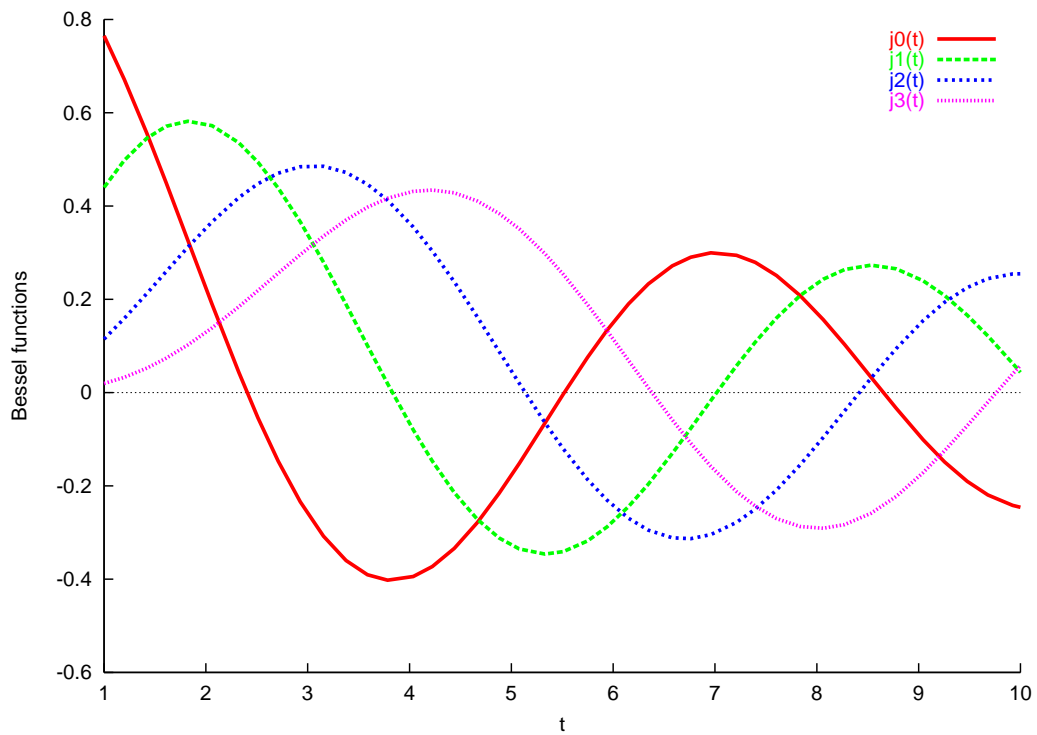


The Ch Language Environment

Version 8.0

Reference Guide



How to Contact SoftIntegration

Mail SoftIntegration, Inc.
 216 F Street, #68
 Davis, CA 95616
Phone + 1 530 297 7398
Fax + 1 530 297 7392
Web <http://www.softintegration.com>
Email info@softintegration.com

Copyright ©2016 by SoftIntegration, Inc. All rights reserved.
Revision 8.0, November 2016

Permission is granted for registered users to make one copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited.

SoftIntegration, Inc. is the holder of the copyright to the Ch language environment described in this document, including without limitation such aspects of the system as its code, structure, sequence, organization, programming language, header files, function and command files, object modules, static and dynamic loaded libraries of object modules, compilation of command and library names, interface with other languages and object modules of static and dynamic libraries. Use of the system unless pursuant to the terms of a license granted by SoftIntegration or as otherwise authorized by law is an infringement of the copyright.

SoftIntegration, Inc. makes no representations, expressed or implied, with respect to this documentation, or the software it describes, including without limitations, any implied warranty merchantability or fitness for a particular purpose, all of which are expressly disclaimed. Users should be aware that included in the terms and conditions under which SoftIntegration is willing to license the Ch language environment as a provision that SoftIntegration, and their distribution licensees, distributors and dealers shall in no event be liable for any indirect, incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of the Ch language environment, and that liability for direct damages shall be limited to the amount of purchase price paid for the Ch language environment.

In addition to the foregoing, users should recognize that all complex software systems and their documentation contain errors and omissions. SoftIntegration shall not be responsible under any circumstances for providing information on or corrections to errors and omissions discovered at any time in this documentation or the software it describes, even if SoftIntegration has been advised of the errors or omissions. The Ch language environment is not designed or licensed for use in the on-line control of aircraft, air traffic, or navigation or aircraft communications; or for use in the design, construction, operation or maintenance of any nuclear facility.

Ch, ChIDE, SoftIntegration, and One Language for All are either registered trademarks or trademarks of SoftIntegration, Inc. in the United States and/or other countries. Microsoft, MS-DOS, Windows, Windows 2000, Windows XP, Windows Vista, and Windows 7 are trademarks of Microsoft Corporation. Solaris and Sun are trademarks of Sun Microsystems, Inc. Unix is a trademark of the Open Group. HP-UX is either a registered trademark or a trademark of Hewlett-Packard Co. Linux is a trademark of Linus Torvalds. Mac OS X and Darwin are trademarks of Apple Computers, Inc. QNX is a trademark of QNX Software Systems. AIX is a trademark of IBM. All other trademarks belong to their respective holders.

Preface

Ch is a C compatible cross-platform scripting language environment. Ch is also a C virtual machine and a superset of C interpreter with salient features from C++, other languages, and software packages. Descriptions of functions and classes of Ch are given in this reference guide. The header files in the distribution of the Ch language environment are listed in Table 1. Functions not explained in this reference manual follow the interpretation of ISO and de facto standards such as ISO C and POSIX standards.

Typographical Conventions

The following list defines and illustrates typographical conventions used as visual cues for specific elements of the text throughout this document.

- Interface components are window titles, button and icon names, menu names and selections, and other options that appear on the monitor screen or display. They are presented in boldface. A sequence of pointing and clicking with the mouse is presented by a sequence of boldface words.

Example: Click **OK**

Example: The sequence **Start**→**Programs**→**Ch7.0**→**Ch** indicates that you first select **Start**. Then select submenu **Programs** by pointing the mouse on **Programs**, followed by **Ch7.0**. Finally, select **Ch**.

- Keycaps, the labeling that appears on the keys of a keyboard, are enclosed in angle brackets. The label of a keycap is presented in typewriter-like typeface.

Example: Press <Enter>

- Key combination is a series of keys to be pressed simultaneously (unless otherwise indicated) to perform a single function. The label of the keycaps is presented in typewriter-like typeface.

Example: Press <Ctrl><Alt><Enter>

- Commands presented in lowercase boldface are for reference only and are not intended to be typed at that particular point in the discussion.

Example: “Use the **install** command to install...”

In contrast, commands presented in the typewriter-like typeface are intended to be typed as part of an instruction.

Example: “Type `install` to install the software in the current directory.”

- Command Syntax lines consist of a command and all its possible parameters. Commands are displayed in lowercase bold; variable parameters (those for which you substitute a value) are displayed

Table 1: Library summary.

Header file	Description	C	POSIX	Ch
aio.h	Asynchronous input and output		X	X
arpa/inet.h	Internet operations			X
array.h	Computational arrays			X
assert.h	Diagnostics	X	X	X
chplot.h	2D and 3D plotting			X
chshell.h	Ch Shell functions			X
complex.h	Complex numbers	X		X
cpio.h	Cpio archive values			X
crypt.h	Encryption functions			X
ctype.h	Character handling	X	X	X
dirent.h	Format of directory entries		X	X
dlfcn.h	Dynamically loaded functions			X
errno.h	Error numbers	X	X	X
fcntl.h	Interprocess communication functions		X	X
fenv.h	Floating-point environment	X		X
float.h	platform-dependent floating-point limits	X	X	X
glob.h	Pathname pattern-matching types			X
grp.h	Group structure		X	X
inttypes.h	Fixed size integral types	X		X
iostream.h	input and output stream in C++ style			X
iso646.h	Alternative spellings	X		X
libintl.h	Message catalogs for internationalization			X
limits.h	platform-dependent integral limits	X	X	X
locale.h	Locale functions	X	X	X
malloc.h	Dynamic memory management functions			X
math.h	Mathematical functions	X	X	X
mqueue.h	Message queues		X	X
netconfig.h	Network configuration database			X
netdb.h	Network database operations			X
netdir.h	Name-to-address mapping for transport protocols			X
netinet/in.h	Internet Protocol family			X
new.h	Memory allocation error handling in C++ style			X
numeric.h	Numerical analysis			X
poll.h	Definitions for the poll() function			X
pthread.h	Threads		X	
pwd.h	Password structure		X	X
re_comp.h	regular-expression-matching functions for re_comp()			X
readline.h	Readline function			X
regex.h	regular-expression-matching types			X
sched.h	execution scheduling		X	X
semaphore.h	Semaphore functions		X	X
setjmp.h	Non-local jumps	X	X	X

Table 1: Library summary (continued).

Header file	Description	C	POSIX	Ch
signal.h	Signal handling	X	X	X
stdarg.h	Variable argument lists	X	X	X
stdbool.h	Boolean numbers	X		X
stddef.h	Miscellaneous functions and macros	X	X	X
stdint.h	Integer types	X	X	X
stdio.h	Input and output	X	X	X
stdlib.h	Utility functions	X	X	X
string.h	String functions	X	X	X
stropts.h	streams interface			X
sys/acct.h	Process accounting			X
sys/fcntl.h	Control file			X
sys/file.h	Accessing the file struct array			X
sys/ioctl.h	Control device			X
sys/ipc.h	Interprocess communication access structure			X
sys/lock.h	Locking processes			X
sys/mman.h	Memory management declarations		X	X
sys/msg.h	Message queue structures			X
sys/procset.h	Set processes			X
sys/resource.h	XSI resource operations			X
sys/sem.h	Semaphore facility			X
sys/shm.h	Shared memory facility			X
sys/socket.h	Internet Protocol family			X
sys/stat.h	File structure function		X	X
sys/time.h	Time types			X
sys/times.h	File access and modification times structure		X	X
sys/types.h	Data types		X	X
sys/uio.h	Vector I/O operations			X
sys/un.h	Unix-domain sockets			X
sys/utsname.h	System name structure		X	X
sys/wait.h	Evaluating exit statuses		X	X
syslog.h	System error logging			X
tar.h	Extended tar definitions			X
termios.h	Define values for termios		X	X
tgmath.h	Type-generic mathematical functions	X		X
time.h	Time and date functions	X	X	X
tiuser.h	Transport layer interface			X
unistd.h	System and process functions		X	X
utime.h	Access and modification times structure		X	X
wait.h	Wait for child process to stop or terminate			X
wchar.h	Multibyte I/O and string functions	X		X
wctype.h	Multibyte character class tests	X		X

Note: the symbol ‘X’ indicates that the library is supported by the standard.

in lowercase italics; constant parameters are displayed in lowercase bold. The brackets indicate items that are optional.

Example: `ls [-aAbcCdfFgilLmnopqrRstux1] [file ...]`

- Command lines consist of a command and may include one or more of the command's possible parameters. Command lines are presented in the typewriter-like typeface.

Example: `ls /home/username`

- Screen text is a text that appears on the screen of your display or external monitor. It can be a system message, for example, or it can be a text that you are instructed to type as part of a command (referred to as a command line). Screen text is presented in the typewriter-like typeface.

Example: The following message appears on your screen

```
usage:  rm [-fiRr] file ...
```

```
ls [-aAbcCdfFgilLmnopqrRstux1] [file ... ]
```

- Function prototype consists of return type, function name, and arguments with data type and parameters. Keywords of the C language, typedefed names, and function names are presented in boldface. Parameters of the function arguments are presented in italic. The brackets indicate items that are optional.

Example: `double derivative(double (*func)(double), double x, ... [double *err, double h]);`

- Source code of programs is presented in the typewriter-like typeface.

Example: The program `hello.ch` with code

```
int main() {
    printf("Hello, world!\n");
}
```

will produce the output `Hello, world!` on the screen.

- Variables are symbols for which you substitute a value. They are presented in italics.

Example: module `n` (where `n` represents the memory module number)

- System Variables and System Filenames are presented in boldface.

Example: startup file `/home/username/.chrc` or `.chrc` in directory `/home/username` in Unix and `C:\ >_chrc` or `.chrc` in directory `C:\ >` in Windows.

- Identifiers declared in a program are presented in typewriter-like typeface when they are used inside a text.

Example: variable `var` is declared in the program.

- Directories are presented in typewriter-like typeface when they are used inside a text.

Example: Ch is installed in the directory `/usr/local/ch` in Unix and `C:/Ch` in Windows.

- Environment Variables are the system level variables. They are presented in boldface.

Example: Environment variable `PATH` contains the directory `/usr/ch`.

Other Relevant Documentations

The core Ch documentation set consists of the following titles. These documentation come with the CD and are installed in CHHOME/docs, where CHHOME is the Ch home directory.

- *The Ch Language Environment — Installation and System Administration Guide*, version 8.0, SoftIntegration, Inc., 2016.
This document covers system installation and configuration, as well as setup of Ch for Web servers.
- *The Ch Language Environment, — User's Guide*, version 8.0, SoftIntegration, Inc., 2016.
This document presents language features of Ch for various applications.
- *The Ch Language Environment, — Reference Guide*, version 8.0, SoftIntegration, Inc., 2016.
This document gives detailed references of functions, classes and commands along with sample source code.
- *The Ch Language Environment, — SDK User's Guide*, version 8.0, SoftIntegration, Inc., 2016.
This document presents Software Development Kit for interfacing with C/C++ functions in static or dynamical libraries.
- *The Ch Language Environment CGI Toolkit User's Guide*, version 3.5, SoftIntegration, Inc., 2016.
This document describes Common Gateway Interface in CGI classes with detailed references for each member function of the classes.

Table of Contents

Preface	i
1 Diagnostic Test — <assert.h>	1
assert	2
2 2D and 3D plotting <chplot.h>	3
CPlot Class	3
CPlot Class Member Functions:	10
arc	10
arcSector	11
arcSegment	12
arrow	16
autoScale	20
axis	21
axisRange	24
axes	28
backgroundColor	30
barGraph	31
barGraphCategory	35
barGraphColor	36
barGraphLegend	36
barGraphValue	37
barSize	38
border	38
borderOffsets	41
boxAndWhisker	42
boxBorder	44
boxFill	46
boxWidth	48
changeViewAngle	49
circle	51
circleClosed	54
circleOpen	56
colorBox	56
contourLabel	58
contourLevels	60
contourMode	62

coordSystem	65
data	70
data2D	73
data2DCurve	77
data3D	78
data3DCurve	85
data3DSurface	87
dataFile	89
dataSetNum	93
deleteData	94
deletePlots	95
dimension	95
displayTime	96
dot	97
ellipse	98
enhanceText	100
expr	104
exprPoint	105
exprxy	107
fillColor	108
fillCurve	109
fillOpacity	113
fillPattern	114
fillStyle	116
font	119
fontScale	120
func2D	121
func2DPoint	122
func3D	124
funcp2D	126
funcp3D	127
getGrid	128
getLabel	129
getOutputType	130
getSubplot	131
getTitle	135
grid	136
histogram	139
histogramColor	141
histogramValue	142
inequal	142
inequalAnd	145
inequalOffset	149
isUsed	150
label	150
legend	152
legendLocation	155
legendOption	157

line	160
line1D	162
line1DOffset	164
lineStyle	164
lineType	166
margins	168
mathCoord	169
noFill	172
noStrokeColor	173
numberLine	173
numberLineScatter	175
numberLineScattern	176
origin	177
outputType	178
plotType	188
plotting	216
point	217
pointStyle	220
pointType	222
polarPlot	223
polygon	226
quad	229
rectangle	230
rectanglexy	233
regularPolygon	234
removeHiddenLine	236
scaleType	238
scattern	240
scatter	241
showMesh	243
size	245
size3D	246
sizeOutput	247
sizeRatio	248
smooth	250
star	252
strokeColor	253
strokeWidth	256
subplot	258
text	259
textColor	260
tics	261
ticsDay	262
ticsDirection	263
ticsFormat	265
ticsLabel	266
ticsLevel	268
ticsLocation	270

ticsMirror	272
ticsMonth	273
ticsPosition	274
ticsRange	275
title	276
triangle	278
Plotting Functions:	279
fplotxy	280
fplotxyz	282
plotxy	284
plotxyf	290
plotxyz	292
plotxyzf	297
3 Shell Functions — <chshell.h>	300
chinfo	302
iskey	304
isstudent	306
isvar	307
sizeofelement	308
4 Complex Functions — <complex.h>	310
cabs	312
cacos	313
cacosh	314
carg	315
casin	316
casinh	317
catan	318
catanh	319
ccos	320
ccosh	321
cexp	322
cimag	323
clog	324
conj	325
cpow	326
creal	327
csin	328
csinh	329
csqrt	330
ctan	331
ctanh	332
iscnan	333

5	Character Handling — <ctype.h>	334
	isalnum	335
	isalpha	336
	iscntrl	337
	isdigit	338
	isgraph	339
	islower	340
	isprint	341
	ispunct	342
	isspace	343
	isupper	344
	isxdigit	345
	tolower	346
	toupper	347
6	Errors — <errno.h>	348
7	Characteristics of floating types <float.h>	349
8	Sizes of integer types — <limits.h>	352
9	Localization <locale.h>	354
	localeconv	356
	setlocale	361
10	Mathematics <math.h>	363
	acos	367
	acosh	368
	asin	369
	asinh	370
	atan	371
	atan2	372
	atanh	374
	cbrt	375
	ceil	376
	copysign	377
	cos	378
	cosh	379
	def2rad	380
	erf	381
	erfc	382
	exp	383
	exp2	384
	expm1	385
	fabs	386
	fdim	387
	floor	388
	fma	389

fmax	390
fmin	391
fmod	392
fpclassify	393
frexp	394
hypot	395
ilogb	396
isfinite	397
isgreater	398
isgreaterequal	399
isinf	400
isless	401
islessequal	402
islessgreater	403
isnan	404
isnormal	405
isunordered	406
ldexp	407
lgamma	408
log	409
log10	410
log1p	411
log2	412
logb	413
lrint	414
lround	415
modf	416
nan	417
nearbyint	418
nextafter	419
nexttoward	420
pow	421
def2rad	422
remainder	423
remquo	424
rint	425
round	426
scalbn and scalbln	427
signbit	428
sin	429
sinh	430
sqrt	431
tan	432
tanh	433
tgamma	434
trunc	435

11 Numeric Analysis — <numeric.h>	436
balance	441
ccompanionmatrix	444
cdeterminant	445
cdiagonal	447
cdiagonalmatrix	450
cdivisor	452
cdivisornum	453
cfevalarray	454
cfunm	456
charpolycoef	458
choldecomp	460
cinverse	464
clinsolve	466
cmean	468
combination	470
companionmatrix	471
complexsolve	472
complexsolvePP	478
complexsolvePR	480
complexsolveRP	483
complexsolveRR	485
complexsolveRRz	488
condnum	490
constrain	492
conv	493
conv2	498
corrcoef	501
correlation2	503
covariance	505
cpolyeval	508
cproduct	510
cquadratic	512
cross	514
csum	515
ctrace	517
ctriangularmatrix	518
cumprod	521
cumsum	523
curvefit	525
deconv	529
derivative	532
derivatives	534
determinant	537
diagonal	539
diagonalmatrix	542
difference	544
divisor	545

divisornum	546
dot	547
eigen	548
eigensystem	553
expm	554
expfit	556
factor	559
factorial	560
factornum	561
fevalarray	562
fft	564
filter	569
filter2	574
findvalue	576
fliplr	578
flipud	580
fminimum	582
fminimums	585
fsolve	588
funm	591
fzero	593
fzero1	595
fzero2	596
gcd	598
gcd	599
getnum	601
hessdecomp	602
histogram	605
householdermatrix	608
identitymatrix	610
ifft	611
integral1	613
integralp1	616
integral2	618
integral3	620
integration2	622
integration3	624
interp1	627
interp2	629
inverse	632
isprime	634
lcm	635
lcmarray	636
lindata	637
linearcorrcoef	639
linearfit	640
linrange	643
linsolve	644

linsolvesi	646
linsolvestd	648
linspace	650
llsqcovsolve	653
llsqnonnegsolve	655
llsqsolve	657
logm	660
logdata	662
logrange	663
logspace	664
ludecomp	666
maxloc	670
maxv	671
mean	672
median	675
minloc	677
minv	678
mode	679
norm	681
nullspace	685
oderk	688
oderungekutta	697
odesolve	699
orthonormalbase	701
pinverse	704
polycoef	709
polyder	712
polyder2	714
polyeval	717
polyevalarray	719
polyevalm	721
polyfit	723
polygonperi	727
product	728
qrdecomp	730
qrdelete	736
qrinsert	740
quadratic	743
quadraticfit	745
quartiles	748
randcolor	750
randdouble	751
randint	752
randintfreq	753
rank	754
rcondnum	756
regularpolygon	758
residue	760

rgb2hex	765
roots	766
rot90	768
rotatepoint	770
rotatepolygon	771
rotmatrix	773
rsf2csf	774
schurdecomp	777
sign	780
sort	781
specialmatrix	784
Cauchy	784
ChebyshevVandermonde	785
Chow	787
Circul	788
Clement	789
DenavitHartenberg	790
DenavitHartenberg2	793
Dramadah	794
Fiedler	796
Frank	797
Gear	798
Hadamard	799
Hankel	801
Hilbert	802
InverseHilbert	803
Magic	803
Pascal	805
Rosser	806
Toeplitz	807
Vandermonde	809
Wilkinson	809
sqrtm	811
std	813
sum	815
svd	817
trace	824
triangularmatrix	826
unwrap	829
unwrapdeg	831
urand	832
vertex	833
xcorr	834
12 Non-local jumps <setjmp.h>	837
setjmp	839
longjmp	841

13	Signal handling <signal.h>	842
	signal	844
	raise	846
14	Variable Argument Lists — <stdarg.h>	847
	arraycopy	851
	va_arg	853
	va_arraydim	854
	va_arrayextent	855
	va_arraynum	856
	va_arraytype	857
	va_copy	859
	va_count	861
	va_datatype	862
	va_dim	863
	va_elementtype	864
	va_end	865
	va_extent	866
	va_start	867
	va_tagname	869
15	Boolean type and values <stdbool.h>	871
16	Common Definitions <stddef.h>	872
17	Input/Output <stdio.h>	874
	clearerr	879
	fclose	880
	feof	881
	ferror	882
	fflush	883
	fgetc	884
	fgetpos	885
	fgets	886
	fopen	887
	fprintf	888
	fputc	892
	fputs	893
	fread	894
	freopen	895
	fscanf	896
	fseek	899
	fsetpos	901
	ftell	902
	fwrite	904
	getchar	905
	getc	906
	gets	907

perror	908
printf	910
putchar	911
putc	912
puts	913
remove	914
rename	915
rewind	917
scanf	918
setbuf	920
setvbuf	921
snprintf	922
sprintf	923
sscanf	924
tmpfile	925
tmpnam	926
ungetc	927
vfprintf	929
vprintf	930
vsprintf	931
vsprintf	932

18 String Handling — <stdlib.h> 933

abort	936
abs	937
labs	937
llabs	937
atexit	938
atoc	939
atof	940
atoi	941
atol	941
atoll	941
bsearch	942
calloc	944
div	945
exit	946
free	947
getenv	948
iscnum	949
isenv	950
isnum	951
iswnum	952
malloc	953
mblen	954
mbstowcs	955
mbtowc	957
qsort	959

rand	961
realloc	962
remenv	964
srand	965
strtod	966
strtof	966
strtold	966
strtol	968
strtoll	968
strtoul	968
strtoull	968
system	970
wcstombs	971
wctomb	972

19 String Functions — <string.h> 973

memchr	975
memcmp	976
memcpy	977
memmove	978
memset	979
str2ascii	980
str2mat	981
stradd	983
strcasecmp	984
strcat	985
strchr	986
strcmp	987
strcoll	988
strconcat	989
strcpy	990
strcspn	991
strdup	992
strerror	993
strgetc	994
strjoin	995
strlen	996
strncasecmp	997
strncat	998
strncmp	999
strncpy	1000
strpbrk	1001
strputc	1002
strchr	1003
strep	1004
strspn	1005
strstr	1006
strtok_r	1007

strtok	1009
strxfrm	1011
20 Date and Time Functions — <time.h>	1012
asctime	1014
clock	1016
ctime	1017
difftime	1018
gmtime	1019
localtime	1020
mktime	1021
strftime	1022
time	1026
21 Extended Multibyte and Wide Character Functions — <wchar.h>	1027
_wfopen	1030
btowc	1031
fgetwc	1032
fgetws	1033
fputwc	1034
fputws	1036
fwide	1037
getwc	1038
getwchar	1039
mbrlen	1040
mbrtowc	1041
mbsinit	1043
mbsrtowcs	1044
putwc	1046
putwchar	1047
ungetwc	1048
wcrtomb	1050
wscat	1052
wcschr	1053
wscmp	1055
wscoll	1057
wscpy	1059
wscspn	1060
wcsftime	1061
wcslen	1063
wcsncat	1064
wcsncmp	1065
wcsncpy	1067
wcspbrk	1068
wcsrchr	1070
wcsrtombs	1072
wcsspn	1074
wcsstr	1075

wctod	1077
wctok	1079
wctol	1081
wcsxfrm	1083
wctob	1084
wmemchr	1085
wmemcmp	1087
wmemcpy	1089
wmemmove	1091
wmemset	1093
22 Wide Character Functions — <wctype.h>	1095
iswalnum	1098
iswalpha	1099
iswcntrl	1100
iswctype	1101
iswdigit	1103
iswgraph	1104
iswlower	1105
iswprint	1106
iswpunct	1107
iswspace	1108
iswupper	1109
iswxdigit	1110
towctrans	1111
towlower	1112
towupper	1113
wctrans	1114
wctype	1115
A Functions Not Supported in Specific Platforms	1116
A.1 HPUX	1116
A.2 Linux	1118
A.3 Solaris	1121
A.4 Windows	1122
Index	1130

Chapter 1

Diagnostic Test — `<assert.h>`

The header `assert.h` defines the `assert` macro and refers to another macro,

`NDEBUG`

which is *not* defined by `assert.h`. If `NDEBUG` is defined as a macro at the point in the source file where `assert.h` is included, the `assert` macro is defined simply as

```
# define assert(ignore) ((void)0)
```

The `assert` macro is redefined according to the current state of `NDEBUG` each time that `assert.h` is included.

Macros

The following macros are defined by the `asserth` header file.

Macro	Description
<code>assert</code>	Inserts diagnostic tests into program.

Portability

This header has no known portability problem.

assert

Synopsis

```
#include <assert.h>
```

```
void assert(scalar expression);
```

Purpose

Insert diagnostic tests into program.

Return Value

The **assert** macro returns no value.

Parameters

expression argument.

Description

The **assert** macro puts diagnostic tests into programs; it expands to a void expression. When it is executed, if *expression* (which shall have a scalar type) is false (that is, compares equal to 0), the **assert** macro writes information about the particular call that failed (including the text of the argument, the name of the source file, the source line number, and the name of the enclosing function – the latter are respectively the values of the preprocessing macro **__FILE__** and **__LINE__** and of the identifier **__func__**) on the standard output. It then calls the **abort** function.

Example

```
/* a example for assert(). The program will
abort while i < 0 and display error message.*/
#include <assert.h>

int main() {
    int i;
    for(i = 5; ; i--)
    {
        printf("i= %d\n", i);
        assert(i >= 0);
    }
}
```

Output

```
i= 5
i= 4
i= 3
i= 2
i= 1
i= 0
i= -1
Assertion failed: i >= 0, file assert_func.c, line 10, function main()
```

See Also

abort()

Chapter 2

2D and 3D plotting <chplot.h>

The high-level plotting features described in this chapter is available only in Ch Professional Edition. The header file **chplot.h** contains the definition of the plotting **CPlot** class, defined macros used with **CPlot**, and definitions of high-level plotting functions that are based on **CPlot** class .

The plotting class **CPlot** allows for high-level creation and manipulation of plots within the Ch language environment. **CPlot** can be used directly within many types of Ch programs, including applications, function files, and CGI programs. Plots can be generated from data arrays or files, and can be displayed on a screen, saved in a large number of different file formats, or generated as a stdout stream in png or gif file format on the Web.

The high-level plotting functions **fplotxy()**, **fplotxyz()**, **plotxy()**, **plotxyf()**, **plotxyz()**, and **plotxyzf()** are designed to be easy to use and allow plots to be created quickly. These functions can be used in conjunction with the **CPlot** member functions to create more sophisticated plots.

CPlot Class

The **CPlot** class can be used to produce two dimensional (2D) and three dimensional (3D) plots through a Ch program. Member functions of the **CPlot** class generate actual plots using a plotting engine. Gnuplot is used internally as the plotting engine for display in this release of the Ch language environment.

Public Data

None.

Public Member Functions

Function	Description
CPlot()	Class constructor. Creates and initializes a new instance of the class.
~CPlot()	Class destructor. Frees memory associated with a instance of the class.
arc()	Add an arc to a 2D plot.
arcSector()	Add an arc sector to a 2D plot.
arcSegment()	Add an arc segment to a 2D plot.
arrow()	Add an arrow to a plot.

autoScale()	Enable or disable autoscaling of plot axes.
axis()	Enable or disable drawing of x-y axis on 2D plots.
axisRange()	Set the range for a plot axis.
axes()	Specify the axes for a data set.
backgroundColor()	Specify the color of the background.
barGraph()	Draw a bar graph.
barGraphCategory()	Label each category for a bar graph.
barGraphColor()	Specify the color for each category for a bar graph.
barGraphLegend()	Specify the legend for each data set for a bar graph.
barGraphValue()	Add a value for each category on a bar graph.
barSize()	Set the size of error bars.
border()	Enable or disable drawing of a border around the plot.
borderOffsets()	Set plot offsets of the plot border.
boundingBoxOrigin()	Obsolete, use <code>CPlot::origin()</code> .
boxAndWhisker()	Draw a Box-and-Whiskers plot.
boxBorder()	Enable or disable drawing of a border for a plot of box type.
boxFill()	Obsolete, use <code>fillStyle()</code> . Fill a box or filled curve with a solid color or pattern.
boxWidth()	Set the width for a box.
changeViewAngle()	Change the view angles for a 3D plot.
circle()	Add a circle to a 2D plot.
circleClosed()	Draw a small closed circle to a 2D plot.
circleOpen()	Draw a small open circle to a 2D plot.
colorBox()	Enable or disable the drawing of a color box for 3D surface plots.
contourLabel()	Enable or disable contour labels for 3D surface plots.
contourLevels()	Set contour levels for 3D plot to be displayed at specific locations.
contourMode()	Set the contour display mode for 3D surface plots.
coordSystem()	Set the coordinate system for a 3D plot.
data()	Add 2D, 3D, or multi-dimensional data to an instance of the CPlot class.
data2D()	Add one or more 2D data sets to an instance of the CPlot class.
data2DCurve()	Add a set of data for 2D curve to an instance of the CPlot class.
data3D()	Add one or more 3D data sets to an instance of the CPlot class.
data3DCurve()	Add a set of data for 3D curve to an instance of the CPlot class.
data3DSurface()	Add a set of data for 3D surface to an instance of the CPlot class.
dataFile()	Add a data file to an instance of the CPlot class.
dataSetNum()	Obtain the current data set number in an instance of the CPlot class.
deleteData()	Remove data from a previously used instance of the CPlot class.
deletePlots()	Remove any data from a previously used instance of the CPlot class and reinitialize option to default values.
dimension()	Set plot dimension to 2D or 3D.
displayTime()	Display the current time and date on the plot.
dot()	Draw a dot plot.
ellipse()	Add an ellipse to a 2D plot.
enhanceText()	Use enhanced text for special symbols.
expr()	Add a set of 2D data based on an expression for plotting a curve.
exprPoint()	Add a set of 2D data based on an expression for plotting points.
exprxy()	Add a set of 3D data based on an expression to an instance of the CPlot class.
fillCurve()	Fill a line or curve with color with respect to a horizontal line or other reference.
fillColor()	Fill a color for an object.

fillOpacity()	Set the opacity (transparency) of a filled solid color for an object.
fillPattern()	Fill a pattern for an object.
fillStyle()	Fill a box or filled curve with a solid color or pattern.
font()	Set the font name and size used in all texts in the plot.
fontScale()	Set the font scale factor used in all texts in the plot.
func2D()	Add a set of 2D data using a function for plotting a curve.
func2DPoint()	Add a set of 2D data using a function for plotting points.
func3D()	Add a set of 3D data using a function to an instance of the CPlot class.
funcp2D()	Add a set of 2D data using a function with a parameter to an instance of the CPlot class.
funcp3D()	Add a set of 3D data using a function with a parameter to an instance of the CPlot class.
getGrid()	Get the grid information the plot.
getLabel()	Get the label of an axis.
getOutputType()	Get the output type of a plot.
getSubplot()	Get a pointer to an element of a subplot.
getTitle()	Get the title of the plot.
grid()	Enable or disable display of a grid.
histogram()	Draw a histogram.
histogramColor()	Specify the color for each interval for a histogram.
histogramValue()	Add a value for each interval on a histogram.
inequal()	Add an inequality.
inequalAnd()	Add two inequalities with the AND relational operation.
inequalOffset()	Add an offset for each inequality on the plot.
isUsed()	Test if an instance of the CPlot class has been used.
label()	Set axis labels.
legend()	Add a legend for a data set.
legendLocation()	Specify the plot legend location.
legendOption()	Set options for legends of a plot.
line()	Add a line to a plot.
line1D()	Draw a line segment in a one-dimensional plot.
line1DOffset()	Set the vertical offset for drawing a line segment using line1D() .
lineStyle()	Set the line style, such as solid or dotted line, for lines and curves.
lineType()	Set the line type, width, and color for lines, impulses, steps, etc.
margins()	Set plot margins.
mathCoord()	Set proper attributes for the math coordinates.
noFill()	Do not fill a color or pattern for an object.
noStrokeColor()	Do not add outline for an object. No affect on point and text.
numberLine()	Draw a number Line in a one-dimensional plot.
numberLineScatter()	Draw a number Line in scatter plot.
numberLineScattern()	Draw a number Line in scatter plot.
origin()	Set the location of the origin for the bounding box of the plot.
outputType()	Set the plot output type.
plotType()	Set the plot type.
plotting()	Produce a plot from an instance of the CPlot class.
point()	Add a point to a plot.
pointStyle()	Set the point style, such as circle, filled circle, square.
pointType()	Set the point type, size, and color.
polarPlot()	Set a 2D plot to use the polar coordinate system.
polygon()	Add a polygon to a plot.

<code>()</code>	Add a quadrilateral to a 2D plot.
<code>rectangle()</code>	Add a rectangle to a 2D plot.
<code>rectanglexy()</code>	Add a rectangle to a 2D plot.
<code>regularPolygon()</code>	Add a regular polygon to a plot.
<code>removeHiddenLine()</code>	Enable or disable hidden line removal for 3D plots.
<code>scaleType()</code>	Set the axis scale type for a plot.
<code>scatter()</code>	Add a scatter plot to a 2D plot.
<code>scattern()</code>	Add a scatter plot to a 2D plot.
<code>showMesh()</code>	Enable or disable display of mesh of a 3D plot.
<code>size()</code>	Scale the plot itself relative to the size of the output file or canvas.
<code>size3D()</code>	Change the size of a 3D plot.
<code>sizeOutput()</code>	Change the size of the output file.
<code>sizeRatio()</code>	Change the aspect ratio of a plot.
<code>smooth()</code>	Smooth plotting curves by interpolation and approximation of data.
<code>star()</code>	Add a star (pentagram) to a plot.
<code>strokeColor()</code>	Set the color of a point, line, or outline of an object.
<code>strokeWidth()</code>	Set the thickness of a point, line, or outline of an object.
<code>subplot()</code>	Create a group of subplots.
<code>text()</code>	Add a text string to a plot.
<code>textColor()</code>	Set the color of text.
<code>tics()</code>	Enable or disable display of axis tics.
<code>ticsDay()</code>	Set axis tic-mark labels to days of the week.
<code>ticsDirection()</code>	Set the direction in which axis tic-marks are drawn.
<code>ticsFormat()</code>	Set the number format for tic labels.
<code>ticsLabel()</code>	Set location and text label for arbitrary axis labels.
<code>ticsLevel()</code>	Set the z-axis offset for drawing of tics in 3D plots.
<code>ticsLocation()</code>	Specify the location of axis tic marks to be on the border or the axis.
<code>ticsMirror()</code>	Enable or disable display of axis tics on the opposite axis.
<code>ticsMonth()</code>	Set axis tic-mark labels to months.
<code>ticsPosition()</code>	Add tic-marks at the specified positions to an axis.
<code>ticsRange()</code>	Specify the range for a series of tics on an axis.
<code>title()</code>	Set the plot title.
<code>triangle()</code>	Add a triangle to a 2D plot.

Math Symbols

The Ch plotting uses the enhanced text for strings such as title, labels, and legends. Special symbols for the enhanced text can be found in the member function `enhanceText()`. The strings "`<=`" and "`>`" represent \leq and \geq , respectively. To print a superscript on a plot, the symbol `^` is used, such as `x^2` for x^2 . To print a subscript on a plot, the symbol `_` is used, such as `x_2` for x_2 . The string "`M_PI`" is for π . The string "`/Symbol @{\140}\326}x`" is used to print a square root function sign \sqrt{x} .

Macros

The following macros are defined for the `CPlot` class.

Macro	Description
-------	-------------

PLOT_ANGLE_DEG	Select units in degree for angular values.
PLOT_ANGLE_RAD	Select units in radian for angular values.
PLOT_AXIS_X	Select the x axis only.
PLOT_AXIS_X2	Select the x2 axis on the top only.
PLOT_AXIS_XY	Select the x and y axes.
PLOT_AXIS_XYZ	Select the x, y, and z axes.
PLOT_AXIS_Y	Select the y axis only.
PLOT_AXIS_Y2	Select the y2 axis on the right only.
PLOT_AXIS_Z	Select the z axis only.
PLOT_BORDER_BOTTOM	The bottom of the plot.
PLOT_BORDER_LEFT	The left side of the plot.
PLOT_BORDER_TOP	The top of the plot.
PLOT_BORDER_RIGHT	The right side of the plot.
PLOT_BORDER_ALL	All sides of the plot.
PLOT_BOXFILL_EMPTY	Do not fill a box.
PLOT_BOXFILL_SOLID	Fill a box with a solid color.
PLOT_BOXFILL_TRANSPARENT_SOLID	Fill a box with a transparent solid color.
PLOT_BOXFILL_PATTERN	Fill a box with a pattern.
PLOT_BOXFILL_TRANSPARENT_PATTERN	Fill a box with a transparent pattern.
PLOT_CONTOUR_BASE	Draw contour lines for a surface plot on the x-y plane.
PLOT_CONTOUR_SURFACE	Draw contour lines for a surface plot on the surface.
PLOT_COORD_CARTESIAN	Use a Cartesian coordinate system for a 3D plot.
PLOT_COORD_CYLINDRICAL	Use a cylindrical coordinate system in a 3D plot.
PLOT_COORD_SPHERICAL	Use a spherical coordinate system in a 3D plot.
PLOT_FILLSTYLE_EMPTY	Do not fill a box.
PLOT_FILLSTYLE_SOLID	Fill a box with a solid color.
PLOT_FILLSTYLE_TRANSPARENT_SOLID	Fill a box with a transparent solid color.
PLOT_FILLSTYLE_PATTERN	Fill a box with a pattern.
PLOT_FILLSTYLE_TRANSPARENT_PATTERN	Fill a box with a transparent pattern.
PLOT_OFF	Flag to disable an option.
PLOT_ON	Flag to enable an option.
PLOT_OUTPUTTYPE_DISPLAY	Display the plot on a screen.
PLOT_OUTPUTTYPE_FILE	Output the plot to a file.
PLOT_OUTPUTTYPE_STREAM	Output the plot as a stdout stream.
PLOT_PLOTTYPE_BOXERRORBARS	It is a combination of the PLOT_PLOTTYPE_BOXES and PLOT_PLOTTYPE_YERRORBARS plot types.
PLOT_PLOTTYPE_BOXES	Draw a box centered about the given x coordinate.
PLOT_PLOTTYPE_BOXXYERRORBARS	A combination of PLOT_PLOTTYPE_BOXES and PLOT_PLOTTYPE_XYERRORBARS plot types.
PLOT_PLOTTYPE_CANDLESTICKS	Display box-and-whisker plots of financial or statistical data.
PLOT_PLOTTYPE_DOTS	Use dots to mark each data point.
PLOT_PLOTTYPE_FILLED_CURVES	Fill an area bounded in a side by a curve with a solid color or pattern.
PLOT_PLOTTYPE_FINANCEBARS	Display financial data.
PLOT_PLOTTYPE_FSTEPS	Adjacent points are connected with two line segments, one from (x1,y1) to (x1,y2), and a second from (x1,y2) to (x2,y2).
PLOT_PLOTTYPE_HISTEPS	The point x1 is represented by a horizontal line from ((x0+x1)/2,y1) to ((x1+x2)/2,y1). Adjacent lines are

PLOT_PLOTTYPE_IMPULSES	connected with a vertical line from $((x1+x2)/2, y1)$ to $((x1+x2)/2, y2)$.
PLOT_PLOTTYPE_LINES	Display vertical lines from the x-axis (for 2D plots) or the x-y plane (for 3D plots) to the data points.
PLOT_PLOTTYPE_LINESPOINTS	Data points are connected with a line.
PLOT_PLOTTYPE_POINTS	Markers are displayed at each data point and connected with a line.
PLOT_PLOTTYPE_STEPS	Markers are displayed at each data point.
	Adjacent points are connected with two line segments, one from $(x1, y1)$ to $(x2, y1)$, and a second from $(x2, y1)$ to $(x2, y2)$.
PLOT_PLOTTYPE_SURFACES	Data points are connected and smoothed as a surface. For 3D plot only.
PLOT_PLOTTYPE_VECTORS	Display vectors.
PLOT_PLOTTYPE_XERRORBARS	Display dots with horizontal error bars.
PLOT_PLOTTYPE_XERRORLINES	Display linepoints with horizontal error lines.
PLOT_PLOTTYPE_XYERRORBARS	Display dots with horizontal and vertical error bars.
PLOT_PLOTTYPE_XYERRORLINES	Display linepoints with horizontal and vertical error lines.
PLOT_PLOTTYPE_YERRORBARS	Display points with vertical error bars.
PLOT_PLOTTYPE_YERRORLINES	Display linepoints with vertical error lines.
PLOT_SCALETYPE_LINEAR	Use a linear scale for a specified axis.
PLOT_SCALETYPE_LOG	Use a logarithmic scale for a specified axis.
PLOT_TEXT_CENTER	Center text at a specified point.
PLOT_TEXT_LEFT	Left justify text at a specified point.
PLOT_TEXT_RIGHT	Right justify text at a specified point.
PLOT_TICS_IN	Draw axis tic-marks inward.
PLOT_TICS_OUT	Draw axis tic-marks outward.

Functions

The following functions are implemented using the **CPlot** class.

Function	Description
fplotxy()	Plot a 2D function of x in a specified range or initialize an instance of the CPlot class.
fplotxyz()	Plot a 3D function of x and y in a specified range or initialize an instance of the CPlot class.
plotxy()	Plot a 2D data set or initialize an instance of the CPlot class.
plotxyf()	Plot 2D data from a file or initialize an instance of the CPlot class.
plotxyz()	Plot a 3D data set or initialize an instance of the CPlot class.
plotxyzf()	Plot 3D data from a file or initialize an instance of the CPlot class.

References

Chapter 2: 2D and 3D plotting <chplot.h>

T. Williams, C. Kelley, D. Denholm, D. Crawford, et al., *Gnuplot — An Interactive plotting Program*, Version 3.7, December 3, 1998, <ftp://ftp.gnuplot.vt.edu/>.

Copyright Notice of Gnuplot

```
/* [  
 * Copyright 1986 - 1993, 1998   Thomas Williams, Colin Kelley  
 *  
 * Permission to use, copy, and distribute this software and its  
 * documentation for any purpose with or without fee is hereby granted,  
 * provided that the above copyright notice appear in all copies and  
 * that both that copyright notice and this permission notice appear  
 * in supporting documentation.  
 *  
 * Permission to modify the software is granted, but not the right to  
 * distribute the complete modified source code. Modifications are to  
 * be distributed as patches to the released version. Permission to  
 * distribute binaries produced by compiling modified sources is granted,  
 * provided you  
 * 1. distribute the corresponding source modifications from the  
 * released version in the form of a patch file along with the binaries,  
 * 2. add special version identification to distinguish your version  
 * in addition to the base release version number,  
 * 3. provide your name and address as the primary contact for the  
 * support of your modified version, and  
 * 4. retain our contact information in regard to use of the base  
 * software.  
 * Permission to distribute the released version of the source code  
 * along with corresponding source modifications in the form of a patch  
 * file is granted with same provisions 2 through 4 for binary  
 * distributions.  
 *  
 * This software is provided "as is" without express or implied warranty  
 * to the extent permitted by applicable law.  
 */
```

CPlot::arc

Synopsis

```
#include <chplot.h>
```

```
int arc(double x_center, double y_center, double r, double startAngle, double endAngle);
```

Purpose

Add an arc to a 2D plot.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x_center The x coordinate of the center of the arc.

y_center The y coordinate of the center of the arc.

r The radius of the arc.

startAngle The start angle from the x-axis in radians.

endAngle The end angle from the x-axis in radians.

Description

This function adds an arc to a 2D plot. It is a convenience function for creation of a geometric primitive. An arc added with this function is counted as a data set for later calls to **CPlot::legend()** and **CPlot::plotType()**. The *x* and *y* are the coordinates of the center of the arc and *r* is the radius of the arc, all specified in units of the *x* and *y* axes. The *startAngle* is the start angle from the x-axis in radians. The *endAngle* is the end angle from the x-axis in radians. The value of *endAngle* shall be large than the value of *startAngle*. Otherwise, no arc will be drawn and the member function returns -1.

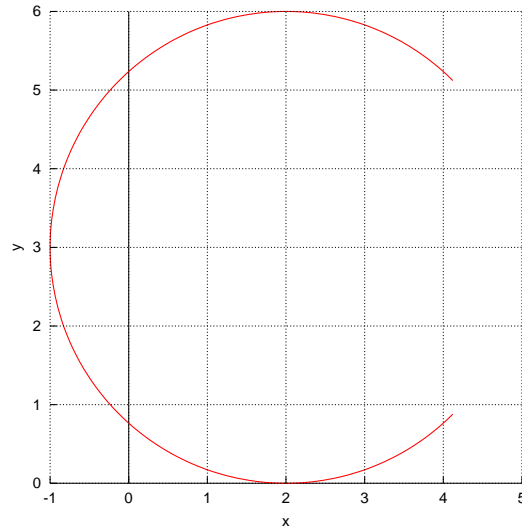
Example 1

```
/* File: arc.ch */
#include <chplot.h>

double x_center = 2, y_center = 3, r = 3;
double startAngle = 45, endAngle = 315;
CPlot plot;

plot.arc(x_center, y_center, r, startAngle, endAngle);
plot.sizeRatio(1);
plot.plotting();
```

Output

**See Also**

CPlot::arcSector(), **CPlot::arcSegment()**, **CPlot::circle()**, **CPlot::data2D()**, **CPlot::data2DCurve()**, **CPlot::ellipse()**, **CPlot::line()**, **CPlot::point()**.

CPlot::arcSector

Synopsis

```
#include <chplot.h>
```

```
int arcSector(double x_center, double y_center, double r, double startAngle, double endAngle);
```

Purpose

Add an arc sector to a 2D plot.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x_center The x coordinate of the center of the arc sector.

y_center The y coordinate of the center of the arc sector.

r The radius of the arc sector.

startAngle The start angle from the x-axis in radians.

endAngle The end angle from the x-axis in radians.

Description

This function adds an arc sector to a 2D plot. It is a convenient function for creation of a geometric primitive. The *x* and *y* are the coordinates of the center of the arc sector and *r* is the radius of the arc sector, all specified in units of the *x* and *y* axes. The *startAngle* is the start angle from the x-axis in radians. The *endAngle* is the end angle from the x-axis in radians. The value of *endAngle* shall be large than the value of *startAngle*. Otherwise, no arcSector will be drawn and the member function returns -1.

Please read the section **Handling Outline and Filled Color of Objects in CPlot** described in `CPlot::circle()` on page 51 about how to handle the outline and filled color for an arc sector.

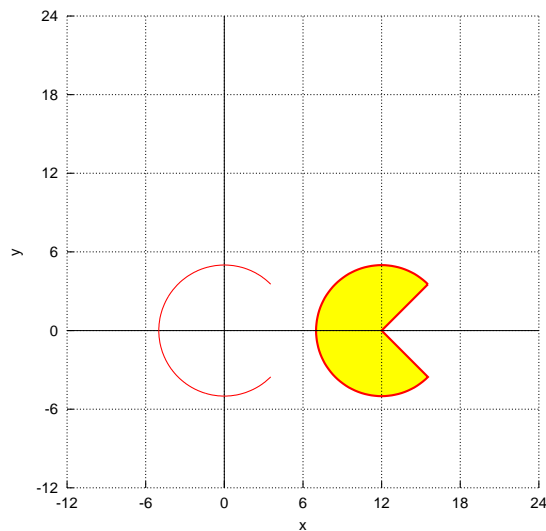
Example 1

```
/* File: arcSector.ch */
#include <chplot.h>
CPlot plot;
double x_center = 0, y_center = 0, r = 5;
double startAngle = 45, endAngle = 315;

plot.fillColor("yellow");
plot.arc(x_center, y_center, r, startAngle, endAngle);
plot.arcSector(x_center+12, y_center, r, startAngle, endAngle);

plot.axisRange(PLOT_AXIS_X, -12, 24);
plot.axisRange(PLOT_AXIS_Y, -12, 24);
plot.ticsRange(PLOT_AXIS_X, 6);
plot.ticsRange(PLOT_AXIS_Y, 6);
plot.sizeRatio(1);
plot.plotting();
```

Output



See Also

`CPlot::arc()`, `CPlot::arcSegment()`, `CPlot::circle()`, `CPlot::ellipse()`, `CPlot::polygon()`, `CPlot::quad()`, `CPlot::rectangle()` `CPlot::rectanglexy()`, `CPlot::triangle()`.

CPlot::arcSegment

Synopsis

```
#include <chplot.h>
```

```
int arcSegment(double x_center, double y_center, double r, double startAngle, double endAngle, ...
               /* [string_t color] */);
```

Syntax

arcSegment(*x_center*, *y_center*, *r*, *startAngle*, *endAngle*)

arcSegment(*x_center*, *y_center*, *r*, *startAngle*, *endAngle*, *color*)

Purpose

Add an arc segment to a 2D plot.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x_center The x coordinate of the center of the arc segment.

y_center The y coordinate of the center of the arc segment.

r The radius of the arc segment.

startAngle The start angle from the x-axis in radians.

endAngle The end angle from the x-axis in radians.

color The color for the triangular part of the arc sector.

Description

This function adds an arc segment to a 2D plot. It is a convenient function for creation of a geometric primitive. The *x* and *y* are the coordinates of the center of the arc segment and *r* is the radius of the arc segment, all specified in units of the x and y axes. The [startAngle] is the start angle from the x-axis in radians. The *endAngle* is the end angle from the x-axis in radians.

By default the triangular part of the arc sector is plotted with the default background color. This default color can be changed by the optional sixth argument of the member function **CPlot::arcSegment()**.

Please read the section **Handling Outline and Filled Color of Objects in CPlot** described in **CPlot::circle()** on page 51 about how to handle the outline and filled color for an arc segment.

Example 1

```
/* File: arcSegment.ch */
#include <chplot.h>
CPlot plot;

double x = 0, y = 0, r = 3;
double startAngle = 0, endAngle = 90;
double startAngle2 = 100, endAngle2 = 190;
double startAngle3 = 200, endAngle3 = 290;

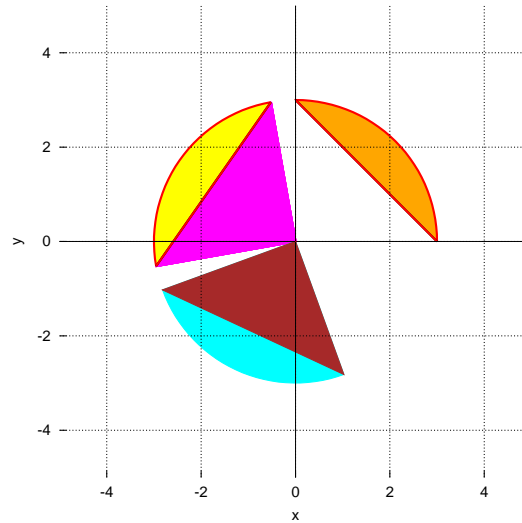
plot.strokeColor("red");
plot.fillColor("orange");
plot.arcSegment(x, y, r, startAngle, endAngle);

plot.fillColor("yellow");
plot.arcSegment(x, y, r, startAngle2, endAngle2, "magenta");

plot.noStrokeColor();
plot.fillColor("cyan");
plot.arcSegment(x, y, r, startAngle3, endAngle3, "brown");
```

```
plot.sizeRatio(1);
plot.plotting();
```

Output



Example 2

```
/* File: arcSegment_2.ch */
#include <chplot.h>
CPlot plot;

double x = 4, y = 5, r = 3;
double startAngle = 0, endAngle = 90;
double startAngle2 = 100, endAngle2 = 190;
double startAngle3 = 200, endAngle3 = 290;

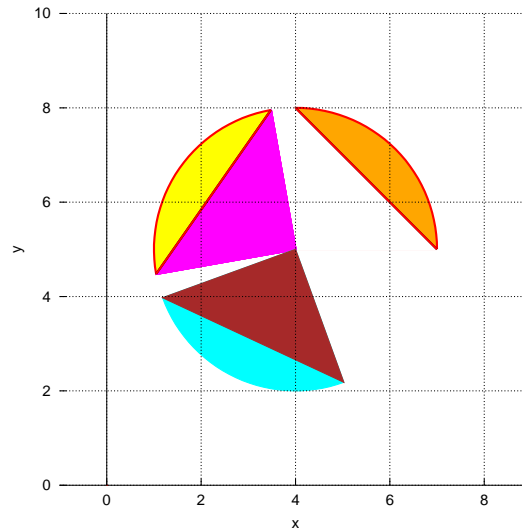
plot.strokeColor("red");
plot.fillColor("orange");
plot.arcSegment(x, y, r, startAngle, endAngle);

plot.fillColor("yellow");
plot.arcSegment(x, y, r, startAngle2, endAngle2, "magenta");

plot.noStrokeColor();
plot.fillColor("cyan");
plot.arcSegment(x, y, r, startAngle3, endAngle3, "brown");

plot.sizeRatio(1);
plot.plotting();
```

Output



Example 3: The difference between the end angle and start angle is larger than 180 degrees.

```

/* File: arcSegment_3.ch */
#include <chplot.h>
CPlot plot;

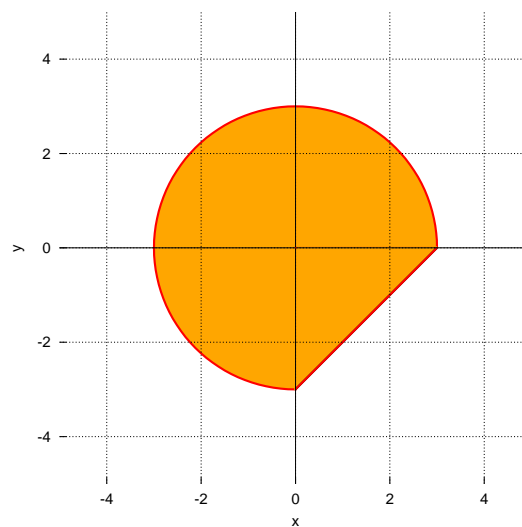
double x = 0, y = 0, r = 3;
double startAngle = 0, endAngle = 270;

plot.strokeColor("red");
plot.fillColor("orange");
plot.arcSegment(x, y, r, startAngle, endAngle);
//plot.arcSegment(x, y, r, startAngle, endAngle, "blue");

plot.sizeRatio(1);
plot.plotting();

```

Output



See Also

CPlot::arc(), **CPlot::arcSector()**, **CPlot::circle()**, **CPlot::ellipse()**, **CPlot::polygon()**, **CPlot::quad()**,
CPlot::rectangle() **CPlot::rectanglexy()**, **CPlot::triangle()**.

CPlot::arrow

Synopsis

```
#include <chplot.h>
```

```
void arrow(double x_tail, double y_tail, double z_tail, double x_head, double y_head, double z_head, ...
           /* [char option] */);
```

Syntax

```
arrow(x_tail, y_tail, x_head, y_head)
```

```
arrow(x_tail, y_tail, x_head, y_head, option)
```

```
arrow(x_tail, y_tail, z_head, x_head, y_tail, z_tail)
```

```
arrow(x_tail, y_tail, z_tail, x_head, y_head, z_head, option)
```

Purpose

Add an arrow to a plot.

Return Value

None.

Parameters

x_tail For x coordinate of the tail of the arrow.

y_tail The y coordinate of the tail of the arrow.

z_tail For 2D plots this is ignored. For 3D plots, the z coordinate of the head of the arrow.

x_head The x coordinate of the head of the arrow.

y_head The y coordinate of the head of the arrow.

z_head For 2D plots this is ignored. For 3D plots, the z coordinate of the tail of the arrow.

option The option for the arrow.

Description

This function adds an arrow to a plot. The arrow points from (x_tail, y_tail, z_tail) to (x_head, y_head, z_head) . These coordinates are specified using the same coordinate system as the curves of the plot.

An arrow is not counted as a curve. Therefore, it does not affect the number of legends added by **CPlot::legend**(legend, num).

The optional argument *option* of string type with the following values can be used to fine tune the arrow based on the argument for set arrow command of the gnuplot.

```
{ {nohead | head | backhead | heads}
  {size <length>, <angle>{, <backangle>}}
  {filled | empty | nofilled}
  {front | back}
  { {linestyle | ls <line_style>}
```

```
| {linetype | lt <line_type>}
| {linewidth | lw <line_width>} } } }
```

Specifying ‘nohead’ produces an arrow drawn without a head—a line segment. This gives you yet another way to draw a line segment on the plot. By default, an arrow has a head at its end. Specifying ‘backhead’ draws an arrow head at the start point of the arrow while ‘heads’ draws arrow heads on both ends of the line. Not all terminal types support double-ended arrows.

Head size can be controlled by ‘size <length>, <angle>’ or ‘size <length>, <angle>, <backangle>’, where ‘<length>’ defines length of each branch of the arrow head and ‘<angle>’ the angle (in degrees) they make with the arrow. ‘<length>’ is in x-axis units; this can be changed by ‘first’, ‘second’, ‘graph’, ‘screen’, or ‘character’ before the <length>; see ‘coordinates’ for details. ‘<backangle>’ only takes effect when ‘filled’ or ‘empty’ is also used. Then, ‘<backangle>’ is the angle (in degrees) the back branches make with the arrow (in the same direction as ‘<angle>’). The ‘fig’ terminal has a restricted backangle function. It supports three different angles. There are two thresholds: Below 70 degrees, the arrow head gets an indented back angle. Above 110 degrees, the arrow head has an acute back angle. Between these thresholds, the back line is straight.

Specifying ‘filled’ produces filled arrow heads (if heads are used). Filling is supported on filled-polygon capable terminals, otherwise the arrow heads are closed but not filled. The same result (closed but not filled arrow head) is reached by specifying ‘empty’.

If ‘front’ is given, the arrow is written on top of the graphed data. If ‘back’ is given (the default), the arrow is written underneath the graphed data. Using ‘front’ will prevent an arrow from being obscured by dense data.

The ‘linetype’ is followed by an integer index representing the line type for drawing. The line type varies depending on the terminal type used (see **CPlot::outputType**). Typically, changing the line type will change the color of the line or make it dashed or dotted. All terminals support at least six different line types. The ‘linewidth’ is followed by a scaling factor for the line width. The line width is ‘linewidth’ multiplied by the default width. Typically the default width is one pixel.

Example 1

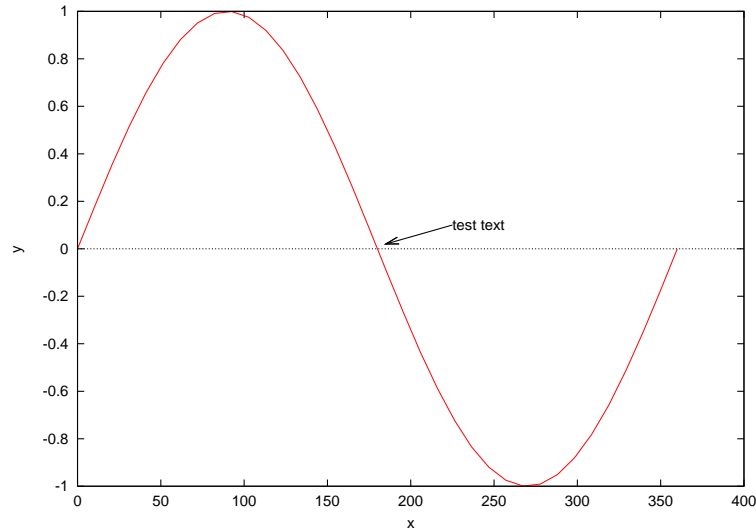
Compare with output for examples in **CPlot::data2D()** and **CPlot::data2DCurve()**.

```
/* File: arrow_1.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    lindata(0, 360, x);
    y = sin(x*M_PI/180);
    plot.arrow(225, 0.1, 185, 0.02);
    plot.text("test text", PLOT_TEXT_LEFT, 225, 0.1);
    plot.data2D(x, y);
    plot.plotting();
}
```

Output

**Example 2**

Compare with the output for examples in **CPlot::data3D()** and **CPlot::data3DSurface()**.

```

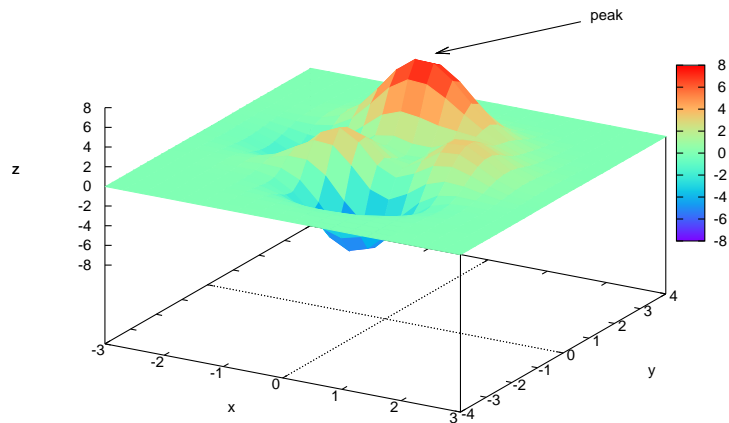
/* File: arrow_2.ch */
#include <math.h>
#include <chplot.h>

int main() {
    double x[20], y[30], z[600];
    int i, j;
    class CPlot plot;

    lindata(-3, 3, x);
    lindata(-4, 4, y);
    for(i=0; i<20; i++) {
        for(j=0; j<30; j++) {
            z[30*i+j] = 3*(1-x[i])*(1-x[i])*exp(-(x[i]*x[i])-(y[j]+1)*(y[j]+1))
                - 10*(x[i]/5 - x[i]*x[i]*x[i]-pow(y[j],5))*exp(-x[i]*x[i]-y[j]*y[j])
                - 1/3*exp(-(x[i]+1)*(x[i]+1)-y[j]*y[j]));
        }
    }
    plot.data3D(x, y, z);
    plot.arrow(2, 3, 12, 0, 2, 8);
    plot.text("peak", PLOT_TEXT_LEFT, 2.1, 3.15, 12.6);
    plot.plotting();
}

```

Output



Example 3

```

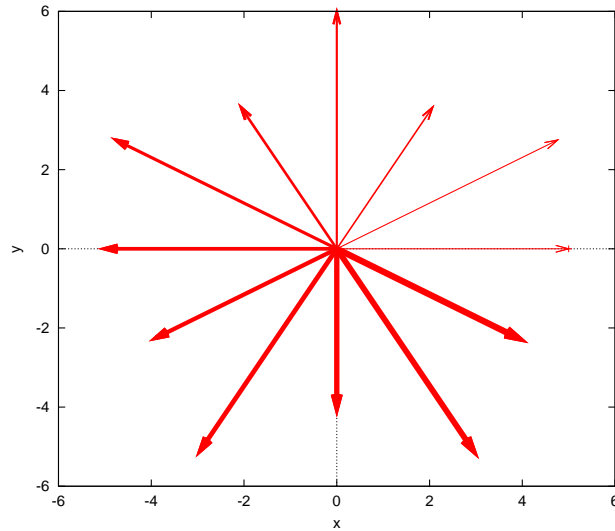
/* File: arrow_3.ch */
#include <math.h>
#include <chplot.h>

int main() {
    double complex z[12];
    char option[64];
    int i;
    class CPlot plot;

    for (i=0; i<12; i++) {
        real(z[i]) = (5+sin(150*i*M_PI/180))*cos(30*i*M_PI/180);
        imag(z[i]) = (5+sin(150*i*M_PI/180))*sin(30*i*M_PI/180);
        sprintf(option, "linetype 1 linewidth %d", i+1);
        plot.arrow(0, 0, 0, real(z[i]), imag(z[i]), 0, option);
    }
    plot.axisRange(PLOT_AXIS_XY, -6, 6); /* one point cannot do autorange */
    plot.point(real(z[0]), imag(z[0]), 0); /* CPlot::arrow() itself is not a data set */
    plot.plotType(PLOT_PLOTTYPE_POINTS, 0);
    plot.sizeRatio(-1);
    plot.plotting();
}

```

Output

**Example 4**

See an example on page 207 for **CPlot::plotType()** on how `option` is used in comparison with the plot type **PLOT_PLOTTYPE_VECTORS**.

See Also

CPlot::circle(), **CPlot::data2D()**, **CPlot::outputType()**, **CPlot::plotType()**, **CPlot::point()**, **CPlot::polygon()**, **CPlot::rectangle()**.

CPlot::autoScale

Synopsis

```
#include <chplot.h>
void autoScale(int axis, int flag);
```

Purpose

Set autoscaling of axes on or off.

Return Value

None.

Parameters

axis The axis to be set. Valid values are:

- PLOT_AXIS_X** Select the x axis only.
- PLOT_AXIS_X2** Select the x2 axis only.
- PLOT_AXIS_Y** Select the y axis only.
- PLOT_AXIS_Y2** Select the y2 axis only.
- PLOT_AXIS_Z** Select the z axis only.
- PLOT_AXIS_XY** Select the x and y axes.
- PLOT_AXIS_XYZ** Select the x, y, and z axes.

flag Enable or disable auto scaling.

PLOT_ON The option is enabled.

PLOT_OFF The option is disabled.

Description

Autoscaling of the axes can be **PLOT_ON** or **PLOT_OFF**. Default is **PLOT_ON**. If autoscaling for an axis is disabled, an axis range of [-10:10] is used.

Example

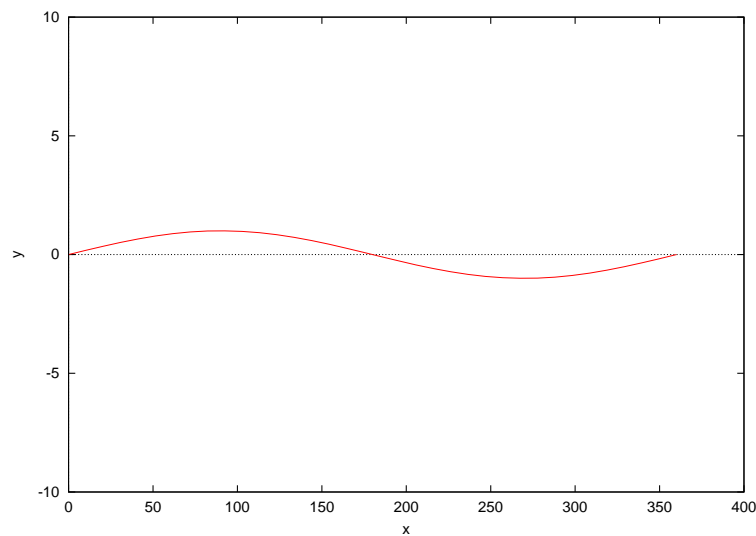
Compare with the output for examples in **CPlot::data2D()** and **CPlot::data2DCurve()**.

```
/* File: autoScale.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    lindata(0, 360, x);
    y = sin(x*M_PI/180);
    plot.autoScale(PLOT_AXIS_Y, PLOT_OFF);
    plot.data2D(x, y);
    plot.plotting();
}
```

Output



CPlot::axis

Synopsis

```
#include <chplot.h>
```

```
void axis(int axis, int flag, ... /* [ int linetype, int linewidth, char linecolor] */);
```

Syntax

```
axis(axis, flag);
```

```
axis(axis, linetype, linewidth, linecolor);
```

Purpose

Enable or disable drawing of x-y axis on 2D plots.

Return Value

None.

Parameters

axis The *axis* parameter can take one of the following values:

PLOT_AXIS_X Select the x axis only.

PLOT_AXIS_X2 Select the x2 axis only.

PLOT_AXIS_Y Select the y axis only.

PLOT_AXIS_Y2 Select the y2 axis only.

PLOT_AXIS_XY Select the x and y axes.

flag This parameter can be set to:

PLOT_ON Enable drawing of the specified axis.

PLOT_OFF Disable drawing of the specified axis.

linetype An integer index representing the line type for drawing. Use the same value for different curves so that each curve with the same style, and same color by default.

linewidth A scaling factor for the line width. The line width is *linewidth* multiplied by the default width.

linecolor color for the line.

Description

Enable or disable the drawing of the x-y axes on 2D plots. By default, the x and y axes are drawn with the dotted line using the `linetype 0`.

The line type, width, and color for x-y axes can be changed in the same manner as the member function **CPlot::lineType()**. See **CPlot::lineType()** on how to setup the arguments *linetype*, *linewidth*, and *linecolor*.

Example

Compare with the output for the example in **CPlot::axisRange()**.

```
/* File: axis.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

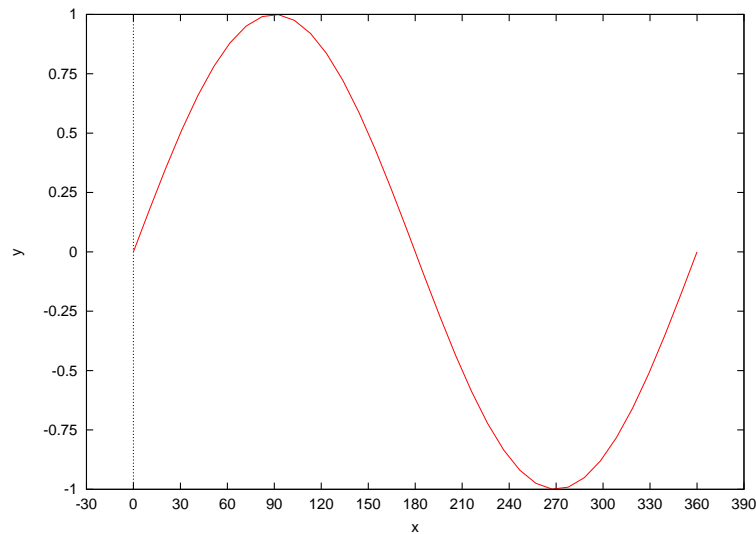
    lndata(0, 360, x);
    y = sin(x*M_PI/180);           // Y-axis data.
```

```

    plot.axisRange(PLOT_AXIS_X, -30, 390);
    plot.ticsRange(PLOT_AXIS_X, 30, -30, 390);
    plot.axisRange(PLOT_AXIS_Y, -1, 1);
    plot.ticsRange(PLOT_AXIS_Y, .25, -1, 1);
    plot.axis(PLOT_AXIS_X, PLOT_OFF);
    plot.data2D(x, y);
    plot.plotting();
}

```

Output



Example 2

Compare with the output from Example 1.

```

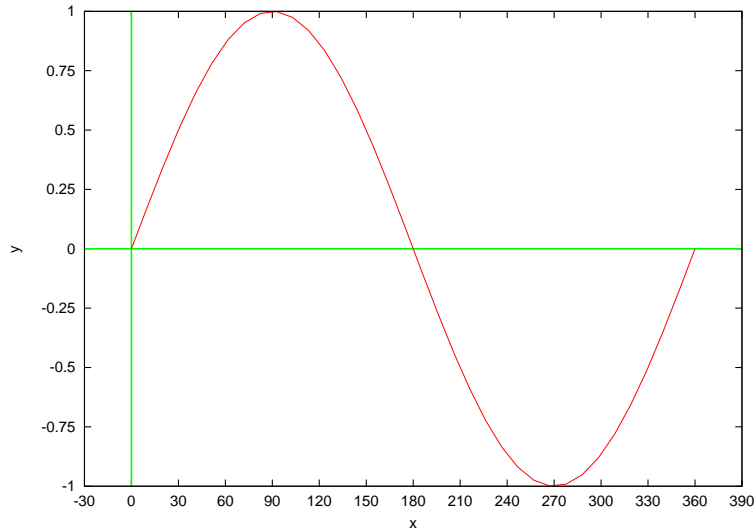
/* File: axis_2.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    int linetype=1, linewidth=3;
    char linecolor[]="green";
    class CPlot plot;

    lindata(0, 360, x);
    y = sin(x*M_PI/180); // Y-axis data.
    plot.axisRange(PLOT_AXIS_X, -30, 390);
    plot.ticsRange(PLOT_AXIS_X, 30, -30, 390);
    plot.axisRange(PLOT_AXIS_Y, -1, 1);
    plot.ticsRange(PLOT_AXIS_Y, .25, -1, 1);
    plot.axis(PLOT_AXIS_XY, linetype, linewidth, linecolor);
    plot.data2D(x, y);
    plot.plotting();
    return 0;
}

```

Output

**See Also**

CPlot::lineType().

CPlot::axisRange

Synopsis

```
#include <chplot.h>
```

```
void axisRange(int axis, double minimum, double maximum, ... /* [double incr] */);
```

Syntax

```
axisRange(axis, minimum, maximum)
```

```
axisRange(axis, minimum, maximum, incr)
```

Purpose

Specify the range for an axis.

Return Value

None.

Parameters

axis The *axis* parameter can take one of the following values:

PLOT_AXIS_X Select the x axis only.

PLOT_AXIS_X2 Select the x2 axis only.

PLOT_AXIS_Y Select the y axis only.

PLOT_AXIS_Y2 Select the y2 axis only.

PLOT_AXIS_Z Select the z axis only.

PLOT_AXIS_XY Select the x and y axes.

PLOT_AXIS_XYZ Select the x, y, and z axes.

minimum The axis minimum.

maximum The axis maximum.

incr The increment between tic marks. By default or when *incr* is 0, the increment between tic marks is calculated internally.

Description

The range for an axis can be explicitly specified with this function. Autoscaling for the specified axis is disabled and any previously specified labeled tic-marks are overridden. If the axis is logarithmic specified by the member function **scaleType()**, the increment will be used as a multiplicative factor.

Note that

```
plot.axisRange(axis, min, max, incr);
```

is obsolete. Use

```
plot.axisRange(axis, min, max);
plot.ticsRange(axis, incr);
```

or

```
plot.ticsRange(axis, incr, start);
plot.ticsRange(axis, incr, start, end);
```

Example 1

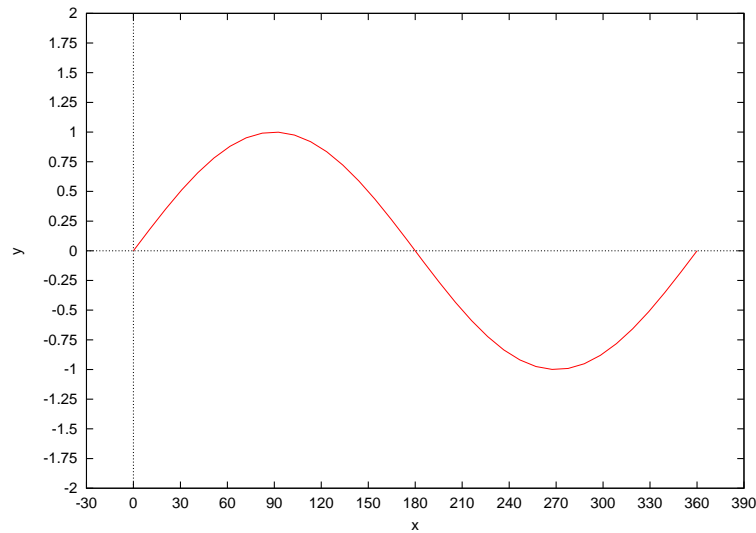
Compare with the output for the examples in **CPlot::axis()** and **CPlot::grid()**.

```
/* File: axisRange.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    lindata(0, 360, x);
    y = sin(x*M_PI/180); // Y-axis data.
    plot.axisRange(PLOT_AXIS_X, -30, 390);
    plot.ticsRange(PLOT_AXIS_X, 30);
    plot.axisRange(PLOT_AXIS_Y, -2, 2);
    plot.ticsRange(PLOT_AXIS_Y, 0.25);
    plot.data2D(x, y);
    plot.plotting();
}
```

Output

**Example 2**

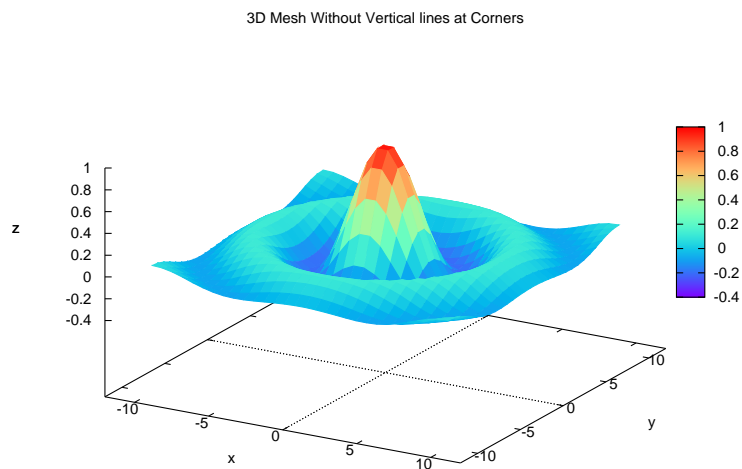
3D mesh plot without vertical lines at the corners. Compare with the output for examples in `CPlot::data3D()` and `CPlot::data3DSurface()`.

```
/* File: axisRange_2.ch */
#include <chplot.h>
#include <math.h>

int main() {
    double x[30], y[30], z[900];
    double r;
    int i, j;
    class CPlot plot;

    lindata(-10, 10, x);
    lindata(-10, 10, y);
    for(i=0; i<30; i++) {
        for(j=0; j<30; j++) {
            r = sqrt(x[i]*x[i]+y[j]*y[j]);
            z[30*i+j] = sin(r)/r;
        }
    }
    plot.data3D(x, y, z);
    plot.axisRange(PLOT_AXIS_XY, -12, 12);
    plot.title("3D Mesh Without Vertical lines at Corners");
    plot.plotting();
}
```

Output



Example 3

X axis range is reversed.

```

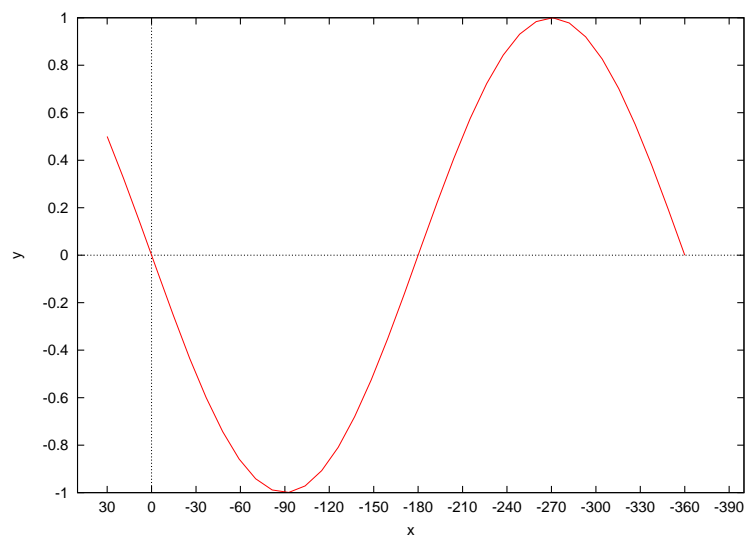
/* File: axisRange_3.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    lndata(-360, 30, x);
    y = sin(x*M_PI/180); // Y-axis data.
    plot.axisRange(PLOT_AXIS_X, 50, -400);
    plot.ticsRange(PLOT_AXIS_X, -30);
    plot.data2D(x, y);
    plot.plotting();
}

```

Output



See Also**CPlot::ticsRange()**, **CPlot::autoScale()**, **CPlot::ticsLabel()**.

CPlot::axes

Synopsis**#include** <chplot.h>**void axes**(int *num*, string_t *axes*);**Syntax****axes**(*num*, *axes*)**Purpose**

Specify the axes for a data set.

Return Value

None.

Parameters*num* The data set the axes are specified.*axes* The axes for the specified data set.**Description**

A **CPlot** class lets you use each of the four borders – x (bottom), x2 (top), y (left) and y2 (right) – as an independent axis. The **axes()** function lets you choose which pair of axes a given set of data specified in *num* is plotted against.

There are four possible sets of axes available. The argument *axes* is used to select the axes for which a particular line should be scaled. The string "x1y1" refers to the axes on the bottom and left; "x2y2" to those on the top and right; "x1y2" to those on the bottom and right; and "x2y1" to those on the top and left.

Other options such as labels and ranges can be specified other member functions by selecting a proper axis with one of following macros.

PLOT_AXIS_X Select the x axis only.**PLOT_AXIS_X2** Select the x2 axis only.**PLOT_AXIS_Y** Select the y axis only.**PLOT_AXIS_Y2** Select the y2 axis only.**PLOT_AXIS_Z** Select the z axis only.**PLOT_AXIS_XY** Select the x and y axes.**PLOT_AXIS_XYZ** Select the x, y, and z axes.**Example 1**

```

/* File: axes.ch */
#include <chplot.h>
#include <math.h>

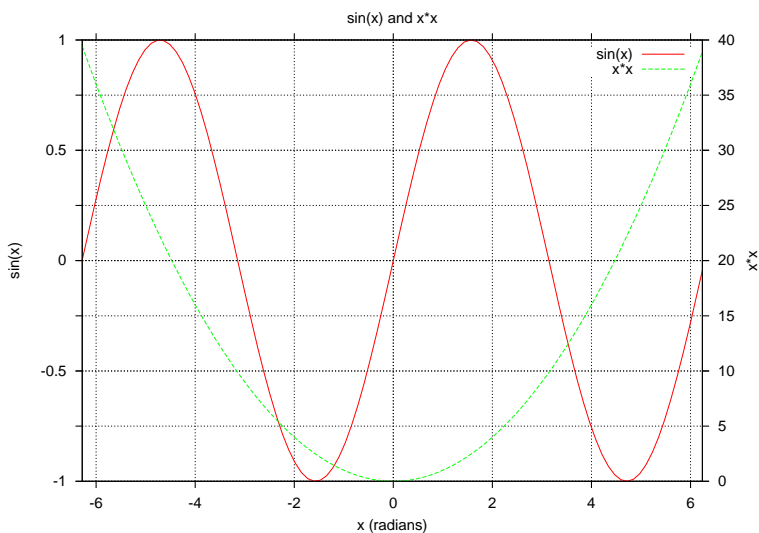
double func1(double x) {
    return sin(x);
}
double func2(double x) {
    return x*x;
}

int main() {
    class CPlot plot;
    double x0= -6.28, xf = 6.24;
    int num = 100;

    plot.title("sin(x) and x*x");
    plot.label(PLOT_AXIS_X, "x (radians)");
    plot.label(PLOT_AXIS_Y, "sin(x)");
    plot.label(PLOT_AXIS_Y2, "x*x");
    plot.axisRange(PLOT_AXIS_X, x0, xf);
    plot.ticsMirror(PLOT_AXIS_X, PLOT_ON);
    plot.axisRange(PLOT_AXIS_X2, x0, xf);
    plot.ticsRange(PLOT_AXIS_Y, 0.5);
    plot.border(PLOT_BORDER_ALL, PLOT_ON);
    plot.ticsMirror(PLOT_AXIS_Y2, PLOT_ON);
    plot.ticsDirection(PLOT_TICS_OUT);
    plot.tics(PLOT_AXIS_Y2, PLOT_ON);
    plot.ticsRange(PLOT_AXIS_Y2, 5);
    plot.grid(PLOT_ON, "x y y2");
    plot.func2D(x0, xf, num, func1);
    plot.legend("sin(x)", 0);
    plot.func2D(x0, xf, num, func2);
    plot.legend("x*x", 1);
    plot.axes(1, "x2y2");
    plot.plotting();
}

```

Output



Example 2

See an example on page 211 for plot type `PLOT_PLOTTYPE_FINANCEBARS` in `CPlot::plotType()`. In that example, the `y2` axis is used to display different data.

See Also

`CPlot::legend()`, `CPlot::axisRange()`.

CPlot::backgroundColor

Synopsis

```
#include <chplot.h>
void backgroundColor(string_t color1);
```

Purpose

Specify the color of the background.

Return Value

None.

Parameters

color The color for the background in a string using a color name or a hexadecimal value.

Description

By default, the background of a plot is white. This member function `plot.backgroundColor()` is used to change the color of the background.

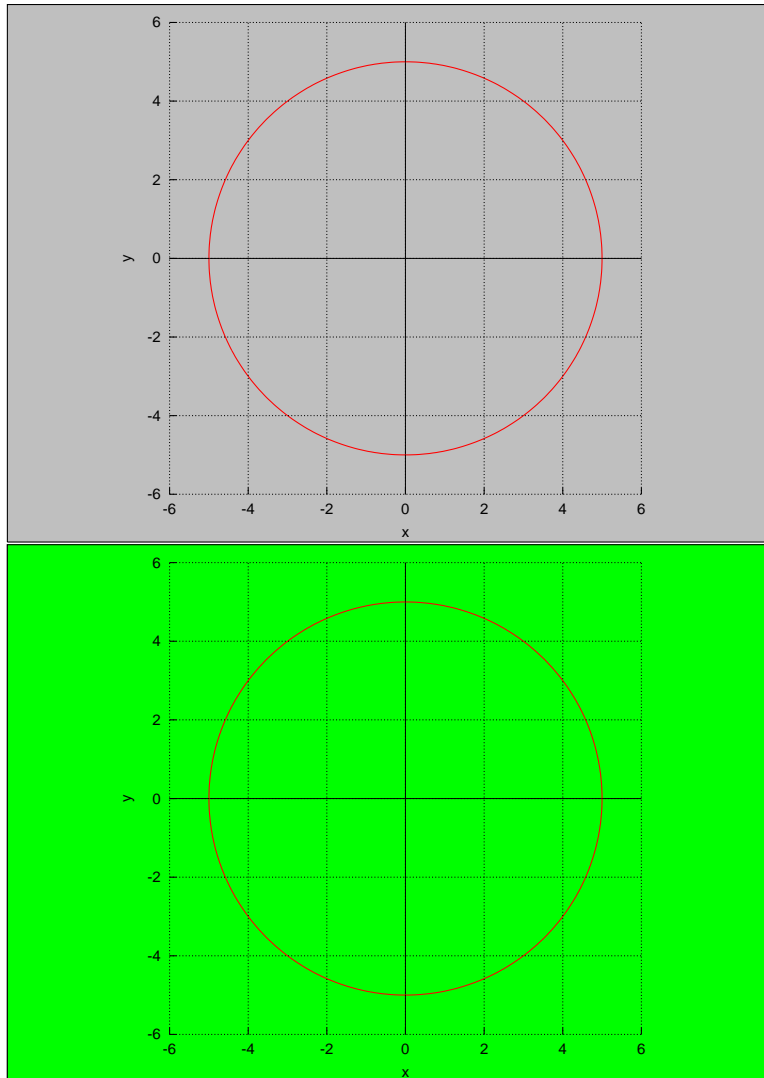
Example

```
/* File: backgroundColor.ch */
#include <chplot.h>
CPlot plot;

plot.circle(0, 0, 5);
plot.sizeRatio(1);

plot.backgroundColor("grey");
plot.plotting();
plot.backgroundColor("#00FF00"); // green
plot.plotting();
```

Output



See Also

CPlot::lineType().

CPlot::barGraph

Synopsis

```
#include <chplot.h>
```

```
void barGraph(double x[:], ... /* [int n,] double x2[:], ...,*/);
```

Syntax

```
barGraph(x);
```

```
barGraph(x, n);
```

```
barGraph(x1, x2);
```

```
barGraph(x1, x2, x3);
```

Purpose

Draw a bar graph.

Return Value

None.

Parameters

x An array containing the values for the bar graph.

n The number of elements in array *x*.

x2 An array containing the values for the second category for the bar graph.

x3 An array containing the values for the third category for the bar graph.

Description

This member function draws a bar graph using the data in the array *x*. It draws a series of vertical rectangles, one next to each other separated with a space, as needed to represent the values of each category in the data.

The member function **plot.barGraphCategory()** is used to specify the category on the x-axis.

The member function **plot.barGraphColor()** can be used to specify the color for each rectangle on the bar graph.

By default, horizontal lines are drawn for each ticks on the y-axis. The member function **plot.barGraphValue()** can be used to suppress the horizontal lines and place a category value on the top of each rectangle.

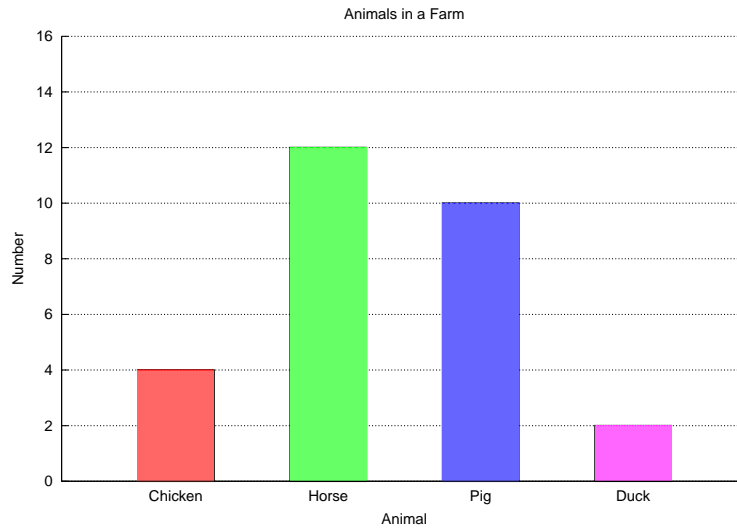
More than one data set can be added using additional arguments of arrays. In this case, the member function **plot.barGraphColor()** can be used to specify the color for each data set and the member function **plot.barGraphLegend()** can be used to add legends for each data set.

Example 1

```
/* File: barGraph.ch */
#include <chplot.h>
CPlot plot;
double x[] = {4, 12, 10, 2};

plot.title("Animals in a Farm");
plot.label(PLOT_AXIS_X, "Animal");
plot.label(PLOT_AXIS_Y, "Number");
plot.barGraphCategory("Chicken", "Horse", "Pig", "Duck");
plot.barGraph(x);
plot.axisRange(PLOT_AXIS_Y, 0, 16);
plot.plotting();
```

Output



Example 2

```

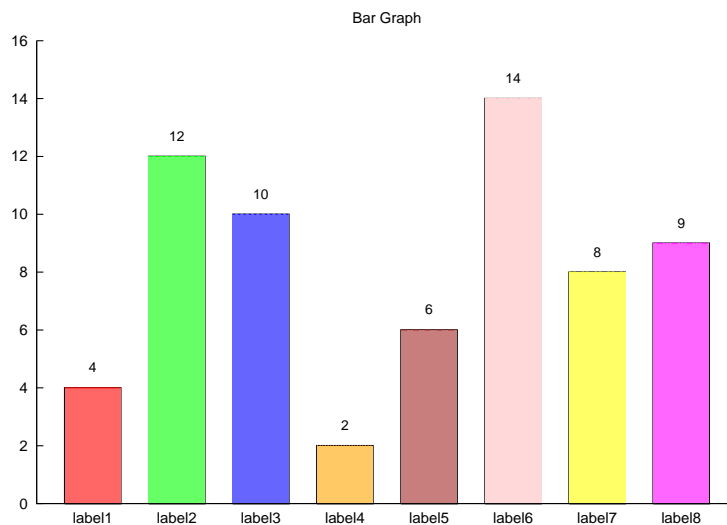
/* File: barGraphColor.ch
   with more than 5 items */
#include <chplot.h>
CPlot plot;
double x[] = {4, 12, 10, 2, 6, 14, 8, 9};

plot.title("Bar Graph");
plot.label(PLOT_AXIS_X, "");
plot.label(PLOT_AXIS_Y, "");
plot.barGraphColor("red", "green", "blue", "orange", "brown", "pink", "yellow", "magenta");
plot.barGraphValue();
plot.barGraphCategory("label1", "label2", "label3", "label4", "label5",
                      "label6", "label7", "label8");

plot.barGraph(x);
plot.grid(PLOT_OFF);
plot.axisRange(PLOT_AXIS_Y, 0, 16);
plot.plotting();

```

Output



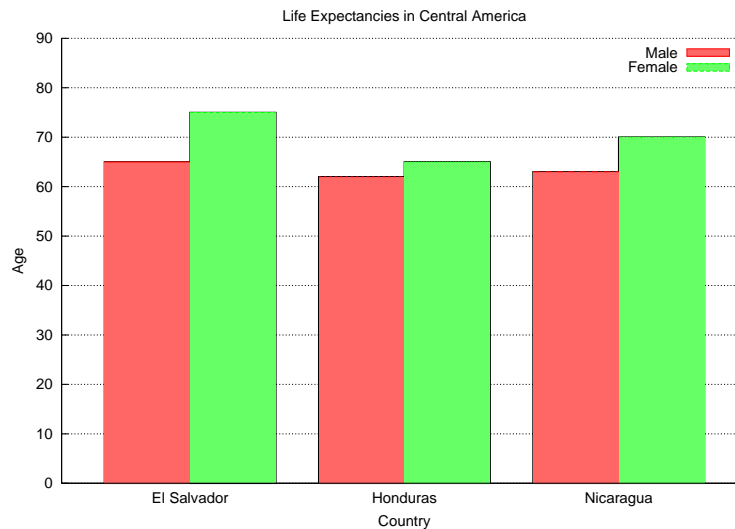
Example 3

```

/* File: barGraphLegend.ch */
#include <chplot.h>
CPlot plot;
double male[] = {65, 62, 63}; // life expectancy for male
double female[] = {75, 65, 70}; // life expectancy for female

plot.title("Life Expectancies in Central America");
plot.label(PLOT_AXIS_X, "Country");
plot.label(PLOT_AXIS_Y, "Age");
plot.barGraphCategory("El Salvador", "Honduras", "Nicaragua");
plot.barGraphColor("red", "green");
plot.barGraphLegend("Male", "Female");
plot.barGraph(male, female);
plot.axisRange(PLOT_AXIS_Y, 0, 90);
plot.plotting();

```

Output**Example 4**

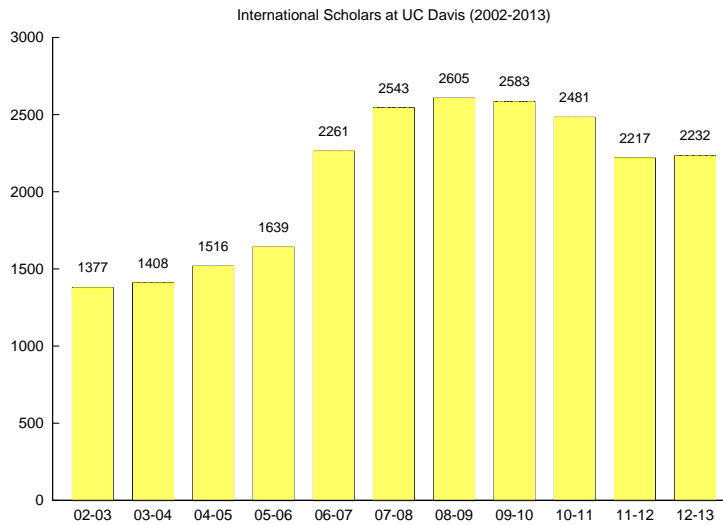
```

/* File: barGraphValue.ch
   with more than 5 items */
#include <chplot.h>
CPlot plot;
double x[] = {1377, 1408, 1516, 1639, 2261, 2543, 2605, 2583, 2481, 2217, 2232};

plot.title("International Scholars at UC Davis (2002-2013)");
plot.label(PLOT_AXIS_X, "");
plot.label(PLOT_AXIS_Y, "");
plot.barGraphValue();
plot.barGraphColor("yellow", "yellow", "yellow", "yellow", "yellow", "yellow",
                  "yellow", "yellow", "yellow", "yellow", "yellow");
plot.barGraphCategory("02-03", "03-04", "04-05", "05-06", "06-07",
                      "07-08", "08-09", "09-10", "10-11", "11-12", "12-13");

plot.barGraph(x);
plot.grid(PLOT_OFF);
plot.axisRange(PLOT_AXIS_Y, 0, 3000);
plot.plotting();

```


Output**See Also**

CPlot::barGraphCategory(), CPlot::barGraphColor(), CPlot::barGraphLegend(), CPlot::barGraphValue(), CPlot::histogram().

CPlot::barGraphCategory

Synopsis

```
#include <chplot.h>
```

```
void barGraphCategory(string_t item1[], ... /* string_t item2);
```

Syntax

```
barGraphCategory(item1);
```

```
barGraphCategory(item1, item2);
```

```
barGraphCategory(item1, item2, item3);
```

Purpose

Label each category for a bar graph.

Return Value

None.

Parameters

item1 The label for the first category of the bar graph.

item2 The label for the second category of the bar graph.

item3 The label for the third category of the bar graph.

Description

The member function **plot.barGraph** draws a bar graph using the data in an array. The member function

`plot.barGraphCategory()` is used to specify the each category in the data set.

Example

See `CPlot::barGraph()`.

See Also

`CPlot::barGraph()`, `CPlot::barGraphColor()`, `CPlot::barGraphLegend()`, `CPlot::barGraphValue()`.

CPlot::barGraphColor

Synopsis

```
#include <chplot.h>
```

```
void barGraphColor(string_t color1[:], ... /* string_t color2);
```

Syntax

```
barGraphColor(color1);
```

```
barGraphColor(color1, color2);
```

```
barGraphColor(color1, color2, color3);
```

Purpose

Specify the color for each category for a bar graph.

Return Value

None.

Parameters

color1 The color for the first category of the bar graph.

color2 The color for the second category of the bar graph.

color3 The color for the third category of the bar graph.

Description

The member function `plot.barGraph` draws a bar graph using the data in an array. The member function `plot.barGraphColor()` is used to specify the color for each rectangle on the bar graph. For multiple data sets, the member function `plot.barGraphColor()` is used to specify the color for each data set.

Example

See `CPlot::barGraph()`.

See Also

`CPlot::barGraph()`, `CPlot::barGraphCategory()`, `CPlot::barGraphLegend()`, `CPlot::barGraphValue()`.

CPlot::barGraphLegend

Synopsis

```
#include <chplot.h>
```

```
void barGraphLegend(string_t legend1[:], ... /* string_t legend2);
```

Syntax

```
barGraphLegend(legend1);
barGraphLegend(legend1, legend2);
barGraphLegend(legend1, legend2, legend3);
```

Purpose

Specify the legend for each data set for a bar graph with multiple data sets.

Return Value

None.

Parameters

legend1 The legend for the first data set in the bar graph.

legend2 The legend for the second data set in the bar graph.

legend3 The legend for the third data set in the bar graph.

Description

The member function **plot.barGraph** draws a bar graph using the data in an array. For a bar graph with multiple data sets only, the member function **plot.barGraphLegend()** is used to specify the legend for each data set of the bar graph.

Example

See **CPlot::barGraph()**.

See Also

CPlot::barGraph(), **CPlot::barGraphCategory()**, **CPlot::barGraphColor()**, **CPlot::barGraphValue()**.

CPlot::barGraphValue

Synopsis

```
#include <chplot.h>
void barGraphValue();
```

Purpose

Add a value for each category on a bar graph.

Return Value

None.

Parameters

None.

Description

The member function **plot.barGraph** draws a bar graph using the data in an array. By default, horizontal

lines are drawn for each ticks on the y-axis. The member function **plot.barGraphValue()** can be used to suppress the horizontal lines and place a category value on the top of each rectangle.

Example

See **CPlot::barGraph()**.

See Also

CPlot::barGraph(), **CPlot::barGraphCategory()**, **CPlot::barGraphColor()**, **CPlot::barGraphLegend()**.

CPlot::barSize

Synopsis

```
#include <chplot.h>
void barSize(double size);
```

Syntax

```
barSize(size)
```

Purpose

Set the size of error bars.

Return Value

None.

Parameters

size The size of error bars. The value for *size* is in the range [0, 1.0].

Description

This function specifies the size of error bars for a plot type of **PLOT_PLOTTYPE_XERRORBARS**, **PLOT_PLOTTYPE_XYERRORBARS**, and **PLOT_PLOTTYPE_YERRORBARS**. specified by member function **CPlot::plotType()**. The value for *size* is in the range [0, 1.0]. The value 0 is for no error bar. The value 1.0 is for the largest error bar.

Example

See an example on page 211 for the plot type **PLOT_PLOTTYPE_XYERRORBARS** in **CPlot::plotType()**.

See Also

CPlot::boxWidth(), **CPlot::plotType()**.

CPlot::border()

Synopsis

```
#include <chplot.h>
void border(int location, int flag);
```

Purpose

Enable or disable display of a bounding box around the plot.

Return Value

None.

Parameter*location* The location of the effected border segment.**PLOT_BORDER_BOTTOM** The bottom of the plot.**PLOT_BORDER_LEFT** The left side of the plot.**PLOT_BORDER_TOP** The top of the plot.**PLOT_BORDER_RIGHT** The right side of the plot.**PLOT_BORDER_ALL** All sides of the plot.*flag* Enable or disable display of a box around the boundary or the plot.**PLOT_ON** Enable drawing of the box.**PLOT_OFF** Disable drawing of the box.**Description**

This function turns the display of a border around the plot on or off. By default, the border is drawn on all sides (left, bottom, top, and right) for 2D plots, and on all sides for 3D plots.

Example 1

Compare with the example output in **CPlot::ticsMirror()**.

```

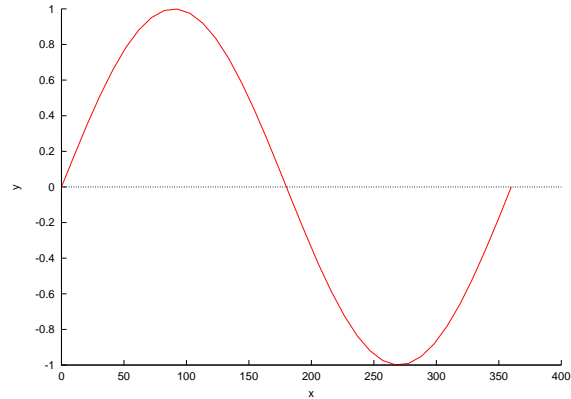
/* File: border.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    lindata(0, 360, x);
    y = sin(x*M_PI/180);
    plot.data2D(x, y);
    //or plot.border(PLOT_BORDER_TOP | PLOT_BORDER_RIGHT, PLOT_OFF);
    plot.border(PLOT_BORDER_TOP, PLOT_OFF);
    plot.border(PLOT_BORDER_RIGHT, PLOT_OFF);
    plot.ticsMirror(PLOT_AXIS_XY, PLOT_OFF);
    plot.plotting();
}

```

Output

**Example 2**

Compare with the example output in `CPlot::polarPlot()`.

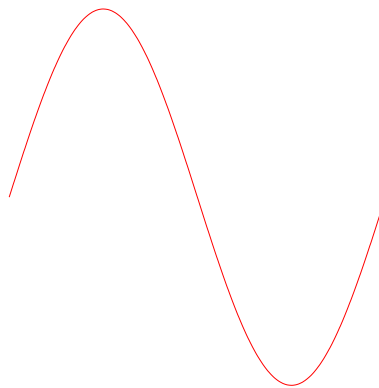
```

/* File: border_2.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 360;
    array double theta[numpoints], r[numpoints];
    class CPlot plot;

    lindata(0, 2*M_PI, theta);
    r = sin(theta);           // Y-axis data.
    plot.data2D(theta, r);
    plot.label(PLOT_AXIS_XY, NULL);
    plot.sizeRatio(1);
    plot.border(PLOT_BORDER_ALL, PLOT_OFF);
    plot.tics(PLOT_AXIS_XY, PLOT_OFF);
    plot.axis(PLOT_AXIS_XY, PLOT_OFF);
    plot.plotting();
}

```

Output**See Also**

`CPlot::ticsMirror()`.

CPlot::borderOffsets

Synopsis

```
#include <chplot.h>
```

```
void borderOffsets(double left, double right, double top, double bottom);
```

Purpose

Specify the size of offsets around the data points of a 2D plot in the same units as the plot axis.

Return Value

None.

Parameters

left The offset on the left side of the plot.

right The offset on the right side of the plot.

top The offset on the top of the plot.

bottom The offset on the bottom of the plot.

Description

For 2D plots, this function specifies the size of offsets around the data points of an autoscaled plot. This function can be used to create empty space around the data. The *left* and *right* arguments are specified in the units of the x-axis. The *top* and *bottom* arguments are specified in the units of the y-axis.

Example

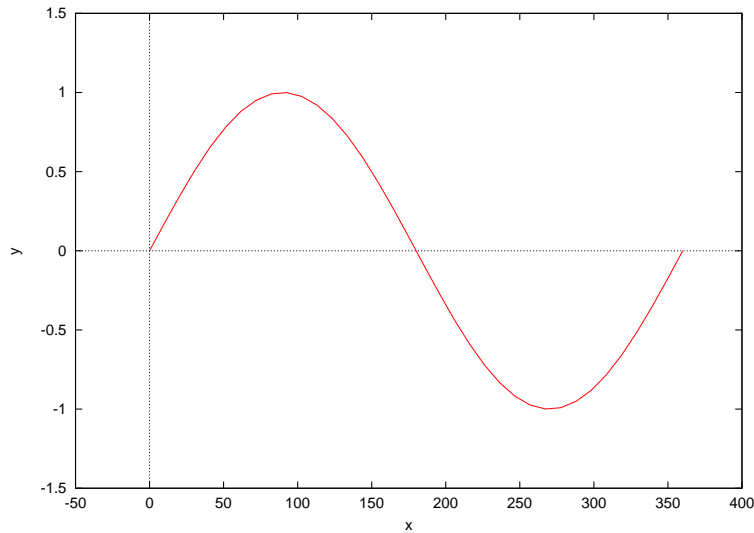
Compare with the output for examples in **CPlot::data2D()** and **CPlot::data2DCurve()**.

```
/* File: borderOffsets.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    lindata(0, 360, x);
    y = sin(x*M_PI/180);
    plot.data2D(x, y);
    plot.borderOffsets(30, 30, .5, .25);
    plot.plotting();
}
```

Output

**See Also**

CPlot::autoScale(), CPlot::axisRange(), CPlot::ticsLabel(), CPlot::margins().

CPlot::boxAndWhisker

Synopsis

```
#include <chplot.h>
```

```
void boxAndWhisker(double x[], ... /* int n */);
```

Syntax

```
boxAndWhisker(x);
```

```
boxAndWhisker(x, n);
```

Purpose

Draw a Box-and-Whiskers plot.

Return Value

None.

Parameters

x An array containing the values used for the Box-and-Whiskers plot.

n The number of elements in array *x*.

Description

This member function draws a Box-and-Whiskers plot for the data in the array *x*. The vertical lines represents the minimum value, lower quartile (the median of the lower half), median, upper quartile (the median of the upper half), and maximum value for the data set.

Example

```
/* File: boxAndWhisker.ch */
```



```
#include <chplot.h>

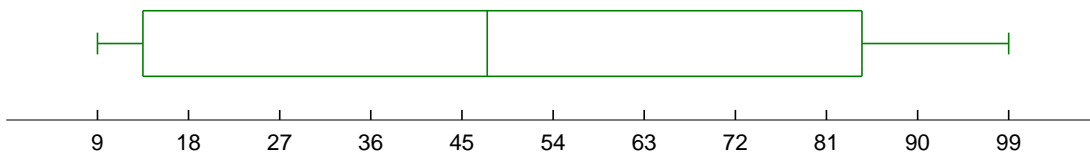
double x[] = {53, 78, 11, 94, 16, 9, 42, 91, 37, 11, 76, 99};
double min, q1, med, q3, max, range, iqr;
CPlot plot;

quartiles(x, min, q1, med, q3, max, range, iqr);
printf("          min = %.2lf\n", min);
printf("      lower quartile = %.2lf\n", q1);
printf("middle quartile (median) = %.2lf\n", med);
printf("          upper quartile = %.2lf\n", q3);
printf("          max = %.2lf\n", max);
printf("          range = %.2lf\n", range);
printf("interquartile range = %.2lf\n", iqr);

//plot.title("Box-and-Whisker Plot");
plot.title("");
plot.label(PLOT_AXIS_X, "");
plot.boxAndWhisker(x);
plot.plotting();
```

Output

```
          min = 9.00
      lower quartile = 13.50
middle quartile (median) = 47.50
          upper quartile = 84.50
          max = 99.00
          range = 90.00
interquartile range = 71.00
```



See Also

quartiles().

CPlot::boxBorder**Synopsis****#include** <chplot.h>**void boxBorder**(int num, ... /* [int linetype] */);**Syntax****boxBorder**(num)**boxBorder**(num, linetype)**Purpose**

Enable or disable drawing of a border for a plot of box type.

Return Value

None.

Parameters*num* The data set of plot type of box or filled curve to be bounded with a border. If *num* is -1, the border handling will be applied to all boxes and filled curves for the plot.*linetype* The *linetype* parameter specifies the line type used to draw bounding lines.**Description**

This function specifies how borders for boxes and filled curves will be handled for a plot type of **PLOT_PLOTTYPE_BOXES**, **PLOT_PLOTTYPE_BOXERRORBARS**, **PLOT_PLOTTYPE_BOXXYERRORBARS**, **PLOT_PLOTTYPE_CANDLESTICKS**, and **PLOT_PLOTTYPE_FILLEDCURVES** specified by member function **CPlot::plotType()**. By default, the box or filled curve is bounded by a solid line of the current line type. If the line type is not specified by the function call of **boxBorder(num)**, no bounding lines are drawn.

Example

In this example, each box has a width of 30 specified by **CPlot::boxWidth()**. The first box is by default empty. The second box is filled with a solid color. The third one is filled with a solid color of density 0.25. The border of this box uses the line type for the first box specified by **CPlot::boxBorder()**. The fourth box is filled with a pattern. The border of this box is empty. The four boxes are repeated twice by an outer loop with index *j*.

```

/* File: boxBorder.ch */
#include <math.h>
#include <chplot.h>
#include <numeric.h>

#define N 4
#define M 2
int main() {
    int i, j, k, numpoints = 4, linetype, linewidth, patternnum=1;
    array double x[numpoints], y[numpoints];
    class CPlot plot;
    double begin;

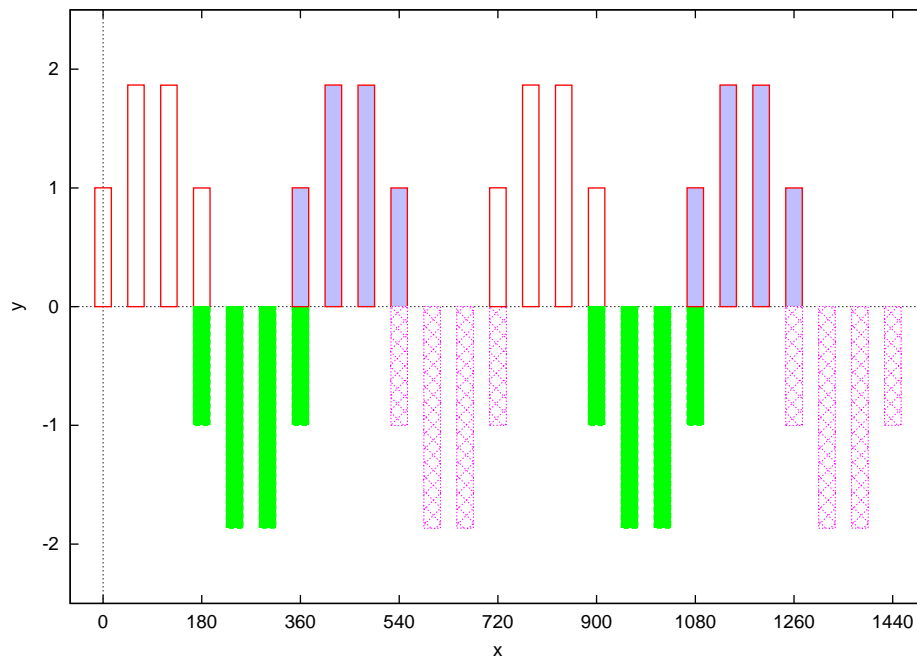
```

```

begin = 0.01;
for(j=0; j< M; j++) {
  for(i=0; i<N; i++) {
    lndata(begin, begin+180-0.01, x);
    y = sin(x*M_PI/180);
    for(k=0; k<numpoints; k++) {
      y[k] += sign(y[k])*1;
    }
    plot.data2D(x, y);
    linetype = i+1;
    linewidth = 2;
    plot.plotType(PLOT_PLOTTYPE_BOXES, i+j*N);
    plot.lineType(i+j*N, linetype, linewidth);
    plot.boxWidth(30);
    if(i==1)
      plot.fillStyle(i+j*N, PLOT_FILLSTYLE_SOLID);
    else if(i==2) {
      plot.fillStyle(i+j*N, PLOT_FILLSTYLE_SOLID, 0.25);
      plot.boxBorder(i+j*N, 1);
    }
    else if(i==3) {
      plot.boxBorder(i+j*N);
      plot.fillStyle(i+j*N, PLOT_FILLSTYLE_PATTERN, patternnum);
    }
    begin += 180;
  }
}
plot.axisRange(PLOT_AXIS_X, -60, N*M*180+60);
plot.ticsRange(PLOT_AXIS_X, 180, 0, N*M*180);
plot.axisRange(PLOT_AXIS_Y, -2.5, 2.5);
plot.plotting();
return 0;
}

```

Output

**See Also**

CPlot::fillStyle(), CPlot::boxWidth(), CPlot::boxFill(), CPlot::plotType().

CPlot::boxFill

Synopsis

```
#include <chplot.h>
```

```
void boxFill(int num, int style, ... /* [double density], int patternnum */);
```

Syntax

```
boxFill(num, PLOT_BOXFILL_EMPTY)
```

```
boxFill(num, PLOT_BOXFILL_SOLID)
```

```
boxFill(num, PLOT_BOXFILL_SOLID, density)
```

```
boxFill(num, PLOT_BOXFILL_TRANSPARENT_SOLID, density)
```

```
boxFill(num, PLOT_BOXFILL_PATTERN)
```

```
boxFill(num, PLOT_BOXFILL_PATTERN, patternnum)
```

```
boxFill(num, PLOT_BOXFILL_TRANSPARENT_PATTERN, patternnum)
```

Purpose

Fill a box or filled curve with a solid color or pattern.

This function is obsolete. Use **CPlot::fillStyle()**.

Return Value

None.

Parameters

num The data set of plot type of box or filled curve to be filled. If *num* is -1, the fill style will be applied to

all boxes and filled curves for the plot.

style The *style* parameter can take one of the following values:

PLOT_BOXFILL_EMPTY Do not fill the box. The default is to have an empty box.

PLOT_BOXFILL_SOLID Fill the boxes or filled curves with a solid color.

PLOT_BOXFILL_TRANSPARENT_SOLID Fill the boxes or filled curves with a transparent solid color.

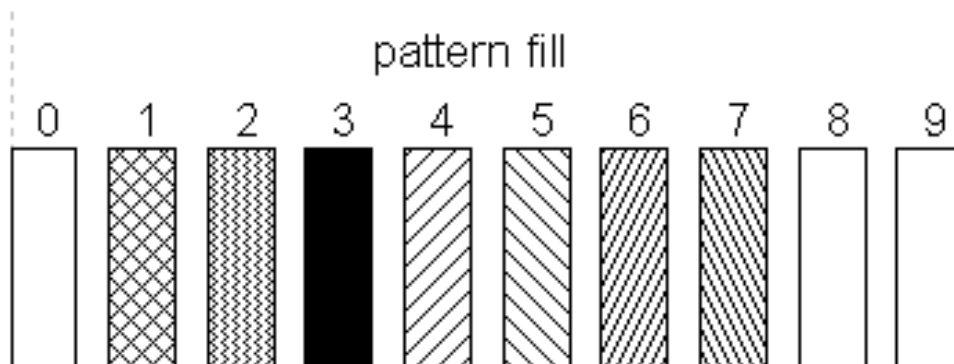
PLOT_BOXFILL_PATTERN Fill the boxes or filled curves with a pattern.

PLOT_BOXFILL_TRANSPARENT_PATTERN Fill the boxes or filled curves with a transparent pattern.

density The density of solid color in the range of [0, 1.0]. If the value for *density* is 0.0, the box is empty. If it is 1.0, the inner area is of the same color as the current line type. If no *density* parameter is given, it defaults to 1.0.

In the case of transparent solid fill areas, the density parameter is interpreted as an alpha value; that is, density 0 is fully transparent, density 1 is fully opaque.

patternnum The parameter *patternnum* option causes filling to be done with a fill pattern supplied by the terminal driver. The kind and number of available fill patterns depend on the terminal driver. If multiple datasets using filled boxes are plotted, the pattern cycles through all available pattern types, starting from *patternnum*, much as the line type cycles for multiple line plots. The available patterns in Windows are shown below.



In the case of transparent pattern fill, the background of the pattern is either fully transparent or fully opaque.

Description

This function specifies how boxes and filled curves are filled with a solid color or pattern. The fill style can be applied to plot types of **PLOT_PLOTTYPE_BOXES**, **PLOT_PLOTTYPE_BOXERRORBARS**, **PLOT_PLOTTYPE_BOXXYERRORBARS**, **PLOT_PLOTTYPE_CANDLESTICKS**, and **PLOT_PLOTTYPE_FILLEDCURVES** specified by member function **CPlot::plotType()**.

There may be additional limitations on the creation or viewing of graphs containing transparent fill areas. For example, the png terminal can only use transparent fill if the "truecolor" option is set by

```
plot.outputType(PLOT_OUTPUTTYPE_FILE, "png truecolor", "filename.png")
```

Postscript files only support transparent pattern-fill areas.

Portability

Not all platforms support `PLOT_BOXFILL_TRANSPARENT_SOLID` and `PLOT_BOXFILL_TRANSPARENT_PATTERN`.

Example

See an example on page 44 for `CPlot::boxBorder()`.

See Also

`CPlot::boxBorder()`, `CPlot::boxWidth()`, `CPlot::fillStyle()`, `CPlot::plotType()`.

CPlot::boxWidth

Synopsis

```
#include <chplot.h>
void boxWidth(double width);
```

Syntax

```
boxWidth(width)
```

Purpose

Set the width for a box.

Return Value

None.

Parameters

width The width of boxes to be drawn.

Description

This function specifies the width of boxes for a plot type of `PLOT_PLOTTYPE_BOXES`, `PLOT_PLOTTYPE_BOXERRORBARS`, `PLOT_PLOTTYPE_BOXXYERRORBARS`, and `PLOT_PLOTTYPE_CANDLESTICKS` specified by member function `CPlot::plotType()`.

By default, adjacent boxes are extended in width until they touch each other. A different default width may be specified using the this member function.

The parameter `width` for the `boxwidth` is interpreted as being a number of units along the current x axis. If the x axis is a log-scale then the value of `boxwidth` is absolute only at `x=1`; this physical width is maintained everywhere along the axis (i.e. the boxes do not become narrower the value of `x` increases). If the range spanned by a log scale x axis is far from `x=1`, some experimentation may be required to find a useful value of `boxwidth`.

The default is superseded by explicit width information taken from an extra data column for a plot type of `PLOT_PLOTTYPE_BOXES`, `PLOT_PLOTTYPE_BOXERRORBARS`, or `PLOT_PLOTTYPE_BOXXYERRORBARS`. In a four-column data set, the fourth column will be interpreted as the box width unless the `width` is set to `-2.0`, in which case the width will be calculated automatically.

To set the box width to automatic for four-column data, use

```
plot.boxwidth(-2);
```

The same effect can be achieved with the option of using keyword for member function **CPlot::dataFile()** as follows.

```
plot.dataFile(datafile, "using 1:2:3:4:(-2)");
```

To set the box width to an absolute value of 30, use

```
plot.boxWidth(2);
```

Example

See an example on page 44 for **CPlot::boxBorder()**.

See Also

CPlot::boxBorder(), **CPlot::fillStyle()**, **CPlot::boxFill()**, **CPlot::plotType()**.

CPlot::changeViewAngle

Synopsis

```
#include <chplot.h>
```

```
void changeViewAngle(double x_rot, double z_rot);
```

Purpose

Change the viewpoint for a 3D plot.

Return Value

None.

Parameters

x_rot The angle of rotation for the plot (in degrees) along an axis horizontal axis of the screen.

z_rot The angle of rotation for the plot (in degrees) along an axis perpendicular to the screen.

Description

This function provides rotation of a 3D plot. *x_rot* and *z_rot* are bounded by the range [0:180] and the range [0:360] respectively. By default, 3D plots are displayed with *x_rot* = 60° and *z_rot* = 30°.

Example

Compare with the output for examples in **CPlot::data3D()** and **CPlot::data3DSurface()**.

```
/* File: changeViewAngle.ch */
#include <math.h>
#include <chplot.h>

int main() {
    array double x[20], y[20], z[400];
    int i, j;
    class CPlot plot;

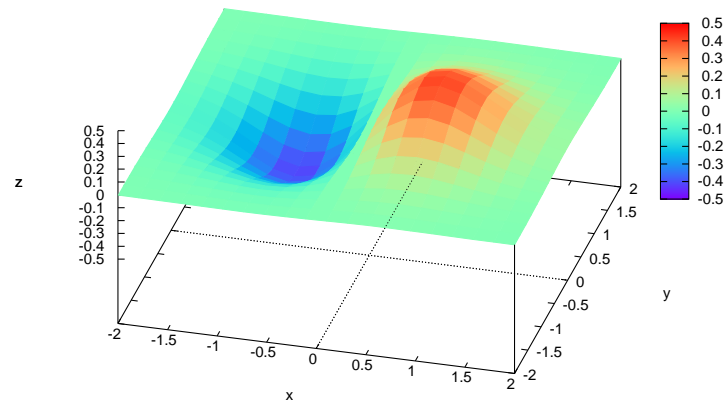
    lndata(-2, 2, x);
    lndata(-2, 2, y);
    for (i=0; i<20; i++) {
```

```

        for(j=0; j<20; j++) {
            z[i*20+j] = x[i]*exp(-x[i]*x[i]-y[j]*y[j]);
        }
    }
    plot.data3D(x, y, z);
    plot.changeViewAngle(45, 15);
    plot.plotting();
}

```

Output



Example

```

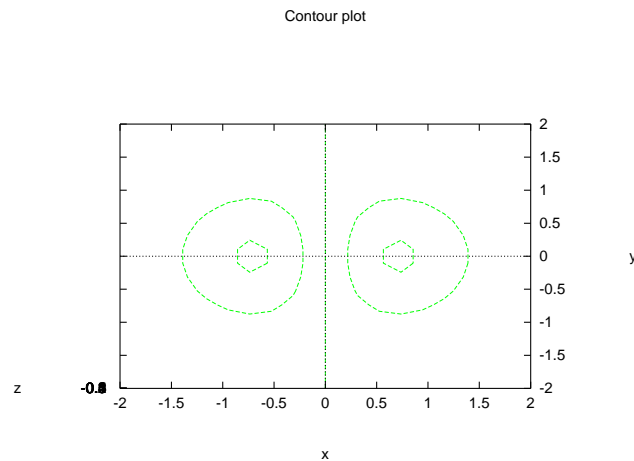
/* File: changeViewAngle_2.ch */
#include <chplot.h>
#include <math.h>

int main() {
    array double x[20], y[20], z[400];
    int datasetnum =0, i, j;
    class CPlot plot;

    lindata(-2, 2, x);
    lindata(-2, 2, y);
    for (i=0; i<20; i++) {
        for(j=0; j<20; j++) {
            z[i*20+j] = x[i]*exp(-x[i]*x[i]-y[j]*y[j]);
        }
    }
    plot.data3D(x, y, z);
    plot.changeViewAngle(0,0);
    plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum);
    plot.contourMode(PLOT_CONTOUR_BASE);
    plot.showMesh(PLOT_OFF);
    plot.title("Contour plot");
    plot.plotting();
}

```

Output

**See Also**

CPlot::data3D(), **CPlot::data3DCurve()**, **CPlot::data3DSurface()**.

CPlot::circle

Synopsis

```
#include <chplot.h>
```

```
int circle(double x_center, double y_center, double r);
```

Purpose

Add a circle to a 2D plot.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x_center The x coordinate of the center of the circle.

y_center The y coordinate of the center of the circle.

r The radius of the circle.

Description

This function adds a circle to a 2D plot. It is a convenience function for creation of geometric primitives. For rectangular plots, *x* and *y* are the coordinates of the center of the circle and *r* is the radius of the circle, all specified in units of the x and y axes. For polar plots, the location of the center of the circle is specified in polar coordinates where *x* is θ and *y* is *r*.

Handling Outline and Filled Color of Objects in CPlot

The member functions **CPlot::arcSector()**, **CPlot::arcSegment()**, **CPlot::circle()**, **CPlot::ellipse()**, **CPlot::rectangle()**, **CPlot::rectanglexy()**, **CPlot::quad()**, **CPlot::polygon()**, **CPlot::regularPolygon()**, **CPlot::star()**, and **CPlot::triangle()** can add corresponding objects to a 2D plot. These objects are not counted as a data set for later calls to **CPlot::legend()** and **CPlot::plotType()**.

By default, the line width of the outline of an object is 2. The color of the outline of an object is red, green, blue, magenta, cyan, yellow, black, orange, or grey. The color will be cycled if more than eight objects are added. The default line width and color of the outline of an object can be changed by **CPlot::strokeWidth()** and **CPlot::strokeColor()**, respectively. The outline of an object can be disabled by **CPlot::noStrokeColor()**.

By default, the interior of an object is not filled. The interior of an object can be filled with a solid color by **CPlot::fillColor()**. The opacity of the filled color can be set by **CPlot::fillOpacity()** in the range from 0 to 1, with 0 for complete transparency and 1 for solid color. The interior of an object can also be filled with a pattern by **CPlot::fillPattern()** when a background color other than white is used. The member function **CPlot::noFill()** will disable the filled color.

Example 1

```
/* File: circle.ch */
#include <chplot.h>

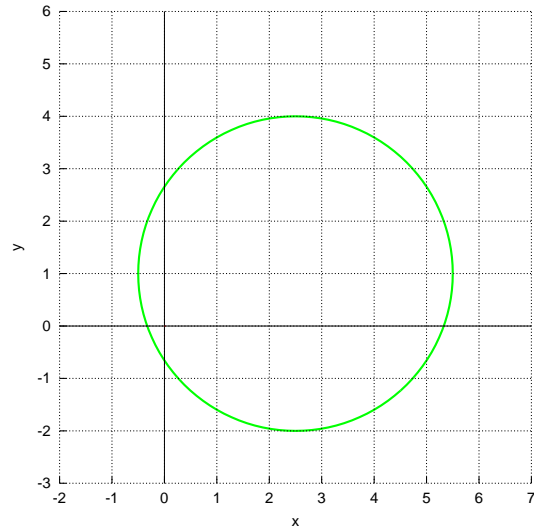
int main() {
    double x_center = 2.5, y_center = 1.0, r = 3;
    int line_width = 4;
    class CPlot plot;

    plot.strokeColor("green");
    plot.strokeWidth(line_width);
    plot.circle(x_center, y_center, r);
    plot.circle(x_center, y_center, r-1);

    plot.strokeColor("blue");
    plot.strokeWidth(10);
    plot.fillColor("yellow");
    plot.circle(x_center, y_center, r-2);

    plot.sizeRatio(1);
    plot.axisRange(PLOT_AXIS_X, -2, 7);
    plot.axisRange(PLOT_AXIS_Y, -3, 6);
    plot.plotting();
}
```

Output



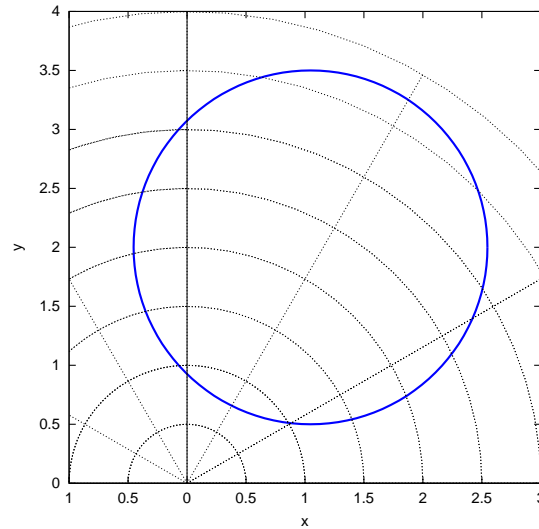
Example 2

```
/* File: circle_2.ch */
#include <chplot.h>
#include <math.h>

int main() {
    double x_center = M_PI/3, y_center = 2.0, r = 1.5;
    class CPlot plot;

    plot.polarPlot(PLOT_ANGLE_RAD); /* Polar coordinate system */
    plot.label(PLOT_AXIS_X, "x");
    plot.label(PLOT_AXIS_Y, "y");
    plot.strokeColor("blue");
    plot.strokeWidth(4);
    plot.circle(x_center, y_center, r);
    plot.sizeRatio(1);
    plot.axisRange(PLOT_AXIS_X, -1, 3);
    plot.axisRange(PLOT_AXIS_Y, 0, 4);
    plot.border(PLOT_BORDER_ALL, PLOT_ON);
    plot.ticsLocation(PLOT_AXIS_XY, "border");
    plot.ticsMirror(PLOT_AXIS_XY, PLOT_ON);
    plot.plotting();
}
```

Output

**See Also**

CPlot::arc(), CPlot::data2D(), CPlot::data2DCurve(), CPlot::ellipse(), CPlot::line(),
 CPlot::outputType(), CPlot::plotType(), CPlot::point(), CPlot::polygon(), CPlot::rectangle(), CPlot::star().

CPlot::circleClosed

Synopsis

```
#include <chplot.h>
void circleClosed(double x, double y, ... /* string_t color */);
```

Syntax

```
circleClosed(x, y);
circleClosed(x, y, "blue");
```

Purpose

Draw a small closed circle to a 2D plot.

Return Value

None.

Parameters

x The x coordinate of the center of the circle.

y The y coordinate of the center of the circle.

color The color of the circle.

Description

This member function adds a small closed circle to a 2D plot. This member function is commonly used in visualizing equalities and piece-wise functions.

Example 1

```

/* File: circleClosed.ch
   Plot the piecewise function  $y = x^2 - 2$  for  $x < 1$ 
                                    $y = -2x + 4$  for  $x \geq 1$ 

   for x from -3 to 5 */
#include <chplot.h>
CPlot plot;

plot.title("");
plot.label(PLOT_AXIS_X, "x");
plot.label(PLOT_AXIS_Y, "y");
plot.ticsRange(PLOT_AXIS_X, 1);
plot.ticsRange(PLOT_AXIS_Y, 1);
//plot.axisRange(PLOT_AXIS_Y, -3, 5);

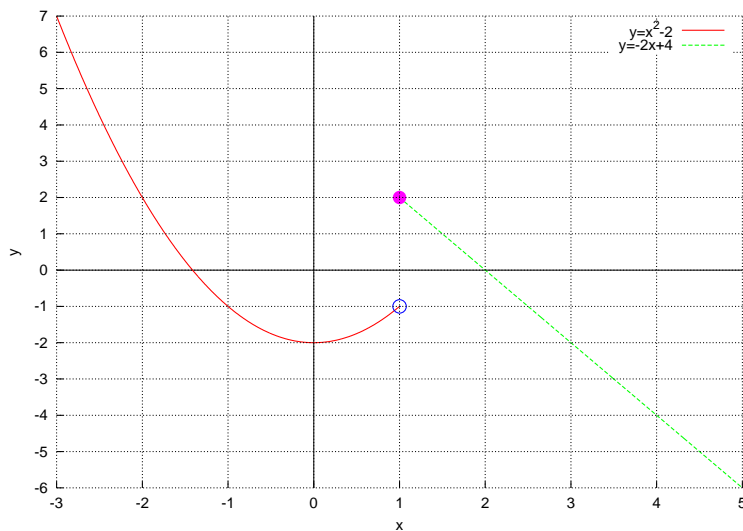
/* draw functions with the default line colors "red", "green", "blue",
   "magenta", "cyan", "brown", "light brown", "light red" */
plot.expr(-3, 1, 500, "x*x-2");
plot.legend("y=x^2-2", 0);
plot.expr(1, 5, 500, "-2*x+4");
plot.legend("y=-2x+4", 1);

/* draw an open circle */
plot.circleOpen(1, 1*1-2);
/* draw a close circle */
plot.circleClosed(1, -2*1+4);

plot.plotting();

```

Output



Example 2

See `plot.line1D()`.

See Also

`CPlot::circle`, `CPlot::circleOpen()`.

CPlot::circleOpen

Synopsis

```
#include <chplot.h>
void circleOpen(double x, double y, ... /* string_t color */);
```

Syntax

```
circleOpen(x, y);
circleOpen(x, y, "blue");
```

Purpose

Draw a small open circle to a 2D plot.

Return Value

None.

Parameters

x The x coordinate of the center of the circle.

y The y coordinate of the center of the circle.

color The color of the circle.

Description

This member function adds a small open circle to a 2D plot. This member function is commonly used in visualizing inequalities and piece-wise functions.

Example

See `plot.circleClosed()`.

See Also

`CPlot::circle()`, `CPlot::circleClosed()`.

CPlot::colorBox

Synopsis

```
#include <chplot.h>
void colorBox(int flag);
```

Purpose

Enable or disable the drawing of a color box for a 3D surface plot.

Return Value

None.

Parameter

flag This parameter can be set to:

PLOT_ON Enable the color box.

PLOT_OFF Disable the color box.

Description

Enable or disable the drawing of a color box, which contains the color scheme, i.e. the gradient of the smooth color with the maximum and minimum values of the color palette. This is applicable only for 3D surface plots. By default, the color box is drawn.

Example 1

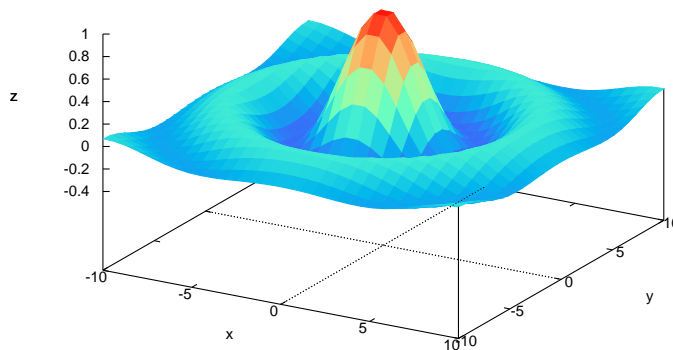
Compare with the output for the example in **CPlot::data3D()**.

```
/* File: colorBox.ch */
#include <chplot.h>
#include <math.h>
#include <numeric.h>

int main() {
    double x[30], y[30], z[900];
    double r;
    int i, j;
    class CPlot plot;

    lindata(-10, 10, x);
    lindata(-10, 10, y);
    for(i=0; i<30; i++) {
        for(j=0; j<30; j++) {
            r = sqrt(x[i]*x[i]+y[j]*y[j]);
            z[30*i+j] = sin(r)/r;
        }
    }
    plot.data3D(x, y, z);
    plot.colorBox(PLOT_OFF);
    plot.plotting();
}
```

Output



See Also

CPlot::data3D(), **CPlot::data3DCurve()**, **CPlot::data3DSurface()**, **CPlot::contourMode()**,

CPlot::plotType().

CPlot::contourLabel

Synopsis

```
#include <chplot.h>
```

```
void contourLabel(int flag);
```

Purpose

Set display of contour line elevation labels for 3D plots on or off.

Return Value

None.

Parameter

flag Enable or disable drawing of contour line labels.

PLOT_ON Enable contour labels.

PLOT_OFF Disable contour labels.

Description

Enable or disable contour line labels for 3D surface plots. labels appear with the plot legend. By default, labels for contours are not displayed.

Example 1

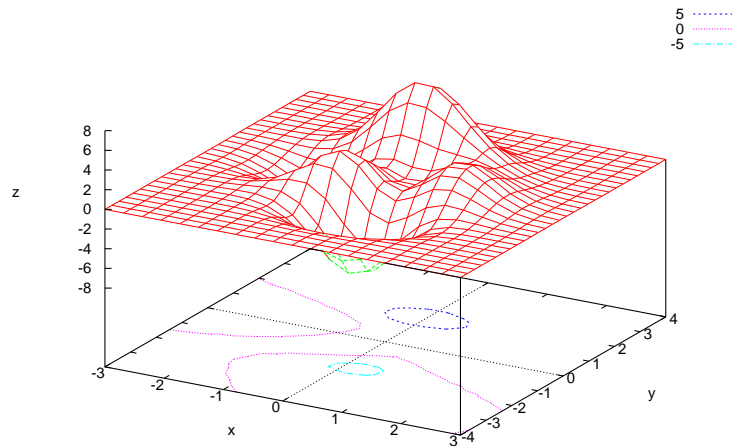
Compare with the output for examples in **CPlot::data3D()**, **CPlot::data3DSurface()**, **CPlot::contourLevels()**, and **CPlot::showMesh()**.

```
/* File: contourLabel.ch */
#include <math.h>
#include <chplot.h>

int main() {
    double x[20], y[30], z[600];
    int datasetnum = 0, i, j;
    class CPlot plot;

    lndata(-3, 3, x);
    lndata(-4, 4, y);
    for(i=0; i<20; i++) {
        for(j=0; j<30; j++) {
            z[30*i+j] = 3*(1-x[i])*(1-x[i])*exp(-(x[i]*x[i])-(y[j]+1)*(y[j]+1))
                - 10*(x[i]/5 - x[i]*x[i]*x[i]-pow(y[j], 5))*exp(-x[i]*x[i]-y[j]*y[j])
                - 1/3*exp(-(x[i]+1)*(x[i]+1)-y[j]*y[j]));
        }
    }
    plot.data3D(x, y, z);
    plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum);
    plot.contourLabel(PLOT_ON);
    plot.contourMode(PLOT_CONTOUR_BASE);
    plot.plotting();
}
```

Output



Example 2

```

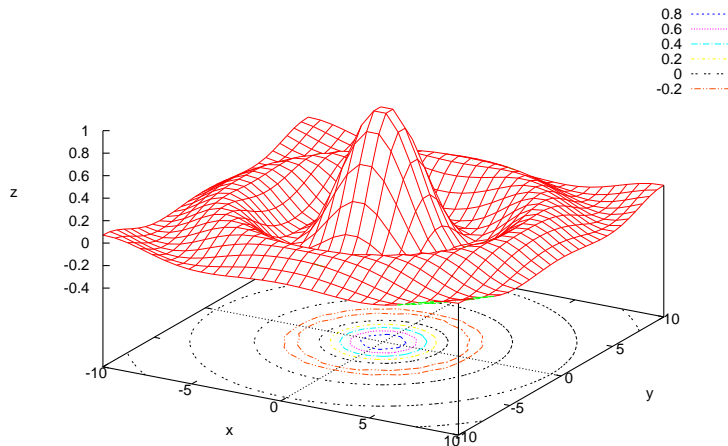
/* File: contourLabel_2.ch */
#include <math.h>
#include <chplot.h>

int main() {
    double x[30], y[30], z[900];
    double r;
    int datasetnum =0, i, j;
    class CPlot plot;

    lndata(-10, 10, x);
    lndata(-10, 10, y);
    for(i=0; i<30; i++) {
        for(j=0; j<30; j++) {
            r = sqrt(x[i]*x[i]+y[j]*y[j]);
            z[30*i+j] = sin(r)/r;
        }
    }
    plot.data3D(x, y, z);
    plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum);
    plot.contourLabel(PLOT_ON);
    plot.contourMode(PLOT_CONTOUR_BASE);
    plot.colorBox(PLOT_OFF);
    plot.plotting();
}

```

Output

**See Also**

CPlot::data3D(), CPlot::contourLevels(), CPlot::contourMode(), CPlot::legend(), CPlot::showMesh().

CPlot::contourLevels

Synopsis

```
#include <chplot.h>
```

```
void contourLevels(double levels[], ... /* [int n] */);
```

Syntax

```
contourLevels(level)
```

```
contourLevels(level, n)
```

Purpose

Set contour levels for 3D plot to be displayed at specific locations.

Return Value

None.

Parameter

levels An array of z-axis levels for contours to be drawn.

n The number of elements of array *levels*.

Description

This function allows contour levels for 3D grid data to be displayed at any desired z-axis position. The contour levels are stored in an array where the lowest contour is in the first array element.

Example 1

Compare with the output from the example in CPlot::contourLabel().

```
/* File: contourLevels.ch */
#include <math.h>
```

```

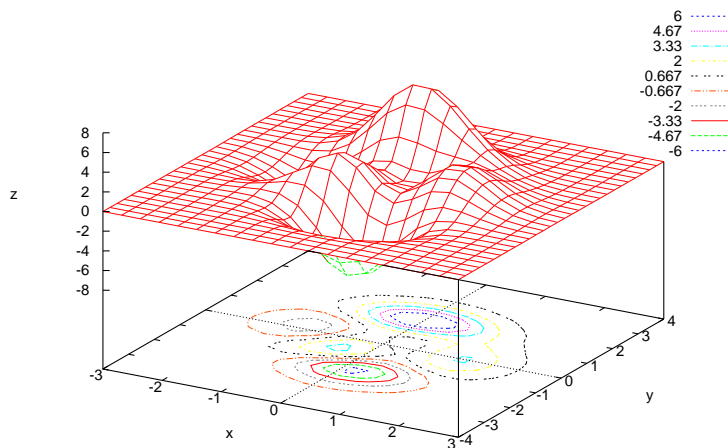
#include <chplot.h>

int main() {
    double x[20], y[30], z[600];
    double levels[10];
    int datasetnum = 0, i, j;
    class CPlot plot;

    lndata(-6, 6, levels);
    lndata(-3, 3, x);
    lndata(-4, 4, y);
    for(i=0; i<20; i++) {
        for(j=0; j<30; j++) {
            z[30*i+j] = 3*(1-x[i])*(1-x[i])*exp(-(x[i]*x[i])-(y[j]+1)*(y[j]+1))
                - 10*(x[i]/5 - x[i]*x[i]*x[i]-pow(y[j], 5))*exp(-x[i]*x[i]-y[j]*y[j])
                - 1/3*exp(-(x[i]+1)*(x[i]+1)-y[j]*y[j]);
        }
    }
    plot.data3D(x, y, z);
    plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum);
    plot.contourLabel(PLOT_ON);
    plot.contourMode(PLOT_CONTOUR_BASE);
    plot.contourLevels(levels);
    plot.colorbar(PLOT_OFF);
    plot.plotting();
}

```

Output



Example 2

```

/* File: contourLevels_2.ch */
#include <math.h>
#include <chplot.h>

int main() {
    double x[30], y[30], z[900];
    double levels[6];
    double r;

```

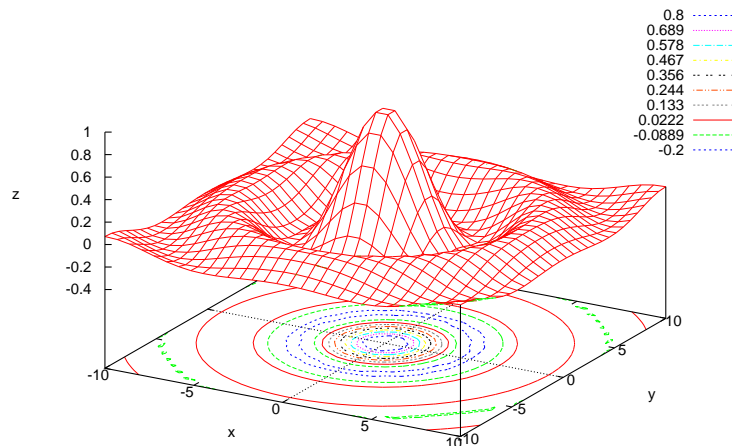
```

int datasetnum =0, i, j;
class CPlot plot;

lindata(-10, 10, x);
lindata(-10, 10, y);
lindata(-0.2, 0.8, levels);
for(i=0; i<30; i++) {
    for(j=0; j<30; j++) {
        r = sqrt(x[i]*x[i]+y[j]*y[j]);
        z[30*i+j] = sin(r)/r;
    }
}
plot.data3D(x, y, z);
plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum);
plot.contourLabel(PLOT_ON);
plot.contourMode(PLOT_CONTOUR_BASE);
plot.contourLevels(levels);
plot.colorbar(PLOT_OFF);
plot.plotting();
}

```

Output



See Also

CPlot::data3D(), CPlot::data3DCurve(), CPlot::data3DSurface(), CPlot::contourLabel(), CPlot::contourMode(), CPlot::showMesh().

CPlot::contourMode

Synopsis

```

#include <chplot.h>
void contourMode(int mode);

```

Purpose

Selects options for the drawing of contour lines in 3D plots.

Return Value

None.

Parameter

mode The following contour modes are available and can be combined using the logical or (|) operator:

PLOT_CONTOUR_BASE Contour lines drawn on the xy plane.

PLOT_CONTOUR_SURFACE Contour lines drawn on the surface of the plot.

Description

This option is available for display of 3D grid data. For the plot type of `PLOT_PLOTTYPE_LINES`, the positions of the contour levels are determined internally unless explicitly set using `CPlot::contourLevels()`. The **PLOT_CONTOUR_SURFACE** option does not work with hidden line removal. The hidden lines are removed by default. It can be disabled by member function `CPlot::removeHiddenLine()`.

For the plot type of `PLOT_PLOTTYPE_SURFACES`, the contour mode of **PLOT_CONTOUR_SURFACE** will add a projected map on the xy plane. the contour mode of **PLOT_CONTOUR_SURFACE** will have a projected map on the xy plane only.

Example

```

/* File: contourMode.ch */
#include <chplot.h>
#include <math.h>

int main() {
    double x[16], y[16], z[256];
    double r;
    int i, j;
    class CPlot subplot, *spl;

    lndata(-10, 10, x);
    lndata(-10, 10, y);
    for(i=0; i<16; i++) {
        for(j=0; j<16; j++) {
            r = sqrt(x[i]*x[i]+y[j]*y[j]);
            z[16*i+j] = sin(r)/r;
        }
    }
    subplot.subplot(2,3);
    spl = subplot.getSubplot(0,0);
    spl->data3D(x, y, z);
    spl->plotType(PLOT_PLOTTYPE_LINES, 0);
    spl->contourMode(PLOT_CONTOUR_BASE);
    spl->removeHiddenLine(PLOT_OFF);
    spl->label(PLOT_AXIS_XYZ, NULL);
    spl->colorBox(PLOT_OFF);
    spl->title("PLOT_CONTOUR_BASE");

    spl = subplot.getSubplot(0,1);
    spl->data3D(x, y, z);
    spl->plotType(PLOT_PLOTTYPE_LINES, 0);
    spl->contourMode(PLOT_CONTOUR_SURFACE);
    spl->removeHiddenLine(PLOT_OFF);
    spl->label(PLOT_AXIS_XYZ, NULL);
}

```

```
spl->colorBox(PLOT_OFF);
spl->title("PLOT_CONTOUR_SURFACE");

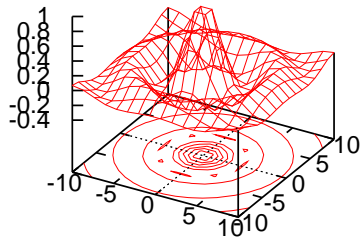
spl = subplot.getSubplot(0,2);
spl->data3D(x, y, z);
spl->plotType(PLOT_PLOTTYPE_LINES, 0);
spl->contourMode(PLOT_CONTOUR_BASE|PLOT_CONTOUR_SURFACE);
spl->removeHiddenLine(PLOT_OFF);
spl->label(PLOT_AXIS_XYZ, NULL);
spl->colorBox(PLOT_OFF);
spl->title("PLOT_CONTOUR_BASE|PLOT_CONTOUR_SURFACE");

spl = subplot.getSubplot(1,0);
spl->data3D(x, y, z);
spl->contourMode(PLOT_CONTOUR_SURFACE);
spl->label(PLOT_AXIS_XYZ, NULL);
spl->colorBox(PLOT_OFF);
spl->title("PLOT_CONTOUR_SURFACE");

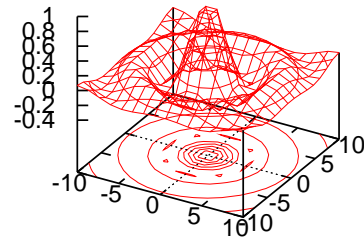
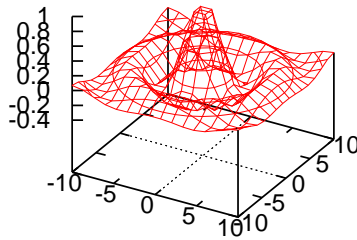
spl = subplot.getSubplot(1,1);
spl->data3D(x, y, z);
spl->contourMode(PLOT_CONTOUR_BASE);
spl->label(PLOT_AXIS_XYZ, NULL);
spl->colorBox(PLOT_OFF);
spl->title("PLOT_CONTOUR_BASE");
subplot.plotting();
}
```

Output

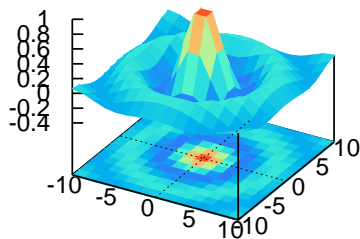
PLOT_CONTOUR_BASE



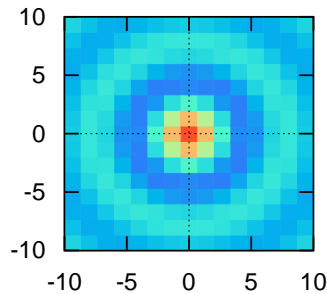
PLOT_CONTOUR_SURFACE|PLOT_CONTOUR_BASE|PLOT_CONTOUR_SURFACE



PLOT_CONTOUR_SURFACE



PLOT_CONTOUR_BASE

**See Also**

CPlot::data3D(), CPlot::data3DCurve(), CPlot::data3DSurface(), CPlot::contourLevels(),
CPlot::removeHiddenLine(), CPlot::showMesh().

CPlot::coordSystem

Synopsis

```
#include <chplot.h>
```

```
void coordSystem(int coord_system, ... /* [int angle_unit] */);
```

Syntax

```
coordSystem(coord_system)
```

```
coordSystem(coord_system, angle_unit)
```

Purpose

Select coordinate system for 3D plots.

Return Value

None.

Parameters

coord_system The coordinate system can be set to any of the following:

PLOT_COORD_CARTESIAN Cartesian coordinate system. Each data point consists of three values (x,y,z).

PLOT_COORD_SPHERICAL Spherical coordinate system. Each data point consists of three values (θ, ϕ, r).

PLOT_COORD_CYLINDRICAL Cylindrical coordinates. Each data point consists of three values (θ, z, r).

angle_unit an optional parameter to specify the unit for measurement of an angular position in **PLOT_COORD_SPHERICAL** and **PLOT_COORD_CYLINDRICAL** coordinate systems. The following options are available:

PLOT_ANGLE_DEG Angles measured in degree.

PLOT_ANGLE_RAD Angles measured in radian.

Description

This function selects the coordinate system for 3D plots. For a spherical coordinate system, points are mapped to Cartesian space by:

$$\begin{aligned}x &= r \cos(\theta) \cos(\phi) \\y &= r \sin(\theta) \cos(\phi) \\z &= r \sin(\phi)\end{aligned}$$

For a cylindrical coordinate system, points are mapped to Cartesian space by:

$$\begin{aligned}x &= r \cos(\theta) \\y &= r \sin(\theta) \\z &= z\end{aligned}$$

The default coordinate system is **PLOT_COORD_CARTESIAN**. For **PLOT_COORD_SPHERICAL** and **PLOT_COORD_CYLINDRICAL**, the default unit for *angle_unit* is **PLOT_ANGLE_RAD**.

Example 1

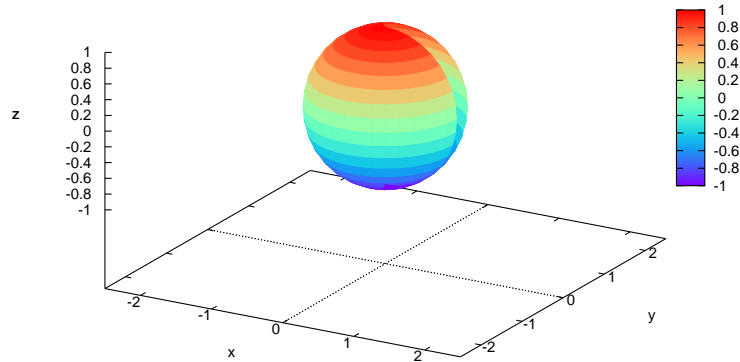
See **CPlot::data3D()**.

Example 2

```
/* File: coordSystem2.ch */
#include <chplot.h>
#include <math.h>

int main() {
    array double theta[37], phi[19], r[703];
    class CPlot plot;

    lindata(0, 2*M_PI, theta);
    lindata(-M_PI/2, M_PI/2, phi);
    r = (array double [703])1;
    plot.data3D(theta, phi, r);
    plot.coordSystem(PLOT_COORD_SPHERICAL);
    plot.axisRange(PLOT_AXIS_XY, -2.5, 2.5);
    plot.plotting();
}
```


Output**Example 3**

```

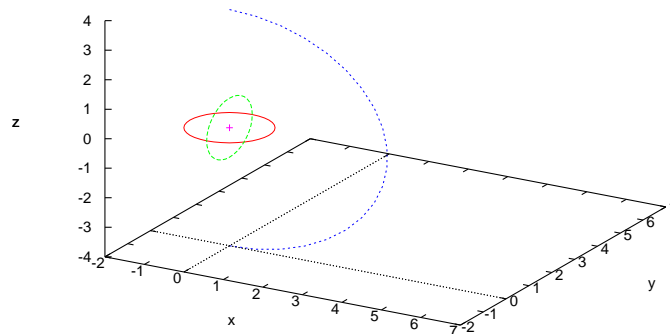
/* File: coordSystem3.ch */
#include <chplot.h>
#include <math.h>

int main() {
    array double theta1[37], phi1[37], r1[37];
    array double theta2[37], phi2[37], r2[37];
    array double theta3[20], phi3[20], r3[20];
    class CPlot plot;

    lndata(0, 2*M_PI, theta1);
    phi1 = (array double [37])0;
    r1 = (array double [37])1;
    theta2 = (array double [37])M_PI/2;
    lndata(phi2, 0, 2*M_PI, phi2);
    r2 = (array double [37])1;
    theta3 = (array double [20])0;
    lndata(-M_PI/2, M_PI/2, phi3);
    r3 = (array double [20])4;
    plot.data3D(theta1, phi1, r1);
    plot.data3D(theta2, phi2, r2);
    plot.data3D(theta3, phi3, r3);
    plot.point(0, 0, 0);
    plot.coordSystem(PLOT_COORD_SPHERICAL);
    plot.axisRange(PLOT_AXIS_XY, -2, 7);
    plot.ticsLevel(0);
    plot.removeHiddenLine(PLOT_OFF);
    plot.plotting();
}

```

Output

**Example 4**

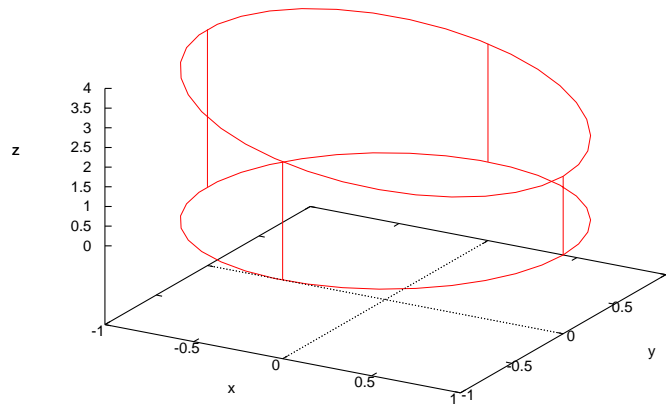
```

/* File: coordSystem4.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double r1[numpoints], theta1[numpoints], z1[numpoints];
    array double r2[numpoints], theta2[numpoints], z2[numpoints];
    class CPlot plot;

    lindata(0, 360, theta1);
    z1=(array double [numpoints])3+
        sin((theta1-(array double [numpoints])90)*M_PI/180);
    r1=(array double [numpoints])1;
    lindata(0, 360, theta2);
    z2=(array double [numpoints])0;
    r2=(array double [numpoints])1;
    plot.data3D(theta1, z1, r1);
    plot.data3D(theta2, z2, r2);
    plot.coordSystem(PLOT_COORD_CYLINDRICAL, PLOT_ANGLE_DEG);
    plot.line(0, 0, 1, 0, 2, 1);
    plot.line(90, 0, 1, 90, 3, 1);
    plot.line(180, 0, 1, 180, 4, 1);
    plot.line(270, 0, 1, 270, 3, 1);
    plot.plotType(PLOT_PLOTTYPE_LINES, 0);
    plot.lineType(0, 1, 1);
    plot.plotType(PLOT_PLOTTYPE_LINES, 1);
    plot.lineType(1, 1, 1);
    plot.plotType(PLOT_PLOTTYPE_LINES, 2);
    plot.lineType(2, 1, 1);
    plot.plotType(PLOT_PLOTTYPE_LINES, 3);
    plot.lineType(3, 1, 1);
    plot.plotType(PLOT_PLOTTYPE_LINES, 4);
    plot.lineType(4, 1, 1);
    plot.plotType(PLOT_PLOTTYPE_LINES, 5);
    plot.lineType(5, 1, 1);
    plot.plotting();
}

```

Output**Example 5**

```

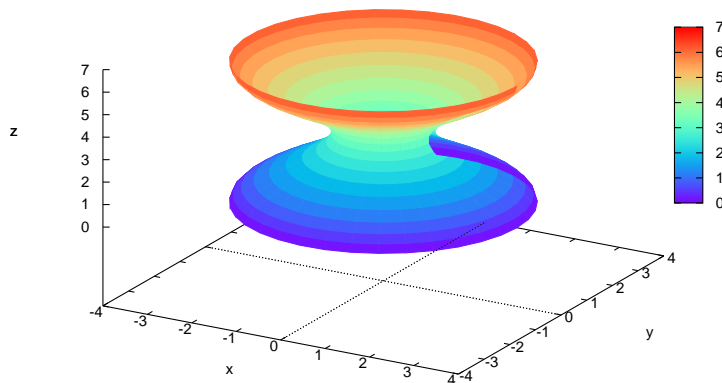
/* File: coordSystem5.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double theta[36], z[20], r[720];
    int i, j;
    class CPlot plot;

    lindata(0, 360, theta);
    lindata(0, 2*M_PI, z);
    for(i=0; i<36; i++) {
        for(j=0; j<20; j++) {
            r[i*20+j] = 2+cos(z[j]);
        }
    }
    plot.data3D(theta, z, r);
    plot.coordSystem(PLOT_COORD_CYLINDRICAL, PLOT_ANGLE_DEG);
    plot.axisRange(PLOT_AXIS_XY, -4, 4);
    plot.plotting();
}

```

Output

**See Also**

CPlot::data3D(), **CPlot::data3DCurve()**, **CPlot::data3DSurface()**.

CPlot::data

Synopsis

```
#include <chplot.h>
```

```
int data(void *x, int row, int col);
```

Purpose

Add 2D, 3D, or multi-dimensional data set to an instance of the **CPlot** class.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x A two-dimensional array of double type used for the plot.

row The number of rows for the array *x*.

col The number of columns for the array *x*.

Description

The data for a plot can be placed in either a file or in the memory. The opaque pointer *x* is the address for a two-dimensional array of double type. The size of the array is specified by its number of rows and columns. The data with multiple columns for a plot type such as candlesticks, finance bars, boxes, etc. can be added by this member function.

Example 1

In this example, a data set for 2D plot is added.

```
/* File: data.ch */
#include <math.h>
#include <chplot.h>
```

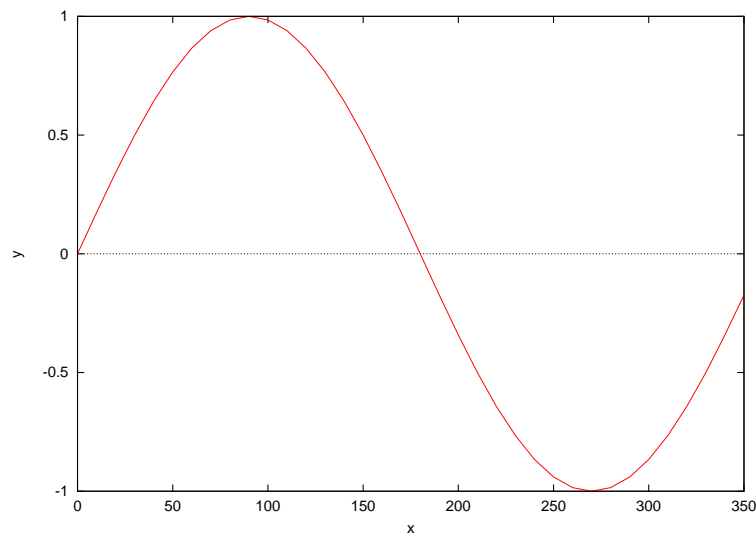
```

#define ROW 36
#define COL 2
int main() {
    int i;
    double a[ROW][COL];
    class CPlot plot;

    for(i=0; i< ROW; i++) {
        a[i][0] = i*10;
        a[i][1] = sin(a[i][0]*M_PI/180);
    }
    plot.data(a, ROW, COL);
    plot.plotting();
    return 0;
}

```

Output



Example 2 In this example, a data set for 3D plot is added.

```

/* File: data_1.ch */
#include <math.h>
#include <chplot.h>

#define ROW 36
#define COL 3
int main() {
    int i;
    double a[ROW][COL];
    class CPlot plot;

    for(i=0; i< ROW; i++) {
        a[i][0] = i*10;
        a[i][1] = sin(a[i][0]*M_PI/180);
        a[i][2] = cos(a[i][0]*M_PI/180);
    }
    plot.data(a, ROW, COL);
    plot.dimension(3);
}

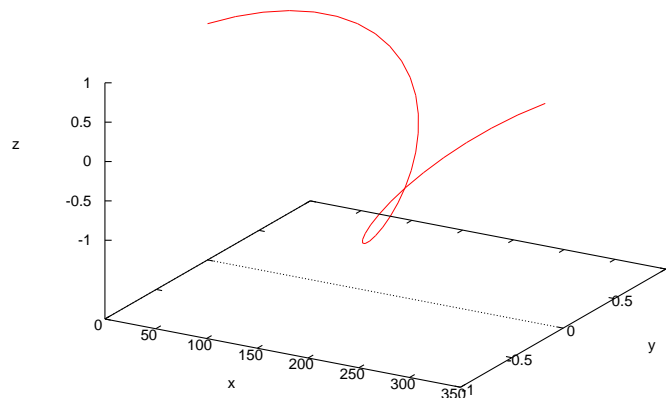
```

```

    plot.plotting();
    return 0;
}

```

Output



Example 3 In this example, a data set for candlesticks is added by `plot.data(a, ROW, COL)` and `plot.data(b, ROW, COL)`.

```

/* File: data_2.ch */
/* File: data_2.cpp to process data in candlesticks.dat */
/*      compare with plotType_cs.ch */
#include <chplot.h>

#define ROW 10
#define COL 5
int main() {
    class CPlot plot;
    double a[ROW][COL], b[ROW][COL];
    char filename[]="candlesticks.dat";
    FILE *stream;
    int i;

    stream = fopen(filename, "r");
    if(stream == NULL) {
        fprintf(stderr, "Error: cannot open file '%s' for reading\n", filename);
        return -1;
    }
    /* "using 1:3:2:6:5" */
    for(i = 0; i<ROW; i++) {
        fscanf(stream, "%lf%lf%lf%*lf%lf%lf",
            &a[i][0], &a[i][2], &a[i][1], &a[i][4], &a[i][3]);
        printf("%lf %lf %lf %lf %lf\n",
            a[i][0], a[i][1], a[i][2], a[i][3], &a[i][4]);
    }
    rewind(stream);
    /* using 1:4:4:4:4 */
    for(i = 0; i<ROW; i++) {
        fscanf(stream, "%lf%*lf%*lf%lf%*lf%*lf", &b[i][0], &b[i][1]);
    }
}

```

```

        b[i][2] = b[i][1];
        b[i][3] = b[i][1];
        b[i][4] = b[i][1];
        printf("%lf %lf\n", b[i][0], b[i][1]);
    }
    fclose(stream);

    plot.label(PLOT_AXIS_X, "");
    plot.label(PLOT_AXIS_Y, "");
    plot.border(PLOT_BORDER_ALL, PLOT_ON);
    plot.ticsMirror(PLOT_AXIS_XY, PLOT_ON);
    plot.title("box-and-whisker plot adding median value as bar");
    plot.axisRange(PLOT_AXIS_X, 0, 11);
    plot.axisRange(PLOT_AXIS_Y, 0, 10);
    //plot.dataFile("candlesticks.dat", "using 1:3:2:6:5");
    //plot.legend("'candlesticks.dat' using 1:3:2:6:5", 0);
    plot.data(a, ROW, COL);
    plot.legend("array a", 0);
    plot.plotType(PLOT_PLOTTYPE_CANDLESTICKS, 0, "linetype 1 linewidth 2 whiskerbars");
    plot.fillStyle(0, PLOT_FILLSTYLE_EMPTY);
    //plot.dataFile("candlesticks.dat", "using 1:4:4:4:4");
    //plot.legend("'candlesticks.dat' using 1:4:4:4:4", 1);
    plot.data(b, ROW, COL);
    plot.legend("array b", 1);
    plot.plotType(PLOT_PLOTTYPE_CANDLESTICKS, 1, "linetype -1 linewidth 2");
    plot.boxWidth(0.2);
    plot.plotting();
    return 0;
}

```

Output

The output is the same as that from program `plotType_cs.ch` on page 209 for `CPlot::plotType()`.

See Also

`CPlot::data2D()`, `CPlot::data2DCurve()`, `CPlot::data3D()`, `CPlot::data3DCurve()`, `CPlot::dataFile()`, `CPlot::plotType()`.

CPlot::data2D

Synopsis

```

#include <chplot.h>
int data2D(array double x[&], array double &y);

```

Purpose

Add one or more 2D data sets to an instance of `CPlot` class.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x a one-dimensional array of reference type used for the x axis of the plot.

y an array of reference type. If the size of array *x* is *n*, array *y* is a one-dimensional array of size *n* or a two-dimensional array of size *m* x *n* containing *m* curves, each of which is plotted against *x*.

Description

This function adds the specified x-y data to a previously declared instance of the **CPlot** class. The parameter *x* is a one-dimensional array of size *n*. The parameter *y* is a one-dimensional array of size *n* or a two-dimensional array of size *m* x *n*. In the case that *y* is *m* x *n*, each of the *m* rows of *y* is plotted against *x*. Each of the rows of *y* is a separate data set. The *x* and *y* arrays can be of any supported data type. Conversion of the data to **double** type is performed internally. Data points with a *y* value of NaN are internally removed before plotting occurs. "Holes" in a data set can be constructed by manually setting elements of *y* to this value. The plot of the data is generated using the **CPlot::plotting** member function.

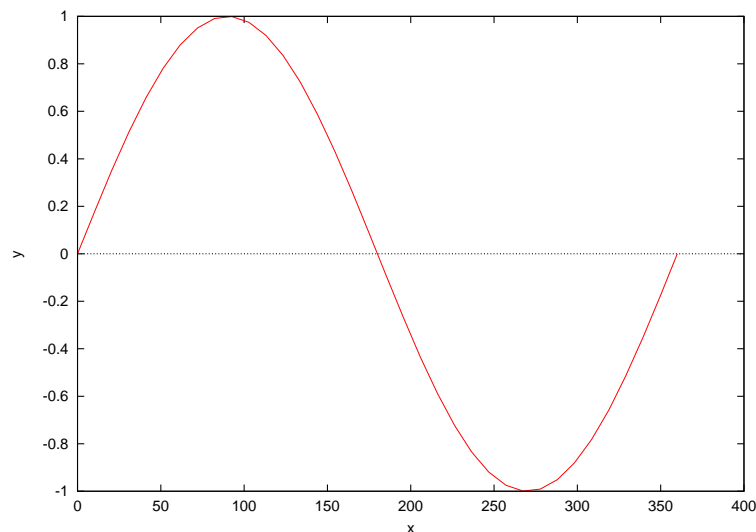
Example 1

Compare with the output for the examples in **CPlot::arrow()**, **CPlot::autoScale()**, **CPlot::borderOffsets()**, **CPlot::displayTime()**, **CPlot::label()**, **CPlot::ticsLabel()**, **CPlot::margins()**, **CPlot::ticsDirection()**, **CPlot::ticsFormat()**, **CPlot::ticsLocation()**, and **CPlot::title()**.

```
/* File: data2D.ch */
#include <math.h>
#include <chplot.h>
#include <numeric.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    lindata(0, 360, x);
    y = sin(x*M_PI/180);
    plot.data2D(x, y);
    plot.plotting();
}
```

Output**Example 2**

```
/* File: data2D_2.ch */
#include <math.h>
#include <chplot.h>
#include <numeric.h>
```



```

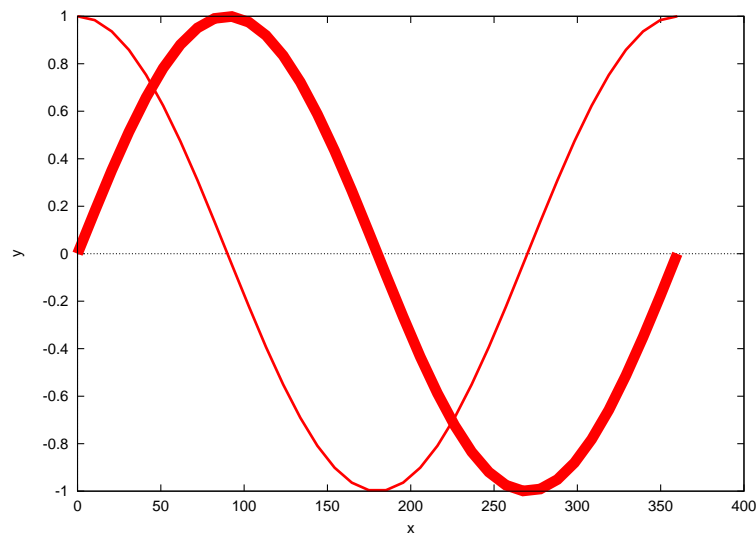
#define CURVE1 0
#define CURVE2 1

int main() {
    int numpoints = 36;
    array double x[numpoints], y[2][numpoints];
    int i;
    class CPlot plot;

    lindata(0, 360, x);
    for(i=0; i<numpoints; i++) {
        y[0][i] = sin(x[i]*M_PI/180);
        y[1][i] = cos(x[i]*M_PI/180);
    }
    plot.data2D(x, y);
    plot.plotType(PLOT_PLOTTYPE_LINES, CURVE1);
    plot.lineType(CURVE1, 1, 20); /* set first data set to use the default line
                                   type and a width of twenty times the
                                   default width */
    plot.plotType(PLOT_PLOTTYPE_LINES, CURVE2);
    plot.lineType(CURVE2, 1, 5); /* set second data set to use the default line
                                   type and a width of five times the default */
    plot.plotting();
}

```

Output



Example 3

```

/* File: data2D_3.ch */
/* Two curves have a phase shift */
#include <math.h>
#include <chplot.h>
#include <numeric.h>

int main() {
    int numpoints = 36;
    array double x1[numpoints], y1[numpoints];
    array double x2[numpoints];

```

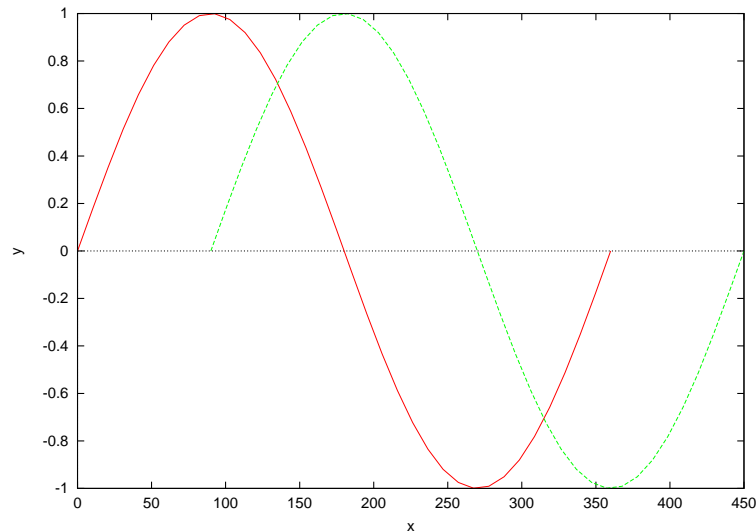
```

class CPlot plot;

lindata(0, 360, x1);
y1 = sin(x1*M_PI/180);
lindata(90, 450, x2);
plot.data2D(x1, y1);
plot.data2D(x2, y1);
plot.plotting();
}

```

Output



Example 4

```

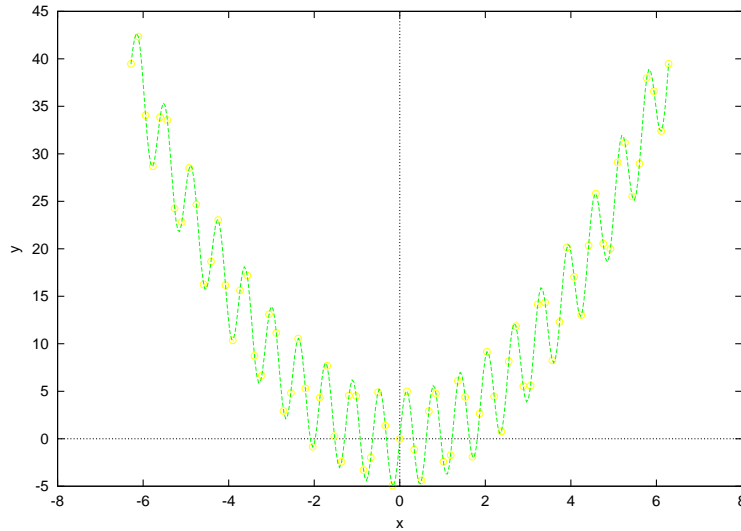
/* File: data2D_4.ch */
#include <chplot.h>
#include <math.h>
#include <numeric.h>

int main() {
    array double x1[75], y1[75];
    array double x2[300], y2[300];
    class CPlot plot;

    lindata(-2*M_PI, 2*M_PI, x1);
    lindata(-2*M_PI, 2*M_PI, x2);
    y1 = x1.*x1+5*sin(10*x1);
    y2 = x2.*x2+5*sin(10*x2);
    plot.data2D(x1, y1);
    plot.data2D(x2, y2);
    plot.plotType(PLOT_PLOTTYPE_POINTS, 0);
    plot.pointType(0, 6, 1);
    plot.plotType(PLOT_PLOTTYPE_LINES, 1);
    plot.lineType(1, 2, 1);
    plot.plotting();
}

```

Output

**See Also**

CPlot::data2DCurve(), **CPlot::data3D()**, **CPlot::data3DCurve()**, **CPlot::data3DSurface()**,
CPlot::dataFile(), **CPlot::plotting()**, **plotxy()**.

CPlot::data2DCurve

Synopsis

```
#include <chplot.h>
```

```
int data2DCurve(double x[], double y[], int n);
```

Purpose

Add a set of data for 2D curve to an instance of **CPlot** class.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x a one-dimensional array of double type used for the x axis of the plot.

y a one-dimensional array of double type for the y axis.

n number of elements of arrays *x* and *y*.

Description

This function adds the specified x-y data to a previously declared instance of the **CPlot** class. The parameter *x* is a one-dimensional array of size *n*. The parameter *y* is a one-dimensional array of size *n*. Data points with a y value of NaN are internally removed before plotting occurs. "Holes" in a data set can be constructed by manually setting elements of *y* to this value. The plot of the data is generated using the **CPlot::plotting** member function.

Example 1

Compare with the output for examples in **CPlot::arrow()**, **CPlot::autoScale()**, **CPlot::borderOffsets()**, **CPlot::data2D()**, **CPlot::displayTime()**, **CPlot::label()**, **CPlot::ticsLabel()**, **CPlot::margins()**, **CPlot::ticsDirection()**, **CPlot::ticsFormat()**, **CPlot::ticsLocation()**, and **CPlot::title()**.

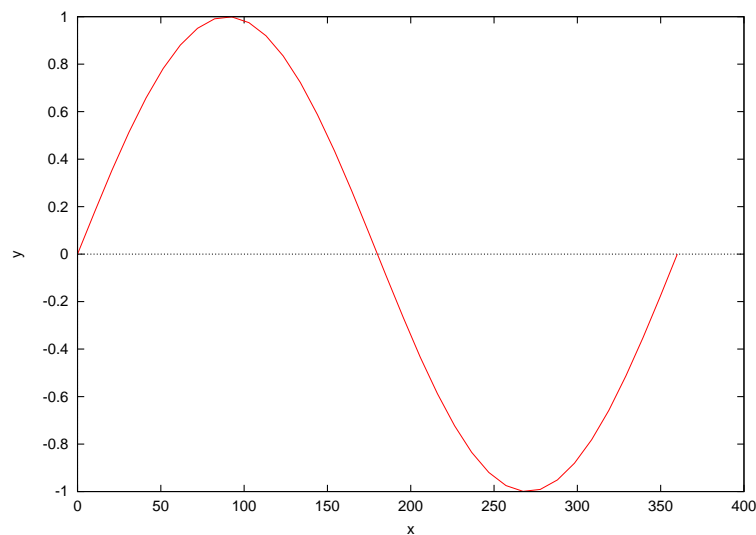
```

/* File: data2DCurve.ch */
#include <math.h>
#include <chplot.h>
#ifndef M_PI
#define M_PI          3.14159265358979323846
#endif

#define NUM 36
int main() {
    int i;
    double x[NUM], y[NUM];
    class CPlot plot;

    for(i=0; i< NUM; i++) {
        x[i] = i*10;
        y[i] = sin(x[i]*M_PI/180);
    }
    plot.data2DCurve(x, y, NUM);
    plot.plotting();
    return 0;
}

```

Output**See Also**

CPlot::data2D(), **CPlot::data3D()**, **CPlot::data3DCurve()**, **CPlot::data3DSurface()**, **CPlot::dataFile()**, **CPlot::plotting()**, **plotxy()**.

CPlot::data3D**Synopsis**

```
#include <chplot.h>
```

```
int data3D(array double x[&], array double y[&], array double &z);
```

Purpose

Add one or more 3D data sets to an instance of **CPlot** class.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x a one-dimensional array of size n_x used for the x axis of the plot.

y a one-dimensional array of size n_y used for the y axis of the plot.

z a one-dimensional array of size n_z , or a two dimensional array of size $m \times n_z$, containing m 3D data sets.

Description

This function adds one or more 3D data sets to an existing plot variable. For Cartesian data, *x* is a one-dimensional array of size n_x and *y* is a one-dimensional array of size n_y . *z* can be of two different sizes depending on what type of data is to be plotted. If the data are for a 3D curve, *z* is a one-dimensional array of size n_z or a two-dimensional array of size $m \times n_z$, with $n_x = n_y = n_z$. If the data are for a 3D surface or grid, *z* is $m \times n_z$, with $n_z = n_x \cdot n_y$. For cylindrical or spherical data *x* is a one dimensional array of size n_x (representing θ), *y* is a one dimensional array of size n_y (representing z or ϕ), and *z* is of the size $m \times n_z$ (representing r) where $n_x = n_y = n_z$. Each of the m rows of *z* are plotted against *x* and *y*, and correspond to a separate data set. In all cases these data arrays can be of any supported data type. Conversion of the data to **double** type is performed internally. For grid data, hidden line removal is enabled automatically (see **CPlot::removeHiddenLine()**). If it is desired to plot both grid data and non-grid data on the same plot, hidden line removal should be disabled manually after all data are added. Data points with a *z* value of NaN are internally removed before plotting occurs. "Holes" in a data set can be constructed by manually setting elements of *z* to this value.

It is important to note that for a 3D grid, the ordering of the *z* data is important. For calculation of the *z* values, the *x* value is held constant while *y* is cycled through its range of values. The *x* value is then incremented and *y* is cycled again. This is repeated until all the data are calculated. So, for a 10x20 grid the data shall be ordered as follows:

```

x1  y1  z1
x1  y2  z2
x1  y3  z3
.
.
.
x1  y18 z18
x1  y19 z19
x1  y20 z20
x2  y1  z21
x2  y2  z22
x2  y3  z23
.
.
.
x10 y18 z198
x10 y19 z199
x10 y20 z200

```

Example 1

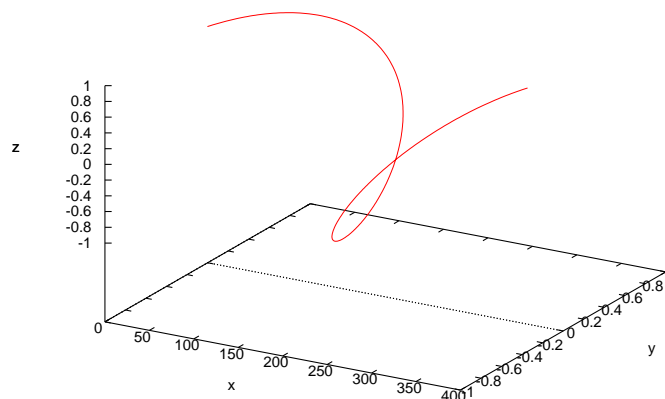
```

/* File: data3D.ch */
#include <math.h>
#include <chplot.h>
#include <numeric.h>

int main() {
    array double x[360], y[360], z[360];
    class CPlot plot;

    lindata(0, 360, x);
    y = sin(x*M_PI/180);
    z = cos(x*M_PI/180);
    plot.data3D(x, y, z);
    plot.plotting();
}

```

Output**Example 2**

```

/* File: data3D_2.ch */
#include <math.h>
#include <chplot.h>
#include <numeric.h>

int main() {
    array double x[360], y[360], z[2][360];
    int i;
    class CPlot plot;

    lindata(0, 360, x);
    y = sin(x*M_PI/180);
    for(i=0; i<360; i++) {
        z[0][i] = cos(x[i]*M_PI/180);
        z[1][i] = y[i];
    }
    plot.data3D(x, y, z);
    plot.plotType(PLOT_PLOTTYPE_LINES, 1);
}

```

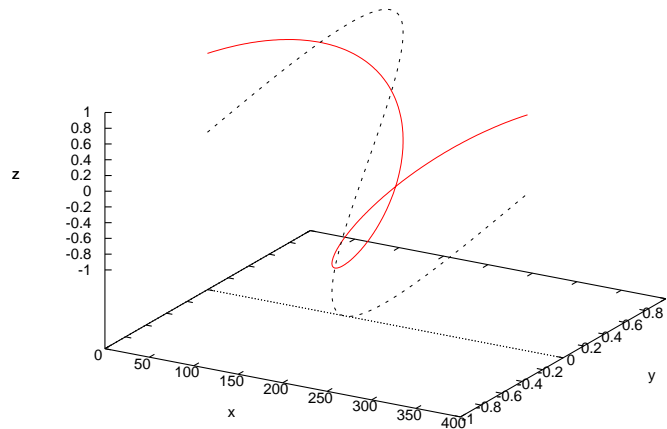
```

    plot.lineType(1, 0, 3); /* set the second data set to
                               use the default line type
                               and a line width three
                               times the default */

    plot.plotting();
}

```

Output



Example 3

Compare with output for examples in `CPlot::arrow()`, `CPlot::contourLabel()`, `CPlot::grid()`, `CPlot::removeHiddenLine()`, `CPlot::size3D()`, `CPlot::changeViewAngle()`, and `CPlot::ticsLevel()`.

```

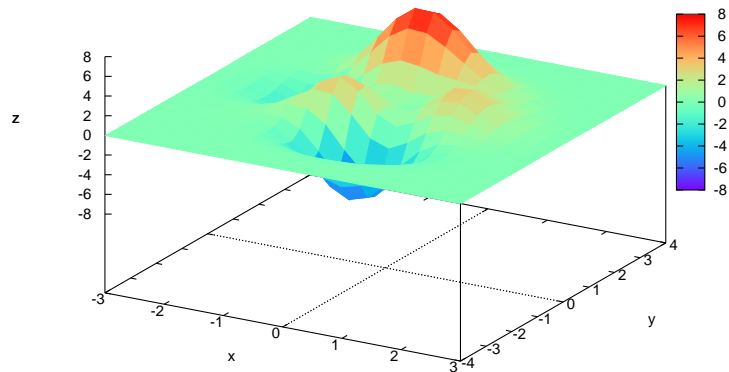
/* File: data3D_3.ch */
#include <math.h>
#include <chplot.h>
#include <numeric.h>

int main() {
    double x[20], y[30], z[600];
    int i,j;
    class CPlot plot;

    lindata(-3, 3, x);
    lindata(-4, 4, y);
    for(i=0; i<20; i++) {
        for(j=0; j<30; j++) {
            z[30*i+j] = 3*(1-x[i])*(1-x[i])*exp(-(x[i]*x[i])-(y[j]+1)*(y[j]+1))
                - 10*(x[i]/5 - x[i]*x[i]*x[i]-pow(y[j],5))*exp(-x[i]*x[i]-y[j]*y[j])
                - 1/3*exp(-(x[i]+1)*(x[i]+1)-y[j]*y[j]));
        }
    }
    plot.data3D(x, y, z);
    plot.plotting();
}

```

Output

**Example 4**

Compare with output for example `CPlot::axisRange()` and `CPlot::colorBox()`.

```

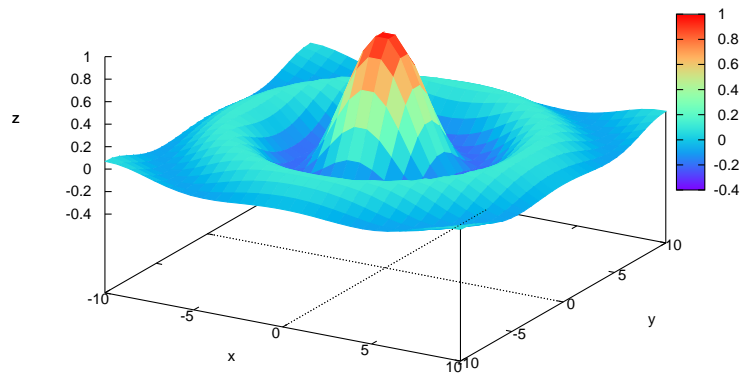
/* File: data3D_4.ch */
#include <chplot.h>
#include <math.h>
#include <numeric.h>

int main() {
    double x[30], y[30], z[900];
    double r;
    int i, j;
    class CPlot plot;

    lindata(-10, 10, x);
    lindata(-10, 10, y);
    for(i=0; i<30; i++) {
        for(j=0; j<30; j++) {
            r = sqrt(x[i]*x[i]+y[j]*y[j]);
            z[30*i+j] = sin(r)/r;
        }
    }
    plot.data3D(x, y, z);
    plot.plotting();
}

```

Output



Example 5

```

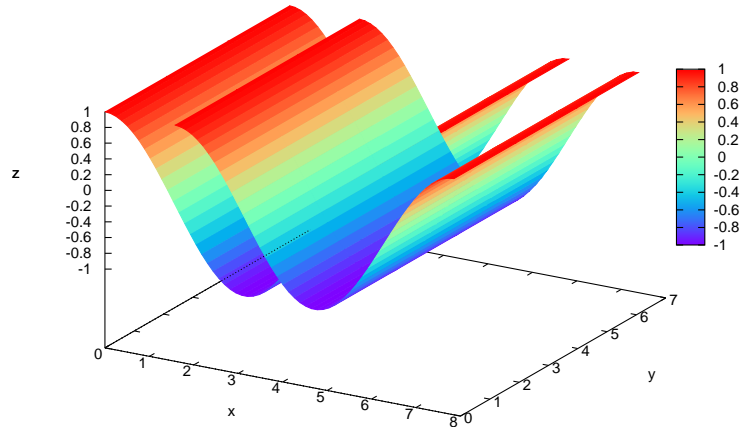
/* File: data3D_5.ch */
#include <chplot.h>
#include <math.h>
#include <numeric.h>

int main() {
    array double x1[50], x2[50], y[4], z[200];
    int i,j,angle;
    class CPlot plot;

    lindata(0, 2*M_PI, x1);
    lindata(M_PI/2, 2*M_PI+M_PI/2, x2);
    lindata(0, 2*M_PI, y);
    for (i=0;i<50;i++) {
        for (j=0;j<4;j++) {
            z[j+4*i]=cos(x1[i]);          // Z-axis data.
        }
    }
    plot.data3D(x1, y, z);
    plot.data3D(x2, y, z);
    plot.plotting();
}

```

Output

**Example 6**

Compare with the output for the example `CPlot::changeViewAngle()`.

```

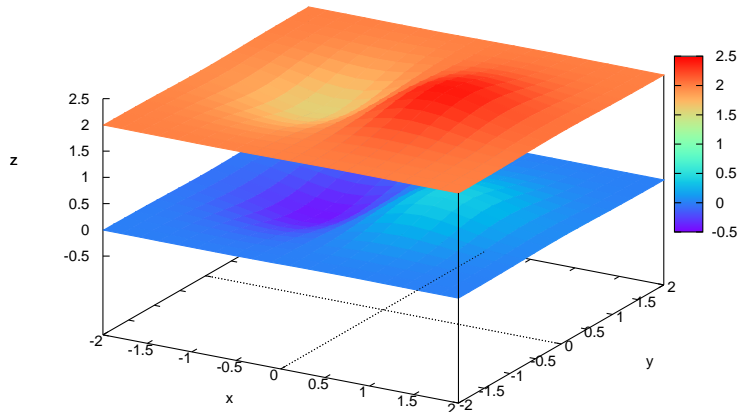
/* File: data3D_6.ch */
#include <chplot.h>
#include <math.h>
#include <numeric.h>

#define NUMX 20
#define NUMY 20
#define NUMCURVE 2
int main() {
    array double x[NUMX], y[NUMY], z[NUMCURVE][NUMX*NUMY];
    int i, j;
    class CPlot plot;

    lindata(-2, 2, x);
    lindata(-2, 2, y);
    for (i=0; i<NUMX; i++) {
        for(j=0; j<NUMY; j++) {
            z[0][i*NUMX+j] = x[i]*exp(-x[i]*x[i]-y[j]*y[j]);
            z[1][i*NUMX+j] = z[0][i*NUMX+j] +2;
        }
    }
    plot.data3D(x, y, z);
    plot.plotting();
}

```

Output

**See Also**

CPlot::data2D(), **CPlot::data2DCurve()**, **CPlot::data3DCurve()**, **CPlot::data3DSurface()**,
CPlot::dataFile(), **CPlot::plotting()**, **plotxyz()**.

CPlot::data3DCurve

Synopsis

```
#include <chplot.h>
```

```
int data3DCurve(double x[], double y[], double z[], int n);
```

Purpose

Add a set of data for 3D curve to an instance of **CPlot** class.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x a one-dimensional array of size *n* used for the x axis of the plot.

y a one-dimensional array of size *n* used for the y axis of the plot.

z a one-dimensional array of size *n* used for the z axis of the plot.

n the number of elements for arrays *x*, *y*, and *z*.

Description

Add a set of data for 3D curve to an instance of **CPlot** class. Arrays *x*, *y*, and *z* have the same number of elements of size *n*. In a Cartesian coordinate system, these arrays represent data in X-Y-Z coordinates. In a cylindrical coordinate system, *x* represents θ , *y* for *z*, and *z* for *r*. In a spherical coordinate system, *x* represents θ , *y* for ϕ , and *z* for *r*. Data points with a *z* value of NaN are internally removed before plotting occurs. "Holes" in a data set can be constructed by manually setting elements of *z* to this value.

Example 1

Compare with output for examples in **CPlot::data3D()**.

```

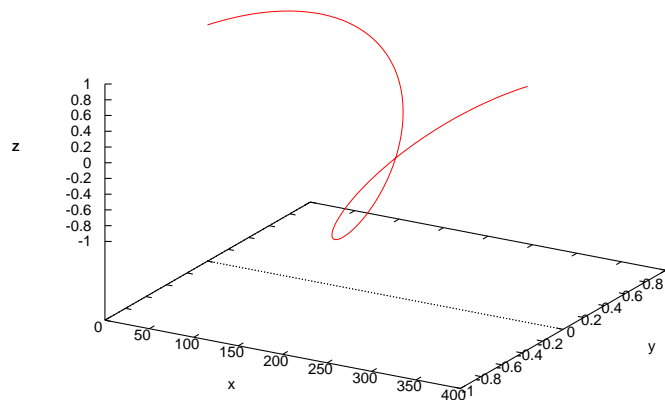
/* File: data3DCurve.ch */
#include <math.h>
#include <chplot.h>
#ifndef M_PI
#define M_PI          3.14159265358979323846
#endif

#define NUM 36
int main() {
    int i;
    double x[NUM], y[NUM], z[NUM];
    class CPlot plot;

    for(i=0; i< NUM; i++) {
        x[i] = i*10;
        y[i] = sin(x[i]*M_PI/180);
        z[i] = cos(x[i]*M_PI/180);
    }
    plot.data3DCurve(x, y, z, NUM);
    plot.plotting();
    return 0;
}

```

Output



Example 2

Compare with output for examples in `CPlot::data3D()`.

```

/* File: data3DCurve_2.ch */
#include <math.h>
#include <chplot.h>
#ifndef M_PI
#define M_PI          3.14159265358979323846
#endif

#define NUM 360
int main() {
    double x[NUM], y[NUM], z1[NUM], z2[NUM];
    int i;
}

```

```

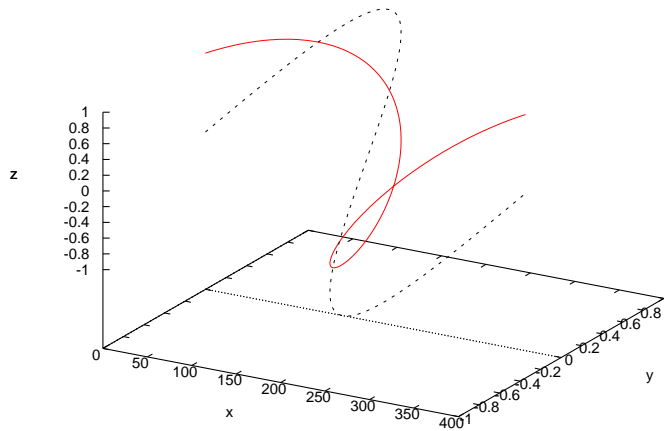
class CPlot plot;

for(i=0; i<360; i++) {
    x[i] = i;
    y[i] = sin(x[i]*M_PI/180);
    z1[i] = cos(x[i]*M_PI/180);
    z2[i] = y[i];
}
plot.data3DCurve(x, y, z1, NUM);
plot.data3DCurve(x, y, z2, NUM);
plot.plotType(PLOT_PLOTTYPE_LINES, 1);
plot.lineType(1, 0, 3); /* set the second data set to
                           use the default line type
                           and a line width three
                           times the default */

plot.plotting();
return 0;
}

```

Output



See Also

CPlot::data2D(), CPlot::data2DCurve(), CPlot::data3D(), CPlot::data3DSurface(), CPlot::dataFile(), CPlot::plotting(), plotxyz().

CPlot::data3DSurface

Synopsis

```
#include <chplot.h>
```

```
int data3DSurface(double x[], double y[], double z[], int n, int m);
```

Purpose

Add a set of data for 3D surface to an instance of **CPlot** class.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x a one-dimensional array of size *n* used for the x axis of the plot.

y a one-dimensional array of size *m* used for the y axis of the plot.

z a one-dimensional array of size $n \times m$

n the number of elements for array *x*.

m the number of elements for array *y*.

Description

Add a set of data for 3D surface plot to an instance of **CPlot** class. If one-dimensional array *x* has the number of elements of size *n*, and *y* has size *m*, *z* shall be a one-dimensional array of size $n_z = n \cdot m$. In a Cartesian coordinate system, arrays *x*, *y*, and *z* represent values in X-Y-Z coordinates, respectively. In a cylindrical coordinate system, arrays *x*, *y*, and *z* represent θ , *z*, and *r* coordinates, respectively. In a spherical coordinate system, arrays *x*, *y*, and *z* represent θ , ϕ , and *r* coordinates, respectively. Hidden line removal is enabled automatically (see **CPlot::removeHiddenLine()**). If it is desired to plot both grid data and non-grid data on the same plot, hidden line removal should be disabled manually after all data are added. Data points with a *z* value of NaN are internally removed before plotting occurs. "Holes" in a data set can be constructed by manually setting elements of *z* to this value.

It is important to note that for a 3D grid, the ordering of the *z* data is important. For calculation of the *z* values, the *x* value is held constant while *y* is cycled through its range of values. The *x* value is then incremented and *y* is cycled again. This is repeated until all the data are calculated. So, for a 10x20 grid the data shall be ordered as follows:

```

x1  y1  z1
x1  y2  z2
x1  y3  z3
.
.
.
x1  y18 z18
x1  y19 z19
x1  y20 z20
x2  y1  z21
x2  y2  z22
x2  y3  z23
.
.
.
x10 y18 z198
x10 y19 z199
x10 y20 z200

```

Example 1

Compare with output for examples in **CPlot::data3D()**, **CPlot::arrow()**, **CPlot::contourLabel()**, **CPlot::grid()**, **CPlot::removeHiddenLine()**, **CPlot::size3D()**, **CPlot::changeViewAngle()**, and **CPlot::ticsLevel()**.

```

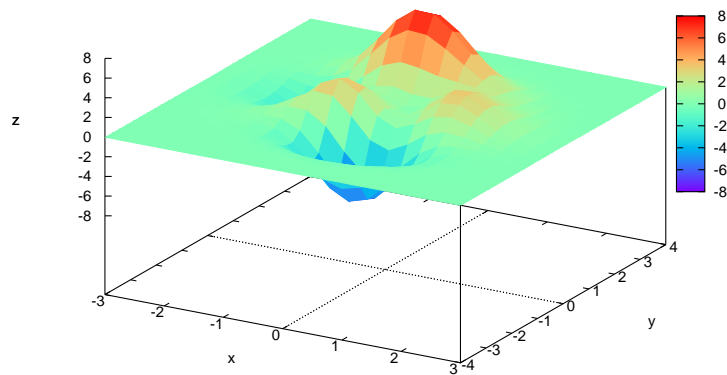
/* File: data3DSurface_3.ch */
#include <math.h>
#include <chplot.h>

#define N 20
#define M 30
int main() {
    double x[N], y[M], z[N*M];
    int i,j;
    class CPlot plot;

    for(i=0; i<N; i++) {
        x[i] = -3 + i*6/19.0; // lndata(-3, 3, x)
    }
    for(i=0; i<M; i++) {
        y[i] = -4 + i*8/29.0; // lndata(-4, 4, y)
    }
    for(i=0; i<N; i++) {
        for(j=0; j<M; j++) {
            z[M*i+j] = 3*(1-x[i])*(1-x[i])*exp(-(x[i]*x[i])-(y[j]+1)*(y[j]+1))
                - 10*(x[i]/5 - x[i]*x[i]*x[i]-pow(y[j],5))*exp(-x[i]*x[i]-y[j]*y[j])
                - 1/3*exp(-(x[i]+1)*(x[i]+1)-y[j]*y[j]));
        }
    }
    plot.data3DSurface(x, y, z, N, M);
    plot.plotting();
    return 0;
}

```

Output



See Also

CPlot::data2D(), **CPlot::data2DCurve()**, **CPlot::data3D()**, **CPlot::data3DCurve()**,
CPlot::dataFile(), **CPlot::plotting()**, **plotxyz()**.

CPlot::dataFile

Synopsis

```
#include <chplot.h>
int dataFile(string_t file, ... /* [char option] */);
```

Syntax

```
dataFile(file)
dataFile(file, option)
```

Purpose

Add a data file to an existing instance of the **CPlot** class.

Return Value

This function returns 0 on success and -1 on failure.

Parameter

file name of the file to be plotted.

option The option for the data file.

Description

Add a data file to an existing plot variable. Each data file corresponds to a single data set. The data file should be formatted with each data point on a separate line. 2D data are specified by two values per point. An empty line in a 2D data file causes a break of the curve in the plot. Multiple curves can be plotted in this manner, however the plot style will be the same for all curves. 3D data are specified by three values per data point.

For a 3D grid or surface data, each row is separated in the data file by a blank line. By default, hidden lines are not removed for 3D plotting using a data file. Use function **CPlot::removeHiddenLine()** to remove hidden lines.

The symbol # will comment out a subsequent text terminated at the end of a line in a data file. For example, a 3 x 3 grid would be represented as follows:

```
# This is a comment line
x1  y1  z1
x1  y2  z2
x1  y3  z3

x2  y1  z4
x2  y2  z5
x2  y3  z6

x3  y1  z7
x3  y2  z8
x3  y3  z9
```

Two empty lines in the data file will cause a break in the plot. Multiple curves or surfaces can be plotted in this manner, however, the plot style will be the same for all curves or surfaces. Member function **CPlot::dimension(3)** must be called before 3D data file can be added.

The option for the data file is as follows.

```
using {<entry> {:<entry> {:<entry> ...}}} {'format' }
```


If a format is specified, each datafile record is read using the C library's `scanf()` function, with the specified format string. Otherwise the record is read and broken into columns at spaces or tabs.

The resulting array of data is then sorted into columns according to the entries. Each <entry> may be a simple column number, which selects the datum, an expression enclosed in parentheses, or empty. The expression can use `$1` to access the first item read, `$2` for the second item, and so on. A column number of 0 generates a number increasing (from zero) with each point, and is reset upon encountering two blank records. A column number of -1 gives the dataline number, which starts at 0, increments at single blank records, and is reset at double blank records. A column number of -2 gives the index number, which is incremented only when two blank records are found. An empty <entry> will default to its order in the list of entries. For example, using `:4\verb` is interpreted as using `1:2:4\verb`.

If the `using\verb` list has but a single entry, that <entry> will be used for y and the data point number is used for x; for example, using `1\verb` is identical to using `0:1\verb`. If the `using\verb` list has two entries, these will be used for x and y. Additional entries are usually plot style of errors in x and/or y. See `CPlot::plotType()` for details about plotting styles that make use of error information.

The C Function `scanf()` accepts several numerical specifications `CPlot` requires all inputs to be double-precision floating-point variables, so `"%lf"` is essentially the only permissible specifier. A format string given by the user must contain at least one such input specifier, and no more than seven of them. `scanf()` expects to see white space—a blank, tab (`"\t"`), newline (`"\n"`), or formfeed (`"\f"`)—between numbers; anything else in the input stream must be explicitly skipped. Note that the use of `"\t"`, `"\n"`, or `"\f"` requires use of double-quotes rather than single-quotes.

Examples:

This creates a plot of the sum of the 2nd and 3rd data against the first: The format string specifies comma- rather than space-separated columns.

```
using 1:($2+$3) '%lf,%lf,%lf'
```

In this example the data are read from a using a more complicated format:

```
using "%*lf%lf%*20[^\n]%lf"
```

The meaning of this format is:

<code>%*lf</code>	ignore a number
<code>%lf</code>	read a double-precision number (x by default)
<code>%*20[^\n]</code>	ignore 20 non-newline characters
<code>%lf</code>	read a double-precision number (y by default)

One trick is to use the C ternary `'?:'\verb` operator to filter data:

```
using 1:($3>10 ? $2 : 1/0)
```

which plots the datum in column two against that in column one provided the datum in column three exceeds ten. `1/0` is undefined; `CPlot` quietly ignores undefined points, so unsuitable points are suppressed.

If timeseries data are being used, the time can span multiple columns. The starting column should be specified. Note that the spaces within the time must be included when calculating starting columns for other

data. E.g., if the first element on a line is a time with an embedded space, the y value should be specified as column three.

It should be noted that for three cases a) without option, b) with option of `using 1:2`, c) with option `using ($1):($2)` can be subtly different: 1) if the datafile has some lines with one column and some with two, the first will invent x values when they are missing, the second will quietly ignore the lines with one column, and the third will store an undefined value for lines with one point (so that in a plot with lines, no line joins points across the bad point); 2) if a line contains text at the first column, the first will abort the plot on an error, but the second and third should quietly skip the garbage.

In fact, it is often possible to plot a file with lots of lines of garbage at the top simply by specifying

```
using 1:2
```

However, if you want to leave text in your data files, it is safer to put the comment character '#' in the first column of the text lines.

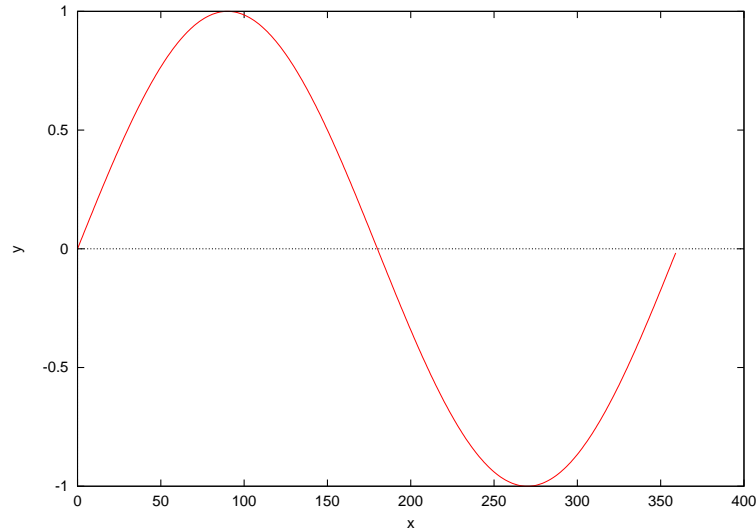
Example

```
/* File: dataFile.ch */
#include <stdio.h>
#include <chplot.h>
#include <math.h>

int main() {
    string_t file;
    int i;
    class CPlot plot;
    FILE *out;

    file = tmpnam(NULL);           //Create temporary file.
    out=fopen (file,"w");         //Write data to file.
    for(i=0;i<=359;i++) fprintf(out,"%i %f \n",i,sin(i*M_PI/180));
    fclose(out);
    plot.dataFile(file);
    plot.plotting();
    remove(file);
}
```

Output

**Examples**

See an example on page 207 for **CPlot::plotType()**. For comparison with data from **CPlot::dataFile()** and **CPlot::data()**, see programs on pages 73 and 209 for plot with candlesticks.

See Also

CPlot::data2D(), **CPlot::data2DCurve()**, **CPlot::data3D**, **CPlot::data3DCurve**, **CPlot::data3DSurface**, **CPlot::outputType()**, **CPlot::plotting()**, **plotxyf()**, **plotxyzf()**.

CPlot::dataSetNum

Synopsis

```
#include <chplot.h>
int dataSetNum();
```

Purpose

Obtain the current data set number in an instance of **CPlot** class.

Return Value

The current data set number in an instance of **CPlot** class. The first data set number is 0. If there is no data in the instance of the **CPlot** class, the return value is -1.

Parameters

None.

Description

This function returns the current data set number in an instance of **CPlot** class.

Example

```
/* File: dataSetNum.ch */
#include <math.h>
#include <chplot.h>
```

```

int main() {
    int num, numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    lindata(0, 360, x, 36);
    y = sin(x*M_PI/180);
    num = plot.dataSetNum();
    printf("The number of data set is %d\n", num);
    plot.data2D(x, y);
    num = plot.dataSetNum();
    printf("The number of data set is %d\n", num);
    plot.data2D(x, 2*y);
    num = plot.dataSetNum();
    printf("The number of data set is %d\n", num);
    plot.plotting();
}

```

Output in console

```

The number of data set is -1
The number of data set is 0
The number of data set is 1

```

CPlot::deleteData**Synopsis**

```

#include <chplot.h>
void deleteData();

```

Purpose

Delete all plot data of an instance of the **CPlot** class.

Return Value

None.

Parameters

None.

Description

This function frees all memory associated with previously allocated plot data, plot type, legends, plot axes, points, lines, polygons, rectangles, and circles. Unlike **CPlot::deletePlots()**, this function does not reset plotting options to their default values. This function allows for the reuse of a single instance of the **CPlot** class to create multiple plots.

See Also

CPlot::arrow(), **CPlot::circle()**, **CPlot::data2D()**, **CPlot::data2DCurve()**, **CPlot::data3D()**, **CPlot::data3DCurve()**, **CPlot::data3DSurface()**, **CPlot::dataFile()**, **CPlot::deletePlots()**, **CPlot::ticsLabel()**, **CPlot::line()**, **CPlot::point()**, **CPlot::polygon**, **CPlot::rectangle()**, **CPlot::text()**.

CPlot::deletePlots

Synopsis

```
#include <chplot.h>
void deletePlots();
```

Purpose

Delete all plot data and reinitialize an instance of the class to default values.

Return Value

None.

Parameters

None.

Description

This function frees all memory associated with previously allocated plot data, plot type, legends, plot axes, text strings, arrows, points, lines, polygons, rectangles, circles, and labeled tic-marks. This function also resets all plotting options to their default values. This function allows for the reuse of a single instance of the **CPlot** class to create multiple plots. This function is used internally by **fplotxy()**, **fplotxyz()**, **plotxy()**, **plotxyz()**, **plotxyf()**, **plotxyzf()**.

See Also

CPlot::arrow(), **CPlot::circle()**, **CPlot::data2D()**, **CPlot::data2DCurve()**, **CPlot::data3D()**, **CPlot::data3DCurve()**, **CPlot::data3DSurface()**, **CPlot::dataFile()**, **CPlot::deleteData()**, **CPlot::ticsLabel()**, **CPlot::line()**, **CPlot::point()**, **CPlot::polygon**, **CPlot::rectangle()**, **CPlot::text()**, **fplotxy()**, **fplotxyz()**, **plotxy()**, **plotxyz()**, **plotxyf()**, **plotxyzf()**.

CPlot::dimension

Synopsis

```
#include <chplot.h>
void dimension(int dim);
```

Purpose

Set plot dimension to 2D or 3D.

Return Value

None.

Parameter

dim 2 for 2D and 3 for 3D. Default is 2.

Description

Set the dimension of the plot. The plot dimension should be set before data are added to the plot if member functions **CPlot::dataThreeD()**, **CPlot::dataThreeDCurve()**, or **CPlot::dataThreeDSurface()** are not called before. This member function must be used when 3D plotting data are added by **CPlot::dataFile()** and

CPlot::polygon().

Example

See **CPlot::polygon()**.

See Also

CPlot::data2D(), **CPlot::data2DCurve()**, **CPlot::data3D()**, **CPlot::data3DCurve()**,
CPlot::data3DSurface(), **CPlot::dataFile()**, **CPlot::polygon()**.

CPlot::displayTime

Synopsis

```
#include <chplot.h>
```

```
void displayTime(double x_offset, double y_offset);
```

Purpose

Display the current time and date.

Return Value

None.

Parameters

x_offset Offset of the time-stamp in the x direction from its default location.

y_offset Offset of the time-stamp in the y direction from its default location.

Description

This function places the current time and date near the left margin. The exact location is device dependent. The offset values, *x_offset* and *y_offset*, are in screen coordinates and are measured in characters. For example, if both *x_offset* and *y_offset* are 2, the time will be moved approximately two characters to the right and two characters above the default location.

Example

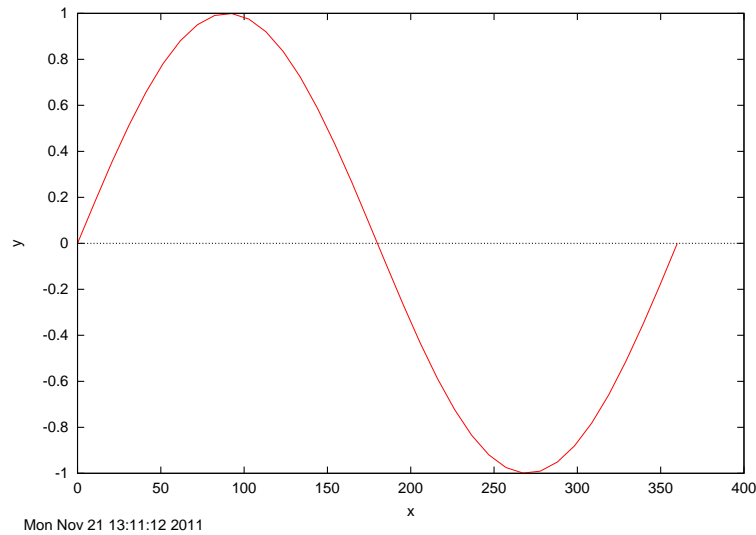
Compare with the output for examples in **CPlot::data2D()** and **CPlot::data2DCurve()**.

```
/* File: displayTime.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    lindata(0, 360, x);
    y = sin(x*M_PI/180);
    plot.displayTime(10,0);
    plot.data2D(x, y);
    plot.plotting();
}
```

Output



CPlot::dot

Synopsis

```
#include <chplot.h>
```

```
void dot(int x[:], ... /* int n */);
```

Syntax

```
dot(x);
```

```
dot(x, n);
```

Purpose

Draw a dot plot.

Return Value

None.

Parameters

x An array containing the values used for the dot plot.

n The number of elements in array *x*.

Description

The dot plot can be used to visually display the frequency of each number in a set of data. This member function places a dot on a graph each time for the corresponding number occurs in the array *x*.

Example

```
#include<chplot.h>

//int a[]={ 0, 2, 3, 5, 5};
//int a[]={ 0, 2, 3, -5, 5};
//int a[]={ 1, 2, 3, -5, 5};
//int a[]={ 2, 3, -5, 5};
```

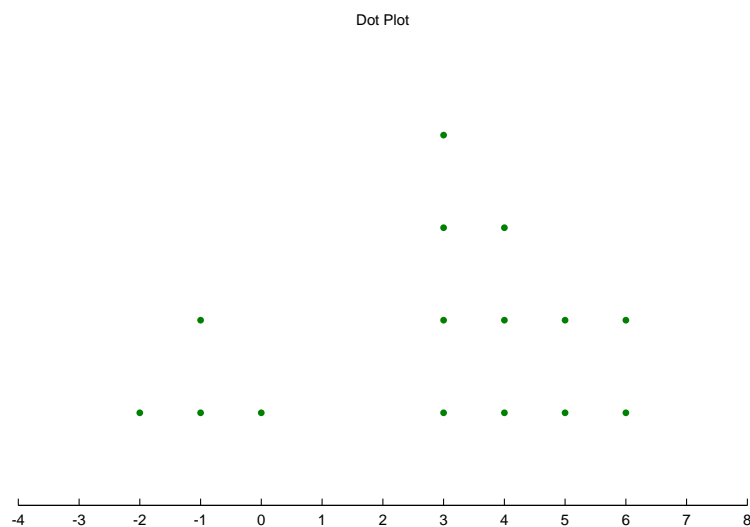
```

//int a[]={ 20, 23, 25, 28};
//int a[]={1, 2, 3, 5, 5};
//int a[]={ 2, 3, 5, 5};
//int a[]={ 1, 2, 3, 5, 5, 7, 3, 5, 4, 3, 5, 5, 5, 7, 13, 20, 4, 3, 5, 5, -5};
int a[]={-2, -1, 5, 0, 3, 4, 5, -1, 3, 3, 4, 6, 4, 3, 6};
CPlot plot;

plot.title("Dot Plot");
//plot.label(PLOT_AXIS_X, "random variable");
plot.label(PLOT_AXIS_X, "");
plot.dot(a);
plot.plotting();

```

Output



See Also

CPlot::point(), CPlot::scattern().

CPlot::ellipse

Synopsis

```
#include <chplot.h>
```

```
int ellipse(double x, double y, double width, double height, double angle);
```

Purpose

Add an ellipse to a 2D plot.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x The x coordinate of the center of the ellipse.

y The y coordinate of the center of the ellipse.

width The radius of the x-axis for the ellipse.

height The radius of the y-axis for the ellipse.

angle The orientation angle between the x-axis and the major diameter of the ellipse.

Description

This function adds an ellipse to a 2D plot. It is a convenience function for creation of geometric primitive. An ellipse added with this function is counted as a data set for later calls to **CPlot::legend()** and **CPlot::plotType()**. The *x* and *y* are the coordinates of the center of the ellipse. The *width* is the radius of the x-axis for the ellipse. The *height* is the radius of the y-axis for the ellipse. The *angle* is the orientation angle between the x-axis and the major diameter of the ellipse.

Please read the section **Handling Outline and Filled Color of Objects in CPlot** described in **CPlot::circle()** on page 51 about how to handle the outline and filled color for an ellipse.

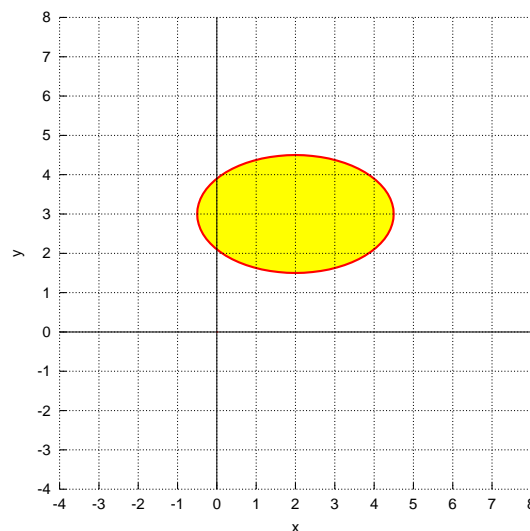
Example 1

```
/* File: ellipse.ch */
#include <chplot.h>

int main() {
    double x= 2, y= 3, width= 5, height = 3, angle = 0;
    CPlot plot;

    plot.axisRange(PLOT_AXIS_X, -4, 8);
    plot.axisRange(PLOT_AXIS_Y, -4, 8);
    plot.ticsRange(PLOT_AXIS_X, 1);
    plot.ticsRange(PLOT_AXIS_Y, 1);
    plot.fillColor("yellow");
    plot.ellipse(x, y, width, height, angle);
    plot.sizeRatio(1);
    plot.plotting();
}
```

Output



See Also

CPlot::arc(), **CPlot::arcSector()**, **CPlot::circle()**, **CPlot::data2D()**, **CPlot::data2DCurve()**,

CPlot::line(), **CPlot::outputType()**, **CPlot::plotType()**, **CPlot::point()**, **CPlot::polygon()**, **CPlot::rectangle()**.

CPlot::enhanceText

Synopsis

```
#include <chplot.h>
void enhanceText();
```

Purpose

Use special symbols in text.

Return Value

None.

Parameters

None.

Description

This function turns on the enhanced text mode for terminal and output files in PostScript, PNG, JPEG, GIF formats that support additional text formatting information embedded in the text string. With this function call, special symbols can be used in text such as those passed arguments of member functions **CPlot::label()**, **CPlot::legend()**, **CPlot::text()**, **CPlot::title()**.

The syntax for the enhanced text is shown in Figure 2.1. The character codes for the enhanced text is shown in Figure 2.2. Braces can be used to place multiple-character text where a single character is expected, for example, 3^{2x} . To change the font and/or size, use the full form

```
{/[fontname][=fontsize | *fontscale] text}
```

For example, `{/Symbol=20 p}` is a 20-point π and `{/*0.5 B}` is a B at an half of whatever fontsize is currently in effect. The `'/'` character *must* be the first character after the `'{'` character to use the enhanced text.

You can access special symbols numerically by specifying `\character-code` in octal number, for example, `{/Symbol \160}` is the symbol for π .

You can escape control characters using `\\`.

The string `"\\326"` is for the symbol $\sqrt{\quad}$ and string `"\\140"` draws a bar on a symbol. The number of token `"\\140"`, inside a pair of braces `'{'` and `'}'`, for a bar can be changed to adjust the length of the bar on the math expression. To place an expression inside the radical symbol $\sqrt{\quad}$, multiple tokens `"\\140"` can be added inside a pair of braces. For example, use the string `"{/Symbol @{\140\140\140\140}\326}x+8"` for $\sqrt{x+8}$. If the radicand has superscript, the font size for the regular text and superscript may need to be changed to display the math expression properly.

For example, the string `"{/Symbol=11 @{\140\140\140\140\140\140\140\140}\326}2x^{/Symbol=6 2}+9x+14"` displays $\sqrt{2x^2 + 9x + 14}$ with 11 points for the regular text and 6 points for superscript.

For simplicity, the strings `"<="` and `">"` represent \leq and \geq , respectively. To print a superscript on a plot, the symbol `^` is used, such as x^2 for x^2 . To print a subscript on a plot, the symbol `_` is used, such as x_2 for x_2 . The string `"M_PI"` is for π .

	text	result
Superscripts are denoted by ^:	'10^{-2}'	10^{-2}
Subscripts are denoted by _:	'A_{j,k}'	$A_{j,k}$
Braces are not needed for single characters:	'e^x'	e^x
Use @ to align sub- and superscripts:	'x@^2_k'	x_k^2
Put the shorter of the two first:	'x@_0^{-3/2}y'	$x_0^{-3/2}y$
...rather than:	'x@^{-3/2}_0y'	$x_0^{-3/2}y$
Font changes are enclosed in braces:	'{/Helvetica m}'	m
...size, too:	'{/=8 m}'	m
...or both:	'{/Helvetica=18 m}'	m
Characters can be specified by code:	'{\120}'	P
...which is how to get nonkeyboard characters:	'{\267}'	•
Use keyboard characters or codes for other fonts:	'{/Symbol p\271 22/7}'	$\pi \neq 22/7$
Everything outside braces is in the default font:	'P = {/Symbol r}kT'	$P = \rho kT$
Space of a given size can be inserted with &:	'<junk>'	<junk>
	'&{junk}>'	< >
Special characters (^,_,{,},@,&,\) can be escaped by \:	'f{x,y\}'	f{x,y}
...or \\ if within a double-quoted string:	"f\\{x,y\\}"	f{x,y}

Everything can be done recursively:

the text

```
'{/Symbol=18 \362@_{/=9.6 0}^{/=12 \245}}
```

```
{/Helvetica e^{-{/Symbol m}^2/2} d}{/Symbol m = (p/2)^{1/2}}
```

produces the result:

$$\int_0^\infty e^{-\mu^2/2} d\mu = (\pi/2)^{1/2}$$

Note how font sizes and definitions are preserved across pairs of braces.

Figure 2.1: Syntax for the enhanced text.

T = text (here Times-Roman) S = Symbol Z = ZapfDingbats E = ISO Latin-1 encoding
 (the "E" character set is accessed via a member function CPlot::nativeCmd("set encoding"))

T	S	Z	E	T	S	Z	E	T	S	Z	E	T	S	Z	E	T	S	Z	E		
040				111	I	I	☆ I	162	r	ρ	□ r	256	fi	→	③ ④	327	.	↑	×		
041	!	!	✂ !	112	J	ϑ	★ J	163	s	σ	▲ s	257	fl	↓	④ -	330	↵	▲	∅		
042	"	∇	✂ "	113	K	K	★ K	164	t	τ	▼ t	260		°	⑤ °	331	^	→	Û		
043	#	#	✂ #	114	L	Λ	★ L	165	u	υ	◆ u	261	-	±	⑥ ±	332	√	↗	Ú		
044	\$	Ξ	✂ \$	115	M	M	★ M	166	v	ϖ	◇ v	262	†	"	⑦ ²	333	↔	→	Û		
045	%	%	✂ %	116	N	N	★ N	167	w	ω	◐ w	263	‡	≥	⑧ ³	334	↔	→	Ü		
046	&	&	✂ &	117	O	O	★ O	170	x	ξ	x	264	.	×	⑨ ´	335	↑	→	Ý		
047	'	ε	✂ '	120	P	Π	☆ P	171	y	ψ	y	265		∞	⑩ μ	336	⇒	→	Ɔ		
050	((✂ (121	Q	Θ	★ Q	172	z	ζ	■ z	266	¶	∂	⑪ ¶	337	↓	→	β		
051))	✂)	122	R	P	✱ R	173	{	{	‘ {	267	.	.	⑫ .	340	◇	→	à		
052	*	*	✂ *	123	S	Σ	* S	174			’	270	,	÷	⑬ ,	341	Æ	<	→	á	
053	+	+	✂ +	124	T	T	* T	175	}	}	“ }	271	„	≠	⑭ ¹	342	®	>	→	â	
054	,	,	✂ ,	125	U	Υ	✂ U	176	~	~	” ~	272	”	≡	⑮ °	343	ª	©	>	→	ã
055	-	-	✂ -	126	V	ς	* V	220			ı	273	»	≈	⑯ »	344	™	▶	→	ä	
056	.	.	✂ .	127	W	Ω	* W	221			`	274	⑰ ¼	345	Σ	→	→	å	
057	/	/	✂ /	130	X	Ξ	* X	222			˘	275	%		⑱ ½	346	{	→	→	æ	
060	0	0	✂ 0	131	Y	Ψ	* Y	223			ˆ	276		—	⑲ ¾	347		→	→	ç	
061	1	1	✂ 1	132	Z	Z	✂ Z	224			˜	277	ı	┘	⑳ ı	350	Ł	{	→	è	
062	2	2	✂ 2	133	[[* [225			-	300	ˆ	⌘	㉑ À	351	∅		→	é	
063	3	3	✂ 3	134	\	\	* \	226			˘	301	˘	⌘	㉒ Á	352	œ		→	ê	
064	4	4	✂ 4	135]]]]	*]	227			˙	302	˘	⌘	㉓ Â	353	°		→	ë	
065	5	5	✂ 5	136	^	⊥	✂ ^	230			¨	303	˘	⌘	㉔ Ã	354	{	→	→	ì	
066	6	6	✂ 6	137	_	_	✂ _	232			°	304	˘	⊗	㉕ Ä	355	{	→	→	í	
067	7	7	✂ 7	140	‘	—	✂ ‘	233			˘	305	-	⊕	㉖ Å	356	{	→	→	î	
070	8	8	✂ 8	141	a	α	✂ a	235			˘	306	˘	∅	㉗ Æ	357		→	→	ï	
071	9	9	✂ 9	142	b	β	* b	236			˘	307	.	∩	㉘ Ç	360				ð	
072	:	:	✂ :	143	c	χ	✂ c	237			˘	310	¨	∪	㉙ È	361	æ	>	→	ñ	
073	;	;	✂ ;	144	d	δ	✂ d	240	€			311	˘	⊃	㉚ É	362	{	→	→	ò	
074	<	<	✂ <	145	e	ε	✂ e	241	ı	Υ	ı	312	˘	⊇	㉛ Ê	363	{	→	→	ó	
075	=	=	✂ =	146	f	φ	✂ f	242	ı	’	ı	313	˘	⊂	㉜ Ë	364		→	→	ô	
076	>	>	✂ >	147	g	γ	✂ g	243	£	≤	ı	314		⊃	㉝ Ì	365	ı	┘	→	õ	
077	?	?	✂ ?	150	h	η	✂ h	244	/	/	ı	315	˘	⊆	㉞ Í	366		→	→	ö	
100	@	≡	✂ @	151	i	ι	* i	245	¥	∞	ı	316	˘	⊆	㉟ Î	367		→	→	÷	
101	A	A	✂ A	152	j	φ	* j	246	f	f	ı	317	˘	⊈	㊱ Ï	370	ı	┘	→	ø	
102	B	B	✂ B	153	k	κ	* k	247	§	♣	ı	320	—	∠	㊲ Ð	371	∅		→	ù	
103	C	X	✂ C	154	l	λ	● l	250	¤	◆	ı	321		∇	㊳ Ñ	372	œ		→	ú	
104	D	Δ	✂ D	155	m	μ	○ m	251	’	♥	ı	322	®	㉑	㊴ Ò	373	β		→	û	
105	E	E	✂ E	156	n	v	■ n	252	“	♠	ı	323	©	㉒	㊵ Ó	374		→	→	ü	
106	F	Φ	✂ F	157	o	o	□ o	253	«	↔	ı	324	™	▶	㊶ Ô	375	}	→	→	ý	
107	G	Γ	✂ G	160	p	π	□ p	254	<	←	㉑	325	Π	→	㊷ Õ	376		→	→	þ	
110	H	H	✂ H	161	q	θ	□ q	255	>	↑	㉒	326	√	↔	㊸ Ö	377				ÿ	

Figure 2.2: character codes for the enhanced text.

Example 1

```

/* File: enhanceText.ch */
#include <math.h>
#include <chplot.h>

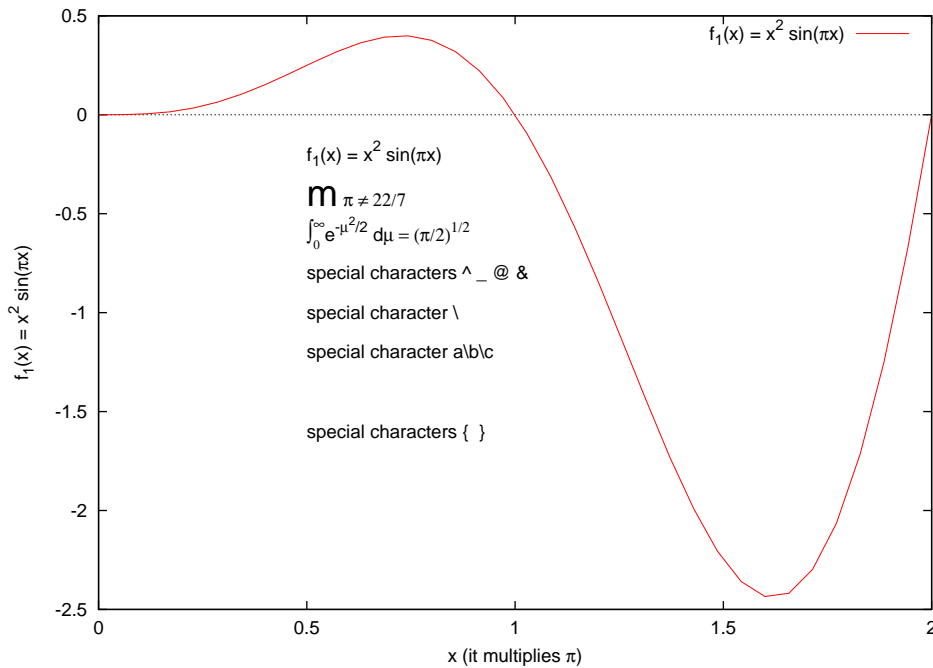
#define NUM 36
int main() {
    int i;
    double x[NUM], y[NUM];
    class CPlot plot;
    char funcname[] = "f_1(x) = x^2 sin({/Symbol p}x)";

    for(i=0; i< NUM; i++) {
        x[i] = i*(2.0)/(NUM-1);
        y[i] = x[i]*x[i]*sin(x[i]*M_PI);
    }
    plot.data2DCurve(x, y, NUM);
    plot.enhanceText();
    plot.label(PLOT_AXIS_X, "x (it multiplies {/Symbol p})");
    plot.label(PLOT_AXIS_Y, funcname);
    plot.legend(funcname, 0);
    plot.text(funcname, PLOT_TEXT_LEFT, 0.5, -0.2, 0);
    plot.text("{/Helvetica=28 m} {/Symbol p} \\271 22/7", PLOT_TEXT_LEFT, 0.5, -0.4, 0);
    plot.text("{/Symbol=18} \\362@_{/=9.6 0}^{/=12} \\245}"
        "{/Helvetica} e^{-{/Symbol m}^2/2} d{/Symbol m} = (p/2)^{1/2}",
        PLOT_TEXT_LEFT, 0.5, -0.6, 0);
    plot.text("special characters \\|^ \\_ \\@ \\&", PLOT_TEXT_LEFT, 0.5, -0.8, 0);
    plot.text("special character \\134 ", PLOT_TEXT_LEFT, 0.5, -1.0, 0);
    plot.text("special character a\\134b\\c", PLOT_TEXT_LEFT, 0.5, -1.2, 0);

    /* for display on the screen */
    plot.text("special characters \\{ \\} ", PLOT_TEXT_LEFT, 0.5, -1.4, 0);
    /* For postscript file use the format below to create '{' and '}' */
    //plot.text("special characters \\173 \\175", PLOT_TEXT_LEFT, 0.5, -1.6, 0);
    //plot.outputType(PLOT_OUTPUTTYPE_FILE, "postscript eps color", "../output/enhanceText.eps");
    plot.plotting();
    return 0;
}

```

Output

**See Also**

CPlot::label(), **CPlot::legend()**, **CPlot::text()**, **CPlot:::**

CPlot::expr

Synopsis

```
#include <chplot.h>
int expr(double x0, double xf, int n, string_t expr);
```

Purpose

Add a set of data based on an expression for 2D curve to an instance of **CPlot** class.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x0 the initial value for the domain of the function.

xf the final value for the domain of the function.

n the number of points for the domain of the function.

expr an expression in terms of *x*. The expression can also contain global variables.

Description

This function adds a set of data using an expression `expr`, in terms of *x*, in the domain from *x0* to *xf* with *n* points to a previously declared instance of the **CPlot** class.

Example 1

```

/* File: expr.ch */
#include<math.h>
#include<chplot.h>

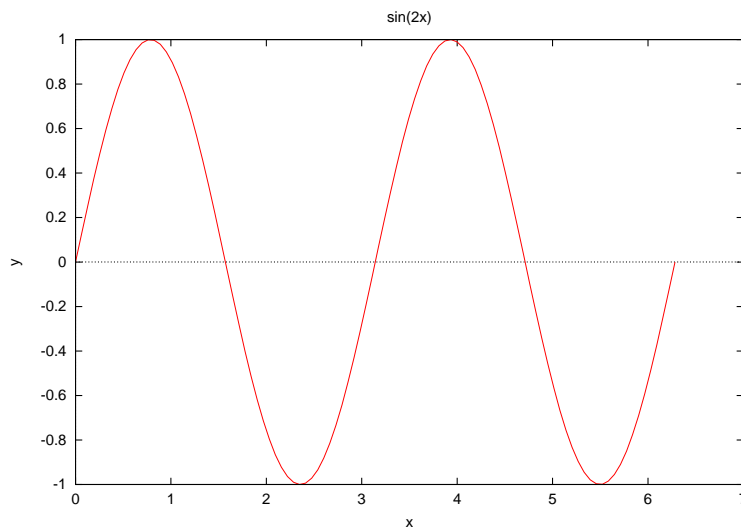
#define N 100

int main() {
    double x0, xf;
    CPlot plot;

    x0 = 0;
    xf = 2*M_PI;
    plot.title("sin(2x)");
    plot.expr(x0, xf, N, "sin(2*x)");
    plot.plotting();
}

```

Output



See Also

CPlot::exprPoint(), **CPlot::exprxy()**, **CPlot::func3D()**, **CPlot::funcp2D()**, **CPlot::funcp3D()**, **CPlot::data2D()**, **CPlot::data3D()**, **CPlot::data3DCurve()**, **CPlot::data3DSurface()**, **fplotxy()**.

CPlot::exprPoint

Synopsis

```
#include <chplot.h>
```

```
int exprPoint(double x0, double xf, int n, string_texpr);
```

Purpose

Add a set of data based on an expression for 2D curve to an instance of **CPlot** class for plotting points.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x_0 the initial value for the domain of the function.

x_f the final value for the domain of the function.

n the number of points for the domain of the function.

$expr$ an expression in terms of x . The expression can also contain global variables.

Description

This function adds a set of data using an expression $expr$, in terms of x , in the domain from x_0 to x_f with n points to a previously declared instance of the **CPlot** class. Unlike **CPlot::expr()**, adjacent plotting points are not connected.

Example 1

```
/* File: exprPoint.ch */
#include <chplot.h>
CPlot plot;

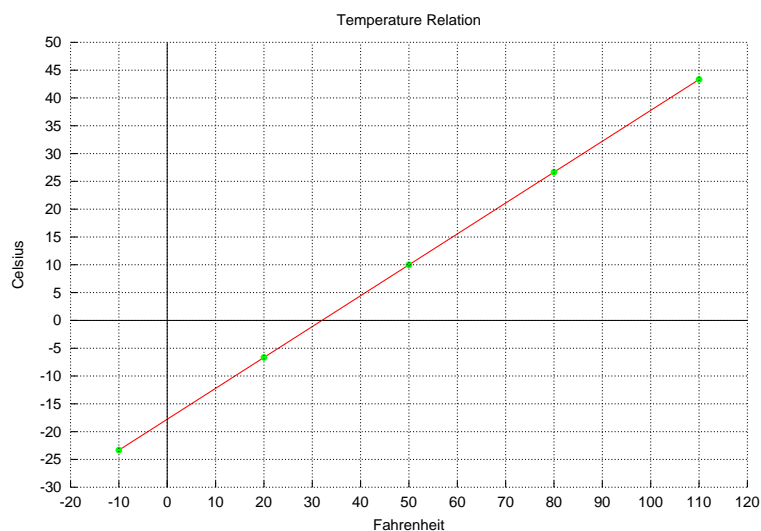
plot.title("Temperature Relation");
plot.label(PLOT_AXIS_X, "Fahrenheit");
plot.label(PLOT_AXIS_Y, "Celsius");
plot.axisRange(PLOT_AXIS_X, -20, 120);
plot.axisRange(PLOT_AXIS_Y, -30, 50);
plot.ticsRange(PLOT_AXIS_X, 10);
plot.ticsRange(PLOT_AXIS_Y, 5);

/* add a line with 2 points to the plot */
plot.expr(-10, 110, 2, "5*(x-32)/9");

/* add 5 points to the plot */
plot.exprPoint(-10, 110, 5, "5*(x-32)/9");

plot.plotting();
```

Output



See Also

CPlot::expr(), CPlot::exprxy(), CPlot::func3D(), CPlot::funcp2D(), CPlot::funcp3D(), CPlot::data2D(), CPlot::data3D(), CPlot::data3DCurve(), CPlot::data3DSurface(), fplotxy().

CPlot::exprxy

Synopsis

```
#include <chplot.h>
```

```
int exprxy(double x0, double xf, double y0, double yf, int nx, int ny, string_texpr);
```

Purpose

Add a set of data based on an expression for 3D surface to an instance of **CPlot** class.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x0 the initial value for the x domain of the function.

xf the final value for the x domain of the function.

y0 the initial value for the y domain of the function.

yf the final value for the y domain of the function.

nx the number of points for the x domain of the function.

ny the number of points for the y domain of the function.

expr an expression in terms of x and y. The expression can also contain global variables.

Description

This function adds a set of data using an expression *expr*, in terms of x and y, in the domain from *x0* to *xf* with *nx* points in the x-coordinate and in the domain from *y0* to *yf* with *ny* points in the y-coordinate to a previously declared instance of the **CPlot** class.

Example 1

```
/* File: exprxy.ch */
#include<math.h>
#include<chplot.h>

#define N 100

int main() {
    double x0, xf;
    double y0, yf;
    CPlot plot;

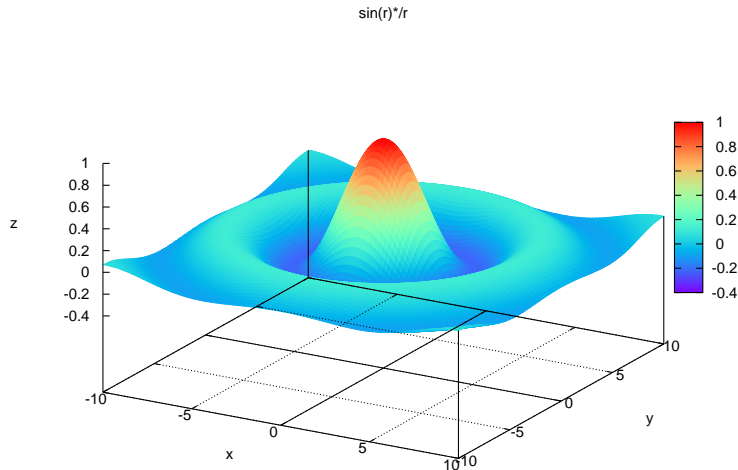
    x0 = -10;
    xf = 10;
    y0 = -10;
```

```

yf = 10;
plot.title("sin(r)*r");
plot.exprxy(x0, xf, y0, yf, N, N, "sin(sqrt(x*x+y*y))/sqrt(x*x+y*y)");
plot.plotting();
}

```

Output



See Also

CPlot::expr(), **CPlot::func3D()**, **CPlot::funcp2D()**, **CPlot::funcp3D()**, **CPlot::data2D()**, **CPlot::data3D()**, **CPlot::data3DCurve()**, **CPlot::data3DSurface()**, **fplotxy()**.

CPlot::fillColor

Synopsis

```

#include <chplot.h>
void fillColor(string_t color);

```

Purpose

Fill the color for interior of an object.

Return Value

None.

Parameters

color The color for the interior of an object.

Description

This function fill the color of the interior of an object, including the arc sector, arc segment, circle, ellipse, rectangle, quadrilateral, triangle, regular polyong, and general polygon. By default, the interior of an object is not filled with a color or pattern. Once the member function **CPlot::fillColor()** is called, the interior color will be kept for drawing objects until it is changed by calling this member function again with a different

filled color, or by the member function **CPlot::fillPattern()** to fill the interior with a pattern, or by the member function **CPlot::noFill()** to disable the filling.

The detailed color specification can be found in the member function **CPlot::strokeColor()**.

Example 1

```
/* File: fillColor.ch */
#include <chplot.h>
CPlot plot;

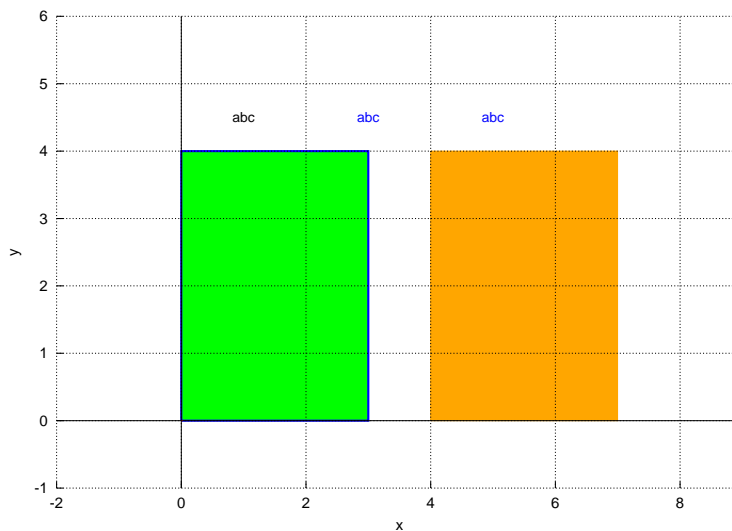
plot.text("abc", PLOT_TEXT_CENTER, 1, 4.5);

plot.strokeColor("blue");
plot.fillColor("green");
plot.rectangle(0, 0, 3, 4);
plot.text("abc", PLOT_TEXT_CENTER, 3, 4.5);

plot.noStrokeColor();
plot.fillColor("orange");
plot.rectangle(4, 0, 7, 4);
plot.text("abc", PLOT_TEXT_CENTER, 5, 4.5);

plot.plotting();
```

Output



See Also

CPlot::fillOpacity(), **CPlot::fillPattern()**, **CPlot::noFill()**, **CPlot::noStrokeColor()**, **CPlot::strokeColor()**, **CPlot::strokeWidth()**.

CPlot::fillCurve

Synopsis

```
#include <chplot.h>
```

```
void fillCurve(string_t option, string_t color);
```

Purpose

Fill the curve with respect to a reference with the color.

Return Value

None.

Parameters

option This option specify how the curve will be filled.

color The color for the filled curve.

Description

This function fills the curve with respect to a reference with the color for a 2D plot.

Three variants are possible. The first two variants require two columns of input data, and may be further modified by the options listed below. The first variant, `closed`, treats the curve itself as a closed polygon. This is the default. The second variant is to fill the area between the curve and a given axis, a horizontal or vertical line, or a point. The third variant requires three columns of input data: the x coordinate and two y coordinates corresponding to two curves sampled at the same set of x coordinates; the area between the two curves is filled.

The option of the filled curve `fillCurve()` is as follows.

```
{ [closed | {above | below} {x1 | x2 | y1 | y2}[=<a>] | xy=<x>,<y>]
  {linetype | lt <line_type>}
}
```

The plot variants for `fillCurve()` can be further modified by the options

```
closed    ... just filled closed curve,
x1        ... x1 axis,
x2        ... x2 axis, etc for y1 and y2 axes,
y1=0     ... line y=0 (at y1 axis) ie parallel to x1 axis,
y2=42    ... line y=42 (at y2 axis) ie parallel to x2, etc,
xy=10,20 ... point 10,20 of x1,y1 axes (arc-like shape).
```

An example of filling the area between two input curves using three columns of data is as follows.

```
plot.fillCurve(option, color);
plot.dataFile("datafile", "using 1:2:3");
```

The above and below in the form

```
above {x1|x2|y1|y2}=<val>
below {x1|x2|y1|y2}=<val>
```

limit the filled area to one side of the bounding line or curve.

If the values of `<a>`, `<x>`, `<y>` are out of the drawing boundary, then they are moved to the graph boundary. Then the actually filled area in the case of option `xy=<x>,<y>` will depend on the x-range and y-range.

The second argument `color` changes the color of the filled area. For example,

```
plot.fillCurve("y1=0", "blue");
```

The detailed color specification can be found in the member function `CPlot::strokeColor()`.

Example 1

```

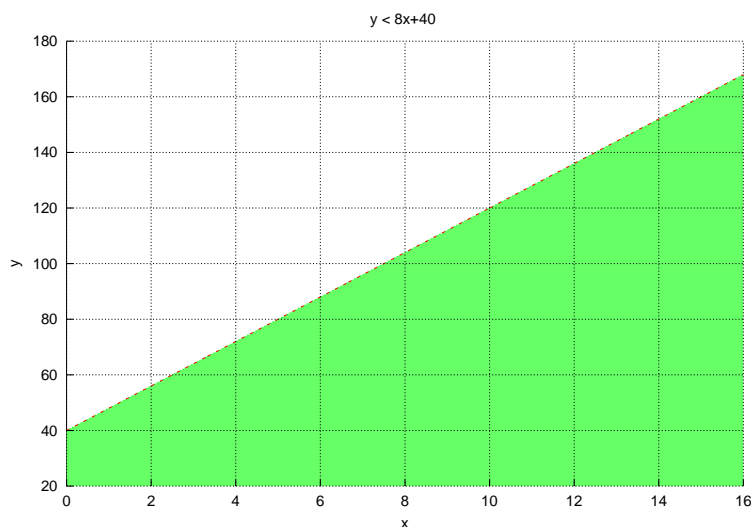
/* File: fillCurve.ch
   Plotting the inequality  $y < 8x+40$  for  $x$  from 0 to 16. */
#include <chplot.h> /* for CPlot */
CPlot plot;

plot.title("y < 8x+40 "); /* for "y < 8x+40" */
plot.label(PLOT_AXIS_X, "x");
plot.label(PLOT_AXIS_Y, "y");
plot.axisRange(PLOT_AXIS_Y, 20, 180);

plot.strokeColor("red");
plot.strokeWidth(3);
plot.lineStyle(".");
plot.fillCurve("y1=20", "green");
plot.expr(0, 16, 500, "8*x+40");
plot.plotting();

```

Output



Example 2

```

/* File: fillCurve2.ch
   Plotting the inequality  $y < 8x+40$  for  $x$  from 0 to 16. */
#include <chplot.h> /* for CPlot */
CPlot plot;

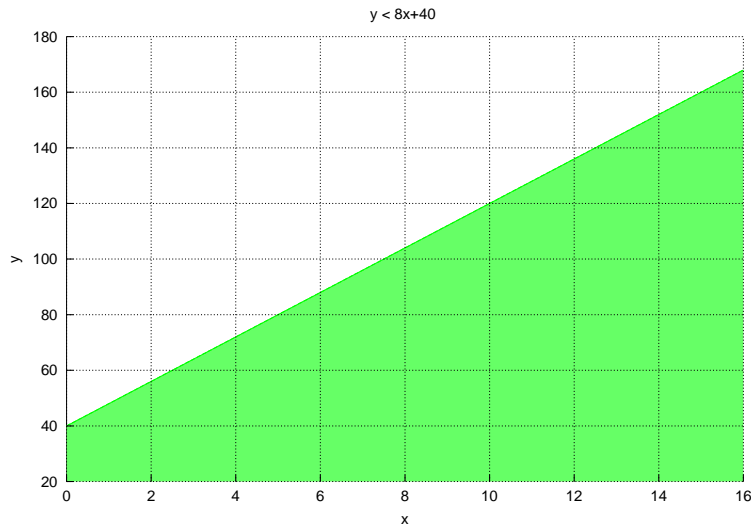
plot.title("y < 8x+40 "); /* for "y < 8x+40" */
plot.label(PLOT_AXIS_X, "x");
plot.label(PLOT_AXIS_Y, "y");
plot.axisRange(PLOT_AXIS_Y, 20, 180);

plot.strokeColor("green"); // stroke color is the same as filled color
//plot.strokeWidth(3);

```

```
//plot.lineStyle(".");
plot.fillCurve("y1=20", "green");
plot.expr(0, 16, 500, "8*x+40");
plot.plotting();
```

Output



Example 3

```
/* File: fillCurve3.ch
   Plotting the inequalities  $y < -x+4$  and  $y \leq x+1$ 
   for  $x$  from  $-5$  to  $5$  and  $y$  from  $-10$  to  $10$ . */
#include <chplot.h> /* for CPlot */
CPlot plot;

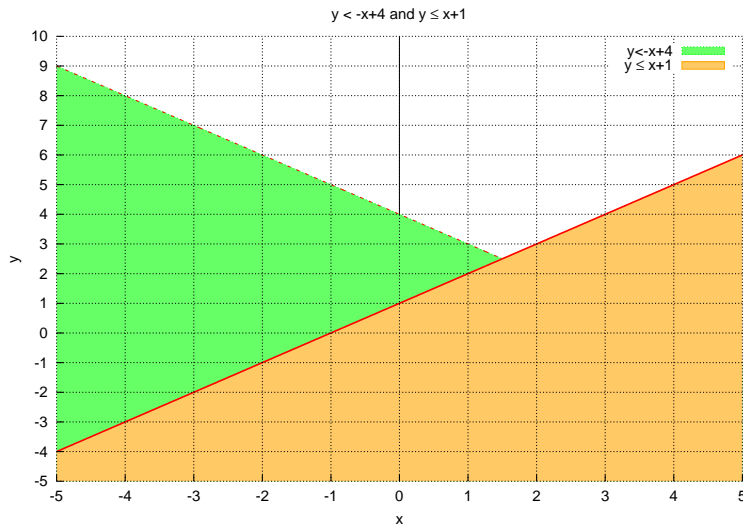
plot.title("y < -x+4 and y <= x+1"); // y > x-3 and y <= x+1
plot.label(PLOT_AXIS_X, "x");
plot.label(PLOT_AXIS_Y, "y");
plot.axisRange(PLOT_AXIS_Y, -5, 10);
plot.ticsRange(PLOT_AXIS_XY, 1);

/* plot  $y < -x+4$  */
plot.strokeColor("red");
plot.strokeWidth(3);
plot.lineStyle(".");
plot.fillCurve("y1=-5", "green");
plot.legend("y<-x+4");
plot.expr(-5, 5, 500, "-x+4");

/* plot  $y \leq x+1$  */
plot.lineStyle("-");
plot.fillCurve("y1=-5", "orange");
plot.legend("y <= x+1");
plot.expr(-5, 5, 500, "x+1");

plot.plotting();
```

Output

**See Also**

CPlot::fillColor(), CPlot::plotType(), CPlot::strokeColor().

CPlot::fillOpacity

Synopsis

```
#include <chplot.h>
```

```
void fillOpacity(double opacity);
```

Purpose

Set the opacity for the filled color for the interior of an object.

Return Value

None.

Parameters

opacity The opacity for the interior of an object from 0 to 1.

Description

The member function **CPlot::fillColor()** will fill the color of the interior of an object, including the arc sector, arc segment, circle, ellipse, rectangle, quadrilateral, triangle, regular polygon, and general polygon. The member function **CPlot::fillOpacity()** will set the opacity for the filled color for an object. When the argument for the opacity is 0, the object is completely transparent. When the opacity is 1, the filled color is solid.

Once the member function **CPlot::fillOpacity()** is called, the opacity for the filled color for the interior of object will be kept for drawing objects until it is changed by calling this member function again with a different opacity.

Example 1

```
/* File: fillOpacity.ch */
#include <chplot.h>
```

```

CPlot plot;

plot.title("Venna Diagram");
plot.fillOpacity(0.5);
//plot.noStrokeColor();
plot.fillOpacity(0.5);

plot.fillColor("red");
plot.circle(3, 4, 8);

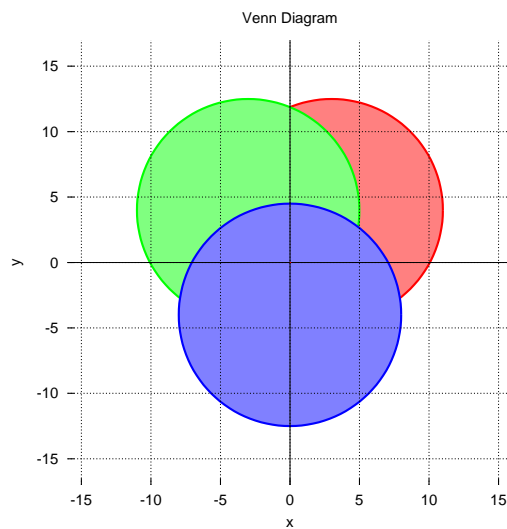
plot.fillColor("green");
plot.circle(-3, 4, 8);

plot.fillColor("blue");
plot.circle(0, -4, 8);

plot.sizeRatio(1);
plot.plotting();

```

Output



See Also

CPlot::fillColor(), **CPlot::fillPattern()**, **CPlot::noFill()**, **CPlot::noStrokeColor()**, **CPlot::strokeColor()**, **CPlot::strokeWidth()**.

CPlot::fillPattern

Synopsis

```

#include <chplot.h>
void fillPattern(int num);

```

Purpose

Fill the interior of an object with a pattern.

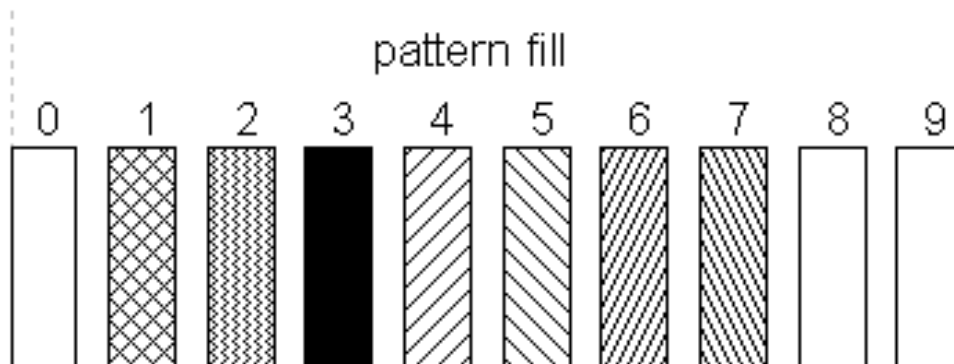
Return Value

None.

Parameters*num* The pattern number for the interior of an object.**Description**

This function fill the interior of an object with a pattern, The object includes the arc sector, arc segment, circle, ellipse, rectangle, quadrilateral, triangle, and polygon. By default, the interior of an object is not filled with a color or pattern. Once the member function **CPlot::fillPattern()** is called, the filled pattern for the interior of object will be kept for drawing objects until it is changed by calling this member function again with a different pattern number, or by the memner function **CPlot::fillColor()** to fill the interior with a color, or by the member function **CPlot::noFill()** to disable the filling.

The kind and number of available fill patterns depend on the terminal driver. If multiple datasets using filled boxes are plotted, the pattern cycles through all available pattern types, starting from *num*, much as the line type cycles for multiple line plots. The available patterns in Windows are shown below.



In this version of Ch, to fill the interior of an object, the background color should be changed from white to other color.

Example 1

```

/* File: fillPattern.ch */
#include <chplot.h>
CPlot plot;

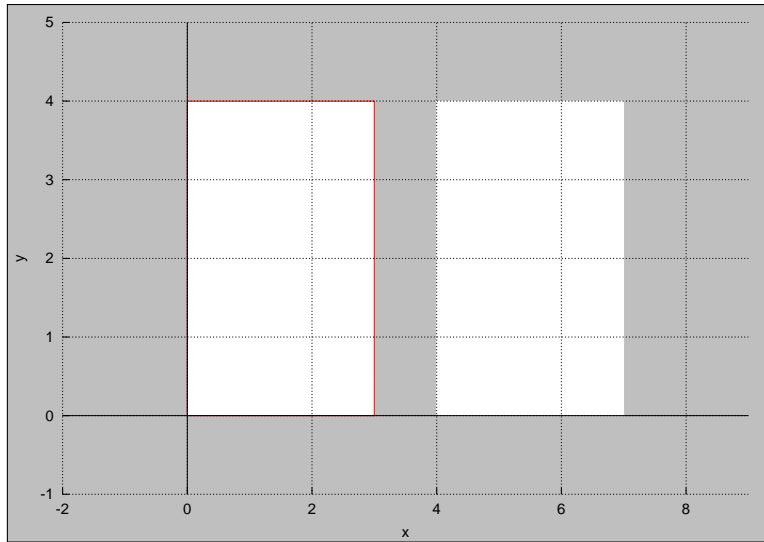
/* need a background color to show a pattern by fillPattern() */
plot.backgroundColor("grey");
plot.fillPattern(1);
plot.rectangle(0, 0, 3, 4);

plot.noStrokeColor();
plot.fillPattern(5);
plot.rectangle(4, 0, 3, 4);

plot.plotting();

```

Output

**See Also**

CPlot::fillColor(), **CPlot::fillOpacity()**, **CPlot::noFill()**, **CPlot::noStrokeColor()**, **CPlot::strokeColor()**, **CPlot::strokeWidth()**.

CPlot::fillStyle

Synopsis

```
#include <chplot.h>
```

```
void fillStyle(int num, int style, ... /* [double density], int patternnum */);
```

Syntax

```
fillStyle(num, PLOT_FILLSTYLE_EMPTY)
```

```
fillStyle(num, PLOT_FILLSTYLE_SOLID)
```

```
fillStyle(num, PLOT_FILLSTYLE_SOLID, density)
```

```
fillStyle(num, PLOT_FILLSTYLE_TRANSPARENT_SOLID, density)
```

```
fillStyle(num, PLOT_FILLSTYLE_PATTERN)
```

```
fillStyle(num, PLOT_FILLSTYLE_PATTERN, patternnum)
```

```
fillStyle(num, PLOT_FILLSTYLE_TRANSPARENT_PATTERN, patternnum)
```

Purpose

Fill a box or filled curve with a solid color or pattern, or transparent solid color or pattern.

Return Value

None.

Parameters

num The data set of plot type of box or filled curve to be filled. If *num* is -1, the fill style will be applied to all boxes and filled curves for the plot.

style The *style* parameter can take one of the following values:

PLOT_FILLSTYLE_EMPTY Do not fill the box. The default is to have an empty box.

PLOT_FILLSTYLE_SOLID Fill the boxes or filled curves with a solid color.

PLOT_FILLSTYLE_TRANSPARENT_SOLID Fill the boxes or filled curves with a transparent solid color.

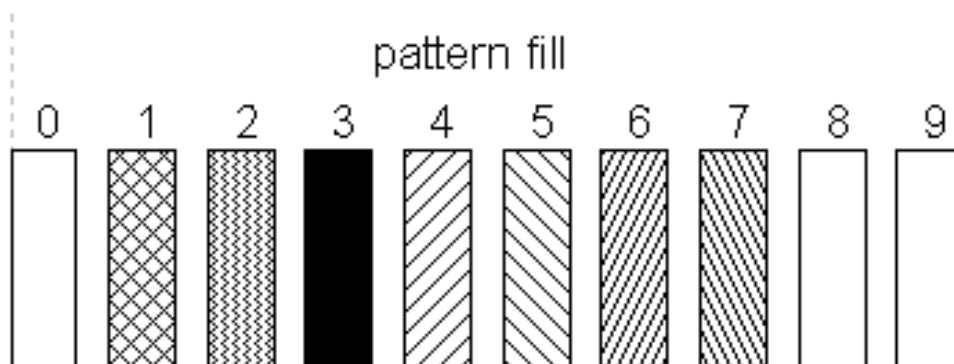
PLOT_FILLSTYLE_PATTERN Fill the boxes or filled curves with a pattern.

PLOT_FILLSTYLE_TRANSPARENT_PATTERN Fill the boxes or filled curves with a transparent pattern.

density The density of solid color in the range of [0, 1.0]. If the value for `density` is 0.0, the box is empty. If it is 1.0, the inner area is of the same color as the current line type. If no `density` parameter is given, it defaults to 1.0.

In the case of transparent solid fill areas, the density parameter is interpreted as an alpha value; that is, density 0 is fully transparent, density 1 is fully opaque.

patternnum The parameter `patternnum` option causes filling to be done with a fill pattern supplied by the terminal driver. The kind and number of available fill patterns depend on the terminal driver. If multiple datasets using filled boxes are plotted, the pattern cycles through all available pattern types, starting from `patternnum`, much as the line type cycles for multiple line plots. The available patterns in Windows are shown below.



In the case of transparent pattern fill, the background of the pattern is either fully transparent or fully opaque.

Description

This function specifies how boxes and filled curves are filled with a solid color or pattern. The fill style can be applied to plot types of **PLOT_PLOTTYPE_BOXES**, **PLOT_PLOTTYPE_BOXERRORBARS**, **PLOT_PLOTTYPE_BOXXYERRORBARS**, **PLOT_PLOTTYPE_CANDLESTICKS**, and **PLOT_PLOTTYPE_FILLEDCURVES** specified by member function **CPlot::plotType()**.

There may be additional limitations on the creation or viewing of graphs containing transparent fill areas. For example, the png terminal can only use transparent fill if the "truecolor" option is set by

```
plot.outputType(PLOT_OUTPUTTYPE_FILE, "png truecolor", "filename.png")
```

Postscript files only support transparent pattern-fill areas.

Portability

Not all platforms support **PLOT_FILLSTYLE_TRANSPARENT_SOLID** and **PLOT_FILLSTYLE_TRANSPARENT_PATTERN**.

Example

This example generates a plot with transparent solid colors. See also `mathCoord2.ch` in `CPlot::mathCoord()`.

```

/* File: fillStyle.ch
   Plotting the inequality  $y < 8x+40$  and  $y < 10x+30$  for  $x$  from 0 to 16. */
#include <chplot.h> /* for CPlot */

double func(double x) {
    double y;
    y = 8 * x + 40;
    return y;
}

double func2(double x) {
    double y;
    y = 10 * x + 30;
    return y;
}

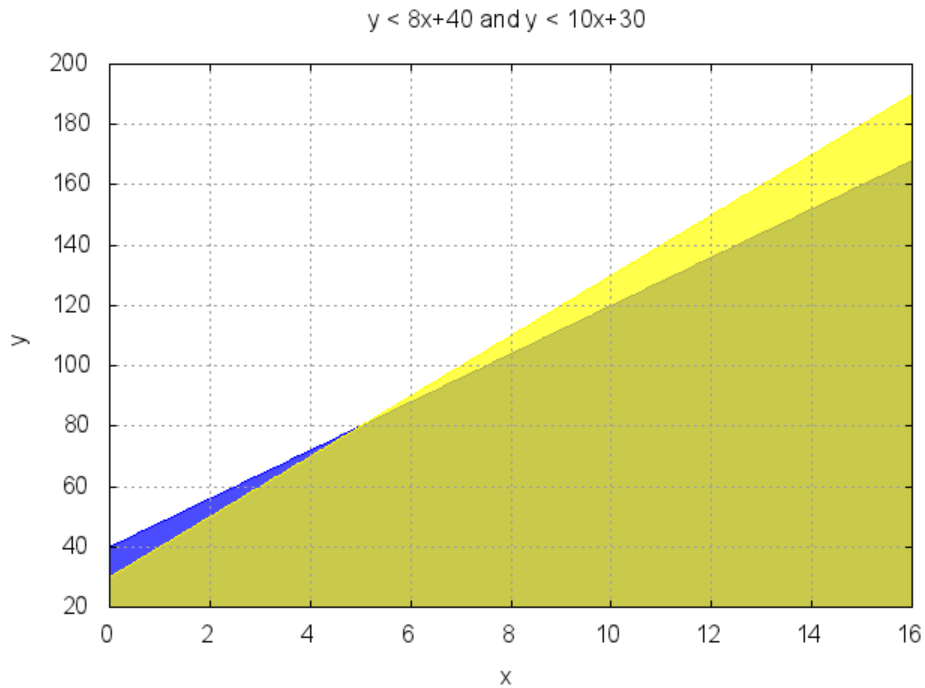
int main() {
    CPlot plot;
    int num = 0;

    plot.title("y < 8x+40 and y < 10x+30");
    plot.label(PLOT_AXIS_X, "x");
    plot.label(PLOT_AXIS_Y, "y");
    plot.axisRange(PLOT_AXIS_Y, 20, 200);
    plot.grid(PLOT_ON, "front");
    plot.func2D(0, 16, 500, func);
    plot.plotType(PLOT_PLOTTYPE_FILLEDCURVES, num, "y1=20", "blue");
    /* transparent solid blue color with density 0.6 */
    plot.fillStyle(num, PLOT_FILLSTYLE_TRANSPARENT_SOLID, 0.6);

    plot.func2D(0, 16, 500, func2);
    num++;
    plot.plotType(PLOT_PLOTTYPE_FILLEDCURVES, num, "y1=20", "yellow");
    /* transparent solid yellow color with density 0.6 */
    plot.fillStyle(num, PLOT_FILLSTYLE_TRANSPARENT_SOLID, 0.6);
    plot.plotting();
    return 0;
}

```

Output

**See Also**

CPlot::boxBorder(), **CPlot::boxWidth()**, **CPlot::mathCoord()**, **CPlot::plotType()**.

CPlot::font

Synopsis

```
#include <chplot.h>
```

```
void font(char *fontname, int fontsize);
```

Purpose

Set the font and font size used for all texts in the plot.

Return Value

None.

Parameter

fontname The font name for the font used in all texts in the plot.

fontsize The font size for the font used in all texts in the plot.

Description

This function sets the font name and size used in all texts such as title, label, legend, and text in the plot. The commonly used font names such as "Sans", "Arial", "Helvetica", "Times", "Times-New-Roman" are available. By default, the font name "Sans" and font size 10 are used.

Example 1

In this example, the smaller font size 8 is used so that tick marks on the x-axis can fit in the space. Compare the output with the that for the example in **CPlot::mathCoord()**.

```

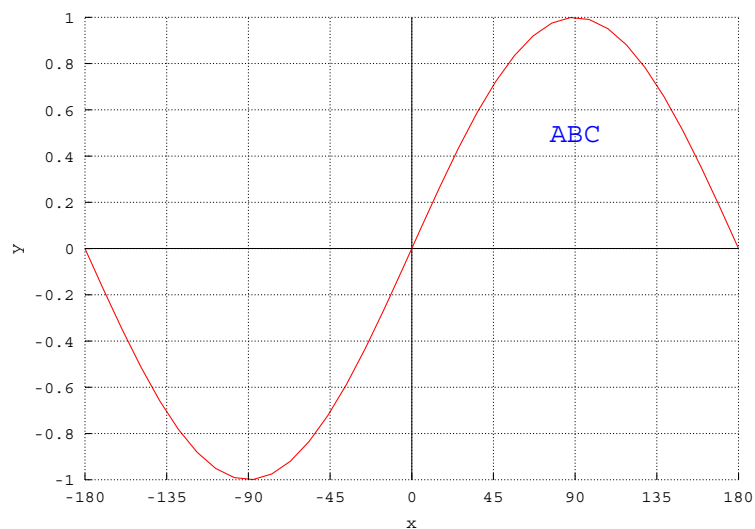
/* File: font.ch */
#include <math.h>
#include <chplot.h>
#include <numeric.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    CPlot plot;

    lndata(-180, 180, x);
    y = sin(x*M_PI/180);
    plot.data2D(x, y);
    plot.ticsRange(PLOT_AXIS_X, 45);
    /* change the default font size from 10 to 8 */
    plot.textColor("blue");
    plot.font("Sans", 8);
    plot.fontScale(2); /* use default font scale */
    plot.text("ABC", PLOT_TEXT_CENTER, 90, 0.5);
    plot.plotting();
}

```

Output



See Also

CPlot::fontScale().

CPlot::fontScale

Synopsis

```

#include <chplot.h>
void fontScale(int scale);

```

Purpose

Scale the font used for all texts in the plot.

Return Value

None.

Parameter

scale The scaling factor for the font used in all texts in the plot. The value should be larger than 0.

Description

This function scales all texts such as title, label, legend, and text in the plot by the factor *scale*. If the scaling factor *scale* is larger than 1, the font size will be bigger than the default one. If the scaling factor *scale* is larger than 0, but less than 1, the font size will be smaller than the default one. By default, the scaling factor is 1.

Portability

Not all platforms support this feature.

Example 1

See an example in `CPlot::font()`.

See Also

`CPlot::font()`.

CPlot::func2D

Synopsis

```
#include <chplot.h>
```

```
int func2D(double x0, double xf, int n, double (*func)(double x));
```

Purpose

Add a set of data using a function for 2D curve to an instance of **CPlot** class.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x0 the initial value for the range of the function.

xf the final value for the range of the function.

n the number of points for the range of the function.

func a pointer to function for adding a set of data.

Description

This function adds a set of data using a function `func()` in the range from *x0* to *xf* with *n* points to a previously declared instance of the **CPlot** class.

Example 1

```
/* File: func2D.ch */
#include<math.h>
```

```

#include<chplot.h>

#define N 100

double omega;
double func(double x) {
    double y;

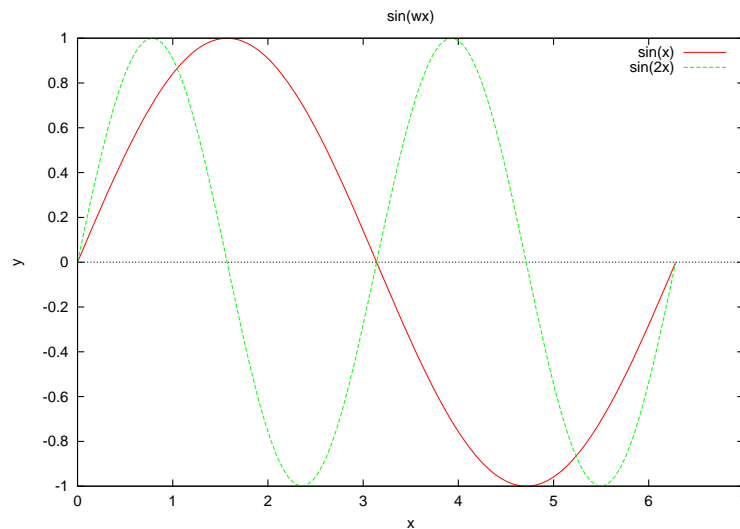
    y = sin(omega*x);
    return y;
}

int main() {
    double x0, xf;
    CPlot plot;

    x0 = 0;
    xf = 2*M_PI;
    plot.title("sin(wx)");
    plot.func2D(x0, xf, N, sin);
    plot.legend("sin(x)", 0);
    omega = 2;
    plot.func2D(x0, xf, N, func);
    plot.legend("sin(2x)", 1);
    plot.plotting();
}

```

Output



See Also

CPlot::func2DPoint(), **CPlot::func3D()**, **CPlot::funcp2D()**, **CPlot::funcp3D()**, **CPlot::data2D()**, **CPlot::data3D()**, **CPlot::data3DCurve()**, **CPlot::data3DSurface()**, **fplotxy()**.

CPlot::func2DPoint

Synopsis

```
#include <chplot.h>
```



```
int func2DPoint(double x0, double xf, int n, double (*func)(double x));
```

Purpose

Add a set of data using a function for 2D curve to an instance of **CPlot** class for plotting points.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x0 the initial value for the range of the function.

xf the final value for the range of the function.

n the number of points for the range of the function.

func a pointer to function for adding a set of data.

Description

This function adds a set of data using a function `func()` in the range from `x0` to `xf` with `n` points to a previously declared instance of the **CPlot** class. Unlike **CPlot::func2DPoint()**, adjacent plotting points are not connected.

Example 1

```
/* File: func2DPoint.ch */
#include <chplot.h>
CPlot plot;

double func(double x) {
    return 5*(x-32)/9;
}

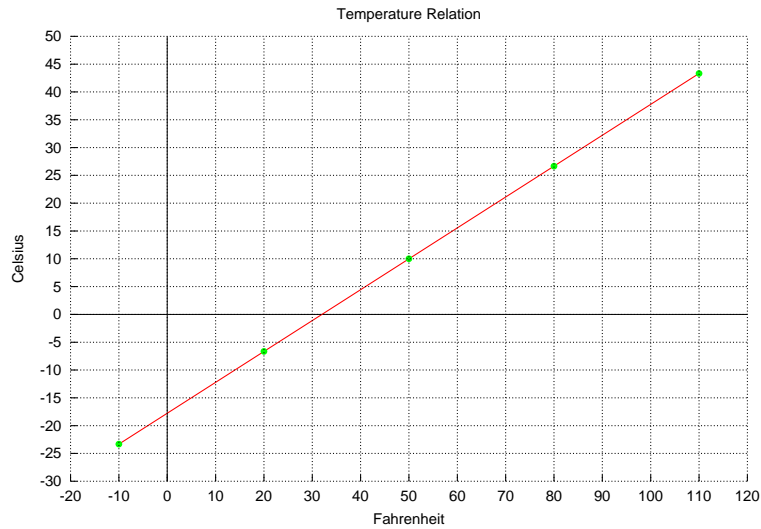
plot.title("Temperature Relation");
plot.label(PLOT_AXIS_X, "Fahrenheit");
plot.label(PLOT_AXIS_Y, "Celsius");
plot.axisRange(PLOT_AXIS_X, -20, 120);
plot.axisRange(PLOT_AXIS_Y, -30, 50);
plot.ticsRange(PLOT_AXIS_X, 10);
plot.ticsRange(PLOT_AXIS_Y, 5);

/* add a line with 2 points to the plot */
plot.func2D(-10, 110, 2, func);

/* add 5 points to the plot */
plot.func2DPoint(-10, 110, 5, func);

plot.plotting();
```

Output

**See Also**

CPlot::func2D(), **CPlot::func3D()**, **CPlot::funcp2D()**, **CPlot::funcp3D()**, **CPlot::data2D()**, **CPlot::data3D()**, **CPlot::data3DCurve()**, **CPlot::data3DSurface()**, **fplotxy()**.

CPlot::func3D

Synopsis

```
#include <chplot.h>
```

```
int func3D(double x0, double xf, double y0, double yf, int nx, int ny, double (*func)(double x, double y));
```

Purpose

Add a set of data using a function for 3D surface to an instance of **CPlot** class.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x0 the initial value for the x range of the function.

xf the final value for the x range of the function.

y0 the initial value for the y range of the function.

yf the final value for the y range of the function.

nx the number of points for the x range of the function.

ny the number of points for the y range of the function.

func a pointer to function for adding a set of data.

Description

This function adds a set of data using a function `func()` with `nx` points in the range from `x0` to `xf` for `x`

and with ny points in the range from y_0 to y_f for y to a previously declared instance of the **CPlot** class.

Example 1

```
/* File: func3D.ch */
#include<math.h>
#include<chplot.h>

#define NX 50
#define NY 50

double offset;

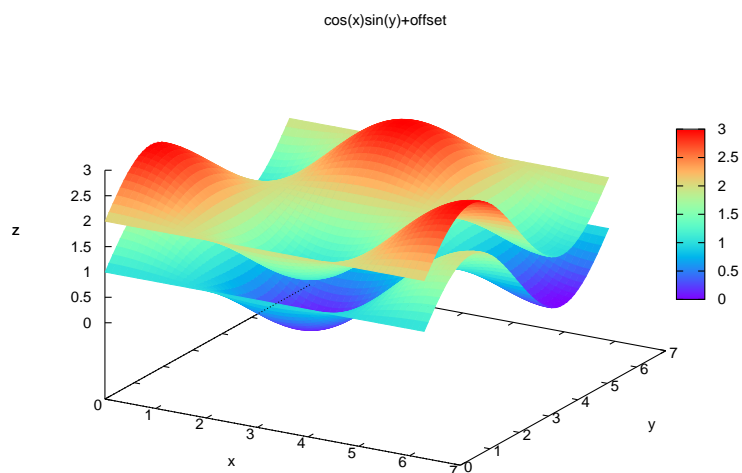
double func(double x, double y) {
    double z;

    z = cos(x)*sin(y) +offset;
    return z;
}

int main() {
    double x0, xf, y0, yf;
    CPlot plot;

    x0 = 0;
    xf = 2*M_PI;
    y0 = 0;
    yf = 2*M_PI;
    offset = 1;
    plot.title("cos(x)sin(y)+offset");
    plot.func3D(x0, xf, y0, yf, NX, NY, func);
    offset = 2;
    plot.func3D(x0, xf, y0, yf, NX, NY, func);
    plot.plotting();
}
```

Output



See Also

CPlot::func2D(), CPlot::func3D(), CPlot::funcp2D(), CPlot::data2D(), CPlot::data3D(), CPlot::data3DCurve(), CPlot::data3DSurface(), fplotxy().

CPlot::funcp2D

Synopsis

```
#include <chplot.h>
```

```
int funcp2D(double x0, double xf, int n, double (*func)(double x, void * param, void * param));
```

Purpose

Add a set of data using a function for 2D curve to an instance of **CPlot** class.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x0 the initial value for the range of the function.

xf the final value for the range of the function.

n the number of points for the range of the function.

func a pointer to function for adding a set of data.

param a parameter passed to the function. If this argument is not used inside function `func()`, NULL can be passed to this parameter.

Description

This function adds a set of data using a function with a parameter `func()` in the range from `x0` to `xf` with `n` points to a previously declared instance of the **CPlot** class.

Example 1

```
/* File: funcp2D.ch */
#include<math.h>
#include<chplot.h>

#define N 100

double func(double x, void *param) {
    double omega, y;

    omega = *(double*)param;
    y = sin(omega*x);
    return y;
}

int main() {
    double x0, xf, omega;
    CPlot plot;
```

```

x0 = 0;
xf = 2*M_PI;
omega = 1;
plot.title("sin(wx)");
plot.funcp2D(x0, xf, N, func, &omega);
plot.legend("sin(x)", 0);
omega = 2;
plot.funcp2D(x0, xf, N, func, &omega);
plot.legend("sin(2x)", 1);
plot.plotting();
}

```

Output

See **CPlot::func2D()**.

See Also

CPlot::func2D(), **CPlot::funcp3D()**, **CPlot::funcp3D()**, **CPlot::data2D()**, **CPlot::data3D()**, **CPlot::data3DCurve()**, **CPlot::data3DSurface()**, **fplotxy()**.

CPlot::funcp3D

Synopsis

```
#include <chplot.h>
```

```
int funcp3D(double x0, double xf, double y0, double yf, int nx, int ny, double (*func)(double x, double y, void * param, void * param));
```

Purpose

Add a set of data using a function for 3D surface to an instance of **CPlot** class.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x0 the initial value for the x range of the function.

xf the final value for the x range of the function.

y0 the initial value for the y range of the function.

yf the final value for the y range of the function.

nx the number of points for the x range of the function.

ny the number of points for the y range of the function.

func a pointer to function for adding a set of data.

param a parameter passed to the function. If this argument is not used inside function `func()`, NULL can be passed to this parameter.

Description

This function adds a set of data using a function `func()` with *nx* points in the range from *x0* to *xf* for *x*

and with `ny` points in the range from `y0` to `yf` for `y` to a previously declared instance of the **CPlot** class. The function has an optional parameter.

Example 1

```

/* File: funcp3D.ch */
#include<math.h>
#include<chplot.h>

#define NX 50
#define NY 50

double func(double x, double y, void *param) {
    double offset, z;

    offset = *(double*)param;
    z = cos(x)*sin(y) +offset;
    return z;
}

int main() {
    double x0, xf, y0, yf, offset;
    CPlot plot;

    x0 = 0;
    xf = 2*M_PI;
    y0 = 0;
    yf = 2*M_PI;
    offset = 1;
    plot.title("cos(x) sin(y)+offset");
    plot.funcp3D(x0, xf, y0, yf, NX, NY, func, &offset);
    offset = 2;
    plot.funcp3D(x0, xf, y0, yf, NX, NY, func, &offset);
    plot.plotting();
}

```

Output

See **CPlot::func3D()**.

See Also

CPlot::func2D(), **CPlot::func3D()**, **CPlot::funcp2D()**, **CPlot::data2D()**, **CPlot::data3D()**, **CPlot::data3DCurve()**, **CPlot::data3DSurface()**, **fplotxy()**.

CPlot::getGrid

Synopsis in Ch and C++

```
#include <chplot.h>
```

```
void getGrid(int *flag, char *option);
```

Purpose

Get the information for the grid of a plot.

Return Value

None.

Parameters

flag The value for turning on or off or grids.

option The additional options for setting the grids.

Description

The member function **CPlot::grid()** can be called by

```
int flag = PLOT_ON;
char option[]="front", char *optionptr;
plot.grid(flag);
plot.grid(flag, option);
```

This **CPlot::getGrid()** gets the values of *flag* and *option* by the function call as follows.

```
plot.getGrid(&flag, optionptr);
```

See Also

CPlot::grid().

CPlot::getLabel

Synopsis

```
#include <chplot.h>
```

```
char * getLabel(int axis);
```

Purpose

Get the label for a plot axis.

Return Value

The label of the axis.

Parameters

axis The axis with its label to be obtained. Valid values are:

PLOT_AXIS_X Select the x axis only.

PLOT_AXIS_X2 Select the x2 axis only.

PLOT_AXIS_Y Select the y axis only.

PLOT_AXIS_Y2 Select the y2 axis only.

PLOT_AXIS_Z Select the z axis only.

Description

Get the label of a plot axis.

Example

```

/* File: getLabel.ch */
#include <math.h>
#include <chplot.h>

int main() {
    class CPlot plot;
    string_t xlabel, ylabel, zlabel, title;

    plot.label(PLOT_AXIS_X, "xlabel");
    xlabel = plot.getLabel(PLOT_AXIS_X);
    ylabel = plot.getLabel(PLOT_AXIS_Y);
    zlabel = plot.getLabel(PLOT_AXIS_Z);
    printf("xlabel = %s\n", xlabel);
    printf("ylabel = %s\n", ylabel);
    printf("zlabel = %s\n", zlabel);
    title = plot.getTitle();
    printf("title = %s\n", title);
    plot.title("New Title");
    title = plot.getTitle();
    printf("title = %s\n", title);
}

```

Output

```

xlabel = xlabel
ylabel = y
zlabel = z
title =
title = New Title

```

See Also

CPlot::label(), **CPlot::title()**, **CPlot::getTitle()**.

CPlot::getOutputType

Synopsis

```

#include <chplot.h>
plotinfo_t getOutputType();

```

Purpose

Get the output type for a plot.

Return Value

The output type in one of the following forms:

PLOT_OUTPUTTYPE_DISPLAY Display the plot on the screen.

PLOT_OUTPUTTYPE_STREAM Output the plot as a standard output stream.

PLOT_OUTPUTTYPE_FILE Output the plot to a file in one of a variety of formats.

Parameters None

Description

Get the output type of a plot.

Example

```
/* File: getOutputType.ch */
#include <math.h>
#include <chplot.h>

int main() {
    class CPlot plot;
    plotinfo_t outputtype;

    outputtype = plot.getOutputType();
    if(outputtype == PLOT_OUTPUTTYPE_DISPLAY)
        printf("output is displayed\n");
    else if(outputtype == PLOT_OUTPUTTYPE_STREAM)
        printf("output is stdout stream\n");
    else if(outputtype == PLOT_OUTPUTTYPE_FILE)
        printf("output is in a file\n");
    return 0;
}
```

Output

output is displayed

See Also

CPlot::outputType().

CPlot::getSubplot

Synopsis

#include <chplot.h>

class CPlot* getSubplot(int row, int col);

Purpose

Get a pointer to an element of a subplot.

Return Value

Returns a pointer to the specified element of the subplot.

Parameters

row The row number of the desired subplot element. Numbering starts with zero.

col The column number of the desired subplot element. Numbering starts with zero.

Description

After the creation of a subplot using CPlot::subplot(), this function can be used to get a pointer to an element of the subplot. This pointer can be used as a CPlot pointer normally would be. Addition of data sets

or selection of plotting options are done normally. However, each option only effects the subplot element currently pointed to.

Example

```

/* File: getSubplot.ch */
#include <chplot.h>
#include <math.h>

#define NVAR 4
#define POINTS 50
int main() {
    void derivs(double t, double y[], double dydt[]) {
        dydt[0] = -y[1];
        dydt[1]=y[0]-(1.0/t)*y[1];
        dydt[2]=y[1]-(2.0/t)*y[2];
        dydt[3]=y[2]-(3.0/t)*y[3];
    }

    double t0=1, tf=10, y0[NVAR];
    double t[POINTS], y1[NVAR][POINTS];
    int points;
    int datasetnum=0, line_type=4, line_width = 2;
    array double theta2[360], r2[360];
    array double x3[360], y3[360], z3[2][360];
    double theta4[36], z4[20], r4[720];
    double r, x5[30], y5[30], z5[900];
    double x6[40], y6[60], z6[2400];

    int i, j;
    class CPlot subplot, *spl;

    /* plot 1 */
    y0[0]=j0(t0);
    y0[1]=j1(t0);
    y0[2]=jn(2,t0);
    y0[3]=jn(3,t0);
    points = odesolve(t, y1, derivs, t0, tf, y0);

    /* plot 2 */
    lindata(0, M_PI, theta2);
    r2 = sin(5*theta2);

    /* plot 3 */
    lindata(0, 360, x3);
    y3 = sin(x3*M_PI/180);
    for(i=0; i<360; i++) {
        z3[0][i] = cos(x3[i]*M_PI/180);
        z3[1][i] = y3[i];
    }

    /* plot 4 */
    lindata(0, 360, theta4);
    lindata(0, 2*M_PI, z4);
    for(i=0; i<36; i++) {
        for(j=0; j<20; j++) {
            r4[i*20+j] = 2+cos(z4[j]);
        }
    }
}

```

```

    }
}

/* plot 5 */
lindata(-10, 10, x5);
lindata(-10, 10, y5);
for(i=0; i<30; i++) {
    for(j=0; j<30; j++) {
        r = sqrt(x5[i]*x5[i]+y5[j]*y5[j]);
        z5[30*i+j] = sin(r)/r;
    }
}

/* plot 6 */
lindata(-3, 3, x6);
lindata(-4, 4, y6);
for(i=0; i<40; i++) {
    for(j=0; j<60; j++) {
        z6[60*i+j] = 3*(1-x6[i])*(1-x6[i])*
            exp(-(x6[i]*x6[i])-(y6[j]+1)*(y6[j]+1))
        - 10*(x6[i]/5 - x6[i]*x6[i]*x6[i]-pow(y6[j],5))*
            exp(-x6[i]*x6[i]-y6[j]*y6[j])
        - 1/3*exp(-(x6[i]+1)*(x6[i]+1)-y6[j]*y6[j]);
    }
}

subplot.subplot(2,3); /* create 2 x 3 subplot */
spl = subplot.getSubplot(0,0); /* get subplot (0,0) */
spl->title("Line");
spl->label(PLOT_AXIS_X,"t");
spl->label(PLOT_AXIS_Y,"Bessel functions");
spl->data2D(t, y1);
spl->legend("j0", 0);
spl->legend("j1", 1);
spl->legend("j2", 2);
spl->legend("j3", 3);

spl = subplot.getSubplot(0,1); /* get subplot (0,1) */
spl->title("Polar");
spl->axisRange(PLOT_AXIS_XY, -1, 1);
spl->ticsRange(PLOT_AXIS_XY, .5, -1, 1);
spl->data2D(theta2, r2);
spl->polarPlot(PLOT_ANGLE_RAD);
spl->sizeRatio(-1);

spl = subplot.getSubplot(0,2); /* get subplot (0,2) */
spl->title("3D curve");
spl->data3D(x3, y3, z3);
spl->colorBox(PLOT_OFF);
spl->axisRange(PLOT_AXIS_X, 0, 400);
spl->ticsRange(PLOT_AXIS_X, 200, 0, 400);
spl->axisRange(PLOT_AXIS_Y, -1, 1);
spl->ticsRange(PLOT_AXIS_Y, 1, -1, 1);
spl->axisRange(PLOT_AXIS_Z, -1, 1);
spl->ticsRange(PLOT_AXIS_Z, 1, -1, 1);

spl = subplot.getSubplot(1,0); /* get subplot (1,0) */
spl->title("Cylindrical");
spl->data3D(theta4, z4, r4);

```

```

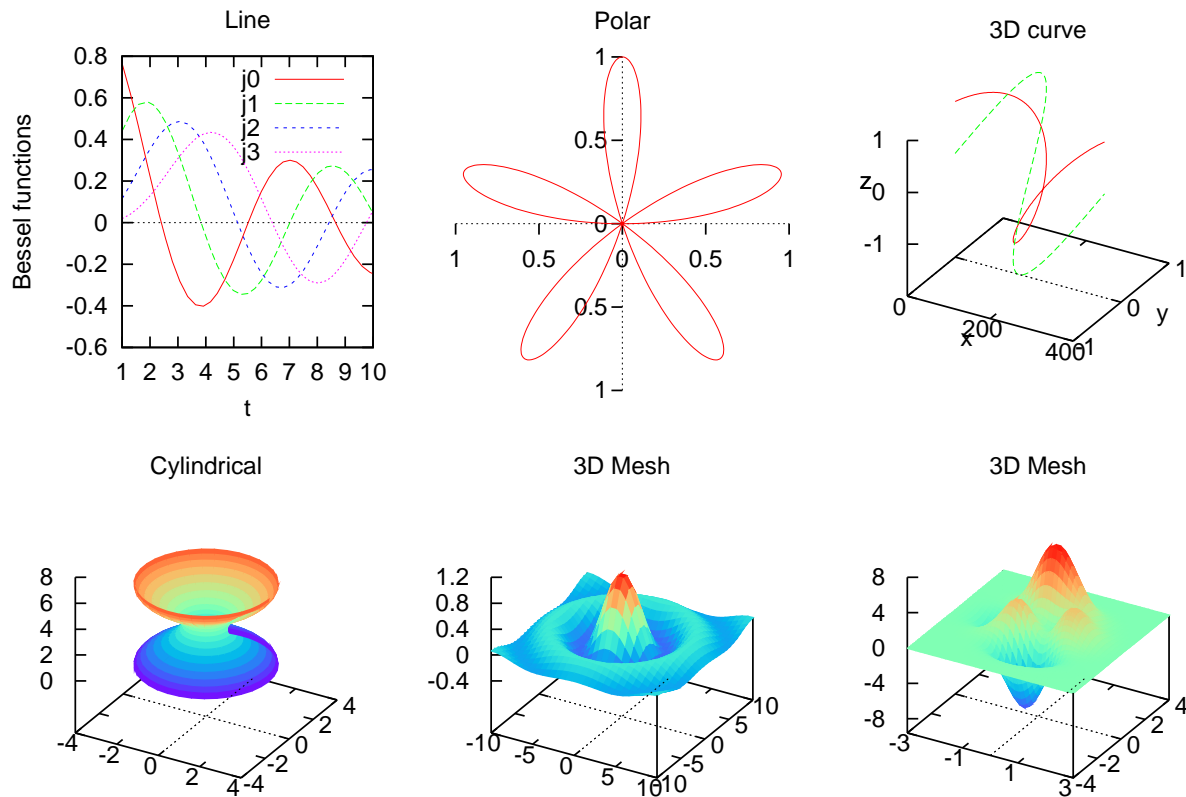
spl->colorBox(PLOT_OFF);
spl->coordSystem(PLOT_COORD_CYLINDRICAL, PLOT_ANGLE_DEG);
spl->axisRange(PLOT_AXIS_Z, 0, 8);
spl->ticsRange(PLOT_AXIS_Z, 2, 0, 8);
spl->axisRange(PLOT_AXIS_XY, -4, 4);
spl->ticsRange(PLOT_AXIS_XY, 2, -4, 4);
spl->label(PLOT_AXIS_XYZ, NULL);

spl = subplot.getSubplot(1,1); /* get subplot (1,1) */
spl->title("3D Mesh");
spl->axisRange(PLOT_AXIS_X, -10, 10);
spl->ticsRange(PLOT_AXIS_X, 5, -10, 10);
spl->axisRange(PLOT_AXIS_Y, -10, 10);
spl->ticsRange(PLOT_AXIS_Y, 5, -10, 10);
spl->axisRange(PLOT_AXIS_Z, -.4, 1.2);
spl->ticsRange(PLOT_AXIS_Z, .4, -.4, 1.2);
spl->data3D(x5, y5, z5);
spl->colorBox(PLOT_OFF);
spl->label(PLOT_AXIS_XYZ, NULL);

spl = subplot.getSubplot(1,2); /* get subplot (1,2) */
spl->title("3D Mesh");
spl->data3D(x6, y6, z6);
spl->axisRange(PLOT_AXIS_X, -3, 3);
spl->ticsRange(PLOT_AXIS_X, 2, -3, 3);
spl->axisRange(PLOT_AXIS_Y, -4, 4);
spl->ticsRange(PLOT_AXIS_Y, 2, -4, 4);
spl->axisRange(PLOT_AXIS_Z, -8, 8);
spl->ticsRange(PLOT_AXIS_Z, 4, -8, 8);
spl->colorBox(PLOT_OFF);
spl->label(PLOT_AXIS_XYZ, NULL);
spl->ticsLevel(0.1);
subplot.plotting();
}

```

Output



See Also
CPlot::subplot().

CPlot::getTitle

Synopsis

```
#include <chplot.h>
string_t getTitle(void);
```

Purpose

Get the plot title.

Return Value

The plot title.

Parameters

None.

Description

Get the title of the plot. If no title, NULL will be returned.

Example

see `CPlot::getLabel()`.

See Also

`CPlot::label()`, `CPlot::getLabel()`, `CPlot::title()`.

CPlot::grid

Synopsis

```
#include <chplot.h>
void grid(int flag, ... /* [char * option] */);
```

Syntax

```
grid(flag)
grid(flag, option)
```

Purpose

Enable or disable the display of a grid on the xy plane.

Return Value

None.

Parameters

flag This parameter can be set to:

- PLOT_ON** Enable the display of the grid.
- PLOT_OFF** Disable the display of the grid.

option The option for the grid.

Description

Enable or disable the display of a grid on the xy plane. By default, the grid is off. For a polar plot, the polar grid is displayed. Otherwise, the grid is rectangular.

The optional argument *option* of string type with the following values can be used to fine tune the grid based on the argument for set grid command of the gnuplot.

```

{{no}{m}xtics} {{no}{m}ytics} {{no}{m}ztics}
{{no}{m}x2tics} {{no}{m}y2tics}
{{no}{m}cbtics}
{polar {<angle>}}
{layerdefault | front | back}
{ {linestyle <major_linestyle>
  | {linetype | lt <major_linetype>}
  {linewidth | lw <major_linewidth>}
  { , {linestyle | ls <minor_linestyle>
    | {linetype | lt <minor_linetype>}
    {linewidth | lw <minor_linewidth>} } } }
```

The grid can be enabled and disabled for the major and/or minor tic marks on any axis, and the linetype and linewidth can be specified for major and minor grid lines.

Additionally, a polar grid can be selected for 2-d plots—circles are drawn to intersect the selected tics, and radial lines are drawn at definable intervals. The interval is given in degrees or radians, depending on the argument of **CPlot::polarPlot()**. The default polar angle is 30 degrees.

The pertinent tics must be enabled. Otherwise, the plotting engine will quietly ignore instructions to draw grid lines at non-existent tics, but they will appear if the tics are subsequently enabled.

The 'linetype' is followed by an integer index representing the line type for drawing. The line type varies depending on the terminal type used (see **CPlot::outputType**). Typically, changing the line type will change the color of the line or make it dashed or dotted. All terminals support at least six different line types. The 'linewidth' is followed by a scaling factor for the line width. The line width is 'linewidth' multiplied by the default width. Typically the default width is one pixel.

If no linetype is specified for the minor gridlines, the same linetype as the major gridlines is used.

If "front" is given, the grid is drawn on top of the graphed data. If "back" is given, the grid is drawn underneath the graphed data. Using "front" will prevent the grid from being obscured by dense data. The default setup, "layerdefault", is equivalent to "back" for 2d plots. In 3D plots the default is to split up the grid and the graph box into two layers: one behind, the other in front of the plotted data and functions.

For 3D plot, Z grid lines are drawn on the bottom of the plot.

Example 1

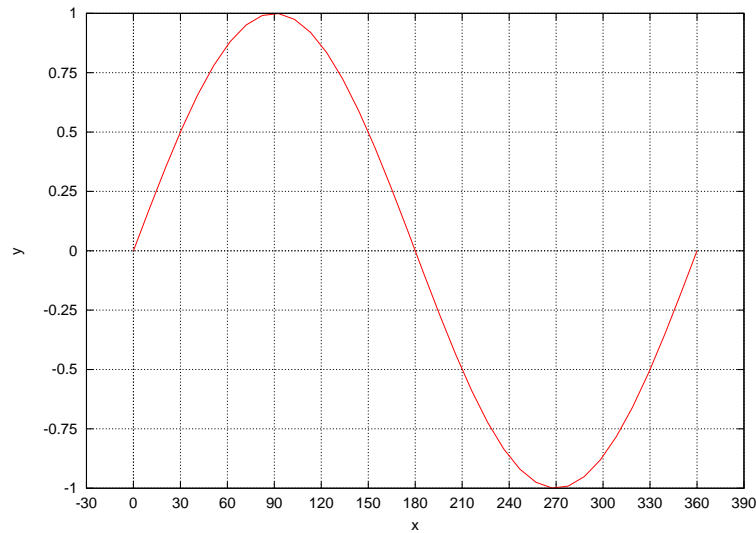
Compare with the output for the example in **CPlot::axisRange()**.

```
/* File: grid.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    int i;
    class CPlot plot;

    lindata(0, 360, x);
    y = sin(x*M_PI/180);           // Y-axis data.
    plot.axisRange(PLOT_AXIS_X, -30, 390);
    plot.ticsRange(PLOT_AXIS_X, 30, -30, 390);
    plot.axisRange(PLOT_AXIS_Y, -1, 1);
    plot.ticsRange(PLOT_AXIS_Y, .25, -1, 1);
    plot.grid(PLOT_ON);
    plot.data2D(x, y);
    plot.plotting();
}
```

Output

**Example 2**

Compare with the output for examples in **CPlot::data3D()** and **CPlot::data3DSurface()**.

```

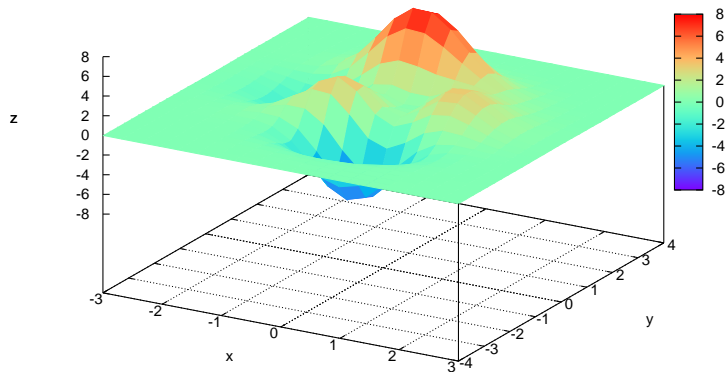
/* File: grid_2.ch */
#include <math.h>
#include <chplot.h>

int main() {
    double x[20], y[30], z[600];
    int i, j;
    class CPlot plot;

    lndata(-3, 3, x);
    lndata(-4, 4, y);
    for(i=0; i<20; i++) {
        for(j=0; j<30; j++) {
            z[30*i+j] = 3*(1-x[i])*(1-x[i])*exp(-(x[i]*x[i])-(y[j]+1)*(y[j]+1))
                - 10*(x[i]/5 - x[i]*x[i]*x[i]-pow(y[j],5))*exp(-x[i]*x[i]-y[j]*y[j])
                - 1/3*exp(-(x[i]+1)*(x[i]+1)-y[j]*y[j]));
        }
    }
    plot.data3D(x, y, z);
    plot.grid(PLOT_ON);
    plot.plotting();
}

```

Output

**Example 3**

An example with grids for both y and y2 axes on page 29 for **CPlot::axes()**.

Example 4

An example with grids on the front for a filled curve on page 215 for **CPlot::plotType()**.

CPlot::histogram

Synopsis

```
#include <chplot.h>
```

```
void histogram(double x[], double begin, double end, double interval, ... /* [int n] */);
```

Syntax

```
histogram(x, begin, end, interval);
```

```
histogram(x, begin, end, interval, n);
```

Purpose

Draw a histogram.

Return Value

None.

Parameters

x An array containing the values for the histogram.

begin The beginning of the class intervals.

end The end of the class intervals.

interval The interval for the histogram.

n The number of elements in array *x*.

Description

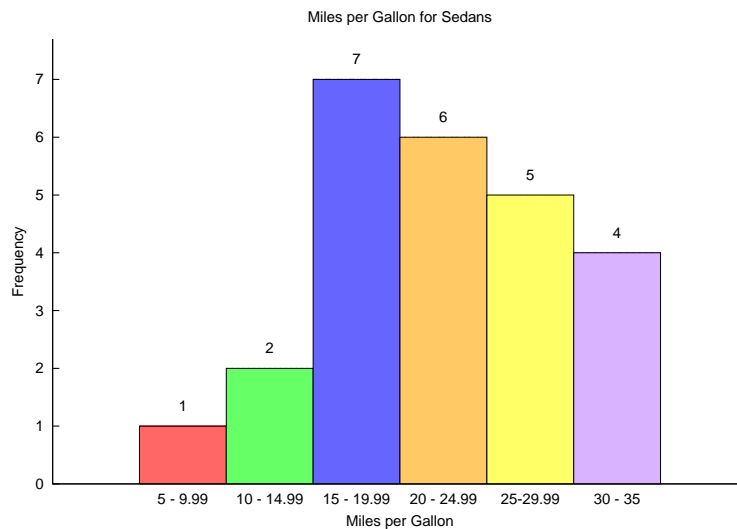
This member function draws a histogram using the data in the array *x*. It darws a series of vertical rectangles,


```

plot.grid(PLOT_OFF);
plot.axisRange(PLOT_AXIS_X, 0, 40);
plot.ticsLabel(PLOT_AXIS_X, "5 - 9.99", 7.5);
plot.ticsLabel(PLOT_AXIS_X, "10 - 14.99", 12.5);
plot.ticsLabel(PLOT_AXIS_X, "15 - 19.99", 17.5);
plot.ticsLabel(PLOT_AXIS_X, "20 - 24.99", 22.5);
plot.ticsLabel(PLOT_AXIS_X, "25-29.99", 27.5);
plot.ticsLabel(PLOT_AXIS_X, "30 - 35", 32.5);
plot.plotting();

```

Output



See Also

CPlot::barGraph(), CPlot::histogramColor(), CPlot::histogramValue().

CPlot::histogramColor

Synopsis

```
#include <chplot.h>
```

```
void histogramColor(string_t color1[:], ... /* string_t color2);
```

Syntax

```
histogramColor(color1);
```

```
histogramColor(color1, color2);
```

```
histogramColor(color1, color2, color3);
```

Purpose

Specify the color for each interval for a histogram.

Return Value

None.

Parameters

color1 The color for the first interval of the histogram.

color2 The color for the second interval of the histogram.

color3 The color for the third interval of the histogram.

Description

This member function **plot.histogram()** draws a histogram using the data in an array. The member function **plot.histogramColor()** is used to specify the color for each rectangle on the histogram.

Example

See **CPlot::histogram()**.

See Also

CPlot::barGraph(), **CPlot::histogram()**, **CPlot::histogramValue()**.

CPlot::histogramValue

Synopsis

```
#include <chplot.h>
void histogramValue();
```

Purpose

Add a value for each interval on a histogram.

Return Value

None.

Parameters

None.

Description

The member function **plot.histogram()** draws a histogram using the data in an array. By default, horizontal lines are drawn for each ticks on the y-axis. The member function **plot.histogramValue()** can be used to suppress the horizontal lines and place a value on the top of each rectangle to represent the number of items in that interval.

Example

See **CPlot::histogram()**.

See Also

CPlot::barGraph(), **CPlot::histogram()**, **CPlot::histogramColor()**.

CPlot::inequal

Synopsis

```
#include <chplot.h>
```

```
void inequal(string_t relop, double value, ... /* string_t color */);
```

Syntax

```
inequal(relop, value);
```

```
inequal(relop, value, "blue");
```

Purpose

Draw an inequality on a number line.

Return Value

None.

Parameters

relop The relational operator for the inequality.

value The value for the relational operation.

color The color of the inequality.

Description

This function draws an inequality on a number line. This member function is used to visualize inequalities with relational operations as listed in the table below.

Operator	Operation	Math	Description
"<"	$x < \text{value}$	$<$	less than
"<="	$x \leq \text{value}$	\leq	less than or equal to
"=="	$x == \text{value}$	$=$	equal to
"!="	$x \neq \text{value}$	\neq	not equal to
">="	$x \geq \text{value}$	\geq	greater than or equal to
">"	$x > \text{value}$	$>$	greater than

Before this member function is called, the member function **plot.axisRange()** must be called first to set the range of the x-axis on the number line.

```
plot.axisRange(PLOT_AXIS_X, xmin, xmax);
```

If the relational operator is "<" or "<=", an arrow from the *value* toward *xmin* (negative infinity) will be drawn. If the relational operator is ">" or ">=", an arrow from the *value* toward *xmax* (positive infinity) will be drawn.

If the relational operator is "<=", "==", or ">=", a closed circle is drawn at the *value*. If the relational operator is "<" or "!=" , or ">" , an open circle is drawn at the *value*.

Unless the member function **plot.inequalOffset()**, when this member function is called multiple times, each inequality is drawn on the same number line. If the member function **plot.inequalOffset()** is called, each inequality has an offset of the value 1 vertically from the previous inequality.

Other plotting member functions such as **plot.label()**, **plot.ticsRange()**, and **plot.strokeColor()** can be called to set some attributes of the plot.

Example 1

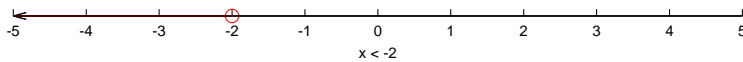
```

/* File: inequal.ch
   Inequality  $x < -2$  */
#include <chplot.h>
CPlot plot;

plot.label(PLOT_AXIS_X, "x < -2");
plot.axisRange(PLOT_AXIS_X, -5, 5);
plot.inequal("<", -2);
plot.plotting();

```

Output



Example 2

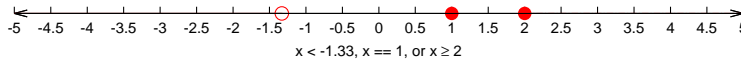
```

/* File: inequal2.ch
   Inequality  $x < -1.33$ ,  $x = 1$ , or  $x \geq 2$  */
#include <chplot.h>
CPlot plot;

plot.label(PLOT_AXIS_X, "x < -1.33, x = 1, or x >= 2");
plot.axisRange(PLOT_AXIS_X, -5, 5);
plot.ticsRange(PLOT_AXIS_X, 0.5);
plot.strokeColor("red");
plot.inequal("<", -1.33);
plot.inequal("=", 1);
plot.inequal(">=", 2);
plot.plotting();

```

Output

**See Also**

CPlot::inequalAnd(), CPlot::inequalOffset(), CPlot::line1D(), CPlot::line1DOffset().

CPlot::inequalAnd

Synopsis

```
#include <chplot.h>
```

```
void inequalAnd(string_t relop1, double value2, ... /* string_t color */);
```

Syntax

```
inequalAnd(relop1, value1, relop2, value2);
```

```
inequalAnd(relop1, value1, relop2, value2, "blue");
```

Purpose

Draw the result of an AND relation operation for two inequalities on a number line.

Return Value

None.

Parameters

relop1 The relational operator for the first inequality.

value1 The value for the first relational operation.

relop2 The relational operator for the second inequality.

value2 The value for the second relational operation.

color The color of the inequalities.

Description

This function draws the result of an AND relation operation for two inequalities x *relop1* *value1* and x *relop2* *value2* on the number line.

This member function is used to visualize the result of the AND operation or conjunction of two inequalities. The valid relational operations are listed in the member function `plot.inequal()`.

Before this member function is called, the member function **plot.axisRange()** must be called first to set the range of the x-axis on the number line.

Unless the member function **plot.inequalAndOffset()**, when this member function is called multiple times, the result is drawn on the same number line. If the member function **plot.inequalAndOffset()** is called, each result has an offset of the value 1 vertically from the previous **inequalAnd**ity.

Other plotting member functions such as **plot.label()**, **plot.ticsRange()**, and **plot.strokeColor()** can be called to set some attributes of the plot.

Example 1

```
/* File: inequalAnd.ch
   Inequality x >= -1.33 AND x <= 2 */
#include <chplot.h>
CPlot plot;

plot.axisRange(PLOT_AXIS_X, -5, 5);
plot.inequalOffset();

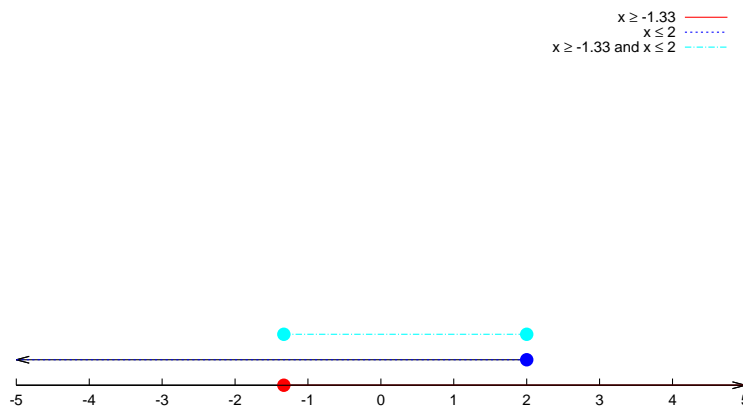
plot.legend("x >= -1.33");
plot.inequal(">=", -1.33);

plot.legend("x <= 2");
plot.inequal("<=", 2);

plot.legend("x >= -1.33 and x <= 2");
plot.inequalAnd(">=", -1.33, "<=", 2);

plot.plotting();
```

Output



Example 2

```
/* File: inequalAnd2.ch
   Inequality x < -1 AND x <= 2 */
#include <chplot.h>
CPlot plot;

plot.axisRange(PLOT_AXIS_X, -5, 5);
plot.inequalOffset();

plot.legend("x < -1");
```



```

plot.inequal("<", -1);

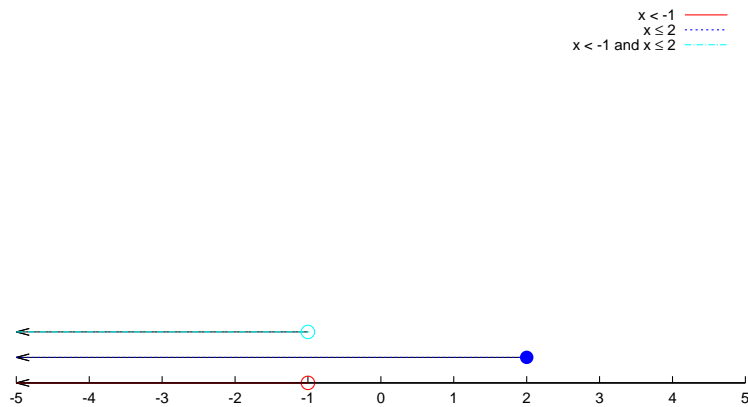
plot.legend("x <= 2");
plot.inequal("<=", 2);

plot.legend("x < -1 and x <= 2");
plot.inequalAnd("<", -1, "<=", 2);

plot.plotting();

```

Output



Example 3

```

/* File: inequalAnd3.ch
   Inequality  $x \leq 2$  AND  $x \geq 2$  */
#include <chplot.h>
CPlot plot;

plot.axisRange(PLOT_AXIS_X, -5, 5);
plot.inequalOffset();

plot.legend("x <= 2");
plot.inequal("<=", 2);

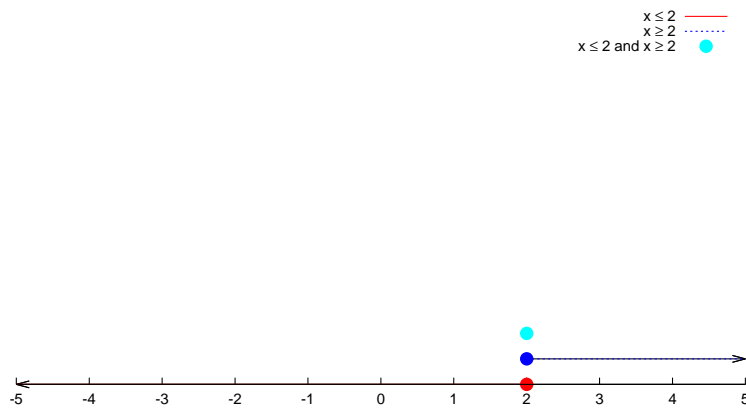
plot.legend("x >= 2");
plot.inequal(">=", 2);

plot.legend("x <= 2 and x >= 2");
plot.inequalAnd("<=", 2, ">=", 2);

plot.plotting();

```

Output

**Example 4**

```

/* File: inequalAnd4.ch
   Inequality  $x \leq 1$  AND  $x \geq 2$ . There is no solution */
#include <chplot.h>
CPlot plot;

plot.axisRange(PLOT_AXIS_X, -5, 5);
plot.inequalOffset();

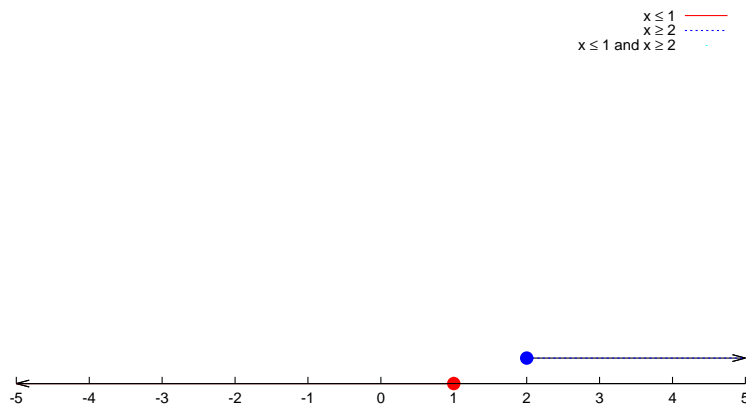
plot.legend("x <= 1");
plot.inequal("<=", 1);

plot.legend("x >= 2");
plot.inequal(">=", 2);

plot.legend("x <= 1 and x >= 2");
plot.inequalAnd("<=", 1, ">=", 2);

plot.plotting();

```

Output**Example 5**

```

/* File: inequalAnd5.ch
   Inequality  $x == 3$  AND  $x \geq 2$ . */
#include <chplot.h>

```

```

CPlot plot;

plot.axisRange(PLOT_AXIS_X, -5, 5);
plot.inequalOffset();

plot.legend("x == 3");
plot.inequal("==", 3);

plot.legend("x != 4");
plot.inequal("!=", 4);

plot.legend("x >= 2");
plot.inequal(">=", 2);

plot.legend("x == 3 and x >= 2");
plot.inequalAnd("==", 3, ">=", 2);

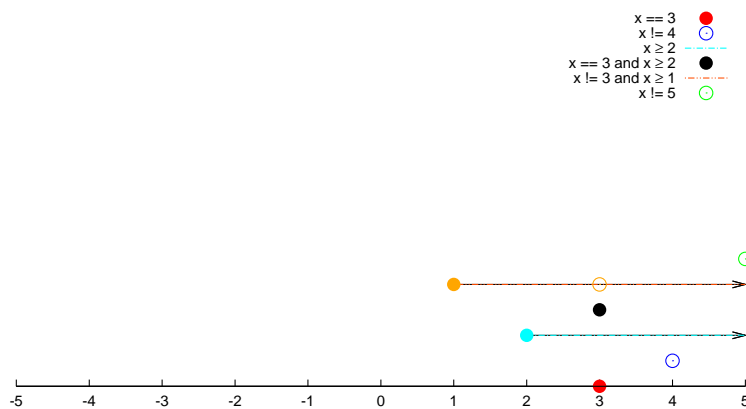
plot.legend("x != 3 and x >= 1");
plot.inequalAnd("!=", 3, ">=", 1);

plot.legend("x != 5");
plot.inequal("!=", 5);

plot.plotting();

```

Output



See Also

CPlot::inequal(), CPlot::inequalOffset(), CPlot::line1D(), CPlot::line1DOffset().

CPlot::inequalOffset

Synopsis

```

#include <chplot.h>
void inequalOffset();

```

Purpose

Set the vertical offset for drawing inequalities in a one-dimensional plot on a number line.

Return Value

None.

Parameters

None.

Description

The members **plot.inequal()** and **plot.inequalAnd()** draw inequalities on a number line as a one-dimensional plot. By default, each inequality is drawn on the same number line. If this member function is called, each inequality has an offset of the value 1 vertically from the previous inequality from the number line.

Example

See **CPlot::inequalAnd()**.

See Also

CPlot::inequal(), **CPlot::inequalAnd()**.

CPlot::isUsed

Synopsis

```
#include <chplot.h>
int isUsed();
```

Purpose

Test if an instance of the **CPlot** class has been used.

Return Value

Returns `true` if the **CPlot** instance has been previously used, returns `false` otherwise.

Parameters

None.

Description

This function determines if an instance of the **CPlot** class has previously been used. The function actually tests if data have previously been added to the instance, by checking if an internal pointer in the **CPlot** class is `NULL`. If the pointer is not `NULL`, then the instance has been used. This function is used internally by **fplotxy()**, **fplotxyz()**, **plotxy()**, **plotxyz()**, **plotxyf()**, and **plotxyzf()**.

See Also

fplotxy(), **fplotxyz()**, **plotxy()**, **plotxyz()**, **plotxyf()**, and **plotxyzf()**.

CPlot::label

Synopsis

```
#include <chplot.h>
void label(int axis, string_t label);
```

Purpose

Set the labels for the plot axes.

Return Value

None.

Parameters

axis The axis to be set. Valid values are:

PLOT_AXIS_X Select the x axis only.

PLOT_AXIS_X2 Select the x2 axis only.

PLOT_AXIS_Y Select the y axis only.

PLOT_AXIS_Y2 Select the y2 axis only.

PLOT_AXIS_Z Select the z axis only.

PLOT_AXIS_XY Select the x and y axes.

PLOT_AXIS_XYZ Select the x, y, and z axes.

label label of the axis.

Description

Set the plot axis labels. The label for the z-axis can be set only for a 3D plot. If no label is desired, the value of `NULL` can be used as an argument. For example, function call `plot.label(PLOT_AXIS_XY, NULL)` will suppress the labels for both x and y axes of the plot. By default, labels are “x”, “y”, and “z” for the first x, y, and z axes, respectively.

Example

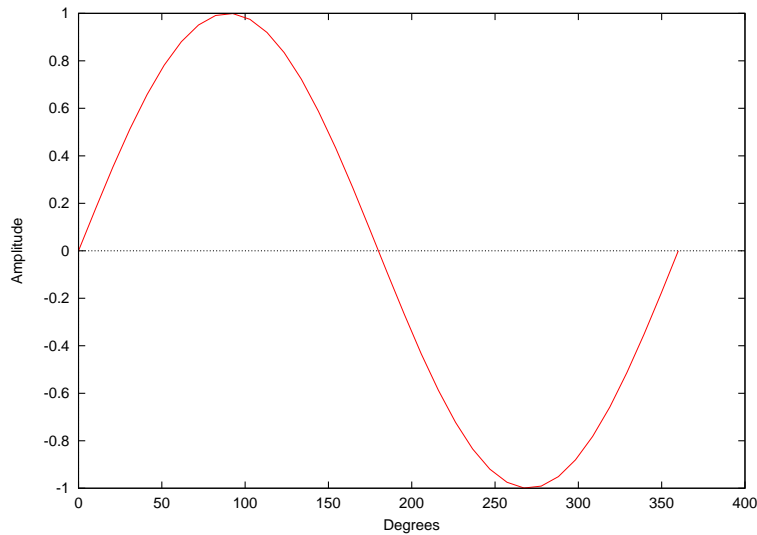
Compare with the output for examples in `CPlot::data2D()` and `CPlot::data2DCurve()`.

```
/* File: label.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    string_t xlabel="Degrees",
            ylabel="Amplitude";
    class CPlot plot;

    lndata(0, 360, x);
    y = sin(x*M_PI/180);
    plot.label(PLOT_AXIS_X, xlabel);
    plot.label(PLOT_AXIS_Y, ylabel);
    plot.data2D(x, y);
    plot.plotting();
}
```

Output

**See Also**

CPlot::getLabel(), **CPlot::title()**.

CPlot::getTitle().

CPlot::legend

Synopsis

```
#include <chplot.h>
```

```
void legend(string_t legend, ... /* int num | string_t legend2 ... */);
```

Syntax

```
legend(legend);
```

```
legend(legend, num);
```

```
legend(legend, legend2);
```

```
legend(legend, legend2, legend3);
```

Purpose

Specify a legend string for a data set. Specify a legend string for a previously added data set. Specify legend strings for data sets.

Return Value

None.

Parameters

legend The legend string.

num The data set the legend is added to.

legend2 The legend string.

legend3 The legend string.

Description

The legend string is added to a plot legend located in the upper-right corner of the plot by default. If the member function is called without the argument *num*, a legend will associate with the corresponding data set in the sequence it was added to the plot. New legends will replace previously specified legends. If the argument *legend* is NULL or "", no legend will be displayed for the curve. In this case, when the next time this function is called, the legend will be associated with the next set of data.

If the argument *num* is used to specify the data set, numbering of the data sets starts with zero. The member function with the argument *num* shall be called after plotting data have been added by member functions **CPlot::data2D()**, **CPlot::data2DCurve()**, **CPlot::data3D()**, **CPlot::data3DCurve()**, **CPlot::data3DSurface()**, **CPlot::dataFile()**, **CPlot::point()**, **CPlot::scatter()**, **CPlot::scattern()**.

Bugs

The legend is not available for objects including the arc sector, arc segment, circle, ellipse, rectangle, quadrilateral, triangle, and polygon.

Examples 1-3

Compare with the output for the example in **CPlot::legendLocation()**.

```

/* File: legend.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints], y2[numpoints];
    class CPlot plot;

    lndata(0, 360, x);
    y = sin(x*M_PI/180);
    y2= cos(x*M_PI/180);
    plot.data2D(x, y);
    plot.legend("sin(x)");
    plot.data2D(x, y2);
    plot.legend("cos(x)");
    /* place legned in location (370, 0.6) inside a box */
    plot.legendOption("at 370, 0.6 box");
    plot.plotting();
}

/* File: legend2.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints], y2[numpoints];
    class CPlot plot;

    lndata(0, 360, x);
    y = sin(x*M_PI/180);
    y2= cos(x*M_PI/180);
    plot.data2D(x, y);
    plot.legend("sin(x)", 0);
    plot.data2D(x, y2);
    plot.legend("cos(x)", 1);
    /* place legned in location (370, 0.6) inside a box */

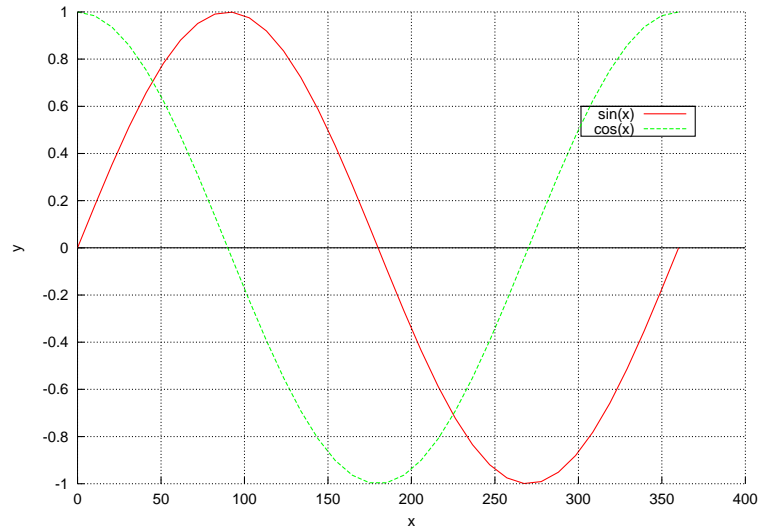
```

```

    plot.legendOption("at 370, 0.6 box");
    plot.plotting();
}

```

Output: Programs `legend.ch`, `legend2.ch`, and `legend3.ch` will produce the same output.



Example 4

```

/* File: legend4.ch
   Plot the rational function y=-1/(x-5)
   for x from 0 to 4.8 and 5.2 to 10 with 500 points. */
#include <chplot.h>
CPlot plot;

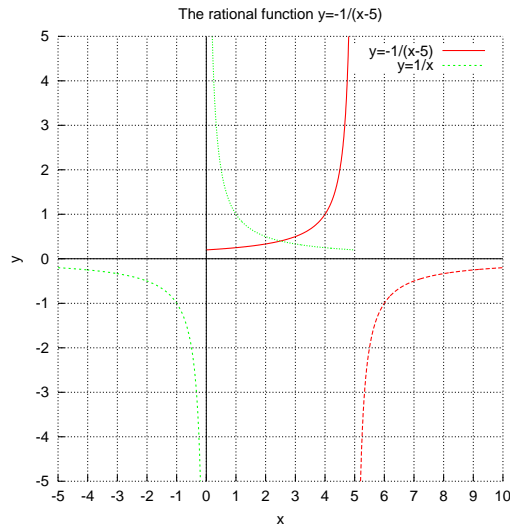
plot.title("The rational function y=-1/(x-5)");
plot.label(PLOT_AXIS_X, "x");
plot.label(PLOT_AXIS_Y, "y");
plot.ticsRange(PLOT_AXIS_X, 1);
plot.ticsRange(PLOT_AXIS_Y, 1);
plot.sizeRatio(1);

plot.strokeColor("red");
plot.expr(0, 4.8, 500, "-1/(x-5)");
plot.legend("y=-1/(x-5)");
plot.expr(5.2, 10, 500, "-1/(x-5)");
plot.legend(""); // SKIP legend

/* plot y=1/x for x from -5 to -0.2 and 0.2 to 5 */
plot.strokeColor("green");
plot.expr(-5, -0.2, 500, "1/x");
plot.legend("y=1/x");
plot.expr(0.2, 5, 500, "1/x");

plot.plotting();

```


**See Also**

CPlot::data2D(), CPlot::data2DCurve(), CPlot::data3D(), CPlot::data3DCurve(),
 CPlot::data3DSurface(), CPlot::dataFile(), CPlot::legendLocation(), CPlot::legendOption().

CPlot::legendLocation

Synopsis

```
#include <chplot.h>
void legendLocation(double x, double y, ... /* [double z] */);
```

Syntax

```
legendLocation(x, y)
legendLocation(x, y, z)
```

Purpose

Specify the plot legend (if any) location

Return Value

None.

Parameters

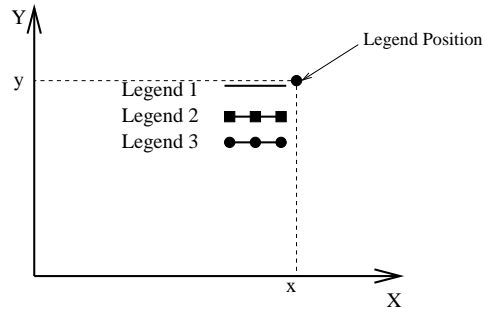
x The x coordinate of the legend.

y The y coordinate of the legend.

z The z coordinate of the legend.

Description

This function specifies the position of the plot legend using plot coordinates. The position specified is the location of the top right of the box for the markers and labels of the legend, as shown below. By default, the location of the legend is near the upper-right corner of the plot. For a two-dimensional plot, the third argument is not necessary.

**Example**

Compare with the output for the example in `CPlot::legend()`.

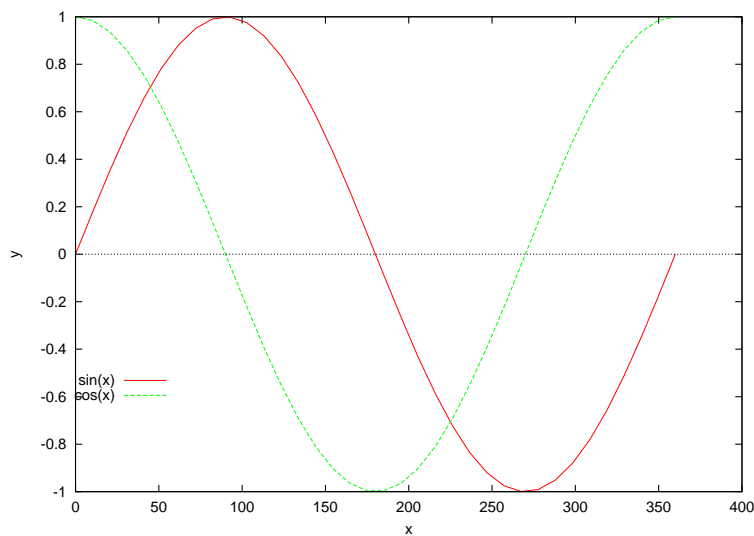
```

/* File: legendLocation.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints], y2[numpoints];
    class CPlot plot;

    lindata(0, 360, x);
    y = sin(x*M_PI/180);
    y2= cos(x*M_PI/180);
    plot.data2D(x, y);
    plot.data2D(x, y2);
    plot.legend("sin(x)", 0);
    plot.legend("cos(x)", 1);
    plot.legendLocation(60, -.5);
    plot.plotting();
}

```

Output**See Also**

`CPlot::legend()`, `CPlot::legendOption()`.

CPlot::legendOption

Synopsis

```
#include <chplot.h>
void legendOption(char * option);
```

Syntax

```
legendOption(option)
```

Purpose

Set options for legends of a plot.

Return Value

None.

Parameters

option The options for legends of a plot.

Description

The optional argument *option* of string type with the following values can be used to fine tune the legends of a plot.

```
{{inside | outside} | {lmargin | rmargin | tmargin | bmargin}
 | {at <position>}}
{left | right | center} {top | bottom | center}
{vertical | horizontal} {Left | Right}
{{no}reverse} {{no}invert}
{samplen <sample_length>} {spacing <vertical_spacing>}
{width <width_increment>}
{height <height_increment>}
{{no}autotitle {columnheader}}
{{no}box { {linestyle | ls <line_style>
          | {linetype | lt <line_type>}
          {linewidth | lw <line_width>}}}}
```

Legends are stacked according to ‘vertical’ or ‘horizontal’. In the case of ‘vertical’, the legends occupy as few columns as possible. That is, legends are aligned in a column until running out of vertical space at which point a new column is started. In the case of ‘horizontal’, the legends occupy as few rows as possible.

By default the legends are placed in the upper right inside corner of the graph. The keywords ‘left’, ‘right’, ‘top’, ‘bottom’, ‘center’, ‘inside’, ‘outside’, ‘lmargin’, ‘rmargin’, ‘tmargin’, ‘bmargin’, ‘above’, ‘over’, ‘below’ and ‘under’) may be used to automatically place the legends in other positions of the graph. Also an ‘at <position>’ may be given to indicate precisely where the plot should be placed. In this case, the keywords ‘left’, ‘right’, ‘top’, ‘bottom’ and ‘center’ serve an analogous purpose for alignment.

To understand positioning, the best concept is to think of a region, i.e., inside/outside, or one of the margins. Along with the region, keywords ‘left/center/right’ (l/c/r) and ‘top/center/bottom’ (t/c/b) control where within the particular region the legends should be placed.

When in ‘inside’ mode, the keywords ‘left’ (l), ‘right’ (r), ‘top’ (t), ‘bottom’ (b), and ‘center’ (c) push the legends out toward the plot boundary as illustrated:

```
t/l   t/c   t/r
c/l   c     c/r
b/l   b/c   b/r
```

When in ‘outside’ mode, automatic placement is similar to the above illustration, but with respect to the view, rather than the graph boundary. That is, a border is moved inward to make room for the key outside of the plotting area, although this may interfere with other labels and may cause an error on some devices. The particular plot border that is moved depends upon the position described above and the stacking direction. For options centered in one of the dimensions, there is no ambiguity about which border to move. For the corners, when the stack direction is ‘vertical’, the left or right border is moved inward appropriately. When the stack direction is ‘horizontal’, the top or bottom border is moved inward appropriately.

The margin syntax allows automatic placement of legends regardless of stack direction. When one of the margins ‘lmargin’ (lm), ‘rmargin’ (rm), ‘tmargin’ (tm), and ‘bmargin’ (bm) is combined with a single, non-conflicting direction keyword, the following illustrated positions may contain the legends:

```
l/tm  c/tm  r/tm
t/lm                      t/rm
c/lm                      c/rm
b/lm                      b/rm
l/bm  c/bm  r/bm
```

Keywords ‘above’ and ‘over’ are synonymous with ‘tmargin’. For version compatibility, ‘above’ or ‘over’ without an additional l/c/r or stack direction keyword uses ‘center’ and ‘horizontal’. Keywords ‘below’ and ‘under’ are synonymous with ‘bmargin’. For compatibility, ‘below’ or ‘under’ without an additional l/c/r or stack direction keyword uses ‘center’ and ‘horizontal’. A further compatibility issue is that ‘outside’ appearing without an additional t/b/c or stack direction keyword uses ‘top’, ‘right’ and ‘vertical’ (i.e., the same as t/rm above).

The <position> can be a simple x,y,z as in previous versions, but these can be preceded by one of five keywords (‘first’, ‘second’, ‘graph’, ‘screen’, ‘character’) which selects the coordinate system in which the position of the first sample line is specified. See ‘coordinates’ for more details. The effect of ‘left’, ‘right’, ‘top’, ‘bottom’, and ‘center’ when <position> is given is to align the legends as though it were text positioned using the label command, i.e., ‘left’ means left align with key to the right of jposition, etc.

Justification of the labels within the key is controlled by ‘Left’ or ‘Right’ (default is ‘Right’). The text and sample can be reversed (‘reverse’) and a box can be drawn around the legend (‘box { . . . }’) in a specified ‘linetype’ and ‘linewidth’. Note that not all terminal drivers support linewidth selection, though.

By default the first plot label is at the top of the legends and successive labels are entered below it. The ‘invert’ option causes the first label to be placed at the bottom of the legends, with successive labels entered above it. This option is useful to force the vertical ordering of labels in the legends to match the order of box types in a stacked histogram.

The length of the sample line can be controlled by 'samplen'. The sample length is computed as the sum of the tic length and <sample_length> times the character width. 'samplen' also affects the positions of point samples in the legends since these are drawn at the midpoint of the sample line, even if the sample line itself is not drawn.

The vertical spacing between lines is controlled by 'spacing'. The spacing is set equal to the product of the pointsize, the vertical tic size, and <vertical_spacing>. The program will guarantee that the vertical spacing is no smaller than the character height.

The <width_increment> is a number of character widths to be added to or subtracted from the length of the string. This is useful only when you are putting a box around the legends and you are using control characters in the text. CPlot class simply counts the number of characters in the string when computing the box width; this allows you to correct it.

The <height_increment> is a number of character heights to be added to or subtracted from the height of the key box. This is useful mainly when you are putting a box around the legends, otherwise it can be used to adjust the vertical shift of automatically chosen legend position by <height_increment>/2.

The defaults for legends are 'on', 'right', 'top', 'vertical', 'Right', 'noreverse', 'noinvert', 'samplen 4', 'spacing 1.25', and 'nobox'. The default <linetype> is the same as that used for the plot borders.

The legends are drawn as a sequence of lines, with one plot described on each line. On the right-hand side (or the left-hand side, if 'reverse' is selected) of each line is a representation that attempts to mimic the way the curve is plotted. On the other side of each line is the text description (the line title), obtained from the member function **CPlot::legend()**. The lines are vertically arranged so that an imaginary straight line divides the left- and right-hand sides of the key. It is the coordinates of the top of this line that are specified in the argument of `option` or member function **CPlot::legendLocation()**. For a 2D plot, only the x and y coordinates are used to specify the line position. For a 3D plot, x, y and z are all used as a 3-d location mapped using the same mapping as the graph itself to form the required 2-d screen position of the imaginary line.

When using the TeX or PostScript drivers, or similar drivers where formatting information is embedded in the string, CPlot class is unable to calculate correctly the width of the string for the legend positioning. If the legends are to be positioned at the left, it may be convenient to use the combination 'left Left reverse'. The box and gap in the grid will be the width of the literal string.

For a 3D contour, the contour labels will be listed in the legends.

Examples:

This places the legends at coordinates 2,3.5,2 in the default (first) coordinate system:

```
plot.legendOption("2,3.5,2");
```

This places the legends below the graph:

```
plot.legendOption("below");
```

This places the legends in the bottom left corner, left-justifies the text, and draws a box around it in linetype 3:

```
plot.legendOption("left bottom Left box 3");
```

Examples

See three examples on pages 211, 212, and 213 for **CPlot::plotType()**.

See Also

CPlot::legend(), **CPlot::legendLocation()**.

CPlot::line

Synopsis

```
#include <chplot.h>
```

```
int line(double x1, double y1, ... /* double x2, double y2; double z1, double x2, double y2, double z2 */);
```

Syntax in Ch

```
line(x1, y1, x2, y2);
```

```
line(x1, y1, z1, x2, y2, z2);
```

Purpose

Add a line to a plot.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x1 The x coordinate of the first endpoint of the line.

y1 The y coordinate of the first endpoint of the line.

z1 The z coordinate of the first endpoint of the line. This argument is ignored for 2D plots.

x2 The x coordinate of the second endpoint of the line.

y2 The y coordinate of the second endpoint of the line.

z2 The z coordinate of the second endpoint of the line. This argument is ignored for 2D plots.

Description

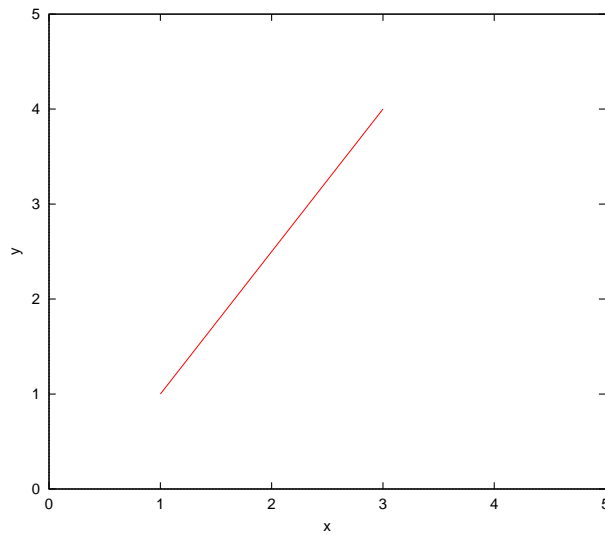
This function adds a line to a plot. It is a convenience function for creation of geometric primitives. A line added with this function is counted as a data set for later calls to **CPlot::legend()** and **CPlot::plotType()**. For 2D rectangular and 3D cartesian plots, (*x1*, *y1*, *z1*) and (*x2*, *y2*, *z2*) are the coordinates of the endpoints of the line, specified in units of the x, y, and z axes. However, for 2D plots, *z1* and *z2* are ignored. For 2D polar and 3D cylindrical plots, the endpoints are specified in polar coordinates where *x* is θ , *y* is *r*, and *z* is unchanged. Again, for 2D plots, *z1* and *z2* are ignored. For 3D plots with spherical coordinates *x* is θ , *y* is ϕ and *z* is *r*.

Example 1

```
/* File: line.ch */
#include <chplot.h>

int main(){
    double x1 = 1, y1 = 1, x2 = 3, y2 = 4;
    class CPlot plot;

    plot.line(x1, y1, x2, y2);
    plot.sizeRatio(-1);
    plot.axisRange(PLOT_AXIS_XY, 0, 5);
    plot.plotting();
}
```

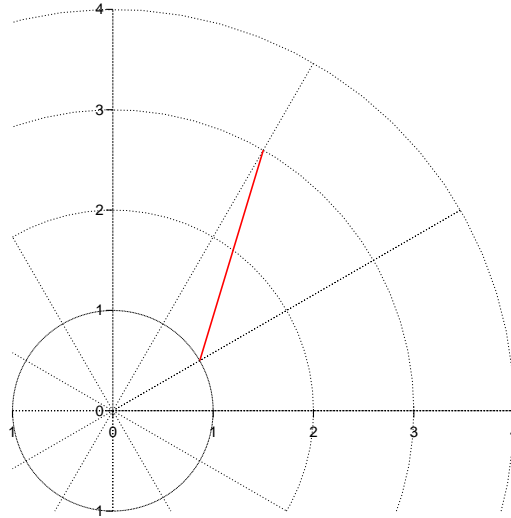
Output**Example 2**

```
/* File: line_2.ch */
#include <chplot.h>
#include <math.h>

int main(){
    double theta1 = 30, theta2 = 60, r1 = 1, r2 = 3;
    class CPlot plot;

    plot.grid(PLOT_ON);
    plot.polarPlot(PLOT_ANGLE_DEG);
    plot.line(theta1, r1, theta2, r2);
    plot.plotType(PLOT_PLOTTYPE_LINES, 0);
    plot.lineType(0, 1, 3);
    plot.sizeRatio(-1);
    plot.axisRange(PLOT_AXIS_XY, -1, 4);
    plot.plotting();
}
```

Output

**See Also**

CPlot::data2D(), **CPlot::data2DCurve()**, **CPlot::data3D()**, **CPlot::data3DCurve()**,
CPlot::data3DSurface(), **CPlot::circle()**, **CPlot::outputType()**, **CPlot::plotType()**, **CPlot::point()**,
CPlot::polygon(), **CPlot::rectangle()**.

CPlot::line1D

Synopsis

```
#include <chplot.h>
void line1D(double x1, double x2, ... /* string_t color */);
```

Syntax

```
line1D(x1, x2);
line1D(x1, x2, "blue");
```

Purpose

Draw a line segment in a one-dimensional plot.

Return Value

None.

Parameters

x1 The first point of the line segment.

x2 The second point of the line segment.

color The color of the line segment.

Description

This function draws a line segment from *x1* to *x2* in a one-dimensional plot. Unless the member function **plot.line1DOffset()**, each line segment is drawn on the same number line. If the member function

plot.line1DOffset() is called, each line segment has an offset of the value 1 vertically from the previous line segment.

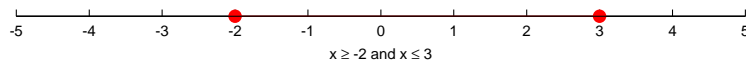
This member function is commonly used to visualize inequalities. Other plotting member functions such as **plot.label()**, **plot.circleClosed()**, **plot.circleOpen()**, **plot.label()**, and **plot.axisRange()** can be called to set some attributes of the plot.

Example 1

```
/* File: line1D.ch
   Inequality  $-2 \leq x \leq 3$  for  $x \geq -2$  and  $x \leq 3$  */
#include <chplot.h>
CPlot plot;

plot.line1D(-2, 3);
plot.axisRange(PLOT_AXIS_X, -5, 5);
plot.circleClosed(-2, 0, "red");
plot.circleClosed(3, 0, "red");
plot.label(PLOT_AXIS_X, "x  $\geq -2$  and  $x \leq 3$ ");
plot.plotting();
```

Output



Example 2

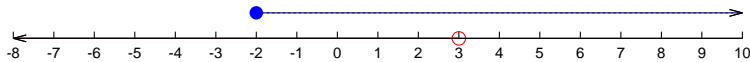
```
/* File: line1DOffset.ch
   Inequality  $x < 3$  for  $x \geq -2$  */
#include <chplot.h>
CPlot plot;

plot.line1DOffset();
plot.line1D(-8, 3);
plot.legend("x < 3", 0);
plot.arrow(3, 0, -8, 0);
plot.circleOpen(3, 0, "red");

plot.line1D(-2, 10);
plot.legend("x  $\geq -2$ ", 2);
plot.arrow(-2, 1, 10, 1);
plot.circleClosed(-2, 1, "blue");
plot.plotting();
```

Output

$x < 3$ ———
 $x \geq -2$ ·····

**See Also**

CPlot::line1DOffset().

CPlot::line1DOffset**Synopsis**

```
#include <chplot.h>
void line1DOffset();
```

Purpose

Set the vertical offset for drawing line segments in a one-dimensional plot.

Return Value

None.

Parameters

None.

Description

The member `plot.line1D()`, draws a line segment on the number line as a one-dimensional plot. Unless the member function `plot.line1DOffsetOffset()`, each line segment is drawn on the same number line. If the member function `plot.line1DOffsetOffset()` is called, each line segment has an offset of the value 1 vertically from the previous line segment.

Example

See `CPlot::line1D()`.

See Also

`CPlot::line1D()`.

CPlot::lineStyle**Synopsis**

```
#include <chplot.h>
```

```
void lineStyle(string_t style);
```

Purpose

Set the line style such as solid and dotted line for lines and curves.

Return Value

None.

Parameters

style The line style.

Description

Set the line style for lines and curves. After this member function is called to set the line style, it will be effective for all lines and curves added afterward. The line style can be changed again later in the program.

Argument	Line Style
" - "	solid line
" . "	dotted line
" - - "	dash line (to be supported)
" . - "	dot-dash line (to be supported)
" . . - "	dot-dot-dash line (to be supported)

Bugs

The line style cannot be set for objects including the arc sector, arc segment, circle, ellipse, rectangle, quadrilateral, triangle, and polygon.

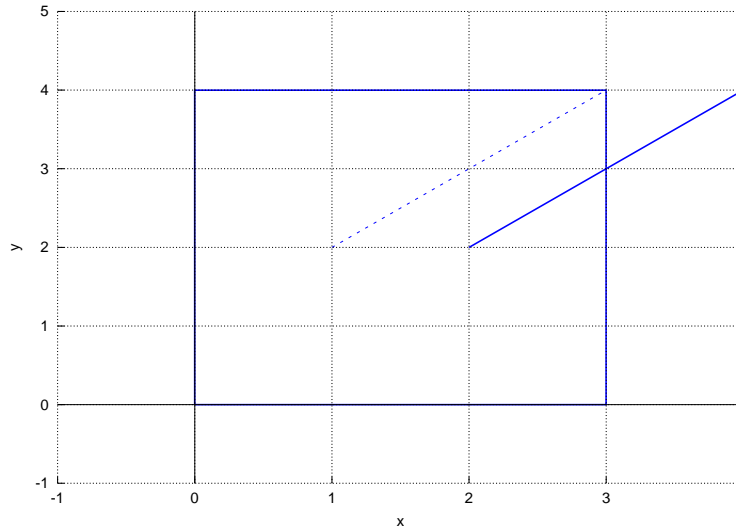
Example 1

```
/* File: lineStyle.ch */
#include <chplot.h>
CPlot plot;

plot.strokeColor("blue");
plot.strokeWidth(3);
plot.lineStyle(".");
plot.line(1, 2, 3, 4);
plot.rectangle(0, 0, 3, 4); // lineStyle() has no effect for objects

plot.lineStyle("-");
plot.line(2, 2, 5, 5);
plot.plotting();
```

Output

**See Also**

CPlot::pointStyle(), CPlot::strokeColor(), CPlot::strokeWidth().

CPlot::lineType

Synopsis

```
#include <chplot.h>
```

```
void lineType(int num, int line_type, int line_width], ... /* [char line_color] */);
```

Syntax

```
lineType(num, line_type, line_width)
```

```
lineType(num, line_type, line_width, line_color)
```

Purpose

Set the line type, width, and color for lines, impulses, steps, etc.

Return Value

None.

Parameters

num The data set to which the line type, width, and color apply.

line_type An integer index representing the line type for drawing. Use the same value for different curves so that each curve with the same style, and same color by default.

line_width A scaling factor for the line width. The line width is *line_width* multiplied by the default width.

line_color color for the line.

Description

Set the desired line type, width, and color for a previously added data set.

Numbering of the data sets starts with zero. The line style and/or marker type for the plot are selected automatically. The default line type, width, and color can be changed by this member function.

The *line_type* specifies an index for the line type used for drawing the line. The line type varies depending on the terminal type used (see **CPlot::outputType**). Typically, changing the line type will change the color of the line or make it dashed or dotted. All terminals support at least six different line types. By default, the line type is 1. The *line_width* specifies the line width. The line width is *line_width* multiplied by the default width.

The default width is two pixels.

The optional fourth argument *line_type* specifies the color of a line by a color name or RGB value, such as "blue" or "#0000ff" for color blue. The default line type, width, and color can be changed by the function call

```
plot.lineType(num, linetype, linewidth, "blue");
```

The color of the line is specified as blue in this example. The detailed color specification can be found in the member function **CPlot::strokeColor()**.

Example 1

```
/* File: lineType.ch */
#include <math.h>
#include <chplot.h>

int main() {
    array double x[36], y[36];
    int line_type = 1, line_width = 4, datasetnum = 0;
    CPlot plot;

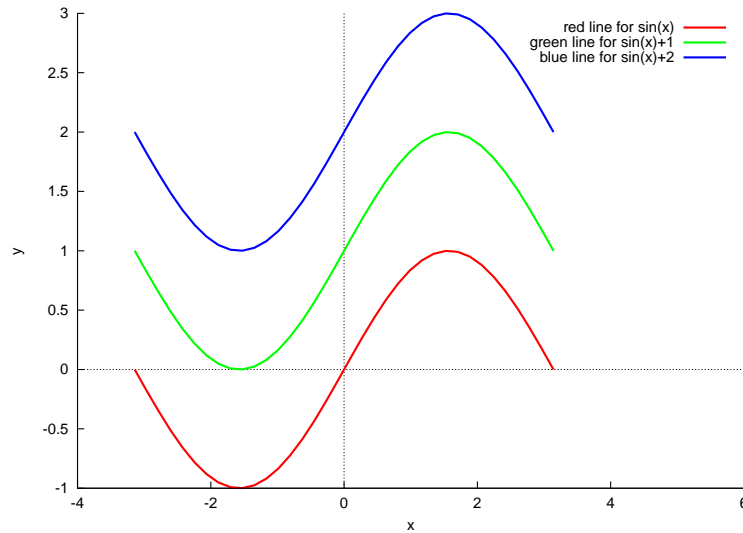
    lindata(-M_PI, M_PI, x);
    y = sin(x);
    plot.data2D(x, y);
    plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum);
    plot.lineType(datasetnum, line_type, line_width, "red");
    plot.legend("red line for sin(x)", datasetnum);

    y = sin(x)+1;
    plot.data2D(x, y);
    plot.plotType(PLOT_PLOTTYPE_LINES, ++datasetnum);
    plot.lineType(datasetnum, line_type, line_width, "green");
    plot.legend("green line for sin(x)+1", datasetnum);

    y = sin(x)+2;
    plot.data2D(x, y);
    plot.plotType(PLOT_PLOTTYPE_LINES, ++datasetnum);
    plot.lineType(datasetnum, line_type, line_width, "blue");
    plot.legend("blue line for sin(x)+2", datasetnum);

    plot.axisRange(PLOT_AXIS_X, -4, 6);
    plot.plotting();
}
```

Output

**See Also**

CPlot::plotType(), **CPlot::pointType()**.

CPlot::margins

Synopsis

```
#include <chplot.h>
```

```
void margins(double left, double right, double top, double bottom);
```

Purpose

Set the size of the margins around the edge of the plot.

Return Value

None.

Parameters

left The size of the left margin in character width.

right The size of the right margin in character width.

top The size of the top margin in character height.

bottom The size of the bottom margin in character height.

Description

By default, the plot margins are calculated automatically. They can be set manually with this function. Specifying a negative value for a margin causes the default value to be used.

Example

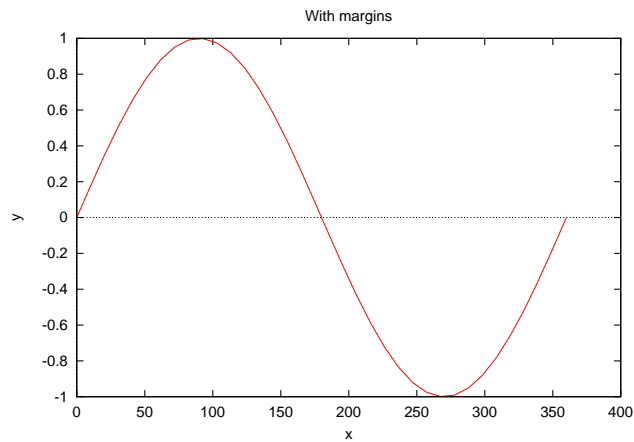
Compare with the output for examples in **CPlot::data2D()** and **CPlot::data2DCurve()**.

```
/* File: margins.ch */
#include <math.h>
```

```
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    lindata(0, 360, x);
    y = sin(x*M_PI/180);
    plot.data2D(x, y);
    plot.margins(15, 10, 5, 7);
    plot.title("With margins");
    plot.plotting();
}
```

Output**See Also**

CPlot::borderOffsets().

CPlot::mathCoord**Synopsis**

```
#include <chplot.h>
void mathCoord();
```

Purpose

Set proper attributes for math coordinates.

Return Value

None.

Parameters

None.

Description

This function sets proper attributes for math coordinates.

Starting in Ch v7.5, this member function is called by default in the CPlot class constructor.

The plot uses solid lines in the black color for x and y axes. The plot will have grids, but no board and no tics on mirror. The plot uses enhanced text with math symbols, such as \leq , \geq , and π , for for all texts including title, label, legend, and text in the plot. See **CPlot::enhanceText()** section for detailed information about enhanced text for math symbols. The function calls the following member functions internally.

```
// use solid lines for x-y axes
lineType=1; lineWidth=2;
plot.axis(PLOT_AXIS_XY, lineType, lineWidth, "black");
plot.border(PLOT_BORDER_ALL, PLOT_OFF); // no border
plot.ticsMirror(PLOT_AXIS_XY, PLOT_OFF); // no tics on mirror
plot.grid(PLOT_ON, "front"); // turn grids on
plot.enhanceText(); // use enhanced text
```

In addition, if **CPlot::plotType()** has been called with **PLOT_PLOTTYPE_FILLEDCURVE** multiple times for filled curves, such as

```
plot.plotType(PLOT_PLOTTYPE_FILLEDCURVES, num, "y1=20", "blue");
```

CPlot::mathCoord() calls

```
plot.fillStyle(num, PLOT_FILLSTYLE_TRANSPARENT_SOLID, 0.6);
```

to fill each area with the transparent solid color.

The appearance of the plot, set by **CPlot::mathCoord()**, can be changed or overwritten by calling member functions.

This function is useful for beginners to learn math using **CPlot**.

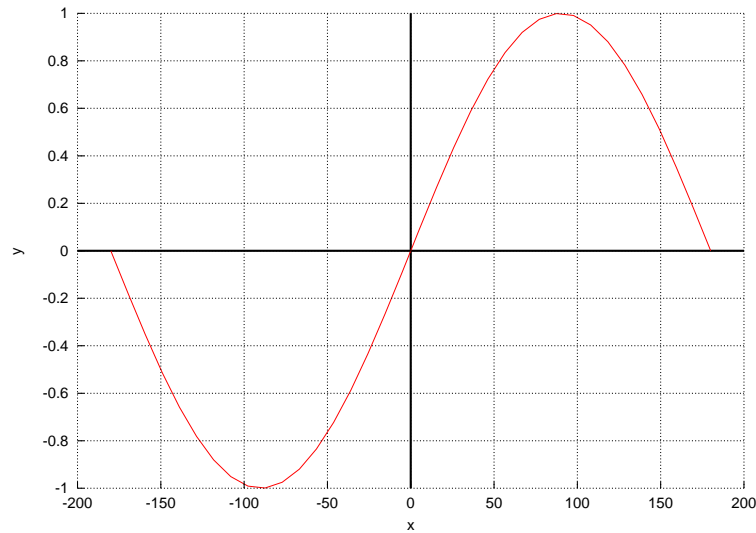
Example 1

```
/* File: mathCoord.ch */
#include <math.h>
#include <chplot.h>
#include <numeric.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    lindata(-180, 180, x);
    y = sin(x*M_PI/180);
    plot.mathCoord();
    plot.data2D(x, y);
    plot.plotting();
}
```

Output

**Example 2**

```

/* File: mathCoord_2.ch
   Plotting the inequality  $y < 8x+40$  and  $y < 10x+30$  for  $x$  from 0 to 16. */
#include <chplot.h> /* for CPlot */

double func(double x) {
    double y;
    y = 8 * x + 40;
    return y;
}

double func2(double x) {
    double y;
    y = 10 * x + 30;
    return y;
}

int main() {
    CPlot plot;
    int num = 0;

    plot.mathCoord();
    plot.title("y < 8x+40 and y < 10x+30");
    plot.label(PLOT_AXIS_X, "x");
    plot.label(PLOT_AXIS_Y, "y");
    plot.axisRange(PLOT_AXIS_Y, 20, 200);
    plot.func2D(0, 16, 500, func);
    plot.plotType(PLOT_PLOTTYPE_FILLEDCURVES, num, "y1=20", "blue");

    plot.func2D(0, 16, 500, func2);
    num++;
    plot.plotType(PLOT_PLOTTYPE_FILLEDCURVES, num, "y1=20", "yellow");
    plot.plotting();
    return 0;
}

```

Output: The output is similar to that from the program `mathCoord_2.cpp` described in `CPlot::fillStyle()` on page 118.

See Also

CPlot::axis(), **CPlot::border()**, **CPlot::enhanceText()**, **CPlot::fillStyle()**, **CPlot::grid()**, **CPlot::plotType()**, **CPlot::ticsMirror()**.

CPlot::noFill

Synopsis

```
#include <chplot.h>
void noFill(void);
```

Purpose

Do not fill the interior of objects.

Return Value

None.

Parameter None**Description**

When the member function **CPlot::noFill()** is called, the interior of objects, including the arc sector, arc segment, circle, ellipse, rectangle, quadrilateral, will not be filled, until it is changed by calling the member function **CPlot::fillColor()** to set the filled color or **CPlot::fillPattern()** to set the filled pattern.

Example 1

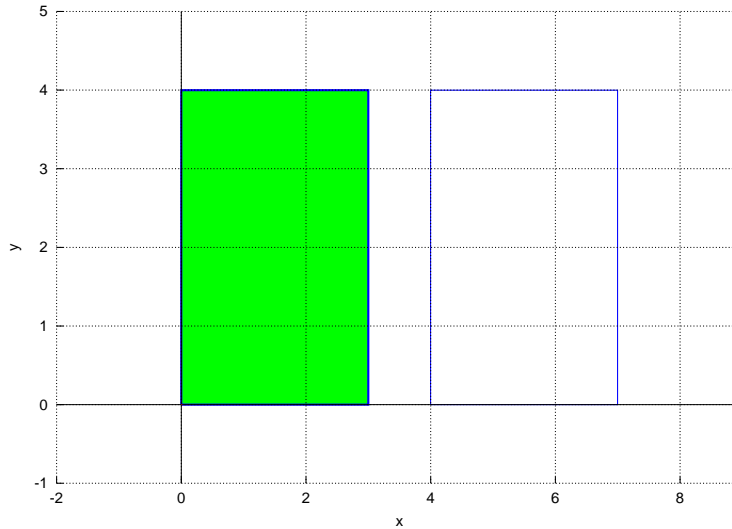
```
/* File: noFill.ch */
#include <chplot.h>
CPlot plot;

plot.strokeColor("blue");
plot.fillColor("green");
plot.rectangle(0, 0, 3, 4);

plot.noFill();
plot.rectangle(4, 0, 3, 4);

plot.plotting();
```

Output

**See Also**

CPlot::fillColor(), **CPlot::fillOpacity()**, **CPlot::fillPattern()**, **CPlot::noStrokeColor()**, **CPlot::strokeColor()**, **CPlot::strokeWidth()**.

CPlot::noStrokeColor

Synopsis

```
#include <chplot.h>
void noStrokeColor(void);
```

Purpose

Do not draw the outline of objects.

Return Value

None.

Parameter None**Description**

When the member function **CPlot::noStrokeColor()** is called, the outline of objects, including the arc sector, arc segment, circle, ellipse, rectangle, quadrilateral, will not be drawn, until it is changed by calling the member function **CPlot::strokeColor()** to set the stroke color.

The member function **CPlot::noStrokeColor()** do not affect the color of a point or line.

Example

See **CPlot::fillColor()**.

See Also

CPlot::fillColor(), **CPlot::fillOpacity()**, **CPlot::fillPattern()**, **CPlot::noFill()**, **CPlot::strokeColor()**, **CPlot::strokeWidth()**.

CPlot::numberLine

Synopsis

```
#include <chplot.h>
```

```
void numberLine(double x1, double x2, ... /* double x3, x4, ... */);
```

Syntax

```
numberLine(x1, x2);
```

```
numberLine(x1, x2, x3);
```

```
numberLine(x1, x2, x3, x4);
```

Purpose

Draw a number Line in a one-dimensional plot.

Return Value

None.

Parameters

x1 The first point on the number line.

x2 The second point on the number line.

x3 The third point on the number line.

x4 The fourth point on the number line.

Description

This function draws a line segment with an arrow for each two adjacent points on the number line, such as from x_1 to x_2 , x_2 to x_3 , etc. Except the first line segment, each line segment has an offset vertically from the number line.

The color sequences of the number line functions `plot.numberLine()`, `plot.numberLineScatter()`, and `plot.numberLineScattern()` are shared.

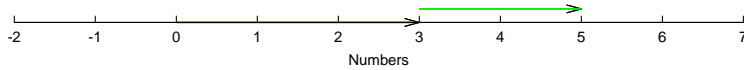
Other plotting member functions such as `plot.label()` and `plot.axisRange()` can be called to set some attributes of the plot.

Example

```
/* File: numberLine.ch */
#include <chplot.h>
CPlot plot;

plot.numberLine(0, 3, 2);
plot.label(PLOT_AXIS_X, "Numbers");
plot.plotting();
```

Output

**See Also**

CPlot::numberLineScatter(), **CPlot::numberLineScattern()**.

CPlot::numberLineScatter

Synopsis

```
#include <chplot.h>
```

```
void numberLineScatter(double x[:], ... /* string_t color */);
```

Syntax

```
numberLineScattern(x);
```

```
numberLineScattern(x, "blue");
```

Purpose

Draw a number Line in scatter plot in a one-dimensional plot.

Return Value

None.

Parameters

x An array containing the values on the number line.

color The color of the points for the scatter plot.

Description

This function draws a scatter plot using the data in the array *x* with an offset from the number line.

The color sequences of the number line functions **plot.numberLine()**, **plot.numberLineScatter()**, and **plot.numberLineScattern()** are shared.

Other plotting member functions such as **plot.label()** and **plot.axisRange()** can be called to set some attributes of the plot.

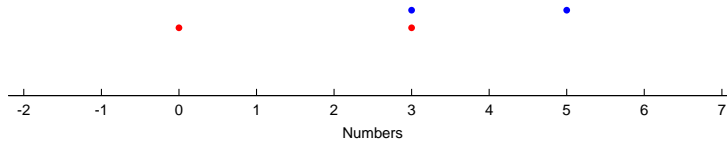
Example

```
/* File: numberLineScatter.ch */
#include <chplot.h>
CPlot plot;
array double x1[] = {0, 3};
array double x2[] = {3, 3+2};

plot.numberLineScatter(x1);
plot.numberLineScatter(x2, "blue");
```

```
plot.label(PLOT_AXIS_X, "Numbers");
plot.plotting();
```

Output



See Also

CPlot::numberLine(), CPlot::numberLineScattern().

CPlot::numberLineScattern

Synopsis

```
#include <chplot.h>
```

```
void numberLineScattern(double x[], int n, ... /* string_t color */);
```

Syntax

```
numberLineScattern(x, n);
```

```
numberLineScattern(x, n, "blue");
```

Purpose

Draw a number Line in scatter plot in a one-dimensional plot.

Return Value

None.

Parameters

x An array containing the values on the number line.

n The number of elements in array *x*. If the argument of *x* is type of pointer to double, the number of elements of the array is specified by this argument.

color The color of the points for the scatter plot.

Description

This function draws a scatter plot using the data in the array *x* with an offset from the number line.

The color sequences of the number line functions **plot.numberLine()**, **plot.numberLineScattern()**, and **plot.numberLineScatter()** are shared.

Other plotting member functions such as **plot.label()** and **plot.axisRange()** can be called to set some attributes of the plot.

Example

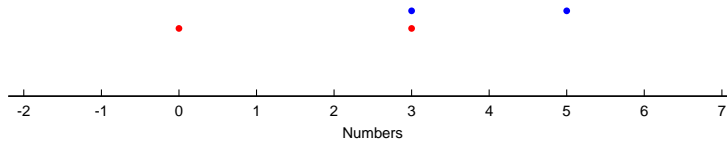
```

/* File: numberLineScatter.ch */
#include <chplot.h>
CPlot plot;
array double x1[2] = {0, 3};
array double x2[2] = {3, 3+2};

plot.numberLineScatter(x1, 2);
plot.numberLineScatter(x2, 2, "blue");
plot.label(PLOT_AXIS_X, "Numbers");
plot.plotting();

```

Output



See Also

CPlot::numberLine(), **CPlot::numberLineScatter()**.

CPlot::origin

Synopsis

```

#include <chplot.h>
void origin(double x_orig, double y_orig);

```

Purpose

Set the location of the origin for the drawing of the plot.

Return Value

None.

Parameters

x_orig The x coordinate of the origin for the drawing of the plot.

y_orig The y coordinate of the origin for the drawing of the plot.

Description

This function specifies the location of the origin for the drawing of the plot. This is the location of the bottom left corner of the bounding box of the plot, not the location of the origin of the plot coordinate system. The values *x_orig* and *y_orig* are specified as numbers between 0 and 1, where (0,0) is the bottom left of the plot area (the plot window) and (1,1) is the top right. This function is used internally by method **CPlot::subplot()** in conjunction with method **CPlot::size()**.

See Also

CPlot::getSubplot(), **CPlot::size()**, **CPlot::subplot()**.

CPlot::outputType

Synopsis

```
#include <chplot.h>
```

```
void outputType(int outputtype, ... /* [string_t terminal, string_t filename] */ );
```

Syntax

```
outputType(outputtype)
```

```
outputType(PLOT_OUTPUTTYPE_DISPLAY)
```

```
outputType(PLOT_OUTPUTTYPE_STREAM, terminal)
```

```
outputType(PLOT_OUTPUTTYPE_FILE, terminal, filename)
```

Purpose

Set the output type for a plot.

Return Value

None.

Parameters

outputtype This can have any of the following values:

PLOT_OUTPUTTYPE_DISPLAY Display the plot on the screen. The plot is displayed in its own separate window. A plot window can be closed by pressing the ‘q’ key in the X-Windows system.

PLOT_OUTPUTTYPE_STREAM Output the plot as a standard output stream. This output type is useful for CGI (Common Gateway Interface) when a Ch program is used as CGI script in a Web server.

PLOT_OUTPUTTYPE_FILE Output the plot to a file in one of a variety of formats. If this output option is selected two additional arguments are necessary: the *terminal* type and *filename*.

terminal Supported terminal types when gnuplot is used as a plotting engine are as follow:

Terminal	Description
aifm	Adobe Illustrator 3.0.
corel	EPS format for CorelDRAW.
dxl	AutoCAD DXF.
dxl800a	Roland DXY800A plotter.
eepic	Extended L ^A T _E Xpicture.
emtex	L ^A T _E Xpicture with emTeX specials.
epson-180dpi	Epson LQ-style 24-pin printer with 180dpi.
epson-60dpi	Epson LQ-style 24-pin printers with 60dpi.
epson-lx800	Epson LX-800, Star NL-10 and NX-100.
excl	Talaris printers.

fig	Xfig 3.1.
gif	GIF file format.
gpic	gpic/groff package.
hp2648	Hewlett Packard HP2647 an HP2648.
hp500c	Hewlett Packard DeskJet 500c.
hpdj	Hewlett Packard DeskJet 500.
hpgl	HPGL output.
hpljii	HP LaserJet II.
hppj	HP PaintJet and HP3630 printers.
latex	L ^A T _E Xpicture.
mf	MetaFont.
mif	Frame Maker MIF 3.00.
nec-cp6	NEC CP6 and Epson LQ-800.
okidata	9-pin OKIDATA 320/321 printers.
pcl5	Hewlett Packard LaserJet III.
pbm	Portable BitMap.
pdf	PDF file.
png	Portable Network Graphics.
pngcairo	Portable Network Graphics via cairo and pango.
postscript	Postscript.
pslatex	L ^A T _E Xpicture with postscript specials.
pstricks	L ^A T _E Xpicture with PSTricks macros.
starc	Star Color Printer.
tandy-60dpi	Tandy DMP-130 series printers.
texdraw	L ^A T _E Xtexdraw format.
tgif	TGIF X-Window drawing format.
tpic	L ^A T _E Xpicture with tpic specials.

`aifm` Output an Adobe Illustrator 3.0 file. The format for the *terminal* string is:

```
"aifm [colormode] [\"fontname\"] [fontsize]"
```

`colormode` can be `color` or `monochrome`.
`fontname` is the name of a valid PostScript font.
`fontsize` is the size of the font in points.

Defaults are `monochrome`, `"Helvetica"` and `14pt`.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "aifm color", "plot.aif");`

`corel` Output EPS format for CorelDRAW.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "corel", "plot.eps");`

`dxfl` Output AutoCAD DXF file.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "dxf", "plot.dxf");`

`dxy800a` Output file for Roland DXY800A plotter.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "dxy800a", "plot.dxy");`

`eepic` Output extended \LaTeX picture.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "eepic", "plot.tex");`

`emtex` Output \LaTeX picture with emTeX specials.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "emtex", "plot.tex");`

`epson-180dpi` Epson LQ-style 24-pin printers with 180dpi.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "epson-180dpi", "plot");`

`epson-60dpi` Epson LQ-style 24-pin printers with 60dpi.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "epson-60dpi", "plot");`

`epson-1x800` Epson LX-800, Star NL-10 and NX-100.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "epson-1x800", "plot");`

`excl` Talaris printers such as EXCL Laser printer and 1590.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "excl", "plot");`

`fig` Xfig 3.1 file. The format for the *terminal* string is:

```
"fig [colormode] [size] [pointsmax num_points] [orientation]
[units] [fontsize fntsize] [size xsize ysize] [thickness width]
[depth layer]"
```

`colormode` can be `monochrome` or `color`

`size` can be `small` or `big`

`num_points` is the maximum number of points per polyline.

`orientation` can be `landscape` or `portrait`. `units` can be `metric` or `inches`.

`fontsize` is the size of the text font in points. Must be preceded by the `fontsize` keyword.

`xsize` and `ysize` set the size of the drawing area. Must be preceded by the `size` keyword.

`width` is the line thickness. Must be preceded by the `thickness` keyword.

`layer` is the line depth. Must be preceded by the `depth` keyword.

Default values are: monochrome small pointsmax 1000 landscape inches
fontsize 10 thickness 1 depth 10.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "fig color big",
"plot.fig");`

`gif` GIF file format. The format for the *terminal* string is:

```
"gif [transparent] [interlace] [font_size] [size x,y]
[color0 color1 color2 ...]
```

Specifying the `transparent` keyword will generate a transparent GIF. By default, white is the transparent color.

Specifying the `interlace` key word will generate an interlaced GIF.

`font_size` is `small` (6x12 pixels), `medium` (7x13 pixels), or `large` (8x16 pixels).

`x` and `y` are the image size in pixels. Must be preceded by the `size` keyword.

`colors` are specified in the format "xrrggbb" where `x` is the character "x" and "rrggbb" are the RGB components of the color in hexadecimal. A maximum of 256 colors can be set. If the GIF is transparent, the first color is used as the transparent color.

The default values are: `small size 640,480`

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "gif size 1024,768",
"plot.gif");`

`gpic` Output for use with the Free Software Foundation `gpic/groff p` package. The format for the *terminal* string is:

```
"gpic [x] [y]"
```

where `x` and `y` are the location of the origin. Default is (0,0).

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "gpic 5 5", "plot.gpic");`

`hp2633a` Hewlett Packard HP2623A.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "hp2633a", "plot");`

`hp2648` Hewlett Packard HP2647 an HP2648.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "hp2648", "plot");`

`hp500c` Hewlett Packard DeskJet 500c. The format for the *terminal* string is:

```
"hp500c [resolution] [compression]"
```

resolution can be 75, 100, 150, or 300 dpi.
compression can be rle or tiff.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "hp500c 100 tiff", "plot");`

`hpdj`: Hewlett Packard DeskJet 500. The format for the *terminal* string is:

`"hp500c [resolution]"`

resolution can be 75, 100, 150, or 300 dpi. Default is 75.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "hpdj 100", "plot");`

`hpgl` Produces HPGL output for devices such as the HP7475A plotter. The format for the *terminal* string is:

`"hpgl [num_of_pens] [eject]"`

`num_of_pens` is the number of available pens. The default is 6.
`eject` is a keyword that tells the plotter to eject a page when done.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "hpgl 4", "plot");`

`hpljii` HP LaserJet II

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "hpljii", "plot");`

`item[hppj]` HP PaintJet and HP3630 printers. The format for the *terminal* string is:

`"hppj [font]"`

font can be FNT5X9, FNT9x17, or FNT13X25. Default is FNT9x17.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "hppj FNT5X9", "plot");`

`latex` Output a \LaTeX picture. The format of the *terminal* string is:

`"latex [font] [size]"`

where `font` can be `courier` or `roman` and `size` can be any point size. Default is `roman 10pt`.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "latex", "plot.tex");`

mf Output file for the MetaFont program.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "mf", "plot.mf");`

mif Output Frame Maker MIF file format version 3.00. The format of the *terminal* string is:

```
"mif pentype curvetype"
```

pentype can be: colour or monochrome.

curvetype can be:

polyline curves drawn as continuous lines

vectors curves drawn as a collection of vectors

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "mif colour vectors", "plot.mif");`

nec-cp6 Generic 24-pin printer such as NEC CP6 and Epson LQ-800.

The format for the *terminal* string is:

```
"nec-cp6a [option]"
```

option can be monochrome, colour, or draft.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "nec-cp6a draft", "plot");`

okidata 9-pin OKIDATA 320/321 printers.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "okidata", "plot");`

pcl5 Hewlett Packard LaserJet III. This actually produces HPGL-2. The format of the *terminal* string is:

```
"pcl5 [mode] [font] [fontsize]"
```

mode is landscape or portrait.

font is stick, univers, or cg_times.

fontsize is the font size in points.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "pcl5 landscape", "plot");`

pbm Output a Portable BitMap. The format for the *terminal* string is:

```
"pbm [fontsize] [colormode]"
```

fontsize is small, medium, or large.
colormode is monochrome, gray, or color.

Example:

```
plot.outputType(PLOT_OUTPUTTYPE_FILE, "pbm medium gray",
               "plot.pbm");
```

png Portable Network Graphics format. The format of the *terminal* string is:

```
"png [fontsize]"
```

fontsize can be small, medium, or large.

Default is small and monochrome with the output size of 640x480 pixel. Use member function **CPlot::size()** to change the size of the plot.

Example:

```
plot.outputType(PLOT_OUTPUTTYPE_FILE, "png", "plot.png");
```

pngcairo Portable Network Graphics format via cairo and pango. It supports enhanced text. The format of the *terminal* string is:

```
"pngcairo {enhanced} {[fontsize]}"
```

fontsize can be small, medium, or large.

Default is small and monochrome with the output size of 640x480 pixel. Use member function **CPlot::size()** to change the size of the plot.

Example:

```
plot.outputType(PLOT_OUTPUTTYPE_FILE, "pngcairo", "plot.png");
```

postscript This produces a postscript file. The format for the *terminal* string is:

```
"postscript [mode] [colormode] [dash] [\"fontname\"] [fontsize]"
```

mode can be landscape, portrait, eps, or default

colormode can be color or monochrome.

dash can be solid or dashed.

fontname is the name of a valid PostScript font.

fontsize is the size of the font in points.

The default mode is landscape, monochrome, dashed, "Helvetica", 14pt.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "postscript eps \"Times\" 11", "plot.eps");`

`pslatex` Output \LaTeX picture with postscript specials.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "pslatex", "plot.tex");`

`pstricks` Output \LaTeX picture with PSTricks macros.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "pstricks", "plot.tex");`

`starc` Star Color Printer.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "starc", "plot");`

`tandy-60dpi` Tandy DMP-130 series printers.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "tandy-60dpi", "plot");`

`texdraw` Output \LaTeX texdraw format.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "texdraw", "plot.tex");`

`tgif` Output TGIF X-Window drawing format.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "tgif", "plot.tgif");`

`tpic` Output \LaTeX picture with tpic specials.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "tpic", "plot.tex");`

filename The filename the plot is saved to. On machines that support `popen()` functions, the output can also be piped to another program by placing the `'|'` character in front of the command name and using it as the *filename*. For example, on Unix systems, setting *terminal* to "postscript" and *filename* to "`|lp`" could be used to send a plot directly to a postscript printer.

Description

This function is used to display a plot on the screen, save a plot to a file, or generate a plot to the stdout stream in GIF format for use on the World Wide Web. By default, the output type is `PLOT_OUTPUTTYPE_DISPLAY`.

Example 1

```
/* File: outputType.ch */
/* a plot is created in postscript file plot.eps */
#include <math.h>
#include <chplot.h>
```

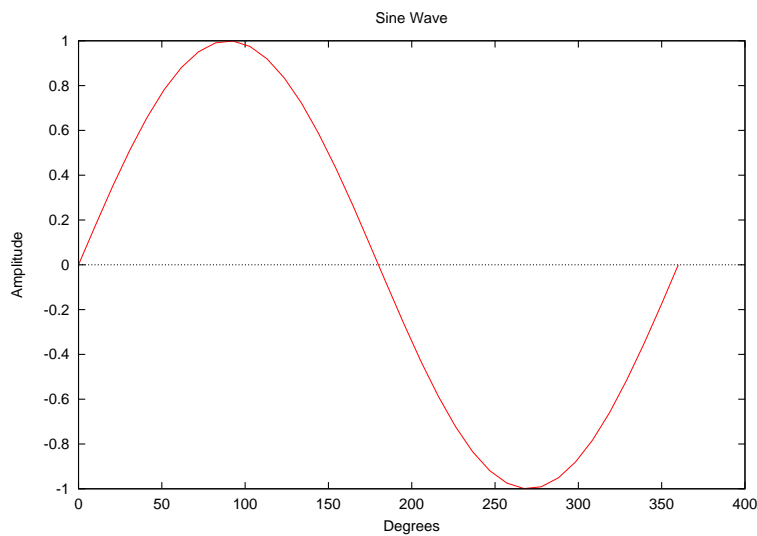
```

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    string_t title="Sine Wave",           // Define labels.
              xlabel="Degrees",
              ylabel="Amplitude";
    class CPlot plot;

    lndata(0, 360, x);
    y = sin(x*M_PI/180);                 // Y-axis data.
    plot.title(title);
    plot.label(PLOT_AXIS_X, xlabel);
    plot.label(PLOT_AXIS_Y, ylabel);
    plot.data2D(x, y);
    plot.outputType(PLOT_OUTPUTTYPE_FILE, "postscript eps color", "plot.eps");
    plot.plotting();
}

```

Output



Example 2

```

/* File: outputType_2.ch */
/* In this example, the plot is saved as an xfig file first.
   Next the xfig program is invoked. The plot can then be edited using xfig in Unix */
#include <math.h>
#include <chplot.h>

int main() {
    double x[20], y[30], z[600];
    int i,j;
    class CPlot plot;

    lndata(-3, 3, x);
    lndata(-4, 4, y);
    for(i=0; i<20; i++) {
        for(j=0; j<30; j++) {
            z[30*i+j] = 3*(1-x[i])*(1-x[i])*exp(-(x[i]*x[i])-(y[j]+1)*(y[j]+1))

```



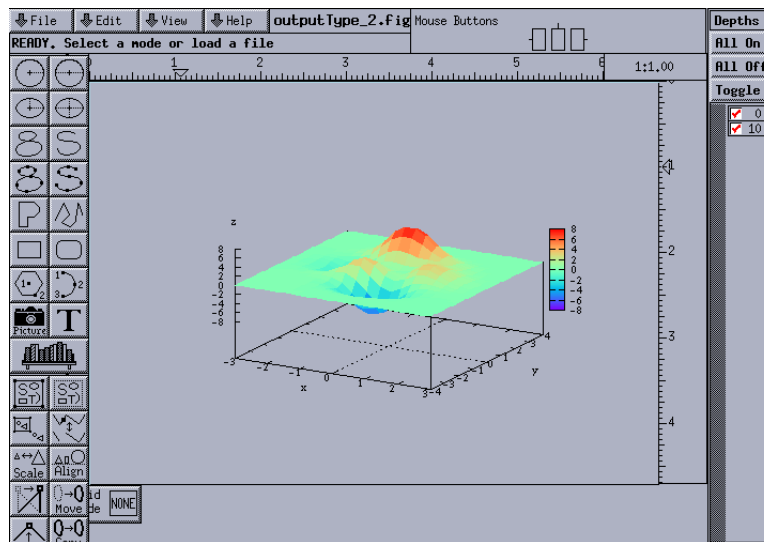
```

        - 10*(x[i]/5 - x[i]*x[i]*x[i]-pow(y[j],5))*exp(-x[i]*x[i]-y[j]*y[j])
        - 1/3*exp(-(x[i]+1)*(x[i]+1)-y[j]*y[j]);
    }
}
plot.data3D(x, y, z);
plot.outputType(PLOT_OUTPUTTYPE_FILE, "fig color", "../output/outputType_2.fig");
plot.plotting();

xfig ../output/outputType_2.fig&
}

```

Output



Example 3

To run this code as a CGI program in a Web server, place `outputType.ch` in your `cgi-bin` directory. If you place `outputType.ch` in a different directory, change `/cgi-bin` to specify the correct location. In your Web browser, open the html file.

```

#!/bin/ch

/* This file is called outputType.ch, a plot is created in png format */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    string_t title="Sine Wave",
             xlabel="Degrees",
             ylabel="Amplitude";
    class CPlot plot;

    lindata(0, 360, x);
    y = sin(x*M_PI/180);
    setbuf(stdout, NULL);
    printf("Content-type: image/png\n\n");
    plot.title(title);
    plot.label(PLOT_AXIS_X, xlabel);
}

```

```

    plot.label(PLOT_AXIS_Y, ylabel);
    plot.data2D(x, y);
    plot.outputType(PLOT_OUTPUTTYPE_STREAM, "png color");
    plot.plotting();
}

```

outputType.html

```

<! this file is called outputType.html >
<html>
<head>
<title>
Example Plot
</title>
</head>



</html>

```

CPlot::plotType

Synopsis

```
#include <chplot.h>
```

```
void plotType(int plot_type, int num, ... /* [char option] */);
```

Obsolete Synopsis

```
#include <chplot.h>
```

```
void plotType(int plot_type, int num, ... /* [ [int line_type, int line_width],
[int point_type, int point_size], [char option] */);
```

Syntax

```

plotType(PLOT_PLOTTYPE_LINES, num)
plotType(PLOT_PLOTTYPE_IMPULSES, num)
plotType(PLOT_PLOTTYPE_FSTEPS, num)
plotType(PLOT_PLOTTYPE_HISTEPS, num)
plotType(PLOT_PLOTTYPE_POINTS, num)
plotType(PLOT_PLOTTYPE_LINESPOINTS, num)
plotType([PLOT_PLOTTYPE_STEPS, num)
plotType(PLOT_PLOTTYPE_DOTS, num)
plotType(PLOT_PLOTTYPE_DOTS, num, dot_type)
plotType(PLOT_PLOTTYPE_SURFACES, num)
plotType(PLOT_PLOTTYPE_FINANCEBARS, num)
plotType(PLOT_PLOTTYPE_BOXES, num)
plotType(PLOT_PLOTTYPE_BOXERRORBARS, num)
plotType(PLOT_PLOTTYPE_BOXXYERRORBARS, num)
plotType(PLOT_PLOTTYPE_XERRORBARS, num)
plotType(PLOT_PLOTTYPE_XYERRORBARS, num)
plotType(PLOT_PLOTTYPE_YERRORBARS, num)
plotType(PLOT_PLOTTYPE_XERRORLINES, num)
plotType(PLOT_PLOTTYPE_XYERRORLINES, num)

```

plotType(PLOT_PLOTTYPE_YERRORLINES, *num*)
plotType(PLOT_PLOTTYPE_VECTORS, *num*)
plotType(PLOT_PLOTTYPE_VECTORS, *num*, *option*)
plotType(PLOT_PLOTTYPE_CANDLESTICKS, *num*)
plotType(PLOT_PLOTTYPE_CANDLESTICKS, *num*, *option*)
plotType(PLOT_PLOTTYPE_FILLEDCURVES, *num*)
plotType(PLOT_PLOTTYPE_FILLEDCURVES, *num*, *option*)
plotType(PLOT_PLOTTYPE_FILLEDCURVES, *num*, *option*, **char** *color*)

Obsolete Syntax

plotType(PLOT_PLOTTYPE_LINES, *num*, *line_type*, *line_width*)
plotType(PLOT_PLOTTYPE_IMPULSES, *num*, *line_type*, *line_width*)
plotType(PLOT_PLOTTYPE_STEPS, *num*, *line_type*, *line_width*)
plotType(PLOT_PLOTTYPE_FSTEPS, *num*, *line_type*, *line_width*)
plotType(PLOT_PLOTTYPE_HISTEPS, *num*, *line_type*, *line_width*)
plotType(PLOT_PLOTTYPE_POINTS, *num*, *point_type*, *point_size*)
plotType(PLOT_PLOTTYPE_LINESPOINTS, *num*, *line_type*, *line_width*, *point_type*, *point_size*)
plotType(PLOT_PLOTTYPE_FINANCEBARS, *num*, *line_type*, *line_width*)
plotType(PLOT_PLOTTYPE_BOXES, *num*, *line_type*, *line_width*)
plotType(PLOT_PLOTTYPE_BOXERRORBARS, *num*, *line_type*, *line_width*)
plotType(PLOT_PLOTTYPE_BOXXYERRORBARS, *num*, *line_type*, *line_width*)
plotType(PLOT_PLOTTYPE_XERRORBARS, *num*, *line_type*, *line_width*)
plotType(PLOT_PLOTTYPE_XYERRORBARS, *num*, *line_type*, *line_width*)
plotType(PLOT_PLOTTYPE_YERRORBARS, *num*, *line_type*, *line_width*)
plotType(PLOT_PLOTTYPE_XERRORLINES, *num*, *line_type*, *line_width*)
plotType(PLOT_PLOTTYPE_XYERRORLINES, *num*, *line_type*, *line_width*)
plotType(PLOT_PLOTTYPE_YERRORLINES, *num*, *line_type*, *line_width*)

Use

lineType(*num*, *line_type*, *line_width*)
lineType(*num*, *line_type*, *line_width*, *line_color*)
pointType(*num*, *point_type*, *point_size*)
pointType(*num*, *point_type*, *point_size*, *point_color*)

Purpose

Set the plot type for a data set.

Return Value

None.

Parameters

plot_type The plot type. Valid values are:

PLOT_PLOTTYPE_LINES Data points are connected with a line. This is the default for 2D plots. When this plot type used for 3D plot, the surface is meshed with wire frames.

PLOT_PLOTTYPE_IMPULSES Display vertical lines from the x-axis (for 2D plots) or the xy plane (for 3D plots) to the data points.

- PLOT_PLOTTYPE_STEPS** Adjacent points are connected with two line segments, one from (x_1, y_1) to (x_2, y_1) , and a second from (x_2, y_1) to (x_2, y_2) . This type is available only for 2D plots.
- PLOT_PLOTTYPE_FSTEPS** Adjacent points are connected with two line segments, one from (x_1, y_1) to (x_1, y_2) , and a second from (x_1, y_2) to (x_2, y_2) . This type is available only for 2D plots.
- PLOT_PLOTTYPE_HISTEPS** This type is intended for plotting histograms. The point x_1 is represented by a horizontal line from $((x_0+x_1)/2, y_1)$ to $((x_1+x_2)/2, y_1)$. Adjacent lines are connected with a vertical line from $((x_1+x_2)/2, y_1)$ to $((x_1+x_2)/2, y_2)$. This type is available only for 2D plots.
- PLOT_PLOTTYPE_POINTS** Markers are displayed at each data point.
- PLOT_PLOTTYPE_LINESPOINTS** Markers are displayed at each data point and connected with a line.
- PLOT_PLOTTYPE_DOTS** A small dot is displayed at each data point. The optional *point_type* argument effects only the color of the dot.
- PLOT_PLOTTYPE_SURFACES** Data points are meshed as a smooth surface. This is the default for 3D plots.
- PLOT_PLOTTYPE_FILLEDCURVES** The `filledcurves` plot type is only relevant to 2D plotting. Three variants are possible. The first two variants require two columns of input data, and may be further modified by the options listed below. The first variant, `closed`, treats the curve itself as a closed polygon. This is the default. The second variant is to fill the area between the curve and a given axis, a horizontal or vertical line, or a point. The third variant requires three columns of input data: the x coordinate and two y coordinates corresponding to two curves sampled at the same set of x coordinates; the area between the two curves is filled.
- PLOT_PLOTTYPE_VECTORS** For a 2D plot, this plot type draws a vector from (x, y) to $(x+x\delta, y+y\delta)$. Thus it requires four columns of data. It also draws a small arrowhead at the end of the vector. A 3D plot of this plot type is similar, but requires six columns of data. It only works for Cartesian 3D plot. The `option` for this plot type is the same as that for the arrow style defined on page 16 for member function `CPlot::arrow()`.
- PLOT_PLOTTYPE_BOXES** The `boxes` plot type is only relevant to 2-d plotting. It draws a box centered about the given x coordinate from the x axis (not the graph border) to the given y coordinate. The width of the box is obtained in one of three ways. If it is a data plot and the data file has a third column, this will be used to set the width of the box. If not, if a width has been set using the member function `CPlot::boxWidth()`, this will be used. If neither of these is available, the width of each box will be calculated automatically so that it touches the adjacent boxes. The interior of the boxes is drawn based on the member function `CPlot::fillStyle()`. By default, the box is filled with the background color.
- PLOT_PLOTTYPE_BOXERRORBARS** The `boxerrorbars` plot type is only relevant to 2-d data plotting. It is a combination of the `boxes` **PLOT_PLOTTYPE_BOXES** and `yerrorbars` **PLOT_PLOTTYPE_YERRORBARS** plot types. The `boxwidth` will come from the fourth column if the y errors are in the form of "ydelta" and the `boxwidth` was not previously set equal to -2.0 by the member function `CPlot::boxWidth()` or from the fifth column if the y errors are in the form of "y_{low} y_{high}". The special case `CPlot::boxWidth(-2.0)` is for four-column data with y errors in the form "y_{low} y_{high}". In this case the `boxwidth` will be calculated so that each box touches the adjacent boxes. The width will also be calculated in cases where three-column data are used. The box height is determined from the y error in the same way as it is for the `yerrorbars` style—either from y-delta to y-delta or from y_{low} to

yhigh, depending on how many data columns are provided. The interior of the boxes is drawn based on the specification by **CPlot::fillStyle()**.

PLOT_PLOTTYPE_BOXXYERRORBARS The `boxxyerrorbars` plot type is only relevant to 2-d data plotting. It is a combination of the boxes **PLOT_PLOTTYPE_BOXES** and `xyerrorbars` **PLOT_PLOTTYPE_XYERRORBARS** plot types. The box width and height are determined from the x and y errors in the same way as they are for the `xyerrorbars` plot type — either from `xlow` to `xhigh` and from `ylow` to `yhigh`, or from `x-xdelta` to `x+xdelta` and from `y-ydelta` to `y+ydelta`, depending on how many data columns are provided. The interior of the boxes is drawn based on the specification by **CPlot::fillStyle()**.

PLOT_PLOTTYPE_CANDLESTICKS The `candlesticks` plot type can be used for 2-d data plotting of financial data or for generating box-and-whisker plots of statistical data. Five columns of data are required; in order, these should be the x coordinate (most likely a date) and the opening, low, high, and closing prices. The symbol is a rectangular box, centered horizontally at the x coordinate and limited vertically by the opening and closing prices. A vertical line segment at the x coordinate extends up from the top of the rectangle to the high price and another down to the low. The vertical line will be unchanged if the low and high prices are interchanged.

The width of the rectangle can be controlled by the member function **CPlot::boxWidth()**.

By default the vertical line segments have no crossbars at the top and bottom. If you want crossbars, which are typically used for box-and-whisker plots, then add the keyword `whiskerbars` to the `option` parameter of the function. By default these whiskerbars extend the full horizontal width of the candlestick, but you can modify this by specifying a fraction of the full width.

By default the rectangle is empty if (`open > close`), and filled with three vertical bars if (`close > open`). If filled-boxes support is present, then the rectangle can be colored by the member function **CPlot::fillStyle()**.

Note: To place additional symbols, such as the median value, on a box-and-whisker plot requires additional function call as shown in the example below.

```
plot.dataFile("candlesticks.dat", "using 1:3:2:6:5");
plot.plotType(PLOT_PLOTTYPE_CANDLESTICKS, 0,
              "linetype 1 linewidth 2 whiskerbars 0.5");
plot.dataFile("candlesticks.dat", "using 1:4:4:4:4");
plot.plotType(PLOT_PLOTTYPE_CANDLESTICKS, 1, "linetype -1 linewidth 2");
```

It assumed that the data in file `candlesticks.dat` contains the following entries

#	X	Min	1stQuartile	Median	3rdQuartile	Max
1	1.5	2		2.4	4	6.
2	1.5	3		3.5	4	5.5
3	4.5	5		5.5	6	6.5
...						

The plot will have crossbars on the whiskers and crossbars are 50% of full width.

PLOT_PLOTTYPE_FINANCEBARS The `financebars` plot type is only relevant for 2-d data plotting of financial data. Five columns of data are required; in order, these should be the x coordinate (most likely a date) and the opening, low, high, and closing prices. The symbol is a vertical line segment, located horizontally at the x coordinate and limited vertically by the high

and low prices. A horizontal tic on the left marks the opening price and one on the right marks the closing price. The length of these tics may be changed by **CPlot::barSize()**. The symbol will be unchanged if the high and low prices are interchanged.

PLOT_PLOTTYPE_XERRORBARS The `xerrorbars` plot type is only relevant to 2D data plotting. The `xerrorbars` is like `dots`, except that a horizontal error bar is also drawn. At each point (x,y) , a line is drawn from (x_{low},y) to (x_{high},y) or from $(x-x_{delta},y)$ to $(x+x_{delta},y)$, depending on how many data columns are provided. A tic mark is placed at the ends of the error bar (unless **CPlot::barSize()** is called).

PLOT_PLOTTYPE_XYERRORBARS The `xyerrorbars` plot type is only relevant to 2D data plotting. The `xyerrorbars` is like `dots`, except that horizontal and vertical error bars are also drawn. At each point (x,y) , lines are drawn from $(x,y-y_{delta})$ to $(x,y+y_{delta})$ and from $(x-x_{delta},y)$ to $(x+x_{delta},y)$ or from (x,y_{low}) to (x,y_{high}) and from (x_{low},y) to (x_{high},y) , depending upon the number of data columns provided. A tic mark is placed at the ends of the error bar (unless **CPlot::barSize()** is called).

If data in a file are provided in an unsupported mixed form, the option `using filter` for **CPlot::dataFile()** should be used to set up the appropriate form. For example, if the data are of the form $(x,y,x_{delta},y_{low},y_{high})$, then you can use

```
plot.dataFile("datafile", "using 1:2:($1-$3):($1+$3):4:5");
plot.plotType(PLOT_PLOTTYPE_XYERRORBARS, plot.dataSetNum());
```

PLOT_PLOTTYPE_YERRORBARS The `yerrorbars` (or `errorbars`) plot type is only relevant to 2D data plotting. The `yerrorbars` is like `points`, except that a vertical error bar is also drawn. At each point (x,y) , a line is drawn from $(x,y-y_{delta})$ to $(x,y+y_{delta})$ or from (x,y_{low}) to (x,y_{high}) , depending on how many data columns are provided. A tic mark is placed at the ends of the error bar (unless **CPlot::barSize()** is called).

PLOT_PLOTTYPE_XERRORLINES The `xerrorlines` plot type is only relevant to 2D data plotting. The `xerrorlines` is like `linespoints`, except that a horizontal error line is also drawn. At each point (x,y) , a line is drawn from (x_{low},y) to (x_{high},y) or from $(x-x_{delta},y)$ to $(x+x_{delta},y)$, depending on how many data columns are provided. A tic mark is placed at the ends of the error bar (unless **CPlot::barSize()** is called).

PLOT_PLOTTYPE_XYERRORLINES The `xyerrorlines` plot type is only relevant to 2D data plotting. The `xyerrorlines` is like `linespoints`, except that a horizontal and vertical error lines are also drawn. At each point (x,y) , lines are drawn from $(x,y-y_{delta})$ to $(x,y+y_{delta})$ and from $(x-x_{delta},y)$ to $(x+x_{delta},y)$ or from (x,y_{low}) to (x,y_{high}) and from (x_{low},y) to (x_{high},y) , depending upon the number of data columns provided. A tic mark is placed at the ends of the error bar (unless **CPlot::barSize()** is called).

If data in a file are provided in an unsupported mixed form, the option `using filter` for **CPlot::dataFile()** should be used to set up the appropriate form. For example, if the data are of the form $(x,y,x_{delta},y_{low},y_{high})$, then you can use

```
plot.dataFile("datafile", "using 1:2:($1-$3):($1+$3):4:5");
plot.plotType(PLOT_PLOTTYPE_XYERRORLINES, plot.dataSetNum());
```

PLOT_PLOTTYPE_YERRORLINES The `yerrorlines` (or `errorlines`) plot type is only relevant to 2D data plotting. The `yerrorlines` is like `linespoints`, except that a vertical error line is also drawn. At each point (x,y) , a line is drawn from $(x,y-y_{delta})$ to $(x,y+y_{delta})$ or from (x,y_{low}) to (x,y_{high}) , depending on how many data columns are provided. A tic mark is placed at the ends of the error bar (unless **CPlot::barSize()** is called).

num The data set to which the plot type applies.

line_type An integer index representing the line type for drawing. Use the same value for different curves so that each curve with the same color and style.

line_width A scaling factor for the line width. The line width is *line_width* multiplied by the default width.

point_type An integer index representing the desired point type.

point_size A scaling factor for the size of the point used. The point size is *point_size* multiplied by the default size.

option An option string for a plot type to fine tune the plot.

1. The *option* for the plot type **PLOT_PLOTTYPE_VECTORS** is the same as that for the arrow style defined on page 16 for member function **CPlot::arrow()**.
2. The *option* for the plot type **PLOT_PLOTTYPE_CANDLESTICKS** is as follows.

```
{ {linetype | lt <line_type>}
  {linewidth | lw <line_width>}
  {whiskerbars [fraction_value]}
}
```

It specifies line type and line width. The *whiskerbars* extend the full horizontal width of the candlestick. The optional *fraction_value* in the range [0, 1.0] specifies a fraction of the full width of whiskerbars.

3. The *option* for the plot type **PLOT_PLOTTYPE_FILLEDCURVES** is as follows.

```
{ [closed | {above | below} {x1 | x2 | y1 | y2}[=<a>] | xy=<x>,<y>]
  {linetype | lt <line_type>}
}
```

The plot variants for **PLOT_PLOTTYPE_FILLEDCURVES** can be further modified by the options

```
closed    ... just filled closed curve,
x1        ... x1 axis,
x2        ... x2 axis, etc for y1 and y2 axes,
y1=0     ... line y=0 (at y1 axis) ie parallel to x1 axis,
y2=42    ... line y=42 (at y2 axis) ie parallel to x2, etc,
xy=10,20 ... point 10,20 of x1,y1 axes (arc-like shape).
```

An example of filling the area between two input curves using three columns of data is as follows.

```
plot.dataFile("datafile", "using 1:2:3");
plot.plotType(PLOT_PLOTTYPE_FILLEDCURVES, plot.dataSetNum());
```

The above and below in the form

```
above {x1|x2|y1|y2}=<val>
below {x1|x2|y1|y2}=<val>
```

limit the filled area to one side of the bounding line or curve.

If the values of <a>, <x>, <y> are out of the drawing boundary, then they are moved to the graph boundary. Then the actually filled area in the case of option `xy=<x>, <y>` will depend on the x-range and y-range.

The optional fourth argument `color` to change the color for the filled area. For example,

```
plot.plotType(PLOT_PLOTTYPE_FILLEDCURVES, num, "y1=0", "blue");
```

The detailed color specification can be found in the member function **CPlot::strokeColor()**.

Alternatively, the option `linetype` in the third argument can be used to change the color for the filled area. For example,

```
plot.plotType(PLOT_PLOTTYPE_FILLEDCURVES, num, "y1=0 linetype 1");
```

Description

Set the desired plot type for a previously added data set. For 3D plots, only **PLOT_PLOTTYPE_LINES**, **PLOT_PLOTTYPE_IMPULSES**, **PLOT_PLOTTYPE_POINTS**, and **PLOT_PLOTTYPE_LINESPOINTS** are valid. If other types are specified, **PLOT_PLOTTYPE_POINTS** is used.

Some 2D plot types need data with more than two columns. If the data are in a file, the `using` option of **CPlot::dataFile()** can be used to set up the correct columns for the plot type you want. In this discussion, “column” will be used to refer to a column in a data file, an entry in the `using` list, or a column in a two-dimensional array. The data for plotting in a two-dimensional array can be added to an instance of **CPlot** class by the member function **CPlot::data()**.

For three columns data, only `xerrorbars`, `yerrorbars`, `xerrorlines`, `yerrorlines`, `boxes`, `boxerrorbars`, and `filledcurves` are allowed. If other plot type is used, the type will be changed to `yerrorbars`.

For four columns, only `xerrorbars`, `yerrorbars`, `xyerrorbars`, `xerrorlines`, `yerrorlines`, `xyerrorlines`, `boxxyerrorbars`, and `boxerrorbars` are allowed. An illegal plot type will be changed to `yerrorbars`.

Five-column data allow only the `boxerrorbars`, `financebars`, and `candlesticks` plot types. An illegal style will be changed to `boxerrorbars` before plotting.

Six- and seven-column data only allow the `xyerrorbars`, `xyerrorlines`, and `boxxyerrorbars` plot types. Illegal styles will be changed to `xyerrorbars` before plotting.

Numbering of the data sets starts with zero. New plot types replace previously specified types. The line style and/or marker type for the plot are selected automatically, unless the appropriate combination of `line_type`, `line_width`, `point_type`, and `point_size` are specified. The `line_type` is an optional argument specifying an index for the line type used for drawing the line. The line type varies depending on the terminal type used (see **CPlot::outputType**). Typically, changing the line type will change the color of the line or make it dashed or dotted. All terminals support at least six different line types. By default, the line type is 1. The `line_width` is an optional argument used to specify the line width. The line width is `line_width` multiplied by the default width. Typically the default width is one pixel. `point_type` is an optional argument used to change the appearance (color and/or marker type) of a point. It is specified with an integer representing the index of the desired point type. All terminals support at least six different point types. `point_size` is an optional argument used to change the size of the point. The point size is `point_size` multiplied by the default size. If `point_type` and `point_size` are set to zero or a negative number, a default value is used.

Portability

The `line_width` and `point_size` options is not supported by all terminal types.

For 3D plots on some systems with output type set to `postscript` (see `CPlot::outputType()`), data may not be displayed for `PLOT_PLOTTYPE_DOTS`.

Example 1

This example shows some of the different point types for the default X-window and the `postscript` terminal types (see `CPlot::outputType`). In this example the points have a point size of five times the default. The appearance of points for different terminal types may be different.

```

/* File: plotType4.ch */
#include <chplot.h>

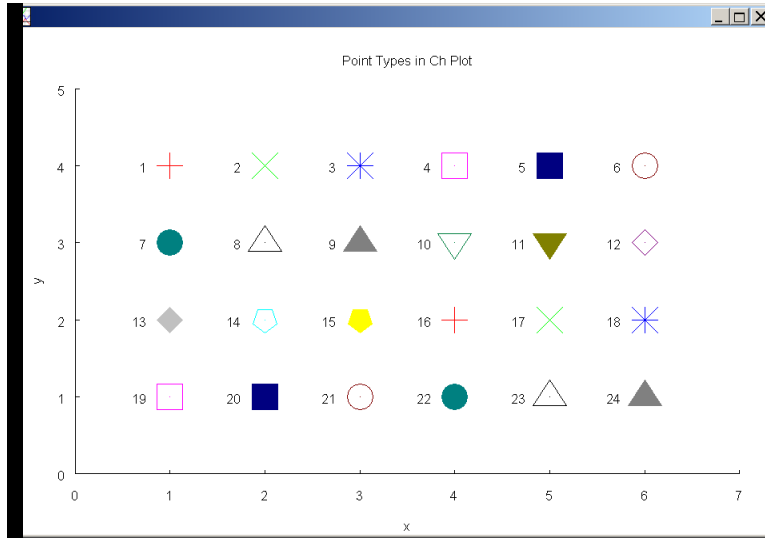
int main() {
    double x, y;
    string_t text;
    int datasetnum=0, point_type = 1, point_size = 5;
    class CPlot plot;

    plot.axisRange(PLOT_AXIS_X, 0, 7);
    plot.axisRange(PLOT_AXIS_Y, 0, 5);
    plot.title("Point Types in Ch Plot");
    for (y = 4; y >= 1; y--) {
        for (x = 1; x <= 6; x++) {
            sprintf(text, "%d", point_type);
            plot.point(x, y, 0);
            plot.plotType(PLOT_PLOTTYPE_POINTS, datasetnum);
            plot.pointType(datasetnum, point_type, point_size);
            plot.text(text, PLOT_TEXT_RIGHT, x-.25, y, 0);
            datasetnum++; point_type++;
        }
    }
    plot.plotting();
}

```

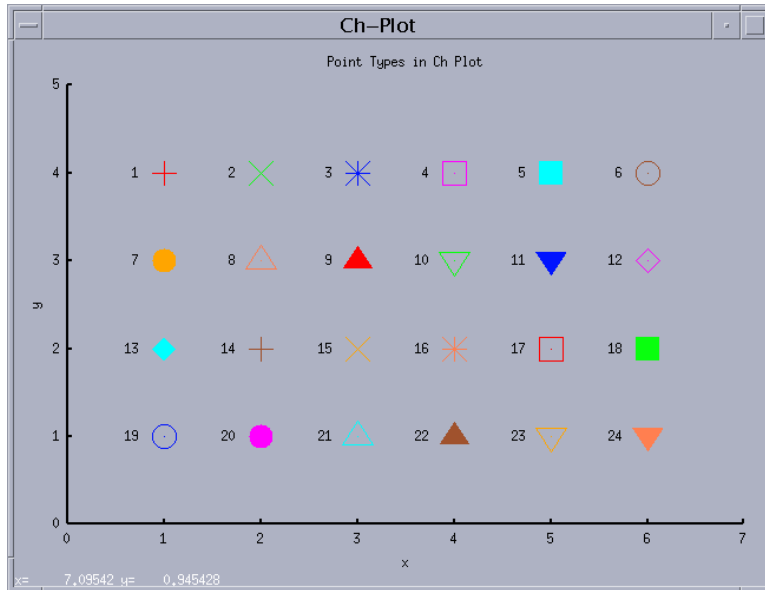
Output

Output displayed in Window.



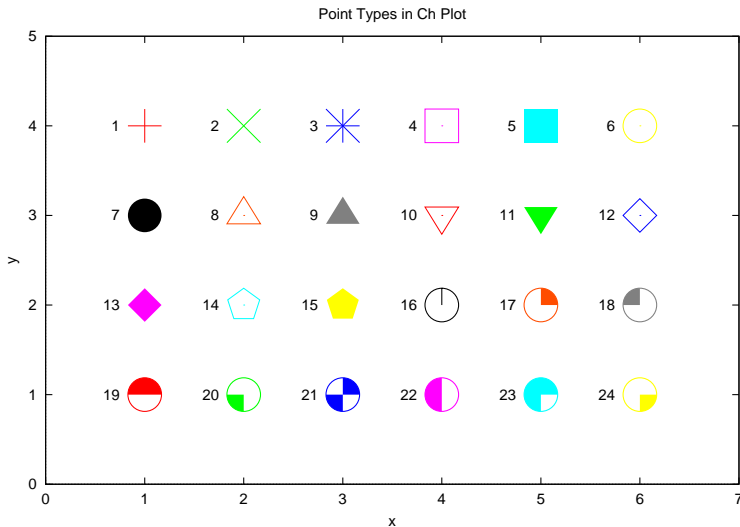
Output

Output displayed in X-window.

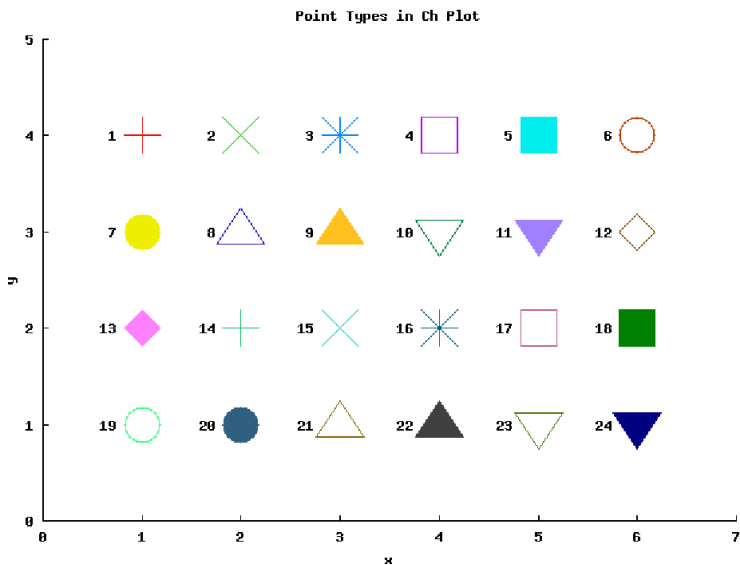


Output

Output as a postscript file.



Output
Output as a PNG file.



Example 2
This example shows some of the different line types for the `postscript` terminal type (see `CPlot::outputType`). The appearance of lines for different terminal types may be different.

```

/* File: plotType5.ch */
#include <chplot.h>

int main() {
    double x, y, xx[2], yy[2];
    string_t text;
    int line_type = -1, line_width = 2, datasetnum = 0;
    class CPlot plot;

```

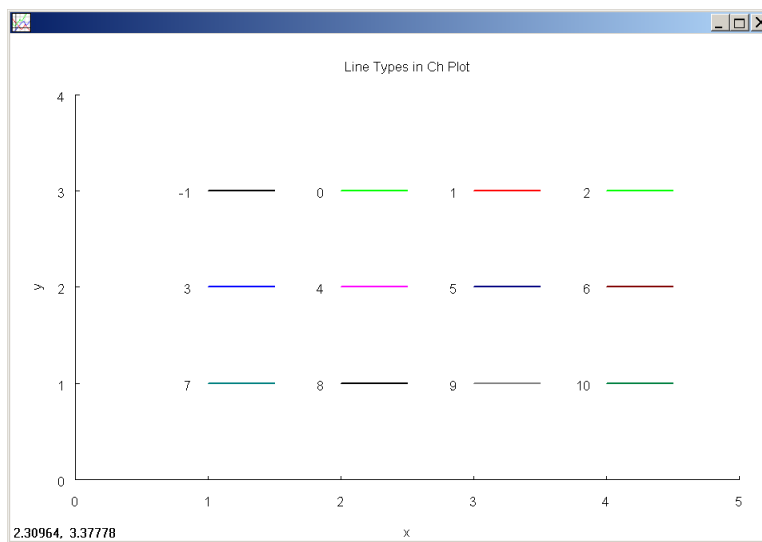
```

plot.axisRange(PLOT_AXIS_X, 0, 5);
plot.axisRange(PLOT_AXIS_Y, 0, 4);
plot.ticsRange(PLOT_AXIS_Y, 1, 0, 4);
plot.title("Line Types in Ch Plot");
for (y = 3; y >= 1; y--) {
    for (x = 1; x <= 4; x++) {
        sprintf(text, "%d", line_type);
        lindata(x, x+.5, xx);
        lindata(y, y, yy);
        plot.data2D(xx, yy);
        plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum);
        plot.lineType(datasetnum, line_type, line_width);
        plot.text(text, PLOT_TEXT_RIGHT, x-.125, y, 0);
        datasetnum++;
        line_type++;
    }
}
plot.plotting();
}

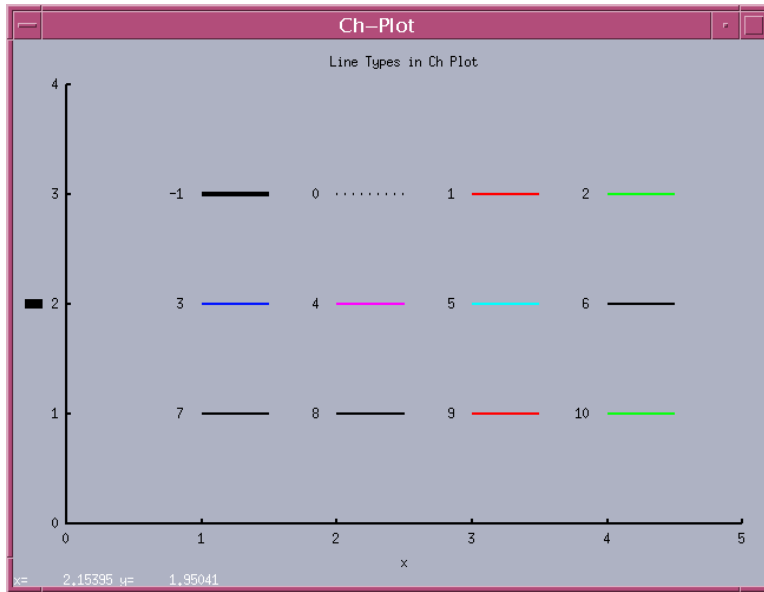
```

Output

Output displayed in Window.

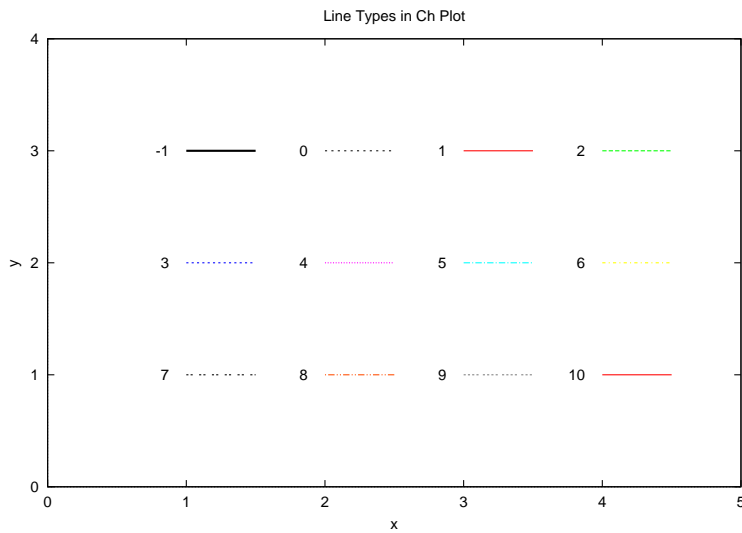
**Output**

Output displayed in X-window.



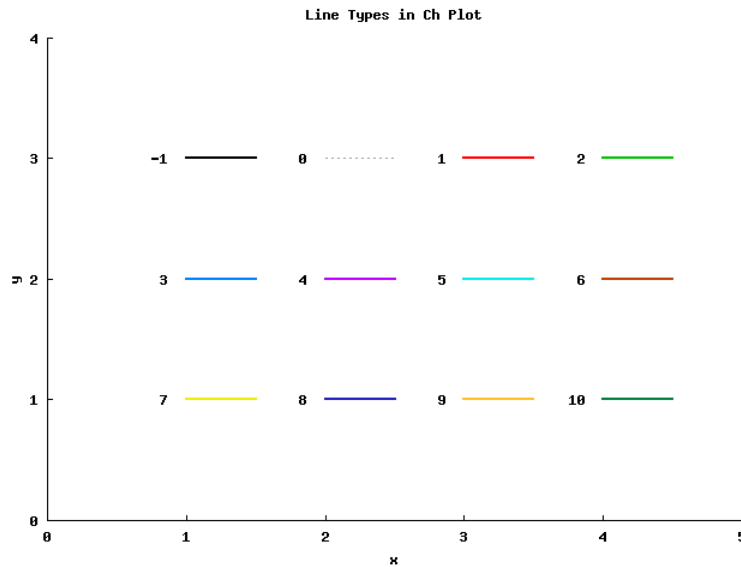
Output

Output as a postscript file.



Output

Output as a PNG file.

**Example 3**

This example shows some of the different point sizes for the `postscript` terminal type (see `CPlot::outputType`). The appearance of points for different terminal types may be different.

```

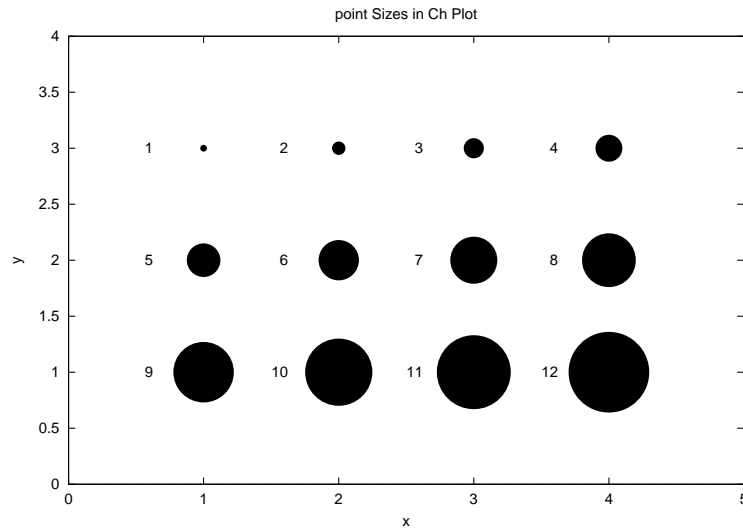
/* File: plotType6.ch */
#include <chplot.h>

int main() {
    double x, y;
    string_t text;
    int datasetnum=0, point_type = 7, point_size = 1;
    class CPlot plot;

    plot.axisRange(PLOT_AXIS_X, 0, 5);
    plot.axisRange(PLOT_AXIS_Y, 0, 4);
    plot.title("point Sizes in Ch Plot");
    for (y = 3; y >= 1; y--) {
        for (x = 1; x <= 4; x++) {
            sprintf(text, "%d", point_size);
            plot.point(x, y, 0);
            plot.plotType(PLOT_PLOTTYPE_POINTS, datasetnum);
            plot.pointType(datasetnum, point_type, point_size);
            plot.text(text, PLOT_TEXT_RIGHT, x-.375, y, 0);
            datasetnum++; point_size++;
        }
    }
    plot.plotting();
}

```

Output

**Example 4**

This example shows some of the different line sizes for the `postscript` terminal type (see `CPlot::outputType`). The appearance of lines for different terminal types may be different.

```

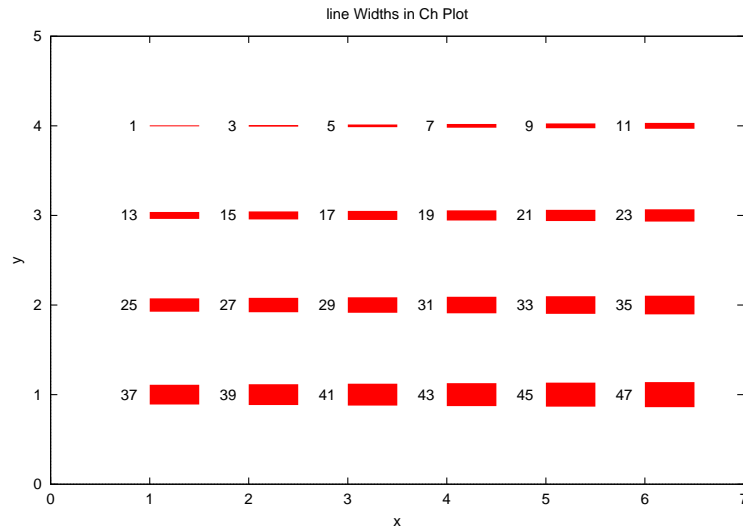
/* File: plotType7.ch */
#include <chplot.h>

int main() {
    double x, y, xx[2], yy[2];
    string_t text;
    int line_type = 1, line_width = 1, datasetnum = 0;
    class CPlot plot;

    plot.axisRange(PLOT_AXIS_X, 0, 7);
    plot.axisRange(PLOT_AXIS_Y, 0, 5);
    plot.title("line Widths in Ch Plot");
    for (y = 4; y >= 1; y--) {
        for (x = 1; x <= 6; x++) {
            sprintf(text, "%d", line_width);
            lndata(x, x+.5, xx);
            lndata(y, y, yy);
            plot.data2D(xx, yy);
            plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum);
            plot.lineType(datasetnum, line_type, line_width);
            plot.text(text, PLOT_TEXT_RIGHT, x-.125, y, 0);
            datasetnum++;
            line_width+=2;
        }
    }
    plot.plotting();
}

```

Output

**Example 5**

```

/* File: plotType1.ch */
#include <chplot.h>
#include <math.h>

int main() {
    class CPlot subplot, *spl;
    array double x1[30], y1[30];
    int line_type = 1, line_width = 1;
    int point_type = 7, point_size = 1;

    lindata(-M_PI, M_PI, x1);
    y1 = sin(x1);
    subplot.subplot(3,3);
    spl = subplot.getSubplot(0,0);
    spl->data2D(x1, y1);
    spl->axisRange(PLOT_AXIS_Y, -1, 1);
    spl->ticsRange(PLOT_AXIS_Y, 0.5, -1, 1);
    spl->plotType(PLOT_PLOTTYPE_LINES, 0);
    spl->label(PLOT_AXIS_XY, NULL);
    spl->title("PLOT_PLOTTYPE_LINES");
    spl = subplot.getSubplot(0,1);
    spl->data2D(x1, y1);
    spl->axisRange(PLOT_AXIS_Y, -1, 1);
    spl->ticsRange(PLOT_AXIS_Y, 0.5, -1, 1);
    spl->plotType(PLOT_PLOTTYPE_IMPULSES, 0);
    spl->label(PLOT_AXIS_XY, NULL);
    spl->title("PLOT_PLOTTYPE_IMPULSES");
    spl = subplot.getSubplot(0,2);
    spl->data2D(x1, y1);
    spl->axisRange(PLOT_AXIS_Y, -1, 1);
    spl->ticsRange(PLOT_AXIS_Y, 0.5, -1, 1);
    spl->plotType(PLOT_PLOTTYPE_STEPS, 0);
    spl->label(PLOT_AXIS_XY, NULL);
    spl->title("PLOT_PLOTTYPE_STEPS");
    spl = subplot.getSubplot(1,0);
    spl->data2D(x1, y1);
    spl->axisRange(PLOT_AXIS_Y, -1, 1);
    spl->ticsRange(PLOT_AXIS_Y, 0.5, -1, 1);

```

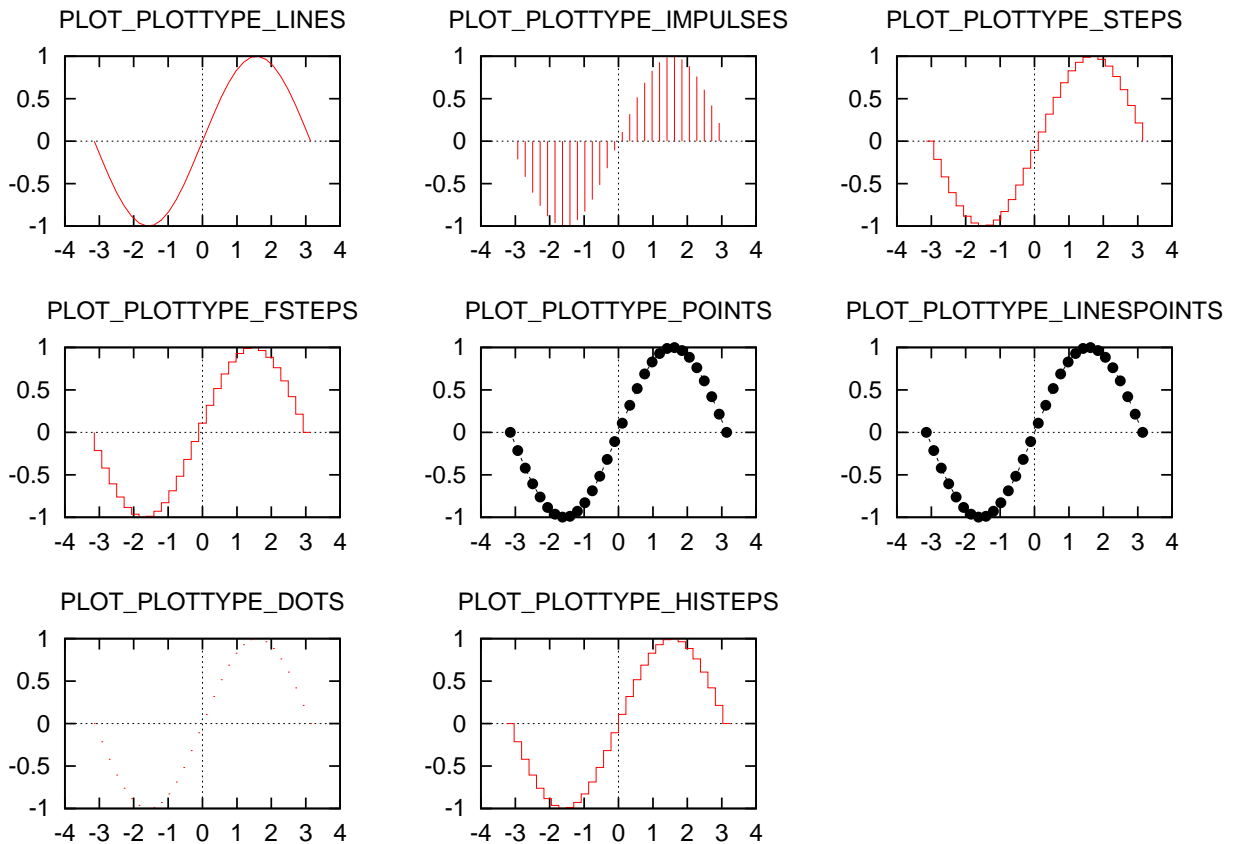


```

spl->plotType(PLOT_PLOTTYPE_FSTEPS, 0);
spl->label(PLOT_AXIS_XY, NULL);
spl->title("PLOT_PLOTTYPE_FSTEPS");
spl = subplot.getSubplot(1,1);
spl->data2D(x1, y1);
spl->axisRange(PLOT_AXIS_Y, -1, 1);
spl->ticsRange(PLOT_AXIS_Y, 0.5, -1, 1);
spl->plotType(PLOT_PLOTTYPE_POINTS, 0);
spl->pointType(0, point_type, point_size);
spl->label(PLOT_AXIS_XY, NULL);
spl->title("PLOT_PLOTTYPE_POINTS");
spl = subplot.getSubplot(1,2);
spl->data2D(x1, y1);
spl->axisRange(PLOT_AXIS_Y, -1, 1);
spl->ticsRange(PLOT_AXIS_Y, 0.5, -1, 1);
spl->plotType(PLOT_PLOTTYPE_LINESPOINTS, 0);
spl->lineType(0, line_type, line_width);
spl->pointType(0, point_type, point_size);
spl->label(PLOT_AXIS_XY, NULL);
spl->title("PLOT_PLOTTYPE_LINESPOINTS");
spl = subplot.getSubplot(2,0);
spl->data2D(x1, y1);
spl->axisRange(PLOT_AXIS_Y, -1, 1);
spl->ticsRange(PLOT_AXIS_Y, 0.5, -1, 1);
spl->plotType(PLOT_PLOTTYPE_DOTS, 0);
spl->label(PLOT_AXIS_XY, NULL);
spl->title("PLOT_PLOTTYPE_DOTS");
spl = subplot.getSubplot(2,1);
spl->data2D(x1, y1);
spl->axisRange(PLOT_AXIS_Y, -1, 1);
spl->ticsRange(PLOT_AXIS_Y, 0.5, -1, 1);
spl->plotType(PLOT_PLOTTYPE_HISTEPS, 0);
spl->label(PLOT_AXIS_XY, NULL);
spl->title("PLOT_PLOTTYPE_HISTEPS");
subplot.plotting();
}

```

Output

**Example 6**

```

/* File: plotType2.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 360,          // number of data points
        numpts=10;              // number of data points
    array double x0[numpoints], y0[numpoints];
    array double x1[numpoints], y1[numpoints];
    array double x2[numpts], y2[numpts];
    array double x3[numpoints], y3[numpoints];
    int line_type=1, line_width =5;
    class CPlot plot;

    lindata(0, 360, x0);        // assign x0 with -10 <= x0 <= 360 linearly
    y0 = sin(x0*M_PI/180);      // array y0 is sine of array x0, element-wise
    lindata(0, 90, x1);        // assign x1 with 0 <= x1 <= 90 linearly
    y1 = sin(x1*M_PI/180);      // array y1 is sine of array x1, element-wise
    lindata(90, 180, x2);      // assign x2 with 90 <= x2 <= 180 linearly
    y2 = sin(x2*M_PI/180);      // array y2 is sine of array x2, element-wise
    lindata(270, 360, x3);     // assign x3 with 90 <= x3 <= 180 linearly
    y3 = sin(x3*M_PI/180);      // array y3 is sine of array x3, element-wise
    plot.data2D(x0, y0);
    plot.data2D(x1, y1);
    plot.data2D(x2, y2);
    plot.data2D(x3, y3);
}

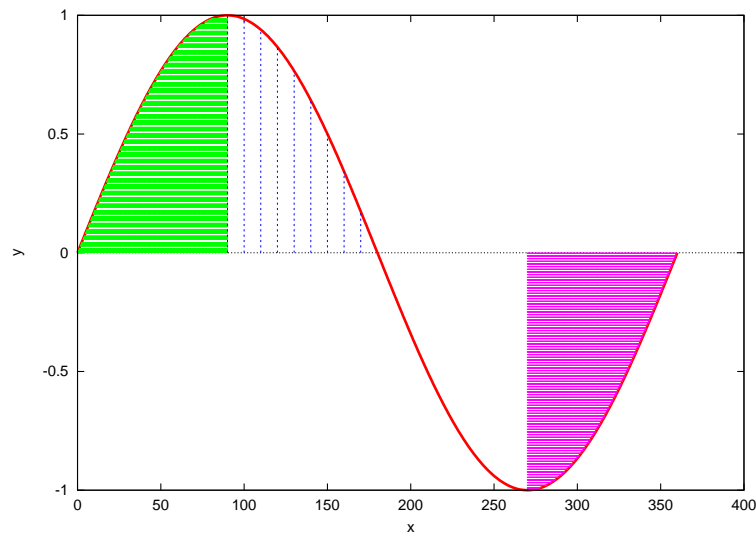
```

```

plot.plotType(PLOT_PLOTTYPE_LINES, 0);           // line for (x,y)
plot.lineType(0, line_type, line_width);
plot.plotType(PLOT_PLOTTYPE_IMPULSES, 1);       // impulse plot for (x1, y1)
plot.plotType(PLOT_PLOTTYPE_IMPULSES, 2);       // impulse plot for (x2, y2)
plot.plotType(PLOT_PLOTTYPE_IMPULSES, 3);       // impulse plot for (x3, y3)
plot.plotting();                                 // get the plotting job done
}

```

Output



Example 7

```

/* File: plotType3.ch */
#include <chplot.h>
#include <math.h>

int main() {
    int line_type = 1, line_width = 1;
    int point_type = 7, point_size = 1;
    double x[16], y[16], z[256];
    double r;
    int i, j;
    class CPlot subplot, *spl;

    lindata(-10, 10, x);
    lindata(-10, 10, y);
    for(i=0; i<16; i++) {
        for(j=0; j<16; j++) {
            r = sqrt(x[i]*x[i]+y[j]*y[j]);
            z[16*i+j] = sin(r)/r;
        }
    }
    subplot.subplot(2,3);
    spl = subplot.getSubplot(0,0);
    spl->data3D(x, y, z);
    spl->axisRange(PLOT_AXIS_Z, -.4, 1.2);
    spl->ticsRange(PLOT_AXIS_Z, .4, -.4, 1.2);
    spl->axisRange(PLOT_AXIS_XY, -10, 10);
    spl->ticsRange(PLOT_AXIS_XY, 5, -10, 10);
}

```

```

spl->plotType(PLOT_PLOTTYPE_LINES, 0);
spl->colorBox(PLOT_OFF);
spl->label(PLOT_AXIS_XYZ, NULL);
spl->title("PLOT_PLOTTYPE_LINES");

spl = subplot.getSubplot(0,1);
spl->data3D(x, y, z);
spl->axisRange(PLOT_AXIS_Z, -.4, 1.2);
spl->ticsRange(PLOT_AXIS_Z, .4, -.4, 1.2);
spl->axisRange(PLOT_AXIS_XY, -10, 10);
spl->ticsRange(PLOT_AXIS_XY, 5, -10, 10);
spl->plotType(PLOT_PLOTTYPE_IMPULSES, 0);
spl->colorBox(PLOT_OFF);
spl->label(PLOT_AXIS_XYZ, NULL);
spl->title("PLOT_PLOTTYPE_IMPULSES");

spl = subplot.getSubplot(0,2);
spl->data3D(x, y, z);
spl->axisRange(PLOT_AXIS_Z, -.4, 1.2);
spl->ticsRange(PLOT_AXIS_Z, .4, -.4, 1.2);
spl->axisRange(PLOT_AXIS_XY, -10, 10);
spl->ticsRange(PLOT_AXIS_XY, 5, -10, 10);
spl->plotType(PLOT_PLOTTYPE_POINTS, 0);
spl->pointType(0, point_type, point_size);
spl->colorBox(PLOT_OFF);
spl->label(PLOT_AXIS_XYZ, NULL);
spl->title("PLOT_PLOTTYPE_POINTS");

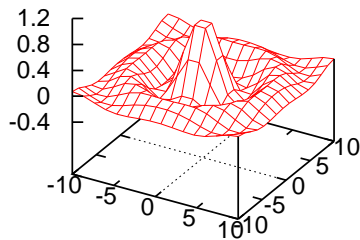
spl = subplot.getSubplot(1,0);
spl->data3D(x, y, z);
spl->axisRange(PLOT_AXIS_Z, -.4, 1.2);
spl->ticsRange(PLOT_AXIS_Z, .4, -.4, 1.2);
spl->axisRange(PLOT_AXIS_XY, -10, 10);
spl->ticsRange(PLOT_AXIS_XY, 5, -10, 10);
spl->plotType(PLOT_PLOTTYPE_LINESPOINTS, 0);
spl->lineType(0, line_type, line_width);
spl->pointType(0, point_type, point_size);
spl->colorBox(PLOT_OFF);
spl->label(PLOT_AXIS_XYZ, NULL);
spl->title("PLOT_PLOTTYPE_LINESPOINTS");

spl = subplot.getSubplot(1,1);
spl->data3D(x, y, z);
spl->axisRange(PLOT_AXIS_Z, -.4, 1.2);
spl->ticsRange(PLOT_AXIS_Z, .4, -.4, 1.2);
spl->axisRange(PLOT_AXIS_XY, -10, 10);
spl->ticsRange(PLOT_AXIS_XY, 5, -10, 10);
spl->plotType(PLOT_PLOTTYPE_SURFACES, 0);
spl->label(PLOT_AXIS_XYZ, NULL);
spl->title("PLOT_PLOTTYPE_SURFACES");
subplot.plotting();
}

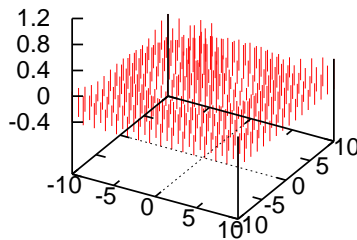
```

Output

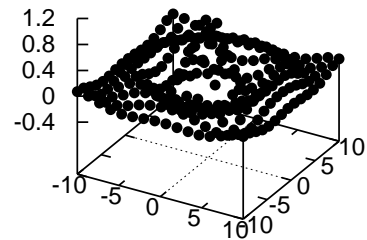
PLOT_PLOTTYPE_LINES



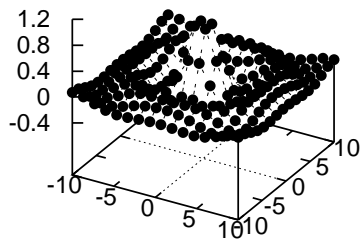
PLOT_PLOTTYPE_IMPULSES



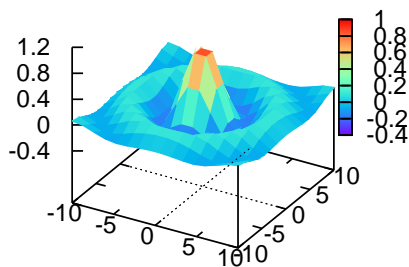
PLOT_PLOTTYPE_POINTS



PLOT_PLOTTYPE_LINESPOINTS



PLOT_PLOTTYPE_SURFACES

**Example 8**

This example illustrates the plot type **PLOT_PLOTTYPE_VECTORS**. The arrow style is defined based on the specification in **CPlot::arrow()** on page 16.

```

/* File: plotType_v.ch */
#include <chplot.h>

char *arrowstyle[] = {
    //"head filled size screen 0.025,30,45 linetype 1 linewidth 1",
    "head filled size screen 0.025,30,45 linetype 1",
    "head nofilled size screen 0.03,15 linetype 3 linewidth 1",
    "head filled size screen 0.03,15,45 linetype 1 linewidth 2",
    "head filled size screen 0.03,15 linetype 3 linewidth 2",
    "heads filled size screen 0.03,15,135 linetype 1 linewidth 3",
    "head empty size screen 0.03,15,135 linetype 3 linewidth 3",
    "nohead linetype 1 linewidth 4",
    "heads size screen 0.008,90 linetype 3 linewidth 4"
};

int main () {
    class CPlot plot;
    int i;

    plot.label(PLOT_AXIS_X, "");
    plot.label(PLOT_AXIS_Y, "");
    plot.axisRange(PLOT_AXIS_X, -1000, 1000);
    plot.axisRange(PLOT_AXIS_Y, -178, 86);
    plot.border(PLOT_BORDER_ALL, PLOT_ON);

```

```

plot.ticsMirror(PLOT_AXIS_XY, PLOT_ON);

for(i=0; i<sizeof(arrowstyle)/sizeof(char*); i++) {
    plot.arrow(-500,-100-10*i,0, 500,-100-10*i,0, arrowstyle[i]);
}
plot.dataFile("arrowstyle.dat", "using 1:2:(0):3");
plot.plotType(PLOT_PLOTTYPE_VECTORS, 0, arrowstyle[0]);
plot.legend("arrow style 0", 0);
plot.plotting();
}

```

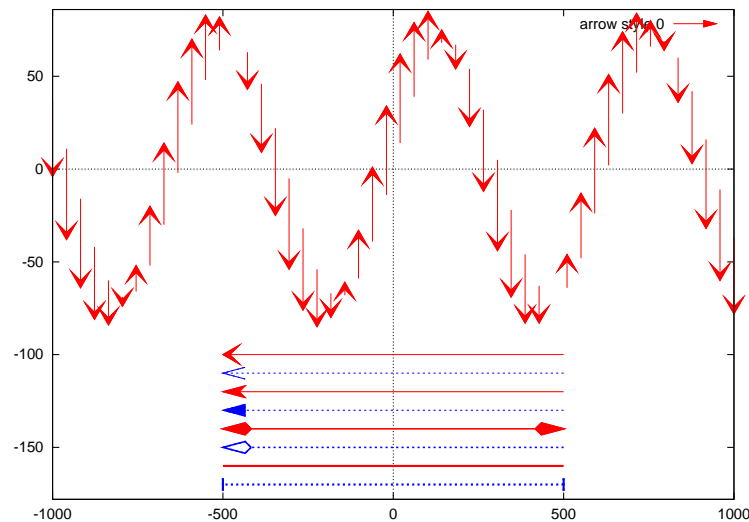
The contents of data file arrowstyle.dat in Example 8

```

-1000    37    -41
-959     11    -49
-918    -16    -48
...

```

Output



Example 9

This example illustrates the plot type **PLOT_PLOTTYPE_CANDLESTICKS**.

```

/* File: plotType_cs.ch */
/* File: plotType_cs.ch to process data in candlesticks.dat */
#include <chplot.h>

int main() {
    class CPlot plot;

    plot.label(PLOT_AXIS_X, "");
    plot.label(PLOT_AXIS_Y, "");
    plot.border(PLOT_BORDER_ALL, PLOT_ON);
    plot.ticsMirror(PLOT_AXIS_XY, PLOT_ON);
    plot.title("box-and-whisker plot adding median value as bar");
    plot.axisRange(PLOT_AXIS_X, 0, 11);
    plot.axisRange(PLOT_AXIS_Y, 0, 10);
    plot.dataFile("candlesticks.dat", "using 1:3:2:6:5");
    plot.plotType(PLOT_PLOTTYPE_CANDLESTICKS, 0, "linetype 1 linewidth 2 whiskerbars");
}

```

```

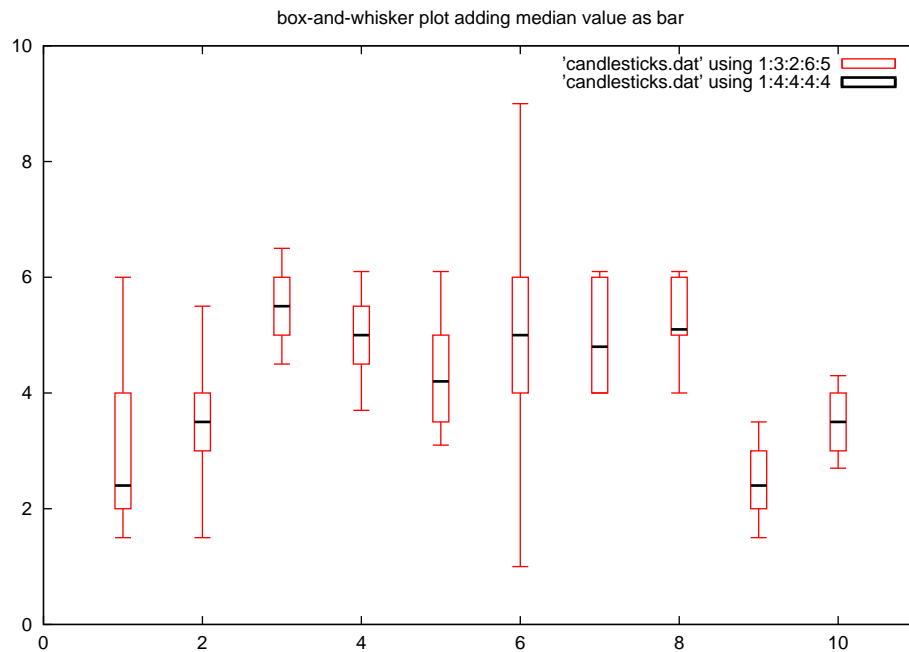
    plot.fillStyle(0, PLOT_FILLSTYLE_EMPTY);
    plot.legend("'candlesticks.dat' using 1:3:2:6:5", 0);
    plot.dataFile("candlesticks.dat", "using 1:4:4:4:4");
    plot.plotType(PLOT_PLOTTYPE_CANDLESTICKS, 1, "linetype -1 linewidth 2");
    plot.legend("'candlesticks.dat' using 1:4:4:4:4", 1);
    plot.boxWidth(0.2);
    plot.plotting();
    return 0;
}

```

The contents of data file candlesticks.dat in Example 9

1	1.5	2	2.4	4	6.
2	1.5	3	3.5	4	5.5
3	4.5	5	5.5	6	6.5
...					

Output



Example 10

This example illustrates the plot type **PLOT_PLOTTYPE_FINANCEBARS**.

```

/* File: plotType_fb.ch */
/* File: plotType_fb.ch to process data in finance.dat

data and indicators in finance.dat
# data file layout:
# date, open, high, low, close, volume,
# 50-day moving average volume, Intraday Intensity,
# 20-day moving average close,
# upper Bollinger Band, lower Bollinger Band
*/

#include <chplot.h>

```

```

int main() {
    class CPlot subplot, *plot1, *plot2;

    subplot.subplot(2,1);
    subplot.title("Finance Bar");
    plot1 = subplot.getSubplot(0,0);
    plot1->label(PLOT_AXIS_X, "");
    plot1->label(PLOT_AXIS_Y, "price");
    plot1->border(PLOT_BORDER_ALL, PLOT_ON);
    plot1->ticsMirror(PLOT_AXIS_XY, PLOT_ON);
    plot1->dataFile("finance.dat", "using 0:2:3:4:5");
    plot1->plotType(PLOT_PLOTTYPE_FINANCEBARS, 0);
    plot1->axisRange(PLOT_AXIS_X, 50, 253);
    plot1->axisRange(PLOT_AXIS_Y, 75, 105);
    plot1->grid(PLOT_ON);
    plot1->ticsPosition(PLOT_AXIS_X, 66, 87, 109, 130, 151, 174, 193, 215, 235);
    plot1->ticsPosition(PLOT_AXIS_Y, 105, 100, 95, 90, 85, 80);
    plot1->ticsFormat(PLOT_AXIS_X, "");
    plot1->scaleType(PLOT_AXIS_Y, PLOT_SCALETYPE_LOG);
    plot1->dataFile("finance.dat", "using 0:9");
    plot1->dataFile("finance.dat", "using 0:10");
    plot1->dataFile("finance.dat", "using 0:11");
    plot1->dataFile("finance.dat", "using 0:8");
    plot1->axes(plot1->dataSetNum(), "x1y2");
    plot1->text("Courtesy of Bollinger Capital", PLOT_TEXT_CENTER, 83, 76.7, 0);
    plot1->text("www.BollingerBands.com", PLOT_TEXT_CENTER, 83, 75.7, 0);
    plot1->size(1, 0.7); // 70% for the top plot
    plot1->margins(9, -1, -1, 0); // left margin is 9, bottom margin is 0
    plot1->origin(0, 0.3); // origin for Y is 0.3
    plot1->text("Acme Widgets", PLOT_TEXT_CENTER, 150, 102, 0);

    plot2 = subplot.getSubplot(1,0);
    plot2->label(PLOT_AXIS_X, "");
    plot2->label(PLOT_AXIS_Y, "volume (x10000)");
    plot2->dataFile("finance.dat", "using 0:($6/10000)");
    plot2->plotType(PLOT_PLOTTYPE_IMPULSES, 0);
    plot2->dataFile("finance.dat", "using 0:($7/10000)");
    plot2->border(PLOT_BORDER_ALL, PLOT_ON);
    plot2->ticsMirror(PLOT_AXIS_XY, PLOT_ON);
    plot2->grid(PLOT_ON);
    plot2->axisRange(PLOT_AXIS_X, 50, 253);
    plot2->ticsRange(PLOT_AXIS_Y, 500);
    plot2->ticsFormat(PLOT_AXIS_Y, "%1.0f");
    plot2->ticsPosition(PLOT_AXIS_X, 66, 87, 109, 130, 151, 174, 193, 215, 235);
    /* use the code below for X-label for C++ or Ch */
    /*
    plot2->ticsLabel(PLOT_AXIS_X, "6/03", 66);
    plot2->ticsLabel(PLOT_AXIS_X, "7/03", 87);
    plot2->ticsLabel(PLOT_AXIS_X, "8/03", 109);
    plot2->ticsLabel(PLOT_AXIS_X, "9/03", 130);
    plot2->ticsLabel(PLOT_AXIS_X, "10/03", 151);
    plot2->ticsLabel(PLOT_AXIS_X, "11/03", 174);
    plot2->ticsLabel(PLOT_AXIS_X, "12/03", 193);
    plot2->ticsLabel(PLOT_AXIS_X, "1/04", 215);
    plot2->ticsLabel(PLOT_AXIS_X, "2/04", 235);
    */
    plot2->size(1, 0.3); // 30% for the bottom plot
    plot2->margins(9, -1, 0, -1); // left margin is 9, top margin is 0

```



```

subplot.plotting();
return 0;
}

```

The contents of data file `finance.dat` in Example 10

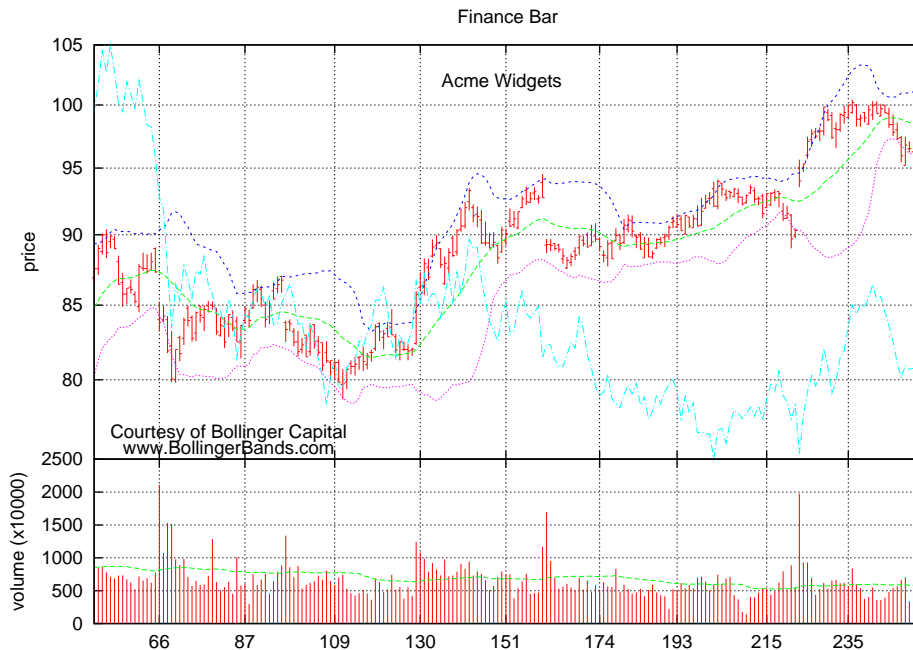
```

# data and indicators in finance.dat
# data file layout:
# date, open, high, low, close, volume,
# 50-day moving average volume, Intraday Intensity,
# 20-day moving average close,
# upper Bollinger Band, lower Bollinger Band
2/27/2003 77.9 78.59 76.75 77.28 9927900 0 -4208566 0 0 0
2/28/2003 78.15 78.47 77 77.95 6556100 0 -2290796 0 0 0
3/3/2003 78.6 79 77.12 77.33 6618300 0 -7430539 0 0 0
...

```

This data file is also used to plot stock prices in Example 14 below.

Output



Example 11

This example illustrates the plot type `PLOT_PLOTTYPE_YERRORBARS`.

```

/* File: plotType_eb.ch */
#include<chplot.h>

int main() {
    CPlot plot;

    plot.title("Errorbar type ");
    plot.label(PLOT_AXIS_X, "Angle (deg)");
    plot.label(PLOT_AXIS_Y, "Amplitude");
}

```

```

plot.border(PLOT_BORDER_ALL, PLOT_ON);
plot.ticsMirror(PLOT_AXIS_XY, PLOT_ON);
plot.legendOption("box");
plot.dataFile("big_peak.dat");
plot.plotType(PLOT_PLOTTYPE_YERRORBARS, 0);
plot.legend("Rate", 0);
plot.dataFile("big_peak.dat");
plot.smooth(1, "csplines");
plot.legend("Average", 1);
plot.barSize(1.0);
plot.plotting();
}

```

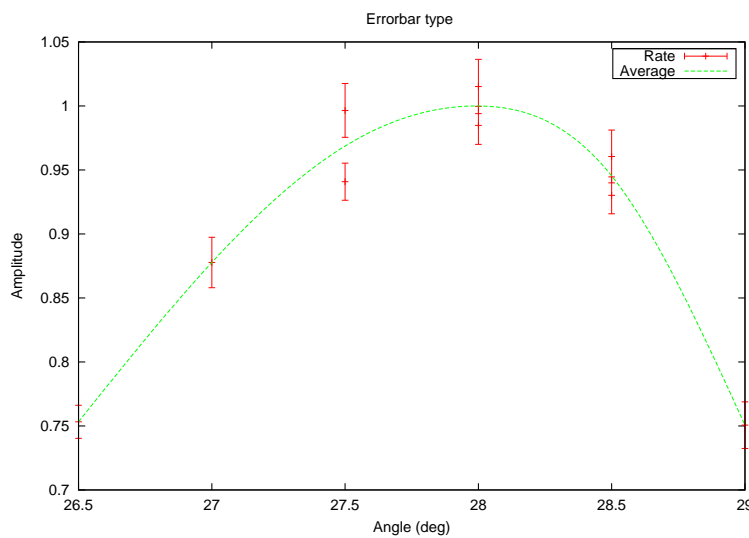
The contents of data file `big_peak.dat` in Example 11

```

26.500000 0.753252 0.012953
27.000000 0.877710 0.019712
27.500000 0.996531 0.021018
...

```

Output



Example 12

This example illustrates the plot type `PLOT_PLOTTYPE_FILLEDCURVES`.

```

/* File: filledcolors.ch */
#include<math.h>
#include<chplot.h>

#define N 100

double func(double x) {
    double y;

    y = 1;
    return y;
}

int main() {

```

```

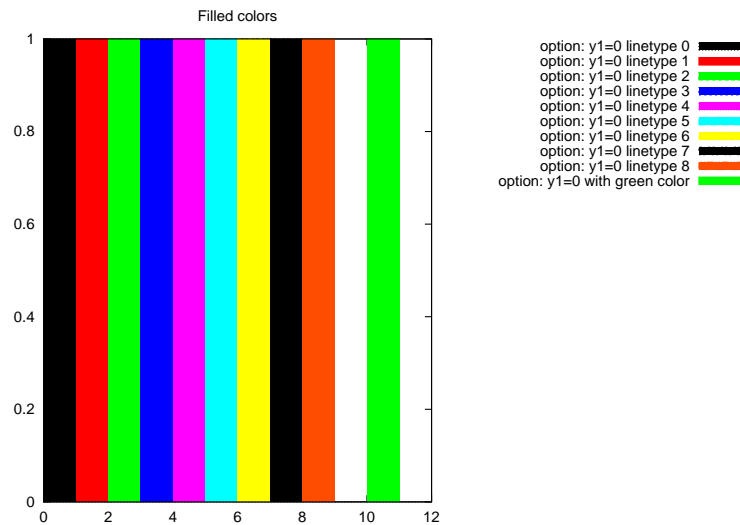
CPlot plot;
double x0, xf;
int i;
char option[64], legend[64];

plot.title("Filled colors");
plot.legendOption("outside");
plot.label(PLOT_AXIS_X, "");
plot.label(PLOT_AXIS_Y, "");
plot.axisRange(PLOT_AXIS_X, 0, 12);
plot.axisRange(PLOT_AXIS_Y, 0, 1);

for(i=0; i<9; i++) {
    x0 = i;
    xf = i+1;
    sprintf(option, "y1=0 linetype %d", i);
    sprintf(legend, "option: %s", option);
    plot.func2D(x0, xf, N, func);
    plot.plotType(PLOT_PLOTTYPE_FILLEDCURVES, i, option);
    plot.legend(legend, i);
}
x0 = i+1;
xf = i+2;
plot.func2D(x0, xf, N, func);
plot.plotType(PLOT_PLOTTYPE_FILLEDCURVES, 9, "y1=0", "green");
plot.legend("option: y1=0 with green color", 9);
plot.plotting();
}

```

Output



Example 13

This example illustrates the plot type `PLOT_PLOTTYPE_FILLEDCURVES`.

```

/* File: filledcolors2.ch */
#include<math.h>
#include<chplot.h>

#define N 100

```

```

double func(double x, void *param) {
    double offset;
    double y;

    offset = *(double*)param;
    y = offset;
    return y;
}

int main() {
    CPlot plot;
    double offset, x0, xf;
    int i;
    char option[64], legend[64];

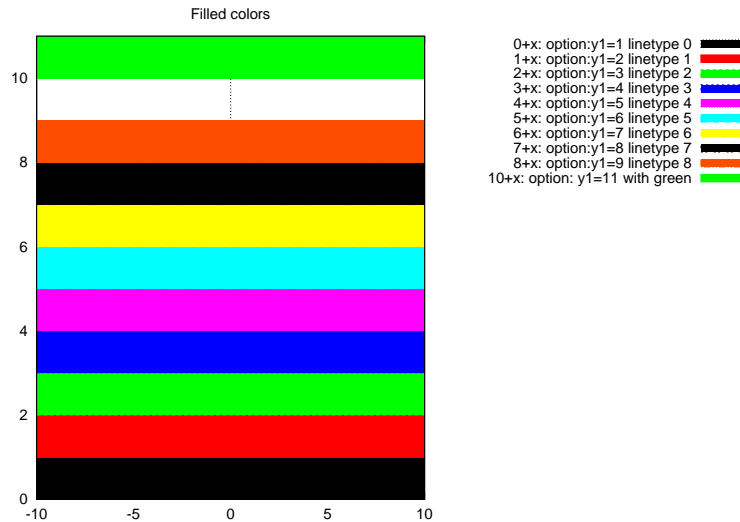
    x0 = -10;
    xf = 10;

    plot.title("Filled colors");
    plot.legendOption("outside");
    plot.label(PLOT_AXIS_X, "");
    plot.label(PLOT_AXIS_Y, "");
    plot.axisRange(PLOT_AXIS_X, -10, 10);
    plot.axisRange(PLOT_AXIS_Y, 0, 11);

    for(i=0; i<9; i++) {
        offset = i;
        sprintf(option, "y1=%d linetype %d", i+1, i);
        sprintf(legend, "%d+x: option:%s", i, option);
        plot.funcp2D(x0, xf, N, func, &offset);
        plot.plotType(PLOT_PLOTTYPE_FILLEDCURVES, i, option);
        plot.legend(legend, i);
    }
    offset = 10;
    plot.funcp2D(x0, xf, N, func, &offset);
    plot.plotType(PLOT_PLOTTYPE_FILLEDCURVES, 9, "y1=11", "green");
    plot.legend("10+x: option: y1=11 with green", 9);
    plot.plotting();
}

```

Output

**Example 14**

This example illustrates the plot type `PLOT_PLOTTYPE_FILLEDCURVES`. This plot for stock prices uses the same data file in Example 10.

```

/* File: stockprice.ch */
/* File: finance.ch to process data in finance.dat

data and indicators in finance.dat
# data file layout:
# date, open, high, low, close, volume,
# 50-day moving average volume, Intraday Intensity,
# 20-day moving average close,
# upper Bollinger Band, lower Bollinger Band
*/

#include <chplot.h>

int main() {
    class CPlot plot;
    int linetype, linewidth;

    plot.title("Stock price for company X (in semi-log scale)");
    plot.label(PLOT_AXIS_X, "");
    plot.label(PLOT_AXIS_Y, "price");
    plot.border(PLOT_BORDER_ALL, PLOT_ON);
    plot.ticsMirror(PLOT_AXIS_XY, PLOT_ON);
    plot.dataFile("finance.dat", "using 0:5");
    plot.plotType(PLOT_PLOTTYPE_FILLEDCURVES, 0, "y1=0", "green");
    plot.dataFile("finance.dat", "using 0:5");
    linetype = 3;
    linewidth = 2;
    plot.plotType(PLOT_PLOTTYPE_LINES, 1);
    plot.lineType(1, linetype, linewidth);
    plot.axisRange(PLOT_AXIS_X, 50, 253);
    plot.axisRange(PLOT_AXIS_Y, 75, 105);
    plot.scaleType(PLOT_AXIS_Y, PLOT_SCALETYPETYPE_LOG);
    plot.grid(PLOT_ON, "front linewidth 2");
    plot.ticsPosition(PLOT_AXIS_Y, 105, 100, 95, 90, 85, 80);
    /* use the code below for X-label for C++ or Ch */
    /*

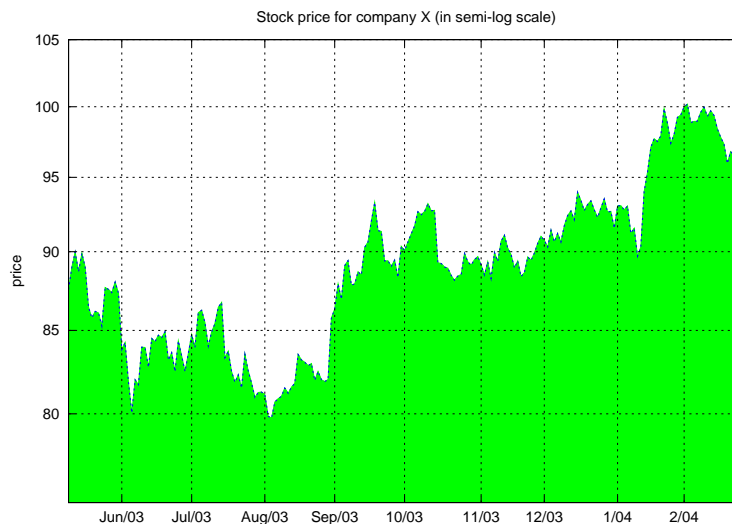
```

```

    plot.ticsPosition(PLOT_AXIS_Y, 105);
    plot.ticsPosition(PLOT_AXIS_Y, 100);
    plot.ticsPosition(PLOT_AXIS_Y, 95);
    plot.ticsPosition(PLOT_AXIS_Y, 90);
    plot.ticsPosition(PLOT_AXIS_Y, 85);
    plot.ticsPosition(PLOT_AXIS_Y, 80);
    */
    plot.ticsLabel(PLOT_AXIS_X, "Jun/03", 66, "Jul/03", 87, "Aug/03", 109, "Sep/03", 130,
                  "10/03", 151, "11/03", 174, "12/03", 193, "1/04", 215, "2/04", 235);
;
    /* use the code below for X-label for C++ or Ch */
    /*
    plot.ticsLabel(PLOT_AXIS_X, "6/03", 66);
    plot.ticsLabel(PLOT_AXIS_X, "7/03", 87);
    plot.ticsLabel(PLOT_AXIS_X, "8/03", 109);
    plot.ticsLabel(PLOT_AXIS_X, "9/03", 130);
    plot.ticsLabel(PLOT_AXIS_X, "10/03", 151);
    plot.ticsLabel(PLOT_AXIS_X, "11/03", 174);
    plot.ticsLabel(PLOT_AXIS_X, "12/03", 193);
    plot.ticsLabel(PLOT_AXIS_X, "1/04", 215);
    plot.ticsLabel(PLOT_AXIS_X, "2/04", 235);
    */
    plot.plotting();
}

```

Output



Example 15

The example illustrates the plot type **PLOT_PLOTTYPE_BOXES** can be found on page 44.

See Also

CPlot::arrow(), **CPlot::lineType()**, **CPlot::fillStyle()**, **CPlot::pointType()**.

CPlot::plotting

Synopsis

```
#include <chplot.h>
void plotting();
```

Purpose

Generate a plot file or display a plot.

Return Value

None.

Parameters

None.

Description

The plot is displayed or a file is generated containing the plot when this function is called. It shall be called after all the desired plot options are set.

Example

See CPlot::data2D().

CPlot::point

Synopsis

```
#include <chplot.h>
int point(double x, double y, ... /* double z */);
```

Syntax

```
point(x, y);
point(x, y, z);
```

Purpose

Add a point to a 2D or 3D plot.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x The x coordinate of the point.

y The y coordinate of the point.

z The z coordinate of the point.

Description

This function adds a point to a plot. It is a convenience function for creation of geometric primitives. A point added with this function counts as a data set for later calls to CPlot::legend() and CPlot::plotType(). For 2D rectangular and 3D cartesian plots, *x*, *y*, and *z* are the coordinates of the point specified in units of the *x*, *y* and *z* axes. However, for 2D polar and 3D cylindrical plots, the point is

specified in polar coordinates where x is θ , y is r , and z is unchanged. Again, for 2D plots, z is ignored. For 3D plots with spherical coordinates x is θ , y is ϕ and z is r . For 3D plots with points, hidden line removal should be disabled (see **CPlot::removeHiddenLine()**) after all data are added.

The size and color of a point can be changed by **CPlot::strokeWidth()** and **CPlot::strokeColor()**, respectively. By default, the color of a point is red. The member function **CPlot::noStrokeColor()** only disables the outline of an object. It has no effect on the color of a point.

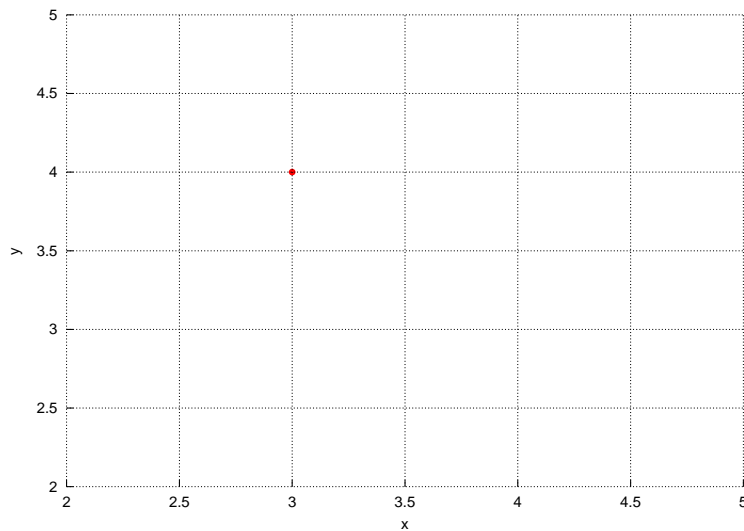
Example 1

```
/* File: point.ch */
#include <chplot.h>

int main() {
    double x = 3, y = 4;
    class CPlot plot;

    plot.axisRange(PLOT_AXIS_XY, 2, 5); /* one point cannot do autorange */
    plot.point(x, y);
    plot.point(x, y+0.5);
    plot.plotting();
}
```

Output



Example 2

```
/* File: point_2.ch */
#include <chplot.h>

int main() {
    double x = 3, y = 4;
    class CPlot plot;

    plot.axisRange(PLOT_AXIS_XY, 2, 5); /* one point cannot do autorange */
    plot.strokeColor("black");
    plot.point(x, y-1);
    plot.point(x, y-0.5);
}
```

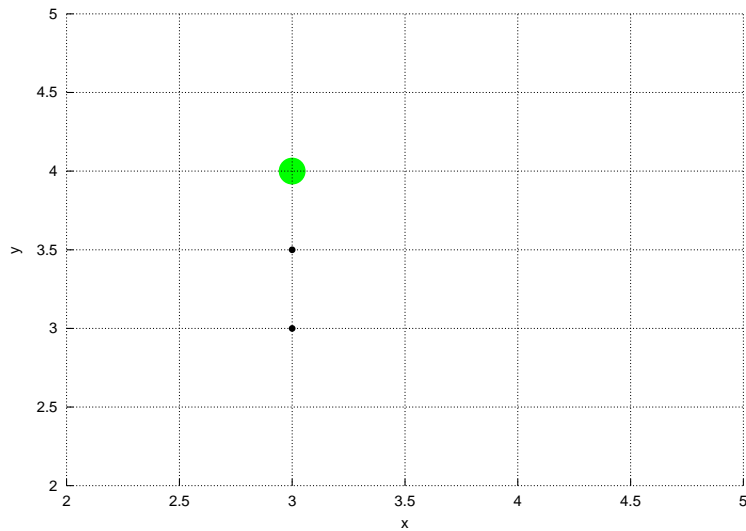


```

    plot.strokeColor("green");
    plot.strokeWidth(5);
    plot.point(x, y);
    plot.plotting();
}

```

Output



Example 3

```

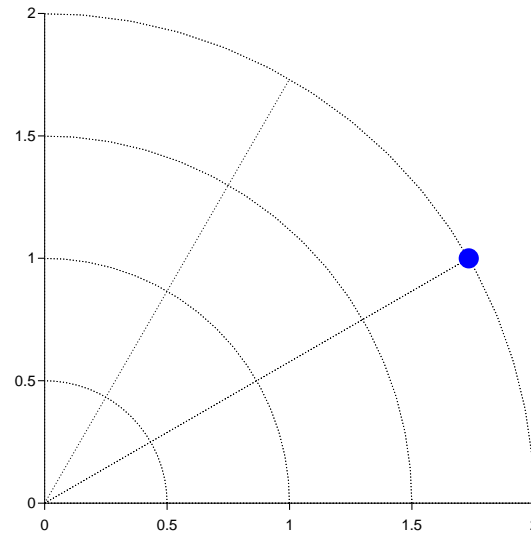
/* File: point_3.ch */
#include <chplot.h>

int main() {
    double theta1 = 30, r1 = 2;
    class CPlot plot;
    int point_type = 7, point_size = 3;
    char *point_color = "blue";

    plot.grid(PLOT_ON);
    plot.polarPlot(PLOT_ANGLE_DEG);
    plot.strokeWidth(3);
    plot.strokeColor(point_color);
    plot.point(theta1, r1);
    //plot.plotType(PLOT_PLOTTYPE_POINTS, 0);
    //plot.pointType(0, point_type, point_size, point_color);
    plot.axisRange(PLOT_AXIS_XY, 0, 2);
    plot.sizeRatio(1);
    plot.plotting();
}

```

Output

**See Also**

CPlot::data2D(), **CPlot::data2DCurve()**, **CPlot::data3D()**, **CPlot::data3DCurve()**,
CPlot::data3DSurface(), **CPlot::circle()**, **CPlot::line()**, **CPlot::outputType()**, **CPlot::plotType()**,
CPlot::pointStyle(), **CPlot::polygon()**, **CPlot::rectangle()**, **CPlot::scatter()**, **CPlot::scattern()**.

CPlot::pointStyle

Synopsis

```
#include <chplot.h>
void pointStyle(string_t style);
```

Purpose

Set the point style such as circle, filled circle, etc.

Return Value

None.

Parameters

style The point style.

Description

Set the point style for points and scatter plots, After this member function is called to set the point style, it will be effective for all points added afterward. The point style can be changed again later in the program.

Argument	Point Style
"."	point
"+"	plus
"x"	cross
"*"	asterisk
"s"	square
"sf"	filled square
"o"	circle
"of"	filled circle
"^"	upward-pointing triangle
"^f"	filled upward-pointing triangle
"v"	downward-pointing triangle
"vf"	filled downward-pointing triangle
"d"	diamond
"df"	filled diamond

Example 1

```

/* File: pointStyle.ch */
#include <chplot.h>

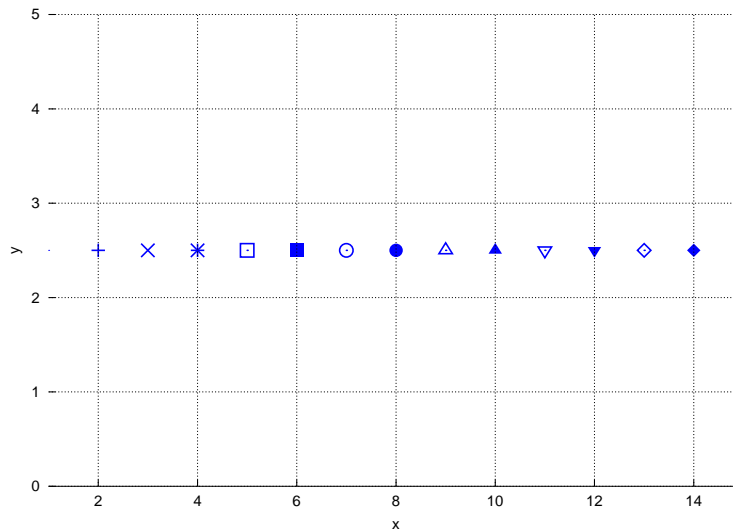
double x = 1, y = 2.5;
CPlot plot;

plot.axisRange(PLOT_AXIS_X, 1, 15); /* one point cannot do autorange */
plot.axisRange(PLOT_AXIS_Y, 0, 5); /* one point cannot do autorange */
plot.strokeColor("blue");
plot.strokeWidth(3);
plot.pointStyle(".");
plot.point(x, y);
plot.pointStyle("+");
plot.point(x+1, y);
plot.pointStyle("x");
plot.point(x+2, y);
plot.pointStyle("*");
plot.point(x+3, y);
plot.pointStyle("s");
plot.point(x+4, y);
plot.pointStyle("sf");
plot.point(x+5, y);
plot.pointStyle("o");
plot.point(x+6, y);
plot.pointStyle("of");
plot.point(x+7, y);
plot.pointStyle("^");
plot.point(x+8, y);
plot.pointStyle("^f");
plot.point(x+9, y);
plot.pointStyle("v");
plot.point(x+10, y);
plot.pointStyle("vf");
plot.point(x+11, y);
plot.pointStyle("d");
plot.point(x+12, y);
plot.pointStyle("df");
plot.point(x+13, y);

```

```
plot.plotting();
```

Output



See Also

CPlot::lineStyle(), CPlot::point(), CPlot::scatter(), CPlot::scattern(), CPlot::strokeColor(), CPlot::strokeWidth().

CPlot::pointType

Synopsis

```
#include <chplot.h>
```

```
void pointType(int num, int point_type, int point_size], ... /* [char point_color] */);
```

Syntax

```
pointType(num, point_type, point_size)
```

```
pointType(num, point_type, point_size, point_color)
```

Purpose

Set the point type, size, and color for points, line-points, etc.

Return Value

None.

Parameters

num The data set to which the point type, size, and color apply.

point_type An integer index representing the desired point type.

point_size A scaling factor for the size of the point used. The point size is *point_size* multiplied by the default size.

point_color color for the point.

Description

Set the desired point type, size, and color for a previously added data set.

Numbering of the data sets starts with zero. The point style and/or marker type for the plot are selected automatically. The default point type, size, and color can be changed by this member function.

The *point_type* specifies an index for the point type used for drawing the point. The point type varies depending on the terminal type used (see **CPlot::outputType**). The value *point_type* is used to change the appearance (color and/or marker type) of a point. It is specified with an integer representing the index of the desired point type. All terminals support at least six different point types. *point_size* is an optional argument used to change the size of the point. The point size is *point_size* multiplied by the default size. If *point_type* and *point_size* are set to zero or a negative number, a default value is used.

An optional fourth argument can specify the color of a point by a color name or RGB value, such as "blue" or "#0000ff" for color blue. The default point type, size, and color can be changed by the function call

```
plot.pointType(num, pointtype, pointsize, "blue");
```

The color of the point is specified as blue in this example.

The detailed color specification can be found in the member function **CPlot::strokeColor()**.

Examples

See programs and their generated figures for **CPlot::point()** and **CPlot::plotType()**.

See Also

CPlot::lineType(), **CPlot::point()**, **CPlot::pointStyle()**, **CPlot::plotType()**.

CPlot::polarPlot

Synopsis

```
#include <chplot.h>
```

```
void polarPlot(int angle_unit);
```

Purpose

Set a 2D plot to use polar coordinates.

Return Value

None.

Parameter

angle_unit Specify the unit for measurement of an angular position. The following options are available:

PLOT_ANGLE_DEG Angles measured in degree.

PLOT_ANGLE_RAD Angles measured in radian.

Description

Set a 2D plot to polar coordinates. In polar mode, two numbers, θ and r , are required for each point. First two arguments in member functions **CPlot::data2D()** and **CPlot::data2DCurve()** are the phase angle θ and magnitude r of points to be plotted.

The plot is generated by calling the following member functions internally.

```

plot.sizeRatio(1); // aspect ratio 1
plot.label(PLOT_AXIS_XY, NULL); // no label
plot.border(PLOT_BORDER_ALL, PLOT_OFF); // no border
plot.ticsLocation(PLOT_AXIS_XY, "axis"); // tics on axis
plot.ticsMirror(PLOT_AXIS_XY, PLOT_OFF); // no tics on mirror

```

You can change the appearance of the function by calling the member functions to change the default look.

Example 1

Compare with the example output in `CPlot::border()`.

```

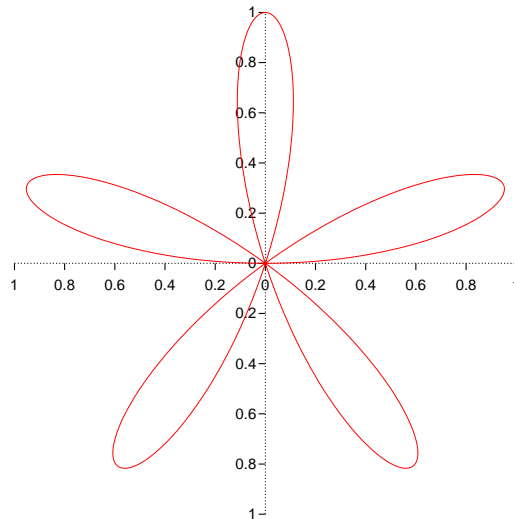
/* File: polarPlot.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 360;
    array double theta[numpoints], r[numpoints];
    class CPlot plot;

    lindata(0, M_PI, theta);
    r = sin(5*theta); // Y-axis data.
    plot.polarPlot(PLOT_ANGLE_RAD);
    plot.data2D(theta, r);
    plot.plotting();
}

```

Output



Example 2

```

/* File: polarPlot_2.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 360;
    array double theta[numpoints], r[numpoints];
    class CPlot plot;

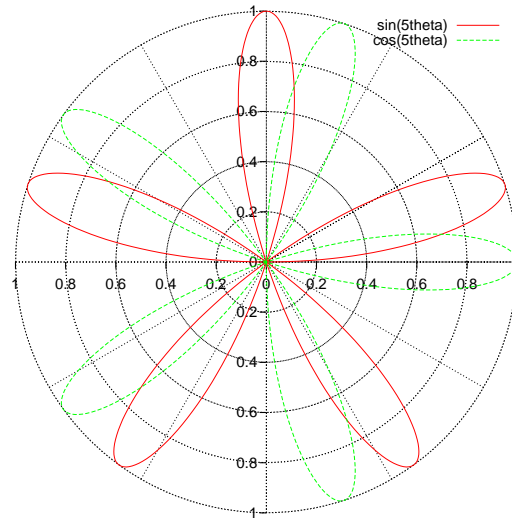
```

```

plot.polarPlot(PLOT_ANGLE_RAD);
plot.grid(PLOT_ON);
lindata(0, M_PI, theta);
r = sin(5*theta); // Y-axis data.
plot.data2D(theta, r);
plot.legend("sin(5theta)", 0);
r = cos(5*theta); // Y-axis data.
plot.data2D(theta, r);
plot.legend("cos(5theta)", 1);
plot.plotting();
}

```

Output



Example 3

```

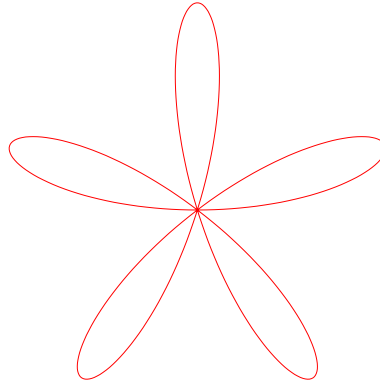
/* File: polarPlot_3.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 360;
    array double theta[numpoints], r[numpoints];
    class CPlot plot;

    lindata(0, M_PI, theta);
    r = sin(5*theta); // Y-axis data.
    plot.polarPlot(PLOT_ANGLE_RAD);
    plot.data2D(theta, r);
    plot.tics(PLOT_AXIS_XY, PLOT_OFF);
    plot.axis(PLOT_AXIS_XY, PLOT_OFF);
    plot.plotting();
}

```

Output

**See Also**

CPlot::grid().

CPlot::polygon

Synopsis

```
#include <chplot.h>
```

```
int polygon(double px[:], double py[:], ... /* double pz[:]; [int num] */);
```

Syntax

```
polygon(px, py)
```

```
polygon(px, py, num)
```

```
polygon(px, py, pz)
```

```
polygon(px, py, pz, num)
```

Purpose

Add a polygon to a plot.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

px An array containing the x coordinates of the vertices of the polygon.

py An array containing the y coordinates of the vertices of the polygon.

pz An array containing the z coordinates of the vertices of the polygon.

num The number of elements of arrays *x*, *y*, and *z*.

Description

This function adds a polygon to a plot. It is a convenience function for creation of geometric primitives. For 2D rectangular plots and 3D cartesian plots, *x*, *y*, and *z* contain the polygon vertices specified in units of the *x*, *y*, and *z* axes. However, for 2D plots, *z* is ignored. For 2D polar and 3D cylindrical plots, the locations of the vertices are specified in polar coordinates where *x* is θ , *y* is *r*, and *z* is unchanged. Again, for 2D plots, *z* is ignored. For 3D plots with spherical coordinates *x* is θ , *y* is ϕ and *z* is *r*. Each of the points is connected to the next in a closed chain. The line type and width vary depending on the terminal type

used (see **CPlot::outputType**). Typically, changing the line type will change the color of the line or make it dashed or dotted. All terminals support at least six different line types.

Please read the section **Handling Outline and Filled Color of Objects in CPlot** described in **CPlot::circle()** on page 51 about how to handle the outline and filled color for a polygon.

Example 1

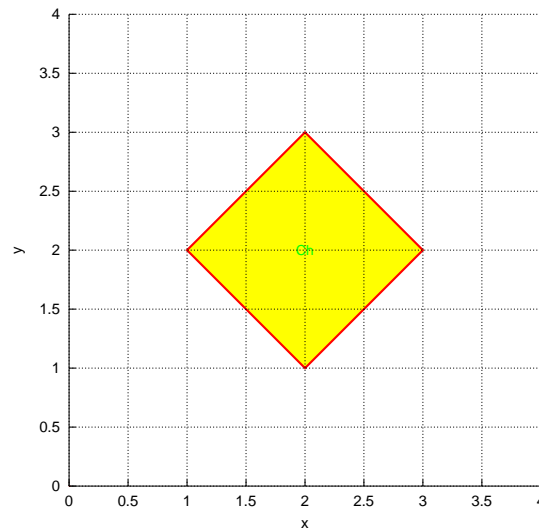
```
/* File: polygon.ch */
#include <chplot.h>

int main() {
    double px[4] = {3, 2, 1, 2}, py[4] = {2, 3, 2, 1};
    class CPlot plot;

    plot.fillColor("yellow");
    plot.polygon(px, py);
    plot.sizeRatio(-1);
    plot.axisRange(PLOT_AXIS_XY, 0, 4);

    plot.strokeColor("green");
    plot.text("Ch", PLOT_TEXT_CENTER, 2, 2);
    plot.plotting();
}
```

Output



Example 2

```
/* File: polygon_2.ch */
#include <chplot.h>
#include <math.h>

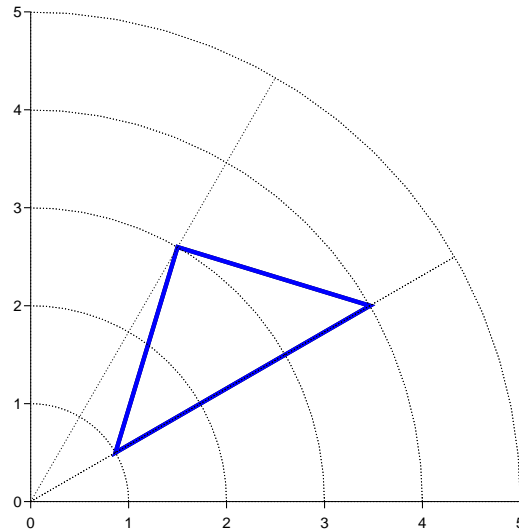
int main(){
    double theta[3] = {30, 60, 30}, r[3] = {1, 3, 4};
    class CPlot plot;

    plot.polarPlot(PLOT_ANGLE_DEG);
    plot.strokeColor("blue");
    plot.strokeWidth(4);
}
```

```

    plot.polygon(theta, r);
    plot.sizeRatio(1);
    plot.axisRange(PLOT_AXIS_XY, 0, 5);
    plot.plotting();
}

```

Output**Example 3**

```

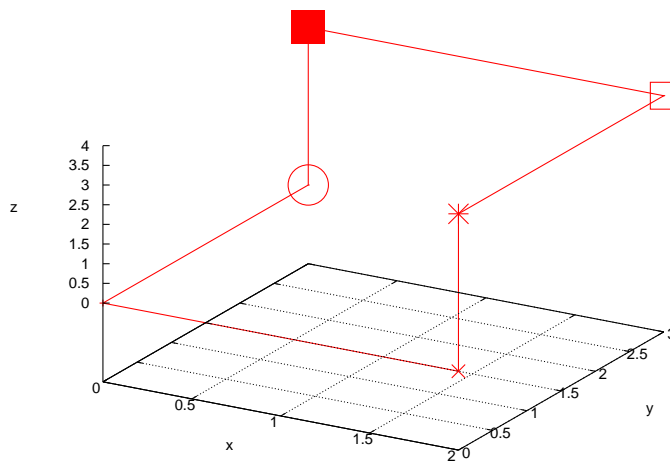
/* File: polygon_3.ch */
#include <chplot.h>

int main() {
    double px[6] = {0, 2, 2, 2, 0, 0}, py[6] = {0, 0, 0, 3, 3, 3},
           pz[6] = {0, 0, 4, 4, 4, 0};
    class CPlot plot;
    int datasetnum, pointtype, pointsize;

    plot.dimension(3);
    plot.polygon(px, py, pz);
    plot.point(0, 0, 0);
    plot.point(2, 0, 0);
    plot.point(2, 0, 4);
    plot.point(2, 3, 4);
    plot.point(0, 3, 4);
    plot.point(0, 3, 0);
    for(datasetnum=1, pointtype=1, pointsize=1;
        datasetnum <= 6;
        datasetnum++, pointtype++, pointsize++)
        plot.plotType(PLOT_PLOTTYPE_POINTS,
                    datasetnum, pointtype, pointsize);
    plot.removeHiddenLine(PLOT_OFF);
    plot.colorBox(PLOT_OFF);
    plot.plotting();
}

```

Output

**See Also**

CPlot::data2D(), **CPlot::data2DCurve()**, **CPlot::data3D()**, **CPlot::data3DCurve()**,
CPlot::data3DSurface(), **CPlot::circle()**, **CPlot::line()**, **CPlot::outputType()**, **CPlot::plotType()**,
CPlot::point(), **CPlot::rectangle()**, **polygonperi()**.

CPlot::quad

Synopsis

```
#include <chplot.h>
```

```
int quad(double x1, double y1, double x2, double y2, double x3, double y3, double x4, double y4);
```

Purpose

Add a quadrilateral to a 2D plot.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x1 The x coordinate of the vertex (x1, y1) of the quadrilateral.

y1 The y coordinate of the vertex (x1, y1) of the quadrilateral.

x2 The x coordinate of the vertex (x2, y2) of the quadrilateral.

y2 The y coordinate of the vertex (x2, y2) of the quadrilateral.

x3 The x coordinate of the vertex (x3, y3) of the quadrilateral.

y3 The y coordinate of the vertex (x3, y3) of the quadrilateral.

x4 The x coordinate of the vertex (x4, y4) of the quadrilateral.

y4 The y coordinate of the vertex (x4, y4) of the quadrilateral.

Description

This function draws a quadrilateral by beginning at the ordered pair (x_1, y_1) and connecting that point to the other three vertices in order from (x_2, y_2) to (x_3, y_3) to (x_4, y_4) and back to the original vertex at (x_1, y_1) . It is a convenience function for creation of geometric primitives.

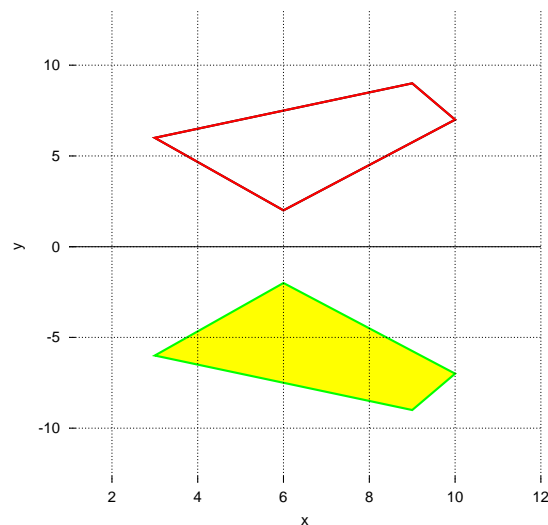
Please read the section **Handling Outline and Filled Color of Objects in CPlot** described in **CPlot::circle()** on page 51 about how to handle the outline and filled color for a quadrilateral.

Example

```
/* File: quad.ch */
#include <chplot.h>

int main() {
    double x1 = 6, y1 = 2, x2 = 3, y2 = 6, x3 = 9, y3 = 9, x4 = 10, y4 = 7;
    CPlot plot;

    plot.quad(x1, y1, x2, y2, x3, y3, x4, y4);
    plot.fillColor("yellow");
    plot.quad(x1, -y1, x2, -y2, x3, -y3, x4, -y4);
    plot.sizeRatio(1);
    plot.plotting();
}
```

Output**See Also**

CPlot::data2D(), **CPlot::data2DCurve()**, **CPlot::data3D()**, **CPlot::data3DCurve()**,
CPlot::data3DSurface(), **CPlot::circle()**, **CPlot::line()**, **CPlot::outputType()**, **CPlot::plotType()**,
CPlot::point(), **CPlot::polygon()**, **CPlot::rectanglexy()**, **CPlot::triangle()**.

CPlot::rectangle**Synopsis**

```
#include <chplot.h>
int rectangle(double x, double y, double width, double height);
```

Purpose

Add a rectangle to a 2D plot.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x The x coordinate of the lower-left corner of the rectangle.

y The y coordinate of the lower-left corner of the rectangle.

width The width of the rectangle.

height The height of the rectangle.

Description

This function adds a rectangle to a 2D plot. It is a convenience function for creation of geometric primitives. For rectangular plots, *x* and *y* are the coordinates of the lower-left corner of the rectangle. For polar plots, the coordinates of the lower-left corner are given in polar coordinates where *x* is theta and *y* is r. In both cases the *width* and *height* are the dimensions of the rectangle in rectangular coordinates.

Please read the section **Handling Outline and Filled Color of Objects in CPlot** described in **CPlot::circle()** on page 51 about how to handle the outline and filled color for a rectangle.

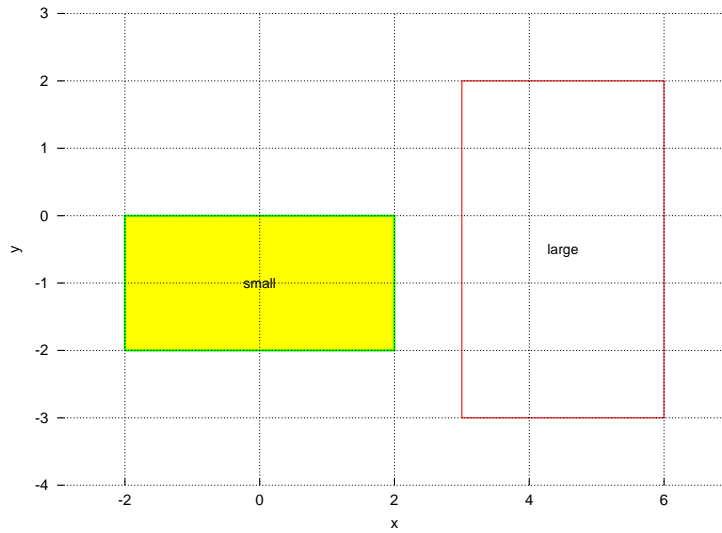
Example 1

```
/* File: rectangle.ch */
#include <chplot.h>

int main() {
    double x1 = 3, y1 = -3, length1 = 3, width1 = 5;
    double x2 = -2, y2 = -2, length2 = 4, width2 = 2;
    class CPlot plot;

    plot.rectangle(x1, y1, length1, width1);
    plot.fillColor("yellow");
    plot.rectangle(x2, y2, length2, width2);
    plot.sizeRatio(-1);
    plot.axisRange(PLOT_AXIS_X, -3, 7);
    plot.axisRange(PLOT_AXIS_Y, -4, 3);
    plot.axis(PLOT_AXIS_XY, PLOT_OFF);
    plot.text("large", PLOT_TEXT_CENTER, 4.5, -0.5);
    plot.text("small", PLOT_TEXT_CENTER, 0, -1);
    plot.plotting();
}
```

Output



Example 2

```

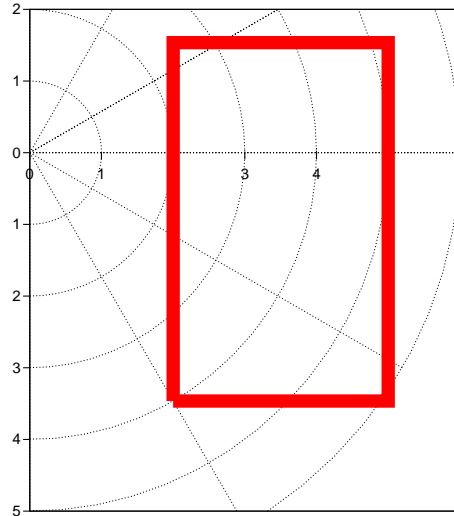
/* File: rectangle_2.ch */
#include <chplot.h>

int main() {
    double thetal = -60, r1 = 4, length1 = 3, width1 = 5;
    class CPlot plot;

    plot.grid(PLOT_ON);
    plot.polarPlot(PLOT_ANGLE_DEG);
    plot.rectangle(thetal, r1, length1, width1);
    plot.plotType(PLOT_PLOTTYPE_LINES, 0);
    plot.lineType(0, 1, 25);
    plot.sizeRatio(-1);
    plot.axisRange(PLOT_AXIS_X, 0, 6);
    plot.axisRange(PLOT_AXIS_Y, -5, 2);
    plot.axis(PLOT_AXIS_XY, PLOT_OFF);
    plot.border(PLOT_BORDER_ALL, PLOT_ON);
    plot.plotting();
}

```

Output

**See Also**

CPlot::data2D(), CPlot::data2DCurve(), CPlot::data3D(), CPlot::data3DCurve(),
 CPlot::data3DSurface(), CPlot::circle(), CPlot::line(), CPlot::outputType(), CPlot::plotType(),
 CPlot::point(), CPlot::polygon().

CPlot::rectanglexy

Synopsis

```
#include <chplot.h>
int rectanglexy(double x1, double y1, double x2, double y2);
```

Purpose

Add a rectangle to a 2D plot.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

- x1* The x coordinate of the lower-left corner of the rectangle.
- y1* The y coordinate of the lower-left corner of the rectangle.
- x2* The x coordinate of the upper-right corner of the rectangle.
- y2* The y coordinate of the upper-right corner of the rectangle.

Description

This function adds a rectangle to a 2D plot. It is a convenience function for creation of geometric primitives.

Please read the section **Handling Outline and Filled Color of Objects in CPlot** described in `CPlot::circle()` on page 51 about how to handle the outline and filled color for a rectangle.

Example

```

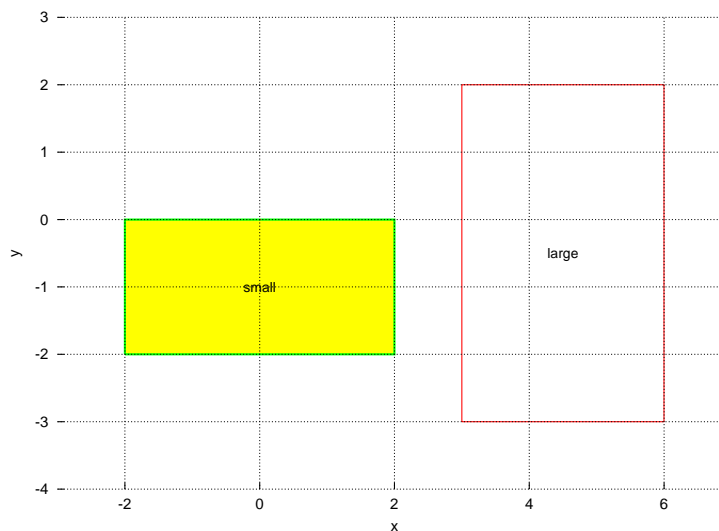
/* File: rectanglexy.ch */
#include <chplot.h>

int main() {
    double x1=3,  y1=-3, x2=6, y2=2;
    double x3=-2, y3 = -2, x4=2, y4=0;
    class CPlot plot;

    plot.rectanglexy(x1, y1, x2, y2);
    plot.fillColor("yellow");
    plot.rectanglexy(x3, y3, x4, y4);
    plot.sizeRatio(-1);
    plot.axisRange(PLOT_AXIS_X, -3, 7);
    plot.axisRange(PLOT_AXIS_Y, -4, 3);
    plot.axis(PLOT_AXIS_XY, PLOT_OFF);
    plot.text("large", PLOT_TEXT_CENTER, 4.5, -0.5);
    plot.text("small", PLOT_TEXT_CENTER, 0, -1);
    plot.plotting();
}

```

Output



See Also

CPlot::data2D(), **CPlot::data2DCurve()**, **CPlot::data3D()**, **CPlot::data3DCurve()**,
CPlot::data3DSurface(), **CPlot::circle()**, **CPlot::line()**, **CPlot::outputType()**, **CPlot::plotType()**,
CPlot::point(), **CPlot::polygon()**, **CPlot::rectangle()**.

CPlot::regularPolygon

Synopsis

```
#include <chplot.h>
```

```
int regularPolygon(double x, double y, int n, double length, double angle);
```

Purpose

Add a regularPolygon to a plot.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x The x coordinate for the center of the regular polygon.

y The y coordinate for the center of the regular polygon.

n The number of sides of the regular polygon.

length The length for each side of the regular polygon.

angle The angle for the regular polygon rotated about the center of the polygon.

Description

This function adds a regular polygon (all angles are the same and all sides have the same length) to a plot. The regular polygon has *n* sides with the *length* for each side. By default, a side of a regular polygon at the bottom is parallel to the x-axis. The regular polygon is rotated with respect to its center by an *angle* in degrees.

This is a convenient function for creation of geometric primitives.

Please read the section **Handling Outline and Filled Color of Objects in CPlot** described in `CPlot::circle()` on page 51 about how to handle the outline and filled color for a regular polygon.

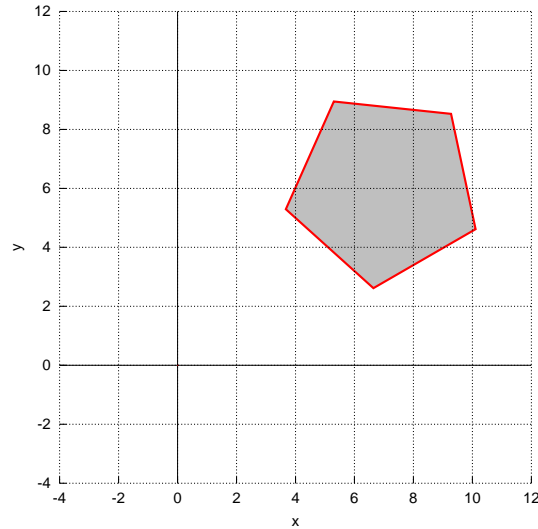
Example

```
/* File: regularPolygon.ch */
#include <chplot.h>
CPlot plot;

plot.fillColor("grey");
/* create a regular polygon centered at (7, 6),
   5 sides with side length 4, rotated by 30 degrees */
plot.regularPolygon(7, 6, 5, 4, 30);

plot.axisRange(PLOT_AXIS_XY, -4, 12);
plot.sizeRatio(1);
plot.plotting();
```

Output

**See Also**

CPlot::data2D(), **CPlot::data2DCurve()**, **CPlot::data3D()**, **CPlot::data3DCurve()**,
CPlot::data3DSurface(), **CPlot::circle()**, **CPlot::line()**, **CPlot::outputType()**, **CPlot::plotType()**,
CPlot::point(), **CPlot::rectangle()**, **CPlot::polygon()**, **polygonperi()**.

CPlot::removeHiddenLine

Synopsis

```
#include <chplot.h>
void removeHiddenLine(int flag);
```

Purpose

Enable or disable hidden line removal for 3D surface plots.

Return Value

None.

Parameter

flag This parameter can be set to:

PLOT_ON Enable hidden line removal.

PLOT_OFF Disable hidden line removal.

Description

Enable or disable hidden line removal for 3D surface plots. This option is only valid for 3D plots with a plot type set to **PLOT_PLOTTYPE_LINES** or **PLOT_PLOTTYPE_LINESPOINTS** with surface display enabled. By default hidden line removal is enabled. This function should be called after data set are added to the plot. The **PLOT_CONTOUR_SURFACE** option for **CPlot::contourMode()** does not work when hidden line removal is enabled. **CPlot::point()** cannot be used when hidden line removal is enabled. By default, the hidden lines are removed.

Example 1

Compare with the output for the example in `CPlot::data3D()`, `CPlot::contourLabel()`, and `CPlot::contourLevels()`.

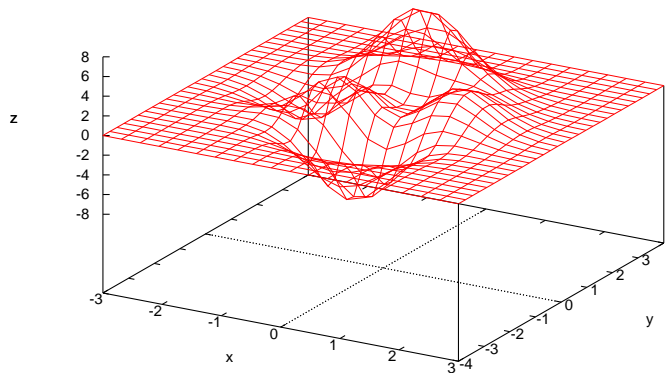
```

/* File: removeHiddenLine.ch */
#include <math.h>
#include <chplot.h>

int main() {
    double x[20], y[30], z[600];
    int i, j;
    class CPlot plot;

    lndata(-3, 3, x);
    lndata(-4, 4, y);
    for(i=0; i<20; i++) {
        for(j=0; j<30; j++) {
            z[30*i+j] = 3*(1-x[i])*(1-x[i])*exp(-(x[i]*x[i])-(y[j]+1)*(y[j]+1))
                - 10*(x[i]/5 - x[i]*x[i]*x[i]-pow(y[j],5))*exp(-x[i]*x[i]-y[j]*y[j])
                - 1/3*exp(-(x[i]+1)*(x[i]+1)-y[j]*y[j]));
        }
    }
    plot.data3D(x, y, z);
    plot.plotType(PLOT_PLOTTYPE_LINES, 0);
    plot.removeHiddenLine(PLOT_OFF);
    plot.plotting();
}

```

Output**Example 2**

```

/* File: removeHiddenLine_2.ch */
#include <math.h>
#include <chplot.h>

int main() {
    double x[30], y[30], z[900];
    double r;
    int i, j;

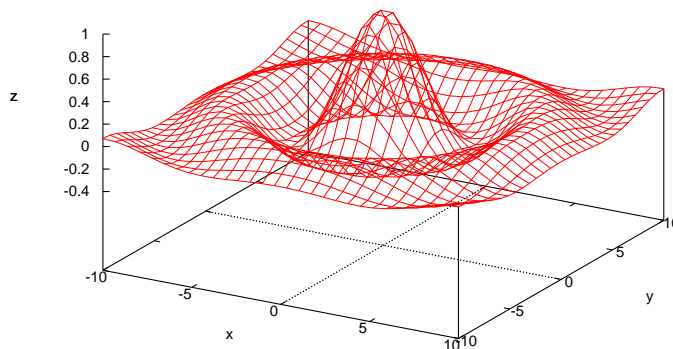
```

```

class CPlot plot;

lindata(-10, 10, x);
lindata(-10, 10, y);
for(i=0; i<30; i++) {
    for(j=0; j<30; j++) {
        r = sqrt(x[i]*x[i]+y[j]*y[j]);
        z[30*i+j] = sin(r)/r;
    }
}
plot.data3D(x, y, z);
plot.removeHiddenLine(PLOT_OFF);
plot.plotting();
}

```

Output**See Also**

CPlot::data3D(), **CPlot::data3DCurve()**, **CPlot::data3DSurface()**, **CPlot::contourMode()**, **CPlot::plotType()**, **CPlot::point()**, **CPlot::showMesh()**.

CPlot::scaleType

Synopsis

```
#include <chplot.h>
```

```
void scaleType(int axis, int scale_type, ... /* [double base] */);
```

Syntax

```
scaleType(axis, scale_type)
```

```
scaleType(axis, scale_type, base)
```

Purpose

Set the axis scale type for a plot.

Return Value

None.

Parameters*axis* The axis to be modified. Valid values are:**PLOT_AXIS_X** Select the x axis only.**PLOT_AXIS_X2** Select the x2 axis only.**PLOT_AXIS_Y** Select the y axis only.**PLOT_AXIS_Y2** Select the y2 axis only.**PLOT_AXIS_Z** Select the z axis only.**PLOT_AXIS_XY** Select the x and y axes.**PLOT_AXIS_XYZ** Select the x, y, and z axes.*scale_type* The scale type for the specified axis. Valid values are:**PLOT_SCALETYPE_LINEAR** Use a linear scale for a specified axis.**PLOT_SCALETYPE_LOG** Use a logarithmic scale for a specified axis.*base* The base for a log scale. For log scales the default base is 10.**Description**

Using this function an axis can be modified to have a logarithmic scale. By default, axes are in linear scale. For a semilog scale in the x-coordinate, call the member function

```
plot.scaleType(PLOT_AXIS_X, PLOT_SCALETYPE_LOG);
```

For a logarithmic base 2, call the member function

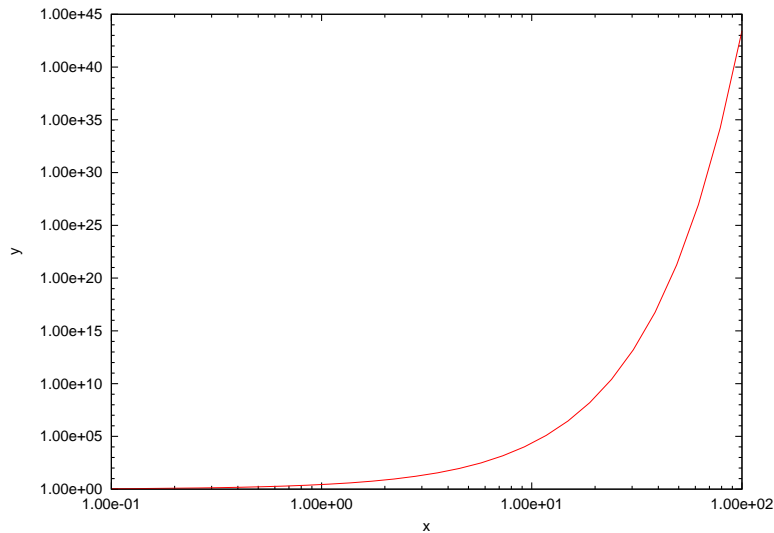
```
plot.scaleType(PLOT_AXIS_X, PLOT_SCALETYPE_LOG, 2); // log base = 2
```

Example

```
/* File: scaleType.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 30;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    logspace(x, -1, 2);
    y = exp(x); // Y-axis data.
    plot.scaleType(PLOT_AXIS_XY, PLOT_SCALETYPE_LOG);
    plot.ticsFormat(PLOT_AXIS_XY, "%.2e");
    plot.data2D(x, y);
    plot.plotting();
}
```

Output

CPlot::scattern**Synopsis**

```
#include <chplot.h>
```

```
void scattern(double x[], double y[], int n, ... /* string_t color */);
```

Syntax

```
scattern(x, y, n);
```

```
scattern(x, y, n);
```

```
scattern(x, y, n, "blue");
```

Purpose

Add a scattern plot to a 2D plot.

Return Value

None

Parameters

x An array containing the x coordinates of points for the scattern plot.

y An array containing the y coordinates of points for the scattern plot.

n The number of elements in arrays *x* and *y*. If the arguments of *x* and *y* are type of pointer to double, the number of elements of the arrays is specified by this argument.

color The color of the points for the scattern plot.

Description

This function draws a scattern plot with the coordinate for each point specified by arrays *x* and *y*.

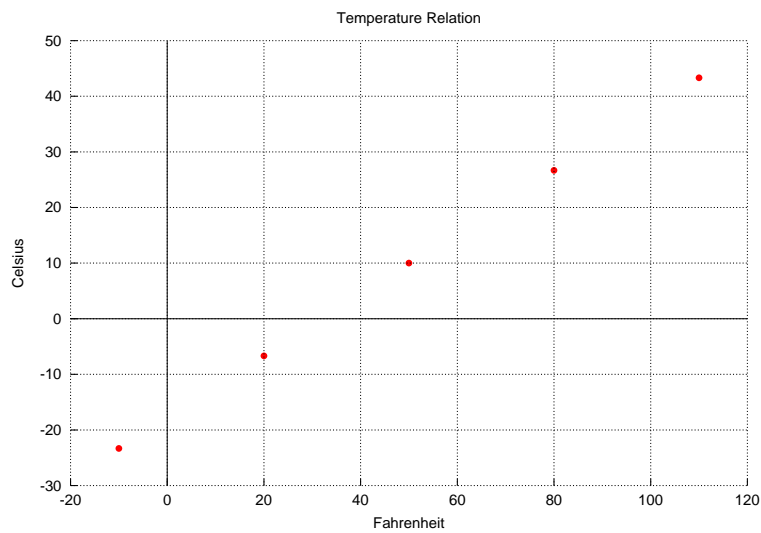
Example

```

/* File: scattern.ch */
#include <chplot.h> /* for CPlot */
CPlot plot;
double x[5] = {-10, 20, 50, 80, 110};
double y[5] = {-23.33, -6.67, 10, 26.67, 43.33};

/* add labels */
plot.title("Temperature Relation");
plot.label(PLOT_AXIS_X, "Fahrenheit");
plot.label(PLOT_AXIS_Y, "Celsius");
plot.scatter(x, y, 5);
plot.plotting();

```

Output**See Also**

CPlot::point(), CPlot::pointStyle(), CPlot::pointType(), CPlot::scatter().

CPlot::scatter**Synopsis**

```

#include <chplot.h>
void scatter(double x[], double y[], ... /* string_t color */);

```

Syntax

```

scatter(x, y);
scatter(x, y, "blue");

```

Purpose

Add a scatter plot to a 2D plot.

Return Value

None

Parameters

x An array containing the x coordinates of points for the scatter plot.

y An array containing the y coordinates of points for the scatter plot.

color The color of the points for the scatter plot.

Description

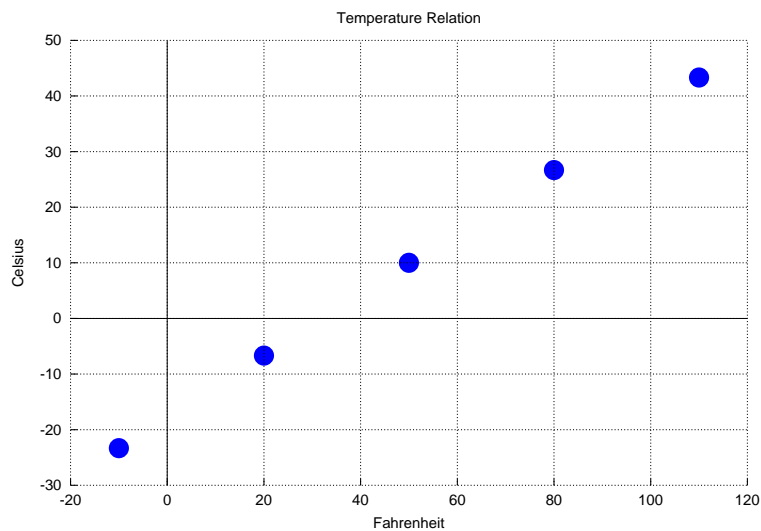
This function draws a scatter plot with the coordinate for each point specified by arrays *x* and *y*.

Example

```
/* File: scatter.ch */
#include <chplot.h> /* for CPlot */
CPlot plot;
double x[] = {-10, 20, 50, 80, 110};
double y[] = {-23.33, -6.67, 10, 26.67, 43.33};

/* add labels */
plot.title("Temperature Relation");
plot.label(PLOT_AXIS_X, "Fahrenheit");
plot.label(PLOT_AXIS_Y, "Celsius");

plot.strokeColor("blue");
plot.strokeWidth(4);
plot.pointStyle("of");
plot.scatter(x, y);
plot.plotting();
```

Output**See Also**

CPlot::point(), **CPlot::pointStyle()**, **CPlot::pointType()**, **CPlot::scattern()**.

CPlot::showMesh

Synopsis

```
#include <chplot.h>
void showMesh(int flag);
```

Purpose

Display 3D data.

Return Value

None.

Parameters

flag This parameter can be set to:

PLOT_ON Enable the display of 3D data.

PLOT_OFF Disable the display of 3D data.

Description

Enable or disable the display of 3D data. If this option is disabled, data points or lines will not be drawn. This option is useful with **CPlot::contourMode()** to display contour lines without the display of a surface grid.

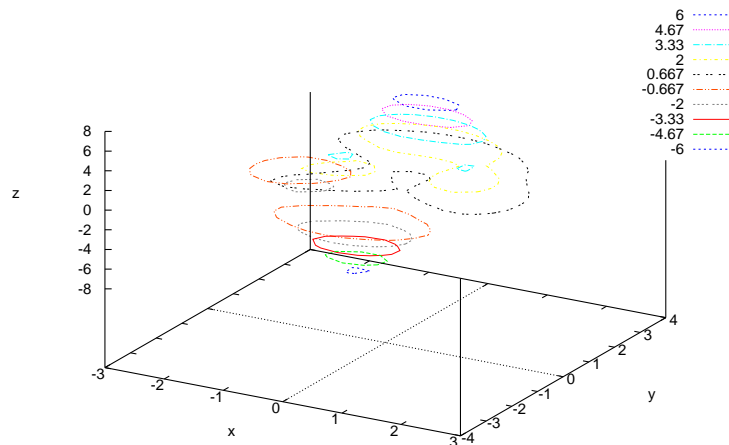
Example 1

Compare with the output for the example in **CPlot::contourLabel()**.

```
/* File: showMesh.ch */
#include <math.h>
#include <chplot.h>

int main() {
    double x[20], y[30], z[600];
    double levels[10];
    int datasetnum = 0, i, j;
    class CPlot plot;

    lindata(-6, 6, levels);
    lindata(-3, 3, x);
    lindata(-4, 4, y);
    for(i=0; i<20; i++) {
        for(j=0; j<30; j++) {
            z[30*i+j] = 3*(1-x[i])*(1-x[i])*exp(-(x[i]*x[i])-(y[j]+1)*(y[j]+1))
                - 10*(x[i]/5 - x[i]*x[i]*x[i]-pow(y[j],5))*exp(-x[i]*x[i]-y[j]*y[j])
                - 1/3*exp(-(x[i]+1)*(x[i]+1)-y[j]*y[j]));
        }
    }
    plot.data3D(x, y, z);
    plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum);
    plot.contourLabel(PLOT_ON);
    plot.showMesh(PLOT_OFF);
    plot.contourMode(PLOT_CONTOUR_SURFACE);
    plot.contourLevels(levels);
    plot.colorBox(PLOT_OFF);
    plot.plotting();
}
```

Output**Example 2**

```

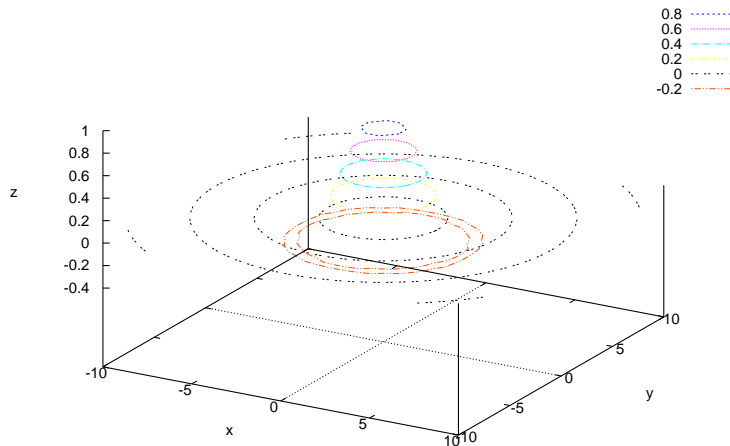
/* File: showMesh_2.ch */
#include <math.h>
#include <chplot.h>

int main() {
    double x[30], y[30], z[900];
    double r;
    int datasetnum = 0, i, j;
    class CPlot plot;

    lindata(-10, 10, x);
    lindata(-10, 10, y);
    for(i=0; i<30; i++) {
        for(j=0; j<30; j++) {
            r = sqrt(x[i]*x[i]+y[j]*y[j]);
            z[30*i+j] = sin(r)/r;
        }
    }
    plot.data3D(x, y, z);
    plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum);
    plot.contourLabel(PLOT_ON);
    plot.showMesh(PLOT_OFF);
    plot.contourMode(PLOT_CONTOUR_SURFACE);
    plot.colorbar(PLOT_OFF);
    plot.plotting();
}

```

Output

**See Also**

CPlot::data3D(), **CPlot::data3DCurve()**, **CPlot::data3DSurface()**, **CPlot::contourMode()**, **CPlot::plotType()**.

CPlot::size

Synopsis

```
#include <chplot.h>
void size(double x_scale, double y_scale);
```

Purpose

Scale the plot itself relative to the size of the output file or canvas.

Return Value

None.

Parameters

x_scale A positive multiplicative scaling factor in the range (0, 1) for the x direction.

y_scale A positive multiplicative scaling factor in the range (0, 1) for the y direction.

Description

This function can be used to scale a plot itself relative to the size of the output file or canvas. If the plot is displayed on a screen, the plot is scaled relative to the size of its canvas. If the plot is saved to a file, or output to the stdout stream, the size of the plot image is scaled relative to the output file. For a plot with subplots, this function should be called before **CPlot::subplot()** is called.

Example

```
/* File: size.ch */
#include <math.h>
#include <chplot.h>

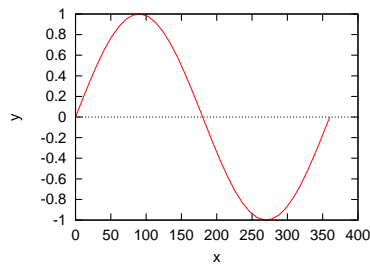
int main() {
```

```

int numpoints = 36;
array double x[numpoints], y[numpoints];
class CPlot plot;

lindata(0, 360, x);
y = sin(x*M_PI/180);
plot.data2D(x, y);
plot.size(.5, .5);
plot.plotting();
}

```

Output**See Also**

CPlot::size3D(), **CPlot::sizeOutput()**, and **CPlot::sizeRatio()**.

CPlot::size3D

Synopsis

```

#include <chplot.h>
void size3D(float scale, float z_scale);

```

Purpose

Scale a 3D plot.

Return Value

None.

Parameters

scale A positive multiplicative scaling factor that is applied to the entire plot.

z_scale A positive multiplicative scaling factor that is applied to the z-axis only.

Description

This function can be used to scale a 3D plot to the desired size. By default, *scale* and *z_scale* are both 1.

Example

Compare with the output for examples in **CPlot::data3D()** and **CPlot::data3DSurface()**.

```

/* File: size3D.ch */
#include <math.h>
#include <chplot.h>

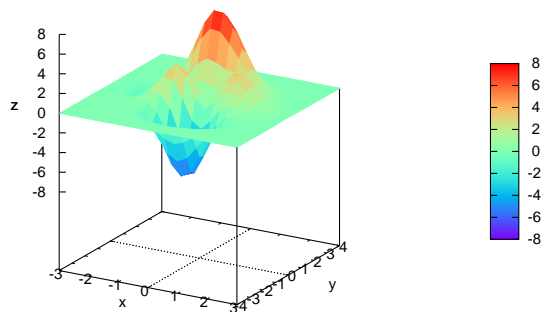
```

```

int main() {
    double x[20], y[30], z[600];
    int i, j;
    class CPlot plot;

    lndata(-3, 3, x);
    lndata(-4, 4, y);
    for(i=0; i<20; i++) {
        for(j=0; j<30; j++) {
            z[30*i+j] = 3*(1-x[i])*(1-x[i])*exp(-(x[i]*x[i])-(y[j]+1)*(y[j]+1))
                - 10*(x[i]/5 - x[i]*x[i]*x[i]-pow(y[j], 5))*exp(-x[i]*x[i]-y[j]*y[j])
                - 1/3*exp(-(x[i]+1)*(x[i]+1)-y[j]*y[j]);
        }
    }
    plot.data3D(x, y, z);
    plot.size3D(0.5, 2);
    plot.plotting();
}

```

Output**See Also**

CPlot::size().

CPlot::sizeOutput

Synopsis

```
#include <chplot.h>
```

```
void sizeOutput(int xpixels, int ypixels);
```

Purpose

Change the size of an output file.

Return Value

None.

Parameters

xpixels A positive integral number for the number of pixels in the x direction.

ypixels A positive integral number for the number of pixels in the y direction.

Description

This function can be used to change the default size (640x480) of an output image file for the plot.

Example

```
/* File: sizeOutput.ch */
#include<math.h>
#include<chplot.h>

#define NUM 36
int main() {
    int i, xpixels = 400, ypixels=600;
    double x[NUM], y[NUM];
    class CPlot plot;

    for(i=0; i<NUM; i++) {
        x[i]= 0 + i*360.0/(NUM-1); // linspace(x, 0, 360)
        y[i] = sin(x[i]*M_PI/180); // Y-axis data
    }
    plot.data2DCurve(x, y, NUM);
    plot.outputType(PLOT_OUTPUTTYPE_FILE, "png", "sizeOutput.png");
    plot.sizeOutput(xpixels, ypixels); // size of image is 400x600 pixels
    plot.plotting();
    return 0;
}
```

See Also

CPlot::size(), CPlot::size3D(), and CPlot::sizeRatio().

CPlot::sizeRatio**Synopsis**

```
#include <chplot.h>
```

```
void sizeRatio(float ratio);
```

Purpose

Change the aspect ratio for a plot.

Return Value

None.

Parameter

ratio The aspect ratio for the plot.

Description

The function sets the aspect ratio for the plot. The meaning of *ratio* changes depending on its value. For a positive *ratio*, it is the ratio of the length of the y-axis to the length of the x-axis. So, if *ratio* is 2, the y-axis will be twice as long as the x-axis. If *ratio* is zero, the default aspect ratio for the terminal type is used. If it

is negative, *ratio* is the ratio of the y-axis units to the x-axis units. So, if *ratio* is -2, one unit on the y-axis will be twice as long as one unit on the x-axis.

Portability

The aspect ratio specified is not exact on some terminals. To get a true ratio of 1, for example, some adjustment may be necessary.

Example

Compare with output for example in `plotxy()`.

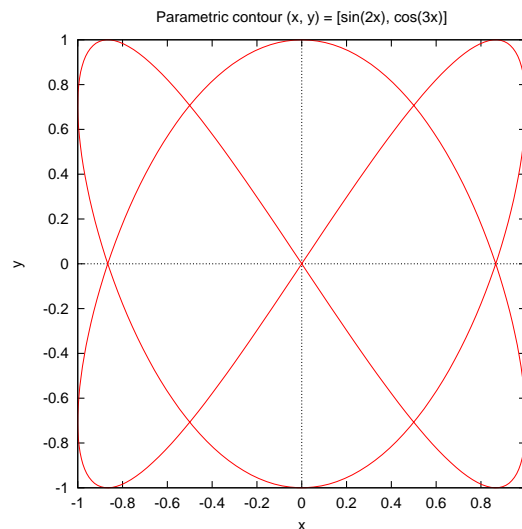
```
/* File: sizeRatio.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 360;
    array double t[numpoints], x[numpoints], y[numpoints];
    class CPlot plot;

    lindata(0, 2*M_PI, t);
    x = sin(2*t);
    y = cos(3*t);
    plot.data2D(x, y);
    plot.title("Parametric contour (x, y) = [sin(2x), cos(3x)]");
    plot.label(PLOT_AXIS_X, "x");
    plot.label(PLOT_AXIS_Y, "y");

    /* both x and y axes the same length */
    plot.sizeRatio(1);
    plot.plotting();
}
```

Output



See Also

`CPlot::size()`, `CPlot::size3D()`.

CPlot::smooth

Synopsis

```
#include <chplot.h>
void smooth(int num, char * option);
```

Syntax

```
smooth(num, option)
```

Purpose

Smooth a plotting curve by interpolation or approximation of data.

Return Value

None.

Parameters

num The data set for the curve to be smooth.

option The options smooth a curve.

Description

This function can be used to readily plot a smooth curve through your data points for a 2D plotting by interpolation and approximation of data. However, sophisticated data processing may be performed by preprocessing the data in your Ch program.

The argument *option* of string type with the following values can be used to smooth the data points.

```
{unique | frequency | csplines | acsplines | bezier | sbezier}
```

The *unique* and *frequency* plot the data after making them monotonic. Each of the other options uses the data to determine the coefficients of a continuous curve between the end points of the data. This curve is then plotted in the same manner as a function, that is, by finding its value at uniform intervals along the abscissa and connecting these points with straight line segments (if a line style is chosen).

If the axis range is determined automatically, the ranges will be computed such that the plotted curve lies within the borders of the graph.

If **CPlot::axisRange()** is called, and the smooth option is either *acspline* or *cspline*, the sampling of the generated curve is done across the intersection of the x range covered by the input data and the fixed abscissa range as defined by **CPlot::axisRange()** for x-axis.

If too few points are available to allow the selected option to be applied, an error message is produced. The minimum number is one for *unique* and *frequency* four for *acsplines*, and three for the others.

"*acsplines*" The *acsplines* option approximates the data with a "natural smoothing spline". After the data are made monotonic in x a curve is piecewise constructed from segments of cubic polynomials whose coefficients are found by the weighting the data points; the weights are taken from the third column in the data file or data in the memory. If the data is from a data file, that default can be modified by the third entry in the option *using list* for **CPlot::dataFile()** as shown below.

```
plot.dataFile("datafile", "using 1:2:(1.0)");
plot.smooth(plot.dataSetNum(), "acsplines");
```


`bezier` The `bezier` option approximates the data with a Bezier curve of degree `n` (the number of data points) that connects the endpoints.

`"csplines"` The `csplines` option connects consecutive points by natural cubic splines after rendering the data monotonic.

`"sbezier"` The `sbezier` option first renders the data monotonic and then applies the `bezier` algorithm.

`"unique"` The `unique` option makes the data monotonic in `x`; points with the same `x`-value are replaced by a single point having the average `y`-value. The resulting points are then connected by straight line segments.

`"frequency"` The `frequency` option makes the data monotonic in `x`; points with the same `x`-value are replaced by a single point having the summed `y`-values. The resulting points are then connected by straight line segments.

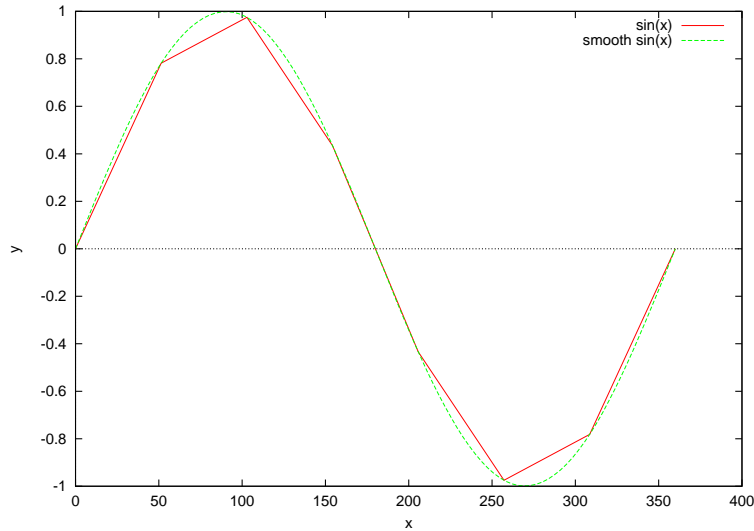
Example 1

```
/* File: smooth.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 8;
    array double x[numpoints], y[numpoints], y2[numpoints];
    class CPlot plot;

    lndata(0, 360, x);
    y = sin(x*M_PI/180);
    y2 = sin(x*M_PI/180);
    plot.data2D(x, y);
    plot.data2D(x, y2);
    plot.smooth(1, "csplines");
    plot.legend("sin(x)", 0);
    plot.legend("smooth sin(x)", 1);
    plot.plotting();
}
```

Output

**Example 2**

See an example on page 211 for **CPlot::plotType()** on how data for an error bar are smoothed by the option `csplines`.

CPlot::star

Synopsis

```
#include <chplot.h>
```

```
int star(double x, double y, double length, double angle);
```

Purpose

Add a star (pentagram) to a plot.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x The x coordinate for the center of the star.

y The y coordinate for the center of the star.

n The number of sides of the regular polygon.

length The point-to-point edge length.

angle The angle for the star rotated about the center of the star.

Description

This function adds a star (pentagram) to a plot. By default, one point of the pentagram is on the top. The star is rotated with respect to its center by an `angle` in degrees.

This is a convenient function for creation of geometric primitives, implemented using a polygon.

Please read the section **Handling Outline and Filled Color of Objects in CPlot** described in **CPlot::circle()** on page 51 about how to handle the outline and filled color for a star.

Example

```

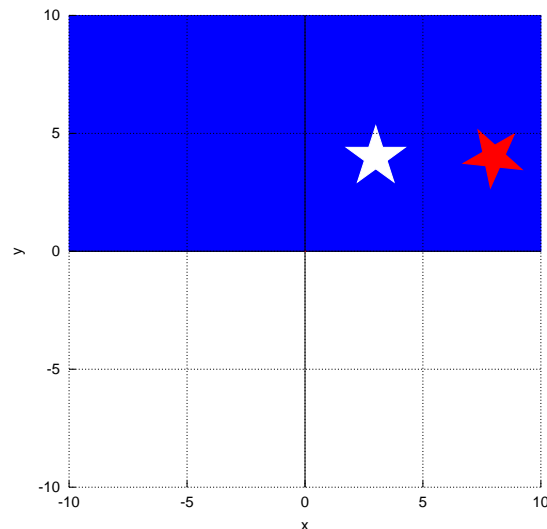
/* File: star.ch */
#include <chplot.h>
CPlot plot;
double length = 2.5, x = 3, y = 4, angle = 30;

plot.noStrokeColor();
plot.fillColor("blue");
plot.rectangle(-10, 0, 20, 10);

plot.fillColor("white");
plot.star(x, y, length, 0);
plot.fillColor("red");
plot.star(x+5, y, length, angle);

plot.sizeRatio(1);
plot.axisRange(PLOT_AXIS_XY, -10, 10);
plot.plotting();

```

Output**See Also**

CPlot::data2D(), **CPlot::data2DCurve()**, **CPlot::data3D()**, **CPlot::data3DCurve()**,
CPlot::data3DSurface(), **CPlot::circle()**, **CPlot::line()**, **CPlot::outputType()**, **CPlot::plotType()**,
CPlot::point(), **CPlot::rectangle()**, **CPlot::regularPolygon()**, **CPlot::polygon()**, **polygonperi()**.

CPlot::strokeColor**Synopsis**

```

#include <chplot.h>
void strokeColor(string_t color);

```

Purpose

Set the color of a point, line, or outline of an object.

Return Value

None.

Parameters*color* The color of a point, line, or outline of an object.**Description**

This function sets the color of a point, line, or outline of an object, including the arc sector, arc segment, circle, ellipse, rectangle, quadrilateral, triangle, and polygon.

By default, the color for points is red. The colors for lines are red, green, blue, magenta, cyan, yellow, black, orange, or grey. The color will be cycled if more than nine lines are added. Similarly, the color of the outline of an object is red, green, blue, magenta, cyan, yellow, black, orange, or grey. The color will be cycled if more than nine objects are added.

Once the member function **CPlot::strokeColor()** is called, the stroke color will be kept for drawing until it is changed by calling this member function again to make a change.

The valid color names and their corresponding GRB values are listed below.

Color Name	Hexadecimal	R	G	B
values				
white	#ffffff	= 255	255	255
black	#000000	= 0	0	0
gray0	#000000	= 0	0	0
grey0	#000000	= 0	0	0
gray10	#1a1a1a	= 26	26	26
grey10	#1a1a1a	= 26	26	26
gray20	#333333	= 51	51	51
grey20	#333333	= 51	51	51
gray30	#4d4d4d	= 77	77	77
grey30	#4d4d4d	= 77	77	77
gray40	#666666	= 102	102	102
grey40	#666666	= 102	102	102
gray50	#7f7f7f	= 127	127	127
grey50	#7f7f7f	= 127	127	127
gray60	#999999	= 153	153	153
grey60	#999999	= 153	153	153
gray70	#b3b3b3	= 179	179	179
grey70	#b3b3b3	= 179	179	179
gray80	#cccccc	= 204	204	204
grey80	#cccccc	= 204	204	204
gray90	#e5e5e5	= 229	229	229
grey90	#e5e5e5	= 229	229	229
gray100	#ffffff	= 255	255	255
grey100	#ffffff	= 255	255	255
gray	#bebebe	= 190	190	190
grey	#bebebe	= 190	190	190
light-gray	#d3d3d3	= 211	211	211

light-grey	#d3d3d3	=	211	211	211
dark-gray	#a9a9a9	=	169	169	169
dark-grey	#a9a9a9	=	169	169	169
red	#ff0000	=	255	0	0
light-red	#f03232	=	240	50	50
dark-red	#8b0000	=	139	0	0
yellow	#ffff00	=	255	255	0
light-yellow	#ffffe0	=	255	255	224
dark-yellow	#c8c800	=	200	200	0
green	#00ff00	=	0	255	0
light-green	#90ee90	=	144	238	144
dark-green	#006400	=	0	100	0
spring-green	#00ff7f	=	0	255	127
forest-green	#228b22	=	34	139	34
sea-green	#2e8b57	=	46	139	87
blue	#0000ff	=	0	0	255
light-blue	#add8e6	=	173	216	230
dark-blue	#00008b	=	0	0	139
midnight-blue	#191970	=	25	25	112
navy	#000080	=	0	0	128
medium-blue	#0000cd	=	0	0	205
royalblue	#4169e1	=	65	105	225
skyblue	#87ceeb	=	135	206	235
cyan	#00ffff	=	0	255	255
light-cyan	#e0ffff	=	224	255	255
dark-cyan	#008b8b	=	0	139	139
magenta	#ff00ff	=	255	0	255
light-magenta	#f055f0	=	240	85	240
dark-magenta	#8b008b	=	139	0	139
turquoise	#40e0d0	=	64	224	208
light-turquoise	#afeeee	=	175	238	238
dark-turquoise	#00ced1	=	0	206	209
pink	#ffc0cb	=	255	192	203
light-pink	#ffb6c1	=	255	182	193
dark-pink	#ff1493	=	255	20	147
coral	#ff7f50	=	255	127	80
light-coral	#f08080	=	240	128	128
orange-red	#ff4500	=	255	69	0
salmon	#fa8072	=	250	128	114
light-salmon	#ffa07a	=	255	160	122
dark-salmon	#e9967a	=	233	150	122
aquamarine	#7fffd4	=	127	255	212
khaki	#f0e68c	=	240	230	140
dark-khaki	#bdb76b	=	189	183	107
goldenrod	#daa520	=	218	165	32
light-goldenrod	#eedd82	=	238	221	130
dark-goldenrod	#b8860b	=	184	134	11
gold	#ffd700	=	255	215	0

beige	#f5f5dc = 245 245 220
brown	#a52a2a = 165 42 42
orange	#ffa500 = 255 165 0
dark-orange	#ff8c00 = 255 140 0
violet	#ee82ee = 238 130 238
dark-violet	#9400d3 = 148 0 211
plum	#dda0dd = 221 160 221
purple	#a020f0 = 160 32 240

Example 1

```

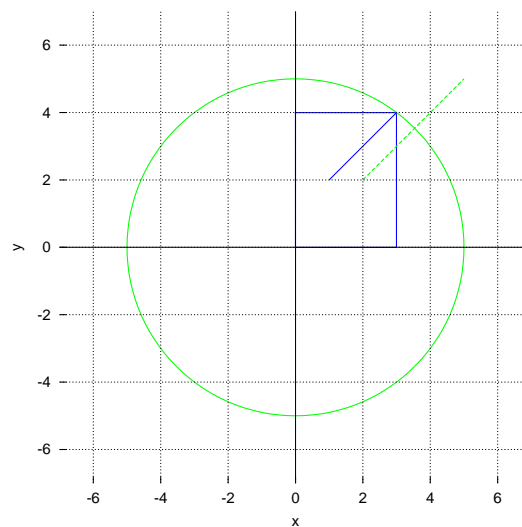
/* File: strokeColor.ch */
#include <chplot.h>
CPlot plot;

plot.strokeColor("blue");
plot.line(1, 2, 3, 4);
plot.rectangle(0, 0, 3, 4);

plot.strokeColor("green");
plot.line(2, 2, 5, 5);
plot.circle(0, 0, 5);

plot.sizeRatio(1);
plot.plotting();

```

Output**See Also**

CPlot::fillColor(), **CPlot::fillOpacity()**, **CPlot::fillPattern()**, **CPlot::noFill()**, **CPlot::noStrokeColor()**, **CPlot::strokeWidth()**.

CPlot::strokeWidth

Synopsis

```
#include <chplot.h>
```

```
void strokeWidth(int width);
```

Purpose

Set the thickness of a point, line, or outline of an object.

Return Value

None.

Parameters

width The thickness of a point, line, or outline of an object.

Description

This function sets the thickness of a point, line, or outline of an object, including the arc sector, arc segment, circle, ellipse, rectangle, quadrilateral, triangle, and polygon. By default, the thickness for a stroke is 2. Once the member function **CPlot::strokeWidth()** is called, the stroke width will be kept for drawing until it is changed by calling this member function again to make a change.

Example 1

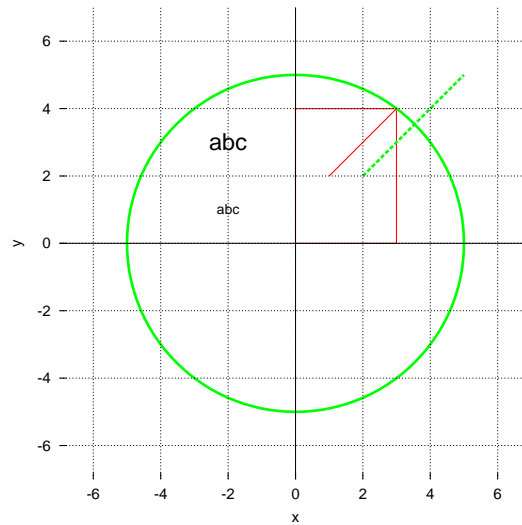
```
/* File: strokeWidth.ch */
#include <chplot.h>
CPlot plot;

plot.strokeWidth(2);
plot.line(1, 2, 3, 4);
plot.rectangle(0, 0, 3, 4);
plot.text("abc", PLOT_TEXT_CENTER, -2, 1);

plot.strokeWidth(5);
plot.line(2, 2, 5, 5);
plot.circle(0, 0, 5);
plot.text("abc", PLOT_TEXT_CENTER, -2, 3);

plot.sizeRatio(1);
plot.plotting();
```

Output

**See Also**

CPlot::fillColor(), **CPlot::fillOpacity()**, **CPlot::fillPattern()**, **CPlot::noFill()**, **CPlot::noStrokeColor()**, **CPlot::fontScale()**, **CPlot::strokeColor()**.

CPlot::subplot

Synopsis

```
#include <chplot.h>
int subplot(int row, int col);
```

Purpose

Create a group of subplots.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

row The number of rows in the subplot.

col The number of columns in the subplot.

Description

This function allocates memory for $(row * col) - 1$ instances of the **CPlot** class to be used in a subplot. The location of the drawing origin and the scale for each element of the subplot is set automatically. The first element of the subplot (an instance of the **CPlot** class) is created normally by the user before this function is called. The remaining elements of the subplot are created and stored internally when this function is called. These elements are accessible through a pointer returned by the **CPlot::getSubplot()** function. Calling **CPlot::subplot()** with $row = col = 1$ has no effect.

The user interface and font size for text for the displayed plot with subplots are different from those for a single plot.

Example

See `CPlot::getSubplot()`.

See Also

`CPlot::origin()`, `CPlot::getSubplot()`, `CPlot::size()`.

CPlot::text

Synopsis

```
#include <chplot.h>
```

```
void text(string_t string, int just, double x, double y, ... /* double z */);
```

Purpose

Add a text string to a plot.

Return Value

None.

Parameters

string The string to be added at location (x,y) for 2D plots or (x,y,z) for 3D plots. The location of the text is in plot coordinates.

just The justification of the text. Valid values are:

PLOT_TEXT_LEFT The specified location is the left side of the text string.

PLOT_TEXT_RIGHT The specified location is the right side of the text string.

PLOT_TEXT_CENTER The specified location is the center of the text string.

x The x position of the text.

y The y position of the text.

z The z position of the text. This argument is ignored for a 2D plot.

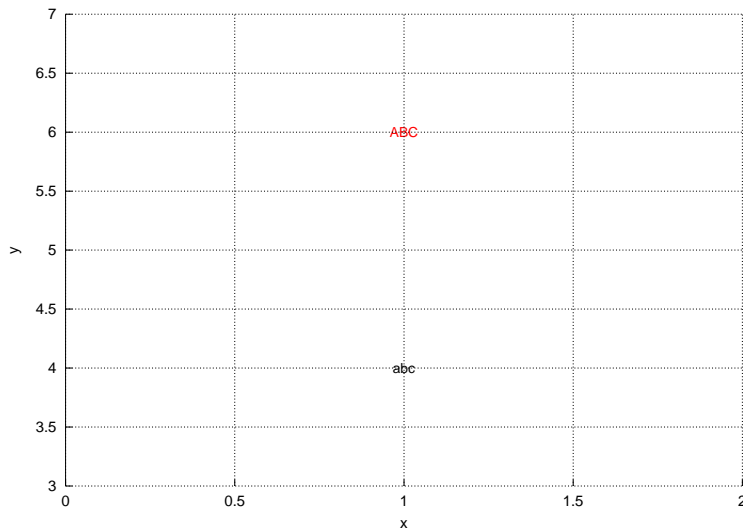
Description

This function can be used to add text to a plot at an arbitrary location. The size of a text can be changed by `CPlot::fontScale()` The color of a text can be changed by `CPlot::textColor()`. By default, the color of a text is black.

Example 1

```
/* File: text.ch */
#include <chplot.h>
CPlot plot;

plot.text("abc", PLOT_TEXT_CENTER, 1, 4);
plot.textColor("red");
plot.text("ABC", PLOT_TEXT_CENTER, 1, 6);
plot.plotting();
```

Output**More examples**

See `CPlot::textColor()`, `CPlot::font()`, `CPlot::fontScale()`, `CPlot::arrow()`, `CPlot::polygon()`, and `CPlot::rectangle()`.

CPlot::textColor

Synopsis

```
#include <chplot.h>
```

```
void textColor(string_t color);
```

Purpose

Set the color of text.

Return Value

None.

Parameters

color The color of text.

Description

This function sets the text color. By default, the text color is black.

Once the member function `CPlot::textColor()` is called, the text color will be kept for drawing until it is changed by calling this member function again to make a change.

The valid color names and their corresponding GRB values are listed in `CPlot::strokeColor()`.

Example

See `CPlot::font()` and `CPlot::text()`.

See Also

`CPlot::text()`, `CPlot::font()`, `CPlot::fontScale()`, `CPlot::strokeColor()`.

CPlot::tics

Synopsis

```
#include <chplot.h>
void tics(int axis, int flag)
```

Purpose

Enable or disable display of axis tics.

Return Value

None.

Parameters

axis The axis which labels are added to. This parameter can take one of the following values:

PLOT_AXIS_X Select the x axis only.

PLOT_AXIS_X2 Select the x2 axis only.

PLOT_AXIS_Y Select the y axis only.

PLOT_AXIS_Y2 Select the y2 axis only.

PLOT_AXIS_Z Select the z axis only.

PLOT_AXIS_XY Select the x and y axes.

PLOT_AXIS_XYZ Select the x, y, and z axes.

flag This parameter can be set to:

PLOT_ON Enable drawing of tics for the specified axis.

PLOT_OFF Disable drawing of tics for the specified axis.

Description

Enable or disable the display of tics and numerical labels for an axis. By default, tics are displayed for the x and y axes on 2D plots and for the x, y and z axes on 3D plots.

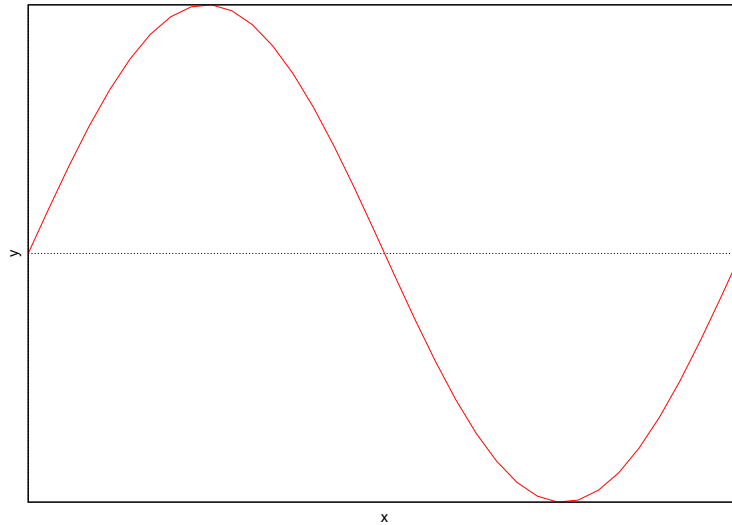
Example

Compare with the output for examples in **CPlot::data2D()** and **CPlot::data2DCurve()**.

```
/* File: tics.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    lindata(0, 360, x);
    y = sin(x*M_PI/180);
    plot.data2D(x, y);
    plot.border(PLOT_BORDER_ALL, PLOT_ON);
    plot.tics(PLOT_AXIS_XY, PLOT_OFF);
    plot.plotting();
}
```

Output**See Also**

CPlot::ticsDirection(), **CPlot::ticsFormat()**, **CPlot::ticsLabel()**, **CPlot::ticsLevel()**, **CPlot::ticsLocation()**, and **CPlot::ticsMirror()**.

CPlot::ticsDay

Synopsis

```
#include <chplot.h>
void ticsDay(int axis);
```

Purpose

Set axis tic-marks to days of the week.

Return Value

None.

Parameter

axis The *axis* parameter can take one of the following values:

- PLOT_AXIS_X** Select the x axis only.
- PLOT_AXIS_X2** Select the x2 axis only.
- PLOT_AXIS_Y** Select the y axis only.
- PLOT_AXIS_Y2** Select the y2 axis only.
- PLOT_AXIS_Z** Select the z axis only.
- PLOT_AXIS_XY** Select the x and y axes.
- PLOT_AXIS_XYZ** Select the x, y, and z axes.

Description

Sets axis tic marks to days of the week (0=Sunday, 6=Saturday). Values greater than 6 are converted into

the value of modulo 7.

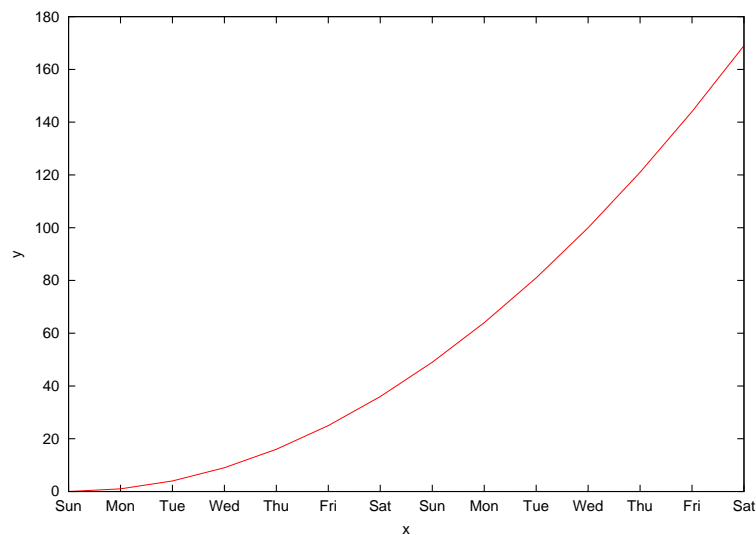
Example

```
/* File: ticsDay.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 14;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    lindata(0, numpoints-1, x);
    y = x.*x;
    plot.ticsDay(PLOT_ON);
    plot.data2D(x, y);
    plot.plotting();
}
```

Output



See Also

CPlot::ticsMonth(), CPlot::ticsLabel().

CPlot::ticsDirection

Synopsis

```
#include <chplot.h>
void ticsDirection(int direction);
```

Purpose

Set the direction in which the tic-marks are drawn for an axis.

Return Value

None.

Parameter

direction Direction tic-marks are drawn. Can be set to:

PLOT_TICS_IN Draw axis tic-marks inward.

PLOT_TICS_OUT Draw axis tic-marks outward.

Description

Set the direction in which tic-marks are drawn in the inward or outward direction from the axes. The default is **PLOT_TICS_IN**.

Example

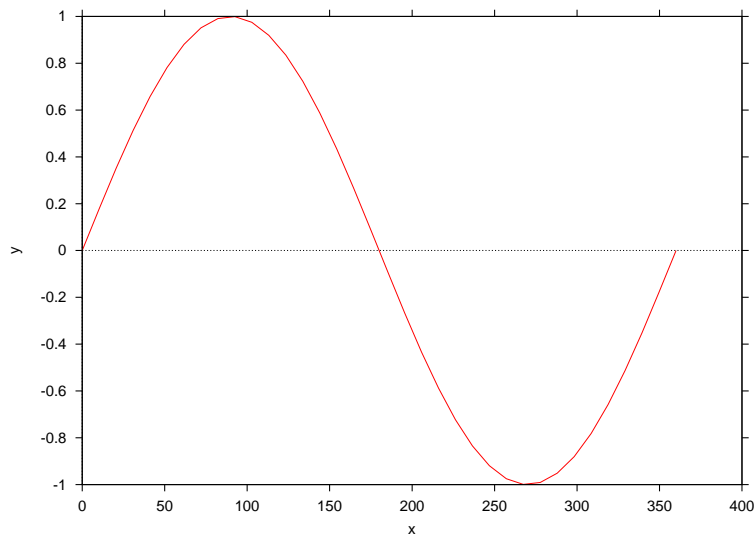
Compare with the output for examples in **CPlot::data2D()** and **CPlot::data2DCurve()**.

```
/* File: ticsDirection.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    lindata(0, 360, x);
    y = sin(x*M_PI/180);
    plot.ticsDirection(PLOT_TICS_OUT);
    plot.data2D(x, y);
    plot.plotting();
}
```

Output



See Also

CPlot::ticsDay(), **CPlot::ticsLabel()**, **CPlot::ticsLocation()**, and **CPlot::ticsMonth()**.

CPlot::ticsFormat

Synopsis

```
#include <chplot.h>
```

```
void ticsFormat(int axis, string_t format);
```

Purpose

Set the number format for tic labels.

Return Value

None.

Parameters

axis The axis to be modified. Valid values are:

PLOT_AXIS_X Select the x axis only.

PLOT_AXIS_X2 Select the x2 axis only.

PLOT_AXIS_Y Select the y axis only.

PLOT_AXIS_Y2 Select the y2 axis only.

PLOT_AXIS_Z Select the z axis only.

PLOT_AXIS_XY Select the x and y axes.

PLOT_AXIS_XYZ Select the x, y, and z axes.

format A C-style conversion specifier. Any conversion specifier suitable for double precision floats is acceptable, but other formats are available.

Description

This function allows for custom number formats for tic-mark labels. The default format is "%g". The table below gives some of the available tics formats in C style.

Format	Effect
%f	Decimal notation for a floating point number. By default, 6 digits are displayed after the decimal point.
%e or %E	Scientific notation for a floating point value. There is always one digit to the left of the decimal point. By default, 6 digits are displayed to the right of the decimal point.
%g or %G	A floating point number. If the value requires an exponent less than -4 or greater than the precision, e or E is used, otherwise f is used.

These format specifiers can be used with the standard C flag characters (-, +, #, etc.), minimum field width specifications and precision specifiers for function **fprintf()**. See **fprintf()** for details.

Examples:

format	number	output
"%f"	234.5678	234.567800
"%.2f"	234.5678	234.57
"%e"	123.456	0.123456e+03
"%E"	123.456	0.123456E+03
"%5.0f"	125.0	125
"%#5.0f"	125.0	125.

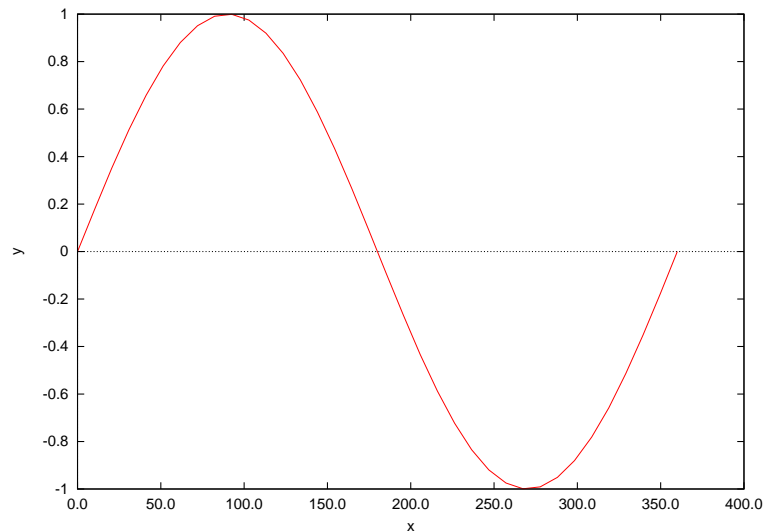
Example

Compare with the output for examples in **CPlot::data2D()** and **CPlot::data2DCurve()**.

```
/* File: ticsFormat.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    lindata(0, 360, x);
    y = sin(x*M_PI/180);
    plot.data2D(x, y);
    plot.ticsFormat(PLOT_AXIS_X, "%.1f");
    plot.plotting();
}
```

Output**See Also**

CPlot::ticsLabel(), **fprintf()**.

CPlot::ticsLabel

Synopsis

```
#include <chplot.h>
```

```
void ticsLabel(int axis, ... /* [ [string_t label, double position], ... ] */);
```

Syntax

```
ticsLabel(axis) /* does nothing */
```

```
ticsLabel(axis, label, position)
```

```
ticsLabel(axis, label, position, label2, position2)
```

etc.

Purpose

Add tic-marks with arbitrary labels to an axis.

Return Value

None.

Parameters

axis The axis which labels are added to. This parameter can take one of the following values:

PLOT_AXIS_X Select the x axis only.

PLOT_AXIS_X2 Select the x2 axis only.

PLOT_AXIS_Y Select the y axis only.

PLOT_AXIS_Y2 Select the y2 axis only.

PLOT_AXIS_Z Select the z axis only.

PLOT_AXIS_XY Select the x and y axes.

PLOT_AXIS_XYZ Select the x, y, and z axes.

label The tic-mark label.

position The position of the tic-mark on the axis.

Description

Add tic marks with arbitrary labels to an axis. The axis specification is followed by one or more pairs of arguments. Each pair consists of a label string and a double precision floating point position. This function disables numerical labels for the specified axis. This function can be called multiple times to set tic labels for an axis.

Example

Compare with the output for examples in **CPlot::data2D()** and **CPlot::data2DCurve()**.

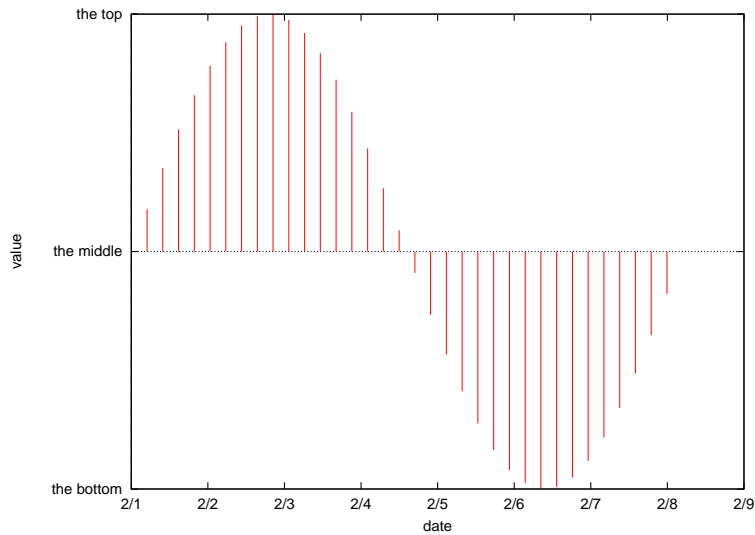
```

/* File: ticsLabel.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    lindata(0, 360, x);
    y = sin(x*M_PI/180);
    plot.label(PLOT_AXIS_X, "date");
    plot.label(PLOT_AXIS_Y, "value");
    plot.data2D(x, y);
    plot.plotType(PLOT_PLOTTYPE_IMPULSES, 0);
    plot.axisRange(PLOT_AXIS_X, 0, 400);
    plot.ticsLabel(PLOT_AXIS_X, "2/1", 0, "2/2", 50,
                  "2/3", 100, "2/4", 150, "2/5", 200, "2/6", 250,
                  "2/7", 300, "2/8", 350, "2/9", 400);
    plot.ticsLabel(PLOT_AXIS_Y, "the bottom", -1, "the middle", 0,
                  "the top", 1);
    plot.plotting();
}

```

Output**See Also**

CPlot::ticsDirection(), CPlot::ticsFormat(), CPlot::ticsLevel(), CPlot::ticsLocation().

CPlot::ticsLevel

Synopsis

```
#include <chplot.h>
void ticsLevel(double level);
```

Purpose

Set the z-axis offset for drawing of tics in 3D plots.

Return Value

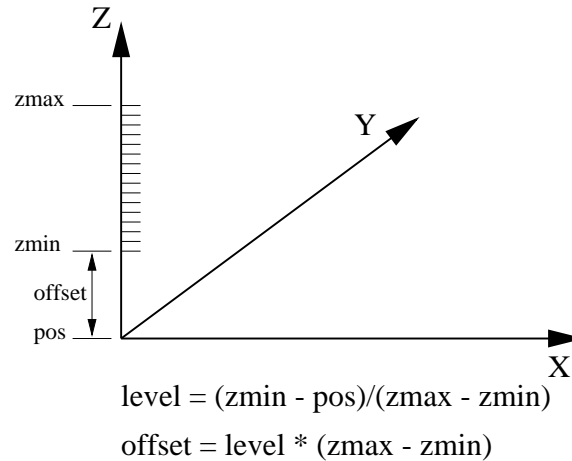
None.

Parameters

level The distance between the xy plane and the start of tic-marks on the z axis as a multiple of the full z range. This can be any non-negative number.

Description

This function specifies an offset between the xy plane and the start of z-axis tics-marks as a multiple of the full z range. By default the value for *level* is 0.5, so the z offset is a half of the z axis range. To place the xy-plane at the specified position *pos* on the z-axis, *level* shall equal $(zmin-pos)/(zmax-zmin)$.

**Example**

Compare with the output for examples in `CPlot::data3D()` and `CPlot::data3DSurface()`.

```

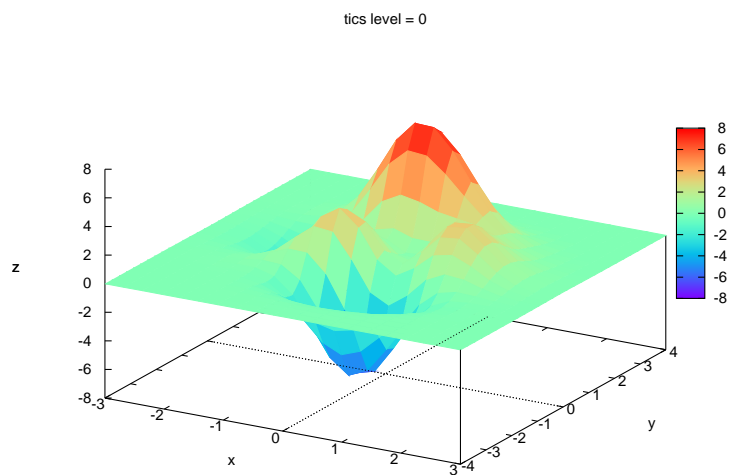
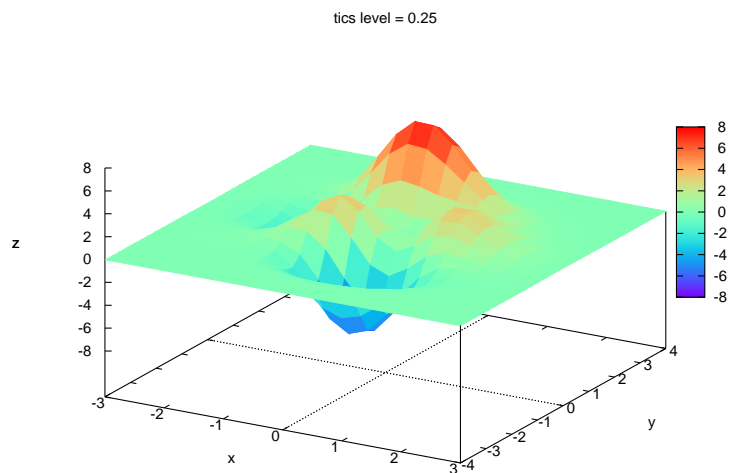
/* File: ticsLevel.ch */
#include <math.h>
#include <chplot.h>

int main() {
    double x[20], y[30], z[600];
    int i,j;
    class CPlot plot;

    lindata(-3, 3, x);
    lindata(-4, 4, y);
    for(i=0; i<20; i++) {
        for(j=0; j<30; j++) {
            z[30*i+j] = 3*(1-x[i])*(1-x[i])*exp(-(x[i]*x[i])-(y[j]+1)*(y[j]+1))
                - 10*(x[i]/5 - x[i]*x[i]*x[i]-pow(y[j],5))*exp(-x[i]*x[i]-y[j]*y[j])
                - 1/3*exp(-(x[i]+1)*(x[i]+1)-y[j]*y[j]));
        }
    }
    plot.data3D(x, y, z);
    plot.ticsLevel(.25);
    plot.title("tics level = 0.25");
    plot.plotting();
    plot.ticsLevel(0);
    plot.title("tics level = 0");
    plot.plotting();
}

```

Output



CPlot::ticsLocation

Synopsis

```
#include <chplot.h>
```

```
void ticsLocation(int axis, string_t location)
```

Purpose

Specify the location of axis tic marks to be on the border or the axis.

Return Value

None.

Parameters

axis The *axis* parameter can take one of the following values:

PLOT_AXIS_X Select the x axis only.

PLOT_AXIS_X2 Select the x2 axis only.

PLOT_AXIS_Y Select the y axis only.

PLOT_AXIS_Y2 Select the y2 axis only.

PLOT_AXIS_XY Select the x and y axes.

location Tic marks are placed on the plot border with "border" or on the axis itself with "axis". By default, tic marks are on the border.

Description

Specify the location of axis tic marks to be on the plot border or the axis itself.

Example

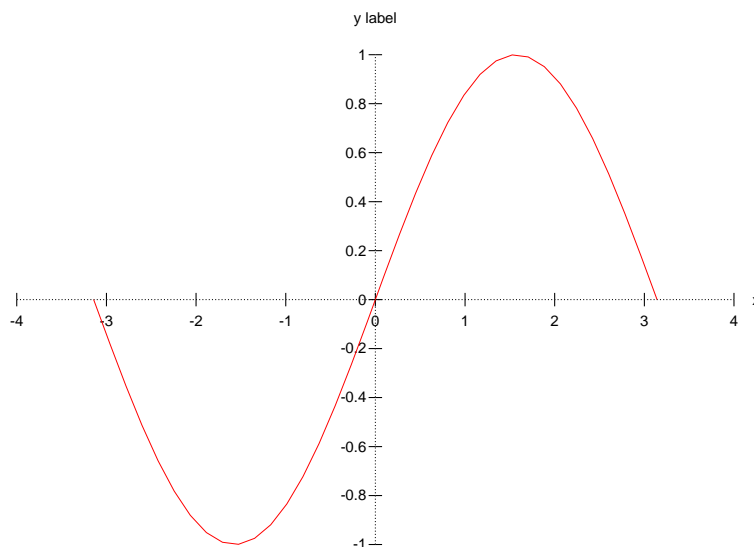
Compare with the output for examples in **CPlot::data2D()** and **CPlot::data2DCurve()**.

```
/* File: ticsLocation.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    lindata(-M_PI, M_PI, x);
    y = sin(x);
    plot.data2D(x, y);
    plot.ticsLocation(PLOT_AXIS_XY, "axis");
    plot.border(PLOT_BORDER_ALL, PLOT_OFF);
    plot.label(PLOT_AXIS_XY, NULL);
    plot.text("y label", PLOT_TEXT_CENTER, 0, 1.15, 0);
    plot.text("x", PLOT_TEXT_CENTER, 4.25, 0, 0);
    plot.margins(-1, -1, 2, -1); /* adjust top margin for y label */
    plot.plotting();
}
```

Output



See Also

CPlot::tics(), **CPlot::ticsDirection()**, **CPlot::ticsFormat()**, **CPlot::ticsLabel**, **CPlot::ticsLevel()**, **CPlot::ticsLocation**, and **CPlot::ticsMirror()**.

CPlot::ticsMirror

Synopsis

```
#include <chplot.h>
```

```
void ticsMirror(int axis, int flag)
```

Purpose

Enable or disable the display of axis tics on the opposite axis.

Return Value

None.

Parameters

axis The axis which labels are added to. This parameter can take one of the following values:

PLOT_AXIS_X Select the x axis only.

PLOT_AXIS_X2 Select the x2 axis only.

PLOT_AXIS_Y Select the y axis only.

PLOT_AXIS_Y2 Select the y2 axis only.

PLOT_AXIS_Z Select the z axis only.

PLOT_AXIS_XY Select the x and y axes.

PLOT_AXIS_XYZ Select the x, y, and z axes.

flag This parameter can be set to:

PLOT_ON Enable drawing of tics for the specified axis.

PLOT_OFF Disable drawing of tics for the specified axis.

Description

Enable or disable the display of tics on the opposite (mirror) axis. By default, on both 2D and 3D plots, tics on the opposite axis are displayed.

Example

Compare with output for example in **CPlot::border()**.

```
/* File: ticsMirror.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

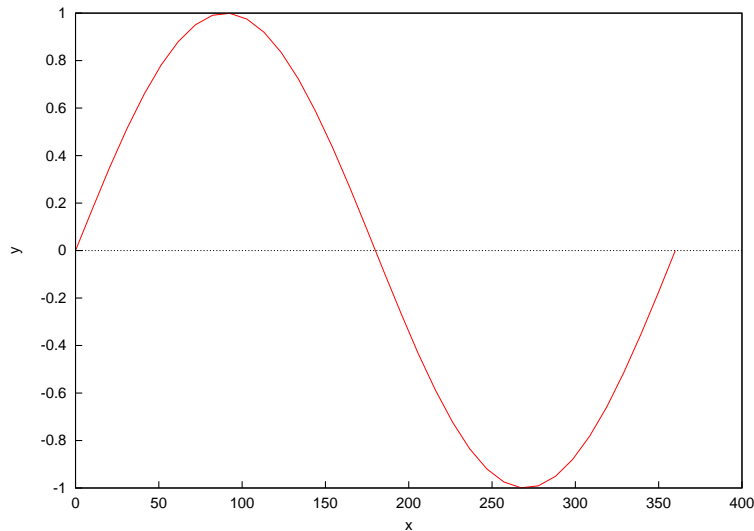
    lindata(0, 360, x);
    y = sin(x*M_PI/180);
```

```

    plot.data2D(x, y);
    plot.border(PLOT_BORDER_ALL, PLOT_ON);
    plot.ticsMirror(PLOT_AXIS_XY, PLOT_OFF);
    plot.plotting();
}

```

Output



See Also

CPlot::tics(), **CPlot::ticsDirection()**, **CPlot::ticsFormat()**, **CPlot::ticsLabel()**, **CPlot::ticsLevel()**, **CPlot::border()**, and **CPlot::ticsLocation()**.

CPlot::ticsMonth

Synopsis

```

#include <chplot.h>
void ticsMonth(int axis);

```

Purpose

Set axis tic-marks to months.

Return Value

None.

Parameter

axis The axis to be changed. Valid values are:

- PLOT_AXIS_X** Select the x axis only.
- PLOT_AXIS_X2** Select the x2 axis only.
- PLOT_AXIS_Y** Select the y axis only.
- PLOT_AXIS_Y2** Select the y2 axis only.
- PLOT_AXIS_Z** Select the z axis only.

PLOT_AXIS_XY Select the x and y axes.

PLOT_AXIS_XYZ Select the x, y, and z axes.

Description

Sets axis tic marks to months of the year (1=January, 12=December). Values greater than 12 are converted into the value of modulo 12.

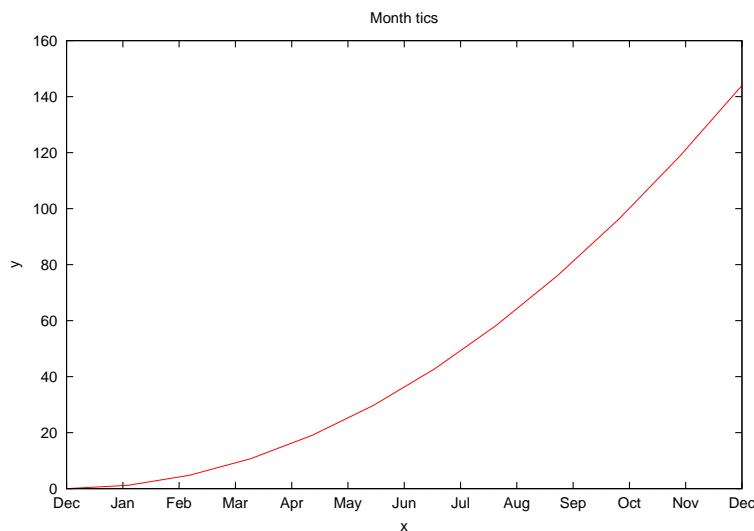
Example

```
/* File: ticsMonth.ch */
#include <math.h>
#include <chplot.h>

int main() {
    array double x[12], y[12];
    string_t title="Month tics",           // Define labels.
               xlabel="x",
               ylabel="y";
    class CPlot plot;

    lndata(0, 12, x);
    y = x.*x;
    plot.ticsMonth(PLOT_AXIS_X);
    plot.title(title);
    plot.label(PLOT_AXIS_X, xlabel);
    plot.label(PLOT_AXIS_Y, ylabel);
    plot.data2D(x, y);
    plot.plotting();
}
```

Output



See Also

CPlot::ticsDay(), CPlot::ticsLabel().

CPlot::ticsPosition

Synopsis

```
#include <chplot.h>
```

```
void ticsPosition(int axis, double position1, ... /* [double position2], ... ] */);
```

Syntax

```
ticsPosition(axis, position)
```

```
ticsPosition(axis, position1, position2)
```

etc.

Purpose

Add tic-marks at the specified positions to an axis.

Return Value

None.

Parameters

axis The axis which tics are added to. This parameter can take one of the following values:

PLOT_AXIS_X Select the x axis only.

PLOT_AXIS_X2 Select the x2 axis only.

PLOT_AXIS_Y Select the y axis only.

PLOT_AXIS_Y2 Select the y2 axis only.

PLOT_AXIS_Z Select the z axis only.

PLOT_AXIS_XY Select the x and y axes.

PLOT_AXIS_XYZ Select the x, y, and z axes.

position The position of the tic-mark on the axis.

Description

Add tic marks at the specified positions to an axis. The axis specification is followed by one or more position values of double precision floating point numbers. This function disables numerical labels for the specified axis. This function can be called multiple times to set tic positions for an axis. In this form, the tics do not need to be listed in numerical order.

Examples

See an example on page 211 using for **CPlot::ticsPosition()** for the x-axis for date.

See Also

CPlot::ticsDirection(), **CPlot::ticsFormat()**, **CPlot::ticsLevel()**, **CPlot::ticsLabel()**, **CPlot::ticsLocation()**, **CPlot::ticsRange()**.

CPlot::ticsRange

Synopsis

```
#include <chplot.h>
```

```
void ticsRange(int axis, double incr, ... /* [double start], [double end] */);
```

Syntax**ticsRange**(*axis*, *incr*) **ticsRange**(*axis*, *incr*, *start*)**ticsRange**(*axis*, *incr*, *start*, *end*)**Purpose**

Specify the range for a series of tics on an axis.

Return Value

None.

Parameters*axis* The *axis* parameter can take one of the following values:**PLOT_AXIS_X** Select the x axis only.**PLOT_AXIS_X2** Select the x2 axis only.**PLOT_AXIS_Y** Select the y axis only.**PLOT_AXIS_Y2** Select the y2 axis only.**PLOT_AXIS_Z** Select the z axis only.**PLOT_AXIS_XY** Select the x and y axes.**PLOT_AXIS_XYZ** Select the x, y, and z axes.*incr* The increment between tic marks. By default or when *incr* is 0, the increment between tic marks is calculated internally.*start* The starting value for tics.*end* The end value for tics.**Description**

The range for a series of tics on an axis can be explicitly specified with this function. Any previously specified labeled tic-marks are overridden. The implicit *start*, *incr*, *end* form specifies that a series of tics will be plotted on the axis between the values *start* and *end* with an increment of *incr*. If *end* is not given, it is assumed to be infinity. The increment may be negative. If neither *start* nor *end* is given, *start* is assumed to be negative infinity, *end* is assumed to be positive infinity, and the tics will be drawn at integral multiples of *incr*. If the axis is logarithmic specified by the member function **scaleType**(), the increment will be used as a multiplicative factor.

ExampleSee **CPlot::axisRange**().**See Also****CPlot::axisRange**(), **CPlot::ticsPosition**(), **CPlot::ticsLabel**().

CPlot::title**Synopsis**

#include <chplot.h>

void title(string_t *title*);

Purpose

Set the plot title.

Return Value

None.

Parameters

title The plot title.

Description

Add a title string to an existing plot variable. For no title, NULL can be specified. By default, no title is specified.

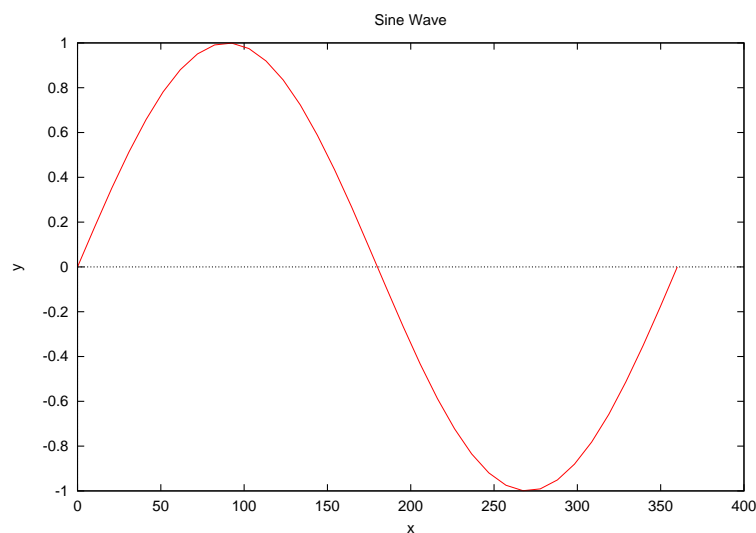
Example

Compare with the output for examples in `CPlot::data2D()` and `CPlot::data2DCurve()`.

```
/* File: title.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    string_t title="Sine Wave"; // Define labels.
    class CPlot plot;

    lndata(0, 360, x);
    y = sin(x*M_PI/180);
    plot.title(title);
    plot.data2D(x, y);
    plot.plotting();
}
```

Output**See Also**

`CPlot::label()`, `CPlot::getLabel()`, `CPlot::getTitle()`.

CPlot::triangle

Synopsis

```
#include <chplot.h>
```

```
int triangle(double x1, double y1, double x2, double y2, double x3, double y3);
```

Purpose

Add a triangle to a 2D plot.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x1 The x coordinate of the vertex (*x1*, *y1*) of the triangle.

y1 The y coordinate of the vertex (*x1*, *y1*) of the triangle.

x2 The x coordinate of the vertex (*x2*, *y2*) of the triangle.

y2 The y coordinate of the vertex (*x2*, *y2*) of the triangle.

x3 The x coordinate of the vertex (*x3*, *y3*) of the triangle.

y3 The y coordinate of the vertex (*x3*, *y3*) of the triangle.

Description

This function adds a triangle defined by three vertices (*x1*, *y1*), (*x2*, *y2*), and (*x3*, *y3*) to a 2D plot. It is a convenience function for creation of geometric primitives.

Please read the section **Handling Outline and Filled Color of Objects in CPlot** described in **CPlot::circle()** on page 51 about how to handle the outline and filled color for a triangle.

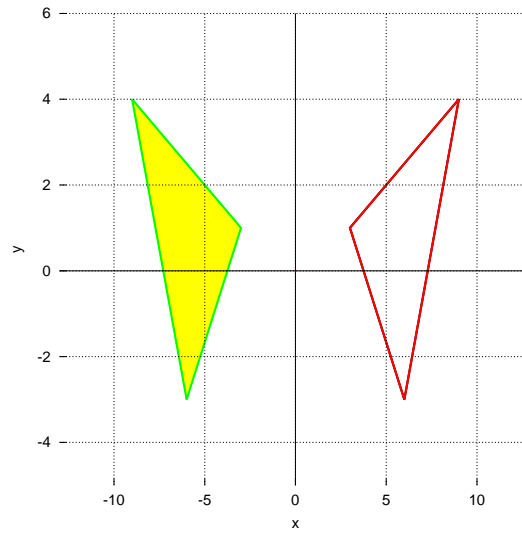
Example

```
/* File: triangle.ch */
#include <chplot.h>

int main() {
    double x1 = 6, y1 = -3, x2 = 3, y2 = 1, x3 = 9, y3 = 4;
    class CPlot plot;

    plot.triangle(x1, y1, x2, y2, x3, y3);
    plot.fillColor("yellow");
    plot.triangle(-x1, y1, -x2, y2, -x3, y3);
    plot.sizeRatio(1);
    plot.plotting();
}
```

Output



See Also

**CPlot::data2D(), CPlot::data2DCurve(), CPlot::data3D(), CPlot::data3DCurve(),
CPlot::data3DSurface(), CPlot::circle(), CPlot::line(), CPlot::outputType(), CPlot::plotType(),
CPlot::point(), CPlot::polygon(), CPlot::rectanglexy().**

fplotxy

Synopsis

```
#include <chplot.h>
```

```
int fplotxy(double (*func)(double x), double x0, double xf, ...
            /* [int num, [string_t title, string_t xlabel, string_t ylabel], [class CPlot *pl]] */ );
```

Syntax

```
fplotxy(func, x0, xf)
```

```
fplotxy(func, x0, xf, num)
```

```
fplotxy(func, x0, xf, num, title, xlabel, ylabel)
```

```
fplotxy(func, x0, xf, num, &plot)
```

```
fplotxy(func, x0, xf, num, title, xlabel, ylabel, &plot)
```

Purpose

Plot a 2D function of x in the range $x0 \leq x \leq xf$.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

func A pointer to a function that takes a `double` as an argument and returns a `double`.

x0 The lower bound of the range to be plotted.

xf The upper bound of the range to be plotted.

num The number of points to be plotted. The default is 100.

title The title of the plot.

xlabel The x-axis label.

ylabel The y-axis label.

pl A pointer to an instance of the **CPlot** class.

Description

Plot a 2D function of x in the range $x0 \leq x \leq xf$. The function to be plotted, *func*, is specified as a pointer to a function that takes a *double* as an argument and returns a *double*. The arguments *x0* and *xf* are the end-points of the range to be plotted. The optional argument *num* specifies how many points in the range are to be plotted. The number of points plotted are evenly spaced in the range. By default, 100 points are plotted. The *title*, *xlabel*, and *ylabel* for the plot can also optionally be specified. A pointer to a plot structure can also be passed to this function. If a non-NULL pointer is passed, it will be initialized with the function parameters. The plot can then be displayed using the **CPlot::plotting()** member function. If a previously initialized *plot* variable is passed, it will be re-initialized with the function parameters. If no pointer or a NULL pointer is passed, an internal **CPlot** variable will be used and the plot will be displayed without calling the **CPlot::plotting()** member function.

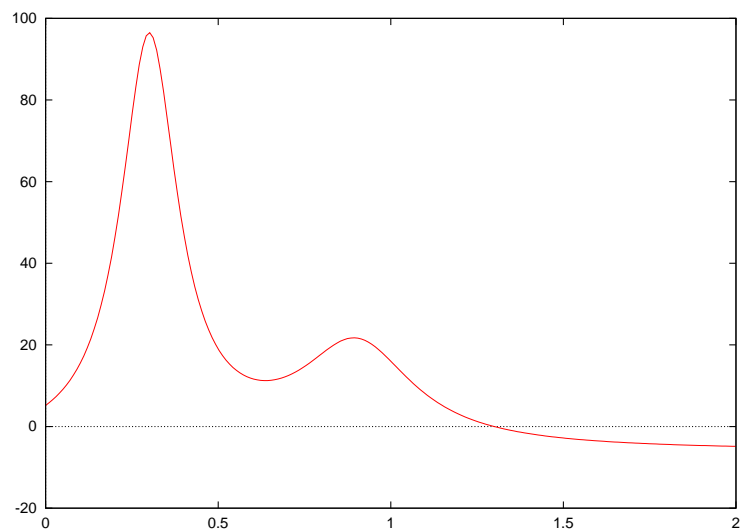
Example

```
/* File: fplotxy.ch */
// Demonstrates the usage of the fplotxy() function.
// Print out a sine wave with appropriate labels and grid range.
#include <math.h>
#include <chplot.h>

int main() {
    double x0 = 0, xf = 2;
    int num = 200;

    /* local function to be plotted */
    double func2(double x) {
        return 1/((x-0.3)*(x-0.3)+0.01) + 1/((x-0.9)*(x-0.9)+0.04) - 6;
    }
    fplotxy(func2,x0,xf,num);
}
```

Output



See Also

CPlot, **fplotxyz()**, **plotxy()**, **plotxyf()**, **plotxyz()**, **plotxyzf()**.

fplotxyz

Synopsis

```
#include <chplot.h>
```

```
int fplotxyz(double (*func)(double x, double y), double x0, double xf, double y0, double yf, ...
             /* [int x_num, int y_num, string_t title, string_t xlabel, string_t ylabel, string_t zlabel],
                [class CPlot *pl]] */);
```

Syntax

```
fplotxyz(func, x0, xf, y0, yf)
```

```
fplotxyz(func, x0, xf, y0, yf, x_num, y_num)
```

```
fplotxyz(func, x0, xf, y0, yf, x_num, y_num, &plot)
```

```
fplotxyz(func, x0, xf, y0, yf, x_num, y_num, title, xlabel, ylabel, zlabel)
```

```
fplotxyz(func, x0, xf, y0, yf, x_num, y_num, title, xlabel, ylabel, zlabel, &plot)
```

Purpose

Plot a 3D function of x and y in the range $x_0 \leq x \leq x_f$ and $y_0 \leq y \leq y_f$.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

func A pointer to a function that takes two **double** arguments and returns a **double**.

x0 The lower bound of the x range to be plotted.

xf The upper bound of the x range to be plotted.

y0 The lower bound of the y range to be plotted.

yf The upper bound of the y range to be plotted.

x_num The number of points to be plotted. The default is 25.

y_num The number of points to be plotted. The default is 25.

title The title of the plot.

xlabel The x -axis label.

ylabel The y -axis label.

zlabel The z -axis label.

pl A pointer to an instance of the **CPlot** class.

Description

Plot a 3D function of x and y in the range $x_0 \leq x \leq x_f$ and $y_0 \leq y \leq y_f$. The function to be plotted, *func*, is specified as a pointer to a function that takes two **double** arguments and returns a **double**. x_0 and x_f are the end-points of the x range to be plotted. y_0 and y_f are the end-points of the y range to be plotted. The optional arguments *x_num* and *y_num* specify how many points in the x and y ranges are to be plotted. The number of

points plotted are evenly spaced in the ranges. By default, *x_num* and *y_num* are 25. The *title*, *xlabel*, *ylabel*, and *zlabel* for the plot can also optionally be specified. A pointer to a plot structure can also be passed to this function. If a non-NULL pointer is passed, it will be initialized with the function parameters. The plot can then be displayed using the **CPlot::plotting()** member function. If a previously initialized **CPlot** variable is passed, it will be re-initialized with the function parameters. If no pointer or a NULL pointer is passed, an internal **CPlot** variable will be used and the plot will be displayed without calling the **CPlot::plotting()** member function. This function can only be used to plot 3D grid or scatter data, it cannot be used to plot 3D paths.

Example

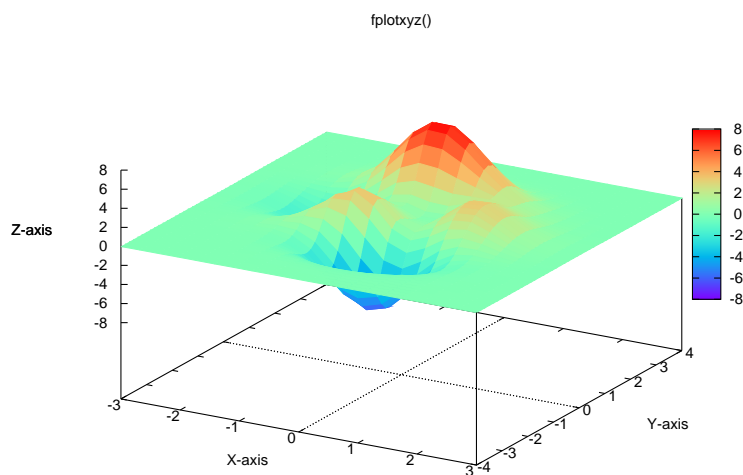
```
/* File: fplotxyz.ch */
// Demonstrates the usage of the fplotxyz() function.
// Print out a sine wave with appropriate labels and grid range.
#include <math.h>
#include <chplot.h>

int main() {
    string_t title="fplotxyz()",           // Define labels.
              xlabel="X-axis",
              ylabel="Y-axis",
              zlabel="Z-axis";
    double x0 = -3, xf = 3, y0 = -4, yf = 4;
    int x_num = 20, y_num = 50;

    double func(double x, double y) {     // function to be plotted

        return 3*(1-x)*(1-x)*exp(-(x*x) - (y+1)*(y+1) )
            - 10*(x/5 - x*x*x - pow(y,5))*exp(-x*x-y*y)
            - 1/3*exp(-(x+1)*(x+1) - y*y);
    }
    fplotxyz(func, x0, xf, y0, yf, x_num, y_num, title, xlabel, ylabel, zlabel);
}
```

Output



See Also

CPlot, **fplotxy()**, **plotxy()**, **plotxyf()**, **plotxyz()**, **plotxyzf()**.

plotxy

Synopsis

```
#include <chplot.h>
int plotxy(double x[&], array double &y, ...
/* [int n] [string_t title, string_t xlabel, string_t ylabel],
   [class CPlot *pl] */);
```

Syntax

```
plotxy(x, y)
plotxy(x, y, title, xlabel, ylabel)
plotxy(x, y, title, xlabel, ylabel, &plot)
plotxy(x, y, n)
plotxy(x, y, n, title, xlabel, ylabel)
plotxy(x, y, n, title, xlabel, ylabel, &plot)
```

Purpose

Plot a 2D data set or initialize an instance of the **CPlot** class.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x A one-dimensional array of size *n*. The value of each element is used for the x-axis of the plot.

y A *m* x *n* dimensional array containing *m* curves, each of which is plotted against *x*.

n An integer for the number of elements of array *x*.

title The *title* of the plot.

xlabel The x-axis label.

ylabel The y-axis label.

pl A pointer to an instance of the **CPlot** class.

Description

The arrays *x* and *y* can be of any supported data type of real numbers. Conversion of the data to **double** type is performed internally. The optional argument *n* for the number of elements for array *x* is for the compatibility with SIGL C++ graphical library. The *title*, *xlabel*, and *ylabel* for the plot can also optionally be specified. A pointer to a plot structure can also be passed to this function. If a non-NULL pointer is passed, it will be initialized with the function parameters. The plot can then be displayed using the **CPlot::plotting()** member function. If a previously initialized **CPlot** variable is passed, it will be re-initialized with the function parameters. If no pointer or a NULL pointer is passed, an internal **CPlot** variable will be used and the plot will be displayed without calling the **CPlot::plotting()** member function.

The following code segment

```
class CPlot plot;
plotxy(x, y, "title", "xlabel", "ylabel", &plot);
```

is equivalent to

```
class CPlot plot;
plot.data2D(x, y);
plot.title("title");
plot.label(PLOT_AXIS_X, "xlabel");
plot.label(PLOT_AXIS_Y, "ylabel");
```

The code segment

```
class CPlot plot;
plotxy(x, y, n, "title", "xlabel", "ylabel", &plot);
```

is equivalent to

```
class CPlot plot;
plot.data2DCurve(x, y, n);
plot.title("title");
plot.label(PLOT_AXIS_X, "xlabel");
plot.label(PLOT_AXIS_Y, "ylabel");
```

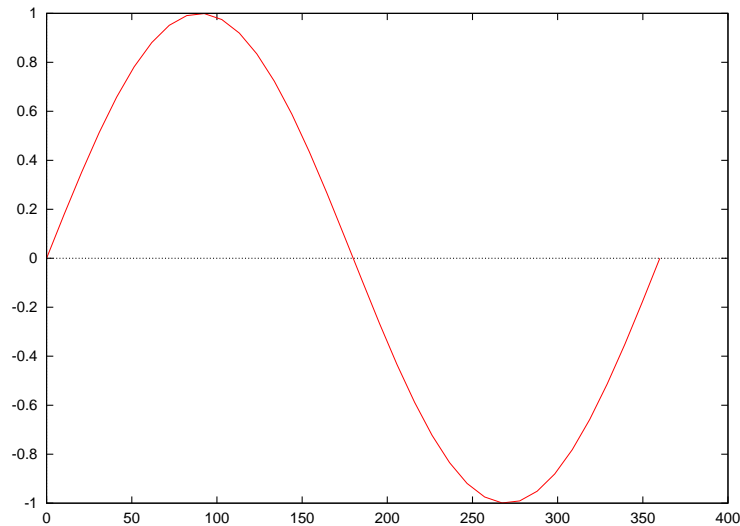
Example 1

```
/* File: plotxy_5.ch */
#include <math.h>
#include <chplot.h>

int main() {
    array double x[36];

    lindata(0, 360, x);
    plotxy(x, sin(x*M_PI/180));
}
```

Output

**Example 2**

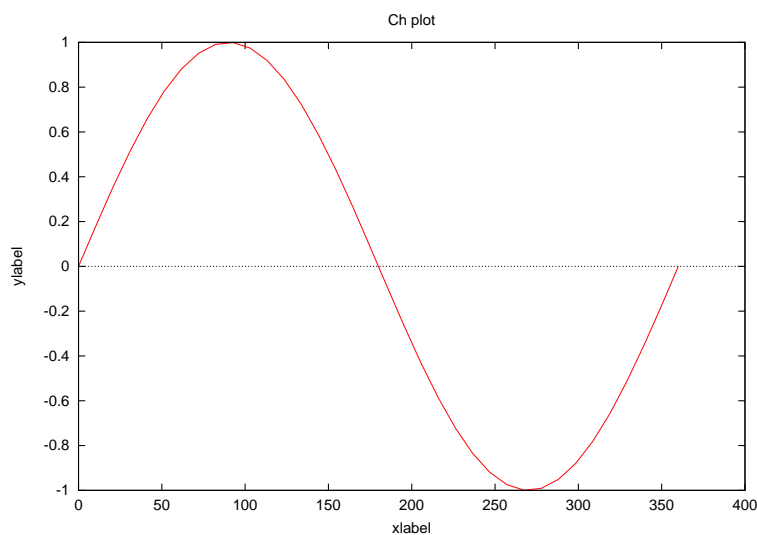
```

/* File: plotxy.ch */
/* File: plotxy.ch
   Plot data in computational arrays using function plotxu() */
#include <math.h>
#include <chplot.h>

int main() {
    int n = 37;          /* number of points for the plot */
    array double x[n], y[n]; /* arrays with data for plotting */

    lindata(0, 360, x, n); /* fill array x with n points from 0 to 360 */
    y = sin(x*M_PI/180);   /* y = sin(x) with x in radian */
    plotxy(x, y, n, "function sin(x)", "x (degree)", "sin(x)");
    return 0;
}

```

Output**Example 3**

```

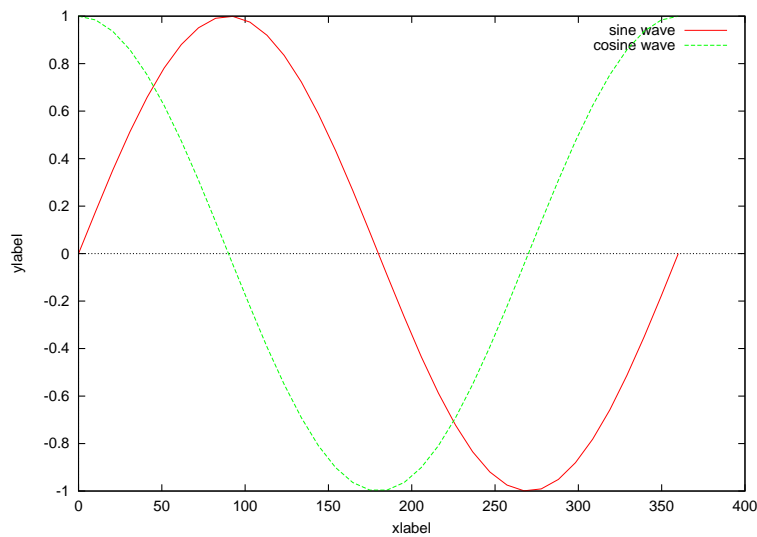
/* File: plotxy_2.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[2][numpoints];
    int i;
    class CPlot plot;

    lndata(0, 360, x);
    for(i=0; i<numpoints; i++) {
        y[0][i] = sin(x[i]*M_PI/180);
        y[1][i] = cos(x[i]*M_PI/180);
    }
    plotxy(x, y, NULL, "xlabel", "ylabel", &plot);
    plot.legend("sine wave", 0);
    plot.legend("cosine wave", 1);
    plot.plotting();
}

```

Output



Example 4

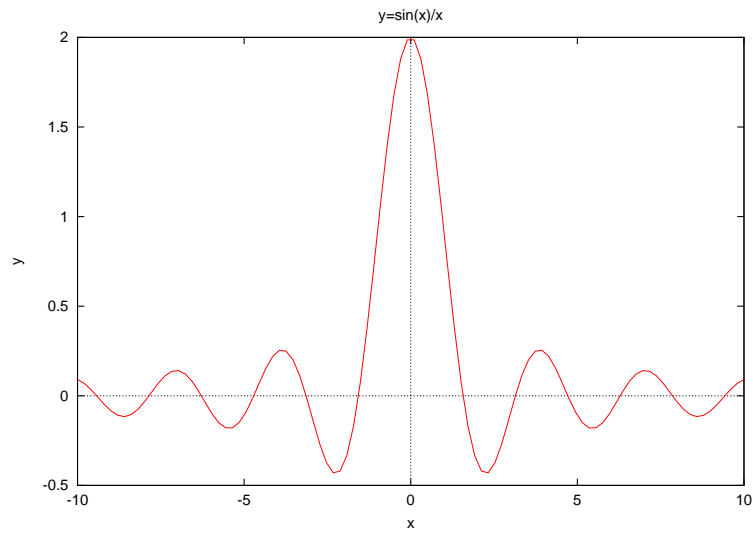
```

/* File: plotxy_4.ch */
#include <math.h>
#include <float.h>
#include <chplot.h>

int main() {
    int numpoints = 100;
    array double x[numpoints], y[numpoints];

    lndata(-10, 10, x);
    x = x+(x==0)*FLT_EPSILON; /* if x==0, x becomes epsilon */
    y = sin(2*x)./x;
    plotxy(x, y, "y=sin(x)/x", "x", "y");
}

```

Output**Example 5**

Compare with the output in the example in `CPlot::sizeRatio()`.

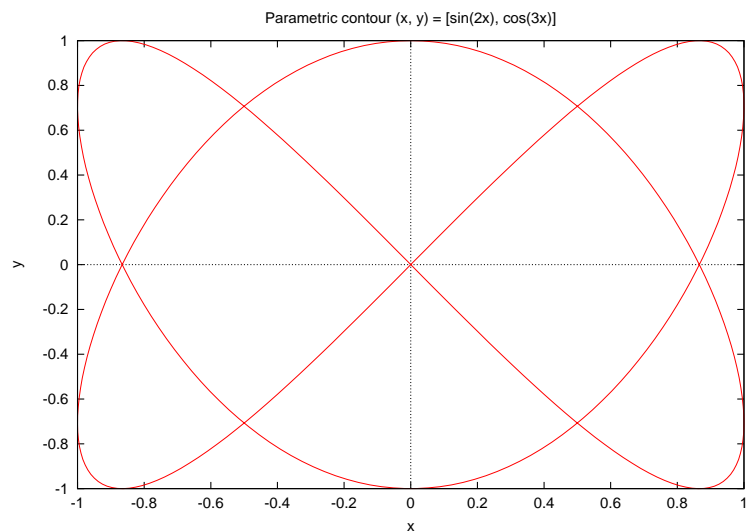
```

/* File: plotxy_3.ch */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 360;
    array double t[numpoints], x[numpoints], y[numpoints];

    lindata(0, 2*M_PI, t);
    x = sin(2*t);
    y = cos(3*t);
    plotxy(x, y, "Parametric contour (x, y) = [sin(2x), cos(3x)]", "x", "y");
}

```

Output

See Also

CPlot, CPlot::data2D(), CPlot::data2DCurve(), fplotxy(), fplotxyz(), plotxyf(), plotxyz(), plotxyzf().

plotxyf

Synopsis

```
#include <chplot.h>
```

```
int plotxyf(string_t file, ... /* [string_t title, string_t xlabel, string_t ylabel], [class CPlot *pl] */ );
```

Syntax

```
plotxyf(file)
```

```
plotxyf(file, title, xlabel, ylabel)
```

```
plotxyf(file, title, xlabel, ylabel, &plot)
```

Purpose

Plot 2D data from a file or initialize an instance of the **CPlot** class.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

file The file containing the data to be plotted.

title The title of the plot.

xlabel The x-axis label.

ylabel The y-axis label.

pl A pointer to an instance of the **CPlot** class.

Description

Plot 2D data from a file or initialize a **CPlot** variable. The data file should be formatted with each data point on a separate line. 2D data are specified by two values per point. The *title*, *xlabel*, and *ylabel*, for the plot can also optionally be specified. A pointer to a plot structure can also be passed to this function. If a non-NULL pointer is passed, it will be initialized with the function parameters. The plot can then be displayed using the **CPlot::plotting** member function. If a previously initialized **CPlot** variable is passed, it will be re-initialized with the function parameters. If no pointer or a NULL pointer is passed, an internal **CPlot** variable will be used and the plot will be displayed without calling the **CPlot::plotting()** member function. An empty line in the data file causes a break in the plot. Multiple curves can be plotted in this manner, however, the plot style will be the same for all curves.

The following code segment

```
class CPlot plot;
plotxyf("datafile", "title", "xlabel", "ylabel", &plot);
```

is equivalent to

```
class CPlot plot;
plot.dataFile("datafile");
```



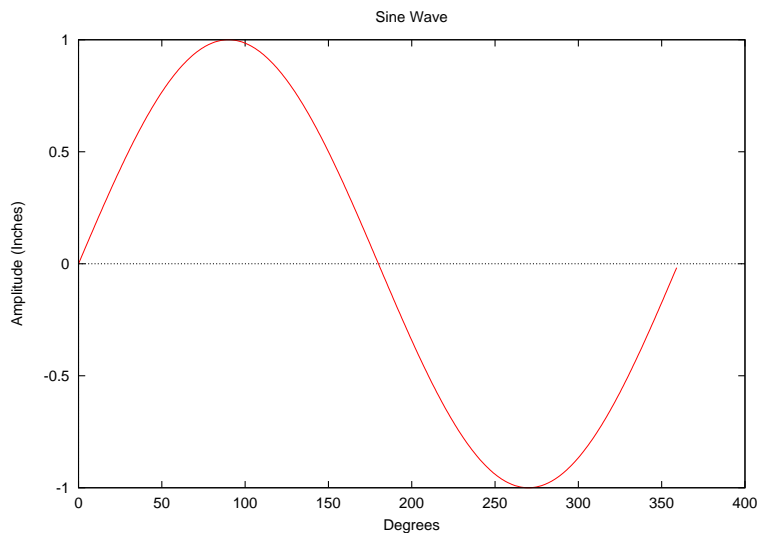
```
plot.title("title");
plot.label(PLOT_AXIS_X, "xlabel");
plot.label(PLOT_AXIS_Y, "ylabel");
```

Example

```
/* File: plotxyf.ch */
// Demonstrates the usage of the plotxyf() function.
#include <stdio.h>
#include <chplot.h>
#include <math.h>

int main() {
    string_t title="Sine Wave",           //Define
              xlabel="Degrees",
              ylabel="Amplitude (Inches)";
    string_t file;
    file = tmpnam(NULL);                 //Create temporary file.
    int i;
    FILE *out;
    out=fopen (file,"w");                //Write data to file.
    for(i=0;i<=359;i++) fprintf(out,"%i %f \n",i,sin(i*M_PI/180));
    fclose(out);
    plotxyf(file,title,xlabel,ylabel); //Call plotting function.
    remove(file);
}
```

Output



See Also

CPlot, **CPlot::dataFile**, **fplotxy()**, **fplotxyz()**, **plotxy()**, **plotxyz()**, **plotxyzf()**.

plotxyz

Synopsis

```
#include <chplot.h>
```

```
int plotxyz(double x[&], double y[&], array double &z, ... /* [int n] [int nx, int ny]
    [string_t title, string_t xlabel, string_t ylabel, string_t zlabel], [class CPlot *pl] */);
```

Syntax

```
plotxyz(x, y, z)
plotxyz(x, y, z, title, xlabel, ylabel, zlabel)
plotxyz(x, y, z, title, xlabel, ylabel, zlabel, &plot)
plotxyz(x, y, z, n)
plotxyz(x, y, z, n, title, xlabel, ylabel, zlabel)
plotxyz(x, y, z, n, title, xlabel, ylabel, zlabel, &plot)
plotxyz(x, y, z, nx, ny)
plotxyz(x, y, z, nx, ny, title, xlabel, ylabel, zlabel)
plotxyz(x, y, z, nx, ny, title, xlabel, ylabel, zlabel, &plot)
```

Purpose

Plot a 3D data set or initialize an instance of the **CPlot** class.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x A one-dimensional array of size n_x . The value of each element is used for the x-axis of the plot.

y A one-dimensional array of size n_y . The value of each element is used for the y-axis of the plot.

z If the data are for a 3D curve, *z* is a $m \times n_z$ dimensional array, and $n_x = n_y = n_z$. If the data are for a 3D surface or grid, *z* is a $m \times n_z$ dimensional array, and $n_z = n_x \cdot n_y$.

n The number of data points for a 3D curve.

nx The number of data points in the x-coordinates for a 3D surface.

ny The number of data points in the y-coordinates for a 3D surface.

title The title of the plot.

xlabel The x-axis label.

ylabel The y-axis label.

zlabel The y-axis label.

pl A pointer to an instance of the **CPlot** class.

Description

Plot a 3D data set or initialize a **CPlot** variable. For Cartesian data, *x* is a one-dimensional array of size n_x

and y is a one-dimensional array of size n_y . z can be of two different dimensions depending on what type of data is to be plotted. If the data is for a 3D curve, z is a $m \times n_z$ dimensional array, and $n_x = n_y = n_z$ for the optional argument n . If the data is for a 3D surface or grid, z is a $m \times n_z$ dimensional array, and $n_z = n_x \cdot n_y$ with optional arguments n_x and n_y . For cylindrical or spherical data x is a one dimensional array of size n_x (representing θ), y is a one dimensional array of size n_y (representing z or ϕ), and z is a $m \times n_z$ dimensional array (representing r). In all cases these data arrays can be of any supported data type. Conversion of the data to **double** type is performed internally. The *title*, *xlabel*, *ylabel*, and *zlabel* for the plot can also optionally be specified. A pointer to a plot structure can also be passed to this function. If a non-NULL pointer is passed, it will be initialized with the function parameters. The plot can then be displayed using the **CPlot::plotting** member function. If a previously initialized **CPlot** variable is passed, it will be re-initialized with the function parameters. If no pointer or a NULL pointer is passed, an internal **CPlot** variable will be used and the plot will be displayed without calling the **CPlot::plotting()** member function.

The following code segment

```
class CPlot plot;
plotxyz(x, y, z, "title", "xlabel", "ylabel", "zlabel", &plot);
```

is equivalent to

```
class CPlot plot;
plot.data3D(x, y, z);
plot.title("title");
plot.label(PLOT_AXIS_X, "xlabel");
plot.label(PLOT_AXIS_Y, "ylabel");
plot.label(PLOT_AXIS_Z, "zlabel");
```

The following code segment for plotting a 3D curve

```
class CPlot plot;
plotxyz(x, y, z, n);
```

is equivalent to

```
class CPlot plot;
plot.data3DCurve(x, y, z, n);
```

The following code segment for plotting a 3D surface

```
class CPlot plot;
plotxyz(x, y, z, nx, ny);
```

is equivalent to

```
class CPlot plot;
plot.data3DSurface(x, y, z, nx, ny);
```

Example 1

```

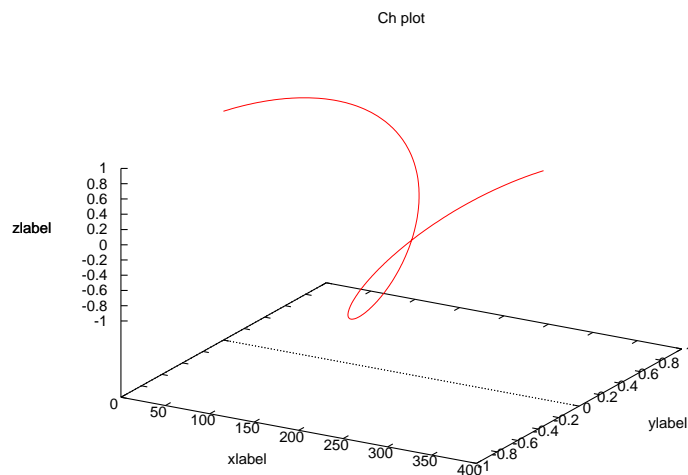
/* File: plotxyz.ch */
#include <math.h>
#include <chplot.h>

int main() {
    array double x[360], y[360], z[360];

    lindata(0, 360, x);
    y = sin(x*M_PI/180);
    z = cos(x*M_PI/180);
    plotxyz(x, y, z, "Ch plot", "xlabel", "ylabel", "zlabel");
    /* or
    plotxyz(x, y, z, 360, "Ch plot", "xlabel", "ylabel", "zlabel"); */
}

```

Output



Example 2

```

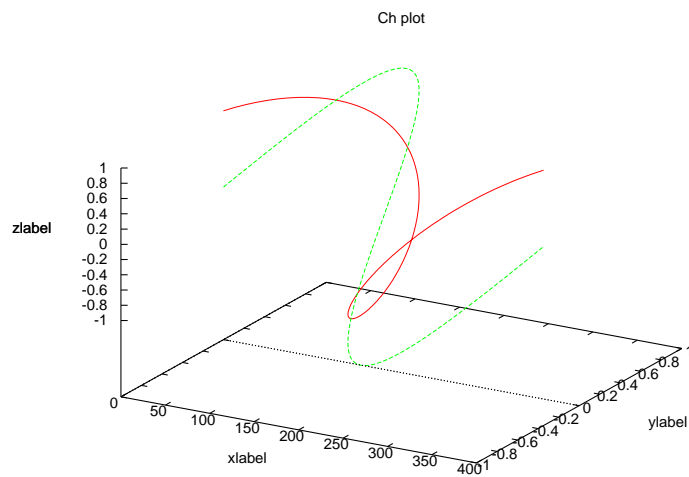
/* File: plotxyz_2.ch */
#include <math.h>
#include <chplot.h>

int main() {
    array double x[360], y[360], z[2][360];
    int i;

    lindata(0, 360, x);
    y = sin(x*M_PI/180);
    for(i=0; i<360; i++) {
        z[0][i] = cos(x[i]*M_PI/180);
        z[1][i] = y[i];
    }
    plotxyz(x, y, z, "Ch plot", "xlabel", "ylabel", "zlabel");
    /* or
    plotxyz(x, y, z, 360, "Ch plot", "xlabel", "ylabel", "zlabel"); */
}

```

Output



Example 3

```

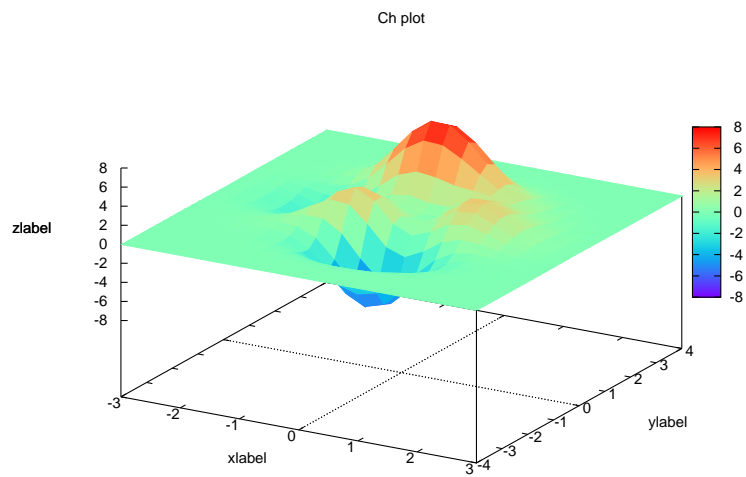
/* File: plotxyz_3.ch */
#include <math.h>
#include <chplot.h>

int main() {
    double x[20], y[30], z[600];
    int i, j;

    lndata(-3, 3, x);
    lndata(-4, 4, y);
    for(i=0; i<20; i++) {
        for(j=0; j<30; j++) {
            z[30*i+j] = 3*(1-x[i])*(1-x[i])*exp(-(x[i]*x[i])-(y[j]+1)*(y[j]+1))
                - 10*(x[i]/5 - x[i]*x[i]*x[i]-pow(y[j],5))*exp(-x[i]*x[i]-y[j]*y[j])
                - 1/3*exp(-(x[i]+1)*(x[i]+1)-y[j]*y[j]);
        }
    }
    plotxyz(x, y, z, "Ch plot", "xlabel", "ylabel", "zlabel");
    /* or
    plotxyz(x, y, z, 20, 30, "Ch plot", "xlabel", "ylabel", "zlabel"); */
}

```

Output

**See Also**

CPlot, **CPlot::data3D()**, **CPlot::data3DCurve()**, **CPlot::data3DSurface()**, **fplotxyz()**, **fplotxyz()**, **plotxy()**, **plotxyf()**, **plotxyzf()**.

plotxyzf

Synopsis

```
#include <chplot.h>
```

```
int plotxyzf(string_t file, ... /* [string_t title, string_t xlabel, string_t ylabel, string_t zlabel],
                             [class CPlot *pl] */);
```

Syntax

```
plotxyzf(file)
```

```
plotxyzf(file, title, xlabel, ylabel, zlabel)
```

```
plotxyzf(file, title, xlabel, ylabel, zlabel, &plot)
```

Purpose

Plot 3D data from a file or initialize an instance of the **CPlot** class.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

file The file containing the data to be plotted.

title The title of the plot.

xlabel The x-axis label.

ylabel The y-axis label.

zlabel The z-axis label.

pl A pointer to an instance of the **CPlot** class.

Description

Plot 3D data from a file or initialize a **CPlot** variable. The data file should be formatted with each data point on a separate line. 3D data is specified by three values per data point. For a 3D grid or surface data, each row is separated in the data file by a blank line. For example, a 3 x 3 grid would be represented as follows:

```
x1  y1  z1
x1  y2  z2
x1  y3  z3

x2  y1  z4
x2  y2  z5
x2  y3  z6

x3  y1  z7
x3  y2  z8
x3  y3  z9
```

This function can only be used to plot Cartesian grid data. The *title*, *xlabel*, *ylabel*, and *zlabel* for the plot can also optionally be specified. A pointer to a plot structure can also be passed to this function. If a non-NULL pointer is passed, it will be initialized with the function parameters. The plot can then be displayed

using the **CPlot::plotting** member function. If a previously initialized **CPlot** variable is passed, it will be re-initialized with the function parameters. If no pointer or a NULL pointer is passed, an internal **CPlot** variable will be used and the plot will be displayed without calling the **CPlot::plotting()** member function. Two empty lines in the data file will cause a break in the plot. Multiple curves or surfaces can be plotted in this manner however, the plot style will be the same for all curves or surfaces.

The following code segment

```
class CPlot plot;
plotxyzf("datafile", "title", "xlabel", "ylabel", "zlabel", &plot);
```

is equivalent to

```
class CPlot plot;
plot.dimension(3);
plot.dataFile("datafile");
plot.title("title");
plot.label(PLOT_AXIS_X, "xlabel");
plot.label(PLOT_AXIS_Y, "ylabel");
plot.label(PLOT_AXIS_Z, "zlabel");
```

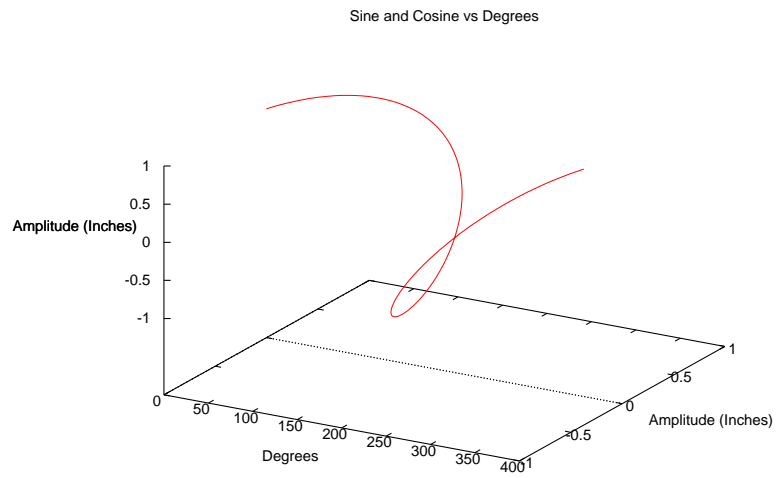
Example

```
/* File: plotxyzf.ch */
#include <chplot.h>
#include <stdio.h>
#include <math.h>

int main() {
    string_t title="Sine and Cosine vs Degrees",
              xlabel="Degrees",
              ylabel="Amplitude (Inches)",
              zlabel="Amplitude (Inches)";
    string_t file;
    file = tmpnam(file); //temporary file
    int i;
    FILE *out;

    out=fopen (file,"w"); // write data to file
    for(i=0;i<=359;i++) fprintf(out,"%i %f %f\n",i,sin(i*M_PI/180),cos(i*M_PI/180));
    fclose(out);
    plotxyzf(file,title,xlabel,ylabel,zlabel); // call plotting function:
    remove(file);
}
```

Output



See Also

CPlot, CPlot::dataFile(), fplotxy(), fplotxyz(), plotxy(), plotxyz(), plotxyz().

Chapter 3

Shell Functions — <chshell.h>

Header file **chshell.h** contains the definitions of Ch shell functions. The following functions are designed for Ch shell programming.

Functions

Function	Description
chinfo()	Get information about the Ch language environment.
iskey()	Find out if a string is a keyword.
isstudent()	Find out if the Ch language environment is a student version.
isvar()	Find the property of a string.
sizeofelement()	Find the size of an element of an array.

Constants

The following macros are defined in **chshell.h**.

Constant	Description
CH_NOTVAR	Defined as 0, indicating the string is not a variable
CH_SYSTEMCONST	Indicating the string is a system constant.
CH_SYSTEMVAR	Indicating the string is a system variable.
CH_SYSTEMFUN	Indicating the string is a system function.
CH_EXTERNVAR	Indicating the string is a external variable.

Structure

Structure **chinfo_t** shall contain the following members

```
Struct chinfo_t {  
  
    string_t edition;           // "Professional" or "Student"  
    string_t releasedate;      // "October 1 1999"  
    string_t version;          // "1.0"  
    unsigned intvermajor;      // 1  
    unsigned intverminor;      // 0  
  
}
```

Datatype

The data type **chinfo_t** holds the parameters of the Ch language environment described above.

chinfo

Synopsis

```
#include <chshell.h>
int chinfo(chinfo_t *info);
```

Purpose

Get information about the Ch language environment.

Return Value

This function returns 0 if it succeeds and -1 if it fails.

Parameters

info A pointer to a structure containing the edition, release date, version, major version number, minor version number, micro version number, and build number.

Description

This function gets the information about the Ch language environment including the edition, release date, version, , major version number, minor version number, micro version number, and build number.

Example

```
#include<chshell.h>

int main() {
    chinfo_t info;

    if(chinfo(&info)==-1)
        printf("Error: chinfo() error\n");
    printf("info.edition = %s\n", info.edition);
    printf("info.releasedate = %s\n", info.releasedate);
    printf("info.version = %s\n", info.version);
    printf("info.vermajor = %d\n", info.vermajor);
    printf("info.verminor = %d\n", info.verminor);
    printf("info.vermicro = %d\n", info.vermicro);
    printf("info.verbuild = %d\n", info.verbuild);
    printf("Ch %s Edition v%d.%d.%d.%d %s released by %s\n", info.edition,
        info.vermajor, info.verminor, info.vermicro, info.verbuild,
        info.version, info.releasedate);
}
```

Output

```
info.edition = Professional
info.releasedate = March 1, 2007
info.version =
info.vermajor = 5
info.verminor = 5
info.vermicro = 0
info.verbuild = 13151
Ch Professional Edition v5.5.0.13151 released by March 1, 2007
```

See Also
isstudent().

iskey

Synopsis

```
#include <chshell.h>
int iskey(string_t name);
```

Purpose

Find out if a string is a keyword.

Return Value

This function returns 1 if the string is a generic function, 2 if it is a keyword starting with alphabetical letter, 3 if it is a other keyword, 0 otherwise.

Parameters

name The input string.

Description

This function determines if the input string *name* is not a keyword.

Example

```
/* access element of a string */
#include <stdio.h>
#include <string.h>
#include <chshell.h>

int g;
int main() {
    int i;

    printf("iskey(\"sin\") = %d\n", iskey("sin"));
    printf("iskey(\"int\") = %d\n", iskey("int"));
    printf("iskey(\"=\") = %d\n", iskey("="));
    printf("iskey(\"unknown\") = %d\n", iskey("unknown"));

    printf("isstudent() = %d\n", isstudent());

    printf("isvar(\"i\") == CH_NOTVAR = %d\n", isvar("i") == CH_NOTVAR);
    printf("isvar(\"NaN\") == CH_SYSTEMCONST = %d\n", isvar("NaN") == CH_SYSTEMCONST);
    printf("isvar(\"_path\") == CH_SYSTEMVAR = %d\n", isvar("_path") == CH_SYSTEMVAR);
    printf("isvar(\"g\") == CH_EXTERNVAR = %d\n", isvar("g") == CH_EXTERNVAR);
    printf("isvar(\"isvar\") == CH_EXTERNFUN = %d\n", isvar("isvar") == CH_EXTERNFUN);
    printf("isvar(\"unknown\") = %d\n", isvar("unknown"));
}
```

Output

```
iskey("sin") = 1
iskey("int") = 2
iskey("=") = 3
iskey("unknown") = 0
isstudent() = 0
isvar("i") == CH_NOTVAR = 1
isvar("NaN") == CH_SYSTEMCONST = 1
```

```
isvar("_path") == CH_SYSTEMVAR = 1
isvar("g") == CH_EXTERNVAR = 1
isvar("isvar") == CH_EXTERNFUN = 1
isvar("unknown") = 0
```

See Also

isenv(), isnum(), isstudent(), isvar().

isstudent

Synopsis

```
#include <chshell.h>  
int isstudent(void);
```

Purpose

Find out if the Ch language environment is a student version.

Return Value

This function returns 1 if it is a student version and 0 if it is a professional version.

Description

This function determines if the current Ch language environment is a student version.

Example

See `iskey()`.

See Also

`isenv()`, `isnum()`, `iskey()`, `isvar()`.

isvar

Synopsis

```
#include <chshell.h>
```

```
int isvar(string_t name);
```

Purpose

Find the property of a string.

Return Value

This function returns one of the following values:

CH_NOTVAR The string is not a variable.

CH_SYSTEMCONST The string is a system constant.

CH_SYSTEMVAR The string is a system variable.

CH_SYSTEMFUN The string is a system function.

CH_EXTERNVAR The string is a external variable.

Parameters

name The input string.

Description

This function returns an integer to indicate the characteristic of the input string *name*.

Example

See `iskey()`

See Also

`isenv()`, `iskey()`, `isnum()`, `isstudent()`.

sizeofelement

Synopsis

```
#include <chshell.h>
```

```
int sizeofelement(int etype);
```

Purpose

Find the size of elements of an array.

Return Value

This function returns the number of characters of the data type represented in integral value.

Parameters

etype An integer indicating the data type of the element of array.

Description

This function finds the size of an element of an array. The argument *etype* is the data type of the array, and it can be obtained by calling built-in function **elementtype()**.

Example

```
/* sizeofelement() returns the size of the data type of the
   its argument. If it is an array, the data type of the array element
   is used */

#include <stdio.h>

void func1(double a[&]) {
    int size;

    size =sizeofelement(elementtype(a));
    printf("sizeofelement(a) = %d\n", size);
}

void func2(array double &a) {
    int size;

    size =sizeofelement(elementtype(a));
    printf("sizeofelement(a) = %d\n", size);
}

int main() {
    int    a1[3][4], b1[3];
    double a2[3][4], b2[3];
    func1(b1);
    func1(b2);
    func2(a1);
    func2(b1);
    func2(a2);
    func2(b2);
}
```

Output

```
sizeofelement(a) = 4
sizeofelement(a) = 8
```

```
sizeofelement(a) = 4  
sizeofelement(a) = 4  
sizeofelement(a) = 8  
sizeofelement(a) = 8
```

See Also

elementtype().

Chapter 4

Complex Functions — `<complex.h>`

The header **complex.h** defines macros and declares functions that support complex arithmetic in the latest ISO C standard. Each synopsis specifies a family of functions consisting of a principal function with one or more double complex parameters and a double complex, or double, return value; and other functions with the same name but with `f` and `l` suffixes which are corresponding functions with float and long double parameters and return values.

Functions

The following functions are declared in the **complex.h** header file.

Function	Description
cabs()	Calculate complex absolute value.
ccos()	Calculate complex arc cosine.
ccosh()	Calculate complex arc hyperbolic cosine.
carg()	Calculate complex argument.
casin()	Calculate complex arc sine.
casinh()	Calculate complex arc hyperbolic sine.
catan()	Calculate complex arc tangent.
catanh()	Calculate complex arc hyperbolic tangent.
ccos()	Calculate complex cosine.
ccosh()	Calculate complex hyperbolic cosine.
cexp()	Calculate complex exponential.
cimag()	Obtain the imaginary part of a complex number.
clog()	Calculate complex logarithm.
conj()	Calculate complex conjugate.
cpow()	Calculate complex power of two variables.
creal()	Obtain the real part of a complex number.
csin()	Calculate complex sine.
csinh()	Calculate complex hyperbolic sine.
ctan()	Calculate complex tangent.
ctanh()	Calculate complex hyperbolic tangent.
iscnan()	Test if the argument is a complex Not-a-Number.

Macros

The following macro is defined by the **complex.h** header file.

Macro	Description
I	Imaginary number I expands to complex(0.0, 1.0) .

Portability

This header has no known portability problem.

Differences between C and Ch

Function **conj** is a built-in function in Ch.

cabs

Synopsis

```
#include <complex.h>
double cabs(double complex z);
float cabsf(float complex z);
long double cabsl(long double complex z);
```

Purpose

Calculates an absolute value of a complex number.

Return Value

The **cabs** functions return the complex absolute value.

Parameters

z Complex argument.

Description

The **cabs** functions compute the complex absolute value (also called norm, modulus or magnitude) of *z*.

It is recommended that the polymorphic generic function **abs()**, instead of the **cabs()** function, be used to compute the complex absolute value of *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
    double complex z = complex(1, 2);
    printf("cabs(z) = %f\n", cabs(z));
}
```

Output

```
cabs(z) = 2.236068
```

See Also

abs().

cacos

Synopsis

```
#include <complex.h>
double complex cacos(double complex z);
float complex cacosf(float complex z);
long double complex cacosl(long double complex z);
```

Purpose

Calculates an arc cosine of a complex number.

Return Value

The **cacos** functions return the complex arc cosine value, in the range of a strip mathematically unbound along the imaginary axis, and in the interval $[0, \pi]$ along the real axis.

Parameters

z Complex argument.

Description

The **cacos** functions compute the complex arc cosine of *z*, with branch cuts outside the interval $[-1, 1]$ along the real axis.

It is recommended that the polymorphic generic function **acos()** instead of the **cacos()** function, be used to compute the complex arc cosine of *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
    double complex z = complex(1, 2);
    printf("cacos(z) = %f\n", cacos(z));
}
```

Output

```
cacos(z) = complex(1.143718, -1.528571)
```

See Also

cos(), **ccos()**, **cacosh()**, **ccosh()**.

cacosh

Synopsis

```
#include <complex.h>
double complex cacosh(double complex z);
float complex cacoshf(float complex z);
long double complex cacoshl(long double complex z);
```

Purpose

Calculates an arc hyperbolic cosine of a complex number.

Return Value

The **cacosh** functions return the complex arc hyperbolic cosine value, in the range of a half-strip of non-negative values along the real axis, and in the interval $[-i\pi, i\pi]$ along the imaginary axis.

Parameters

z Complex argument.

Description

The **cacosh** functions compute the complex arc cosine of z , with branch a cut at values less than 1 along the real axis.

It is recommended that the polymorphic generic function **acosh()**, instead of the **cacosh()** function, be used to compute the complex arc hyperbolic cosine of z .

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
    double complex z = complex(1, 2);
    printf("cacosh(z) = %f\n", cacosh(z));
}
```

Output

```
cacosh(z) = complex(1.528571, 1.143718)
```

See Also

cos(), **cacos()**, **ccos()**, **ccosh()**.

carg

Synopsis

```
#include <complex.h>
double carg(double complex z);
float cargf(float complex z);
long double cargl(long double complex z);
```

Purpose

Calculates an argument of a complex number.

Return Value

The **carg** functions return the value of the argument in the range $[-\pi, \pi]$.

Parameters

z Complex argument.

Description

The **carg** functions compute the argument (also called phase angle) of *z*, with a branch cut along the negative real axis.

It is recommended that the polymorphic generic function **arg()**, instead of the **carg()** function, be used to compute the argument of a complex number *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
    int i=7;
    double f = 30;
    double d = -10;
    complex z = complex(3, 4);
    printf("carg(%d) = %f radian \n", i, carg(i));
    printf("carg(%f) = %f radian \n", f, carg(f));
    printf("carg(%lf) = %f radian \n", d, carg(d));
    printf("carg(%f) = %f radian \n", z, carg(z));
}
```

Output

```
carg(7) = 0.000000 radian
carg(30.000000) = 0.000000 radian
carg(-10.000000) = 3.141593 radian
carg(complex(3.000000,4.000000)) = 0.927295 radian
```

See Also

casin

Synopsis

```
#include <complex.h>
double complex casin(double complex z);
float complex casinf(float complex z);
long double complex casinl(long double complex z);
```

Purpose

Calculates an arc sine of a complex number.

Return Value

The **casin** functions return the complex arc sine value, in the range of a strip mathematically unbounded along the imaginary axis, and in the interval $[-\pi/2, \pi/2]$ along the real axis.

Parameters

z Complex argument.

Description

The **casin** functions compute the complex arc sine of *z*, with branch cuts outside the interval $[-1, 1]$ along the real axis.

It is recommended that the polymorphic generic function **asin()**, instead of the **casin()** function, be used to compute the complex arc sine of *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
    double complex z = complex(1, 2);
    printf("casin(z) = %f\n", casin(z));
}
```

Output

```
casin(z) = complex(0.427079,1.528571)
```

See Also

sin(), **csin()**, **casinh()**, **csinh()**.

casinh

Synopsis

```
#include <complex.h>
double complex casinh(double complex z);
float complex casinhf(float complex z);
long double complex casinhl(long double complex z);
```

Purpose

Calculates an arc hyperbolic sine of a complex number.

Return Value

The **casinh** functions return the complex arc hyperbolic sine value, in the range of a strip mathematically unbounded along the real axis, and in the interval $[-i\pi/2, i\pi/2]$ along the imaginary axis.

Parameters

z Complex argument.

Description

The **casinh** functions compute the complex arc hyperbolic sine of *z*, with branch cuts outside the interval $[-i, i]$ along the imaginary axis.

It is recommended that the polymorphic generic function **asinh()**, instead of the **casinh()** function, be used to compute the complex arc hyperbolic sine of *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
    double complex z = complex(1, 2);
    printf("casinh(z) = %f\n", casinh(z));
}
```

Output

```
casinh(z) = complex(1.469352,1.063440)
```

See Also

sin(), **casin()**, **csin()**, **csinh()**.

catan

Synopsis

```
#include <complex.h>
double complex catan(double complex z);
float complex catanf(float complex z);
long double complex catanl(long double complex z);
```

Purpose

Calculates an arc tangent of a complex number.

Return Value

The **catan** functions return the complex arc tangent value, in the range of a strip mathematically unbounded along the imaginary axis, and in the interval $[-\pi/2, \pi/2]$.

Parameters

z Complex argument.

Description

The **catan** functions compute the complex arc tangent of z , with branch cuts outside the interval $[-i, i]$ along the imaginary axis.

It is recommended that the polymorphic generic **atan()** function, instead of the **catan()** function, be used to compute the complex arc tangent of z .

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
    double complex z = complex(1, 2);
    printf("catan(z) = %f\n", catan(z));
}
```

Output

```
catan(z) = complex(1.338973,0.402359)
```

See Also

tan(), **ctan()**, **catanh()**, **ctanh()**.

catanh

Synopsis

```
#include <complex.h>
double complex catanh(double complex z);
float complex catanhf(float complex z);
long double complex catanhl(long double complex z);
```

Purpose

Calculates an arc hyperbolic tangent of a complex number.

Return Value

The **catanh** functions return the complex arc hyperbolic tangent value, in the range of a strip mathematically unbounded along the real axis, and in the interval $[-i\pi/2, i\pi/2]$ along the imaginary axis.

Parameters

z Complex argument.

Description

The **catanh** functions compute the complex arc hyperbolic tangent of *z*, with branch cuts outside the interval $[-1, 1]$ along the real axis.

It is recommended that the polymorphic generic **atanh()** function, instead of the **catanh()** functions, be used to compute the complex arc hyperbolic tangent of *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
    double complex z = complex(1, 2);
    printf("catanh(z) = %f\n", catanh(z));
}
```

Output

```
catanh(z) = complex(0.173287,1.178097)
```

See Also

tan(), **catan()**, **ctan()**, **ctanh()**.

CCOS

Synopsis

```
#include <complex.h>
double complex ccos(double complex z);
float complex ccosf(float complex z);
long double complex ccosl(long double complex z);
```

Purpose

Calculates a cosine of a complex number.

Return Value

The **ccos** functions return the complex cosine.

Parameters

z Complex argument.

Description

The **cacos** functions compute the complex cosine of *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
    double complex z = complex(1, 2);
    printf("ccos(z) = %f\n", ccos(z));
}
```

Output

```
ccos(z) = complex(2.032723,-3.051898)
```

See Also

cos(), **cacos()**, **cacosh()**, **ccosh()**.

ccosh

Synopsis

```
#include <complex.h>
double complex ccosh(double complex z);
float complex ccoshf(float complex z);
long double complex ccoshl(long double complex z);
```

Purpose

Calculates a hyperbolic cosine of a complex number.

Return Value

The **ccosh** functions return the complex hyperbolic cosine.

Parameters

z Complex argument.

Description

The **cacosh** functions compute the complex hyperbolic cosine of *z*.

It is recommended that the polymorphic generic function **cosh()**, instead of the **ccosh()** function, be used to compute the complex hyperbolic cosine of *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
    double complex z = complex(1, 2);
    printf("ccosh(z) = %f\n", ccosh(z));
}
```

Output

```
ccosh(z) = complex(-0.642148, 1.068607)
```

See Also

cos(), **acos()**, **ccos()**, **cacosh()**.

cexp

Synopsis

```
#include <complex.h>
double complex cexp(double complex z);
float complex cexpf(float complex z);
long double complex cexpl(long double complex z);
```

Purpose

Calculates a base- e exponential of a complex number.

Return Value

The **cexp** functions return the complex base- e exponential value.

Parameters

z Complex argument.

Description

The **cexp** functions compute the complex base- e exponential of z .

It is recommended that the polymorphic generic function **exp()**, instead of the **cexp()** function, be used to compute the base- e exponential of complex number z .

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
    double complex z = complex(1, 2);
    printf("cexp(z) = %f\n", cexp(z));
}
```

Output

```
cexp(z) = complex(-1.131204, 2.471727)
```

See Also

exp().

cimag

Synopsis

```
#include <complex.h>
double complex cimag(double complex z);
float complex cimagf(float complex z);
long double complex cimagl(long double complex z);
```

Purpose

Get the imaginary part of a complex number.

Return Value

The **cimag** functions return the imaginary part value (as a real).

Parameters

z Complex argument.

Description

The **cimag** functions compute the imaginary part of *z*.

It is recommended that the polymorphic generic function **imag()**, instead of the **cimag()** function, be used to obtain the imaginary part of a complex number *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
    double complex z = complex(1, 2);
    printf("cimag(z) = %f\n", cimag(z));
}
```

Output

```
cimag(z) = 2.000000
```

See Also

imag().

clog

Synopsis

```
#include <complex.h>
double complex clog(double complex z);
float complex clogf(float complex z);
long double complex clogl(long double complex z);
```

Purpose

Calculates a natural base- e logarithm of a complex number.

Return Value

The **clog** functions return the complex natural base- e logarithm value, in the range of a strip mathematically unbounded along the real axis, and in the interval $[-i\pi, i\pi]$ along the imaginary axis.

Parameters

z Complex argument.

Description

The **clog** functions compute the complex natural base- e logarithm of z , with a branch cut along the negative real axis.

It is recommended that the polymorphic generic **log()** functions, instead of the **clog()** functions, be used to compute the natural base- e logarithm of the complex number z .

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
    double complex z = complex(1, 2);
    printf("clog(z) = %f\n", clog(z));
}
```

Output

```
clog(z) = complex(0.628609,1.107149)
```

See Also

log().

conj

Synopsis

```
#include <complex.h>
double complex conj(double complex z);
float complex conjf(float complex z);
long double complex conjl(long double complex z);
```

Purpose

Calculates the conjugate of a complex number.

Return Value

The **conj** functions return the complex conjugate value.

Parameters

z Complex argument.

Description

The **conj** functions compute the complex conjugate of *z*, by reversing the sign of its imaginary part.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
    double complex z = complex(1, 2);
    printf("conj(z) = %f\n", conj(z));
}
```

Output

```
conj(z) = complex(1.000000,-2.000000)
```

See Also

cpow

Synopsis

```
#include <complex.h>
```

```
double complex cpow(double complex z, double complex y);
```

```
float complex cpowf(float complex z, float complex y);
```

```
long double complex cpowl(long double complex z, long double complex y);
```

Purpose

Calculates the power function of a complex number.

Return Value

The **cpow** functions return the complex power function value.

Parameters

z Complex argument.

y Complex argument.

Description

The **cpow** functions compute the complex power function x^y , with a branch cut for the first parameter along the negative real axis.

It is recommended that the polymorphic generic function **pow()**, instead of the **cpow()** function, be used to compute the power function of a complex number *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
    double complex z = complex(1, 2);
    double complex y = complex(3, 4);
    printf("cpow(z,y) = %f\n", cpow(z,y));
}
```

Output

```
cpow(z,y) = complex(0.129010,0.033924)
```

See Also

pow().

creal

Synopsis

```
#include <complex.h>
double creal(double complex z);
float crealf(float complex z);
long double creall(long double complex z);
```

Purpose

Get the real part of a complex number.

Return Value

The **creal** functions return the real part value.

Parameters

z Complex argument.

Description

The **creal** functions compute the real part of *z*.

It is recommended that the polymorphic generic function **real()**, instead of the **creal()** function, be used to compute the real part of a complex number *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
    double complex z = complex(1, 2);
    printf("conj(z) = %f\n", conj(z));
}
```

Output

```
conj(z) = complex(1.000000,-2.000000)
```

See Also

real().

csin

Synopsis

```
#include <complex.h>
```

```
double complex csin(double complex z);
```

```
float complex csinf(float complex z);
```

```
long double complex csinl(long double complex z);
```

Purpose

Calculates a sine of a complex number.

Return Value

The **csin** functions return the complex sine.

Parameters

z Complex argument.

Description

The **csin** functions compute the complex sine of *z*.

It is recommended that the polymorphic generic **sin()** functions, instead of the **csin()** functions, be used to compute the complex sine of *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
    double complex z = complex(1, 2);
    printf("csin(z) = %f\n", csin(z));
}
```

Output

```
csin(z) = complex(3.165779,1.959601)
```

See Also

sin(), **casin()**, **casinh()**, **csinh()**.

csinh

Synopsis

```
#include <complex.h>
double complex csinh(double complex z);
float complex csinhf(float complex z);
long double complex csinhl(long double complex z);
```

Purpose

Calculates a hyperbolic sine of a complex number.

Return Value

The **csinh** functions return the complex hyperbolic sine.

Parameters

z Complex argument.

Description

The **csinh** functions compute the complex hyperbolic sine of *z*.

It is recommended that the polymorphic generic **sinh()** functions, instead of the **csinh()** functions, be used to compute the complex hyperbolic sine of *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
    double complex z = complex(1, 2);
    printf("csinh(z) = %f\n", csinh(z));
}
```

Output

```
csinh(z) = complex(-0.489056, 1.403119)
```

See Also

sin(), **casin()**, **csin()**, **casinh()**.

csqrt

Synopsis

```
#include <complex.h>
double complex csqrt(double complex z);
float complex csqrtf(float complex z);
long double complex csqrtl(long double complex z);
```

Purpose

Calculates a square root of a complex number.

Return Value

The **csqrt** functions return the complex square root value, in the range of the right half plane (including the imaginary axis).

Parameters

z Complex argument.

Description

The **csqrt** functions compute the complex square root of *z*, with a branch cut along the negative real axis.

It is recommended that the polymorphic generic **sqrt()** functions, instead of the **csqrt()** function, be used to compute the complex square root of *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
    double complex z = complex(1, 2);
    printf("csqrt(z) = %f\n", csqrt(z));
}
```

Output

```
csqrt(z) = complex(1.272020,0.786151)
```

See Also

ctan

Synopsis

```
#include <complex.h>
double complex ctan(double complex z);
float complex ctanf(float complex z);
long double complex ctanl(long double complex z);
```

Purpose

Calculates a tangent of a complex number.

Return Value

The **ctan** functions return the complex tangent.

Parameters

z Complex argument.

Description

The **ctan** functions compute the complex tangent of *z*.

It is recommended that the polymorphic generic **tan()** functions, instead of the **ctan()** functions, be used to compute the complex tangent of *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
    double complex z = complex(1, 2);
    printf("ctan(z) = %f\n", ctan(z));
}
```

Output

```
ctan(z) = complex(0.033813,1.014794)
```

See Also

tan(), **catan()**, **catanh()**, **ctanh()**.

ctanh

Synopsis

```
#include <complex.h>
double complex ctanh(double complex z);
float complex ctanhf(float complex z);
long double complex ctanhl(long double complex z);
```

Purpose

Calculates a hyperbolic tangent of a complex number.

Return Value

The **ctanh** functions return the complex hyperbolic tangent.

Parameters

z Complex argument.

Description

The **ctanh** functions compute the complex hyperbolic tangent of *z*.

It is recommended that the polymorphic generic **tanh()** functions, instead of the **ctanh()** functions, be used to compute the complex hyperbolic tangent of *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
    double complex z = complex(1, 2);
    printf("ctanh(z) = %f\n", ctanh(z));
}
```

Output

```
ctanh(z) = complex(1.166736,-0.243458)
```

See Also

tan(), **catan()**, **ctan()**, **catanh()**.

iscnan

Synopsis

```
#include <complex.h>
int iscnan(complex z);
```

Purpose

Tests whether the complex number is a complex Not-a-Number, ComplexNaN.

Return Value

The **iscnan** function returns 1 value when z is a complex Not-a-Number.

Parameters

z complex number.

Description

The **iscnan** function tests whether the complex number is a complex Not-a-Number.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
    double complex z1 = complex(1, 2);
    double complex z2 = ComplexNaN;
    complex z3 = complex(1,2);
    complex z4 = ComplexNaN;
    printf("iscnan(z1) = %d\n", iscnan(z1));
    printf("iscnan(z2) = %d\n", iscnan(z2));
    printf("iscnan(z3) = %d\n", iscnan(z3));
    printf("iscnan(z4) = %d\n", iscnan(z4));
    printf("iscnan(ComplexNaN) = %d\n", iscnan(ComplexNaN));
    printf("iscnan(ComplexInf) = %d\n", iscnan(ComplexInf));
}
```

Output

```
iscnan(z1) = 0
iscnan(z2) = 1
iscnan(z3) = 0
iscnan(z4) = 1
iscnan(ComplexNaN) = 1
iscnan(ComplexInf) = 0
```

See Also

Chapter 5

Character Handling — `<ctype.h>`

The header `ctype.h` declares several functions useful for testing and mapping characters. In all cases the argument is of `int` type, the value of which shall be representable as an **unsigned character** or shall equal the value of the macro `EOF`, defined in header file `stdio.h` to represent end-of-file. If the argument has any other value, the behavior is undefined. These functions shall return a nonzero value for true or zero for false.

The behavior of these functions is affected by the current locale. Those functions that have locale-specific aspects only when not in the “C” locale are noted below. The term *printing character* refers to a member of locale-specific set of characters, each of which occupies one printing position on a display device; the term *control character* refers to a member of a locale-specific set of characters that are not printing characters.

Functions

Function	Description
<code>isalnum()</code>	Tests if an argument is a digit or alphabetic character.
<code>isalpha()</code>	Tests if an argument is an alphabetic character.
<code>iscntrl()</code>	Tests if an argument is a control character.
<code>isdigit()</code>	Tests if an argument is a decimal-digit character.
<code>isgraph()</code>	Tests if an argument is a printing character.
<code>islower()</code>	Tests if an argument is a lowercase character.
<code>isprint()</code>	Tests if an argument is a printing character.
<code>ispunct()</code>	Tests if an argument is a punctuation character.
<code>isspace()</code>	Tests if an argument is a white-space character.
<code>isupper()</code>	Tests if an argument is an uppercase character.
<code>isxdigit()</code>	Tests if an argument is a hexadecimal-digit character.
<code>tolower()</code>	Converts an uppercase character to a lowercase character.
<code>toupper()</code>	Converts a lowercase character to an uppercase character.

Portability

This header has no known portability problem.

isalnum

Synopsis

```
#include <ctype.h>
int isalnum(int c);
```

Purpose

Tests whether the character is a digit or alphabetic character.

Return Value

The **isalnum** function returns a nonzero value when *c* is a digit or alphabetic character.

Parameters

c Character.

Description

The **isalnum** function tests for any character for which **isalpha** or **isdigit** is true.

Example

```
/* a sample program testing isalnum() */
#include <ctype.h>
#include <stdio.h>

int main() {
    char c='A';
    printf("isalnum('%c') returns %d\n", c, isalnum(c));
    if(isalnum(c) > 0)
        printf("'%c' is an alphanumeric character.\n", c);
    else
        printf("'%c' is not an alphanumeric character.\n", c);
    c = '4';
    printf("isalnum('%c') returns %d\n", c, isalnum(c));
    if(isalnum(c) > 0)
        printf("'%c' is an alphanumeric character.\n", c);
    else
        printf("'%c' is not an alphanumeric character.\n", c);
}
```

Output

```
isalnum('A') returns 1
'A' is an alphanumeric character.
isalnum('4') returns 1
'4' is an alphanumeric character.
```

See Also

isalpha

Synopsis

```
#include <ctype.h>
int isalpha(int c);
```

Purpose

Tests whether the character is an alphabetic character.

Return Value

The **isalpha** function returns a non-zero value when *c* is an alphabetic character.

Parameters

c Character.

Description

The **isalpha** function tests for any character for which **isupper** or **islower** is true, or any character that is one of a locale-specific set of alphabetic characters for which none of **isctrl**, **isdigit**, **ispunct**, or **isspace** is true. In the “C” locale, **isalpha** returns true only when **isupper** or **islower** is true.

Example

```
/* a sample program testing isalpha() */
#include <ctype.h>
#include <stdio.h>

int main() {
    char c='A';
    printf("isalpha('%c') returns %d\n", c, isalpha(c));
    if(isalpha(c) > 0)
        printf("'c' is an alphabetical character.\n", c);
    else
        printf("'c' is not an alphabetical character.\n",c);
    c = ' ';
    printf("isalpha('%c') returns %d\n", c, isalpha(c));
    if(isalpha(c) > 0)
        printf("'c' is an alphabetical character.\n", c);
    else
        printf("'c' is not an alphabetical character.\n",c);
}
```

Output

```
isalpha('A') returns 1
'A' is an alphabetical character.
isalpha(' ') returns 0
' ' is not an alphabetical character.
```

See Also

isctrl

Synopsis

```
#include <ctype.h>
int isctrl(int c);
```

Purpose

Tests whether the character is a control character.

Return Value

The **isctrl** function returns a non-zero value when *c* is a control character.

Parameters

c Character.

Description

The **isctrl** function test for any control character.

Example

```
/* a sample program testing isctrl() */
#include <ctype.h>
#include <stdio.h>

int main() {
    char c = 'A';
    printf("isctrl('%c') returns %d\n", c, isctrl(c));
    if(isctrl(c) > 0)
        printf("%c is a control character.\n", c);
    else
        printf("%c is not a control character.\n",c);
}
```

Output

```
isctrl('A') returns 0
'A' is not a control character.
```

See Also

isdigit

Synopsis

```
#include <ctype.h>
int isdigit(int c);
```

Purpose

Tests whether the character is a decimal-digit character.

Return Value

The **isdigit** function returns a non-zero value when *c* is a decimal-digit character.

Parameters

c Character.

Description

The **isdigit** function tests for any decimal-digit character.

Example

```
/* a sample program testing isdigit() */
#include <ctype.h>
#include <stdio.h>

int main() {
    char c = 'A';
    printf("isdigit('%c') returns %d\n", c, isdigit(c));
    if(isdigit(c) > 0)
        printf("%c is a digit.\n", c);
    else
        printf("%c is not a digit.\n", c);
    c = '1';
    printf("isdigit('%c') returns %d\n", c, isdigit(c));
    if(isdigit(c) > 0)
        printf("%c is a digit.\n", c);
    else
        printf("%c is not a digit.\n", c);
}
```

Output

```
isdigit('A') returns 0
'A' is not a digit.
isdigit('1') returns 1
'1' is a digit.
```

See Also

isgraph

Synopsis

```
#include <ctype.h>
int isgraph(int c);
```

Purpose

Tests whether the character is a printing character.

Return Value

The **isgraph** function returns a non-zero value when *c* is a printing character.

Parameters

c Character.

Description

The **isgraph** function tests for any printing character, except space (' ').

Example

```
/* a sample program testing isgraph() */
#include <ctype.h>
#include <stdio.h>

int main() {
    char c = 'A';
    printf("isgraph('%c') returns %d\n", c, isgraph(c));
    if(isgraph(c) > 0)
        printf("'%'c' is a printable character.\n", c);
    else
        printf("'%'c' is not a printable character or it is a space.\n", c);
    c = ' ';
    printf("isgraph('%c') returns %d\n", c, isgraph(c));
    if(isgraph(c) > 0)
        printf("'%'c' is a printable character.\n", c);
    else
        printf("'%'c' is not a printable character or it is a space.\n", c);
}
```

Output

```
isgraph('A') returns 1
'A' is a printable character.
isgraph(' ') returns 0
' ' is not a printable character or it is a space.
```

See Also

islower

Synopsis

```
#include <ctype.h>
int islower(int c);
```

Purpose

Tests whether the character is a lowercase letter.

Return Value

The **islower** functions return a non-zero value when *c* is a lowercase character.

Parameters

c Character.

Description

The **islower** function tests for any character that is a lowercase letter or is one of a locale-specific set of characters for which none of **iscntrl**, **isdigit**, **ispunct**, or **isspace** is true. In the “C” locale, **islower** returns true only for the characters defined as lowercase letters.

Example

```
/* a sample program testing islower() */
#include <ctype.h>
#include <stdio.h>

int main() {
    char c = 'A';
    printf("islower('%c') returns %d\n", c, islower(c));
    if(islower(c) > 0)
        printf("'c' is a lower case character.\n", c);
    else
        printf("'c' is not a lower case character.\n", c);
    c = 'a';
    printf("islower('%c') returns %d\n", c, islower(c));
    if(islower(c) > 0)
        printf("'c' is a lower case character.\n", c);
    else
        printf("'c' is not a lower case character.\n", c);
}
```

Output

```
islower('A') returns 0
'A' is not a lower case character.
islower('a') returns 1
'a' is a lower case character.
```

See Also

isprint

Synopsis

```
#include <ctype.h>
int isprint(int c);
```

Purpose

Tests whether the character is a printing character.

Return Value

The **isprint** function returns a non-zero value when *c* is a printing character.

Parameters

c Character.

Description

The **isprint** function test for any printing character, including space (' ').

Example

```
/* a sample program testing isprint() */
#include <ctype.h>
#include <stdio.h>

int main() {
    char c = 'A';
    printf("isprint('%c') returns %d\n", c, isprint(c));
    if(isprint(c) > 0)
        printf("'%'c' is a printable character.\n", c);
    else
        printf("'%'c' is not a printable character.\n", c);
    c = ' ';
    printf("isprint('%c') returns %d\n", c, isprint(c));
    if(isprint(c) > 0)
        printf("'%'c' is a printable character.\n", c);
    else
        printf("'%'c' is not a printable character.\n", c);
}
```

Output

```
isprint('A') returns 1
'A' is a printable character.
isprint(' ') returns 1
' ' is a printable character.
```

See Also

ispunct

Synopsis

```
#include <ctype.h>
int ispunct(int c);
```

Purpose

Tests whether the character is a punctuation character.

Return Value

The **ispunct** functions return a non-zero value when *c* is a punctuation character.

Parameters

c Character.

Description

The **ispunct** function tests for any printing character that is one of a locale-specific set of punctuation characters for which neither **isspace** nor **isalnum** is true. In the “C” locale, **ispunct** returns true for every printing character for which neither **isspace** nor **isalnum** is true.

Example

```
/* a sample program testing ispunct() */
#include <ctype.h>
#include <stdio.h>

int main() {
    char c = 'A';
    printf("ispunct('%c') returns %d\n", c, ispunct(c));
    if (ispunct(c) > 0)
        printf("'%c' is a punctuation mark.\n", c);
    else
        printf("'%c' is not a punctuation mark.\n", c);
    c = ';';
    printf("ispunct('%c') returns %d\n", c, ispunct(c));
    if (ispunct(c) > 0)
        printf("'%c' is a punctuation mark.\n", c);
    else
        printf("'%c' is not a punctuation mark.\n", c);
}
```

Output

```
ispunct('A') returns 0
'A' is not a punctuation mark.
ispunct(';') returns 1
';' is a punctuation mark.
```

See Also

isspace

Synopsis

```
#include <ctype.h>
int isspace(int c);
```

Purpose

Tests whether the character is a white-space character.

Return Value

The **isspace** functions return a non-zero value when *c* is a white-space character.

Parameters

c Character.

Description

The **isspace** function tests for any character that is a standard white-space character or is one of a locale-specific set of characters for which **isalnum** is false. The standard white-space characters are the following: space (' '), form feed ('\f'), new-line ('\n'), carriage return ('\r'), horizontal tab ('\t'), and vertical tab ('\v'). In the "C" locale, **isspace** returns true only for the standard white-space characters.

Example

```
/* a sample program testing isspace() */
#include <ctype.h>
#include <stdio.h>

int main() {
    char c = 'A';
    printf("isspace('%c') returns %d\n", c, isspace(c));
    if(isspace(c) > 0)
        printf("'%c' is a white space character.\n", c);
    else
        printf("'%c' is not a white space character.\n", c);
    c = ' ';
    printf("isspace('%c') returns %d\n", c, isspace(c));
    if(isspace(c) > 0)
        printf("'%c' is a white space character.\n", c);
    else
        printf("'%c' is not a white space character.\n", c);
}
```

Output

```
isspace('A') returns 0
'A' is not a white space character.
isspace(' ') returns 1
' ' is a white space character.
```

See Also

isupper

Synopsis

```
#include <ctype.h>
int isupper(int c);
```

Purpose

Tests whether the character is an uppercase character.

Return Value

The **isupper** function returns a non-zero value when *c* is an uppercase character.

Parameters

c Character.

Description

The **isupper** function tests for any character that is an uppercase letter or is one of a locale-specific set of characters for which none of **isctrl**, **isdigit**, **ispunct**, or **isspace** is true. In the “C” locale, **isupper** returns true only for the characters defined as uppercase letters.

Example

```
/* a sample program testing isupper() */
#include <ctype.h>
#include <stdio.h>

int main() {
    char c = 'A';
    printf("isupper('%c') returns %d\n", c, isupper(c));
    if(isupper(c) > 0)
        printf("'c' is a upper case character.\n", c);
    else
        printf("'c' is not a upper case character.\n", c);
    c = 'a';
    printf("isupper('%c') returns %d\n", c, isupper(c));
    if(isupper(c) > 0)
        printf("'c' is a upper case character.\n", c);
    else
        printf("'c' is not a upper case character.\n", c);
}
```

Output

```
isupper('A') returns 1
'A' is a upper case character.
isupper('a') returns 0
'a' is not a upper case character.
```

See Also

isxdigit

Synopsis

```
#include <ctype.h>
int isxdigit(int c);
```

Purpose

Tests whether the character is a hexadecimal-digit character.

Return Value

The **isxdigit** functions return a non-zero value when *c* is an hexadecimal-digit character.

Parameters

c Character.

Description

The **isxdigit** function tests for any hexadecimal-digit character.

Example

```
/* a sample program testing isxdigit() */
#include <ctype.h>
#include <stdio.h>

int main() {
    char c = 'T';
    printf("isxdigit('%c') returns %d\n", c, isxdigit(c));
    if(isxdigit(c) > 0)
        printf("%c is a hexadecimal-digit character.\n", c);
    else
        printf("%c is not a hexadecimal-digit character.\n", c);
    c = 'a';
    printf("isxdigit('%c') returns %d\n", c, isxdigit(c));
    if(isxdigit(c) > 0)
        printf("%c is a hexadecimal-digit character.\n", c);
    else
        printf("%c is not a hexadecimal-digit character.\n", c);
}
```

Output

```
isxdigit('T') returns 0
'T' is not a hexadecimal-digit character.
isxdigit('a') returns 1
'a' is a hexadecimal-digit character.
```

See Also

tolower

Synopsis

```
#include <ctype.h>
int tolower(int c);
```

Purpose

Converts an uppercase character to a lowercase character.

Return Value

If the argument for which **isupper** is true, and there are one or more corresponding characters, as specified by the current locale, for which **islower** is true, the **tolower** function returns one of the corresponding characters (always the same one for any given locale); otherwise, the argument is returned unchanged.

Parameters

c Character.

Description

The **tolower** function converts an uppercase letter to a corresponding lowercase letter.

Example

```
/* a sample program that displays lower case of the letter. */
#include <stdio.h>
#include <ctype.h>

int main() {
    char c = 'Q';
    printf("tolower('%c') = %c\n", c, tolower('Q'));
}
```

Output

```
tolower('Q') = q
```

See Also

toupper

Synopsis

```
#include <ctype.h>
int toupper(int c);
```

Purpose

Converts a lowercase character to an uppercase character.

Return Value

If the argument for which **islower** is true, and there are one or more corresponding characters, as specified by the current locale, for which **isupper** is true, the **toupper** function returns one of the corresponding characters (always the same one for any given locale); otherwise, the argument is returned unchanged.

Parameters

c Character.

Description

The **toupper** function converts a lowercase character to a corresponding uppercase character.

Example

```
/* a sample program that displays uppercase of the letter. */
#include <stdio.h>
#include <ctype.h>

int main() {
    char c = 'a';
    printf("toupper('%c') = %c\n", c, toupper(c));
}
```

Output

```
toupper('a') = A
```

See Also

Chapter 6

Errors — `<errno.h>`

The header **errno.h** defines several macros, all relating to the reporting of error conditions.

The macros are

EDOM
EILSEQ
ERANGE

which expand to integer constant expressions with type **int**, distinct positive values, and which are suitable for use in **#if** preprocessing directives; and

errno

which expands to a modifiable lvalue that has type **int**, the value of which is set to a positive error number by several library functions.

The value of **errno** is zero at program startup, but is never set to zero by any library function. The value of **errno** may be set to nonzero by a library function call whether or not there is an error, provided the use of **errno** is not documented in the description of the function in this International Standard.

Additional macro definitions, beginning with **E** and a digit or **E** and an uppercase letter, are specified in the *errno.h* header file.

Macros

The following macros are defined by the **errno.h** header file.

Macro	Description
EDOM	Math arg out of domain of func
EILSEQ	Illegal byte sequence.
ERANGE	Math result not representable

Portability

This header has no known portability problem.

Chapter 7

Characteristics of floating types <float.h>

The header **float.h** defines several macros that expand to various limits and parameters of the standard floating-point types.

The macros, their meanings, and the constraints (or restrictions) on their values are listed below.

Macros

The following macros are defined by the **float.h** header file.

Macro	Constraint on the Value	Description
FLT_RADIX	2	radix of exponent representation.

Macro	Constraint on the Value	Description
FLT_MANT_DIG	24	number of base- FLT_RADIX digits in the floating-point significant.
DBL_MANT_DIG	53	
LDBL_MANT_DIG		

Macro	Constraint on the Value	Description
FLT_DIG	6	number of decimal digits.
DBL_DIG	15	
LDBL_DIG		

Macro	Constraint on the Value	Description
-------	-------------------------	-------------

FLT_MIN_EXP	(- 125)	minimum negative integer such that FLT_RADIX raised to one less than that power is a normalized
DBL_MIN_EXP	(- 1021)	
LDBL_MIN_EXP		

Macro	Constraint on the Value	Description
FLT_MIN_10_EXP	(- 37)	minimum negative integer such that 10 raised to that power is in the range of no number.
DBL_MIN_10_EXP	(- 307)	
LDBL_MIN_10_EXP		

Macro	Constraint on the Value	Description
FLT_MAX_EXP	128	maximum integer such that FLT_RADIX raised to one less than that power is a representable finite floating-point number.
DBL_MAX_EXP	1024	
LDBL_MAX_EXP		

Macro	Constraint on the Value	Description
FLT_MAX_10_EXP	38	maximum integer such that 10 raised to that power is in the range of representable finite floating-point numbers.
DBL_MAX_10_EXP	308	
LDBL_MAX_10_EXP		

Macro	Constraint on the Value	Description
FLT_MAX	3.40282347E+38F	maximum representable finite floating-point number.
DBL_MAX	1.797693134862315E+308	
LDBL_MAX		

Macro	Constraint on the Value	Description
FLT_EPSILON	1.19209290E-07F	the difference between 1 and the least value greater than 1 that is representable between 1 and the least.
DBL_EPSILON	2.2204460492503131E-16	
LDBL_EPSILON		

Macro	Constraint on the Value	Description
FLT_MIN	1.17549435E-38F	minimum normalized positive floating-point number.
DBL_MIN	2.2250738585072014E-308	
LDBL_MIN		

Portability

This header has no known portability problem.

Chapter 8

Sizes of integer types — `<limits.h>`

The header **limits.h** defines several macros that expand to various limits and parameters of the standard integer types.

The macros, their meanings, and the constraints (or restrictions) on their values are listed below.

Portability

This header has no known portability problem.

Macros

The following macros are defined by the **limits.h** header file.

Macro	Description Constraint on the Value
CHAR_BIT	number of bits for smallest object that is not a bit_filed(byte) 8
SCHAR_MIN	minimum value for an object of type signed char -128 // -2⁷
SCHAR_MAX	maximum value for an object of type signed char +127 // 2⁷-1
UCHAR_MAX	maximum value for an object of type unsigned char 255 // 2⁸-1
CHAR_MIN	minimum value for an object of type char -128 // -2⁷
CHAR_MAX	maximum value for an object of type char +127 // 2⁷-1
MB_LEN_MAX	maximum number of bytes in a multibyte character,for any supported locale 1 (This value is platform-dependent.)
SHRT_MIN	minimum value for an object of type short int -32768 // -2¹⁵
SHRT_MAX	maximum value for an object of type short int +32767 // 2¹⁵-1
USHRT_MAX	maximum value for an object of type unsigned short int 65535 // 2¹⁶-1

INT_MIN	minimum value for an object of type int -2147483648 // -2³¹
INT_MAX	maximum value for an object of type int +2147483647 // 2³¹-1
UINT_MAX	maximum value for an object of type unsigned int 4294967295 // 2³²-1
LONG_MIN	minimum value for an object of type long int -2147483648 // -2³¹
LONG_MAX	maximum value for an object of type long int 2147483647 // 2³¹-1
ULONG_MAX	maximum value for an object of type unsigned long int 4294967295 // 2³²-1
LLONG_MIN	minimum value for an object of type long long int -9223372036854775808 // -2⁶³
LLONG_MAX	maximum value for an object of type long long int +9223372036854775807 // 2⁶³-1
ULLONG_MAX	maximum value for an object of type unsigned long long int 18446744073709551615 // 2⁶⁴-1

Chapter 9

Localization <locale.h>

The header **locale.h** declares two functions, one type, and defines several macros.

The type is

struct lconv

which contains members related to the formatting of numeric values. The structure shall contain at least the following members, in any order. The semantics of the members and their normal ranges are explained later.

In the C locale, the members shall have the values specified in the comments.

```
char *decimal_point;    // "."
char *thousands_sep;   // ""
char *grouping;         // ""
char *mon_decimal_point // ""
char *mon_thousands_sep // ""
char *mon_grouping;     // ""
char *positive_sign;    // ""
char *negative_sign;    // ""
char *currency_symbol;  // ""
char frac_digits;       // CHAR_MAX
char p_cs_precedes;     // CHAR_MAX
char n_cs_precedes;     // CHAR_MAX
char p_sep_by_space;    // CHAR_MAX
char n_sep_by_space;    // CHAR_MAX
char p_sign_posn;       // CHAR_MAX
char n_sign_posn;       // CHAR_MAX
char *int_curr_symbol;  // ""
char int_frac_digits;   // CHAR_MAX
char int_p_cs_precedes; // CHAR_MAX
char int_n_cs_precedes; // CHAR_MAX
char int_p_sep_by_space; // CHAR_MAX
char int_n_sep_by_space; // CHAR_MAX
char int_p_sign_posn;   // CHAR_MAX
char int_n_sign_posn;   // CHAR_MAX
```

The macros defined are

```
LC_ALL
LC_COLLATE
```


LC_CTYPE

LC_MONETARY

LC_NUMERIC

LC_TIME

which expand to integer constant expressions with distinct values, suitable for use as the first argument to the **setlocale** function. Additional macro definitions, beginning with the characters **LC_** and an uppercase letter, may be added in the future version.

Functions

The following functions are defined by the **locale.h** header file.

Function	Description
localeconv()	Returns a pointer containing the values appropriate for the formatting of the numeric quantities according to the current locale.
setlocale()	Selects the appropriate piece of the program's locale as specified by the category, and may be used to change or query the program's international environment.

Macros

The following macros are defined by the **locale.h** header file.

Macro	Description
LC_ALL	Determines all localization categories.
LC_COLLATE	Affects the behavior of the string collation functions strcoll() and strxfrm() .
LC_CTYPE	Affects the behavior of the character classification and conversion functions.
LC_MONETARY	Affects the monetary unit format.
LC_NUMERIC	Determines the decimal-point character for formatted input/output functions.
LC_TIME	Determines the behavior of the strftime() function.

Declared Types

The following types are defined by the **locale.h** header file.

Type	Description
lconv	struct - holds values related to the formatting of numerical values.

Portability

This header has no known portability problem.

localeconv

Synopsis

```
#include <locale.h>
structlconv *localeconv(void);
```

Purpose

Numeric formatting convention inquiry.

Return Value

The **localeconv** function returns a pointer to the filled-in object. The structure pointed to by the return value shall not be modified by the program, but may be overwritten by a subsequent call to the **localeconv** function. In addition, calls to the **setlocale** function with categories **LC_ALL**, **LC_MONETARY**, or **LC_NUMERIC** may overwrite the contents of the structure.

Parameters

No argument.

Description

The **localeconv** function sets the components of an object with type **struct lconv** with values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale.

The members of the structure with type **char *** are pointers to strings, any of which (except **decimal_point**) can point to "", to indicate that the value is not available in the current locale or is of zero length. Apart from **grouping** and **mon_grouping**, the strings shall start and end in the initial shift state. The members with type **char** are nonnegative numbers, any of which can be **CHAR_MAX** to indicate that the value is not available in the current locale. The members include the following:

char *decimal_point

The decimal-point character used to format nonmonetary quantities.

char *thousands_sep

The character used to separate groups of digits before the decimal-point character in formatted nonmonetary quantities.

char *grouping

A string whose elements indicate the size of each group of digits in formatted nonmonetary quantities.

char *mon_decimal_point

The decimal-point used to format monetary quantities.

char *mon_thousands_sep

The separator for groups of digits before the decimal-point in formatted monetary quantities.

char *mon_grouping

A string whose elements indicate the size of each group of digits in formatted monetary quantities.

char *positive_sign

The string used to indicate a nonnegative-valued formatted monetary quantity.

char *negative_sign

The string used to indicate a negative-valued formatted monetary quantity.

char *currency_symbol

The local currency symbol applicable to the current locale.

char frac_digits

The number of fractional digits (those after the decimal-point) to be displayed in a locally formatted monetary quantity.

char p_cs_precedes

Set to 1 or 0 if the `currency_symbol` respectively precedes or succeeds the value for a nonnegative locally formatted monetary quantity.

char n_cs_precedes

Set to 1 or 0 if the `currency_symbol` respectively precedes or succeeds the value for a negative locally formatted monetary quantity.

char p_sep_by_space

Set to a value indicating the separation of the `currency_symbol`, the sign string, and the value for a nonnegative locally formatted monetary quantity.

char n_sep_by_space

Set to a value indicating the separation of the `currency_symbol`, the sign string, and the value for a negative locally formatted monetary quantity.

char p_sign_posn

Set to a value indicating the positioning of the `positive_sign` for a nonnegative locally formatted monetary quantity.

char n_sign_posn

Set to a value indicating the positioning of the `negative_sign` for a negative locally formatted monetary quantity.

char *int_curr_symbol

The international currency symbol applicable to the current locale. The first three characters contain the alphabetic international currency symbol in accordance with those specified in ISO 4217:1995. The fourth character (immediately preceding the null character) is the character used to separate the international currency symbol from the monetary quantity.

char int_frac_digits

The number of fractional digits (those after the decimal-point) to be displayed in an internationally formatted monetary quantity.

char int_p_cs_precedes

Set to 1 or 0 if the **int_currency_symbol** respectively precedes or succeeds the value for a nonnegative internationally formatted monetary quantity.

char int_n_cs_precedes

Set to 1 or 0 if the **int_currency_symbol** respectively precedes or succeeds the value for a negative internationally formatted monetary quantity.

char int_p_sep_by_space

Set to a value indicating the separation of the **int_currency_symbol**, the sign string, and the value for a nonnegative internationally formatted monetary quantity.

char int_n_sep_by_space

Set to a value indicating the separation of the **int_currency_symbol**, the sign string, and the value for a negative internationally formatted monetary quantity.

char int_p_sign_posn

Set to a value indicating the positioning of the **positive_sign** for a nonnegative internationally formatted monetary quantity.

char int_n_sign_posn

Set to a value indicating the positioning of the **negative_sign** for a negative internationally formatted monetary quantity.

The elements of **grouping** and **mon_grouping** are interpreted according to the following:

CHAR_MAX	No further grouping is to be performed.
0	The previous element is to be repeatedly used for the remainder of the digits.
<i>other</i>	The integer value is the number of digits that compose the current group. The next element is examined to determine the size of the next group of digits before the current group.

The values of **p_sep_by_space**, **n_sep_by_space**, **int_p_sep_by_space**, and **int_n_sep_by_space** are interpreted according to the following:

- 0** No space separates the currency symbol and value.
- 1** If the currency symbol and sign string are adjacent, a space separates them from the value; otherwise, a space separates the currency symbol from the value.
- 2** If the currency symbol and sign string are adjacent, a space separates them; otherwise, a space separates the sign string from the value.

The values of **p_sign_posn**, **n_sign_posn**, **int_p_sign_posn**, and **int_n_sign_posn** are interpreted according to the following:

- 0 Parentheses surround the quantity and currency symbol.
- 1 The sign string precedes the quantity and currency symbol.
- 2 The sign string succeeds the quantity and currency symbol.
- 3 The sign string immediately precedes the currency symbol.
- 4 The sign string immediately succeeds the currency symbol.

The implementation shall behave as if no library function calls the **localeconv** function.

EXAMPLE 1 The following table illustrates the rules which may well be used by four countries to format monetary quantities.

Country	Local format		International format	
	Positive	Negative	Positive	Negative
Finland	1.234,56 mk	-1.234,56 mk	FIM 1.234,56	FIM -1.234,56
Italy	L.1.234	-L.1.234	ITL 1.234	-ITL 1.234
Netherlands	f 1.234,56	f -1.234,56	NLG 1.234,56	NLG -1.234,56
Switzerland	SFrs.1,234.56	SFrs.1,234.56	CHF 1,234.56	CHF -1,234.56

For those four countries, the respective values for the monetary members of the structure returned by **localeconv** are:

	Finland	Italy	Netherlands	Switzerland
mon_decimal_point	","	","	","	."
mon_thousands_sep	","	","	","	","
mon_grouping	"\3"	"\3"	"\3"	"\3"
positive_sign	""	""	""	""
negative_sign	"_"	"_"	"_"	"C"
currency_symbol	"mk"	"L."	"\u0192"	"SFrs."
frac_digits	2	0	2	0
p_cs_precedes	0	1	1	1
n_cs_precedes	0	1	1	1
p_sep_by_space	1	0	1	0
n_sep_by_space	1	0	1	0
p_sign_posn	1	1	1	1
n_sign_posn	1	1	4	2
int_curr_symbol	"FIM"	"ITL"	"NLG"	"CHF"
int_facr_digits	2	0	2	2
int_p_cs_precedes	1	1	1	1
int_n_cs_precedes	1	1	1	1
int_p_sep_by_space	0	0	0	0
int_n_sep_by_space	0	0	0	0
int_p_sign_posn	1	1	1	1
int_n_sign_posn	4	1	4	2

EXAMPLE 2 The following table illustrates how the `p_cs_precedes`, `p_sep_by_space`, and `p_sign_posn` members affect the formatted value.

p_cs_precedes	p_sign_posn	p_sep_by_space		
		0	1	2
0	0	(1.25\$)	(1.25 \$)	(1.25\$)
	1	+1.25\$	+1.25 \$	+ 1.25\$
	2	1.25\$+	1.25 \$+	1.25 \$ +
	3	1.25+\$	1.25 +\$	1.25 + \$
	4	1.25\$+	1.25 \$+	1.25 \$ +
1	0	(1.25)	(\$ 1.25)	(\$1.25)
	1	+\$1.25	+\$ 1.25	+\$1.25
	2	\$1.25+	\$ 1.25+	\$1.25 +
	3	+\$1.25	+\$ 1.25	+\$1.25
	4	\$+1.25	\$+1.25	\$+1.25

Example

```
/* a sample program that displays the decimal point
character used by the current locale. */
#include <stdio.h>
#include <locale.h>

int main() {
    struct lconv *lc;
    lc = localeconv();
    printf("Decimal symbol is: %s\n", lc->decimal_point);
    printf("Thousands separator is: %s\n", lc->thousands_sep);
    printf("Currency symbol is: %s\n", lc->currency_symbol);
}
```

Output

```
Decimal symbol is: .
Thousands separator is:
Currency symbol is:
```

See Also

setlocale

Synopsis

```
#include <locale.h>
char *setlocale(int category, const char* locale);
```

Purpose

Locale control.

Return Value

If a pointer to a string is given for *locale* and the selection can be honored, the **setlocale** function returns a pointer to the string associated with the specified *category* for the new locale. If the selection cannot be honored, the **setlocale** function returns a null pointer and the program's locale is not changed.

A null pointer for *locale* causes the **setlocale** function to return a pointer to the string associated with the *category* for the program's current locale; the program's locale is not changed.

The pointer to string returned by the **setlocale** function is such that a subsequent call with that string value and its associated category will restore that part of the program's locale. The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the **setlocale** function.

Parameters

category Category of locales.

locale Locale value.

Description

The **setlocale** function selects the appropriate portion of the program's locale as specified by the category and locale arguments. The **setlocale** function may be used to change or query the program's entire current locale or portions thereof. The value **LC_ALL** for category names the program's entire locale; the other values for category name only a portion of the program's locale. **LC_COLLATE** affects the behavior of the **strcoll** and **strxfrm** functions. **LC_CTYPE** affects the behavior of the character handling functions and the multibyte and wide-character functions. **LC_MONETARY** affects the monetary formatting information returned by the **localeconv** function. **LC_NUMERIC** affects the decimal-point character for the formatted input/output functions and the string conversion functions, as well as the nonmonetary formatting information returned by the **localeconv** function. **LC_TIME** affects the behavior of the **strftime** function.

A value of "C" for locale specifies the minimal environment for C^H translation; a value of "" for locale specifies the locale-specific native environment. Other implementation-defined strings may be passed as the second argument to **setlocale**.

At program startup, the equivalent of

```
setlocale(LC_ALL, "C");
```

is executed.

The implementation shall behave as if no library function calls the **setlocale** function.

Example

```
/* a sample program that displays the current locale setting.*/
#include <locale.h>
#include <stdio.h>

int main() {
    printf("local locale is %s\n", setlocale(LC_ALL, ""));
}
```

Output

```
local locale is C
```

See Also

strcoll(), strftime(), strxfrm().

Chapter 10

Mathematics <math.h>

The header **math.h** declares two types, several mathematical functions, and defines several macros. Most synopses specify a family of functions consisting of a principal function with one or more double parameters, a double return value, or both; and other functions with the same name but with **f** and **l** suffixes which are corresponding functions with float and long double parameters, return values, or both. Integer arithmetic functions and conversion functions are discussed later.

The macro

HUGE_VAL

expands to a positive double constant expression, not necessarily representable as a **float**.

The macro

INFINITY

expands to a constant expression of type float representing positive or unsigned infinity, if available; else to a positive constant of type float that overflows at translation time.

The macro

NAN

is defined if and only if the implementation supports quiet NaNs for the **float** type. It expands to a constant expression of type **float** representing a quiet **NaN**.

The macros

FP_ILOGB0

FP_ILOGBNAN

expand to integer constant expressions whose values are returned by **ilogb(x)** if **x** is zero or NaN, respectively. The value of **FP_ILOGB0** shall be either **INT_MIN** or **INT_MAX**. The value of **FP_ILOGBNAN** shall be either **INT_MAX** or **INT_MIN**.

Functions

The following functions are defined by the **math.h** header file.

Function	Description
acos()	Calculates the arc cosine of an argument.
acosh()	Calculates the arc hyperbolic cosine of an argument.
asin()	Calculates the arc sine of an argument.
asinh()	Calculates the arc hyperbolic sine of an argument.
atan()	Calculates the arc tangent of an argument in the range $[-\pi/2, \pi/2]$.
atan2()	Calculates the arc tangent of an argument in the range $[-\pi, \pi]$.
atanh()	Calculates the arc hyperbolic tangent of an argument.
cbrt()	Calculates the cubed root of an argument.
ceil()	Calculates the smallest integer value not less than the argument.
copysign()	Switches signs of two arguments.
cos()	Calculates the cosine of an argument.
cosh()	Calculates the hyperbolic cosine of an argument.
deg2rad()	Convert an angle in degrees to an angle in radians.
erf()	Calculates the error function.
erfc()	Calculates a modified error function.
exp()	Calculates the base- e exponential of an argument.
exp2()	Calculates the base-2 exponential of an argument.
expm1()	Calculates the base- e -1 exponential of an argument.
fabs()	Calculates the absolute value of an argument.
fdim()	Calculates the difference between two arguments.
floor()	Calculates the largest integer value not greater than the argument.
fma()	Calculates $(x+y) \times z$.
fmax()	Calculates the maximum value of two arguments.
fmin()	Calculates the minimum value of two arguments.
fmod()	Calculates the remainder of two arguments.
frexp()	Calculates the normalized fractions of an argument.
hypot()	Calculates $\sqrt{z^2 + y^2}$.
ilogb()	Calculates the exponent of the argument as a signed int value.
ldexp()	Calculates $x \times 2^{exp}$.
lgamma()	Calculates the absolute logarithmic gamma value of an argument.
log()	Calculates the logarithm of an argument.
log10()	Calculates the base-10 logarithm of an argument.
log1p()	Calculates the base- e logarithm of one plus the argument.
log2()	Calculates the base-2 logarithm of an argument.
logb()	Calculates the signed exponent of the argument.
lrint()	Calculates the rounded integer value of an argument according to the current rounding direction.
lround()	Calculates the rounded integer value of an argument, rounding halfway cases to zero, regardless of rounding direction.
modf()	Calculates the signed fractional part of the argument.
nan()	Returns a quiet NaN.
nearbyint()	Calculates the rounded integer value of an argument, using the current rounding

	direction.
nextafter()	Calculates the next representable value of an argument.
nexttoward()	Calculates the next representable value of an argument.
pow()	Calculates x^y .
rad2deg()	Convert an angle in radians to an angle in degrees.
remainder()	Calculates the remainder of two arguments.
remquo()	Calculates the remainder of two arguments.
rint()	Calculates the rounded integer value of an argument.
round()	Calculates the rounded integer value of an argument, rounding halfway cases away from zero, regardless of current rounding direction.
scalbn()	Calculates $x \times \text{FLT_RADIX}^n$.
sin()	Calculates the sine of an argument.
sinh()	Calculates the hyperbolic sine of an argument.
sqrt()	Calculates the square root of an argument.
tan()	Calculates the tangent of an argument.
tanh()	Calculates the hyperbolic tangent of an argument.
tgamma()	Calculates the gamma function of an argument.
trunc()	Calculates the truncated integer value of an argument.

Macros

The following macros are defined by the **math.h** header file.

Macro	Description
HUGE_VAL	double - Positive constant expression.
INFINITY	float - Positive infinity.
NAN	float - Not-a-Number.
isfinite	int - Returns a nonzero value if the argument is finite.
isgreater	double - Returns the value of $x > y$.
isgreaterequal	double - Returns the value of $x \geq y$.
isinf	int - Returns a nonzero value if the argument is infinite.
isless	double - Returns the value of $x < y$.
islessequal	double - Returns the value of $x \leq y$.
islessgreater	double - Determines if x is less than, greater than, or equal to y .
isnan	int - Returns a nonzero value if the argument is NaN.
isnormal	int - Returns a nonzero value if the argument is normal, (neither zero, subnormal, infinite, nor NaN).
isunordered	Returns 1 if the argument is unordered.
signbit	int - Returns a nonzero value if the argument has a negative value.

Portability

Functions **isnormal()**, **fpclassify()**, **signbit()** are not available in all platforms. Functions **isunordered()**,

lrint(), **lround()**, **NAN()**, **nearbyint()**, **nexttoward()**, **remquo()**, **round()**, **tgamma()**, and **trunc()** are not supported in any platform.

acos

Synopsis

```
#include <math.h>
type acos(type x);
double acos(double x);
float acosf(float x);
long double acosl(long double x);
```

Purpose

Calculate arc cosine.

Return Value

The **acos** functions return the arc cosine in the range $[0, \pi]$ radians.

Parameters

x Argument.

Description

The **acos** functions compute the arc cosine of *x*. A domain error occurs for arguments not in the range $[-1, +1]$. The function returns the following values related to special numbers:

- **acos**($\pm\infty$) returns NaN.
- **acos**(NaN) returns NaN.
- **acos**(± 0.0) returns $\pi/2$.
- **acos**(*x*) returns NaN for $|x| > 1$.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("acos(x) = %f\n", acos(x));
}
```

Output

```
acos(x) = 1.266104
```

See Also

cos(), **acosh()**, **cosh()**.

acosh

Synopsis

```
#include <math.h>
type acosh(type x);
double acosh(double x);
float acoshf(float x);
long double acoshl(long double x);
```

Purpose

Calculate arc hyperbolic cosine.

Return Value

The **acosh** functions return the arc hyperbolic cosine in the range $[0, \infty]$.

Parameters

x Argument.

Description

The **acosh** functions compute the (nonnegative) arc hyperbolic cosine of *x*. A domain error occurs for arguments less than 1. The function returns the following values related to special numbers:

- **acosh**($-\infty$) returns NaN.
- **acosh**($+\infty$) returns $+\infty$.
- **acosh**(NaN) returns NaN.
- **acosh**($-x$) returns NaN, for $x > 0$.
- **acosh**(*x*) returns NaN, for $x < 1.0$.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("acosh(x) = %f\n", acosh(x));
}
```

Output

```
acosh(x) = NaN
```

See Also

cos(), **acos()**, **cosh()**.

asin

Synopsis

```
#include <math.h>
type asin(type x);
double asin(double x);
float asinf(float x);
long double asinl(long double x);
```

Purpose

Calculate arc sine.

Return Value

The **asin** functions return the arc sine in the range $[-\pi/2, \pi/2]$.

Parameters

x Argument.

Description

The **asin** functions compute the principal value of the arc sine of *x*. A domain error occurs for arguments not in the range $[-1, +1]$. The function returns the following values related to special numbers:

- **asin**($\pm\infty$) returns NaN.
- **asin**(NaN) returns NaN.
- **asin**(± 0.0) returns ± 0.0 .
- **asin**(*x*) returns NaN for $|x| > 1$.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("asin(x) = %f\n", asin(x));
}
```

Output

```
asin(x) = 0.304693
```

See Also

sin(), **asinh()**, **sinh()**.

asinh

Synopsis

```
#include <math.h>
type asinh(type x);
double asinh(double x);
float asinhf(float x);
long double asinhl(long double x);
```

Purpose

Calculate arc sine.

Return Value

The **asinh** functions return the arc hyperbolic sine value. The function returns the following values related to special numbers:

- **asinh**($\pm\infty$) returns $\pm\infty$.
- **asinh**(NaN) returns NaN.
- **asinh**(± 0.0) returns ± 0.0 .
- **asinh**($-x$) returns **-asinh**(x), for $x > 0$.

Parameters

x Argument.

Description

The **asinh** functions compute the arc hyperbolic sine of x .

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("asinh(x) = %f\n", asinh(x));
}
```

Output

```
asinh(x) = 0.295673
```

See Also

sin(), **asin()**, **sinh()**.

atan

Synopsis

```
#include <math.h>
type atan(type x);
double atan(double x);
float atanf(float x);
long double atanl(long double x);
```

Purpose

Calculate arc tangent.

Return Value

The **atan** functions return the arc tangent in the range $[-\pi/2, \pi/2]$ radians.

Parameters

x Argument.

Description

The **atan** functions compute the arc tangent of *x*. The function returns the following values related to special numbers:

- **atan**($\pm\infty$) returns $\pm\pi/2$.
- **atan**(± 0.0) returns ± 0.0 .
- **atan**(NaN) returns NaN.
- **atan**($-x$) returns **-atan**(*x*), for $x > 0$.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("atan(x) = %f\n", atan(x));
}
```

Output

```
atan(x) = 0.291457
```

See Also

tan(), **atanh()**, **atanh()**.

atan2

Synopsis

```
#include <math.h>
type atan2(type y, type x);
double atan2(double y, double x);
float atanf(float y, float x);
long double atanl(long double y, long double x);
```

Purpose

Calculate arc tangent.

Return Value

The **atan2** functions return the arc tangent of y/x , in the range $[-\pi, \pi]$ radians.

Parameters

x Argument.

y Argument.

Description

The **atan2** functions compute the principal value of the arc tangent of y/x , using the signs of both arguments to determine the quadrant of the return value. A domain error may occur if both arguments are zero. The function returns the following values related to special numbers:

- **atan2**($\pm 0, x$) returns ± 0 , for $x > 0$.
- **atan2**($\pm 0, +0$) returns ± 0 .
- **atan2**($\pm 0, x$) returns $\pm \pi$, for $x < 0$.
- **atan2**($\pm 0, -0$) returns $\pm \pi$.
- **atan2**($y, \pm 0$) returns $\pi/2$ for $y > 0$.
- **atan2**($y, \pm 0$) returns $-\pi/2$ for $y < 0$.
- **atan2**($\pm y, \infty$) returns ± 0 for $y > 0$.
- **atan2**($\pm \infty, x$) returns $\pm \pi/2$, for finite x .
- **atan2**($\pm y, -\infty$) returns $\pm \pi$, for $y > 0$.
- **atan2**($\pm \infty, \infty$) returns $\pm \pi/4$.
- **atan2**($\pm \infty, -\infty$) returns $\pm 3\pi/4$.
- **atan2**(NaN, NaN) returns NaN.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    double y = 0.5;
    printf("atan2(x,y) = %f\n", atan2(x,y));
}
```

Output

```
atan2(x, y) = 0.540420
```

See Also

tan(), atan2h(), tan2h().

atanh

Synopsis

```
#include <math.h>
type atanh(type x);
double atanh(double x);
float atanhf(float x);
long double atanhl(long double x);
```

Purpose

Calculate arc hyperbolic tangent.

Return Value

The **atanh** functions return the arc hyperbolic tangent value.

Parameters

x Argument.

Description

The **atanh** functions compute the arc hyperbolic tangent of *x*. A domain error occurs for arguments not in the range $[-1, 1]$. A range error may occur if the argument equals -1 or $+1$. The function returns the following values related to special numbers:

- **atanh**(± 0) returns ± 0 .
- **atanh**(± 1) returns $\pm \infty$.
- **atanh**(*x*) returns NaN if $|x| > 1$.
- **atanh**($\pm \infty$) returns NaN.
- **atanh**(NaN) returns NaN.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("atanh(x) = %f\n", atanh(x));
}
```

Output

```
atanh(x) = 0.309520
```

See Also

tan(), **atan()**, **tanh()**.

cbrt

Synopsis

```
#include <math.h>
double cbrt(double x);
float cbrtf(float x);
long double cbrtl(long double x);
```

Purpose

Calculate cubed root.

Return Value

The **cbrt** functions return the value of the cube root.

Parameters

x Argument.

Description

The **cbrt** functions compute the real cube root of *x*. The function returns the following values related to special numbers:

- cbrt**($\pm\infty$) returns $\pm\infty$.
- cbrt**(± 0) returns ± 0 .
- cbrt**(NaN) returns NaN.

Example

```
/* a sample program that solves the cube root */
#include <stdio.h>
#include <math.h>

int main() {
    double x= 8.0;
    printf( "the cube root of 8 is %f\n",cbrt(x));
}
```

Output

```
the cube root of 8 is 2.000000
```

See Also

ceil

Synopsis

```
#include <math.h>
type ceil(type x);
double ceil(double x);
float ceilbf(float x);
long double ceilbl(long double x);
```

Purpose

Calculate smallest integer.

Return Value

The **ceil** functions return the smallest integer value not less than x , expressed as a floating-point number.

The function returns the following values related to special numbers:

- **ceil**($\pm\infty$) returns $\pm\infty$.
- **ceil**(± 0) returns ± 0 .
- **ceil**(NaN) returns NaN.

Parameters

x Argument.

Description

The **ceil** functions compute the smallest integer value not less than x .

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("ceil(x) = %f\n", ceil(x));
}
```

Output

```
ceil(x) = 1.000000
```

See Also

copysign

Synopsis

```
#include <math.h>
double copysign(double x, double y);
float copysignf(float x, float y);
long double copysignl(long double x, long double y);
```

Purpose

Switch signs of values.

Return Value

The **copysign** functions return a value with magnitude of x and the sign of y .

Parameters

x Argument.

y Argument.

Description

The **copysign** functions produce a value with the magnitude of x and the sign of y . They produce a NaN (with the sign of y) if x is a NaN. On implementations that represent a signed zero, but do not treat negative zero consistently in arithmetic operations, the **copysign** functions regard the sign of zero as positive.

Example

```
/* a sample program that changes the first number's
sign to the second number's */
#include <stdio.h>
#include <math.h>
#include <float.h>

int main() {
    double x,y,z;

    x = -1.563;
    y = 3.7664;
    z = copysign(x,y);
    printf("before copysign x = %f, y = %f \n",x,y);
    printf("after copysign z = %f, y = %f \n",z,y);
}
```

Output

```
before copysign x = -1.563000, y = 3.766400
after copysign z = 1.563000, y = 3.766400
```

See Also

COS

Synopsis

```
#include <math.h>
type cos(type x);
double cos(double x);
float cosf(float x);
long double cosl(long double x);
```

Purpose

Calculate cosine.

Return Value

The **cos** functions return the cosine value.

Parameters

x Argument.

Description

The **cos** functions compute the cosine of *x* (measured in radians). The function returns following values related to special numbers:

- **cos**(±0) returns 1.
- **cos**(±∞) returns NaN.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("cos(x) = %f\n", cos(x));
}
```

Output

```
cos(x) = 0.955336
```

See Also

acos(), **acosh**(), **cosh**().

cosh

Synopsis

```
#include <math.h>
type cosh(type x);
double cosh(double x);
float coshf(float x);
long double coshl(long double x);
```

Purpose

Calculate hyperbolic cosine.

Return Value

The **cosh** functions return the hyperbolic cosine value.

Parameters

x Argument.

Description

The **cosh** functions compute the hyperbolic cosine of *x*. A range error occurs if the magnitude of *x* is too large. The function returns following values related to special numbers:

- $\cosh(\pm 0)$ returns 1.
- $\cosh(\pm \infty)$ returns $\pm \infty$.
- $\cosh(\text{Nan})$ returns NaN.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("cosh(x) = %f\n", cosh(x));
}
```

Output

```
cosh(x) = 1.045339
```

See Also

cos(), **acosh()**, **acos()**.

deg2rad

Synopsis

```
#include <math.h>
double deg2rad(double x);
```

Purpose

Convert an angle in degrees to an angle in radians.

Return Value

The **deg2rad** function returns the angle in radians.

Parameters

x An angle in degrees.

Description

The function **deg2rad()** converts the angle *x* in degrees to an angle in radians by the formula

$$\text{radian} = \text{degree} * 180/\pi$$

Example

```
/* a sample program that use deg2rad() and rad2deg() */
#include <math.h>

int main() {
    double radian, degree;

    radian = deg2rad(90);
    printf("90 degrees are %lf radians\n", radian);
    degree= rad2deg(M_PI);
    printf("PI (%lf) in radian are %.1f degrees\n", M_PI, degree);
}
```

Output

```
90 degrees are 1.570796 radians
PI (3.141593) in radian are 180 degrees
```

See Also

rad2deg().

erf

Synopsis

```
#include <math.h>
double erf(double x);
float erff(float x);
long double erfl(long double x);
```

Purpose

Calculate error function.

Return Value

The **erf** functions return the error function value.

Parameters

x Argument.

Description

The **erf** functions compute the error function of x : $\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$. The function returns following values related to special numbers:

- **erf**(±0) returns ±0.
- **erf**(±∞) returns ±1.

Example

```
/* a sample program that returns the error fuction of x */
#include <stdio.h>
#include <math.h>

int main() {
    double x,y;

    x = 1;
    y = erf(x);
    printf("the error fuction of x= %f is %f\n",x,y);
    y = erfc(x);
    printf("the complementary value of x= %f is %f\n",x,y);
}
```

Output

```
the error fuction of x= 1.000000 is 0.842701
the complementary value of x= 1.000000 is 0.157299
```

See Also

erfc

Synopsis

```
#include <math.h>
double erfc(double x);
float erfcf(float x);
long double erfcl(long double x);
```

Purpose

Calculate complementary error function.

Return Value

The **erfc** functions return the complementary error function value.

Parameters

x Argument.

Description

The **erfc** functions compute the complementary error function of x : $\frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2} dt$. A range error occurs if x is too large. The function returns following values related to special numbers:

- **erfc**($+\infty$) returns ± 0 .
- **erfc**($-\infty$) returns ± 2 .

Example

See **erf**().

See Also

exp

Synopsis

```
#include <math.h>
type exp(type x);
double exp(double x);
float expf(float x);
long double expl(long double x);
```

Purpose

Calculate base- e exponential.

Return Value

The **exp** functions return the exponential value.

Parameters

x Argument.

Description

The **exp** functions compute the base- e exponential of x : e^x . A range error occurs if the magnitude of x is too large. The function returns following values related to special numbers:

- **exp**(± 0) returns 1.
- **exp**($+\infty$) returns $+\infty$.
- **exp**($-\infty$) returns +0.
- **exp**(NaN) returns NaN.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("exp(x) = %f\n", exp(x));
}
```

Output

```
exp(x) = 1.349859
```

See Also

exp2(), **expm1()**, **frexp()**.

exp2

Synopsis

```
#include <math.h>
double exp2(double x);
float exp2f(float x);
long double exp2l(long double x);
```

Purpose

Calculate base-2 exponential.

Return Value

The **exp2** functions return the base-2 exponential value.

Parameters

x Argument.

Description

The **exp2** functions compute the base-2 exponential of x : 2^x . A range error occurs if the magnitude of x is too large. The function returns following values related to special numbers:

- **exp2**(± 0) returns 1.
- **exp2**($+\infty$) returns $+\infty$.
- **exp2**($-\infty$) returns $+0$.
- **exp2**(NaN) returns NaN.

Example

```
#include <math.h>
#include <stdio.h>

int main() {
    double x = 3.0;
    printf("exp2(%f) = %f\n", x, exp2(x));
}
```

Output

```
exp2(3.000000) = 8.000000
```

See Also

exp(), **expm1()**, **frexp()**.

expm1

Synopsis

```
#include <math.h>
double expm1(double x);
float expm1f(float x);
long double expm1l(long double x);
```

Purpose

Calculate base- e -1 exponential.

Return Value

The **expm1** functions return the value of $e^x - 1$.

Parameters

x Argument.

Description

The **expm1** functions compute the base- e exponential of the argument, minus 1: $e^x - 1$. A range error occurs if x is too large. The function returns following values related to special numbers:

- **expm1**(± 0) returns 0.
- **expm1**($+\infty$) returns $+\infty$.
- **expm1**($-\infty$) returns -1 .
- **expm1**(NaN) returns NaN.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("expm1(x) = %f\n", expm1(x));
}
```

Output

```
expm1(x) = 0.349859
```

See Also

exp(), **exp2()**, **frexp()**.

fabs

Synopsis

```
#include <math.h>
double fabs(double x);
float fabsf(float x);
long double fabsl(long double x);
```

Purpose

Calculate absolute value.

Return Value

The **fabs** functions return the absolute value of x .

Parameters

x Argument.

Description

The **fabs** functions compute the absolute value of a floating-point number x . The function returns following values related to special numbers:

- **fabs**(± 0) returns $+0$.
- **fabs**($\pm \infty$) returns $+\infty$.
- **fabs**(NaN) returns NaN.

Example

```
/* a sample program that returns the absolute
value of x. */
#include <stdio.h>
#include <math.h>

int main() {
    float x;

    x = -12.5;
    printf("The absolute of %3.1f is %3.1f \n", x, fabs(x));
}
```

Output

The absolute of -12.5 is 12.5

See Also

fdim

Synopsis

```
#include <math.h>
double fdim(double x, double y);
float fdimf(float x, float y);
long double fdiml(long double x, long double y);
```

Purpose

Calculate difference value.

Return Value

The **fdim** functions return the positive difference value.

Parameters

x Argument.

y Argument.

Description

The **fdim** functions determine the *positive difference* between their arguments:

$x - y$ if $x \geq y$
 $+0$ if $x < y$

A range error may occur.

Example

```
#include <stdio.h>
#include <math.h>

int main() {
    double x[2] = {1.0, 3.0};
    double y[2] = {3.0, 1.0};
    printf("fdim(x[0],y[0]) = %f\n", fdim(x[0], y[0]));
    printf("fdim(x[1],y[1]) = %f\n", fdim(x[1], y[1]));
}
```

Output

```
fdim(x[0],y[0]) = 0.000000
fdim(x[1],y[1]) = 2.000000
```

See Also

floor

Synopsis

```
#include <math.h>
type floor(type x);
double floor(double x);
float floorf(float x);
long double floorbl(long double x);
```

Purpose

Calculate largest integer.

Return Value

The **floor** functions return the largest integer value not greater than x , expressed as a floating-point number.

Parameters

x Argument.

Description

The **floor** functions compute the largest integer not greater than x . The function returns following values related to special numbers:

— **floor**(x) returns x if x is $\pm\infty$ or ± 0 .

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("floor(x) = %f\n", floor(x));
}
```

Output

```
floor(x) = 0.000000
```

See Also

fma

Synopsis

```
#include <math.h>
double fma(double x, double y, double z);
float fmaf(float x, float y, float z);
long double fmal(long double x, long double y, long double z);
```

Purpose

Calculate sum plus product.

Return Value

The **fma** functions return the sum z plus the product x times y , rounded as one ternary operation.

Parameters

x Argument.

y Argument.

z Argument.

Description

The **fma** functions compute the sum of z plus the product x times y , rounded as one ternary operation: they compute the sum z plus the product x times y (as if) to infinite precision and round once to the result format, according to the rounding mode characterized by the value **FLT_ROUNDS**. The function returns following values related to special numbers:

— **fma**(x, y, z) returns NaN if one of x and y is infinite, the other is zero.

— **fma**(x, y, z) returns NaN if one of x and y is an exact infinite and z is also an infinity but with the opposite sign.

Example

```
#include <stdio.h>
#include <math.h>

int main() {
    double x = 1.0, y = 2.0, z = 3.0;
    printf("fma(%f,%f,%f) = %f\n", x, y, z, fma(x, y, z));
}
```

Output

```
fma(1.000000,2.000000,3.000000) = 5.000000
```

See Also

fmax

Synopsis

```
#include <math.h>
double fmax(double x, double y);
float fmaxf(float x, float y);
long double fmaxl(long double x, long double y);
```

Purpose

Calculate maximum numeric value.

Return Value

The **fmax** functions return the maximum numeric value of their arguments.

Parameters

x Argument.

y Argument.

Description

The **fmax** functions determine the maximum numeric value of their arguments. The function returns the following values related to special numbers:

— If just one argument is NaN, the **fmax** returns the other argument. If both arguments are NaN, the function returns a NaN.

Example

```
/* a sample program that decides which of the two values is a max. */
#include <math.h>
#include <stdio.h>

int main() {
    double x, y, z;

    x = 1.0;
    y = 2.0;
    z =fmax(x, y);
    printf("Value 1: %f\n", x);
    printf("Value 2: %f\n", y);
    printf("%f is the maximum numeric value.\n",z);
}
```

Output

```
Value 1: 1.000000
Value 2: 2.000000
2.000000 is the maximum numeric value.
```

See Also

fmin

Synopsis

```
#include <math.h>
double fmin(double x, double y);
float fminf(float x, float y);
long double fminl(long double x, long double y);
```

Purpose

Calculate minimum numeric value.

Return Value

The **fmin** functions return the minimum numeric value of their arguments.

Parameters

x Argument.

y Argument.

Description

The **fmin** functions determine the minimum numeric value of their arguments. The function returns the following values related to special numbers:

— If just one argument is NaN, the **fmin** returns the other argument. If both arguments are NaN, the function returns a NaN.

Example

```
/* a sample program that decides which of the two values is a min. */
#include <math.h>
#include <stdio.h>

int main() {
    double x, y, z;

    x = 1.0;
    y = 2.0;
    z =fmin(x, y);
    printf("Value 1: %f\n", x);
    printf("Value 2: %f\n", y);
    printf("%f is the minimum numeric value.\n",z);
}
```

Output

```
Value 1: 1.000000
Value 2: 2.000000
1.000000 is the minimum numeric value.
```

See Also

fmod

Synopsis

```
#include <math.h>
type fmod(type x, double y);
double fmod(double x, double y);
float fmodf(float x, float y);
long double fmodl(long double x, long double y);
```

Purpose

Calculate remainder.

Return Value

The **fmod** functions compute the floating-point remainder of x/y .

Parameters

x Argument.

y Argument.

Description

The **fmod** functions return the value $x-ny$, for some integer n such that, if y is nonzero, the result has the same sign as x and magnitude less than the magnitude of y . If y is zero, whether a domain error occurs or the **fmod** functions return zero is implementation defined. The function returns the following values related to special numbers:

- **fmod**(± 0 , y) returns ± 0 if y is not zero.
- **fmod**(x , y) returns a NaN if x is infinite y is zero.
- **fmod**(x , $\pm\infty$) returns x if x is not infinite.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    double y = 1.7;
    printf("fmod(x,y) = %f\n", fmod(x,y));
}
```

Output

```
fmod(x,y) = 0.300000
```

See Also

fpclassify

Synopsis

```
#include <math.h>
int fpclassify(double x);
```

Purpose

Classifies argument.

Return Value

The **fpclassify** macro returns the value of the number classification macro appropriate to the value of its argument.

Parameters

x Argument.

Description

The **fpclassify** macro classifies its argument value as NaN, infinite, normal, subnormal, or zero. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then classification is based on the type of the argument.

See Also

frexp

Synopsis

```
#include <math.h>
double frexp(double value, int *exp);
float frexpf(float value, int *exp);
long double frexpl(long double value, int *exp);
```

Purpose

Converts number into normalized fractions.

Return Value

The **frexp** functions return the value x , such that x has a magnitude in the interval $[1/2, 1]$ or zero, and *value* equal $x \times 2^{\text{exp}}$. If *value* is zero, both parts of the result are zero.

Parameters

value Argument.

exp Argument.

Description

The **frexp** functions break a floating-point number into a normalized fraction and an integral power of 2. They store the integer in the **int** object pointed to by *exp*. The function returns following values related to special numbers:

- **frexp**(± 0 , *exp*) returns ± 0 .
- **frexp**($\pm \infty$, *exp*) returns $\pm \infty$.
- **frexp**(NaN, *exp*) returns NaN.

Example

```
#include <stdio.h>
#include <math.h>

int main() {
    double value = 3.1415926;
    double x;
    int expt;
    x = frexp(value, &expt);
    printf("x = %f\n", x);
    printf("*exp = %d\n", expt);
}
```

Output

```
x = 0.785398
*exp = 2
```

See Also

exp2(), **expm1()**, **exp()**.

hypot

Synopsis

```
#include <math.h>
double hypot(double x, double y);
float hypotf(float x, float y);
long double hypotl(long double x, long double y);
```

Purpose

Calculate sum of squares.

Return Value

The **hypot** functions return the value of the square root of the sum of the squares.

Parameters

x Argument.

y Argument.

Description

The **hypot** functions compute the square root of the sum of the squares of *x* and *y*, without undue overflow or underflow. A range error may occur. The function returns the following values related to special numbers:

- **hypot**(*x*, *y*) returns $\pm\infty$ if *x* is infinite, even if *y* is a NaN.
- **hypot**(*x*, ± 0) is equivalent to **fabs**(*x*).

Example

```
/* a sample program that use hypot() */
#include <stdlib.h>
#include <math.h>

int main() {
    double z, y;
    double x;

    x = 3;
    y = 4;
    z = hypot(x, y);
    printf("The magnitude of the complex number %2.1f+i%2.1f is %2.1f\n", x, y, z);
}
```

Output

```
the model of the complex 3.0+4.0i is 5.0
```

See Also

ilogb

Synopsis

```
#include <math.h>
int ilogb(double x);
int ilogbf(float x);
int double ilogbl(long double x);
```

Purpose

Calculate exponent.

Return Value

The **ilogb** functions return the exponent of x as a signed **int** value.

Parameters

x Argument.

Description

The **ilogb** functions extract the exponent of x as a signed **int** value. If x is zero they compute the value **FP_ILOGB0**; if x is infinite they compute the value **INT_MAX**; if x is a NaN they compute the value **FP_ILOGBNAN**; otherwise, they are equivalent to calling the corresponding **logb** function and casting the returned value to type **int**. A range error may occur if x is 0.

Example

```
/* The ilogb functions return the signed exponent of x as a signed int value */
#include <stdlib.h>
#include <math.h>

int main() {
    double x = 10.0;
    printf("The result is %d\n", ilogb(x));
}
```

Output

The result is 3

See Also

isfinite

Synopsis

```
#include <math.h>
int isfinite(double x);
```

Purpose

Determines if an argument has a finite value.

Return Value

The **isfinite** macro returns a nonzero value if and only if its argument has a finite value.

Parameters

x Argument.

Description

The **isfinite** macro determines whether its argument has a finite value (zero, subnormal, or normal, and not infinite or NaN). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Example

```
/* a sample program that decides if x is finite. */
#include <math.h>
#include <stdio.h>

int main() {
    double x;

    x = 1.0;
    if(isfinite(x))
        printf("%f is a finite number\n",x);
    else
        printf("%f is not a finite value\n", x);
}
```

Output

```
1.000000 is a finite number
```

See Also

isgreater

Synopsis

```
#include <math.h>
```

```
double isgreater(real-floating x, real-floating y);
```

Purpose

Calculate greater value.

Return Value

The **isgreater** macro returns the value for $(x) > (y)$.

Parameters

x Argument.

y Argument.

Description

The **isgreater** macro determines whether its first argument is greater than its second argument. The value of **isgreater**(*x*,*y*) is always equal to $(x) > (y)$; however, unlike $(x) > (y)$, **isgreater**(*x*,*y*) does not raise the *invalid* exception when *x* and *y* are unordered.

Example

```
#include <stdio.h>
#include <math.h>

int main() {
    double x = 2.0, y=1.0;
    printf("isgreater(%f %f)= %d\n", x, y, isgreater(x, y));
}
```

Output

```
isgreater(2.000000 1.000000)= 1
```

See Also

isgreaterequal

Synopsis

```
#include <math.h>
```

```
double isgreaterequal(real-floating x, real-floating y);
```

Purpose

Calculate greater or equal value.

Return Value

The **isgreaterequal** macro returns the value of $(x) \geq (y)$.

Parameters

x Argument.

y Argument.

Description

The **isgreaterequal** macro determines whether its first argument is greater than or equal to its second argument. The value of **isgreaterequal**(*x*,*y*) is always equal to $(x) \geq (y)$; however, unlike $(x) \geq (y)$, **isgreaterequal**(*x*,*y*) does not raise the *invalid* exception when *x* and *y* are unordered.

Example

```
#include <stdio.h>
#include <math.h>

int main() {
    double x = 2.0, y=1.0, z = 2.0;
    printf("isgreaterequal(%f %f)= %d\n", x, y, isgreaterequal(x, y));
    printf("isgreaterequal(%f %f)= %d\n", x, z, isgreaterequal(x, z));
}
```

Output

```
isgreaterequal(2.000000 1.000000)= 1
isgreaterequal(2.000000 2.000000)= 1
```

See Also

isinf

Synopsis

```
#include <math.h>
int isinf(double x);
```

Purpose

Determines if argument has an infinite value.

Return Value

The **isinf** macro returns a nonzero value if and only if its argument has a infinite value.

Parameters

x Argument.

Description

The **isinf** macro determines whether its argument is an infinity (positive or negative). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Example

```
/* a sample program that decides if x is infinite. */
#include <math.h>
#include <stdio.h>

int main() {
    double x;

    x = 1.0;
    if(isinf(x))
        printf("%f is infinite.\n",x);
    else
        printf("%f is not infinite. \n", x);
    x = +INFINITY;
    if(isinf(x))
        printf("%f is infinite.\n",x);
    else
        printf("%f is not infinite. \n", x);
}
```

Output

```
1.000000 is not infinite.
Inf is infinite.
```

See Also

isless

Synopsis

```
#include <math.h>
```

```
double isless(real-floating x, real-floating y);
```

Purpose

Calculate smaller value.

Return Value

The **isless** macro returns the value for $(x) < (y)$.

Parameters

x Argument.

y Argument.

Description

The **isless** macro determines whether its first argument is less than its second argument. The value of **isless**(*x*,*y*) is always equal to $(x) < (y)$; however, unlike $(x) < (y)$, **isless**(*x*,*y*) does not raise the *invalid* exception when *x* and *y* are unordered.

Example

```
#include <stdio.h>
#include <math.h>

int main() {
    double x = 2.0, y=1.0;
    printf("isless(%f %f)= %d\n", x, y, isless(x, y));
}
```

Output

```
isless(2.000000 1.000000)= 0
```

See Also

islessequal

Synopsis

```
#include <math.h>
```

```
double islessequal(real-floating x, real-floating y);
```

Purpose

Calculate less or equal value.

Return Value

The **islessequal** macro returns the value of $(x) \leq (y)$.

Parameters

x Argument.

y Argument.

Description

The **islessequal** macro determines whether its first argument is less than or equal to its second argument. The value of **islessequal**(*x*,*y*) is always equal to $(x) \leq (y)$; however, unlike $(x) \leq (y)$, **islessequal**(*x*,*y*) does not raise the *invalid* exception when *x* and *y* are unordered.

Example

```
#include <stdio.h>
#include <math.h>

int main() {
    double x = 2.0, y=1.0, z = 2.0;
    printf("islessequal(%f %f)= %d\n", x, y, islessequal(x, y));
    printf("islessequal(%f %f)= %d\n", x, z, islessequal(x, z));
}
```

Output

```
islessequal(2.000000 1.000000)= 0
islessequal(2.000000 2.000000)= 1
```

See Also

islessgreater

Synopsis

```
#include <math.h>
```

```
double islessgreater(real-floating x, real-floating y);
```

Purpose

Determine which value is greater.

Return Value

The **islessgreater** macro returns the value of $(x) \leq (y) \parallel (x) \geq (y)$.

Parameters

x Argument.

y Argument.

Description

The **islessgreater** macro determines whether its first argument is less than or greater than its second argument. The **islessgreater** macro is similar to $(x) \leq (y) \parallel (x) \geq (y)$; however, **islessgreater**(*x*,*y*) does not raise the *invalid* exception when *x* and *y* are unordered (nor does it evaluate *x* and *y* twice).

Example

```
#include <stdio.h>
#include <math.h>

int main() {
    double x = 2.0, y=1.0, z = 2.0;
    printf("islessgreater(%f %f)= %d\n", x, y, islessgreater(x, y));
    printf("islessgreater(%f %f)= %d\n", y, z, islessgreater(y, z));
}
```

Output

```
islessgreater(2.000000 1.000000)= 1
islessgreater(1.000000 2.000000)= 1
```

See Also

isnan

Synopsis

```
#include <math.h>
```

```
int isnan(real-floating x);
```

Purpose

Determines if argument has an NaN value.

Return Value

The **isnan** macro returns a nonzero value if and only if its argument has a NaN value.

Parameters

x Argument.

Description

The **isnan** macro determines whether its argument is a NaN. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Example

```
/* a sample program that assigns NaN (not a number) to x
and decide if it is NaN. */
#include <math.h>
#include <stdio.h>

int main() {
    double x;

    x = NaN;
    if(isnan(x)) printf("%f is not a number\n",x);
}
```

Output

```
NaN is not a number
```

See Also

isnormal

Synopsis

```
#include <math.h>
int isnormal(double x);
```

Purpose

Determines if argument has a normal value.

Return Value

The **isnormal** macro returns a nonzero value if and only if its argument has a normal value.

Parameters

x Argument.

Description

The **isnormal** macro determines whether its argument value is normal (neither zero, subnormal, infinite, nor NaN). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Example

```
#include <stdio.h>
#include <math.h>

int main() {
    printf("isnormal(3) = %d\n", isnormal(3));
    printf("isnormal(-3) = %d\n", isnormal(-3));
    printf("isnormal(0) = %d\n", isnormal(0));
    printf("isnormal(-0) = %d\n", isnormal(-0));
    printf("isnormal(Inf) = %d\n", isnormal(Inf));
    printf("isnormal(-Inf) = %d\n", isnormal(-Inf));
    printf("isnormal(NaN) = %d\n", isnormal(NaN));
}
```

Output

```
isnormal(3) = 1
isnormal(-3) = 1
isnormal(0) = 0
isnormal(-0) = 0
isnormal(Inf) = 0
isnormal(-Inf) = 0
isnormal(NaN) = 0
```

See Also

isunordered

Synopsis

```
#include <math.h>
```

```
double isunordered(real-floating x, real-floating y);
```

Purpose

Checks for order.

Return Value

The **isunordered** macro determines whether its arguments are unordered.

Parameters

x Argument.

y Argument.

Description

The **isunordered** macro returns 1 if its arguments are unordered and 0 otherwise.

Example

Output

See Also

ldexp

Synopsis

```
#include <math.h>
double ldexp(double x, int exp);
float ldexpf(float x, int exp);
long double ldexpl(long double x, int exp);
```

Purpose

Multiply floating-point number by power of 2.

Return Value

The **ldexp** functions return the value of $x \times 2^{\text{exp}}$.

Parameters

x Argument.

exp Exponent.

Description

The **ldexp** functions multiply a floating-point number by an integral power of 2. A range error may occur.

Example

Output

```
ldexp(3.000000, 2) = 12.000000
```

See Also

lgamma

Synopsis

```
#include <math.h>
double lgamma(double x);
float lgammaf(float x);
long double lgammal(long double x);
```

Purpose

Calculate absolute logarithmic gamma value.

Return Value

The **lgamma** functions return the gamma function value.

Parameters

x Argument.

Description

The **lgamma** functions compute the natural logarithm of the absolute value of gamma of *x*: $\log_e |\Gamma(x)|$. A range error may occur if the magnitude of *x* is too large or too small.

Example

```
/* The lgamma functions compute the natural logarithm of
   the absolute value of Gamma of x. */
#include <stdio.h>
#include <math.h>

int main() {
    double x = -12.5;

    printf("ln(|Gamma (%2.1f)| is %2.1f\n", x, lgamma(x));
}
```

Output

```
ln(|Gamma (-12.5)| is -20.1
```

See Also

log

Synopsis

```
#include <math.h>
double log(double x);
float logf(float x);
long double logl(long double x);
```

Purpose

Calculate base- e logarithm.

Return Value

The **log** functions return the base- e logarithm value.

Parameters

x Argument.

Description

The **log** functions compute the base- e (natural) logarithm of x . A domain error occurs if the argument is negative. A range error may occur if the argument is zero. The function returns the following values related to special numbers:

- **log**(± 0) returns $-\infty$.
- **log**(1) returns +0.
- **log**(x) returns NaN if $x < 0$.
- **log**($+\infty$) returns $+\infty$.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("log(x) = %f\n", log(x));
}
```

Output

```
log(x) = -1.203973
```

See Also

log10

Synopsis

```
#include <math.h>
double log10(double x);
float loggf(float x);
long double loggl(long double x);
```

Purpose

Calculate base-10 logarithm.

Return Value

The **log10** functions return the base-10 logarithm value.

Parameters

x Argument.

Description

The **log10** functions compute the base-10 (common) logarithm of *x*. A domain error occurs if the argument is negative. A range error may occur if the argument is zero. The function returns the following values related to special numbers:

- **log10**(±0) returns $-\infty$.
- **log10**(1) returns +0.
- **log10**(*x*) returns NaN if $x < 0$.
- **log10**($+\infty$) returns $+\infty$.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("log10(x) = %f\n", log10(x));
}
```

Output

```
log10(x) = -0.522879
```

See Also

log1p

Synopsis

```
#include <math.h>
double log1p(double x);
float log1pf(float x);
long double log1pl(long double x);
```

Purpose

Calculate base- e logarithm.

Return Value

The **log1p** functions return the value of the base- e logarithm of 1 plus the argument.

Parameters

x Argument.

Description

The **log1p** functions compute the base- e (natural) logarithm of 1 plus the argument. A domain error occurs if the argument is less than -1. A range error may occur if the argument equals -1. The function returns the following values related to special numbers:

- **log**(± 0) returns ± 0 .
- **log**(-1) returns $-\infty$.
- **log**(x) returns NaN if $x < -1$.
- **log**($+\infty$) returns $+\infty$.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("log(x) = %f\n", log(x));
}
```

Output

```
log(x) = -1.203973
```

See Also

log2

Synopsis

```
#include <math.h>
double log2(double x);
float log2f(float x);
long double log2l(long double x);
```

Purpose

Calculate base- e logarithm.

Return Value

The **log2** functions return the base-2 logarithm value.

Parameters

x Argument.

Description

The **log2** functions compute the base-2 logarithm of x . A domain error occurs if the argument is less than zero. A range error may occur if the argument is zero. The function returns the following values related to special numbers:

- **log10**(± 0) returns $-\infty$.
- **log10**(x) returns NaN if $x < 0$.
- **log10**($+\infty$) returns $+\infty$.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 2;
    printf("log2(x) = %f\n", log2(x));
}
```

Output

```
log2(x) = 1.000000
```

See Also

logb

Synopsis

```
#include <math.h>
double logb(double x);
float logbf(float x);
long double logbl(long double x);
```

Purpose

Returns signed exponent.

Return Value

The **logb** functions return the signed exponent of x .

Parameters

x Argument.

Description

The **logb** functions extract the exponent of x , as a signed integer value in floating-point format. If x is sub-normal it is treated as though it were normalized; thus, for positive finite x ,

$$1 \leq x \times \text{FLT_RADIX}^{-\text{log}(x)} \leq \text{FLT_RADIX}.$$

A domain error may occur if the argument is zero. The function returns the following values related to special numbers:

- **log10**(± 0) returns $-\infty$.
- **log10**($\pm \infty$) returns $+\infty$.

Example

```
/* The logb functions return the signed exponent of x. */
#include <stdlib.h>
#include <math.h>

int main() {
    double x = 10.0;
    printf("The result is %f\n", logb(x));
}
```

Output

The result is 3.000000

See Also

lrint

Synopsis

```
#include <math.h>
long int lrint(double x);
long int lrintf(float x);
long int lrintl(long double x);
long long int llrint(double x);
long long int llrintf(float x);
long long int llrintl(long double x);
```

Purpose

Calculate rounded integer.

Return Value

The **lrint** and **llrint** functions return the rounded integer value.

Parameters

x Argument.

Description

The **lrint** and the **llrint** functions round their argument to the nearest integer value, rounding according to the current rounding direction. If the rounded value is outside the range of the return type, the numeric result is unspecified. A range error may occur if the magnitude of *x* is too large.

Example

Output

See Also

lround

Synopsis

```
#include <math.h>
long int lround(double x);
long int lroundf(float x);
long int lroundl(long double x);
long long int llround(double x);
long long int llroundf(float x);
long long int llroundl(long double x);
```

Purpose

Calculate rounded integer.

Return Value

The **lround** and **llround** functions return the rounded integer value.

Parameters

x Argument.

Description

The **lround** and the **llround** functions round their argument to the nearest integer value, rounding halfway cases away from zero, regardless of the current rounding direction. If the rounded value is outside the range of the return type, the numeric result is unspecified. A range error may occur if the magnitude of *x* is too large.

Example

Output

See Also

modf

Synopsis

```
#include <math.h>
double modf(double value, double *iptr);
float modff(float value, float *iptr);
long double modfl(long double value, long double *iptr);
```

Purpose

Return fractional value.

Return Value

The **modf** functions return the value of the signed fractional part of *value*.

Parameters

value Argument.

iptr Pointer.

Description

The **modf** functions break the argument *value* into integral and fractional parts, each of which has the same type and sign as the argument. They store the integral part (in floating-point format) in the object pointed to by *iptr*. The function returns the following values related to special numbers:

- **modf**(*value*, *iptr*) returns a result with the same sign as the argument *value*.
- **modf**($\pm\infty$, *iptr*) returns ± 0 and stores $\pm\infty$ in the object pointed to by *iptr*.
- **modf** of a NaN argument stores a NaN in the object pointed to by *iptr* (and returns a NaN).

Example

Output

See Also

nan

Synopsis

```
#include <math.h>
double nan(const char *tagp);
float nanf(const char *tagp);
long double nanl(const char *tagp);
```

Purpose

Return a NaN.

Return Value

The **nan** functions return a quiet NaN, if available, with content indicated through *tagp*. If the implementation does not support quiet NaNs, the functions return zero.

Parameters

tagp Argument.

Description

The call **nan**("n-char-sequence") is equivalent to **strtod**("nan(n-char-sequence)", (char**) NULL); the call **nan**("") is equivalent to **strtod**"nan()", (char**) NULL). IF *tagp* does not point to an n-char-sequence or an empty string, the call is equivalent to **strtod**("nan", (char**) NULL). Calls to **nanf** and **nanl** are equivalent to the corresponding calls to **strtof** and **strtold**.

Example

Output

See Also

nearbyint

Synopsis

```
#include <math.h>
double nearbyint(double x);
float nearbyintf(float x);
long double nearbyintl(long double x);
```

Purpose

Calculate rounded integer.

Return Value

The **nearbyint** functions return the rounded integer value.

Parameters

x Argument.

Description

The **nearbyint** functions round their argument to an integer value in floating-point format, using the current rounding direction and without raising the *inexact* exception.

Example

Output

See Also

nextafter

Synopsis

```
#include <math.h>
double nextafter(double x, double y);
float nextafterf(float x, float y);
long double nextafterl(long double x, long double y);
```

Purpose

Returns next value.

Return Value

The **nextafter** functions return the next representable value in the specified format after x in the direction of y .

Parameters

x Argument.

y Argument.

Description

The **nextafter** functions determine the next representable value, in the type of the function, after x in the direction of y , where x and y are first converted to the type of the function. The **nextafter** functions return y if x equals y . A range error may occur if the magnitude of x is the largest finite value representable in the type, or the result is infinite or not representable in the type.

Example

```
/* a sample program that computes the next representable value */
#include <math.h>
#include <stdio.h>

int main() {
    double x, y, z;

    x = 5.0;
    y = 10.0;
    z = nextafter(x, y);
    printf("%f is the next representable value.\n", z);
}
```

Output

```
5.000000 is the next representable value.
```

See Also

nexttoward

Synopsis

```
#include <math.h>
```

```
double nexttoward(double x, long double y);
```

```
float nexttowardf(float x, long double y);
```

```
long double nexttowardl(long double x, long double y);
```

Purpose

Return fractional value.

Return Value

Parameters

x Argument.

y Argument.

Description

The **nexttoward** functions are equivalent to the **nextafter** functions except that the second parameter has type **long double**.

Example

Output

See Also

pow

Synopsis

```
#include <math.h>
type pow(type x, type y);
double pow(double x, double y);
float powf(float x, float y);
long double powl(long double x, long double y);
```

Purpose

Calculate value to a power.

Return Value

The **pow** functions return the value of x raised to the power y .

Parameters

x Argument.

y Argument.

Description

The **pow** functions compute x raised to the power y . A domain error occurs if x is negative and y is finite and not an integer value. A domain error occurs if the result cannot be represented when x is zero and y is less than or equal to zero. The function returns the following values related to special numbers:

- **pow**(x , ± 0) returns 1 for any x , even a NaN.
- **pow**(x , $+\infty$) returns $+\infty$ for $|x| > 1$.
- **pow**(x , $+\infty$) returns +0 for $|x| < 1$.
- **pow**(x , $-\infty$) returns +0 for $|x| > 1$.
- **pow**(x , $-\infty$) returns $+\infty$ for $|x| < 1$.
- **pow**($+\infty$, y) returns $+\infty$ for $|y| > 0$.
- **pow**($+\infty$, y) returns +0 for $|y| < 0$.
- **pow**($-\infty$, y) returns $-\infty$ for y an odd integer > 0 .
- **pow**($-\infty$, y) returns $+\infty$ for $y > 0$ and not an odd integer.
- **pow**($-\infty$, y) returns -0 for y an odd integer < 0 .
- **pow**($-\infty$, y) returns +0 for $y < 0$ and not an odd integer.
- **pow**(± 1 , $\pm\infty$) returns a NaN.
- **pow**(x , y) returns a NaN for finite $x < 0$ and finite non-integer y .
- **pow**(± 0 , y) returns $\pm\infty$ for y an odd integer < 0 .
- **pow**(± 0 , y) returns $+\infty$ for $y < 0$ and not an odd integer.
- **pow**(± 0 , y) returns ± 0 for y an odd integer > 0 .
- **pow**(± 0 , y) returns +0 for $y > 0$ and not an odd integer.

Example Output

```
pow(2.000000 3.000000) = 8.000000
```

rad2deg

Synopsis

```
#include <math.h>
```

```
double rad2deg(double x);
```

Purpose

Convert an angle in degrees to an angle in radians.

Return Value

The **rad2deg** function returns the angle in radians.

Parameters

x An angle in degrees.

Description

The function **rad2deg**() converts the angle *x* in degrees to an angle in radians by the formula

$$\text{radian} = \text{degree} * 180/\pi$$

Example

See an example in **deg2rad**().

See Also

deg2rad().

remainder

Synopsis

```
#include <math.h>
double remainder(double x, double y);
float remainderf(float x, float y);
long double remainderl(long double x, long double y);
```

Purpose

Calculate remainder.

Return Value

The **remainder** functions return the value of $x \text{ REM } y$.

Parameters

x Argument.

y Argument.

Description

The **remainder** functions compute the remainder $x \text{ REM } y$.

Example

```
/* The function returns the value of  $r = x \text{ REM } y$ .  $r = x - ny$ , where  $n$  is the
   integer nearest the exact value of  $x/y$ . */
#include <math.h>
#include <stdio.h>

int main() {
    double x, y, z;

    x = 5.0;
    y = 2.0;
    z = remainder(x, y);
    printf("x = %f\n", x);
    printf("y = %f\n", y);
    printf("Remainder = %f.\n", z);
}
```

Output

```
x = 5.000000
y = 2.000000
Remainder = 1.000000.
```

See Also

remquo

Synopsis

```
#include <math.h>
double remquo(double x, double y, int*quo);
float remquof(float x, float y, int*quo);
long double remquol(long double x, long double y, int*quo);
```

Purpose

Return fractional value.

Return Value

The **remquo** functions return the value of $x \text{ REM } y$.

Parameters

x Argument.

y Argument.

quo Pointer.

Description

The **remquo** functions compute the same remainder as the **remainder** functions. In the object pointed to by *quo* they store a value whose sign is the sign of x/y and whose magnitude is congruent modulo 2^n to the magnitude of the integral quotient of x/y , where n is an implementation-defined integer greater than or equal to 3.

Example

Output

See Also

rint

Synopsis

```
#include <math.h>
double rint(double x);
float rintf(float x);
long double rintl(long double x);
```

Purpose

Calculate rounded integer.

Return Value

The **rint** functions return the rounded integer value.

Parameters

x Argument.

Description

The **rint** functions differ from the **nearbyint** functions only in that the **rint** functions may raise the inexact exception if the result differs in value from the argument. The function returns the following values related to special numbers:

- **rint**(± 0) returns ± 0 (for all rounding directions).
- **rint**($\pm \infty$) returns $\pm \infty$ (for all rounding directions).

Example

```
/* a sample program that rounds a double number. */
#include <stdlib.h>
#include <math.h>

int main() {
    double x = 213.645;
    printf("%3.3f is rounded to %f \n", x, rint(x));
}
```

Output

```
213.645 is rounded to 214.000000
```

See Also

round

Synopsis

```
#include <math.h>
double round(double x);
float roundf(float x);
long double roundl(long double x);
```

Purpose

Calculate rounded integer.

Return Value

The **round** functions return the rounded integer value.

Parameters

x Argument.

Description

The **round** functions round their argument to the nearest integer value in floating-point format, rounding halfway cases away from zero, regardless of the current rounding direction.

Example

Output

See Also

scalbn and scalbln

Synopsis

```
#include <math.h>
double scalbn(double x, int n);
float scalbnf(float x, int n);
long double scalbnl(long double x, int n);
double scalbln(double x, int n);
float scalblnf(float x, int n);
long double scalblnl(long double x, int n);
```

Purpose

Return the value of $x \times \text{FLT_RADIX}^n$.

Return Value

The **scalbn** and **scalbln** functions return the value of $x \times \text{FLT_RADIX}^n$.

Parameters

x Argument.

n Integer.

Description

The **scalbn** and **scalbln** functions compute $x \times \text{FLT_RADIX}^n$ efficiently, not normally, by computing FLT_RADIX^n explicitly. The function returns the following values related to special numbers:

- **scalbn**(*x*, *n*) returns *x* if *x* is infinite or zero.
- **scalbn**(*x*, 0) returns *x*.

Example

```
/* a sample program that decides which of the two values is a max. */
#include <math.h>
#include <stdio.h>

int main() {
    double x, z;
    int y;
    x = 5.0;
    y = 2;
    z =scalbn(x, y);
    printf("Value 1: %f\n", x);
    printf("Value 2: %d\n", y);
    printf("%f is the maximum numeric value.\n", z);
}
```

Output

```
Value 1: 5.000000
Value 2: 2
20.000000 is the maximum numeric value.
```

See Also

signbit

Synopsis

```
#include <math.h>
int signbit(double x);
```

Purpose

Determines if argument has a negative value.

Return Value

The **signbit** macro returns a nonzero value if and only if its argument has a negative value.

Parameters

x Argument.

Description

The **signbit** macro determines whether its argument value is negative.

Example

```
#include <stdio.h>
#include <math.h>

int main() {
    printf("signbit(3) = %d\n", signbit(3));
    printf("signbit(-3) = %d\n", signbit(-3));
}
```

Output

See Also

sin

Synopsis

```
#include <math.h>
```

```
type sin(type x);
```

```
double sin(double x);
```

```
float sinf(float x);
```

```
long double sinl(long double x);
```

Purpose

Calculate the sine.

Return Value

The **sin** functions return the sine value.

Parameters

x Argument.

Description

The **sin** functions compute the sine of *x* (measured in radians). The function returns the following values related to special numbers:

— **sin**(±0) returns ±0.

— **sin**(±∞) returns a NaN.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("sin(x) = %f\n", sin(x));
}
```

Output

```
sin(x) = 0.295520
```

See Also

asin(), **asinh**(), **sinh**().

sinh

Synopsis

```
#include <math.h>
type sinh(type x);
double sinh(double x);
float sinhf(float x);
long double sinhl(long double x);
```

Purpose

Calculate the hyperbolic sine.

Return Value

The **sinh** functions return the hyperbolic sine value.

Parameters

x Argument.

Description

The **sinh** functions compute the hyperbolic sine of *x*. A range error occurs if the magnitude of *x* is too large.

The function returns the following values related to special numbers:

- **sinh**(±0) returns ±0.
- **sinh**(±∞) returns ±∞.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("sinh(x) = %f\n", sinh(x));
}
```

Output

```
sinh(x) = 0.304520
```

See Also

sin(), **asinh()**, **asin()**.

sqrt

Synopsis

```
#include <math.h>
double sqrt(double x);
float sqrtf(float x);
long double sqrtl(long double x);
```

Purpose

Calculate square root.

Return Value

The **sqrt** functions return the value of the square root.

Parameters

x Argument.

Description

The **sqrt** functions compute the non-negative square root of *x*. A domain error occurs if the argument is less than zero.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("sqrt(x) = %f\n", sqrt(x));
}
```

Output

```
sqrt(x) = 0.547723
```

See Also

tan

Synopsis

```
#include <math.h>
type tan(type x);
double tan(double x);
float tanf(float x);
long double tanl(long double x);
```

Purpose

Calculate the tangent.

Return Value

The **tan** functions return the tangent value.

Parameters

x Argument.

Description

The **tan** functions compute the tangent of *x* (measured in radians). The function returns the following values related to special numbers:

- **tan**(± 0) returns ± 0 .
- **tan**($\pm\infty$) returns NaN.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("tan(x) = %f\n", tan(x));
}
```

Output

```
tan(x) = 0.309336
```

See Also

atan(), **atanh()**, **tanh()**.

tanh

Synopsis

```
#include <math.h>
type tanh(type x);
double tanh(double x);
float atanhf(float x);
long double atanh1(long double x);
```

Purpose

Calculate the hyperbolic tangent.

Return Value

The **tanh** functions return the hyperboilc tangent value.

Parameters

x Argument.

Description

The **tanh** functions compute the hyperboilc tangent of *x*. The function returns the following values related to special numbers:

- **tanh**(±0) returns ±0.
- **tanh**(±∞) returns ±1.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("tanh(x) = %f\n", tanh(x));
}
```

Output

```
tanh(x) = 0.291313
```

See Also

tan(), **atanh()**, **atan()**.

tgamma

Synopsis

```
#include <math.h>
double tgamma(double x);
float tgammaf(float x);
long double tgammal(long double x);
```

Purpose

Calculate gamma value.

Return Value

The **tgamma** functions return the gamma function value.

Parameters

x Argument.

Description

The **tgamma** functions compute the gamma function of x : $\Gamma(x)$. A domain error occurs if x is a negative integer or zero. A range error may occur if the magnitude of x is too large or too small. The function returns the following values related to special numbers:

- **tgamma**($+\infty$) returns $+\infty$.
- **tgamma**(x) returns $+\infty$ if x is a negative integer or zero.
- **tgamma**($-\infty$) returns a NaN.

Example

Output

See Also

trunc

Synopsis

```
#include <math.h>
double trunc(double x);
float truncf(float x);
long double trunc(long double x);
```

Purpose

Calculate truncated integer.

Return Value

The **trunc** functions return the truncated integer value.

Parameters

x Argument.

Description

The **trunc** functions round their argument to the integer value, in floating-point format, nearest to but no larger in magnitude than the argument.

Example

Output

See Also

Chapter 11

Numeric Analysis — <numeric.h>

Header file **numeric.h** contains the definitions of high-level numeric analysis functions, which are available only in Ch Professional Edition. It includes the following header files **math.h**, **stdarg.h**, **array.h** and **dlfcn.h**. The following high-level functions are designed for numeric analysis.

Function	Description
balance()	Balances a general real matrix to improve accuracy of computed eigenvalues.
ccompanionmatrix()	Calculate companion matrix with complex number.
cdeterminant()	Calculate the determinant of a complex matrix.
cdiagonal()	Get a vector with diagonals of a complex matrix.
cdiagonalmatrix()	Create diagonal matrix from a complex vector.
cdivisor()	Calculate common divisors of two integers.
cdivisornum()	Calculate the number of common divisors of two integers.
cfevalarray()	Complex array function evaluation.
cfunm()	Evaluate general complex matrix function.
charpolycoef()	Calculate the coefficients of characteristic polynomial of a matrix.
choldecomp()	Computes the Cholesky factorization of a symmetric positive definite matrix A.
cinverse()	Calculate the inverse of a complex square matrix.
clinsolve()	Solve the linear system of complex equations by LU decomposition.
cmean()	Calculate the mean value of all the elements and mean values of each row of a complex array.
combination()	Calculate the number of combination of n things taken k at a time.
companionmatrix()	Calculate a companion matrix.
complexsolve()	Solve a complex equation.
complexsolvePP()	Solve a complex equation.
complexsolvePR()	Solve a complex equation.
complexsolveRP()	Solve a complex equation.
complexsolveRR()	Solve a complex equation.
complexsolveRRz()	Solve a complex equation.
condnum()	Calculate condition number of a matrix.
constrain()	Constrain a number to be within a range.

conv()	One-dimensional Discrete Fourier Transform (DFT) based convolution.
conv2()	Two-dimensional Discrete Fourier Transform (DFT) based convolution.
corrcoef()	Correlation coefficients calculation.
correlation2()	Obtain a two-dimensional correlation coefficient.
cpolyeval()	Calculate the value of a complex polynomial at a complex point.
cproduct()	Calculate product of all elements or products of the elements of a complex array of any dimension and the products of each row of a two-dimensional complex array.
cquadratic()	Compute the solution of a quadratic equation in complex number.
cross()	Calculate vector cross product.
csum()	Calculate the sum of all elements of a complex array of any dimension or sums of elements in each row of a two-dimensional complex array.
ctrace()	Calculate the sum of diagonal elements of a complex matrix.
ctriangularmatrix()	Get the upper or lower triangular part of a complex matrix.
cumprod()	Calculate cumulative product of elements in an array
cumsum()	Calculate cumulative sum of elements in an array
curvefit()	Fit a set of data points x and y to a linear combination of specified base functions.
covariance()	Covariance matrix.
deconv()	One-dimensional Discrete Fourier Transform (DFT) based deconvolution.
derivative()	Calculate numerical derivative of a function at a given point.
derivatives()	Calculate numerical derivatives of a function at multiple points.
determinant()	Calculate the determinant of a matrix.
diagonal()	Get a vector with diagonals of a matrix.
diagonalmatrix()	Create diagonal matrix of a vector.
difference()	Calculate differences between adjacent elements of array.
divisor()	Calculate divisors of a number.
divisornum()	Calculate the number of divisors of a number.
dot()	Calculate vector dot product.
eigen()	Find eigenvalues and eigenvectors.
eigensystem()	Find eigenvalues and eigenvectors.
expfit()	Fit a set of data points to an exponential function $y = ab^x$.
expm()	Computes a matrix exponential.
factor()	Calculate the prime factors for a number.
factornum()	Calculate the number of prime factors for a number.
factorial()	Factorial function.
fevalarray()	Array function evaluation.
fft()	N-dimensional Fast Fourier Transform (FFT) calculation.
filter()	Filters the data.
filter2()	Two-dimensional Discrete Fourier Transform based FIR filter.
findvalue()	Obtain indices of nonzero elements of an array.
fliplr()	Flip matrix in left/right direction.
flipud()	Flip matrix in up/down direction.
fminimum()	Find the minimum value of a one-dimensional function and its corresponding position for the minimum value.
fminimums()	Find the minimum position and value of an n-dimensional function.
fsolve()	Find a zero position of a nonlinear system of equations.
funm()	Evaluate general real matrix function.

fzero()	Find a zero position of a nonlinear function with one variable.
fzero1()	Find a zero position of a nonlinear function with one variable.
fzero2()	Find a zero position of a nonlinear function with one variable.
gcd()	Obtain the greatest common divisor of two integers
gcdarray()	Obtain the greatest common divisor of the corresponding value of two integers in two arrays of integral type.
getnum()	Get a number with default value from the console.
hessdecomp()	Reduces a real general matrix to upper Hessenberg form by an orthogonal/unitary matrix similarity transformation.
histogram()	Calculate and plot histograms.
householdermatrix()	Get the Householder matrix.
identitymatrix()	Create an identity matrix.
ifft()	N-dimensional inverse Fast Fourier Transform (FFT) calculation.
integral1()	Numerical integration of a one-dimensional function.
integralp1()	Numerical integration of a one-dimensional function with a parameter.
integral2()	Numerical integration of a two-dimensional function.
integral3()	Numerical integration of a three-dimensional function.
integration2()	Numerical integration of a two-dimensional function.
integration3()	Numerical integration of a three-dimensional function.
interp1()	One-dimensional interpolation.
interp2()	Two-dimensional interpolation.
isprime()	Test if a number is a prime number.
inverse()	Calculate the inverse of a square matrix.
lcm()	Obtain the least common multiple of two integers
lcmarray()	Obtain the least common multiple of the corresponding value of two numbers for two arrays of integral type.
lindata()	Generate a linearly spaced data.
linearcorrcoef()	Calculate the linear correlation coefficient between two variables.
linearfit()	Linear fit for solving m and b with $y=mx+b$ using the linear regression.
linsolve()	Solve a system of linear equations by LU decomposition.
linsolvesi()	Solve a system of two linear equations in slope-intercept form.
linsolvestd()	Solve a system of two linear equations in standard form.
linrange()	Generate a linearly spaced array of double type.
linspace()	Generate a linearly spaced array. Obsolete, use lindata()
llsqcovsolve()	Solve linear system of equations based on linear least-squares with known covariance.
llsqnonnegsolve()	Solve linear system of equations with non-negative values based on linear least-squares method.
llsqsolve()	Least squares solution of sets of linear equations.
logm()	Computes matrix natural logarithm.
logdata()	Generate a logarithmically spaced data.
logrange()	Generate a logarithmically spaced array of double type.
logspace()	Generate a logarithmically spaced array. Obsolete, use logdata()
ludcomp()	LU decomposition of general m-by-n matrix.
maxloc()	Find the location of maximum value of an array.
maxv()	Find the maximum values of the elements of each row of a matrix.
mean()	Calculate the mean value of all the elements and mean values of each row in an array.

median()	Find the median value of all the elements and median values of elements in each row of an array.
minloc()	Find the location of the minimum value of an array.
minv()	Find the minimum values of each row in a two-dimensional array.
mode()	Find the mode for a set of data.
norm()	Calculate vector and matrix norms.
nullspace()	Calculate null space of a matrix.
oderk()	Solve ordinary differential equation using Runge-Kutta Method.
oderungekutta()	Solve ordinary differential equation using Runge-Kutta Method. This function is obsolete.
odesolve()	Solve ordinary differential equation. This function is obsolete.
orthonormalbase()	Find orthonormal bases of a matrix.
pinverse()	Calculate Moore-Penrose pseudoinverse of a matrix.
polyder()	Take derivative of a polynomial.
polyder2()	Calculate the derivative of product or quotient of two polynomials.
polyeval()	Calculate the value of a polynomial and its derivatives.
polyevalarray()	Polynomial evaluation for a sequence of points.
polyevalm()	Calculate the value of a matrix polynomial.
polyfit()	Fit a set of data points to a polynomial function.
polygonperi()	Calculate the perimeter of a polygon.
product()	Calculate product of all elements of an array of any dimension or products of the elements of each row of a two-dimensional array.
qrdecomp()	Compute the orthogonal-triangular QR decomposition of a matrix.
qrdelete()	Remove a column in a QR factorized matrix.
qrinsert()	Insert a column in a QR factorized matrix.
quadratic()	Compute the solution of a quadratic equation.
quadraticfit()	Fit a set of data points to a quadratic function $y = ax^2 + bx + c$.
quartiles()	Calculate seven key statistical values.
randcolor()	Obtain a random color in a string represented in a hex number.
randidouble()	Generate a random float-in-point number of double type.
randint()	Generate a random integer.
randintfreq()	Find the frequency for an integer in a set of random numbers.
rank()	Find the rank of a matrix.
residue()	Partial-fraction expansion or residue computation.
regularpolygon()	Create two arrays of data for a regular polygon.
rcondnum()	Matrix reciprocal condition number estimate.
roots()	Find the roots of a polynomial.
rot90()	Rotate matrix 90 degrees.
rotatepoint()	Rotate a point with respect to another point by an angle.
rotatepolygon()	Rotate a polygon with two arrays of data by an angle.
rotmatrix()	A two-dimensional rotation matrix.
rgb2hex()	Change a color in RGB to color in a string represented in a hex number.
rsf2csf()	Convert a real Schur form to a complex Schur form.
specialmatrix()	Generate a special matrix.
Cauchy	Generate a Cauchy matrix.
ChebyshevVandermonde	Generate a Vandermonde-like matrix for the Chebyshev polynomials.
Chow	Generate a Chow matrix.
Circul	Generate a Circul matrix.
Clement	Generate a Clement matrix.

DenavitHartenberg	Generate a Denavit-Hartenberg matrix.
DenavitHartenberg2	Generate a Denavit-Hartenberg matrix.
Dramadah	Generate a Dramadah matrix.
Fiedler	Generate a Fiedler matrix.
Frank	Generate a Frank matrix.
Gear	Generate a Gear matrix.
Hadamard	Generate a Hadamard matrix.
Hankel	Generate a Hankel matrix.
Hilbert	Generate a Hilbert matrix.
InverseHilbert	Generate a inverse Hilbert matrix.
Magic	Generate a Magic matrix.
Pascal	Generate a Pascal matrix.
Rosser	Generate a Rosser matrix.
Toeplitz	Generate a Toeplitz matrix.
Vandermonde	Generate a Vandermonde matrix.
Wilkinson	Generate a Wilkinson matrix.
schurdecomp()	Compute Schur decomposition.
sign()	Sign function.
sort()	Sorting and ranking elements in ascending order.
sqrtn()	Computes the square root of a matrix.
std()	Calculate the standard deviation of a data set.
sum()	Calculate the sum of all elements or sums of elements in each row of an array.
svd()	Singular value decomposition.
trace()	Calculate the sum of diagonal elements.
triangularmatrix()	Get the upper or lower triangular part of a matrix.
unwrap()	Unwrap radian phase of each element of input array by changing its absolute jump greater than π to its $2 * \pi$ complement.
unwrapdeg()	Unwrap degree phase of each element of input array by changing its absolute jump greater than 180 to its $2 * 180$ complement.
urand()	Uniform random-number generator.
vertex()	Compute the vertex of a quadratic polynomial.
xcorr()	Obtain a one-dimensional cross-correlation vector.

balance

Synopsis

```
#include <numeric.h>
```

```
int balance(array double complex a[&][&], array double complex b[&][&], ...
           /* [array double complex d[&][&]] */);
```

Syntax

```
balance(a, b); balance(a, b, d);
```

Purpose

Balances a general matrix to improve accuracy of computed eigenvalues.

Return Value

This function returns 0 on success and negative value on failure.

Parameters

a An $n \times n$ square matrix to be balanced.

l An output square matrix which contains the balanced matrix of matrix *a*.

d An optional output square matrix which contains a diagonal matrix.

Description

This function computes the balance matrix *b* and diagonal matrix *d* from input matrix *a* so that $b = d^{-1} * a * d$. If *a* is symmetric, then $b == a$ and *d* is the identity matrix.

In this function, square matrix *a* could be any supported arithmetic data type. Output *b* is the same dimension and data type as input *a*. If the input *a* is a real matrix, the optional output *d* shall only be **double** data type. If the input *a* is a complex matrix, *d* shall be **complex** or **double complex**.

Example1

balance a real general matrix.

```
#include <numeric.h>
int main() {
    int m = 5;

    array double a[5][5] = {
        1, 10, 100, 1000, 10000,
        0.1, 1, 10, 100, 1000,
        0.01, 0.1, 1, 10, 100,
        0.001, 0.01, 0.1, 1, 10,
        0.0001, 0.001, 0.01, 0.1, 1};

    int status;
    array double b[m][m], d[m][m];
    status = balance(a, b, d);
    if (status == 0) {
        printf("a =\n%f\n", a);
        printf("b =\n%f\n", b);
        printf("d =\n%f\n", d);
        printf("inverse(d)*a*d - b =\n%f\n", inverse(d)*a*d-b);
    }
    else
        printf("error: calculation error in balance().\n");
}
```

```
}

```

Output1

```
a =
1.000000 10.000000 100.000000 1000.000000 10000.000000
0.100000 1.000000 10.000000 100.000000 1000.000000
0.010000 0.100000 1.000000 10.000000 100.000000
0.001000 0.010000 0.100000 1.000000 10.000000
0.000100 0.001000 0.010000 0.100000 1.000000

```

```
b =
1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000

```

```
d =
1000.000000 0.000000 0.000000 0.000000 0.000000
0.000000 100.000000 0.000000 0.000000 0.000000
0.000000 0.000000 10.000000 0.000000 0.000000
0.000000 0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 0.000000 0.100000

```

```
inverse(d)*a*d - b =
0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000 0.000000
-0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000 0.000000
-0.000000 0.000000 0.000000 0.000000 0.000000

```

Example2

Balance a complex general matrix.

```
#include <numeric.h>

int main() {
    int m = 4;

    array double complex a[4][4] = {
        0, 0, complex(1,1), 0,
        0, 0, 0, complex(1,1),
        complex(11,1), complex(10,1), 0, 0,
        complex(10,1), complex(11,1), 0, 0};

    int status;
    array double complex b[m][m];
    array double d[m][m];
    status = balance(a, b, d);
    if (status == 0) {
        printf("a =\n%4.2f\n", a);
        printf("b =\n%4.2f\n", b);
        printf("d =\n%4.2f\n", d);
        printf("inverse(d)*a*d -b =\n%4.2f\n", inverse(d)*a*d-b);
    }
    else
        printf("error: calculation error in balance().\n");
}

```



```
}

```

Output2

```
a =
complex(0.00,0.00) complex(0.00,0.00) complex(1.00,1.00) complex(0.00,0.00)
complex(0.00,0.00) complex(0.00,0.00) complex(0.00,0.00) complex(1.00,1.00)
complex(11.00,1.00) complex(10.00,1.00) complex(0.00,0.00) complex(0.00,0.00)
complex(10.00,1.00) complex(11.00,1.00) complex(0.00,0.00) complex(0.00,0.00)

```

```
b =
complex(0.00,0.00) complex(0.00,0.00) complex(10.00,10.00) complex(0.00,0.00)
complex(0.00,0.00) complex(0.00,0.00) complex(0.00,0.00) complex(10.00,10.00)
complex(1.10,0.10) complex(1.00,0.10) complex(0.00,0.00) complex(0.00,0.00)
complex(1.00,0.10) complex(1.10,0.10) complex(0.00,0.00) complex(0.00,0.00)

```

```
d =
0.10 0.00 0.00 0.00
0.00 0.10 0.00 0.00
0.00 0.00 1.00 0.00
0.00 0.00 0.00 1.00

```

```
inverse(d)*a*d -b =
complex(0.00,0.00) complex(0.00,0.00) complex(0.00,0.00) complex(0.00,0.00)
complex(0.00,0.00) complex(0.00,0.00) complex(0.00,0.00) complex(0.00,0.00)
complex(0.00,0.00) complex(0.00,0.00) complex(0.00,0.00) complex(0.00,0.00)
complex(0.00,0.00) complex(0.00,0.00) complex(0.00,0.00) complex(0.00,0.00)

```

See Also

eigensystem().

References

E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

ccompanionmatrix

Synopsis

```
#include <numeric.h>
```

```
array double complex ccompanionmatrix(array double complex v[&])[:,:];
```

Purpose

Find companion matrix with complex number.

Return Value

This function returns the complex companion matrix.

Parameters

v Input array containing the complex coefficients of a polynomial.

Description

This function returns the corresponding complex companion matrix of the complex array *v* which contains the coefficients of a polynomial. The eigenvalues of companion matrix are roots of the polynomial.

Example

```
#include <numeric.h>
int main() {
    array double complex v[4] = {complex(1,2),3,4,5}; /* (1+i2)x^3 + 3x^2 + 4x + 5 */
    int n = 4;
    array double complex a[n-1][n-1];

    a = ccompanionmatrix(v);
    printf("ccompanionmatrix(a) =\n%f\n", a);
}
```

Output

```
ccompanionmatrix(a) =
complex(-0.600000,1.200000) complex(-0.800000,1.600000) complex(-1.000000,2.000000)

complex(1.000000,0.000000) complex(0.000000,0.000000) complex(0.000000,0.000000)

complex(0.000000,0.000000) complex(1.000000,0.000000) complex(0.000000,0.000000)
```

See Also

companionmatrix(), **roots()**, **eigensystem()**, **polycoef()**.

References

cdeterminant

Synopsis

```
#include <numeric.h>
```

```
double complex cdeterminant(array double complex a[&][&]);
```

Purpose

Calculate the determinant of a complex matrix.

Return Value

This function returns the determinant of the complex matrix *a*.

Parameters

a The input complex matrix.

Description

The function **cdeterminant()** returns the determinant of complex matrix *a*. If the input matrix is not a square matrix, the function will return NaN.

Example

Calculate the determinant of complex matrices with different data type.

```
#include <numeric.h>
int main() {
    array double complex a[2][2] = {complex(1,2), 4,
                                    3, 7};
    /* a2 is an ill-detection matrix */
    array double a2[2][2] = {2, 4,
                             2.001, 4.0001};
    /* a3 is singular */
    array double a3[2][2] = {2, 4,
                             4, 8};
    array complex b[2][2] = {2, 4,
                             3, 7};
    array double c[3][3] = {-1,5,6,
                             3,-6,1,
                             6,8, 9} ; /* n-by-n matrix */

    double complex det;

    det = cdeterminant(a);
    printf("cdeterminant(a) = %g\n", det);
    det = cdeterminant(a2);
    printf("cdeterminant(a2) = %g\n", det);
    det = cdeterminant(a3);
    printf("cdeterminant(a3) = %g\n", det);
    det = cdeterminant(b);
    printf("cdeterminant(b) = %g\n", det);
    det = cdeterminant(c);
    printf("cdeterminant(c) = %g\n", det);
}
```

Output

```
cdeterminant(a) = complex(-5,14)
cdeterminant(a2) = complex(-0.0038,-0)
```

```
cdeterminant(a3) = complex(-0,-0)
cdeterminant(b) = complex(-2,0)
cdeterminant(c) = complex(317,-0)
```

See Also

determinant(), **inverse()**, **diagonal()**, **ludcomp()**, **rcondnum()**.

References

E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

cdiagonal

Synopsis

```
#include <numeric.h>
```

```
array double complex cdiagonal(array double complex a[&][&],... /* [int k] */[:]);
```

Syntax

```
cdiagonal(a);
```

```
cdiagonal(a, k);
```

Purpose

Form a vector with diagonals of a complex matrix.

Return Value

This function returns a column vector formed from the elements of the k th diagonal of the complex matrix a .

Parameters

a An input complex matrix.

k An input integer indicating which element the vector is formed from.

Description

The function `cdiagonal()` produces a row vector formed from the elements of k th diagonal of the complex matrix a .

Example1

```
#include <numeric.h>
int main() {
    array double complex a[3][3] = {complex(1,2), 2, 3,
                                     4, 5, 6,
                                     7, 8, 9};

    int n =3, k = 1;
    array double complex d[n], d2[n-(abs(k))];

    d = cdiagonal(a);
    printf("cdiagonal(a) = %0.2f\n", d);

    d2 = cdiagonal(a, k);
    printf("cdiagonal(a, 1) = %0.2f\n", d2);

    k = -1;
    d2 = cdiagonal(a, k);
    printf("cdiagonal(a, -1) = %0.2f\n", d2);
}
```

Output1

```
cdiagonal(a) = complex(1.00,2.00) complex(5.00,0.00) complex(9.00,0.00)
```

```
cdiagonal(a, 1) = complex(2.00,0.00) complex(6.00,0.00)
```

```
cdiagonal(a, -1) = complex(4.00,0.00) complex(8.00,0.00)
```

Example2

```
#include <numeric.h>
int main() {
    array double complex a[4][3] = {complex(1,2), 2, 3,
                                   4, 5, 6,
                                   7, 8, 9,
                                   4, 4, 4};
    int m=4, n = 3, k1 = 1, k2 = -1;
    array double complex d[min(m, n)];
    array double complex d1[min(m, n-k1)];
    array double complex d2[min(m+k2, n)];

    d = cdiagonal(a);
    printf("cdiagonal(a) = %0.2f\n", d);

    d1 = cdiagonal(a, k1);
    printf("cdiagonal(a, 1) = %0.2f\n", d1);

    d2 = cdiagonal(a, k2);
    printf("cdiagonal(a, -1) = %0.2f\n", d2);
}
```

Output2

```
cdiagonal(a) = complex(1.00,2.00) complex(5.00,0.00) complex(9.00,0.00)
cdiagonal(a, 1) = complex(2.00,0.00) complex(6.00,0.00)
cdiagonal(a, -1) = complex(4.00,0.00) complex(8.00,0.00) complex(4.00,0.00)
```

Example3

```
#include <numeric.h>
int main() {
    array double complex a[3][4] = {complex(1,2), 2, 3, 4,
                                   5, 6, 7, 8,
                                   9, 4, 4, 4};
    int m=3, n = 4, k1 = 1, k2 = -1;
    array double complex d[min(m, n)];
    array double complex d1[min(m, n-k1)];
    array double complex d2[min(m+k2, n)];

    d = cdiagonal(a);
    printf("cdiagonal(a) = %0.2f\n", d);

    d1 = cdiagonal(a, k1);
    printf("cdiagonal(a, 1) = %0.2f\n", d1);

    d2 = cdiagonal(a, k2);
    printf("cdiagonal(a, -1) = %0.2f\n", d2);
}
```

Output3

```
cdiagonal(a) = complex(1.00,2.00) complex(6.00,0.00) complex(4.00,0.00)
cdiagonal(a, 1) = complex(2.00,0.00) complex(7.00,0.00) complex(4.00,0.00)
cdiagonal(a, -1) = complex(5.00,0.00) complex(4.00,0.00)
```

See Also

diagonal(), **cdiagonalmatrix()**.

References

cdiagonalmatrix

Synopsis

```
#include <numeric.h>
```

```
array double complex cdiagonalmatrix(array double complex v[&],... /* [int k] */)[:][:];
```

Syntax

```
cdiagonalmatrix(v)
```

```
cdiagonalmatrix(v, k)
```

Purpose

Form a complex diagonal matrix.

Return Value

This function returns a complex square matrix with specified diagonal elements.

Parameters

v An input complex vector.

k An input integer indicating specified diagonal elements of the matrix.

Description

The function **cdiagonalmatrix()** returns a complex square matrix of order $n + \text{abs}(k)$, with the elements of *v* on the the *k*th diagonal. $k = 0$ represents the main diagonal, $k > 0$ above the main diagonal, and $k < 0$ below the main diagonal.

Example

```
#include <numeric.h>
int main() {
    array double complex v[3] = {1,2,3};
    int n = 3, k;
    array double complex a[n][n];

    a = cdiagonalmatrix(v);
    printf("cdiagonal matrix a =\n%f\n", a);

    k = 0;
    a = cdiagonalmatrix(v,k);
    printf("cdiagonalmatrix(a, 0) =\n%f\n", a);

    k = 1;
    array double a2[n+abs(k)][n+abs(k)];
    a2 = cdiagonalmatrix(v,k);
    printf("cdiagonalmatrix(a2, 1) =\n%f\n", a2);

    k = -1;
    array double a3[n+abs(k)][n+abs(k)];
    a3 = cdiagonalmatrix(v,k);
    printf("cdiagonalmatrix(a3, -1) =\n%f\n", a3);
}
```


Output

```
cdiagonal matrix a =  
complex(1.000000,0.000000) complex(0.000000,0.000000) complex(0.000000,0.000000)  
complex(0.000000,0.000000) complex(2.000000,0.000000) complex(0.000000,0.000000)  
complex(0.000000,0.000000) complex(0.000000,0.000000) complex(3.000000,0.000000)
```

```
cdiagonalmatrix(a, 0) =  
complex(1.000000,0.000000) complex(0.000000,0.000000) complex(0.000000,0.000000)  
complex(0.000000,0.000000) complex(2.000000,0.000000) complex(0.000000,0.000000)  
complex(0.000000,0.000000) complex(0.000000,0.000000) complex(3.000000,0.000000)
```

```
cdiagonalmatrix(a2, 1) =  
0.000000 1.000000 0.000000 0.000000  
0.000000 0.000000 2.000000 0.000000  
0.000000 0.000000 0.000000 3.000000  
0.000000 0.000000 0.000000 0.000000
```

```
cdiagonalmatrix(a3, -1) =  
0.000000 0.000000 0.000000 0.000000  
1.000000 0.000000 0.000000 0.000000  
0.000000 2.000000 0.000000 0.000000  
0.000000 0.000000 3.000000 0.000000
```

See Also

diagonalmatrix(), **diagonal()**.

References

cdivisor()

Synopsis

```
#include <numeric.h>
```

```
array<int, cdivisor(int num1, int num2)>::;
```

Purpose

Calculate the common divisors of two integers.

Return Value

This function returns an array of common divisors of two integers *num1* and *num2*. If one of two integers is less or equal to 0, it returns an array of one element with the value of -1.

Parameter

num1 An integer.

num2 Another integer.

Description

This function returns the common divisors in an array for two integers *num1* and *num2*. If one of two integers is less or equal to 0, it returns an array of one element with the value of -1.

Example

See `gcd()`.

See Also

`gcd()`, `divisor()`, `divisornum()`, `cdivisornum()`.

cdvisornum()

Synopsis

```
#include <numeric.h>
int cdvisornum(int num, int num2);
```

Purpose

Calculate the number of common divisors of two integers.

Return Value

This function returns the number of common divisors of two integers. If one of two integers is less or equal to 0, it returns -1.

Parameter

num1 An integer.

num2 Another integer.

Description

This function returns the number of common divisors integers *num1* and *num2*. If one of two integers is less or equal to 0, it returns -1.

Example

See `divisornum()`.

See Also

`gcd()`, `divisor()`, `divisornum()`, `cdvisor()`.

cfevalarray

Synopsis

```
#include <numeric.h>
```

```
int cfevalarray(array double &y, double complex (*func)(double complex x), array double &x, ...
                /* [array int &mask, double complex value] */);
```

Syntax

```
cfevalarray(y, func, x)
```

```
cfevalarray(y, func, x, mask, value)
```

Purpose

Complex array function evaluation.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

y A **complex** output array which contains calculated function values.

x A **complex** input array at which the function values will be evaluated.

func A function routine given by the user.

mask An input array of integer. An entry with 1 indicates the corresponding function value will be evaluated, and an entry with 0 indicates that its output value is given by argument *value*.

value The default **complex** value which is used to replace the function value when the entry of *mask* is 0.

Description

This function evaluates the value of a user given function corresponding to the input array *x*. The input array can be of any data type and any dimension. The given function prototype shall be **double complex func(double complex)**.

Example

Evaluation of function $f(x) = x^2$ for complex array of different dimensions.

```
#include <numeric.h>
#define N 2
#define M 3

double complex func(double complex x) {
    return x*x;
}

int main() {
    array float complex y2[N][M], x2[N][M]; /* diff data types, diff dim */
    array int mask1[N][M] = {1, 0, 1, 0, 1, 1};

    linspace(x2, 1, N*M);
    x2[1][2]=complex(2,3);
    cfevalarray(y2, func, x2);
}
```

```
printf("y2 = %5.2f\n", y2);

cfevalarray(y2, func, x2, mask1, -1);
printf("y2 = %5.2f\n", y2);
}
```

Output

```
y2 = complex( 1.00, 0.00) complex( 4.00, 0.00) complex( 9.00, 0.00)
complex(16.00, 0.00) complex(25.00, 0.00) complex(-5.00,12.00)
```

```
y2 = complex( 1.00, 0.00) complex(-1.00, 0.00) complex( 9.00, 0.00)
complex(-1.00, 0.00) complex(25.00, 0.00) complex(-5.00,12.00)
```

See Also**fevalarray()**.

cfunm

Synopsis

```
#include <numeric.h>
```

```
int cfunm(array double complex y[&][&], double complex (*func)(double complex ),
          array double complex x[&][&]);
```

Syntax

```
cfunm(y, func, x)
```

Purpose

Evaluate general complex matrix function.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x Input square matrix. It contains data to be evaluated.

func A function routine given by the user.

y Output square matrix which contains data of the calculated function values.

Description

This function evaluates the complex matrix version of the function specified by the parameter *func*. In this function, The input matrix *x* shall be **double complex** data type and the specified function prototype shall be **double complex** *func*(**double complex**). The output matrix *y* could be a **real** or **complex** data type as required.

Example

A complex matrix evaluation.

```
#include <numeric.h>
double complex mylog(double complex x) {
    return log(x);
}

int main() {
    array double complex zx[3][3]={complex(1,1),complex(2,2),0,
                                   3,complex(4,1),complex(2,5),
                                   0,0,0};
    array double complex zy[3][3];

    expm(zy, zx);
    printf("zx = \n%5.1f", zx);
    printf("zy = \n%5.1f", zy);
    cfunm(zx, mylog, zy);
    printf("zx = \n%5.1f", zx);
}
```

Output

```
zx =
complex( 1.0, 1.0) complex( 2.0, 2.0) complex( 0.0, 0.0)
complex( 3.0, 0.0) complex( 4.0, 1.0) complex( 2.0, 5.0)
complex( 0.0, 0.0) complex( 0.0, 0.0) complex( 0.0, 0.0)
zy =
complex(-44.9, 56.8) complex(-87.5, 70.9) complex(-101.5,-18.9)
complex(-12.5,118.7) complex(-57.4,175.5) complex(-155.5, 67.8)
complex( 0.0, 0.0) complex( 0.0, 0.0) complex( 1.0, 0.0)
zx =
complex( 1.0, 1.0) complex( 2.0, 2.0) complex( 0.0, 0.0)
complex( 3.0, 0.0) complex( 4.0, 1.0) complex( 2.0, 5.0)
complex( 0.0, 0.0) complex( 0.0, 0.0) complex( 0.0, 0.0)
```

See Also

cfunm(), **expm()**, **logm()**, **funm()**, **sqrtm()**.

References

G. H. Golub, C. F. Van Loan, *Matrix Computations* Third edition, The Johns Hopkins University Press, 1996

charpolycoef

Synopsis

```
#include <numeric.h>
```

```
int charpolycoef(array double complex p[&], array double complex a[&][&]);
```

Purpose

Calculates the coefficients of characteristic polynomial of a matrix.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

p An output array which contains the calculated coefficients of the characteristic polynomial of a matrix.

a An input square matrix.

Description

This function calculates the coefficients of the characteristic polynomial of matrix **A** which is defined as $\det(x\mathbf{I} - \mathbf{A})$. The coefficients are in the order of descending powers. The polynomial it represents is $p_0x^n + \dots + p_{n-1}x + p_n$.

Example

This example calculates the polynomial coefficients for three different matrices *a1*, *a2* and *a3*. *a1* is symmetrical with real eigenvalues, *a2* is non-symmetrical with real eigenvalues, *a3* is non-symmetrical with complex eigenvalues, and *a4* is a complex matrix.

```
#include <numeric.h>
int main() {
    /* eigenvalues of a symmetrical matrix are always real numbers */
    array double a1[3][3] = {0.8, 0.2, 0.1,
                            0.2, 0.7, 0.3,
                            0.1, 0.3, 0.6};
    /* eigenvalues of a non-symmetrical matrix can be either real numbers or
       complex numbers */
    /* this non-symmetrical matrix has real eigenvalues */
    array double a2[3][3] = {0.8, 0.2, 0.1,
                            0.1, 0.7, 0.3,
                            0.1, 0.1, 0.6};
    /* this non-symmetrical matrix has complex eigenvalues */
    array double a3[3][3] = {3, 9, 23,
                            2, 2, 1,
                            -7, 1, -9};
    array double complex a4[2][2] = {complex(1,2), 4,
                                     3, 7};

    array double charpolynomial[4];
    array double complex zcharpolynomial[4];
    array double complex zcharpolynomial2[3];

    charpolycoef(charpolynomial, a1);
    printf("charpolynomial from charpolycoef(a1) =\n%f\n", charpolynomial);
```



```
charpolycoef(charpolynomial, a2);
printf("charpolynomial from charpolycoef(a2) =\n%f\n", charpolynomial);

charpolycoef(charpolynomial, a3);
printf("charpolynomial from charpolycoef(a3) =\n%f\n", charpolynomial);

charpolycoef(zcharpolynomial, a3);
printf("zcharpolynomial from charpolycoef(zcharpolynomial, a3) =\n%f\n",
       zcharpolynomial);

charpolycoef(zcharpolynomial2, a4);
printf("zcharpolynomial2 from charpolycoef(zcharpolynomial2, a4) =\n%f\n",
       zcharpolynomial2);
}
```

Output

```
charpolynomial from charpolycoef(a1) =
1.000000 -2.100000 1.320000 -0.245000

charpolynomial from charpolycoef(a2) =
1.000000 -2.100000 1.400000 -0.300000

charpolynomial from charpolycoef(a3) =
1.000000 4.000000 103.000000 -410.000000

zcharpolynomial from charpolycoef(zcharpolynomial, a3) =
complex(1.000000,0.000000) complex(4.000000,0.000000) complex(103.000000,0.000000)
complex(-410.000000,0.000000)

zcharpolynomial2 from charpolycoef(zcharpolynomial2, a4) =
complex(1.000000,0.000000) complex(-8.000000,-2.000000) complex(-5.000000,14.000000)
```

See Also

roots(), **polycoef()**.

References

choldecomp

Synopsis

```
#include <numeric.h>
```

```
int choldecomp(array double complex a[&][&], array double complex l[&][&], ...
               /* [char mode] */);
```

Syntax

```
choldecomp(a, l); choldecomp(a, l, "mode");
```

Purpose

Computes the Cholesky factorization of a symmetric, positive, definite matrix **A**.

Return Value

This function returns 0 on success, a negative value on failure, and positive value *i* indicates that the leading minor of order *i* is not positive definite and the factorization could not be completed.

Parameters

a A symmetric positive definite two-dimensional matrix.

l A two-dimensional output matrix which contains the upper or lower triangle of the symmetric matrix *a*.

mode The it mode specifies the calculation of upper or lower triangle matrix.

'L' or 'l'- lower triangle is calculated.

otherwise the upper triangle matrix is calculated. By default, the upper triangle matrix is calculated.

Description

This function computes Cholesky factorization of a symmetric positive definite matrix **A**. Cholesky decomposition factors a symmetric positive definite matrix into two matrices. For a symmetric positive finite matrix **A** of real type, Cholesky decomposition can produce matrix **L** so that

$$\mathbf{A} = \mathbf{L}^T \mathbf{L}$$

for upper triangle factorization or

$$\mathbf{A} = \mathbf{L} \mathbf{L}^T$$

for lower triangle factorization. where L^T is the transpose of matrix **L**. For a symmetric positive finite matrix **A** of complex type, instead of L^T , the Hermitian L^H of matrix **L** shall be used.

Example1

Cholesky factorization of a real matrix.

```
#include <numeric.h>
int main() {
    int m = 5;
    array double a[5][5] = { 1, 1, 1, 1, 1,
                            1, 2, 3, 4, 5,
                            1, 3, 6,10,15,
                            1, 4, 10,20,35,
                            1, 5, 15,35,70};
    array double a1[5][5] = { -1, 1, 1, 1, 1,
                              1, 2, 3, 4, 5,
                              1, 3, 6,10,15,
                              1, 4, 10,20,35,
```

```

                                1, 5, 15,35,70});
int status;
array double l[m][m];

status = choldecomp(a, l);
if (status == 0) {
    printf("upper triangle l =\n%f\n", l);
    printf("transpose(l)*l - a =\n%f\n", transpose(l)*l-a);
}
else
    printf("error: Matrix must be positive definite.\n"
           "The %d order of matrix is not positive definite.\n",status);

status = choldecomp(a, l, 'l');
if (status == 0) {
    printf("lower triangle l =\n%f\n", l);
    printf("l*transpose(l) - a =\n%f\n", l*transpose(l)-a);
}
else
    printf("error: Matrix must be positive definite.\n"
           "The %d order of matrix is not positive definite.\n",status);

status = choldecomp(a1, l);
if (status == 0) {
    printf("l =\n%f\n", l);
}
else {
    printf("a =\n%f", a1);
    printf("error: Matrix must be positive definite.\n"
           "The %d order of matrix is not positive definite.\n",status);
}
}

```

Output1

```

upper triangle l =
1.000000 1.000000 1.000000 1.000000 1.000000
0.000000 1.000000 2.000000 3.000000 4.000000
0.000000 0.000000 1.000000 3.000000 6.000000
0.000000 0.000000 0.000000 1.000000 4.000000
0.000000 0.000000 0.000000 0.000000 1.000000

transpose(l)*l - a =
0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000 0.000000

lower triangle l =
1.000000 0.000000 0.000000 0.000000 0.000000
1.000000 1.000000 0.000000 0.000000 0.000000
1.000000 2.000000 1.000000 0.000000 0.000000
1.000000 3.000000 3.000000 1.000000 0.000000
1.000000 4.000000 6.000000 4.000000 1.000000

l*transpose(l) - a =
0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000 0.000000

```

```
0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000 0.000000
```

```
a =
-1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 2.000000 3.000000 4.000000 5.000000
1.000000 3.000000 6.000000 10.000000 15.000000
1.000000 4.000000 10.000000 20.000000 35.000000
1.000000 5.000000 15.000000 35.000000 70.000000
error: Matrix must be positive definite.
The 1 order of matrix is not positive definite.
```

Example2

Cholesky factorization of a complex matrix.

```
#include <numeric.h>

int main() {
    int m = 3;
    array double complex a[3][3] =
        { complex(2,0), complex(0,-1), complex(0,0),
          complex(0,1), complex(2,0), complex(0,0),
          complex(0,0), complex(0,0), complex(3,0) };

    int status;
    array double complex l[m][m];

    status = choldecomp(a, l);
    if (status == 0) {
        printf("upper triangle l =\n%5.3f\n", l);
        printf("transpose(l)*l - a =\n%5.3f\n", conj(transpose(l))*l-a);
    }
    else
        printf("error: Matrix must be positive definite.\n"
              "The %d order of matrix is not positive definite.\n",status);

    status = choldecomp(a, l, 'l');
    if (status == 0) {
        printf("lower triangle l =\n%5.3f\n", l);
        printf("l*transpose(l) - a =\n%5.3f\n", l*conj(transpose(l))-a);
    }
    else
        printf("error: Matrix must be positive definite.\n"
              "The %d order of matrix is not positive definite.\n",status);
}
```

Output2

```
upper triangle l =
complex(1.414,0.000) complex(0.000,-0.707) complex(0.000,0.000)
complex(0.000,0.000) complex(1.225,0.000) complex(0.000,0.000)
complex(0.000,0.000) complex(0.000,0.000) complex(1.732,0.000)

transpose(l)*l - a =
complex(0.000,0.000) complex(0.000,0.000) complex(0.000,0.000)
complex(0.000,0.000) complex(-0.000,0.000) complex(0.000,0.000)
complex(0.000,0.000) complex(0.000,0.000) complex(-0.000,0.000)
```

```
lower triangle l =
complex(1.414,0.000) complex(0.000,0.000) complex(0.000,0.000)
complex(0.000,0.707) complex(1.225,0.000) complex(0.000,0.000)
complex(0.000,0.000) complex(0.000,0.000) complex(1.732,0.000)

l*transpose(l) - a =
complex(0.000,0.000) complex(0.000,0.000) complex(0.000,0.000)
complex(0.000,0.000) complex(-0.000,0.000) complex(0.000,0.000)
complex(0.000,0.000) complex(0.000,0.000) complex(-0.000,0.000)
```

References

E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

cinverse

Synopsis

```
#include <numeric.h>
```

```
array double complex cinverse(array double complex a[:][:], ... /* [int *status] */)[:][:];
```

Syntax

```
cinverse(a)
```

```
cinverse(a, status)
```

Purpose

Calculate the inverse of a complex square matrix.

Return Value

This function returns the inverse matrix.

Parameters

a Input square matrix of double complex type.

status Output integer indicating the status of calculation.

Description

This function calculates the inverse matrix of a complex square matrix. If calculation is successful, *status* = 0, otherwise *status* ≠ 0.

Example

```
#include <numeric.h>
int main() {
    array double complex a[2][2] = {complex(2,-3), 4,
                                    3, 7}; // n-by-n matrix

    array double complex inv[2][2];
    int status;

    inv = cinverse(a);
    printf("cinverse(a) =\n%f\n", inv);
    printf("a =\n%f\n", a);
    printf("cinverse(a)*a =\n%f\n", inv*a);
    printf("a*cinverse(a) =\n%f\n", a*inv);

    inv = cinverse(a, &status);
    if(status == 0)
        printf("cinverse(a, &status) = %f\n", inv);
    else
        printf("error: numerical error in cinverse()\n");
}
```

Output

```
cinverse(a) =
complex(0.031461,0.330337) complex(-0.017978,-0.188764)
complex(-0.013483,-0.141573) complex(0.150562,0.080899)
```

```
a =
complex(2.000000,-3.000000) complex(4.000000,0.000000)
complex(3.000000,0.000000) complex(7.000000,0.000000)

cinverse(a)*a =
complex(1.000000,-0.000000) complex(0.000000,0.000000)
complex(-0.000000,-0.000000) complex(1.000000,0.000000)

a*cinverse(a) =
complex(1.000000,-0.000000) complex(0.000000,0.000000)
complex(0.000000,-0.000000) complex(1.000000,0.000000)

cinverse(a, &status) = complex(0.031461,0.330337) complex(-0.017978,-0.188764)
complex(-0.013483,-0.141573) complex(0.150562,0.080899)
```

See Also

inverse(), **ludcomp()**, **clinsolve()**.

References

E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

clinsolve

Synopsis

```
#include <numeric.h>
```

```
int clinsolve(array double complex x[], array double complex a[:][:], array double complex b[:]);
```

Purpose

Solve the linear system of equations by LU decomposition.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x Output array which contains the solution of equations.

a Input two-dimensional array contains the coefficients of a linear system of equations.

b Input one-dimensional array for linear equations.

Description

This function solves the system of complex linear equations by LU decomposition. The input matrix *a* should be n-by-n. The function can handle the equation set with complex numbers. The function returns 0 if there is a solution for the equation set, otherwise it returns -1.

Algorithm

This function uses the subroutines **zgesv()** from LAPACK. It calculates the solution to a real/complex system of linear equations $a * X = b$, where *a* is an n-by-n matrix. The LU decomposition with partial pivoting and row interchanges is used to factor *a* as $a = P * L * U$, where *P* is a permutation matrix, *L* is unit lower triangular, and *U* is upper triangular. The factored form of *a* is then used to solve the system of equation $a * X = b$.

Example

Solution of a linear system of equations with complex coefficients.

```
#include <numeric.h>
int main() {
    array double complex a[3][3] = {complex(3,4), 0, 6,
                                     0, 2, 1,
                                     1, 0, 1};
    array double complex b[3] = {2,
                                 13,
                                 25};

    array double complex x[3];
    int status;

    clinsolve(x, a, b);
    printf("linsolve(x, a,b) =\n%f\n", x);

    status = clinsolve(x, a, b);
    if(status == 0)
        printf("clinsolve(x,a,b) =\n%f\n", x);
    else
```



```
        printf("error: numerical error in clinsolve()\n");  
    }
```

Output

```
linsolve(x, a,b) =  
complex(17.760000,23.680000) complex(2.880000,11.840000) complex(7.240000,-23.680000)
```

```
clinsolve(x, a,b) =  
complex(17.760000,23.680000) complex(2.880000,11.840000) complex(7.240000,-23.680000)
```

See Also

llsqsolve(), **linsolve()**, **inverse()**.

References

E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

cmean

Synopsis

```
#include <numeric.h>
```

```
double complex cmean(array double complex &a, ... /*[array double complex v[&]] */);
```

Syntax

```
cmean(a)
```

```
cmean(a, v)
```

Purpose

Calculate the mean value of all elements of an array of complex type and mean values of the elements of each row of a two-dimensional array of complex type.

Return Value

This function returns the mean value of all the elements of complex type.

Parameters

a An input array of any dimension.

v An output array which contains the mean values of each row of a two-dimensional array of complex type. *v* shall be an array of double complex type.

Description

This function calculates the complex mean value of all elements in an array of any dimension. If the array is a two-dimensional matrix, the function can calculate complex mean values of each row. The complex mean values of each row are passed out by argument *v*.

Example

Calculate the mean values of all elements of arrays of complex type with different dimensions. The mean values of each row are passed by argument *v*.

```
#include <numeric.h>
#define N 2
#define M 3

int main() {
    complex a[3] = {1, 2, complex(3.435, 6.35765)};
    double complex c[2][3][5];
    c[0][0][0] = complex(53.327, 92.5310);
    c[0][0][1] = 20;
    complex a1[N][M] = {1, complex(3.54, 54.43), 3,
                       4, 5, complex(3.455, 466)};
    array double complex b1[3][4] = {1, 2, 3, complex(4, 64.54),
                                     complex(5, 6.45), 6, 7, 8,
                                     1, complex(4.35, 54.43), 3, 4};
    array double complex meana1[N], meanb1[3];
    double complex meanval;

    meanval = cmean(a);
    printf("mean(a) = %f\n", meanval);
```

```
meanval = cmean(c);
printf("mean(c) = %f\n", meanval);

/* Note: second argument of cmean() must be double complex data type */

meanval = cmean(a1, meana1);
printf("meanval = mean(a1, meana1) = %f\n", meanval);
printf("mean(a1, meana1) = %f\n", meana1);

meanval = cmean(b1, meanb1);
printf("meanval = mean(b1, meanb1) = %f\n", meanval);
printf("mean(b1, meanb1) = %0.2f\n", meanb1);
}
```

Output

```
mean(a) = complex(2.145000,2.119217)
mean(c) = complex(2.444233,3.084367)
meanval = mean(a1, meana1) = complex(3.332500,86.738333)
mean(a1, meana1) = complex(2.513333,18.143333) complex(4.151667,155.333333)

meanval = mean(b1, meanb1) = complex(4.029167,10.451667)
mean(b1, meanb1) = complex(2.50,16.14) complex(6.50,1.61) complex(3.09,13.61)
```

See Also

mean().

combination

Synopsis

```
#include <numeric.h>
```

```
unsigned long long combination(unsigned int n, unsigned int k);
```

Purpose

Calculate the number of combinations of n different things taken k at a time without repetitions.

Return Value

This function returns the number of combinations.

Parameters

n The number of different things.

k The number of items taken from n different things.

Description

This function calculates the number of combination of n different things taken k at a time without repetitions. The number of combination is the number of sets that can be made up from n things, each set containing k different things and no two sets containing exactly the same k things.

Algorithm

The number of combination of n different things taken k at a time without repetitions is defined as

$$C_k^n = \frac{n!}{(n-k)!k!}$$

Example

```
#include <numeric.h>
int main() {
    unsigned long long f;

    f = combination(3, 2);
    printf("combination(3, 2) = %llu\n", combination(3, 2));
}
```

Output

```
combination(3, 2) = 3
```

See Also

factorial().

References

companionmatrix

Synopsis

```
#include <numeric.h>
```

```
array double companionmatrix(array double v[&])[:][:];
```

Purpose

Find a companion matrix.

Return Value

This function returns a companion matrix.

Parameters

v Input array containing the coefficients of a polynomial.

Description

This function returns the corresponding companion matrix of array *v* which contains the coefficients of a polynomial. The eigenvalues of companion matrix are roots of the polynomial.

Example

```
#include <numeric.h>
int main() {
    array double v[4] = {2,3,4,5}; /* 2x^3 + 3x^2 + 4x + 5 */
    int n = 4;
    array double a[n-1][n-1];

    a = companionmatrix(v);
    printf("companionmatrix(a) =\n%f\n", a);
}
```

Output

```
companionmatrix(a) =
-1.500000 -2.000000 -2.500000
1.000000 0.000000 0.000000
0.000000 1.000000 0.000000
```

See Also

[ccompanionmatrix\(\)](#), [eigensystem\(\)](#), [polycoef\(\)](#).

References

complexsolve**Synopsis****#include** <numeric.h>**int** complexsolve(**int** *n1*, **int** *n2*, **double** *phi_or_r1*, **double** *phi_or_r2*, **double** **complex** *z3*,
double &*xx1*, **double** &*xx2*, **double** &*xx3*, **double** &*xx4*);**Purpose**

Solve a complex equation in the polar format.

Return Value

This function returns the number of solutions with 0, 1, or 2.

Parameters*n1* The first position of the two unknowns on the left hand side of equation(1). It's value can be 1, 2, 3, or 4.*n2* The second position of the two unknowns on the left hand side of equation(1). It's value can be 1, 2, 3, or 4.*phi_or_r1* The value of the first known on the left hand side.*phi_or_r2* The value of the second known on the left hand side.*z3* Complex number on the right hand side.*xx1* First unknown on the left hand side.*xx2* Second unknown on the left hand side.*xx3* First unknown on the left hand side for the second solution, if there are more than one set of solution.*xx4* Second unknown on the left hand side for the second solution, if there are more than one set of solution.**Description**

This function is used to solve a complex equation in polar format. The equation is in the form of

$$R_1 e^{i\phi_1} + R_2 e^{i\phi_2} = x_3 + iy_3 \quad (11.1)$$

Because it is a complex equation that can be partitioned into real and imaginary parts, two unknowns out of four parameters R_1 , ϕ_1 , R_2 , and ϕ_2 can be solved in this equation. The parameters R_1 , ϕ_1 , R_2 , and ϕ_2 are in positions 1, 2, 3, and 4, respectively.**Algorithm**

Two unknowns can be solved by decomposing equation (11.1) into real and imaginary parts

$$R_1 \cos \phi_1 + R_2 \cos \phi_2 = x_3 \quad (11.2)$$

$$R_1 \sin \phi_1 + R_2 \sin \phi_2 = y_3 \quad (11.3)$$

Case 1: $n_1 = 1, n_2 = 2$, Solve for R_1 and ϕ_1 , given R_2, ϕ_2, R_3, ϕ_3 or R_2, ϕ_2, x_3, y_3 . Where, n_1 and n_2 are the first and second positions of two unknowns on the left hand side of equation (11.1), respectively.

From equations (11.2) and equations (11.3), we get

$$R_1 \cos \phi_1 = x_3 - R_2 \cos \phi_2 = a \quad (11.4)$$

$$R_1 \sin \phi_1 = y_3 - R_2 \sin \phi_2 = b \quad (11.5)$$

R_1 and ϕ_1 can be calculated as

$$R_1 = \sqrt{a^2 + b^2} \quad (11.6)$$

$$\phi_1 = \text{atan2}(b, a) \quad (11.7)$$

Case 2: $n_1 = 1, n_2 = 3$, Given ϕ_1 and ϕ_2 , R_1 and R_2 are solved.

Multiplying equation (11.1) by $e^{-i\phi_2}$ and $e^{-i\phi_1}$ gives

$$R_1 e^{i(\phi_1 - \phi_2)} + R_2 = R_3 e^{i(\phi_3 - \phi_2)} \quad (11.8)$$

$$R_1 + R_2 e^{i(\phi_2 - \phi_1)} = R_3 e^{i(\phi_3 - \phi_1)} \quad (11.9)$$

Imaginary parts of equations (11.8) and equation (11.9) are

$$R_1 \sin(\phi_1 - \phi_2) = R_3 \sin(\phi_3 - \phi_2) \quad (11.10)$$

$$R_2 \sin(\phi_2 - \phi_1) = R_3 \sin(\phi_3 - \phi_1) \quad (11.11)$$

From equation (11.10) and equation (11.11) we get

$$R_1 = R_3 \frac{\sin(\phi_3 - \phi_2)}{\sin(\phi_1 - \phi_2)} \quad (11.12)$$

$$R_2 = R_3 \frac{\sin(\phi_3 - \phi_1)}{\sin(\phi_2 - \phi_1)} \quad (11.13)$$

Case 3: $n_1 = 1, n_2 = 4$, Given ϕ_1 and R_2 , R_1 and ϕ_2 are solved.

From equation (11.2) we have

$$R_1 = \frac{x_3 - R_2 \cos \phi_2}{\cos \phi_1} \quad (11.14)$$

substitute equation (11.14) into equation (11.3) we get

$$(x_3 - R_2 \cos \phi_2) \sin \phi_1 + R_2 \sin \phi_2 \cos \phi_1 = y_3 \cos \phi_1 \quad (11.15)$$

equation (11.15) can be simplified as

$$\sin(\phi_2 - \phi_1) = \frac{y_3 \cos \phi_1 - x_3 \sin \phi_1}{R_2} = a \quad (11.16)$$

then

$$\phi_2 = \phi_1 + \sin^{-1}(a) \phi_2 = \phi_1 + \pi - \sin^{-1}(a) \quad (11.17)$$

If $\cos \phi_1$ is larger than machine epsilon, ε , R_1 can be obtained using equation (11.14). Otherwise R_1 will be obtained using equation (11.3)

if $(|\cos \phi_1| > \text{FLT_EPSILON})$

$$R_1 = \frac{(x_3 - R_2 \cos \phi_2)}{\cos \phi_1} \quad (11.18)$$

else

$$R_1 = \frac{(y_3 - R_2 \sin \phi_2)}{\sin \phi_1} \quad (11.19)$$

Case 4: $n_1 = 2, n_2 = 4$, given R_1 and R_2 , ϕ_1 and ϕ_2 in equation (11.1) can be solved.

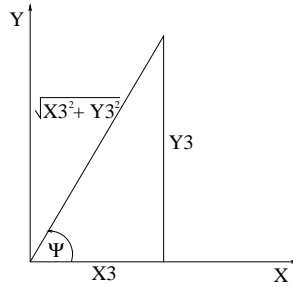
From equation (11.1), we get

$$\cos \phi_1 = \frac{x_3 - R_2 \cos \phi_2}{R_1}, \quad \sin \phi_1 = \frac{y_3 - R_2 \sin \phi_2}{R_1} \quad (11.20)$$

Substituting these results into the identity equation $\sin^2 \phi_1 + \cos^2 \phi_1 = 1$ and simplifying the resultant equation, we get

$$y_3 \sin \phi_2 + x_3 \cos \phi_2 = \frac{x_3^2 + y_3^2 + R_2^2 - R_1^2}{2R_2} \quad (11.21)$$

from equation (11.21), we can derive the formulas for ϕ_1 and ϕ_2 as follows:



$$\tan(\psi) = \frac{y_3}{x_3}, \quad \cos(\psi) = \frac{x_3}{\sqrt{x_3^2 + y_3^2}}, \quad \sin(\psi) = \frac{y_3}{\sqrt{x_3^2 + y_3^2}}$$

Equation (11.21) becomes,

$$\frac{y_3}{\sqrt{x_3^2 + y_3^2}} \sin(\phi_2) + \frac{x_3}{\sqrt{x_3^2 + y_3^2}} \cos(\phi_2) = \frac{x_3^2 + y_3^2 + R_2^2 - R_1^2}{2R_2 \sqrt{x_3^2 + y_3^2}} \quad (11.22)$$

$$a = \sin(\psi) \sin(\phi_2) + \cos(\psi) \cos(\phi_2)$$

Let,

$$a = \frac{x_3^2 + y_3^2 + R_2^2 - R_1^2}{2R_2 \sqrt{x_3^2 + y_3^2}} \quad (11.23)$$

Becomes,

$$\cos(\phi_2 - \psi) = a \quad (11.24)$$

$$\phi_2 = \psi \pm \cos^{-1}(a) \quad (11.25)$$

$$= \operatorname{atan2}(y_3, x_3) \pm \operatorname{acos} \left(\frac{x_3^2 + y_3^2 + R_2^2 - R_1^2}{2R_2\sqrt{x_3^2 + y_3^2}} \right) \quad (11.26)$$

ϕ_1 can be obtained using equation (11.20)

$$\phi_1 = \operatorname{atan2}(\sin \phi_1, \cos \phi_1) \quad (11.27)$$

or use identity equation,

$$\tan\left(\frac{\phi}{2}\right) = \frac{1 - \cos(\phi)}{\sin(\phi)} \text{ or } \tan\left(\frac{\phi}{2}\right) = \frac{\sin(\phi)}{1 + \cos(\phi)} \quad (11.28)$$

$$\phi = 2 \tan^{-1}\left(\frac{1 - \cos(\phi)}{\sin(\phi)}\right) \text{ or } \phi = 2 \tan^{-1}\left(\frac{\sin(\phi)}{1 + \cos(\phi)}\right) \quad (11.29)$$

Case 5: $n_1 = 2, n_2 = 3$, given R_1 and ϕ_2, ϕ_1 and R_2 are solved. This case is similar to Case 3.

Case 6: $n_1 = 3, n_2 = 4$, given R_1 and ϕ_1, R_2 and ϕ_2 are solved. This case is similar to case 1.

Example 1

A complex equation

$$R_1 e^{i\phi_1} + R_2 e^{i\phi_2} = 1 + i2$$

is solved for the 6 cases described above.

```
#include <numeric.h>
#include <complex.h>

int main() {
    double x1, x2, x3, x4;
    int n1, n2, num;
    double phi_or_r1, phi_or_r2;
    complex z3, error;

    phi_or_r1 = 3;
    phi_or_r2 = 4;
    z3 = complex(1,2);

    /***** test n1 = 1, n2 = 2 *****/
    n1 = 1,
    n2 = 2;
    num = complexsolve(n1, n2, phi_or_r1, phi_or_r2, z3, x1, x2, x3, x4);
    error = polar(x1, x2) + polar(phi_or_r1, phi_or_r2) - z3;
    printf("For n1=1, n2=2, num = %d\n", num);
    printf("For n1=1, n2=2, the output x is %f %f %f %f \n", x1, x2, x3, x4);
    printf("the residual error is %f \n", error);
```

```

/***** test n1 = 1, n2 = 3 *****/
n1 = 1,
n2 = 3;
num = complexsolve(n1, n2, phi_or_r1, phi_or_r2, z3, x1, x2, x3, x4);
printf("For n1=1, n2=3, num = %d\n", num);
printf("For n1=1, n2=3, the output x is %f %f %f %f \n", x1, x2, x3, x4);

/***** test n1 = 1, n2 = 4 *****/
n1 = 1,
n2 = 4;
num = complexsolve(n1, n2, phi_or_r1, phi_or_r2, z3, x1, x2, x3, x4);
printf("For n1=1, n2=4, num = %d\n", num);
printf("For n1=1, n2=4, the output x is %f %f %f %f \n", x1, x2, x3, x4);

/***** test n1 = 2, n2 = 3 *****/
n1 = 2,
n2 = 3;
num = complexsolve(n1, n2, phi_or_r1, phi_or_r2, z3, x1, x2, x3, x4);
printf("For n1=2, n2=3, num = %d\n", num);
printf("For n1=2, n2=3, the output x is %f %f %f %f \n", x1, x2, x3, x4);

/***** test n1 = 2, n2 = 4 *****/
n1 = 2,
n2 = 4;
num = complexsolve(n1, n2, phi_or_r1, phi_or_r2, z3, x1, x2, x3, x4);
printf("For n1=2, n2=4, num = %d\n", num);
printf("For n1=2, n2=4, the output x is %f %f %f %f \n", x1, x2, x3, x4);

/***** test n1 = 3, n2 = 4 *****/
n1 = 3,
n2 = 4;
num = complexsolve(n1, n2, phi_or_r1, phi_or_r2, z3, x1, x2, x3, x4);
printf("For n1=3, n2=4, num = %d\n", num);
printf("For n1=3, n2=4, the output x is %f %f %f %f \n", x1, x2, x3, x4);
}

```

Output

```

For n1=1, n2=2, num = 1
For n1=1, n2=2, the output x is 5.196488 0.964540 NaN NaN
the residual error is complex(0.000000,0.000000)
For n1=1, n2=3, num = 1
For n1=1, n2=3, the output x is 0.654193 -2.520711 NaN NaN
For n1=1, n2=4, num = 2
For n1=1, n2=4, the output x is -4.099052 2.441074 2.683547 0.417334
For n1=2, n2=3, num = 2
For n1=2, n2=3, the output x is -2.467726 -5.116311 1.042948 0.781813
For n1=2, n2=4, num = 2
For n1=2, n2=4, the output x is -0.613277 1.942631 2.827574 0.271667
For n1=3, n2=4, num = 1
For n1=3, n2=4, the output x is 5.196488 0.964540 NaN NaN

```

Example 2

For a complex equation

$$R_1 e^{i\phi_1} + R_2 e^{i\phi_2} = 1 + i2$$

- (1) Given $R_1 = 3$ and $\phi_1 = 4$ radian, find R_2 and ϕ_2
- (2) Given $R_1 = 3$ and $R_2 = 4$, find ϕ_1 and ϕ_2

```

#include <numeric.h>
#include <complex.h>

int main () {
    double x1, x2, x3, x4;
    int n1, n2, num;
    double phi_or_r1, phi_or_r2;
    complex z3, error;

    /* problem (1) n1 = 3, n2 = 4 */
    phi_or_r1 = 3;
    phi_or_r2 = 4;
    z3 = complex(1,2);
    n1 = 3,
    n2 = 4;
    num = complexsolve(n1, n2, phi_or_r1, phi_or_r2, z3, x1, x2, x3, x4);
    printf("For n1=3, n2=4, the number of solution is = %d\n", num);
    printf("R2 = %f phi2 = %f\n", x1, x2);
    printf("R2 = %f phi2 = %f\n", x3, x4);

    /* problem (2), n1 = 2, n2 = 4 */
    n1 = 2,
    n2 = 4;
    num = complexsolve(n1, n2, phi_or_r1, phi_or_r2, z3, x1, x2, x3, x4);
    printf("For n1=2, n2=4, the number of solution is = %d\n", num);
    printf("phi1 = %f phi2 = %f\n", x1, x2);
    printf("phi1 = %f phi2 = %f\n", x3, x4);
}

```

Output

```

For n1=3, n2=4, the number of solution is = 1
R2 = 5.196488 phi2 = 0.964540
R2 = NaN phi2 = NaN
For n1=2, n2=4, the number of solution is = 2
phi1 = -0.613277 phi2 = 1.942631
phi1 = 2.827574 phi2 = 0.271667

```

See Also

complexsolvePP(), complexsolvePR(), complexsolveRP(), complexsolveRR(), complexsolveRRz(), polar().

References

complexsolvePP**Synopsis****#include** <numeric.h>**int** complexsolvePP(double *phi1*, double *phi2*, double complex *z3*,
double &*r1*, double &*r2*);**Purpose**Solve for r_1 and r_2 in a complex equation in the polar format.**Return Value**

This function returns the number of solution with the value 1.

Parameters*phi1* The known value on the left hand side.*phi2* The known value on the left hand side.*z3* Complex number on the right hand side.*r1* r_1 for the solution on the left hand site.*r2* r_2 for the solution on the left hand site.**Description**This function is used to solve for r_1 and r_2 in a complex equation in the polar format. The equation is in the form of

$$R_1 e^{i\phi_1} + R_2 e^{i\phi_2} = x_3 + iy_3 \quad (11.30)$$

Because it is a complex equation that can be partitioned into real and imaginary parts, two unknowns R_1 and ϕ_2 can be solved in this equation.**Algorithm**Multiplying equation (11.30) by $e^{-i\phi_2}$ and $e^{-i\phi_1}$ gives

$$R_1 e^{i(\phi_1 - \phi_2)} + R_2 = R_3 e^{i(\phi_3 - \phi_2)} \quad (11.31)$$

$$R_1 + R_2 e^{i(\phi_2 - \phi_1)} = R_3 e^{i(\phi_3 - \phi_1)} \quad (11.32)$$

Imaginary parts of equations (11.31) and equation (11.32) are

$$R_1 \sin(\phi_1 - \phi_2) = R_3 \sin(\phi_3 - \phi_2) \quad (11.33)$$

$$R_2 \sin(\phi_2 - \phi_1) = R_3 \sin(\phi_3 - \phi_1) \quad (11.34)$$

From equation (11.33) and equation (11.34) we get

$$R_1 = R_3 \frac{\sin(\phi_3 - \phi_2)}{\sin(\phi_1 - \phi_2)} \quad (11.35)$$

$$R_2 = R_3 \frac{\sin(\phi_3 - \phi_1)}{\sin(\phi_2 - \phi_1)} \quad (11.36)$$

Note

The function call

```
complexsolvePP(phi1, phi2, z3, r1, r2);
```

is equivalent to

```
complexsolve(1, 3, phi1, phi2, z3, r1, r2, 0, 0);
```

Example

For a complex equation

$$R_1 e^{i\phi_1} + R_2 e^{i\phi_2} = 1 + i2$$

Given $\phi_1 = 3$ radian and $\phi_2 = 4$ radian, find R_1 and ϕ_2 .

```
/* solve for r1 and r2 in
   polar(r1, phi1) + polar(r2, phi2) = z3
   with phi1 = 3, phi2 = 4, z3 = 1+i2.
   The output of this program is as follows:
   r1 = 0.6542 (m), r2 = -2.5207 (m)
*/
#include <stdio.h>
#include <numeric.h>

int main () {
    double phi1, phi2;
    double complex z3;
    double r1, r2;

    phi1 = 3;
    phi2 = 4;
    z3 = complex(1, 2);
    complexsolvePP(phi1, phi2, z3, r1, r2);
    printf("r1 = %7.4f (m), r2 = %7.4f (m)\n", r1, r2);
    return 0;
}
```

See Also

`complexsolve()`, `complexsolvePR()`, `complexsolveRP()`, `complexsolveRR()`, `complexsolveRRz()`, `polar()`.

References

complexsolvePR

Synopsis

```
#include <numeric.h>
```

```
int complexsolvePR(double phi1, double r2, double complex z3,
                  double &r1, double &phi2, double &r1_2, double &phi2_2);
```

Purpose

Solve for r_1 and ϕ_2 in a complex equation in the polar format.

Return Value

This function returns the number of solutions with the value 2.

Parameters

phi1 The known value on the left hand side.

r2 The known value on the left hand side.

z3 Complex number on the right hand side.

r1 r_1 for the first solution on the left hand side.

phi2 ϕ_2 for the first solution on the left hand side.

r1_2 r_1 for the second solution on the left hand side.

phi2_2 ϕ_2 for the second solution on the left hand side.

Description

This function is used to solve for r_1 and ϕ_2 in a complex equation in the polar format. The equation is in the form of

$$R_1 e^{i\phi_1} + R_2 e^{i\phi_2} = x_3 + iy_3 \quad (11.37)$$

Because it is a complex equation that can be partitioned into real and imaginary parts, two unknowns R_1 and ϕ_2 can be solved in this equation.

Algorithm

Two unknowns can be solved by decomposing equation (11.37) into real and imaginary parts

$$R_1 \cos \phi_1 + R_2 \cos \phi_2 = x_3 \quad (11.38)$$

$$R_1 \sin \phi_1 + R_2 \sin \phi_2 = y_3 \quad (11.39)$$

From equation (11.38) we have

$$R_1 = \frac{x_3 - R_2 \cos \phi_2}{\cos \phi_1} \quad (11.40)$$

substitute equation (11.40) into equation (11.39), we get

$$(x_3 - R_2 \cos \phi_2) \sin \phi_1 + R_2 \sin \phi_2 \cos \phi_1 = y_3 \cos \phi_1 \quad (11.41)$$

equation (11.41) can be simplified as

$$\sin(\phi_2 - \phi_1) = \frac{y_3 \cos \phi_1 - x_3 \sin \phi_1}{R_2} = a \quad (11.42)$$

then

$$\phi_2 = \phi_1 + \sin^{-1}(a) \quad (11.43)$$

$$\phi_2 = \phi_1 + \pi - \sin^{-1}(a) \quad (11.44)$$

If $\cos \phi_1$ is larger than machine epsilon, ε , R_1 can be obtained using equation (11.40). Otherwise R_1 will be obtained using equation (11.39)

if $(|\cos \phi_1| > \text{FLT_EPSILON})$

$$R_1 = \frac{x_3 - R_2 \cos \phi_2}{\cos \phi_1} \quad (11.45)$$

else

$$R_1 = \frac{y_3 - R_2 \sin \phi_2}{\sin \phi_1} \quad (11.46)$$

Note

The function call

```
complexsolvePR(phi1, r2, z3, r1, phi2, r1_2, phi2_2);
```

is equivalent to

```
complexsolve(1, 4, phi1, r2, z3, r1, phi2, r1_2, phi2_2);
```

Example

For a complex equation

$$R_1 e^{i\phi_1} + R_2 e^{i\phi_2} = 1 + i2$$

Given $\phi_1 = 3$ radian and $r_2 = 4$ meters, find R_1 and ϕ_2 .

```
/* solve for r1 and phi2 in
   polar(r1, phi1) + polar(r2, phi2) = z3
   or
   polar(r2, phi2) + polar(r1, phi1) = z3
   with phi1 = 3, r2 = 4, z3 = 1+i2.
   The output of this program is as follows:
   r1 = -4.0991 (m), phi2 = 2.4411 (radian)
   r1 = 2.6835 (m), phi2 = 0.4173 (radian)
*/
#include <stdio.h>
#include <numeric.h>

int main () {
    double phi1, r2;
    double complex z3;
    double r1, phi2, r1_2, phi2_2;

    phi1 = 3;
    r2 = 4;
    z3 = complex(1, 2);
    complexsolvePR(phi1, r2, z3, r1, phi2, r1_2, phi2_2);
    printf("r1 = %7.4f (m), phi2 = %7.4f (radian)\n", r1, phi2);
    printf("r1 = %7.4f (m), phi2 = %7.4f (radian)\n", r1_2, phi2_2);
    return 0;
}
```

See Also

complexsolve(), **complexsolvePP()**, **complexsolveRP()**, **complexsolveRR()**, **complexsolveRRz()**, **polar()**.

References

complexsolveRP**Synopsis****#include** <numeric.h>**int** complexsolveRP(double *a*, double complex *z*,
double &*r*, double &*theta*, double &*r_2*, double &*theta_2*);**Purpose**Solve for r and θ in a complex equation in the polar format.**Return Value**

This function returns the number of solutions with the value 2 if successful. Otherwise, it returns -1.

Parameters*a* The known value on the left hand side.*z* The complex number on the right hand side.*r* r for the first solution on the left hand side.*theta* θ for the first solution on the left hand side.*r_2* r for the second solution on the left hand side.*theta_2* θ for the second solution on the left hand side.**Description**This function is used to solve for r and θ in a complex equation in the polar format. The equation is in the form of

$$(a + ir)e^{i\theta} = x + iy \quad (11.47)$$

Because it is a complex equation that can be partitioned into real and imaginary parts, two unknowns r and θ can be solved in this equation.**Algorithm**

Two unknowns can be solved by decomposing equation (11.47) into real and imaginary parts

$$a \cos \theta - r \sin \theta = x \quad (11.48)$$

$$a \sin \theta + r \cos \theta = y \quad (11.49)$$

From equations (11.48) and (11.49), we have

$$(a \cos \theta - r \sin \theta)^2 + (a \sin \theta + r \cos \theta)^2 = x^2 + y^2 \quad (11.50)$$

which can be simplified to

$$a^2 + r^2 = x^2 + y^2 \quad (11.51)$$

From equation (11.50), we can find two solutions for r as follows:

$$r = \pm \sqrt{x^2 + y^2 - a^2} \quad (11.52)$$

From equations (11.48) and (11.49), we can derive the formulas for $\sin \theta$ and $\cos \theta$ as follows:

$$\sin \theta = \frac{ay - rx}{r^2 + a^2} \quad (11.53)$$

$$\cos \theta = \frac{ry + ax}{r^2 + a^2} \quad (11.54)$$

For each r , we can use the function **atan2()** to find θ .

Example

For a complex equation

$$(a + ir)e^{i\theta} = 1 + i2$$

Given $a = 3$ meters and $r_2 = 4$ meters, find r and θ .

```

/* solve for r and theta in
   (a+i r)e^(i theta) = z
   with a = 3 and z = 5+i3.
   The output of this program is as follows:
   r = 5.0000 (m), theta = -0.4900 (radian)
   r = -5.0000 (m), theta = 1.5708 (radian)
*/
#include <stdio.h>
#include <numeric.h>

int main () {
    double a;
    double complex z;
    double r, theta, r_2, theta_2;

    a = 3;
    z = complex(5, 3);
    complexsolveRP(a, z, r, theta, r_2, theta_2);
    printf("r = %7.4f (m), theta = %7.4f (radian)\n", r, theta);
    printf("r = %7.4f (m), theta = %7.4f (radian)\n", r_2, theta_2);
    return 0;
}

```

See Also

complexsolve(), **complexsolvePP()**, **complexsolvePR()**, **complexsolveRR()**, **complexsolveRRz()**, **polar()**.

References

complexsolveRR**Synopsis****#include** <numeric.h>**int** complexsolveRR(double *r1*, double *r2*, double complex *z3*,
double &*phi1*, double &*phi2*, double &*phi1_2*, double &*phi2_2*);**Purpose**Solve for ϕ_1 and ϕ_2 in a complex equation in the polar format.**Return Value**

This function returns the number of solutions with the value 2.

Parameters*r1* The known value on the left hand side.*r2* The known value on the left hand side.*z3* Complex number on the right hand side.*phi1* ϕ_1 for the first solution on the left hand side.*phi2* ϕ_2 for the first solution on the left hand side.*phi1_2* ϕ_1 for the second solution on the left hand side.*phi2_2* ϕ_2 for the second solution on the left hand side.**Description**This function is used to solve for ϕ_1 and ϕ_2 in a complex equation in polar format. The equation is in the form of

$$R_1 e^{i\phi_1} + R_2 e^{i\phi_2} = x_3 + iy_3 \quad (11.55)$$

Because it is a complex equation that can be partitioned into real and imaginary parts, two unknowns ϕ_1 and ϕ_2 can be solved in this equation.**Algorithm**

Two unknowns can be solved by decomposing equation (11.55) into real and imaginary parts

$$R_1 \cos \phi_1 + R_2 \cos \phi_2 = x_3 \quad (11.56)$$

$$R_1 \sin \phi_1 + R_2 \sin \phi_2 = y_3 \quad (11.57)$$

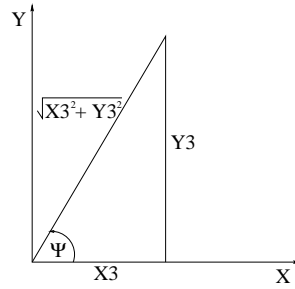
From equation (11.55), we get

$$\cos \phi_1 = \frac{x_3 - R_2 \cos \phi_2}{R_1}, \quad \sin \phi_1 = \frac{y_3 - R_2 \sin \phi_2}{R_1} \quad (11.58)$$

Substituting these results into the identity equation $\sin^2 \phi_1 + \cos^2 \phi_1 = 1$ and simplifying the resultant equation, we get

$$y_3 \sin \phi_2 + x_3 \cos \phi_2 = \frac{x_3^2 + y_3^2 + R_2^2 - R_1^2}{2R_2} \quad (11.59)$$

from equation (11.59), we can derive the formulas for ϕ_2 as follows:



$$\tan(\psi) = \frac{y_3}{x_3}, \cos(\psi) = \frac{x_3}{\sqrt{x_3^2 + y_3^2}}, \sin(\psi) = \frac{y_3}{\sqrt{x_3^2 + y_3^2}}$$

Equation (11.59) becomes,

$$\frac{y_3}{\sqrt{x_3^2 + y_3^2}} \sin(\phi_2) + \frac{x_3}{\sqrt{x_3^2 + y_3^2}} \cos(\phi_2) = \frac{x_3^2 + y_3^2 + R_2^2 - R_1^2}{2R_2\sqrt{x_3^2 + y_3^2}} \quad (11.60)$$

Let

$$a = \frac{x_3^2 + y_3^2 + R_2^2 - R_1^2}{2R_2\sqrt{x_3^2 + y_3^2}} \quad (11.61)$$

we get

$$\sin(\psi) \sin(\phi_2) + \cos(\psi) \cos(\phi_2) = a \quad (11.62)$$

which can be simplified as

$$\cos(\phi_2 - \psi) = a \quad (11.63)$$

$$\phi_2 = \psi \pm \cos^{-1}(a) \quad (11.64)$$

$$= \text{atan2}(y_3, x_3) \pm \text{acos} \left(\frac{x_3^2 + y_3^2 + R_2^2 - R_1^2}{2R_2\sqrt{x_3^2 + y_3^2}} \right) \quad (11.65)$$

ϕ_1 can be obtained using equation (11.58)

$$\phi_1 = \text{atan2}(\sin \phi_1, \cos \phi_1) \quad (11.66)$$

or use identity equation,

$$\tan\left(\frac{\phi}{2}\right) = \frac{1 - \cos(\phi)}{\sin(\phi)} \text{ or } \tan\left(\frac{\phi}{2}\right) = \frac{\sin(\phi)}{1 + \cos(\phi)} \quad (11.67)$$

$$\phi_1 = 2 \tan^{-1}\left(\frac{1 - \cos(\phi_1)}{\sin(\phi_1)}\right) \text{ or } \phi_1 = 2 \tan^{-1}\left(\frac{\sin(\phi_1)}{1 + \cos(\phi_1)}\right) \quad (11.68)$$

Note

The function call

```
complexsolveRR(r1, r2, z3, phi1, phi2, phi1_2, phi2_2);
```

is equivalent to

```
complexsolve(2, 4, r1, r2, z3, phi1, phi2, phi1_2, phi2_2);
```

Example

For a complex equation

$$R_1 e^{i\phi_1} + R_2 e^{i\phi_2} = 1 + i2$$

Given $R_1 = 3$ and $R_2 = 4$, find ϕ_1 and ϕ_2

```
/* solve for phi1 and phi2 in
   polar(r1, phi1) + polar(r2, phi2) = z3
   or
   polar(r2, phi2) + polar(r1, phi1) = z3
   with r1 = 3, r2 = 4, z3 = 1+i2.
   The output of this program is as follows:
   phi1 = -0.6133 (radian), phi2 = 1.9426 (radian)
   phi1 = 2.8276 (radian), phi2 = 0.2717 (radian)
*/
#include <stdio.h>
#include <numeric.h>

int main () {
    double r1, r2;
    double complex z3;
    double phi1, phi2, phi1_2, phi2_2;

    r1 = 3;
    r2 = 4;
    z3 = complex(1, 2);
    complexsolveRR(r1, r2, z3, phi1, phi2, phi1_2, phi2_2);
    printf("phi1 = %7.4f (radian), phi2 = %7.4f (radian)\n", phi1, phi2);
    printf("phi1 = %7.4f (radian), phi2 = %7.4f (radian)\n", phi1_2, phi2_2);
    return 0;
}
```

See Also

complexsolve(), **complexsolvePP()**, **complexsolvePR()**, **complexsolveRP()**, **complexsolveRRz()**, **polar()**.

References

complexsolveRRz**Synopsis****#include** <numeric.h>**int** complexsolveRRz(double complex *z1*, double complex *r2*, double complex *z3*,
double *&phi1*, double *&phi2*, double *&phi1_2*, double *&phi2_2*);**Purpose**Solve for ϕ_1 and ϕ_2 in a complex equation in the polar format.**Return Value**

This function returns the number of solutions with the value 2.

Parameters*z1* complex number on the left hand side.*z2* complex number on the left hand side.*z3* Complex number on the right hand side.*phi1* ϕ_1 for the first solution on the left hand side.*phi2* ϕ_2 for the first solution on the left hand side.*phi1_2* ϕ_1 for the second solution on the left hand side.*phi2_2* ϕ_2 for the second solution on the left hand side.**Description**This function is used to solve for ϕ_1 and ϕ_2 in a complex equation in polar format. The equation is in the form of

$$z_1 e^{i\phi_1} + z_2 e^{i\phi_2} = z_3 \quad (11.69)$$

Because it is a complex equation that can be partitioned into real and imaginary parts, two unknowns ϕ_1 and ϕ_2 can be solved in this equation.**Algorithm**

Two unknowns can be solved by using equation (11.70) and its conjugate equation (11.71)

$$z_2 e^{i\phi_2} = z_3 - z_1 e^{i\phi_1} \quad (11.70)$$

$$\bar{z}_2 e^{-i\phi_2} = \bar{z}_3 - \bar{z}_1 e^{-i\phi_1} \quad (11.71)$$

Multiplying equations (11.70) and (11.71), we get

$$z_2 \bar{z}_2 = z_3 \bar{z}_3 - z_1 \bar{z}_1 - z_3 \bar{z}_1 e^{-i\phi_1} - \bar{z}_3 z_1 e^{i\phi_1} \quad (11.72)$$

Multiplying equation (11.72) by $e^{i\phi_1}$, we get

$$\bar{z}_3 z_1 (e^{i\phi_1})^2 + (z_2 \bar{z}_2 - z_3 \bar{z}_3 - z_1 \bar{z}_1) e^{i\phi_1} + z_3 \bar{z}_1 = 0 \quad (11.73)$$

We can write equation (11.73) as

$$ap^2 + bp + c = 0 \quad (11.74)$$

with $a = \bar{z}_3 z_1$, $b = z_2 \bar{z}_2 - z_3 \bar{z}_3 - z_1 \bar{z}_1$, $c = z_3 \bar{z}_1$, and $p = e^{i\phi_1}$. We can solve for two solutions of p using the complex quadratic equation (11.74). With

$$\cos \phi_1 = \text{real}(p), \sin \phi_1 = \text{imag}(p), \quad (11.75)$$

we can find $\phi_1 = \text{atan2}(p_y, p_x)$.

For each ϕ_1 , we can solve for ϕ_2 by equation (11.76)

$$e^{i\phi_2} = q = (z_3 - z_1 e^{i\phi_1})/z_2 \quad (11.76)$$

we can find $\phi_2 = \text{atan2}(q_y, q_x)$.

Example

For a complex equation

$$z_1 e^{i\phi_1} + z_2 e^{i\phi_2} = z_3$$

Given $z_1 = 3$, $z_2 = 4$, and $z_3 = 1 + i2$, find ϕ_1 and ϕ_2

```

/* solve for phi1 and phi2 in
   z1 e^(i phi1) + z2 e^(i phi2) = z3
   with z1 = 3, z2 = 4, z3 = 1+i2.
   The output of this program is as follows:
   phi1 =  2.8276 (radian), phi2 =  0.2717 (radian)
   phi1 = -0.6133 (radian), phi2 =  1.9426 (radian)
*/
#include <stdio.h>
#include <numeric.h>

int main () {
    double complex z1, z2, z3;
    double phi1, phi2, phi1_2, phi2_2;

    z1 = complex(3, 0);
    z2 = complex(4, 0);
    z3 = complex(1, 2);
    complexsolveRRz(z1, z2, z3, phi1, phi2, phi1_2, phi2_2);
    printf("phi1 = %7.4f (radian), phi2 = %7.4f (radian)\n", phi1, phi2);
    printf("phi1 = %7.4f (radian), phi2 = %7.4f (radian)\n", phi1_2, phi2_2);
    return 0;
}

```

See Also

complexsolve(), **complexsolvePP()**, **complexsolvePR()**, **complexsolveRP()**, **complexsolveRR()**, **polar()**.

References

condnum

Synopsis

```
#include <numeric.h>
```

```
double condnum(array double complex a[&][&]);
```

Purpose

Calculate the condition number of a matrix.

Return Value

This function returns the condition number.

Parameters

a Input matrix.

Description

The condition number of a matrix measures the sensitivity of the solution of a system of linear equations to errors in the data. It gives an indication of the accuracy of the results from matrix inversion and numerical solution of the linear system of equations solution. A value of the condition number near 1 indicates a well-conditioned matrix.

Algorithm

The function `condnum()` uses the the singular value decomposition function `svd()`. By function `svd()`, the matrix *a* is decomposed as

$$\mathbf{a} = \mathbf{u}\mathbf{sv}^t$$

The condition number is defined as $\max(s)/\min(s)$.

Example

```
#include <numeric.h>
int main() {
    array double a[2][2] = {2, 4,
                           3, 7};
    /* a2 is an ill-conditioned matrix */
    array double a2[2][2] = {2, 4,
                             2.001, 4.0001};
    array float b[2][2] = {2, 4,
                           3, 7};

    array double complex z[2][2] = {2, complex(4, 3),
                                    3, 7};
    /* z2 is an ill-conditioned matrix */
    array double complex z2[2][2] = {2, complex(4, 3),
                                      2.001, complex(4.0001, 3)};
    array complex z3[2][2] = {2, complex(4, 3),
                              3, 7};

    double cond;

    cond = condnum(a);
    printf("condnum(a) = %g\n", cond);
}
```



```
cond = condnum(a2);
printf("condnum(a2) = %g\n", cond);

cond = condnum(b);
printf("condnum(b) = %g\n", cond);

cond = condnum(z);
printf("condnum(z) = %g\n", cond);
cond = condnum(z2);
printf("condnum(z2) = %g\n", cond);
cond = condnum(z3);
printf("condnum(z3) = %g\n", cond);
}
```

Output

```
condnum(a) = 38.9743
condnum(a2) = 10527.6
condnum(b) = 38.9743
condnum(z) = 9.32929
condnum(z2) = 11980.8
condnum(z3) = 9.32929
```

See Also

svd(), **norm()**, **rank()**, **rcondnum()**.

References

constrain

Synopsis

```
#include <numeric.h>
```

```
double constrain(double x, double a, double b);
```

Purpose

Constrain a number to be within a range.

Return Value

This function returns a value with the constrained range.

Parameters

x the number to be constrained.

a the lower end of the range.

b the upper end of the range.

Description

This function constrains a number to be within a range. The function returns $\min(\max(x, a), b)$. If *x* is between *a* and *b*, the function returns *x*. If *x* is less than *a*, the function returns *a*. If *x* is greater than *b*, the function returns *b*.

Example

```
#include <numeric.h>
double x, a = 3.5, b = 10.5, y;
x = 8;
y = constrain(x, a, b);
printf("y = %lf\n", y);
y = constrain(2.5, a, b);
printf("y = %lf\n", y);
y = constrain(12.5, a, b);
printf("y = %lf\n", y);
```

Output

```
y = 8.000000
y = 3.500000
y = 10.500000
```

See Also

`getline()`.

References

conv**Synopsis****#include** <numeric.h>**int conv(array double complex c[&], array double complex x[&], array double complex y[&]);****Syntax**conv(*c*, *x*, *y*)**Purpose**

One-dimensional Discrete Fourier Transform (DFT) based convolution.

Return Value

This function returns 0 on success and -1 on failure.

Parameters*c* A one-dimensional array of size $n + m - 1$. It contains the result of convolution of *x* and *y*.*x* A one-dimensional array of size n . It contains data used for convolution.*y* A one-dimensional array of size m . It contains data used for convolution.**Description**

The input **array** *x* and *y* can be of any supported arithmetic data type and sizes n and m , respectively. Conversion of the data to **double complex** type is performed internally. If both *x* and *y* are real type, the result is a one-dimensional array *c* of size $n + m - 1$. If either one of *x* and *y* is complex type, the result *c* is complex type.

If *x* and *y* are considered as two vectors of polynomial coefficients, the convolution of *x* and *y* is equivalent to the multiplication of these two polynomials.

AlgorithmThe convolution of two functions $x(t)$ and $y(t)$, denoted as $x * y$, is defined by

$$x * y \equiv \int_{-\infty}^{\infty} x(\tau)y(t - \tau)d\tau$$

in the time domain with $x * y = y * x$. According to the theorem of convolution, if $X(f)$ and $Y(f)$ are Fourier transforms of $x(t)$ and $y(t)$, that is,

$$x(t) \iff X(f) \quad \text{and} \quad y(t) \iff Y(f)$$

then

$$x * y \iff X(f)Y(f)$$

According to the theorem of discrete convolution,

if $X_d(f)$ and $Y_d(f)$ are discrete Fourier transforms of sequences $x[nT]$ and $y[nT]$, that is,

$$x[nT] \iff X_d(f) \quad \text{and} \quad y[nT] \iff Y_d(f)$$

then

$$x[nT] * y[nT] \iff X_d(f)Y_d(f)$$

Where T is the sample interval and n is an index used in the summation.

Based on the theorem of discrete convolution, DFT can be used to compute discrete convolutions. The procedure is first to compute the DFT of $x[nT]$ and $y[nT]$ and then use the theorem to compute

$$c[nT] = x[nT] * y[nT] = F_d^{-1}[X_d(f)Y_d(f)]$$

to get the convolution result.

Suppose $x[nT]$ and $y[nT]$ have the same length N for n from 0 to $N - 1$. Then the DFT of $x[nT]$ and $y[nT]$ have N points, while $c[nT]$ has length of $N + N - 1$. Therefore, the signal produced by $x[nT] * y[nT]$ has length that is longer than the DFT length. The length difference produces time aliasing when the inverse DFT of $X_d(f)Y_d(f)$ is used to obtain $x[nT] * y[nT]$. Thus, $c[nT] = x[nT]y[nT]$ is a time-aliased version of the sequences convolution. To solve the time-aliasing problem, DFT can be used to compute discrete convolution of two signals if zero padded to the signals to extend their lengths to equal the discrete convolution length. That is, if the lengths of $x[nT]$ and $y[nT]$ are N_x and N_y , respectively, then $x[nT]$ and $y[nT]$ can be padded with $N_y - 1$ and $N_x - 1$ zeroes, respectively. This zero padding produces sample signals that are both of length $N_x + N_y - 1$. DFT can be used with $(N_x + N_y - 1)$ points to compute $X_d(f)$ and $Y_d(f)$. The length, $N_x + N_y - 1$, of the sample sequence obtained from the inverse DFT $F_d^{-1}[X_d(f)Y_d(f)]$ equals the length of the discrete convolution so that the discrete convolution is obtained.

In this numeric function, the sizes of two convolution arrays x and y are expanded to $m + n - 1$ and zero padded internally. The DFT algorithm is used to compute the discrete Fourier transform of x and y . Multiplying two transforms together component by component, then using the inverse DFT algorithm to take the inverse discrete Fourier transform of the products, the answer is the convolution $x(nT) * y(nT)$.

Example 1

Assume $x(t)$ and $y(t)$ are polynomial functions, convolution of x and y is equivalent to multiplication of the two polynomials.

$$\begin{aligned} x(t) &= t^5 + 2 * t^4 + 3 * t^3 + 4 * t^2 + 5 * t + 6; \\ y(t) &= 6 * t + 7; \end{aligned}$$

Convolution of $x(t) * y(t)$ or multiplication of polynomials $x(t)$ and $y(t)$ is equal to

$$c(t) = x(t) * y(t) = 6 * t^6 + 19 * t^5 + 32 * t^4 + 45 * t^3 + 58 * t^2 + 71 * t + 42$$

```
#include <stdio.h>
#include <numeric.h>

#define N 6          /* data x array size */
#define M 2          /* data y array size */
#define N2 (N+M-1)

int main() {
    int i;
    array double c[N2], x[N]={1, 2, 3, 4, 5, 6}, y[M]={6, 7};

    conv(c, x, y); /* float data convolution */
    printf("Polynomial multiplication\n");
    printf("x=%6.3f\n", x);
    printf("y=%6.3f\n", y);
    printf("c=%6.3f\n", c);
}
```

Output

```
Polynomial multiplication
x= 1.000  2.000  3.000  4.000  5.000  6.000

y= 6.000  7.000

c= 6.000 19.000 32.000 45.000 58.000 71.000 42.000
```

Example 2

The sequences corresponding to the input and unit pulse response of a filter in a sample data control system are given as

$$x[n] = \begin{cases} n+1 & 0 \leq n < 2 \\ 5-n & 2 \leq n \leq 3 \\ 0 & \text{elsewhere} \end{cases}$$

$$y[n] = \begin{cases} -n/2 & 2 \leq n \leq 4 \\ 0 & \text{elsewhere} \end{cases}$$

The convolution sum or output sequence becomes

$$\begin{aligned} c[n] = x[n] * y[n] &= \sum_{m=-\infty}^{\infty} x[m]y[n-m] \\ &= \begin{cases} \sum_{m=2}^4 x[m]y[n-m] & 2 \leq n \leq 7 \\ 0 & \text{elsewhere} \end{cases} \end{aligned}$$

The six values of $c[n]$ for $i = 2$ to 7 can be calculated as follows

$$\begin{aligned} c[2] &= x[2]h[0] + x[3]h[-1] + x[4]h[-2] = -1 \\ c[3] &= x[2]h[1] + x[3]h[0] + x[4]h[-1] = -3.5 \\ c[4] &= x[2]h[2] + x[3]h[1] + x[4]h[0] = -8 \\ c[5] &= x[2]h[3] + x[3]h[2] + x[4]h[1] = -10.5 \\ c[6] &= x[2]h[4] + x[3]h[3] + x[4]h[2] = -9 \\ c[7] &= x[2]h[5] + x[3]h[4] + x[4]h[3] = -4 \end{aligned}$$

```
#include <stdio.h>
#include <chplot.h>
#include <numeric.h>

#define N 4          /* data x array size */
#define M 5          /* data y array size */
#define N2 (N+M-1)

int main() {
    int i,n1[N],n2[M],n3[N2];
    array double c[N2],x[N]={1,2,3,2},y[M]={0,0,-1,-1.5,-2};
    class CPlot plot;
    string_t labelx="x",
             labely="y";
```

```

    linspace(n1,0,N-1);
    linspace(n2,0,M-1);
    linspace(n3,0,N2-1);
    conv(c,x,y);
    printf("x=%6.3f\n",x);
    printf("y=%6.3f\n",y);
    printf("c=%6.3f\n",c);

    plot.data2D(n1,x);
    plot.plotType(PLOT_PLOTTYPE_IMPULSES, 0);
    plot.axisRange(PLOT_AXIS_X, 0,7,1);
    plot.axisRange(PLOT_AXIS_Y, -12,4,1);
    plot.label(PLOT_AXIS_X,"n");
    plot.label(PLOT_AXIS_Y,"x[n]");
    plot.title("Input sequence");
    plot.plotting();
    plot.deletePlots();
    plot.data2D(n2,y);
    plot.plotType(PLOT_PLOTTYPE_IMPULSES, 0);
    plot.axisRange(PLOT_AXIS_X, 0,7,1);
    plot.axisRange(PLOT_AXIS_Y, -12,4,1);
    plot.label(PLOT_AXIS_X,"n");
    plot.label(PLOT_AXIS_Y,"y[n]");
    plot.title("Unit Pulse Response sequence");
    plot.plotting();
    plot.deletePlots();
    plot.data2D(n3,c);
    plot.plotType(PLOT_PLOTTYPE_IMPULSES, 0);
    plot.axisRange(PLOT_AXIS_X, 0,7,1);
    plot.axisRange(PLOT_AXIS_Y, -12,4,1);
    plot.label(PLOT_AXIS_X,"n");
    plot.label(PLOT_AXIS_Y,"c[n]");
    plot.title("Output Response sequence");
    plot.plotting();
}

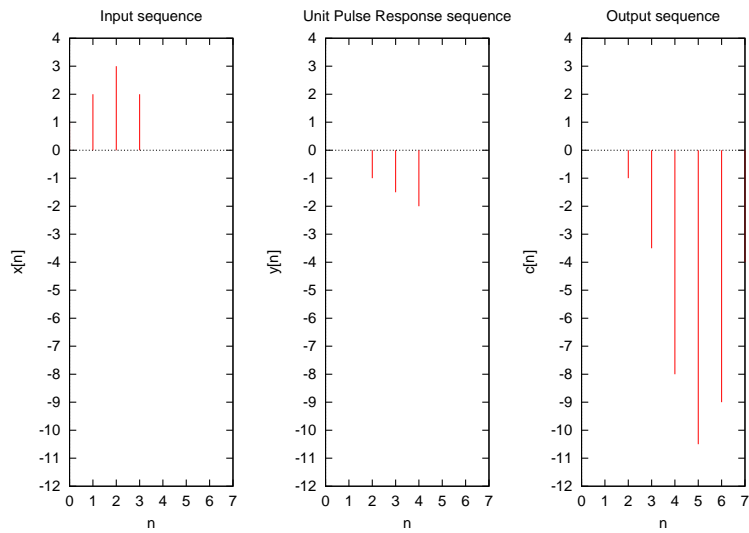
```

Output

```
x= 1.000  2.000  3.000  2.000
```

```
y= 0.000  0.000 -1.000 -1.500 -2.000
```

```
c= 0.000  0.000 -1.000 -3.500 -8.000 -10.500 -9.000 -4.000
```

**See Also**

`fft()`, `ifft()`, `conv2()`, `deconv()`.

References

Nussbaumer, H. J., *Fast Fourier Transform and Convolution Algorithms*, New York: Springer-Verlay, 1982

conv2

Synopsis

```
#include <numeric.h>
```

```
int conv2(array double complex c[&][&], array double complex f[&][&],
          array double complex g[&][&], ... /* [string_t method] */);
```

Syntax

```
conv2(c, f, g)
conv2(c, g, g, method);
```

Purpose

Two-dimensional Discrete Fourier Transform (DFT) based convolution.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

c A two-dimensional **array** with size of a section of $(na + nb - 1) \times (ma + mb - 1)$ specified by "method". It contains the result of convolution of *f* and *g*.

f A two-dimensional **array** of size $(na) \times (ma)$ used for convolution.

g A two-dimensional **array** of size $(nb) \times (mb)$ used for convolution.

method The **string_t** *method* used to specify the convolution result.

"full" - (default) returns the full 2-D convolution. The size is $(na + nb - 1) \times (ma + mb - 1)$.

"same" - returns the central part of 2-D convolution. That is the same size as *f*.

"valid" - returns only those parts of 2-D convolution that are computed without the zero-padded edges, the size of *c* is $(na - nb + 1) \times (ma - mb + 1)$ where the size of *f* must be bigger than the size of *g*. That is, $\text{size}(f) > \text{size}(g)$.

Description

The input two-dimensional **array** *f* and *g* can be of any supported arithmetic data type and sizes, respectively. Conversion of the data to **double complex** is performed internally. If both of *f* and *g* are **real** data, the result is a two-dimensional **real array** *c*. If either one of *f* and *g* is **complex**, the result **array** *c* will be **complex** data type.

If size of *f* and size of *g* are $(ma) \times (na)$ and $(mb) \times (nb)$, respectively, then

method = "full" (default) the size of *c* is $(ma + mb - 1) \times (na + nb - 1)$;

method = "same" the size of *c* is $(ma) \times (na)$;

method = "valid" the size of *c* is $(ma - mb + 1) \times (na - nb + 1)$ where the size of *f* must be bigger than the size of *g*. That is, $\text{size}(f) > \text{size}(g)$.

Algorithm

Two-dimensional convolution is analogous in form to one-dimensional convolution. Thus for two functions $f(x, y)$ and $g(x, y)$, we define

$$f(x, y) * g(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(\alpha, \beta) g(x - \alpha, y - \beta) d\alpha d\beta$$

The convolution theorem in two dimensions, then, is given by the relations

$$f(x, y) * g(x, y) \iff F(u, v)G(u, v)$$

where $F(u, v)$ and $G(u, v)$ are two-dimensional Fourier transform of $f(x, y)$ and $g(x, y)$, respectively. The discrete convolution of the two functions $f(mT, nT)$ and $g(mT, nT)$ is given by the relation

$$f(mT, nT) * g(mT, nT) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(mT, nT)g(k_1 - m, k_2 - n)$$

for $k_1 = 0, 1, 2, \dots, M - 1$ and $k_2 = 0, 1, 2, \dots, N - 1$. Where T is the sampling interval, M and N are periods in x and y directions, respectively.

Similarly, the discrete convolution theorem in two dimensions is given as

$$c(mT, nT) = f(mT, nT) * g(mT, nT) \iff F_d(f_1, f_2)G_d(f_1, f_2)$$

where $F_d(f_1, f_2)$ and $G_d(f_1, f_2)$ are discrete Fourier transform of $f(mT, nT)$ and $g(mT, nT)$, respectively. We can compute the DFT of $f(mT, nT)$ and $g(mT, nT)$ and then use the convolution theorem to compute

$$c(mT, nT) = f(mT, nT) * g(mT, nT) = F_d^{-1}[F_d(f_1, f_2)G_d(f_1, f_2)]$$

to get the two-dimensional result.

Similar to the one-dimensional convolution, if periods M and N are used to compute the convolution, there is a time-aliasing in the signals. To solve the time-aliasing problem, zeroes are padded to the signal to extend their lengths to equal the discrete convolution length. That is, if the lengths of $f(nT, mT)$ is $(na) \times (ma)$ and $g(nT, mT)$ is $(nb) \times (mb)$, then $f(nT, mT)$ and $g(nT, mT)$ are padded with $(nb - 1) \times (mb - 1)$ and $(na - 1) \times (ma - 1)$ zeroes, respectively. This zero padding produces sample signals that are both of length $(na + nb - 1) \times (ma + mb - 1)$. A two-dimensional DFT with $(na + nb - 1) \times (ma + mb - 1)$ points is used to compute $F_d(f_1, f_2)$ and $G_d(F_1, F_2)$. The length, $(na + nb - 1) \times (ma + mb - 1)$, of the sample sequence obtained from the inverse DFT $F_d^{-1}[F_d(f_1, f_2)G_d(f_1, f_2)]$ equals the length of the discrete convolution so that the discrete convolution is produced.

In this numeric function, the size of two convolution arrays f and g are expanded to $(na + nb - 1) \times (ma + mb - 1)$ and zero padded internally. The DFT algorithm is used to compute the two-dimensional discrete Fourier transform of f and g . Multiplying two transforms together component by component, then using the inverse DFT algorithm to take the inverse discrete Fourier transform of the products, the answer is the two-dimensional convolution $f(mT, nT) * g(mT, nT)$.

Example

In image processing, a Sobel filter is a simple approximation to the concept of edge detection. Convoluting the original two-dimensional image data with a Sobel filter

$$g_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

detects the edges in x-direction. Similarly, convoluting a Sobel filter

$$g_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

which is the transpose of matrix g_x , edges in y-direction can be detected.

```

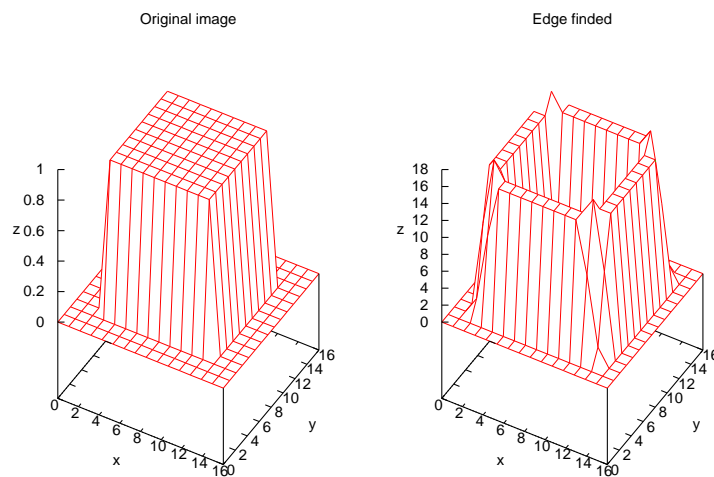
#include <math.h>
#include <chplot.h>
#include <numeric.h>

int main() {
    int i, j;
    array double g[3][3]={{-1,0,1},{-2,0,2},{-1,0,1}};
    array double x[16], y[16], z1[256], f[16][16], H[18][18], V[18][18], Z[18][18], z1[256];

    linspace(x,0,16);
    linspace(y,0,16);
    for(i=3; i<13; i++)
        for(j=3; j<13; j++) {
            z1[i*16+j]=1;
            f[i][j] = 1;
        }
    plotxyz(x,y,z1); /* original image */
    conv2(H,f,g);
    conv2(V,f,transpose(g));
    Z = H .* H + V .* V; /* magnitude of the pixel value */
    for(i = 0; i<16; i++)
        for(j=0; j<16; j++)
            Z1[i*16+j] = Z[i+1][j+1];
    plotxyz(x,y,Z1); /* edge finded image */
}

```

Output



See Also

fft(), ifft(), conv(), deconv().

References

Rafael C. Gonzalez, Paul Wintz, Digital Image Processing, Second Edition, Addison-Wesley Publishing Company, 1987.

corrcoef

Synopsis

```
#include <numeric.h>
```

```
int corrcoef(array double &c, array double &x, ... /* [array double &y] */);
```

Syntax

```
corrcoef(c, x)
```

```
corrcoef(c, x, y)
```

Purpose

Correlation coefficients calculation.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

c Correlation coefficient matrix formed from array *x*.

x A vector of observation or a two-dimensional **array** whose column is an observation and row is a variable.

y Optional argument. *y* is a vector or two-dimensional array whose length or column length shall be the same as the column length of *x*.

Description

The input array *x* can be of any supported arithmetic data type of vector or two-dimensional array of any size $n \times m$ (if *x* is a vector, regard the size as $1 \times m$). Each column of *x* is an observation and each row is a variable. Optional input *y* can only be of **double** data type and same column length as *x*. That is, the size of *y* is $n_1 \times m$ or regard as $1 \times m$ if *y* is a vector. Attachment of *y* to row *a* of *x* as a new expanding matrix [**X**] is performed internally. **corrcoef**(*c*, *x*, *y*) is equivalent to **corrcoef**(*c*, [**X**]) where **X** is defined as $\mathbf{X} = \begin{pmatrix} x \\ y \end{pmatrix}$. The size of [**X**] is $(n + n_1) \times m$. Conversion of the data to **double** is performed internally. The result *c* is a matrix of correlation coefficients.

Algorithm

For array *x* of size $N \times M$, which has *N* variable and each variable has total *M* observation, correlation coefficient matrix *c* is related to the covariance matrix *C* by

$$c_{ij} = \frac{C_{ij}}{\sqrt{C_{ii} * C_{jj}}}$$

where *C* is a covariance matrix. It is calculated by

$$C = \frac{1}{N - 1} u * u'$$

where element *u* is defined as

$$u_{ij} = x_{ij} - \mu_i; \quad i = 0, 1, \dots, N; j = 0, 1, \dots, M$$

and

$$\mu_i = \frac{1}{M} \sum_{j=1}^M x_{ij}; \quad i = 0, 1, \dots, N$$

where x_{ij} is the element of the input matrix.

Example

Consider an array $x[4][5]$.

$$x = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 0 & 0 & 0 & 0 & 1 \\ 2 & 3 & 4 & 5 & 6 \end{bmatrix}$$

where each row of the x is a variable and column is an observation. The result array c is the correlation coefficient matrix.

```
#include <stdio.h>
#include <numeric.h>

#define N 4          /* data array size */
#define N1 5
#define N2 4

int main(){
    int i, j;
    array double c[N2][N2];
    array double x[N][N1]={ {1,2,3,4,5},
                             {1,2,3,4,5},
                             {0,0,0,0,1},
                             {2,3,4,5,6}};

    corrcoef(c,x);
    printf("x=%6.3f\n", x);
    printf("correlation coefficient matrix\n");
    printf("%6.3f\n", c);
}
```

Output

```
x= 1.000  2.000  3.000  4.000  5.000
   1.000  2.000  3.000  4.000  5.000
   0.000  0.000  0.000  0.000  1.000
   2.000  3.000  4.000  5.000  6.000

correlation coefficient matrix
 1.000  1.000  0.707  1.000
 1.000  1.000  0.707  1.000
 0.707  0.707  1.000  0.707
 1.000  1.000  0.707  1.000
```

See Also

covariance().

correlation2

Synopsis

```
#include <numeric.h>
```

```
double correlation2(array double x[&][&], array double y[&][&]);
```

Syntax

```
correlation2(x, y)
```

Purpose

Two-dimensional correlation coefficient.

Return Value

This function returns the two-dimensional correlation coefficient.

Parameters

x A square matrix of size $n \times n$. It contains the original data for correlation coefficient calculation.

y A square matrix of the same size as *x*. It contains the original data for correlation coefficient calculation.

Algorithm

This function is used to calculate the two-dimensional correlation coefficient. The two-dimensional correlation coefficient is defined as

$$c = \frac{\sum_{i=1}^n \sum_{j=1}^n (xx_{ij} * yy_{ij})}{\sqrt{(\sum_{i=1}^n \sum_{j=1}^n xx_{ij}^2) * (\sum_{i=1}^n \sum_{j=1}^n yy_{ij}^2)}}$$

where

$$xx_{ij} = x_{ij} - \mu_x$$

$$yy_{ij} = y_{ij} - \mu_y$$

μ_x and μ_y are the mean values of the matrix *x* and *y*.

$$\mu_x = \frac{1}{n * n} \sum_{i=1}^n \sum_{j=1}^n x_{ij}$$

$$\mu_y = \frac{1}{n * n} \sum_{i=1}^n \sum_{j=1}^n y_{ij}$$

Example

Calculate two square matrix correlation coefficient.

```
#include <stdio.h>
#include <numeric.h>

#define NC 6
#define ND 3
```

```
#define N 20

int main() {
    int i,j,k;
    array double x[3][3]={1,2,3,
                          3,4,5,
                          6,7,8};
    array double y[3][3]={3,2,2,
                          3,8,5,
                          6,2,5};

    double c;
    c = correlation2(x,y);
    printf("x=\n%f",x);
    printf("y=\n%f",y);
    printf("c=\n%f\n",c);
}
```

Output

```
x=
1.000000 2.000000 3.000000
3.000000 4.000000 5.000000
6.000000 7.000000 8.000000
y=
3.000000 2.000000 2.000000
3.000000 8.000000 5.000000
6.000000 2.000000 5.000000
c=
0.326637
```

See Also
correlation().

covariance

Synopsis

```
#include <numeric.h>
```

```
int covariance(array double &c, array double &x, ... /* [array double &y] */);
```

Syntax

```
covariance(c, x)
```

```
covariance(c, x, y)
```

Purpose

Covariance matrix.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

c Covariance matrix. If there is only one input *x* and it is a vector, this is the variance. If *x* is a matrix, where each column is an observation and each row is a variable, *c* is the covariance matrix. The diagonal of *c* is a vector of variances for each row. The square root of diagonal of *c* is a vector of standard deviations.

x A vector whose elements are an observation, or a two-dimensional **array** whose column is an observation and row is a variable.

y Optional argument. *y* could be a vector or a two-dimensional **array** whose length or column length shall be same as the column length of *x*.

Description

The input **array** *x* can be of any supported arithmetic data type of vector or matrix of any size $n \times m$ (if *x* is a vector, regard the size as $1 \times m$). Each column of *x* is an observation and each row is a variable. Optional input *y* can only be of **double** data type and same column length as *x*. That is, the size of *y* is $n_1 \times m$ or regard as $1 \times m$ if *y* is a vector. Attachment of *y* to row of *x* as a new expanding matrix **[X]** is performed internally. **covariance**(*c*, *x*, *y*) is equivalent to **covariance**(*c*, **[X]**) where **X** is defined as $\mathbf{X} = \begin{pmatrix} x \\ y \end{pmatrix}$. The size of **[X]** is $(n + n_1) \times (m)$. Conversion of the data to **double** is performed internally. The result *c* is a variance or a covariance matrix depending on *x* and *y*. If *x* is a vector and no optional input *y*, the result *c* is a variance. Otherwise it is a covariance. Diagonal of *c* is a vector of variances for each row. The square root of the diagonal of *c* is a vector of standard deviations.

Algorithm

For each of **array** $x[N][M]$ of *N* variable which has total *M* observation, covariance function is defined as

$$\text{covariance}(x_i, x_j) = E[(x_i - \mu_i)(x_j - \mu_j)] \quad i, j = 1, 2, \dots, N;$$

where x_i and x_j are the *i*th and *j*th row elements. *E* is the mathematical expectation and $\mu_i = Ex_i$.

In function **covariance**(), the mean from each column is removed before calculating the result.

$$u_{ij} = x_{ij} - \mu_i; \quad i = 0, 1, \dots, N; j = 0, 1, \dots, M$$

where

$$\mu_i = \frac{1}{M} \sum_{j=1}^M x_{ij}; \quad i = 0, 1, \dots, N$$

Then the covariance matrix is calculated by

$$c = \frac{1}{N-1} u * u'$$

Example

Consider an array $x[4][5]$.

$$x = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 0 & 0 & 0 & 0 & 1 \\ 2 & 3 & 4 & 5 & 6 \end{bmatrix}$$

where each row of the x is a variable and column is an observation. The result **array** c is the covariance matrix. The diagonal elements $c(i, j)$ represent the variances for the rows of x . The off-diagonal element $c(i, j)$ represents the covariances of rows i and j .

```
#include <stdio.h>
#include <numeric.h>

#define N 4          /* data array size */
#define N1 5
#define N2 4

int main() {
    int i, j;
    array double c[N2][N2];
    array double x[N][N1]={{1,2,3,4,5},
                           {1,2,3,4,5},
                           {0,0,0,0,1},
                           {2,3,4,5,6}};

    covariance(c,x);    /* covariance of matrix */

    printf("x=%8.3f\n", x);
    printf("c=%8.3f\n", c);
}
```

Output

```
x=  1.000  2.000  3.000  4.000  5.000
    1.000  2.000  3.000  4.000  5.000
    0.000  0.000  0.000  0.000  1.000
    2.000  3.000  4.000  5.000  6.000

c=  2.500  2.500  0.500  2.500
    2.500  2.500  0.500  2.500
    0.500  0.500  0.200  0.500
    2.500  2.500  0.500  2.500
```


See Also
corrcoef().

cpolyeval

Synopsis

```
#include <numeric.h>
```

```
double complex cpolyeval(array double complex c[&], double complex z);
```

Syntax

```
cpolyeval(c, z)
```

Purpose

Calculate the value of a polynomial at point z .

Return Value

This function returns the **complex** value of a polynomial at a given complex point z .

Parameters

c A vector of coefficients of a polynomial.

z A value of point x in which polynomial is evaluated.

Description

The vector of coefficient of polynomial c can be of any supported arithmetic data type and extent size. The value of z can be any data type. Conversion of the data to **double complex** is performed internally. The return value is **double complex** data of the value of the polynomial.

Algorithm

For polynomial

$$P(z) = c_0 z^n + c_1 z^{n-1} + c_2 z^{n-2} + \cdots + c_{n-1} z + c_n$$

The calculation of the polynomial value is implemented using the follow algorithm

$$P = (((\cdots (c_0 z + c_1) z + c_2) z + c_3) z + \cdots + c_{n-1}) z + c_n$$

Example

Evaluate polynormal

$$P(z) = z^5 - 5z^4 + 10z^3 - 10z^2 + 5z - 1$$

at complex point $z = 0.1 + i0.1$.

```
#include <stdio.h>
#include <numeric.h>

#define NC 6
#define ND 3

int main() {
    int i, j, k;
```

```
double c[NC]={1.0,-5.0,10.0,-10.0,5.0,-1.0};
double complex z=complex(0.1,1),val;

val = cpolyeval(c,z);
printf("z = %f    function value = %f\n",z,val);
}
```

Output

```
z = complex(0.100000,1.000000)    function value = complex(2.199510,-3.819500)
```

see also

polyeval(), **polyevalarray()**.

References

William H. Press, et al, *Numerical Recipes in C*, second edition, Cambridge University Press, 1997.

cproduct

Synopsis

```
#include <numeric.h>
```

```
double complex cproduct(array double complex &a, ... /* [array double complex v[:]] */);
```

Syntax

```
cproduct(a)
```

```
cproduct(a, v)
```

Purpose

Calculate products of all elements and products of the elements of each rows of a complex array.

Return Value

This function returns the product of all elements.

Parameters

a The input complex array for which the products are calculated.

v The output array which contains the products of each row of a two-dimensional complex array.

Description

This function calculates the product of all elements in a complex array of any dimension. If the array is a two-dimensional matrix, the function can calculate the products of each row. The product of all elements of the array is returned by the function and the products of the elements of each row are passed out by the argument *v*. The input array can be of any dimension and any arithmetic data types.

Example1

Calculate the products of all elements of the complex arrays with different data types and dimensions.

```
#include <numeric.h>
int main() {
    complex double a[3] = {complex(1,2), complex(2,3), complex(1,2)};
    complex double b[2][3] = {1,2,3,4,5,6};
    int b1[2][3] = {1,2,3,4,5,6};
    double c[2][3][5];
    c[0][0][0] = 10;
    c[0][0][1] = 20;
    double complex products;

    products = cproduct(a);
    printf("cproduct(a) = %f\n", products);
    products = cproduct(b);
    printf("cproduct(b) = %f\n", products);
    products = cproduct(b1);
    printf("cproduct(b) = %f\n", products);
    products = cproduct(c);
    printf("cproduct(c) = %f\n", products);
}
```

Output1

```

cproduct(a) = complex(-18.000000,-1.000000)
cproduct(b) = complex(720.000000,0.000000)
cproduct(b) = complex(720.000000,0.000000)
cproduct(c) = complex(0.000000,0.000000)

```

Example2

Calculate the products of the elements of each row of two-dimensional complex arrays with different data types and dimensions.

```

#include <numeric.h>
int main() {
    double complex a[2][3] = {complex(1,2),2,3,
                              4,5,6};
    array double complex b[3][4] = {1,2,3,4,
                                    5,6,7,8,
                                    1,2,3,4};
    array double complex cproductv1[2], cproductv2[3];

    cproduct(a, cproductv1);
    printf("cproduct(a, cproductv1) = %f\n", cproductv1);
    cproduct(b, cproductv2);
    printf("cproduct(b, cproductv2) = %f\n", cproductv2);
}

```

Output2

```

cproduct(a, cproductv1) = complex(6.000000,12.000000) complex(120.000000,0.000000)

```

```

cproduct(b, cproductv2) = complex(24.000000,0.000000) complex(1680.000000,0.000000) complex(24.000000,0.000000)

```

See Also

product(), **cumprod()**, **mean()**, **median()**, **sum()**, **cumsum()**.

References

cquadratic

Synopsis

```
#include <numeric.h>
```

```
int cquadratic(double a, double b, double c, double complex & x1, double complex & x2);
```

Purpose

Solve for the solutions of the quadratic equation $ax^2 + bx + c = 0$.

Return Value

This function returns the number of solutions for the quadratic equation $ax^2 + bx + c = 0$.

Parameters

a The coefficient *a* of $ax^2 + bx + c = 0$.

b The coefficient *a* of $ax^2 + bx + c = 0$.

c The coefficient *a* of $ax^2 + bx + c = 0$.

x1 The first solution of the quadratic equation $ax^2 + bx + c = 0$, expressed in complex number .

x2 The second solution of the quadratic equation $ax^2 + bx + c = 0$, expressed in complex number.

Description

This function calculates the solutions of the quadratic equation $ax^2 + bx + c = 0$. The solutions in complex number are passed back to the calling function through the fourth and fifth arguments of the function by reference. If the discriminant $b^2 - 4ac$ is zero, there is one solution and the function returns 1. Otherwise, there are two solutions and the function returns 2.

Example

```
/* File: cquadratic.ch */
#include <numeric.h>
int main() {
    /* y = x*x -5x +6 */
    double a=1, b=-5, c = 6;
    double complex x1, x2;

    quadratic(a, b, c, x1, x2);
    printf("Solutions for x*x-5x+6=0 are %.2f and %.2f\n", x1, x2);

    c = 7.25;
    cquadratic(a, b, c, x1, x2);
    printf("Solutions for x*x-5x+7.25=0 are %.2f and %.2f\n", x1, x2);
}
```

Output

Solutions for $x^2-5x+6=0$ are `complex(3.00,0.00)` and `complex(2.00,0.00)`
Solutions for $x^2-5x+7.25=0$ are `complex(2.50,1.00)` and `complex(2.50,-1.00)`

See Also

quadratic(), **vertex()**.

CROSS

Synopsis

```
#include <numeric.h>
```

```
array double cross(array double a[&], array double b[&])[3];
```

Purpose

Calculate the cross product of two vectors.

Return Value

This function returns the cross product of vectors *a* and *b*.

Parameters

a An input array which contains the first vector.

b An input array which contains the second vector.

Description

The function `cross()` returns the cross product of vectors *a* and *b*. The number of elements of each vector shall be 3.

Example

```
#include <numeric.h>
int main() {
    array double a[3] = {1, 2, 3};
    array double b[ ] = {1, 2, 3};
    array double c[3] = {2, 3, 4};
    array double crossprod[3];

    crossprod = cross(a, b);
    printf("cross(a,b) = %f\n", crossprod);
    crossprod = cross(a, c);
    printf("cross(a,c) = %f\n", crossprod);
}
```

Output

```
cross(a,b) = 0.000000 0.000000 0.000000
```

```
cross(a,c) = -1.000000 2.000000 -1.000000
```

See Also

`dot()`.

References

csum**Synopsis****#include** <numeric.h>**double complex csum**(array double complex &*a*, ... /*[array double complex *v*[:]]*/);**Syntax****csum**(*a*)**csum**(*a*, *v*)**Purpose**

Calculate the sum of all elements of an array and the sums of each row of a two-dimensional complex array.

Return Value

This function returns the sum of all elements of the complex array.

Parameters*a* The input complex array.*v* The output array which contains the sums of each row of array.**Description**

The function calculates the sum of all elements in a complex array of any dimension. If the array is a two-dimensional matrix, the function can calculate the sum of each row. The input array can be any arithmetic data type and of any dimension.

Example1

Calculate the sum of all elements of a complex arrays of different data types.

```
#include <numeric.h>
int main() {
    complex double a[3] = {complex(1,2), complex(2,3)};
    complex double b[2][3] = {1,2,3,4,5,6};
    int b1[2][3] = {1,2,3,4,5,6};
    double c[2][3][5];
    c[0][0][0] = 10;
    c[0][0][1] = 20;
    double complex sums;

    sums = csum(a);
    printf("csum(a) = %f\n", sums);
    sums = csum(b);
    printf("csum(b) = %f\n", sums);
    sums = csum(b1);
    printf("csum(b) = %f\n", sums);
    sums = csum(c);
    printf("csum(c) = %f\n", sums);
}
```

Output1

```

csum(a) = complex(3.000000,5.000000)
csum(b) = complex(21.000000,0.000000)
csum(b) = complex(21.000000,0.000000)
csum(c) = complex(30.000000,0.000000)

```

Example2

Calculate the sums of each elements of the complex arrays

```

#include <numeric.h>
int main() {
    double complex a[2][3] = {complex(1,2),2,3,
                              4,5,6};
    array double complex b[3][4] = {1,2,3,4,
                                    5,6,7,8,
                                    1,2,3,4};
    array int    b1[3][4] = {1,2,3,4,
                            5,6,7,8,
                            1,2,3,4};
    array double complex csumv1[2], csumv2[3];

    csum(a, csumv1);
    printf("csum(a, csumv1) = %.3f\n", csumv1);
    csum(b, csumv2);
    printf("csum(b, csumv2) = %.3f\n", csumv2);
    csum(b1, csumv2);
    printf("csum(b1, csumv2) = %.3f\n", csumv2);
}

```

Output2

```

csum(a, csumv1) = complex(6.000,2.000) complex(15.000,0.000)
csum(b, csumv2) = complex(10.000,0.000) complex(26.000,0.000) complex(10.000,0.000)
csum(b1, csumv2) = complex(10.000,0.000) complex(26.000,0.000) complex(10.000,0.000)

```

See Also

sum(), cumsum(), trace(), product().

References

ctrace

Synopsis

```
#include <numeric.h>
```

```
double complex ctrace(array double complex a[&][&]);
```

Purpose

Calculate the sum of diagonal elements of a complex matrix.

Return Value

This function returns the sum.

Parameters

a Input two-dimensional complex array.

Description

This function calculates the sum of the diagonal elements of the complex matrix.

Algorithm

For a matrix *a*, the trace is defined as

$$trace = \sum_{i=1}^n a_{ii}$$

Example

```
#include <numeric.h>
int main() {
    array double complex a[2][2] = {complex(2,3), -4,
                                    3, -7};
    array double complex a2[3][2] = {complex(2,3), -4,
                                    3, -7,
                                    1, 1};
    array double b[2][2] = {2, -4,
                           3, -7};
    double complex t;

    t = ctrace(a);
    printf("ctrace(a) = %f\n", t);
    t = ctrace(a2);
    printf("ctrace(a2) = %f\n", t);
    t = ctrace(b);
    printf("ctrace(b) = %f\n", t);
}
```

Output

```
ctrace(a) = complex(-5.000000,3.000000)
ctrace(a2) = complex(-5.000000,3.000000)
ctrace(b) = complex(-5.000000,0.000000)
```

See Also

[trace\(\)](#), [sum\(\)](#), [mean\(\)](#), [median\(\)](#).

References

ctriangularmatrix

Synopsis

```
#include <numeric.h>
```

```
array complex double ctriangularmatrix(string_t pos, array double complex a[&][&], ... /* [int k]
*/)[:][:];
```

Syntax

```
ctriangularmatrix(pos, a)
```

```
ctriangularmatrix(pos, a, k)
```

Purpose

Get the upper or lower triangular part of a complex matrix.

Return Value

This function returns the upper or lower triangular part of a complex matrix.

Parameters

pos Input string indicating which part of the matrix to be returned.

a Input 2-dimensional array.

k Input integer. A matrix is returned on and below the *k*th diagonal of the input matrix.

Description

This function gets the triangular matrix on and below the *k*th diagonal of matrix *a*. For *pos* of “upper” the function returns the upper complex triangular part of the matrix, and for *pos* of “lower” the function returns the lower complex part of the matrix. *k* indicates the offset of the triangular matrix to the lower *k*th diagonal of the matrix.

Example

```
#include <numeric.h>
int main() {
    array double complex a[4][3] = {1,2,3,
                                    4,5,6,
                                    7,8,9,
                                    3,6,4};

    int n = 4, m=3, k;
    array double complex t[n][m];

    t = ctriangularmatrix("upper",a);
    printf("ctriangularmatrix(\"upper\", t) =\n%5.3f\n", t);

    k = 0;
    t = ctriangularmatrix("upper",a,k);
    printf("ctriangularmatrix(\"upper\", t, 0) =\n%5.3f\n", t);

    k = 1;
    t = ctriangularmatrix("upper",a,k);
    printf("ctriangularmatrix(\"upper\", t, 1) =\n%5.3f\n", t);
```

```

k = -1;
t = ctriangularmatrix("upper", a, k);
printf("ctriangularmatrix(\"upper\", t, -1) =\n%5.3f\n", t);

t = ctriangularmatrix("lower", a);
printf("ctriangularmatrix(\"lower\", t) =\n%5.3f\n", t);

k = 0;
t = ctriangularmatrix("lower", a, k);
printf("ctriangularmatrix(\"lower\", t, 0) =\n%5.3f\n", t);

k = 1;
t = ctriangularmatrix("lower", a, k);
printf("ctriangularmatrix(\"lower\", t, 1) =\n%5.3f\n", t);

k = -1;
t = ctriangularmatrix("lower", a, k);
printf("ctriangularmatrix(\"lower\", t, -1) =\n%5.3f\n", t);
}

```

Output

```

ctriangularmatrix("upper", t) =
complex(1.000,0.000) complex(2.000,0.000) complex(3.000,0.000)
complex(0.000,0.000) complex(5.000,0.000) complex(6.000,0.000)
complex(0.000,0.000) complex(0.000,0.000) complex(9.000,0.000)
complex(0.000,0.000) complex(0.000,0.000) complex(0.000,0.000)

ctriangularmatrix("upper", t, 0) =
complex(1.000,0.000) complex(2.000,0.000) complex(3.000,0.000)
complex(0.000,0.000) complex(5.000,0.000) complex(6.000,0.000)
complex(0.000,0.000) complex(0.000,0.000) complex(9.000,0.000)
complex(0.000,0.000) complex(0.000,0.000) complex(0.000,0.000)

ctriangularmatrix("upper", t, 1) =
complex(0.000,0.000) complex(2.000,0.000) complex(3.000,0.000)
complex(0.000,0.000) complex(0.000,0.000) complex(6.000,0.000)
complex(0.000,0.000) complex(0.000,0.000) complex(0.000,0.000)
complex(0.000,0.000) complex(0.000,0.000) complex(0.000,0.000)

ctriangularmatrix("upper", t, -1) =
complex(1.000,0.000) complex(2.000,0.000) complex(3.000,0.000)
complex(4.000,0.000) complex(5.000,0.000) complex(6.000,0.000)
complex(0.000,0.000) complex(8.000,0.000) complex(9.000,0.000)
complex(0.000,0.000) complex(0.000,0.000) complex(4.000,0.000)

ctriangularmatrix("lower", t) =
complex(1.000,0.000) complex(0.000,0.000) complex(0.000,0.000)
complex(4.000,0.000) complex(5.000,0.000) complex(0.000,0.000)
complex(7.000,0.000) complex(8.000,0.000) complex(9.000,0.000)
complex(3.000,0.000) complex(6.000,0.000) complex(4.000,0.000)

ctriangularmatrix("lower", t, 0) =
complex(1.000,0.000) complex(0.000,0.000) complex(0.000,0.000)
complex(4.000,0.000) complex(5.000,0.000) complex(0.000,0.000)
complex(7.000,0.000) complex(8.000,0.000) complex(9.000,0.000)
complex(3.000,0.000) complex(6.000,0.000) complex(4.000,0.000)

```

```
ctriangularmatrix("lower", t, 1) =  
complex(1.000,0.000) complex(2.000,0.000) complex(0.000,0.000)  
complex(4.000,0.000) complex(5.000,0.000) complex(6.000,0.000)  
complex(7.000,0.000) complex(8.000,0.000) complex(9.000,0.000)  
complex(3.000,0.000) complex(6.000,0.000) complex(4.000,0.000)  
  
ctriangularmatrix("lower", t, -1) =  
complex(0.000,0.000) complex(0.000,0.000) complex(0.000,0.000)  
complex(4.000,0.000) complex(0.000,0.000) complex(0.000,0.000)  
complex(7.000,0.000) complex(8.000,0.000) complex(0.000,0.000)  
complex(3.000,0.000) complex(6.000,0.000) complex(4.000,0.000)
```

See Also**triangularmatrix(), diagonal(), cdiagonal()**

cumprod

Synopsis

```
#include <numeric.h>
```

```
int cumprod(array double complex &y, array double complex &x);
```

Syntax

```
cumprod(y, x)
```

Purpose

Calculate cumulative product of elements in an array.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x Input array with dimension less than three. It contains data to be calculated.

y Output array of same dimension as *x*. It contains the result of cumulative product.

Description

This function computes the cumulative product of the input array *x*. If the input *x* is a vector, it calculates the cumulative product of the elements of *x*. If the input *x* is a two-dimensional matrix, it calculates the cumulative product over each row. If the input *x* is a three-dimensional array, it calculates the cumulative product over the the first dimension. It is invalid for calculation of cumulative product for arrays with more than three dimensions.

Algorithm

For a vector $x = [x_1, x_2, \dots, x_n]$, cumulative productive vector $y = [y_1, y_2, \dots, y_n]$ is defined by

$$y_i = x_1 * x_2 * \dots * x_i \quad i = 1, 2, \dots, n$$

Example

Calculation of cumulative products of arrays of different dimensions.

```
#include <numeric.h>
int main() {
    array double x[6]={1,2,3,4,5,6}, y[6];
    array double complex zx[2][3]={complex(1,1),2,3,complex(2,2),5,6}, zy[2][3];
    array double x1[2][3][4]={{1,2,3,4,
                               5,6,7,8,
                               5,6,7,8},
                              {10,11,12,13,
                               10,11,12,13,
                               14,15,16,17}};
    array double y1[2][3][4];

    cumprod(y, x);
    printf("x=%f", x);
    printf("y=%f", y);
    printf("\n");
```

```

    cumprod(zy, zx);
    printf("zx=%5.2f", zx);
    printf("zy=%5.2f", zy);
    printf("\n");

    cumprod(y1, x1);
    printf("x1=%f", x1);
    printf("y1=%f", y1);
}

```

Output

```

x=1.000000 2.000000 3.000000 4.000000 5.000000 6.000000
y=1.000000 2.000000 6.000000 24.000000 120.000000 720.000000

zx=complex( 1.00, 1.00) complex( 2.00, 0.00) complex( 3.00, 0.00)
complex( 2.00, 2.00) complex( 5.00, 0.00) complex( 6.00, 0.00)
zy=complex( 1.00, 1.00) complex( 2.00, 2.00) complex( 6.00, 6.00)
complex( 2.00, 2.00) complex(10.00,10.00) complex(60.00,60.00)

x1=1.000000 2.000000 3.000000 4.000000
5.000000 6.000000 7.000000 8.000000
5.000000 6.000000 7.000000 8.000000

10.000000 11.000000 12.000000 13.000000
10.000000 11.000000 12.000000 13.000000
14.000000 15.000000 16.000000 17.000000
y1=1.000000 2.000000 6.000000 24.000000
5.000000 30.000000 210.000000 1680.000000
5.000000 30.000000 210.000000 1680.000000

10.000000 110.000000 1320.000000 17160.000000
10.000000 110.000000 1320.000000 17160.000000
14.000000 210.000000 3360.000000 57120.000000

```

See Also

cumsum(), **product()**, **cproduct()**, **sum()**, **csum()**.

cumsum

Synopsis

```
#include <numeric.h>
```

```
int cumsum(array double complex &y, array double complex &x);
```

Syntax

```
cumsum(y, x)
```

Purpose

Calculate cumulative sum of elements in an array.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x Input array whose dimension are no more than three. It contains data to be calculated.

y Output array of same dimensional as *x*. It contains data the result of cumulative sum.

Description

This function computes the cumulative sum of the input array *x*. If the input *x* is a vector, it calculates the cumulative sum of the elements of *x*. If the input *x* is a two-dimensional matrix, it calculates the cumulative sum over each row. If the input *x* is a three-dimensional array, it calculates the cumulative sum over the first dimension. It is invalid for calculation of cumulative sum for arrays with more than three dimensions.

Algorithm

For a vector $x = [x_1, x_2, \dots, x_n]$, cumulative sum vector $y = [y_1, y_2, \dots, y_n]$ is defined by

$$y_i = x_1 + x_2 + \dots + x_i \quad i = 1, 2, \dots, n$$

Example

Calculation of cumulative sum of arrays of different dimensions.

```
#include <numeric.h>
int main() {
    array double x[6]={1,2,3,4,5,6}, y[6];
    array double complex zx[2][3]={complex(1,1),2,3,complex(2,2),5,6}, zy[2][3];
    array double x1[2][3][4]={{1,2,3,4,
                               5,6,7,8,
                               5,6,7,8},
                              {10,11,12,13,
                               10,11,12,13,
                               14,15,16,17}};

    array double y1[2][3][4];

    cumsum(y, x);
    printf("x=%f", x);
    printf("y=%f", y);
    printf("\n");
```

```

    cumsum(zy, zx);
    printf("zx=%5.2f", zx);
    printf("zy=%5.2f", zy);
    printf("\n");

    cumsum(y1, x1);
    printf("x1=%f", x1);
    printf("y1=%f", y1);
}

```

Output

```

x=1.000000 2.000000 3.000000 4.000000 5.000000 6.000000
y=1.000000 3.000000 6.000000 10.000000 15.000000 21.000000

zx=complex( 1.00, 1.00) complex( 2.00, 0.00) complex( 3.00, 0.00)
complex( 2.00, 2.00) complex( 5.00, 0.00) complex( 6.00, 0.00)
zy=complex( 1.00, 1.00) complex( 3.00, 1.00) complex( 6.00, 1.00)
complex( 2.00, 2.00) complex( 7.00, 2.00) complex(13.00, 2.00)

x1=1.000000 2.000000 3.000000 4.000000
5.000000 6.000000 7.000000 8.000000
5.000000 6.000000 7.000000 8.000000

10.000000 11.000000 12.000000 13.000000
10.000000 11.000000 12.000000 13.000000
14.000000 15.000000 16.000000 17.000000
y1=1.000000 3.000000 6.000000 10.000000
5.000000 11.000000 18.000000 26.000000
5.000000 11.000000 18.000000 26.000000

10.000000 21.000000 33.000000 46.000000
10.000000 21.000000 33.000000 46.000000
14.000000 29.000000 45.000000 62.000000

```

See Also

cumprod(), **product()**, **cproduct()**, **sum()**, **csum()**.

curvefit

Synopsis

```
#include <numeric.h>
```

```
int curvefit(double a[&], double x[&], double y[&], void (*funcs)(double, double [ ]), ...
             /*[double sig[ ], int ia[ ], double covar[:][:], double *chisq ]*/);
```

Syntax

```
curvefit(a, x, y, funcs)
```

```
curvefit(a, x, y, funcs, sig)
```

```
curvefit(a, x, y, funcs, sig, ia)
```

```
curvefit(a, x, y, funcs, sig, ia, covar)
```

```
curvefit(a, x, y, funcs, sig, ia, covar, chisq)
```

Purpose

Fit a set of data points x and y to a linear combination of specified functions of x .

Return Value

This function returns 0 on success and -1 on failure.

Parameters

a An output array which contains coefficients for curve fitting.

x An input array which contains variable values of data set.

y An input array which contains function values of data set.

$funcs$ The routine given by the user that passes the values of base functions evaluated at x .

sig An input array which contains individual standard (measurement error) deviations of data set. It can be set to 1 if unknown.

ia An input array. Those components of nonzero entries shall be fitted, and those components of zero entries shall be held at their input values.

$covar$ The covariance matrix of fitting results, describing the fit of data to the model. See algorithm for definition.

$chisq$ Chi-square of fitting.

Description

Given a set of data points in arrays x and y with individual standard deviations in array sig , use χ^2 minimization to fit for some or all of the coefficients in array a of a function that depends linearly on, $y = a_i * funcs_i(x)$. The fitted coefficients are passed by array a . The program can also pass χ^2 and covariance matrix $covar$. If the values for array argument ia are constants, the corresponding components of the covariance matrix will be zero. The user supplies a routine $funcs(x, func)$ that passes the function values evaluated at x by the array $func$.

Algorithm

The general form of the fitting formula is

$$y(x) = \sum_{k=1}^M a_k f_k(x)$$

where $f_1(x), \dots, f_M(x)$ are arbitrary fixed function of x , called base functions. $f_k(x)$ can be a nonlinear function of x . The coefficients a_k are determined by minimizing chi-square which is defined as

$$\chi^2 = \sum_{i=1}^N \left(\frac{y_i - \sum_{k=1}^M a_k f_k(x_i)}{\sigma_i} \right)^2$$

where σ_i is the standard deviation of the i th data point. If the standard deviations are unknown, they can be set to the constant value $\sigma = 1$. By defining a $n \times m$ matrix α with element (k, j) as

$$\alpha_{kj} = \sum_{i=1}^N \frac{f_j(x_i) f_k(x_i)}{\sigma_i^2}$$

The element (k, j) of the covariance matrix can be expressed as

$$cov_{kj} = [\alpha]_{kj}^{-1}$$

Example 1

Fitting of the data points generated by $f(x) = 4 \cos(x) + 3 \sin(x) + 2e^x + 1$ with uniform random deviation to the base functions $\cos(x)$, $\sin(x)$, e^x and 1. This example uses **curvfit**($a, x, y, funcs$) only.

```
#include <stdio.h>
#include <numeric.h>
#define NPT 100
#define SPREAD 0.1
#define NTERM 4

void funcs(double x, double func[]) {
    func[0]=cos(x);
    func[1]=sin(x);
    func[2]=exp(x);
    func[3]=1.0;
}
int main(void) {
    int i, j;
    array double a[NTERM], x[NPT], y[NPT], sig[NPT], u[NPT];

    linspace(x, 0, 10);
    urand(u);
    y = 4*cos(x) + 3*sin(x) + 2*exp(x) + (array double [NPT])1.0 + SPREAD*u;
    curvfit(a, x, y, funcs);
    printf("\n%11s\n", "parameters");
    for (i=0; i<NTERM; i++)
        printf(" a[%1d] = %f\n", i, a[i]);
}
```

Output1

```

parameters
a[0] = 3.990255
a[1] = 2.994660
a[2] = 2.000000
a[3] = 1.051525

```

Example 2

Fitting of the data points generated by $f(x) = 4 \cos(x) + 3 \sin(x) + 2e^x + 1$ with uniform random deviation to the base functions of $\cos(x)$, $\sin(x)$, e^x and 1. This example uses all the options of arguments of function `curvefit()`. The values of 3 and 1 for coefficients $a[2]$ and $a[0]$ are fixed, respectively.

```

#include <stdio.h>
#include <numeric.h>

#define NPT 100
#define SPREAD 0.1
#define NTERM 4

void funcs(double x, double func[]) {
    func[0]=cos(x);
    func[1]=sin(x);
    func[2]=exp(x);
    func[3]=1.0;
}

int main(void) {
    int i,j;
    array int ia[NTERM];
    array double a[NTERM],x[NPT],y[NPT],sig[NPT],covar[NTERM][NTERM],u[NPT];
    double chisq;

    linspace(x, 0, 10);
    urand(u);
    y = 4*cos(x) + 3*sin(x) + 2*exp(x) +(array double [NPT])1.0 + SPREAD*u;
    sig=(array double[NPT]) (SPREAD);

    ia[0] = 0; ia[1] = 1; ia[2] = 0; ia[3] = 1;
    a[0] = 4; a[1] = 3; a[2] = 2; a[3] = 1;

    curvefit(a,x,y,funcs,sig,ia,covar,&chisq);
    printf("\n%11s %21s\n", "parameter", "uncertainty");
    for (i=0;i<NTERM;i++)
        printf(" a[%1d] = %8.6f %12.6f\n",i,a[i],sqrt(covar[i][i]));
    printf("chi-square = %12f\n",chisq);
    printf("full covariance matrix\n");
    printf("%12f",covar);
}

```

Output 2

```

parameter          uncertainty
a[0] = 4.000000    0.000000
a[1] = 2.994009    0.015044
a[2] = 2.000000    0.000000
a[3] = 1.052622    0.010357
chi-square =      7.538080
full covariance matrix
    0.000000    0.000000    0.000000    0.000000

```

```
0.000000    0.000226    0.000000    -0.000041
0.000000    0.000000    0.000000    0.000000
0.000000   -0.000041    0.000000    0.000107
```

See Also

expfit(), linearfit(), polyfit(), quadraticfit(), std().

References

William H. Press, Saul A. Teukolsky, William T. Vetterling and Brian P. Flannery, *Numeric Recipes in C*, Second Edition, Cambridge University Press, 1997.

deconv

Synopsis

```
#include <numeric.h>
```

```
int deconv(array double complex u[&], array double complex v[&], array double complex q[&], ...
          /* [array double complex r[&]] */);
```

Syntax

```
deconv(u, v, q)
```

```
deconv(u, v, q, r)
```

Purpose

One-dimensional Discrete Fourier Transform (DFT) based deconvolution.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

u A one-dimensional array of size *n*. It contains data used for deconvolution.

v A one-dimensional array of size *m*. It contains data used for deconvolution.

q A one-dimensional array of size $(n - m + 1)$. It contains the result of deconvolution of *u* and *v*.

r A one-dimensional array of size *n*. It contains the remainder item if user specified.

Description

The input arrays *u* and *v* can be of any supported arithmetic data type and sizes *n* and *m*. Conversion of the data to **double complex** type is performed internally. If both *u* and *v* are **real** data type, the deconvolution result is a one-dimensional **real array** *q* of size $(n + m - 1)$ and the remaining item is a **real array** *r* if specified. If either one of *u* and *v* is a **complex** type, then the result **array** *q* and remained item *r* will be a **complex** data type.

If *u* and *v* are vectors of polynomial coefficients, deconvolution of *u* and *v* is equivalent to the division the two polynomials.

Algorithm

According to the theorem of convolution, if $X(f)$ and $Y(f)$ are Fourier transforms of $x(t)$ and $y(t)$, that is,

$$x(t) \iff X(f) \quad \text{and} \quad y(t) \iff Y(f)$$

then

$$x * y \iff X(f)Y(f)$$

The theorem of convolution of discrete convolution is described as, if $X_d(f)$ and $Y_d(f)$ are discrete Fourier transforms of sequences $x[nT]$ and $y[nT]$, that is,

$$x[nT] \iff X_d(f) \quad \text{and} \quad y[nT] \iff Y_d(f)$$

then

$$x[nT] * y[nT] \iff X_f(f)Y_d(f)$$

where T is the sample interval and n is an index used in the summation.

In function **deconv()**, the convolution result u is given and one sequence of convolution data v is known. Based on the theorem of discrete convolution, DFT can be used to compute discrete deconvolution. The procedure is first to compute the DFT of $u[nT]$ and $v[nT]$ and then use the theorem to compute

$$q[nT] + r[nT] = F_d^{-1}[U_d(f)]/F_d^{-1}[V_d(f)]$$

where $U_d(f)$ and $V_d(f)$ are DFT of sequences $u[nT]$ and $v[nT]$, respectively. In the algorithm the long division of two polynomial is used to get the result. The quotient is the convolution sequence $q[nT]$ and the remainder is $r[nT]$.

Example

There are two polynomial U and V whose coefficients u and v are $[6, 19, 32, 35, 58, 71, 42]$ and $[6, 7]$ respectively. The deconvolution of u and v is equivalent to the division of polynomial U by polynomial V .

$$\begin{aligned} U &= 6 * t^6 + 19 * t^5 + 32 * t^4 + 45 * t^3 + 58 * t^2 + 71 * t + 42 \\ V &= 6 * t + 7 \end{aligned}$$

then U/V equals

$$Q = t^5 + 2 * t^4 + 3 * t^3 + 4 * t^2 + 5 * t + 6$$

```
#include <stdio.h>
#include <numeric.h>

#define N 7          /* u array size */
#define M 2          /* response function dimension */
#define N2 N-M+1

int main() {
    int i, j, n, m;
    array double u[N] = {6, 19, 32, 45, 58, 71, 42},
                v[M] = {6, 7};
    array double q[N2], r[N];

    printf("u=%f\n", u);
    printf("v=%f\n", v);

    deconv(u, v, q); /* real v, not specify remaind result */
    deconv(u, v, q, r); /* real v, specified remaind result */

    printf("q=%f\n", q);
    printf("r=%f\n", r);
}
```

Output

```
u=6.000000 19.000000 32.000000 45.000000 58.000000 71.000000 42.000000
v=6.000000 7.000000
```



```
q=1.000000 2.000000 3.000000 4.000000 5.000000 6.000000  
r=0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 -0.000000
```

See Also

conv(), **conv2()**, **filter()**.

References

Nussbaumer, H. J, *Fast Fourier Transform and Convolution Algorithms*, New York: Springer-Verlay, 1982.

derivative

Synopsis

```
#include <numeric.h>
```

```
double derivative(double (*func)(double), double x, ... /* [double &err], [double h] */);
```

Syntax

```
derivative(func, x)
```

```
derivative(func, x, &err)
```

```
derivative(func, x, &err, h)
```

Purpose

Calculate numerical derivative of a function at a given point.

Return Value

This function returns a derivative value of function *func* at point *x* in **double** data type.

Parameters

func The function taking derivative.

x A point at which the derivative is evaluated.

err An estimated error in the derivative will return as *err* if this argument is passed.

h An initial estimated step size. It does not have to be small, but it should be an increment in *x* over which *func* changes substantially. If the user does not specify this value, the value of $0.02 * x$ or 0.0001 (if $x < 0.0001$) is used by default.

Description

The function taken derivative, *func*, is specified as a pointer to a function that takes a **double** as an argument and returns a **double** value at *x*. The returned value is the derivative of the function at point *x*. The optional argument *err* contains the estimate of the error in the calculation of derivative. It helps the user to estimate the result. If the optional argument *h* is given, calculation of the derivative uses the initial step size of *h*. Otherwise it uses the default value.

Algorithm

The symmetrized form of the definition of the derivate is defined as the limit

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

when $h \rightarrow 0$. For this formula, the truncation and roundoff errors are $e_t \sim h^2 f'''$ and $e_r \sim \epsilon_f |f(x)/h|$, respectively. The optimal choice of *h* is

$$h \sim \left(\frac{\epsilon_f f}{f'''}\right)^{1/3} \sim (\epsilon_f)^{1/3} x_c$$

and the relative error is

$$(e_r + e_t)/|f'| \sim (\epsilon_f)^{2/3} f^{2/3} (f''')^{1/3} / f' \sim (\epsilon_f)^{2/3}$$

Example

Calculate the derivative of function $x \sin(x)$ at point $x = 2.5$

$$\frac{d}{dx}x \sin(x)|_{x=2.5} = \{\sin(x) - x \cos(x)\}|_{x=2.5} = -1.404387$$

```
#include <stdio.h>
#include <math.h>
#include <numeric.h>

double func(double x) {
    return x*sin(x);
}

int main() {
    double yprime, func(double), x, h, err;

    x = 2.5;
    yprime = derivative(func, x);
    /* derivative of func = x*sin(x) is sin(x)+x*cos(x) */
    printf("yprime = %f\n", sin(x)+x*cos(x));
    printf("derivative(func, x) = %f\n", yprime);
    yprime = derivative(func, x, &err);
    printf("derivative(func, x, &err) = %f, err = %f\n", yprime, err);
    h = 0.1; // default h = 0.001
    yprime = derivative(func, x, &err, h);
    printf("derivative(func, x, &err, h) = %f, err = %f\n", yprime, err);
}
```

Output

```
yprime = -1.404387
derivative(func, x) = -1.404387
derivative(func, x, &err) = -1.404387, err = 0.000000
derivative(func, x, &err, h) = -1.404387, err = 0.000000
```

see also

derivatives(), **integral1()**, **integral2()**, **integral3()**, **integration2()**, **integration3()**.

References

Ridder, C. J. F., *Advances in Engineering Software*, 1982, vol. 4, no. 2, pp. 75-76. [2].

derivatives

Synopsis

```
#include <numeric.h>
```

```
array double derivatives(double (*func)(double), double x[&], ... /* [double &err], [double h] */)[:];
```

Syntax

```
derivative(func, x)
```

```
derivative(func, x, &err)
```

```
derivative(func, x, &err, h)
```

Purpose

Calculate derivatives of a function numerically at multiple points.

Return Value

This function returns an **array** of derivative values of function *func* at points *x[&]*.

Parameters

func Function its derivatives are calculated.

x A series of points in which derivatives are calculated.

err An estimated error in the derivative which will return as *err* if the user specified.

h An initial estimated step size. It does not have to be small, but rather should be an increment in *x* over which *func* changes substantially. If the user do not specify this value, the value of $0.02 * x$ or 0.0001 (if $x < 0.0001$) is used by default.

Description

The function taking derivative, *func*, is specified as a pointer to a function that takes an **array** of **double** data as an argument and returns an **array double** values at *x[&]*. The values of *x* can be of any real type. Conversion of the data to **double** is performed internally. The return value is an **array** of derivatives of the function at points specified by **array** *x*. If the optional argument *h* is specified, which shall be **double** data type, the derivative uses initial step size of *h*. Otherwise it uses the default value. Optional argument *err* shall be a **double** data and it means the estimate of the error in the derivate. This helps the user to estimate the result.

Algorithm

Using a for-loop to evaluate the derivative values specified in **array** *x* and in the loop using the same algorithm as **derivative()**.

Example

Take the derivatives of function

$$f(x) = x * \sin(x) * \cos(x)$$

at 36 different positions meanly spaced between 0 and 2π .
Analytically, the first derivative of this function is

$$f'(x) = 0.5 * \sin(2x) + x \cos(2x)$$

```
#include <stdio.h>
#include <math.h>
#include <chplot.h>
#include <numeric.h>

#define N 36

double func(double x) {
    return x*sin(x)*cos(x);
}

int main() {
    array double x[N], y[N], yprime[N];
    double err, h;
    int i;
    class CPlot plot;

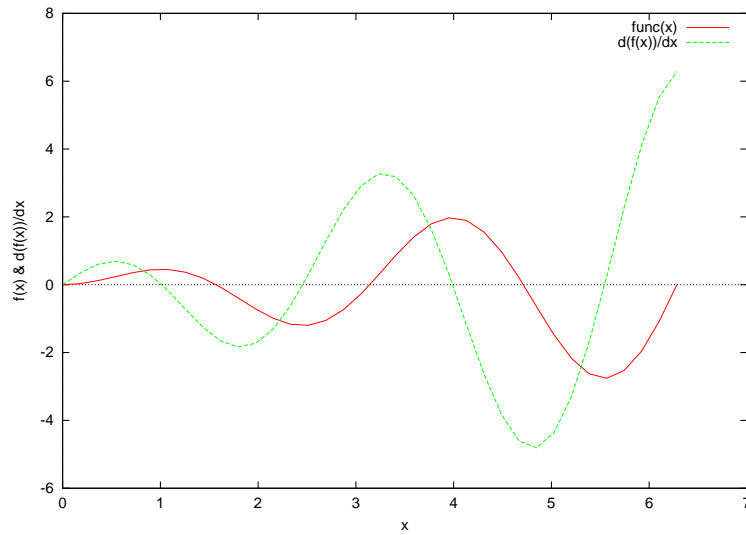
    linspace(x, 0, 2*M_PI);
    printf("      x      derivative() value      Cal. value\n");
    // /* Specify the initial step h and return estimate error */
    // yprime =derivatives(func,x,&err,0.8);
    yprime =derivatives(func,x); /* use default step value and do not return err */

    fevalarray(y,func,x);

    for (i=0; i<2; i++)
        printf("%12.6f      %12.6f      %12.6f\n",x[i],yprime[i],
            (0.5*sin(2*x[i])+x[i]*cos(2*x[i])));
    printf("      -----      -----      -----\n");
    for (i=34; i<36; i++)
        printf("%12.6f      %12.6f      %12.6f\n",x[i],yprime[i],
            (0.5*sin(2*x[i])+x[i]*cos(2*x[i])));
    plot.label(PLOT_AXIS_X, "x");
    plot.label(PLOT_AXIS_Y, "f(x) & d(f(x))/dx");
    plot.data2D(x,y);
    plot.data2D(x,yprime);
    plot.legend("f(x)",0);
    plot.legend("d(f(x))/dx",1);
    plot.plotting();
}
```

Output

x	derivative() value	Cal. value
0.000000	0.000000	0.000000
0.179520	0.343760	0.343760
-----	-----	-----
6.103666	5.538781	5.538781
6.283186	6.283187	6.283187



see also

`derivative()`, `integral1()`, `integral2()`, `integral3()`, `integration2()`, `integration3()`.

References

Ridder, C. J. F., *Advances in Engineering Software*, 1982, vol. 4, no. 2, pp. 75-76.

determinant

Synopsis

```
#include <numeric.h>
```

```
double determinant(array double a[&][&]);
```

Purpose

Calculate the determinant of a matrix.

Return Value

This function returns the determinant of matrix *a*.

Parameters

a The input matrix.

Description

The function **determinant()** returns the determinant of matrix *a*. If the input matrix is not a square matrix, the function will return NaN.

Example

Calculate the determinant of matrices with different data type.

```
#include <numeric.h>
int main() {
    array double a[2][2] = {2, 4,
                           3, 7};
    /* a2 is an ill-detection matrix */
    array double a2[2][2] = {2, 4,
                             2.001, 4.0001};

    /* a3 is singular */
    array double a3[2][2] = {2, 4,
                             4, 8};

    array float b[2][2] = {2, 4,
                          3, 7};

    array double c[3][3] = {-1, 5, 6,
                            3, -6, 1,
                            6, 8, 9} ; /* n-by-n matrix */

    array double d[3][3] = {2, 1, -2,
                          4, -1, 2,
                          2, -1, 1} ; /* n-by-n matrix */

    double det;

    det = determinant(a);
    printf("determinant(a) = %g\n", det);
    det = determinant(a2);
    printf("determinant(a2) = %g\n", det);
    det = determinant(a3);
    printf("determinant(a3) = %g\n", det);
    det = determinant(b);
    printf("determinant(b) = %g\n", det);
    det = determinant(c);
    printf("determinant(c) = %g\n", det);
    det = determinant(d);
```

```
    printf("determinant(d) = %g\n", det);  
}
```

Output

```
determinant(a) = 2  
determinant(a2) = -0.0038  
determinant(a3) = -0  
determinant(b) = 2  
determinant(c) = 317  
determinant(d) = 6
```

See Also

cdeterminant(), **inverse()**, **diagonal()**, **ludcomp()**, **rcondnum()**.

References

E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

diagonal

Synopsis

```
#include <numeric.h>
```

```
array double diagonal(array double a[&][&], ... /* [int k] */)[:];
```

Syntax

```
diagonal(a);
```

```
diagonal(a, k);
```

Purpose

Form a vector with diagonals of a matrix.

Return Value

This function returns a column vector formed from the elements of the k th diagonal of the matrix a . By default, k is 0.

Parameters

a An input matrix.

k An input integer indicating which element the vector is formed from.

Description

The function `diagonal()` produces a column vector formed from the elements of the k th diagonal of the matrix a .

Example1

```
#include <numeric.h>
int main() {
    array double a[3][3] = {1, 2, 3,
                           4, 5, 6,
                           7, 8, 9};

    int n =3, k = 1;
    array double d[n], d2[n-(abs(k))];

    d = diagonal(a);
    printf("diagonal(a) = %f\n", d);

    d2 = diagonal(a, k);
    printf("diagonal(a, 1) = %f\n", d2);

    k = -1;
    d2 = diagonal(a, k);
    printf("diagonal(a, -1) = %f\n", d2);
}
```

Output1

```
diagonal(a) = 1.000000 5.000000 9.000000
```

```
diagonal(a, 1) = 2.000000 6.000000
```

```
diagonal(a, -1) = 4.000000 8.000000
```

Example2

```
#include <numeric.h>
int main() {
    array double a[4][3] = {1, 2, 3,
                           4, 5, 6,
                           7, 8, 9,
                           4, 4, 4};

    int m=4, n = 3, k1 = 1, k2 = -1;
    array double d[min(m, n)];
    array double d1[min(m, n-k1)];
    array double d2[min(m+k2, n)];

    d = diagonal(a);
    printf("diagonal(a) = %f\n", d);

    d1 = diagonal(a, k1);
    printf("diagonal(a, 1) = %f\n", d1);

    d2 = diagonal(a, k2);
    printf("diagonal(a, -1) = %f\n", d2);
}
```

Output2

```
diagonal(a) = 1.000000 5.000000 9.000000

diagonal(a, 1) = 2.000000 6.000000

diagonal(a, -1) = 4.000000 8.000000 4.000000
```

Example3

```
#include <numeric.h>
int main() {
    array double a[3][4] = {1, 2, 3, 4,
                           5, 6, 7, 8,
                           9, 4, 4, 4};

    int m=3, n = 4, k1 = 1, k2 = -1;
    array double d[min(m, n)];
    array double d1[min(m, n-k1)];
    array double d2[min(m+k2, n)];

    d = diagonal(a);
    printf("diagonal(a) = %f\n", d);

    d1 = diagonal(a, k1);
    printf("diagonal(a, 1) = %f\n", d1);

    d2 = diagonal(a, k2);
    printf("diagonal(a, -1) = %f\n", d2);
}
```

Output3

```
diagonal(a) = 1.000000 6.000000 4.000000

diagonal(a, 1) = 2.000000 7.000000 4.000000

diagonal(a, -1) = 5.000000 4.000000
```

See Also

diagonalmatrix().

cdiagonal(), cdiagonalmatrix().

References

diagonalmatrix

Synopsis

```
#include <numeric.h>
```

```
array double diagonalmatrix(array double v[&], ... /* [int k] */)[:][:];
```

Syntax

```
diagonalmatrix(v)
```

```
diagonalmatrix(v, k)
```

Purpose

Form a diagonal matrix.

Return Value

This function returns a square matrix with specified diagonal elements.

Parameters

v An input vector.

k An input integer indicating specified diagonal elements of the matrix.

Description

The function **diagonalmatrix()** returns a square matrix of order $n + \text{abs}(k)$, with the elements of *v* on the *k*th diagonal. $k = 0$ represents the main diagonal, $k > 0$ above the main diagonal, and $k < 0$ below the main diagonal.

Example

```
#include <numeric.h>
int main() {
    array double v[3] = {1,2,3};
    int n = 3, k;
    array double a[n][n];

    a = diagonalmatrix(v);
    printf("diagonal matrix a =\n%f\n", a);

    k = 0;
    a = diagonalmatrix(v,k);
    printf("diagonalmatrix(a, 0) =\n%f\n", a);

    k = 1;
    array double a2[n+abs(k)][n+abs(k)];
    a2 = diagonalmatrix(v,k);
    printf("diagonalmatrix(a2, 1) =\n%f\n", a2);

    k = -1;
    array double a3[n+abs(k)][n+abs(k)];
    a3 = diagonalmatrix(v,k);
    printf("diagonalmatrix(a3, -1) =\n%f\n", a3);
}
```

Output

```
diagonal matrix a =  
1.000000 0.000000 0.000000  
0.000000 2.000000 0.000000  
0.000000 0.000000 3.000000
```

```
diagonalmatrix(a, 0) =  
1.000000 0.000000 0.000000  
0.000000 2.000000 0.000000  
0.000000 0.000000 3.000000
```

```
diagonalmatrix(a2, 1) =  
0.000000 1.000000 0.000000 0.000000  
0.000000 0.000000 2.000000 0.000000  
0.000000 0.000000 0.000000 3.000000  
0.000000 0.000000 0.000000 0.000000
```

```
diagonalmatrix(a3, -1) =  
0.000000 0.000000 0.000000 0.000000  
1.000000 0.000000 0.000000 0.000000  
0.000000 2.000000 0.000000 0.000000  
0.000000 0.000000 3.000000 0.000000
```

See Also

cdiagonalmatrix(), diagonal() cdiagonal().

References

difference

Synopsis

```
#include <numeric.h>
```

```
array double difference(array double a[&])[:];
```

Purpose

Calculate differences between the adjacent elements of an array.

Return Value

This function returns the array of differences of adjacent elements of array *a*.

Parameters

a The input array.

Description

The function **difference()** calculates the differences of adjacent elements of an array. The input array can be of any arithmetic data types.

Example

Calculate the differences of the adjacent elements of the arrays with different data types.

```
#include <numeric.h>
#define N 6
int main() {
    array double a[N] = {1, 2, 10, 4, 5, 6};
    array float f[N] = {1, 2, 10, 4, 5, 6};
    array double d[N-1];

    d = difference(a);
    printf("difference(a) = %f\n", d);
    d = difference(f);
    printf("difference(f) = %f\n", d);
}
```

Output

```
difference(a) = 1.000000 8.000000 -6.000000 1.000000 1.000000
```

```
difference(f) = 1.000000 8.000000 -6.000000 1.000000 1.000000
```

See Also

derivative(), **derivatives()**, **sum()**, **product()**.

References

divisor()

Synopsis

```
#include <numeric.h>
array int divisor(int num)[:];
```

Purpose

Calculate the divisors of an integer.

Return Value

This function returns an array of divisors of an integer. If the integer is less or equal to 0, it returns an array of one element with the value of -1.

Parameter

num An integer.

Description

This function returns the divisors in an array for the integer *num*. If the integer is less or equal to 0, it returns an array of one element with the value of -1.

Example

See `gcd()`.

See Also

`gcd()`, `divisornum()`, `cdivisor()`, `cdivisornum()`.

divisornum()

Synopsis

```
#include <numeric.h>
int divisornum(int num);
```

Purpose

Calculate the number of divisors of an integer.

Return Value

This function returns the number of divisors of an integer. If the integer is less or equal to 0, it returns -1.

Parameter

num An integer.

Description

This function returns the number of divisors for the integer *num*. If the integer is less or equal to 0, it returns -1.

Example

```
/* File: divisornum.ch */
#include <numeric.h>
int num = 54;
int n;

n = divisornum(num);
array int div[n];
div = divisor(num);
printf("The number of divisors of %d is %d\n", num, n);
printf("The divisors of %d are\n%d", num, div);

n = cdivisornum(54, 24);
array int cdiv[n];
cdiv = cdivisor(54, 24);
printf("The number of common divisors for 54 and 24 is %d\n", n);
printf("The common divisors of 54 and 24 are\n%d", cdiv);
```

Output

```
The number of divisors of 54 is 8
The divisors of 54 are
1 2 3 6 9 18 27 54
The number of common divisors for 54 and 24 is 4
The common divisors of are
1 2 3 6
```

See Also

gcd(), **divisor()**, **cdivisor()**, **cdivisornum()**.

dot

Synopsis

```
#include <numeric.h>
```

```
double dot(array double a[&], array double b[&]);
```

Purpose

Calculate the dot product of a vector.

Return Value

This function returns the dot product of two vectors.

Parameters

a An input one-dimensional array.

b Another input one-dimensional array.

Description

The function calculates the dot product of two vectors. The number of elements of two vectors should be the same. Otherwise the function returns NaN.

Example

```
#include <numeric.h>
int main() {
    array double a[4] = {1, 2, 3, 4};
    array double b[ ] = {1, 2, 3, 4};
    double dotprod;

    dotprod = dot(a, b);
    printf("dot(a,b) = %f\n", dotprod);
}
```

Output

```
dot(a,b) = 30.000000
```

See Also

`cross()`.

References

eigen

Synopsis

```
#include <numeric.h>
```

```
int eigen(... /* array double [complex] a[:][:], array double [complex] values[:],
              array double [complex] evecors[:][:], string_t mode */);
```

Syntax

```
eigen(a, values); eigen(a, values, evecors); eigen(a, values, evecors, mode);
```

Purpose

Find eigenvalues and eigenvectors.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

a Input square matrix for which the eigenvalues and eigenvectors are calculated.

values Output one-dimensional array which contains the calculated eigenvalues.

evecors Output square matrix which contains the calculated eigenvectors.

mode The optional **string_t mode** used to specify if using a preliminary balancing step before the calculation of eigenvalues and eigenvectors.

Description

This function calculates the eigenvalues and eigenvectors of n-by-n matrix *a* so that $a*v = \lambda*v$, where λ and *v* are the vector of eigenvalues and eigenvectors, respectively. The computed eigenvectors are normalized so that the norm of each eigenvector equals 1. *mode* is used to specify if a preliminary balancing step before the calculation is performed or not. Generally, balancing improves the condition of the input matrix, enabling more accurate computation of the eigenvalues and eigenvectors. But, it also may lead to incorrect eigenvectors in special cases. By default, a preliminary balancing step is taken.

Example1

This example calculates the eigenvalues and eigenvectors of a non-symmetrical matrix. The eigenvalues of this matrix is real number. The formula $a * v - values * v$ is calculated to verify the results.

```
#include <numeric.h>
#define N 5

int main() {
    array double a[5][5] = {2,      30,    100,  1000, 10000,
                          0.3,    1,     10,   100,  1000,
                          0.04,   0.1,   1,    10,   100,
                          0.001,  0.01,  0.1,   1,    10,
                          0.0002, 0.003, 0.01, 0.1,  1};

    array double values[N];
    array double evecors[N][N];
    array double *v, vv[N][N];
    int i, status;

    eigen(a, values, evecors);
    printf("eigenvalues of a =\n%e\n", values);
```

```

printf("eigenvectors of a =\n%e\n", evectors);

eigen(a, values, evectors, "nobalance");
printf("no balance: eigennvalues of a =\n%e\n", values);
printf("no balance: eigenvectors of a =\n%e\n", evectors);

vv = transpose(evectors);
for(i=0; i<N; i++) {
    v = (array double [:]) (double [N])&vv[i][0];
    printf("A*v-evalues[i]*v = %f\n", a*v-evalues[i]*v);
}
}

```

Output1

```

eigennvalues of a =
7.327477e+00 -1.508424e+00 1.809473e-01 -1.555781e-15 2.032652e-16

eigenvectors of a =
-9.959858e-01 -9.945522e-01 -9.966546e-01 -7.564396e-14 4.879875e-14
-8.892060e-02 1.028639e-01 -5.396722e-02 -9.891903e-15 -2.815928e-14
-1.025131e-02 1.687971e-02 -6.047655e-02 9.948856e-01 8.268104e-01
-6.173564e-04 -2.900250e-04 1.047629e-02 -1.010080e-01 5.588107e-01
-9.959858e-05 -9.945522e-05 -9.966546e-05 1.519397e-04 -6.414918e-02

no balance: eigennvalues of a =
7.327477e+00 1.809473e-01 -1.508424e+00 1.227397e-15 -4.649855e-16

no balance: eigenvectors of a =
9.959858e-01 9.966546e-01 -9.945522e-01 1.586027e-13 -2.911653e-16
8.892060e-02 5.396722e-02 1.028639e-01 2.746367e-15 -1.029631e-14
1.025131e-02 6.047655e-02 1.687971e-02 9.842796e-01 9.714797e-01
6.173564e-04 -1.047629e-02 -2.900250e-04 -1.764452e-01 -2.367118e-01
9.959858e-05 9.966546e-05 -9.945522e-05 7.801720e-03 1.395638e-02

A*v-evalues[i]*v = -0.000000 0.000000 0.000000 0.000000 -0.000000

A*v-evalues[i]*v = 0.000000 0.000000 0.000000 0.000000 -0.000000

A*v-evalues[i]*v = 0.000000 0.000000 0.000000 0.000000 0.000000

A*v-evalues[i]*v = 0.000000 0.000000 0.000000 0.000000 0.000000

A*v-evalues[i]*v = -0.000000 0.000000 0.000000 0.000000 -0.000000

```

Example2

The eigenvalues and eigenvectors of symmetrical and non-symmetrical matrices are calculated. The symmetrical matrix has real eigenvalues and a non-symmetrical matrix has real or complex eigenvalues.

```

/* File: eigen_1.ch
   Find eigenvalues and eigenvectors */
#include <numeric.h>
int main() {
    /* eigenvalues of a symmetrical matrix are always real numbers */
    array double a[3][3] = {0.8, 0.2, 0.1,
                           0.2, 0.7, 0.3,
                           0.1, 0.3, 0.6};
}

```

```

/* eigenvalues of a non-symmetrical matrix can be either real numbers or
   complex numbers */
/* this non-symmetrical matrix has real eigenvalues */
array double a2[3][3] = {0.8, 0.2, 0.1,
                        0.1, 0.7, 0.3,
                        0.1, 0.1, 0.6};
/* this non-symmetrical matrix has complex eigenvalues and eigenvectors*/
array double a3[3][3] = {3, 9, 23,
                        2, 2, 1,
                        -7, 1, -9};
array double evalues[3],eectors[3][3];
array double complex zevalues[3], zevectors[3][3];

eigen(a, evalues);
printf("evalues =\n%f\n", evalues);
eigen(a2, evalues);
printf("evalues =\n%f\n", evalues);
eigen(a2, zevalues, eectors);
printf("zevalues =\n%f\n", zevalues);
printf("eectors =\n%f\n", eectors);
eigen(a3, evalues);
printf("evalues =\n%f\n", evalues);
eigen(a3, zevalues, zevectors);
printf("evalues =\n%f\n", zevalues);
printf("eectors =\n%f\n", zevectors);
}

```

Output2

```

evalues =
1.108807 0.652624 0.338569

evalues =
1.000000 0.600000 0.500000

zevalues =
complex(1.000000,0.000000) complex(0.600000,0.000000) complex(0.500000,0.000000)

eectors =
-0.744845 -0.707107 0.408248
-0.579324 0.707107 -0.816497
-0.331042 0.000000 0.408248

evalues =
NaN NaN 3.241729

evalues =
complex(-3.620864,10.647303) complex(-3.620864,-10.647303) complex(3.241729,0.000000)

eectors =
complex(0.854461,0.000000) complex(0.854461,0.000000) complex(0.604364,0.000000)
complex(-0.024440,-0.125397) complex(-0.024440,0.125397) complex(0.744064,0.000000)
complex(-0.236405,0.444621) complex(-0.236405,-0.444621) complex(-0.284803,0.000000)

```

Example3

The eigenvalues and eigenvectors of real and complex matrices are calculated in this example.

```
#include <numeric.h>
int main() {
    array double complex a[3][3] = {0.8, 0.2, 0.1,
                                    0.1, 0.7, 0.3,
                                    0.1, 0.1, 0.6};

    array double complex a2[3][3] = {complex(0.8,-1), 0.2, 0.1,
                                    0.1, 0.7, 0.3,
                                    0.1, 0.1, 0.6};
    /* this non-symmtrical matrix has complex eigenvalues and eigenvectors*/
    array double complex a3[3][3] = {3, 9, 23,
                                    2, 2, 1,
                                    -7, 1, -9};

    array double complex evalues[3];
    array double complex evectors[3][3];

    eigen(a, evalues, evectors);
    printf("eigenvalues of a =\n%f\n", evalues);
    printf("eigenvectors of a =\n%f\n", evectors);

    eigen(a2, evalues, evectors);
    printf("eigenvalues of a =\n%f\n", evalues);
    printf("eigenvectors of a =\n%f\n", evectors);

    eigen(a3, evalues, evectors);
    printf("eigenvalues of a =\n%f\n", evalues);
    printf("eigenvectors of a =\n%f\n", evectors);
}
```

Output3

```
eigenvalues of a =
complex(1.000000,0.000000) complex(0.600000,0.000000) complex(0.500000,0.000000)

eigenvectors of a =
complex(0.744845,0.000000) complex(-0.707107,0.000000) complex(-0.408248,0.000000)
complex(0.579324,0.000000) complex(0.707107,0.000000) complex(0.816497,0.000000)
complex(0.331042,0.000000) complex(0.000000,0.000000) complex(-0.408248,0.000000)

eigenvalues of a =
complex(0.796911,-0.968453) complex(0.832118,-0.037024) complex(0.470970,0.005477)

eigenvectors of a =
complex(0.989164,0.000000) complex(0.003912,-0.225655) complex(-0.025122,-0.090331)
complex(-0.019749,0.106886) complex(0.891481,0.000000) complex(0.794571,0.000000)
complex(0.008946,0.098280) complex(0.391299,-0.034802) complex(-0.598227,0.044616)
```

```
eigenvalues of a =  
complex(-3.620864,10.647303) complex(-3.620864,-10.647303) complex(3.241729,-0.000000)
```

```
eigenvectors of a =  
complex(0.854461,0.000000) complex(0.854461,0.000000) complex(0.604364,0.000000)  
complex(-0.024440,-0.125397) complex(-0.024440,0.125397) complex(0.744064,0.000000)  
complex(-0.236405,0.444621) complex(-0.236405,-0.444621) complex(-0.284803,-0.000000)
```

See Also

balance(), **choldecomp()**, **hessdecomp()**.

References

E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

eigensystem

Synopsis

```
#include <numeric.h>
```

```
int eigensystem(array double complex values[&], array double complex evecs[&][&],  
                array double complex a[&][&], ... /* [string_t mode] */);
```

Syntax

```
eigensystem(values, NULL, a); eigensystem(values, evecs, a);
```

Purpose

Find eigenvalues and eigenvectors. This function is obsolete. Use function **eigen()**.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

values Output one-dimensional array which contains the calculated eigenvalues.

evecs Output square matrix which contains the calculated eigenvectors.

a Input square matrix for which the eigenvalues and eigenvectors are calculated.

mode The optional **string_t mode** used to specify if using a preliminary balancing step before the calculation of eigenvalues and eigenvectors.

Description

This function calculates the eigenvalues and eigenvectors of n-by-n matrix *a* so that $a * v = \lambda * v$, where λ and *v* are the vector of eigenvalues and eigenvectors, respectively. The computed eigenvectors are normalized so that the norm of each eigenvector equals 1. *mode* is used to specify if a preliminary balancing step before the calculation is performed or not. Generally, balancing improves the condition of the input matrix, enabling more accurate computation of the eigenvalues and eigenvectors. But, it also may lead to incorrect eigenvectors in special cases. By default, a preliminary balancing step is taken.

See Also

eigen().

References

E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

expm

Synopsis

```
#include <numeric.h>
```

```
int expm(array double complex y[&][&], array double complex x[&][&]);
```

Syntax

```
expm(y, x)
```

Purpose

Computes the matrix exponential.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x Input square matrix. It contains data to be calculated.

y Output square matrix which contains data of the result of *x* exponential.

Description

This function computes the matrix exponential of the input matrix *x* using a scaling and squaring algorithm with a Pade approximation. If any of the eigenvalues of input matrix *x* are positive, the output matrix *y* is a real matrix. If *x* has nonpositive eigenvalues, the results *y* shall be a complex matrix.

Example

Calculation of exponentials of real and complex matrix.

```
#include <numeric.h>
int main() {
    array double x[3][3]={1,2,3,
                          3,4,5,
                          6,7,8};
    array double complex zx[3][3]={complex(1,1),complex(2,2),0,
                                   3,complex(4,1),complex(2,5),
                                   0,0,0};
    array double complex zy[3][3];
    array double y[3][3];

    expm(y, x);
    printf("x = \n%f", x);
    printf("y = \n%f", y);
    printf("\n");

    expm(zy, zx);
    printf("zx = \n%5.1f", zx);
    printf("zy = \n%5.1f", zy);
}
```

Output


```
x =
1.000000 2.000000 3.000000
3.000000 4.000000 5.000000
6.000000 7.000000 8.000000
y =
157372.953093 200034.605129 242697.257164
290995.910241 369883.552084 448769.193928
491431.845963 624654.472518 757878.099072

zx =
complex( 1.0, 1.0) complex( 2.0, 2.0) complex( 0.0, 0.0)
complex( 3.0, 0.0) complex( 4.0, 1.0) complex( 2.0, 5.0)
complex( 0.0, 0.0) complex( 0.0, 0.0) complex( 0.0, 0.0)
zy =
complex(-44.9, 56.8) complex(-87.5, 70.9) complex(-101.5,-18.9)
complex(-12.5,118.7) complex(-57.4,175.5) complex(-155.5, 67.8)
complex( 0.0, 0.0) complex( 0.0, 0.0) complex( 1.0, 0.0)
```

See Also

logm(), funm(), cfunm(), sqrtm().

References

G. H. Golub, C. F. Van Loan, *Matrix Computations* Third edition, The Johns Hopkins University Press, 1996

expfit

Synopsis

```
#include <numeric.h>
```

```
int expfit(double x[&], double y[&], double &a, double &b, double r[:]);
```

Purpose

Fit a set of data points x, y to an exponential function $y = ab^x$.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x An array which contains variable values of the data set.

y An array which contains function values of the data set.

a The constant a for $y = ab^x$.

b The constant b for $y = ab^x$.

r The residual error for the exponential fitting.

Description

Given a set of data points expressed in arrays x and y with the same dimension, fit the data set using the χ^2 minimization. The fitted coefficients are passed by arguments a and b .

Algorithm

Use the χ^2 minimization algorithm to find a and b for $y = ab^x$.

Example 1

```

/* File: expfit.ch
   Find the exponential fit for the growth of bacteria */
#include <chplot.h>
CPlot plot;
array double t[11] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
array double y[11] = {1, 2, 4, 7, 17, 31, 65, 126, 254, 511, 1025};
array double r[11];
double a, b, t1, y1;

/* solve for a and b in y = a*pow(b, x) */
expfit(t, y, a, b, r);
printf("a = %.2lf\n", a);
printf("b = %.4lf\n", b);
printf("r = %.4lf\n", r);

/* calculate y for t=24 */
t1 = 24;
y1 = a*pow(b, t1);
printf("(t1, y1) = (%.2lf, %.2lf)\n", t1, y1);

```

```

plot.title("The Growth of Bacteria");
plot.label(PLOT_AXIS_X, "Time (hours)");
plot.label(PLOT_AXIS_Y, "Number of bacteria");
plot.axisRange(PLOT_AXIS_X, 0, 10);
//plot.axisRange(PLOT_AXIS_Y, 0, 1200);
plot.expr(0, 10, 50, "a*pow(b, x)");
plot.scatter(t, y);
plot.plotting();

```

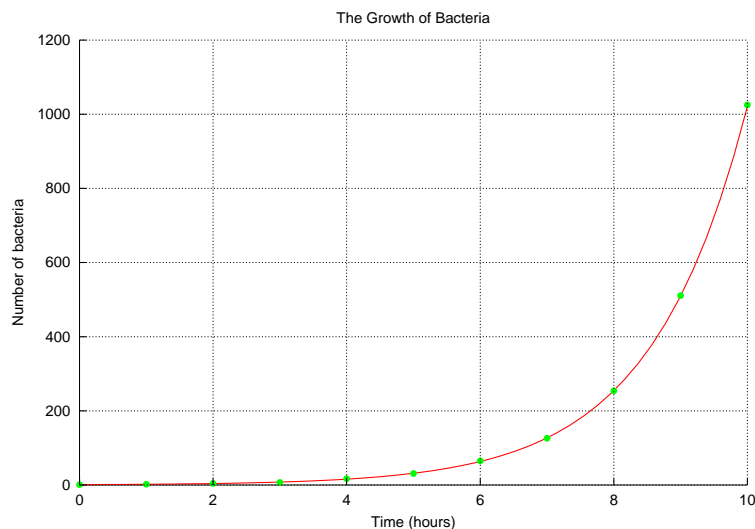
Output

```

a = 0.98
b = 2.0030
r = 0.0176 0.0323 0.0588 -0.8941 1.1881 -0.6709 1.5638 -1.0618 -0.5026 1.2354 3.9498

```

```
(t1, y1) = (24.00, 17081681.44)
```



Example 2

```

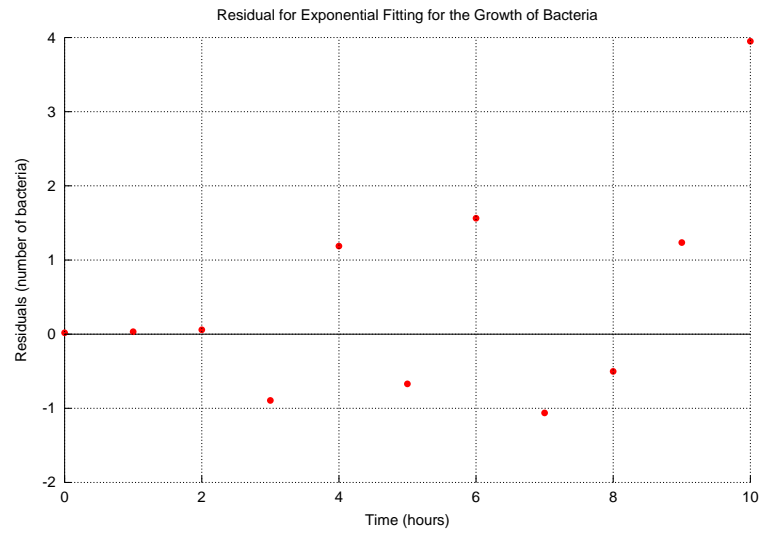
/* File: expfit2.ch
   Find the residuals of an exponential relation between the hours and number of
   bacteria */
#include <chplot.h>
CPlot plot;
array double t[11] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
array double y[11] = {1, 2, 4, 7, 17, 31, 65, 126, 254, 511, 1025};
array double r[11];
double a, b;

/* solve for a and b in y = a*pow(b, x) */
expfit(t, y, a, b, r);
printf("a = %.2lf\n", a);
printf("b = %.4lf\n", b);
printf("r = %.4lf\n", r);

plot.title("Residual for Exponential Fitting for the Growth of Bacteria");
plot.label(PLOT_AXIS_X, "Time (hours)");
plot.label(PLOT_AXIS_Y, "Residuals (number of bacteria)");

```

```
plot.scatter(t, r);  
plot.axisRange(PLOT_AXIS_X, 0, 10);  
//plot.axisRange(PLOT_AXIS_Y, -5, 5);  
plot.plotting();
```

**Output****See Also**

curvefit(), linearfit(), polyfit(), quadraticfit().

factor()

Synopsis

```
#include <numeric.h>
array int factor(int num)[:];
```

Purpose

Calculate the prime factors of an integer.

Return Value

This function returns an array of prime factors of an integer. If the integer is less or equal to 0, it returns an array of one element with the value of -1.

Parameter

num An integer.

Description

This function returns the prime factors in an array for the integer *num*. If the integer is less or equal to 0, it returns an array of one element with the value of -1.

Example

```
/* File: factor.ch */
#include <numeric.h>
int num1 = 54;
int num2 = 24;

printf("The number of prime factors of %d is %d\n", num1, factornum(num1));
printf("The prime factors of %d are\n%d", num1, factor(num1));

int n;
n = factornum(num2);
array int fact[n];
fact = factor(num2);
printf("The number of prime factors of %d is %d\n", num2, n);
printf("The prime factors of %d are\n%d", num2, fact);
```

Output

```
The number of prime factors of 54 is 4
The prime factors of 54 are
2 3 3 3
The number of prime factors of 24 is 4
The prime factors of 24 are
2 2 2 3
```

See Also

divisor(), **cdivisor()**, **factornum()**.

factorial

Synopsis

```
#include <numeric.h>
```

```
unsigned long long factorial(unsigned int n);
```

Purpose

Calculate factorial of an unsigned integer.

Return Value

This function returns factorial of integer n .

Parameters

n Input unsigned integer.

Description

The function **factorial** calculates the product of all the integers from 1 to n . The factorial is defined as

$$f = n!$$

Example

```
#include <numeric.h>
int main() {
    unsigned long long f;

    f = factorial(3);
    printf("factorial(3) = %llu\n", factorial(3));
}
```

Output

```
factorial(3) = 6
```

See Also

combination().

References

factornum()

Synopsis

```
#include <numeric.h>
int factornum(int num);
```

Purpose

Calculate the number of prime factors of an integer.

Return Value

This function returns the number of prime factors of an integer. If the integer is less or equal to 0, it returns -1.

Parameter

num An integer.

Description

This function returns the number of prime factors for the integer *num*. If the integer is less or equal to 0, it returns -1.

Example

See `factor()`.

See Also

`factor()`, `divisornum()`, `cdivisornum()`.

fevalarray

Synopsis

```
#include <numeric.h>
```

```
int fevalarray(array double &y, double (*func)(double x), array double &x, ... /* [array int &mask,
double value] */);
```

Syntax

```
fevalarray(y, func, x)
```

```
fevalarray(y, func, x, mask, value)
```

Purpose

Array function evaluation.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

y An output array which contains the result of function evaluation.

x An input array at which the function values will be evaluated.

func A pointer to function given by the user.

mask An input array of integer. If the value of an element of array *mask* is 1, the function evaluation will be applied to that element. If the value of an element of array is 0, the value from the optional fifth argument *value* will be used for that element.

value The default value used to replace the function evaluation when the entry of *mask* is 0.

Description

This function evaluates a user function applied to each element of the input array *x*. The input array can be of any arithmetic data type and dimension.

Example

Evaluation of function $f(x) = x^2$ for arrays of different dimensions.

```
#include <numeric.h>
#define N 4
#define M 5

double func(double x) {
    return x*x;
}

int main() {
    array double y1[N],    x1[N];
    array float y2[N][M], x2[N][M]; /* diff data types, diff dim */
    array int mask1[N] = {1, 0, 1, 0};

    linspace(x1, 1, N);
    fevalarray(y1, func, x1);
```



```
printf("y1 = %f\n", y1);

linspace(x1, 1, N);
fevalarray(y1, func, x1, mask1, -1);
printf("y1 = %f\n", y1);

linspace(x2, 1, N*M);
fevalarray(y2, func, x2);
printf("y2 = %f\n", y2);
}
```

Output

```
y1 = 1.000000 4.000000 9.000000 16.000000
```

```
y1 = 1.000000 -1.000000 9.000000 -1.000000
```

```
y2 = 1.000000 4.000000 9.000000 16.000000 25.000000
36.000000 49.000000 64.000000 81.000000 100.000000
121.000000 144.000000 169.000000 196.000000 225.000000
256.000000 289.000000 324.000000 361.000000 400.000000
```

See Also

cfevalarray(), **streval()**.

References

fft

Synopsis

```
#include <numeric.h>
```

```
int fft(array double complex &y, array double complex &x, ... /* [int n [int dim[&]]] */);
```

Syntax

```
fft(y, x)
```

```
fft(y, x, n)
```

```
fft(y, x, dim);
```

Purpose

N-dimensional Fast Fourier Transform (FFT) calculation.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

y Result of FFT, the same dimension and size **array** as *x*.

x An n-dimensional array of data used for FFT.

n Optional argument, user specified FFT points for one-dimensional data.

dim A one-dimensional array of optional arguments with **int** type. It contains user specified FFT points. Each element corresponds to data in that dimension. For example, there is a three-dimensional data $x[m][n][l]$. The FFT points of the array is specified by m, n, l . Then the array *dim* is given values of $\text{dim}[0] = m, \text{dim}[1] = n$ and $\text{dim}[2] = l$.

Description

The multi-dimensional (maximum dimension is three) array *x* can be of any supported arithmetic data type and size. Conversion of the data to double complex is performed internally. Array *y* with the same dimension as array *x* contains the result of the fast Fourier transform. The optional argument *n* of int type is used to specify the number of points for FFT of one-dimensional data. For multi-dimensional data, The optional array argument *dim* of int type. contains the value for user specified FFT points. The dimensions of the input array *x* are contained in array *dim*. If no optional argument is passed, the number of FFT points is obtained from the input array *x*.

Algorithm

Mixed-radix Fast Fourier Transform algorithm developed by R. C. Singleton (Stanford Research Institute, Sept, 1968). A discrete Fourier transform is defined by

$$Y(n) = \sum_{j=0}^{N-1} X(j) \exp\left(\frac{-2\pi i j n}{N}\right), (n = 0, 1, \dots, N - 1);$$

and the inverse discrete Fourier transform is defined by

$$Z(n) = \sum_{j=0}^{N-1} X(j) \exp\left(\frac{2\pi i j n}{N}\right), (n = 0, 1, \dots, N - 1);$$

satisfying the condition of $Z(j) = NX(j)$, ($j = 0, 1, \dots, N - 1$). Function `fft()` evaluates these sums using fast Fourier transform technique. It is not required that N be a power of 2. One-, two-, and three-dimensional transforms can be performed.

Example 1

This example illustrates the use of the FFT. There is a signal

$$f(t) = \begin{cases} 9e^{-9t} & t \geq 0 \\ 0 & t < 0 \end{cases}$$

Analytically, the Fourier transform of function $f(t)$ is given by

$$F(\omega) = \frac{9}{9 + j\omega}$$

This example illustrates an approach to estimate the Fourier transform of less common signals because the sample frequency is limited. In this example, total 256 points of signal are sampled in 4 seconds. So the sample rate is $256/4 = 64\text{Hz}$.

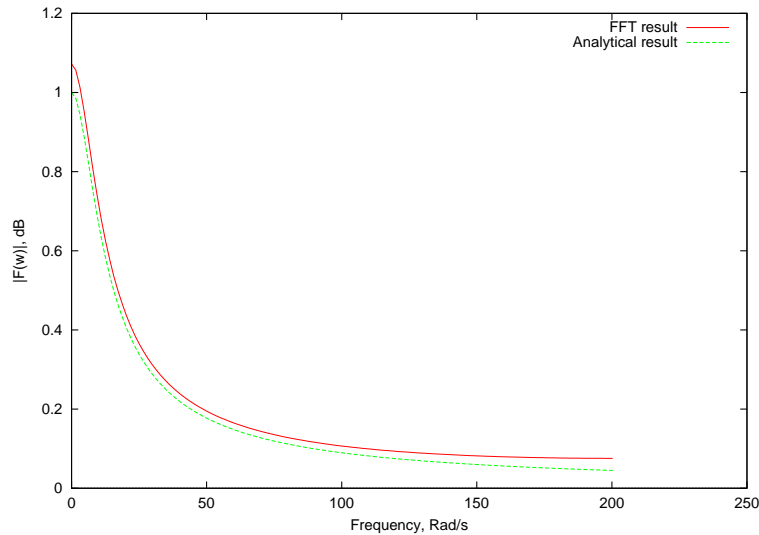
```
#include <stdio.h>
#include <math.h>
#include <chplot.h>
#include <numeric.h>

#define N 256    /* total sample points */
#define M 4     /* sample time, 4 seconds */

int main() {
    int i;
    array double t[N], f[N], E[N/2], W[N/2+1], y[2][N/2+1];
    array double complex F[N], Fp[N/2+1], Fa[N/2+1];
    double Ts, Ws;

    linspace(t, 0, M);
    f = 9*exp(-9*t);
    Ts = t[1] - t[0];
    Ws = 2*M_PI/Ts;
    fft(F, f);
    linspace(W, 0, N/2);
    W = W*Ws/N;
    for (i=0; i<N/2+1; i++) {
        Fp[i] = F[i]*Ts;
        Fa[i] = 9.0/(9+W[i]*complex(0,1));
        y[0][i] = abs(Fp[i]);
        y[1][i] = abs(Fa[i]);
    }
    plotxy(W, y, "FFT example", "Frequency, Rad/s", "|F(w)|, dB");
}
```

Output

**Example 2**

This example shows the one-dimensional fft usage. When the optional argument n is specified, it takes n points FFT. If it is not specified, a total number of elements of x is taken to FFT by default.

```
#include <stdio.h>
#include <math.h>
#include <numeric.h>

#define N 16
#define M 4

int main() {
    int i, j, k, retval;
    array double x1[N];
    array double complex y1[N], y2[M], x2[N], x3[M];
    int n = M;

    linspace(x1, 0, 1);

    fft(y1, x1);
    ifft(x2, y1);
    printf("FFT number of points by default\n");
    printf("Original data    FFT result    FFT + iFFT result\n");
    for (i=0; i<N; i++)
        printf("%5.3f  %5.3f  %5.3f\n", x1[i], y1[i], x2[i]);

    printf("FFT number of points is specified by n=%d\n", n);
    fft(y2, x1, n);
    ifft(x3, y2, n);
    printf("\n\nOriginal data    FFT result    FFT + iFFT result\n");
    for (i=0; i<M; i++)
        printf("%5.3f  %5.3f  %5.3f\n", x1[i], y2[i], x3[i]);
}
```

Output

```
FFT number of points by default
Original data    FFT result    FFT + iFFT result
0.000  complex(8.000,0.000)  complex(-0.000,-0.000)
```

```

0.067 complex(-0.533,-2.681) complex(0.067,0.000)
0.133 complex(-0.533,-1.288) complex(0.133,-0.000)
0.200 complex(-0.533,-0.798) complex(0.200,0.000)
0.267 complex(-0.533,-0.533) complex(0.267,0.000)
0.333 complex(-0.533,-0.356) complex(0.333,0.000)
0.400 complex(-0.533,-0.221) complex(0.400,-0.000)
0.467 complex(-0.533,-0.106) complex(0.467,0.000)
0.533 complex(-0.533,0.000) complex(0.533,0.000)
0.600 complex(-0.533,0.106) complex(0.600,0.000)
0.667 complex(-0.533,0.221) complex(0.667,0.000)
0.733 complex(-0.533,0.356) complex(0.733,0.000)
0.800 complex(-0.533,0.533) complex(0.800,0.000)
0.867 complex(-0.533,0.798) complex(0.867,0.000)
0.933 complex(-0.533,1.288) complex(0.933,0.000)
1.000 complex(-0.533,2.681) complex(1.000,-0.000)
FFT number of points is specified by n=4

```

```

Original data   FFT result   FFT + iFFT result
0.000 complex(0.400,0.000) complex(0.000,0.000)
0.067 complex(-0.133,-0.133) complex(0.067,0.000)
0.133 complex(-0.133,0.000) complex(0.133,0.000)
0.200 complex(-0.133,0.133) complex(0.200,0.000)

```

Example 3

This example shows the multi-dimensional fft usage. When the optional array argument *dim* is specified, the value for each element is used to specify the number of elements for each dimension of the array for performing FFT. If it is not specified, a total number of elements of *x* is taken to FFT by default.

```

#include <stdio.h>
#include <math.h>
#include <numeric.h>

#define N1 4
#define N2 3
#define M 3

int main() {
    int i, j, k, retval;
    array double complex x1[N1][N2], y1[N1][N2], x3[N1][N2];
    array double complex y[M][M], x2[M][M];
    int dim[2];

    linspace(x1, 0, 1);
    dim[0]=M;dim[1]=M;
    fft(y, x1, dim);
    dim[0]=M;dim[1]=M;
    ifft(x2, y, dim);

    printf("Input data =\n");
    printf("%5.3f", x1);
    printf("\nFFT result data =\n");
    printf("%5.3f", y);
    printf("\nFFT + iFFT result data =\n");
    printf("%5.3f", x2);

    fft(y1, x1);
    ifft(x3, y1);

    printf("\n\nInput data =\n");

```

```

    printf("%5.3f",x1);
    printf("\nFFT result data =\n");
    printf("%5.3f",y1);
    printf("\nFFT + iFFT result data =\n");
    printf("%5.3f",x3);
}

```

Output

```

Input data =
complex(0.000,0.000) complex(0.091,0.000) complex(0.182,0.000)
complex(0.273,0.000) complex(0.364,0.000) complex(0.455,0.000)
complex(0.545,0.000) complex(0.636,0.000) complex(0.727,0.000)
complex(0.818,0.000) complex(0.909,0.000) complex(1.000,0.000)

FFT result data =
complex(3.273,0.000) complex(-0.409,-0.236) complex(-0.409,0.236)
complex(-1.227,-0.709) complex(0.000,0.000) complex(0.000,0.000)
complex(-1.227,0.709) complex(0.000,-0.000) complex(0.000,-0.000)

FFT + iFFT result data =
complex(0.000,0.000) complex(0.091,0.000) complex(0.182,0.000)
complex(0.273,0.000) complex(0.364,0.000) complex(0.455,0.000)
complex(0.545,0.000) complex(0.636,0.000) complex(0.727,0.000)

```

```

Input data =
complex(0.000,0.000) complex(0.091,0.000) complex(0.182,0.000)
complex(0.273,0.000) complex(0.364,0.000) complex(0.455,0.000)
complex(0.545,0.000) complex(0.636,0.000) complex(0.727,0.000)
complex(0.818,0.000) complex(0.909,0.000) complex(1.000,0.000)

FFT result data =
complex(6.000,0.000) complex(-0.545,-0.315) complex(-0.545,0.315)
complex(-1.636,-1.636) complex(0.000,-0.000) complex(0.000,-0.000)
complex(-1.636,0.000) complex(-0.000,-0.000) complex(-0.000,0.000)
complex(-1.636,1.636) complex(0.000,0.000) complex(0.000,0.000)

FFT + iFFT result data =
complex(-0.000,0.000) complex(0.091,0.000) complex(0.182,0.000)
complex(0.273,0.000) complex(0.364,0.000) complex(0.455,0.000)
complex(0.545,0.000) complex(0.636,0.000) complex(0.727,0.000)
complex(0.818,0.000) complex(0.909,0.000) complex(1.000,0.000)

```

See Also

ifft().

References

R. C. Singleton, *An Algorithm for Computing the Mixed Radix F. F. T.*, IEEE Trans, Audio Electroacoust., AU-1(1969) 93-107.
 MJ Oleson, *Netlib Repository at UTK and ORNL*, go/fft-olesen.tar.gz, <http://www.netlib.org>.

filter

Synopsis

```
#include <numeric.h>
int filter(array double complex v[&], array double complex u[&],
          array double complex x[&], array double complex y[&], ...
          /* [array double complex zi[&], [array double complex zf[&]] */);
```

Syntax

```
filter(v, u, x, y)
filter(v, u, x, y, zi, zf)
```

Purpose

Filters the data in vector x with a filter represented by vectors u and v to create the filtered data y . The filter is a direct form II transposed implementation of the standard difference equation:

$$y(n) = v_0 * x(n) + v_1 * x(n - 1) + \dots + v_{nb} * x(n - nb - 1) \\ - u_1 * y(n - 1) - \dots - u_{na} * y(n - na - 1)$$

The input-output description of this filtering operation in the z -transform domain is a rational transfer function

$$Y(z) = \frac{v_0 + v_1 z^{-1} + \dots + v_{nb-1} z^{-nb-1}}{1 + u_1 z^{-1} + \dots + u_{na-1} z^{-na-1}} X(z)$$

Return Value

This function returns 0 on success and -1 on failure.

Parameters

v A vector of size nb . It contains the numerator of polynomial coefficients of the filter.

u A vector of size na . It contains the denominator of polynomial coefficients of the filter.

x A vector of size N . It contains the raw data.

y A vector of size N . It contains the filtered data.

zi A vector of size $(\max(na, nb) - 1)$. It contains initial delays of the filter.

zf A vector of size $\max(na, nb)$. It contains the final delays of the filter.

Description

The numerator coefficients with zeros of the system transfer function v , denominator coefficients with poles of the system transfer function u , and vector x of input data can be of any supported arithmetic data type and size. Conversion of the data to double complex is performed internally. The vector y which is the same size as x contains the result of filtered output. The optional arguments zi and zf are used to set the initial values of delays and get the final delays of the filter. They shall be double complex data type. The leading coefficient of denominator u_0 must be non-zero, as the other coefficients are divided by u_0 .

Algorithm

The block diagram of the direct form II transposed implementation is shown in Figure 1.

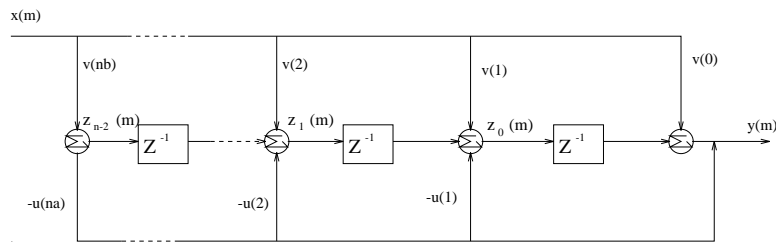


Figure 11.1: Direct form II transposed implementation

The operation of filter at sample m is given by the time domain difference equations

$$\begin{aligned}
 y(m) &= v_0 x(m) + z_0(m-1) \\
 z_0(m) &= v_1 x(m) - u_1 y(m) + z_1(m-1) \\
 &\vdots = \vdots \quad \vdots \\
 z_{n-3}(m) &= v_{n-2} x(m) - u_{n-2} y(m) + z_{n-2}(m-1) \\
 z_{n-2}(m) &= v_{n-1} x(m) - u_{n-1} y(m)
 \end{aligned}$$

where $n = \max(na, nb)$ and $m = 0, 1, \dots, N$. Initial delay state $z_0(0), z_0(1), \dots, z_0(n-2)$ are contained in vector z_i . Final delay states $z_{(N-1)}(0), z_{(N-1)}(1), \text{ and } z_{(N-1)}(n-1)$ are kept in vector z_f .

Example 1

This example shows the usage of function of `filter()`, specification of the initial delays z_i and the final delays z_f are passed back to the calling function.

```

#include <stdio.h>
#include <numeric.h>

#define N1 6          /* x array size */
#define N2 1          /* B term coefficient number */
#define N3 4          /* A term coefficient number */

int main()
{
    int i, j, n, m;
    array double complex x[N1], v[N2], u[N3], zi [N3-1], zf [N3-1];
    array double x1 [N1], v1 [N2], u1 [N3], y1 [N1];
    array double complex y [N1];
    for (i=0; i<N1; i++) {
        real(x[i])=i+1;
        imag(x[i])=i-1;
        x1[i] = i+1;
    }
    for (i=0; i<N3; i++) {
        real(u[i])=i+1;
        imag(u[i])=i+1;
        u1[i] = i+1;
    }
    for (i=0; i<N2; i++) {
        v[i]=i+1;
        v1[i] = i+1;
    }
}

```



```

    }
    for (i=0;i<N3-1;i++) {
        zi[i]=1;
    }

    filter(v,u,x,y,zi,zf); /* User specified initial delays */

    printf("u=%6.3f\n",u);
    printf("v=%6.3f\n",v);
    printf("x=%6.3f\n",x);
    printf("y=%6.3f\n",y);
    printf("zi=%6.3f\n",zi);
    printf("zf=%6.3f\n",zf);

    filter(v1,u1,x1,y1); /* User did not specify initial delays */

    printf("u1=%6.3f\n",u1);
    printf("v1=%6.3f\n",v1);
    printf("x1=%6.3f\n",x1);
    printf("y1=%6.3f\n",y1);

    return 0;
}

```

Output

```

u=complex( 1.000, 1.000) complex( 2.000, 2.000) complex( 3.000, 3.000)
complex( 4.000, 4.000)

v=complex( 1.000, 0.000)

x=complex( 1.000,-1.000) complex( 2.000, 0.000) complex( 3.000, 1.000)
complex( 4.000, 2.000) complex( 5.000, 3.000) complex( 6.000, 4.000)

y=complex( 1.000,-1.000) complex( 0.000, 1.000) complex( 0.000, 0.000)
complex(-1.000, 0.000) complex( 6.000,-5.000) complex(-4.000, 9.000)

zi=complex( 1.000, 0.000) complex( 1.000, 0.000) complex( 1.000, 0.000)

zf=complex(-6.000,-3.000) complex(-12.000,-7.000) complex(16.000,-36.000)

u1= 1.000  2.000  3.000  4.000

v1= 1.000

x1= 1.000  2.000  3.000  4.000  5.000  6.000

y1= 1.000  0.000  0.000  0.000  5.000 -4.000

```

Example 2

This example shows how to use the FFT algorithm to find the spectrum of signals which is buried in noises and use the filter algorithm to filter unwanted signals.

```
#include <stdio.h>
```

```

#include <math.h>
#include <chplot.h>
#include <numeric.h>

#define N 512

int main() {
    array double t[N], x[N], y[N], Pyy[N/2], f[N/2], u[7], v[7];
    array double complex Y[N];
    int i;
    class CPlot plot;

    /* The filter arguments u[i] and v[i] are designed according to the characteristic
       of filter specified. In this example program, we use the designed filter to
       filter the raw data. */
    u[0]=1;u[1]=-5.66792131;u[2]=13.48109005;u[3]=-17.22250511;
    u[4]=12.46418230;u[5]=-4.84534157;u[6]=0.79051978;
    v[0]=0.00598202; v[1]=-0.02219918; v[2]=0.02645738; v[3]=0;
    v[4]=-0.02645738;v[5]=0.02219918;v[6]=-0.00598202;

    linspace(t,0,N-1);
    t = t/N;
    for (i=0; i<N; i++) {
        x[i] = sin(2*M_PI*5*t[i]) + sin(2*M_PI*15*t[i]) + sin(2*M_PI*t[i]*30);
        x[i]=x[i]+3*(urand(NULL)-0.5);
    }

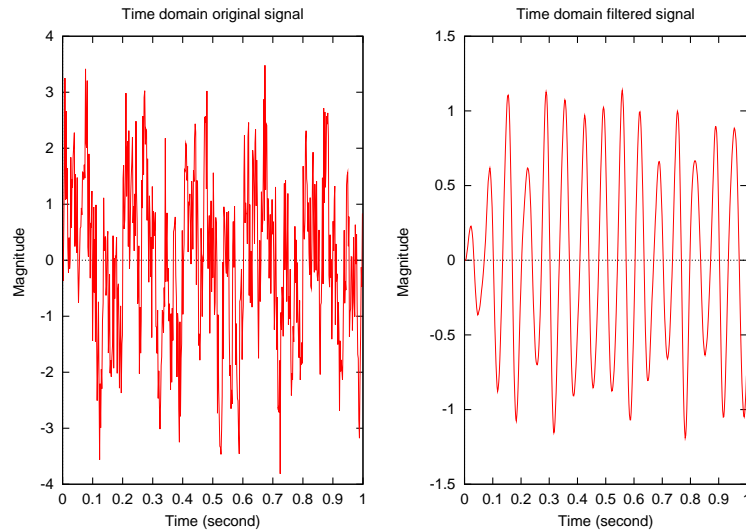
    filter(v,u,x,y);
    plotxy(t,x,"Time domain original signal","Time (second)","Magnitude ");
    plotxy(t,y,"Time domain filtered signal","Time (second)","Magnitude ");

    fft(Y,x);
    for (i=0; i<N/2; i++)
        Pyy[i] = abs(Y[i]);
    linspace(f,0,N/2);
    plotxy(f,Pyy,"Frequency domain original signal","frequency (Hz)","Magnitude (db)");
    fft(Y,y);
    for (i=0; i<256; i++)
        Pyy[i] = abs(Y[i]);
    linspace(f,0,255);
    plotxy(f,Pyy,"Frequency domain filtered signal","frequency (Hz)","Magnitude (db)");
}

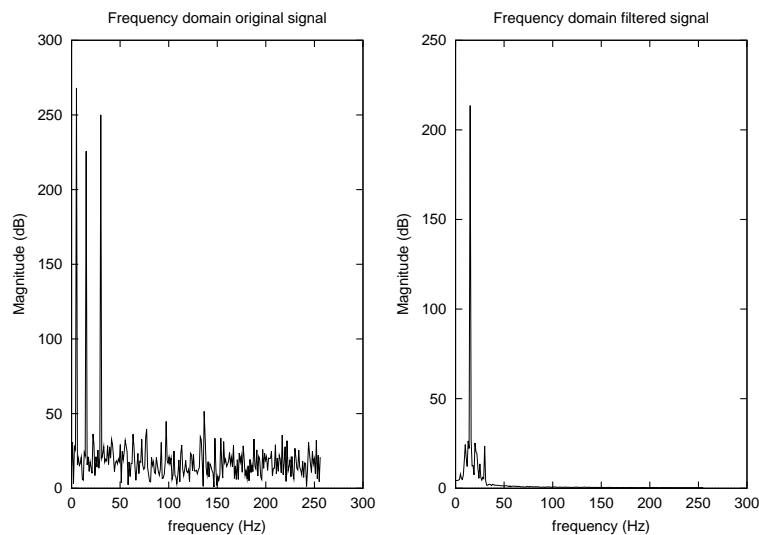
```

Output

This figure shows signals in the time domain. The figure on the left hand side is the original signals with three sinusoidal components at frequency of 5, 15, 30 Hz which are buried in white noises. The figure on the right hand side is signals after a sixth order IIR filter with a passband of 10 to 20 Hz. We hope to keep the 15 Hz sinusoidal signals and get rid of the 5 and 30 Hz sinusoids and other white noise using a filter with coefficients u and v given in the program.



The signals in the frequency domain are obtained by a fast Fourier transform algorithm using function `fft()`. This figure shows the original and filtered signals in the frequency domain. The figure on the left hand side shows the original signals with three separate main frequency spectrums and some noise frequency. After filtering, the signals, as shown on the right hand side of the figure, contain mainly a frequency of 15 Hz and a few noise components.



See Also
`filter2()`.

References

Alon V. Oppenheim, *Discrete-time Signal Processing*, Prentice Hall, 1998.

filter2

Synopsis

```
#include <numeric.h>
```

```
int filter2(array double complex y[&][&], array double complex u[&][&], array double complex x[&][&],
            ... /* [string_t method] */);
```

Syntax

```
filter2(y, u, x)
```

```
filter2(y, u, x, method);
```

Purpose

Two-dimensional Discrete Fourier Transform based FIR filter.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

y Two-dimensional **array** of filtered data.

u Two-dimensional **array** of size $nu \times mu$ of FIR filter.

x Two-dimensional **array** of size $nx \times mx$ of raw data.

method A string which describes the filter output method.

Description

Filter the data *x* with the 2-D FIR filter in the matrix *u*. By default, the result *y* is computed using 2-D convolution with the same size as *x*. If option *method* is specified, *y* is computed via 2-D convolution with size specified by "method":

"same" - (default) returns the central part of the convolution that is the same size as *x*.

"valid" - returns only those parts of the convolution that are computed without the zero-padded edges, the size of *y* is $(nx - nu + 1) \times (mx - mu + 1)$. The size of *x* must be bigger than the size of *u*.

"full" - returns the full 2-D convolution, that is the size of *y* is $(nx + nu - 1) \times (mx + mu - 1)$.

Algorithm

This function calls `conv2()`, the two-dimensional convolution function, to implement the filtering operation.

See `conv2()`.

Example

In image processing, Sobel filter often used to smoothing image. Convoluting the original two-dimensional image data with Sobel mask

$$s = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

The image can be smoothed.

```

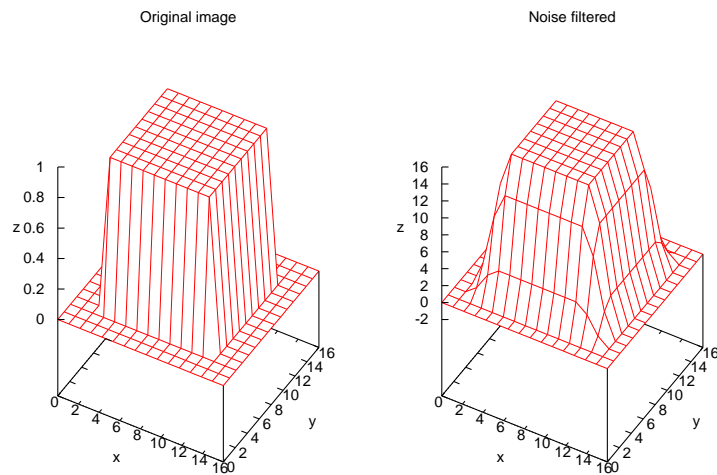
#include <math.h>
#include <chplot.h>
#include <numeric.h>

int main() {
    int i, j;
    array double u[3][3]={{1,2,1},{2,4,2},{1,2,1}};
    array double x[16],y[16],z1[256],z[16][16],Z[18][18],Z1[256];

    linspace(x,0,16);
    linspace(y,0,16);
    for(i=3; i<13; i++)
        for(j=3; j<13; j++) {
            z1[i*16+j]=1;
            z[i][j] = 1;
        }
    plotxyz(x,y,z1);
    filter2(Z,u,z,"full");
    for(i = 0; i<16; i++)
        for(j=0; j<16; j++)
            Z1[i*16+j] = Z[i+1][j+1];
    plotxyz(x,y,Z1);
}

```

Output



See Also

filter(), **conv2()**.

findvalue()

Synopsis

```
#include <numeric.h>
```

```
int findvalue(array int y[&], array double complex &x);
```

Syntax

```
findvalue(y, x)
```

Purpose

Obtain indices of nonzero elements of an array.

Return Value

This function returns the number of nonzero elements of an array.

Parameters

y A vector the size of the total number of elements of array *x*. It contains the indices of nonzero elements of array *x*.

x Any-dimensional input array which contains the original data.

Description

The original data **array** *x* can be of any supported arithmetic data type and extent sizes. The total number of nonzero elements is returned by the function call of **findvalue()**. Vector *y* is of integer data type with the total number of elements of *x*. It contains the indices of nonzero elements of array *x*. The remaining elements of *y* contain a value of -1 . If *x* is an array of complex data type, a zero element has a value of zero for both its real and imaginary parts.

Example

```
#include <numeric.h>
int main() {
    array double x1[5]={0,43.4,-7,-2.478,7};
    array double x[4][2]={1,3,
                          2.45,8.56,
                          -3,5,
                          6,8};
    array double complex zx[3][2]={complex(1,1),0,
                                   3,complex(4,1),
                                   complex(1,0),complex(0,1)};
    array int zy[6], y1[5],y[8];
    int i,j;

    j=findvalue(y1,x1);
    printf("x1 = \n%f",x1);
    printf("Total %d of x1 are not eq. 0. They are located at y1 = \n",j);
    for (i=0; i<j; i++) printf("%d  ",y1[i]);
    printf("\n");
    j=findvalue(y,x<2.0);
    printf("x = \n%f",x);
    printf("Total %d of x are less than 2.0. They are located at y = \n",j);
```

```

    for (i=0; i<j; i++) printf("%d  ",y[i]);
    printf("\n");
    j=findvalue(zy,zx);
    printf("zx = \n%f",zx);
    printf("Total %d of zx are not eq. 0. They are located at zy = \n",j);
    for (i=0; i<j; i++) printf("%d  ",zy[i]);
    printf("\n");
}

```

Output

```

x1 =
0.000000 43.400000 -7.000000 -2.478000 7.000000
Total 4 of x1 are not eq. 0. They are located at y1 =
1  2  3  4
x =
1.000000 3.000000
2.450000 8.560000
-3.000000 5.000000
6.000000 8.000000
Total 2 of x are less than 2.0. They are located at y =
0  4
zx =
complex(1.000000,1.000000) complex(0.000000,0.000000)
complex(3.000000,0.000000) complex(4.000000,1.000000)
complex(1.000000,0.000000) complex(0.000000,1.000000)
Total 5 of zx are not eq. 0. They are located at zy =
0  2  3  4  5

```

fliplr()

Synopsis

```
#include <numeric.h>
```

```
int fliplr(array double complex y[&][&], array double complex x[&][&]);
```

Syntax

```
fliplr(y, x)
```

Purpose

Flip matrix in left/right direction.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x Two-dimensional matrix which contains the original data.

y Two-dimensional matrix of the same data type and size as *x*, which contains the flipped result of input matrix *x*.

Description

This function flips matrix *x* in the left/right direction with rows being preserved and columns flipped.

Example

```
#include <numeric.h>
int main() {
    array double x[2][4]={1, 3, 2.45, 8.56,
                        3, 5, 6, 8};
    array double complex zx[2][3]={complex(1, 1), 0, 3,
                                   complex(4, 1), 6, 0};
    array double complex zy[2][3];
    array double y[2][4];

    fliplr(y, x);
    printf("x = \n%f", x);
    printf("y = \n%f", y);
    printf("\n");

    fliplr(zy, zx);
    printf("zx = \n%5.1f", zx);
    printf("zy = \n%5.1f", zy);
}
```

Output

```
x =
1.000000 3.000000 2.450000 8.560000
3.000000 5.000000 6.000000 8.000000
y =
```



```
8.560000 2.450000 3.000000 1.000000  
8.000000 6.000000 5.000000 3.000000
```

```
zx =
```

```
complex( 1.0, 1.0) complex( 0.0, 0.0) complex( 3.0, 0.0)  
complex( 4.0, 1.0) complex( 6.0, 0.0) complex( 0.0, 0.0)
```

```
zy =
```

```
complex( 3.0, 0.0) complex( 0.0, 0.0) complex( 1.0, 1.0)  
complex( 0.0, 0.0) complex( 6.0, 0.0) complex( 4.0, 1.0)
```

See Also

flipud(), rot90().

flipud()

Synopsis

```
#include <numeric.h>
```

```
int flipud(array double complex y[&][&], array double complex x[&][&]);
```

Syntax

```
flipud(y, x)
```

Purpose

Flip matrix in up/down direction.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x Two-dimensional matrix which contains the original data.

y Two-dimensional matrix of the same data type and size as *x*, which contains the flipped result of input matrix *x*.

Description

This function flips matrix *x* in the up/down direction with columns being preserved and rows flipped.

Example

```
#include <numeric.h>
int main() {
    array double x[4][2]={1, 3,
                          2.45, 8.56,
                          3, 5,
                          6, 8};
    array double complex zx[3][2]={complex(1, 1), 0,
                                   3, complex(4, 1),
                                   6, 0};
    array double complex zy[3][2];
    array double y[4][2];

    flipud(y, x);
    printf("x = \n%f", x);
    printf("y = \n%f", y);
    printf("\n");

    flipud(zy, zx);
    printf("zx = \n%5.1f", zx);
    printf("zy = \n%5.1f", zy);
}
```

Output

x =

```
1.000000 3.000000
2.450000 8.560000
3.000000 5.000000
6.000000 8.000000
y =
6.000000 8.000000
3.000000 5.000000
2.450000 8.560000
1.000000 3.000000
```

```
zx =
complex( 1.0, 1.0) complex( 0.0, 0.0)
complex( 3.0, 0.0) complex( 4.0, 1.0)
complex( 6.0, 0.0) complex( 0.0, 0.0)
zy =
complex( 6.0, 0.0) complex( 0.0, 0.0)
complex( 3.0, 0.0) complex( 4.0, 1.0)
complex( 1.0, 1.0) complex( 0.0, 0.0)
```

See Also**fliplr(), rot90().**

fminimum

Synopsis

```
#include <numeric.h>
```

```
int fminimum (double *fminval, double *xmin, double (*func)(double), double x0, double xf, ...
              /* [ double rel_tol, double abs_tol] */);
```

Syntax

```
fminimum(fminval, xmin, func, x0, xf)
```

```
fminimum(fminval, xmin, func, x0, xf, rel_tol)
```

```
fminimum(fminval, xmin, func, x0, xf, rel_tolm, abs_tol)
```

Purpose

Find the minimum value of a one-dimensional function and its corresponding position for the minimum value.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

fminval A pointer to the variable which passes the calculated minimum value of the function.

xmin A pointer to the variable which contains the calculated position with the minimum value of the function.

func The function given by user for minimization.

x0 The beginning-point of the interval on which a minimum value is searched for.

xf The end-point of the interval on which a minimum value is searched for.

rel_tol The relative tolerance of the calculation.

abs_tol The absolute tolerance of the calculation.

Description

Determines a point between *x0* and *xf* at which the real function *func* assumes a minimum value. The function *func* given by the user has an argument for the *x* value input. The argument *fminval* passes the calculated minimum value of the function. The argument *xmin* contains the position where the minimum value of the function is found. The tolerance is defined as a function of *x*: $|x|rel_tol + abs_tol$, where *rel_tol* is the relative precision and *abs_tol* is the absolute precision which should not be chosen equal to zero. The default value for *rel_tol* and *abs_tol* is 10^{-6} .

Limitations

In the interval at which the function assumes a minimum value, it is assumed that for some point *u* either (a) *func* is strictly monotonically decreasing on [a,u) and strictly monotonically increasing on [u,b] or (b) these two intervals may be replaced by [a,u) and (u,b] respectively.

Example

Find the minimum of the function $f(x) = -\frac{1}{(x-0.3)^2+0.01} - \frac{1}{(x-0.9)^2+0.04} - 6$ at intervals [-1,0], [0,0.8], [0.4,1] and [0,1] with the default value for tolerance. For interval [0,1] this function can only find the local minimum at 0.892716. This example also finds the minimum of the function at the interval [0,0.8] with the given tolerance *rel_tol* and *abs_tol* equal to 0.01.

```
#include <numeric.h>
#include <stdio.h>

double func(double x) {
    double y;
    y = 1/((x-0.3)*(x-0.3)+0.01) + 1/((x-0.9)*(x-0.9)+0.04) - 6;
    return -y;
}

int main() {
    double x0, xf, rel_tol, abs_tol;
    double func(double);
    double xmin, fminval;
    int status;
    string_t title = "f =-1/((x-0.3)*(x-0.3)+0.01) - 1/((x-0.9)*(x-0.9)+0.04) + 6 ";

    fplotxy(func, -1, 1, 50, title, "x", "y");
    x0 = -1; xf = 0; /* fminimum is at 0.0 */
    status = fminimum(&fminval, &xmin, func, x0, xf);
    printf("Interval = [%3.2f, %3.2f]\n", x0, xf);
    printf("status = %d\n", status);
    printf("xmin = %f\n", xmin);
    printf("m = %f\n\n", fminval);

    x0 = 0; xf = 0.8; /* fminimum is around 0.3 */
    status = fminimum(&fminval, &xmin, func, x0, xf);
    printf("Interval = [%3.2f, %3.2f]\n", x0, xf);
    printf("status = %d\n", status);
    printf("xmin = %f\n", xmin);
    printf("m = %f\n\n", fminval);

    rel_tol = 1E-2; /* default tolerance is 1E-6 */
    abs_tol = 1E-2; /* default tolerance is 1E-6 */
    x0 = 0; xf = 0.8; /* fminimum is around 0.6 */
    status = fminimum(&fminval, &xmin, func, x0, xf, rel_tol, abs_tol);
    printf("Interval = [%3.2f, %3.2f]\n", x0, xf);
    printf("status = %d\n", status);
    printf("xmin = %f\n", xmin);
    printf("m = %f\n\n", fminval);

    x0 = 0.4; xf = 1; /* fminimum is at 0.4 */
    status = fminimum(&fminval, &xmin, func, x0, xf);
    printf("Interval = [%3.2f, %3.2f]\n", x0, xf);
    printf("status = %d\n", status);
    printf("xmin = %f\n", xmin);
    printf("m = %f\n\n", fminval);

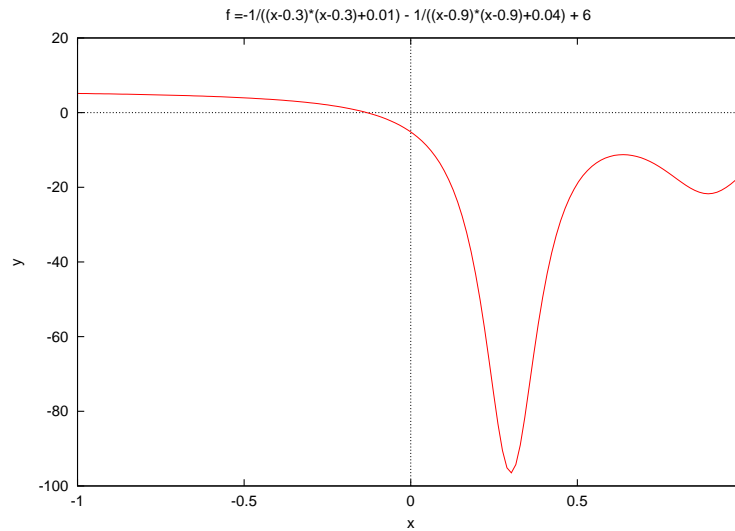
    x0 = 0; xf = 1; /* Only local minimum at 0.892716 is found */
    status = fminimum(&fminval, &xmin, func, x0, xf);
    printf("Interval = [%3.2f, %3.2f]\n", x0, xf);
    printf("status = %d\n", status);
```

```

    printf("xmin = %f\n", xmin);
    printf("m = %f\n\n", fminval);
}

```

Output



```

Interval = [-1.00, 0.00]
status = 0
xmin = 0.000000
m = -5.176471

```

```

Interval = [0.00, 0.80]
status = 0
xmin = 0.300375
m = -96.501409

```

```

Interval = [0.00, 0.80]
status = 0
xmin = 0.305573
m = -96.232705

```

```

Interval = [0.40, 1.00]
status = 0
xmin = 0.400000
m = -47.448276

```

```

Interval = [0.00, 1.00]
status = 0
xmin = 0.892717
m = -21.734573

```

See Also

fminimums().

References

H. T. Lau, *A Numerical Library in C for Scientists and Engineers*, CRC Press, Inc. 1995.

fminimums

Synopsis

```
#include <numeric.h>
```

```
int fminimums(double *fminval, double xmin[&], double (*func)(int, double[ ]), double x0[&], ...
              /*[ double rel_tol, double abs_tol, int numfuneval] */);
```

Syntax

```
fminimums(fminimumval, xmin, func, x0)
```

```
fminimums(fminimumval, xmin, func, x0, rel_tol)
```

```
fminimums(fminimumval, xmin, func, x0, rel_tol, abs_tol)
```

```
fminimums(fminimumval, xmin, func, x0, rel_tol, abs_tol, numfuneval)
```

Purpose

Find the minimum position and value of an n-dimensional function.

Return Value

This function returns one of the following values: **Parameters**

0 On successful.

-1 The process is broken off because at the end of an iteration step the number of calls of *func* exceeded the value of *numfuneval*.

-2 The process is broken off because the condition of the problem is too bad.

-3 Function not supported.

Parameters

fminval A pointer to the variable which contains the calculated minimum value of the function.

xmin A pointer to the array which contains the calculated positions with the minimum value of the function.

func An n-dimensional function given by the user for minimization.

x0 An input array of approximation for variables *x* where the function gives the minimum value.

rel_tol The relative tolerance of the calculation.

abs_tol The absolute tolerance of the calculation.

numfuneval The maximum number of function evaluation allowed.

Description

Given an n-dimensional function and an array which contains initial estimate by the user as input, this function calculates the array of the position at which the function has a minimum value. The number of dimension is taken from input array *x* internally. The function *func* should deliver the value of the function to be minimized, at the points given by *x*. The tolerance is defined as a function of *x* as $|x|rel_tol + abs_tol$, where *rel_tol* is the relative precision and *abs_tol* is the absolute precision which should not be chosen equal to zero. The argument *numfuneval* is the maximum number of function evaluations allowed. The default values for *rel_tol*, *abs_tol* and *numfuneval* are 10^{-6} , 10^{-6} , and 250, respectively.

Example

Find the minimum value of two dimension function $100(x_1 - x_0^2)^2 + (1.0 - x_0)^2$ with the initial guess of $x_0 = -1.2$ and $x_1 = 1.0$.

```
#include <numeric.h>
#include <stdio.h>
#define N 2

double func(double x[]) {
    double temp;
    temp=x[1]-x[0]*x[0];
    return 100*temp*temp+(1.0-x[0])*(1.0-x[0]);
}

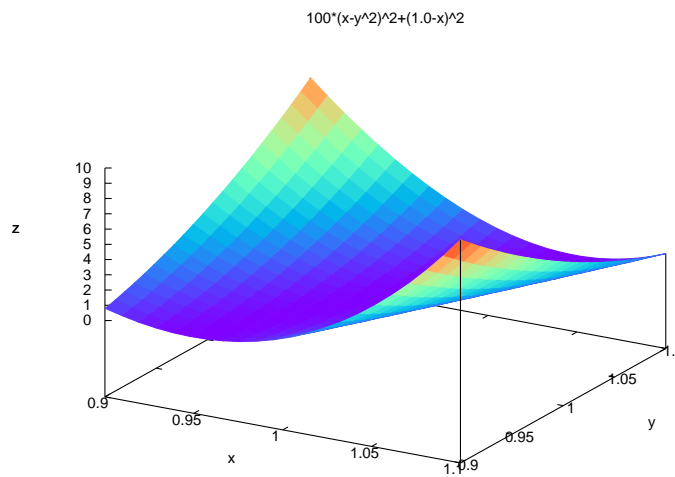
int main () {
    double func(double x[]);
    double fminval,xmin[N],x0[N],rel_tol, abs_tol;
    int numfuneval, status;
    x0[0] = -1.2;
    x0[1]=1.0;

    /* Plot a 2 dimensional graph of the function
       for which the minimum will be found */
    double funcp(double x, y)
    {
        double temp;
        temp=y-x*x;
        return 100*temp*temp+(1.0-x)*(1.0-x);
    }
    fplotxyz(funcp, 0.9, 1.1, 0.9, 1.1, 22,22,"100*(x-y*y)*(x-y*y)+(1.0-x)*(1.0-x)",
            "x","y","z");

    status = fminimums(&fminval, xmin, func,x0);
    printf("status = %d\n", status);
    printf("fminval = %f\n", fminval);
    printf("xmin[0] = %f\n", xmin[0]);
    printf("xmin[1] = %f\n", xmin[1]);

    rel_tol = 1.0e-3; /* not default value */
    abs_tol = 1.0e-3; /* not default value */
    numfuneval = 250; /* not default value */
    status = fminimums(&fminval, xmin, func,x0,rel_tol, abs_tol,numfuneval);
    printf("status = %d\n", status);
    printf("fminval = %f\n", fminval);
    printf("xmin[0] = %f\n", xmin[0]);
    printf("xmin[1] = %f\n", xmin[1]);
}
```

Output



```
status = 0
fminval = 0.000000
xmin[0] = 1.000000
xmin[1] = 1.000000
status = 0
fminval = 0.000000
xmin[0] = 0.999999
xmin[1] = 1.000028
```

See Also
fminimum().

References

H. T. Lau, *A Numerical Library in C for Scientists and Engineers*, CRC Press, Inc. 1995.

fsolve

Synopsis

```
#include <numeric.h>
```

```
int fsolve (double x[:], void(*func)(double x[ ], double y[ ]), double x0[:]);
```

Purpose

Find a zero position of a nonlinear system of equations.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x An n-dimensional array which contains the calculated zero position.

x0 An n-dimensional array which contains the initial guess for a zero position.

func A pointer to the function given by the user.

Description

Function **fsolve()** finds a zero position of function *func(x)* provided by the user. The user function has two arguments, the first one for input and the second one for output. The input argument is an n-dimensional array, and the function values calculated will be delivered by the second array argument of the same dimension as output. The number of dimension is taken from the function given by the user internally.

Algorithm

The multidimensional nonlinear system of equations is solved by a multidimensional secant method: Broyden's method. Since the Jacobian matrix which is needed for solution of the equations is approximated by Quasi-Newton method inside the function, the user does not need to provide analytic formula for derivatives of equations. Broyden's method converges superlinearly once you get close enough to the root.

Example

Solve the following nonlinear system of two equations

$$\begin{aligned} f_0 &= -(x_0^2 + x_1^2 - 2.0) = 0 \\ f_1 &= e^{x_0-1.0} + x_1^3 - 2.0 = 0 \end{aligned}$$

with the initial guesses of zero point at $x_0 = 2.0$, and $x_1 = 0.5$.

```
/* Driver for routine fsolve: for 2 dimentional equation*/
#include <stdio.h>
#include <numeric.h>
#include <chplot.h>
#define N 2
#define NP 22

void func(double x[], double f[]){
    f[0]=-(x[0]*x[0]+x[1]*x[1]-2.0);
    f[1]=exp(x[0]-1.0)+x[1]*x[1]*x[1]-2.0;
}
```

```

int main(){
    double x[N], x0[N], f[N];
    double x1[NP], x2[NP], f1[NP*NP], f2[NP*NP];
    int i,j,status, datasetnum=0;
    class CPlot plot;

    linspace(x1, 0, 2);
    linspace(x2, 0, 2);

    for(i=0; i<NP; i++) {
        for(j=0; j<NP; j++){
            f1[NP*i+j] = -(x1[i]*x1[i]+x2[j]*x2[j] - 2.0);
            f2[NP*i+j] = exp(x1[i]-1.0) + x2[j]*x2[j]*x2[j]-2.0;
        }
    }
    plot.data3D(x1, x2, f1);
    plot.data3D(x1, x2, f2);
    plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum);
    plot.legend("f1", datasetnum);
    datasetnum++;
    plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum);
    plot.legend("f2", datasetnum);
    plot.label(PLOT_AXIS_X, "x");
    plot.label(PLOT_AXIS_Y, "y");
    plot.label(PLOT_AXIS_Z, "z");
    plot.title("f1 = 2.0 - x*x -y*y, f2 = exp(x-1.0)+y*y*y-2.0");
    plot.ticsLevel(0);
    plot.colorBox(PLOT_OFF);
    plot.plotting();

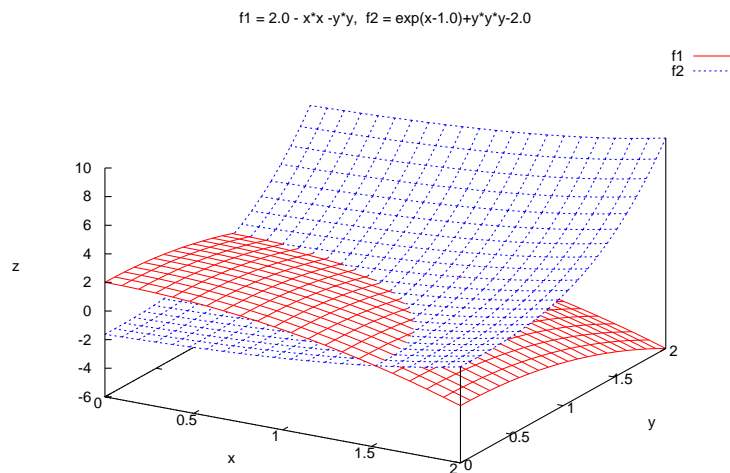
    x0[0]=2.0;
    x0[1]=0.5;
    status=fsolve(x, func, x0);

    func(x, f);

    printf("status = %d\n",status);
    if (status===-1)
        fprintf(stderr, "Convergence problems.\n");
    printf("%7s %3s %12s\n", "Index", "x", "f");
    for (i=0; i<2; i++)
        printf("%5d %12.6f %12.6f\n", i, x[i], f[i]);
}

```

Output



```
status = 0
  Index  x          f
    0    1.000016  -0.000020
    1    0.999994  -0.000002
```

See Also
fzero().

References

William H. Press, Saul A. Teukolsky, William T. Vetterling and Brian P. Flannery, *Numeric Recipes in C*, Second Edition, Cambridge University Press, 1997.

funm

Synopsis

```
#include <numeric.h>
```

```
int funm(array double y[&][&], double (*func)(double ), array double x[&][&]);
```

Syntax

```
funm(y, func, x)
```

Purpose

Evaluate general real matrix function.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x Input square matrix. It contains data to be evaluated.

func A function routine given by the user.

y Output square matrix which contains data of the calculated function values.

Description

This function evaluates the matrix version of the function specified by argument *func*. In this function, The input matrix *x* shall be **double** data type and the specified function prototype shall be **double func(double)**. The output matrix *y* could be **real** or **complex** data type as required.

Example

A real matrix evaluation.

```
#include <numeric.h>
double mylog(double x) {
    return log(x);
}

int main() {
    array double x[3][3]={1,2,3,
                          3,4,5,
                          6,7,8};
    array double y[3][3];

    expm(y, x);
    printf("x = \n%f", x);
    printf("y = \n%f", y);
    funm(x, mylog, y);
    printf("x = \n%f", x);
}
```

Output

```
x =
1.000000 2.000000 3.000000
3.000000 4.000000 5.000000
6.000000 7.000000 8.000000
y =
157372.953093 200034.605129 242697.257164
290995.910241 369883.552084 448769.193928
491431.845963 624654.472518 757878.099072
x =
1.000000 2.000000 3.000000
3.000000 4.000000 5.000000
6.000000 7.000000 8.000000
```

See Also

expm(), **logm()**, **cfunm()**, **sqrtnm()**.

References

G. H. Golub, C. F. Van Loan, *Matrix Computations* Third edition, The Johns Hopkins University Press, 1996

fzero

Synopsis

```
#include <numeric.h>
```

```
int fzero(double *x, double(*func)(double x), ... /* [double x0] | [double x02[2]*/] );
```

Purpose

Find a zero position of a nonlinear function with one variable.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x Output of the calculated zero position.

func A pointer to the function given by the user.

x0 Input of the initial guess for zero position.

x02 Input of a vector of length 2 and double type. The function shall be bracketed in the interval of [*x02*[0], *x02*[1]] so that the sign of *func*(*x02*[0]) differs from the sign of *func*(*x02*[1]). Otherwise, an error occurs.

Description

Function **fzero()** finds a zero position of function *func*(*x*) provided by the user. The input argument of function *func*() is *x* value.

Algorithm

If *x0* is given, the algorithm of one-dimensional zero finding is based on the algorithm for finding zero of the multi-dimensional nonlinear system of equations. Based on **fsolve()**, the number of dimension is set to 1 for **fzero()**. See algorithm for **fsolve()**. If *x02* is given, a bisection method is used to find the zero position.

Example

Find zero of function $f = x^2 - 2$ with initial guess $x = 2.0$.

```
#include <stdio.h>
#include <chplot.h>
#include <numeric.h>

double func(double x) {
    return x*x-2.0;
}

int main() {
    double x, x0, f, x02[2];
    int i, status;
    double func(double);
    fplotxy(func, -3, 3, 50, "f = x*x-2", "x", "y");

    x0=-2.0;
    status=fzero(&x, func, x0); // x0 is a scalar
    f=func(x);
```

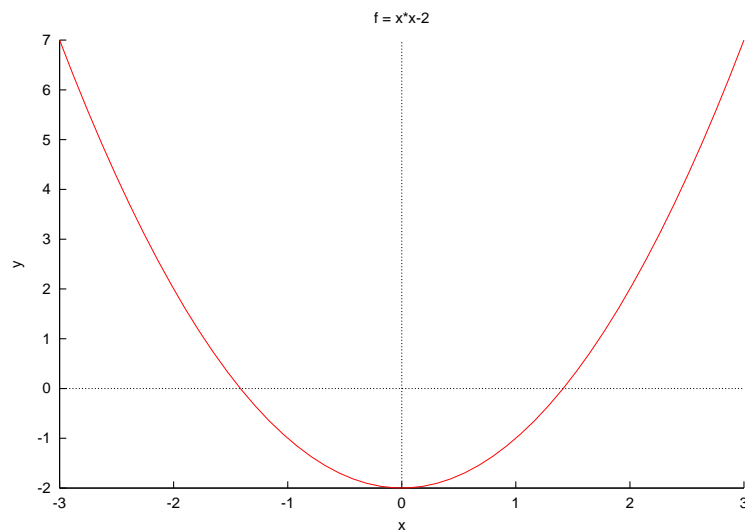
```

    if (status<0)
        printf("fzero() failed.\n");
    printf(" %6s %12s\n", "x", "f");
    printf("%10.6f %12.6f\n", x, f);

    x02[0]=-2.0;
    x02[1]=0;
    status=fzero(&x, func, x02); // x02 is an array
    f=func(x);
    if (status<0)
        printf("fzero() failed.\n");
    printf(" %6s %12s\n", "x", "f");
    printf("%10.6f %12.6f\n", x, f);
}

```

Output



x	f
1.414213	-0.000000
x	f
-1.414213	-0.000000

See Also

fsolve().

References

William H. Press, Saul A. Teukolsky, William T. Vetterling and Brian P. Flannery, *Numeric Recipes in C*, Second Edition, Cambridge University Press, 1997.

fzero1

Synopsis

```
#include <numeric.h>
```

```
int fzero1(double &x, double(func)(double x), double x0);
```

Purpose

Find a zero position of a nonlinear function with one variable.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x Output of the calculated zero position.

func A pointer to the function given by the user.

x0 Input of the initial guess for the zero position.

Description

Function **fzero1()** finds a zero position of function $func(x)$ provided by the user. The input argument of function $func()$ is x value.

Algorithm

The algorithm of one-dimensional zero finding is based on the algorithm for finding zero of the multi-dimensional nonlinear system of equations. Based on **fsolve()**, the number of dimension is set to 1 for **fzero1()** using **fzero()**. See algorithm for **fsolve()**.

Example

Find zero of function $f = x^2 - 2$ with initial guess $x = 2.0$.

```
/* File: fzero1.ch
   Solve a nonlinear equation f(x) = x*x-2 with an initial guess */
#include <numeric.h>

double func(double x) {
    return x*x-2;
}

double x;
double x0 = 1;          // initial guess
fzero1(x, func, x0);
printf("x = %lf\n", x);
```

Output

```
x = 1.414213
```

See Also

fsolve(), **fzero()**, **fzero2()**.

fzero2**Synopsis****#include** <numeric.h>**int fzero2(double & x, double(func)(double x), double x1, double x2);****Purpose**

Find a zero position of a nonlinear function with one variable.

Return Value

This function returns 0 on success and -1 on failure.

Parameters*x* Output of the calculated zero position.*func* A pointer to the function given by the user.*x1* The starting value of the interval with a zero position.*x2* The ending value of the interval with a zero position. The function shall be bracketed in the interval of [*x1*, *x2*] so that the sign of *func(x1)* differs from the sign of *func(x2)*. Otherwise, an error occurs.**Description**Function **fzero2()** finds a zero position of function *func(x)* provided by the user. The input argument of function *func()* is *x* value.**Algorithm**

A bisection method is used to find the zero position.

ExampleFind zero of function $f = x^2 - 2$ with in the interval $[x_1, x_2] = [-2, -1]$.

```

/* File: fzero2.ch
   Solve a nonlinear equation f(x) = x*x-2 in an interval [x1, x2].
   The sign of f(x1) should be different from that of f(x2) */
#include <numeric.h>

double func(double x) {
    return x*x-2;
}

double x;
double x1, x2;
x1 = -2;
x2 = -1;
fzero2(x, func, x1, x2); // find the solution in [-2, -1]
printf("x = %lf\n", x);
fzero2(x, func, 1, 2); // find the solution in [1, 2]
printf("x = %lf\n", x);

```

Output

```
x = -1.414213  
x = 1.414213
```

See Also

fsolve(), **fzero()**, **fzero1()**.

gcd()

Synopsis

```
#include <numeric.h>
int gcd(int num1, int num2);
```

Purpose

Obtain the greatest common divisor of two integers.
arrays of integer type.

Return Value

This function returns 0 on success and -1 on failure if one of two integer arguments is less or equal to 0.

Parameters

num1 An integer.

num2 Another integer.

g The greatest common divisor of two integers.

Description

This function calculates the greatest common divisor of two integers *num1* and *num2*. If one of two integer arguments is less or equal to 0, the function returns -1.

Example

```
/* File: gcd.ch */
#include <numeric.h>
int num1 = 54;
int num2 = 24;

printf("The divisors of %d are\n%d", num1, divisor(num1));

printf("The divisors of %d are\n%d", num2, divisor(num2));

printf("The common divisors of %d and %d are\n%d", num1, num2, cdivisor(num1, num2));
printf("The greatest common divisor of %d and %d is %d\n", num1, num2, gcd(num1, num2));
```

Output

```
The divisors of 54 are
1 2 3 6 9 18 27 54
The divisors of 24 are
1 2 3 4 6 8 12 24
The common divisors of 54 and 24 are
1 2 3 6
The greatest common divisor of 54 and 24 is 6
```

See Also

gcdarray(), **lcm()**, **lcmarray()**.

gcdarray()

Synopsis

```
#include <numeric.h>
```

```
int gcdarray(array int &u, array int &v, array int &g, ... /* [array int c[&], array int d[&]]*/);
```

Syntax

```
gcdarray(u, v, g)
```

```
gcdarray(u, v, g, c, d)
```

Purpose

Obtain the greatest common divisor of the corresponding elements of two arrays of integer type.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

u Any-dimensional array which contains non-negative integer elements for the greatest common divisor calculation.

v An array the same dimension and size as *u*. It contains another non-negative integer elements for the greatest common divisor calculation.

g An output array the same dimension and size as *u*. It contains the result of the greatest common divisor calculation of *u* and *v*.

c An output array the same dimension and size as *u*.

d An output array the same dimension and size as *u*.

Description

This function calculates the greatest common divisor of corresponding elements *u* and *v*. The arrays *u* and *v* shall be the same size and contain non-negative integer data. The output array *g* is a positive integer array of the same size as *u*. The optional output arrays *c* and *d* are of the same size as *u* and satisfy the equation $u.*c + v.*d = g$.

Example

```
#include <numeric.h>
int main() {
    array int u[2][3] = {1,2,7,15,3,4};
    array int v[2][3] = {2,4,8,3,8,3};
    array int g[2][3], c[2][3], d[2][3];
    gcdarray(u,v,g,c,d);
    printf("g=%d", g);
    printf("c=%d", c);
    printf("d=%d", d);
    printf("u.*c+v.*d = %d", u.*c+v.*d);
}
```

Output

```
g=1 2 1
3 1 1
c=1 1 -1
0 3 1
d=0 0 1
1 -1 -1
u.*c+v.*d = 1 2 1
3 1 1
```

See Also

gcd(), **lcm()**, **lcmarray()**.

References

Knuth, Donald Ervin, *The art of computer programming*, Vol. 2, Addison-Wesley Pub. Co., 1973.

getnum

Synopsis

```
#include <numeric.h>
```

```
double getnum(string_t msg, double d);
```

Purpose

Obtain a number from the console through standard input stream *stdin*.

Return Value

This function returns the default number or input number from the console.

Parameters

msg Message printed out in the standard output stream *stdout*.

d Default number.

Description

This function returns the default number when carriage RETURN is entered as input or new number from *stdin* as a double-precision floating-point number. However, if an invalid number is entered, a number is requested. The message in string *msg* will be printed out.

Example

```
#include <numeric.h>
/* The input numbers typed in on the console are 100 200 */

int main() {
    double d=2.0;

    printf("\nEnter a number [%lf]: ", d);
    d = getnum(NULL, d);
    printf("\nThe number you entered: %lf\n\n", d);
    d = getnum("Please enter a number[10.0]: ", 10);
    printf("\nThe number you entered: %lf\n\n", d);
}
```

Output

```
Enter a number [2.000000]:
The number you entered: 100.000000

Please enter a number[10.0]:
The number you entered: 200.000000
```

See Also

[getline\(\)](#).

References

hessdecomp

Synopsis

```
#include <numeric.h>
```

```
int hessdecomp(array double complex a[&][&], array double complex h[&][&], ...
               /* [array double complex p[&][&]] */);
```

Syntax

```
hessdecomp(a, h);
hessdecomp(a, h, p);
```

Purpose

Reduces a real general matrix a to upper Hessenberg form h by an orthogonal/unitary matrix p similarity transformation: $p^T * a * p = h$ or $p^H * a * p = h$ for real and complex, respectively. The p^H is Hermitian of matrix p .

Return Value

This function returns 0 on success and negative value on failure.

Parameters

a A $n \times n$ square matrix to be decomposed.

h An output two-dimensional matrix which contains the upper Hessenberg matrix of matrix a .

p An optional output two-dimensional array which contains an orthogonal or unitary matrix.

Description

This function computes the Hessenberg matrix h and an orthogonal/unitary matrix p so that $h = p^T * a * p$ and $p^T * p = I$ for real matrix, and $h = p^H * a * p$ and $p^H * p = I$ for complex matrix. Each element of a Hessenberg matrix below the first subdiagonal is zero. If the matrix is symmetric or Hermitian, the form is tridiagonal. This matrix has the same eigenvalues as the original one, but less computation is needed to calculate them.

In this function, square matrix a could be any supported arithmetic data type. Output h is the same dimension and data type as input a . If the input a is of real type, the optional output p shall only be double type. If the input a is complex type, p shall be **complex** or **double complex** type.

Example1

Reduce a real general matrix to upper Hessenberg form.

```
#include <numeric.h>
int main() {
    int m = 3;
    array double a[3][3] = { 0.8,0.2,0.1,
                           0.1,0.7,0.3,
                           0.1,0.1,0.6};

    int status;
    array double p[m][m], h[m][m];
    status = hessdecomp(a, h);
    if (status == 0) {
        printf("h =\n%f\n", h);
    }
}
```



```

    }
    else
        printf(" Hessenberg matrix calculation error.\n");
    status = hessdecomp(a, h, p);
    if (status == 0) {
        printf("h =\n%f\n", h);
        printf("p =\n%f\n", p);
        printf("transpose(p)*p =\n%f\n", transpose(p)*p );
        printf("transpose(p)*a*p - h =\n%f\n", transpose(p)*a*p - h);
    }
    else
        printf(" Hessenberg matrix calculation error.\n");
}

```

Output1

```

h =
0.800000 -0.212132 -0.070711
-0.141421 0.850000 -0.050000
0.000000 0.150000 0.450000

```

```

h =
0.800000 -0.212132 -0.070711
-0.141421 0.850000 -0.050000
0.000000 0.150000 0.450000

```

```

p =
1.000000 0.000000 0.000000
0.000000 -0.707107 -0.707107
0.000000 -0.707107 0.707107

```

```

transpose(p)*p =
1.000000 0.000000 0.000000
0.000000 1.000000 -0.000000
0.000000 -0.000000 1.000000

```

```

transpose(p)*a*p - h =
0.000000 -0.000000 -0.000000
0.000000 0.000000 -0.000000
0.000000 0.000000 0.000000

```

Example2

Reduce a complex general matrix to upper Hessenberg form.

```

#include <numeric.h>
int main() {
    int m = 3;

    array double complex a[3][3] = { complex(-149,1), -50, -154,
                                     537, 180, 546,
                                     -27, -9, -25};

    int status;
    array double complex h[m][m];
    array double complex p[m][m];
    status = hessdecomp(a, h);
    if (status == 0) {
        printf("h =\n%5.2f\n", h);
    }
}

```

```

else
    printf("Hessenberg matrix calculation error.\n");
status = hessdecomp(a, h, p);
if (status == 0) {
    printf("h =\n%5.2f\n", h);
    printf("p =\n%5.2f\n", p);
    printf("transpose(p)*p =\n%5.2f\n", transpose(p)*p);
    printf("conj(transpose(p))*a*p - h =\n%5.2f\n", conj(transpose(p)*a*p - h));
}
else
    printf("Hessenberg matrix calculation error.\n");
}

```

Output2

```

h =
complex(-149.00, 1.00) complex(42.20, 0.00) complex(-156.32, 0.00)
complex(-537.68, 0.00) complex(152.55, 0.00) complex(-554.93, 0.00)
complex( 0.00, 0.00) complex( 0.07, 0.00) complex( 2.45, 0.00)

h =
complex(-149.00, 1.00) complex(42.20, 0.00) complex(-156.32, 0.00)
complex(-537.68, 0.00) complex(152.55, 0.00) complex(-554.93, 0.00)
complex( 0.00, 0.00) complex( 0.07, 0.00) complex( 2.45, 0.00)

p =
complex( 1.00, 0.00) complex( 0.00, 0.00) complex( 0.00, 0.00)
complex( 0.00, 0.00) complex(-1.00, 0.00) complex( 0.05, 0.00)
complex( 0.00, 0.00) complex( 0.05, 0.00) complex( 1.00, 0.00)

transpose(p)*p =
complex( 1.00, 0.00) complex( 0.00, 0.00) complex( 0.00, 0.00)
complex( 0.00, 0.00) complex( 1.00, 0.00) complex( 0.00, 0.00)
complex( 0.00, 0.00) complex( 0.00, 0.00) complex( 1.00, 0.00)

conj(transpose(p))*a*p - h =
complex( 0.00,-0.00) complex( 0.00,-0.00) complex( 0.00,-0.00)
complex( 0.00,-0.00) complex( 0.00,-0.00) complex( 0.00,-0.00)
complex( 0.00,-0.00) complex( 0.00,-0.00) complex(-0.00,-0.00)

```

See Also

eigensystem().

References

E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

histogram

Synopsis**#include** <numeric.h>**int** histogram(array double &y, array double x[&], ... /* [array double hist[:]] */);**Syntax****histogram**(y, x)**histogram**(y, x, hist)**Purpose**

Calculate and plot histograms.

Return Value

This function returns 0 on success and -1 on failure.

Parameters*y* An input array of data set.*x* An input array which contains the bins of the histogram.*hist* An output array which contains the calculated data of the histogram.**Description**

This function calculates and plots the histogram of data set *y* with the bins defined in array *x*. The array of data set *y* can be of any dimensions. If the function is called without the optional argument *hist*, the histogram will be plotted. Otherwise the function only outputs the data of histogram without plotting.

Example1

The histograms of the data set generated from function $\sin(y)$ are calculated and plotted in this example. Note that the histograms are the same for the data set *y1* and *y2* due to the symmetric nature of the $\sin()$ function, even though their periods of the data are different.

```
#define N 300
#define N1 10
#define N2 30
#define M 21

#include<numeric.h>
int main() {
    array double y1[N], x[M], hist[M];
    array double y2[N1][N2];

    linspace(x, -1, 1);
    linspace(y1, 0, 2*M_PI);
    linspace(y2, 0, 4*M_PI);

    y1 = sin(y1);
    y2 = sin(y2);
    histogram(y1, x); // chplot.histogram
    histogram(y2, x); // chplot.histogram

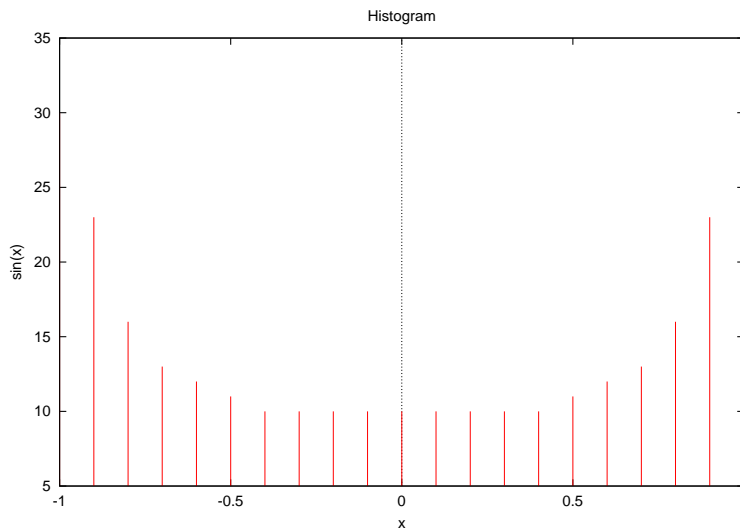
    histogram(y1, x, hist);
```

```

printf("x hist = %f %f\n", x, hist);
histogram(y2, x, hist);
printf("x hist = %f %f\n", x, hist);
}

```

Output1



```

x hist = -1.000000 -0.900000 -0.800000 -0.700000 -0.600000 -0.500000 -0.400000 -0.300000
-0.200000 -0.100000 0.000000 0.100000 0.200000 0.300000 0.400000 0.500000 0.600000
0.700000 0.800000 0.900000 1.000000
30.000000 23.000000 16.000000 13.000000 12.000000 11.000000 10.000000 10.000000
10.000000 10.000000 10.000000 10.000000 10.000000 10.000000 10.000000 11.000000
12.000000 13.000000 16.000000 23.000000 30.000000

```

```

x hist = -1.000000 -0.900000 -0.800000 -0.700000 -0.600000 -0.500000 -0.400000 -0.300000
-0.200000 -0.100000 0.000000 0.100000 0.200000 0.300000 0.400000 0.500000 0.600000
0.700000 0.800000 0.900000 1.000000
30.000000 23.000000 16.000000 13.000000 12.000000 11.000000 10.000000 10.000000
10.000000 10.000000 10.000000 10.000000 10.000000 10.000000 10.000000 11.000000
12.000000 13.000000 16.000000 23.000000 30.000000

```

Example2

The histograms of the same data sets $y1$ and $y2$ as in example 1 are calculated in this example. The histograms are drawn by function `plotxy()` according to the data calculated by function `histogram()`. The histograms are the same as that of example 1.

```

#define N 300
#define N1 10
#define N2 30
#define M 21

#include <numeric.h>
#include <chplot.h>
int main() {
    array double y1[N], x[M], hist[M];

```

```
array double y2[N1][N2];
class CPlot plot;

linspace(x, -1, 1);
linspace(y1, 0, 2*M_PI);
linspace(y2, 0, 4*M_PI);

y1 = sin(y1);
y2 = sin(y2);

histogram(y1, x, hist);
plotxy(x, hist, "Histogram", "x", "sin(x)", &plot);
plot.axisRange(PLOT_AXIS_Y, min(hist)-5, max(hist)+5);
plot.plotType(PLOT_PLOTTYPE_IMPULSES, 0);
plot.plotting();

histogram(y2, x, hist);
plotxy(x, hist, "Histogram", "x", "sin(x)", &plot);
plot.axisRange(PLOT_AXIS_Y, min(hist)-5, max(hist)+5);
plot.plotType(PLOT_PLOTTYPE_IMPULSES, 0);
plot.plotting();
}
```

Output2

The output histograms are the same as that of example 1.

See Also

plotxy().

References

householdermatrix

Synopsis

```
#include <numeric.h>
```

```
int householdermatrix(array double complex x[&], array double complex v[&], ...
                      /* [double *beta] */);
```

Syntax

```
householdermatrix(x, v)
```

```
householdermatrix(x, v, beta)
```

Purpose

Get the Householder matrix.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x Input vector of *n* elements.

v Output vector of *n* elements.

beta Optional output value of pointer to double.

Description

This function passes a vector *x* as an input argument and gets the vector *v* and optional output value *beta* such that

$$H = I - \text{beta} * v * v^T$$

where *H* is a Householder matrix, and *I* is an identity matrix. A Householder matrix *H* satisfies the equation

$$H * x = -\text{sign}(x[0]) * \text{norm}(x) * E$$

where vector $E = [1, 0, 0, \dots, 0]$ has *n* elements. If *x* is a complex vector, then $\text{sign}(x[0])$ is defined

$$\text{sign}(x[0]) = \frac{x[0]}{\text{abs}(x[0])}$$

Example

```
#include <numeric.h>
int main() {
    array double complex x[5] = {complex(-0.3, 0.5), 54, 25.3, 25.46, 83.47};
    array double complex y[5], h[5][5];
    array double x1[5] = {-0.3, 54, 25.3, 25.46, 83.47};
    array double e[5]={1,0,0,0,0};
    array double y1[5], h1[5][5];
    double beta;

    householdermatrix(x1,y1);
    printf("x1=\n%5.2f", x1);
```

```

printf("y1=\n%5.2f", y1);
householdermatrix(x1, y1, &beta);
h1 = identitymatrix(5) - beta*y1*transpose(y1);
printf("y1=\n%5.2f", y1);
printf("beta=%5.2f\n", beta);
printf("h1*x1+sign(x1[0])*norm(x1)*e =\n%5.2f", h1*x1+sign(x1[0])*norm(x1, "2")*e);

householdermatrix(x, y);
printf("x=\n%5.2f", x);
printf("y=\n%5.2f", y);
householdermatrix(x, y, &beta);
h = identitymatrix(5) - beta*y*conj(transpose(y));
printf("y=\n%5.2f", y);
printf("beta=%5.2f\n", beta);
printf("h*x+sign(x[0])*norm(x)*e =\n%5.2f", h*x+x[0]/abs(x[0])*norm(x, "2")*e);
}

```

Output

```

x1=
-0.30 54.00 25.30 25.46 83.47
y1=
-106.00 54.00 25.30 25.46 83.47
y1=
-106.00 54.00 25.30 25.46 83.47
beta= 0.00
h1*x1+sign(x1[0])*norm(x1)*e =
 0.00 0.00 0.00 -0.00 -0.00
x=
complex(-0.30, 0.50) complex(54.00, 0.00) complex(25.30, 0.00) complex(25.46, 0.00)
complex(83.47, 0.00)
y=
complex(-54.68, 91.13) complex(54.00, 0.00) complex(25.30, 0.00) complex(25.46, 0.00)
complex(83.47, 0.00)
y=
complex(-54.68, 91.13) complex(54.00, 0.00) complex(25.30, 0.00) complex(25.46, 0.00)
complex(83.47, 0.00)
beta= 0.00
h*x+sign(x[0])*norm(x)*e =
complex( 0.00, 0.00) complex( 0.00,-0.00) complex( 0.00,-0.00) complex( 0.00,-0.00)
complex(-0.00,-0.00)

```

References

G. H. Golub and C. F. van Loan, *Matrix Computations*, third edition, Johns Hopkins University Press, Baltimore, Maryland, 1996

identitymatrix

Synopsis

```
#include <numeric.h>
```

```
array double identitymatrix(int n)[:][:];
```

Purpose

Generate an identity matrix.

Return Value

This function returns the identity matrix.

Parameters

n Input integer.

Description

The function returns an $n \times n$ identity matrix.

Example

```
#include <numeric.h>
int main() {
    int n = 3;
    array double a[n][n];

    a = identitymatrix(n);
    printf("identitymatrix(n) =\n%f\n", a);
}
```

Output

```
identitymatrix(n) =
1.000000 0.000000 0.000000
0.000000 1.000000 0.000000
0.000000 0.000000 1.000000
```

See Also

rank().

References

ifft

Synopsis

```
#include <numeric.h>
```

```
int ifft(array double complex &y, array double complex &x, ... /* [int n [int dim[&]]] */);
```

Syntax

```
ifft(y, x)
```

```
ifft(y, x, n)
```

```
ifft(y, x, dim);
```

Purpose

N-dimensional inverse Fast Fourier Transform (FFT) calculation.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

y Result of inverse FFT, the array of the same dimension and size as *x*.

x An n-dimensional array used for calculating the inverse FFT.

n Optional argument, user specified inverse FFT points for one-dimensional data.

dim A one-dimensional array of optional arguments with int type. It contains the user specified inverse FFT points. Every element corresponds to a data dimension. For example, there is a three-dimensional data $x[M][N][L]$. The FFT points are specified m, n, l , respectively. Then the array *dim* is given values of $\text{dim}[0] = m, \text{dim}[1] = n, \text{dim}[2] = l$.

Description

The multi-dimensional **array** *x* can be of any supported arithmetic data type and size. Conversion of the data to **double complex** is performed internally. The same multi-dimensional, **double complex array** *y* contains the result of inverse fast Fourier transform. The result normalized by the total number of points of the transform is performed internally. The optional argument *n* or *dim[]* is used for specifying the number of points of the inverse FFT. If the user does not specify the number of points of FFT, it will be initialized according to the length of the input data.

Algorithm

see `fft()`.

Example

see `fft()`.

See Also

`fft()`.

References

R. C. Singleton, *An Algorithm for Computing the Mixed Radix F. F. T.*, IEEE Trans, Audio Electroacoust., AU-1(1969) 93-107.

MJ Oleson, *Netlib Repository at UTK and ORNL*, go/fft-olesen.tar.gz, <http://www.netlib.org>

integral1

Synopsis

```
#include <numeric.h>
```

```
double integral1(double (*func)(double), double x1, double x2, ... /* [double tol] */);
```

Syntax

```
integral1(func, x1, x2)
```

```
integral1(func, x1, x2, tol)
```

Purpose

Numerical integration of a function.

Return Value

This function returns an integral value of a function *func* integrated from *x1* to *x2*.

Parameters

func Function to be integrated.

x1 An initial point of integral.

x2 A final point of integral.

tol The user specified tolerance. If the user does not specify this optional value, the value of `10*FLT_EPSILON` is used by default, where `FLT_EPSILON` is defined in header file **float.h**.

Description

The function to be integrated, *func*, is specified as a pointer to a function that takes a **double** as an argument and returns a **double** functional value. *x1* and *x2* are the end-points of the range to be integrated. Conversion of the data to **double** are performed internally. The return value is a **double** data of integral evaluation. If the optional argument *tol* is specified, the integral uses the user specified tolerance to decide iteration stop. Otherwise the default value is used.

Algorithm

The following trapezoidal algorithm for numerical integration of *func* $f(x)$ is used

$$\int_{x_1}^{x_2} f(x)dx = h\left[\frac{1}{2}f_1 + \frac{1}{2}f_2\right] + O(h^3 f'')$$

where f'' represents the second derivative of the function $f(x)$ evaluated at a point within $x_1 \leq x \leq x_2$.

Example

Evaluate the integration

$$\int_0^{\frac{\pi}{2}} x^2(x^2 - 2) \sin(x) dx$$

```

#include <stdio.h>
#include <math.h>
#include <chplot.h>
#include <numeric.h>

/* Test function */
double func(double x) {
    return x*x*(x*x-2.0)*sin(x);
}

/* Integral of test function */
double fint(double x) {
    return 4.0*x*(x*x-7.0)*sin(x)-(pow(x,4.0)-14.0*x*x+28.0)*cos(x);
}

int main() {
    double x1=0.0,x2=M_PI/2,s;
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    linspace(x, x1, x2);
    fevalarray(y, func, x);
    printf("Actual value of the integration is %12.6lf\n",fint(x2)-fint(x1));
    s=integrall(func,x1,x2);
    printf("Result from the function integrall is %11.6lf\n",s);

    plot.data2D(x, y);
    plot.data2D(x, y);
    plot.plotType(PLOT_PLOTTYPE_IMPULSES, 1);
    plot.plotting();
}

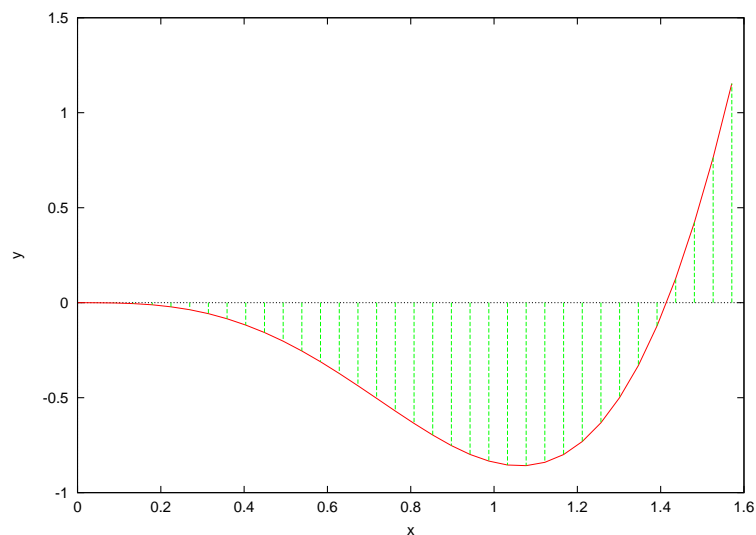
```

Output

```

Actual value of integraion is    -0.479159
Result from the function integration is    -0.479158

```



see also

integralp1(), **integral2()**, **integral3()**, **integration2()**, **integration3()**, **derivative()**.

References

William H. Press, et al, *Numerical Recipes in C*, second edition, Cambridge University Press, 1997.

integralp1

Synopsis

```
#include <numeric.h>
```

```
double integralp1(double (*func)(double), double x1, double x2, ... void *param, /* [double tol] */);
```

Syntax

```
integralp1(func, x1, x2, param)
```

```
integralp1(func, x1, x2, param, tol)
```

Purpose

Numerical integration of a function with a parameter.

Return Value

This function returns an integral value of a function *func()* with a parameter integrated from *x1* to *x2*.

Parameters

func Function to be integrated.

x1 An initial point of integral.

x2 A final point of integral.

param The parameter of the function.

tol The user specified tolerance. If the user does not specify this optional value, the value of `10*FLT_EPSILON` is used by default, where `FLT_EPSILON` is defined in header file **float.h**.

Description

The function to be integrated, *func*, is specified as a pointer to a function that takes a **double** as an argument and returns a **double** functional value. *x1* and *x2* are the end-points of the range to be integrated. The parameter for the function is passed as the argument `param`. Conversion of the data to **double** are performed internally. The return value is a **double** data of integral evaluation. If the optional argument *tol* is specified, the integral uses the user specified tolerance to decide iteration stop. Otherwise the default value is used.

Algorithm

The following trapezoidal algorithm for numerical integration of *func f(x)* is used

$$\int_{x_1}^{x_2} f(x)dx = h\left[\frac{1}{2}f_1 + \frac{1}{2}f_2\right] + O(h^3 f'')$$

where f'' represents the second derivative of the function $f(x)$ evaluated at a point within $x_1 \leq x \leq x_2$.

Example

Evaluate the integration

$$\int_0^{\frac{\pi}{2}} f(x, m)dx$$

for the function

$$f(x, mu) = x^2(x^2 - 2)\sin(x) + mu$$

with $mu = 2.5$ and $mu = 3.5$.

```
#include <stdio.h>
#include <math.h>
#include <numeric.h>

/* Test function */
double func(double x, void *param) {
    double mu;
    mu = *(double*)param;
    return x*x*(x*x-2.0)*sin(x) + mu;
}

int main() {
    double x1=0.0,x2=M_PI/2,s;
    double mu;

    mu = 2.5;
    s=integral1(func,x1,x2, &mu);
    printf("Result for the integraton of function f(x, 2.5) is %lf\n",s);
    mu = 3.5;
    s=integral1(func,x1,x2, &mu);
    printf("Result for the integraton of function f(x, 3.5) is %lf\n",s);
}
```

Output

```
Result for the integraton of function f(x, 2.5) is 3.447839
Result for the integraton of function f(x, 3.5) is 5.018636
```

see also

integral1(), integral2(), integral3(), integration2(), integration3(), derivative().

integral2

Synopsis

```
#include <numeric.h>
```

```
double integral2(double (*func)(double x, double y), double x1, double x2, double y1, double y2);
```

Syntax

```
integral2(func, x1, x2, y1, y2)
```

Purpose

Numerical integration of a function of two variables with constant limits for both x and y .

Return Value

This function returns an integral value of a function $func$ integrated from $y1$ to $y2$ in y and $x1$ to $x2$ in x .

Parameters

$func$ Function to be integrated.

$x1$ The lower limit of integration in x .

$x2$ The upper limit of integration in x .

$y1$ The lower limit of integration in y .

$y2$ The upper limit of integration in y .

Description

The function to be integrated, $func$, is specified as a pointer to a two-dimensional function that takes two arguments and returns the functional value of double type. The values $x1$ and $x2$ are the end-points of the range in x to be integrated. The values $y1$ and $y2$ are the upper and lower limits in y , respectively. The return value is the integral of double type.

Algorithm

The integration is carried out by (1) its lower and upper limits in y denoted as $y1$ and $y2$; (2) its lower and upper limits in x denoted as $x1$ and $x2$, respectively, so that

$$I = \int \int dx dy f(x, y) = \int_{x_1}^{x_2} dx \int_{y_1}^{y_2} dy f(x, y)$$

Example

Integrate $f(x, y) = \sin(x) \cos(y) + 1$ over the limits $0 \leq x \leq \pi$ and $-\pi \leq y \leq \pi$ by

$$I = \int_0^\pi \int_{-\pi}^\pi (\sin(x) * \cos(y) + 1) dx dy$$

```
#include <math.h>
#include <numeric.h>
```

```
double func(double x, double y) {
    return sin(x) * cos(y) + 1;
}
```



```
}
int main() {
    double x1 = 0, x2 = M_PI;
    double y1 = -M_PI, y2 = M_PI;
    double v;
    v = integral2(func,x1, x2, y1,y2);
    printf("integral2() gives %f\n", v);
}
```

Output

```
integral2() gives 19.739212
```

see also

integral1(), **integral3()**, **integration2()**, **integration3()**, **derivative()**.

References

William H. Press, et al, *Numerical Recipes in C*, second edition, Cambridge University Press, 1997.

integral3

Synopsis

```
#include <numeric.h>
```

```
double integral3(double (*func)(double x, double y, double z), double x1, double x2, double y1,
                double y2, double z1, double z2);
```

Syntax

```
integration3(func, x1, x2, y1, y2, z1, z2)
```

Purpose

Numerical integration of a function of three-dimensions with constant limits in x , y and z .

Return Value

This function returns an integral value of a function $func$ integrated from $z1$ to $z2$ in z , $y1$ to $y2$ in y and $x1$ to $x2$ in x .

Parameters

$func$ Function to be integrated.

$x1$ The lower limit of integration in x .

$x2$ The upper limit of integration in x .

$y1$ The lower limit of integration in y .

$y2$ The upper limit of integration in y .

$z1$ The lower limit of integration in z .

$z2$ The upper limit of integration in z .

Description

The function to be integrated, $func$, is specified as a pointer to a three-dimensional function that takes three arguments and returns a functional value of double type. The values $y1$ and $y2$ are the upper and lower limits in y , the values $z1$ and $z2$ are the upper and lower limits in z , and the values $x1$ and $x2$ are the end-points of the range in x to be integrated, respectively. The return value is the integral of double type.

Algorithm

The integration is carried out by (1) its lower and upper limits in z , denoted $z1$ and $z2$, (2) its lower and upper limits in y , denoted $y1$ and $y2$; and (3) its lower and upper limits in x , which we will denote $x1$ and $x2$, so that

$$I = \int \int \int dx dy dz f(x, y, z) = \int_{x_1}^{x_2} dx \int_{y_1}^{y_2} dy \int_{z_1}^{z_2} dz f(x, y, z)$$

Example

Integrate $f(x, y, z) = \sin(x) \cos(y) \sin(z) + 1$ over the limits of $0 \leq x \leq \pi$, $-\pi \leq y \leq \pi$ and $0 \leq z \leq \pi$.

$$I = \int_0^\pi \int_{-\pi}^\pi \int_0^\pi (\sin(x) \cos(y) \sin(z) + 1) dx dy dz$$

```
#include <math.h>
#include <numeric.h>

double func(double x,double y, double z) {
    return sin(x)*cos(y)*sin(z)+1;
}

int main() {
    double x1 = 0, x2 = M_PI;
    double y1 = -M_PI, y2 = M_PI;
    double z1 = 0, z2 = M_PI;
    double v;
    v = integral3(func, x1,x2, y1, y2, z1, z2);
    printf("integral3() gives %f\n", v);
}
```

Output

```
integral3() gives 62.012571
```

see also

integral1(), integral2(), integral3(), integration2(), integration3(), derivative().

References

William H. Press, et al, *Numerical Recipes in C*, second edition, Cambridge University Press, 1997.

integration2

Synopsis

```
#include <numeric.h>
```

```
double integration2(double (*func)(double x, double y), double x1, double x2, double (*y1)(double x),
                   double (*y2)(double x));
```

Syntax

```
integration2(func, x1, x2, y1, y2)
```

Purpose

Numerical integration of a function of two variables.

Return Value

This function returns an integral value of a function *func* integrated from $y_1(x)$ to $y_2(x)$ and $x1$ to $x2$ in x .

Parameters

func Function to be integrated.

x1 Function for the lower limit of integration in x .

x2 Function for the upper limit of integration in x .

y1 Function for the lower limit of integration in y .

y2 Function for the upper limit of integration in y .

Description

The function to be integrated, *func*, is specified as a pointer to a two-dimensional function that takes two arguments and returns the functional value of double type. The values $x1$ and $x2$ are the end-points of the range in x to be integrated. The values $y_1(x)$ and $y_2(x)$ are user-supplied functions for the upper and lower limits in y . The return value is the integral of double type.

Algorithm

The integration is carried out by (1) its lower and upper limits in y , denoted as $y1(x)$ and $y2(x)$; (2) its lower and upper limits in x , denoted as $x1$ and $x2$, respectively, so that

$$I = \int \int dx dy f(x, y) = \int_{x_1}^{x_2} dx \int_{y_1(x)}^{y_2(x)} dy f(x, y)$$

Now we can define a function $H(x, y)$ that does the innermost integral,

$$H(x, y) = \int_{y_1(x)}^{y_2(x)} f(x, y) dy$$

and finally our answer as an integral over $H(x)$

$$I = \int_{x_1}^{x_2} H(x) dx$$

Example

Integrate r^2 over a circular area with a radius of $r = 2$.

$$I = \int_{-r}^r \int_{-\sqrt{r^2-x^2}}^{\sqrt{r^2-x^2}} (x^2 + y^2) dx dy = \int_0^r \int_0^\pi \rho^2 d\theta d\rho = \frac{2\pi r^4}{4}$$

```
#include <stdio.h>
#include <math.h>
#include <numeric.h>

double r = 2;
double func(double x,double y) {
    return x*x+y*y;
}
double y1(double x) {
    return -sqrt(r*r-x*x);
}
double y2(double x) {
    return sqrt(r*r-x*x);
}

int main() {
    double x1=-r, x2=r, s;
    s=integration2(func,x1,x2, y1,y2);
    printf("integration2() = %f\n", s);
    printf("actual integral = %f\n", M_PI*pow(r,4.0)/2.0);
}
```

Output

```
integration2() = 25.156167
actual integral = 25.132744
```

see also

integral1(), **integral2()**, **integral3()**, **integration3()**, **derivative()**.

References

William H. Press, et al, *Numerical Recipes in C*, second edition, Cambridge University Press, 1997.

integration3

Synopsis

```
#include <numeric.h>
```

```
double integration3(double (*func)(double x, double y, double z), double x1, double x2,
                   double (*y1)(double x), double (*y2)(double x),
                   double (*z1)(double x, double y), double (*z2)(double x, double y));
```

Syntax

```
integration3(func, x1, x2, y1, y2, z1, z2)
```

Purpose

Numerical integration of a function of three-dimensions.

Return Value

This function returns an integral value of a function *func* integrated from *z1* to *z2* in *z*, *y1* to *y2* in *y* and *x1* to *x2* in *x*.

Parameters

func Function to be integrated.

x1 Function for the lower limit of integration in *x*.

x2 Function for the upper limit of integration in *x*.

y1 Function for the lower limit of integration in *y*.

y2 Function for the upper limit of integration in *y*.

z1 Function for the lower limit of integration in *z*.

z2 Function for the upper limit of integration in *z*.

Description

The function to be integrated, *func*, is specified as a pointer to a three-dimensional function that takes three arguments and returns a functional value of double type. The values $y_1(x)$ and $y_2(x)$ are user-supplied functions for the upper and lower limits in *y*, the values $z_1(x)$ and $z_2(x)$ are user-supplied functions for the upper and lower limits in *z*, and the values *x1* and *x2* are the end-points of the range in *x* to be integrated, respectively. The return value is the integral of double type.

Algorithm

The integration is carried out by (1) its lower and upper limits in *z* at specified *x* and *y*, denoted $z1(x,y)$ and $z2(x,y)$, (2) its lower and upper limits in *y*, denoted $y1(x)$ and $y2(x)$; (3) its lower and upper limits in *x*, which we will denote *x1* and *x2*, so that

$$I = \int \int \int dx dy dz f(x, y, z) = \int_{x_1}^{x_2} dx \int_{y_1(x)}^{y_2(x)} dy \int_{z_1(x,y)}^{z_2(x,y)} dz f(x, y, z)$$

Now we can define a function $G(x,y)$ that does the innermost integral,

$$G(x,y) = \int_{z_1(x,y)}^{z_2(x,y)} f(x,y,z)dz$$

and a function $H(x)$ that does the integral of $G(x,y)$,

$$H(x,y) = \int_{y_1(x)}^{y_2(x)} G(x,y)dy$$

and finally our answer as an integral over $H(x)$

$$I = \int_{x_1}^{x_2} H(x)dx$$

Example

Integrate r^2 over a spherical volume with a radius of $r = 2$.

$$I = \int_{-r}^r \int_{-\sqrt{r^2-x^2}}^{\sqrt{r^2-x^2}} \int_{-\sqrt{r^2-x^2-y^2}}^{\sqrt{r^2-x^2-y^2}} (x^2 + y^2 + z^2) dx dy dz = \int_0^{2\pi} \int_0^\pi \int_0^r \rho^4 \sin(\theta) d\rho d\theta d\phi = \frac{4\pi r^5}{5}$$

```
#include <stdio.h>
#include <math.h>
#include <numeric.h>

double r;
double func(double x,double y,double z) {
    return x*x+y*y+z*z;
}
double y1(double x) {
    return -sqrt(r*r-x*x);
}
double y2(double x) {
    return sqrt(r*r-x*x);
}
double z1(double x,double y) {
    return -sqrt(r*r-x*x-y*y);
}
double z2(double x,double y) {
    return sqrt(r*r-x*x-y*y);
}

int main() {
    double x1=-2, x2= 2, s;
    r = 2;
    s=integration3(func,x1,x2,y1,y2,z1,z2);
    printf("integration3() = %f\n", s);
    printf("actual integral = %f\n", 4.0*M_PI*pow(r,5.0)/5.0);
}
```

Output

```
integration3() = 80.486994
actual integral = 80.424781
```

see also

integral1(), integral2(), integral3(), integration2(), derivative().

References

William H. Press, et al, *Numerical Recipes in C*, second edition, Cambridge University Press, 1997.

interp1

Synopsis

```
#include <numeric.h>
```

```
int interp1(double y[&], double x[&], double xa[&], double ya[&], char *method);
```

Purpose

This function finds the values of the function, which is expressed in terms of two arrays *xa* and *ya*, at points expressed in array *x* by linear or cubic spline interpolation.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

y An output array of function values.

x An input array of variable values.

ya An array which contains function values corresponding to *xa*.

xa An array which contains tabulated *x* values of the function.

method A string which specifies the method of interpolation. The string "linear" or "spline" is for linear or spline interpolation, respectively.

Description

This function takes two arrays *xa* and *ya* of the same length that express a known function, and calculates the values, at some other points *x* by linear or cubic spline interpolation. The interpolation method can be chosen by input argument *method* of string with "linear" or "spline".

Example

Calculate function values of $\sin(x)$ by linear and cubic spline interpolation.

```
#include <stdio.h>
#include <numeric.h>
#define N 10
#define NUM_X 5

int main() {
    int i,nfunc;
    array double f[NUM_X],x[NUM_X],y[NUM_X];
    array float xa[N],ya[N];

    /* Generation of interpolation tables of sin(x) function */
    linspace(xa, 0, M_PI);
    ya = sin(xa);
    linspace(x, 0.15, M_PI-0.15);
    f=sin(x);

    /*Spline interpolation */
    printf("\nCubic spline interpolation of function sin(x)\n");
    printf("\n%9s %13s %17s\n", "x", "f(x)", "interpolation");
    interp1(y, x, xa, ya, "spline");
```

```

for (i=0; i<NUM_X; i++)
    printf("%12.6f %12.6f %12.6f\n", x[i], f[i], y[i]);

/*linear interpolation */
printf("\nlinear interpolation of function sin(x)\n");
printf("\n%9s %13s %17s\n", "x", "f(x)", "interpolation");
interp1(y, x, xa, ya, "linear");
for (i=0; i<NUM_X; i++)
    printf("%12.6f %12.6f %12.6f\n", x[i], f[i], y[i]);
}

```

Output

Cubic spline interpolation of function sin(x)

x	f(x)	interpolation
0.150000	0.149438	0.149431
0.860398	0.758102	0.758072
1.570797	1.000000	0.999960
2.281195	0.758102	0.758072
2.991593	0.149438	0.149430

linear interpolation of function sin(x)

x	f(x)	interpolation
0.150000	0.149438	0.146972
0.860398	0.758102	0.746562
1.570797	1.000000	0.984808
2.281195	0.758102	0.746562
2.991593	0.149438	0.146972

See Also

interp2(), **CSpline::Interp()**, **CSpline::Interpm()**.

References

William H. Press, Saul A. Teukolsky, William T. Vetterling and Brian P. Flannery, *Numeric Recipes in C*, Second Edition, Cambridge University Press, 1997.

interp2

Synopsis

```
#include <numeric.h>
```

```
int interp2(double z[&][&], double x[&], double y[&], double xa[&], double ya[&],
            double za[&][&], char *method);
```

Purpose

This function finds the values of a two-dimensional function, at points indicated by two one-dimensional arrays x and y , by two dimensional linear or cubic spline interpolation. The function is expressed in terms of tabulated values.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

z An output two-dimensional matrix which contains calculated function values

x An array of x values at which the function values will be calculated by interpolation.

y An array of y values at which the function values will be calculated by interpolation.

xa An input one-dimensional array which contains tabulated values of the first variable.

ya An input one-dimensional array which contains tabulated values of the second variable.

za An input two-dimensional matrix which contains function values corresponding to xa and ya .

$method$ A string which specifies the method of interpolation. The string "linear" and "spline" is for linear or spline interpolation, respectively.

Description

This function takes two arrays, xa of dimension m , and ya of dimension n , and a matrix of function value za of dimension $m \times n$ tabulated at the grid points defined by xa and ya , and calculates the values of the function, at points defined by arrays of x and y by linear or cubic spline interpolation. The dimensions for arrays xa , ya , x and y can be different.

Example

Interpolate a two-dimensional function

$$f(x, y) = 3(1 - x)^2 e^{-x^2 - y^2 + 1} - 10\left(\frac{x}{5} - x^3 - y^5\right) e^{-x^2 - y^2} - \frac{1}{3} e^{-(x+1)^2 - y^2}$$

by linear and cubic spline interpolations.

```
#include <stdio.h>
#include <chplot.h>
#include <numeric.h>
#define M 15
#define N 11
#define NUM_X 22
#define NUM_Y 22

int main() {
```

```

int i, j;
array double z_s1[NUM_X*NUM_Y], z_l1[NUM_X*NUM_Y];
array double x[NUM_X], y[NUM_Y], xa[M], ya[N];
array double za[M][N], zald[M*N], z_s[NUM_X][NUM_Y], z_l[NUM_X][NUM_Y];
class CPlot plot;

/* Construct data set of the peaks function */
linspace(xa, -3, 3);
linspace(ya, -4, 4);
for(i=0; i<M; i++) {
    for(j=0; j<N; j++) {
        za[i][j] = 3*(1-xa[i])*(1-xa[i])*
            exp(-(xa[i]*xa[i])-(ya[j]+1)*(ya[j]+1))
            - 10*(xa[i]/5 - xa[i]*xa[i]*xa[i]-
            pow(ya[j],5))*exp(-xa[i]*xa[i]-ya[j]*ya[j])
            - 1/3*exp(-(xa[i]+1)*(xa[i]+1)-ya[j]*ya[j]);
    }
}
zald = (array double[M*N])za;

linspace(x, -2.9, 2.9);
linspace(y, -3.9, 3.9);

/* test 2-dimensional cubic spline interpolation*/
interp2(z_s,x,y,xa,ya,za,"spline");

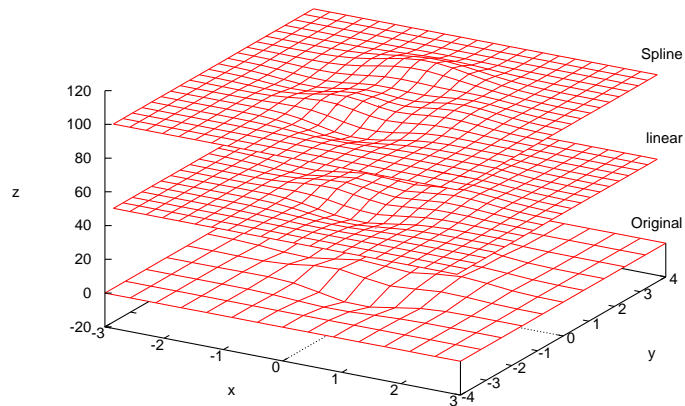
/* test 2-dimensional linear interpolation */
interp2(z_l,x,y,xa,ya,za,"linear");

/* add offset for display */
z_l1 = (array double[NUM_X*NUM_Y])z_l + (array double[NUM_X*NUM_Y])50;
z_s1 = (array double[NUM_X*NUM_Y])z_s + (array double[NUM_X*NUM_Y])100;

plot.data3D(xa, ya, zald);
plot.data3D(x, y, z_s1);
plot.data3D(x, y, z_l1);
plot.label(PLOT_AXIS_X, "x");
plot.label(PLOT_AXIS_Y, "y");
plot.label(PLOT_AXIS_Z, "z");
plot.ticsLevel(0);
plot.text("Spline", PLOT_TEXT_RIGHT, 3.5, 3.5, 120);
plot.text("linear", PLOT_TEXT_RIGHT, 3.5, 3.5, 70);
plot.text("Original", PLOT_TEXT_RIGHT, 3.5, 3.5, 20);
plot.plotType(PLOT_PLOTTYPE_LINES, 0, 1, 1);
plot.plotType(PLOT_PLOTTYPE_LINES, 1, 1, 1);
plot.plotType(PLOT_PLOTTYPE_LINES, 2, 1, 1);
plot.colorBox(PLOT_OFF);
plot.plotting();
}

```

Output

**See Also**

interp1(), **CSpline::Interp()**, **CSpline::Interpm()**.

References

William H. Press, Saul A. Teukolsky, William T. Vetterling and Brian P. Flannery, *Numeric Recipes in C*, Second Edition, Cambridge University Press, 1997.

inverse

Synopsis

```
#include <numeric.h>
```

```
array double inverse(array double a[:][:], ... /* [int *status] */)[:][:];
```

Syntax

```
inverse(a)
```

```
inverse(a, status)
```

Purpose

Calculate the inverse of a square matrix of double type.

Return Value

This function returns the inverse matrix.

Parameters

a Input square matrix.

status Output integer indicating the status of calculation.

Description

This function calculates the inverse matrix of a square matrix. If calculation is successful, *status* = 0, otherwise *status* ≠ 0. To calculate the inverse of a matrix of complex numbers, use function **cinverse**().

Example1

```
#include <numeric.h>
int main() {
    array double a[2][2] = {2, 4,
                           3, 7};

    array double inv[2][2];
    int status;

    inv = inverse(a);
    printf("inverse(a) =\n%f\n", inv);
    printf("inverse(a)*a =\n%f\n", inv*a);
    printf("a*inverse(a) =\n%f\n", a*inv);

    inv = inverse(a, &status);
    if(status == 0)
        printf("inverse(a, &status) = %f\n", inv);
    else
        printf("error: numerical error in inverse()\n");
}
```

Output1

```
inverse(a) =
3.500000 -2.000000
-1.500000 1.000000
```

```
inverse(a)*a =
1.000000 0.000000
0.000000 1.000000

a*inverse(a) =
1.000000 0.000000
-0.000000 1.000000

inverse(a, &status) = 3.500000 -2.000000
-1.500000 1.000000
```

Example2

```
#include <numeric.h>
array double func(array double B[2][2])[2][2] {
    return inverse(B);
}

int main() {
    array double a[2][2] = {2, 4,
                           3, 7};
    array double inv[2][2];
    int status;

    inv = func(a);
    printf("inverse(a) =\n%f\n", inv);
    printf("inverse(a)*a =\n%f\n", inv*a);
    printf("a*inverse(a) =\n%f\n", a*inv);
}
```

Output2

```
inverse(a) =
3.500000 -2.000000
-1.500000 1.000000

inverse(a)*a =
1.000000 0.000000
0.000000 1.000000

a*inverse(a) =
1.000000 0.000000
-0.000000 1.000000
```

See Also

cinverse(), **ludcomp()**, **linsolve()**.

References

E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

isprime

Synopsis

```
#include <numeric.h>
int isprime(int x);
```

Purpose

Determines if the argument is a prime number.

Return Value

The **isprime** macro returns 1 value if and only if its argument is a prime number.

Parameters

x Argument.

Description

The **isprime** macro determines whether its argument is a prime number).

Example

```
#include <numeric.h>

int n = 17;
if(isprime(n))
    printf("%d is a prime number\n", n);
else
    printf("%d is not a prime number\n", n);

n = 18;
if(isprime(n))
    printf("%d is a prime number\n", n);
else
    printf("%d is not a prime number\n", n);
```

Output

```
17 is a prime number
18 is not a prime number
```

See Also

lcm()

Synopsis

```
#include <numeric.h>
int lcm(int num1, int num2);
```

Purpose

Obtain the least common multiple of two integers.
arrays of integer type.

Return Value

This function returns 0 on success and -1 on failure if one of two integer arguments is less or equal to 0.

Parameters

num1 An integer.

num2 Another integer.

g The least common multiple of two integers.

Description

This function calculates the least common multiple of two integers *num1* and *num2*. If one of two integer arguments is less or equal to 0, the function returns -1.

Example

```
/* File: lcm.ch */
#include <numeric.h>
int num1 = 54;
int num2 = 24;

printf("The divisors of %d are\n%d", num1, divisor(num1));
printf("The divisors of %d are\n%d", num2, divisor(num2));
printf("The common divisors of %d and %d are %d", num1, num2, cdivisor(num1, num2));
printf("The least common multiple of %d and %d is %d\n", num1, num2, lcm(num1, num2));
```

Output

```
The divisors of 54 are
1 2 3 6 9 18 27 54
The divisors of 24 are
1 2 3 4 6 8 12 24
The common divisors of 54 and 24 are 1 2 3 6
The least common multiple of 54 and 24 is 216
```

See Also

`gcd()`, `gcdarray()`, `lcmarray()`.

lcmarray()

Synopsis

```
#include <numeric.h>
int lcmarray(array int &g, array int &u, array int &v);
```

Syntax

```
lcmarray(g, u, v)
```

Purpose

Obtain the least common multiple of the corresponding elements of two arrays of integer type.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

g An output array which is the same dimension and size as *u*. It contains the result of the least common multiple calculation of *u* and *v*.

u Any-dimensional array which contains positive integer elements for the least common multiple calculation.

v An array which is the same dimension and size as *u*. It contains another positive integer elements for the least common multiple calculation.

Description

This function calculates the least common multiple of corresponding elements *u* and *v*. The arrays *u* and *v* shall be the same size and contains positive integer values. The output array *g* is a positive integer array of same size as *u*.

Example

```
#include <numeric.h>
int main() {
    array int u[2][3] = {1,2,7,15,3,4};
    array int v[2][3] = {2,4,8,3,8,3};
    array int l[2][3];
    lcmarray(l,u,v);
    printf("l=%d",l);
}
```

Output

```
g=2 4 56
15 24 12
```

See Also

`lcmarray()`, `gcd()`, `gcdarray()`.

References

Knuth, Donald Ervin, *The art of computer programming*, Vol. 2, Addison-Wesley Pub. Co., 1973.

lindata

Synopsis

```
#include <numeric.h>
```

```
int lindata(double first, double last, ... /* [[array] type a[:]...[:]] [type *a, int n] */);
```

Syntax

```
lindata(first, last, a);
```

```
lindata(first, last, a, n);
```

Purpose

Get linearly spaced data for an array passed from the third argument.

Return Value

This function returns the number of elements in the passed array *a*.

Parameters

first Starting point of the linearly spaced array.

last Ending point of the linearly spaced array.

a An output passed from the third argument with linearly spaced data. It shall be an array or pointer to arithmetic type.

n An input passed from the fourth argument for the number of elements.

Description

This function obtains a linearly spaced values starting with *first* and ending with *last* for the array *a* passed from the third argument. The number of points is taken internally from the passed array *a*. If the third argument is a pointer to arithmetic type, the fourth argument contains the number of elements for the passed object in the third argument.

Example

```
#include <numeric.h>

int main () {
    double a[6], *p;
    array float b[6];
    array int c[6];
    array double aa[6], d[2][3];

    lindata(2, 12, a);
    printf("a[0] = %f\n", a[0]);
    aa = a;
    printf("aa = \n%f\n", aa);
    p = &aa[0];
    lindata(20, 120, p, 6);
    printf("aa = \n%f\n", aa);
    lindata(2, 12, b);
    printf("b = \n%f\n", b);
    lindata(2, 12, c);
    printf("c = \n%d\n", c);
    lindata(2, 12, d);
    printf("d = \n%f\n", d);
}
```

Output

```
a[0] = 2.000000
aa =
2.000000 4.000000 6.000000 8.000000 10.000000 12.000000

aa =
20.000000 40.000000 60.000000 80.000000 100.000000 120.000000

b =
2.000000 4.000000 6.000000 8.000000 10.000000 12.000000

c =
2 4 6 8 10 12

d =
2.000000 4.000000 6.000000
8.000000 10.000000 12.000000
```

See Also
logdata().

linearcorrcoef

Synopsis

```
#include <numeric.h>
```

```
double linearcorrcoef(array double x[:], array double y[:]);
```

Syntax

```
linearcorrcoef(x, y)
```

Purpose

Calculate the linear correlation coefficient between two variables.

Return Value

This function returns the linear correlation coefficient between two variables.

Parameters

x A vector of size $n \times n$. It contains the original data for correlation coefficient calculation.

y A vector of the same size as *x*. It contains the original data for correlation coefficient calculation.

Algorithm

This function is used to calculate the linear correlation coefficient. The linear correlation coefficient is defined as

$$c = \frac{n \sum_{i=1}^n xy - (\sum_{i=1}^n x)(\sum_{i=1}^n y)}{\sqrt{n(\sum_{i=1}^n x^2) - (\sum_{i=1}^n x)^2} \sqrt{n(\sum_{i=1}^n y^2) - (\sum_{i=1}^n y)^2}}$$

Example

Calculate the linear correlation coefficient.

```
/* File: linearcorrcoef.ch */
#include <numeric.h>

double x1[] = {-10, 20, 50, 80, 110};
double y1[] = {-23.33, -6.67, 10, 26.67, 43.33};
double x2[] = {3, 5, 6, 1, 2, 7, 1, 7};
double y2[] = {8, 9, 8, 5, 7, 9, 4, 10};
double r;

r = linearcorrcoef(x1, y1);
printf("Linear Correlation coefficient r = %.2lf\n", r);

r = linearcorrcoef(x2, y2);
printf("Linear Correlation coefficient r = %.2lf\n", r);
```

Output

```
Linear Correlation coefficient r = 1.00
Linear Correlation coefficient r = 0.89
```

See Also

corrcoef().

linearfit

Synopsis

```
#include <numeric.h>
```

```
int linearfit(double x[], double y[], double &m, double &b, double r[]);
```

Purpose

Fit a set of data points x, y to a linear function $y = mx + b$.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x An array which contains variable values of the data set.

y An array which contains function values of the data set.

m The slope m for $y = mx + b$.

b The intercept b for $y = mx + b$.

r The residual error for the linear fitting.

Description

Given a set of data points expressed in arrays x and y with the same dimension, fit the data set using the linear regression. The fitted coefficients are passed by arguments m and b .

Algorithm

Use the regression algorithm to find m and b for $y=mx+b$ described in the book *C for Engineers and Scientists: An Interpretive Approach* by Harry H. Cheng, published by McGraw-Hill, 2009, ISBN: 978-0-07-337605-9.

Example 1

```
/* File: linearfit.ch
   Find the linear relation between the hours studied and quiz scores using
   linear regression */
#include <chplot.h>
CPlot plot;
array double x[8] = {3, 5, 6, 1, 2, 7, 1, 7};
array double y[8] = {9, 9, 8, 5, 7, 9, 4, 10};
array double r[8];
double m, b, x1, y1, x2, y2;

/* solve for m and b in y = mx+b */
linearfit(x, y, m, b, r);
printf("m = %.2lf\n", m);
printf("b = %.2lf\n", b);
printf("r = %.2lf\n", r);
```

```

/* Calculate y for x = 1 and x = 7 */
x1 = 1;
y1 = m*x1 + b;
printf("(x1, y1) = (%.21f, %.21f)\n", x1, y1);
x2 = 7;
y2 = m*x2 + b;
printf("(x2, y2) = (%.21f, %.21f)\n", x2, y2);

plot.title("Does Studying Help Quiz Score?");
plot.label(PLOT_AXIS_X, "Hours Studied");
plot.label(PLOT_AXIS_Y, "Quiz Scores");
plot.axisRange(PLOT_AXIS_X, 0, 8);
plot.axisRange(PLOT_AXIS_Y, 2, 12);
plot.line(x1, y1, x2, y2);
plot.scatter(x, y);
plot.plotting();

```

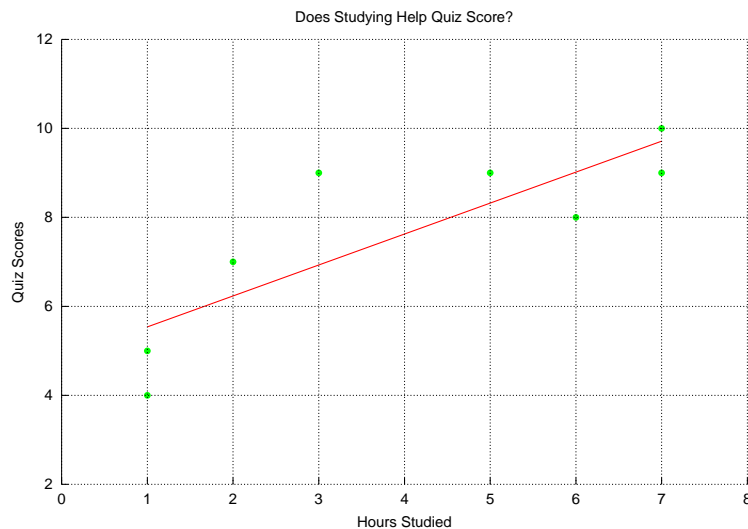
Output

```

m = 0.70
b = 4.84
r = 2.07 0.68 -1.02 -0.54 0.77 -0.71 -1.54 0.29

(x1, y1) = (1.00, 5.54)
(x2, y2) = (7.00, 9.71)

```



Example 2

```

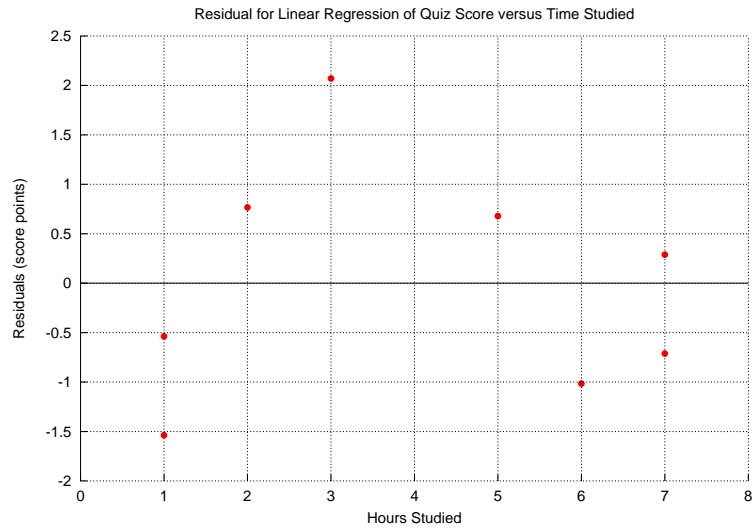
/* File: linearfit2.ch
   Find the residuals of a linear relation between the hours studied
   and quiz scores using linear regression */
#include <chplot.h>
CPlot plot;
array double x[8] = {3, 5, 6, 1, 2, 7, 1, 7};
array double y[8] = {9, 9, 8, 5, 7, 9, 4, 10};
array double r[8];
double m, b;

/* solve for m and b in y = mx+b */
linearfit(x, y, m, b, r);

```

```
printf("m = %.2lf\n", m);
printf("b = %.2lf\n", b);
printf("r = %.2lf\n", r);

plot.title("Residual for Linear Regression of Quiz Score versus Time Studied");
plot.label(PLOT_AXIS_X, "Hours Studied");
plot.label(PLOT_AXIS_Y, "Residuals (score points)");
plot.scatter(x, r);
plot.axisRange(PLOT_AXIS_X, 0, 8);
//plot.axisRange(PLOT_AXIS_Y, -3, 3);
plot.plotting();
```

**Output****See Also**

`curvefit()`, `expfit()`, `polyfit()`, `quadraticfit()`.

linrange

Synopsis

```
#include <numeric.h>
```

```
array double linrange(double first, double last, int n)[:];
```

Purpose

Generate a linearly spaced array of the column vector.

Return Value

This function returns an array a linearly spaced column vector with *n* elements.

Parameters

first Starting point of the linearly spaced array.

last Ending point of the linearly spaced array.

n The number of elements of the array.

Description

This function generates a linearly spaced array of *n* elements with the first element of *first* and the last element of *last*.

Example

```
#include <numeric.h>
int main() {
    array double x[10];
    array int y[10];
    int n = 10;

    x = linrange(1, 10, n);
    printf("x = %lf\n", x);
    y = linrange(1, 10, n);
    printf("y = %d\n", y);
}
```

Output

```
x = 1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000 9.000000
10.000000
```

```
y = 1 2 3 4 5 6 7 8 9 10
```

See Also

lindata(), **linspace()**, **logrange()**.

References

linsolve

Synopsis

```
#include <numeric.h>
```

```
int linsolve(array double x[], array double a[:][:], array double b[:]);
```

Purpose

Solve the linear system of equations by LU decomposition.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x Output array which contains the solution of equations.

a Input two-dimensional array contains the coefficients of a system of linear equations.

b Input one-dimensional array for linear equations.

Description

This function solves the system of linear equations by LU decomposition. The input matrix *a* should be n-by-n. The function returns 0 if there is a solution for the equation set, otherwise it returns -1.

It calculates the solution to a real/complex system of linear equations $a * X = b$, where *a* is an n-by-n matrix. The LU decomposition with partial pivoting and row interchanges is used to factor *a* as $a = P * L * U$, where *P* is a permutation matrix, *L* is unit lower triangular, and *U* is upper triangular. The factored form of *a* is then used to solve the system of equation $a * X = b$.

Example1

Solution of a linear system of equations with real coefficients.

```
#include <numeric.h>
int main() {
    array double a[3][3] = {3, 0, 6,
                           0, 2, 1,
                           1, 0, 1};
    array double b[3] = {2,
                        13,
                        25};
    array double x[3];
    int status;

    linsolve(x, a, b);
    printf("x =\n%f\n", x);

    status = linsolve(x, a, b);
    if(status == 0)
        printf("x =\n%f\n", x);
    else
        printf("error: numerical error in linsolve()\n");
}
```

Output1

```
x =  
49.333333 18.666667 -24.333333
```

```
x =  
49.333333 18.666667 -24.333333
```

See Also

linsolvesi(), **linsolvestd()**, **llsqsolve()**, **clinsolve()**, **inverse()**.

References

E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

linsolvesi

Synopsis

```
#include <numeric.h>
```

```
void linsolvesi(double m1, double b1, double m2, double b2, double &x, double &y);
```

Purpose

Solve for the solution of a system of two linear equations in slope-intercept form.

Return Value

None.

Parameters

m1 The slope of the first linear equation m_1 of $y = m_1x + b_1$.

b1 The y-intercept of the first linear equation m_1 of $y = m_1x + b_1$.

m2 The slope of the second linear equation m_2 of $y = m_2x + b_2$.

b2 The y-intercept of the second linear equation m_2 of $y = m_2x + b_2$.

x The x part of the solution (x, y).

y The y part of the solution (x, y).

Description

This function calculates the solution of a system of two linear equations in slope intercept form

$$y = m_1x + b_1 \quad (11.77)$$

$$y = m_2x + b_2 \quad (11.78)$$

The solutions (x, y) are passed back to the calling function through the last two arguments.

Example

```
/* File: linsolvesi.ch */
#include <numeric.h>
int main() {
    /* y = 2x + 4
       y = -0.5x - 2 */
    double m1 = 2, b1 = 4, m2 = -0.5, b2 = -2;
    double x, y;

    linsolvesi(m1, b1, m2, b2, x, y);
    printf("The solution for y = 2x + 4x, y = -0.5x - 2 is (x, y) = (%lf, %lf)\n", x, y);
}
```

Output

The solution for $y = 2x + 4x$, $y = -0.5x - 2$ is $(x, y) = (-2.400000, -0.800000)$

See Also

linsolvestd(), linsolve().

linsolvestd

Synopsis

```
#include <numeric.h>
```

```
void linsolvestd(double a1, double d1, double c1, double a2, double d2, double c2, double & x, double & y);
```

Purpose

Solve for the solution of a system of two linear equations in standard form.

Return Value

None.

Parameters

a1 The a_1 of the first linear equation $a_1x + d_1y = c_1$ in standard form.

d1 The d_1 of the first linear equation $a_1x + d_1y = c_1$ in standard form.

c1 The c_1 of the first linear equation $a_1x + d_1y = c_1$ in standard form.

a2 The a_2 of the first linear equation $a_2x + d_2y = c_2$ in standard form.

d2 The d_2 of the first linear equation $a_2x + d_2y = c_2$ in standard form.

c2 The c_2 of the first linear equation $a_2x + d_2y = c_2$ in standard form.

x The x part of the solution (x, y).

y The y part of the solution (x, y).

Description

This function calculates the solution of a system of two linear equations in standard form

$$a_1x + d_1y = c_1 \tag{11.79}$$

$$a_2x + d_2y = c_2 \tag{11.80}$$

The solutions (x, y) are passed back to the calling function through the last two arguments.

Example

```
/* File: linsolvestd.ch */
#include <numeric.h>
int main() {
    /* -2x + y = 4
```

```
    x + 2y = -4 */
double a1=-2, d1=1, c1 =4, a2=1, d2=2, c2=-4;
double x, y;

    linsolvestd(a1, d1, c1, a2, d2, c2, x, y);
printf("The solution for -2x + y = 4, x + 2y = -4 is (x, y) = (%lf, %lf)\n", x, y);
}
```

Output

The solution for -2x + y = 4, x + 2y = -4 is (x, y) = (-2.400000, -0.800000)

See Also

linsolvesi(), linsolve().

linspace

Synopsis

```
#include <numeric.h>
```

```
int linspace(array double &x, double first, double last);
```

Purpose

Get linearly spaced values for an array.

Return Value

This function returns the number of elements in array *x*.

Parameters

x An output with linearly spaced values.

first Starting point of the linearly spaced array.

last Ending point of the linearly spaced array.

Description

This function obtains a linearly spaced values starting with *first* and ending with *last* for the array *x*. The number of points is taken internally from the array *x*.

Note

`linspace()` is obsolete. Use `linspace()`.

Example

```
#include <numeric.h>
int main() {
    array double x[11], x2[2][3], x3[2][3][2];
    array float y[11];
    array int z[11];
    array unsigned int ui[11];
    array unsigned short uh[11];
    array short h[11];
    array unsigned char uc[11];
    array char c[11];
    int n;

    linspace(x, 1, sizeof(x)/sizeofelement(elementtype(x)));
    printf("x = %f\n", x);

    linspace(x2, 1, sizeof(x2)/sizeofelement(elementtype(x2)));
    printf("x2 = %f\n", x2);

    linspace(x3, 1, sizeof(x3)/sizeofelement(elementtype(x3)));
    printf("x3 = %f\n", x3);

    linspace(x, 0, M_PI);
    printf("x = %f\n", x);
    n = linspace(x, 0, M_PI);
    printf("n = %d\n", n);
```



```

linspace(y, 0, M_PI); /* y is float */
printf("y = %f\n", y);

linspace(z, 0, 90);
printf("z = %d\n", z);

linspace(ui, 0, 90);
printf("ui = %d\n", ui);

linspace(uh, 0, 90);
printf("uh = %d\n", uh);

linspace(h, 0, 90);
printf("h = %d\n", h);

linspace(c, 0, 90);
printf("c = %d\n", c);

linspace(uc, 0, 90);
printf("uc = %d\n", uc);
}

```

Output

```

x = 1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000 9.000000
10.000000 11.000000

```

```

x2 = 1.000000 2.000000 3.000000
4.000000 5.000000 6.000000

```

```

x3 = 1.000000 2.000000
3.000000 4.000000
5.000000 6.000000

```

```

7.000000 8.000000
9.000000 10.000000
11.000000 12.000000

```

```

x = 0.000000 0.314159 0.628319 0.942478 1.256637 1.570797 1.884956 2.199115 2.513274
2.827434 3.141593

```

```

n = 11
y = 0.000000 0.314159 0.628319 0.942478 1.256637 1.570796 1.884956 2.199115 2.513274
2.827434 3.141593

```

```

z = 0 9 18 27 36 45 54 63 72 81 90

```

```

ui = 0 9 18 27 36 45 54 63 72 81 90

```

```

uh = 0 9 18 27 36 45 54 63 72 81 90

```

```

h = 0 9 18 27 36 45 54 63 72 81 90

```

```

c = 0 9 18 27 36 45 54 63 72 81 90

```

```

uc = 0 9 18 27 36 45 54 63 72 81 90

```

See Also

logspace().

References

llsqcovsolve()

Synopsis

```
#include <numeric.h>
```

```
int llsqcovsolve(array double x[&], array double a[&][&], array double b[&],
                array double v[&][&], ... /* [array double p[&]] */);
```

Syntax

```
llsqcovsolve(a, b, v, x)
```

```
llsqcovsolve(a, b, v, x, p)
```

Purpose

Solve linear system of equations based on linear least-squares with known covariance.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

a A two-dimensional array size of $m \times n$ which contains input data. It is an overdetermined least-squares problem. So m must be larger than n .

b A vector with m elements. It contains input data.

v An input array of size $m \times m$. It contains covariance value.

x An output vector with n elements. It contains the linear least-squares results.

p An optional output vector with n elements. It contains the standard errors of x .

Description

This function calculates the vector x that satisfies $a * x = b + e$ where e is a normally distributed error with zero mean and covariance v using the linear least-squares method. This is an overdetermined linear least-squares problem. So the number of rows m must be larger than the number of column n . If optional output vector p is specified, it will pass out the standard errors of x .

Algorithm

The vector x can be calculated through the minimizes the quantity $(ax - b)^T v^{-1} (ax - b)$. The classical linear algebra solution to this problem is

$$x = (a^T v^{-1} a)^{-1} a^T v^{-1} b$$

and the standard errors of x is

$$\begin{aligned} \text{mse} &= b^T (v^{-1} - v^{-1} a (a^T v^{-1} a)^{-1} a^T v^{-1}) b / (m - n) \\ p &= \sqrt{\text{diagonal}(a^T v^{-1} a)^{-1} * \text{mse}} \end{aligned}$$

Example

```

#include <numeric.h>

int main() {
    array double a[4][3] = {1, 2, 3,
                           4, 5, 6,
                           7, 8, 9,
                           11,12,3}; /* a[m][n] */
    array double b[4] = {4,4,54,6};
    array double v[4][4] = { 1, 1, 1, 1,
                            1, 2, 3, 4,
                            1, 3, 6,10,
                            1, 4,10,20};

    array double x[3], dx[3];

    llsqcovsolve(x, a, b, v, dx);
    printf("a = \n%f", a);
    printf("b = \n%f", b);
    printf("v = \n%f", v);
    printf("x = \n%f", x);
    printf("a*x = \n%f", a*x);
    printf("dx = \n%f", dx);
}

```

Output

```

a =
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000
7.000000 8.000000 9.000000
11.000000 12.000000 3.000000
b =
4.000000 4.000000 54.000000 6.000000
v =
1.000000 1.000000 1.000000 1.000000
1.000000 2.000000 3.000000 4.000000
1.000000 3.000000 6.000000 10.000000
1.000000 4.000000 10.000000 20.000000
x =
10.800000 -25.600000 14.800000
a*x =
4.000000 4.000000 4.000000 -144.000000
dx =
61.237244 54.772256 5.270463

```

See Also

qrdecomp(), **llsqnonnegsolve()**, **llsqsolve()**.

References

Franklin A. Graybill, *Theory and Application of the Linear Model*, Duxbury Press, 1976.
 Gilbert Strang, *Introduction to Applied Mathematics*, Wellesley-Cambridge Press, 1986.

llsqnonnegsolve()

Synopsis

```
#include <numeric.h>
```

```
int llsqnonnegsolve(array double x[&], array double a[&][&], array double b[&], ...
                    /* [double tol, array double w[&]] */);
```

Syntax

```
llsqnonnegsolve(a, b, x)
```

```
llsqnonnegsolve(a, b, x, tol)
```

```
llsqnonnegsolve(a, b, x, tol, p)
```

Purpose

Solve linear system of equation with non-negative values based on linear least-squares method.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

a A two-dimensional array size of $m \times n$ which contains input data.

b A vector with m elements. It contains input data.

x An output vector with n elements. It contains the non-negative least squares results.

tol An optional input which specifies the tolerance of the solution. If the user does not specify this argument or specify zero, $tol = 10 * \max(m, n) * \text{norm}(u, "1") * \text{FLT_EPSILON}$ is used by default. `FLT_EPSILON` is defined in header file **float.h**.

w An optional output vector with n elements. It contains a vector where $w[i] < 0$ when $x[i] = 0$ and $w[i] \cong 0$ when $x[i] > 0$.

Description

This function solves the equations $ax = b$ using the linear least-squares method, while it is subject to the constraint that the solution vector x has non-negative elements. That is, $x[i] \geq 0$ for $i = 0, 1, \dots, n - 1$.

Algorithm

This function uses the algorithm described in the reference book, Chapter 23.

Example

```
#include <numeric.h>
int main () {

array double a[15][5]={-0.1340555,-0.2016283,-0.1693078,-0.1897199,-0.1738723,
                      -0.1037948,-0.1576634,-0.1334626,-0.1484855,-0.1359769,
                      -0.0877960,-0.1288387,-0.1068301,-0.1201180,-0.1093297,
                      0.02058554,0.00335331,-0.0164127,0.00078606,0.00271659,
                      -0.0324809,-0.0187680,0.00410639,-0.0140589,-0.0138439,
                      0.05967662,0.06667714,0.04352153,0.05740438,0.05024962,
```

```

    0.06712457,0.07352437,0.04489770,0.06471862,0.05876445,
    0.08687186,0.09368296,0.05672327,0.08141043,0.07302320,
    0.02149662,0.06222662,0.07213486,0.06200069,0.05570931,
    0.06687407,0.10344506,0.09153849,0.09508223,0.08393667,
    0.15879069,0.18088339,0.11540692,0.16160727,0.14796479,
    0.17842887,0.20361830,0.13057860,0.18385729,0.17005549,
    0.11414080,0.17259611,0.14816471,0.16007466,0.17005549,
    0.07846038,0.14669563,0.14365800,0.14003842,0.12571277,
    0.10803175,0.16994623,0.14971519,0.15885312,0.14301547},
    b[15]={-0.4361,-0.3437,-0.2657,-0.0392,0.0193,0.0747,0.0935,0.1079,0.1930,
           0.2058,0.2606,0.3142,0.3529,0.3615,0.3647};
    array double x[5], w[5];
    llsqnnonnegsolve(x,a,b,0.0,w);
    printf("a=%6.4f",a);
    printf("b=%6.4f",b);
    printf("x=%6.4f",x);
    printf("w=%6.4f",w);
}

```

Output

```

a=-0.1341 -0.2016 -0.1693 -0.1897 -0.1739
-0.1038 -0.1577 -0.1335 -0.1485 -0.1360
-0.0878 -0.1288 -0.1068 -0.1201 -0.1093
0.0206 0.0034 -0.0164 0.0008 0.0027
-0.0325 -0.0188 0.0041 -0.0141 -0.0138
0.0597 0.0667 0.0435 0.0574 0.0502
0.0671 0.0735 0.0449 0.0647 0.0588
0.0869 0.0937 0.0567 0.0814 0.0730
0.0215 0.0622 0.0721 0.0620 0.0557
0.0669 0.1034 0.0915 0.0951 0.0839
0.1588 0.1809 0.1154 0.1616 0.1480
0.1784 0.2036 0.1306 0.1839 0.1701
0.1141 0.1726 0.1482 0.1601 0.1701
0.0785 0.1467 0.1437 0.1400 0.1257
0.1080 0.1699 0.1497 0.1589 0.1430
b=-0.4361 -0.3437 -0.2657 -0.0392 0.0193 0.0747 0.0935 0.1079 0.1930 0.2058 0.2606
0.3142 0.3529 0.3615 0.3647
x=0.0000 0.0000 2.4393 0.0000 0.0000
w=-0.0055 -0.0035 0.0000 -0.0024 -0.0023

```

See Also

llsqcovsolve(), **llsqsolve()**.

References

Lawson, C. L. and R. J. Hanson, *Solving Least Squares Problems*, Prentice-Hall, 1974.

llsqsolve

Synopsis

```
#include <numeric.h>
```

```
int llsqsolve(array double complex x[&], array double complex a[&][&],
              array double complex b[&]);
```

Purpose

Linear least-squares solution of a linear system of equations.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x Output array which contains the solution of a linear system of equations.

a Input matrix for a linear system of equations.

b Input array for constants of a linear system of equations.

Description

This function solves the linear system of equations by the method called *linear least-squares* method. The input matrix *x* can be of different number of rows and columns. But, the number of elements for *x* shall be the same as the number of columns in matrix *a*. The number of elements for *b* shall be the same as the number of rows in matrix *a*. The function can handle the equation set with complex numbers. The function returns 0 if there is a solution for the system of linear equations, otherwise it returns -1.

Algorithm

The linear system of equations

$$a * X = b$$

is solved by minimizing the squared error in $(a * X - b)$.

Example1

Solutions of linear system of equations with $a[3][3]$ and $a[3][4]$.

```
#include <numeric.h>
#define N1 3
#define N2 4
int main() {
    array double a[N1][3] = {3, 0, 6,
                             0, 2, 1,
                             1, 0, 1};
    array double a2[N2][3] = {3, 0, 6,
                              0, 2, 1,
                              1, 0, 1,
                              4, 5, 2};
    array double b[N1] = {2,
                          13,
                          25};
    array double b2[N2] = {2,
```

```

                13,
                25,
                1});
array double x[3], x2[3];
int status;

llsqsolve(x, a, b);
printf("llsqsolve(x,a,b) = \n%f\n", x);
printf("a*x = \n%f\n", a*x);
llsqsolve(x2, a2, b2);
printf("llsqsolve(x2, a2,b2) = \n%f\n", x2);
printf("a2*x2 = \n%f\n", a2*x2);

status = llsqsolve(x, a, b);
if(status == 0)
    printf("llsqsolve(x, a,b) =\n%f\n", x);
else
    printf("error: numerical error in llsqsolve()\n");
}

```

Output1

```

llsqsolve(x,a,b) =
49.333333 18.666667 -24.333333

a*x =
2.000000 13.000000 25.000000

llsqsolve(x2, a2,b2) =
-2.256881 1.715596 2.198777

a2*x2 =
6.422018 5.629969 -0.058104 3.948012

llsqsolve(x, a,b) =
49.333333 18.666667 -24.333333

```

Example2

Solution of linear system of equations with complex numbers.

```

#include <numeric.h>
int main() {
    array double complex a[3][3] = {complex(3,2), 0, 6,
                                    0, 2, 1,
                                    1, 0, 1};
    array double complex a2[3][4] = {complex(3,2), 0, 6, 0,
                                    2, 1, 1, 0,
                                    1, 4, 5, 2};

    array double complex b[3] = {2,
                                  13,
                                  25};
    array double complex b2[3] = {2,
                                  13,
                                  25};

    array double complex x[3], x2[4];
    int status;

    llsqsolve(x, a, b);
    printf("llsqsolve(x, a,b) = \n%f\n", x);
}

```



```

llsqsolve(x2, a2, b2);
printf("a*x = \n%f\n", a*x);
printf("llsqsolve(x2, a2,b2) = \n%f\n", x2);
printf("a2*x2 = \n%f\n", a2*x2);

status = llsgsolve(x, a, b);
if(status == 0)
    printf("llsgsolve(x,a,b) is ok\n");
else
    printf("error: numerical error in llsgsolve()\n");
}

```

Output2

```

llsgsolve(x, a,b) =
complex(34.153846,22.769231) complex(11.076923,11.384615) complex(-9.153846,-22.769231)

```

```

a*x =
complex(2.000000,0.000000) complex(13.000000,0.000000) complex(25.000000,0.000000)

```

```

llsgsolve(x2, a2,b2) =
complex(3.953216,0.102924) complex(6.702534,1.163353) complex(-1.608967,-1.369201)
complex(1.140741,1.044834)

```

```

a2*x2 =
complex(2.000000,0.000000) complex(13.000000,-0.000000) complex(25.000000,0.000000)

```

```

llsgsolve(x,a,b) is ok

```

See Also**linsolve()**.**References**

E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

logm

Synopsis

```
#include <numeric.h>
```

```
int logm(array double complex y[&][&], array double complex x[&][&]);
```

Syntax

```
logm(y, x)
```

Purpose

Computes the matrix natural logarithm.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x Input square matrix. It contains data to be calculated.

y Output square matrix which contains data of the result of *x* natural logarithm..

Description

This function computes the matrix natural logarithm of the input matrix *x*. It is the inverse function of **expm()**. The complex results of *y* will be produced if *x* has negative eigenvalues.

Example

This example shows the usage of **expm()** and **logm()**. A real or complex matrix will change back to itself when a matrix is calculated by **expm()** and then followed by **logm()**.

```
#include <numeric.h>
int main() {
    array double x[3][3]={1,2,3,
                          3,4,5,
                          6,7,8};
    array double complex zx[3][3]={complex(1,1),complex(2,2),0,
                                   3,complex(4,1),complex(2,5),
                                   0,0,0};
    array double complex zy[3][3];
    array double complex y[3][3];

    expm(y, x);
    printf("x = \n%f", x);
    printf("y = \n%5.3f", y);
    logm(x, y);
    printf("x = \n%f", x);
    printf("\n");

    expm(zx, zy);
    printf("zx = \n%5.3f", zx);
    printf("zy = \n%5.3f", zy);
    logm(zx, zy);
    printf("zx = \n%5.3f", zx);
}
```

Output

```

x =
1.000000 2.000000 3.000000
3.000000 4.000000 5.000000
6.000000 7.000000 8.000000
y =
complex(157372.953,0.000) complex(200034.605,0.000) complex(242697.257,0.000)
complex(290995.910,0.000) complex(369883.552,0.000) complex(448769.194,0.000)
complex(491431.846,0.000) complex(624654.473,0.000) complex(757878.099,0.000)

x =
1.000000 2.000000 3.000000
3.000000 4.000000 5.000000
6.000000 7.000000 8.000000

zx =
complex(1.000,1.000) complex(2.000,2.000) complex(0.000,0.000)
complex(3.000,0.000) complex(4.000,1.000) complex(2.000,5.000)
complex(0.000,0.000) complex(0.000,0.000) complex(0.000,0.000)
zy =
complex(-44.901,56.755) complex(-87.478,70.853) complex(-101.511,-18.910)
complex(-12.469,118.748) complex(-57.370,175.502) complex(-155.470,67.839)

complex(0.000,0.000) complex(0.000,0.000) complex(1.000,0.000)
zx =
complex(1.000,1.000) complex(2.000,2.000) complex(0.000,0.000)
complex(3.000,0.000) complex(4.000,1.000) complex(2.000,5.000)
complex(0.000,0.000) complex(0.000,0.000) complex(0.000,0.000)

```

See Also

expm(), **funm()**, **cfunm()**, **sqrtm()**.

References

G. H. Golub, C. F. Van Loan, *Matrix Computations* Third edition, The Johns Hopkins University Press, 1996

logdata

Synopsis

```
#include <numeric.h>
```

```
int logdata(double first, double last, ... /* [array] type a[:]...[:] */);
```

Syntax

```
logdata(first, last, a);
```

Purpose

Get logarithmically spaced data for an array passed from the third argument.

Return Value

This function returns the number of elements in the passed array *a*.

Parameters

first Starting point of the logarithmically spaced array.

last Ending point of the logarithmically spaced array.

a An output passed from the third argument with logarithmically spaced data.

Description

This function obtains a logarithmically spaced values starting with 10^{first} and ending with 10^{last} for the array *a* passed from the third argument. The number of points is taken internally from the passed array *a*.

Example

```
#include <numeric.h>

int main () {
    double a[6];
    array float b[6];
    array int c[6];
    array double aa[6], d[2][3];

    logdata(0, 2, a);
    printf("a[0] = %f\n", a[0]);
    aa = a;
    printf("aa = \n%f\n", aa);
    logdata(0, 2, b);
    printf("b = \n%f\n", b);
    logdata(0, 2, c);
    printf("c = \n%d\n", c);
    logdata(0, 2, d);
    printf("d = \n%f\n", d);
}
```

Output

See Also

lindata().

logrange

Synopsis

```
#include <numeric.h>
```

```
array double logrange(double first, double last, int n[:]);
```

Purpose

Generate a logarithmically spaced array of the column vector.

Return Value

This function returns an array a logarithmically spaced column vector with *n* elements.

Parameters

first Starting point of the logarithmically spaced array.

last Ending point of the logarithmically spaced array.

n The number of elements of the array.

Description

This function generates a logarithmically spaced array of *n* elements with the first element of 10^{first} and the last element of 10^{last} .

Example

```
#include <numeric.h>
int main() {
    array double x[10];
    int n = 10;

    x = logrange(0, 2, 10);
    printf("x = %lf\n", x);
}
```

Output

```
x = 1.000000 1.668101 2.782559 4.641589 7.742637 12.915497 21.544347 35.938137
59.948425 100.000000
```

See Also

logdata(), **logspace()**, **linrange()**.

References

logspace

Synopsis

```
#include <numeric.h>
```

```
int logspace(array double &x, double first, double last);
```

Purpose

Get logarithmically spaced values for an array.

Return Value

This function returns the number of elements in array *x*.

Parameters

x An output array with logarithmically spaced values.

first Starting point of the logarithmically spaced array.

last Ending point of the logarithmically spaced array.

Description

This function obtains a logarithmically spaced values starting with 10^{first} and ending with 10^{last} for array *x*. The number of points is taken internally from the array *x*.

Note

`logspace()` is obsolete. Use `logdata()`.

Example

```
#include <numeric.h>
int main() {
    array double x[11], x2[2][3];
    array float y[11];
    array complex z[11];
    int n;

    logspace(x, 0, 2);
    printf("x = %f\n", x);
    logspace(x2, 0, 2);
    printf("x2 = %f\n", x2);
    n = logspace(x, 0, 2);
    printf("n = %d\n", n);
    n = logspace(y, 0, 2);
    printf("y = %f\n", y);
    logspace(z, 0, 2);
    printf("z = %f\n", z);
}
```

Output

```
x = 1.000000 1.584893 2.511886 3.981072 6.309573 10.000000 15.848932 25.118864
39.810717 63.095734 100.000000
```

```
x2 = 1.000000 2.511886 6.309573
```

```
15.848932 39.810717 100.000000
```

```
n = 11
```

```
y = 1.000000 1.584893 2.511886 3.981072 6.309574 10.000000 15.848932 25.118864  
39.810719 63.095734 100.000000
```

```
z = complex(1.000000,0.000000) complex(1.584893,0.000000) complex(2.511886,0.000000)  
complex(3.981072,0.000000) complex(6.309574,0.000000) complex(10.000000,0.000000)  
complex(15.848932,0.000000) complex(25.118864,0.000000) complex(39.810719,0.000000)  
complex(63.095734,0.000000) complex(100.000000,0.000000)
```

See Also

linspace().

References

ludcomp

Synopsis

```
#include <numeric.h>
```

```
int ludcomp(array double complex a[&][&], array double complex l[&][&],
            array double complex u[&][&],... /*array int p[[:]][:] */);
```

Syntax

```
ludcomp(a, l, u)
```

```
ludcomp(a, l, u, p)
```

Purpose

LU decomposition of a general m-by-n matrix.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

a Input matrix.

l Output L matrix.

u Output U matrix.

p Output P matrix.

Description

An LU decomposition of a square matrix *a* is calculated using partial pivoting with row interchanges. The factorization has the form $a = P * L * U$, where P is a permutation matrix, L a lower triangular matrix with unit diagonal elements and U an upper triangular matrix.

Example1

LU decomposition of a real 3×3 matrix.

```
#include <numeric.h>
int main() {
    int n = 3;
    array double a[3][3] = {2,1,-2,
                           4,-1,2,
                           2,-1,1} ; /* n-by-n matrix */
    array double a2[3][3] = {-1,5,6,
                             3,-6,1,
                             6,8, 9} ; /* n-by-n matrix */
    array double l[n][n], u[n][n];
    array int p[n][n];
    int status;

    status = ludcomp(a, l, u);
    printf("l =\n%f\n", l);
    printf("u =\n%f\n", u);
    printf("l*u =\n%f\n", l*u);
}
```



```

/* pass the address of p[0][0]
   or use ludcomp(a, l, u, &p);
   or use ludcomp(a, l, u, &p[0][0]); */
status = ludcomp(a, l, u, p);
printf("l =\n%f\n", l);
printf("u =\n%f\n", u);
printf("p =\n%d\n", p);
printf("p*l*u =\n%f\n", p*l*u);

status = ludcomp(a2, l, u);
if(status == 0) {
    printf("l =\n%f\n", l);
    printf("u =\n%f\n", u);
    printf("l*u =\n%f\n", l*u);
}
else
    printf("error: numerical error in ludcomp()\n");

status = ludcomp(a2, l, u, p);
printf("l =\n%f\n", l);
printf("u =\n%f\n", u);
printf("p =\n%d\n", p);
printf("p*l*u =\n%f\n", p*l*u);
}

```

Output1

```

l =
0.500000 1.000000 0.000000
1.000000 0.000000 0.000000
0.500000 -0.333333 1.000000

u =
4.000000 -1.000000 2.000000
0.000000 1.500000 -3.000000
0.000000 0.000000 -1.000000

l*u =
2.000000 1.000000 -2.000000
4.000000 -1.000000 2.000000
2.000000 -1.000000 1.000000

l =
1.000000 0.000000 0.000000
0.500000 1.000000 0.000000
0.500000 -0.333333 1.000000

u =
4.000000 -1.000000 2.000000
0.000000 1.500000 -3.000000
0.000000 0.000000 -1.000000

p =
0 1 0
1 0 0
0 0 1

p*l*u =

```

```

2.000000  1.000000 -2.000000
4.000000 -1.000000  2.000000
2.000000 -1.000000  1.000000

l =
-0.166667 -0.633333  1.000000
0.500000  1.000000  0.000000
1.000000  0.000000  0.000000

u =
6.000000  8.000000  9.000000
0.000000 -10.000000 -3.500000
0.000000  0.000000  5.283333

l*u =
-1.000000  5.000000  6.000000
3.000000 -6.000000  1.000000
6.000000  8.000000  9.000000

l =
1.000000  0.000000  0.000000
0.500000  1.000000  0.000000
-0.166667 -0.633333  1.000000

u =
6.000000  8.000000  9.000000
0.000000 -10.000000 -3.500000
0.000000  0.000000  5.283333

p =
0 0 1
0 1 0
1 0 0

p*l*u =
-1.000000  5.000000  6.000000
3.000000 -6.000000  1.000000
6.000000  8.000000  9.000000

```

Example2

LU decomposition of 3×3 matrix with complex number.

```

#include <numeric.h>
int main() {
    int n = 3;
    array double complex a[3][3] = {complex(-1,1),5,6,
                                    3,-6,1,
                                    6,8, 9} ; /* n-by-n matrix */
    array double complex l[n][n], u[n][n];

    ludcomp(a, l, u);
    printf("l =\n%f\n", l);
    printf("u =\n%f\n", u);
    printf("l*u =\n%f\n", l*u);
}

```

Output2

```
l =
complex(-0.166667,0.166667) complex(-0.633333,0.133333) complex(1.000000,0.000000)
complex(0.500000,0.000000) complex(1.000000,0.000000) complex(0.000000,0.000000)
complex(1.000000,0.000000) complex(0.000000,0.000000) complex(0.000000,0.000000)

u =
complex(6.000000,0.000000) complex(8.000000,0.000000) complex(9.000000,0.000000)
complex(0.000000,0.000000) complex(-10.000000,0.000000) complex(-3.500000,0.000000)
complex(0.000000,0.000000) complex(0.000000,0.000000) complex(5.283333,-1.033333)

l*u =
complex(-1.000000,1.000000) complex(5.000000,0.000000) complex(6.000000,0.000000)
complex(3.000000,0.000000) complex(-6.000000,0.000000) complex(1.000000,0.000000)
complex(6.000000,0.000000) complex(8.000000,0.000000) complex(9.000000,0.000000)
```

See Also**linsolve, inverse().****References**

E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

maxloc

Synopsis

```
#include <numeric.h>
```

```
int maxloc(array double &a);
```

Purpose

Find the index of element with the maximum value in an array.

Return Value

This function returns the index for the elements with the maximum value in an array.

Parameters

a The input array in which the maximum value shall be found.

Description

The function finds the maximum element in the array *a*.

Example

```
#include <numeric.h>
int main() {
    double a[3] = {1,2,3};
    double b[2][3] = {1,2,3,10,5,6};
    float  b1[2][3] = {1,2,3,10,5,6};
    int    b2[2][3] = {1,2,3,10,5,6};
    double c[2][3][5];
    c[0][0][3] = 10;
    c[1][0][3] = -10;
    int loc;

    loc = maxloc(a);
    printf("maxloc(a) = %d\n", loc);
    loc = maxloc(b);
    printf("maxloc(b) = %d\n", loc);
    loc = maxloc(b1);
    printf("maxloc(b1) = %d\n", loc);
    loc = maxloc(b2);
    printf("maxloc(b2) = %d\n", loc);
    loc = maxloc(c);
    printf("maxloc(c) = %d\n", loc);
}
```

Output

```
maxloc(a) = 2
maxloc(b) = 3
maxloc(b1) = 3
maxloc(b2) = 3
maxloc(c) = 3
```

See Also

[minloc\(\)](#), [maxv\(\)](#).

References

maxv**Synopsis****#include** <numeric.h>**array double maxv**(array double *a*[&][&])[:];**Purpose**

Find the maximum values of the elements of each row of a matrix.

Return Value

This function returns an array which contains the maximum values of each row of the matrix.

Parameters*a* Input matrix.**Description**

This function finds the maximum values of each row of a matrix. The matrix can be of any arithmetic data type.

Example

```
#include <numeric.h>
int main() {
    double a[2][3] = {1,2,3,
                     4,5,6};
    array double b[3][4] = {1,2,3,4,
                           5,6,7,8,
                           1,2,3,4};
    array double maxv1[2], maxv2[3];

    maxv1 = maxv(a);
    printf("maxv(a) = %f\n", maxv1);
    maxv2 = maxv(b);
    printf("maxv(b) = %f\n", maxv2);
}
```

Output

maxv(a) = 3.000000 6.000000

maxv(b) = 4.000000 8.000000 4.000000

See Also**maxloc()**, **minloc()**.**References**

mean

Synopsis

```
#include <numeric.h>
```

```
double mean(array double &a, ... /*[array double v[:]] */);
```

Syntax

```
mean(a)
```

```
mean(a, v)
```

Purpose

Calculate the mean value of all elements of an array and mean values of the elements of each row of a two-dimensional array.

Return Value

This function returns the mean value of all the elements.

Parameters

a An input array of any dimension.

v An output array which contains the mean values of each row of a two-dimensional array.

Description

The function calculates the mean value of all elements in an array of any dimension. If the array is a two-dimensional matrix, the function can calculate mean values of each row. The mean values of each row are passed out by argument *v*.

Example1

Calculate the mean values of all elements of arrays with different dimensions.

```
#include <numeric.h>
int main() {
    double a[3] = {1,2,3};
    int b[2][3] = {1,2,3,4,5,6};
    double c[2][3][5];
    c[0][0][0] = 10;
    c[0][0][1] = 20;
    double meanval;

    meanval = mean(a);
    printf("mean(a) = %f\n", meanval);
    meanval = mean(b);
    printf("mean(b) = %f\n", meanval);
    meanval = mean(c);
    printf("mean(c) = %f\n", meanval);
}
```

Output1

```
mean(a) = 2.000000
mean(b) = 3.500000
mean(c) = 1.000000
```

Example2

Calculate the mean values of all elements of arrays with different dimensions. The mean values of each row are passed by argument *v*.

```
#include <numeric.h>
#define N 2
#define M 3
int main() {
    double a[N][M] = {1,2,3,
                      4,5,6};
    array double b[3][4] = {1,2,3,4,
                           5,6,7,8,
                           1,2,3,4};
    array double meana1[N], meanb1[3];
    double meanval;

    /* Note: second argument of mean() must be double data type */

    meanval = mean(a, meana1);
    printf("meanval = mean(a, meana1) = %f\n", meanval);
    printf("mean(a, meana1) = %f\n", meana1);

    meanval = mean(b, meanb1);
    printf("meanval = mean(b, meanb1) = %f\n", meanval);
    printf("mean(b, meanb1) = %f\n", meanb1);
}
```

Output2

```
meanval = mean(a, meana1) = 3.500000
mean(a, meana1) = 2.000000 5.000000

meanval = mean(b, meanb1) = 3.833333
mean(b, meanb1) = 2.500000 6.500000 2.500000
```

Example3

Calculate the mean values of each column of an array. The calculation is carried out by transposing the array.

```
#include <numeric.h>
int main() {
    double a[2][3] = {1,2,3,
                     4,5,6};
    array double b[3][4] = {1,2,3,4,
                           5,6,7,8,
                           1,2,3,4};
    array double meana1[3], meanb1[4];
    double meanval;

    mean(transpose(a), meana1);
    printf("mean(traspose(a), meana1) = %f\n", meana1);

    meanval = mean(transpose(b), meanb1);
    printf("mean(transpose(b), meanb1) = %f\n", meanb1);
}
```

Output3

```
mean(transpose(a), meana1) = 2.500000 3.500000 4.500000
```

```
mean(transpose(b), meanb1) = 2.333333 3.333333 4.333333 5.333333
```

See Also

median(), minloc(), maxloc().

References

median

Synopsis

```
#include <numeric.h>
```

```
double median(array double &a, ... /*[array double v[:]] */);
```

Syntax

```
median(a)
```

```
median(a, v)
```

Purpose

Find the median value of all elements of an array and median values of the elements in each row of a two-dimensional array.

Return Value

This function returns the median value of all the elements.

Parameters

a An input array of any dimension.

v An output array which contains the median values of each row.

Description

The function calculates the median value of all elements in an array of any dimension. If the array is a two-dimensional matrix, the function can calculate the median values of each row. The median values of each row are passed out by argument *v*.

Example1

Find the median value of all the elements of arrays.

```
#include <numeric.h>
int main() {
    double a[3] = {1,2,3};
    double b[2][3] = {1,2,3,4,5,6};
    double c[2][3][5];
    c[0][0][0] = 10;
    c[0][0][1] = 20;
    double medianval;

    medianval = median(a);
    printf("median(a) = %f\n", medianval);
    medianval = median(b);
    printf("median(b) = %f\n", medianval);
    medianval = median(c);
    printf("median(c) = %f\n", medianval);
}
```

Output

```
median(a) = 2.000000
median(b) = 3.500000
median(c) = 0.000000
```

Example2

Find the median values of each row of arrays.

```
#include <numeric.h>
int main() {
    double a[2][3] = {1,2,3,
                     4,5,6};
    array double b[3][4] = {1,2,3,4,
                           5,6,7,8,
                           1,2,3,4};
    array double medianv1[2], medianv2[3];

    median(a, medianv1);
    printf("median(a, medianv1) = %f\n", medianv1);
    median(b, medianv2);
    printf("median(b, medianv2) = %f\n", medianv2);
}
```

Output

```
median(a, medianv1) = 2.000000 5.000000
```

```
median(b, medianv2) = 2.500000 6.500000 2.500000
```

See Also

mean(), minv().

References

minloc

Synopsis

```
#include <numeric.h>
```

```
int minloc(array double &a);
```

Purpose

Find the index of the element with the minimum value in an array.

Return Value

This function returns the index for the element with the minimum value in an array.

Parameters

a The input array in which the index of the element with the minimum value shall be found.

Description

The function finds the index of the element with the minimum value in the array *a*. The array can be of any dimension.

Example

```
#include <numeric.h>
int main() {
    double a[3] = {1,2,3};
    double b[2][3] = {1,2,3,-10,5,6};
    double c[2][3][5];
    c[0][0][3] = 10;
    c[1][0][3] = -10;
    int loc;

    loc = minloc(a);
    printf("minloc(a) = %d\n", loc);
    loc = minloc(b);
    printf("minloc(b) = %d\n", loc);
    loc = minloc(c);
    printf("minloc(c) = %d\n", loc);
}
```

Output

```
minloc(a) = 0
minloc(b) = 3
minloc(c) = 18
```

See Also

maxloc().

References

minv

Synopsis

```
#include <numeric.h>
```

```
array double minv(array double a[&][&][:];
```

Purpose

Find the minimum values of each row in a two-dimensional array.

Return Value

This function returns the array of minimum values.

Parameters

a Input two-dimensional array in which the minimum values of each row will be found.

Description

The function finds the minimum values of each row of a two-dimensional array. The input array can be of any arithmetic data type.

Example

```
#include <numeric.h>
int main() {
    double a[2][3] = {1,2,3,
                     4,5,6};
    array double b[3][4] = {1,2,3,4,
                           5,6,7,8,
                           1,2,3,4};
    array double minv1[2], minv2[3];

    minv1 = minv(a);
    printf("minv(a) = %f\n", minv1);
    minv2 = minv(b);
    printf("minv(b) = %f\n", minv2);
}
```

Output

```
minv(a) = 1.000000 4.000000
```

```
minv(b) = 1.000000 5.000000 1.000000
```

See Also

maxloc(), **minloc()**, **median()**.

References

mode

Synopsis

```
#include <numeric.h>
```

```
int mode(array int a[:], array int b[:], array int freq[:], array double modeval[:]);
```

Purpose

Calculate the mode for a set of data.

Return Value

This function returns 0 or the number of mode(s).

Parameters

a An array which contains variable values of the data set.

b An array which contains the sorted data in ascending order, similar to the result fo the function **sort()**,

freq The array which contains the frequency for each value in the data set.

modeval The array which contains the mode(s) at the beginning elements. The remaining elements are filled with NaN.

Description

The *mode* is the most frequent number that occurs in a data set. Given a set of data passed in array *a*, this function sorts the array, calculates the frequency for each value in the data set and the mode(s) of the array.

Algorithm

The function will sort the data first, then calculate the frequency for each value in the data set and the mode(s) of the array.

Example

```

/* File: mode.ch
   Find the mode for elements in an array */
#define N 8
array double a[N] = {13, 18, 20, 15, 20, 13, 20, 12};
array double b[N];
array int frequency[N];
array double modeval[N];

mode(a, b, frequency, modeval);

printf("Original Data:%.2lf", a);
printf("Sorted Data:  %.2lf", b);
printf("Frequency:    %5d", frequency);
printf("Mode:        %5.2lf", modeval);

```

Output

```

Original Data:13.00 18.00 20.00 15.00 20.00 13.00 20.00 12.00
Sorted Data: 12.00 13.00 13.00 15.00 18.00 20.00 20.00 20.00
Frequency:   1    2    2    1    1    3    3    3
Mode:       20.00  NaN  NaN  NaN  NaN  NaN  NaN  NaN

```

See Also

max(), mean(), min(), sort(), quartiles()

norm**Synopsis**

```
#include <numeric.h>
```

```
double norm(array double complex &a, char * mode);
```

Purpose

Calculate norm of a vector or matrix.

Return Value

This function returns a norm.

Parameters

a Input, one or two-dimensional array for which a norm is calculated.

mode A character array indicating the type of norm to be calculated.

Description

The norm of a vector or matrix is a scalar that gives some measure of the magnitude of the elements of the vector or matrix. The **norm** function calculates norms of different types for a vector or matrix according to the argument *mode*.

Algorithm

The mode of various different norms for both vectors and matrices are defined below.

Mode	Norm Type	Algorithm
Norm for Vectors		
"1"	1-norm	$\ a\ _1 = a_1 + a_2 + \dots + a_n $
"2"	2-norm	$\ a\ _2 = (a_1 ^2 + a_2 ^2 + \dots + a_n ^2)^{1/2}$
"p"	p-norm	$\ a\ _p = (a_1 ^p + a_2 ^p + \dots + a_n ^p)^{1/p}$ "p" is a floating-point number.
"i"	infinity norm	$\ a\ _\infty = \max_i a_i $
"-i"	negative infinity norm	$\ a\ _{-\infty} = \min_i a_i $
Norm for m-by-n Matrices		
"1"	1-norm	$\ a\ _1 = \max_j \sum_{i=1}^m a_{ij} $
"2"	2-norm	$\ a\ _2 = \text{maximum singular value of } a$
"i"	infinity norm	$\ a\ _\infty = \max_i \sum_{j=1}^n a_{ij} $
"f"	Frobenius norm	$\ a\ _F^2 = \sum_{i=1}^m \sum_{j=1}^n a_{ij} ^2$
"m"	norm	$\ a\ = \max(\text{abs}(A[i][j]))$

Example 1

Calculate the norms of real vectors and matrices.

```
#include <numeric.h>
int main() {
    array double a[2][2] = {2, -4,
                           3, -7};
    array double b[3][2] = {2, -4,
                           3, -7,
                           1, 5};
}
```

```

    array double c[2][3] = {2, -4, 3,
                          -7, 1, 5};
    array double complex d[3] = {2, -4, 3};
    double anorm;

/* a is nxn */
    anorm = norm(a, "1");
    printf("1-norm of a = %f\n", anorm);
    anorm = norm(a, "2");
    printf("2-norm of a = %f\n", anorm);
    anorm = norm(a, "i");
    printf("infinity-norm of a = %f\n", anorm);
    anorm = norm(a, "f");
    printf("Frobenius-norm of a = %f\n", anorm);
    anorm = norm(a, "m");
    printf("max(abs(a)) = %f\n\n", anorm);

/* a is nxm */
    anorm = norm(b, "1");
    printf("1-norm of b = %f\n", anorm);
    anorm = norm(b, "2");
    printf("2-norm of b = %f\n", anorm);
    anorm = norm(b, "i");
    printf("infinity-norm of b = %f\n", anorm);
    anorm = norm(b, "f");
    printf("Frobenius-norm of b = %f\n", anorm);
    anorm = norm(b, "m");
    printf("max(abs(a)) = %f\n\n", anorm);

/* a is mxn */
    anorm = norm(c, "1");
    printf("1-norm of c = %f\n", anorm);
    anorm = norm(c, "2");
    printf("2-norm of c = %f\n", anorm);
    anorm = norm(c, "i");
    printf("infinity-norm of c = %f\n", anorm);
    anorm = norm(c, "f");
    printf("Frobenius-norm of c = %f\n", anorm);
    anorm = norm(c, "m");
    printf("max(abs(a)) = %f\n\n", anorm);

/* a is 1-dim */
    anorm = norm(d, "1");
    printf("1-norm of d = %f\n", anorm);
    anorm = norm(d, "2");
    printf("2-norm of d = %f\n", anorm);
    anorm = norm(d, "2.1");
    printf("2.1-norm of d = %f\n", anorm);
    anorm = norm(d, "i");
    printf("infinity-norm of d = %f\n", anorm);
    anorm = norm(d, "-i");
    printf("negative infinity-norm of d = %f\n", anorm);
}

```

Output1

```

1-norm of a = 11.000000
2-norm of a = 8.828855
infinity-norm of a = 10.000000

```



```
Frobenius-norm of a = 8.831761
max(abs(a)) = 7.000000
```

```
1-norm of b = 16.000000
2-norm of b = 8.671495
infinity-norm of b = 10.000000
Frobenius-norm of b = 10.198039
max(abs(a)) = 7.000000
```

```
1-norm of c = 9.000000
2-norm of c = 9.846035
infinity-norm of c = 13.000000
Frobenius-norm of c = 10.198039
max(abs(a)) = 7.000000
```

```
1-norm of d = 9.000000
2-norm of d = 5.385165
2.1-norm of d = 5.263628
infinity-norm of d = 4.000000
negative infinity-norm of d = 2.000000
```

Example 2

Calculate the norms of complex vectors and matrices.

```
#include <numeric.h>
int main() {
    array double complex z[2][2] = {2, -4,
                                    3, -7};
    array complex z1[2][2]         = {2, -4,
                                    3, -7};
    array double complex z2[2][2] = {2, -4,
                                    3, complex(1,-7)};
    array double complex z3[4] = {2, complex(-4,1),3};
    double anorm;

    anorm = norm(z, "1");
    printf("1-norm of a = %f\n", anorm);
    anorm = norm(z, "i");
    printf("infinity-norm of a = %f\n", anorm);
    anorm = norm(z, "f");
    printf("Frobenius-norm of a = %f\n", anorm);
    anorm = norm(z, "m");
    printf("max(abs(a)) = %f\n\n", anorm);

    anorm = norm(z1, "1");
    printf("1-norm of a = %f\n", anorm);
    anorm = norm(z1, "i");
    printf("infinity-norm of a = %f\n", anorm);
    anorm = norm(z1, "f");
    printf("Frobenius-norm of a = %f\n", anorm);
    anorm = norm(z1, "m");
    printf("max(abs(a)) = %f\n\n", anorm);

    anorm = norm(z2, "1");
    printf("1-norm of z2 = %f\n", anorm);
    anorm = norm(z2, "i");
    printf("infinity-norm of z2 = %f\n", anorm);
    anorm = norm(z2, "f");
    printf("Frobenius-norm of z2 = %f\n", anorm);
```

```

    anorm = norm(z2, "m");
    printf("max(abs(z2)) = %f\n\n", anorm);

    anorm = norm(z3, "1");
    printf("1-norm of d = %f\n", anorm);
    anorm = norm(z3, "2");
    printf("2-norm of d = %f\n", anorm);
    anorm = norm(z3, "2.1");
    printf("2.1-norm of d = %f\n", anorm);
    anorm = norm(z3, "i");
    printf("infinity-norm of d = %f\n", anorm);
    anorm = norm(z3, "-i");
    printf("negative infinity-norm of d = %f\n", anorm);
}

```

Output2

```

1-norm of a = 11.000000
infinity-norm of a = 10.000000
Frobenius-norm of a = 8.831761
max(abs(a)) = 7.000000

```

```

1-norm of a = 11.000000
infinity-norm of a = 10.000000
Frobenius-norm of a = 8.831761
max(abs(a)) = 7.000000

```

```

1-norm of z2 = 11.071068
infinity-norm of z2 = 10.071068
Frobenius-norm of z2 = 8.888194
max(abs(z2)) = 7.071068

```

```

1-norm of d = 9.123106
2-norm of d = 5.477226
2.1-norm of d = 5.355310
infinity-norm of d = 4.123106
negative infinity-norm of d = 0.000000

```

See Also

condnum(), **rcondnum()**, **svd()**, **maxloc()**, **maxv()**, **minloc()**, **minv()**.

References

E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

nullspace

Synopsis

```
#include <numeric.h>
```

```
int nullspace(array double complex null[&][&], array double complex a[&][&]);
```

Purpose

Calculate null space of matrix a .

Return Value

This function returns 0 on success and -1 on failure.

Parameters

a Input matrix.

$null$ Output array which contains the null space of matrix a .

Description

This function calculates an orthonormal bases for null space of matrix a . The null space s meets the following condition

$$s^t * s = I$$

and

$$a * s = 0$$

Algorithm

The computation of null space of matrix a is based on the singular value decomposition of matrix a . The SVD is formulated as

$$A = USV^T$$

Where S is an m -by- n matrix which is zero except for its $\min(m,n)$ diagonal elements, U is an m -by- m orthogonal matrix, and V is an n -by- n orthogonal matrix. The null space can be obtained by

$$orth(i, j) = V^T(i, j),$$

with $i=0, 1, 2, \dots, n-1$, and $j=r, r+1, \dots, n-1$, where r is the number of non-zero singular values with tolerance $tol = \max(m, n) \times \max(S) \times DBL_EPSILON$.

Example

Calculate null space of a singular matrix with $\text{rank}(a) == 2$.

```
#include <numeric.h>
#define M 3
#define N 3
int main() {
    /* singular matrix rank(a) == 2 */
    array double a[M][N] = {1, 2, 3,
                           4, 5, 6,
                           7, 8, 9};
```

```

int r = rank(a);
array double null[M][N-r];

nullspace(null, a);
printf("rank(a) = %d\n", r);
printf("null from nullspace(null, a) = \n%f\n", null);
printf("transpose(null)*null = \n%f\n", transpose(null)*null);
}

```

Output

```

rank(a) = 2
null from nullspace(null, a) =
-0.408248
0.816497
-0.408248

transpose(null)*null =
1.000000

```

Example2

Calculate null space of a singular matrix with $\text{rank}(a) == 1$.

```

#include <numeric.h>
#define M 3
#define N 3
int main() {
    /* singular matrix rank(a) == 1 */
    array double a[M][N] = {1, 2, 3,
                             1, 2, 3,
                             2, 4, 6};

    int r = rank(a);
    array double null[M][N-r];

    nullspace(null, a);
    printf("rank(a) = %d\n", r);
    printf("null from nullspace(null, a) = \n%f\n", null);
    printf("transpose(null)*null = \n%f\n", transpose(null)*null);
}

```

Output2

```

rank(a) = 1
null from nullspace(null, a) =
-0.872872 0.408248
-0.218218 -0.816497
0.436436 0.408248

transpose(null)*null =
1.000000 -0.000000
-0.000000 1.000000

```

Example3

Calculate null space of a non-singular matrix with $\text{rank}(a) == 3$.

```

#include <numeric.h>
#define M 3
#define N 3
int main() {

```

```
/* non-singular matrix rank(a) == 3 */
array double a[M][N] = {1, 2, 3,
                        5, 5, 6,
                        7, 8, 9};

int r = rank(a);
if(r == 3) {
    printf("rank(a) is 3, empty null space\n");
    return 0;
}
else {
    array double null[M][N-r];
    nullspace(null, a);
    printf("rank(a) = %d\n", r);
    printf("null from nullspace(null, a) = \n%f\n", null);
    printf("transpose(null)*null = \n%f\n", transpose(null)*null);
}
}
```

Output3

```
rank(a) is 3, empty null space
```

See Also

svd(), **rank()**.

References

E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

oderk

Synopsis

```
#include <numeric.h>
int oderk(double (*func)(double, double[], double[], void*),
          double t0, double tf, double y0[], void *param,
          double t[], double *y, ... /* [double eps] */);
```

Syntax

```
oderk(func, t0, tf, y0, param, t, y)
oderk(func, t0, tf, y0, param, t, y, eps)
```

Purpose

Numerical solution for ordinary differential equations (ODE) using Runge-Kutta method.

Return Value

If successful, this function returns the number of points calculated in the interval between $t0$ and tf . Otherwise, it returns -1.

Parameters

func A pointer to function, the function shall be first-order differential equation.

t0 A start point of t .

tf A final point of t .

y0 A vector which contains the initial values of ODE functions.

param An option parameter for passing information to ODE functions.

t A vector which contains the t value in the interval

y The address for a one or two-dimensional array which contains the solution of the ODE function. For a two-dimensional array, each row of y corresponds to each derivative function and each column corresponds to the interval value t_i between $t0$ and tf .

eps The user specified tolerance. If the user does not specify this option, the value of 10^{-8} is used by default.

Description

The derivative functions *func* is specified as a pointer to a function which shall be first-order differential equation. The end-points $t0$ and tf and initial value of differential equations $y0$ shall be array of double type. The returned **int** value is the effective number of points calculated in the interval between $t0$ and tf . Vector t contains the values between $t0$ and tf in which the ODE function is solved and the results are stored in a one-dimensional array y for an ordinary differential equation or in a two-dimensional array for a system of ordinary differential equations. The argument `param` is used to pass information from an application program to the ODE functions. If the optional argument *eps* is passed, the algorithm uses the user specified tolerance to decide when to stop the iterations. Otherwise, the value of 10^{-8} is used by default.

Algorithm

Fifth-order Runge-Kutta with adaptive stepsize control.

$$y_{n+1} = y_n + c_1k_1 + c_2k_2 + c_3k_3 + c_4k_4 + c_5k_5 + c_6k_6 + O(h^6)$$

Where

$$\begin{aligned} k_1 &= hf(t_n, y_n) \\ k_2 &= hf(t_n + a_2h, y_n + b_{21}k_1) \\ &\vdots \\ k_6 &= hf(t_n + a_6h, y_n + b_{61}k_1 + \cdots + b_{65}k_5) \end{aligned}$$

The particular values of the various constants are obtained by Cash and Karp, and given in the table below.

Cash-Karp Parameters for Embedded Runge-Kutta Method							
i	a_i	b_i					c_i
1							$\frac{37}{378}$
2	$\frac{1}{3}$	$\frac{1}{3}$					0
3	$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$				$\frac{250}{621}$
4	$\frac{3}{5}$	$\frac{3}{10}$	$-\frac{9}{10}$	$\frac{6}{5}$			$\frac{125}{594}$
5	1	$-\frac{11}{54}$	$\frac{5}{2}$	$-\frac{70}{27}$	$\frac{35}{27}$		0
6	$\frac{7}{8}$	$\frac{1631}{55296}$	$\frac{175}{512}$	$\frac{575}{13824}$	$\frac{44275}{110592}$	$\frac{253}{4096}$	$\frac{512}{1771}$
j	=	1	2	3	4	5	

Example 1

This example shows how to solve the first-order derivative function using function `oderk()`. We want to solve ODE function

$$\frac{dy}{dt} = \cos(t)$$

with $t_0 = 0, t_f = 2\pi, y_0 = 0$ and no more than 50 points.

```
#include <stdio.h>
#include <numeric.h>

#define NVAR 1
#define POINTS 50

int main() {
    void func(double t, double y[], double dydt[], void *param);
    double t0=0, tf=2*M_PI, y0[NVAR]={0};
    double t[POINTS], y[POINTS];
    int i, points;

    /* return -1: divergent
    return >1: ok, num of points
    */
    points = oderk(func, t0, tf, y0, NULL, t, y);
    printf("points = %d\n", points);
}
```

```

if(points > 0) {
    printf("\n%8s %18s %15s\n", "t", "oderk()", "sin(t)");
    for (i=0; i<points; i++)
        printf("%10.4f %14.6f %14.6f\n", t[i], y[i], sin(t[i]));
    plotxy(t, y, "result of oderk", "t", "y");
}
else
    printf("oderk() failed\n");
}

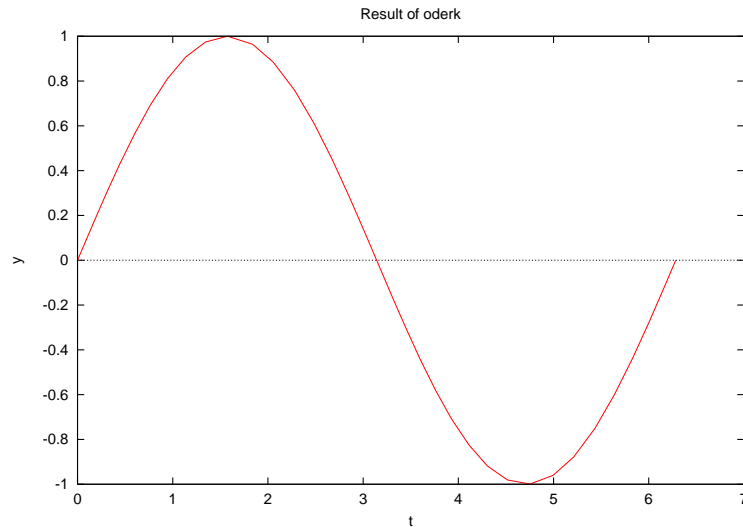
void func(double t, double y[], double dydt[], void *param) {
    /* y' = cos(t) */
    dydt[0] = cos(t);
}

```

Output

points = 35

t	oderk()	sin(t)
0.0000	0.000000	0.000000
0.1647	0.163958	0.163958
0.2930	0.288791	0.288791
0.4373	0.423485	0.423485
0.5945	0.560114	0.560114
0.7633	0.691328	0.691328
0.9435	0.809628	0.809628
1.1361	0.907004	0.907004
1.3441	0.974410	0.974410
1.5755	0.999989	0.999989
1.8381	0.964487	0.964487
2.0552	0.884972	0.884972
2.2800	0.758904	0.758904
2.4850	0.610393	0.610393
2.6762	0.448786	0.448786
2.8559	0.281835	0.281835
3.0246	0.116702	0.116702
3.1814	-0.039813	-0.039813
3.3231	-0.180529	-0.180529
3.4539	-0.307275	-0.307275
3.6002	-0.442691	-0.442691
3.7591	-0.579004	-0.579004
3.9295	-0.708868	-0.708868
4.1113	-0.824733	-0.824733
4.3058	-0.918487	-0.918487
4.5164	-0.980855	-0.980855
4.7526	-0.999192	-0.999192
4.9945	-0.960467	-0.960467
5.2116	-0.877948	-0.877948
5.4347	-0.750252	-0.750252
5.6388	-0.600703	-0.600703
5.8292	-0.438549	-0.438549
6.0082	-0.271508	-0.271508
6.1763	-0.106713	-0.106713
6.2832	0.000000	-0.000000

**Example 2**

This example shows how to specify the tolerance. We want to solve ODE function

$$\frac{dy}{dt} = \cos(t)$$

with $t_0 = 0, t_f = 2\pi, y_0 = 0$ and no more than 50 points. The numerical tolerance is FLT_EPSILON instead of 10^{-8} by default.

```
#include <stdio.h>
#include <float.h>
#include <numeric.h>

#define NVAR 1
#define POINTS 50

int main() {
    void func(double t, double y[], double dydt[], void *param);
    double t0=0, tf=2*M_PI, y0[NVAR]={0};
    double t[POINTS], y[POINTS];
    double tol;
    int i, points;

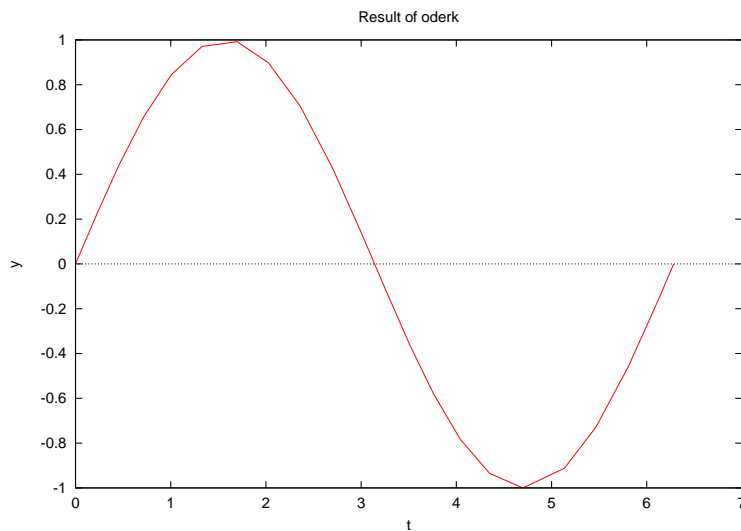
    tol = FLT_EPSILON; /* default tolerance is 1e-6 */
    //points = oderk(t, y, NULL, func, t0, tf, y0, tol);
    points = oderk(func, t0, tf, y0, NULL, t, y);
    printf("points = %d\n", points);
    printf("\n%8s %18s %15s\n", "t", "oderk()", "sin(t)");
    for (i=0; i<points; i++)
        printf("%10.4f %14.6f %14.6f\n", t[i], y[i], sin(t[i]));
    plotxy(t, y, "result of oderk", "t", "y");
}

void func(double t, double y[], double dydt[], void *param) {
    /* y' = cos(t) */
    dydt[0] = cos(t);
}
```

Output

points = 22

t	oderk ()	sin(t)
0.0000	0.000000	0.000000
0.2319	0.229803	0.229803
0.4576	0.441772	0.441772
0.7181	0.657918	0.657918
1.0080	0.845762	0.845762
1.3295	0.971026	0.971026
1.6979	0.991935	0.991935
2.0250	0.898604	0.898604
2.3583	0.705591	0.705591
2.6974	0.429731	0.429731
3.0073	0.133918	0.133918
3.2598	-0.117907	-0.117907
3.5128	-0.362780	-0.362781
3.7617	-0.581135	-0.581135
4.0411	-0.783034	-0.783034
4.3504	-0.935198	-0.935198
4.6982	-0.999900	-0.999900
5.1342	-0.912350	-0.912350
5.4667	-0.728744	-0.728744
5.8095	-0.456146	-0.456146
6.1219	-0.160581	-0.160581
6.2832	-0.000000	-0.000000

**Example 3**

This example shows how to solve the Van der Pol equation

$$\frac{d^2u}{dt^2} - \mu(1 - u^2)\frac{du}{dt} + u = 0$$

where $\mu = 2$, $t_0 = 1$, $t_f = 30$, $u(t_0) = 1$, and $u'(t_0) = 0$;

The Van der Pol equation is reformulated as a set of first-order differential equations first.

Let

$$\begin{aligned} y_0 &= u \\ y_1 &= \frac{du}{dt} \end{aligned}$$

then

$$\begin{aligned} \frac{dy_0}{dt} &= \frac{du}{dt} \\ \frac{dy_1}{dt} &= \frac{d^2u}{dt^2} = \mu(1 - u^2) \frac{du}{dt} + u = \mu(1 - y_0^2)y_1 - y_0 \end{aligned}$$

Use `oderk()` to solve this system of ordinary differential equations. The parameter `param` is used to pass μ from the function `main()` to the ODE function `func()`.

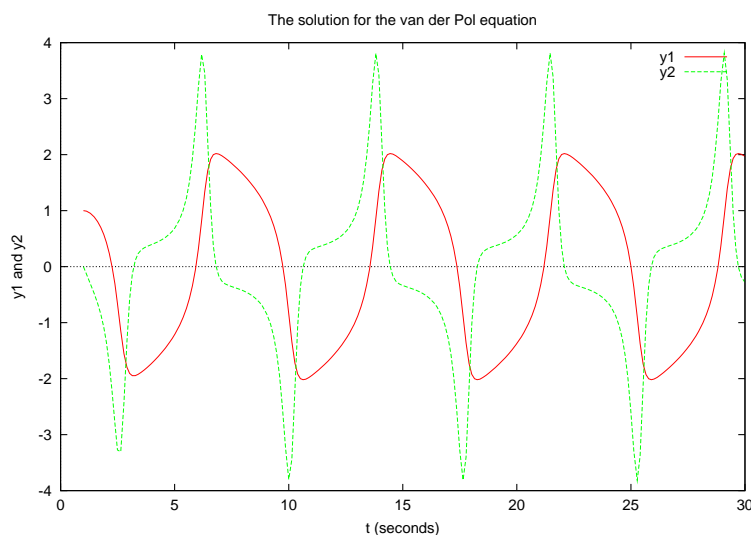
```
#include <chplot.h>
#include <numeric.h>

#define NVAR 2
#define POINTS 256
void func(double t, double y[], double dydt[], void *param) {
    double mu;
    mu = *(double*)param;
    dydt[0] = y[1];
    dydt[1] = mu*(1-y[0]*y[0])*y[1] - y[0];
}

int main() {
    double t0=1, tf=30, y0[NVAR] = {1, 0};
    double t[POINTS], y[NVAR][POINTS];
    double mu = 2;

    oderk(func, t0, tf, y0, &mu, t, y);
    plotxy(t, y, "The solution for the van der Pol equation", "t (seconds)", "y1 and y2");
}
```

Output



Example 4

Function **oderk()** returns the number of points calculated in the interval t_0 and t_f . Generally the user should guess how many points it would produce and define the size of **array** t and y a little bit bigger than the guessed number of points. Function **oderk()** will automatically pad the leftover space in t and y with results at t_f .

```
#include <numeric.h>
#include <stdio.h>
#include <chplot.h>

#define NVAR 4
#define POINTS 50

int main() {
    double t0=1, tf=10, y0[NVAR];
    void func(double t, double y[], double dydt[], void *param);
    double t[POINTS], y[NVAR][POINTS];
    int i, points;
    class CPlot plot;

    y0[0]=j0(t0);
    y0[1]=j1(t0);
    y0[2]=jn(2,t0);
    y0[3]=jn(3,t0);

    points = oderk(func, t0, tf, y0, NULL, t, y);
    printf("points = %d\n", points);
    printf("\n%8s %18s %15s\n", "t", "integral", "bessj(2,t)");
    for (i=0; i<points; i++)
        printf("%10.4f %14.6f %14.6f\n", t[i],y[2][i],jn(2,t[i]));

    printf("\nThe leftover space in t and y are padded with results at tf\n");
    for (i=points; i<POINTS; i++)
        printf("%10.4f %14.6f %14.6f\n", t[i],y[2][i],jn(2,t[i]));

    plotxy(t, y, "solving ode using oderk()", "t", "yi", &plot);
    plot.legend("j0", 0);
    plot.legend("j1", 1);
    plot.legend("j2", 2);
    plot.legend("j3", 3);
    plot.plotting();
}

void func(double t, double y[], double dydt[], void *param) {
    dydt[0] = -y[1];
    dydt[1]=y[0]-(1.0/t)*y[1];
    dydt[2]=y[1]-(2.0/t)*y[2];
    dydt[3]=y[2]-(3.0/t)*y[3];
}
```

Output

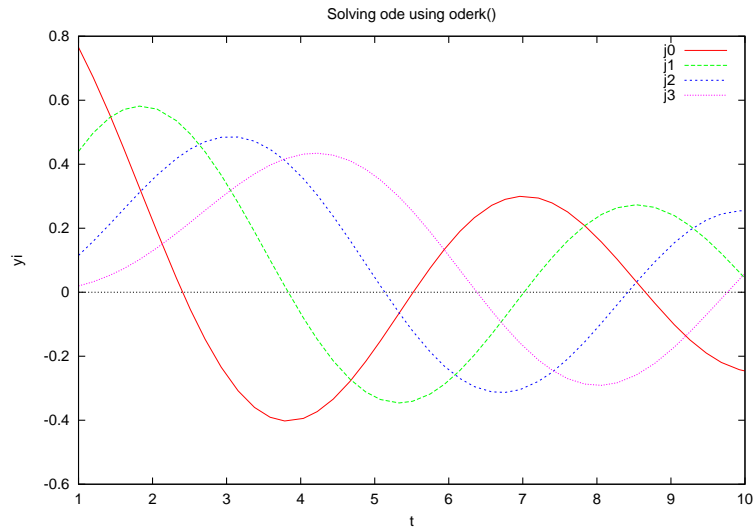
```
points = 43

      t          integral      bessj(2,t)
  1.0000      0.114903      0.114903
  1.1995      0.159225      0.159225
```

1.4208	0.212473	0.212473
1.6107	0.259628	0.259628
1.8230	0.311687	0.311687
2.0595	0.365944	0.365944
2.3217	0.417776	0.417776
2.5117	0.447689	0.447689
2.7100	0.470487	0.470487
2.9266	0.484255	0.484255
3.1546	0.485094	0.485094
3.3763	0.471942	0.471942
3.5876	0.446661	0.446661
3.7842	0.412467	0.412467
4.0366	0.354910	0.354910
4.2242	0.303546	0.303546
4.4402	0.237267	0.237267
4.6658	0.162285	0.162285
4.8797	0.088340	0.088340
5.0780	0.019679	0.019679
5.3301	-0.064486	-0.064486
5.5121	-0.120926	-0.120926
5.7491	-0.186326	-0.186326
5.9414	-0.230973	-0.230973
6.1517	-0.269627	-0.269627
6.3458	-0.294896	-0.294896
6.5806	-0.311270	-0.311270
6.7618	-0.313089	-0.313089
6.9592	-0.304516	-0.304516
7.2117	-0.278417	-0.278417
7.3951	-0.249841	-0.249841
7.6042	-0.208797	-0.208797
7.8362	-0.154879	-0.154879
8.0631	-0.096271	-0.096271
8.2728	-0.039407	-0.039407
8.5323	0.030949	0.030949
8.7792	0.094212	0.094212
9.0451	0.154300	0.154300
9.2558	0.193600	0.193600
9.4769	0.225200	0.225200
9.6799	0.244472	0.244472
9.9231	0.254481	0.254481
10.0000	0.254630	0.254630

The leftover space in t and y are padded with results at tf

10.0000	0.254630	0.254630
10.0000	0.254630	0.254630
10.0000	0.254630	0.254630
10.0000	0.254630	0.254630
10.0000	0.254630	0.254630
10.0000	0.254630	0.254630
10.0000	0.254630	0.254630

**See Also**

derivative(), **integral1()**, **integral2()**, **integral3()**, **integration2()**, **integration3()**.

References

Cash, J. R., and Karp, A. H., *ACM Transactions on Mathematical Software*, 1990. vol. 16, pp. 201-222.

oderungekutta

Synopsis

```
#include <numeric.h>
int oderungekutta(double t[], double &y, void *param,
                 double (*func)(double, double[], double[], void*),
                 double t0, double tf, double y0[&], ... /* [double eps] */);
```

Syntax

```
oderungekutta(t, y, param, func, t0, tf, y0)
oderungekutta(t, y, param, func, t0, tf, y0, eps)
```

Purpose

Numerical solution for ordinary differential equations (ODE) using Runge-Kutta method. This function is obsolete.

Use function

oderk().

Return Value

If successful, this function returns the number of points calculated in the interval between *t0* and *tf*. Otherwise, it returns -1.

Parameters

t A vector which contains the *t* value in the interval between *t0* and *xf*.

y A one or two-dimensional array which contains the solution of the ODE function. For a two-dimensional array, each row of *y* corresponds to each derivative function and each column corresponds to the interval value t_i

param An option parameter for passing information to ODE functions.

func A pointer to function, the function shall be first-order differential equation.

t0 A start point of *t*.

tf A final point of *t*.

y0 A vector which contains the initial values of ODE functions.

eps The user specified tolerance. If the user does not specify this option, the value of 10^{-8} is used by default.

Description

The derivative functions *func* is specified as a pointer to a function which shall be first-order differential equation. The end-points *t0* and *tf* and initial value of differential equations *y0* can be any **real** data type. Conversion of the data to double type is performed internally. The returned **int** value is the effective number of points calculated in the interval between *t0* and *tf*. Vector *t* contains the values between *t0* and *tf* in which the ODE function is solved and the results are stored in a one-dimensional array *y* for an ordinary differential equation or in a two-dimensional array for a system of ordinary differential equations. The argument *param* is used to pass information from an application program to the ODE functions. If the optional argument *eps*

is passed, the algorithm uses the user specified tolerance to decide when to stop the iterations. Otherwise, the value of 10^{-8} is used by default.

Algorithm

Fifth-order Runge-Kutta with adaptive stepsize control.

$$y_{n+1} = y_n + c_1k_1 + c_2k_2 + c_3k_3 + c_4k_4 + c_5k_5 + c_6k_6 + O(h^6)$$

Where

$$\begin{aligned} k_1 &= hf(t_n, y_n) \\ k_2 &= hf(t_n + a_2h, y_n + b_{21}k_1) \\ &\vdots \quad \quad \quad \vdots \\ k_6 &= hf(t_n + a_6h, y_n + b_{61}k_1 + \cdots + b_{65}k_5) \end{aligned}$$

The particular values of the various constants are obtained by Cash and Karp, and given in the table below.

Cash-Karp Parameters for Embedded Runge-Kutta Method							
i	a_i	b_i					c_i
1							$\frac{37}{378}$
2	$\frac{1}{3}$	$\frac{1}{3}$					0
3	$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$				$\frac{250}{621}$
4	$\frac{3}{5}$	$\frac{3}{10}$	$-\frac{9}{10}$	$\frac{6}{5}$			$\frac{125}{594}$
5	1	$-\frac{11}{54}$	$\frac{5}{2}$	$-\frac{70}{27}$	$\frac{35}{27}$		0
6	$\frac{7}{8}$	$\frac{1631}{55296}$	$\frac{175}{512}$	$\frac{575}{13824}$	$\frac{44275}{110592}$	$\frac{253}{4096}$	$\frac{512}{1771}$
j	=	1	2	3	4	5	

See Also

oderk().

References

Cash, J. R., and Karp, A. H., *ACM Transactions on Mathematical Software*, 1990. vol. 16, pp. 201-222.

odesolve

Synopsis

```
#include <numeric.h>
```

```
int odesolve(double t[&], double &y, double (*func)(double, double[], double[]), double t0, double tf,
double y0[&],
... /* [double eps] */);
```

Syntax

```
odesolve(t, y, func, t0, tf, y0)
```

```
odesolve(t, y, func, t0, tf, y0, eps)
```

Purpose

Numerical solution for ordinary differential equations (ODE). This function is obsolete. Use function `oderk()`.

Return Value

If successful, this function returns the number of points calculated in the interval between $t0$ and tf . Otherwise, it returns -1.

Parameters

t A vector which contains the t value in the interval between $t0$ and tf .

y A one or two-dimensional array which contains the solution of the ODE function. For a two-dimensional array, each row of y corresponds to each derivative function and each column corresponds to the interval value t_i

$func$ A pointer to function, the function shall be first-order differential equation.

$t0$ A start point of t .

tf A final point of t .

$y0$ A vector which contains the initial values of ODE functions.

eps The user specified tolerance. If the user does not specify this option, the value of 10^{-8} is used by default.

Description

The derivative functions $func$ is specified as a pointer to a function which shall be first-order differential equation. The end-points $t0$ and tf and initial value of differential equations $y0$ can be any **real** data type. Conversion of the data to double type is performed internally. The returned **int** value is the effective number of points calculated in the interval between $t0$ and tf . Vector t contains the values between $t0$ and tf in which the ODE function is solved and the results are stored in a one-dimensional array y for an ordinary differential equation or in a two-dimensional array for a system of ordinary differential equations. If the optional argument eps is passed, the algorithm uses the user specified tolerance to decide when to stop the iterations. Otherwise, the value of 10^{-8} is used by default.

Algorithm

Fifth-order Runge-Kutta with adaptive stepsize control.

$$y_{n+1} = y_n + c_1k_1 + c_2k_2 + c_3k_3 + c_4k_4 + c_5k_5 + c_6k_6 + O(h^6)$$

Where

$$\begin{aligned} k_1 &= hf(t_n, y_n) \\ k_2 &= hf(t_n + a_2h, y_n + b_{21}k_1) \\ &\vdots \\ k_6 &= hf(t_n + a_6h, y_n + b_{61}k_1 + \dots + b_{65}k_5) \end{aligned}$$

The particular values of the various constants are obtained by Cash and Karp, and given in the table below.

Cash-Karp Parameters for Embedded Runge-Kutta Method							
i	a_i	b_i					c_i
1							$\frac{37}{378}$
2	$\frac{1}{3}$	$\frac{1}{3}$					0
3	$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$				$\frac{250}{621}$
4	$\frac{3}{5}$	$\frac{3}{10}$	$-\frac{9}{10}$	$\frac{6}{5}$			$\frac{125}{594}$
5	1	$-\frac{11}{54}$	$\frac{5}{2}$	$-\frac{70}{27}$	$\frac{35}{27}$		0
6	$\frac{7}{8}$	$\frac{1631}{55296}$	$\frac{175}{512}$	$\frac{575}{13824}$	$\frac{44275}{110592}$	$\frac{253}{4096}$	$\frac{512}{1771}$
j	=	1	2	3	4	5	

see also `oderk()`.

References

Cash, J. R., and Karp, A. H., *ACM Transactions on Mathematical Software*, 1990. vol. 16, pp. 201-222.

orthonormalbase

Synopsis

```
#include <numeric.h>
```

```
int orthonormalbase(array double complex orth[&][&], array double complex a[&][&]);
```

Purpose

Find orthonormal bases of a matrix.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

a The input matrix.

orth An output matrix of orthonormal bases for the matrix *a*.

Description

This function calculates the orthonormal bases for a matrix. The columns of *orth* are orthonormal, and have the same space as columns of *a*. The number of columns of *orth* is the rank of *a*.

Algorithm

The computation of orthonormal bases for matrix *a* is based on the singular value decomposition(SVD) of matrix *a*. The SVD is formulated as

$$A = USV^T$$

Where *S* is an m-by-n matrix which is zero except for its min(m,n) diagonal elements, *U* is an m-by-m orthogonal matrix, and *V* is an n-by-n orthogonal matrix. The orthonormal bases can be obtained by

$$orth(i, j) = U(i, j),$$

with *i*=0, 1, 2, ... n-1, and *j*=0, 1, 2, ... r-1, where *r* is the number of non-zero singular values with tolerance $tol = \max(m, n) \times \max(S) \times DBL_EPSILON$.

Example1

Calculate the orthonormal bases of a singular 3-by-3 matrix with rank(a) == 2.

```
#include <numeric.h>
#define M 3
#define N 3
int main() {
    /* singular matrix rank(a) == 2 */
    array double a[M][N] = {1, 2, 3,
                           4, 5, 6,
                           7, 8, 9};

    int r = rank(a);
    array double orth[M][r];

    orthonormalbase(orth, a);
    printf("rank(a) = %d\n", r);
    printf("orth from orthonormalbase(orth, a) = \n%f\n", orth);
    printf("transpose(orth)*orth = \n%f\n", transpose(orth)*orth);
}
```

Output1

```
rank(a) = 2
orth from orthonormalbase(orth, a) =
-0.214837 0.887231
-0.520587 0.249644
-0.826338 -0.387943
```

```
transpose(orth)*orth =
1.000000 -0.000000
-0.000000 1.000000
```

Example2

Calculate the orthonormal bases of a singular 3-by-3 matrix with $\text{rank}(a) == 1$.

```
#include <numeric.h>
#define M 3
#define N 3
int main() {
    /* singular matrix rank(a) == 1 */
    array double a[M][N] = {1, 2, 3,
                            1, 2, 3,
                            2, 4, 6};

    int r = rank(a);
    array double orth[M][r];

    orthonormalbase(orth, a);
    printf("rank(a) = %d\n", r);
    printf("orth from orthonormalbase(orth, a) = \n%f\n", orth);
    printf("transpose(orth)*orth = \n%f\n", transpose(orth)*orth);
}
```

Output2

```
rank(a) = 1
orth from orthonormalbase(orth, a) =
-0.408248
-0.408248
-0.816497
```

```
transpose(orth)*orth =
1.000000
```

Example3

Calculate the orthonormal bases of a singular 3-by-3 matrix with $\text{rank}(a) == 3$.

```
#include <numeric.h>
#define M 3
#define N 3
int main() {
    /* singular matrix rank(a) == 3 */
    array double a[M][N] = {1, 2, 3,
                            5, 5, 6,
                            7, 8, 9};

    int r = rank(a);
    array double orth[M][r];

    orthonormalbase(orth, a);
```

```
printf("rank(a) = %d\n", r);
printf("orth from orthonormalbase(orth, a) = \n%f\n", orth);
printf("transpose(orth)*orth = \n%f\n", transpose(orth)*orth);
}
```

Output3

```
rank(a) = 3
orth from orthonormalbase(orth, a) =
-0.210138 0.955081 -0.208954
-0.541507 -0.291649 -0.788486
-0.814010 -0.052541 0.578470
```

```
transpose(orth)*orth =
1.000000 0.000000 0.000000
0.000000 1.000000 0.000000
0.000000 0.000000 1.000000
```

See Also

nullspace(), **svd()**, **rank()**.

References

E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

pinverse

Synopsis

```
#include <numeric.h>
```

```
array double pinverse(array double a[:][:][:][:]);
```

Purpose

Calculate MoorePenrose pseudo inverse of a matrix.

Return Value

This function returns the pseudo inverse of a matrix.

Parameters

a An input matrix.

Description

The Moore_Penrose pseudo inverse matrix *b* of matrix *a* has to meet the following four conditions:

$$\begin{aligned}
 a * b * a &= a, \\
 b * a * b &= b, \\
 a * b &\text{ is Hermitian} \\
 b * a &\text{ is Hermitian}
 \end{aligned}$$

Algorithm

The algorithm of this function is based on the function `svd()`. Any singular value less than `DBL_EPSILON` is treated as zero.

Example1

Calculate the Moore-Penrose pseudo inverse of a 3-by-3 singular matrix with `rank(a) == 2`.

```

#include <numeric.h>
#define M 3
#define N 3
/* pseudoinverse */
int main() {
    /* singular matrix rank(a) == 2 */
    array double a[M][N] = {1, 2, 3,
                             4, 5, 6,
                             7, 8, 9};

    int r;
    array double pinv[N][M];

    r = rank(a);
    pinv = pinverse(a);
    printf("rank(a) = %d\n", r);
    printf("pinverse(a) = \n%f\n", pinv);
    printf("a*pinverse(a)*a = \n%f\n", a*pinv*a);
    printf("pinverse(a)*a*pinverse(a) = \n%f\n", pinv*a*pinv);
}

```

Output1

```
rank(a) = 2
pinverse(a) =
-0.638889 -0.166667 0.305556
-0.055556 0.000000 0.055556
0.527778 0.166667 -0.194444

a*pinverse(a)*a =
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000
7.000000 8.000000 9.000000

pinverse(a)*a*pinverse(a) =
-0.638889 -0.166667 0.305556
-0.055556 0.000000 0.055556
0.527778 0.166667 -0.194444
```

Example2

Calculate the Moore-Penrose pseudo inverse of a 3-by-3 singular matrix with $\text{rank}(a) == 1$.

```
#include <numeric.h>
#define M 3
#define N 3
/* pseudoinverse */
int main() {
    /* singular matrix rank(a) == 1 */
    array double a[M][N] = {1, 2, 3,
                             1, 2, 3,
                             2, 4, 6};

    int r;
    array double pinv[N][M];

    r = rank(a);
    pinv = pinverse(a);
    printf("rank(a) = %d\n", r);
    printf("pinverse(a) = \n%f\n", pinv);
    printf("a*pinverse(a)*a = \n%f\n", a*pinv*a);
    printf("pinverse(a)*a*pinverse(a) = \n%f\n", pinv*a*pinv);
}
```

Output2

```
rank(a) = 1
pinverse(a) =
0.011905 0.011905 0.023810
0.023810 0.023810 0.047619
0.035714 0.035714 0.071429

a*pinverse(a)*a =
1.000000 2.000000 3.000000
1.000000 2.000000 3.000000
2.000000 4.000000 6.000000

pinverse(a)*a*pinverse(a) =
0.011905 0.011905 0.023810
0.023810 0.023810 0.047619
0.035714 0.035714 0.071429
```

Example3

Calculate the Moore-Penrose pseudo inverse of a 3-by-3 non-singular matrix with $\text{rank}(a) == 3$.

```
#include <numeric.h>
#define M 3
#define N 3
/* pseudoinverse */
int main() {
    /* non-singular matrix rank(a) == 3 */
    array double a[M][N] = {1, 2, 3,
                            5, 5, 6,
                            7, 8, 9};

    int r;
    array double pinv[N][M];

    r = rank(a);
    pinv = pinverse(a);
    printf("rank(a) = %d\n", r);
    printf("pinverse(a) = \n%f\n", pinv);
    printf("a*pinverse(a)*a = \n%f\n", a*pinv*a);
    printf("pinverse(a)*a*pinverse(a) = \n%f\n", pinv*a*pinv);
}
```

Output3

```
rank(a) = 3
pinverse(a) =
-0.500000 1.000000 -0.500000
-0.500000 -2.000000 1.500000
0.833333 1.000000 -0.833333

a*pinverse(a)*a =
1.000000 2.000000 3.000000
5.000000 5.000000 6.000000
7.000000 8.000000 9.000000

pinverse(a)*a*pinverse(a) =
-0.500000 1.000000 -0.500000
-0.500000 -2.000000 1.500000
0.833333 1.000000 -0.833333
```

Example4

Calculate the Moore-Penrose pseudo inverse of a 2-by-3 matrix.

```
#include <numeric.h>
#define M 2
#define N 3
/* pseudoinverse */
int main() {
    array double a[M][N] = {7, 8, 1,
                            3, 6, 4}; /* m-by-n matrix */

    int r;
    array double pinv[N][M];

    r = rank(a);
    pinv = pinverse(a);
    printf("rank(a) = %d\n", r);
```



```

    printf("pinverse(a) = \n%f\n", pinv);
    printf("a*pinverse(a)*a = \n%f\n", a*pinv*a);
    printf("pinverse(a)*a*pinverse(a) = \n%f\n", pinv*a*pinv);
}

```

Output4

```

rank(a) = 2
pinverse(a) =
0.128000 -0.104000
0.030769 0.061538
-0.142154 0.235692

a*pinverse(a)*a =
7.000000 8.000000 1.000000
3.000000 6.000000 4.000000

pinverse(a)*a*pinverse(a) =
0.128000 -0.104000
0.030769 0.061538
-0.142154 0.235692

```

Example5

Calculate the Moore-Penrose pseudo inverse of a 3-by-2 matrix.

```

#include <numeric.h>
#define M 3
#define N 2
/* pseudoinverse */
int main() {
    array double a[M][N] = {7, 8,
                             1, 3,
                             6, 4}; /* m-by-n matrix */

    int r;
    array double pinv[N][M];

    r = rank(a);
    pinv = pinverse(a);
    printf("rank(a) = %d\n", r);
    printf("pinverse(a) = \n%f\n", pinv);
    printf("a*pinverse(a)*a = \n%f\n", a*pinv*a);
    printf("pinverse(a)*a*pinverse(a) = \n%f\n", pinv*a*pinv);
}

```

Output5

```

rank(a) = 2
pinverse(a) =
-0.053595 -0.209150 0.264052
0.139869 0.228758 -0.201307

a*pinverse(a)*a =
7.000000 8.000000
1.000000 3.000000
6.000000 4.000000

pinverse(a)*a*pinverse(a) =
-0.053595 -0.209150 0.264052
0.139869 0.228758 -0.201307

```

See Also

rank(), **svd()**, **qrdecomp()**, **inverse()**,
cinverse().

References

polycoef

Synopsis

```
#include <numeric.h>
```

```
int polycoef(array double complex p[&], array double complex x[&]);
```

Purpose

Calculates the coefficients of the polynomial with specified roots.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

p An output array containing the calculated coefficients of the polynomial.

x An input one-dimensional array containing the roots of the polynomial.

Description

This function calculates the coefficients of a polynomial with specified roots expressed as an array *x*. The coefficients are in the order of descending powers. The polynomial it represents is $p_0x^n + \dots + p_{n-1}x + p_n$.

Example1

Calculate the coefficients of polynomials

$$x^3 - 6x^2 - 72x - 27 = 0$$

and

$$x^4 - 12x^3 - 25x + 116 = 0$$

The roots of the polynomials are calculated by function **roots()**.

```
#include <numeric.h>
int main() {
    array double x1[3], p1[4] = {1, -6, -72, -27}; /* x^3-6x^2-72x-27 =0*/
    array double x2[4], p2[5] = {1, -12, 0, 25, 116}; /* x^4-12x^3-25x+116 =0 */
    array double complex z2[4], zp2[5];
    array double complex z3[4], p3[5] = {complex(3,4), complex(4,2),
                                         complex(5,3), complex(2,4), complex(1,5)};

    int status;

    roots(x1, p1);
    polycoef(p1, x1);
    printf("p1 from polycoef(p1, x1) = %0.2f\n", p1);

    /* x^4-12x^3-25x+116 =0 has two complex roots and two real roots */
    roots(x2, p2);
    roots(z2, p2);
    polycoef(p2, x2);
}
```

```

printf("p2 from polycoef(p2, x2) = %0.2f\n", p2);
polycoef(zp2, z2);
printf("zp2 from polycoef(zp2, z2) =\n %0.2f\n", zp2);

roots(z3, p3);
status = polycoef(p3, z3);
if(status == 0) {
    printf("p3 from polycoef(p3, z3) = \n %0.2f\n", p3);
    printf("complex(3,4)*p3 from polycoef(p3, z3) = \n %0.2f\n", complex(3,4)*p3);
    roots(z3, p3);
    printf("z3 from roots(z3, p3) = \n %0.2f\n", z3);
}
else
    printf("polycoef(p3, z3) failed\n");
}

```

Output1

```

p1 from polycoef(p1, x1) = 1.00 -6.00 -72.00 -27.00

p2 from polycoef(p2, x2) = 1.00 NaN NaN NaN NaN

zp2 from polycoef(zp2, z2) =
    complex(1.00,0.00) complex(-12.00,0.00) complex(0.00,0.00) complex(25.00,0.00)
    complex(116.00,0.00)

p3 from polycoef(p3, z3) =
    complex(1.00,0.00) complex(0.80,-0.40) complex(1.08,-0.44) complex(0.88,0.16)
    complex(0.92,0.44)

complex(3,4)*p3 from polycoef(p3, z3) =
    complex(3.00,4.00) complex(4.00,2.00) complex(5.00,3.00) complex(2.00,4.00)
    complex(1.00,5.00)

z3 from roots(z3, p3) =
    complex(0.23,1.28) complex(0.43,-0.73) complex(-0.75,-0.71) complex(-0.70,0.55)

```

Example2

The coefficients of polynomial with linearly spaced roots are calculated.

```

#include <numeric.h>
/* Wilkinson's famous example */
#define N 10
int main() {
    array double x[N], p[N+1];
    linspace(x, 1, N);
    printf("x = %f\n", x);
    polycoef(p, x);
    printf("p = %f\n", p);
    roots(x, p);
    printf("x = %f\n", x);
}

```

Output2

```

x = 1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000 9.000000
10.000000

```

```
p = 1.000000 -55.000000 1320.000000 -18150.000000 157773.000000 -902055.000000
3416930.000000 -8409500.000000 12753576.000000 -10628640.000000 3628800.000000
```

```
x = 9.000000 10.000000 8.000000 7.000000 6.000000 5.000000 4.000000 3.000000 2.000000
1.000000
```

Example3

The roots of a characteristic polynomial are calculated by function **eigensystem()** from a matrix, and then the coefficients are calculated again using function **polycoef()**.

```
#include <numeric.h>
int main() {
    array double a[3][3] = {0.8, 0.2, 0.1,
                           0.1, 0.7, 0.3,
                           0.1, 0.1, 0.6};
    array double evalues[3], evector[3][3];
    array double p[4];

    eigensystem(evalues, NULL, a);
    polycoef(p, evalues);
    printf("evalues = %f\n", evalues);
    printf("characteristic polynomial of matrix a = %f\n", p);
}
```

Output3

```
evalues = 1.000000 0.600000 0.500000
```

```
characteristic polynomial of matrix a = 1.000000 -2.100000 1.400000 -0.300000
```

See Also

roots(), **charpolycoef()**.

References

polyder

Synopsis

```
#include <numeric.h>
```

```
int polyder(array double complex y[&], array double complex x[&]);
```

Syntax

```
polyder(y, x)
```

Purpose

Take derivative of a polynomial.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

y A vector of coefficients of the derivative of the original polynomial.

x A vector of coefficients of the original polynomial.

Description

The vector of coefficient of polynomial *x* can be of any supported arithmetic data type and size. Conversion of the data to double complex type is performed internally. If vector *x* is of real type with size *n*, then the vector *y* for the derivative is of real type with size $(n - 1)$. If vector *x* is of complex type, then the vector *y* for the derivative is of complex type.

Algorithm

Given a polynomial $P(t)$ with coefficients of x_i , for $i = 0, 1, \dots, N$. That is

$$P(t) = x_0 t^n + x_1 t^{n-1} + x_2 t^{n-2} + \dots + x_{n-1} t + x_n$$

Then the first derivative of this polynomial is

$$\begin{aligned} P'(t) &= n * x_0 t^{n-1} + (n - 1) * x_1 t^{n-2} + (n - 2) * x_2 t^{n-3} + \dots + x_{n-1} \\ &= y_0 t^{n-1} + y_1 t^{n-2} + y_2 t^{n-3} + \dots + y_{n-1} \end{aligned}$$

Example

In this example:

polynomial: $(1 - i)x^2 + 2x + (3 + i) = 0$

derivative : $(2 + i2)x + 2 = 0$

polynomial: $x^2 + 2x + 3 = 0$

derivative : $2x + 2 = 0$

```
#include <stdio.h>
#include <numeric.h>
```

```

#define N 3          /* data array size */

int main() {
    int i;
    array double complex x[N],y[N-1];
    array double x1[N],y1[N-1];
    for (i=0;i<N;i++) {
        x[i]=complex(1+i,i-1);
        x1[i]=i+1;
    }

    printf("Coef. of polynomial b\n");
    printf("x=%6.3f\n", x);

    polyder(y,x);          /* derivative of x polynomial */
    printf("Coef. of derivative of polynomial of b\n");
    printf("y=%6.3f\n", y);

    printf("Coef. of polynomial d\n");
    printf("x1=%6.3f",x1);

    polyder(y1,x1);       /* derivative of B polynomial */
    printf("Coef. of derivative of polynomial of d\n");

    printf("y1=%6.3f",y1);
}

```

Output

```

Coef. of polynomial b
x=complex( 1.000,-1.000) complex( 2.000, 0.000) complex( 3.000, 1.000)

Coef. of derivative of polynomial of b
y=complex( 2.000,-2.000) complex( 2.000, 0.000)

Coef. of polynomial d
x1= 1.000 2.000 3.000
Coef. of derivative of polynomial of d
y1= 2.000 2.000

```

See Also**polyder2()**.**References**William H. Press, et al, *Numerical Recipes in C*, second edition, Cambridge University Press, 1997.

polyder2()

Synopsis

```
#include <numeric.h>
```

```
int polyder2(array double complex q[&], array double complex r[&], array double complex u[&],
             array double complex v[&]);
```

Syntax

```
polyder2(q, NULL, u, v)
```

```
polyder2(q, r, u, v)
```

Purpose

Calculate the derivative of product or quotient of two polynomials.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

q A vector contains the coefficients of the derivative of the product $(u * v)'$ of two polynomials u and v or coefficients of the numerator of the quotient $(u/v)'$ of two polynomials u and v .

r If input r is NULL the function implements the derivative of the product $(u * v)'$ of two polynomials u and v . Otherwise, it is a vector which contains the coefficients of the denominator from the derivative of the quotient $(u/v)'$ of two polynomials u and v .

u A vector of polynomial coefficients.

v A vector of polynomial coefficients.

Description

The operation of the derivative of a product or quotient of two polynomials u and v depends on argument r . If NULL is passed to argument r , function **polyder2()** calculates the derivative of the product $(u * v)'$ of two polynomials u and v . Otherwise, it calculates the derivative of the quotient $(u/v)'$. The coefficient vector of polynomials u and v can be of any supported arithmetic data type with sizes n and m , respectively. Conversion of the data to double complex type is performed internally. If both vectors u and v are real type, the vector q with size $n + m - 2$ and r with size $2 * m - 1$ are real type. If one of vectors u and v is complex type, vectors q and r are complex type.

Algorithm

Given two polynomials $U(x)$ and $V(x)$

$$\begin{aligned} U(x) &= u_0x^n + u_1x^{n-1} + u_2x^{n-2} + \cdots + u_{n-1}x + u_n \\ V(x) &= v_0x^m + v_1x^{m-1} + v_2x^{m-2} + \cdots + v_{m-1}x + v_m \end{aligned}$$

The derivative of the product of two polynomials $U(x) * V(x)$ becomes

$$\begin{aligned} Q(x) &= (U(x)V(x))' = U'(x)V(x) + V'(x)U(x) \\ R(x) &= NULL \end{aligned}$$

The derivative of the quotient of two polynomials $U(x)/V(x)$ becomes

$$\left(\frac{U(x)}{V(x)}\right)' = \frac{Q(x)}{R(x)} = \frac{U'(x)V(x) - V'(x)U(x)}{V^2(x)}$$

where $Q(x)$ and $R(x)$ are the numerator and denominator of the derivative, respectively. The polynomials for $Q(x)$, applicable to derivative of either the product or quotient of two polynomials, and $R(x)$ can be represented as

$$\begin{aligned} Q(x) &= q_0x^{n+m-1} + q_1x^{n+m-2} + \cdots + q_{n+m-2}x + q_{n+m-1} \\ R(x) &= r_0x^{2m-1} + r_1x^{2m-2} + \cdots + r_{2m-2}x + r_{2m-1} \end{aligned}$$

Example

Given

$$\begin{aligned} U(x) &= x + 2 \\ V(x) &= x^2 + 2x \end{aligned}$$

then

$$\begin{aligned} (U(x)V(x))' &= 3x^2 + 8x + 4 \\ \left(\frac{U(x)}{V(x)}\right)' &= \frac{-x^2 - 4x - 4}{x^4 + 4x^3 + 4x^2} \end{aligned}$$

```
#include <stdio.h>
#include <numeric.h>

#define N 3      /* data array size */
#define M 2      /* response array size */

int main() {
    int i;
    array double complex v[N], u[M], q[N+M-2], r[2*N-1];
    array double v1[N], u1[M], q1[N+M-2], r1[2*N-1];
    v[0]=complex(1,1); v[1]=complex(2,2); v[2]=0;
    v1[0]=1; v1[1]=2; v1[2]=0;
    for (i=0; i<M; i++) {
        u[i]=complex(i+1,i);
        u1[i]=i+1;
    }

    printf("Coef. of polynomial v=%4.2f\n", v);
    printf("Coef. of polynomial u=%4.2f\n", u);

    polyder2(q, NULL, u, v);      /* derivative of v*u polynomial */
    printf("Coef. of deriv of poly v*u=%4.2f\n", q);
    printf("\n");

    polyder2(q, r, u, v);      /* derivative of u/v polynomial */
    printf("Coef. of derivative of polynomial of u/v\n");
    printf("Numerator: %4.2f\n", q);
    printf("Denormantor:%4.2f\n", r);

    printf("Coef. of polynomial v=%6.3f\n", v1);
    printf("Coef. of polynomial u=%6.3f\n", u1);
}
```

```

polyder2(q1, NULL, u1, v1);          /* derivative of b*a polynomial */
printf("Coef. of derivative of polynomial of b*a=%6.3f\n", q1);
printf("\n");

polyder2(q1, r1, u1, v1);          /* derivative of a/b polynomial */
printf("Coef. of derivative of polynomial of a/b=\n");
printf("Numerator: %6.3f\n", q1);
printf("Denormator:%6.3f\n", r1);
}

```

Output

```

Coef. of polynomial v=complex(1.00,1.00) complex(2.00,2.00) complex(0.00,0.00)

Coef. of polynomial u=complex(1.00,0.00) complex(2.00,1.00)

Coef. of deriv of poly v*u=complex(3.00,3.00) complex(6.00,10.00) complex(2.00,6.00)

Coef. of derivative of polynomial of u/v
Numerator: complex(-1.00,-1.00) complex(-2.00,-6.00) complex(-2.00,-6.00)

Denormator:complex(0.00,2.00) complex(0.00,8.00) complex(0.00,8.00) complex(0.00,0.00)
complex(0.00,-0.00)

Coef. of polynomial v= 1.000  2.000  0.000

Coef. of polynomial u= 1.000  2.000

Coef. of derivative of polynomial of b*a= 3.000  8.000  4.000

Coef. of derivative of polynomial of a/b=
Numerator: -1.000 -4.000 -4.000

Denormator: 1.000  4.000  4.000  0.000 -0.000

```

See Also**polyder()**.**References**

William H. Press, et al, *Numerical Recipes in C*, second edition, Cambridge University Press, 1997.

polyeval

Synopsis

```
#include <numeric.h>
```

```
double polyeval(array double c[&], double x, ... /* [array double dp[&] */);
```

Syntax

```
polyeval(c, x)
```

```
polyeval(c, x, dp)
```

Purpose

Calculate the value of a polynomial and its derivatives.

Return Value

This function returns the value of a polynomial and its derivatives at a given point.

Parameters

c A vector of coefficients of a polynomial.

x A value of point *x* in which the value of polynomial is evaluated.

dp A vector of dimension *m*, it contains the values from the 1st to *m*th order derivatives of the polynomial at point *x*.

Description

The vector of coefficients of polynomial *c* can be of any supported real type and dimension *n*. The value of *x* can be of real type. Conversion of the data to double is performed internally. The return data is double type of the value of the polynomial. If the optional argument *dp* is specified, which shall be double type of dimension *m*, it passes back the values of 1st to *m*th order derivatives of the polynomial.

Algorithm

Given a polynomial

$$P(x) = c_0x^n + c_1x^{n-1} + \cdots + c_{n-3}x^3 + c_{n-2}x^2 + c_{n-1}x + c_n$$

The calculation of the polynomial value and its derivatives are implemented using the following algorithm

$$\begin{aligned} P &= (((\cdots (c_0x + c_1)x + \cdots + c_{n-3})x + c_{n-2})x + c_{n-1})x + c_n \\ P' &= (((\cdots (nc_0x + (n-1)c_1)x + \cdots + 3c_{n-3})x + 2c_{n-2})x + c_{n-1} \\ P'' &= (\cdots (n(n-1)c_0x + (n-1)(n-2)c_1)x + \cdots + 3 * 2c_{n-3})x + 2c_{n-2} \\ &\vdots = \vdots \quad \vdots \\ P^{(n-1)} &= n(n-1)(n-2) \cdots 3 * 2c_0x + (n-1)(n-2) \cdots 3 * 2 * 1c_1 \\ P^{(n)} &= n(n-1)(n-2) \cdots 3 * 2 * 1c_0 \\ P^{(n+1)} &= 0 \end{aligned}$$

Example

Calculate the value of the polynomial

$$C(x) = x^5 - 5x^4 + 10x^3 - 10x^2 + 5x - 1$$

at point $x = 0.1$ and its first, second and third order derivatives.

```
#include <numeric.h>
#include <stdio.h>

#define NC 6
#define ND 3
#define NP 10

int main() {
    int i, j, k;
    float c[NC]={1.0, -5.0, 10.0, -10.0, 5.0, -1.0};
    float vp[ND], d[ND+1], x;
    string_t title = "    x    polynomial    first deriv    second deriv    third deriv";

    x=0.1;
    d[0] = polyeval(c, x, vp);
    for (j=0; j<ND; j++) d[j+1]=vp[j];
    printf("%s\n", title);
    printf("%4.3f", x);
    for (i=0; i<=ND; i++) {
        printf("%14.6f", d[i]);
    }
    printf("\n\n");

    x=0.1;
    printf("x=%f\tf(x)=%f\n", x, polyeval(c, x, vp));
}
```

Output

```
    x    polynomial    first deriv    second deriv    third deriv
0.100    -0.590490     3.280500     -14.580000     48.599998
```

```
x=0.100000 f(x)=-0.590490
```

see also

polyevalarray(), **cpolyeval()**.

References

William H. Press, et al, *Numerical Recipes in C*, second edition, Cambridge University Press, 1997.

polyevalarray

Synopsis

```
#include <numeric.h>
int polyevalarray(array double complex &val, c[&], &x);
```

Syntax

```
polyevalarray(val, c, x)
```

Purpose

Polynomial evaluation for a sequence of points.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

val The array of output which contains the polynomial evaluation values for points *x*. The dimension of array *val* is the same as that of **array** *x*.

c A vector of coefficients for the polynomial.

x An array which contains the value of points in which the polynomial is evaluated.

Description

The vector *c* with the coefficients of a polynomial can be of any supported arithmetic data type and size. The value of *x* can be any arithmetic type. Conversion of the data to double complex is performed internally. Array *val*, the same dimension and size as *x*, contains the evaluation results. If both *c* and *x* are real type, *val* is real type. Otherwise, it shall be complex type.

Algorithm

Given polynomial

$$Y(x_i) = c_0x_i^n + c_1x_i^{n-1} + c_2x_i^{n-1} + \cdots + c_{n-1}x_i + c_n$$

the polynomial evaluation is implemented

$$Y_i = (((\cdots (c_0x_i + c_1)x_i + c_2)x_i + c_3)x_i + \cdots + c_{n-1})x_i + c_n$$

where subscript *i* traverses all elements of array *x*.

Example

Evaluate polynormal

$$C(x) = x^5 - 5x^4 + 10x^3 - 10x^2 + 5x - 1$$

at points $x = \{0.1, 0.2, \cdots, 1.9, 2.0\}$.

```
#include <numeric.h>
#include <stdio.h>

#define NC 6
#define N 10

int main()
{
    int i,j,k;
    array double c[NC]={1.0,-5.0,10.0,-10.0,5.0,-1.0};
    array double x[N],val[N];

    for (i=0; i<N; i++)
        x[i]=0.2*(i+1);
    printf("x value      function evaluate\n");
    polyevalarray(val,c,x);
    for (i=0; i<N; i++)
        printf("%f      %f\n",x[i],val[i]);
}
```

Output

```
x value      function evaluate
0.200000    -0.327680
0.400000    -0.077760
0.600000    -0.010240
0.800000    -0.000320
1.000000     0.000000
1.200000     0.000320
1.400000     0.010240
1.600000     0.077760
1.800000     0.327680
2.000000     1.000000
```

see also

polyeval(), **cpolyeval()**.

References

William H. Press, et al, *Numerical Recipes in C*, second edition, Cambridge University Press, 1997.

polyevalm

Synopsis

```
#include <numeric.h>
```

```
int polyevalm(array double complex y[&][&], array double complex c[&], array double complex x[&][&]);
```

Syntax

```
polyevalm(y, c, x)
```

Purpose

Evaluate a real or complex polynomial with matrix argument.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x Input square matrix. It contains data to be evaluated.

c A vector of coefficients of a polynomial. It can be **real** or **complex** data type.

y Output square matrix which contains data of the evaluated polynomial with given matrix argument *x*.

Description

This function evaluates the polynomial with given matrix argument *x*. In this function, The input square matrix *x* can be of any supported arithmetic data type. The polynomial is specified by the coefficient vector *c*, which can be any supported arithmetic data type. The output matrix *y* could be **real** or **complex** data type as required.

Example

Evaluation of real/complex polynomials of real/complex matrices.

```
#include <numeric.h>
#include <stdio.h>

#define NC 6
#define ND 3
#define N 20

int main()
{
    int i, j, k;
    array double c[4]={1,2,3,4};
    array double complex c1[4]={complex(2,1),2,3,4};
    array double x[3][3]={1,2,3,
                          3,4,5,
                          6,7,8};

    array double y[3][3];
    array double complex x1[3][3]={1,2,complex(2,3),
                                   3,4,5,
                                   6,7,complex(5,8)};
```

```

    array double complex y1[3][3];
    polyevalm(y, c, x);
    printf("x=\n%f\n", x);
    printf("y=\n%f\n", y);
    polyevalm(y1, c1, x1);
    printf("x1=\n%5.2f\n", x1);
    printf("y1=\n%5.2f\n", y1);

    return 0;
}

```

Output

```

x=
1.000000 2.000000 3.000000
3.000000 4.000000 5.000000
6.000000 7.000000 8.000000

y=
397.000000 501.000000 609.000000
729.000000 931.000000 1125.000000
1233.000000 1566.000000 1903.000000

x1=
complex( 1.00, 0.00) complex( 2.00, 0.00) complex( 2.00, 3.00)
complex( 3.00, 0.00) complex( 4.00, 0.00) complex( 5.00, 0.00)
complex( 6.00, 0.00) complex( 7.00, 0.00) complex( 5.00, 8.00)

y1=
complex(-82.00, 685.00) complex(-64.00, 878.00) complex(-706.00, 603.00)

complex(849.00, 1110.00) complex(1137.00, 1361.00) complex(-195.00, 1912.00)

complex(-12.00, 2034.00) complex(105.00, 2594.00) complex(-1864.00, 2023.00)

```

See Also

logm(), cfunm(), polyeval(), polyevalarray(), cpolyeval(), sqrtm().

References

G. H. Golub, C. F. Van Loan, *Matrix Computations* Third edition, The Johns Hopkins University Press, 1996

polyfit

Synopsis

```
#include <numeric.h>
```

```
int polyfit(double a[&], double x[&], double y[&], ... /* [double sig[ ], int ia[ ],
      double covar[ ][ ], double *chisq] */);
```

Syntax

```
polyfit(a, x, y)
```

```
polyfit(a, x, y, sig)
```

```
polyfit(a, x, y, sig, ia)
```

```
polyfit(a, x, y, sig, ia, covar)
```

```
polyfit(a, x, y, sig, ia, covar, chisq)
```

Purpose

Fit a set of data points x, y to a polynomial function.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

a An output array which contains coefficients of polynomial.

x An array which contains variable values of the data set.

y An array which contains function values of the data set.

sig An input array which contains individual standard (measurement error) deviations of data set. It can be set to 1 if unknown.

ia An input array. Those components of nonzero entries shall be fitted, and those components of zero entries shall be held at their input values.

$covar$ The Covariance matrix of fitting results, describing the fit of data to the model. See algorithm for definition.

$chisq$ Chi-square of fitting.

Description

Given a set of data points expressed in arrays x and y with the same dimension, use χ^2 minimization to fit the data set using a polynomial coefficients a . The fitted coefficients are passed by array a . The program also passes χ^2 , covariance matrix $covar$. If the parameters of ia are held fixed, the corresponding components of covariance matrix will be zero.

Algorithm

The algorithm of the function **polyfit()** is based on that of the function **curvefit()**. For polynomial fitting, the base functions for **curvefit()** are set to the terms of polynomials internally. The general form of the fitting formula is

$$y(x) = \sum_{k=1}^M a_k x^k$$

The coefficients a_k are determined by minimizing chi-square which is defined as

$$\chi^2 = \sum_{i=1}^N \left(\frac{y_i - \sum_{k=1}^M a_k x_i^k}{\sigma_i} \right)^2$$

where σ_i is the standard deviation of the i th data point. If the standard deviations are unknown, they can be set to the constant value $\sigma = 1$. By defining a $n \times m$ matrix α with element (k, j) as

$$\alpha_{kj} = \sum_{i=1}^N \frac{x_i^j x_i^k}{\sigma_i^2}$$

The element (k, j) of the covariance matrix can be expressed as

$$cover_{kj} = [\alpha]_{kj}^{-1}$$

Example1

Fit the data set created by the polynomial

$$8x^4 + 5x^3 + 3x^2 + 6x + 7$$

The deviation with a fraction of a uniform random number is added to data set.

```
#include <numeric.h>
#include <stdio.h>
#define NPT 100          /* Number of data points */
#define NTERM 5         /* Number of terms */

int main() {
    int i, j, status;
    array double u[NPT], x[NPT], y[NPT], a[NTERM];

    /* Create a data set of NTERM order polynomial with uniform random deviation*/
    linspace(x, 0.1, 0.1*NPT);
    y = 8*x.*x.*x.*x + 5*x.*x.*x + 3*x.*x + 6*x + (array double[NPT])(7);

    /* put uniform random deviation on data set */
    urand(u);
    y += 0.1*u;
    status=polyfit(a, x, y);
    if(status) printf("Abnormal fit");
    printf("\n%11s\n", "Coefficients");
    for (i=0; i<NTERM; i++)
        printf("  a[%1d] = %8.6f \n", i, a[i]);
    printf("y = %f at x = %f\n", polyeval(a, 2.0), 2.0);
}
```

Output1

```
Coefficients
a[0] = 8.000048
a[1] = 4.999537
a[2] = 2.998743
a[3] = 6.015915
a[4] = 7.033636
y = 199.057500 at x = 2.000000
```

Example2

Fit the data set created by the polynomial

$$8x^4 + 5x^3 + 3x^2 + 6x + 7$$

The deviation with a fraction of a uniform random number is added to data set. This example uses all the options of arguments of function **polyfit()**. The coefficients $a[1]$ and $a[3]$ are fixed and the standard deviation of data set is assumed to be a constant 0.1. The uncertainty which is defined as $dev[i] = \sqrt{covar[i][i]}$ is calculated.

```
#include <stdio.h>
#include <numeric.h>
#define NPT 100                /* Number of data points */
#define NTERM 5                /* Number of term */
#define SPREAD 0.1            /* Number of term */

int main() {
    int i, j, status, ia[NTERM];
    array double u[NPT], x[NPT], y[NPT], a[NTERM], dev[NTERM];
    array double sig[NPT], covar[NTERM][NTERM];
    double chisq;

    /* Create a data set of NTERM order polynomial with gaussian deviation*/
    linspace(x, 0.1, 0.1*NPT);
    y = 8*x.*x.*x.*x + 5*x.*x.*x + 3*x.*x + 6*x + (array double[NPT])(7);

    /* put uniform random deviation on data set */
    urand(u);
    y += 0.1*u;
    sig=(array double[NPT])(SPREAD);

    ia[0] = 1; ia[1] = 0; ia[2] = 1; ia[3] = 0, ia[4] = 1;
    a[1] = 5; a[3] = 6;

    status=polyfit(a, x, y, sig, ia, covar, &chisq);
    dev = sqrt(diagonal(covar));
    if(status) printf("Abnormal fit");
    printf("\n%11s%23s\n", "Coefficients");
    printf("\n%11s %21s\n", "parameter", "uncertainty");
    for (i=0; i<NTERM; i++)
        printf("  a[%1d] = %8.6f %12.6f\n",
              i, a[i], dev[i]);
    printf("chi-square = %12f\n", chisq);
    printf("full covariance matrix\n");
    printf("%12f", covar);
}

```

Output2

```
Coefficients
parameter      uncertainty
a[0] = 8.000007    0.000013
a[1] = 5.000000    0.000000
a[2] = 2.999413    0.001165
a[3] = 6.000000    0.000000
a[4] = 7.057378    0.018870
chi-square =      7.414554

```

```
full covariance matrix
  0.000000    0.000000   -0.000000    0.000000    0.000000
  0.000000    0.000000    0.000000    0.000000    0.000000
 -0.000000    0.000000    0.000001    0.000000   -0.000016
  0.000000    0.000000    0.000000    0.000000    0.000000
  0.000000    0.000000   -0.000016    0.000000    0.000356
```

See Also

curvefit(), **expfit()**, **linearfit()**, **polyeval()**, **quadraticfit()**, **std()**.

References

William H. Press, Saul A. Teukolsky, William T. Vetterling and Brian P. Flannery, *Numeric Recipes in C*, Second Edition, Cambridge University Press, 1997.

polygonperi

Synopsis

```
#include <numeric.h>
```

```
double polygonperi(array double px[], array double py[]);
```

Syntax

```
polygonperi(px, py)
```

Purpose

Calculate the perimeter of a polygon.

Return Value

This function returns the perimeter of a polygon.

Parameters

px An array containing the x coordinates of the vertices of the polygon.

py An array containing the y coordinates of the vertices of the polygon.

Description

The function runs the sum of the distance between two adjacent vertices of a polygon.

Algorithm

Use the function **hypot()** to calculate the distance between two points on the coordinate plane.

Example

```
/* File: polygonperi.ch */
#include <numeric.h>
#include <chplot.h>
double px[] = {3, 6, 6, 3};
double py[] = {2, 2, 4, 4};
double perimeter;
CPlot plot;

perimeter = polygonperi(px, py);
printf("perimeter = %.2lf\n", perimeter);

plot.polygon(px, py);
plot.axisRange(PLOT_AXIS_XY, 0, 8);
plot.plotting();
```

Output

```
perimeter = 10.00
```

see also

plot.polygon().

product

Synopsis

```
#include <numeric.h>
```

```
double product(array double &a, ... /* [array double v[:]] */);
```

Syntax

```
product(a)
```

```
product(a, v)
```

Purpose

Calculate products of all elements and products of the elements of each rows of an array.

Return Value

This function returns the product of all elements.

Parameters

a The input array for which the products are calculated.

v The output array which contains the products of each row of a two-dimensional array.

Description

This function calculates the product of all elements in an array of any dimension. If the array is a two-dimensional matrix, the function can calculate the products of each row. The product of all elements of the array is returned by the function and the products of the elements of each row are passed out by the argument *v*. The input array can be of any dimension and any arithmetic data types.

Example1

Calculate the products of all elements of arrays with different data types and dimensions.

```
#include <numeric.h>
int main() {
    double a[3] = {1,2,3};
    double b[2][3] = {1,2,3,4,5,6};
    int b1[2][3] = {1,2,3,4,5,6};
    double c[2][3][5];
    c[0][0][0] = 10;
    double prod;

    prod = product(a);
    printf("prod(a) = %f\n", prod);
    prod = product(b);
    printf("product(b) = %f\n", prod);
    prod = product(b1);
    printf("product(b1) = %f\n", prod);
    prod = product(c);
    printf("product(c) = %f\n", prod);
}
```

Output1

```
prod(a) = 6.000000
product(b) = 720.000000
product(b1) = 720.000000
product(c) = 0.000000
```

Example2

Calculate the products of the elements of each row of the two-dimensional arrays with different data types and dimensions.

```
#include <numeric.h>
int main() {
    double a[2][3] = {1,2,3,
                     4,5,6};
    array double b[3][4] = {1,2,3,4,
                           5,6,7,8,
                           1,2,3,4};
    array double productv1[2], productv2[3];

    product(a, productv1);
    printf("product(a, productv1) = %f\n", productv1);
    product(b, productv2);
    printf("product(b, productv2) = %f\n", productv2);
}
```

Output2

```
product(a, productv1) = 6.000000 120.000000

product(b, productv2) = 24.000000 1680.000000 24.000000
```

See Also

cproduct(), **cumprod()**, **mean()**, **median()**, **sum()**.

References

qrdecomp

Synopsis

```
#include <numeric.h>
```

```
int qrdecomp(array double complex a[&][&], array double complex q[&][&],
             array double complex r[&][&]);
```

Purpose

Perform the orthogonal-triangular QR decomposition of a matrix.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

a Input matrix.

q Output unitary matrix.

r Output upper triangular matrix.

Description

This function performs a QR decomposition of a matrix a so that $a = qr$. The output q is a unitary matrix and r is an upper triangular matrix. Assume the size of input matrix a is $m \times n$. If $m \leq n$, there is only one type of output matrices q and r . The size of output matrix q is $m \times m$ and matrix r is $m \times n$. If $m > n$, there are two types of output matrices q and r . One is full size in which matrix q is $m \times m$ and r is $m \times n$. Another is an economy size in which matrix q is $m \times n$ and r is $n \times n$. This function checks the size of arguments and automatically selects the corresponding output type.

Example1

Perform QR decomposition of 4×4 , 2×4 and 4×2 matrices. The 4×2 matrices have two different types of output corresponding to the argument sizes passed in.

```
#include <numeric.h>
int main() {
    int m = m, n = 4;
    array double a[4][4] = {1, 5, -7, 4,
                           3, 2, 5, 2,
                           3, 2, 5, 2,
                           3, 9, 5, 2}; /* a[m][n] */

    array double q[n][n];
    array double r[n][n];

    qrdecomp(a, q, r);
    printf("q = \n%f\n", q);
    printf("r = \n%f\n", r);
    printf("q*r = \n%f\n", q*r);

    m = 2;
    n = 4;
    array double a1[2][4] = {1, 5, -7, 4,
                             3, 2, 5, 2}; /* a[m][n] */

    array double q1[m][m]; // m<n
    array double r1[m][n];
```



```

qrdecomp(a1, q1, r1);
printf("q1^T*q1 = \n%f\n", transpose(q1)*q1);
printf("q1 = \n%f\n", q1);
printf("r1 = \n%f\n", r1);
printf("q1*r1 = \n%f\n", q1*r1);

array double a2[4][2] = {1, 5,
                        -7, 4,
                        3, 2,
                        5, 2}; /* a[m][n] */

m = 4;
n = 2;
array double q2[m][n], q21[m][m]; // m>=n
array double r2[n][n], r21[m][n];
int status;

status = qrdecomp(a2, q21, r21);
if(status == 0) {
    printf("q21^T*q21 = \n%f\n", transpose(q21)*q21);
    printf("q21 = \n%f\n", q21);
    printf("r21 = \n%f\n", r21);
    printf("q21*r21 = \n%f\n", q21*r21);
}
else
    printf("error: numerical error in qrdecomp()\n");

status = qrdecomp(a2, q2, r2);
if(status == 0) {
    printf("q2^T*q2 = \n%f\n", transpose(q2)*q2);
    printf("q2 = \n%f\n", q2);
    printf("r2 = \n%f\n", r2);
    printf("q2*r2 = \n%f\n", q2*r2);
}
else
    printf("error: numerical error in qrdecomp()\n");
}

```

Output1

```

q =
-0.188982 0.511914 0.837991 0.000000
-0.566947 -0.405266 0.119713 0.707107
-0.566947 -0.405266 0.119713 -0.707107
-0.566947 0.639893 -0.518756 -0.000000

r =
-5.291503 -8.315218 -7.181325 -4.157609
0.000000 6.697548 -4.436592 1.706382
0.000000 0.000000 -7.262591 2.793304
0.000000 0.000000 0.000000 0.000000

q*r =
1.000000 5.000000 -7.000000 4.000000
3.000000 2.000000 5.000000 2.000000
3.000000 2.000000 5.000000 2.000000
3.000000 9.000000 5.000000 2.000000

q1^T*q1 =
1.000000 0.000000

```

```
0.000000 1.000000
```

```
q1 =
-0.316228 -0.948683
-0.948683 0.316228
```

```
r1 =
-3.162278 -3.478505 -2.529822 -3.162278
0.000000 -4.110961 8.221922 -3.162278
```

```
q1*r1 =
1.000000 5.000000 -7.000000 4.000000
3.000000 2.000000 5.000000 2.000000
```

```
q21^T*q21 =
1.000000 0.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 1.000000
```

```
q21 =
-0.109109 -0.730552 -0.408074 -0.536530
0.763763 -0.491027 0.173257 0.381499
-0.327327 -0.323359 0.873744 -0.157694
-0.545545 -0.347312 -0.200072 0.736021
```

```
r21 =
-9.165151 0.763763
0.000000 -6.958209
0.000000 0.000000
0.000000 0.000000
```

```
q21*r21 =
1.000000 5.000000
-7.000000 4.000000
3.000000 2.000000
5.000000 2.000000
```

```
q2^T*q2 =
1.000000 0.000000
0.000000 1.000000
```

```
q2 =
-0.109109 -0.730552
0.763763 -0.491027
-0.327327 -0.323359
-0.545545 -0.347312
```

```
r2 =
-9.165151 0.763763
0.000000 -6.958209
```

```
q2*r2 =
1.000000 5.000000
-7.000000 4.000000
3.000000 2.000000
5.000000 2.000000
```

Example2Perform QR decomposition of 2×4 complex matrix

```

#include <numeric.h>
int main() {
    array double complex a[2][4] = {1, 5, -7, 4,
                                     3, 3, 2, -13}; /* a[m][n] */
    array double complex a1[2][4] = {complex(1,2), 5, -7, 4,
                                     3, 3, 2, -13}; /* a[m][n] */

    int m = 2, n = 4;
    array double complex q[m][m]; // in the case of m<n
    array double complex r[m][n] ;

    qrdecomp(a, q, r);
    printf("q^H*q = \n%f\n", conj(transpose(q))*q);
    printf("q = \n%f\n", q);
    printf("r = \n%f\n", r);
    printf("q*r = \n%f\n", q*r);

    qrdecomp(a1, q, r);
    printf("q^H*q = \n%f\n", conj(transpose(q))*q);
    printf("q = \n%f\n", q);
    printf("r = \n%f\n", r);
    printf("q*r = \n%f\n", q*r);

    m = 4;
    n = 2;
    array double complex a2[4][2] = {complex(1,2), complex(3,5),
                                     complex(-7,2), complex(3,4),
                                     complex(3,0), complex(2,3),
                                     complex(2,0), complex(2,-13)};
                                     // a[m][n]
    array double complex q2[m][m]; // in the case of m>=n
    array double complex r2[m][n] ;
    qrdecomp(a2, q2, r2);
    printf("q2^H*q2 = \n%f\n", conj(transpose(q2))*q2);
    printf("q2 = \n%f\n", q2);
    printf("r2 = \n%f\n", r2);
    printf("q2*r2 = \n%f\n", q2*r2);
}

```

Output2

```

q^H*q =
complex(1.000000,0.000000) complex(0.000000,0.000000)
complex(0.000000,0.000000) complex(1.000000,0.000000)

q =
complex(-0.316228,0.000000) complex(-0.948683,0.000000)
complex(-0.948683,-0.000000) complex(0.316228,0.000000)

r =
complex(-3.162278,0.000000) complex(-4.427189,0.000000) complex(0.316228,0.000000)
complex(11.067972,0.000000)
complex(0.000000,0.000000) complex(-3.794733,0.000000) complex(7.273239,0.000000)
complex(-7.905694,0.000000)

q*r =
complex(1.000000,0.000000) complex(5.000000,0.000000) complex(-7.000000,0.000000)
complex(4.000000,0.000000)

```

```

complex(3.000000,0.000000) complex(3.000000,0.000000) complex(2.000000,0.000000)
complex(-13.000000,0.000000)

q^H*q =
complex(1.000000,0.000000) complex(-0.000000,0.000000)
complex(-0.000000,-0.000000) complex(1.000000,0.000000)

q =
complex(-0.267261,-0.534522) complex(0.717137,-0.358569)
complex(-0.801784,0.000000) complex(-0.000000,0.597614)

r =
complex(-3.741657,0.000000) complex(-3.741657,2.672612) complex(0.267261,-3.741657)
complex(9.354143,2.138090)
complex(0.000000,0.000000) complex(3.585686,0.000000) complex(-5.019960,-3.705209)
complex(2.868549,9.203260)

q*r =
complex(1.000000,2.000000) complex(5.000000,0.000000) complex(-7.000000,0.000000)
complex(4.000000,-0.000000)
complex(3.000000,-0.000000) complex(3.000000,0.000000) complex(2.000000,0.000000)
complex(-13.000000,-0.000000)

q2^H*q2 =
complex(1.000000,0.000000) complex(0.000000,0.000000) complex(0.000000,-0.000000)
complex(-0.000000,0.000000)
complex(0.000000,-0.000000) complex(1.000000,0.000000) complex(0.000000,-0.000000)
complex(0.000000,0.000000)
complex(0.000000,0.000000) complex(0.000000,0.000000) complex(1.000000,0.000000)
complex(-0.000000,0.000000)
complex(-0.000000,-0.000000) complex(0.000000,-0.000000) complex(-0.000000,-0.000000)
complex(1.000000,0.000000)

q2 =
complex(-0.118678,-0.237356) complex(-0.097267,-0.380224) complex(-0.535487,-0.061199)
complex(0.376291,-0.586055)
complex(0.830747,-0.237356) complex(-0.175866,0.098249) complex(0.279893,0.090757)
complex(0.027154,-0.354437)
complex(-0.356034,0.000000) complex(-0.110039,-0.362539) complex(0.784519,0.011697)
complex(0.238045,-0.239634)
complex(-0.237356,0.000000) complex(-0.119864,0.804660) complex(0.041031,0.075111)
complex(0.490320,-0.184604)

r2 =
complex(-8.426150,0.000000) complex(-1.186782,6.171265)
complex(0.000000,0.000000) complex(-14.335517,0.000000)
complex(0.000000,0.000000) complex(0.000000,0.000000)
complex(0.000000,0.000000) complex(0.000000,0.000000)

q2*r2 =
complex(1.000000,2.000000) complex(3.000000,5.000000)
complex(-7.000000,2.000000) complex(3.000000,4.000000)
complex(3.000000,-0.000000) complex(2.000000,3.000000)
complex(2.000000,0.000000) complex(2.000000,-13.000000)

```

See Also

orthonormalbase(), ludcomp(), nullspace() qrdelete(), qrinsert().

References

E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

qrdelete

Synopsis

```
#include <numeric.h>
```

```
int qrdelete(array double complex qout[&][&], array double complex rout[&][&],
             array double complex q[&][&], array double complex r[&][&], int j);
```

Purpose

Remove a column in QR factorized matrix.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

qout Output Q matrix in QR factorization of the matrix with a column removed.

rout Output R matrix in QR factorization of the matrix with a column removed.

q Input Q matrix in QR factorization of the original matrix.

r Input R matrix in QR factorization of the original matrix.

j Index specifying which column to be removed.

Description

This function changes Q and R to be the factorization of the matrix a with one of its columns deleted. Assume that Q and R is the original factorization of the matrix a which has n columns, this function returns $qout$ and $rout$ that are the factorization of the matrix obtained by removing the $(j + 1)$ th column of a .

Example1

Change QR decomposition of 6×6 complex matrix a to the decomposition of the matrix obtained by removing the third column of a .

```
#include <numeric.h>
#include <stdio.h>
int main() {
    array double complex a[6][6] = {1, 2, 3, 7.8, 8.9, 10,
                                     4, 5, 6, 1.2, 2.3, 3,
                                     7, 8, 9, 4.5, 5.6, 6,
                                     11, 15, 2.5, 5.5, 8.5, 0.1,
                                     13, 10, 7.5, 4.5, 1.5, 10,
                                     3.5, 6.5, 9.5, 10.5, 0.2, 31};

    array double complex q[6][6], r[6][6];
    array double complex qout[6][6], rout[6][5];

    qrdecomp(a, q, r);
    qrdelete(qout, rout, q, r, 2);

    printf("qout = %.4f\nrout = %.4f\n", qout, rout);
    printf("qout*rout = %.4f\n", qout*rout);

    return 0;
}
```

Output1

```

qout = complex(-0.0521,0.0000) complex(-0.1565,0.0000) complex(0.6493,0.0000)
complex(0.7075,0.0000) complex(-0.0466,0.0000) complex(0.2201,0.0000)

complex(-0.2084,-0.0000) complex(-0.1186,0.0000) complex(-0.1817,0.0000)
complex(0.0216,0.0000) complex(0.8093,0.0000) complex(0.5040,0.0000)

complex(-0.3648,-0.0000) complex(-0.0807,0.0000) complex(0.0181,0.0000)
complex(0.2167,0.0000) complex(0.3968,0.0000) complex(-0.8098,0.0000)

complex(-0.5732,-0.0000) complex(-0.5377,0.0000) complex(-0.4142,0.0000)
complex(0.1481,0.0000) complex(-0.4111,0.0000) complex(0.1408,0.0000)

complex(-0.6774,-0.0000) complex(0.6717,0.0000) complex(0.1903,0.0000)
complex(-0.1300,0.0000) complex(-0.1219,0.0000) complex(0.1479,0.0000)

complex(-0.1824,-0.0000) complex(-0.4633,0.0000) complex(0.5807,0.0000)
complex(-0.6428,0.0000) complex(0.0396,0.0000) complex(-0.0112,0.0000)

rout = complex(-19.1898,0.0000) complex(-20.6229,0.0000) complex(-10.4144,0.0000)
complex(-8.9110,0.0000) complex(-15.8209,0.0000)
complex(0.0000,0.0000) complex(-5.9115,0.0000) complex(-6.5255,0.0000)
complex(-5.7735,0.0000) complex(-10.1036,0.0000)
complex(0.0000,0.0000) complex(0.0000,0.0000) complex(9.6041,0.0000)
complex(2.3431,0.0000) complex(25.9217,0.0000)
complex(0.0000,0.0000) complex(0.0000,0.0000) complex(0.0000,0.0000)
complex(8.4954,0.0000) complex(-12.7715,0.0000)
complex(0.0000,0.0000) complex(0.0000,0.0000) complex(0.0000,0.0000)
complex(0.0000,0.0000) complex(4.3108,0.0000)
complex(0.0000,0.0000) complex(0.0000,0.0000) complex(0.0000,0.0000)
complex(0.0000,0.0000) complex(0.0000,0.0000)

qout*rout = complex(1.0000,0.0000) complex(2.0000,0.0000) complex(7.8000,0.0000)
complex(8.9000,0.0000) complex(10.0000,0.0000)
complex(4.0000,0.0000) complex(5.0000,0.0000) complex(1.2000,0.0000)
complex(2.3000,0.0000) complex(3.0000,0.0000)
complex(7.0000,0.0000) complex(8.0000,0.0000) complex(4.5000,0.0000)
complex(5.6000,0.0000) complex(6.0000,0.0000)
complex(11.0000,0.0000) complex(15.0000,0.0000) complex(5.5000,0.0000)
complex(8.5000,0.0000) complex(0.1000,0.0000)
complex(13.0000,0.0000) complex(10.0000,0.0000) complex(4.5000,0.0000)
complex(1.5000,0.0000) complex(10.0000,0.0000)
complex(3.5000,0.0000) complex(6.5000,0.0000) complex(10.5000,0.0000)
complex(0.2000,0.0000) complex(31.0000,0.0000)

```

Example2

Change QR decomposition of 4×4 complex matrix a to the decomposition of the matrix obtained by removing the first column of a .

```

#include <numeric.h>
#include <stdio.h>
int main() {
    array double complex a[4][4] = {
        -0.7605, -0.2862, complex(-0.2748, 0.6488), complex(-0.2748,-0.6488),
        -0.4394, 0.6886, complex(-0.4194, -0.4555), complex(-0.4194,0.4555),
        0.1572, 0.3920, complex(0.0360, 0.0915), complex(0.0360,-0.0915),
        -0.4514, -0.5388, complex(-0.3322,-0.0089), complex(-0.3322, 0.0089)
    }
}

```

```

};
array double complex q[4][4], r[4][4];
array double complex qout[4][4], rout[4][3];

qrdecomp(a, q, r);
qrdelete(qout, rout, q, r, 0);
printf("qout = %.4f\nrout = %.4f\n", qout, rout);
printf("qout*rout = %.4f\n", qout*rout);

return 0;
}

```

Output2

```

qout = complex(-0.2862,0.0000) complex(-0.3148,0.5825) complex(-0.3580,-0.5464)
complex(0.2299,0.0000)
complex(0.6886,0.0000) complex(-0.4589,-0.1572) complex(-0.4933,0.1320)
complex(0.1727,-0.0000)
complex(0.3920,0.0000) complex(0.0480,0.3054) complex(0.0434,-0.2804)
complex(-0.8187,0.0000)
complex(-0.5388,0.0000) complex(-0.3843,-0.2882) complex(-0.4086,0.2550)
complex(-0.4970,0.0000)

rout = complex(1.0000,0.0000) complex(-0.0170,-0.4587) complex(-0.0170,0.4587)

complex(0.0000,0.0000) complex(0.8884,0.0000) complex(-0.0717,-0.0252)

complex(0.0000,0.0000) complex(0.0000,0.0000) complex(0.8852,0.0000)

complex(0.0000,0.0000) complex(0.0000,0.0000) complex(0.0000,0.0000)

qout*rout = complex(-0.2862,0.0000) complex(-0.2748,0.6488) complex(-0.2748,-0.6488)

complex(0.6886,0.0000) complex(-0.4194,-0.4555) complex(-0.4194,0.4555)

complex(0.3920,0.0000) complex(0.0360,0.0915) complex(0.0360,-0.0915)

complex(-0.5388,0.0000) complex(-0.3322,-0.0089) complex(-0.3322,0.0089)

```

Example3

The same as Example 1, except that all matrices contain real numbers.

```

#include <numeric.h>
#include <stdio.h>
int main() {
    array double a[6][6] = {1, 2, 3, 7.8, 8.9, 10,
                           4, 5, 6, 1.2, 2.3, 3,
                           7, 8, 9, 4.5, 5.6, 6,
                           11, 15, 2.5, 5.5, 8.5, .1,
                           13, 10, 7.5, 4.5, 1.5, 10,
                           3.5, 6.5, 9.5, 10.5, .2, 31};
    array double q[6][6], r[6][6];
    array double qout[6][6], rout[6][5];

    qrdecomp(a, q, r);
    qrdelete(qout, rout, q, r, 2);
}

```



```

    printf("qout = %.4f\nrout = %.4f\n", qout, rout);
    printf("qout*rout = %.4f\n", qout*rout);

    return 0;
}

```

Output3

```

qout = -0.0521 -0.1565 0.6493 0.7075 -0.0466 0.2201
-0.2084 -0.1186 -0.1817 0.0216 0.8093 0.5040
-0.3648 -0.0807 0.0181 0.2167 0.3968 -0.8098
-0.5732 -0.5377 -0.4142 0.1481 -0.4111 0.1408
-0.6774 0.6717 0.1903 -0.1300 -0.1219 0.1479
-0.1824 -0.4633 0.5807 -0.6428 0.0396 -0.0112

rout = -19.1898 -20.6229 -10.4144 -8.9110 -15.8209
0.0000 -5.9115 -6.5255 -5.7735 -10.1036
0.0000 0.0000 9.6041 2.3431 25.9217
0.0000 0.0000 0.0000 8.4954 -12.7715
0.0000 0.0000 0.0000 0.0000 4.3108
0.0000 0.0000 0.0000 0.0000 0.0000

qout*rout = 1.0000 2.0000 7.8000 8.9000 10.0000
4.0000 5.0000 1.2000 2.3000 3.0000
7.0000 8.0000 4.5000 5.6000 6.0000
11.0000 15.0000 5.5000 8.5000 0.1000
13.0000 10.0000 4.5000 1.5000 10.0000
3.5000 6.5000 10.5000 0.2000 31.0000

```

See Also

qrinsert(), **qrdecomp()**.

References

E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

qrinsert

Synopsis

```
#include <numeric.h>
```

```
int qrinsert(array double complex qout[&][&], array double complex rout[&][&],
             array double complex q[&][&], array double complex r[&][&],
             int j, array double complex x[&]);
```

Purpose

Insert a column in QR factorized matrix.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

qout Output Q matrix in QR factorization of the matrix with an inserted column.

rout Output R matrix in QR factorization of the matrix with an inserted column.

q Input Q matrix in QR factorization of the original matrix.

r Input R matrix in QR factorization of the original matrix.

j Index specifying where to insert the extra column.

x The column to be inserted.

Description

This function changes Q and R to be the factorization of the matrix a with an extra column inserted. Assume that Q and R is the original factorization of the matrix a which has n columns, this function returns *qout* and *rout* that are the factorization of the matrix obtained by inserting an extra column x right before the $(j + 1)$ th column of a . If j is n , then x is inserted after the last column of a .

Example1

Change QR decomposition of 6×6 complex matrix a to the decomposition of the matrix obtained by inserting x before the third column of a .

```
#include <numeric.h>
#include <stdio.h>
int main() {
    array double complex a[6][6] = {1, 2, 3, 7.8, 8.9, 10,
                                     4, 5, 6, 1.2, 2.3, 3,
                                     7, 8, 9, 4.5, 5.6, 6,
                                     11, 15, 2.5, 5.5, 8.5, .1,
                                     13, 10, 7.5, 4.5, 1.5, 10,
                                     3.5, 6.5, 9.5, 10.5, .2, 31};

    array double complex q[6][6], r[6][6];
    array double complex qout[6][6], rout[6][7];
    array double complex x[6] = {10, 11, 12, 13, 14, 15};

    qrdecomp(a, q, r);
    qrinsert(qout, rout, q, r, 2, x);
    printf("qout = %.4f\nrout = %.4f\n", qout, rout);
}
```

```

    printf("qout*rout = %.4f\n", qout*rout);

    return 0;
}

```

Output1

```

qout = complex(-0.0521,0.0000) complex(-0.1565,0.0000) complex(0.5705,0.0000)
complex(-0.6514,0.0000) complex(-0.1766,0.0000) complex(0.4379,0.0000)

complex(-0.2084,-0.0000) complex(-0.1186,0.0000) complex(0.3449,0.0000)
complex(-0.0204,0.0000) complex(0.8873,0.0000) complex(-0.1891,0.0000)

complex(-0.3648,-0.0000) complex(-0.0807,0.0000) complex(0.1193,0.0000)
complex(0.6106,0.0000) complex(0.0177,0.0000) complex(0.6877,0.0000)

complex(-0.5732,-0.0000) complex(-0.5377,0.0000) complex(-0.5401,0.0000)
complex(-0.3008,0.0000) complex(-0.0048,0.0000) complex(-0.0062,0.0000)

complex(-0.6774,-0.0000) complex(0.6717,0.0000) complex(0.1117,0.0000)
complex(-0.1035,0.0000) complex(-0.1586,0.0000) complex(-0.2039,0.0000)

complex(-0.1824,-0.0000) complex(-0.4633,0.0000) complex(0.4869,0.0000)
complex(0.3180,0.0000) complex(-0.3950,0.0000) complex(-0.5078,0.0000)

rout = complex(-19.1898,0.0000) complex(-20.6229,0.0000) complex(-26.8632,0.0000)
complex(-12.9365,0.0000) complex(-10.4144,0.0000) complex(-8.9110,0.0000)
complex(-15.8209,0.0000)
complex(0.0000,0.0000) complex(-5.9115,0.0000) complex(-8.3741,0.0000)
complex(-2.6154,0.0000) complex(-6.5255,0.0000) complex(-5.7735,0.0000)
complex(-10.1036,0.0000)
complex(0.0000,0.0000) complex(0.0000,0.0000) complex(12.7767,0.0000)
complex(8.9680,0.0000) complex(8.0453,0.0000) complex(2.2131,0.0000)
complex(23.6127,0.0000)
complex(0.0000,0.0000) complex(0.0000,0.0000) complex(0.0000,0.0000)
complex(4.9123,0.0000) complex(-1.1385,0.0000) complex(-5.0738,0.0000)
complex(5.8830,0.0000)
complex(0.0000,0.0000) complex(0.0000,0.0000) complex(0.0000,0.0000)
complex(0.0000,0.0000) complex(-5.1202,0.0000) complex(0.2106,0.0000)
complex(-12.8281,0.0000)
complex(0.0000,0.0000) complex(0.0000,0.0000) complex(0.0000,0.0000)
complex(0.0000,0.0000) complex(0.0000,0.0000) complex(6.8539,0.0000)
complex(-9.8438,0.0000)

qout*rout = complex(1.0000,0.0000) complex(2.0000,0.0000) complex(10.0000,0.0000)
complex(3.0000,0.0000) complex(7.8000,0.0000) complex(8.9000,0.0000)
complex(10.0000,0.0000)
complex(4.0000,0.0000) complex(5.0000,0.0000) complex(11.0000,0.0000)
complex(6.0000,0.0000) complex(1.2000,0.0000) complex(2.3000,0.0000)
complex(3.0000,0.0000)
complex(7.0000,0.0000) complex(8.0000,0.0000) complex(12.0000,0.0000)
complex(9.0000,0.0000) complex(4.5000,0.0000) complex(5.6000,0.0000)
complex(6.0000,0.0000)
complex(11.0000,0.0000) complex(15.0000,0.0000) complex(13.0000,0.0000)
complex(2.5000,0.0000) complex(5.5000,0.0000) complex(8.5000,0.0000)
complex(0.1000,0.0000)
complex(13.0000,0.0000) complex(10.0000,0.0000) complex(14.0000,0.0000)
complex(7.5000,0.0000) complex(4.5000,0.0000) complex(1.5000,0.0000)
complex(10.0000,0.0000)

```

```
complex(3.5000,0.0000) complex(6.5000,0.0000) complex(15.0000,0.0000)
complex(9.5000,0.0000) complex(10.5000,0.0000) complex(0.2000,0.0000)
complex(31.0000,0.0000)
```

Example2

The same as Example1, Except that all matrices contain numbers.

```
#include <numeric.h>
#include <stdio.h>
int main() {
    array double a[6][6] = {1, 2, 3, 7.8, 8.9, 10,
                           4, 5, 6, 1.2, 2.3, 3,
                           7, 8, 9, 4.5, 5.6, 6,
                           11, 15, 2.5, 5.5, 8.5, .1,
                           13, 10, 7.5, 4.5, 1.5, 10,
                           3.5, 6.5, 9.5, 10.5, .2, 31};

    array double q[6][6], r[6][6];
    array double qout[6][6], rout[6][7];
    array double x[6] = {10, 11, 12, 13, 14, 15};

    qrdecomp(a, q, r);
    qrinsert(qout, rout, q, r, 2, x);
    printf("qout = %f\n\rout = %f\n", qout, rout);
    printf("qout*rout = %.4f\n", qout*rout);

    return 0;
}
```

Output2

```
qout = -0.052111 -0.156528 0.570518 -0.651416 -0.176603 0.437905
-0.208444 -0.118631 0.344933 -0.020393 0.887315 -0.189091
-0.364776 -0.080733 0.119349 0.610631 0.017715 0.687745
-0.573220 -0.537685 -0.540132 -0.300849 -0.004805 -0.006178
-0.677442 0.671704 0.111661 -0.103480 -0.158569 -0.203877
-0.182388 -0.463268 0.486904 0.318045 -0.394971 -0.507826

rout = -19.189841 -20.622891 -26.863172 -12.936532 -10.414365 -8.910965 -15.820871

0.000000 -5.911545 -8.374077 -2.615442 -6.525484 -5.773472 -10.103602
0.000000 0.000000 12.776730 8.968006 8.045269 2.213064 23.612683
0.000000 0.000000 0.000000 4.912276 -1.138537 -5.073806 5.882958
0.000000 0.000000 0.000000 0.000000 -5.120200 0.210568 -12.828086
0.000000 0.000000 0.000000 0.000000 0.000000 6.853920 -9.843752

qout*rout = 1.0000 2.0000 10.0000 3.0000 7.8000 8.9000 10.0000
4.0000 5.0000 11.0000 6.0000 1.2000 2.3000 3.0000
7.0000 8.0000 12.0000 9.0000 4.5000 5.6000 6.0000
11.0000 15.0000 13.0000 2.5000 5.5000 8.5000 0.1000
13.0000 10.0000 14.0000 7.5000 4.5000 1.5000 10.0000
3.5000 6.5000 15.0000 9.5000 10.5000 0.2000 31.0000
```

See Also

qrdelete(), qrdecomp().

References

E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

quadratic

Synopsis

```
#include <numeric.h>
```

```
int quadratic(double a, double b, double c, double & x1, double & x2);
```

Purpose

Solve for the solutions of the quadratic equation $ax^2 + bx + c = 0$.

Return Value

This function returns the number of solutions for the quadratic equation $ax^2 + bx + c = 0$.

Parameters

a The coefficient *a* of $ax^2 + bx + c = 0$.

b The coefficient *b* of $ax^2 + bx + c = 0$.

c The coefficient *c* of $ax^2 + bx + c = 0$.

x1 The first solution of the quadratic equation $ax^2 + bx + c = 0$.

x2 The second solution of the quadratic equation $ax^2 + bx + c = 0$.

Description

This function calculates the solutions of the quadratic equation $ax^2 + bx + c = 0$. The solutions are passed back to the calling function through the fourth and fifth arguments of the function by reference. If the discriminant $b^2 - 4ac$ is larger than 0, the equation has two solutions and the function returns 2. The solutions are sorted in ascending order with *x2* is larger than *x1*. If the discriminant is zero, there is one solution and the function returns 1. If the discriminant is less than 0, there is no solution and the function returns 0. In this case, *x1* and *x2* are NaN.

Example

```
/* File: quadratic.ch */
#include <numeric.h>
int main() {
    /* y = x*x -5x +6 */
    double a=1, b=-5, c = 6;
    double x1, x2;

    quadratic(a, b, c, x1, x2);
    printf("Solutions for x*x-5x+6=0 are %.2f and %.2f\n", x1, x2);

    c = 7.25;
    quadratic(a, b, c, x1, x2);
    printf("Solutions for x*x-5x+7.25=0 are %.2f and %.2f\n", x1, x2);
}
```

Output

Solutions for $x*x-5x+6=0$ are 3.00 and 2.00
Solutions for $x*x-5x+7.25=0$ are nan and nan

See Also

cquadratic(), **vertex()**.

quadraticfit

Synopsis

```
#include <numeric.h>
```

```
int quadraticfit(double x[], double y[], double &a, double &b); double &a, double r[];
```

Purpose

Fit a set of data points x, y to a quadratic function $y = ax^2 + bx + c$.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x An array which contains variable values of the data set.

y An array which contains function values of the data set.

a The coefficient a for $y = ax^2 + bx + c$.

b The coefficient b for $y = ax^2 + bx + c$.

c The constant c for $y = ax^2 + bx + c$.

r The residual error for the quadratic fitting.

Description

Given a set of data points expressed in arrays x and y with the same dimension, fit the data set using the polynomial fit. The fitted coefficients are passed by arguments a, b , and c .

Algorithm

Use the polynomial fit algorithm to find a, b , and c for $y = ax^2 + bx + c$.

Example 1

```

/* File: quadraticfit.ch
   Find the quadratic equation for a trajectory of soccer ball
   kicked from a platform 0.4 meter above the ground. */
#include <chplot.h>
CPlot plot;
array double t[7] = {0, 0.5, 1, 1.5, 2, 2.5, 3};
array double y[7] = {0.4, 7, 10, 12, 10.3, 7.8, 1};
array double r[7];
double a, b, c, t1, y1;

/* solve for a, b, and c in y = a*t*t+b*t+c */
quadraticfit(t, y, a, b, c, r);
printf("a = %.2lf\n", a);
printf("b = %.2lf\n", b);
printf("c = %.2lf\n", c);
printf("r = %.2lf\n", r);

```

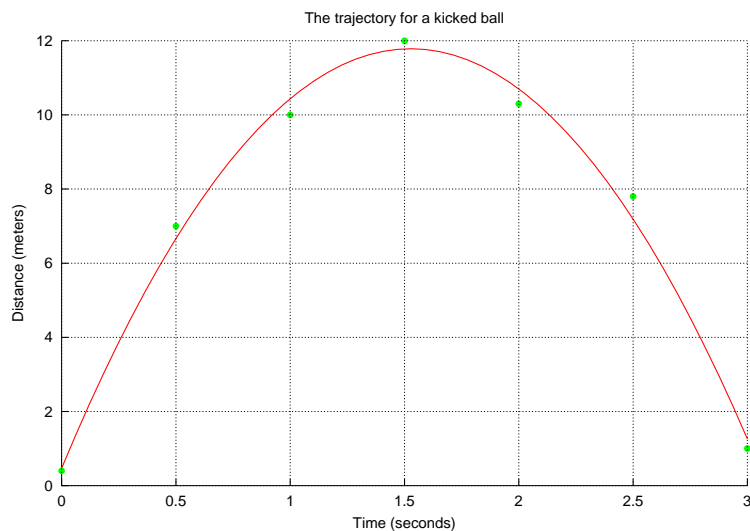
```
/* calculate y for t=1.75 */
t1 = 1.75;
y1 = a*t1*t1 + b*t1 + c;
printf("(t1, y1) = (%.21f, %.21f)\n", t1, y1);

plot.title("The trajectory for a kicked ball");
plot.label(PLOT_AXIS_X, "Time (seconds)");
plot.label(PLOT_AXIS_Y, "Distance (meters)");
plot.axisRange(PLOT_AXIS_X, 0, 3);
//plot.axisRange(PLOT_AXIS_Y, 0, 12);
plot.expr(0, 3, 50, "a*x*x + b*x + c");
plot.scatter(t, y);
plot.plotting();
```

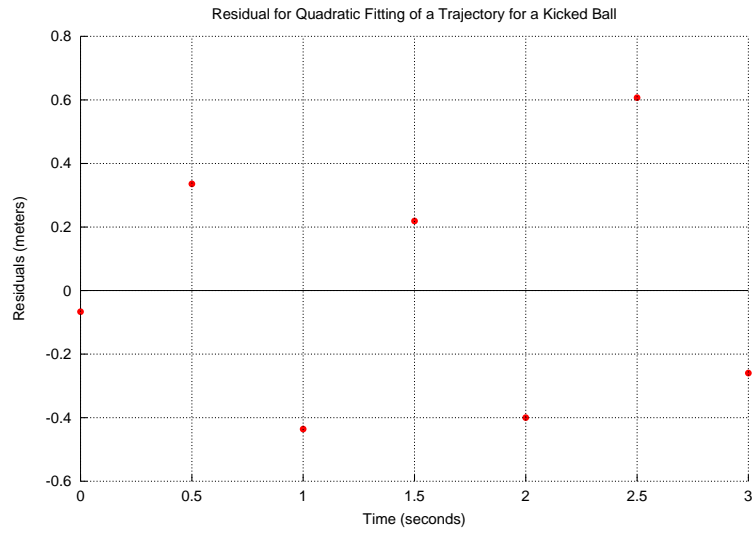
Output

```
a = -4.85
b = 14.82
c = 0.47
r = -0.07 0.34 -0.44 0.22 -0.40 0.61 -0.26
```

```
(t1, y1) = (1.75, 11.54)
```



Example 2



Output

See Also

curvefit(), expfit(), linearfit(), polyfit().

quartiles

Synopsis

```
#include <numeric.h>
```

```
int quartiles(array double x[], double &min, double &q1, double &med, double &q3, double &max,
double &range, double &iqr);
```

Purpose

Calculate the seven key statistical values for a set of data.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x An array which contains variable values of the data set.

min The minimum value of the data set.

q1 The median of the lower half, the lower quartile.

median The median of the data set.

q1 The median of the upper half, the upper quartile.

max The maximum value of the data set.

range The range of the data set. The difference between the maximum and minimum values, range = max-min.

iqr The interquartile range (IQR), the difference between the upper quartile and the lower quartile, iqr = q3-q1.

Description

Given a set of data passed in array *x*, calculate its seven key statistical values and passed them back the calling function.

Algorithm

The function will sort the data first, then calculate the seven key statistical values.

Example

```
/* File: quartiles.ch */
#include <numeric.h>

double x[] = {53, 78, 11, 94, 16, 9, 42, 91, 37, 11, 76, 99};
double min, q1, med, q3, max, range, iqr;

quartiles(x, min, q1, med, q3, max, range, iqr);
range = max - min;
printf("          min = %.21f\n", min);
printf("          lower quartile = %.21f\n", q1);
```

```
printf("middle quartile (median) = %.2lf\n", med);
printf("      upper quartile = %.2lf\n", q3);
printf("      max = %.2lf\n", max);
printf("      range = %.2lf\n", range);
printf("      interquartile range = %.2lf\n", iqr);
```

Output

```
      min = 9.00
      lower quartile = 13.50
middle quartile (median) = 47.50
      upper quartile = 84.50
      max = 99.00
      range = 90.00
interquartile range = 71.00
```

See Also

max(), mean(), min(), sort(), CPlot::boxAndWhisker().

randcolor

Synopsis

```
#include <numeric.h>
string_t randcolor();
```

Purpose

Generate a random color in a hexadecimal number.

Return Value

This function returns a random color in a hexadecimal number.

Parameters

no argument.

Description

This function generates a random integer between and including a and b.

Example

```
#include <numeric.h>
string_t color;

color = randcolor();
printf("a random color is %s\n", color);
color = randcolor();
printf("a random color is %s\n", color);
```

Output1

```
a random color is #8adb6d
a random color is #b37c7b
```

See Also

rand(), **randint()**, **rgb2hex()**, **srand()**.

randdouble

Synopsis

```
#include <numeric.h>
```

```
double randdouble(double a, double b);
```

Purpose

Generate a random floating-point number of double type between and including a and b.

Return Value

This function returns a random floating-point number of double type.

Parameters

a the minimum value of the generated floating-point number.

b the maximum value of the generated floating-point number.

Description

This function generates a random floating-point number between and including a and b.

Example1

```
#include <numeric.h>
int main() {
    double x, a=-1, b = 1;

    x = randdouble(a, b);
    printf("x = %lf\n", x);
    return 0;
}
```

See Also

rand(), **randint()**, **srand()**.

randint

Synopsis

```
#include <numeric.h>
int randint(int a, int b);
```

Purpose

Generate a random integer between and including a and b.

Return Value

This function returns a random integer.

Parameters

a the minimum value of the generated integer.

b the maximum value of the generated integer.

Description

This function generates a random integer between and including a and b.

Example1

```
#include <numeric.h>
int main() {
    int n, a=1, b = 6;

    n = randint(a, b);
    printf("n = %d\n", n);
    return 0;
}
```

See Also

rand(), **randdouble()**, **srand()**.

randintfreq

Synopsis

```
#include <numeric.h>
```

```
array int randintfreq(int n, int num)[:];
```

Purpose

Obtain the frequency for numbers between 1 and n when the num random integers between 1 and n are generated.

Return Value

This function returns an array with the frequency for random numbers.

Parameters

n the maximum value of the generated numbers between 1 and n.

num the number of generated random numbers between 1 and n.

Description

This function obtains the frequency for numbers between 1 and n when the num random integers between 1 and n are generated.

Example1

```

/* File: randintfreq.ch
   Roll a six-sided die 6,000 times randomly,
   calculate the frequencies for each face.
   Use an array to keep frequency for each face */

int n;           /* the number of faces of a die */
int num;        /* the number of times rolling a die */
n = 6;
num = 6000;     /* roll the die 6000 times */
array int freq[n]; /* array of frequencies for each face */

/* roll a die num times */
freq = randintfreq(n, num);

/* display the result */
printf("Face      1      2      3      4      5      6\n");
printf("Frequency  %6d", freq);
printf("Probability %6.2f", freq*100.0/num);

```

Output

Face	1	2	3	4	5	6
Frequency	957	1001	1002	1026	980	1034
Probability	15.95	16.68	16.70	17.10	16.33	17.23

See Also

[rand\(\)](#), [randdouble\(\)](#), [randint\(\)](#), [srand\(\)](#).

rank

Synopsis

```
#include <numeric.h>
```

```
int rank(array double complex a[&][&]);
```

Purpose

Find the rank of a matrix.

Return Value

This function returns the rank of a matrix.

Parameters

a Input two-dimensional array.

Description

This function provides an estimate of the number of linearly independent rows or columns of a matrix. The function can handle complex matrices.

Algorithm

The method used to compute the rank of a matrix in this function is based on the singular value decomposition by function `svd()`. The number of non-zero singular values with tolerance $tol = \max(m, n) \times \max(S) \times \text{DBL_EPSILON}$.

Example1

```
#include <numeric.h>
int main() {
    array double a[2][2] = {2, 4,
                          3, 7};
    /* a2 is an ill-condition matrix */
    array double a2[2][2] = {2, 4,
                          2.001, 4.0001};
    /* singular matrix */
    array float b[3][3] = {2, 4, 4,
                          3, 7, 3,
                          3, 7, 3};

    array double complex z[2][2] = {2, complex(4, 3),
                                    3, 7};
    /* a2 is an ill-condition matrix */
    array double complex z2[2][2] = {2, complex(4, 3),
                                    2.001, complex(4.0001, 3)};
    array complex z3[2][2] = {2, complex(4, 3),
                              3, 7};

    int r;

    r = rank(a);
    printf("rank(a) = %d\n", r);

    r = rank(a2);
    printf("rank(a2) = %d\n", r);
}
```



```

r = rank(b);
printf("rank(b) = %d\n", r);

r = rank(z);
printf("rank(z) = %d\n", r);
r = rank(z2);
printf("rank(z2) = %d\n", r);
r = rank(z3);
printf("rank(z3) = %d\n", r);
}

```

Output1

```

rank(a) = 2
rank(a2) = 2
rank(b) = 2
rank(z) = 2
rank(z2) = 2
rank(z3) = 2

```

Example2

```

#include <numeric.h>
int main() {
    /* singular matrix */
    array float b[3][3] = {1, 2, 3,
                          4, 5, 6,
                          7, 8, 9};

    int r;

    r = rank(b);
    printf("rank(b) = %d\n", r);
    printf("determinant(b) = %f\n", determinant(b));
}

```

Output2

```

rank(b) = 2
determinant(b) = 0.000000

```

See Also

svd().

References

E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

rcondnum

Synopsis

```
#include <numeric.h>
```

```
double rcondnum(array double complex a[&][&]);
```

Purpose

Estimate reciprocal of the condition number of a matrix.

Return Value

This function returns the reciprocal of the condition number of a matrix.

Parameters

a Input matrix.

Description

This function calculates an estimate for the reciprocal of the condition of matrix *a* in 1-norm. If *a* is well conditioned, **rcondnum**(*a*) is near 1.0. If *a* is badly conditioned, **rcondnum**(*a*) is near 0.0. Compared to **condnum**(), function **rcondnum**() is a more efficient, but less reliable, method of estimating the condition of matrix.

Algorithm

The reciprocal of the condition number of a general real/complex matrix *A* is estimated in either the 1-norm or the infinity-norm, using the LU decomposition.

Example

```
#include <numeric.h>
int main() {
    array double a[2][2] = {2, 4,
                           3, 7};
    /* a2 is an ill-condition matrix */
    array double a2[3][3] = {2, 4, 5,
                            2.001, 4.0001, 5.001,
                            6, 8, 3};
    array double complex z[2][2] = {2, 4,
                                    3, 7};
    array double complex z2[2][2] = {complex(2, 3), complex(4,2),
                                     3, 7};

    double rcond;

    rcond = rcondnum(a);
    printf("rcondnum(a) = %f\n", rcond);
    rcond = rcondnum(a2);
    printf("rcondnum(a2) = %f\n", rcond);

    rcond = rcondnum(z);
    printf("rcondnum(z) = %f\n", rcond);
    rcond = rcondnum(z2);
    printf("rcondnum(z2) = %f\n", rcond);
}
```

Output

```
rcondnum(a) = 0.018182  
rcondnum(a2) = 0.000035  
rcondnum(z) = 0.018182  
rcondnum(z2) = 0.131909
```

See Also

condnum(), **norm()**, **rank()**, **svd()**.

References

E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

regularpolygon

Synopsis

```
#include <numeric.h>
```

```
void regularpolygon(array double px[:], array double py[:], double x, double y, int n, double length, double angle);
```

Purpose

Generate two arrays with vertices for a regular polygon.

Return Value

None.

Parameters

px An array for the x-coordinates of vertices of a regular polygon.

py An array for the y-coordinates of vertices of a regular polygon.

x The x coordinate for the center of the regular polygon.

y The y coordinate for the center of the regular polygon.

n The number of sides of the regular polygon.

length The length for each side of the regular polygon.

angle The angle for the regular polygon rotated about the center of the polygon.

Description

The function generates two arrays *px* and *py* with vertices for a regular polygon (all angles are the same and all sides have the same length). The regular polygon has *n* sides with the *length* for each side. The regular polygon is rotated with respect to its center by an *angle* in degrees.

By default, a side of a regular polygon at the bottom is horizontal. The first vertex (*px*[0], *py*[0]) for the regular polygon is the side at the bottom on the right. The subsequent vertices are counted counter-clock wise.

Example

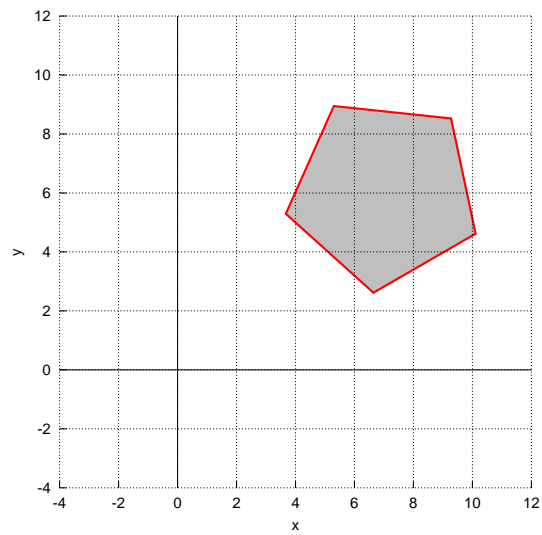
```
/* File: regularpolygon.ch */
#include <chplot.h>
int n = 5; // pentagon
array double px[n], py[n];
double length = 4, angle = 30, x = 7, y = 6;;
CPlot plot;

/* create a regular polygon centered at (7, 6),
   5 sides with side length 4, rotated by 30 degrees */
regularpolygon(px, py, x, y, n, length, angle);
printf("px = %5.2lf", px);
printf("py = %5.2lf", py);
plot.fillColor("grey");
plot.polygon(px, py);
```

```
plot.axisRange(PLOT_AXIS_XY, -4, 12);  
plot.sizeRatio(1);  
plot.plotting();
```

Output

```
px = 10.11  9.28  5.30  3.67  6.64  
py =  4.62  8.53  8.95  5.29  2.62
```

**See Also**

rotatepolygon(), **CPlot::regularPolygon()**.

residue

Synopsis

```
#include <numeric.h>
```

```
int residue(array double u[&], array double v[&], array double complex r[&],
            array double complex p[&], array double k[&]);
```

Syntax

```
residue(u, v, r, p, k)
```

Purpose

Partial-fraction expansion or residue computation.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

u A vector of size *m* which contains the numerator of the ratio of two polynomials.

v A vector of size *n* which contains the denominator of the ratio of two polynomials.

r A vector of size (*n* - 1) which contains the residues of a partial fraction expansion of the ratio of two polynomials.

p A vector of size (*n* - 1) which contains the poles of a partial fraction expansion of the ratio of two polynomials.

k A vector of size (*m* - *n* + 1) which contains the direct term of a partial fraction expansion of the ratio of two polynomials. If *m* ≤ *n*, then *k* contains NULL.

Description

Function **residue()** finds the residues, poles and direct term of a partial fraction expansion of the ratio of two polynomials $V(s)$ and $U(s)$. Given the ratio of two polynomials

$$\frac{U(s)}{V(s)} = \frac{u_0 s^m + u_1 s^{m-1} + \cdots + u_{m-1} s + u_m}{s^n + v_1 s^{n-1} + \cdots + v_{n-1} s + v_n}$$

If there are no multiple roots, the expansion becomes

$$\frac{U(s)}{V(s)} = \frac{r_0}{s - p_0} + \frac{r_1}{s - p_1} + \cdots + \frac{r_{n-1}}{s - p_{n-1}} + K(s)$$

If p_i is a pole of multiplicity l , then the expansion becomes

$$\frac{U(s)}{V(s)} = \frac{r_0}{s - p_0} + \frac{r_1}{s - p_1} + \cdots + \frac{r_i}{(s - p_i)^l} + \cdots + \frac{r_{i+1}}{(s - p_i)^2} + \frac{r_{i+l}}{s - p_i} + \cdots + \frac{r_{n-1}}{s - p_{n-1}} + K(s)$$

where $K(x)$ is the direct term. If $m > n$, $K(s)$ is

$$K(s) = k_0 s^{m-n} + k_1 s^{m-n-1} + \cdots + k_{m-n} s + k_{m-n+1}$$

Otherwise, $K(s)$ is empty. Vectors u and v specify the coefficients of the polynomials in descending powers of s . They can be any real type. Conversion of the data to double type is performed internally. The residues r and poles p are any compatible data type according to the residue computation. If the argument of real type is passed and the result is complex type, the value of NaN will be passed out. The direct term k is always real type.

Algorithm

Given the ratio of two polynomials $U(x)$ and $V(x)$

$$\frac{U(s)}{V(s)} = \frac{u_0 s^m + u_1 s^{m-1} + \cdots + u_{m-1} s + u_m}{s^n + v_1 s^{n-1} + \cdots + v_{n-1} s + v_n}$$

If there are no multiple roots, it can be written as

$$\frac{U(s)}{V(s)} = \frac{U(s)}{(s - p_0)(s - p_1) \cdots (s - p_{n-1})}$$

where p_i can be real or complex number. Then it becomes

$$\frac{U(s)}{V(s)} = \frac{r_0}{s - p_0} + \frac{r_1}{s - p_1} + \cdots + \frac{r_{n-1}}{s - p_{n-1}}$$

where

$$r_i = \left[(s - p_i) \frac{U(s)}{V(s)} \right]_{s=p_i}$$

If l of the $(n - 1)$ poles are identical, or say, the root at $s = p_i$ is of multiplicity l , then

$$\frac{U(s)}{V(s)} = \frac{U(s)}{(s - p_0)(s - p_1) \cdots (s - p_i)^l \cdots (s - p_i)^2 (s - p_i) \cdots (s - p_{n-l})}$$

The equation can be expanded as

$$\frac{U(s)}{V(s)} = \frac{r_0}{s - p_0} + \frac{r_1}{s - p_1} + \cdots + \frac{r_i}{(s - p_i)^l} + \cdots + \frac{r_{i+1}}{(s - p_i)^2} + \frac{r_{i+l}}{s - p_i} + \cdots + \frac{r_{n-1}}{s - p_{n-l}}$$

where the coefficients correspond to simple poles could be calculated by the method described above. The determination of the coefficients that correspond to the multiple-order is given below.

$$r_{i+k} = \frac{1}{(k - 1)!} \frac{d^{k-1}}{ds^{k-1}} \left[(s - p_i)^l \frac{U(s)}{V(s)} \right]_{s=p_i} \quad k = 1, 2, \dots, l$$

Example 1

Partial-fraction expansion with single real roots and without direct term.

$$\frac{10s + 6}{2s^3 + 12s^2 + 22s + 12} = \frac{-1}{s + 1} + \frac{7}{s + 2} + \frac{-6}{s + 3}$$

```
#include <stdio.h>
#include <numeric.h>

#define M 2      /* data array size */
#define N 4      /* response array size */
```

```

int main() {
    array double u[M] = {10, 6};
    array double v[N] = {2, 12, 22, 12};
    array double r[N-1],p[N-1];
    array double k[1];

    residue(u,v,r,p,k);
    printf("coef. of u=%f\n",u);
    printf("coef. of v=%f\n",v);
    printf("coef. of r=%f\n",r);
    printf("coef. of p=%f\n",p);
    printf("coef. of k=%f\n",k);
}

```

Output

```

coef. of u=10.000000 6.000000

coef. of v=2.000000 12.000000 22.000000 12.000000

coef. of r=-1.000000 7.000000 -6.000000

coef. of p=-1.000000 -2.000000 -3.000000

coef. of k=0.000000

```

Example 2

A partial-fraction expansion with single roots of complex numbers and without direct term.

$$\frac{s+3}{s^2+2s+5} = \frac{0.5+i0.5}{s+(1+i2)} + \frac{0.5-i0.5}{s+(1-i2)}$$

```

#include <stdio.h>
#include <numeric.h>

#define M 2          /* data array size */
#define N 3          /* response array size */

int main() {
    array double u[M] = {1, 3};
    array double v[N] = {1, 2, 5};
    array double complex r[N-1],p[N-1];
    array double r1[N-1],p1[N-1];
    array double k[1];

    residue(u,v,r,p,k);
    printf("coef. of u=%f\n",u);
    printf("coef. of v=%f\n",v);
    printf("coef. of r=%f\n",r);
    printf("coef. of p=%f\n",p);
    printf("coef. of k=%f\n",k);

    printf("\n r1, p1 pass by real data, but the result are complex data.\n");
    residue(u,v,r1,p1,k);
    printf("coef. of r=%f\n",r1);
    printf("coef. of p=%f\n",p1);
}

```


Output

```
coef. of u=1.000000 3.000000
coef. of v=1.000000 2.000000 5.000000
coef. of r=complex(0.500000,0.500000) complex(0.500000,-0.500000)
coef. of p=complex(-1.000000,-2.000000) complex(-1.000000,2.000000)
coef. of k=0.000000

r1, p1 pass by real data, but the result are complex data.
coef. of r=NaN NaN
coef. of p=NaN NaN
```

Example 3

A partial-fraction expansion with single roots and has a direct term.

$$\frac{2s^3 + 12s^2 + 22s + 12}{10s + 6} = \frac{0.2688}{s + 0.6} + 0.2s^2 + 1.08s + 1.552$$

```
#include <stdio.h>
#include <numeric.h>

#define N 2          /* data array size */
#define M 4          /* response array size */

int main() {
    array double v[N] = {10, 6};
    array double u[M] = {2, 12, 22, 12};
    array double r[N-1],p[N-1];
    array double k[M-N+1];

    residue(u,v,r,p,k);
    printf("coef. of u=%f\n",u);
    printf("coef. of v=%f\n",v);
    printf("coef. of r=%f\n",r);
    printf("coef. of p=%f\n",p);
    printf("coef. of k=%f\n",k);
}
```

Output

```
coef. of u=2.000000 12.000000 22.000000 12.000000
coef. of v=10.000000 6.000000
coef. of r=0.268800
coef. of p=-0.600000
coef. of k=0.200000 1.080000 1.552000
```

Example 4

A partial-fraction expansion with multiplicity roots and without direct term.

$$\frac{s^2 + 2s + 3}{s^5 + 5s^4 + 9s^3 + 7s^2 + 2s} = \frac{3}{2(s+2)} - \frac{3}{(s+1)^3} - \frac{2}{s+1} + \frac{3}{2s}$$

```
#include <stdio.h>
#include <numeric.h>

#define M 3          /* data array size */
#define N 6          /* response array size */

int main() {
    array double u[M] = {1,2,3};
    array double v[N] = {1, 5, 9, 7, 2, 0};
    array double r[N-1],p[N-1];
    array double k[1];

    residue(u,v,r,p,k);
    printf("coef. of u=%f\n",u);
    printf("coef. of v=%f\n",v);
    printf("coef. of r=%f\n",r);
    printf("coef. of p=%f\n",p);
    printf("coef. of k=%f\n",k);
}
```

Output

```
coef. of u=1.000000 2.000000 3.000000
coef. of v=1.000000 5.000000 9.000000 7.000000 2.000000 0.000000
coef. of r=1.500000 -3.000000 0.000000 -2.000000 1.500000
coef. of p=-2.000000 -1.000000 -1.000000 -1.000000 0.000000
coef. of k=0.000000
```

References

Benjamin C. Kuo, *Automatic Control System*, fifth edition, Prentice-Hall, 1987, p. 28-32.

rgb2hex

Synopsis

```
#include <numeric.h>
string_t rgb2hex(int r, int g, int b);
```

Purpose

Change a represented in RGB values to the color in a hexadecimal number.

Return Value

This function returns a color in a hexadecimal number.

Parameters

r The red component of a color in the range from 0 to 255.

g The green component of a color in the range from 0 to 255.

b The blue component of a color in the range from 0 to 255.

Description

This function returns a color in a hexadecimal number based on the RGB value of the input arguments. If the value of a color component (*r*, *g*, *b*) is less than 0, the value 0 is used. If the value of the a color component is larger than 255, the value 255 is used.

Example

```
#include <numeric.h>
string_t color;
int r, g, b;

r = 255;
g = 0;
b = 17;
color = rgb2hex(r, g, b);
printf("The color in RGB(%d, %d, %d) is color in hexadecimal number %s\n",
       r, g, b, color);
```

Output

```
The color in RGB(255, 0, 17) is color in hexadecimal number #ff0011
```

See Also

randcolor().

roots

Synopsis

```
#include <numeric.h>
```

```
int roots(array double complex x[&], array double complex p[&]);
```

Purpose

Find the roots of a polynomial.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x An output which contains the roots of the polynomial.

p An input which contains the coefficients of the polynomial.

Description

This function finds the roots of the polynomial. The function can handle the polynomial with complex coefficients. The solutions in the array *x* are sorted in ascending order.

Algorithm

The roots of a polynomial are obtained by computing the eigenvalues of the corresponding companion matrix of the polynomial.

Example

```
#include <numeric.h>
int main() {
    array double x1[3], p1[4] = {1, -6, -72, -27}; /* x^3-6x^2-72x-27 =0*/
    array double x2[4], p2[5] = {1, -12, 0, 25, 116}; /* x^4-12x^3-25x+116 =0 */
    array double complex z2[4];
    array double complex z3[4], p3[5] = {complex(3,4), complex(4,2),
                                        complex(5,3), complex(2,4), complex(1,5)};

    int status, i;

    roots(x1, p1);
    printf("x1 from roots(x1, p1) = %.3f\n", x1);
    for(i=0; i<3; i++)
        printf("polyeval(p1, x1[%d]) = %.3f\n", i, polyeval(p1, x1[i]));

    /* x^4-12x^3-25x+116 =0 has two complex roots and two real roots */
    roots(x2, p2);
    printf("x2 from roots(x2, p2) = %.3f\n", x2);
    roots(z2, p2);
    printf("z2 from roots(z2, p2) = \n%.3f\n", z2);

    status = roots(z3, p3);
    if(status == 0)
        printf("z3 from roots(z3, p3) = \n%.3f\n", z3);
    else
```

```
    printf("roots(z3, p3) failed\n");
    for(i=0; i<4; i++)
        printf("cpolyeval(p3, z3[%d]) = %.3f\n", i, cpolyeval(p3, z3[i]));
}
```

Output

```
x1 from roots(x1, p1) = 12.123 -5.735 -0.388

polyeval(p1, x1[0]) = 0.000
polyeval(p1, x1[1]) = -0.000
polyeval(p1, x1[2]) = 0.000
x2 from roots(x2, p2) = 11.747 2.703 NaN NaN

z2 from roots(z2, p2) =
complex(11.747,0.000) complex(2.703,0.000) complex(-1.225,1.467) complex(-1.225,-1.467)

z3 from roots(z3, p3) =
complex(0.226,1.281) complex(0.431,-0.728) complex(-0.754,-0.708) complex(-0.703,0.554)

cpolyeval(p3, z3[0]) = complex(0.000,-0.000)
cpolyeval(p3, z3[1]) = complex(0.000,-0.000)
cpolyeval(p3, z3[2]) = complex(0.000,-0.000)
cpolyeval(p3, z3[3]) = complex(0.000,-0.000)
```

See Also

fzero(), **residue()**.

References

rot90()

Synopsis

```
#include <numeric.h>
```

```
int rot90(array double complex y[&][&], array double complex x[&][&], ... /* [int k] */);
```

Syntax

```
rot90(y, x)
```

Purpose

Rotate matrix 90 degrees.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x Two-dimensional matrix which contains the original data.

y Two-dimensional matrix of the same data type and size correspondence with the size of rotation of matrix *x*. It contains the rotation of input matrix *x*.

k Integral data which indicates the $k * 90$ degrees rotation of *x*. For a positive *k*, the matrix is rotated counter clockwise. For a negative *k*, the matrix is rotated clockwise.

Description

This function rotates matrix *x* $k * 90$ degrees in counter clockwise direction.

Example

```
#include <numeric.h>
int main() {
    array double x[4][2]={1,3,
                          2.45,8.56,
                          3,5,
                          6,8};
    array double complex zx[3][2]={complex(1,1),0,
                                   3,complex(4,1),
                                   0,0};
    array double complex zy[3][2];
    array double y[2][4], y1[4][2];

    rot90(y, x);
    printf("x = \n%f", x);
    printf("y = rotate 90 degrees\n%f", y);
    rot90(y1, x, 2);
    printf("y1 = rotate 180 degrees\n%f", y1);
    rot90(y1, x, 8);
    printf("y1 = rotate 720 degrees\n%f", y1);
    rot90(y, x, -1);
    printf("y = rotate -90/270 degrees\n%f", y);
    printf("\n");
}
```

```

    rot90(zy,zx,2);          /* complex data rotation */
    printf("zx = \n%5.1f",zx);
    printf("zy = \n%5.1f",zy);
}

```

Output

```

x =
1.000000 3.000000
2.450000 8.560000
3.000000 5.000000
6.000000 8.000000
y = rotate 90 degrees
3.000000 8.560000 5.000000 8.000000
1.000000 2.450000 3.000000 6.000000
y1 = rotate 180 degrees
8.000000 6.000000
5.000000 3.000000
8.560000 2.450000
3.000000 1.000000
y1 = rotate 720 degrees
1.000000 3.000000
2.450000 8.560000
3.000000 5.000000
6.000000 8.000000
y = totate -90/270 degrees
6.000000 3.000000 2.450000 1.000000
8.000000 5.000000 8.560000 3.000000

zx =
complex( 1.0, 1.0) complex( 0.0, 0.0)
complex( 3.0, 0.0) complex( 4.0, 1.0)
complex( 0.0, 0.0) complex( 0.0, 0.0)
zy =
complex( 0.0, 0.0) complex( 0.0, 0.0)
complex( 4.0, 1.0) complex( 3.0, 0.0)
complex( 0.0, 0.0) complex( 1.0, 1.0)

```

See Also

flipud(), **fliplr()**.

rotatepoint

Synopsis

```
#include <numeric.h>
```

```
void rotatepoint(double &x2, double &y2, double x1, double y1, double x, double y, double angle);
```

Purpose

Rotate a point with respect to another point by an angle.

Return Value

None.

Parameters

x2 The x-coordinate of the resultant point.

y2 The y-coordinate of the resultant point.

x1 The x-coordinate of the original point.

y1 The y-coordinate of the original point.

x The x coordinate for the point (x, y) that the point (x1, y1) rotates about.

y The y coordinate for the point (x, y) that the point (x1, y1) rotates about.

angle The rotation angle that the point rotates.

Description

The function rotates the point (x1, y1) by an *angle* with respect to the point (x, y). The resultant point is passed by reference through the arguments *x2* and *y2*.

Example

```
/* File: rotatepoint.ch */
double x2, y2;
double x1 = 3, y1 = 4;
double x = 1, y = 2, angle = 30;

rotatepoint(x2, y2, x1, y1, x, y, angle);
printf("x1 = %lf, y1 = %lf\n", x1, y1);
printf("x2 = %lf, y2 = %lf\n", x2, y2);
```

Output

```
x1 = 3.000000, y1 = 4.000000
x2 = 1.732051, y2 = 4.732051
```

See Also

[regularpolygon\(\)](#), [rotatepolygon\(\)](#), [CPlot::polygon\(\)](#), [CPlot::regularPolygon\(\)](#).

rotatepolygon

Synopsis

```
#include <numeric.h>
```

```
void rotatepolygon(array double px2[:], array double py2[:], array double px1[:], array double py1[:],
double x, double y, double angle);
```

Purpose

Rotate two arrays with vertices for a general polygon by an angle.

Return Value

None.

Parameters

px2 An array for the x-coordinates of vertices of the result polygon.

py2 An array for the y-coordinates of vertices of the result polygon.

px1 An array for the x-coordinates of vertices of the original polygon.

py1 An array for the y-coordinates of vertices of the original polygon.

x The x coordinate for the point (x, y) that the polygon rotates about.

y The y coordinate for the point (x, y) that the polygon rotates about.

angle The rotation angle that the polygon rotates.

Description

The function rotates two arrays *px1* and *py1* with vertices for a polygon by an *angle* with respect to the point (x, y). The vertices of the resultant polygon is stored in two arrays *px2* and *py2*.

Example

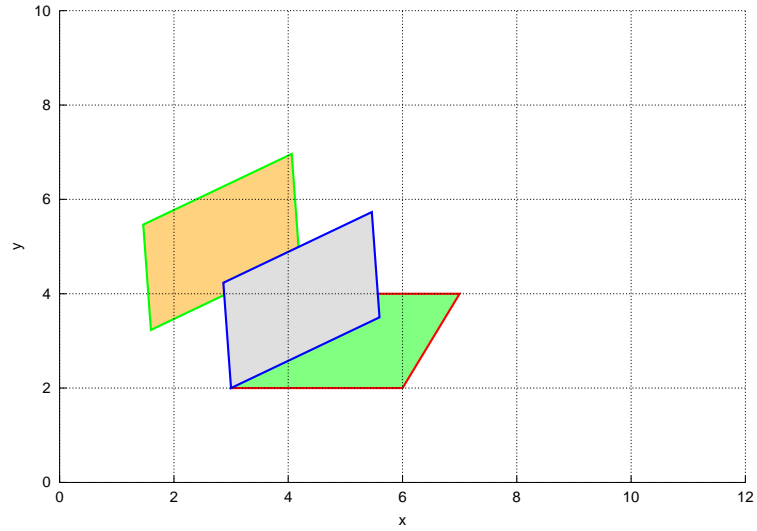
```
/* File: rotatepolygon.ch */
#include <chplot.h>
array double px[] = {3, 6, 7, 4}, px2[4], px3[4];
array double py[] = {2, 2, 4, 4}, py2[4], py3[4];
CPlot plot;

plot.fillOpacity(0.5);
plot.fillColor("green");
plot.polygon(px, py);

/* rotate the polygon about (0, 0) by 30 degrees */
rotatepolygon(px2, py2, px, py, 0, 0, 30);
plot.fillColor("orange");
plot.polygon(px2, py2);

/* rotate the polygon about (2, 3) by 30 degrees */
rotatepolygon(px2, py2, px, py, 3, 2, 30);
plot.fillColor("grey");
plot.polygon(px2, py2);
```

```
plot.axisRange(PLOT_AXIS_X, 0, 12);  
plot.axisRange(PLOT_AXIS_Y, 0, 10);  
plot.plotting();
```

**Output****See Also**

regularpolygon(), **rotatepoint()**, **CPlot::polygon()**, **CPlot::regularPolygon()**.

rotmatrix

Synopsis

```
#include <numeric.h>
array double rotmatrix(double angle);
```

Purpose

Calculate the two-dimensional rotation matrix.

Return Value

This function returns a two-dimensional rotation matrix.

Parameters

angle The angle for the rotation.

Description

The function turns a two-dimensional rotation matrix based on the rotation angle *angle*.

Algorithm

For the rotation angle θ , a two-dimensional matrix R can be calculated by

$$R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

Example

```
/* File: rotmatrix.ch */
array double x1[2] = {3, 4}, x2[2];

printf("x1 = %lf", x1);
printf("x2 = %lf", rotmatrix(90)*x1);
```

Output

```
x1 = 3.000000 4.000000
x2 = -4.000000 3.000000
```

See Also

[rotatepolygon\(\)](#).

rsf2csf

Synopsis

```
#include <numeric.h>
```

```
int rsf2csf(array double complex uc[:][:], array double complex tc[:][:],
            array double complex ur[&][&], array double complex tr[&][&]);
```

Purpose

Convert a real upper quasi-triangular Schur form to a complex upper triangular Schur form.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

uc Output complex unitary matrix.

tc Output complex quasi-triangular Schur matrix.

ur Input real unitary matrix.

tr Input real upper triangular Schur matrix.

Description

This function converts a real upper quasi-triangular Schur form, which is the result of Schur decomposition, to a complex upper triangular Schur form. Assume that (u_r, t_r) is the real Schur form of the real square matrix x . This function can produce the complex Schur form (u_c, t_c) from the real one (u_r, t_r) , such that $x = u_c t_c u_c'$. The matrix t_c is upper triangular with the eigenvalues of x on the diagonal.

Example

Transform the output of function `schurdecomp()` from real Schur form (ur, tr) to complex Schur form (uc, tc) .

```
#include <array.h>

int main() {
    array double x1[4][4] = {1,1,1,3, 1,2,1,1, 1,1,3,1, -2,1,1,4};
    array double ur1[4][4], tr1[4][4];
    array double complex uc1[4][4], tc1[4][4];

    array double x2[3][3] = {2, 3, 4, 5, 6, 7, 8, 9, 10};
    array double ur2[3][3], tr2[3][3];
    array double complex uc2[3][3], tc2[3][3];

    schurdecomp(x1, ur1, tr1);
    printf("ur1 = %.4f\ntr1 = %.4f\n", ur1, tr1);

    rsf2csf(uc1, tc1, ur1, tr1);
    printf("uc1= %.4f\ntc1 = %.4f\n", uc1, tc1);
    printf("norm(uc1) = %.4f\n\n", norm(uc1, "2"));
    printf("uc1*tc1*uc1' = %.4f\n\n", uc1*tc1*transpose(conj(uc1)));

    schurdecomp(x2, ur2, tr2);
    printf("ur2 = %.4f\ntr2 = %.4f\n", ur2, tr2);
```

```

    rsf2csf(uc2, tc2, ur2, tr2);
    printf("uc2 = %.4f\ntc2 = %.4f\n", uc2, tc2);
    printf("norm(uc2) = %.4f\n\n", norm(uc2, "2"));
    printf("uc2*tc2*uc2' = %.4f\n\n", uc2*tc2*transpose(conj(uc2)));

    return 0;
}

```

Output

```

ur1 = 0.8835 0.3186 -0.0197 -0.3428
0.4458 -0.3651 0.1666 0.8001
0.0480 -0.5469 0.7191 -0.4260
-0.1355 0.6827 0.6743 0.2466

tr1 = 1.9202 0.9542 2.9655 0.8572
-2.2777 1.9202 1.1416 1.5630
0.0000 0.0000 4.8121 1.1314
0.0000 0.0000 0.0000 1.3474

uc1= complex(-0.2675,0.4801) complex(0.7417,-0.1731) complex(-0.0197,0.0000)
complex(-0.3428,0.0000)
complex(0.3065,0.2422) complex(0.3743,0.1984) complex(0.1666,0.0000)
complex(0.8001,0.0000)
complex(0.4591,0.0261) complex(0.0403,0.2971) complex(0.7191,0.0000)
complex(-0.4260,0.0000)
complex(-0.5732,-0.0736) complex(-0.1138,-0.3710) complex(0.6743,0.0000)
complex(0.2466,0.0000)

tc1 = complex(1.9202,1.4742) complex(1.3236,0.0000) complex(-0.9583,-1.6113)
complex(-1.3122,-0.4658)
complex(0.0000,0.0000) complex(1.9202,-1.4742) complex(2.4895,0.6203)
complex(0.7196,0.8493)
complex(0.0000,0.0000) complex(0.0000,0.0000) complex(4.8121,0.0000)
complex(1.1314,0.0000)
complex(0.0000,0.0000) complex(0.0000,0.0000) complex(0.0000,0.0000)
complex(1.3474,0.0000)

norm(uc1) = 1.0000

uc1*tc1*uc1' = complex(1.0000,-0.0000) complex(1.0000,-0.0000) complex(1.0000,-0.0000)
complex(3.0000,0.0000)
complex(1.0000,0.0000) complex(2.0000,-0.0000) complex(1.0000,0.0000)
complex(1.0000,0.0000)
complex(1.0000,0.0000) complex(1.0000,-0.0000) complex(3.0000,-0.0000)
complex(1.0000,-0.0000)
complex(-2.0000,-0.0000) complex(1.0000,0.0000) complex(1.0000,-0.0000)
complex(4.0000,0.0000)

ur2 = 0.2826 0.8680 0.4082
0.5383 0.2087 -0.8165
0.7939 -0.4505 0.4082

tr2 = 18.9499 4.8990 0.0000
0.0000 -0.9499 -0.0000
0.0000 0.0000 -0.0000

uc2 = complex(0.2826,0.0000) complex(0.8680,0.0000) complex(0.4082,0.0000)

```

```
complex(0.5383,0.0000) complex(0.2087,0.0000) complex(-0.8165,0.0000)
complex(0.7939,0.0000) complex(-0.4505,0.0000) complex(0.4082,0.0000)

tc2 = complex(18.9499,0.0000) complex(4.8990,0.0000) complex(0.0000,0.0000)
complex(0.0000,0.0000) complex(-0.9499,0.0000) complex(-0.0000,0.0000)
complex(0.0000,0.0000) complex(0.0000,0.0000) complex(-0.0000,0.0000)

norm(uc2) = 1.0000

uc2*tc2*uc2' = complex(2.0000,0.0000) complex(3.0000,0.0000) complex(4.0000,0.0000)
complex(5.0000,0.0000) complex(6.0000,0.0000) complex(7.0000,0.0000)
complex(8.0000,0.0000) complex(9.0000,0.0000) complex(10.0000,0.0000)
```

See Also**schurdecomp()**.**References**

E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

schurdecomp

Synopsis**#include** <numeric.h>

```
int schurdecomp(array double complex a[&][&], array double complex q[&][&],
                array double complex t[&][&]);
```

Purpose

Compute Schur decomposition of a matrix.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

a An input two-dimensional array.

q An output two-dimensional array which contains *q* matrix.

t An output two-dimensional array which contains *t* matrix.

Description

This function computes Schur form of a square matrix. The Schur matrix **T** for a square matrix **A** of real type is defined as

$$A = QTQ^T$$

where matrix **Q** is an orthogonal matrix with $Q^T Q = \mathbf{I}$. For matrix **A** of complex type, the Schur matrix *T* is defined as

$$A = QTQ^H$$

where matrix **Q** is unitary with $Q^H Q = \mathbf{I}$.

Example1

Compute Schur decomposition of a real matrix.

```
#include <numeric.h>
int main() {
    int n = 2;
    array double a[2][2] = {8, -3,
                           -5, 9};
    array double t[n][n];
    array double q[n][n];
    int status;

    status = schurdecomp(a, q, t);
    if(status == 0) {
        printf("Schur vectors q =\n%f\n", q);
        printf("Schur form t =\n%f\n", t);
        printf("q*t*q^T =\n%f\n", q*t*transpose(q));
        printf("q^T*q =\n%f\n", transpose(q)*q);
    }
    else
        printf("error: numerical error in schurdecomp()\n");
    schurdecomp(a, q, NULL);
    printf("Schur vectors q =\n%f\n", q);
}
```

Output1

```
Schur vectors q =
0.661061 -0.750332
0.750332 0.661061
```

```
Schur form t =
4.594875 2.000000
0.000000 12.405125
```

```
q*t*q^T =
8.000000 -3.000000
-5.000000 9.000000
```

```
q^T*q =
1.000000 0.000000
0.000000 1.000000
```

```
Schur vectors q =
0.661061 -0.750332
0.750332 0.661061
```

Example2

Compute Schur decomposition of complex matrix.

```
#include <numeric.h>
int main() {
    int n = 2;
    array double complex a[2][2] = {8, -3,
                                     -5, 9};
    array double complex z[2][2] = {complex(1,2), -3,
                                     -5, 9};
    array double complex t[n][n];
    array double complex q[n][n];

    schurdecomp(a, q, t);
    printf("Schur vectors q =\n%f\n", q);
    printf("Schur form t =\n%f\n", t);
    printf("q*t*q^H =\n%f\n", q*t*conj(transpose(q)));
    printf("q^H*q =\n%f\n", conj(transpose(q))*q);

    schurdecomp(z, q, t);
    printf("Schur vectors q =\n%f\n", q);
    printf("Schur form t =\n%f\n", t);
    printf("q*t*q^H =\n%f\n", q*t*conj(transpose(q)));
    printf("q^H*q =\n%f\n", conj(transpose(q))*q);
}
```

Output2

```
Schur vectors q =
complex(-0.661061,0.000000) complex(-0.750332,0.000000)
complex(-0.750332,0.000000) complex(0.661061,0.000000)
```

```
Schur form t =
complex(4.594875,0.000000) complex(-2.000000,0.000000)
complex(0.000000,0.000000) complex(12.405125,0.000000)
```



```

q*t*q^H =
complex(8.000000,0.000000) complex(-3.000000,0.000000)
complex(-5.000000,0.000000) complex(9.000000,0.000000)

q^H*q =
complex(1.000000,0.000000) complex(-0.000000,0.000000)
complex(-0.000000,0.000000) complex(1.000000,0.000000)

Schur vectors q =
complex(-0.874263,0.158241) complex(-0.452441,0.076946)
complex(-0.458937,0.000000) complex(0.888419,0.009421)

Schur form t =
complex(-0.524869,1.723999) complex(-1.983020,1.403899)
complex(0.000000,0.000000) complex(10.524869,0.276001)

q*t*q^H =
complex(1.000000,2.000000) complex(-3.000000,0.000000)
complex(-5.000000,0.000000) complex(9.000000,0.000000)

q^H*q =
complex(1.000000,0.000000) complex(0.000000,-0.000000)
complex(0.000000,0.000000) complex(1.000000,0.000000)

```

See Also**eigensystem()**.**References**

E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

sign

Synopsis

```
#include <numeric.h>
```

```
double sign(double x)
```

Purpose

Sign function.

Return Value

This function returns 0, 1, -1 to indicate the sign of real number x ;

Parameters

x The input **double** type real number.

Description

The function returns the 1, -1 and 0 for $x > 0$, $x < 0$ and $x = 0$, respectively.

Example

```
#include <numeric.h>
int main() {
    int i = -10;
    float f = -10;
    double d = 10;
    double d2 = 0;
    printf("sign(i) = %d\n", sign(i));
    printf("sign(f) = %d\n", sign(f));
    printf("sign(d) = %d\n", sign(d));
    printf("sign(d2) = %d\n", sign(d2));
}
```

Output

```
sign(i) = -1
sign(f) = -1
sign(d) = 1
sign(d2) = 0
```

See Also

abs(), **real()**, **conj()**.

References

sort()

Synopsis

```
#include <numeric.h>
```

```
int sort(array double complex &y, array double complex &x, ...
```

```
/* [string_t method], [array int &ind] */);
```

Syntax

```
sort(y, x)
```

```
sort(y, x, method)
```

```
sort(y, x, method, ind)
```

Purpose

Sorting and ranking elements in ascending order.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

y An array of any dimension and size which contains the sorted data.

x The same array as *x* which contains the original data.

method Sorting method.

method = "array" - (default) sorted by total elements. That is, if *x* is an array of $n \times m$, sort in total $n \times m$ elements.

method = "row" - sorted by row for two-dimensional array.

method = "column" - sorted by column for two-dimensional array.

ind Index array . Each elements presents the ranking index, starting with 0, corresponding to array *x*.

Description

The original data in array *x* can be of any supported arithmetic data type and dimension. Array *y* is the same data type and size as *x* and it contains sorted data. If *x* is a complex data type, it is sorted by the magnitude of each element. If *x* includes any NaN or ComplexNaN elements, **sort()** places these at the end. The index in array *ind* contains the ranking index corresponding to array *x* and optional argument *method* specifies the sorting method. When *x* is a two-dimensional array, this parameter specifies the sorting method defined as follows: "array" - sorted by total elements; "row" - sorted by row for two-dimensional array; "column" - sorted by column for two-dimensional array. By default, two-dimensional arrays are sorted by total elements. If *x* is not a two-dimensional array, The array is sorted by total elements.

Example

```
#include <numeric.h>

int main() {
    array double x1[4] = {0.1, NaN, -0.1, 3}, y1[4];
```

```

array double complex x2[2][3] = {5, NaN, -3, -6, 3, 5}, y2[2][3];
array int indel[2][3];
array double x4[3][3] = {1, 3, -3, -2, 3, 5, 9, 2, 4}, y4[3][3];
array int inde[3][3];
array double x3[2][3][4], y3[2][3][4];
x3[1][1][2] = 10;
x3[1][2][2] = -10;

sort(y1, x1);          // sort by total elements
printf("sort by total elements\n");
printf("x1 = %5.2f\n", x1);
printf("sort(x1) = %5.2f\n", y1);

sort(y2, x2, "row", indel);
printf("sort by row, gives index array\n");
printf("x2 = %f\n", x2);
printf("sort(x2)\n");
printf("%f\n", y2);
printf("index(x2)\n");
printf("%d\n", indel);

sort(y3, x3, "array");
printf("sort by total elements \n");
printf("x3 = %f\n", x3);
printf("sort(x3)\n");
printf("%f\n", y3);

sort(y4, x4, "column", inde);
printf("sort by column elements, gives index array\n");
printf("x4 = %f\n", x4);
printf("sort(x4)\n");
printf("%f\n", y4);
printf("inde(x4)\n");
printf("%d\n", inde);
}

```

Output

```

sort by total elements
x1 =  0.10   NaN -0.10  3.00

sort(x1) = -0.10  0.10  3.00   NaN

sort by row, gives index array
x2 = complex(5.000000,0.000000) ComplexNaN complex(-3.000000,0.000000)
complex(-6.000000,0.000000) complex(3.000000,0.000000) complex(5.000000,0.000000)

sort(x2)
complex(-3.000000,0.000000) complex(5.000000,0.000000) ComplexNaN
complex(3.000000,0.000000) complex(5.000000,0.000000) complex(-6.000000,0.000000)

index(x2)
2 0 1
1 2 0

sort by total elements
x3 = 0.000000 0.000000 0.000000 0.000000

```

```
0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000

0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 10.000000 0.000000
0.000000 0.000000 -10.000000 0.000000

sort(x3)
-10.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000

0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 10.000000

sort by column elements, gives index array
x4 = 1.000000 3.000000 -3.000000
-2.000000 3.000000 5.000000
9.000000 2.000000 4.000000

sort(x4)
-2.000000 2.000000 -3.000000
1.000000 3.000000 4.000000
9.000000 3.000000 5.000000

inde(x4)
1 2 0
0 0 2
2 1 1
```

References

William H. Press, et al, *Numerical Recipes in C*, second edition, Cambridge University Press, 1997.

specialmatrix()

Synopsis

```
#include <numeric.h>
```

```
array double specialmatrix(string_t name, ... /* [type1 arg1, type2 arg2, ...] */)[:][:];
```

Syntax

```
specialmatrix(name)
```

```
specialmatrix(name, arg1)
```

```
specialmatrix(name, arg1, arg2, arg3)
```

Purpose

Generate a special matrix.

Return Value

This function returns a special matrix.

Parameters

name The special matrix name selected from the table below.

arg1, arg2, ... Input arguments required by a particular special matrix. The number of arguments and their data types are different for different special matrices.

Special Matrices

Cauchy	ChebyshevVandermonde	Chow	Circul	Celement
Dramadah	DenavitHartenberg	DenavitHartenberg2	Fiedler	Frank
Gear	Hadamard	Hankel	Hilbert	InverseHilbert
Magic	Pascal	Rosser	Toeplitz	Vandermonde
Wilkinson				

.....

Cauchy

Synopsis

```
#include <numeric.h>
```

```
array double specialmatrix("Cauchy", array double c[&], ... /* [ array double r[&] ] */);
```

Syntax

```
specialmatrix("Cauchy", c)
```

```
specialmatrix("Cauchy", c, r)
```

Purpose

Generate a Cauchy matrix.

Return Value

This function returns an $n \times n$ Cauchy matrix.

Parameters

c Input vector with n elements.

c An optional input vector with n elements.

Description

This function generates the $n \times n$ Cauchy matrix whose elements are $y[i][j] = 1/(c[i] + r[j])$. When optional input r is not specified, then inside the function, c is assigned to r . That is, element (i, j) is defined by $y[i][j] = 1/(c[i] + c[j])$.

Example

```
#include <numeric.h>
int main() {
    array double r[5] = {3,4,8,6,2};
    array double c[5] = {8,2.3,9,6,2};
    printf("c=\n%f", c);
    printf("r=\n%f", r);
    printf("specialmatrix(\"Cauchy\", c) =\n%5.4f\n", specialmatrix("Cauchy", c));
    printf("specialmatrix(\"Cauchy\", c, r) =\n%5.4f\n", specialmatrix("Cauchy", c, r));
}
```

Output

```
c=
8.000000 2.300000 9.000000 6.000000 2.000000
r=
3.000000 4.000000 8.000000 6.000000 2.000000
specialmatrix("Cauchy", c) =
0.0625 0.0971 0.0588 0.0714 0.1000
0.0971 0.2174 0.0885 0.1205 0.2326
0.0588 0.0885 0.0556 0.0667 0.0909
0.0714 0.1205 0.0667 0.0833 0.1250
0.1000 0.2326 0.0909 0.1250 0.2500

specialmatrix("Cauchy", c, r) =
0.0909 0.0833 0.0625 0.0714 0.1000
0.1887 0.1587 0.0971 0.1205 0.2326
0.0833 0.0769 0.0588 0.0667 0.0909
0.1111 0.1000 0.0714 0.0833 0.1250
0.2000 0.1667 0.1000 0.1250 0.2500
```

References

Knuth, Donald, *The art of computer programming*, Vol. 1, Addison-Wesley Pub. Co., 1973.

ChebyshevVandermonde**Synopsis**

```
#include <numeric.h>
array double specialmatrix("ChebyshevVandermonde", array double c[&], ... /* [ int m ] */);
```

Syntax

```
specialmatrix("ChebyshevVandermonde", c)
specialmatrix("ChebyshevVandermonde", c, m)
```

Purpose

Generate a Vandermonde-like matrix for the Chebyshev polynomials.

Return Value

This function returns a Vandermonde-like matrix.

Parameters

c Input vector with *n* elements.

m An optional input integral number which defines the rows of the output Vandermonde-like matrix.

Description

This function generates the $n \times m$ Chebyshev Vandermonde matrix whose elements are $y[i][j] = T_i(P[j])$. T_{i-1} is the Chebyshev polynomial of *i*th degree. When the optional input *m* is not specified, the output is a square matrix of $n \times n$. If the user specifies the optional input *m*, then a rectangular version of $n \times m$ Chebyshev Vandermonde matrix is generated.

Example

```
#include <numeric.h>
int main() {
    array double v[5] = {2,3,4,5,6};
    printf("v=\n%f", v);
    printf("specialmatrix(\"ChebyshevVandermonde\", v) =\n%5.4f\n",
           specialmatrix("ChebyshevVandermonde", v));
    printf("specialmatrix(\"ChebyshevVandermonde\", v, 3) =\n%5.4f\n",
           specialmatrix("ChebyshevVandermonde", v, 3));
}
```

Output

```
v=
2.000000 3.000000 4.000000 5.000000 6.000000
specialmatrix("ChebyshevVandermonde", v) =
1.0000 1.0000 1.0000 1.0000 1.0000
2.0000 3.0000 4.0000 5.0000 6.0000
7.0000 17.0000 31.0000 49.0000 71.0000
26.0000 99.0000 244.0000 485.0000 846.0000
97.0000 577.0000 1921.0000 4801.0000 10081.0000

specialmatrix("ChebyshevVandermonde", v, 3) =
1.0000 1.0000 1.0000 1.0000 1.0000
2.0000 3.0000 4.0000 5.0000 6.0000
7.0000 17.0000 31.0000 49.0000 71.0000
```

References

N.J. Higham, *Stability analysis of algorithms for solving confluent Vandermonde-like systems*, SIAM J.

Matrix Anal. Appl., 11 (1990).

Knuth, Donald Ervin, *The art of computer programming*, Vol. 1, Addison-Wesley Pub. Co., 1973.

Chow

Synopsis

```
#include <numeric.h>
```

```
array double specialmatrix("Chow", int n, ... /* [double u, double v ] */);
```

Syntax

```
specialmatrix("Chow", n)
```

```
specialmatrix("Chow", n, u)
```

```
specialmatrix("Chow", n, u, v)
```

Purpose

Generate Chow matrix.

Return Value

This function returns a Chow matrix.

Parameters

n Input integral number. It specifies the degree of the matrix. That is, the size of the Chow matrix is $n \times n$.

u An optional input value.

v An optional input value.

Description

This function generates the $n \times n$ Chow matrix. It is a singular Toeplitz lower Hessenberg matrix. This function returns a square matrix y such that $y = h + v * I$, where h is a square matrix whose elements are $h[i][j] = u^{(i-j)+1}$ and I is a $n \times n$ identity matrix. If the user does not specify the optional arguments u and v , $u = 1.0$ and $v = 0.0$ are used by default.

Example

Generate a 5th order Chow matrix with $u = 1, v = 0$ (default), $u = 0.5, v = 0$ and $u = 0.5, v = 0.67$. Verify the result with the formulation $y = h + v * I$.

```
#include <numeric.h>
int main() {
    int i, j, n=5;
    double u=0.5, v=0.67;
    array double h[n][n], y[n][n];

    printf("specialmatrix(\"Chow\", n) =\n%5.4f\n", specialmatrix("Chow", n));
    printf("specialmatrix(\"Chow\", n, u) =\n%5.4f\n", specialmatrix("Chow", n, u));
    printf("specialmatrix(\"Chow\", n, u, v) =\n%5.4f\n", specialmatrix("Chow", n, u, v));
    for (i=0; i<n; i++)
        for (j=0; j<=min(i+1,n-1); j++)
            h[i][j] = pow(u, (i-j+1));
    y = h + v*identitymatrix(n);
    printf("varify matrix=\n%5.4f",y);
}
```

Output

```
specialmatrix("Chow", n) =
1.0000 1.0000 0.0000 0.0000 0.0000
1.0000 1.0000 1.0000 0.0000 0.0000
1.0000 1.0000 1.0000 1.0000 0.0000
1.0000 1.0000 1.0000 1.0000 1.0000
1.0000 1.0000 1.0000 1.0000 1.0000
```

```
specialmatrix("Chow", n, u) =
0.5000 1.0000 0.0000 0.0000 0.0000
0.2500 0.5000 1.0000 0.0000 0.0000
0.1250 0.2500 0.5000 1.0000 0.0000
0.0625 0.1250 0.2500 0.5000 1.0000
0.0312 0.0625 0.1250 0.2500 0.5000
```

```
specialmatrix("Chow", n, u, v) =
1.1700 1.0000 0.0000 0.0000 0.0000
0.2500 1.1700 1.0000 0.0000 0.0000
0.1250 0.2500 1.1700 1.0000 0.0000
0.0625 0.1250 0.2500 1.1700 1.0000
0.0312 0.0625 0.1250 0.2500 1.1700
```

```
varify matrix=
1.1700 1.0000 0.0000 0.0000 0.0000
0.2500 1.1700 1.0000 0.0000 0.0000
0.1250 0.2500 1.1700 1.0000 0.0000
0.0625 0.1250 0.2500 1.1700 1.0000
0.0312 0.0625 0.1250 0.2500 1.1700
```

References

T.S. Chow, A class of Hessenberg matrices with known eigenvalues and inverses, SIAM Review, 11 (1969), pp. 391-395.

Circul**Synopsis**

```
#include <numeric.h>
array double specialmatrix("Circul", array double c[&]);
```

Syntax

```
specialmatrix("Circul", c)
```

Purpose

Generate a Circul matrix.

Return Value

This function returns a $n \times n$ Circul matrix.

Parameters

c Input vector with n elements. It is the first row of Circul matrix.

Description

A circulant matrix has the property that each row is obtained from the previous one by cyclically permuting

the entries one step forward. specifies the degree of the matrix.

Example

```
#include <numeric.h>
int main() {
    array double c[5] = {2,7,1,9,6};
    printf("c=\n%f", c);
    printf("specialmatrix(\"Circul\", c) =\n%5.4f\n", specialmatrix("Circul", c));
}
```

Output

```
c=
2.000000 7.000000 1.000000 9.000000 6.000000
specialmatrix("Circul", c) =
2.0000 7.0000 1.0000 9.0000 6.0000
6.0000 2.0000 7.0000 1.0000 9.0000
9.0000 6.0000 2.0000 7.0000 1.0000
1.0000 9.0000 6.0000 2.0000 7.0000
7.0000 1.0000 9.0000 6.0000 2.0000
```

References

P.J. Davis, Circulant Matrices, John Wiley, 1977.

Clement

Synopsis

```
#include <numeric.h>
array double specialmatrix("Clement", int n /* [int k ] */);
```

Syntax

```
specialmatrix("Clement", n)
specialmatrix("Clement", n, k)
```

Purpose

Generate a Clement matrix.

Return Value

This function returns an $n \times n$ Clement matrix.

Parameters

n Input integral number. It specifies the order of the square matrix produced.

k Integral number 0 or 1. For $k = 0$ or by default, the matrix is unsymmetric. For $k = 1$, it is a symmetric matrix.

Description

Clement matrix is a tridiagonal matrix with zero diagonal elements and known eigenvalues. The eigenvalues

of the matrix are positive and negative numbers of $n - 1, n - 3, n - 5, \dots, 1$ or 0 .

Example

```
#include <numeric.h>
int main() {
    int n1=5, n2=6;
    array double y1[n1][n1], eigenvalue1[n1],eigenvector1[n1][n1];
    array double y2[n2][n2], eigenvalue2[n2],eigenvector2[n2][n2];

    printf("specialmatrix(\"Clement\", n1, 0) =\n%5.4f\n",
           y1=specialmatrix("Clement", n1, 0));
    printf("specialmatrix(\"Clement\", n1, 1) =\n%5.4f",
           y1=specialmatrix("Clement", n1, 1));
    eigensystem(eigenvalue1,eigenvector1, y1);
    printf("eigenvalue=\n%f\n",eigenvalue1);
    printf("specialmatrix(\"Clement\", n2, 1) =\n%5.4f",
           y2=specialmatrix("Clement", n2, 1));
    eigensystem(eigenvalue2,eigenvector2, y2);
    printf("eigenvalue=\n%f",eigenvalue2);
}
```

Output

```
specialmatrix("Clement", n1, 0) =
0.0000 1.0000 0.0000 0.0000 0.0000
4.0000 0.0000 2.0000 0.0000 0.0000
0.0000 3.0000 0.0000 3.0000 0.0000
0.0000 0.0000 2.0000 0.0000 4.0000
0.0000 0.0000 0.0000 1.0000 0.0000

specialmatrix("Clement", n1, 1) =
0.0000 2.0000 0.0000 0.0000 0.0000
2.0000 0.0000 2.4495 0.0000 0.0000
0.0000 2.4495 0.0000 2.4495 0.0000
0.0000 0.0000 2.4495 0.0000 2.0000
0.0000 0.0000 0.0000 2.0000 0.0000
eigenvalue=
4.000000 -4.000000 0.000000 2.000000 -2.000000

specialmatrix("Clement", n2, 1) =
0.0000 2.2361 0.0000 0.0000 0.0000 0.0000
2.2361 0.0000 2.8284 0.0000 0.0000 0.0000
0.0000 2.8284 0.0000 3.0000 0.0000 0.0000
0.0000 0.0000 3.0000 0.0000 2.8284 0.0000
0.0000 0.0000 0.0000 2.8284 0.0000 2.2361
0.0000 0.0000 0.0000 0.0000 2.2361 0.0000
eigenvalue=
5.000000 -5.000000 3.000000 -3.000000 1.000000 -1.000000
```

References

P.A. Clement, A class of triple-diagonal matrices for test purposes, SIAM Review, 1 (1959), pp. 50-52.

DenavitHartenberg

Synopsis**#include** <numeric.h>**array double specialmatrix**("DenavitHartenberg", **double** *theta* , **double** *d*, **double** *alpha*, **double** *a*);**Syntax**specialmatrix("DenavitHartenberg", *theta*, *d*, *alpha*, *a*)**Purpose**

Generate a Denavit-Hartenberg matrix.

Return ValueThis function returns a 4×4 Denavit-Hartenberg matrix.**Parameters***theta* Input θ value.*d* Input d value.*alpha* Input α value.*a* Input a value.**Description**

The Denavit-Hartenberg matrix is used to describe the coordinate transformation between adjacent links in a three dimensional space. In the Denavit-Hartenberg representation, there are three basic rules for determining and establishing the coordinate frame.

1. The z_{i-1} axis lies along the axis of motion of the i th joint.
2. The x_i axis is normal to the z_{i-1} axis, and pointing away from it.
3. The y_i axis completes the right-handed coordinate system as required.

This first Denavit-Hartenberg representation of a rigid link depends on four geometric parameters associated with each link. These four parameters completely describe any revolute or prismatic joint. Referring to the Figure 11.2, these four parameters are defined as follows:

θ_i is the joint angle from the x_{i-1} axis to the x_i axis about the z_{i-1} axis (using the right-hand rule).

d_i is the distance from the origin of the $(i - 1)$ th coordinate frame to the intersection of the z_{i-1} axis with the x_i axis along the z_{i-1} axis.

a_i is the offset distance from the intersection of the z_i axis with the x_i axis to the origin of the i th frame along the x_i axis, or the shortest distance between the z_{i-1} and z_i axis.

α_i is the offset angle from the z_{i-1} axis to the z_i axis about the x_i axis, using the right-hand rule.

Example

```
#include<numeric.h> // with M_PI
int main() {
    array double t1[4][4], t2[4][4], t3[4][4],
                t4[4][4], t5[4][4], t6[4][4], t[4][4];
```

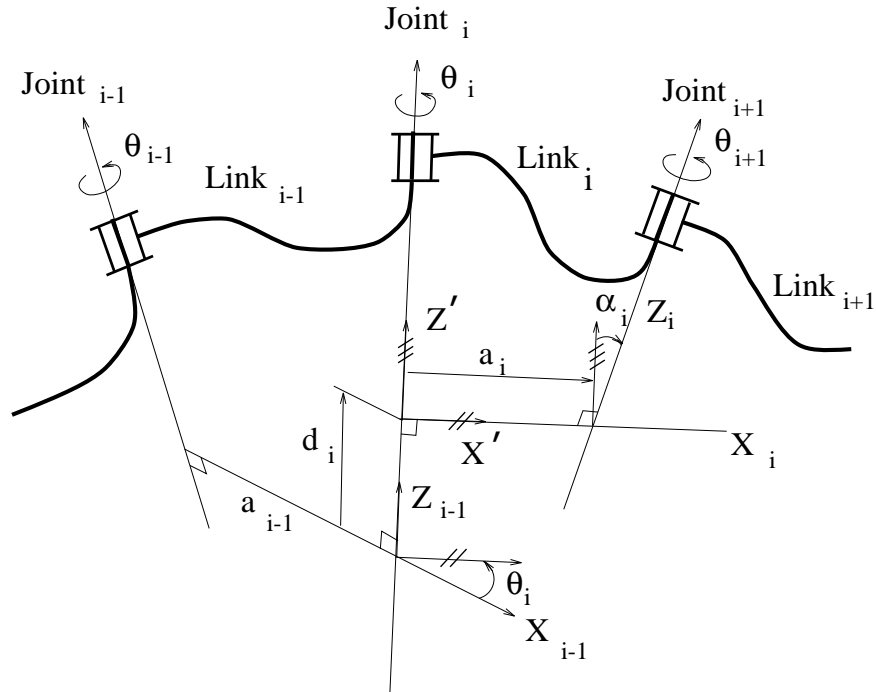


Figure 11.2: Link coordinate system and its parameters

```

double d2r = M_PI/180;
double d[6] = {0, 149.09, 0, 433.07, 0, 165.5};
double alpha[6] = {-90*d2r, 0, 90*d2r, -90*d2r, 90*d2r, 0};
double a[6] = {0, 431.8, -20.32, 0, 0, 0};
double theta[6];

theta[0] = 10*d2r;
theta[1] = 20*d2r;
theta[2] = 30*d2r;
theta[3] = 40*d2r;
theta[4] = 50*d2r;
theta[5] = 60*d2r;
t1 = specialmatrix("DenavitHartenberg",theta[0], d[0], alpha[0], a[0]);
t2 = specialmatrix("DenavitHartenberg",theta[1], d[1], alpha[1], a[1]);
t3 = specialmatrix("DenavitHartenberg",theta[2], d[2], alpha[2], a[2]);
t4 = specialmatrix("DenavitHartenberg",theta[3], d[3], alpha[3], a[3]);
t5 = specialmatrix("DenavitHartenberg",theta[4], d[4], alpha[4], a[4]);
t6 = specialmatrix("DenavitHartenberg",theta[5], d[5], alpha[5], a[5]);
t = t1*t2*t3*t4*t5*t6; // forward kinematics for Puma 560
printf("t = \n%f\n", t);
}

```

Output

```

t =
-0.636562 0.022716 0.770891 815.135915
0.771180 0.029596 0.635929 377.870408
-0.008369 0.999304 -0.036357 140.236602
0.000000 0.000000 0.000000 1.000000

```

References

K. S. Fu, R. C. Gonzalez, C. S. G. Lee, *Robotics: Control, Sensing, Vision, and Intelligence*, McGraw-Hill Pub. Co., 1987.

DenavitHartenberg2

Synopsis

```
#include <numeric.h>
```

```
array double specialmatrix("DenavitHartenberg2", double theta , double d, double alpha, double a);
```

Syntax

```
specialmatrix("DenavitHartenberg2", theta, d, alpha, a)
```

Purpose

generate a Denavit-Hartenberg matrix.

Return Value

This function returns a 4×4 Denavit-Hartenberg matrix.

Parameters

theta Input θ value.

d Input d value.

alpha Input α value.

a Input a value.

Description

This second Denavit-Hartenberg representation of a rigid link depends on four geometric parameters associated with each link. These four parameters completely describe any revolute or prismatic joint. Referring to the Figure11.3, these four parameters are defined as follows:

θ_i is the joint angle from the x_{i-1} axis to the x_i axis about the z_i axis, using the right-hand rule.

d_i is the distance from the origin of the $(i - 1)$ th coordinate frame to the intersection of the z_i axis with the x_i axis along the z_i axis.

a_i is the offset distance from the intersection of the z_i axis with the x_i axis to the origin of the $(i + 1)$ th frame along the x_i axis, or the shortest distance between the z_i and z_{i+1} axis.

α_i is the offset angle from the z_i axis to the z_{i+1} axis about the x_i axis, using the right-hand rule.

Example

```
#include <numeric.h>
int main() {
    printf("specialmatrix(\"DenavitHartenberg2\",M_PI, 0.0, -M_PI, 0.0) =\n%5.4f\n",
        specialmatrix("DenavitHartenberg2",M_PI, 0.0, -M_PI, 0.0));
}
```

Output

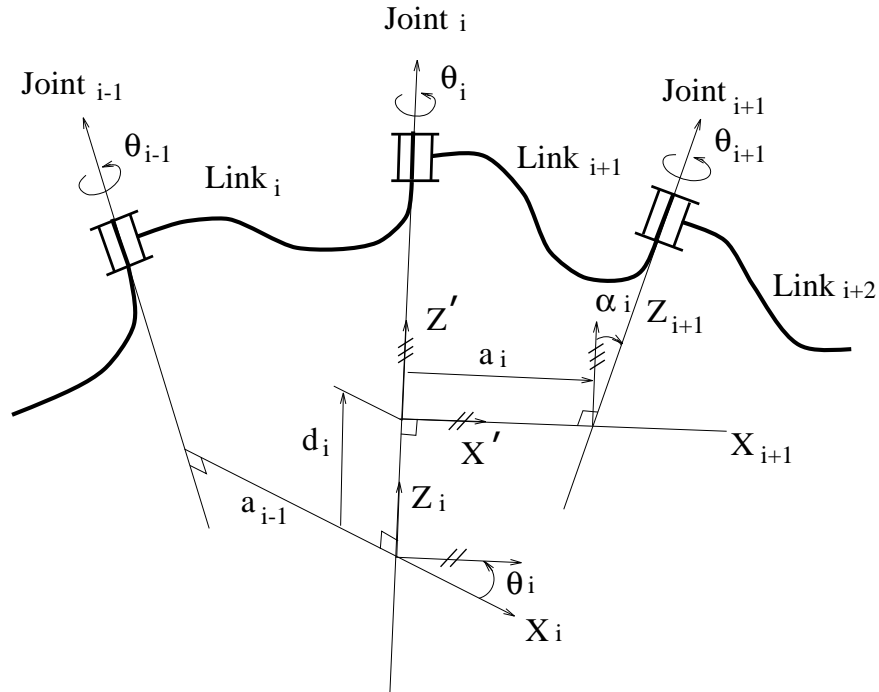


Figure 11.3: Link coordinate system and its parameters

```
specialmatrix("DenavitHartenberg2",M_PI, 0.0, -M_PI, 0.0) =
-1.0000 0.0000 0.0000 0.0000
0.0000 1.0000 -0.0000 -0.0000
-0.0000 -0.0000 -1.0000 -0.0000
0.0000 0.0000 0.0000 1.0000
```

References

John J. Craig, *Robotics: Mechanics & Control*, Addison-Wesley Pub. Co., 1986.

Dramadah**Synopsis**

```
#include <numeric.h>
array double specialmatrix("Dramadah1", int n /* [int k ] */);
```

Syntax

```
specialmatrix("Dramadah1", n)
specialmatrix("Dramadah1", n, k)
```

Purpose

generate a Dramadah matrix.

Return Value

This function returns an $n \times n$ Dramadah matrix.

Parameters

n Input integral number. It specifies the order of the square matrix produced.

k Integral number 1, 2, or 3.

k = 1 or by default, the returned matrix *y* is Toeplitz, with $abs|y|= 1$. Each element of y^{-1} is an integral value.

k = 2, *y* is upper triangular and Toeplitz. Each element of y^{-1} is an integral value.

k = 3, *y* has a maximal determinant among (0,1) lower Hessenberg matrices.

Description

The Dramadah matrix has elements of zeros and ones. The inverse matrix of Dramadah matrix has large integer elements.

Example

```
#include <numeric.h>
int main() {
    int n=4;
    array double y[n][n];
    printf("specialmatrix(\"Dramadah\", n) =\n%5.4f", y=specialmatrix("Dramadah", n));
    printf("absolute value of determinant of y=%f\n",abs(determinant(y)));
    printf("inverse of y=\n%5.4f\n",inverse(y));
    printf("specialmatrix(\"Dramadah\", n,2) =\n%5.4f", y=specialmatrix("Dramadah", n,2));
    printf("inverse of y=\n%5.4f\n",inverse(y));
    printf("specialmatrix(\"Dramadah\", n,3) =\n%5.4f", y=specialmatrix("Dramadah", n,3));
    printf("determinant of y=%f\n",determinant(y));
}
```

Output

```
specialmatrix("Dramadah", n) =
1.0000 1.0000 0.0000 1.0000
0.0000 1.0000 1.0000 0.0000
0.0000 0.0000 1.0000 1.0000
1.0000 0.0000 0.0000 1.0000
absolute value of determinant of y=1.000000
inverse of y=
-1.0000 1.0000 -1.0000 2.0000
1.0000 0.0000 0.0000 -1.0000
-1.0000 1.0000 0.0000 1.0000
1.0000 -1.0000 1.0000 -1.0000

specialmatrix("Dramadah", n,2) =
1.0000 1.0000 0.0000 1.0000
0.0000 1.0000 1.0000 0.0000
0.0000 0.0000 1.0000 1.0000
0.0000 0.0000 0.0000 1.0000
inverse of y=
1.0000 -1.0000 1.0000 -2.0000
0.0000 1.0000 -1.0000 1.0000
0.0000 0.0000 1.0000 -1.0000
0.0000 0.0000 0.0000 1.0000

specialmatrix("Dramadah", n,3) =
1.0000 1.0000 0.0000 0.0000
```

```
0.0000 1.0000 1.0000 0.0000
1.0000 0.0000 1.0000 1.0000
0.0000 1.0000 0.0000 1.0000
determinant of y=3.000000
```

References

R.L. Graham and N.J.A. Sloane, Anti-Hadamard matrices, *Linear Algebra and Appl.*, 62 (1984), pp. 113-137. L. Ching, The maximum determinant of an $n \times n$ lower Hessenberg (0,1) matrix, *Linear Algebra and Appl.*, 183 (1993), pp. 147-153.

Fiedler

Synopsis

```
#include <numeric.h>
array double specialmatrix("Fiedler", array double c[&]);
```

Syntax

```
specialmatrix("Fiedler", c)
```

Purpose

Generate a Fiedler matrix.

Return Value

This function returns an $n \times n$ Fiedler matrix.

Parameters

c Input vector with n elements. The length of the vector n specifies the order of the output matrix. That is, the size of the Fiedler matrix is $n \times n$.

Description

A Fiedler matrix has a dominant positive eigenvalue and all the other eigenvalues are negative. A Fiedler matrix is a symmetric matrix with elements $abs(c[i] - c[j])$.

Example

```
#include <numeric.h>
int main() {
    int n=5;
    array double y[n][n], eigenvalue[n], eigenvector[n][n];
    array double c[5] = {2,-7,1,-9,6};
    printf("specialmatrix(\"Fiedler\",c) =\n%5.4f", y=specialmatrix("Fiedler", c));
    eigensystem(eigenvalue,eigenvector, y);
    printf("eigenvalues =\n%f",eigenvalue);
}
```

Output

```
specialmatrix("Fiedler",c) =
0.0000 9.0000 1.0000 11.0000 4.0000
9.0000 0.0000 8.0000 2.0000 13.0000
```

```

1.0000 8.0000 0.0000 10.0000 5.0000
11.0000 2.0000 10.0000 0.0000 15.0000
4.0000 13.0000 5.0000 15.0000 0.0000
eigenvalues =
-23.319294 32.070623 -0.904663 -5.933579 -1.913088

```

References

G. Szego, Solution to problem 3705, Amer. Math. Monthly, 43 (1936), pp. 246-259.

J. Todd, Basic Numerical Mathematics, Vol. 2: Numerical Algebra, Birkhauser, Basel, and Academic Press, New York, 1977, p. 159.

Frank

Synopsis

```

#include <numeric.h>
array double specialmatrix("Frank", int n /* [int k ] */);

```

Syntax

```

specialmatrix("Frank", n)
specialmatrix("Frank", n, k)

```

Purpose

Generate a Frank matrix.

Return Value

This function returns an $n \times n$ Frank matrix.

Parameters

n Input integral number. It specifies the order of the square matrix produced.

k Integral number 0 or 1. For $k = 0$ or by default, the matrix is an upper Hessenberg with determinant 1. If $k = 1$, the elements are reflected about the anti-diagonal.

Description

A Frank matrix is an upper Hessenberg with ill-conditioned eigenvalues. The eigenvalues of the Frank matrix are positive and occur in reciprocal pairs. If n is odd, 1 is an eigenvalue. The first m , with $m = \text{floor}(n/2)$, eigenvalues in the ascending order are ill-conditioned.

Example

```

#include <numeric.h>
int main() {
    int n1=4, n2=5;
    array double y1[n1][n1], eigenvalue1[n1], eigenvector1[n1][n1];
    array double y2[n2][n2], eigenvalue2[n2], eigenvector2[n2][n2];
    printf("specialmatrix(\"Frank\",n1) =\n%5.4f", y1=specialmatrix("Frank", n1));
    eigensystem(eigenvalue1,eigenvector1,y1);
    printf("eigenvalues =\n%5.4f", eigenvalue1);
    printf("determinant =%5.4f\n",determinant(y1));
    printf("specialmatrix(\"Frank\",n2, 1) =\n%5.4f", y2=specialmatrix("Frank", n2, 1));
    eigensystem(eigenvalue2,eigenvector2,y2);
}

```

```

    printf("eigenvalues =\n%5.4f", eigenvalue2);
    printf("determinant =%5.4f\n", determinant (y2));
}

```

Output

```

specialmatrix("Frank", n1) =
4.0000 3.0000 2.0000 1.0000
3.0000 3.0000 2.0000 1.0000
0.0000 2.0000 2.0000 1.0000
0.0000 0.0000 1.0000 1.0000
eigenvalues =
0.1367 7.3127 2.0666 0.4839
determinant =1.0000
specialmatrix("Frank", n2, 1) =
1.0000 1.0000 1.0000 1.0000 1.0000
1.0000 2.0000 2.0000 2.0000 2.0000
0.0000 2.0000 3.0000 3.0000 3.0000
0.0000 0.0000 3.0000 4.0000 4.0000
0.0000 0.0000 0.0000 4.0000 5.0000
eigenvalues =
3.5566 10.0629 1.0000 0.0994 0.2812
determinant =1.0000

```

References

- W.L. Frank, Computing eigenvalues of complex matrices by determinant evaluation and by methods of Danilewski and Wielandt, *J. Soc. Indust. Appl. Math.*, 6 (1958), pp. 378-392 (see pp. 385 and 388).
- G.H. Golub and J.H. Wilkinson, Ill-conditioned eigensystems and the computation of the Jordan canonical form, *SIAM Review*, 18 (1976), pp. 578-619 (Section 13).
- H. Rutishauser, On test matrices, *Programmation en Mathematiques Numeriques*, Editions Centre Nat. Recherche Sci., Paris, 165, 1966, pp. 349-365. Section 9.
- J.H. Wilkinson, Error analysis of floating-point computation, *Numer. Math.*, 2 (1960), pp. 319-340 (Section 8).
- J.H. Wilkinson, *The Algebraic Eigenvalue Problem*, Oxford University Press, 1965 (pp. 92-93).
- P.J. Eberlein, A note on the matrices denoted by B_n , *SIAM J. Appl. Math.*, 20 (1971), pp. 87-92.
- J.M. Varah, A generalization of the Frank matrix, *SIAM J. Sci. Stat. Comput.*, 7 (1986), pp. 835-839.

Gear

Synopsis

```

#include <numeric.h>
array double specialmatrix("Gear", int n, ... /* [int ni, int nj] */);

```

Syntax

```

specialmatrix("Gear", n)
specialmatrix("Gear", n, ni)
specialmatrix("Gear", n, ni, nj)

```

Purpose

Generate a Gear matrix.

Return Value

This function returns an $n \times n$ Gear matrix.

Parameters

n Input integral number. It specifies the order of the square matrix produced.

ni Integral number $abs(ni) \leq n$.

nj Integral number $abs(nj) \leq n$.

Description

A Gear matrix is an $n \times n$ matrix with ones on the sub- and super-diagonals. The $[0][abs(ni)]$ element is assigned the value $sign(ni)$ and the $[n-1][n-1-abs(nj)]$ is assigned $sign(nj)$. All eigenvalues of Gear matrix are of the form $2 * cos(a)$ and the eigenvectors are of the form $[sin(w + a), sin(w + 2a), \dots, sin(w + Na)]$, where a and w are constants.

Example

```
#include <numeric.h>
int main() {
    printf("specialmatrix(\"Gear\",7) =\n%5.4f\n", specialmatrix("Gear", 7));
    printf("specialmatrix(\"Gear\",7,3,6) =\n%5.4f\n", specialmatrix("Gear", 7, 3, 6));
}
```

Output

```
specialmatrix("Gear",7) =
0.0000 1.0000 0.0000 0.0000 0.0000 0.0000 1.0000
1.0000 0.0000 1.0000 0.0000 0.0000 0.0000 0.0000
0.0000 1.0000 0.0000 1.0000 0.0000 0.0000 0.0000
0.0000 0.0000 1.0000 0.0000 1.0000 0.0000 0.0000
0.0000 0.0000 0.0000 1.0000 0.0000 1.0000 0.0000
0.0000 0.0000 0.0000 0.0000 1.0000 0.0000 1.0000
-1.0000 0.0000 0.0000 0.0000 0.0000 1.0000 0.0000
```

```
specialmatrix("Gear",7,3,6) =
0.0000 1.0000 0.0000 1.0000 0.0000 0.0000 0.0000
1.0000 0.0000 1.0000 0.0000 0.0000 0.0000 0.0000
0.0000 1.0000 0.0000 1.0000 0.0000 0.0000 0.0000
0.0000 0.0000 1.0000 0.0000 1.0000 0.0000 0.0000
0.0000 0.0000 0.0000 1.0000 0.0000 1.0000 0.0000
0.0000 0.0000 0.0000 0.0000 1.0000 0.0000 1.0000
1.0000 0.0000 0.0000 0.0000 0.0000 1.0000 0.0000
```

References

C.W. Gear, A simple set of test matrices for eigenvalue programs, Math. Comp., 23 (1969), pp. 119-125.

Hadamard**Synopsis**

```
#include <numeric.h>
```

```
array double specialmatrix("Hadamard", int n);
```

Syntax

```
specialmatrix("Hadamard", n)
```

Purpose

Generate a Hadamard matrix.

Return Value

This function returns an $n \times n$ Hadamard matrix.

Parameters

n Input integral number. It specifies the degree of the matrix. That is, the size of Hadamard matrix is $n \times n$.

Description

This function generates an $n \times n$ Hadamard matrix. A Hadamard matrix is a matrix y with elements 1 or -1 such that $y^T * y = n * I$, where I is an $n \times n$ identity matrix. An $n \times n$ Hadamard matrix with $n > 2$ exists only if $n \% 4 = 0$. This function handles only the cases where $n, n/12$ or $n/20$ is a power of 2.

Example

```
#include <numeric.h>
int main() {
    int n=8;
    array double y[n][n];
    printf("specialmatrix(\"Hadamard\", n) =\n%4.1f", y=specialmatrix("Hadamard", n));
    printf("verify: transpose(y)*y-n*identitymatrix(n)=0\n%5.4f",
          transpose(y)*y-n*identitymatrix(n));
}
```

Output

```
specialmatrix("Hadamard", n) =
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0 -1.0  1.0 -1.0  1.0 -1.0  1.0 -1.0
 1.0  1.0 -1.0 -1.0  1.0  1.0 -1.0 -1.0
 1.0 -1.0 -1.0  1.0  1.0 -1.0 -1.0  1.0
 1.0  1.0  1.0  1.0 -1.0 -1.0 -1.0 -1.0
 1.0 -1.0  1.0 -1.0 -1.0  1.0 -1.0  1.0
 1.0  1.0 -1.0 -1.0 -1.0 -1.0  1.0  1.0
 1.0 -1.0 -1.0  1.0 -1.0  1.0  1.0 -1.0
verify: transpose(y)*y-n*identitymatrix(n)=0
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
```

References

Reference: S.W. Golomb and L.D. Baumert, The search for Hadamard matrices, Amer. Math. Monthly, 70 (1963) pp. 12-17.

Hankel

Synopsis

```
#include <numeric.h>
```

```
array double specialmatrix("Hankel", array double c[&], ... /* [ array double r[&] ] */);
```

Syntax

```
specialmatrix("Hankel", c)
```

```
specialmatrix("Hankel", c, r)
```

Purpose

Generate a Hankel matrix.

Return Value

This function returns a Hankel matrix.

Parameters

c Input vector with *n* elements.

r An optional input vector with *m* elements.

Description

If the optional input vector *r* is not specified, this function generates a square Hankel matrix whose column is *c* and whose elements are zero below the anti-diagonal. If the optional input vector *r* is specified, it returns an $n \times m$ Hankel matrix whose first column is *c* and last row is *r*. Hankel matrices are symmetric.

Example

```
#include <numeric.h>
int main() {
    array double r[3] = {2, 6, 8};
    array double c[5] = {8, 2.3, 9, 6, 2};
    printf("specialmatrix(\"Hankel\", c) =\n%5.1f\n", specialmatrix("Hankel", c));
    printf("specialmatrix(\"Hankel\", c, r) =\n%5.1f\n", specialmatrix("Hankel", c, r));
    printf("specialmatrix(\"Hankel\", r, c) =\n%5.1f\n", specialmatrix("Hankel", r, c));
}
```

Output

```
specialmatrix("Hankel", c) =
  8.0  2.3  9.0  6.0  2.0
  2.3  9.0  6.0  2.0  0.0
  9.0  6.0  2.0  0.0  0.0
  6.0  2.0  0.0  0.0  0.0
  2.0  0.0  0.0  0.0  0.0

specialmatrix("Hankel", c, r) =
  8.0  2.3  9.0
  2.3  9.0  6.0
  9.0  6.0  2.0
```

```

6.0  2.0  6.0
2.0  6.0  8.0

specialmatrix("Hankel", r, c) =
2.0  6.0  8.0  2.3  9.0
6.0  8.0  2.3  9.0  6.0
8.0  2.3  9.0  6.0  2.0

```

References

.....

Hilbert

Synopsis

```

#include <numeric.h>
array double specialmatrix("Hilbert", int n);

```

Syntax

```
specialmatrix("Hilbert", n)
```

Purpose

Generate a Hilbert matrix.

Return Value

This function returns an $n \times n$ Hilbert matrix.

Parameters

n Input integral number. It specifies the degree of the matrix. That is, the size of Hilbert matrix is $n \times n$.

Description

This function generates an $n \times n$ Hilbert matrix. A Hilbert matrix is a matrix y with elements $y[i][j] = 1/(i + j + 1)$. It is a famous example of a badly conditioned matrix.

Example

```

#include <numeric.h>
int main() {
    printf("specialmatrix(\"Hilbert\", 7) =\n%5.3f\n", specialmatrix("Hilbert", 7));
}

```

Output

```

specialmatrix("Hilbert", 7) =
1.000 0.500 0.333 0.250 0.200 0.167 0.143
0.500 0.333 0.250 0.200 0.167 0.143 0.125
0.333 0.250 0.200 0.167 0.143 0.125 0.111
0.250 0.200 0.167 0.143 0.125 0.111 0.100
0.200 0.167 0.143 0.125 0.111 0.100 0.091
0.167 0.143 0.125 0.111 0.100 0.091 0.083
0.143 0.125 0.111 0.100 0.091 0.083 0.077

```


References

InverseHilbert
Synopsis

```
#include <numeric.h>
```

```
array double specialmatrix("InverseHilbert", int n);
```

Syntax

```
specialmatrix("InverseHilbert", n)
```

Purpose

generate an inverse Hilbert matrix.

Return Value

This function returns an $n \times n$ inverse Hilbert matrix.

Parameters

n Input integral number. It specifies the degree of the matrix. That is, the size of the inverse Hilbert matrix is $n \times n$.

Description

This function generates an $n \times n$ inverse Hilbert matrix. The result is exact for n less than 15.

Example

```
#include <numeric.h>
int main() {
    printf("specialmatrix(\"InverseHilbert\", 7) =\n%8.4g\n",
        specialmatrix("InverseHilbert", 7));
}
```

Output

```
specialmatrix("InverseHilbert", 7) =
    49    -1176    8820 -2.94e+04  4.851e+04 -3.881e+04  1.201e+04
   -1176  3.763e+04 -3.175e+05  1.129e+06 -1.94e+06  1.597e+06 -5.045e+05
    8820 -3.175e+05  2.858e+06 -1.058e+07  1.871e+07 -1.572e+07  5.045e+06
  -2.94e+04  1.129e+06 -1.058e+07  4.032e+07 -7.276e+07  6.209e+07 -2.018e+07
  4.851e+04 -1.94e+06  1.871e+07 -7.276e+07  1.334e+08 -1.153e+08  3.784e+07
 -3.881e+04  1.597e+06 -1.572e+07  6.209e+07 -1.153e+08  1.006e+08 -3.33e+07
  1.201e+04 -5.045e+05  5.045e+06 -2.018e+07  3.784e+07 -3.33e+07  1.11e+07
```

References

Magic

Synopsis**#include** <numeric.h>**array double specialmatrix**("Magic", int *n*);**Syntax**specialmatrix("Magic", *n*)**Purpose**

generate a Magic matrix.

Return ValueThis function returns an $n \times n$ Magic matrix.**Parameters***n* Input integral number. It specifies the degree of the matrix. That is, the size of Magic matrix is $n \times n$.**Description**This function generates an $n \times n$ Magic matrix. A Magic matrix is a matrix *y* with elements from 1 through n^2 with equal row, column, and diagonal sums.**Example**

```
#include <numeric.h>
int main() {
    int i, n=7;
    array double y[n][n], sum1[n];
    printf("specialmatrix(\"Magic\",n) =\n%4.0f", y=specialmatrix("Magic", n));
    sum(y, sum1);
    printf("Sum of columns=\n%4.0f", sum1);
    sum(transpose(y), sum1);
    printf("Sum of rows=\n%4.0f", sum1);
    printf("Sum of diagonal elements=%4.0f\n", trace(y));
    rot90(y, y);
    printf("Sum of anti-diagonal elements=%4.0f\n", trace(y));
}
```

Output

```
specialmatrix("Magic",n) =
 30  38  46   5  13  21  22
 39  47   6  14  15  23  31
 48   7   8  16  24  32  40
  1   9  17  25  33  41  49
 10  18  26  34  42  43   2
 19  27  35  36  44   3  11
 28  29  37  45   4  12  20
Sum of columns=
175 175 175 175 175 175 175
Sum of rows=
175 175 175 175 175 175 175
Sum of diagonal elements= 175
Sum of anti-diagonal elements= 175
```

Pascal

Synopsis

```
#include <numeric.h>
array double specialmatrix("Pascal", int n /* [int k ] */);
```

Syntax

```
specialmatrix("Pascal", n)
specialmatrix("Pascal", n, k)
```

Purpose

generate a Pascal matrix.

Return Value

This function returns an $n \times n$ Pascal matrix.

Parameters

n Input integral number. It specifies the order of the square matrix produced.

k Optional input Integral number 0, 1 or 2.

Description

If the optional input *k* is not specified by default, or specified with 0, it returns a symmetric positive definite matrix with integer elements, which is made up from Pascal's triangular. Its inverse matrix has integer elements. If the optional input *k* is specified with 1, it is a lower triangular Choleskey factor of the Pascal matrix. Its inverse matrix is itself. If the optional input *k* is specified with 2, it is a transposed and permuted version of *specialmatrix("Pascal",n,1)* which is a cubic root of the identity.

Example

```
#include <numeric.h>
int main() {
    int n=5;
    array double y[n][n];

    printf("specialmatrix(\"Pascal\", n) =\n%5.1f", y=specialmatrix("Pascal", n));
    printf("inverse matrix=\n%5.1f\n", inverse(y));
    printf("specialmatrix(\"Pascal\", n, 1) =\n%5.1f", y=specialmatrix("Pascal", n, 1));
    printf("inverse matrix=\n%5.1f\n", inverse(y));
    printf("specialmatrix(\"Pascal\", n, 2) =\n%5.1f", y=specialmatrix("Pascal", n, 2));
}
```

Output

```
specialmatrix("Pascal", n) =
  1.0  1.0  1.0  1.0  1.0
  1.0  2.0  3.0  4.0  5.0
  1.0  3.0  6.0 10.0 15.0
  1.0  4.0 10.0 20.0 35.0
```

```

1.0  5.0  15.0  35.0  70.0
inverse matrix=
5.0 -10.0  10.0  -5.0  1.0
-10.0  30.0 -35.0  19.0  -4.0
10.0 -35.0  46.0 -27.0  6.0
-5.0  19.0 -27.0  17.0  -4.0
1.0  -4.0   6.0  -4.0  1.0

specialmatrix("Pascal", n, 1) =
1.0  0.0  0.0  0.0  0.0
1.0 -1.0  0.0  0.0  0.0
1.0 -2.0  1.0  0.0  0.0
1.0 -3.0  3.0 -1.0  0.0
1.0 -4.0  6.0 -4.0  1.0
inverse matrix=
1.0  0.0  0.0  0.0  0.0
1.0 -1.0  0.0  0.0  0.0
1.0 -2.0  1.0  0.0  0.0
1.0 -3.0  3.0 -1.0  0.0
1.0 -4.0  6.0 -4.0  1.0

specialmatrix("Pascal", n, 2) =
0.0  0.0  0.0  0.0 -1.0
0.0  0.0  0.0 -1.0  4.0
0.0  0.0  1.0  3.0 -6.0
0.0 -1.0 -2.0 -3.0  4.0
-1.0 -1.0 -1.0 -1.0  1.0

```

References**Rosser****Synopsis**

```

#include <numeric.h>
array double specialmatrix("Rosser");

```

Syntax

```
specialmatrix("Rosser")
```

Purpose

Generate a Rosser matrix.

Return Value

This function returns an 8×8 Rosser matrix.

Description

A Rosser matrix is an 8×8 matrix R .

$$\begin{bmatrix} 611 & 196 & -192 & 407 & -8 & -52 & -49 & 29 \\ 196 & 899 & 113 & -192 & -71 & -43 & -8 & -44 \\ -192 & 113 & 899 & 196 & 61 & 49 & 8 & 52 \\ 407 & -192 & 196 & 611 & 8 & 44 & 59 & -23 \\ -8 & -71 & 61 & 8 & 411 & -599 & 208 & 208 \\ -52 & -43 & 49 & 44 & -599 & 411 & 208 & 208 \\ -49 & -8 & 8 & 59 & 208 & 208 & 99 & -911 \\ 29 & -44 & 52 & -23 & 208 & 208 & -911 & 99 \end{bmatrix}$$

This matrix was a challenge for many matrix eigenvalue algorithms. It has:

- Two eigenvalues with the same value.
- Three nearly equal eigenvalues.
- Dominant eigenvalues of opposite sign.
- A zero eigenvalue.
- A small, nonzero eigenvalue.

Example

```
#include <numeric.h>
int main() {
    array double y[8][8], eigenvalue[8], eigenvector[8][8];
    printf("specialmatrix(\"Rosser\") =\n%5.1f\n", y=specialmatrix("Rosser"));
    eigensystem(eigenvalue, eigenvector, y);
    printf("eigenvalue =\n%7.3f", eigenvalue);
}
```

Output

```
specialmatrix("Rosser") =
611.0 196.0 -192.0 407.0 -8.0 -52.0 -49.0 29.0
196.0 899.0 113.0 -192.0 -71.0 -43.0 -8.0 -44.0
-192.0 113.0 899.0 196.0 61.0 49.0 8.0 52.0
407.0 -192.0 196.0 611.0 8.0 44.0 59.0 -23.0
-8.0 -71.0 61.0 8.0 411.0 -599.0 208.0 208.0
-52.0 -43.0 49.0 44.0 -599.0 411.0 208.0 208.0
-49.0 -8.0 8.0 59.0 208.0 208.0 99.0 -911.0
29.0 -44.0 52.0 -23.0 208.0 208.0 -911.0 99.0

eigenvalue =
-1020.049 -0.000 0.098 1020.049 1020.000 1019.902 1000.000 1000.000
.....
```

Toeplitz

Synopsis

```
#include <numeric.h>
```

```
array double specialmatrix("Toeplitz", array double c[&], ... /* [ array double r[&] ] */);
```

Syntax

```
specialmatrix("Toeplitz", c)
specialmatrix("Toeplitz", c, r)
```

Purpose

Generate a Toeplitz matrix.

Return Value

This function returns a Toeplitz matrix.

Parameters

c Input vector with *n* elements.

r An optional input vector with *m* elements.

Description

If the optional input vector *r* is not specified, this function generates a square symmetric Toeplitz matrix whose first column is *c*. If the optional input vector *r* is specified, it returns an $n \times m$ non-symmetric Toeplitz matrix whose first column is *c* and first row is *r*.

Example

```
#include <numeric.h>
int main() {
    array double r[3] = {8, 6, 2};
    array double c[5] = {8, 2.3, 9, 6, 2};
    printf("specialmatrix(\"Toeplitz\", c) =\n%5.1f\n", specialmatrix("Toeplitz", c));
    printf("specialmatrix(\"Toeplitz\", c, r) =\n%5.1f\n",
           specialmatrix("Toeplitz", c, r));
    printf("specialmatrix(\"Toeplitz\", r, c) =\n%5.1f\n",
           specialmatrix("Toeplitz", r, c));
}
```

Output

```
specialmatrix("Toeplitz", c) =
  8.0  2.3  9.0  6.0  2.0
  2.3  8.0  2.3  9.0  6.0
  9.0  2.3  8.0  2.3  9.0
  6.0  9.0  2.3  8.0  2.3
  2.0  6.0  9.0  2.3  8.0

specialmatrix("Toeplitz", c, r) =
  8.0  6.0  2.0
  2.3  8.0  6.0
  9.0  2.3  8.0
  6.0  9.0  2.3
  2.0  6.0  9.0

specialmatrix("Toeplitz", r, c) =
```

```

8.0  2.3  9.0  6.0  2.0
6.0  8.0  2.3  9.0  6.0
2.0  6.0  8.0  2.3  9.0

```

Vandermonde

Synopsis

```
#include <numeric.h>
```

```
array double specialmatrix("Vandermonde", array double v[&])[:, :];
```

```
syntax specialmatrix("Vandermonde", v);
```

Purpose

generate a Calculate the Vandermonde.

Return Value

This function returns an $n \times n$ Vandermonde matrix.

Parameters

v An input vector of size n .

Description

This function calculates the Vandermonde matrix of a vector of n elements. The Vandermonde matrix is an $n \times n$ matrix defined as:

$$a_{i,j} = v_i^{j-1}$$

The Vandermonde matrix can be used in the fitting of polynomial to data.

Example

```

#include <numeric.h>
int main() {
    array double v[3] = {2,3,4};
    printf("specialmatrix(\"Vandermonde\", v) =\n%f\n",
        specialmatrix("Vandermonde", v));
}

```

Output

```

specialmatrix("Vandermonde", v) =
1.000000 2.000000 4.000000
1.000000 3.000000 9.000000
1.000000 4.000000 16.000000

```

See Also

polyfit().

Wilkinson

Synopsis**#include** <numeric.h>**array double** specialmatrix("Wilkinson", int *n*);**Syntax**specialmatrix("Wilkinson", *n*)**Purpose**

generate a Wilkinson matrix.

Return ValueThis function returns an $n \times n$ Wilkinson's eigenvalue testing matrix.**Parameters***n* Input integral number. It specifies the order of the square matrix produced.**Description**

Wilkinson's eigenvalue testing matrix is a symmetric, tridiagonal matrix with pairs of nearly, but not exactly, equal eigenvalues.

Example

```
#include <numeric.h>
int main() {
    int n=7;
    array double y[n][n], eigenvalue[n], eigenvector[n][n];
    printf("specialmatrix(\"Wilkinson\", n) =\n%5.1f", y=specialmatrix("Wilkinson", n));
    eigensystem(eigenvalue, eigenvector, y);
    printf("eigenvalues =\n%5.1f", eigenvalue);
}
```

Output

```
specialmatrix("Wilkinson", n) =
  3.0  1.0  0.0  0.0  0.0  0.0  0.0
  1.0  3.0  1.0  0.0  0.0  0.0  0.0
  0.0  1.0  3.0  1.0  0.0  0.0  0.0
  0.0  0.0  1.0  3.0  1.0  0.0  0.0
  0.0  0.0  0.0  1.0  3.0  1.0  0.0
  0.0  0.0  0.0  0.0  1.0  3.0  1.0
  0.0  0.0  0.0  0.0  0.0  1.0  3.0
eigenvalues =
  4.8  1.2  1.6  4.4  3.0  3.8  2.2
```

sqrtm

Synopsis

```
#include <numeric.h>
```

```
int sqrtm(array double complex y[&][&], array double complex x[&][&]);
```

Syntax

```
sqrtm(y, x)
```

Purpose

Computes the square root of the matrix.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x Input square matrix. It contains data to be calculated.

y Output square matrix which contains data of the result of the principal square root of the matrix *x*.

Description

This function computes the principal square root of the matrix *x*, that is, $x = y * y$ where *x* and *y* are square matrices. If *x* has non-positive eigenvalues, the result *y* is a real matrix. If *x* has any eigenvalue of negative real part then a complex result *y* will be produced.

Example

Calculation of square roots of matrices of different data types.

```
#include <numeric.h>
int main() {
    array double x[3][3]={0.8,0.2,0.1,
                        0.2,0.7,0.3,
                        0.1,0.3,0.6};
    array double complex zx[3][3]={complex(1,1),complex(2,2),0,
                                   3,complex(4,1),complex(2,5),
                                   0,0,0};
    array double complex zy[3][3];
    array double y[3][3];

    sqrtm(y, x);
    printf("x = \n%5.3f", x);
    printf("y = \n%5.3f", y);
    printf("\n");

    sqrtm(zy, zx);
    printf("zx = \n%5.3f", zx);
    printf("zy = \n%5.3f", zy);
}
```

Output

```
x =
0.800 0.200 0.100
0.200 0.700 0.300
0.100 0.300 0.600
y =
0.886 0.113 0.048
0.113 0.807 0.189
0.048 0.189 0.750

zx =
complex(1.000,1.000) complex(2.000,2.000) complex(0.000,0.000)
complex(3.000,0.000) complex(4.000,1.000) complex(2.000,5.000)
complex(0.000,0.000) complex(0.000,0.000) complex(0.000,0.000)
zy =
complex(0.690,0.818) complex(1.003,0.353) complex(-3.155,0.649)
complex(1.017,-0.488) complex(1.708,0.331) complex(3.068,1.046)
complex(0.000,0.000) complex(0.000,0.000) complex(0.000,0.000)
```

See Also

logm(), funm(), cfunm(), expm().

References

G. H. Golub, C. F. Van Loan, *Matrix Computations* Third edition, The Johns Hopkins University Press, 1996

std**Synopsis****#include** <numeric.h>**double std**(array double &a, ... /* [array double v[:]] */);**Syntax****std**(a)**std**(a, v)**Purpose**

Calculate the standard deviation of a data set.

Return Value

This function returns the standard deviation.

Parameters*a* Input array which contains a data set.*v* Output array which contains the standard deviations of each row of the array.**Description**

This function calculates the standard deviation of a data set.

AlgorithmThe standard deviation of the data set σ_i is defined as

$$\sigma = \sqrt{\frac{1}{N-1} \sum_{j=1}^N (X_j - \bar{X})^2}$$

where N is the number of observations of data set x , \bar{x} is the mean of data set x .**Example1**

```
#include <numeric.h>
int main() {
    double a[3] = {1,2,3};
    double b[2][3] = {1,2,3,4,5,6};
    int    b1[2][3] = {1,2,3,4,5,6};
    double c[2][3][5];
    c[0][0][0] = 10;
    c[0][0][1] = 20;
    double stdval;

    stdval = std(a);
    printf("std(a) = %f\n", stdval);
    stdval = std(b);
    printf("std(b) = %f\n", stdval);
    stdval = std(b1);
    printf("std(b1) = %f\n", stdval);
    stdval = std(c);
```

```
    printf("std(c) = %f\n", stdval);  
}
```

Output1

```
std(a) = 1.000000  
std(b) = 1.870829  
std(b1) = 1.870829  
std(c) = 4.025779
```

Example2

```
#include <numeric.h>  
int main() {  
    double a[2][3] = {1,2,3,  
                     4,5,6};  
    array double b[3][4] = {1,2,3,4,  
                           5,6,7,8,  
                           1,2,3,4};  
    array double stdv1[2], stdv2[3];  
  
    std(a, stdv1);  
    printf("std(a, stdv1) = %f\n", stdv1);  
    std(b, stdv2);  
    printf("std(b, stdv2) = %f\n", stdv2);  
}
```

Output2

```
std(a, stdv1) = 1.000000 1.000000  
  
std(b, stdv2) = 1.290994 1.290994 1.290994
```

See Also

mean(), median().

References

sum

Synopsis

```
#include <numeric.h>
```

```
double sum(array double &a, ... /*[array double v[:]]*/);
```

Syntax

```
sum(a)
```

```
sum(a, v)
```

Purpose

Calculate the sum of all elements of an array and the sums of each row of a two-dimensional array.

Return Value

This function returns the sum of all elements of an array.

Parameters

a The input array.

v The output array which contains the sums of each row of array.

Description

This function calculates the sum of all elements in an array of any dimension. If the array is a two-dimensional matrix, the function can calculate the sum of each row. The input array can be any arithmetic data type, and of any dimension.

Example1

Calculate the sum of all elements of arrays of different data types.

```
#include <numeric.h>
int main() {
    double a[3] = {1,2,3};
    double b[2][3] = {1,2,3,4,5,6};
    int b1[2][3] = {1,2,3,4,5,6};
    double c[2][3][5];
    c[0][0][0] = 10;
    c[0][0][1] = 20;
    double sums;

    sums = sum(a);
    printf("sum(a) = %f\n", sums);
    sums = sum(b);
    printf("sum(b) = %f\n", sums);
    sums = sum(b1);
    printf("sum(b) = %f\n", sums);
    sums = sum(c);
    printf("sum(c) = %f\n", sums);
}
```

Output1

```
sum(a) = 6.000000
sum(b) = 21.000000
sum(b) = 21.000000
sum(c) = 30.000000
```

Example2

Calculate the sums of each elements of the arrays

```
#include <numeric.h>
int main() {
    double a[2][3] = {1,2,3,
                     4,5,6};
    array double b[3][4] = {1,2,3,4,
                          5,6,7,8,
                          1,2,3,4};
    array int    b1[3][4] = {1,2,3,4,
                          5,6,7,8,
                          1,2,3,4};
    array double sumv1[2], sumv2[3];
    double s;

    s = sum(a, sumv1);
    printf("sum(a) = %f\n", sumv1);
    sum(b, sumv2);
    printf("sum(b) = %f\n", sumv2);
    sum(b1, sumv2);
    printf("sum(b1) = %f\n", sumv2);
}
```

Output2

```
sum(a) = 6.000000 15.000000
sum(b) = 10.000000 26.000000 10.000000
sum(b1) = 10.000000 26.000000 10.000000
```

See Also

csum(), **cumsum()**, **cumprod()**, **trace()**, **product()**.

References

svd**Synopsis****#include** <numeric.h>**int** svd(array double complex *a*[[&]][&], array double *s*[], array double complex *u*[[&]][&],
array double complex *vt*[[&]][&]);**Purpose**

Compute singular value decomposition of a matrix.

Return Value

This function returns 0 on success and -1 on failure.

Parameters*a* Input matrix.*s* Output S vector.*u* Output U matrix.*vt* Output V matrix.**Description**

The function computes the singular value of a real /complex m-by-n matrix, and the left and/or right singular vectors. The SVD is formulated as

$$A = USV^T$$

Where S is an m-by-n matrix which is zero except for its min(m,n) diagonal elements, *U* is an m-by-m orthogonal matrix, and *V* is an n-by-n orthogonal matrix. The diagonal elements of S are the singular value of *A*; they are real, non-negative, and are in descending order. The first min(m,n) column of *U* and *V* are the left and right singular vectors of *A*.**Example1**

Singular value decomposition of a real n-by-n matrix.

```

#include <numeric.h>
int main() {
    array double a[3][3] = {7, 8, 1,
                           3, 6, 4,
                           3, 5, 7}; /* m-by-n matrix */
    array float b[3][3] = {7, 8, 1,
                          3, 6, 4,
                          3, 5, 7}; /* m-by-n matrix */

    int m = 3, n = 3;
    int lmn = min(m,n);
    array double s[lmn];
    array double u[m][m];
    array double vt[n][n];
    array float fs[lmn];
    array float fu[m][m];
    array float fvt[n][n];
    int status;

```

```

svd(a, s, NULL, NULL);
printf("s =\n%f\n", s);

svd(a, s, u, vt);
printf("u =\n%f\n", u);
printf("s =\n%f\n", s);
printf("vt =\n%f\n", vt);
printf("u*s*v^t =\n%f\n", u*diagonalmatrix(s)*transpose(vt));

status = svd(b, fs, fu, fvt);
if(status ==0) {
    printf("fu =\n%f\n", fu);
    printf("fs =\n%f\n", fs);
    printf("fvt =\n%f\n", fvt);
    printf("u*s*v^t =\n%f\n", fu*diagonalmatrix(fs)*transpose(fvt));
}
else
    printf("error: numerical error in svd()\n");
}

```

Output1

```

s =
15.071167 5.471953 0.957939

u =
-0.660443 0.704551 0.259659
-0.511773 -0.169318 -0.842271
-0.549458 -0.689158 0.472395

s =
15.071167 5.471953 0.957939

vt =
-0.517995 0.430638 0.739075
-0.736603 0.214678 -0.641349
-0.434853 -0.876621 0.206007

u*s*v^t =
7.000000 8.000000 1.000000
3.000000 6.000000 4.000000
3.000000 5.000000 7.000000

fu =
-0.660443 0.704551 0.259659
-0.511773 -0.169318 -0.842271
-0.549458 -0.689158 0.472395

fs =
15.071167 5.471953 0.957939

fvt =
-0.517995 0.430638 0.739075
-0.736603 0.214678 -0.641349
-0.434853 -0.876621 0.206007

u*s*v^t =
7.000000 8.000000 1.000000

```



```
3.000000 6.000000 4.000000
3.000000 5.000000 7.000000
```

Example2Singular value decomposition of a real m -by- n matrix ($m < n$).

```
#include <numeric.h>
int main() {
    array double a[2][3] = {7, 8, 1,
                           3, 6, 4}; /* m-by-n matrix */

    int m = 2, n = 3, i;
    int lmn = min(m,n);
    array double s[lmn], sm[m][n];
    array double u[m][m];
    array double vt[n][n];

    svd(a, s, u, vt);
    printf("u =\n%f\n", u);
    printf("s =\n%f\n", s);
    printf("vt =\n%f\n", vt);
    for(i=0; i<lmn; i++)
        sm[i][i] = s[i];
    printf("u*s*v^t =\n%f\n", u*sm*transpose(vt));
}
```

Output2

```
u =
-0.818910 -0.573922
-0.573922 0.818910

s =
12.851503 3.136698

vt =
-0.580020 -0.497570 0.644981
-0.777715 0.102681 -0.620174
-0.242353 0.861325 0.446525

u*s*v^t =
7.000000 8.000000 1.000000
3.000000 6.000000 4.000000
```

Example3Singular value decomposition of a real m -by- n matrix ($m > n$).

```
#include <numeric.h>
int main() {
    array double a[3][2] = {7, 8,
                           1, 3,
                           6, 4}; /* m-by-n matrix */

    int m = 3, n = 2, i;
    int lmn = min(m,n);
    array double s[lmn], sm[m][n];
    array double u[m][m];
    array double vt[n][n];
```

```

    svd(a, s, u, vt);
    printf("u =\n%f\n", u);
    printf("s =\n%f\n", s);
    printf("vt =\n%f\n", vt);
    for(i=0; i<lmn; i++)
        sm[i][i] = s[i];
    printf("u*s*v^t =\n%f\n", u*sm*transpose(vt));
}

```

Output3

```

u =
-0.812719 -0.288579 -0.506171
-0.217573 -0.655581 0.723102
-0.540508 0.697808 0.470016

```

```

s =
13.058084 2.118123

```

```

vt =
-0.700689 0.713467
-0.713467 -0.700689

```

```

u*s*v^t =
7.000000 8.000000
1.000000 3.000000
6.000000 4.000000

```

Example4

Singular value decomposition of a real n-by-n matrix with complex number.

```

#include <numeric.h>
int main() {
    int m = 3 , n = 3;
    int lmn = min(m,n);
    array double complex a[3][3] = {7, 8, 1,
                                     3, 6, 4,
                                     3, 5, 7} ; /* m-by-n matrix */
    array complex z[3][3] = {complex(1,2), 8, 1,
                             3, 6, 4,
                             3, 5, 7} ; /* m-by-n matrix */
    array double s[lmn];
    array double complex u[m][m];
    array double complex vt[n][n];
    int status;

    svd(a, s, NULL, NULL);
    printf("s =\n%f\n", s);

    svd(a, s, u, vt);
    printf("u =\n%f\n", u);
    printf("s =\n%f\n", s);
    printf("vt =\n%f\n", vt);
    printf("u*s*v^t =\n%f\n", u*diagonalmatrix(s)*transpose(vt));

    status = svd(z, s, u, vt);
    if(status ==0) {
        printf("u =\n%f\n", u);
    }
}

```

```

    printf("s =\n%f\n", s);
    printf("vt =\n%f\n", vt);
    printf("u*s*v^t =\n%f\n", u*diagonalmatrix(s)*transpose(vt));
}
else
    printf("error: numerical error in svd()\n");
}

```

Output4

```

s =
15.071167 5.471953 0.957939

u =
complex(-0.660443,0.000000) complex(0.704551,0.000000) complex(0.259659,0.000000)
complex(-0.511773,0.000000) complex(-0.169318,0.000000) complex(-0.842271,0.000000)
complex(-0.549458,0.000000) complex(-0.689158,0.000000) complex(0.472395,0.000000)

s =
15.071167 5.471953 0.957939

vt =
complex(-0.517995,-0.000000) complex(0.430638,-0.000000) complex(0.739075,0.000000)
complex(-0.736603,-0.000000) complex(0.214678,-0.000000) complex(-0.641349,-0.000000)
complex(-0.434853,-0.000000) complex(-0.876621,-0.000000) complex(0.206007,0.000000)

u*s*v^t =
complex(7.000000,0.000000) complex(8.000000,0.000000) complex(1.000000,0.000000)
complex(3.000000,0.000000) complex(6.000000,0.000000) complex(4.000000,0.000000)
complex(3.000000,0.000000) complex(5.000000,0.000000) complex(7.000000,0.000000)

u =
complex(-0.504577,-0.182672) complex(0.536523,-0.575026) complex(-0.247592,0.179497)
complex(-0.551746,-0.131934) complex(-0.031610,0.061460) complex(0.625115,-0.531624)
complex(-0.607942,-0.143559) complex(-0.372381,0.487898) complex(-0.372831,0.306730)

s =
13.678758 5.090474 0.989266

vt =
complex(-0.317937,-0.000000) complex(-0.358611,-0.000000) complex(0.877675,0.000000)
complex(-0.759338,-0.217182) complex(0.440159,-0.352023) complex(-0.095224,-0.222508)
complex(-0.509341,-0.125401) complex(-0.431508,0.606249) complex(-0.360819,0.202282)

```

```

u*s*v^t =
complex(1.000000,2.000000) complex(4.932809,1.184115) complex(3.850649,4.943457)
complex(3.000000,0.000000) complex(5.202331,3.116323) complex(3.380806,1.947931)
complex(3.000000,0.000000) complex(6.030656,5.110845) complex(3.373340,-0.361984)

```

Example5

Singular value decomposition of a real n-by-n singular matrix.

```

#include <numeric.h>
int main() {
    /* singular matrix */
    array float a[3][3] = {1, 2, 3,
                          4, 5, 6,
                          7, 8, 9};

    int m = 3, n = 3, i;
    int lmn = min(m,n);
    array double s[lmn], sm[m][n];
    array double u[m][m];
    array double vt[n][n];

    svd(a, s, u, vt);
    printf("u =\n%f\n", u);
    printf("s =\n%f\n", s);
    printf("vt =\n%f\n", vt);
    for(i=0; i<lmn; i++)
        sm[i][i] = s[i];
    printf("u*s*v^t =\n%f\n", u*sm*transpose(vt));
}

```

Output5

```

u =
-0.214837 0.887231 0.408248
-0.520587 0.249644 -0.816497
-0.826338 -0.387943 0.408248

s =
16.848103 1.068370 0.000000

vt =
-0.479671 -0.776691 -0.408248
-0.572368 -0.075686 0.816497
-0.665064 0.625318 -0.408248

u*s*v^t =
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000
7.000000 8.000000 9.000000

```

See Also

condnum(), rcondnum().

References

E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

trace

Synopsis

```
#include <numeric.h>
```

```
double trace(array double a[&][&]);
```

Purpose

Calculate the sum of diagonal elements of a matrix.

Return Value

This function returns the sum.

Parameters

a Input two-dimensional array.

Description

This function calculates the sum of the diagonal elements of the matrix.

Algorithm

For a matrix *a*, the trace is defined as

$$trace = \sum_{i=1}^n a_{ii}$$

Example

```
#include <numeric.h>
int main() {
    array double a[2][2] = {2, -4,
                           3, -7};
    array double a2[3][2] = {2, -4,
                             3, -7,
                             1, 1};
    array int b[2][2] = {2, -4,
                        3, -7};
    array int b2[3][2] = {2, -4,
                          3, -7,
                          1, 1};

    double t;

    t = trace(a);
    printf("trace(a) = %f\n", t);
    t = trace(a2);
    printf("trace(a2) = %f\n", t);
    t = trace(b);
    printf("trace(b) = %f\n", t);
    t = trace(b2);
    printf("trace(b2) = %f\n", t);
}
```

Output

```
trace(a) = -5.000000
trace(a2) = -5.000000
trace(b) = -5.000000
trace(b2) = -5.000000
```

See Also

ctrace(), **sum()**, **mean()**, **median()**.

References

triangularmatrix

Synopsis

```
#include <numeric.h>
```

```
double triangularmatrix(string_t pos, array double a[&][&], ... /* [intk] */)[:][:];
```

Syntax

```
triangularmatrix(pos, a)
```

```
triangularmatrix(pos, a, k)
```

Purpose

Get the upper or lower triangular part of a matrix.

Return Value

This function returns the upper or lower triangular part of a matrix.

Parameters

pos Input string indicating which part of the matrix to be returned.

a Input 2-dimensional array.

k Input integer. A matrix is returned on and below the *k*th diagonal of the input matrix. By default, *k* is 0.

Description

This function gets a triangular matrix of matrix *a*. For *pos* of "upper", the function returns the upper triangular part of matrix *a*, on and above the *k*th diagonal of the matrix. Optional argument *k* indicates the offset of the triangular matrix to the upper *k*th diagonal of the matrix. For *pos* of "lower" the function returns the lower part of matrix *a*, on and below the *k*th diagonal of the matrix. Optional argument *k* indicates the offset of the triangular matrix to the lower *k*th diagonal of the matrix.

Example

```
#include <numeric.h>
int main() {
    array double a[4][3] = {1,2,3,
                           4,5,6,
                           7,8,9,
                           6,3,5};

    int n = 4, m = 3, k;
    array double t[n][m];

    t = triangularmatrix("upper", a);
    printf("triangularmatrix(\"upper\", t) =\n%f\n", t);

    k = 0;
    t = triangularmatrix("upper", a, k);
    printf("triangularmatrix(\"upper\", t, 0) =\n%f\n", t);

    k = 1;
    t = triangularmatrix("upper", a, k);
    printf("triangularmatrix(\"upper\", t, 1) =\n%f\n", t);
}
```



```

k = -1;
t = triangularmatrix("upper",a,k);
printf("triangularmatrix(\"upper\", t, -1) =\n%f\n", t);

t = triangularmatrix("lower",a);
printf("triangularmatrix(\"lower\", t) =\n%f\n", t);

k = 0;
t = triangularmatrix("lower",a,k);
printf("triangularmatrix(\"lower\", t, 0) =\n%f\n", t);

k = 1;
t = triangularmatrix("lower",a, k);
printf("triangularmatrix(\"lower\", t, 1) =\n%f\n", t);

k = -1;
t = triangularmatrix("lower",a,k);
printf("triangularmatrix(\"lower\", t, -1) =\n%f\n", t);
}

```

Output

```

triangularmatrix("upper", t) =
1.000000 2.000000 3.000000
0.000000 5.000000 6.000000
0.000000 0.000000 9.000000
0.000000 0.000000 0.000000

triangularmatrix("upper", t, 0) =
1.000000 2.000000 3.000000
0.000000 5.000000 6.000000
0.000000 0.000000 9.000000
0.000000 0.000000 0.000000

triangularmatrix("upper", t, 1) =
0.000000 2.000000 3.000000
0.000000 0.000000 6.000000
0.000000 0.000000 0.000000
0.000000 0.000000 0.000000

triangularmatrix("upper", t, -1) =
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000
0.000000 8.000000 9.000000
0.000000 0.000000 5.000000

triangularmatrix("lower", t) =
1.000000 0.000000 0.000000
4.000000 5.000000 0.000000
7.000000 8.000000 9.000000
6.000000 3.000000 5.000000

triangularmatrix("lower", t, 0) =
1.000000 0.000000 0.000000
4.000000 5.000000 0.000000
7.000000 8.000000 9.000000
6.000000 3.000000 5.000000

```

```
triangularmatrix("lower", t, 1) =  
1.000000 2.000000 0.000000  
4.000000 5.000000 6.000000  
7.000000 8.000000 9.000000  
6.000000 3.000000 5.000000
```

```
triangularmatrix("lower", t, -1) =  
0.000000 0.000000 0.000000  
4.000000 0.000000 0.000000  
7.000000 8.000000 0.000000  
6.000000 3.000000 5.000000
```

See Also

ctriangularmatrix(), diagonal().

unwrap

Synopsis

```
#include <numeric.h>
```

```
int unwrap(array double &y, array double &x, ... /* [ double cutoff] */);
```

Syntax

```
unwrap(y, x)
```

```
unwrap(y, x, cutoff)
```

Purpose

Unwrap radian phase of each element of input array x by changing its absolute jump greater than π to its $2 * \pi$ complement.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x A vector or two-dimensional array which contains the original radian phase data.

y An output array which is the same dimension and size as x . It contains the unwrapped data.

$cutoff$ An optional input data which specifies the jump value. If the user does not specify this input, $cutoff$ has a value of π by default.

Description

The input array x can be of a vector or a two-dimensional array. If it is a two-dimensional array, the function unwraps it through every row of the array.

Example

In motion analysis of a crank-rocker mechanism, the output range of the rocker is within $0 \sim 2\pi$. For a given mechanism, the output may be as shown on the top part in Figure 11.4. There is a jump when θ_4 is π , because $\theta_4 = \pi$ and $\theta_4 = -\pi$ are the same point for the crank-rocker mechanism. If the `unwrap()` function is used, a smooth curve for output angle θ_4 can be obtained as shown on the lower part in Figure 11.4.

```
#include <numeric.h>
#include <complex.h>
#include <chplot.h>

int main(){
    double r[1:4], theta1, theta31;
    int n1=2, n2=4, i;
    double complex z, p, rb;
    double x1, x2, x3, x4;
    array double theta2[36], theta4[36], theta41[36];
    class CPlot subplot, *plot;

    /* four-bar linkage*/
    r[1]=5; r[2]=1.5; r[3]=3.5; r[4]=4;
    theta1=30*M_PI/180;
    linspace(theta2, 0, 2*M_PI);
```

```

for (i=0;i<36;i++) {
    z=polar(r[1],theta1)-polar(r[2],theta2[i]);
    complexsolve(n1,n2,r[3],-r[4],z,x1,x2,x3,x4);

    theta4[i] = x2;
}
unwrap(theta41, theta4);
subplot.subplot(2,1);
plot = subplot.getSubplot(0,0);
plot->data2D(theta2, theta4);
plot->title("Wraped");
plot->label(PLOT_AXIS_X,"Crank input: radians");
plot->label(PLOT_AXIS_Y,"Rocker output: radians");

plot = subplot.getSubplot(1,0);
plot->data2D(theta2, theta41);
plot->title("Unwraped");
plot->label(PLOT_AXIS_X,"Crank input: radians");
plot->label(PLOT_AXIS_Y,"Rocker output: radians");
subplot.plotting();
}

```

Output

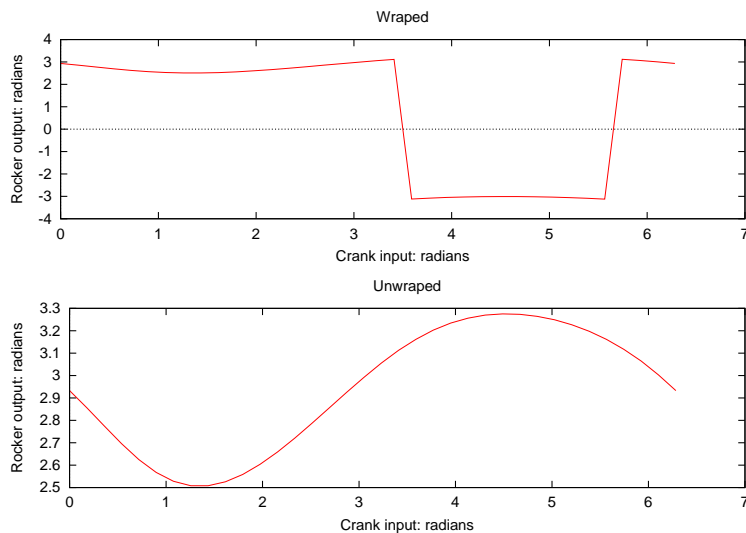


Figure 11.4: Comparison of results with and without using `unwrap()` function.

See Also

`unwrapdeg()`.

References

H. H. Cheng, *Computational Kinematics*, 1999.

unwrapdeg

Synopsis

```
#include <numeric.h>
```

```
int unwrapdeg(array double &y, array double &x, ... /* [ double cutoff] */);
```

Syntax

```
unwrapdeg(y, x)
```

```
unwrapdeg(y, x, cutoff)
```

Purpose

Unwrap degree phase of each element of input array *x* by changing its absolute jump greater than 180 to its $2 * 180$ complement.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x A vector or two-dimensional array which contains the original degree phase data.

y An output array which is the same dimension and size as *x*. It contains the unwrapped data.

cutoff An optional input data which specifies the jump value. If the user does not specify this input, *cutoff* has a value of 180 by default.

Description

The input array *x* can be of a vector or a two-dimensional array. If it is a two-dimensional array, the function unwrapdeg it through every row of the array.

See Also

unwrap().

urand

Synopsis

```
#include <numeric.h>
double urand(array double &x);
```

Syntax

```
urand(&x)
urand(NULL)
```

Purpose

Uniform random-number generator.

Return Value

This function returns a **double** uniform random-number.

Parameter

x An array of any dimensional size. It contains the uniform random numbers. The return value is the first element of array. If the argument of **urand()** is *NULL*, it only returns a uniform random number.

Description

Function **urand()** uses a random-number generator with period 2^{32} that returns successive pseudo-random numbers in the range from 0 to 1.

Example

The following example generates a two-dimensional array of uniform random number and a single uniform random number.

```
#include <stdio.h>
#include <numeric.h>

int main() {
    array double x[3][2];
    double u;

    u = urand(x);
    printf("urand(x) = %f\n", u);
    printf("x of urand(x) = \n%f\n", x);
    printf("urand(NULL) = %f\n", urand(NULL));
}
```

Output

```
urand(x) = 0.513871
x of urand(x) =
0.513871 0.175726
0.308634 0.534532
0.947630 0.171728

urand(NULL) = 0.702231
```

vertex

Synopsis

```
#include <numeric.h>
```

```
void vertex(double a, double b, double c, double &x, double &y);
```

Purpose

Calculate the vertex (x, y) of the second order polynomial $y = ax^2 + bx + c$.

Return Value

None.

Parameters

a The coefficient *a* of $y = ax^2 + bx + c$.

b The coefficient *b* of $y = ax^2 + bx + c$.

c The coefficient *c* of $y = ax^2 + bx + c$.

x The x coordinate of the vertex for $y = ax^2 + bx + c$.

y The y coordinate of the vertex for $y = ax^2 + bx + c$.

Description

This function calculates the vertex (x, y) of the second order polynomial $y = ax^2 + bx + c$. The x coordinate of the vertex is $-b/2a$. The vertex is passed back to the calling function through the fourth and fifth arguments of the function by reference.

Example

```
/* File: vertex.ch */
#include <numeric.h>
int main() {
    /* y = x*x -5x +6 */
    double a=1, b=-5, c = 6;
    double x, y;

    vertex(a, b, c, x, y);
    printf("vertex (x, y) = (%f, %f)\n", x, y);
}
```

Output

```
vertex (x, y) = (2.500000, -0.250000)
```

See Also

[cquadratic\(\)](#), [quadratic\(\)](#).

XCORR**Synopsis**

#include <numeric.h>

int xcorr(array double complex c[&], array double complex x[&], array double complex y[&]);

Syntax

xcorr(c, x, y)

Purpose

One-dimensional cross-correlation function estimation.

Return Value

This function returns 0 on success and -1 on failure.

Parameters*c* A one-dimensional **array** of size $(n+m-1)$. It contains the result of a cross-correlation function estimate of *x* and *y*.*x* A one-dimensional **array** of size *n*. It contains data used for correlation.*y* A one-dimensional **array** of size *m*. It contains data used for correlation.**Description**The input **array** *x* and *y* can be of any supported arithmetic data type and size. Conversion of the data to **double complex** is performed internally. If both *x* and *y* are **real** data, the result is a one-dimensional **real array** *c* of size $(n+m-1)$. If either one of *x* or *y* is **complex**, then the result **array** *c* will be of **complex** data type.**Algorithm**Given two functions $x(t)$ and $y(t)$, and their corresponding Fourier transforms $X(f)$ and $Y(f)$, the cross correlation of these two functions, denoted as $xcorr(x, y)$, is defined by

$$xcorr(x, y) \equiv \int_{-\infty}^{\infty} x(t + \tau)y(t)d\tau$$

$$xcorr(x, y) \iff X(f)Y^*(f)$$

The asterisk denotes a complex conjugate. The correlation of a function with itself is called its autocorrelation. In this case

$$xcorr(x, x) \iff |X(f)|^2$$

In discrete correlation of two sampled sequences x_k and y_k , each periodic with period N , is defined by

$$xcorr(x, y)_j \equiv \sum_{k=0}^{N-1} x_{j+k}y_k$$

The discrete correlation theorem says that this discrete correlation of two real sequences x and y is one member of the discrete Fourier transform pair

$$xcorr(x, y)_j \iff X_k Y_k^*$$

where X_k and Y_k^* are discrete Fourier transforms of x_k and y_k , and the asterisk denotes complex conjugation. This theorem makes the same presumptions about the functions as those encountered for the discrete convolution theorem. The correlation can be computed using the DFT as follows: DFT the two data sequences, multiplying one resulting transform by the complex conjugate of the other, and inverse transforming the product, the result is the correlation. Just as in the case of convolution, time-aliasing is handled by padding with zeroes.

In the numerical implementation, the sizes of two data sequences, **array** x of size n and y of size m , are expanded to $(m+n-1)$ and padded with zeroes internally. The FFT algorithm is used to compute the discrete Fourier transforms of x and y first. Multiplying the complex conjugate of transform of y with the transform of x component by component. Then using the inverse FFT algorithm to take the inverse transform of the products, the result is the cross correlation $xcorr(x, y)$ for arrays x and y .

Example

For two sequences $x[m]$ and $y[n]$, analytically, the cross correlation of x and y can be calculated by

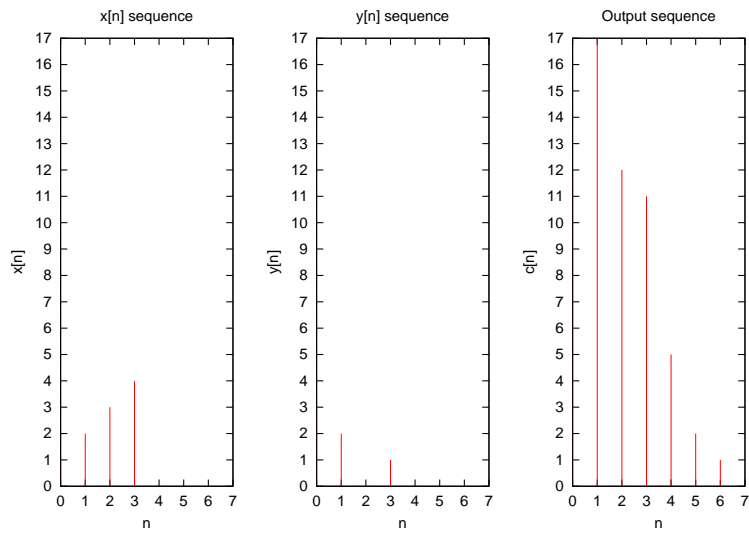
$$c[k] = \sum_{j=\max\{1, k-n+1\}}^{\min\{k, m\}} y[j]x[n+j-k]; \quad k = 1, 2, \dots, n+m-1$$

Given $x = \{1, 2, 3, 4\}$ and $y = \{3, 2, 0, 1\}$, then

$$\begin{aligned} c[1] &= y[1]x[4] = 12 \\ c[2] &= y[1]x[3] + y[2]x[4] = 17 \\ c[3] &= y[1]x[2] + y[2]x[3] + y[3]x[4] = 12 \\ c[4] &= y[1]x[1] + y[2]x[2] + y[3]x[3] + y[4]x[4] = 11 \\ c[5] &= y[2]x[1] + y[3]x[2] + y[4]x[3] = 5 \\ c[6] &= y[3]x[1] + y[4]x[2] = 2 \\ c[7] &= y[4]x[1] = 1 \end{aligned}$$

Output

```
x= 1.000  2.000  3.000  4.000
y= 3.000  2.000  0.000  1.000
c=12.000 17.000 12.000 11.000  5.000  2.000  1.000
```

**See Also**

`fft()`, `ifft()`, `conv()`, `conv2()`, `deconv()`.

References

William H. Press, et al, *Numerical Recipes in C*, second edition, Cambridge University Press, 1997.

Chapter 12

Non-local jumps <setjmp.h>

The header **setjmp.h** defines the macro **setjmp**, and declares one function and one type, for bypassing the normal function call and return discipline.

The type declared is

jmp_buf

which is an array type suitable for holding the information needed to restore a calling environment.

The macro defined is

setjmp

which saves the current environment.

Functions

The following functions are defined by the **setjmp.h** header file.

Function	Description
longjmp	Restores the environment saved by the most recent invocation of the setjmp macro.

Macros

The following macros are defined by the **setjmp.h** header file.

Macro	Description
setjmp	Saves its calling environment in its jmp_buf argument.

Declared Types

The following functions are defined by the **setjmp.h** header file.

Type	Description
jmp_buf	array - holds information needed to restore a calling environment.

Portability

This header has no known portability problem.

setjmp

Synopsis

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

Purpose

Saves environment.

Return Value

If the return is from a direct invocation, the **setjmp** macro returns the value zero. If the return is from a call to the **longjmp** function, the **setjmp** macro returns a nonzero value.

An invocation of the **setjmp** macro shall appear only in one of the following contexts:

- The entire controlling expression of a selection or iteration statement
- The operand of a relational or equality operator with the other operand an integer constant expression, with the resulting expression being the entire controlling expression of a selection or iteration statement.
- The operand of a unary ! operator with the resulting expression being the entire controlling expression of selection or iteration statement; or
- The entire expression of an expression statement (possibly cast to **void**).

Parameters

env Argument.

Description

The **setjmp** macro saves its calling environment in its **jmp_buf** argument for later use by the **longjmp** function.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>

jmp_buf env;
int i = 0;
main ()
{
    void exit();

    if(setjmp(env) != 0) {
        int k;

        printf("value of i on 2nd return from setjmp: %d\n", i);
        i = 2;
        k = setjmp(env);
        printf("k = %d\n", k);
        if(k==0) {
            printf("reset setjmp() \n");
        }
    }
}
```

```
        restore_env();
    }
    else {
        printf("value of i on 3rd return from setjmp: %d\n", i);
        exit(0);
    }
}
(void) printf("value of i on 1st return from setjmp: %d\n", i);
i = 1;
restore_env();
/* NOTREACHED */
}
int restore_env()
{
    longjmp(env, 1);
    /* NOTREACHED */
    return 0;
}
```

Output

```
value of i on 1st return from setjmp: 0
value of i on 2nd return from setjmp: 1
k = 0
reset setjmp()
k = 1
value of i on 3rd return from setjmp: 2
```

See Also

longjmp().

longjmp

Synopsis

```
#include <setjmp.h>
void longjmp(jmp_buf env, int val);
```

Purpose

Restores environment.

Return Value

After **longjmp** is completed, program execution continues as if the corresponding invocation of the **setjmp** macro had just returned the value specified by *val*. The **longjmp** function cannot cause the **setjmp** macro to return the value 0; if *val* is 0, the **setjmp** macro returns the value 1.

Parameters

env Argument.

val Argument.

Description

The **longjmp** function restores the environment saved by the most recent invocation of the **setjmp** macro in the same invocation of the program with the corresponding **jmp_buf** argument. If there has been no such invocation, or if the function containing the invocation of the **setjmp** macros has terminated execution in the interim, or if the invocation of the **setjmp** macro was within the scope of an identifier with variably modified type and execution has left that scope in the interim, the behavior is undefined.

All accessible objects have values as of the time **longjmp** was called, except that the values of objects of automatic storage duration that are local to the function containing the invocation of the corresponding **setjmp** macro that do not have volatile-qualified type and have been changed between the **setjmp** invocation and the **longjmp** call are indeterminate.

Example

see **setjmp()**.

See Also

setjmp().

Chapter 13

Signal handling <signal.h>

The header **signal.h** declares a type and two functions and defines several macros, for handling various *signals* (conditions that may be reported during program execution).

The type defined is

sig_atomic_t

which is the (possibly volatile-qualified) integer type of an object that can be accessed as an atomic entity even in the presence of asynchronous interrupts.

The macros defined are

SIG_DFL

SIG_ERR

SIG_IGN

which expand to constant expressions with distinct values that are type compatible with the second argument to, and the return value of, the **signal** function, and whose values compare unequal to the address of any declarable function; and each of the following, which expand to positive integer constant expressions with type **int** and distinct values that are the signal numbers, correspond to the specified condition:

SIGABRT abnormal termination, such as is initiated by the abort function

SIGFPE an erroneous arithmetic operation, such as zero divide or an operation resulting in overflow

SIGILL detection of an invalid function image, such as an invalid instruction

SIGINT receipt of an interactive attention signal

SIGSEGV an invalid access to storage

SIGTERM a termination request sent to the program

Ch may not generate any of these signals, except as a result of explicit calls to the **raise** function. Additional signals and pointers to undeclarable functions, with macro definitions beginning, respectively, with the letters **SIG** and an uppercase letter or with **SIG_** and an uppercase letter may also be specified by the different platform. implementation. The complete set of signals, their semantics, and their default handling is platform-dependent; all signal numbers shall be positive.

Functions

The following functions are defined by the **signal.h** header file.

Function	Description
signal()	Determines in which way the signal number sig will be handled.

Macros

The following macros and several other macros starting with **SIG** are defined by the **signal.h** header file.

Macro	Description
SIG_DFL	Specify use of default signal handling.
SIG_ERR	Return value of function signal() when it has an error.
SIG_IGN	Ignore the signal.

Declared Types

The following types are defined by the **signal.h** header file.

Type	Description
sig_atomic_t	int - object that can be accessed as an atomic entity.

Portability

This header has no known portability problem.

signal

Synopsis

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

Purpose

Chooses which way to receive a signal.

Return Value

If the request can be honored, the **signal** function returns the value of *func* for the most recent successful call to **signal** for the specified signal *sig*. Otherwise, a value of **SIG_ERR** is returned and a positive value is stored in **errno**.

Parameters

sig Signal number.

func signal handler.

Description

The signal function chooses one of three ways in which receipt of the signal number *sig* is to be subsequently handled. If the value of *func* is **SIG_DFL** default handling for that signal will occur. If the value of *func* is **SIG_IGN**, the signal will be ignored. Otherwise, *func* shall point to a function to be called when that signal occurs. An invocation of such a function because of a signal, or (recursively) of any further functions called by that invocation (other than functions in the standard library), is called a *signal handler*.

When a signal occurs and *func* points to a function, it is implementation-defined whether the equivalent of **signal(sig, SIG_DFL)**; is executed or the implementation prevents some implementation-defined set of signals (at least including *sig*) from occurring until the current signal handling has completed; in the case of **SIGILL**, the implementation may alternatively define that no action is taken. Then the equivalent of **(*func)(sig)**; is executed. If and when the function returns, if the value of *sig* is **SIGFPE SIGILL SIGSEGV** or any other implementation-defined value corresponding to a computational exception, the behavior is undefined; otherwise the program will resume execution at the point it was interrupted.

If the signal occurs as the result of calling the **abort** or **raise** function, the signal handler shall not call the **raise** function.

If the signal occurs other than as the result of calling the **abort** or **raise** function, the behavior is undefined if the signal handler refers to any object with static storage duration other than by assigning a value to an object declared as **volatile sig_atomic_t**, or the signal handler calls any function in the standard library other than the **abort** function or the **signal** function with the first argument equal to the signal number corresponding to the signal that caused the invocation of the handler. Furthermore, if such a call to the **signal** function results in a **SIG_ERR** return, the value of **errno** is indeterminate.

At program startup, the equivalent of **signal(sig, SIG_IGN)**;

may be executed for some signals selected in an implementation-defined manner; the equivalent of **signal(sig, SIG_DFL)**;

is executed for all other signals defined by the implementation.

The implementation shall behave as if no library function calls the **signal** function.

Example

```

/* sample program to show how signal() is used */
#include <stdio.h>
#include <signal.h>
int delay = 1;
void childHandler();

int main()
{
    int pid;
    char *argv[4];

    argv[0]=strdup("ls");
    argv[1]=strdup("-s");
    argv[2]=strdup("signal.c");
    argv[3]=NULL;
    signal(SIGCHLD, childHandler);
    pid=fork();
    if(pid==0)
    {
        execvp("ls",argv);
        perror("limit");
    }
    else
    {
        sleep(delay);
        printf("Child %d exceeded limit and is being killed \n",pid);
        kill(pid, SIGINT);
    }
}

void childHandler()
{
    int childPid, childStatus;
    childPid=wait(&childStatus);
    printf("Child %d terminated within %d seconds \n", childPid, delay);
    exit(0);
}

```

Output

```

2 signal.c
Child 9211 terminated within 1 seconds

```

See Also
raise().

raise

Synopsis

```
#include <signal.h>
```

```
int raise(int sig);
```

Purpose

Sends the signal specified by the *sig* to the executing program.

Return Value

The **raise** function returns zero if successful, nonzero if unsuccessful.

Parameters

sig Signal number.

Description

The **raise** function carries out the actions described by the signal function. If a signal handler is called, the **raise** function shall not return until after the signal handler does.

Example

```
#include<stdio.h>
#include<signal.h>

void func(int sig) {
    printf("sig in func() = %d\n", sig);
}

int main() {
    void (*signal (int sig, void (*disp)(int)))(int);
    void (*fptr)(int);

    fptr = signal(SIGTERM, func);
    printf("fptr = %p\n", fptr);
    raise(SIGTERM);
}
```

Output

```
fptr = 0
sig in func() = 15
```

See Also

signal().

Chapter 14

Variable Argument Lists — `<stdarg.h>`

The header `stdarg.h` declares a type and defines eight macros, for advancing through a list of arguments whose number and types are not known to the called function when it is parsed.

A function may be called with a variable number of arguments of varying types. Its parameter list contains one or more parameters. The rightmost parameter plays a special role in the access mechanism, and will be designated *parmN* in this description.

The declared type `va_list` is an object type suitable for holding information needed by the macros `va_start`, `va_arg`, `va_end`, `va_copy`, `va_count`, `va_datatype`, `va_arraydim`, `va_arrayextent`, `va_arraynum`, `va_arraytype`, and `va_tagname`. If access to the varying arguments is desired, the called function shall declare an object (referred to as *ap* in this subclause) having type `va_list`. The object *ap* may be passed as an argument to another function; if that function invokes the `va_arg` macro with parameter *ap*, the value of *ap* in the calling function is indeterminate and shall be passed to the `va_end` macro prior to any further reference to *ap*.

Variable argument list access macros

The `va_start`, `va_arg`, and `va_copy` macros described in this subclause shall be implemented as macros, not functions. The `va_end` is an identifier declared with external linkage. If a program defines an external identifier with the name `va_end`, the behavior is undefined. Each invocation of the `va_start` or `va_copy` macros shall be matched by a corresponding invocation of the `va_end` macro in the function accepting a varying number of arguments.

Macros

The following macros are defined by the `stdarg.h` header file.

Macro	Description
<code>VA_NOARG</code>	Second argument for <code>va_start()</code> , if no argument is passed to function
<code>va_arg</code>	Expands to an expression that has the specified type and the value of the next argument in the calling function.
<code>va_arraydim</code>	Obtain the dimension of an array of variable arguments.
<code>va_arrayextent</code>	Obtain the number of elements for a dimension of the array of variable argument.
<code>va_arraynum</code>	Obtain the number of elements of the array of variable argument.
<code>va_arraytype</code>	Obtain the array type of the array of the variable argument.
<code>va_copy</code>	Makes a copy of the <code>va_list</code> .

va_count	Obtain the number of variable arguments.
va_datatype	Obtain the data type of a variable argument.
va_dim	Obsolete, use va_arraydim .
va_elementtype	Obsolete, use va_datatype .
va_end	Facilitates a normal return from the function.
va_extent	Obsolete, use va_arrayextent .
va_start	Initializes <i>ap</i> for subsequent use by va_arg and va_end .
va_tagname	Obtain the tag name for struct/class/union argument of the variable argument.

Declared Types

The following types are defined by the **stdarg.h** header file.

Type	Description
va_list	object - holds information needed by functions with the prefix va_ .

Macros

The data type **ChType_t** defined inside the header file **ch.h** has the following values.

Value	Description
CH_UNDEFINETYPE	Undefined type.
CH_CHARTYPE	char type.
CH_UCHARTYPE	unsigned char type.
CH_SHORTTYPE	short int type.
CH_USHORTTYPE	unsigned short int type.
CH_INTTYPE	int/long int type.
CH_UINTTYPE	unsigned int type.
CH_LLINTTYPE	long long int type.
CH_ULLINTTYPE	unsigned long long int type.
CH_FLOATTYPE	float type.
CH_DOUBLETTYPE	double type.
CH_LDOUBLETTYPE	long double type.
CH_COMPLEXTYPE	float complex type.
CH_LCOMPLEXTYPE	long complex type.
CH_STRINGTYPE	string_t type.
CH_FILETYPE	FILE type.
CH_VOIDTYPE	void type.
CH_PROCTYPE	function type.
CH_STRUCTTYPE	struct type.
CH_CLASSTYPE	class type.
CH_UNIONTYPE	union type.
CH_ENUMTYPE	enum type.

CH_CARRAYTYPE	C array type.
CH_CARRAYPTRTYPE	pointer to C array.
CH_CARRAYVLATYPE	C VLA array.
CH_CHARRAYTYPE	Ch array type.
CH_CHARRAYPTRTYPE	pointer to Ch array.
CH_CHARRAYVLATYPE	Ch VLA array.

CH_NULLTYPE	NULL pointer type.
CH_VOIDPTRTYPE	pointer to void type.
CH_CHARPTRTYPE	pointer to char type.
CH_UCHARPTRTYPE	pointer to unsigned char type.
CH_SHORTPTRTYPE	pointer to short int type.
CH_USHORTPTRTYPE	pointer to unsigned short int type.
CH_INTPTRTYPE	pointer to int/long int type.
CH_UINTPTRTYPE	pointer to unsigned int type.
CH_LLINTPTRTYPE	pointer to long long int type.
CH_ULLINTPTRTYPE	pointer to unsigned long long int type.
CH_FLOATPTRTYPE	pointer to float type.
CH_DOUBLEPTRTYPE	pointer to double type.
CH_LDOUBLEPTRTYPE	pointer to long double type.
CH_COMPLEXPTRTYPE	pointer to float complex type.
CH_LCOMPLEXPTRTYPE	pointer to long complex type.
CH_STRINGPTRTYPE	pointer to string_t type.
CH_PROCPTRTYPE	function pointer type.
CH_FILEPTRTYPE	pointer to FILE type.
CH_STRUCTPTRTYPE	pointer to struct type.
CH_CLASSPTRTYPE	pointer to class type.
CH_UNIONPTRTYPE	pointer to union type.
CH_ENUMPTRTYPE	pointer to enum type.

CH_VOIDPTR2TYPE	pointer to pointer to void type.
CH_CHARPTR2TYPE	pointer to pointer to char type.
CH_UCHARPTR2TYPE	pointer to pointer to unsigned char type.
CH_SHORTPTR2TYPE	pointer to pointer to short int type.
CH_USHORTPTR2TYPE	pointer to pointer to unsigned short int type.
CH_INTPTR2TYPE	pointer to pointer to int/long int type.
CH_UINTPTR2TYPE	pointer to pointer to unsigned int type.
CH_LLINTPTR2TYPE	pointer to pointer to long long int type.
CH_ULLINTPTR2TYPE	pointer to pointer to unsigned long long int type.
CH_FLOATPTR2TYPE	pointer to pointer to float type.
CH_DOUBLEPTR2TYPE	pointer to pointer to double type.
CH_LDOUBLEPTR2TYPE	pointer to pointer to long double type.
CH_COMPLEXPTR2TYPE	pointer to pointer to float complex type.
CH_LCOMPLEXPTR2TYPE	pointer to pointer to long complex type.
CH_STRINGPTR2TYPE	pointer to pointer to string_t type.
CH_FILEPTR2TYPE	pointer to pointer to FILE type.
CH_STRUCTPTR2TYPE	pointer to pointer to struct type.
CH_CLASSPTR2TYPE	pointer to pointer to class type.

CH_UNIONPTR2TYPE pointer to pointer to union type.
CH_ENUMPTR2TYPE pointer to pointer to enum type.

Functions

Function	Description
arraycopy()	Copy arrays of of possible different types.

Portability

This header has no known portability problem.

arraycopy

Synopsis

```
#include <stdarg.h>
```

```
int arraycopy(void *des, CType_t destype, void *src, CType_t srctype, int n);
```

Purpose

Copy array with *n* elements of possible different types from *src* to *des*.

Return Value

If successful, this function returns 0; Otherwise, non-zero value is returned.

Parameters

des The address for the array to be copied to.

destype The data type for the array to be copied to. The valid data type is defined in header file **stdarg.h**.

src The address for the array to be copied from.

srctype The data type for the array to be copied from. The valid data type is defined in header file **stdarg.h**.

n The number of elements for the array.

Description

The function **arraycopy()** copies the array with the address of *src* of data type *srctype* to the array with the address of *des* of data type *destype*. It is assumed that both destination and source arrays have the same number of elements, although the data types of these two arrays can be different. The data type of an array passed from a variable argument can be determined by function **va_datatype()**. If the data type *destype* or *srctype* is pointer type, it will be treated as its corresponding non-pointer type.

Example

Function file `lindata.chf`

```
/* Copyright (c) 2001-2015 by SoftIntegration, Inc. All Rights Reserved */
#include <stdarg.h>
#include <stdio.h>
int lindata(double first, double last, ... /* [[array] type a[:]...[:]] [type *p, int n] */)
{
    va_list ap;
    int i, n;
    CType_t dtype;
    double step;
    void *vptr;

    va_start(ap, last);
    dtype = va_datatype(ap);
    if(va_arraytype(ap)) {
        n = va_arraynum(ap);
        vptr = va_arg(ap, void*);
    }
    else {
        dtype = va_datatype(ap);
```

```

    vptr = va_arg(ap, void*);
    n = va_arg(ap, int);
}

double a[n];
step = (last - first)/(n-1);
for(i=0; i<n; i++) {
    a[i]=first+i*step;
}

arraycopy(vptr, dtype, a, CH_DOUBLETTYPE, n);
return n;
}

```

Program arraycopy.ch

```

#include <numeric.h>

int main () {
    int i, a[6], *p;
    array double b[6];

    lndata(2, 12, a);
    printf("a = ");
    for(i=0; i<6; i++) {
        printf("%d ", a[i]);
    }
    p = &a[0];
    lndata(20, 120, p, 6);
    printf("\na = ");
    for(i=0; i<6; i++) {
        printf("%d ", a[i]);
    }
    printf("\nb = ");
    lndata(2, 12, b);
    printf("%g", b);
}

```

Output

```

a = 2 4 6 8 10 12
a = 20 40 60 80 100 120
b = 2 4 6 8 10 12

```

See Also

va_start(), **va_arg()**, **va_count()**, **va_datatype()**, **va_arraytype()**.

va_arg

Synopsis

```
#include <stdarg.h>
type va_arg(va_list ap, type);
```

Purpose

Expand a value in the variable argument list.

Return Value

The first invocation of the **va_arg** macro after that of the **va_start** macro returns the value of the argument after that specified by *parmN*. Successive invocations return the values of the remaining arguments in succession.

Parameters

ap An object having type **va_list**.

type The data type of passed variable argument.

Description

The **va_arg** macro expands to an expression that has the specified type and the value of the next argument in the call. The parameter *ap* shall be the same as the **va_list** *ap* initialized by **va_start**. Each invocation of **va_arg** modifies *ap* so that the values of successive arguments are returned in turn. The parameter *type* shall be a type name specified such that the type of a pointer to an object that has the specified type can be obtained simply by postfixing a * to *type*. If there is no actual next argument, or if *type* is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), the behavior is undefined.

Example

See **va_start()**.

See Also

va_start(), **va_end()**.

va_arraydim

Synopsis

```
#include <stdarg.h>
int va_arraydim(va_list ap);
```

Purpose

Obtain the dimension of an array which is a variable argument.

Return Value

The `va_arraydim` macro returns the dimension of an array.

Parameters

ap An object having type `va_list`.

Description

The `va_arraydim` macro returns the dimension of an array which is a variable argument.

Example

See `va_start()`.

See Also

`va_count()`, `va_datatype()`, `va_arrayextent()`, `va_arraynum()`, `va_arraytype()`.

va_arrayextent

Synopsis

```
#include <stdarg.h>
```

```
int va_arrayextent(va_list ap, int i);
```

Purpose

Obtain the number of elements in the specified dimension of a variable argument.

Return Value

The **va_arrayextent** macro returns the number of elements in the specified dimension of a variable argument.

Parameters

ap An object having type **va_list**.

i An integer specifying in which dimension the number of elements will be obtained.

Description

The **va_arrayextent** macro returns the number of elements in the specified dimension of a variable argument.

Example

See **va_start()**.

See Also

va_count(), **va_datatype()**, **va_arraydim()**, **va_arraynum()**, **va_arraytype()**.

va_arraynum

Synopsis

```
#include <stdarg.h>
int va_arraynum(va_list ap);
```

Purpose

Obtain the number of elements of an array passed from a variable argument.

Return Value

The **va_arraynum** macro returns the number of elements of an array passed from a variable argument. If the passed argument is not of array type, it returns 0.

Parameters

ap An object having type **va_list**.

Description

The **va_arraynum** macro returns the number of elements of an array passed from a variable argument.

Example

See **va_start()**.

See Also

va_count(), **va_datatype()**, **va_arraydim()**, **va_arrayextent()**, **va_arraytype()**.

va_arraytype

Synopsis

```
#include <stdarg.h>
```

```
ChType_t va_arraytype(va_list ap);
```

Purpose

Determine if a passed argument from a variable argument is an array and its array type.

Return Value

Based on its argument, this function returns one of the macros below, defined in header file **stdarg.h**.

Macro	Description	Example
CH_UNDEFINETYPE	not an array.	int i
CH_CARRAYTYPE	C array	int a[3]
CH_CARRAYPTRTYPE	pointer to C array	int (*ap)[3]
CH_CARRAYVLATYPE	C VLA array	int a[n]
CH_CHARRAYTYPE	Ch array	int func(int a[n], int b[:], int c[&]) array int a[3]
CH_CHARRAYPTRTYPE	pointer to Ch array	array int (*ap)[3]
CH_CHARRAYVLATYPE	Ch VLA array	array int a[n]; int fun(array int a[n], array int b[:], array int c[&])

Parameters

ap An object having type **va_list**.

Description

The function **va_arraytype()** determines if a passed argument is array type. The data type of the array shall be determined by function **va_datatype()**.

Example

See **va_start()**.

```
#include <stdarg.h>
#include <array.h>

void fun (int k, ...) {
    int i, ii, vacount;
    va_list ap;
    char *c;
    int *a;
    double (*b)[6];

    va_start(ap, k);
    vacount = va_count(ap);
    for(i = 0; i < vacount; i++) {
        if(va_arraytype(ap) == CH_CARRAYTYPE) {
            if(va_datatype(ap) == CH_CHARTYPE) {
                c = va_arg(ap, char *);
                printf("c = %s\n", c);
            }
            else if(va_datatype(ap) == CH_INTTYPE) {
                a = va_arg(ap, int *);
                printf("a[0] = %d\n", a[0]);
            }
        }
    }
}
```

```
        }
    }
    else if(va_arraytype(ap)==CH_CHARRAYTYPE) {
        b = (double (*)[6])va_arg(ap, double *);
        printf("b[0][0] = %f\n", b[0][0]);
    }
    else
        ii = va_arg(ap, int);
}
va_end(ap);
}

int main() {
    int i;
    char c[10] = "hello";
    int a[4] = {8, 2, 3, 4};
    array double b[5][6] = { 10, 2};

    fun(i, c);
    fun(i, a);
    fun(i, b, a, 10);
}
```

Output

```
c = hello
a[0] = 8
b[0][0] = 10.000000
a[0] = 8
```

See Also

va_count(), **va_datatype()**, **va_arraydim()**, **va_arrayextent()**, **va_arraynum()**.

va_copy

Synopsis

```
#include <stdarg.h>
```

```
void va_copy(va_list dest, va_list src);
```

Purpose

Makes **va_list** *src* a copy of **va_list** *dest*.

Return Value

The **va_copy** macro returns no value.

Parameters

dest A pointer to destination object.

src A pointer to source object.

Description

The **va_copy** macro makes the **va_list** *dest* be a copy of the **va_list** *src*, as if the **va_start** macro had been applied to it followed by the same sequence of uses of the **va_arg** macro as had previously been used to reach the present state of *src*.

Example

```
#include <stdarg.h>

void fun (int i, ...) {
    int i, j, k;;
    float f;
    va_list ap, ap2;

    va_start(ap, i);
    j = va_arg(ap, int);
    printf("j = %d\n", j);
    va_copy(ap2, ap);

    k = va_arg(ap, int);
    printf("k = %d\n", k);
    f = va_arg(ap, float);
    printf("f = %f\n", f);
    va_end(ap);

    /* use ap2 copied from ap */
    k = va_arg(ap2, int);
    printf("k = %d\n", k);
    f = va_arg(ap2, float);
    printf("f = %f\n", f);
    va_end(ap2);
}

int main() {
    fun(1, 2, 3, 4.0);
}
```

Output

```
j = 2
k = 3
f = 4.000000
k = 3
f = 4.000000
```

See Also

`va_arg()`, `va_start()`, `va_end()`.

va_count

Synopsis

```
#include <stdarg.h>
int va_count(va_list ap);
```

Purpose

Obtain the number of variable arguments.

Return Value

The **va_count** macro returns the number of variable arguments.

Parameters

ap An object having type **va_list**.

Description

The **va_count** macro returns the number of variable arguments.

Example

See **va_start()**.

See Also

va_dim(), **va_extent()**, **va_elementtype()**.

va_datatype

Synopsis

```
#include <stdarg.h>
```

```
ChType_t va_datatype(va_list ap);
```

Purpose

Obtain the data type of a variable argument.

Return Value

The **va_datatype** macro returns the data type of a variable argument.

Parameters

ap An object having type **va_list**.

Description

The **va_datatype** macro returns the data type of a variable argument. If a passed argument is of array type, the function will return the type of the element of the array.

Example

See **va_start()**.

See Also

va_count(), **va_arraytype()**,

va_dim

Synopsis

```
#include <stdarg.h>  
int va_dim(va_list ap);
```

Purpose

Obtain the dimension of an array which is a variable argument.

Return Value

The **va_dim** macro returns the dimension of the array.

Parameters

ap An object having type **va_list**.

Description

The **va_dim** macro returns the dimension of an array which is a variable argument.

Note

va_dim() is obsolete. Use **va_arraydim()**.

Example

See **va_start()**.

See Also

va_arraydim().

va_elementtype

Synopsis

```
#include <stdarg.h>
int va_elementtype(va_list ap);
```

Purpose

Obtain the data type of a variable argument.

Return Value

The `va_elementtype` macro returns the data type of a variable argument.

Parameters

ap An object having type `va_list`.

Description

The `va_elementtype` macro returns the data type of a variable argument.

Note

`va_elementtype()` is obsolete. Use `va_datatype()`.

Example

See `va_start()`.

See Also

`va_count()`, `va_dim()`, `va_extent()`.

va_end

Synopsis

```
#include <stdarg.h>
void va_end(va_list ap);
```

Purpose

Creates a normal return from a function.

Return Value

The **va_end** macro returns no value.

Parameters

ap An object having type **va_list**.

Description

The **va_end** macro facilitates a normal return from the function whose variable argument list was referred to by the expansion of **va_start** that initialized the **va_list** *ap*. The **va_end** macro may modify *ap* so that it is no longer usable (without an intervening invocation of **va_start**). If there is no corresponding invocation of the **va_start** macro, or if the **va_end** macro is not invoked before the return, the behavior is undefined.

Example

See **va_start()**.

See Also

va_start(), **va_arg()**.

va_extent

Synopsis

```
#include <stdarg.h>
```

```
int va_extent(va_list ap, int i);
```

Purpose

Obtain the number of elements in the specified dimension of a variable argument.

Return Value

The **va_extent** macro returns the number of elements in the specified dimension of a variable argument.

Parameters

ap An object having type **va_list**.

i An integer specifying in which dimension the number of elements will be obtained.

Description

The **va_extent** macro returns the number of elements in the specified dimension of a variable argument.

Note

va_extent() is obsolete. Use **va_arrayextent()**.

Example

See **va_start()**.

See Also

va_arrayextent().

va_start

Synopsis

```
#include <stdarg.h>
void va_start(va_list ap, parmN);
```

Purpose

Initializes *ap*.

Return Value

The **va_start** macro returns no value.

Parameters

ap An object having type **va_list**.

parmN The identifier of the rightmost parameter in the variable parameter list in the function definition.

Description

The **va_start** macro shall be invoked before any access to the unnamed arguments.

The **va_start** macro initializes *ap* for subsequent use by **va_arg** and **va_end**. **va_start()** shall not be invoked again for the same *ap* without an intervening invocation of **va_end()** for the same *ap*.

The parameter *parmN* is the identifier of the rightmost parameter in the variable parameter list in the function definition (the one just before the , ...). If the parameter *parmN* can be declared with the **register** storage class, with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions. In case there is no parameter in the function, macro **VA_NOARG** shall be used.

Example

```
#include <stdarg.h>
#include <array.h>

void fun (int k, ...) {
    int m, n, vacount, num;
    va_list ap;

    va_start(ap, k);
    vacount = va_count(ap);
    printf("va_count(ap) = %d\n", vacount);
    if(va_arraytype(ap)==CH_CARRAYTYPE ||
        va_arraytype(ap)==CH_CHARRAYTYPE) {
        printf("va_arraydim(ap)= %d\n", va_arraydim(ap));
        num = va_arraynum(ap);
        printf("va_arraynum(ap)= %d\n", num);
        m = va_arrayextent(ap, 0);
        printf("va_arrayextent(ap, 0)= %d\n", m);
        if(va_arraydim(ap) > 1) {
            n = va_arrayextent(ap, 1);
            printf("va_arrayextent(ap, 1)= %d\n", n);
        }
    }
    if(va_datatype(ap) == CH_INTTYPE) {
        int a[num], *p;
        ChType_t dtype;
        void *vptr;
        dtype = va_datatype(ap);
        vptr = va_arg(ap, void *);
        printf("array element is int\n");
    }
}
```

```

    p = vptr;
    printf("p[0] = %d\n", p[0]);
    arraycopy(a, CH_INTTYPE, vptr, dtype, num);
    printf("a[0] = %d\n", a[0]);
}
else if(va_datatype(ap) == CH_DOUBLETYPE) {
    array double b[m][n];
    ChType_t dtype;
    void *vptr;
    dtype = va_datatype(ap);
    vptr = va_arg(ap, void *);
    printf("array element is double\n");
    arraycopy(&b[0][0], CH_DOUBLETYPE, vptr, dtype, num);
    printf("b = \n%f", b);
}
}
else if(va_datatype(ap) == CH_INTTYPE)
    printf("data type is int\n");
else if(va_datatype(ap) == CH_DOUBLETYPE)
    printf("data type is double\n");
else if(va_datatype(ap) == CH_INTPTRTYPE)
    printf("data type is pointer to int\n");
va_end(ap);
}

int main() {
    int i, a[4]={10, 20, 30}, *p;
    double b[2][3]={1, 2, 3, 4, 5, 6};

    p = &i;
    fun(i, a);
    fun(i, b,a);
    fun(i, p);
}

```

Output

```

va_count(ap) = 1
va_arraydim(ap)= 1
va_arraynum(ap)= 4
va_arrayextent(ap, 0)= 4
array element is int
p[0] = 10
a[0] = 10
va_count(ap) = 2
va_arraydim(ap)= 2
va_arraynum(ap)= 6
va_arrayextent(ap, 0)= 2
va_arrayextent(ap, 1)= 3
array element is double
b =
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000
va_count(ap) = 1
data type is pointer to int

```

See Also

va_arg(), **va_end()**.

va_tagname

Synopsis

```
#include <stdarg.h>
char va_tagname(va_list ap);
```

Purpose

Obtain the tag name of a variable argument of struct, class, or union type.

Return Value

The **va_tagname** macro returns the tag name of pointer to char type. If the argument is not struct, class, or union type, the macro returns NULL.

Parameters

ap An object having type **va_list**.

Description

The **va_tagname** macro returns the tag name of pointer to char type. if the argument is not struct, class, or union type. Otherwise, it returns NULL.

Example

Program va_tagname.ch

```
#include<stdio.h>
#include<stdarg.h>

struct tag1 {int i; int j;};
class tag2 {public: int i; int j;};
union tag3 {int i; double d;};

void f1(int i, ...) {
    va_list ap;
    struct tag1 s, *sp;
    class tag2 c, *cp;
    union tag3 u, *up;

    va_start(ap, arg_num);
    if(va_datatype(ap) == CH_STRUCTTYPE) {
        if(!strcmp(va_tagname(ap), "tag1"))
            s = va_arg(ap, struct tag1);
    }
    if(va_datatype(ap) == CH_STRUCTPTRTYPE) {
        if(!strcmp(va_tagname(ap), "tag1"))
            sp = va_arg(ap, struct tag1*);
    }

    if(va_datatype(ap) == CH_CLASSTYPE) {
        if(!strcmp(va_tagname(ap), "tag2"))
            c = va_arg(ap, class tag2);
    }
    if(va_datatype(ap) == CH_CLASSPTRTYPE) {
        if(!strcmp(va_tagname(ap), "tag2"))
            cp = va_arg(ap, class tag2*);
    }

    if(va_datatype(ap) == CH_UNIONTYPE) {
        if(!strcmp(va_tagname(ap), "tag3"))
```

```

        u = va_arg(ap, union tag3);
    }
    if(va_datatype(ap) == CH_UNIONPTRTYPE) {
        if(!strcmp(va_tagname(ap), "tag3"))
            up = va_arg(ap, union tag3*);
    }

    printf("i = %d\n", i);
    printf("s.i = %d\n", s.i);
    printf("s.j = %d\n", s.j);
    printf("sp->i = %d\n", sp->i);
    printf("sp->j = %d\n", sp->j);
    printf("c.i = %d\n", c.i);
    printf("c.j = %d\n", c.j);
    printf("cp->i = %d\n", cp->i);
    printf("cp->j = %d\n", cp->j);
    printf("u.d = %f\n", u.d);
    printf("up->d = %f\n", up->d);
    va_end(ap);
}

int main(){
    struct tag1 s = {1,2};
    class tag2 c = {3,4};
    union tag3 u;
    u.d = 5;

    fl(10, s, &s, c, &c, u, &u);
}

```

Output

```

i = 10
s.i = 1
s.j = 2
sp->i = 1
sp->j = 2
c.i = 3
c.j = 4
cp->i = 3
cp->j = 4
u.d = 5.000000
up->d = 5.000000

```

See Also

va_datatype().

Chapter 15

Boolean type and values <stdbool.h>

The header **stdbool.h** defines four macros.

The macro

bool

expands to **_Bool**.

The remaining three macros are suitable for use in **#if** preprocessing directives. They are

true

which expands to the integer constant 1,

false

which expands to the integer constant 0, and

__bool_true_false_are_defined

which expands to the decimal constant 1.

Notwithstanding the reserved identifiers, a program is permitted to undefine and perhaps then redefine the macros **bool**, **true**, and **false**.

Macros

The following macros are defined by the **stdbool.h** header file.

Macro	Description
bool	Expands to _Bool .
true	Expands to integer constant 1.
false	Expands to integer constant 0.
__bool_true_false_are_defined	Expands to decimal point constant 1.

Chapter 16

Common Definitions <stddef.h>

The following types and macros are defined in the standard header **stddef.h**. Some are also defined in other headers, as noted in their respective subclauses.

The types are

ptrdiff_t

which is the signed integer type of the result of subtracting two pointers;

size_t

which is the unsigned integer type of the result of the **sizeof** operator; and

wchar_t

which is an integer type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales; the null character shall have the code value zero and each member of the basic character set shall have a code value equal to its value when used as the lone character in an integer character constant.

The macros are

offsetof(*type*, *member-designator*)

which expands to an integer constant expression that has type **size_t**, the value of which is the offset in bytes, to the structure member (designated by member-designator), from the beginning of its structure (designated by type). The type and member designator shall be such that given

static type t;

then the expression **&(t.member-designator)** evaluates to an address constant. (If the specified member is a bit-field, the behavior is undefined.)

Macros

The following macros are defined by the **stddef.h** header file.

Macro	Description
offsetof	Expands to integer constant..

Declared Types

The following types are defined by the **stddef.h** header file.

Type	Description
ptrdiff_t	int - results from subtracting two pointers.
size_t	unsigned int - results from the sizeof operator.
wchar_t	int - range of values can represent distinct codes for all members of the largest extended character set specified by the supported locales.

Portability

This header has no known portability problem.

Chapter 17

Input/Output <stdio.h>

The header **stdio.h** declares three types, several macros, and many functions for performing input and output.

The types declared are **size_t** (described by **stddef.h**);

FILE

which is an object type capable of recording all the information needed to control a stream, including its file position indicator, a pointer to its associated buffer (if any), an *error indicator* that records whether a read/write error has occurred, and an *end-of-file indicator* that records whether the end of the file has been reached; and

fpos_t

which is an object type other than an array type capable of recording all the information needed to specify uniquely every position within a file.

The macros are

_IOFBF

_IOLBF

_IONBF

which expand to integer constant expressions with distinct values, suitable for use as the third argument to the **setvbuf** function;

BUFSIZ

which expands to an integer constant expression, which is the size of the buffer used by the **setbuf** function;

EOF

which expands to an integer constant expression, with type **int** and a negative value, that is returned by several functions to indicate *end-of-file*, that is, no more input from a stream;

FOPEN_MAX

which expands to an integer constant expression that is the minimum number of files that the implementation guarantees can be open simultaneously;

FILENAME_MAX

which expands to an integer constant expression that is the size needed for an array of **char** large enough to hold the longest file name string that the implementation guarantees can be opened;

L_tmpnam

which expands to an integer constant expression that is the size needed for an array of **char** large enough to hold a temporary file name string generated by the **tmpnam** function;

SEEK_CUR

SEEK_END

SEEK_SET

which expand to integer constant expressions with distinct values, suitable for use as the third argument to the **fseek** function;

TMP_MAX

which expands to an integer constant expression that is the minimum number of unique file names that can be generated by the **tmpnam** function;

stderr

stdin

stdout

which are expressions of type “pointer to **FILE**” that point to the **FILE** objects associated, respectively, with the standard error, input, and output streams.

The header **wchar.h** declares a number of functions useful for wide-character input and output. The wide-character input/output functions described in that subclause provide operations analogous to most of those described here, except that the fundamental units internal to the program are wide characters. The external representation (in the file) is a sequence of “generalized” multibyte characters.

The input/output functions are given the following collective terms:

- The *wide-character input functions* – those functions described that perform input into wide characters and wide strings: **fgetwc**, **fgetws**, **getwc**, **getwchar**, **fwscanf**, **wscanf**, **vwscanf**, and **vwscanf**.
- The *wide-character output functions* – those functions that perform output from wide characters and wide strings: **fputwc**, **fputws**, **putwc**, **putwchar**, **fwprintf**, **wprintf**, **vwprintf**, and **vwprintf**.
- The *wide-character input/output functions* – the union of the **ungetwc** function, the wide-character input functions, and the wide-character output functions.
- The *byte input/output functions* – those functions described in this subclause that perform input/output: **fgetc**, **fgets**, **fprintf**, **fputc**, **fputs**, **fread**, **fscanf**, **fwrite**, **getc**, **getchar**, **gets**, **printf**, **putc**, **putchar**, **puts**, **scanf**, **ungetc**, **vfprintf**, **vscanf**, **vprintf**, and **vscanf**.

Streams

Input and output, whether to or from physical devices such as terminals and tape drives, or whether to or from files supported on structured storage devices, are mapped into logical data *streams*, whose properties are more uniform than their various inputs and outputs. Two forms of mapping are supported, for *text streams* and for *binary streams*.

A text stream is an ordered sequence of characters composed into *lines*, each line consisting of zero or more characters plus a terminating new-line character. Whether the last line requires a terminating new-line character is platform-dependent. Characters may have to be added, altered, or deleted on input and output to conform to differing conventions for representing text in the host environment. Thus, there need not be a one-to-one correspondence between the characters in a stream and those in the external representation. Data read in from a text stream will necessarily compare equal to the data that were earlier written out to that stream only if: the data consist only of printing characters and the control characters horizontal tab and new-line; no new-line character is immediately preceded by space characters; and the last character is a new-line character. Whether space characters that are written out immediately before a new-line character appear when read in is platform-dependent.

A binary stream is an ordered sequence of characters that can transparently record internal data. Data read in from a binary stream shall compare equal to the data that were earlier written out to that stream, under the same implementation. Such a stream may, however, have a platform-dependent number of null characters appended to the end of the stream.

Each stream has an *orientation*. After a stream is associated with an external file, but before any operations are performed on it, the stream is without orientation. Once a wide-character input/output function has been applied to a stream without orientation, the stream becomes a *wide-oriented stream*. Similarly, once a byte input/output function has been applied to a stream without orientation, the stream becomes a *byte-oriented stream*. Only a call to the **freopen** function or the **fwide** function can otherwise alter the orientation of a stream. (A successful call to **freopen** removes any orientation.)

Byte input/output functions shall not be applied to a wide-oriented stream and wide-character input/output functions shall not be applied to a byte-oriented stream. The remaining stream operations do not affect, and are not affected by, a stream’s orientation, except for the following additional restrictions:

- Binary wide-oriented streams have the file-positioning restrictions ascribed to both text and binary streams.
- For wide-oriented streams, after a successful call to a file-positioning function that leaves the file position indicator prior to the end-of-file, a wide-character output function can overwrite a partial multibyte character; any file contents beyond the byte(s) written are henceforth indeterminate.

Each wide-oriented stream has an associated **mbstate_t** object that stores the current parse state of the stream. A successful call to **fgetpos** stores a representation of the value of this **mbstate_t** object as part of the value of the **fpos_t** object. A later successful call to **fsetpos** using the same stored **fpos_t** value restores the value of the associated **mbstate_t** object as well as the position within the controlled stream.

Environmental limits

Each platform supports text files with lines containing at least 254 characters, including the terminating new-line character. The value of the macro **BUFSIZ** shall be at least 256.

Files

A stream is associated with an external file (which may be a physical device) by *opening* a file, which may involve *creating* a new file. Creating an existing file causes its former contents to be discarded, if necessary. If a file can support positioning requests (such as a disk file, as opposed to a terminal), then a *file position indicator* associated with the stream is positioned at the start (character number zero) of the file, unless the file is opened with append mode in which case it is platform-dependent whether the file position indicator is initially positioned at the beginning or the end of the file. The file position indicator is maintained by subsequent reads, writes, and positioning requests, to facilitate an orderly progression through the file.

Binary files are not truncated, except as defined later. Whether a write on a text stream causes the associated file to be truncated beyond that point is platform-dependent.

When a stream is *unbuffered*, characters are intended to appear from the source or at the destination as soon as possible. Otherwise characters may be accumulated and transmitted to or from the host environment as a block. When a stream is *fully buffered*, characters are intended to be transmitted to or from the host environment as a block when a buffer is filled. When a stream is *line buffered*, characters are intended to be transmitted to or from the host environment as a block when a new-line character is encountered. Furthermore, characters are intended to be transmitted as a block to the host environment when a buffer is filled, when input is requested on an unbuffered stream, or when input is requested on a line buffered stream that requires the transmission of characters from the host environment. Support for these characteristics is platform-dependent, and may be affected via the **setbuf** and **setvbuf** functions.

A file may be disassociated from a controlling stream by *closing* the file. Output streams are flushed (any unwritten buffer contents are transmitted to the host environment) before the stream is disassociated from the file. The value of a pointer to a **FILE** object is indeterminate after the associated file is closed (including the standard text streams). Whether a file of zero length (on which no characters have been written by an output stream) actually exists is platform-dependent.

The file may be subsequently reopened, by the same or another program execution, and its contents reclaimed or modified (if it can be repositioned at its start). If the **main** function returns to its original caller, or if the **exit** function is called, all open files are closed (hence all output streams are flushed) before program termination. Other paths to program termination, such as calling the **abort** function, need not close all files properly.

The address of the **FILE** object used to control a stream may be significant; a copy of a **FILE** object need not serve in place of the original.

At program startup, three text streams are predefined and need not be opened explicitly – *standard input* (for reading conventional input), *standard output* (for writing conventional output), and *standard error* (for writing diagnostic output). As initially opened, the standard error stream is not fully buffered; the standard input and standard output streams are fully buffered if and only if the stream can be determined not to refer to an interactive device.

Functions that open additional (nontemporary) files require a *file name*, which is a string. The rules for composing valid file names are platform-dependent. Whether the same file can be simultaneously open multiple times is also platform-dependent.

Although both text and binary wide-oriented streams are conceptually sequences of wide characters, the external file associated with a wide-oriented stream is a sequence of multibyte characters, generalized as follows:

- Multibyte encodings within files may contain embedded null bytes (unlike multibyte encodings valid for use internal to the program).
- A file need not begin nor end in the initial shift state.

Moreover, the encodings used for multibyte characters may differ among files. Both the nature and choice of such encodings are platform-dependent.

The wide-character input functions read multibyte characters from the stream and convert them to wide characters as if they were read by successive calls to the **fgetcwc** function. Each conversion occurs as if by a call to the **mbrtowc** function, with the conversion state described by the stream's own **mbstate_t** object. The byte input functions read characters from the stream as if by successive calls to the **fgetc** function.

The wide-character output functions convert wide characters to multibyte characters and write them to the stream as if they were written by successive calls to the **fputcwc** function. Each conversion occurs as if by a call to the **wcrtomb** function, with the conversion state described by the stream's own **mbstate_t** object. The byte output functions write characters to the stream as if by successive calls to the **fputc** function.

In some cases, some of the byte input/output functions also perform conversions between multibyte characters and wide characters. These conversions also occur as if by calls to the **mbrtowc** and **wcrtomb** functions.

An *encoding error* occurs if the character sequence presented to the underlying **mbrtowc** function does not form a valid (generalized) multibyte character, or if the code value passed to the underlying **wcrtomb** does not correspond to a valid (generalized) multibyte character. The wide-character input/output functions and the byte input/output functions store the value of the macro **EILSEQ** in **errno** if and only if an encoding error occurs.

Environmental limits

The value of **FOPEN_MAX** shall be at least eight, including the three standard text streams.

Functions

The following functions are defined by the **stdio.h** header file.

Function	Description
clearerr	Clears the end-of-file and error indicators from a stream.
fclose	Flushes a stream and closes the associated file(s).
feof	Tests the end-of-file for a stream.
ferror	Tests the error indicator for a stream.
fflush	Flushes a file.
fgetc	Reads the next character from the input stream.
fgetpos	Stores values of file position indicators.
fgets	Reads $n-1$ characters from the input stream ending with a carriage return character.
fopen	Opens a file.
fprintf	Prints output to given stream.
fputc	Writes a character to the output stream.
fputs	Writes a string to the output stream.
fread	Reads from the stream into an array.
freopen	Opens a file.
fscanf	Reads input from a given stream.
fseek	Sets file position indicator for a stream.
fsetpos	Sets the mbstate_t object and the file position indicator for a stream.
ftell	Obtains the current value of the file position indicator for a stream.
fwrite	Writes from an array into a stream.
getc	Reads a character from the input stream.
getchar	Reads next character from string.
gets	Reads a character from the input stream.
perror	Maps the error number in errno to an error message.
printf	Prints output to the standard output.
putc	Writes a character to the output stream.
putchar	Writes a character to stdout .
puts	Writes a character to a string.
remove	Removes a file.
rename	Renames a file.
rewind	Sets the file position indicator to the beginning of a stream.
scanf	Reads input from the standard input.
setbuf	Sets buffer of a stream.
setvbuf	Sets buffer mode and size of a stream.
snprintf	Prints output to a specified array.
sprintf	Prints output to a specified array.
sscanf	Reads input from a string.
tmpfile	Creates a temporary binary file.
tmpnam	Creates a temporary file name.
ungetc	Pushes character back onto input stream.
vfprintf	Prints output to specified stream.
vprintf	Prints output to specified stream.

vsprintf Prints output to specified array.
vprintf Prints output to specified array.

Macros

The following macros are defined by the **stdio.h** header file.

Macro	Description
_IOFBF	Expands to an integer constant expression suitable for third argument of setvbuf .
_IOLBF	Expands to an integer constant expression suitable for third argument of setvbuf .
_IONBF	Expands to an integer constant expression suitable for third argument of setvbuf .
BUFSIZ	Expands to an integer constant expression, which is the size of the buffer used by setbuf .
EOF	Expands to an integer constant expression, with type int and a negative value, that is returned by several functions to indicate <i>end-of-file</i> .
FILENAME_MAX	Expands to an integer constant expression that is the size needed for an array of char large enough to hold the longest file name string that the implementation guarantees can be opened.
FOPEN_MAX	Expands to an integer constant expression that is the minimum number of files that the implementation guarantees can be open simultaneously.
L_tmpnam	Expands to an integer constant expression large enough to hold the temporary file name string generated by tmpnam .
SEEK_CUR	Expands to an integer constant expression with distinct value suitable for use in fseek .
SEEK_END	Expands to an integer constant expression with distinct value suitable for use in fseek .
SEEK_SET	Expands to an integer constant expression with distinct value suitable for use in fseek .
TMP_MAX	Expands to an integer constant expression that is the minimum number of file names that can be generated by tmpnam .
stderr	Points to file object associated with the standard error.
stdin	Points to file object associated with the standard input.
stdout	Points to file object associated with the standard output.

Declared Types

The following types are defined by the **stdio.h** header file.

Type	Description
size_t	unsigned int - results from the sizeof operator.
FILE	object - records all information needed to control a stream.
fpos_t	object - records all information needed to specify uniquely every position within a file.

Portability

The following functions are not supported in Ch: **vsscanf()**, **vfscanf()**, **vscanf()**.

clearerr

Synopsis

```
#include <stdio.h>
void clearerr(FILE *stream);
```

Purpose

Clear the end-of-file and error indicators from a stream.

Return Value

The **clearerr** function returns no value.

Parameters

stream Pointer to a stream.

Description

The **clearerr** function clears the end-of-file and error indicators for the stream pointed to by *stream*.

Example

```
/* clearerr() resets the error indicator and EOF
   to 0 on the same stream */
#include <stdio.h>

int main() {
    FILE *fp;
    char ch;

    if((fp = fopen("clearerr.c","r")) == NULL) {
        fprintf(stderr, "Error: cannot open file clearerr.c for reading\n");
        exit(1);
    }
    while(!feof(fp)) {
        ch = fgetc(fp);
    }
    printf("feof(fp) is %2d before clearerr(fp) is called.\n", feof(fp));
    clearerr(fp);
    printf("feof(fp) is %2d after clearerr(fp) is called.\n", feof(fp));
}
```

Output

```
feof(fp) is 16 before clearerr(fp) is called.
feof(fp) is 0 after clearerr(fp) is called.
```

See Also

fclose

Synopsis

```
#include <stdio.h>
int fclose(FILE *stream);
```

Purpose

Flushes a stream and closes the associated file(s).

Return Value

The **fclose** function returns zero if the stream was successfully closed, or **EOF** if any errors were detected.

Parameters

stream Pointer to a stream.

Description

The **fclose** function causes the stream pointed to by *stream* to be flushed and the associated file to be closed. Any unwritten buffered data for the stream are delivered to the host environment to be written to the file; and unread buffered data are discarded. The stream is disassociated from the file. If the associated buffer was automatically allocated, it is deallocated.

Example see **clearerr()**.

Output**See Also**

feof

Synopsis

```
#include <stdio.h>
```

```
int feof(FILE *stream);
```

Purpose

Test the end-of-file for a stream.

Return Value

The **feof** function returns nonzero if and only if the end-of-file indicator is set for *stream*.

Parameters

stream Pointer to a stream.

Description

The **feof** function tests the end-of-file indicator for the stream pointed to by *stream*.

Example see **clearerr()**.

Output**See Also**

ferror

Synopsis

```
#include <stdio.h>
```

```
int ferror(FILE *stream);
```

Purpose

Test the error indicator for a stream.

Return Value

The **ferror** function returns nonzero if and only if the end-of-file indicator is set for *stream*.

Parameters

stream Pointer to a stream.

Description

The **ferror** function tests the error indicator for the stream pointed to by *stream*.

Example see **clearerr()**.

Output**See Also**

fflush

Synopsis

```
#include <stdio.h>
int fflush(FILE *stream);
```

Purpose

Flush a file.

Return Value

The **fflush** function sets the error indicator for the stream and returns **EOF** if a write error occurs, otherwise, it returns zero.

Parameters

stream Pointer to a stream.

Description

If *stream* points to an output stream or an update stream in which the most recent operation was no input, the **fflush** function causes any unwritten data for that stream to be delivered to the host environment to be written to the file; otherwise, the behavior is undefined.

If *stream* is a null pointer, the **fflush** function performs this flushing action on all streams for which the behavior is defined above.

Example

see **fwrite()**.

Output**See Also**

fgetc

Synopsis

```
#include <stdio.h>
int fgetc(FILE *stream);
```

Purpose

Read input from stream.

Return Value

If the end-of-file indicator is set, or if the stream is at end-of-file, the end-of-file indicator for the stream is set and **fgetc** returns **EOF**. Otherwise, the **fgetc** function returns the next character from the input stream pointed to by *stream*. If a read error occurs, the error indicator for the stream is set and **fgetc** returns **EOF**.

Parameters

stream Pointer to a stream.

Description

If the end-of-file indicator for the input stream pointed to by *stream* is not set and a next character is present, the **fgetc** function obtains that character as an **unsigned char** converted to an **int** and advances the associated file position indicator for the stream (if defined).

Example

```
/* a sample program that reads a file then
displays it's content. */
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *stream;
    char ch;

    /* open file for reading only */
    if((stream=fopen("fgetc.c", "r")) == NULL){
        printf("cannot open file\n");
        exit(1);
    }
    ch = fgetc(stream);
    printf("%c\n", ch);
    fclose(stream);
}
```

Output

/

See Also

fgetpos

Synopsis

```
#include <stdio.h>
int fgetpos(FILE * restrict stream, fpos_t * resitric pos);
```

Purpose

Store current values of file position indicators.

Return Value

If successful, the **fgetpos** function returns 0; on failure, the **fgetpos** function returns nonzero and stores an implementation-defined positive value in **errno**.

Parameters

stream Pointer to a stream.

pos Pointer to an object.

Description

The **fgetpos** function stores the current values of the parse state (if any) and file position indicator for the stream pointed to by *stream* in the object pointed to by *pos*. The values stored contain unspecified information usable by the **fsetpos** function for repositioning the stream to its position at the time of the call to the **fgetpos** function.

Example

```
/* a sample program that uses the fgetpos() to read the fourth
   character from a file. */
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *stream;
    fpos_t pos;
    char ch;
    long int offset = 3;
    /* open file for reading */
    if((stream = fopen("fgetpos.c", "r")) == NULL){
        printf("cannot open file\n");
        exit(1);
    }
    fseek(stream, offset ,SEEK_SET);
    fgetpos(stream, &pos);
    fsetpos(stream, &pos);
    fread(&ch, sizeof(char), 1, stream); //read a character
    printf("The current character in file is '%c'\n", ch);
    fclose(stream);
}
```

Output

The current character in file is 'a'

See Also

fgets

Synopsis**#include** <stdio.h>**char *fgets**(**char *restrict** *s*, **int** *n*, **FILE *restrict** *stream*);**Purpose**

Read input from a stream into the memory.

Return Value

The **fgets** function returns *s* if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read error occurs during operation, the array contents are indeterminate and a null pointer is returned.

Parameters*s* Pointer to an array.*n* Pointer to an integer.*stream* Pointer to a stream.**Description**

The **fgets** function reads at most one less than the number of characters specified by *n* from the stream pointed to by *stream* into the array pointed to by *s*. No additional characters are read after a new-line character (which is retained) or after end-of-file. A null character is written immediately after the last character read into the array.

Example**Output****See Also**

fopen

Synopsis

```
#include <stdio.h>
```

```
FILE *fopen(const char *filename, const char *mode);
```

Purpose

Open a file.

Return Value

The **fopen** function returns a pointer to the object controlling the stream. If the open operation fails, **fopen** returns a null pointer.

Parameters

filename Pointer to a file.

mode Points to a string.

Description

The **fopen** function opens the file whose name is the string pointed to by *filename*, and associates a stream with it.

The argument *mode* points to a string. If the string is one of the following, the file is open in the indicated mode. Otherwise, the behavior is undefined.

r	open a text file for reading
w	truncate to zero length or create a text file for writing
a	append; open or create a text file for writing at end-of-file
rb	open a binary file for reading
wb	truncate to zero length or create a binary file for writing
ab	append; open or create a binary file for writing at end-of-file
r+	open a text file for update (reading and writing)
w+	truncate to zero length or create a text file for update
a+	append; open or create a text file for update, writing at-end of-file
r+b or rb+	open a binary file for update (reading and writing)
w+b or wb+	truncate to zero length or create a binary file for updating
a+b or ab+	append; open or create a binary file for update, writing at end-of-file

Opening a file with read mode (**'r'** as the first character in the mode argument) fails if the file does not exist or cannot be read.

Opening a file with append mode (**'a'** as the first character in the *mode* argument) causes all subsequent writes to the file to be forced to the then current end-of-file, regardless of intervening calls to the **fseek** function. In some implementations, opening a binary file with the append mode (**'b'** as the second or third character in the above list of *mode* argument values) may initially position the file position indicator for the stream beyond the last data written, because of null character padding.

When a file is opened with update mode (**'+'** as the second or third character in the above list of *mode* argument values), both input and output may be performed on the associated stream. However, output shall not be directly followed by input without an intervening call to the **fflush** function or to a file positioning function (**fseek**, **fsetpos**, or **rewind**), and input shall not be directly followed by output without an intervening call to a file positioning function, unless the input operation encounters end-of-file. Opening (or creating) a text file with update mode may instead open (or create) a binary stream in some implementations.

When opened, a stream is fully buffered if and only if it can be determined not to refer to an interactive device. The error and end-of-file indicators for the stream are cleared.

Example

see **clearerr()**.

Output**See Also**

fprintf

Synopsis

```
#include <stdio.h>
```

```
int fprintf(FILE *restrict stream, const char *restrict format, ...);
```

Purpose

Write output to a stream using the given format.

Return Value

The **fprintf** function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

Parameters

stream Pointer to a stream.

format Points to a character.

Description

The **fprintf** function writes output to the stream pointed to by *stream*, under control of the string pointed to by *format* that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored. The **fprintf** function returns when the end of the format string is encountered.

The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: ordinary multibyte characters (not %), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments, converting them, if applicable, according to the corresponding conversion specifier, and then writing the result to the output stream.

Each conversion specification is introduced by the character %. After the %, the following appear in sequence:

- Zero or more *flags* (in any order) that modify the meaning of the conversion specification.
- An optional minimum *field width*. If the converted value has fewer characters than the field width, it is padded with spaces (by default) on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The field width takes the form of an asterisk (described later) or a decimal integer.
- An optional precision that gives the minimum number of digits to appear for the **d**, **i**, **o**, **u**, **x**, and **X** conversions, the number of digits to appear after the decimal-point character for **a**, **A**, **e**, **E**, **f**, and **F** conversions, the maximum number of significant digits for the **g** and **G** conversions, or the maximum number of characters to be written from a string in **s** conversions. The precision takes the form of a period (.) followed either by an asterisk (described later) or by an optional decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.
- An optional length modifier that specifies the size of the argument.
- A conversion specifier character that specifies the type of conversion to be applied.

As noted above, a field width, or precision, or both, may be indicated by an asterisk. In this case, an **int** argument supplies the field width or precision. The arguments specifying field width, or precision, or both, shall appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a - flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

The flag characters and their meanings are:

- The result of the conversion is left-justified within the field. (It is right-justified if this flag is not specified).
- + The result of a signed conversion always begins with a plus or minus sign. (It begins with a sign only when a negative value is converted if this flag is not specified).
- space* If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space is prefixed to the result. If the *space* and + flags both appear, the *space* flag is ignored.
- # The result is converted to an “alternative form”. For **o** conversion, it increases the precision, if and only if necessary, to force the first digit of the result to be a zero (if the value and precision are both 0, a single 0 is printed). For **x** (or **X**) conversion, a nonzero result has **0x** (or **0X**) prefixed to it. For **a**, **A**, **e**, **E**, **f**, **F**, **g**, and **G** conversions, the result of converting a floating-point number always contains a decimal-point character, even if no digits follow it. (Normally, a decimal-point character appears in the result of these conversions only if a digit follows it.) For **g** and **G** conversions, trailing zeros are not removed from the result. For other conversions, the behavior is undefined.
- 0** For **d**, **i**, **o**, **u**, **x**, **X**, **a**, **A**, **e**, **E**, **f**, **F**, **g**, and **G** conversions, leading zeros (following any indication of sign or base) are used to pad to the field width rather than performing space padding, except when converting an infinity or NaN. If the **0** and - flags both appear, the **0** flag is ignored. For **d**, **i**, **o**, **u**, **x**, and **X** conversions, if a precision is specified, the **0** flag is ignored. For other conversions, the behavior is undefined.

The length modifiers and their meanings are:

- hh** Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **signed char** or **unsigned char** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **signed char** or **unsigned char** before printing); or that a following **n** conversion specifier applies to a pointer to a **signed char** argument.
- h** Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **short int** or **unsigned short int** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **short int** or **unsigned short int** before printing); or that a following **n** conversion specifier applies to a pointer to a **short int** argument.
- l** (ell) Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **long int** or **unsigned long int** argument; that a following **n** conversion specifier applies to a pointer to a **long int** argument; that a following **c** conversion specifier applies to a **wint_t** argument; that a following **s** conversion specifier applies to a pointer to a **wchar_t** argument; or has no effect on a following **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G** conversion specifier.
- ll** (ell-ell) Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **long long int** or **unsigned long long int** argument; or that a following **n** conversion specifier applies to a pointer to a **long long int** argument.
- j** Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to an **intmax_t** or **uintmax_t** argument; or that a following **n** conversion specifier applies to a pointer to an **intmax_t** argument.
- z** Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **size_t** or the corresponding signed integer type argument; or that a following **n** conversion specifier applies to a pointer to a signed integer type corresponding to a **size_t** argument.
- t** Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **ptrdiff_t** or the corresponding unsigned integer type argument; or that a following **n** conversion specifier applies to a pointer to a **ptrdiff_t** argument.
- L** Specifies that a following **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G** conversion specifier applies to a **long double** argument.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

The conversion specifiers and their meanings are:

- d,i** The **int** argument is converted to signed decimal in the style *[-]ddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.
- o,u,x,X** The **unsigned int** argument is converted to unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal notation (**x** or **X**) in the style *ddd*; the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.
- f,F** A **double** argument representing a floating-point number is converted to decimal notation in the style *[-]ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the **#** flag is not specified, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.
A **double** argument representing an infinity is converted in one of the styles *[-]inf* or *[-]infinity* – which style is implementation-defined. A **double** argument representing a NaN is converted in one of the styles *[-]nan* or *[-]nan(n-char-sequence)* – which style, and the meaning of any n-char- sequence, is implementation-defined. The **F** conversion specifier produces **INF**, **INFINITY**, or **NAN** instead of **inf**, **infinity**, or **nan**, respectively.
- e,E** A **double** argument representing a floating-point number is converted in the style *[-]d.ddde±dd*, where there is one digit (which is nonzero if the argument is nonzero) before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero and the **#** flag is not specified, no decimal-point character appears. The value is rounded to the appropriate number of digits. The **E** conversion specifier produces a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits, and only as many more digits as necessary to represent the exponent. If the value is zero, the exponent is zero.
A **double** argument representing an infinity or NaN is converted in the style of an **f** or **F** conversion specifier.
- g,G** A **double** argument representing a floating-point number is converted in style **f** or **e** (or in style **F** or **E** in the case of a **G** conversion specifier), with the precision specifying the number of significant digits. If the precision is zero, it is taken as 1. The style used depends on the value converted; style **e** (or **E**) is used only if the exponent resulting from such a conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result unless the **#** flag is specified; a decimal-point character appears only if it is followed by a digit.
A **double** argument representing an infinity or NaN is converted in the style of an **f** or **F** conversion specifier.
- a,A** A **double** argument representing a floating-point number is converted in the style *[-]0xh.hhhp±d*, where there is one hexadecimal digit (which is nonzero if the argument is a normalized floating- point number and is otherwise unspecified) before the decimal-point character²²¹ and the number of hexadecimal digits after it is equal to the precision; if the precision is missing and **FLT_RADIX** is a power of 2, then the precision is sufficient for an exact representation of the value; if the precision is missing and **FLT_RADIX** is not a power of 2, then the precision is sufficient to distinguish values of type **double**, except that trailing zeros may be omitted; if the precision is zero and the **#** flag is not specified, no decimal- point character appears. The letters **abcdef** are used for a conversion and the letters **ABCDEF** for **A** conversion. The **A** conversion specifier produces a number with **X** and **P** instead of **x** and **p**. The exponent always contains at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent is zero.
A **double** argument representing an infinity or NaN is converted in the style of an **f** or **F** conversion specifier.
- c** If no **l** length modifier is present, the **int** argument is converted to an **unsigned char**, and the resulting character is written.
If an **l** length modifier is present, the **wint_t** argument is converted as if by an **ls** conversion specification with no precision and an argument that points to the initial element of a two-element array of **wchar_t**, the first element containing the **wint_t** argument to the **lc** conversion specification and the second a null wide character.

- s** If no **l** length modifier is present, the argument shall be a pointer to the initial element of an array of character type. Characters from the array are written up to (but not including) the terminating null character. If the precision is specified, no more than that many characters are written. If the precision is not specified or is greater than the size of the array, the array shall contain a null character.
If an **l** length modifier is present, the argument shall be a pointer to the initial element of an array of **wchar_t** type. Wide characters from the array are converted to multibyte characters (each as if by a call to the **wcrtomb** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first wide character is converted) up to and including a terminating null wide character. The resulting multibyte characters are written up to (but not including) the terminating null character (byte). If no precision is specified, the array shall contain a null wide character. If a precision is specified, no more than that many characters (bytes) are written (including shift sequences, if any), and the array shall contain a null wide character if, to equal the multibyte character sequence length given by the precision, the function would need to access a wide character one past the end of the array. In no case is a partial multibyte character written.
- p** The argument shall be a pointer to **void**. The value of the pointer is converted to a sequence of printing characters, in an implementation-defined manner.
- n** The argument shall be a pointer to signed integer into which is *written* the number of characters written to the output stream so far by this call to **fprintf**. No argument is converted, but one is consumed. If the conversion specification includes any flags, a field width, or a precision, the behavior is undefined.
- %** A **%** character is written. No argument is converted. The complete conversion specification shall be **%%**.

If a conversion specification is invalid, the behavior is undefined. If any argument is not the correct type for the corresponding conversion specification, the behavior is undefined.

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

For **a** and **A** conversions, if **FLT_RADIX** is a power of 2, the value is correctly rounded to a hexadecimal floating number with the given precision.

If **FLT_RADIX** is not a power of 2, the result should be one of the two adjacent numbers in hexadecimal floating style with the given precision, with the extra stipulation that the error should have a correct sign for the current rounding direction.

For **e**, **E**, **f**, **F**, **g**, and **G** conversions, if the number of significant decimal digits is at most **DECIMAL_DIG**, then the result should be correctly rounded. If the number of significant decimal digits is more than **DECIMAL_DIG** but the source value is exactly representable with **DECIMAL_DIG** digits, then the result should be an exact representation with trailing zeros. Otherwise, the source value is bounded by two adjacent decimal strings $L < U$, both having **DECIMAL_DIG** significant digits; the value of the resultant decimal string D should satisfy $L \leq D \leq U$, with the extra stipulation that the error should have a correct sign for the current rounding direction.

Example
Output
See Also

fputc

Synopsis**#include** <stdio.h>**int** *fputc(int *c*, FILE **stream*);**Purpose**Print output to **stdout**.**Return Value**The **fputc** function returns the character written. If a write error occurs, the error indicator for the stream is set and **fputc** returns **EOF**.**Parameters***c* Pointer to an integer.*stream* Pointer to a stream.**Description**

The **fputc** function writes the character specified by *c* (converted to an **unsigned char**) to the output stream pointed to by *stream*, at the position indicated by the associated file position indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream.

Example

```
/* a sample program that opens a temporary file and
write a character and a string into it. */
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *stream;
    char ch='A';
    char *name;

    if((name = tmpnam(NULL)) == NULL)
        printf("Error in creating temp file.\n");
    /* open file for reading only */
    if((stream=fopen(name, "w")) == NULL) {
        printf("cannot open file\n");
        exit(1);
    }
    else {
        fputc(ch, stream);
        fputs("This is a test\n", stream);
        fputs("This is a test\n", stdout);
        fclose(stream);
    }
}
```

Output

This is a test

See Also

fputs

Synopsis**#include** <stdio.h>**int** *fputs(const char * restrict *s*, **FILE** * restrict *stream*);**Purpose**

Print output to a string.

Return ValueThe **fputs** function writes the string pointed to by *s* to the stream pointed to by *stream*. The terminating null character is not written.**Parameters***s* Pointer to a character.*stream* Pointer to a stream.**Description**The **fputs** function writes the string pointed to by *s* pointed to by *stream*. The terminating null character is not written.**Example**see **fputc()**.**Output****See Also****fputc()**

fread

Synopsis

#include <stdio.h>

int fread(void * restrict *ptr*, **size_t** *size*, **size_t** *nmemb*, **FILE** * restrict *stream*);

Purpose

Read input from string into an array.

Return Value

The **fread** function returns the number of elements successfully read, which may be less than *nmemb* if a read error or end-of-file is encountered. If *size* or *nmemb* is zero, *fread* returns zero and the contents of the array and the state of the stream remain unchanged.

Parameters

ptr Pointer.

size Argument.

nmemb Argument.

stream Pointer to a stream.

Description

The **fread** function reads, into the array pointed to by *ptr*, up to *nmemb* elements whose size is specified by *size*, from the stream pointed to by *stream*. The file position indicator for the stream (if defined) is advanced by the number of characters successfully read. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate. If a partial element is read, its value is indeterminate.

Example

Output

See Also

freopen

Synopsis

#include <stdio.h>

FILE *freopen(const char *filename, const char *mode, FILE * restrict stream);

Purpose

Open a file.

Return Value

The **freopen** function returns a null pointer if the open operation fails. Otherwise, **freopen** returns the value of *stream*.

Parameters

filename Pointer to a file.

mode Points to a string.

stream Points to a stream.

Description

The **freopen** function opens the file whose name is the string pointed to by *filename* and associates the stream pointed to by *stream* with it. The *mode* argument is used just as in the **fopen** function.

If the *filename* is a null pointer, the **freopen** function attempts to change the mode of the stream to that specified by *mode*, as if the name of the file currently associated with the stream had been used. It is implementation-defined which changes of mode are permitted (if any), and under what circumstances.

The **freopen** function first attempts to close any file that is associated with the specified stream. Failure to close the file is ignored. The error and end-of-file indicators for the stream are cleared.

Example

Output

See Also

fscanf

Synopsis

```
#include <stdio.h>
```

```
int fscanf(FILE *restrict stream, const char *restrict format, ...);
```

Purpose

Read the input from a stream using the given format.

Return Value

The **fscanf** function returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, the function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of any early matching failure.

Parameters

stream Pointer to a stream.

format Points to a character.

Description

The **fscanf** function reads input from the stream pointed to by *stream*, under control of the string pointed to by *format* that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored.

The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: one or more white-space characters, an ordinary multibyte character (neither a % nor a white-space character), or a conversion specification. Each conversion specification is introduced by the character

After the %, the following appear in sequence:

- An optional assignment-suppressing character *****.
- An optional nonzero decimal integer that specifies the maximum field width (in characters).
- An optional *length modifier* that specifies the size of the receiving object.
- A *conversion specifier* character that specifies the type of conversion to be applied.

The **fscanf** function executes each directive of the format in turn. If a directive fails, as detailed below, the function returns. Failures are described as input failures (due to the occurrence of an encoding error or the unavailability of input characters), or matching failures (due to inappropriate input).

A directive composed of white-space character(s) is executed by reading input up to the first non-white-space character (which remains unread), or until no more characters can be read.

A directive that is an ordinary multibyte character is executed by reading the next characters of the stream. If any of those characters differ from the ones composing the directive, the directive fails and the differing and subsequent characters remain unread.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

Input white-space characters (as specified by the **isspace** function) are skipped, unless the specification includes a **[**, **c**, or **n** specifier.

An input item is read from the stream, unless the specification includes an **n** specifier. An input item is defined as the longest sequence of input characters which does not exceed any specified field width and which is, or is a prefix of, a matching input sequence. The first character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails; this condition is a matching failure unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.

Except in the case of a % specifier, the input item (or, in the case of a %n directive, the count of input characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless assignment suppression was indicated by a *, the result of the conversion is placed in the object pointed to by the first argument following the *format* argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the object, the behavior is undefined.

The length modifiers and their meanings are:

- hh** Specifies that a following **d, i, o, u, x, X,** or **n** conversion specifier applies to an argument with type pointer to **signed char** or **unsigned char**.
- h** Specifies that a following **d, i, o, u, x, X,** or **n** conversion specifier applies to an argument with type pointer to **short int** or **unsigned short int**.
- l (ell)** Specifies that a following **d, i, o, u, x, X,** or **n** conversion specifier applies to an argument with type pointer to **long int** or **unsigned long int**; that a following **a, A, e, E, f, F, g,** or **G** conversion specifier applies to an argument with type pointer to **double**; or that a following **c, s,** or **[** conversion specifier applies to an argument with type pointer to **wchar_t**.
- ll (ell-ell)** Specifies that a following **d, i, o, u, x, X,** or **n** conversion specifier applies to an argument with type pointer to **long long int** or **unsigned long long int**.
- j** Specifies that a following **d, i, o, u, x, X,** or **n** conversion specifier applies to an argument with type pointer to **intmax_t** or **uintmax_t**.
- z** Specifies that a following **d, i, o, u, x, X,** or **n** conversion specifier applies to an argument with type pointer to **size_t** or the corresponding signed integer type.

- t** Specifies that a following **d, i, o, u, x, X,** or **n** conversion specifier applies to an argument with type pointer to **ptrdiff_t** or the corresponding unsigned integer type.
- L** Specifies that a following **a, A, e, E, f, F, g,** or **G** conversion specifier applies to an argument with type pointer to **long double**.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

The conversion specifiers and their meanings are:

- d** Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 10 for the **base** argument. The corresponding argument shall be a pointer to signed integer.
- i** Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 0 for the **base** argument. The corresponding argument shall be a pointer to signed integer.
- o** Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 8 for the **base** argument. The corresponding argument shall be a pointer to unsigned integer.
- u** Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 10 for the **base** argument. The corresponding argument shall be a pointer to unsigned integer.
- x** Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 16 for the **base** argument. The corresponding argument shall be a pointer to unsigned integer.
- a,e,f,g** Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected for the subject sequence of the **strtod** function. The corresponding argument shall be a pointer to floating-point number.
- c** Matches a sequence of characters of exactly the number specified by the field width (1 if no field width is present in the directive).
 If no **l** length modifier is present, the corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence. No null character is added.
 If an **l** length modifier is present, the input shall be a sequence of multibyte characters that begins in the initial shift state. Each multibyte character in the sequence is converted to a wide character as if by a call to the **mbrtowc** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first multibyte character is converted. The corresponding argument shall be a pointer to the initial element of an array of **wchar_t** large enough to accept the resulting sequence of wide characters. No null wide character is added.

- s** Matches a sequence of non-white-space characters.
If no **l** length modifier is present, the corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically.
If an **l** length modifier is present, the input shall be a sequence of multibyte characters that begins in the initial shift state. Each multibyte character is converted to a wide character as if by a call to the **mbrtowc** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first multibyte character is converted. The corresponding argument shall be a pointer to the initial element of an array of **wchar_t** large enough to accept the sequence and the terminating null wide character, which will be added automatically.
- [** Matches a nonempty sequence of characters from a set of expected characters (the scanset).
If no **l** length modifier is present, the corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically.
If an **l** length modifier is present, the input shall be a sequence of multibyte characters that begins in the initial shift state. Each multibyte character is converted to a wide character as if by a call to the **mbrtowc** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first multibyte character is converted. The corresponding argument shall be a pointer to the initial element of an array of **wchar_t** large enough to accept the sequence and the terminating null wide character, which will be added automatically.
The conversion specifier includes all subsequent characters in the **format** string, up to and including the matching right bracket (**]**). The characters between the brackets (the *scanlist*) compose the scanset, unless the character after the left bracket is a circumflex (**^**), in which case the scanset contains all characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with **[** or **[^**, the right bracket character is in the scanlist and the next following right bracket character is the matching right bracket that ends the specification; otherwise the first following right bracket character is the one that ends the specification. If a character is in the scanlist and is not the first, nor the second where the first character is a **^**, nor the last character, the behavior is implementation-defined.
- p** Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the **%p** conversion of the **fprintf** function. The corresponding argument shall be a pointer to a pointer to **void**. The input item is converted to a pointer value in an implementation-defined manner. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the **%p** conversion is undefined.
- n** No input is consumed. The corresponding argument shall be a pointer to signed integer into which is to be written the number of characters read from the input stream so far by this call to the **fscanf** function. Execution of a **%n** directive does not increment the assignment count returned at the completion of execution of the **fscanf** function. No argument is converted, but one is consumed. If the conversion specification includes an assignment-suppressing character or a field width, the behavior is undefined.
- %** Matches a single **%** character; no conversion or assignment occurs. The complete conversion specification shall be **%%**.

If a conversion specification is invalid, the behavior is undefined.

The conversion specifiers **A**, **E**, **F**, **G**, and **X** are also valid and behave the same as, respectively, **a**, **e**, **f**, **g**, and **x**.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (other than **%n**, if any) is terminated with an input failure.

Trailing white space (including new-line characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the **%n** directive.

If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream.

Example
Output
See Also

fseek

Synopsis

```
#include <stdio.h>
```

```
int fseek(FILE *stream, long int offset, int whence);
```

Purpose

Store current values of file position indicators.

Return Value

the **fseek** function returns nonzero only for a request that cannot be satisfied.

Parameters

stream Pointer to a stream.

offset Argument.

whence Argument.

Description

The **fseek** function sets the file position indicator for the stream pointed to by *stream*. If a read or write error occurs, the error indicator for the stream is set and **fseek** fails.

For a binary stream, the new position, measured in characters from the beginning of the file, is obtained by adding *offset* to the position specified by *whence*. The specified position is the beginning of the file if *whence* is **SEEK_SET**, the current value of the file position indicator if **SEEK_CUR**, or end-of-file if **SEEK_END**. A binary stream need not meaningfully support **fseek** calls with a *whence* value of **SEEK_END**.

For a text stream, either *offset* shall be zero, or *offset* shall be a value returned by an earlier successful call to the **ftell** function on a stream associated with the same file, and *whence* shall be **SEEK_SET**.

After determining the new position, a successful call to the **fseek** function undoes any effects of the **ungetc** function on the stream, clears the end-of-file indicator for the stream, and then establishes the new position. After a successful **fseek** call, the next operation on an update stream may be either input or output.

Example

```
/* a sample program that reads the third character
in a file. */
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *stream;
    char ch;
    long int offset;

    offset = 2*sizeof(char);
    /* open file for reading and writing */
    if((stream = fopen("fseek.c", "r")) == NULL){
        printf("cannot open file\n");
        exit(1);
    }
    fseek(stream,offset,0);    //search the third character
    fread(&ch,sizeof(char),1,stream);    //read a character
    printf("The third character in file is '%c'\n",ch);
    rewind(stream);    //set file indicator to first
    fread(&ch,sizeof(char),1,stream);
    printf("The first character in file is '%c'\n",ch);
}
```

```
    fclose(stream);  
}
```

Output

```
The third character in file is ' '  
The first character in file is '/'
```

See Also

fsetpos

Synopsis

```
#include <stdio.h>
```

```
int fsetpos(FILE * restrict stream, fpos_t * restrict pos);
```

Purpose

Store current values of file position indicators.

Return Value

If successful, the **fsetpos** function returns 0; on failure, the **fsetpos** function returns nonzero and stores an implementation-defined positive value in **errno**.

Parameters

stream Pointer to a stream.

pos Pointer to an object.

Description

The **fsetpos** function sets the **mbstate_t** object (if any) and file position indicator for the *stream* pointed to by *stream* according to the value of the object pointed to by *pos*, which shall be a value obtained from an earlier successful call to the **fgetpos** function on a stream associated with the same file. If a read or write error occurs, the error indicator for the stream is set and **fsetpos** fails.

A successful call to the **fsetpos** function undoes any effects of the **ungetc** function on the stream, clears the end-of-file indicator for the stream, and then establishes the new parse state and position. After a successful **fsetpos** call, the next operation on an update stream may be either input or output.

Example see **fgetpos()**.

Output

See Also

ftell

Synopsis

```
#include <stdio.h>
long int ftell(FILE *stream);
```

Purpose

Obtain the current value of the file position indicator for a stream.

Return Value

If successful, the **ftell** function returns the current value of the file position indicator for the stream. On failure, the **ftell** function returns -1L and stores an implementation-defined positive value in **errno**.

Parameters

stream Pointer to a stream.

Description

The **ftell** function obtains the current value of the file position indicator for the stream pointed to by *stream*. For a binary stream, the value is the number of characters from the beginning of the file. For a text stream, its file position indicator contains unspecified information, usable by the **fseek** function for returning the file position indicator for the stream to its position at the time of the **ftell** call; the difference between two such return values is not necessarily a meaningful measure of the number of characters written or read.

Example

```
/* a sample program that reads the third character
in a file. */
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *stream;
    char ch;
    int cur_indicator;
    long int offset = 2*sizeof(char);
    /* open file for reading and writing */
    if((stream = fopen("ftell.c", "r")) == NULL){
        printf("cannot open file\n");
        exit(1);
    }
    fseek(stream,offset,0);    //search the third character
    cur_indicator = ftell(stream);    // read current indicator
    fread(&ch,sizeof(char),1,stream); //read a character
    printf("The third character in file is '%c'\n",ch);
    fseek(stream,cur_indicator-1,0); // set the file indicator to
    // the number before the current
    fread(&ch,sizeof(char),1,stream); //read a character
    printf("The second character in file is '%c'\n",ch);
    rewind(stream);    //set file indicator to first
    fread(&ch,sizeof(char),1,stream);
    printf("The first character in file is '%c'\n",ch);
    fclose(stream);
}
```

Output

```
The third character in file is ' '
The second character in file is '*'
The first character in file is '/'
```

See Also

fwrite

Synopsis

#include <stdio.h>

int **fwrite**(**void** * *restrict ptr*, **size_t** *size*, **size_t** *nmemb*, **FILE** * *restrict stream*);

Purpose

Read input from string into an array.

Return Value

The **fwrite** function returns the number of elements successfully read, which may be less than *nmemb* if a read error or end-of-file is encountered. If *size* or *nmemb* is zero, *fwrite* returns zero and the contents of the array and the state of the stream remain unchanged.

Parameters

ptr Pointer.

size Argument.

nmemb Argument.

stream Pointer to a stream.

Description

The **fwrite** function writes, from the array pointed to by *ptr*, up to *nmemb* elements whose size is specified by *size*, from the stream pointed to by *stream*. The file position indicator for the stream (if defined) is advanced by the number of characters successfully written. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate.

Example

```
/* a sample program that flushes the buffer
after write operation. */
#include <stdio.h>
#include <stdlib.h>
#define SIZE 80

int main() {
    FILE *stream;
    char buf[SIZE];
    char *name;
    size_t num;

    if((name = tmpnam(NULL)) == NULL)
        printf("Error in creating temp file.\n");
    /* open file for reading and writing */
    if((stream = fopen(name,"w")) == NULL) {
        printf("cannot open file\n");
        exit(1);
    }
    strcpy(buf, "1111111111111111");
    num = fwrite(buf, sizeof(char), 10, stream);
    printf("Number of elements written to temp file: %d\n", num);
    fflush(stream);      // flush the buffer
    fclose(stream);
    remove(name);
}
```

Output

```
Number of elements written to temp file: 10
```

See Also

getchar

Synopsis

```
#include <stdio.h>
```

```
int getchar(void);
```

Purpose

Read next character from stream.

Return Value

The **getchar** function returns the next character from the input stream pointed to by **stdin**. If the stream is at end-of-file, the end-of-file indicator for the stream is set and **getchar** returns **EOF**. If a read error occurs, the error indicator for the stream is set and **getchar** returns **EOF**.

Parameters

No argument.

Description

The **getchar** function is equivalent to **getc** with the argument **stdin**.

Example**Output****See Also**

getc

Synopsis

```
#include <stdio.h>
int getc(FILE *stream);
```

Purpose

Read input from a stream.

Return Value

The **getc** function returns the next character from the input stream pointed to by *stream*. If the stream is at end-of-file, the end-of-file indicator for the stream is set and **getc** returns **EOF**. If a read error occurs, the error indicator for the stream is set and **getc** returns **EOF**.

Parameters

stream Pointer to a stream.

Description

The **getc** function is equivalent to **fgetc**, except that if it is implemented as a macro, it may evaluate *stream* more than once, so the argument should never be an expression with side effects.

Example

```
/* a sample program that flushes the buffer
after write operation. */
#include <stdio.h>
#include <stdlib.h>

int main(int argc , char *argv[]) {
    char choice;

    do {
        printf("1: Check spelling\n");
        printf("2: Correct spelling\n");
        printf("3: Lookup a word in the directory\n");
        printf("4: Quit\n");

        printf("\nEnter your selection: ");
        choice = getc(stdin);
    } while (!strchr("1234",choice));
    printf("You entered: %c\n", choice);
    return (0);
}
```

Output

```
1: Check spelling
2: Correct spelling
3: Lookup a word in the directory
4: Quit
```

```
Enter your selection: You entered: 4
```

See Also

gets

Synopsis

```
#include <stdio.h>
int *gets(char *s);
```

Purpose

Read characters from input streams into arrays.

Return Value

The **gets** function returns *s* if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read error occurs during operation, the array contents are indeterminate and a null pointer is returned.

Parameters

s Pointer to a character.

Description

The **gets** function reads characters from the input stream pointed to by *stdin* into the array pointed to by *s*, until end-of-file is encountered or a new-line character is read. Any new-line character is discarded, and a null character is written immediately after the last character read into the array.

Example**Output****See Also**

perror

Synopsis

```
#include <stdio.h>
void perror(const char *s);
```

Purpose

Map the error number in **errno** to an error message.

Return Value

The **perror** function returns no value.

Parameters

stream Pointer to a stream.

Description

The **perror** function maps the error number in the integer expression **errno** to an error message. It writes a sequence of characters to the standard error stream thus: first (if *s* is not a null pointer and the character pointed to by *s* is not the null character), the string pointed to by *s* followed by a colon (:), a space, then an appropriate error message string followed by a new-line character. The contents of the error message strings are the same as those returned by the **strerror** function with argument **errno**.

Example

```
/* Uses the perror function to display an error message. */
#include <stdio.h>
#include <signal.h>
int delay = 1;
void childHandler();

int main()
{
    int pid;
    char *argv[3];

    argv[0]="-1";
    signal(SIGCHLD, childHandler);
    pid=fork();
    if(pid==0)
    {
        execvp("i",argv);
        perror("Error:");
    }
    else
    {
        sleep(delay);
        printf("Child %d exceeded limit and is being killed \n",pid);
        kill(pid, SIGINT);
    }
}

void childHandler()
{
    int childPid, childStatus;
    childPid=wait(&childStatus);
    printf("Child %d terminated within %d seconds \n", childPid, delay);
    exit(0);
}
```

Output

```
Error:: No such file or directory  
Child 10698 terminated within 1 seconds
```

See Also

printf

Synopsis**#include** <stdio.h>**int printf**(const char *restrict* format, ...);**Purpose**Print output to **stdout**.**Return Value**The **printf** function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.**Parameters***format* Pointer to a character.**Description**The **printf** function is equivalent to **fprintf** with the argument **stdout** interposed before the arguments to **printf**.**Example**

```
/* this program prints a string.
*/

#include <stdio.h>

int main() {
    char *c = "This is a test string";
    printf("%s\n", c);
}
```

Output

This is a test string

See Also

putchar

Synopsis

```
#include <stdio.h>
int putchar(int c);
```

Purpose

Read next character from stream.

Return Value

The **putchar** function returns the character written. If a read error occurs, the error indicator for the stream is set and **putchar** returns **EOF**.

Parameters

c Character.

Description

The **putchar** function is equivalent to **putc** with the argument **stdout**.

Example

```
/* a sample program that writes a string
to stdout. */
#include <stdio.h>
#include <stdlib.h>

int main() {
    char str[] = "This is a test.\n";
    char *p;

    p = str;
    for(; *p; p++) putchar(*p);
}
```

Output

This is a test.

See Also

putc

Synopsis

```
#include <stdio.h>
```

```
int putc(int c, FILE *stream);
```

Purpose

Print output to stream.

Return Value

The **putc** function returns the character written. If a write error occurs, the error indicator for the stream is set and **putc** returns **EOF**.

Parameters

c Pointer to an integer.

stream Pointer to a stream.

Description

The **putc** function is equivalent to **fputc**, except that if it is implemented as a macro, it may evaluate *stream* more than once, so that argument should never be an expression with side effects.

Example

```
/* a sample program that writes a string
to a file. */
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *stream;
    char *name;
    int c, i;

    c = 'a';
    if((name = tmpnam(NULL)) == NULL)
        printf("temp file name could not be made. \n");
    /* open file for reading and writing */
    if((stream = fopen(name,"w")) == NULL) {
        printf("cannot open file\n");
        exit(1);
    }
    for(i =0; i <10; i++) {
        putc(c, stream);
        printf("%c", (char)c);
    }
    printf("\n");
    fclose(stream);
    remove(name);
}
```

Output

```
aaaaaaaaaa
```

See Also

puts

Synopsis

```
#include <stdio.h>
int puts(const char *s);
```

Purpose

Write output to a string.

Return Value

The **puts** function returns **EOF** if a write error occurs; otherwise it returns a nonnegative value.

Parameters

s Pointer to a character.

Description

The **puts** function writes the string pointed to by *s* to the stream pointed to by **stdout**, and appends a new-line character to the output. The terminating null character is not written.

Example

```
/* a sample program that writes a string
to stdout. */
#include <stdio.h>
#include <stdlib.h>

int main() {
    char str[80];

    strcpy(str, "this is a test.\n");
    puts(str);
}
```

Output

```
this is a test.
```

See Also

remove

Synopsis

```
#include <stdio.h>
```

```
int remove(const char *filename);
```

Purpose

Remove a file.

Return Value

The **remove** function returns zero if the operation succeeds, nonzero if it fails.

Parameters

filename Pointer to a file.

Description

The **remove** function causes the file whose name is the string pointed to by *filename* to be no longer accessible by that name. A subsequent attempt to open that file using that name will fail, unless it is created anew. If the file is open, the behavior of the **remove** function is implementation-defined.

Example

Output

See Also

rename

Synopsis

```
#include <stdio.h>
```

```
int rename(const char *old, const char *new);
```

Purpose

Remove a file.

Return Value

The **rename** function returns zero if the operation succeeds, nonzero if it fails, in which case if the file existed previously it is still known by its original name.

Parameters

old Pointer to the old file name.

new Pointer to the new file name.

Description

The **rename** function causes the file whose name is the string pointed to by *old* to be henceforth known by the name given by the string pointed to by *new*. The file named *old* is no longer accessible by that name. If a file named by the string pointed to by *new* exists prior to the call to the **rename** function, the behavior is implementation-defined.

Example

```
/* a sample program that renames the file specified as the first
command line argument to that specified by the second command
line argument. this program rename.c is executed by command
    rename.c < rename.in
Input file rename.in contains the following data: rename.c temp
*/
#include <stdio.h>

int main() {
    char name1[80], name2[80];

    printf("Enter filename: ");
    scanf("%s", name1);
    printf("%s\n", name1);
    printf("Enter new name: ");
    scanf("%s", name2);
    printf("%s\n", name2);
    if(rename(name1, name2)!=0)
        printf("Rename error\n");
    else {
        printf("Old file name: %s\n", name1);
        printf("New file name: %s\n", name2);
    }
    if(rename(name2, name1) !=0)
        printf("Rename error\n");
    else {
        printf("Old file name: %s\n", name2);
        printf("New file name: %s\n", name1);
    }
    return (0);
}
```

Output

```
Enter filename: rename.c
Enter new name: temp
Old file name: rename.c
New file name: temp
Old file name: temp
New file name: rename.c
```

See Also

rewind

Synopsis**#include** <stdio.h>**void** **rewind**(FILE **stream*);**Purpose**

Set the file position indicator to the beginning of a stream.

Return ValueThe **rewind** function returns no value.**Parameters***stream* Pointer to a stream.**Description**The **rewind** function sets the file position indicator for the stream pointed to by *stream* to the beginning of the file. It is equivalent to

(**void**)**fseek**(*stream*, 0L, *SEEK_SET*)
except that the error indicator for the stream is also cleared.

Example see **fseek**().**Output****See Also**

scanf

Synopsis**#include** <stdio.h>**int** scanf(const char *restrict* format, ...);**Purpose**Scan input from **stdin**.**Return Value**

The **scanf** function returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, the **scanf** function returns the number of input items assigned, which can be fewer than provided for, even zero, in the event of an early matching failure.

Parameters*format* Pointer to a character.**Description**

The **scanf** function is equivalent to **fscanf** with the argument **stdin** interposed before the arguments to **scanf**.

Example

```

/* this program scanf.c is executed by command
   scanf.c < scanf.in
   Input file scanf.in contains the following data
   1.0 2.0 3.0 4.0
*/
#include <stdio.h>

int main() {
    float f;
    double d;

    printf("input f\n");
    scanf("%f",&f);
    printf("Your input of f is %f\n", f);
    printf("input d\n");
    scanf("%lf",&d);
    printf("Your input of d is %f\n", d);
    printf("input f\n");
    scanf("%lf",&f); /* ERROR: should use %f for float */
    printf("Your input of d is not what you typed in f = %f\n", f);
    printf("input d\n");
    scanf("%f",&d); /* ERROR: should use %lf for double */
    printf("Your input of d is not what you typed in d = %f\n", d);
}

```

Output

```

input f
Your input of f is 1.000000
input d
Your input of d is 2.000000
input f
Your input of d is not what you typed in f = 2.125000
input d
Your input of d is not what you typed in d = 512.000000

```

See Also

setbuf

Synopsis**#include** <stdio.h>**void setbuf**(FILE *restrict stream, char * restrict buf);**Purpose**

Specify which buffer a stream will use.

Return ValueThe **setbuf** function returns no value.**Parameters***stream* Points to a stream.*buf* Points to a character.**Description**Except that it returns no value, the **setbuf** function is equivalent to the **setvbuf** function invoked with the values **_IOFBF** for *mode* and **BUFSIZ** for *size*, or (if *buf* is a null pointer), with the value **_IONBF** for *mode*.**Example**

```
/* a sample program that reads a file then
displays it's content. */
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *stream;
    char buffer[BUFSIZ], ch;

    /* open file for reading only */
    if((stream=fopen("setbuf.c", "r")) == NULL) {
        printf("cannot open file\n");
        exit(1);
    }
    setbuf(stream,buffer); /* associate a buffer with the stream */
    ch = fgetc(stream);
    printf("%c\n", ch);
    fclose(stream);
}
```

Output

/

See Also

setvbuf

Synopsis

```
#include <stdio.h>
```

```
void setvbuf(FILE * restrict stream, char * restrict buf);
```

Purpose

Specify the mode and buffer size of the given stream.

Return Value

The **setvbuf** function returns zero on success, or nonzero if an invalid value is given for mode or if the request cannot be honored.

Parameters

stream Points to a stream.

buf Points to a character.

mode Holds how stream will be buffered.

size Holds the size of the buffer.

Description

The **setvbuf** function may be used only after the *stream* pointed to by stream has been associated with an open file and before any other operation (other than an unsuccessful call to **setvbuf**) is performed on the stream. The argument *mode* determines how the stream will be buffered, as follows: **_IOFBF** causes input/output to be fully buffered; **_IOLBF** causes input/output to be line buffered; **_IONBF** causes input/output to be unbuffered. If *buf* is not a null pointer, the array it points to may be used instead of a buffer allocated by the **setvbuf** function and the argument *size* specifies the size of the array; otherwise, *size* may determine the size of a buffer allocated by the **setvbuf** function. The contents of the array at any time are indeterminate.

Example

```
/* a sample program that reads a file then
displays it's content. */
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *stream;
    char buffer[500], ch;
    size_t size = 500;

    /* open file for reading only */
    if((stream=fopen("setvbuf.c", "r")) == NULL){
        printf("cannot open file\n");
        exit(1);
    }
    setvbuf(stream,buffer,_IOFBF,size); /* associate a buffer with the stream */
    ch = fgetc(stream);
    printf("%c\n",ch);
    fclose(stream);
}
```

Output

```
/
```

See Also

snprintf

Synopsis

```
#include <stdio.h>
```

```
int snprintf(char * restrict s, size_t n, const char * restrict format, ...);
```

Purpose

Print output to an array.

Return Value

The **snprintf** function returns the number of characters that would have been written had **n** been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than **n**.

Parameters

s Pointer to an array.

n Argument.

format Pointer to a character.

Description

The **snprintf** function is equivalent to **fprintf**, except that the output is written into an array (specified by argument *s*) rather than to a stream. If *n* is zero, nothing is written, and *s* may be a null pointer. Otherwise, output characters beyond the **n**-1st are discarded rather than being written into the array. If copying takes place between objects that overlap, the behavior is undefined.

Example**Output****See Also**

sprintf

Synopsis**#include** <stdio.h>**int** sprintf(**char** * *restrict* *s*, **const char** * *restrict* *format*, ...);**Purpose**

Print output to an array.

Return ValueThe **sprintf** function returns the number of characters written in the array, not counting the terminating null character, or negative value if an encoding error occurred.**Parameters***s* Pointer to an array.*format* Pointer to a character.**Description**The **sprintf** function is equivalent to **fprintf**, except that the output is written to an array (specified by arguments **s**) rather than to a stream. A null character is written at the end of the characters written; it is not counted as part of the returned value. If copying takes place between objects that overlap, the behavior is undefined.**Example****Output****See Also**

sscanf

Synopsis**#include** <stdio.h>**int** **sscanf**(**FILE** * *restrict* *s*, **const char** * *restrict* *format*, ...);**Purpose**

Read input from a stream.

Return Value

The **sscanf** function returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, the **sscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

Parameters*s* Pointer to an array.*format* Pointer to a character.**Description**

The **sscanf** function is equivalent to **fscanf**, except that the input is obtained from a string (specified by argument *s*) rather than from a stream. Reaching the end of the string is equivalent to encountering end-of-file for the **fscanf** function. If copying takes place between objects that overlap, the behavior is undefined.

Example**Output****See Also**

tmpfile

Synopsis

```
#include <stdio.h>
FILE *tmpfile(void);
```

Purpose

Create a temporary file.

Return Value

The **tmpfile** function returns a pointer to the stream of the file that it created. If the file cannot be created, the **tmpfile** function returns a null pointer.

Parameters

No argument.

Description

The **tmpfile** function creates a temporary file that will automatically be removed when it is closed or at program termination. If the program terminates abnormally, whether an open temporary file is removed is implementation-defined. The file is opened for update with the "wb+" mode.

Example

```
/* a sample program that creates a temporary working file. */
#include <stdlib.h>
#include <stdio.h>

int main() {
    FILE *temp;

    if((temp = tmpfile()) == NULL) {
        printf("Cannot open temporary work file.\n");
        exit(1);
    }
}
```

Output**See Also**

tmpnam

Synopsis

```
#include <stdio.h>
char *tmpnam(char *s);
```

Purpose

Create a temporary string for a file name.

Return Value

If the argument is a null pointer, the **tmpnam** function leaves its result in an internal static object and returns a pointer to that object. Subsequent calls to the **tmpnam** function may modify the same object. If the argument is not a null pointer, it is assumed to point to an array of at least **L_tmpnam char**'s; the **tmpnam** function writes its result in that array and returns the argument as its value.

Parameters

s Pointer to a character.

Description

The **tmpnam** function generates a string that is a valid file name and that is not the same as the name of an existing file.

The **tmpnam** function generates a different string each time it is called, up to **TMP_MAX** times. If it is called more than **TMP_MAX** times, the behavior is implementation-defined.

The implementation shall behave as if no library function calls the **tmpnam** function.

Example

```
/* a sample program that displays three unique temporary
file name. */
#include <stdio.h>

int main() {
    char name[40];
    int i;

    for(i = 0; i < 3; i++) {
        tmpnam(name);
        printf("%s\n", name);
    }
}
```

Output

```
/var/tmp/aaaxraG7u
/var/tmp/baayraG7u
/var/tmp/caazraG7u
```

See Also

ungetc

Synopsis

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

Purpose

Push a character back onto the input stream.

Return Value

The **ungetc** function returns the next character pushed back after conversion, or **EOF** if the operation fails.

Parameters

c Pointer to an integer.

stream Pointer to a stream.

Description

The **ungetc** function pushes the character specified by *c* (converted to an **unsigned char**) back onto the input stream pointed to by *stream*. Pushed-back characters will be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by *stream*) to a file positioning function (**fseek**, **fsetpos**, or **rewind**) discards any pushed-back characters for the stream. The external storage corresponding to the stream is unchanged.

One character of pushback is guaranteed. If the **ungetc** function is called too many times on the same stream without an intervening read or file positioning operation on that stream, the operation may fail.

If the value of *c* equals that of the macro **EOF**, the operation fails and the input stream is unchanged.

A successful call to the **ungetc** function clears the end-of-file indicator for the stream. The value of the file position indicator for the stream after reading or discarding all pushed-back characters shall be the same as it was before the characters were pushed back. For a text stream, the value of its file position indicator after a successful call to the **ungetc** function is unspecified until all pushed-back characters are read or discarded. For a binary stream, its file position indicator is decremented by each successful call to the **ungetc** function; if its value was zero before a call, it is indeterminate after the call.

Example

```
/* a sample program that inserts a character into the
buffer of a file then reads it and displays it. */
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#define BUF_SIZE 40

int main() {
    char buf[BUF_SIZE], c;
    FILE *stream;

    if((stream = fopen("ungetc.c", "r")) != NULL) {
        c = fgetc(stream);
        printf("The character pulled from the file is: %c\n", c);
        printf("ungetc(c,stream) = ");
        fputc( ungetc(c,stream), stdout);
        printf("\n");
    }
    else
        printf("Error in opening file!\n");
    fclose(stream);
}
```

```
}
```

Output

```
The character pulled from the file is: /  
ungetc(c,stream) = /
```

See Also

vfprintf

Synopsis**#include** <stdio.h>**int** vfprintf(**FILE** * *restrict stream*, **const char** * *restrict format*, **va_list** *arg*);**Purpose**

Print output to a given stream.

Return ValueThe **vfprintf** function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.**Parameters***stream* Pointer to a stream.*format* Pointer to a character.*arg* Argument.**Description**The **vfprintf** function is equivalent to **fprintf**, with the variable argument list replaced by *arg*, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vfprintf** function does not invoke the **va_end** macro.**Example****Output****See Also**

vprintf

Synopsis**#include** <stdio.h>**int vprintf**(const char * *restrict* *format*, **va_list** *arg*);**Purpose**

Print output to a given stream.

Return ValueThe **vprintf** function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.**Parameters***format* Pointer to a character.*arg* Argument.**Description**The **vprintf** function is equivalent to **printf**, with the variable argument list replaced by *arg*, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vprintf** function does not invoke the **va_end** macro.**Example****Output****See Also**

vsnprintf

Synopsis

```
#include <stdio.h>
```

```
int vsnprintf(char * restrict s, size_t n, const char * restrict format, va_list arg);
```

Purpose

Print output to an array.

Return Value

The **vsnprintf** function returns the number of characters that would have been written had *n* been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than *n*.

Parameters

s Pointer to a character.

n Argument.

format Pointer to a character.

arg Argument.

Description

The **vsnprintf** function is equivalent to **snprintf**, with the variable argument list replaced by *arg*, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vsnprintf** function does not invoke the **va_end** macro. If copying takes place between objects that overlap, the behavior is undefined.

See Also

vsprintf

Synopsis

#include <stdio.h>

int vsprintf(*char* * *restrict* *s*, **const** *char* * *restrict* *format*, **va_list** *arg*);

Purpose

Print output to an array.

Return Value

The **vsprintf** function returns the number of characters that would have been written had *n* been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred.

Parameters

s Pointer to a character.

format Pointer to a character.

arg Argument.

Description

The **vsprintf** function is equivalent to **sprintf**, with the variable argument list replaced by *arg*, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vsprintf** function does not invoke the **va_end** macro. If copying takes place between objects that overlap, the behavior is undefined.

Example

Output

See Also

Chapter 18

String Handling — `<stdlib.h>`

The header `stdlib.h` declares five types and several functions of general utility, and defines several macros.

The types declared are `size_t` and `wchar_t`,

div_t

which is a structure type that is the type of the value returned by the `div` function,

ldiv_t

which is a structure type that is the type of the value returned by the `ldiv` function, and

lldiv_t

which is a structure type that is the type of the value returned by the `lldiv` function.

The macros defined are

EXIT_FAILURE

and

EXIT_SUCCESS

which expand to integer constant expressions that may be used as the argument to the `exit` function to return unsuccessful or successful termination status, respectively, to the host environment;

RAND_MAX

which expands to an integer constant expression, the value of which is the maximum value returned by the `rand` function; and

MB_CUR_MAX

which expands to a positive integer expression with type `size_t` whose value is the maximum number of bytes in a multibyte character for the extended character set specified by the current locale (category `LC_TYPE`), and whose value is never greater than `MB_LEN_MAX`.

Public Data

None.

Functions

The following functions are prototyped in the header file `stdlib.h`.

Functions	Description
abort()	Terminates the program.
abs()	Computes an absolute value.
atexit()	Registers the termination of a program.
atoc()	Converts portion of a string to double complex representation.
atof()	Converts portion of a string to double representation.
atoi()	Converts portion of a string to int representation.
atol()	Converts portion of a string to long int representation.
atoll()	Converts portion of a string to long long int representation.
bsearch()	Searches for matching objects.
calloc()	Allocates space for an array.
div()	Computes a quotient and a remainder.
exit()	Causes normal program termination.
free()	Deallocates a space.
getenv()	Searches for matching strings.
iscnum()	Find out if a string is a valid complex number.
isenv()	Find out if a string is an environment variable.
isnum()	Find out if a string is a valid numerical number.
iswnum()	Find out if a wide character string is a valid numerical number.
labs() ()	Computes an absolute value.
llabs()	Computes an absolute value.
malloc()	Allocates space for an object.
mblen()	Determine the number of bytes in a multibyte character.
mbstowcs()	Converts a sequence of multibyte characters into corresponding codes.
mbtowc()	Determines the number of bytes in a multibyte character.
qsort()	Sorts an array of objects.
rand()	Returns a psuedo-random integer.
realloc()	Deallocates pointer from old object to new object.
remenv()	Remove an environment variable.
srand()	Returns a new psuedo-random integer.
strtod()	Converts initial portion of a string.
strtof()	Converts initial portion of a string.
strtol()	Converts initial portion of a string.
strtold()	Converts initial portion of a string.
strtoll()	Converts initial portion of a string.
strtoul()	Converts initial portion of a string.
strtoull()	Converts initial portion of a string.
system()	Execute a program through a command processor.
wcstombs()	Converts a sequence of codes into corresponding multibyte characters.
wctomb()	Determines the number of bytes needed to represent a multibyte character.

Macros

The following macros are defined for the **stdlib** header.

Macro	Description
EXIT_FAILURE	Returns unsuccessful termination status.
EXIT_SUCCESS	Returns successful termination status.
RAND_MAX	Integer which holds the maximum value returned by the rand function.
MB_CUR_MAX	Unsigned integer which holds maximum number of bytes specified by the current locale.

Declared Types

The following types are declared in the **stdlib** header.

Declared Types	Description
size_t	Unsigned integer.
wchar_t	Integer.
div_t	Structure of type returned by div .
ldiv_t	Structure of type returned by ldiv .
lldiv_t	Structure of type returned by lldiv .

Portability

This header has no known portability problem.

abort

Synopsis

```
#include <stdlib.h>
void abort(void);
```

Purpose

Terminates the program.

Return Value

The **abort()** function does not return to its caller.

Parameters

no argument.

Description

The **abort()** function causes abnormal program termination to occur, unless the signal **SIGABRT** is being caught and the signal handler does not return. Whether open output streams are flushed or open streams closed or temporary files removed is implementation-defined. An implementation-defined form of the status *unsuccessful termination* is returned to the host environment by means of the function call **raise(SIGABRT)**.

Example

```
/* a sample program that use abort() */
#include <stdlib.h>
#include <stdio.h>

int main() {
    printf("abort() is called\n");
    abort();
    printf("This is not called\n");
}
```

Output

```
abort() is called
```

See Also

abs

Synopsis

```
#include <stdlib.h>
```

```
int abs(int j);
```

```
long int labs(long int j);
```

```
long long int llabs(long long int j);
```

Purpose

Computes an absolute value.

Return Value

The **abs()**, **labs()**, and **llabs()** functions return the absolute value.

Parameters

j Integer argument.

Description

The **abs()**, **labs()**, and **llabs()** functions compute the absolute value of an integer *j*. If the result cannot be represented, the behavior is undefined.

Example

```
/* A sample program to compute absolute values.*/
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int num = -2;
    printf("Number is %d\n", num);
    printf("but after running abs()\n");
    num = abs(num);
    printf("Number is %d\n", num);
}
```

Output**See Also**

atexit

Synopsis

```
#include <stdlib.h>
```

```
int atexit(void (*func)(void));
```

Purpose

Registers the termination of a program.

Return Value

The `atexit()` function returns zero if the registration succeeds, nonzero if it fails.

Parameters

func A registered function to be called at normal program termination.

Description

The `atexit()` function registers the function pointed to by *func*, to be called without arguments at normal program termination.

Environmental Limits

The number of functions that can be registered is not limited.

Example

```
/* this is a sample that use atexit() */
#include <stdio.h>
#include <stdlib.h>
void atexitHandler();

int main()
{
    printf("The program is running. \n");
    atexit(atexitHandler); /* call atexitHandler before exit */
}

void atexitHandler()
{
    printf("Call function to exit. \n ");
}
```

Output

```
The program is running.
Call function to exit.
```

See Also

`exit()`

atoc

Synopsis

```
#include <stdlib.h>
```

```
double complex atoc(const char *nptr);
```

Purpose

Converts portion of a string to double complex representation.

Return Value

The `atoc()` function returns the converted value.

Parameters

nptr Pointer to a character.

Description

The `atoc()` function converts the initial portion of the string pointed to by *nptr* to **double complex** representation. The complex number can be represented as `complex(re, im)` or in conventional mathematical notation such as a , $a + ib$, $a + i * b$, $ib + a$, $i * b$, where imaginary number i can be replaced by j or I , and a and b are constants, and plus sign '+' can be replaced by minus sign '-'.

Example

```
/* a sample program that changes a string to double complex*/
#include <stdlib.h>
#include <complex.h>

int main() {
    char *number = "1.27859-I*32.6532";
    char *number2 = "1.27859+i32.6532";
    char *number3 = "complex(3,4)";
    double complex outnumber;

    outnumber = atoc(number);
    printf("the complex number is %f \n",outnumber);
    outnumber = atoc(number2);
    printf("the complex number is %f \n",outnumber);
    outnumber = atoc(number3);
    printf("the complex number is %f \n",outnumber);
}
```

Output

```
the complex number is complex(1.278590,-32.653200)
the complex number is complex(1.278590,32.653200)
the complex number is complex(3.000000,4.000000)
```

See Also

`atof()`, `atoi()`, `strtod()`, `strtof()`, `strtold()`.

atof

Synopsis

```
#include <stdlib.h>
```

```
double atof(const char *nptr);
```

Purpose

Converts portion of a string to double representation.

Return Value

The `atof()` function returns the converted value.

Parameters

nptr Pointer to a character.

Description

The `atof()` function converts the initial portion of the string pointed to by *nptr* to **double** representation. Except for the behavior on error, it is equivalent to

```
strtod(nptr, (char **)NULL)
```

Example

```
/* a sample program that changes a string to double */
#include <stdlib.h>

int main() {
    char *number = "1.27859";
    double outnumber;

    outnumber = atof(number);
    printf("the out double number is %f \n", outnumber);
}
```

Output

```
the out double number is 1.278590
```

See Also

`atoc()`, `atoi()`, `strtod()`, `strtof()`, `strtold()`.

atoi

Synopsis

```
#include <stdlib.h>
```

```
int atoi(const char *nptr);
```

```
long int atol(const char *nptr);
```

```
long long int atoll(const char *nptr);
```

Purpose

Converts portion of a string to int, long int, or long long int representation.

Return Value

The `atoi()`, `atol()`, and `atoll()` functions return the converted value.

Parameters

nptr Pointer to a character.

Description

The `atoi()`, `atol()`, and `atoll()` functions convert the initial portion of the string pointed to by *nptr* to **int**, **long int**, and **long long int** representation, respectively. Except for the behavior on error, they are equivalent to

```
atoi(): (int)strtol(nptr, (char **)NULL, 10)
```

```
atol(): strtol(nptr, (char **)NULL, 10)
```

```
atoll(): strtoll(nptr, (char **)NULL, 10)
```

Example

```
/* a sample program that changes a string to integer */
#include <stdlib.h>

int main() {
    char *number = "1278";
    int outnumber;

    outnumber = atoi(number);
    printf("the out integer number is %d \n", outnumber);
}
```

Output

```
the out integer number is 1278
```

See Also

`atoc()`, `atof()`, `strtol()`, `strtoll()`, `strtoul()`, `strtoull()`.

bsearch

Synopsis

```
#include <stdlib.h>
```

```
void * bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));
```

Purpose

Searches for matching objects.

Return Value

The `bsearch()` function returns a pointer to a matching element of the array, or a null pointer if no match is found. If two elements compare as equal, which element is matched is unspecified.

Parameters

key Pointer to a void.

base Pointer to a void.

nmemb Integer type argument.

size Integer type argument.

compar Pointer to a function with two arguments and return value of int type.

Description

The `bsearch()` function searches an array of *nmemb* objects, the initial element of which is pointed to by *base*, for an element that matches the object pointed to by *key*. The size of each element of the array is specified by *size*.

The comparison function pointed to by *compar* is called with two arguments that point to the *key* object and to an array element, in that order. The function shall return an integer less than, equal to, or greater than zero if the *key* object is considered, respectively, to be less than, equal to, or greater than the array element. The array shall consist of: all the elements that compare less than, all the elements that compare equal to, and all the elements that compare greater than the *key* object, in that order.

Example

```
/* a sample program that decides if the character
   is in the alphabet */
#include <stdio.h>
#include <stdlib.h>

int main() {
    char *alpha="abcdefghijklmnopqrstuvwxy";
    char ch, *p;
    int comp();

    ch='m';
    p=(char *) bsearch(&ch,alpha,26,1,comp);
    if(p)
        printf("%c is in alphabet\n", ch);
    else
        printf("%c is not in alphabet\n", ch);
}

/* compare two characters */
int comp(char *ch, char *s)
{
    return (*ch)-(*s);
}
```

Output

```
m is in alphabet
```

See Also

calloc

Synopsis**#include** <stdlib.h>**void *** calloc(size_t *nmemb*, size_t *size*);**Purpose**

Allocates space for an array.

Return ValueThe **calloc()** function returns either a null pointer or a pointer to the allocated space.**Parameters***nmemb* Unsigned integer argument.*size* Integer argument.**Description**The **calloc()** function allocates space for an array of *nmemb* objects, each of whose size is *size*. The space is initialized to all bits zero.**Example**

```
/* a sample program that allocate 100 floats in memory */
#include <stdio.h>
#include <malloc.h>

int main() {
    float *p;

    p = (float *) calloc(100, sizeof(float));
    if(!p){
        printf("allocation failure - aborting\n");
        exit(1);
    }
    printf ("%f\n", p[8]);
    p[8] = 1.2;
    printf ("%f\n", p[8]);
    free(p);
}
```

Output

```
0.000000
1.200000
```

See Also

div

Synopsis

```
#include <stdlib.h>
div_t div(int numer, int denom);
ldiv_t ldiv(long int numer, long int denom);
lldiv_t lldiv(long long int numer, long long int denom);
```

Purpose

Computes a quotient and a remainder.

Return Value

The `div()`, `ldiv()`, and `lldiv()` functions return a structure of type `div_t`, `ldiv_t`, and `lldiv_t`, respectively, comprising both the quotient and the remainder. The structures shall contain (in either order) the members *quot* (the quotient) and *rem* (the remainder), each of which have the same type as the arguments *numer* and *denom*. If either part of the result cannot be represented, the behavior is undefined.

Parameters

numer Integer argument.
denom Integer argument.

Description

The `div()`, `ldiv()`, and `lldiv()` functions compute $numer / denom$ and $numer \% denom$ in a single operation.

Example

```
/* a sample program that computes the quotient
and remainder of the division of the numerator
by the denominator. */
#include <stdio.h>
#include <stdlib.h>

int main() {
    int number=400;
    int denom=5;
    div_t result;

    result=div(number,denom);
    printf("%d is divided by %d\n",number,denom);
    printf("the quotient is %d\n",result.quot);
    printf("the remainder is %d\n",result.rem);
}
```

Output

```
400 is divided by 5
the quotient is 80
the remainder is 0
```

See Also

exit

Synopsis

```
#include <stdlib.h>
void exit(int status);
```

Purpose

Causes normal program termination.

Return Value

The `exit()` function can not return to its caller.

Parameters

status Integer argument.

Description

The `exit()` function causes normal program termination to occur. If more than one call to the `exit()` function is executed by a program, the behavior is undefined.

First, all functions registered by the `atexit()` function are called, in the reverse order of their registration.

Next, all open streams with unwritten buffered data are flushed, all open streams are closed, and all files created by the `tmpfile()` function are removed.

Finally, control is returned to the host environment. If the value of *status* is zero or `EXIT_SUCCESS`, an implementation-defined form of the status *successful termination* is returned. If the value of *status* is `EXIT_FAILURE`, an implementation-defined form of the status *unsuccessful termination* is returned. Otherwise the status returned is implementation-defined.

Example

```
/* a sample program that use exit() */
#include <stdlib.h>
#include <stdio.h>

int main() {
    printf("exit() is called\n");
    exit(EXIT_SUCCESS);
    printf("This is not called\n");
}
```

Output

```
exit() is called
```

See Also

free

Synopsis

```
#include <stdlib.h>
void free(void *ptr);
```

Purpose

Deallocates a space.

Return Value

The `free()` function returns either a null pointer or a pointer to the allocated space.

Parameters

ptr Pointer to an argument of void type.

Description

The `free()` function causes the space pointed to by *ptr* to be deallocated, that is, made available for further allocation. If *ptr* is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by the `calloc()`, `malloc()`, or `realloc()` function, or if the space has been deallocated by a call to `free()` or `realloc()`, the behavior is undefined.

Example

```
/* a sample program that allocate 100 floats in memory */
#include <stdio.h>
#include <malloc.h>

int main() {
    float *p;

    p = (float *) calloc(100, sizeof(float));
    if(!p){
        printf("allocation failure - aborting\n");
        exit(1);
    }
    printf("Will now free 100 floats from memory.\n");
    free(p);
}
```

Output

Will now free 100 floats from memory.

See Also

getenv

Synopsis

```
#include <stdlib.h>
```

```
char * getenv(const char *name);
```

Purpose

Searches for matching strings.

Return Value

The `getenv()` function returns a pointer to a string associated with the matched list member. The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the `getenv()` function. If the specified **name** cannot be found, a null pointer is returned.

Parameters

name Pointer to a character.

Description

The `getenv()` function searched an *environmental list*, provided by the host environment, for a string that matches the string pointed to by *name*. The set of environment names and the method for altering the environment list are implementation-defined.

The implementation shall behave as if no library function calls the `getenv()` function.

Example

Output

See Also

iscnum

Synopsis

```
#include <stdlib.h>
int iscnum(string_t pattern);
```

Purpose

Find out if a string is a complex number or real number.

Return Value

This function returns 1 if the string is a valid complex number or real number or 0 if not.

Parameters

name The input string.

Description

This function determines if the input string *pattern* is a valid complex number, a pure imaginary number, or a real number. The complex number can be represented as `complex(re, im)` or in conventional mathematical notation such as a , $a + ib$, $a + i * b$, $ib + a$, $i * b$, where imaginary number i can be replaced by j or I , and a and b are constants, and plus sign '+' can be replaced by minus sign '-'.

Example

```
#include <complex.h>
#include <stdlib.h>
int main() {
    string_t s="3+I*4";

    printf("isnum(\"0x30\") = %d\n", isnum("0x30"));
    printf("isnum(\"abc\") = %d\n", isnum("abc"));
    printf("iswnum(L\"0x30\") = %d\n", iswnum(L"0x30"));
    printf("iswnum(L\"abc\") = %d\n", iswnum(L"abc"));

    printf("iscnum(\"10+I*4\") = %d\n", iscnum(s));
    printf("atoc(s) = %f\n", atoc(s));
    printf("iscnum(\"I*4\") = %d\n", iscnum("I*4"));
    printf("atoc(\"I*4\") = %f\n", atoc("I*4"));
    printf("iscnum(\"complex(1,2)\") = %d\n", iscnum("complex(1,2)"));
    printf("atoc(\"complex(1,2)\") = %f\n", atoc("complex(1,2)"));
    printf("iscnum(\"4\") = %d\n", iscnum("4"));
}
```

Output

```
isnum("0x30") = 1
isnum("abc") = 0
iswnum(L"0x30") = 1
iswnum(L"abc") = 0
iscnum("10+I*4") = 1
atoc(s) = complex(3.000000,4.000000)
iscnum("I*4") = 1
atoc("I*4") = complex(0.000000,4.000000)
iscnum("complex(1,2)") = 1
atoc("complex(1,2)") = complex(1.000000,2.000000)
iscnum("4") = 0
```

See Also

`isnum()`, `iswnum()`, `isenv()`, `iskey()`, `isvar()`, `isstudent()`.

isenv

Synopsis

```
#include <stdlib.h>
int isenv(string_t name);
```

Purpose

Find out if a string is an environment variable.

Return Value

This function returns 1 if the string is an environment variable or 0 if it is not.

Parameters

name The input string.

Description

This function determines if the input string *name* is an environmental variable.

Example

```
/* test if a string is an environment variable */
#include <stdlib.h>
#include <string.h>

int main() {
    int i;

    printf("isenv(\"CHHOME\") = %d\n", isenv("CHHOME"));
    printf("isenv(\"unknown\") = %d\n", isenv("unknown"));
}
```

Output

```
isenv("CHHOME") = 1
isenv("unknown") = 0
```

See Also

iskey(), **isnum()**, **isstudent()**, **isvar()**.

isnum

Synopsis

```
#include <stdlib.h>
int isnum(string_t pattern);
```

Purpose

Find out if a string is a number.

Return Value

This function returns 1 if the string is a valid number or 0 if not.

Parameters

name The input string.

Description

This function determines if the input string *pattern* is a valid number.

Example

See `isenv()`.

See Also

`iscnum()`, `iswnum()`, `isenv()`, `iskey()`, `isvar()`, `isstudent()`.

iswnum

Synopsis

```
#include <stdlib.h>
```

```
int iswnum(wchar_t pattern);
```

Purpose

Find out if a wide character string is a number.

Return Value

This function returns 1 if the string is a valid number or 0 if not.

Parameters

name The input string.

Description

This function determines if the input string *pattern* is a valid number.

Example

See `isenv()`.

See Also

`isnum()`, `isalnum()`, `isenv()`, `iskey()`, `isvar()`, `isstudent()`.

malloc

Synopsis

```
#include <stdlib.h>
void * malloc(size_t size);
```

Purpose

Allocates space for an object.

Return Value

The `malloc()` function returns either a null pointer or a pointer to the allocated space.

Parameters

size Integer argument.

Description

The `malloc()` function allocates space for an object whose size is specified by *size* and whose value is indeterminate.

Example

```
/* a sample program that allocates space for a string
   dynamically, request user input, and then print the
   string backwards. This is executed with the following
   statement: malloc.c < malloc.in. The contents of
   malloc.in are: abcd */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main() {
    char *s;
    int t;

    s = malloc(80);
    printf("Please enter a string: ");
    if(!s) {
        printf("Memory request failed.\n");
        exit(1);
    }
    fgets(s, 80, stdin);
    printf("%s", s);
    printf("Converted string: ");
    for(t = strlen(s)-2; t>=0; t--)
        putchar(s[t]);
    printf("\n");
    free(s);
    return (0);
}
```

Output

```
Please enter a string: abcd
Converted string: dcba
```

See Also

mblen

Synopsis

```
#include <stdlib.h>
```

```
int mblen(const char *s, size_t n);
```

Purpose

Determines the number of bytes in a multibyte character.

Return Value

If *s* is a null pointer, the **mblen()** function returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If *s* is not a null pointer, the **mblen()** function either returns 0 (if *s* points to the null character), or returns the number of bytes that are contained in the multibyte character (if the next *n* or fewer bytes form a valid multibyte character), or returns -1 (if they do not form a valid multibyte character).

Parameters

s Pointer to a character.

n Unsigned integer argument.

Description

If *s* is not a null pointer, the **mblen()** function determines the number of bytes contained in the multibyte character pointed to by *s*. Except that the shift state of the **mbtowc** function is not affected, it is equivalent to

```
mbtowc((wchar_t *)0, s, n);
```

The implementation shall behave as if no library function calls the **mblen()** function.

Example

```
/* a sample program that displays the length of the
multibyte character pointed by mb. */
#include <stdio.h>
#include <stdlib.h>

int main() {
    char mb[] = "This is a test";

    printf("The length of the multibyte character pointed by str is %d\n" \
, mblen(mb, sizeof(char)));
}
```

Output

The length of the multibyte character pointed by str is 1

See Also

mbstowcs

Synopsis

```
#include <stdlib.h>
```

```
size_t mbstowcs(wchar_t * restrict pwcs, const char restrict *s, size_t n);
```

Purpose

Converts a sequence of multibyte characters into corresponding codes.

Return Value

If an invalid multibyte character is encountered, the **mbstowcs()** function returns $(\text{size_t}) - 1$. Otherwise, the **mbstowcs()** function returns the number of array elements modified, not including a terminating zero code, if any.

Parameters

pwcs Integer argument.

s Pointer to a character.

n Unsigned integer argument.

Description

The **mbstowcs()** function converts a sequence of multibyte characters that begins in the initial shift state from the array pointed to by *s* into a sequence of corresponding codes and stores not more than *n* codes into the array pointed to by *pwcs*. No multibyte characters that follow a null character (which is converted into a code with value zero) will be examined or converted. Each multibyte character is converted as if by a call to the **—mbtowc** function, except that the shift state of the **mbtowc()** function is not effected.

No more than *n* elements will be modified in the array pointed to by *pwcs*. If copying takes place between objects that overlap, the behavior is undefined.

Example

```
/* a sample program that converts the first 4 characters
in the multibyte character pointed by mb and puts the result
in str. */
#include <wchar.h>
#include <stdlib.h>

int main() {
    char s[] = "This is a test";
    wchar_t pwcs[20];
    size_t n;

    n = strlen(s)+1;
    printf("Original string: %s\n", s);
    printf("Number of characters to convert: %d\n", n);
    if(mbstowcs(pwcs,s,n) == -1) {
        printf("error convert\n");
        exit(1);
    }
    printf("The converted characters are: ");
    fputws(pwcs, stdout);
    fputs("\n", stdout);
}
```

Output

```
Original string: This is a test
Number of characters to convert: 15
The converted characters are: This is a test
```

See Also

mbtowc

Synopsis

```
#include <stdlib.h>
```

```
int mbtowc(wchar_t * restrict pwc, const char restrict *s, size_t n);
```

Purpose

Determines the number of bytes in a multibyte character.

Return Value

If *s* is a null pointer, the **mbtowc()** function returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If *s* is not a null pointer, the **mbtowc()** function either returns 0 (if *s* points to the null character), or returns the number of bytes that are contained in the converted multibyte character (if the next *n* or fewer bytes form a valid multibyte character), or returns -1 (if they do not form a valid multibyte character).

In no case will the value returned be greater than *n* or the value of the **MB_CUR_MAX** macro.

Parameters

pwc Integer argument.

s Pointer to a character.

n Unsigned integer argument.

Description

The *s* is not a null pointer, the **mbtowc()** function determines the number of bytes that are contained in the multibyte character pointed to by *s*. It then determines the code for the value of type **wchar_t** that corresponds to that multibyte character. (The value of the code corresponding to the null character is zero.) If the multibyte character is valid and *pwc* is not a null pointer, the **mbtowc()** function stores the code in the object pointed to by *pwc*. At most *n* bytes of the array pointed to by *s* will be examined.

The implementation shall behave as if no library function calls the **mbtowc()** function.

Example

```
/* a sample program that converts the multibyte character in s
into its equivalent wide character and puts the result in the
array pointed to by pwcs. */
#include <wchar.h>
#include <stdlib.h>

int main() {
    char s[] = "This is a test";
    wchar_t pwcs[8];
    size_t n;

    n = 1;
    if(mbtowc(pwcs,s,n) == -1) {
        printf("error convert\n");
        exit(1);
    }
    printf("The original string is: %s\n", s);
    printf("The converted characters are: ");
    fputws(pwcs, stdout);
    fputs("\n", stdout);
}
```

Output

```
The original string is: This is a test
The converted characters are: T
```

See Also

qsort

Synopsis

```
#include <stdlib.h>
```

```
void qsort(const void *base, size_t nmemb, size_t size, int (*compar)(ocnst void *, const void *));
```

Purpose

Sorts an array of objects.

Return Value

The `qsort()` function returns no value.

Parameters

base Pointer to a void.

nmemb Integer type argument.

size Integer type argument.

compar Pointer to a function.

Description

The `qsort()` function sorts an array of *nmemb* objects, the initial element of which is pointed to by *base*. The size of each object is specified by *size*.

The contents of the array are sorted into ascending order according to a comparison function pointed to by *compar*, the function is called with two arguments that point to the objects being compared. The function shall return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

Example

```
/* a sample program that sorts a list of integers and
displays the result. */
#include <stdio.h>
#include <stdlib.h>

int num[10] = { 1,3,6,5,8,7,9,6,2,0};
int comp(const void *,const void *);

int main() {
    int i;

    printf("Original array:\t");
    for(i=0; i<10; i++)
        printf("%d ",num[i]);
    qsort(num,10,sizeof(int),comp);
    printf("\nSorted array:\t");
    for(i=0; i<10; i++)
        printf("%d ",num[i]);
    printf("\n");
}

/* compare the integers */
int comp(const void *i, const void *j) {
    return *(int *)i - *(int *)j;
}
```

Output

```
Original array: 1 3 6 5 8 7 9 6 2 0
Sorted array: 0 1 2 3 5 6 7 6 8 9
```

See Also

rand

Synopsis**#include** <stdlib.h>**int rand**(void);**Purpose**

Returns a psuedo-random integer.

Return ValueThe **rand**() function returns a psuedo-random integer.**Parameters**

no argument.

DescriptionThe **rand**() function computes a sequence of psuedo-random integers in the range of 0 to **RAND_MAX**.**Example**

```
/* a sample program that displays ten pseudorandom numbers. */
#include <stdio.h>
#include <stdlib.h>

int main() {
    int i;

    for(i=0; i<10; i++) printf("%d ",rand());
    printf ("\n");
}
```

Output

16838 5758 10113 17515 31051 5627 23010 7419 16212 4086

See Also

realloc

Synopsis

```
#include <stdlib.h>
```

```
void * realloc(void *ptr, size_t size);
```

Purpose

Deallocates pointer from old object to new object.

Return Value

The **realloc()** function returns a pointer to the new object (which may have the same value as a pointer to the old object), or a null pointer if the new object could not be allocated.

Parameters

ptr Pointer to a void.

size Integer argument.

Description

The **realloc()** function deallocates the old object pointed to by *ptr* and returns a pointer to a new object that has the size specified by *size*. The contents of the new object shall be the same as that of the old object prior to deallocation, up to the lesser of the new and old sizes. Any bytes in the new object beyond the size of the old object have indeterminate values.

If *ptr* is a null pointer, the **realloc()** function behaves like the **malloc()** function for the specified size. Otherwise, if *ptr* does not match a pointer earlier returned by the **calloc()**, **malloc()**, or **realloc()** function, or if the space has been deallocated by a call to the **free()** or **realloc()** function, the behavior is undefined. If memory for the new object cannot be allocated, the old object is not deallocated and its value is unchanged.

Example

```
/* a sample program that first allocates 16 characters,
copies the string "This is 16 chars" into them, and then
use realloc() to increase the size to 17 in order to place
a period at the end. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    char *p;

    p = malloc(16);
    if(!p) {
        printf("Allocation error - aborting.");
        exit(1);
    }
    strcpy(p, "This is 16 chars");
    p = realloc(p, 17);
    if(!p) {
        printf("allocation error - aborting.");
        exit(1);
    }
    strcat(p, ".");
    printf(p);
    printf("\n");
    free(p);
}
```


Output

This is 16 chars.

See Also

remenv

Synopsis

```
#include <stdlib.h>
int remenv(string_t name);
```

Purpose

Remove an environment variable.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

name A string containing the name of the environment variable to be removed.

Description

This function removes an environment variable with the name in string *name*.

Example

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    string_t s;
    putenv("TESINGENV=testingenv");
    s = getenv("TESINGENV");
    printf("s = %s\n", s);

    remenv("TESINGENV");
    s = getenv("TESINGENV");
    if(s == NULL)
        printf("ok: s = NULL\n");
    else
        printf("error: s = %s\n", s);
}
```

Output

```
s = testingenv
ok: s = NULL
```

See Also

putenv().

srand

Synopsis

```
#include <stdlib.h>
```

```
void srand(unsigned int seed);
```

Purpose

Returns a new psuedo-random integer.

Return Value

The **srand()** function returns no value.

Parameters

seed Unsigned integer argument.

Description

The **srand()** function uses the argument as a seed for a new sequence of psuedo-random numbers to be returned by subsequent calls to **rand()**. If **srand()** is then called with the same seed value, the sequence of psuedo-random numbers shall be repeated. If **rand()** is called before any calls to **srand()** have been made, the same sequence shall be generated as when **srand()** is first called with a seed value of 1.

The implementation shall behave as if no library calls the **srand()** function.

Example

```
/* a sample program that displays ten pseudorandom numbers. */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    int i, stime;
    long ltime;

    ltime = time(NULL); /* get current calendar time */
    stime = (unsigned int) ltime/2;
    srand(stime);
    for(i=0; i<10; i++) printf("%d ", rand());
    printf("\n");
}
```

Output

```
17875 24317 11838 16984 8536 4774 910 5469 11847 28683
```

See Also

strtod

Synopsis

```
#include <stdlib.h>
double strtod(const char * restrict nptr, char ** restrict endptr);
float strtof(const char * restrict nptr, char ** restrict endptr);
long double strtold(const char * restrict nptr, char ** restrict endptr);
```

Purpose

Converts portion of a string to double representation.

Return Value

The functions return the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, plus or minus **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL** is returned (according to the return type and sign of the value), and the value of the macro **ERANGE** is stored in **errno**. If the result underflows, the functions return a value whose magnitude is no greater than the smallest normalized positive number in the return type; whether **errno()** acquires the value **ERANGE** is implementation-defined.

Parameters

nptr Pointer to a character.
endptr Pointer to a character.

Description

The **strtod()**, **strtof()**, and **strtold()** functions convert the initial portion of the string pointed to by **nptr** to **double**, **float**, and **long double** representation, respectively. First, they decompose the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the **isspace()** function), a subject sequence resembling a floating-point constant or representing an infinity or NaN; and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then, they attempt to convert the subject sequence to a floating-point number, and return the result.

The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- a nonempty sequence of decimal digits optionally containing a decimal-point character, then an optional exponent part.
- a **0x** or **0X**, then a nonempty sequence of hexadecimal digits optionally containing a decimal-point character, then an optional binary exponent part.
- one of **inf** or **infinity**, ignoring case.
- one of **NAN** or **NAN(*n-char-sequence_{opt}*)**, ignoring case in the **NAN** part, where:

```
n-char-sequence
  digit
  nondigit
  n-char-sequence digit
  n-char-sequence nondigit
```

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is not of the expected form.

If the subject sequence has the expected form for a floating-point number, the sequence of characters starting with the first digit or the decimal-point character (whichever occurs first) is interpreted as a floating constant, except that the decimal-point character is used in place of a period, and that if neither an exponent part nor a decimal-point character appears in a decimal floating-point number, or is a binary exponent part does not appear in a binary floating-point number, an exponent part of the appropriate type with value zero is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the sequence is

interpreted as negated. A character sequence **INF** or **INFINITY** is interpreted as an infinity, if representable in the return type, else like a floating constant that is too large for the range of the return type. A character sequence **NAN** or **NAN** (*n-char-sequence_{opt}*), is interpreted as a quiet NaN, if supported in the return type, else like a subject sequence part that does not have the expected form; the meaning of the n-char sequences is implementation-defined. A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

If the subject sequence has the hexadecimal form and **FLT_RADIX** is a power of 2, the value resulting from the conversion is correctly rounded.

In other than the “C” locale, additional locale-specific subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

Recommended Practice

If the subject sequence has the hexadecimal form and **FLT_RADIX** is not a power of 2, the result should be one of the two numbers in the appropriate internal format that are adjacent to the hexadecimal floating source value, with the extra stipulation that the error should have a correct sign for the current rounding direction.

If the subject sequence has the decimal form and at most **DECIMAL_DIG** (defined in **float.h**) significant digits, the result should be correctly rounded. If the subject sequence *D* has the decimal form and more than **DECIMAL_DIG** significant digits, consider the two bounding, adjacent decimal strings *L* and *U*, both having **DECIMAL_DIG** significant digits, such that the values of *L*, *D*, and *U* satisfy $L \leq D \leq U$. The result should be one of the (equal or adjacent) values that would be obtained by correctly rounding *L* and *U* according to the current rounding direction, with the extra stipulation that the error with respect to *D* should have a correct sign for the current rounding direction.

Example

Output

See Also

strtol

Synopsis

```
#include <stdlib.h>
```

```
long int strtol(const char * restrict nptr, char ** restrict endptr, int base);
```

```
long long int strtoll(const char * restrict nptr, char ** restrict endptr, int base);
```

```
unsigned long int strtoul(const char * restrict nptr, char ** restrict endptr, int base);
```

```
unsigned long long int strtoull(const char * restrict nptr, char ** restrict endptr, int base);
```

Purpose

Converts a portion of a string to double representation.

Return Value

The `strtol()`, `strtoll()`, `strtoul()`, and `strtoull()` functions return the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, `LONG_MIN`, `LONG_MAX`, `LLONG_MIN`, `LLONG_MAX`, `ULONG_MAX`, or `ULLONG_MAX` is returned (according to the return type and sign of the value, if any), and the value of the macro `ERANGE` is stored in `errno`.

Parameters

nptr Pointer to a character.

endptr Pointer to a character.

base Integer argument.

Description

The `strtol()`, `strtoll()`, `strtoul()` and `strtoull()` functions convert the initial portion of the string pointed to by *nptr* to **long int**, **long long int**, **unsigned long int**, and **unsigned long long int** representation, respectively. First, they decompose the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the `isspace` function), a subject sequence resembling an integer represented in some radix determined by the value of *base*, and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then, they attempt to convert the subject sequence to an integer, and return the result.

If the value of *base* is zero, the expected form of the subject sequence is that of an integer constant, optionally preceded by a plus or minus sign, but not including an integer suffix. If the value of *base* is between 2 and 36 (inclusive), the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from **a** (or **A**) through **z** (or **Z**) are ascribed the values of 10 through 35; only letters and digits whose ascribed values are less than that of *base* are permitted. If the value of *base* is 16, the characters **0x** or **0X** may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of *base* is zero, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated (in the return type). A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

In other than the “C” locale, additional locale-specific subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

Example Output

See Also

system

Synopsis

```
#include <stdlib.h>
```

```
int system(const char *string);
```

Purpose

Execute a program through a command processor.

Return Value

If the argument is a null pointer, the **system()** function returns nonzero only if a command processor is available. If the argument is not a null pointer, and the **system()** function does return, it returns an implementation-defined value.

Parameters

string Pointer to a character.

Description

If *string* is a null pointer, the **system()** function determines whether the host environment has a *command processor*. If *string* is not a null pointer, the **system()** function passes the string pointed to by *string* to that command processor to be executed in a manner which the implementation shall document; this might then cause the program calling **system()** to behave in a non-conforming manner or to terminate.

Example

Output

See Also

wcstombs

Synopsis

```
#include <stdlib.h>
```

```
size_t wcstombs(char restrict *s, const wchar_t * restrict pwcs, size_t n);
```

Purpose

Converts a sequence of codes into corresponding multibyte characters.

Return Value

If a code is encountered that does not correspond to a valid multibyte character, the `wcstombs()` function returns `(size_t) - 1`. Otherwise, the `wcstombs()` function returns the number of bytes modified, not including a terminating null character, if any.

Parameters

pwcs Integer argument.

s Pointer to a character.

n Unsigned integer argument.

Description

The `wcstombs()` function converts a sequence of codes that correspond to multibyte characters from the array pointed to by *pwcs* into a sequence of multibyte characters that begins in the initial shift state and stores these multibyte characters into the array pointed to by *s*, stopping if a multibyte character would exceed the limit of *n* total bytes or if a null character is stored. Each code is converted as if by a call to the `wctomb()` function, except that the shift state of the `wctomb()` function is not affected.

No more than *n* elements will be modified in the array pointed to by *pwcs*. If copying takes place between objects that overlap, the behavior is undefined.

Example

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    char s[8];
    wchar_t pwcs[8] = L"123abc\n";
    size_t retval;

    retval = wcstombs(s,pwcs, wcslen(pwcs)+1);
    if(retval != (size_t) -1) {
        printf("the original string is: ");
        fputws(pwcs, stdout);
        printf("The converted result is: %s\n",s);
    }
    else
    printf("Error\n");
}
```

Output

```
the original string is: 123abc
The converted result is: 123abc
```

See Also

wctomb

Synopsis

```
#include <stdlib.h>
```

```
int wctomb(char *s, wchar_t wchar);
```

Purpose

Determines the number of bytes needed to represent a multibyte character.

Return Value

If *s* is a null pointer, the **wctomb()** function returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If *s* is not a null pointer, the **wctomb()** function returns -1 if the value of *wchar* does not correspond to a valid multibyte character, or returns the number of bytes that are contained in the multibyte character corresponding to the value of *wchar*.

In no case will the value returned be greater than the value of the **MB_CUR_MAX** macro.

Parameters

s Pointer to a character.

wchar Integer argument.

Description

The **wctomb()** function determines the number of bytes needed to represent the multibyte character corresponding to the code whose value is *wchar* (including any change in shift state). It stores the multibyte character representation in the array object pointed to by *s* (if *s* is not a null pointer). At most **MB_CUR_MAX** characters are stored. If the value of *wchar* is zero, the **wctomb** function is left in the initial shift state.

The implementation shall behave as if no library function calls the **wctomb()** function.

Example

```
/* This is an example of wctomb(char*, wchar_t). This function converts
   a wide character to the corresponding multibyte character.
*/
#include <stdio.h>
#include <stdlib.h>

int main() {
    char s[128];
    wchar_t pwcs;
    int retval,x;
    pwcs = 'T';
    wctomb(s,pwcs);
    if(retval != -1) {
        printf("The converted result is: %s\n",s);
    }
    else
    printf("Error\n");
}
```

Output

The converted result is: T

See Also

Chapter 19

String Functions — <string.h>

The header **string.h** declares one type and several functions. The type is **size_t**. Various methods are used for determining the lengths of the arrays, but in all cases a **char *** or **void *** argument points to the initial (lowest addressed) character of the array. If an array is accessed beyond the end of an object, the behavior is undefined.

Where an argument declared as **size_t n** specifies the length of the array for a function, **n** can have the value zero on a call to that function. Unless explicitly stated otherwise in the description of a particular function in this subclause, pointer arguments on such a call shall still have valid values. On such a call, if a function that locates a character finds no occurrence, a function that copies characters copies zero characters.

Functions

The following functions are implemented using the **string** header.

Function	Description
memchr	Locates the first occurrence of a specified character in a stream.
memcmp	Compares the first characters in two objects.
memcpy	Copies characters from one object to another.
memmove	Copies characters from one object to another.
memset	Copies a single value into specified number of locations in an object.
str2ascii	Returns the ASCII value for each character.
str2mat	Create a matrix of char from individual strings.
stradd	Adds two strings together.
strcasecmp	Compares two strings after converting all characters to lowercase.
strcat	Appends a copy of one string to another string.
strchr	Locates the first occurrence of a character in a string.
strcmp	Compares two strings.
strcoll	Compares two strings.
strconcat	Appends copies of strings to a string.
strcpy	Copies a string to an array.
strcspn	Computes the length of the maximum initial segment of a string.
strdup	Duplicates a string.
strerror	Maps values of errno .
strgetc	Returns a character from a specified location.
strjoin	Combines strings as a string.
strlen	Computes the length of a string.
strncasecmp	Compares two strings after converting all characters to lowercase.
strncat	Appends characters from an array to a wide string.
strncmp	Compares characters between two arrays.
strncpy	Copies characters from one array to another array.
strpbrk	Locates first occurrence of a character from one string in another string.
strputc	Replaces a character in a specified location.

strrchr	Locates last occurrence of a character in a string.
strrep	Search and replace strings inside another string.
strspn	Computes the length of the maximum initial segment of a string.
strstr	Locates the first occurrence of an object in one string in another string.
strtok	Breaks a string into a sequence of tokens.
strtok_r	Breaks a string into a sequence of tokens, reentrant.
strxfrm	Transforms string and places it in an array.

Macros

No macro is defined for the **string** header.

Declared Types

The following types are declared in the **string** header.

Declared Types	Description
size_t	Unsigned integer.

Portability

Differences between C and Ch

Functions **strep**, **strncpy**, **strcat**, **strncat**, **strcmp**, **strcoll**, **strtolc**, **memcpy**, and **memmove** are built-in functions in C.
Functions **str2ascii**, **str2mat**, **strgetc**, **strputc**, and **strrep** are available only in Ch.

memchr

Synopsis

```
#include <string.h>
```

```
void *memchr(const void *s, int c, size_t n);
```

Purpose

Locates the first occurrence of a specified character in a stream.

Return Value

The `memchr()` function returns a pointer to the located character, or a null pointer if the character does not occur in the object.

Parameters

s Pointer to void argument.

c Integer argument.

n Unsigned integer argument.

Description

The `memchr()` function locates the first occurrence of `c` (converted to an **unsigned char**) in the initial `n` characters (each interpreted as **unsigned char**) of the object pointed to by `s`.

Example

```
/* a sample program that prints 'is a test' on the screen.*/
#include <stdio.h>
#include <string.h>

int main() {
    char p[] = "this is a test\n";
    char *s;

    s = memchr(p, ' ', strlen(p));
    printf(s);
}
```

Output

```
is a test
```

See Also

memcmp

Synopsis

```
#include <string.h>
```

```
int memcmp(const void *s1, const void *s2, size_t n);
```

Purpose

Compares the first characters in two objects.

Return Value

The `memcmp()` function returns an integer greater than, equal to, or less than zero, accordingly as the object pointed to by `s1` is greater than, equal to, or less than the object pointed to by `s2`.

Parameters

`s1` Pointer to void argument.

`s2` Pointer to void argument.

`n` Unsigned integer argument.

Description

The `memcmp()` function compares the first `n` characters of the object pointed to by `s1` to the first `n` characters of the object pointed to by `s2`.

Example

```
/* a sample program that shows the outcome of a comparison
of its two strings.*/
#include <stdio.h>
#include <string.h>

int main() {
    char s1[] = "abcdefghijk1";
    char s2[] = "abcdefkl";
    int outcome;
    int l1, l2;
    int len;

    len = (l1 = strlen(s1)) < (l2 = strlen(s2)) ? l1:l2;
    outcome = memcmp(s1, s2, len);
    if(! outcome) printf("equal\n");
    else if(outcome<0) printf("first less than second\n");
    else printf("first greater than second\n");
}
```

Output

```
first less than second
```

See Also

memcpy

Synopsis

```
#include <string.h>
```

```
void *memcpy(void * restrict s1, const void * restrict s2, size_t n);
```

Purpose

Copies characters from one object to another.

Return Value

The `memcpy()` returns the value of `s1`.

Parameters

s1 Pointer to void argument.

s2 Pointer to void argument.

n Unsigned integer argument.

Description

The `memcpy()` function copies `n` characters from the object pointed to by `s2` to the object pointed to by `s1`. If copying takes place between objects that overlap, the behavior is undefined.

Example**Output****See Also**

memmove

Synopsis

```
#include <string.h>
```

```
void *memmove(void *s1, const void *s2, size_t n);
```

Purpose

Copies characters from one object to another.

Return Value

The `memmove()` returns the value of `s1`.

Parameters

`s1` Pointer to void argument.

`s2` Pointer to void argument.

`n` Unsigned integer argument.

Description

The `memmove()` function copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. Copying takes place as if the `n` characters from the object pointed to by `s2` are first copied into a temporary array of `n` characters that does not overlap the objects pointed to by `s1` or `s2`, and then the `n` characters from the temporary array are copied into the object pointed to by `s1`.

Example

Output

See Also

memset

Synopsis

```
#include <wchar.h>
```

```
void *memset(void *s, int c, size_t n);
```

Purpose

Copies a single value into specified number of locations in an object.

Return Value

The `memset()` function returns the value of `s`.

Parameters

s Pointer to a void argument.

c Integer argument.

n Unsigned integer argument.

Description

The `memset()` function copies the value of `c` (converted to an **unsigned char**) into each of the first `n` characters of the object pointed to by `s`.

Example

Output

See Also

str2ascii

Synopsis**#include** <string.h>**unsigned int** str2ascii(string_t s);**Purpose**

Get the ASCII number of a string.

Return Value

This function returns the sum of ASCII value for each character of a string.

Parameters*s* The input string.**Description**The function **str2ascii()** returns the sum of ASCII value for each character of a string.**Example**

```
#include <stdio.h>
#include <string.h>

int main() {
    int ascii;
    ascii = str2ascii("abcd");
    printf("ASCII num for string `abcd` is %d\n", ascii);
}
```

Output

ASCII num for string `abcd` is 394

See Also

str2mat

Synopsis

```
#include <string.h>
int str2mat(array char mat[:][:], string_t s, ...);
```

Purpose

Create a matrix of char from individual strings.

Return Value

This function returns 0 on successful or -1 on failure.

Parameters

mat A two-dimensional array containing the created matrix.
s The individual string.

Description

This function creates a matrix of char containing the string *s* and strings passed from variable number arguments. The total number of strings including *s* and variable number arguments must equal the number of elements in the first dimension of array *mat*. If the number of strings is less than that of elements of array *mat*, the function call will fail and return -1. The strings more than the number of elements will be ignored.

Example

```
#include <stdio.h>
#include <string.h>
#include <array.h>

int main() {
    array char mat[3][10];
    int status;

    mat = (array char [3][10])' '; /* blank space */
    status = str2mat(mat, "abcd");
    printf("status = %d\nmat = %c\n", status, mat);
    str2mat(mat, "abcd", "efghi", "0123456789");
    printf("mat = \n%c\n", mat);
    status = str2mat(mat, "ABCD", "EFGH", "ab23456789", "too many strings");
    printf("status = %d\nmat = \n%c\n", status, mat);
}
```

Output

```
status = 0
mat = a b c d
```

```
mat =
a b c d
e f g h i
0 1 2 3 4 5 6 7 8 9
```

```
status = -1
mat =
A B C D
E F G H
a b 2 3 4 5 6 7 8 9
```

See Also
strputc().

stradd

Synopsis

```
#include <string.h>
```

```
string_t stradd(string_t s, ...);
```

Purpose

Adds strings together.

Return Value

The `stradd()` function returns the new string.

Parameters

s string argument.

Description

The `stradd()` function adds the string *s* and subsequent strings passed by variable number of arguments.

Example

```
/** add strings */
#include <stdio.h>
#include <string.h>

int main() {
    string_t s, s1 = "ABCD";
    char *s2="cbcd";
    char s3[]="1234";

    s = stradd(s1, s2, s3);
    printf("s = %s\n", s);
    s = stradd(s2, s3);
    printf("s = %s\n", s);
}
```

Output

```
s = ABCDcbcd1234
s = cbcd1234
```

See Also

strcasecmp

Synopsis

```
#include <string.h>
```

```
int strcasecmp(const char *s1, const char *s2);
```

Purpose

Compares two strings.

Return Value

The **strcasecmp()** function returns an integer greater than, equal to, or less than zero, accordingly as the string pointed to by **s1** is greater than, equal to, or less than the string pointed to by **s2**.

Parameters

s1 Pointer to a character argument.

s2 Pointer to a character argument.

Description

The **strcasecmp()** function compares the string pointed to by **s1** to the string pointed to by **s2**. Before the strings are compared, the **tolower()** function is called to change uppercase characters to lowercase.

Example

```
/* a sample program that shows the outcome of a comparison
of its two strings.*/
#include <stdio.h>
#include <string.h>

int main() {
    char s1[] = "abc";
    char s2[] = "abc";
    int outcome;
    int l1,l2;

    outcome = strcasecmp(s1,s2);
    if(! outcome) printf("equal\n");
    else if(outcome < 0) printf("first less than second\n");
    else printf("first greater than second\n");
}
```

Output

```
equal
```

See Also

strcmp(), **strncmp()**, **strncasecmp()**.

strcat

Synopsis

```
#include <string.h>
```

```
char *strcat(char * restrict s1, const char * restrict s2);
```

Purpose

Appends a copy of one string to another string.

Return Value

The `strcat()` returns the value of `s1`.

Parameters

`s1` Pointer to a character argument.

`s2` Pointer to a character argument.

Description

The `strcat()` function appends a copy of the string pointed to by `s2` (including the terminating null character) to the end of the string pointed to by `s1`. The initial character of `s2` overwrites the null character at the end of `s1`. If copying takes place between objects that overlap, the behavior is undefined.

Example**Output****See Also**

`strconcat()`, `strjoin()`, `strncat()`.

strchr

Synopsis**#include** <string.h>**char** strchr(const char *s, int c)**Purpose**

Locates the first occurrence of a character in a string.

Return ValueThe **strchr()** function returns a pointer to the located character, or a null pointer if the character does not occur in the string.**Parameters***s* Pointer to a character argument.*c* Integer argument.**Description**The **strchr()** function locates the first occurrence of **c** (converted to a **char**) in the string pointed to by *s*. The terminating null character is considered to be part of the string.**Example**

```
/* a sample program that uses strchr to return the remaining
string after and including the first space */
#include <stdio.h>
#include <string.h>

int main() {
    char *p;
    int c;

    c = ' ';
    p = "this This is a test";
    printf("Original string: %s\n", p);
    printf("Character to search for: '%c'\n", (char)c);
    p = strchr(p,c);
    printf("After: %s\n",p);
}
```

Output

```
Original string: this This is a test
Character to search for: ' '
After: This is a test
```

See Also

strcmp

Synopsis

```
#include <string.h>
```

```
int strcmp(const char *s1, const char *s2);
```

Purpose

Compares two strings.

Return Value

The `strcmp()` function returns an integer greater than, equal to, or less than zero, accordingly as the string pointed to by `s1` is greater than, equal to, or less than the string pointed to by `s2`. The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of bytes that differ in the strings being compared.

Parameters

`s1` Pointer to a character argument.

`s2` Pointer to a character argument.

Description

The `strcmp()` function compares the string pointed to by `s1` to the string pointed to by `s2`.

Example**Output****See Also**

strcoll

Synopsis

```
#include <string.h>
```

```
int strcoll(const char *s1, const char *s2);
```

Purpose

Compares two strings.

Return Value

The `strcoll()` function returns an integer greater than, equal to, or less than zero, accordingly as the string pointed to by `s1` is greater than, equal to, or less than the string pointed to by `s2` when both are interpreted as appropriate to the current locale.

Parameters

`s1` Pointer to a character.

`s2` Pointer to a character.

Description

The `strcoll()` function compares the string pointed to by `s1` to the string pointed to by `s2`, both interpreted as appropriate to the `LC_COLLATE` category of the current locale.

Example**Output****See Also**

strconcat

Synopsis

```
#include <string.h>
```

```
char *strconcat(const char * string1, ...);
```

Purpose

Concatenates strings.

Return Value

The `strconcat()` returns a string with dynamically allocated memory.

Parameters

string1 Pointer to character argument.

Description

The `strconcat()` function concatenates all strings in the argument list, and puts the result into the returned string with a dynamically allocated memory. The dynamically allocated memory needs to be freed by the user.

Example

```
#include <string.h>
#include <stdarg.h>

int main() {
    char *buffer;
    char test1[90] = "abcd";

    buffer = strconcat(test1, "test2", "test3");
    printf("strconcat=%s\n",buffer);
    printf("test1=%s\n",test1);
    free (buffer);
    return 0;
}
```

Output

```
strconcat=abcdtest2test3
test1=abcd
```

See Also

`strcat()`, `strncat()`, `strjoin()`.

strcpy

Synopsis

```
#include <string.h>
```

```
char *strcpy(char * restrict s1, const char * restrict s2);
```

Purpose

Copies a string to an array.

Return Value

The `strcpy()` returns the value of `s1`.

Parameters

`s1` Integer pointer argument.

`s2` Integer pointer argument.

Description

The `strcpy()` function copies the string pointed to by `s2` (including the terminating null character) into the array pointed to by `s1`. If copying takes place between objects that overlap, the behavior is undefined.

Example

Output

See Also

strcspn

Synopsis

```
#include <string.h>
```

```
size_t *strcspn(const char *s1, const char *s2);
```

Purpose

Computes the length of the maximum initial segment of a string.

Return Value

The `strcspn()` function returns the length of the segment.

Parameters

s1 Pointer to a character argument.

s2 Pointer to a character argument.

Description

The `strcspn()` function computes the length of the maximum initial segment of the string pointed to by `s1` which consists entirely of characters *not* from the string pointed to by `s2`.

Example

```
/* a sample program that prints the number 8 which is
the index of the first character in first string. */
#include <stdio.h>
#include <string.h>

int main() {
    int len;

    len = strcspn("this is a test", "ab");
    printf("%d\n", len);
}
```

Output

```
8
```

See Also

strdup

Synopsis

```
#include <string.h>
```

```
char *strdup(const char * s);
```

Purpose

The function duplicates a string.

Return Value

The **strdup()** returns a pointer to the duplicate string.

Parameters

s Character pointer argument.

Description

The **strdup()** function creates a duplicate of a string pointed to by *s* and stores it in memory obtained by using **malloc**.

Example

```
/* a sample program that displays a string. */
#include <stdio.h>
#include <string.h>

int main() {
    char str[80], *p;

    strcpy(str, "this is a test");
    p = strdup(str); /* copy the content of str */
    if(p) {
        printf("%s\n", p);
        free(p);
    }
    else {
        printf("strdup() failed\n");
    }
}
```

Output

```
this is a test
```

See Also

strerror

Synopsis

```
#include <string.h>
char *strerror(int errnum);
```

Purpose

Maps values of **errnum**.

Return Value

The **strerror()** function returns a pointer to the string, the contents of which are locale-specific. The array pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the **strerror** function.

Parameters

errnum Integer argument.

Description

The **strerror()** function maps the number in **errnum** to a message string. Typically, the values for **errnum** come from **errno**, but **strerror** shall map any value of type **int** to a message.

The implementation shall behave as if no library function calls the **strerror** function.

Example

Output

See Also

strgetc

Synopsis

```
#include <string.h>
```

```
int strgetc(string_t & s, int i);
```

Purpose

Get a character in a specified position of a string.

Return Value

This function returns the character if successful, or EOF if failed.

Parameters

s A reference to a string.

i An integer indicating the position of character to be obtained in the string.

Description

This function returns the $(i+1)$ th character of the string *s*. This function is for Ch **string_t**.

Example

```
/** access element of a string */
#include <stdio.h>
#include <string.h>

int main() {
    string_t s = "ABCD";

    printf("strgetc(s, 1) = %c \n", strgetc(s, 1));
    strputc(s, 2, 'm');
    printf("s = %s\n", s);
}
```

Output

```
strgetc(s, 1) = B
s = ABmD
```

See Also

strputc().

strjoin

Synopsis

```
#include <string.h>
char *strjoin(const char * separator, ...);
```

Purpose

Combines strings to a string separated by the specified delimiter string.

Return Value

The `strjoin()` returns a string with dynamically allocated memory.

Parameters

separator Pointer to character argument.

Description

The `strjoin()` function combines all strings in the argument list, and puts the result into the returned string with dynamically allocated memory. The returned string is separated by the delimiter specified by the first argument **separator**. The dynamically allocated memory needs to be freed by the user.

Example

```
#include <string.h>
#include <stdarg.h>

int main() {
    char *buffer;

    buffer = strjoin("+", "test1", "test2", "test3");
    printf("strjoin =%s\n",buffer);
    free (buffer);
    return 0;
}
```

Output

```
strjoin =test1+test2+test3
```

See Also

`strcat()`, `strncat()`, `strconcat()`.

strlen

Synopsis

```
#include <string.h>
size_t strlen(const char *s);
```

Purpose

Computes the length of a string.

Return Value

The `strlen()` function returns the number of characters that precede the terminating null character.

Parameters

s Pointer to a character argument.

Description

The `strlen()` function computes the length of the string pointed to by *s*.

Example**Output****See Also**

strncasecmp

Synopsis

```
#include <string.h>
```

```
int strncasecmp(const char *s1, const char *s2, size_t n);
```

Purpose

Compares characters between two arrays.

Return Value

The `strncasecmp()` function returns an integer greater than, equal to, or less than zero, accordingly as the possibly null-terminated array pointed to by `s1` is greater than, equal to, or less than the possibly null-terminated array pointed to by `s2`. Before the arguments are compared, all of the characters are converted to lowercase by `tolower()`.

Parameters

`s1` Pointer to a character argument.

`s2` Pointer to a character argument.

`n` Unsigned integer argument.

Description

The `strncasecmp()` function compares not more than `n` characters (characters that follow a null character are not compared) from the array pointed to by `s1` to the array pointed to by `s2`.

Example

```
/* a sample program that shows the outcome of a comparison
of its two strings.*/
#include <stdio.h>
#include <string.h>

int main() {
    char s1[] = "abcdefG8";
    char s2[] = "abcdefg";
    int outcome;
    int l1,l2;
    int len;

    len = (l1 = strlen(s1)) < (l2 = strlen(s2)) ? l1:l2;
    outcome = strncasecmp(s1,s2,len);
    if(! outcome) printf("equal\n");
    else if(outcome < 0) printf("first less than second\n");
    else printf("first greater than second\n");
}
```

Output

```
equal
```

See Also

strncat

Synopsis

```
#include <string.h>
```

```
char *strncat(char * restrict s1, const char * restrict s2, size_t n);
```

Purpose

Appends characters from an array to a wide string.

Return Value

The `strncat()` returns the value of `s1`.

Parameters

`s1` Pointer to character argument.

`s2` Pointer to character argument.

`n` Unsigned integer argument.

Description

The `strncat()` function appends not more than `n` characters (a null character and characters that follow it are not appended) from the array pointed to by `s2` to the end of the string pointed to by `s1`. The initial character of `s2` overwrites the null character at the end of `s1`. A terminating null character is always appended to the result. If copying takes place between objects that overlap, the behavior is undefined.

Example

Output

See Also

`strcat()`, `strconcat()`, `strjoin()`.

strncmp

Synopsis

```
#include <string.h>
```

```
int strncmp(const char *s1, const char *s2, size_t n);
```

Purpose

Compares characters between two arrays.

Return Value

The **strncmp()** function returns an integer greater than, equal to, or less than zero, accordingly as the possibly null-terminated array pointed to by **s1** is greater than, equal to, or less than the possibly null-terminated array pointed to by **s2**.

Parameters

s1 Pointer to a character argument.

s2 Pointer to a character argument.

n Unsigned integer argument.

Description

The **strncmp()** function compares not more than **n** characters (characters that follow a null character are not compared) from the array pointed to by **s1** to the array pointed to by **s2**.

Example

```
/* a sample program that shows the outcome of a comparison
of its two strings.*/
#include <stdio.h>
#include <string.h>

int main() {
    char s1[] = "abcdefghijk1";
    char s2[] = "abcdefk1";
    int outcome;
    int l1, l2;
    int len;

    len = (l1 = strlen(s1)) < (l2 = strlen(s2)) ? l1:l2;
    outcome = strncmp(s1, s2, len);
    if(! outcome) printf("equal\n");
    else if(outcome < 0) printf("first less than second\n");
    else printf("first greater than second\n");
}
```

Output

```
first less than second
```

See Also

strncpy

Synopsis

```
#include <string.h>
char *strncpy(char * restrict s1, const char * restrict s2, size_t n);
```

Purpose

Copies characters from one array to another array.

Return Value

The `strncpy()` returns the value of `s1`.

Parameters

`s1` Character pointer argument.
`s2` Character pointer argument.
`n` Unsigned integer argument.

Description

The `strncpy()` function copies not more than `n` characters (characters that follow a null character are not copied) from the array pointed to by `s2` to the array pointed to by `s1`. If copying takes place between objects that overlap, the behavior is undefined.

If the array pointed to by `s2` is a string that is shorter than `n` characters, null characters are appended to the copy in the array pointed to by `s1`, until `n` characters in all have been written.

Example

Output

See Also

strpbrk

Synopsis

```
#include <string.h>
```

```
char *strpbrk(const char *s1, const char *s2);
```

Purpose

Locates first occurrence of a character from one string in another string.

Return Value

The `strpbrk()` function returns a pointer to the character, or a null pointer if no character from `s2` occurs in `s1`.

Parameters

`s1` Pointer to a character argument.

`s2` Pointer to a character argument.

Description

The `strpbrk()` function locates the first occurrence in the string pointed to by `s1` of any character from the string pointed to by `s2`.

Example

```
/* a sample program that prints the message 's is a test'
on the screen. */
#include <stdio.h>
#include <string.h>

int main() {
    char *p;

    p = strpbrk("this is a test", " absj");
    printf("%s\n", p);
}
```

Output

```
s is a test
```

See Also

strputc

Synopsis

```
#include <string.h>
```

```
int strputc(string_t &s, int i, char c);
```

Purpose

Replace a character in a specified position of a string with another character.

Return Value

This function returns 0 on successful and -1 in failure.

Parameters

s Reference to the string.

i Input integer indicating the position of the character to be replaced.

c Input character to be used to replace the character in the string.

Description

This function replaces the $(i+1)$ th character of string *s* with character *c*. This function is for Ch **string_t**

Example

See **strgetc**.

See Also

strgetc().

strchr

Synopsis

```
#include <string.h>
```

```
char *strchr(const char *s, int c);
```

Purpose

Locates last occurrence of a character in a string.

Return Value

The `strchr()` function returns a pointer to the character, or a null pointer if `c` does not occur in the string.

Parameters

s Pointer to a character.

c Integer argument.

Description

The `strchr()` function locates the last occurrence of `c` (converted to a `char`) in the string pointed to by `s`. The terminating null character is considered to be part of the string.

Example

```
/* a sample program that prints the message 'is a test'
on the screen. */
#include <stdio.h>
#include <string.h>

int main() {
    char *p;

    p = strchr("this is a test",'i');
    printf("%s\n",p);
}
```

Output

```
is a test
```

See Also

strrep

Synopsis

```
#include <string.h>
```

```
string_t strrep(string_t s1, string_t s2, string_t s3);
```

Purpose

Search and replace strings inside another string.

Return Value

This function returns the new string in which all matched strings are replaced.

Parameters

s1 A string in which the string *s2* will be searched and replaced.

s2 A string to be searched.

s3 A string used to replace the string *s2*.

Description

This function searches and replaces the string *s2* with string *s3* inside the string *s1*. The replaced string is returned and string *s1* is not changed.

Example

```
#include <stdio.h>
#include <string.h>

int main() {
    string_t s, s1 = "He is smart, but you are smarter",
            s2 = "smart", s3="great", s4="tall";

    printf("s1 = %s\n", s1);
    s = strrep(s1, s2, s3);
    printf("s1 = %s\n", s1);
    printf("s = %s\n", s);
    s = strrep(s1, s2, s3);
    printf("s1 = %s\n", s1);
    printf("s = %s\n", s);
    s = strrep(s1, s2, s4);
    printf("s1 = %s\n", s1);
    printf("s = %s\n", s);
}
```

Output

```
s1 = He is smart, but you are smarter
s1 = He is smart, but you are smarter
s = He is great, but you are greater
s1 = He is smart, but you are smarter
s = He is great, but you are greater
s1 = He is smart, but you are smarter
s = He is tall, but you are taller
```

See Also

[strgetc\(\)](#).

strspn

Synopsis

```
#include <string.h>
```

```
size_t strspn(const char *s1, const char *s2);
```

Purpose

Computes the length of the maximum initial segment of a string.

Return Value

The `strspn()` function returns the length of the segment.

Parameters

s1 Pointer to a character argument.

s2 Pointer to a character argument.

Description

The `strspn()` function computes the length of the maximum initial segment of the string pointed to by `s1` which consists entirely of characters from the string pointed to by `s2`.

Example

```
/* a sample program that prints the index of the first
character in the string pointed to by str1 that does not
match any of the characters in str2.*/
#include <stdio.h>
#include <string.h>

int main() {
    unsigned int len;

    len = strspn("this is a test", "siht ");
    printf("%d\n", len);
}
```

Output

8

See Also

strstr

Synopsis

```
#include <string.h>
```

```
char *strstr(const char *s1, const char *s2);
```

Purpose

Locates the first occurrence of an object in one string in the other string.

Return Value

The `strstr()` function returns a pointer to the located string, or a null pointer if the string is not found. If `s2` points to a string with zero length, the function returns `s1`.

Parameters

`s1` Pointer to a character argument.

`s2` Pointer to a character argument.

Description

The `strstr()` function locates the first occurrence in the string pointed to by `s1` of the sequence of characters (excluding the terminating null character) in the string pointed to by `s2`.

Example

```
/* a sample program that prints 'is is a test'. */
#include <stdio.h>
#include <string.h>

int main() {
    char *p;

    p = strstr("this is a test", "is");
    printf("%s\n", p);
}
```

Output

```
is is a test
```

See Also

strtok_r

Synopsis

```
#include <string.h>
char *strtok_r(char * s, const char * sep, char **lasts);
```

Purpose

Finds all of the character strings that are separated by spaces.

Return Value

The `strtok_r()` function returns a pointer to the token found, or a null pointer if there is no token.

Parameters

s Pointer to a character argument.
sep Pointer to a character argument.
lasts Pointer to a user-provided pointer.

Description

The function considers the null-terminated string *s* as a sequence of zero or more text tokens separated by one or more characters from the string *sep*. *lasts* points to stored information necessary for `/strtok_r/` to continue scanning the same string.

In the first call, *s* points to a null-terminated string, *sep* points to a null-terminated string of separator characters, and *lasts* is ignored. The function returns a pointer to the first character of the token, writes a null character into *s* followed by the returned token, and updates the pointer to which *lasts* points.

In subsequent calls, *s* is a **NULL** pointer and *lasts* is unchanged from previous calls so that subsequent calls will move through the string *s*, returning successive tokens until no tokens remain. The separator string *sep* may vary from call to call. When no tokens remain in *s*, a **NULL** pointer is returned.

Example

```
/* a sample program that tokenizes the string, "The summer soldier,
the sunshine patriot" with spaces and commas being the delimiters.
The output is The|summer|soldier|the sunshine|patriot. */
#include <stdio.h>
#include <string.h>

int main() {
    char *token, *str1, *delimiter;
    char *endptr = NULL;

    str1 = "The summer soldier, the sunshine patriot";
    delimiter = " ,";
    for(token = strtok_r(str1, delimiter, &endptr); token != NULL;
        token = strtok_r(NULL, delimiter, &endptr)) {
        printf("token = %s\n", token);
    }
}
```

Output

```
token = The
token = summer
token = soldier
token = the
token = sunshine
token = patriot
```

See Also

strtok

Synopsis

```
#include <string.h>
char *strtok(char * restrict s1, const char * restrict s2);
```

Purpose

Breaks a string into a sequence of tokens.

Return Value

The **strtok()** function returns a pointer to the first character of a token, or a null pointer if there is no token.

Parameters

s1 Pointer to a character argument.
s2 Pointer to a character argument.

Description

A sequence of calls to the **strtok()** function breaks the string pointed to by **s1** into a sequence of tokens, each of which is delimited by a character from the string pointed to by **s2**. The first call in the sequence has a non-null first argument; subsequent calls in the sequence have a null first argument. The separator string pointed to by **s2** may be different from call to call.

The first call in the sequence searches the string pointed to by **s1** for the first character that is *not* contained in the current separator string pointed to by **s2**. If no such character is found, then there are no tokens in the string pointed to by **s1** and the **strtok** function returns a null pointer. If such a character is found, it is the start of the first token.

The **strtok()** function then searches from there for a character that *is* contained in the current separator string. If no such character is found, the current token extends to the end of the string pointed to by **s1**, and the subsequent searches for a token will return a null pointer. If such a character is found, it is overwritten by a null character, which terminates the current token. The **strtok** function saves a pointer to the following character, from which the next search for a token will start.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

The implementation shall behave as if no library function calls the **strtok** function.

Example

```
/* a sample program that tokenizes the string, "The summer soldier,
the sunshine patriot" with spaces and commas being the delimiters.*/
#include <stdio.h>
#include <string.h>

int main() {
    char *token, *str1, *delimit;

    str1 = "The summer soldier, the sunshine patriot";
    delimit = " ,";
    for(token = strtok(str1, delimit); token != NULL;
        token = strtok(NULL, delimit)) {
        printf("token = %s\n", token);
    }
}
```

Output

```
token = The
```

```
token = summer  
token = soldier  
token = the  
token = sunshine  
token = patriot
```

See Also

strxfrm

Synopsis

```
#include <string.h>
```

```
size_t strxfrm(char * restrict s1, const char * restrict s2, size_t n);
```

Purpose

Transform a string and place it in an array.

Return Value

The **strxfrm()** function returns the length of the transformed string (not including the terminating null character). If the value returned is **n** or greater, the contents of the array pointed to by **s1** are indeterminate.

Parameters

s1 Pointer to a character argument.

s2 Pointer to a character argument.

n Unsigned integer argument.

Description

The **strxfrm()** function transforms the string pointed to by **s2** and places the resulting string into the array pointed to by **s1**. The transformation is such that if the **strcmp** function is applied to two transformed strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the **strcmp** function applied to the same two original strings. No more than **n** characters are placed into the resulting array pointed to by **s1**, including the terminating null character, if **n** is permitted to be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

Example

Output

See Also

Chapter 20

Date and Time Functions — `<time.h>`

The header `time.h` defines one macro, and declares several types and functions for manipulating time. Many functions deal with a *calender time* that represents the current date (according to the Gregorian calender) and time. Some functions deal with *local time*, which is the calender time expressed for some specific time zone, and with *Daylight Saving Time*, which is a temporary change in the algorithm for determining local time. The local timezone and Daylight Saving Time are implementation-defined.

The macro defined is `CLOCKS_PER_SEC` which expands to a constant expression with type `clock_t` described below, and which is the number per second of the value returned by the `clock` function.

The types declared are `size_t`;

`clock_t`

and

`time_t`

which are arithmetic types capable of representing times; and

`struct tm`

which holds the components of a calender time, called the *broken-down time*.

The `tm` structure shall contain at least the following members, in any order. The semantics of the members and their normal ranges are expressed in the comments.

```
int tm_sec; // seconds after the minute—[0,60]
int tm_min; // minutes after the hour—[0,59]
int tm_hour; // hours since midnight—[0,23]
int tm_mday; // day of the month—[1,31]
int tm_mon; // months since January—[0,11]
int tm_year; // years since 1900
int tm_wday; // days since Sunday—[0,6]
int tm_yday; // days since January 1—[0,365]
int tm_isdst; // Daylight Saving Time flag
```

The value of `tm_isdst` is positive if Daylight Saving Time is in effect, zero if Daylight Saving Time is not in effect, and negative if the information is not available.

Public Data

None.

Functions

The following functions are implemented using the **time** header.

Functions	Description
asctime()	Converts broken-down time into a string.
clock()	Determines the processor time used.
ctime()	Converts from calendar time to local time.
difftime()	Computes the difference between two calendar times.
gmtime()	Converts from calendar time to broken time.
localtime()	Converts from calendar time to broken-down time.
mktime()	Converts broken-down time to calendar time.
strftime()	Places characters into an array.
time()	Determines the current calendar time.

Macros

The following macros are defined for the **time** header.

Macro	Description
CLOCKS_PER_SEC	Holds the value of number per second of the value returned.

Declared Types

The following types are declared in the **time** header.

Declared Types	Description
size_t	Unsigned integer.
clock_t	Arithmetic type capable of representing times.
time_t	Arithmetic type capable of representing times.
struct tm	Structure which holds various date and time elements.

Portability

This header has no known portability problem.

asctime

Synopsis

```
#include <time.h>
```

```
char * asctime(const struct tm *timeptr);
```

Purpose

Converts broken-down time into a string.

Return Value

The `asctime()` function returns a pointer to the string.

Parameters

timeptr Pointer to a structure of type `tm`

Description

The `asctime()` function converts the broken-down time in the structure pointed to by *timeptr* into a string in the form

```
Sun Sep 16 01:03:52 1973\n\n0
using the equivalent of the following algorithm.
```

```
char *asctime(const struct tm *timeptr)
{
    static const char wday_name[7][3] = {
        "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
    };
    static const char mon_name[12][3] = {
        "Jan", "Feb", "Mar", "Apr", "May", "Jun",
        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
    };
    static char result[26];

    sprintf(result, "%.3s %.3s%3d %.2d:%.2d%.2d %d\n",
        wday_name[timeptr->tm_wday],
        mon_name[timeptr->tm_mon],
        timeptr->tm_mday, timeptr->tm_hour,
        timeptr->tm_min, timeptr->tm_sec,
        1900 + timeptr->tm_year);
    return result;
}
```

Example 1

```
/* a sample program that return local time. */
#include <stdio.h>
#include <time.h>

int main() {
    struct tm *ptr;
    time_t lt;

    lt = time(NULL); //return system time
    printf("time(NULL) = %d\n", lt);
    ptr = localtime(&lt); // return time in the form of tm structure
    printf(asctime(ptr));
}
```

Output

```
time(NULL) = 941830207
Fri Nov  5 14:30:07 1999
```

Example 2

```
/* a sample program that return local time. */
#include <stdio.h>
#include <time.h>
#define SIZE 26

int main() {
    struct tm *ptr;
    time_t lt;
    char buf[26];

    lt = time(NULL);          //return system time
    printf("time(NULL) = %d\n", lt);
    ptr = localtime(&lt);    // return time in the form of tm structure
    asctime_r(ptr,buf,SIZE);
    printf(buf);
}
```

Output

```
time(NULL) = 941830207
Fri Nov  5 14:30:07 1999
```

See Also

clock

Synopsis

```
#include <time.h>
clock_t clock(void);
```

Purpose

Determines the processor time used.

Return Value

The `clock()` function returns the implementation's best approximation to the processor time used by the program since the beginning of an implementation-defined era related only to the program invocation. To determine the time in seconds, the value returned by the `clock()` function should be divided by the value of the macro `CLOCKS_PER_SEC`. If the processor time used is not available or its value cannot be represented, the function returns the value `(clock_t) - 1`.

Parameters

No argument.

Description

The `clock()` function determines the processor time used.

Example

```
/* a sample program that displays the current execution time. */
#include <stdio.h>
#include <time.h>
void elapsed_time(void);

int main() {
    int i;

    clock();    /// first call clock
    for(i = 0; i<100000;i++) {};
    elapsed_time();
}

void elapsed_time(void) {
    clock_t time;

    time = clock();
    printf("Elapsed time: %d micro seconds or %f seconds.\n",time,
          (double)time/CLOCKS_PER_SEC); // return the time between first call
                                        // and this call .
}
```

Output

```
Elapsed time: 30000 microsecs.
```

ctime

Synopsis

```
#include <time.h>
char * ctime(const time_t *timer);
```

Purpose

Converts from calendar time to local time.

Return Value

The **ctime()** function returns the pointer returned by the **asctime()** function with that broken-down time as the argument.

Parameters

timer Variable pointer used to hold a calendar time.

Description

The **ctime()** function converts the calendar time pointed to by **timer** to local time in the form of a string. It is equivalent to

```
asctime(localtime(timer))
```

Example

```
/* a sample program that return local time. */
#include <stdio.h>
#include <time.h>

int main() {
    time_t *timer;

    timer = malloc(sizeof(time_t));
    *timer = time(NULL); //return system time
    printf("%s", ctime(timer));
}
```

Output

```
Fri Nov 5 14:30:09 1999
```

See Also

difftime

Synopsis

```
#include <time.h>
```

```
double difftime(time_t time1, time_t time0);
```

Purpose

Computes the difference between two calendar times.

Return Value

The `difftime()` function returns the difference expressed in seconds as a **double**.

Parameters

time1 Variable used to hold a calendar time.

time0 Variable used to hold a calendar time.

Description

The `difftime()` function computes the difference between two calendar times: *time1* – *time0*.

Example

```
/* a sample program that returns the difference in
seconds between two calendar times. */
#include <stdio.h>
#include <time.h>

int main() {
    time_t time0, time1;
    int i;

    time0 = time(NULL);          /* return system time */
    for(i = 0; i < 100000; i++) {};
    time1 = time(NULL);
    printf("the 100000 loops time is %f sec\n", difftime(time1, time0));
}
```

Output

```
the 100000 loops time is 8.000000 sec
```

See Also

gmtime

Synopsis

```
#include <time.h>
```

```
struct tm * gmtime(const time_t *timer);
```

Purpose

Converts from calendar time to broken time.

Return Value

The `gmtime()` function returns a pointer to the broken-down time, or a null pointer if the specified time cannot be converted to UTC.

Parameters

timer Variable pointer used to hold a calendar time.

Description

The `gmtime()` function converts the calendar time pointed to by *timer* into a broken-down time, expressed as UTC.

Example

```
/* a sample program that prints both the local time and
the UTC of the system. */
#include <stdio.h>
#include <time.h>

int main() {
    struct tm *local_time, *gm;
    time_t t;

    t = time(NULL);
    local_time = localtime(&t);
    printf("Local time and date: %s\n", asctime(local_time));
    gm = gmtime(&t);
    printf("Coordinated Universal Time and date: %s\n", asctime(gm));
}
```

Output

```
Local time and date: Fri Nov  5 14:30:18 1999
```

```
Coordinated Universal Time and date: Fri Nov  5 19:30:18 1999
```

See Also

localtime

Synopsis

```
#include <time.h>
```

```
char * localtime(const time_t *timer);
```

Purpose

Converts from calendar time to broken-down time.

Return Value

The `localtime()` function returns a pointer to the broken-down time, or a null pointer if the specified time cannot be converted to local time.

Parameters

timer Variable pointer used to hold a calendar time.

Description

The `localtime()` function converts the calendar time pointed to by *timer* into a broken-down time, expressed as local time.

Example

```
/* a sample program that prints both the local time and
the UTC of the system. */
#include <stdio.h>
#include <time.h>

int main() {
    struct tm *local_time, *gm;
    time_t t;

    t = time(NULL);
    local_time = localtime(&t);
    printf("Local time and date: %s\n", asctime(local_time));
    gm = gmtime(&t);
    printf("Coordinated Universal Time and date: %s\n", asctime(gm));
}
```

Output

```
Local time and date: Fri Nov  5 14:30:19 1999
```

```
Coordinated Universal Time and date: Fri Nov  5 19:30:19 1999
```

See Also

mktime

Synopsis

```
#include <time.h>
```

```
time_t mktime(struct tm *timeptr);
```

Purpose

Converts broken-down time to calendar time.

Return Value

The **mktime()** function returns the specified calendar time encoded as a value of type **time_t**. If the calendar time cannot be represented, the function returns the value **(time_t) - 1**.

Parameters

timeptr Pointer to a structure of type **tm**

Description

The **mktime()** function converts the broken-down time, expressed as local time, in the structure pointed to by *timeptr* into a calendar time value with the same encoding as that of the values returned by the **time()** function. the original **tm_wday** and **tm_yday** components of the structure are ignored, and the original values of the other components are not restricted to the ranges indicated above. On successful completion, the values of the **tm_wday** and **tm_yday** components of the structure are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to the ranges indicated above; the final value of the **tm_mday** is not set until **tm_mon** and **tm_year** are determined.

If the call is successful, a second call to the **mktime()** function with the resulting **struct tm** value shall always leave it unchanged and return the same value as the first call. Furthermore, if the normalized time is exactly representable as a **time_t** value, then the normalized broken-down time and the broken-down time generated by converting the result of the **mktime()** function by a call to **localtime()** shall be identical.

Example

```
/* a sample program that shows the use of mktime() */
#include <stdio.h>
#include <time.h>

int main() {
    struct tm t;
    time_t t_of_day;

    t.tm_year = 1999-1900;
    t.tm_mon = 6;
    t.tm_mday = 12;
    t.tm_hour = 2;
    t.tm_min = 30;
    t.tm_sec = 1;
    t.tm_isdst = 0;
    t_of_day = mktime(&t);
    printf(ctime(&t_of_day));
}
```

Output

```
Mon Jul 12 03:30:01 1999
```

strptime

Synopsis

```
#include <time.h>
```

```
char strptime(char * restrict s, size_t maxsize, const char * restrict format, const struct tm * restrict timeptr);
```

Purpose

Places characters into an array.

Return Value

If the total number of resulting characters including the terminating null character is not more than *maxsize*, the **strptime()** function returns the number of characters placed into the array pointed to by *s* not including the terminating null character. Otherwise, zero is returned and the contents of the array are indeterminate.

Parameters

s Character pointer argument.

maxsize Unsigned integer argument.

format Character pointer argument.

timeptr Pointer to structure.

Description

The **strptime()** function places characters into the array pointed to by *s* as controlled by the string pointed to by *format*. The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The *format* string consists of zero or more conversion specifiers and ordinary multibyte characters. A conversion specifier consists of a **%** character, possibly followed by an **E** or **O** modifier character (described below), followed by a character that determines the behavior of the conversion specifier. All ordinary multibyte characters (including the terminating null character) are copied unchanged into the array. If copying takes place between objects that overlap, the behavior is undefined. No more than *maxsize* characters are placed into the array.

Each conversion specifier is replaced by appropriate characters as described in the following list. The appropriate characters are determined using the **LC_TIME** category of the current locale and by the values of zero or more members of the broken-down time structure pointed to by **timeptr**, as specified in brackets in the description. If any of the specified values is outside the normal range, the characters stored are unspecified.

%a is replaced by the locale's abbreviated weekday name. [**tm_wday**]

%A is replaced by the locale's full weekday name. [**tm_wday**]

%b is replaced by the locale's abbreviated month name. [**tm_mon**]

%B is replaced by the locale's full month name. [**tm_mon**]

%c is replaced by the locale's appropriate date and time representation.

%C is replaced by the year divided by 100 and truncated to an integer, as a decimal number (**00–99**). [**tm_year**]

%d is replaced by the day of the month as a decimal number (**01–31**). [**tm_mday**]

%D is equivalent to “**%m\%d\%y**”. [**tm_mon, tm_mday, tm_year**]

%e is replaced by day of the month as a decimal number (**01–31**); a single digit is preceded by a space. [**tm_mday**]

%F is equivalent to “**%Y – %m – %d**” (the ISO 8601 date format). [**tm_year, tm_mon, tm_mday**]

%g is replaced by the last two digits of the week-based year (see below) as a decimal number (**00–99**). [**tm_year**, **tm_wday**, **tm_yday**]

%G is replaced by the week-based year (see below) as a decimal number (e.g., 1997). [**tm_year**, **tm_wday**, **tm_yday**]

%h is equivalent to “**%b**”. [**tm_mon**]

%H is replaced by the hour (24-hour clock) as a decimal number (**01–23**). [**tm_hour**]

%I is replaced by the hour (12-hour clock) as a decimal number (**01–12**). [**tm_hour**]

%j is replaced by the day of the year as a decimal number (**001–366**). [**tm_yday**]

%m is replaced by the month as a decimal number (**01–12**). [**tm_mon**]

%M is replaced by the minute as a decimal number (**00–59**). [**tm_min**]

%n is replaced by a new-line character.

%p is replaced by the locale’s equivalent of the AM/PM designations associated with a 12-hour clock. [**tm_hour**]

%r is replaced by the locale’s 12-hour clock time. [**tm_hour**, **tm_min**, **tm_sec**]

%R is equivalent to “**%H:%M**”. [**tm_hour**, **tm_min**]

%S is replaced by the second as a decimal number (**00–60**). [**tm_sec**]

%t is replaced by a horizontal-tab character.

%T is equivalent to “**%H:%M:%S**” (the ISO 8601 time format). [**tm_hour**, **tm_min**, **tm_sec**]

%u is replaced by the ISO 8601 weekday as a decimal number (**1–7**), where Monday is 1. [**tm_wday**]

%U is replaced by the week number of the year (the first Sunday as the first day of week 1) as a decimal number (**00–53**). [**tm_year**, **tm_wday**, **tm_yday**]

%V is replaced by the ISO 8601 week number (see below) as a decimal number (**01–53**). [**tm_year**, **tm_wday**, **tm_yday**]

%w is replaced by the weekday as a decimal number (**0–6**), where Sunday is 0. [**tm_wday**]

%W is replaced by the week number of the year (the first Monday as the first day of week 1) as a decimal number (**00–53**). [**tm_year**, **tm_wday**, **tm_yday**]

%x is replaced by the locale’s appropriate date representation.

%X is replaced by the locale’s appropriate time representation.

%y is replaced by the last 2 digits of the year as a decimal number (**00–99**). [**tm_year**]

%Y is replaced by the year as a decimal number (e.g., 1997). [**tm_year**]

%z is replaced by the offset from UTC in the ISO 8601 format “-0430” (meaning 4 hours 30 minutes behind UTC, west of Greenwich), or by no characters if no time zone is determinable. [**tm_isdst**]

%Z is replaced by the locale’s time zone name or abbreviation, or by no characters if no time zone is determinable. [**tm_isdst**]

%% is replaced by **%**.

Some conversion specifiers can be modified by the inclusion of an **E** or **O** modifier character to indicate an alternative format or specification. If the alternative format or specification does not exist for the current locale, the modifier is ignored.

%Ec is replaced by the locale’s alternative date and time representation.

%EC is replaced by the name of the base year (period) in the locale’s alternative representation.

%Ex is replaced by the locale’s alternative date representation.

%EX is replaced by the locale’s alternative time representation.

%Ey is replaced by the offset from **%EC** (year only) in the locale’s alternative representation.

%EY is replaced by the locale’s full alternative year representation.

%Od is replaced by the day of the month, using the locale’s alternative numeric symbols (filled as needed with leading zeros, or with leading spaces if there is no alternative symbol for zero).

%Oe is replaced by the day of the month, using the locale’s alternative numeric symbols (filled as needed with leading spaces).

%OH is replaced by the hour (24-hour clock), using the locale’s alternative numeric symbols.

%OI is replaced by the hour (12-hour clock), using the locale’s alternative numeric symbols.

%Om is replaced by the month, using the locale’s alternative numeric symbols.

%OM is replaced by the minutes, using the locale’s alternative numeric symbols.

%OS is replaced by the seconds, using the locale’s alternative numeric symbols.

%Ou is replaced by the ISO 8601 weekday as a number in the locale’s alternative representation, where Monday is 1.

%OU is replaced by the week number, using the locale’s alternative numeric symbols.

%OV is replaced by the ISO 8601 week number, using the locale’s alternative numeric symbols.

%Ow is replaced by the weekday as a number, using the locale’s alternative numeric symbols.

%OW is replaced by the week number of the year, using the locale’s alternative numeric symbols.

%Oy is replaced by the last 2 digits of the year, using the locale’s alternative numeric symbols.

%g, **%G**, and **%V** give values according to the ISO 8601 week-based year. In this system, weeks begin on a Monday and week 1 of the year is the week that includes January 4th, which is also the week that includes the first Thursday of the year, and is also the first week that contains at least four days in the year. If the first Monday of January is the 2nd, 3rd, or 4th, the preceding days are part of the last week of the preceding year; thus, for Saturday 2nd January 1999, **%G** is replaced by **1998** and **%V** is replaced by **53**. If December 29th, 30th, or 31st is a Monday, it and any following days are part of week 1 of the following year. Thus, for Tuesday 30th December 1997, **%G** is replaced by **1998** and **%V** is replaced by **1**.

If a conversion specifier is not one of the above, the behavior is undefined.

In the “**C**” locale, the **E** and **O** modifiers are ignored and the replacement strings for the following specifiers are:

%a the first three characters of **%A**.

%A one of “**Sunday**”, “**Monday**”, ... , “**Saturday**”.

%b one of the first three characters of **%B**.

%B one of “**January**”, “**February**”, ... , “**December**”.

%c equivalent to “**%A %B %d %T %Y**”.

%p one of “**am**” or “**pm**”.

%r equivalent to “**%I:%M:%S:%p**”.

%x equivalent to “**%A:%B:%d:%Y**”.

%X equivalent to “**%T**”.

%Z implementation-defined.

Example

```
/* a sample program that displays current time. */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    struct tm *ptr;
    time_t ltime;
    char str[80];

    ltime = time(NULL); /* get current calendar time */
    ptr = localtime(&ltime); // return time in the form of tm structure
    strftime(str, 80, "It is now %H:%M %p.", ptr);
    printf("%s\n", str);
}
```

Output

It is now 14:30 PM.

time

Synopsis

```
#include <time.h>
time_t time(time_t timer);
```

Purpose

Determines the current calendar time.

Return Value

The `time()` function returns the implementation's best approximation to the current calendar time. The value `(time_t) -1` is returned if the calendar time is not available. If `timer` is not a null pointer, the return value is also assigned to the object it points to.

Parameters

timer Variable used to hold a calendar time.

Description

The `time()` function determines the current calendar time. The encoding of the value is unspecified.

Example

See `asctime()`.

Output

Chapter 21

Extended Multibyte and Wide Character Functions — `<wchar.h>`

The header `wchar.h` declares four data types, one tag, four macros, and many functions.

The types declared are `wchar_t` and `size_t` described in Chapter 16;

`mbstate_t`

which is an object type other than an array type that can hold the conversion state information necessary to convert between sequences of multibyte characters and wide characters;

`wint_t` described in Chapter 22;

and

`struct tm`

which is declared as an incomplete structure type. the contents of which are described in Chapter 20;

The macros defined are

`WCHAR_MAX`

which is the maximum value representable by an object of type `wchar_t`;

`WCHAR_MIN`

which is the minimum value representable by an object of type `wchar_t`; and

`WEOF` described in Chapter 22.

The functions declared are grouped as follows:

- Functions that perform input and output of wide characters, or multibyte characters, or both;
- Functions that provide wide-string numeric conversions;
- Functions that perform general wide-string manipulation;
- Functions for wide-string date and time conversions; and
- Functions that provide extended capabilities for conversion between multibyte and wide-character sequences.

Unless explicitly stated otherwise, if the execution of a function described in this subclause causes copying to take place between objects that overlap, the behavior is undefined.

Public Data

None.

Functions

The following functions are implemented using the **wchar** header.

Functions	Description
_w fopen()	Opens a file.
btowc()	Determines if character is multibyte character.
fgetwc()	Obtains wide character and advances one position.
fgetws()	Reads wide characters from stream.
fputwc()	Writes the wide character to the output stream.
fputws()	Writes a wide string to stream.
fwide()	Determines orientation of stream.
getwc()	Gets wide character and advances pointer.
getwchar()	Gets wide character and advances file position indicator.
mbrlen()	Determines the number of bytes in a multibyte character.
mbrtowc()	Determines the number of bytes needed to complete the next multibyte character.
mbsinit()	Determines if pointed-to object describes an initial conversion state.
mbsrtowcs()	Converts a sequence of multibyte characters from an array into a sequence.
putwc()	Writes wide character to output stream.
putwchar()	Writes wide character to output stream.
ungetwc()	Pushes wide character back onto input stream.
wcrtomb()	Determines the number of bytes needed to represent the next multibyte character.
wscat()	Appends a copy of one wide string to another wide string.
wcschr()	Locates first occurrence of a wide character in a string.
wscmp()	Compares two wide strings.
wscoll()	Compares two wide strings.
wscpy()	Copies a wide string to an array.
wcsspn()	Computes the length of the maximum initial segment of a wide string.
wcsftime()	Places wide characters into an array.
wcslen()	Computes the length of a wide string.
wcsncat()	Appends wide characters from an array to a wide string.
wcsncmp()	Compares wide characters between two arrays.
wcsncpy()	Copies wide characters from one array to another array.
wcspbrk()	Locates first occurrence of a wide character from one string in another string.
wcsrchr()	Locates last occurrence of a wide character in a wide string.
wcsrtombs()	Converts sequence of wide characters from array into a sequence of multibyte characters.
wcsspn()	Computes the length of the maximum initial segment of a wide string.
wcsstr()	Locates the first sequence found in one wide string in the other wide string.
wcstod()	Converts initial portion of wide string to double, float, and long double.
wcstok()	Breaks a wide string into a sequence of tokens.
wcstol()	Converts wide string to long int, long long int, unsigned long int, or unsigned long long int.
wcsxfrm()	Transforms wide string and places it in an array.
wctob()	Determines if character corresponds to member of extended character set.
wmemchr()	Locates the first occurrence of a specified wide character in a stream.
wmemcmp()	Compares the first wide characters in two objects.
wmemcpy()	Copies wide characters from one object to another.
wmemmove()	Copies wide characters from one object to another.
wmemset()	Copies a single value into specified number of locations in an object.

Macros

The following macros are defined for the **wchar** header.

Macro	Description
WCHAR_MAX	Maximum value representable by object of type wchar_t .
WCHAR_MIN	Minimum value representable by object of type wchar_t .
WEOF	Used to indicate <i>end-of-file</i> .

Declared Types

The following types are declared in the **wchar** header.

Declared Types	Description
wchar_t	int-can hold codes for all the extended character set.
size_t	unsigned int
wint_t	int-holds values corresponding to members of the extended set.
mbstate_t	object type-holds conversion state information.
struct tm	incomplete structure type-holds the components of calendar time.

Portability

The following functions are not supported in the current release of Ch: **fwprintf()**, **fwscanf()**, **swprintf()**, **swscanf()**, **vfwprintf()**, **vfwscanf()**, **vswprintf()**, **vswscanf()**, **vwprintf()**, **vwscanf()**, **wprintf()**, **wscanf()**.

_w fopen

Synopsis**#include** <stdio.h>**FILE** ***_w fopen**(const wchar_t **filename*, const wchar_t **mode*);**Purpose**

Open a file.

Return ValueThe **_w fopen** function returns a pointer to the object controlling the stream. If the open operation fails, **_w fopen** returns a null pointer.**Parameters***filename* Pointer to a file.*mode* Points to a string.**Description**The **_w fopen** function opens the file whose name is the string pointed to by *filename*, and associates a stream with it.The argument *mode* points to a string. If the string is one of the following, the file is open in the indicated mode. Otherwise, the behavior is undefined.The arguments of **_w fopen()** are wide-character strings. The functions **_w fopen()** and **fopen()** behave identical, otherwise.**Portability**The function **_w fopen()** is available in Windows only.**See Also****fopen()**.

btowc

Synopsis

```
#include <wchar.h>
#include <stdio.h>
wint_t btowc(int c);
```

Purpose

Determines if character is multibyte character.

Return Value

The `btowc()` function returns **WEOF** if `c` has the value **EOF** or if (**unsigned char**) `c` does not constitute a valid (one-byte) multibyte character in the initial shift state. Otherwise, it returns the wide-character representation of the character.

Parameters

`c` Integer argument.

Description

The `btowc()` function determines whether `c` constitutes a valid (one-byte) multibyte character in the initial shift state.

Example

```
/* This is an example of btowc(int). The function determines whether c is a
   valid (one-byte) character in the initial shift state.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    int c;
    wint_t retval;
    c='r';

    retval = btowc(c);
    if(retval != WEOF) {
        printf("btowc() = ");
        fputwc(c, stdout);
        printf("\n");
    }
    else
        printf("Error\n");
}
```

Output

```
btowc() = r
```

See Also

fgetwc

Synopsis

```
#include <wchar.h>
#include <stdio.h>
wint_t fgetwc(FILE *stream);
```

Purpose

Obtains wide character and advances one position.

Return Value

The **fgetwc()** function returns the next wide character from the input stream pointed to by *stream*. If the stream is at end-of-file, the end-of-file indicator for the stream is set and **fgetwc()** returns **WEOF**. If a read error occurs, the error indicator for the stream is set and **fgetwc()** returns **WEOF**. If an encoding error occurs (including too few bytes), the value of the macro **EILSEQ** is stored in **errno()** and **fgetwc()** returns **WEOF**.

Parameters

stream FILE pointer argument.

Description

If a next wide character is present from the input stream pointed to by *stream*, the **fgetwc()** function obtains that wide character and advances the associated file position indicator for the stream (if defined).

Example

```
/*This is an example of how to use the function fgetwc(FILE *). It will
   retrieve a wide character from an input stream.
*/
#include <wchar.h>

int main() {
    FILE *stream;

    printf("Displays the first character from the file, "wctype.c"\n");
    if((stream = fopen("fgetwc.c", "r")) != NULL) {
        printf("fgetwc(stream) = ");
        fputwc( fgetwc(stream), stdout);
        printf("\n");
    }
    else
        printf("Error in opening file!\n");
    fclose(stream);
}
```

Output

Displays the first character from the file, wctype.c
fgetwc(stream) = /

See Also

fgetws

Synopsis

```
#include <wchar.h>
#include <stdio.h>
wchar_t fgetws(wchar_t * restrict s, int n, FILE * restrict stream);
```

Purpose

Reads wide characters from stream.

Return Value

The `fgetws()` function returns `s` if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read or encoding error occurs during the operation, the array contents are indeterminate and a null pointer is returned.

Parameters

`s` Integer pointer argument.

`n` Integer argument.

`stream` FILE pointer argument.

Description

The `fgetws()` function reads at most one less than the number of wide characters specified by `n` from the stream pointed to by `stream` into the array pointed to by `s`. No additional wide characters are read after a new-line wide character (which is retained) or after end-of-file. A null wide character is written immediately after the last wide character read into the array.

Example

```
/* This is an example of the function fgetws(wchar_t*, int, FILE *).
   This will retrieve an entire string of the max length specified by the user
   from an input stream.
*/
#include <wchar.h>

int main() {
    FILE *stream;
    int i;
    wchar_t str[100];

    i =10;
    printf("Retrieving a string of length %d from \"Makefile\".\n", i);
    if((stream = fopen("Makefile", "r")) != NULL) {
        fgetws(str, i, stream);
        fputws(str, stdout);
        fputs("\n", stdout);
    }
    else
        printf("Error in opening file!\n");
    fclose(stream);
}
```

Output

```
Retrieving a string of length 10 from "Makefile".
default a
```

See Also

fputcw

Synopsis

```
#include <wchar.h>
#include <stdio.h>
wint_t fputcw(wchar_t c, FILE *stream);
```

Purpose

Writes the wide character to the output stream.

Return Value

The `fputcw()` function returns the next wide character written. If a write error occurs, the error indicator for the stream is set and `fputcw()` returns `WEOF`. If an encoding error occurs, the value of the macro `EILSEQ` is stored in `errno()` and `fputcw()` returns `WEOF`.

Parameters

c Integer argument.

stream FILE pointer argument.

Description

The `fputcw()` function writes the wide character specified by *c* to the output stream pointed to by *stream*, at the position indicated by the associated file position indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream.

Example

```
/* This is an example of the function fputcw(wint_t, FILE *). This will insert
   a wide character into the specified output destination.
*/
#include <wchar.h>

int main() {
    FILE *stream;
    wint_t wc;
    char *name;

    if((name = tmpnam(NULL)) == NULL)
        printf("Error in creating temp file.\n");
    else {
        wc = 'i';
        printf("Will write the character 'i' to to a temporary file.\n");
        if((stream = fopen(name, "w")) != NULL) {
            printf("fputcw(stream) = ");
            fputcw(wc, stream);
            fputcw(wc, stdout);
            printf("\n");
        }
        else
            printf("Error in opening file!\n");
    }
    fclose(stream);
    remove(name);
}
```

Output

Will write the character 'i' to to a temporary file.
fputwc(stream) = i

See Also

fputws

Synopsis

```
#include <wchar.h>
#include <stdio.h>
wchar_t fputws(const wchar_t * restrict s, FILE * restrict stream);
```

Purpose

Writes a wide string to stream.

Return Value

The `fputws()` function returns **EOF** if a write or encoding error occurs; otherwise, it returns a nonnegative value.

Parameters

s Integer pointer argument.

stream FILE pointer argument.

Description

The `fputws()` function writes the wide string pointed to by *s* to the stream pointed to by *stream*. The terminating null wide character is not written.

Example

```
/* This is an example for the function fputws(const wchar_t *, FILE *).
   It will place a wide character string into the specified file.
   If the file does not exist, one will be created.
*/
#include <wchar.h>

int main() {
    FILE *stream;
    wchar_t line[100];
    char *temp, *name;
    int len;

    if((name = tmpnam(NULL)) == NULL)
        printf("temp file name could not be made. \n");
    else {
        temp = "Have a Nice Day!";
        len = strlen(temp);
        mbstowcs(line, temp, len);
        if((stream = fopen(name, "w")) != NULL) {
            fputws(line, stream);
            fputws(line, stdout);
            fputs("\n", stdout);
        }
        else
            printf("Error in opening file!\n");
    }
    fclose(stream);
    remove(name);
}
```

Output

Have a Nice Day!

See Also

fwide

Synopsis

```
#include <wchar.h>
#include <stdio.h>
int fwide(FILE *stream, int mode);
```

Purpose

Determines orientation of stream.

Return Value

The `fwide()` function returns a value greater than zero if, after the call, the stream has wide orientation, a value less than zero if the stream has byte orientation, or zero if the stream has no orientation.

Parameters

stream FILE pointer argument.
mode Integer argument.

Description

The `fwide()` function determines the orientation of the stream pointed to by *stream*. If *mode* is greater than zero, the function first attempts to make the stream wide oriented. If *mode* is less than zero, the function first attempts to make the stream byte oriented. Otherwise, *mode* is zero and the function does not alter the orientation of the stream.

Example

```
/* This is an example of the function fwide(FILE *, int). The function will
   alter the stream according to the desired mode.
*/
#include <wchar.h>

int main() {
    FILE *stream;
    int c, retval;

    c = 11;
    if((stream = fopen("fwide.c", "r")) != NULL)
    {
        retval = fwide(stream, c);
        if(retval < 0)
            printf("Stream has byte orientation.\n");
        else if(retval > 0)
            printf("Stream has wide orientation.\n");
        else
            printf("Stream has no orientation.\n");
    }
    else
        printf("Error in opening file!\n");
    fclose(stream);
}
```

Output

Stream has wide orientation.

See Also

getwc

Synopsis

```
#include <wchar.h>
#include <stdio.h>
wint_t getwc(FILE *stream);
```

Purpose

Gets a wide character and advances the pointer.

Return Value

The `getwc()` function returns the next wide character from the input stream pointed to by *stream*, or **WEOF**.

Parameters

stream FILE pointer argument.

Description

The `getwc()` function is equivalent to `fgetwc()`, except that if it is implemented as a macro, it may evaluate *stream* more than once, so the argument should never be an expression with side effects.

Example

```
/* This is an example for the function getwc(FILE *). This retrieves a wide
   character from the input stream.
*/
#include <wchar.h>

int main() {
    FILE *stream;
    int i;
    wint_t ch;

    printf("Retrieves the first 10 characters from the file \"wctype.c\".\n");
    if((stream = fopen("getwc.c", "r")) != NULL) {
        for(i = 0; i < 10; i++) {
            ch = getwc(stream);
            fputwc(ch, stdout);
        }
        printf("\n");
    }
    else
        printf("Error in opening file!\n");
    fclose(stream);
}
```

Output

```
Retrieves the first 10 characters from the file wctype.c.
/* This is
```

See Also

getwchar

Synopsis

```
#include <wchar.h>
wint_t getwchar(void);
```

Purpose

Gets a wide character and advances the file position indicator.

Return Value

The `getwchar()` function returns the next wide character from the input stream pointed to by `stdin`, or `WEOF`.

Parameters

void Void argument.

Description

The `getwchar()` function is equivalent to `getwc` with the argument `stdin`.

Example

```
/* This is an example for getwchar(void). This function will retrieve a wide
   character from the input stream, such as from stdin. This program is
   executed by: getwchar.c < getwchar.in
   The input file getwchar.in contains: 'i'
*/
#include <wchar.h>

int main() {
    int i, ch;
    char buffer[80];

    printf("Enter a character. \n");
    for(i =0; (i<80) && (ch !='\n'); i++) {
        ch = getwchar();
        buffer[i]= (char)ch;
    }
    printf("You entered: %s\n",buffer);
}
```

Output

```
Enter a character.
You entered: i
```

See Also

mbrlen

Synopsis

```
#include <wchar.h>
size_t mbrlen(const char * restrict s, size_t n, mbstate_t * restrict ps);
```

Purpose

Determines the number of bytes in a multibyte character.

Return Value

The `mbrlen()` function returns a value between zero and n , inclusive, $(\text{size_t}) - 2$, or $(\text{size_t}) - 1$.

Parameters

s Integer pointer argument.
 n Unsigned integer argument.
 ps Object type pointer argument.

Description

The `mbrlen()` function is equivalent to the call:

```
mbrtowc(NULL, s, n, ps! = NULL?ps : &internal)
```

where `internal` is the `mbstate_t` object for the `mbrlen()` function, except that the expression designated by ps is evaluated only once.

Example

```
/* This is an example of the function mbrlen(const char*, size_t, mbstate_t).
   This returns a value between zero and length.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    char *str;
    size_t retval, length;
    mbstate_t *ps;

    str = "hello";
    ps = NULL;
    length = 5;
    retval = mbrlen(str, length, ps);
    if(retval != (size_t) -1) {
        printf("Number of bytes of next character: %d\n", retval);
    }
    else
        printf("Error\n");
}
```

Output

```
Number of bytes of next character: 1
```

See Also

`mbrtowc()`

mbrtowc

Synopsis

```
#include <wchar.h>
```

```
size_t mbrtowc(const char * restrict pwc, const char * restrict s, size_t n, mbstate_t * restrict ps);
```

Purpose

Determines the number of bytes needed to complete the next multibyte character.

Return Value

The `mbrtowc()` function returns the first of the following that applies (given the current conversion state)

- 0** if the next *n* or fewer bytes complete the multibyte character that corresponds to the null wide character (which is the value stored).
- positive** if the next *n* or fewer bytes complete a valid multibyte character (which is the value stored); the value returned is the number of bytes that complete the multibyte character.
- (size_t) - 2** if the next *n* bytes contribute to an incomplete (but potentially valid) multibyte character, and all *n* bytes have been processed (no value is stored).
- (size_t) - 1** if an encoding error occurs, in which case the next *n* or fewer bytes do not contribute to a complete and valid multibyte character (no value is stored); the value of the macro `EILSEQ` is stored in `errno()`, and the conversion state is undefined.

Parameters

pwc Character pointer argument.

s Character pointer argument.

n Unsigned integer argument.

ps Integer argument.

Description

If *s* is a null pointer, the `mbrtowc()` function is equivalent to the call:

```
mbrtowc(NULL, "", 1, ps)
```

In this case, the values of the parameters *pwc* and *n* are ignored.

If *s* is not a null pointer, the `mbrtowc()` function inspects at most *n* bytes beginning with the byte pointed to by *s* to determine the number of bytes needed to complete the next multibyte character (including any shift sequences). If the function determines that the next multibyte character is completed, it determines the value of the corresponding wide character and then, if *pwc* is not a null pointer, stores that value in the object pointed to by *pwc*. If the corresponding wide character is the null wide character, the resulting state described is the initial conversion state.

Example

```
/* This is an example of the function mbrtowc(const char*, size_t, mbstate_t).
   the mbrtowc function inspects at most n bytes beginning with the byte
   pointed to by s to determine the number of bytes needed to complete the
   next multibyte character
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    char *str2;
    wchar_t *str;
    size_t retval, length;
```

```
mbstate_t *ps;

str2 = "hello";
ps = NULL;
length = 5;
retval = mbrtowc(str, str2, length, ps);
if(retval != (size_t) -1)
    printf("Number of bytes of the next character: %d\n", retval);
else
    printf("error\n");
}
```

Output

Number of bytes of the next character: 1

See Also
mbrtowc

mbsinit

Synopsis

```
#include <wchar.h>
#include <stdio.h>
int mbsinit(const mbstate_t *ps);
```

Purpose

Determines if pointed-to object describes an initial conversion state.

Return Value

The **mbsinit()** function returns nonzero if *ps* is a null pointer or if the pointed-to object describes an initial state; otherwise it returns zero.

Parameters

ps Integer pointer argument.

Description

If *ps* is not a null pointer, the **mbsinit()** function determines whether the pointed-to **mbstate_t** object describes an initial conversion state.

Example

```
/* This is an example of the function mbsinit(mbstate_t).
   This function determines whether the object pointed to by ps describes an
   initial conversion state.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    int retval;
    mbstate_t *ps;

    ps = NULL;
    retval = mbsinit(ps);
    if(retval != 0) {
        printf("mbsinit(ps) returns nonzero if ps is a null pointer \n");
        printf("or ps is pointing to an initial conversion state.\n");
        printf("mbsinit(ps)= %d\n", retval);
    }
    else
        printf("Error \n");
}
```

Output

```
mbsinit(ps) returns nonzero if ps is a null pointer
or ps is pointing to an initial conversion state.
mbsinit(ps)= 1
```

See Also

mbsrtowcs

Synopsis

```
#include <wchar.h>
```

```
size_t mbsrtowcs(wchar_t * restrict dst, const char ** restrict src, size_t len, mbstate_t * restrict ps);
```

Purpose

Converts a sequence of multibyte characters from an array into a sequence.

Return Value

If the input conversion encounters a sequence of bytes that do not form a valid multibyte character, an encoding error occurs: the **mbsrtowcs()** function stores the value of the macro **EILSEQ** in **errno()** and returns **(size_t) - 1**; the conversion state is undefined. Otherwise, it returns the number of multibyte characters successfully converted, not including the terminating null (if any).

Parameters

dst Character pointer argument.

src Character pointer argument.

len Unsigned integer argument.

ps Object type pointer.

Description

The **mbsrtowcs()** function converts a sequence of multibyte characters, beginning in the conversion state described by the object pointed to by *ps*, from the array indirectly pointed to by *src* into a sequence of corresponding wide characters. If *dst* is not a null pointer, the converted characters are stored into the array pointed to by *dst*. Conversion continues up to and including a terminating null character, which is also stored. Conversion stops earlier in two cases: when a sequence of bytes is encountered that does not form a valid multibyte character, or (if *dst* is not a null pointer) when *len* codes have been stored into the array pointed to by *dst*. Each conversion takes place as if by call to the **mbrtowc** function.

If *dst* is not a null pointer, the pointer object pointed to by *src* is assigned either a null pointer (if conversion stopped due to reaching a terminating null character) or the address just past the last multibyte character converted (if any). If conversion stopped due to reaching a terminating null character and if *dst* is not a null pointer, the resulting state described is the initial conversion state.

Example

```
/* This is an example of the function mbsrtowcs(wchar_t*, const char**,
    size_t, mbstate_t). Converts a sequence of characters
    into a sequence of corresponding wide-characters.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    wchar_t str1[100];
    const char *str2;
    size_t retval, length;
    mbstate_t *ps;

    str2 = "hello";
    ps = NULL;
    length = 5;
    retval = mbsrtowcs(str1, &str2, length, ps);
    if(retval != (size_t) -1) {
        printf("Number of multibyte characters");
        printf(" successfully converted: %d\n", retval);
    }
    else
        printf("Error\n");
}
```

```
}
```

Output

Number of multibyte characters successfully converted: 5

See Also

putwc

Synopsis

```
#include <wchar.h>
#include <stdio.h>
wint_t putwc(wchar_t c, FILE *stream);
```

Purpose

Writes a wide character to an output stream.

Return Value

The `putwc()` function returns the next wide character written, or `WEOF`.

Parameters

c Integer argument.

stream FILE pointer argument.

Description

The `putwc()` function is equivalent to `fputwc()`, except that if it is implemented as a macro, it may evaluate *stream* more than once, so that argument should never be an expression with side effects.

Example

```
/* This is an example for putwc(wint_t, FILE*). This function will write a
   character to a stream.
*/
#include <wchar.h>

int main() {
    FILE *stream;
    wint_t wc;
    char *name;

    wc = 'i';
    if((name = tmpnam(NULL)) == NULL)
        printf("Error in creating temp file.\n");
    else {
        fputs("Will write the letter '", stdout);
        fputwc(wc, stdout);
        fputs("' to a temporary file.\n", stdout);
        if((stream = fopen(name, "w")) != NULL) {
            printf("putwc(stream) = %c\n", putwc(wc, stream));
        }
        else
            printf("Error in opening file!\n");
    }
    fclose(stream);
    remove(name);
}
```

Output

```
Will write the letter 'i' to a temporary file.
putwc(stream) = i
```

See Also

putwchar

Synopsis

```
#include <wchar.h>
wint_t putwchar(wchar_t c);
```

Purpose

Writes a wide character to an output stream.

Return Value

The `putwchar()` function returns the character written, or **WEOF**.

Parameters

c Integer argument.

Description

The `putwchar()` function is equivalent to `putwc()` with the second argument **stdout**.

Example

```
/* This is an example of the function fputwchar(wint_t). This will write a
   wide character to stdout.
*/
#include <wchar.h>

int main() {
    wint_t wc;
    wc = 's';

    printf("Will place the character '");
    fputwc(wc, stdout);
    printf("' on to stdout.\n");
    printf("putwchar(wc) = ");
    putwchar(wc);
    printf("\n");
}
```

Output

```
Will place the character 's' on to stdout.
putwchar(wc) = s
```

See Also

ungetwc

Synopsis

```
#include <wchar.h>
#include <stdio.h>
wint_t ungetwc(wint_t c, FILE *stream);
```

Purpose

Pushes a wide character back onto an input stream.

Return Value

The `ungetwc()` function returns the character pushed back, or `WEOF` if the operation fails.

Parameters

c Integer argument.

stream FILE pointer argument.

Description

The `ungetwc()` function pushes the wide character specified by *c* back onto the input stream pointed to by *stream*. Pushed-back wide characters will be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by *stream*) to a file positioning function (`fseek()`, `fsetpos()`, or `rewind()`) discards any pushed-back wide characters for the stream. The external storage corresponding to the stream is unchanged.

One wide character of pushback is guaranteed, even if the call to the `ungetwc()` function follows just after a call to a formatted wide character input function `fwscanf()`, `vfwscanf()`, `vwscanf()`, or `wscanf()`. If the `ungetwc()` function is called too many times on the same stream without an intervening read or file positioning operation on that stream, the operation may fail.

If the value of *c* equals that of the macro `WEOF`, the operation fails and the input stream is unchanged.

A successful call to the `ungetwc()` function clears the end-of-file indicator for the stream. The value of the file position indicator for the stream after reading or discarding all pushed-back wide characters is the same as it was before the wide characters were pushed back. For a text or binary stream, the value of its file position indicator after a successful call to the `ungetwc()` function is unspecified until all pushed-back wide characters are read or discarded.

Example

```
/* This is an example of the function ungetwc(wint_t, FILE *). This pushes
   a character back onto the stream.
*/
#include <wchar.h>

int main() {
    FILE *stream;
    wint_t wc;

    if((stream = fopen("ungetwc.c", "r")) != NULL) {
        wc = fgetwc(stream);
        printf("ungetwc(wc, stream) = ");
        fputwc(ungetwc(wc, stream), stdout);
        printf("\n");
    }
    else
        printf("Error in opening file!\n");
    fclose(stream);
}
```

Output

```
ungetwc(wc, stream) = /
```

See Also

wrtomb

Synopsis

```
#include <wchar.h>
```

```
size_t wrtomb(const char * restrict s, wchar_t n, mbstate_t * restrict ps);
```

Purpose

Determines the number of bytes needed to represent the next multibyte character.

Return Value

The `wrtomb()` function returns the number of bytes stored in the array object (including any shift sequences). When `wc` is not a valid wide character, an encoding error occurs: the function stores the value of the macro `EILSEQ` in `errno()` and returns (*size_t*) `-1`; the conversion state is undefined.

Parameters

s Character pointer argument.

n Character argument.

ps Integer pointer argument.

Description

If *s* is a null pointer, the `wrtomb()` function is equivalent to the call:

```
wrtomb(buf, L'\0', 1, ps)
```

where `buf` is an internal buffer.

If *s* is not a null pointer, the `wrtomb()` function determines the number of bytes needed to represent the multibyte character given by *wc* (including any shift sequences), and stores the resulting bytes in the array whose first element is pointed to by *s*. At most `MB_CUR_MAX` bytes are stored. If *wc* is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state; the resulting state described is the initial conversion state.

Example

```
/* This is an example of the function wrtomb(char*, wchar_t, mbstate_t).
   The wrtomb function determines the number of bytes needed to represent
   the multibyte character that corresponds to the wide character given by wc.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    char *str;
    size_t retval;
    wchar_t c;
    mbstate_t *ps;

    str = "hello";
    c = 'e';
    ps = NULL;
    retval = wrtomb(str, c, ps);
    if(retval != (size_t) -1) {
        printf("Number of bytes needed to represent the \n");
        printf("next character: %d\n", retval);
    }
    else
        printf("Error\n");
}
```


Output

Number of bytes needed to represent the
next character: 1

See Also

wscat

Synopsis

```
#include <wchar.h>
wchar_t wscat(wchar_t * restrict s1, const wchar_t * restrict s2);
```

Purpose

Appends a copy of one wide string to another wide string.

Return Value

The `wscat()` returns the value of `s1`.

Parameters

`s1` Integer pointer argument.
`s2` Integer pointer argument.

Description

The `wscat()` function appends a copy of the wide string pointed to by `s2` (including the terminating null wide character) to the end of the wide string pointed to by `s1`. The initial wide character of `s2` overwrites the null character at the end of `s1`.

Example

```
/* This is an example of the function wscat(wchar_t*, const wchar_t*). This
   appends a string with another wide character string.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    wchar_t str1[100];
    wchar_t str2[100];
    char *line;
    int retval, len;
    line = "Hello";
    len = strlen(line);

    mbstowcs(str1, line, len );
    mbstowcs(str2, line, len);
    fputs("String1: ", stdout);
    fputws(str1, stdout);
    fputs("\nString2: ", stdout);
    fputws(str2, stdout);
    wscat(str1, str2);
    fputs("\nAfter wide character appending: ", stdout);
    fputws(str1, stdout);
    fputs("\n", stdout);
}
```

Output

```
String1: Hello
String2: Hello
After wide character appending: HelloHello
```

See Also

wcschr

Synopsis**#include** <wchar.h>**wchar_t** wcschr(const wchar_t *s, wchar_t c);**Purpose**

Locates first occurrence of a wide character in a string.

DescriptionThe **wcschr()** function locates the first occurrence of *c* in the wide string pointed to by *s*. The terminating null wide character is considered to be part of the wide string.**Parameters***s* Integer pointer argument.*c* Integer argument.**Return Value**The **wcschr()** function returns a pointer to the located wide string character, or a null pointer if the wide character does not occur in the wide string.**Example**

```

/* This is an example of the function wcschr(const wchar_t*, wchar_t*).
   This function finds a character in a string.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    wchar_t str[100], c;
    char *line;
    wchar_t *retval;
    int result, len;
    line = "Hello";
    c = 'l';

    len = strlen(line);
    mbstowcs(str, line, len);
    retval = wcschr(str, c);
    result = retval - str + 1;
    if(result <= 0) {
printf("There is no '");
        fputwc(c, stdout);
        printf("' in this string.\n");
    }
    else {
printf("The first occurrence of '");
        fputwc(c, stdout);
        printf("' is at position: %d\n", result);
    }
}

```

Output

The first occurrence of 'l' is at position: 3

See Also

wcscmp

Synopsis

```
#include <wchar.h>
```

```
int wcscmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```

Purpose

Compares two wide strings.

Return Value

The `wcscmp()` function returns an integer greater than, equal to, or less than zero, accordingly as the wide string pointed to by `s1` is greater than, equal to, or less than the wide string pointed to by `s2`.

Parameters

`s1` Integer pointer argument.

`s2` Integer pointer argument.

`n` Unsigned integer argument.

Description

The `wcscmp()` function compares the wide string pointed to by `s1` to the wide string pointed to by `s2`.

Example

```
/* This is an example of the function wcscmp(const wchar_t*, const wchar_t*).
   This function compares two strings.  If string1 < string2, then a negative
   number will be returned.  If string1 = string2, zero will be returned.
   If string1 > string2, a positive number will be returned.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    const char *line, *line2;
    wchar_t str1[100], str2[100];
    int retval, len1, len2;

    line = "hello";
    line2 = "hello";
    len1 = strlen(line);
    len2 = strlen(line2);
    mbstowcs(str1, line, len1);
    mbstowcs(str2, line2, len2);
    retval = wcscmp(str1, str2);
    if(retval > 0)
        printf("String1 is greater than String2\n");
    else if(retval == 0)
        printf("String1 is identical to String2\n");
    else if(retval < 0)
        printf("String1 is less than String2\n");
    else
        printf("Error!\n");
}
```

Output

```
String1 is identical to String2
```

See Also

wscoll

Synopsis

```
#include <wchar.h>
```

```
int wscoll(const wchar_t *s1, const wchar_t *s2);
```

Purpose

Compares two wide strings.

Return Value

The `wscoll()` function returns an integer greater than, equal to, or less than zero, accordingly as the wide string pointed to by `s1` is greater than, equal to, or less than the wide string pointed to by `s2` when both are interpreted as appropriate to the current locale.

Parameters

`s1` Integer pointer argument.

`s2` Integer pointer argument.

Description

The `wscoll()` function compares the wide string pointed to by `s1` to the wide string pointed to by `s2`, both interpreted as appropriate to the `LC_COLLATE` category of the current locale.

Example

```
/* This is an example of the function wscoll(const wchar_t*, const wchar_t*).
   If string1 < string2, a negative number will be returned.
   If string1 = string2, zero will be returned.  If
   string1 > string2, a positive number will be returned.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    const char *line, *line2;
    wchar_t str1[100], str2[100];
    int retval, len1, len2;

    line = "hello";
    line2 = "hello";
    len1 = strlen(line);
    len2 = strlen(line2);
    mbstowcs(str1, line, len1);
    mbstowcs(str2, line2, len2);
    retval = wscoll(str1, str2);
    if(retval > 0)
        printf("String1 is greater than String2\n");
    else if(retval == 0)
        printf("String1 is identical to String2\n");
    else if(retval < 0)
        printf("String1 is less than String2\n");
    else
        printf("Error!\n");
}
```

Output

```
String1 is identical to String2
```

See Also

wscpy

Synopsis

```
#include <wchar.h>
```

```
wchar_t wscpy(wchar_t * restrict s1, const wchar_t * restrict s2);
```

Purpose

Copies a wide string to an array.

Return Value

The `wscpy()` returns the value of `s1`.

Parameters

`s1` Integer pointer argument.

`s2` Integer pointer argument.

Description

The `wscpy()` function copies the wide string pointed to by `s2` (including the terminating null wide character) into the array pointed to by `s1`.

Example

```
/* This is an example of the function wscpy(wchar_t*, const wchar_t*). This
   make a copy of the second string and store into the first parameter.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    wchar_t str1[100], str2[100];
    char *line;
    int retval, len;
    line = "Hello";
    len = strlen(line);

    mbstowcs(str2, line, len);
    wscpy(str1, str2);
    fputws(str1, stdout);
    fputs("\n", stdout);
}
```

Output

```
Hello
```

See Also

wcspsn

Synopsis

```
#include <wchar.h>
```

```
size_t wcspsn(const wchar_t *s1, const wchar_t *s2);
```

Purpose

Computes the length of the maximum initial segment of a wide string.

Return Value

The `wcspsn()` function returns the length of the segment.

Parameters

s1 Integer pointer argument.

s2 Integer pointer argument.

Description

The `wcspsn()` function computes the length of the maximum initial segment of the wide string pointed to by *s1* which consists entirely of wide characters *not* from the wide string pointed to by *s2*.

Example

```
/* This is an example of the function wcspsn(const wchar_t*, const wchar_t*).
   This finds a substring. The function returns the length of the first
   substring that is made up of characters from string 2.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    char *line, *line2;
    wchar_t str1[100], str2[100];
    size_t retval;
    int len1, len2;
    line = "hello";
    line2 = "ll";

    len1 = strlen(line);
    len2 = strlen(line2);
    mbstowcs(str1, line, len1);
    mbstowcs(str2, line2, len2);
    retval = wcspsn(str1, str2);
    printf("Substring starts at position: %d\n", retval);
}
```

Output

```
Substring starts at position: 2
```

See Also

wcsftime

Synopsis

```
#include <wchar.h>
```

```
size_t wcsftime(wchar_t * restrict s, size_t maxsize, const wchar_t * restrict format, const struct tm * restrict timeptr);
```

Purpose

Places wide characters into an array.

Return Value

If the total number of resulting wide characters, including the terminating null wide character is not more than *maxsize*, the **wcsftime()** function returns the number of wide characters placed into the array pointed to by *s* not including the terminating null wide character. Otherwise, zero is returned and the contents of the array are indeterminate.

Parameters

s Integer pointer argument.

maxsize Unsigned integer argument.

format Integer pointer argument.

timeptr Struct argument.

Description

The **wcsftime()** function is equivalent to the **strftime()** function, except that:

- The argument *s* points to the initial element of an array of wide characters into which the generated output is to be placed.
- The argument *maxsize* indicates the limiting number of wide characters into which the generated output is to be placed.
- The argument *format* is a wide string and the conversion specifiers are replaced by corresponding sequences of wide characters.
- The return value indicates the number of wide characters.

Example

```
/* This is an example of the function wcsftime(wchar_t*, size_t, const wchar_t*,
    const struct tm*). This function formats a time string.
*/
#include <wchar.h>
#include <stdlib.h>
#include <time.h>

int main() {
    size_t retval, maxsize;
    wchar_t str1[100];
    wchar_t format[100];
    char *line;
    struct tm *timeptr;
    int len;
    time_t aclock;
    time(&aclock);
    maxsize = 100;

    line = "%c";
    len = strlen(line);
    mbstowcs(format, line, len);
    timeptr = localtime(&aclock);
```

```
    retval = wcsftime(str1, maxsize, format, timeptr );  
    fputws(str1, stdout);  
    fputs("\n", stdout);  
}
```

Output

Mon Feb 21 14:56:31 2000

See Also

wcslen

Synopsis

```
#include <wchar.h>
```

```
size_t wcslen(const wchar_t *s);
```

Purpose

Computes the length of a wide string.

Return Value

The `wcslen()` function returns the number of wide characters that precede the terminating null wide character.

Parameters

s Integer pointer argument.

Description

The `wcslen()` function computes the length of the wide string pointed to by *s*.

Example

```
/* This is an example of the function wcslen(const wchar_t*). This
   returns the length of the string.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    char *line;
    wchar_t str1[100];
    size_t retval;
    int len;

    line = "hello";
    len = strlen(line);
    mbstowcs(str1, line, len);
    retval = wcslen(str1);
    printf("Number of characters in str1: %d\n", retval);
}
```

Output

```
Number of characters in str1: 5
```

See Also

wcsncat

Synopsis

```
#include <wchar.h>
```

```
wchar_t wcsncat(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
```

Purpose

Appends wide characters from an array to a wide string.

Return Value

The `wcsncat()` returns the value of `s1`.

Parameters

`s1` Integer pointer argument.

`s2` Integer pointer argument.

`n` Unsigned integer argument.

Description

The `wcsncat()` function appends not more than `n` wide characters (a null wide character and those that follow it are not appended) from the array pointed to by `s2` to the end of the wide string pointed to by `s1`. The initial wide character of `s2` overwrites the null wide character at the end of `s1`. A terminating null wide character is always appended to the result.

Example

```
/* This is an example of the function wcsncat(wchar_t*, const wchar_t*, size_t).
   This function appends the specified number of characters from str2 to str1.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    wchar_t str1[100], str2[100];
    char *line, *line2;
    wchar_t *retval;
    size_t count;
    int len1, len2;

    line = "Hello, ";
    line2 = "how are you today?";
    len1 = strlen(line);
    len2 = strlen(line2);
    count = 18;
    mbstowcs(str1, line, len1);
    mbstowcs(str2, line2, len2);
    wcsncat(str1, str2, count);
    fputws(str1, stdout);
    fputs("\n", stdout);
}
```

Output

```
Hello, how are you today?
```

See Also

wcsncmp

Synopsis

```
#include <wchar.h>
```

```
int wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```

Purpose

Compares wide characters between two arrays.

Return Value

The `wcsncmp()` function returns an integer greater than, equal to, or less than zero, accordingly as the possibly null-terminated array pointed to by `s1` is greater than, equal to, or less than the possibly null-terminated array pointed to by `s2`.

Parameters

`s1` Integer pointer argument.

`s2` Integer pointer argument.

`n` Unsigned integer argument.

Description

The `wcsncmp()` function compares not more than `n` wide characters (those that follow a null wide character are not compared) from the array pointed to by `s1` to the array pointed to by `s2`.

Example

```

/* This is an example of the function wcsncmp(const wchar_t*, const wchar_t*,
   size_t). This compares the specified number of characters of two strings.
   If string1 substring < string2 substring, then a negative value is returned.
   If string1 substring = string2 substring, then zero is returned.
   If string1 substring > string2 substring, then a positive value is returned.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    char *line, *line2;
    wchar_t str1[100], str2[100];
    size_t count;
    int retval, len1, len2;
    count = 5;
    line = "hellohello";
    line2 = "hello";
    len1 = strlen(line);
    len2 = strlen(line2);

    mbstowcs(str1, line, len1);
    mbstowcs(str2, line2, len2);
    fputs("String1: ", stdout);
    fputws(str1, stdout);
    fputs("\nString2: ", stdout);
    fputws(str2, stdout);
    printf("\nNumber of characters to compare: %d\n", count);
    retval = wcsncmp(str1, str2, count);
    if(retval > 0)
        printf("String1 is greater than String2\n");
    else if(retval == 0)
        printf("String1 is identical to String2\n");
    else if(retval < 0)

```

```
        printf("String1 is less than String2\n");
    else
        printf("Error!\n");
    printf("wcsncmp(str1,str2,count) = %d\n", retval);
}
```

Output

```
String1: hellohello
String2: hello
Number of characters to compare: 5
String1 is identical to String2
wcsncmp(str1,str2,count) = 0
```

See Also

wcsncpy

Synopsis

```
#include <wchar.h>
```

```
wchar_t wcsncpy(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
```

Purpose

Copies wide characters from one array to another array.

Return Value

The `wcsncpy()` returns the value of `s1`.

Parameters

`s1` Integer pointer argument.

`s2` Integer pointer argument.

`n` Unsigned integer argument.

Description

The `wcsncpy()` function copies not more than `n` wide characters (those that follow a null wide character are not copied) from the array pointed to by `s2` to the array pointed to by `s1`.

If the array pointed to by `s2` is a wide string that is shorter than `n` wide characters, null wide characters are appended to the copy in the array pointed to by `s1`, until `n` wide characters in all have been written.

Example

```
/* This is an example of the function wcsncpy(wchar_t*, const wchar_t*,
size_t). This copies the specified number of characters from string2
to string1.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    wchar_t str1[100], str2[100];
    char *line;
    wchar_t *retval;
    size_t count;
    int len;

    line = "How are you today?";
    count = 18;
    len = strlen(line);
    mbstowcs(str2, line, len);
    wcsncpy(str1, str2, count);
    fputws(str1, stdout);
    fputs("\n", stdout);
}
```

Output

```
How are you today?
```

See Also

wcpbrk

Synopsis

```
#include <wchar.h>
wchar_t wcpbrk(const wchar_t *s1, const wchar_t *s2);
```

Purpose

Locates the first occurrence of a wide character from one string in another string.

Return Value

The `wcpbrk()` function returns a pointer to the wide character in `s1`, or a null pointer if no wide character from `s2` occurs in `s1`.

Parameters

`s1` Integer pointer argument.
`s2` Integer pointer argument.

Description

The `wcpbrk()` function locates the first occurrence in the wide string pointed to by `s1` of any wide character from the wide string pointed to by `s2`.

Example

```
/* This is an example of the function wcpbrk(const wchar_t*, const wchar_t).
   Returns a pointer to the first character in string1 that also belongs to
   string2.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    wchar_t str1[100], str2[100];
    char *line, *line2;
    wchar_t *retval;
    int len1, len2;
    line = "Today is a very fine day.\n";
    line2 = "abc";
    len1 = strlen(line);
    len2 = strlen(line2);

    mbstowcs(str1, line, len1);
    mbstowcs(str2, line2, len2);
    retval = wcpbrk(str1, str2);
    fputs("Original: ", stdout);
    fputws(str1, stdout);
    fputs("Characters to search for: ", stdout);
    fputws(str2, stdout);
    fputs("\nFirst occurrence of a character belonging to string2: ", stdout);
    fputws(retval, stdout);
    fputs("\n", stdout);
}
```

Output

```
Original: Today is a very fine day.
Characters to search for: abc
First occurrence of a character belonging to string2: ay is a very fine day.
```

See Also

wcsrchr

Synopsis

```
#include <wchar.h>
wchar_t wcsrchr(const wchar_t *s, wchar_t c);
```

Purpose

Locates the last occurrence of a wide character in a wide string.

Return Value

The `wcsrchr()` function returns a pointer to the located wide character, or a null pointer if the wide character does not occur in the wide string.

Parameters

s Integer pointer argument.
c Integer argument.

Description

The `wcsrchr()` function locates the last occurrence of *c* in the wide string pointed to by *s*. The terminating null wide character is considered to be part of the wide string.

Example

```
/* This is an example of the function wcsrchr(const char_t*, wchar_t*).
   This returns a pointer to the last occurrence of the search character.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    wchar_t str1[100];
    wchar_t c, *retval;
    char *line;
    int result, len;
    line = "Hello, how are you doing?";
    c = 'l';

    len = strlen(line);
    mbstowcs(str1, line, len);
    fputs("String to be searched: ", stdout);
    fputws(str1, stdout);
    printf("\nCharacter to be searched for: ");
    fputwc(c, stdout);
    printf("\n");
    retval = wcsrchr(str1, c);
    result = retval - str1 + 1;
    if(result <= 0) {
        printf("There is no '");
        fputwc(c, stdout);
        printf("' in this string.\n");
    }
    else {
        printf("The last occurrence of '");
        fputwc(c, stdout);
        printf("' is at position: %d\n", result);
    }
}
```

Output

```
String to be searched: Hello, how are you doing?  
Character to be searched for: l  
The last occurrence of 'l' is at position: 4
```

See Also

wcsrtombs

Synopsis

```
#include <wchar.h>
```

```
size_t wcsrtombs(wchar_t * restrict dst, const wchar_t ** restrict src, size_t len, mbstate_t * restrict ps);
```

Purpose

Converts a sequence of wide characters from an array into a sequence of multibyte characters.

Return Value

If conversion stops because a code is reached that does not correspond to a valid multibyte character, an encoding error occurs: the `wcsrtombs()` function stores the value of the macro `EILSEQ` in `errno()` and returns $(size_t) - 1$; the conversion state is undefined. Otherwise, it returns the number of bytes in the resulting multibyte character sequence, not including the terminating null (if any).

Parameters

dst Character pointer argument.

src Character pointer argument.

len Unsigned integer argument.

ps Object type pointer.

Description

The `wcsrtombs()` function converts a sequence of wide characters from the array indirectly pointed to by *src* into a sequence of corresponding multibyte characters, beginning in the conversion state described by the object pointed to by *ps*. If *dst* is not a null pointer, the converted characters are then stored into the array pointed to by *dst*. Conversion continues up to and including a terminating null character, which is also stored. Conversion stops earlier in two cases: when a code is reached that does not correspond to a valid multibyte character, or (if *dst* is not a null pointer) when the next multibyte character would exceed the limit of *len* total bytes to be stored into the array pointed to by *dst*. Each conversion takes place as if by call to the `wcrtomb()` function.

If *dst* is not a null pointer, the pointer object pointed to by *src* is assigned either a null pointer (if conversion stopped due to reaching a terminating null character) or the address just past the last wide character converted (if any). If conversion stopped due to reaching a terminating null character, the resulting state described is the initial conversion state.

Example

```
/* This is an example of the function wcsrtombs(char*, const wchar_t**,
    size_t, mbstate_t).
   The wcsrtombs function converts a sequence of wide
   characters from the array indirectly pointed to by src into
   a sequence of corresponding multibyte characters, beginning
   in the conversion state described by the object pointed to
   by ps.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    char *str1, *line;
    const wchar_t *str2;
    wchar_t str3[100];
    size_t retval, length;
    mbstate_t *ps;
    int len;

    line = "hello";
    len = strlen(line);
    mbstowcs(str3, line, len);
    str2 = str3;
```

```
ps = NULL;
length = 5;
retval = wcsrtombs(str1, &str2, length, ps);
if(retval != (size_t) -1) {
    printf("Number of multibyte characters successfully");
    printf(" converted: %d\n", retval);
}
else
printf("Error\n");
}
```

Output

Number of multibyte characters successfully converted: 5

See Also

wcssp

Synopsis

```
#include <wchar.h>
```

```
size_t wcssp(const wchar_t *s1, const wchar_t *s2);
```

Purpose

Computes the length of the maximum initial segment of a wide string.

Return Value

The `wcssp()` function returns the length of the segment.

Parameters

s1 Integer pointer argument.

s2 Integer pointer argument.

Description

The `wcssp()` function computes the length of the maximum initial segment of the wide string pointed to by *s1* which consists entirely of wide characters from the wide string pointed to by *s2*.

Example

```
/* This is an example of the function wcssp(const wchar_t*, const wchar_t*).
   This returns the length of the substring found in str1.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    char *line, *line2;
    wchar_t str1[100], str2[100];
    size_t retval;
    int len1, len2;

    line = "hello, how are you?";
    line2 = "hel";
    len1 = strlen(line);
    len2 = strlen(line2);
    mbstowcs(str1, line, len1);
    mbstowcs(str2, line2, len2);
    fputs("String: ", stdout);
    fputws(str1, stdout);
    fputs("\nSubstring to search for: ", stdout);
    fputws(str2, stdout);
    retval = wcssp(str1, str2);
    printf("\nThe length of the substring is:  %d\n", retval);
}
```

Output

```
String: hello, how are you?
Substring to search for: hel
The length of the substring is:  4
```

See Also

wcsstr

Synopsis

```
#include <wchar.h>
```

```
wchar_t wcsstr(const wchar_t *s1, const wchar_t *s2);
```

Purpose

Locates the first sequence found in one wide string in the other wide string.

Return Value

The `wcsstr()` function returns a pointer to the located wide string, or a null pointer if the wide string is not found. If `s2` points to a wide string with zero length, the function returns `s1`.

Parameters

`s1` Integer pointer argument.

`s2` Integer pointer argument.

Description

The `wcsstr()` function locates the last occurrence in the wide string pointed to by `s1` of the sequence of wide characters (excluding the terminating null wide character) in the wide string pointed to by `s2`.

Example

```

/* This is the function wcsstr(const wchar_t*, const wchar_t*).
   This function finds the first occurrence of a wide string from str2 in str1.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    wchar_t str1[100], str2[100];
    wchar_t *retval;
    char *line, *line2;
    int len1, len2;
    line = "Today is a very fine day.";
    line2 = "is";
    len1 = strlen(line);
    len2 = strlen(line2);

    mbstowcs(str1, line, len1);
    mbstowcs(str2, line2, len2);
    retval = wcsstr(str1, str2);
    if(retval != NULL) {
        fputs("Original: ", stdout);
        fputws(str1, stdout);
        fputs("\nCharacters to search for: ", stdout);
        fputws(str2, stdout);
        printf("\nFirst occurrence of a character ");
        printf("belonging to string2: ");
        fputws(retval, stdout);
        fputs("\n", stdout);
    }
    else
        printf("Error\n");
}

```

Output

```
Original: Today is a very fine day.  
Characters to search for: is  
First occurrence of a character belonging to string2: is a very fine day.
```

See Also

wcstod

Synopsis

```
#include <wchar.h>
```

```
double wcstod(const wchar_t * restrict nptr, wchar_t ** restrict endptr);
```

```
float wcstof(const wchar_t * restrict nptr, wchar_t ** restrict endptr);
```

```
long double wcstold(const wchar_t * restrict nptr, wchar_t ** restrict endptr);
```

Purpose

Converts an initial portion of a wide string to double, float, and long double, respectively.

Return Value

The functions return the converted value, if any. If no conversions could be performed, zero is returned. If the correct value is outside the range of representable values, plus or minus **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL** is returned (according to the return type and the sign of the value) and the value of the macro **ERANGE** is stored in **errno()**. If the result underflows, the functions return a value whose magnitude is no greater than the smallest normalized positive number in the return type; whether **errno()** acquires the value **ERANGE** is implementation defined.

Parameters

nptr Integer pointer argument.

endptr Integer pointer argument.

Description

The **wcstod()**, **wcstof()**, and **wcstold()** functions convert the initial portion of the wide string pointed to by *nptr* to **double**, **float**, and **long double** representation, respectively. First, they decompose the input string into three parts: an initial, possibly empty, sequence of white-space wide characters (as specified by the **iswspace()** function) a subject sequence resembling a floating-point constant or representing an infinity or NaN; and a final wide string of one or more unrecognized wide characters, including the terminating null wide character of the input wide string. Then, they attempt to convert the subject sequence to a floating-point number, and return the result.

The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- a nonempty sequence of decimal digits optionally containing a decimal-point wide character, then an optional exponent part as defined for the corresponding single-byte characters;
- a **0x** or **0X**, then a nonempty sequence of decimal digits optionally containing a decimal-point wide character, then an optional binary exponent part;
- one of **INF** or **INFINITY**, or any other wide string equivalent.
- one of **NAN** or **NAN(*n* — *wchar* — *sequence_{opt}*)**, or any other wide string equivalent except for case in the NAN part, where:
 - n-wchar-sequence*:
 - digit*
 - nondigit*
 - n-wchar-sequence digit*
 - n-wchar-sequence nondigit*

The subject sequence is defined as the longest initial subsequence of the input wide string, starting with the first non-white-space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide string is not of the expected form.

If the subject sequence has the expected form for a floating-point number, the sequence of wide characters starting with the first digit or the decimal-point wide character (whichever occurs first) is interpreted as a floating constant, except that the decimal-point wide character is used in place of a period, and that if neither an exponent part nor a decimal-point wide character appears in a decimal floating point number, or if a binary exponent part does not appear in a binary floating point number, an exponent part of the appropriate type with the value zero is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the sequence is interpreted as negated. A wide character sequence **INF** or **INFINITY** is interpreted as an infinity, if

representable in the return type, else like a floating constant that is too large for the range of the return type. A wide character sequence **NAN** or **NAN**(*n - wchar - sequence_{opt}*) is interpreted as a quiet NaN, if supported in the return type, else like a subject sequence part that does not have the expected form; the meaning of the *n-wchar* sequences is implementation-defined. A pointer to the final wide string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

If the subject sequence has the hexadecimal form and **FLT_RADIX** is a power of 2, the value resulting from the conversion is correctly rounded.

In other than the “C” locale, additional locale-specific subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

Recommended practice If the subject sequence has the hexadecimal form and **FLT_RADIX** is not a power of 2, the result should be one of the two numbers in the appropriate internal format that are adjacent to the hexadecimal floating source value, with the extra stipulation that the error should have a correct sign for the current rounding direction.

If the subject sequence has the decimal form and at the most **DECIMAL_DIG** (defined in **float.h**) significant digits, the result should be correctly rounded. If the subject sequence *D* has the decimal form and more than **DECIMAL_DIG** significant digits, such that the values of *L* and *U*, both having **DECIMAL_DIG** significant digits, such that the values of *L*, *D*, and *U* satisfy $L \leq D \leq U$. The result should be one of the (equal or adjacent) values that would be obtained by correctly rounding *L* and *U* according to the current rounding direction, with the extra stipulation that the error with respect to *D* should have a correct sign for the current rounding direction.

Example

```
/* This is an example of the function wcstod(const wchar_t*, wchar_t**).
   This converts a string to a double value.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    wchar_t nptr[100];
    wchar_t *endptr;
    double x;
    char *line;
    int len;

    line = "3.1415926This stopped it";
    len = strlen(line);
    mbstowcs(nptr, line, len);
    x = wcstod(nptr, &endptr);
    printf("Original string: ");
    fputws(nptr, stdout);
    printf("\nDouble representation: %f\n", x);
}
```

Output

```
Original string: 3.1415926This stopped it
Double representation: 3.141593
```

See Also

wcstok

Synopsis

```
#include <wchar.h>
```

```
wchar_t wcstok(wchar_t * restrict s1, const wchar_t * restrict s2, wchar_t ** restrict ptr);
```

Purpose

Breaks a wide string into a sequence of tokens.

Return Value

The `wcstok()` function returns a pointer to the first wide character of a token, or a null pointer if there is no token.

Parameters

s1 Integer pointer argument.

s2 Integer pointer argument.

ptr Integer pointer argument.

Description

A sequence of calls to the `wcstok()` function breaks the wide string pointed to by *s1* into a sequence of tokens, each of which is delimited by a wide character from the wide string pointed to by *s2*. The third argument points to a caller-provided `wchar_t` pointer into which the `wcstok()` function stores information necessary for it to continue scanning the same wide string.

The first call in a sequence has a non-null first argument and stores an initial value in the object pointed to by *ptr*. Subsequent calls in the sequence have a null first argument and the object pointed to by *ptr* is required to have the value stored by the previous call in the sequence, which is then updated. The separator wide string pointed to by *s2* may be different from call to call.

The first call in the sequence searches the wide string pointed to by *s1* for the first wide character that is *not* contained in the current separator wide string pointed to by *s2*. If no such wide character is found, then there are no tokens in the wide string pointed to by *s1* and the `wcstok` function returns a null pointer. If such a wide character is found, it is the start of the first token.

The `wcstok()` function then searches from there for a wide character that *is* contained in the current separator wide string. If no such wide character is found, the current token extends to the end of the wide string pointed to by *s1*, and the subsequent searches in the same wide string for a token returns a null pointer. If such a wide character is found, it is overwritten by a null character, which terminates the current token.

In all cases, the `wcstok()` function stores sufficient information in the pointer pointed to by *ptr* so that subsequent calls, with a null pointer for *s1* and the unmodified pointer value for *ptr*, shall start searching just past the element overwritten by a null wide character (if any).

Example

```
/* This is an example of the function wcstok(wchar_t*, const wchar_t*,
                                             wchar_t**). This pulls out
   the tokens from a given string. There is a difference between the number
   of arguments depending on where you compile. Portability difference
   between HP and Sun.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    char *line, *line2;
    wchar_t str1[100], str2[100];
    wchar_t *endptr = NULL;
    wchar_t *token;
    int len1, len2;
```

```
line = "hello goodbye now";
line2 = " ";
len1 = strlen(line);
len2 = strlen(line2);
mbstowcs(str1, line, len1);
mbstowcs(str2, line2, len2);
fputs("Original string: ", stdout);
fputws(str1, stdout); fputs("\n", stdout);
for (token = wcstok(str1, str2, &endptr);
     token != NULL; token= wcstok(NULL, str2, &endptr)) {
    fputs("token = ", stdout); fputws(token, stdout); fputs("\n", stdout);
}
}
```

Output

```
Original string: hello goodbye now
token = hello
token = goodbye
token = now
```

See Also

wcstol

Synopsis

```
#include <wchar.h>
```

```
long int wcstol(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);
```

```
long long int wcstoll(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);
```

```
unsigned long int wcstoul(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);
```

```
unsigned long long int wcstoull(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);
```

Purpose

Converts wide string to long int, long long int, unsigned long int, or unsigned long long int.

Return Value

The `wcstol()`, `wcstoll()`, `wcstoul()`, and `wcstoull()` functions return the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, `LONG_MIN`, `LONG_MAX`, `LLONG_MIN`, `LLONG_MAX`, `ULONG_MAX`, or `ULLONG_MAX` is returned (according to the return type sign of the value, if any) and the value of the macro `ERANGE` is stored in `errno`.

Parameters

nptr Integer pointer argument.

endptr Integer pointer argument.

base Integer argument.

Description

The `wcstol()`, `wcstoll()`, `wcstoul()`, and `wcstoull()` functions convert the initial portion of the wide string pointed to by *nptr* to **long int**, **long long int**, **unsigned long int**, and **unsigned long long int** representation, respectively. First, they decompose the input string into three parts: an initial, possibly empty, sequence of white-space wide characters (as specified by the `iswspace()` function) a subject sequence resembling an integer represented in some radix determined by the value of *base*, and a final wide string of one or more unrecognized wide characters, including the terminating null wide character of the input wide string. Then, they attempt to convert the subject sequence to an integer, and return the result.

If the value of *base* is zero, the expected form of the subject sequence is that of an integer constant as described for the corresponding single-byte characters, optionally preceded by a plus or minus sign, but not including an integer suffix. If the value of *base* is between 2 and 36 (inclusive) the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters **a** (or **A**) through **z** (or **Z**) are ascribed the values of 10 through 35; only letters and digits whose ascribed values are less than that of *base* are permitted. If the value of *base* is 16, the wide characters **0x** or **0X** may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input wide string, starting with the first non-white-space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide string is empty or consists entirely of white space, or if the first non-white-space wide character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of *base* is zero, the sequence of wide characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated (in the return type). A pointer to the final wide string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

In other than the “C” locale, additional locale-specific subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

Example

```
/* This is an example of the function wcstol(const wchar_t*, wchar_t** , int).
```

```
    This converts a string to a long integer in the specified base.
*/
#include <wchar.h>

int main() {
    wchar_t nptr[100];
    wchar_t *endptr;
    int base, len;
    long retval;
    char *line;

    line= "-10110134932This stopped it";
    base = 10;
    len = strlen(line);
    mbstowcs(nptr, line, len);
    fputs("Original string: ", stdout);
    fputws(nptr, stdout);
    printf("\nLong integer representation in base %d : ", base);
    retval = wcstol(nptr, &endptr, base);
    printf("%ld\n", retval);
}
```

Output

```
Original string: -10110134932This stopped it
Long integer representation in base 10 : -2147483648
```

See Also

wcsxfrm

Synopsis

```
#include <wchar.h>
```

```
int wcsxfrm(const wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
```

Purpose

Transforms a wide string and places it in an array.

Return Value

The `wcsxfrm()` function returns the length of the transformed wide string (not including the terminating null wide character). If the value returned in `n` is greater, the contents of the array pointed to by `s1` are indeterminate.

Parameters

`s1` Integer pointer argument.

`s2` Integer pointer argument.

`n` Unsigned integer argument.

Description

The `wcsxfrm()` function transforms the wide string pointed to by `s2` and places the resulting wide string into the array pointed to by `s1`. The transformation is such that if the `wcscmp()` function is applied to two transformed wide strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the `wcscoll()` function applied to the same two original wide strings. No more than `n` wide characters are placed into the resulting array pointed to by `s1`, including the terminating null wide character. If `n` is permitted to be a null pointer.

Example

```
/* This is an example of the function wcsxfrm(wchar_t*, const wchar_t*, size_t).
   This transforms a string based on locale-specific information. It returns
   the length of the transformed string.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    char *line, *line2;
    wchar_t str1[100], str2[100];
    size_t retval, count;
    int len1;

    line = "abc";
    len1 = strlen(line);
    mbstowcs(str2, line, len1);
    fputs("Source string: ", stdout);
    fputws(str2, stdout);
    retval = wcsxfrm(str1, str2, count);
    printf("\nThe length of the transformed string is: %d\n", retval);
}
```

Output

```
Source string: abc
The length of the transformed string is: 3
```

See Also

wctob

Synopsis

```
#include <wchar.h>
#include <stdio.h>
int wctob(wint_t ic);
```

Purpose

Determines if a character corresponds to member of extended character set.

Description

The `wctob()` function determines whether `c` corresponds to a member of the extended character set whose multibyte character representation is a single byte when in the initial shift state.

Parameters

`c` Integer argument.

Return Value

The `wctob()` function returns **EOF** if `c` does not correspond to multibyte character with length one in the initial shift state. Otherwise, it returns the single-byte representation of that character as an **unsigned char** converted to an **int**.

Example

```
/* This is an example of the function wctomb(char*, wchar_t). This function
   determines the number of bytes needed to represent the equivalent multibyte
   character.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    wint_t wc;
    int retval;
    wc=L'r';

    retval = wctob(wc);
    if(retval != EOF) {
        printf("wctob() = ");
        fputc(retval, stdout);
        printf("\n");
    }
    else
        printf("error\n");
}
```

Output

```
wctob() = r
```

See Also

wmemchr

Synopsis**#include** <wchar.h>**wchar_t** wmemchr(const wchar_t *s, wchar_t c, size_t n);**Purpose**

Locates the first occurrence of a specified wide character in a stream.

Return ValueThe **wmemchr()** function returns a pointer to the located wide character, or a null pointer if the wide character does not occur in the object.**Parameters***s* Integer pointer argument.*c* Integer argument.*n* Unsigned integer argument.**Description**The **wmemchr()** function locates the first occurrence of *c* in the initial *n* wide characters of the object pointed to by *s*.**Example**

```

/* This is an example of wmemchr(const wchar_t*, wchar_t, size_t).
   This function finds and returns the located specified wide character 'c'.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    int len, result;
    size_t count;
    wchar_t *retval, str1[100], c;
    char *line;
    c='r';
    count = 20;

    line = "Hello how are you today?";
    len = strlen(line);
    mbstowcs(str1, line, len);
    retval = wmemchr(str1,c, count);
    if(retval != NULL) {
        printf("%s\n", line);
        result = retval - str1 +1;
        if(result <= 0)
            printf("There is no %c in this string.\n",c);
        else
            printf("The first '%c' is at position: %d\n", c, result);
    }
    else
        printf("Error\n");
}

```

Output

```

Hello how are you today?
The first 'r' is at position: 12

```

See Also

wmemcmp

Synopsis

```
#include <wchar.h>
```

```
int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```

Purpose

Compares the first wide characters in two objects.

Return Value

The `wmemcmp()` function returns an integer greater than, equal to, or less than zero, accordingly as the object pointed to by `s1` is greater than, equal to, or less than the object pointed to by `s2`.

Parameters

`s1` Integer pointer argument.

`s2` Integer pointer argument.

`n` Unsigned integer argument.

Description

The `wmemcmp()` function compares the first `n` wide characters of the object pointed to by `s1` to the first `n` wide characters of the object pointed to by `s2`.

Example

```
/* This is an example of the function wmemcmp(const wchar_t*, const wchar_t*,
size_t). This compares the specified number of characters of two strings.
If string1 substring < string2 substring, then a negative value is returned.
If string1 substring = string2 substring, then zero is returned.
If string1 substring > string2 substring, then a positive value is returned.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    const char *line, *line2;
    wchar_t str1[100], str2[100];
    size_t count;
    int retval, len1, len2;
    count = 6;
    line = "hellohello";
    line2 = "hello";
    len1 = strlen(line);
    len2 = strlen(line2);

    mbstowcs(str1, line, len1);
    mbstowcs(str2, line2, len2);
    retval = wmemcmp(str1, str2, count);
    if(retval != NULL) {
        fputs("String1: ", stdout);
        fputws(str1, stdout);
        fputs("\nString2: ", stdout);
        fputws(str2, stdout);
        printf("\nNumber of characters to compare: %d\n", count);
        if(retval > 0)
            printf("String1 is greater than String2\n");
        else if(retval == 0)
            printf("String1 is identical to String2\n");
    }
}
```

```
    else if(retval < 0)
        printf("String1 is less than String2\n");
    else
        printf("Error!\n");
    printf("wmemcmp(str1,str2,count) = %d\n", retval);
}
else
    printf("Error\n");
}
```

Output

```
String1: hellohello
String2: hello
Number of characters to compare: 6
String1 is greater than String2
wmemcmp(str1,str2,count) = 104
```

See Also

wmemcpy

Synopsis

```
#include <wchar.h>
```

```
wchar_t wmemcpy(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
```

Purpose

Copies wide characters from one object to another.

Return Value

The `wmemcpy()` returns the value of `s1`.

Parameters

`s1` Integer pointer argument.

`s2` Integer pointer argument.

`n` Unsigned integer argument.

Description

The `wmemcpy()` function copies `n` wide characters from the object pointed to by `s2` to the object pointed to by `s1`.

Example

```

/* This is an example of the function wmemcpy(wchar_t*, const wchar_t*,
                                             size_t).
   This copies the specified number of characters from string2 to string1.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    wchar_t str1[100], str2[100];
    char *line;
    wchar_t *retval;
    size_t count;
    int len;

    line = "How are you today?";
    count = 11;
    len = strlen(line);
    mbstowcs(str2, line, len);
    retval = wmemcpy(str1, str2, count);
    if(retval != NULL) {
        printf("Original string: ");
        fputws(str2, stdout);
        printf("\nNumber of characters to copy: %d", count);
        printf("\nCopied string: ");
        fputws(str1, stdout);
        fputs("\n", stdout);
    }
    else
        printf("Error\n");
}

```

Output

Original string: How are you today?

```
Number of characters to copy: 11  
Copied string: How are you
```

See Also

wmemmove

Synopsis

```
#include <wchar.h>
```

```
wchar_t wmemmove(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
```

Purpose

Copies wide characters from one object to another.

Return Value

The `wmemmove()` returns the value of `s1`.

Parameters

`s1` Integer pointer argument.

`s2` Integer pointer argument.

`n` Unsigned integer argument.

Description

The `wmemmove()` function copies `n` wide characters from the object pointed to by `s2` to the object pointed to by `s1`. Copying takes place as if the `n` wide characters from the object pointed to by `s2` are first copied into a temporary array of `n` wide characters that does not overlap the objects pointed to by `s1` or `s2`, and then the `n` wide characters from the temporary array are copied into the object pointed to by `s1`.

Example

```

/* This is an example of the function wmemmove(wchar_t*, const wchar_t*,
                                             size_t)
   This copies the specified number of characters from string2 to string1.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    wchar_t str1[100], str2[100];
    char *line, *line2;
    wchar_t *retval;
    size_t count;
    int len;

    line = "What is your name?";
    count = 15;
    len = strlen(line);
    retval = wmemmove(str1, str2, count);
    if(retval != NULL) {
        printf("Original line: %s\n", line );
        mbstowcs(str2, line, len);
        printf("Number of characters to move: %d", count);
        printf("\nNew string1: ");
        fputws(wmemmove(str1, str2, count), stdout);
        fputs("\n", stdout);
    }
    else
        printf("Error\n");
}

```

Output

```
Original line: What is your name?  
Number of characters to move: 15  
New string1: What is your na
```

See Also

wmemset

Synopsis

```
#include <wchar.h>
```

```
wchar_t wmemset(const wchar_t *s, wchar_t c, size_t n);
```

Purpose

Copies a single value into specified number of locations in an object.

Return Value

The `wmemset()` function returns the value of `s`.

Parameters

`s` Integer pointer argument.

`c` Integer argument.

`n` Unsigned integer argument.

Description

The `wmemset()` function copies the value of `c` into each of the first `n` wide characters of the object pointed to by `s`.

Example

```
/* This is an example of wmemset(wchar_t*, wchar_t, size_t). This function
   copies the specified wide character into the first n positions of str1.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    int len;
    size_t count;
    wchar_t *retval, str1[100], c;
    char *line;
    c='r';
    count = 10;

    line = "Hello how are you today?";
    len = strlen(line);
    mbstowcs(str1, line, len);
    retval = wmemset(str1,c, count);
    if(retval != NULL) {
        printf("Original string: %s\n", line);
        printf("Character to replace with: ");
        fputwc(c, stdout);
        printf("\nThe number of characters to replace: %d\n", count);
        printf("The new string is: ");
        fputws(retval, stdout);
        fputs("\n", stdout);
    }
    else
        printf("Error\n");
}
```

Output

```
Original string: Hello how are you today?
```

Chapter 21: Wide Character Functions — <wctype.h>

Character to replace with: r
The number of characters to replace: 10
The new string is: rrrrrrrrrrare you today?

See Also

Chapter 22

Wide Character Functions — `<wctype.h>`

The header `wctype.h` declares three data types, one macro, and many functions.

The types declared are

wint_t

which is an integer type unchanged by default argument promotions that can hold any value corresponding to members of the extended character set, as well as at least one value that does not correspond to any member of the extended character set (see **WEOF** below);

wctrans_t

which is a scalar type that can hold values which represent locale-specific character mappings; and

wctype_t

which is a scalar type that can hold values which represent locale-specific character classifications.

The macro defined is

WEOF

which expands to a constant expression of type **wint_t** whose value does not correspond to any member of the extended character set. It is accepted (and returned) by several functions in this subclause to indicate *end-of-file*, that is, no more input from a stream. It is also used as a wide-character value that does not correspond to any member of the extended character set.

The functions declared are grouped as follows:

- Functions that provide wide-character classification;
- Extensible functions that provide wide-character classification;
- Functions that provide wide-character case mapping;
- Extensible functions that provide wide-character mapping.

For all functions described in this subclause that accept an argument of type **wint_t**, the value shall be representable as a **whar_t** or shall equal the value of the macro **WEOF**. If this argument has any other value, the behavior is undefined.

The behavior of these functions is affected by the **LC_TYPE** category of the current locale.

Public Data

None.

Functions

The following functions are prototyped in the header file **wctype.h**.

Functions	Description
iswalnum()	Tests for alpha-numeric wide character.
iswalpha()	Tests for upper or lower case alphabetic wide character.
iswcntrl()	Tests for control wide character.
iswctype()	Determines whether character has properties to be a wide character.
iswdigit()	Tests for decimal-digit wide characters.
iswgraph()	Tests for printing wide character.
iswlower()	Tests for lowercase wide characters.
iswprint()	Tests for printing wide character.
iswpunct()	Tests for punctuation wide characters.
iswspace()	Tests for white-space wide characters.
iswupper()	Tests for uppercase letter wide characters.
iswxdigit()	Tests for hexadecimal-digit wide characters.
towctrans()	Maps wide characters.
towlower()	Converts uppercase letter to corresponding lowercase letter.
towupper()	Converts lowercase letter to corresponding uppercase letter.
wctrans()	Describes mapping between wide characters.
wctype()	Describes a class of wide characters.

Macros

The following macros are defined for the **wctype** header.

Macro	Description
WEOF	Used to indicate <i>end-of-file</i> .

Declared Types

The following types are declared in the **wctype** header.

Declared Types	Description
wint_t	int-holds values corresponding to members of the extended set.
wctrans_t	scalar-holds values which represent locale-specific character mappings.
wctype_t	scalar-holds values which represent locale-specific character classifications.

Portability

This header has no known portability problem.

iswalnum

Synopsis

```
#include <wctype.h>
int iswalnum(wint_t wc);
```

Purpose

Tests to see if a character is a wide alpha-numeric character.

Return Value

The `iswalnum()` function returns a boolean value. It returns true if the character is a wide alpha-numeric character.

Parameters

`wc` Integer argument.

Description

The `iswalnum()` function tests for any wide character for which `iswalpha()` or `iswdigit()` is true.

Example

```
/* This is an example of the function iswalnum(wint_t ). This function checks
   if the parameter is an alphanumeric character. If true, the function
   returns a positive number, or zero if false.
*/
#include <wctype.h>
#include <wchar.h>

int main() {
    wint_t wc;

    wc = 'i';
    fputwc(wc, stdout);
    if(iswalnum(wc) != 0)
        printf(": is a alphanumeric character.\n");
    else
        printf(": is not an alphanumeric character.\n");
}
```

Output

```
i: is a alphanumeric character.
```

See Also

iswalpha

Synopsis

```
#include <wctype.h>
int iswalpha(wint_t wc);
```

Purpose

Tests to see if a character is an upper or lower case alphabetic wide character.

Return Value

The **iswalpha()** function returns a boolean value. It returns true if the character is an upper or lower case alphabetic wide character.

Parameters

wc Integer argument.

Description

The **iswalpha()** function tests for any wide character for which **iswupper()** or **iswlower()** is true, or any wide character that is one of a locale-specific set of alphabetic wide characters for which none of **iswcntrl()**, **iswdigit()**, **iswpunct()**, or **iswspace()** is true.

Example

```
/* This is an example for iswalpha(wint_t). This function will return a
   non-zero value if the parameter is an alphabetic character.
*/
#include <wctype.h>
#include <wchar.h>

int main() {
    wint_t wc;

    wc = 'i';
    fputwc(wc, stdout);
    if(iswalpha(wc) > 0)
        printf(": is an alphabetic character\n");
    else
        printf(": is not an alphabetic character\n");
}
```

Output

```
i: is an alphabetic character
```

See Also

iswcntrl

Synopsis

```
#include <wctype.h>
int iswcntrl(wint_t wc);
```

Purpose

Tests for a control wide character.

Return Value

The `iswcntrl()` function returns a boolean value. If the character is a control wide character the function returns a true.

Parameters

`wc` Integer argument.

Description

The `iswcntrl()` function tests for any control wide character.

Example

```
/* This is an example for iswcntrl(wint_t). This returns a non-zero number
   if it is a control wide character. If not, it will return zero.
*/
#include <wctype.h>
#include <wchar.h>

int main() {
    wint_t wc;

    wc = 'i';
    fputwc(wc, stdout);
    if(iswcntrl(wc) > 0)
        printf(": is a wide control character.\n");
    else
        printf(": is not a wide control character.\n");
}
```

Output

```
i: is not a wide control character.
```

See Also

iswctype

Synopsis

```
#include <wctype.h>
int iswctype(wint_t wc, wctype_t desc);
```

Purpose

Determines whether a character has the properties to be a wide character.

Return Value

The `iswctype()` function returns nonzero (true) if and only if the value of the wide character `wc` has the property described by `desc`.

Parameters

`wc` Integer argument.
`desc` Scalar argument.

Description

The `iswctype()` function determines whether the wide character `wc` has the property described by `desc`. The current setting of the `LC_CTYPE` category shall be the same as during the call to `wctype()` that returned the value `desc`.

Each of the following expressions has a truth-value equivalent to the call to the wide-character classification function in the comment that follows the expression:

```
iswctype(wc, wctype("alnum")) // iswgraph(wc)
iswctype(wc, wctype("graph")) // iswgraph(wc)
iswctype(wc, wctype("cntrl")) // iswcntrl(wc)
iswctype(wc, wctype("digit")) // iswdigit(wc)
iswctype(wc, wctype("graph")) // iswgraph(wc)
iswctype(wc, wctype("lower")) // iswlower(wc)
iswctype(wc, wctype("print")) // iswprint(wc)
iswctype(wc, wctype("punct")) // iswpunct(wc)
iswctype(wc, wctype("space")) // iswspace(wc)
iswctype(wc, wctype("upper")) // iswupper(wc)
iswctype(wc, wctype("xdigit")) // iswxdigit(wc)
```

Example

```
/* This is an example for iswctype(wint_t, wctype_t). This test the wint_t
   parameter for the specified property. If the wide character has the
   specified property, than a nonzero value will be returned. If it does not,
   then zero will be returned.
*/
#include <wctype.h>
#include <wchar.h>

int main() {
    wint_t wc;
    char *prop;

    prop = "upper";
    wc = 'I';
    fputs("The wide character '", stdout);
    fputwc(wc, stdout);
    if (iswctype(wc, wctype(prop)) > 0)
        printf("' has the property: %s\n", prop);
    else
        printf("' does not have the property: %s\n", prop);
}
```

```
}
```

Output

The wide character 'I' has the property: upper

See Also

iswdigit

Synopsis

```
#include <wctype.h>
int iswdigit(wint_t wc);
```

Purpose

Tests for decimal-digit wide characters.

Return Value

The **iswdigit()** function returns a boolean value which is true when the character is a decimal-digit character.

Parameters

wc Integer argument.

Description

The **iswdigit()** function tests for any wide character that corresponds to a decimal-digit character.

Example

```
/* This is an example for iswdigit(wint_t). This function will return a
   non-zero value if the parameter is digit.
*/
#include <wctype.h>
#include <wchar.h>

int main() {
    wint_t wc;

    wc = '1';
    fputwc(wc, stdout);
    if(iswdigit(wc) > 0)
        printf(": is a digit.\n");
    else
        printf(": is not a digit.\n");
}
```

Output

```
1: is a digit.
```

See Also

iswgraph

Synopsis

```
#include <wctype.h>
int iswgraph(wint_t wc);
```

Purpose

Tests for printing wide characters.

Return Value

The **iswgraph()** function returns a boolean value which is true when the character is a printing wide character.

Parameters

wc Integer argument.

Description

The **iswgraph()** function tests for any wide character for which **iswprint()** is true and **iswspace()** is false.

Example

```
/* This is an example for iswgraph(wint_t). This function will return a
   non-zero value if the parameter is a printable character except space.
*/
#include <wctype.h>
#include <wchar.h>

int main() {
    wint_t wc;

    wc = 'i';
    fputwc(wc, stdout);
    if(iswgraph(wc) != 0)
printf(": is a printable character.\n");
    else
printf(": is not a printable character.\n");
}
```

Output

```
i: is a printable character.
```

See Also

iswlower

Synopsis

```
#include <wctype.h>
int iswlower(wint_t wc);
```

Purpose

Tests for lowercase wide characters.

Return Value

The **iswlower()** function returns a boolean value which is true when the argument is a lower case letter.

Parameters

wc Integer argument.

Description

The **iswlower()** function tests for any wide character that corresponds to a lowercase letter or is one of a locale-specific set of wide characters for which none of **iswcntrl()**, **iswdigit()**, **iswpunct()**, or **iswspace()** is true.

Example

```
/* This is an example for iswlower(wint_t). This function will return a
   non-zero value if the parameter is a lowercase letter.
*/
#include <wctype.h>
#include <wchar.h>

int main() {
    wint_t wc;

    wc = 'i';
    fputwc(wc, stdout);
    if(iswlower(wc) != 0)
        printf(": is a lowercase letter.\n");
    else
        printf(": is not a lowercase letter.\n");
}
```

Output

```
i: is a lowercase letter.
```

See Also

iswprint

Synopsis

```
#include <wctype.h>
int iswprint(wint_t wc);
```

Purpose

Tests for printing wide characters.

Return Value

The `iswprint()` function returns a boolean value which is true when the argument is a printing wide character.

Parameters

`wc` Integer argument.

Description

The `iswprint()` function tests for any printing wide character.

Example

```
/* This is an example for iswprint(wint_t). This function will return a
   non-zero value if the parameter is a printable character including space
   */
#include <wctype.h>
#include <wchar.h>

int main() {
    wint_t wc;

    wc = 'i';
    fputwc(wc, stdout);
    if(iswprint(wc) != 0)
        printf(": is a printable character.\n");
    else
        printf(": is not a printable character.\n");
}
```

Output

```
i: is a printable character.
```

See Also

iswpunct

Synopsis

```
#include <wctype.h>
int iswpunct(wint_t wc);
```

Purpose

Tests for punctuation wide characters.

Return Value

The `iswpunct()` function returns a boolean value which is true when the argument is a punctuation wide character.

Parameters

`wc` Integer argument.

Description

The `iswpunct()` function tests for any printing wide character that is one of a locale-specific set of punctuation wide characters for which neither `iswspace()` nor `iswalnum()` is true.

Example

```
/* This is an example for iswpunct(wint_t). This function will return a
   non-zero value if the parameter is x punctuation character.
*/
#include <wctype.h>
#include <wchar.h>

int main() {
    wint_t wc;

    wc = '.';
    fputwc(wc, stdout);
    if(iswpunct(wc) != 0)
        printf(": is a punctuation mark.\n");
    else
        printf(": is not a punctuation mark.\n");
}
```

Output

```
.: is a punctuation mark.
```

See Also

iswspace

Synopsis

```
#include <wctype.h>
int iswspace(wint_t wc);
```

Purpose

Tests for white-space wide characters.

Return Value

The **iswspace()** function returns a boolean value which is true when the argument is a white-space wide characters.

Parameters

wc Integer argument.

Description

The **iswspace()** function tests for any wide character that corresponds to a locale-specific set of white-space wide characters for which none of **iswalnum()**, **iswgraph()**, or **iswpunct()** is true.

Example

```
/* This is an example for iswspace(wint_t). This function will return a
   non-zero value if the parameter is a white space character.
*/
#include <wctype.h>
#include <wchar.h>

int main() {
    wint_t wc;

    wc = ' ';
    fputwc(wc, stdout);
    if(iswspace(wc) != 0)
        printf(": is a white space character.\n");
    else
        printf(": is not a white space character.\n");
}
```

Output

```
: is a white space character.
```

See Also

iswupper

Synopsis

```
#include <wctype.h>
int iswupper(wint_t wc);
```

Purpose

Tests for uppercase letter wide characters.

Return Value

The **iswupper()** function returns a boolean value which is true when the argument is an uppercase letter wide character.

Parameters

wc Integer argument.

Description

The **iswupper()** function tests for any wide character that corresponds to an uppercase letter or is one of a locale-specific set of wide characters for which none of **iswcntrl()**, **iswdigit()**, **iswpunct()**, or **iswspace()** is true.

Example

```
/* This is an example for the function iswupper(wint_t). This function returns
   a non negative number if the input is an uppercase letter.
*/
#include <wctype.h>
#include <wchar.h>

int main() {
    wint_t wc;

    wc = 'I';
    fputwc(wc, stdout);
    if(iswupper(wc) != 0)
        printf(": is an uppercase letter.\n");
    else
        printf(": is not an uppercase letter.\n");
}
```

Output

I: is an uppercase letter.

See Also

iswxdigit

Synopsis

```
#include <wctype.h>
int iswxdigit(wint_t wc);
```

Purpose

Tests for hexadecimal-digit wide characters.

Return Value

The `iswxdigit()` function returns a boolean value which is true when the argument is a hexadecimal-digit character.

Parameters

`wc` Integer argument.

Description

The `iswxdigit()` function tests for any wide character that corresponds to a hexadecimal-digit character.

Example

```
/* This is an example for iswxdigit(wint_t). This function will return a
   non-zero value if the parameter is a hexadecimal digit.
*/
#include <wctype.h>
#include <wchar.h>

int main() {
    wint_t wc;

    wc = 'A';
    fputwc(wc, stdout);
    if(iswxdigit(wc) != 0)
        printf(": is a hexadecimal digit.\n");
    else
        printf(": is not a hexadecimal digit.\n");
}
```

Output

A: is a hexadecimal digit.

See Also

towctrans

Synopsis

```
#include <towctrans.h>
wint_t towctrans(wint_t wc);
```

Purpose

Maps wide characters.

Return Value

The `towctrans()` function returns the mapped value of `wc` using the mapping described by `desc`.

Parameters

`wc` Integer argument.
`desc` Scalar argument.

Description

The `towctrans()` function maps the wide character `wc` using the mapping described by `desc`. The current setting of the `LC_TYPE` category shall be the same as during the call to `wctrans()` that returned the value `desc`.

Each of the following expressions behaves the same as the call to the wide-character case- mapping function in the comment that follows the expression:

```
towctrans(wc, wctrans("tolower")) /* tolower(wc) */
towctrans(wc, wctrans("toupper")) /* toupper(wc) */
```

Example

```
/* This is an example for towctrans(wint_t, wctrans_t). This test the wint_t
   parameter for the specified property. If the wide character has the
   specified property, than a nonzero value will be returned. If it does not,
   then zero will be returned.
*/
#include <wctype.h>
#include <wchar.h>

int main() {
    wint_t wc;
    char *prop;

    prop = "toupper";
    wc = 'i';
    fputs("The wide character '", stdout);
    fputwc(wc, stdout);
    printf("' has been changed to: '");
    fputwc(towctrans(wc, wctrans(prop)), stdout);
    printf "'. \nBy the property of %s.\n", prop);
}
```

Output

The wide character 'i' has been changed to: 'I'.
 By the property of toupper.

See Also

tolower

Synopsis

```
#include <tolower.h>
wint_t tolower(wint_t wc);
```

Purpose

Converts uppercase letter to corresponding lowercase letter.

Return Value

If the argument is a wide character for which **iswupper()** is true and there are one or more corresponding wide characters, as specified by the current locale, for which **iswlower()** is true, the **tolower()** function returns one of the corresponding wide characters; otherwise, the argument is returned unchanged.

Parameters

wc Integer argument.

Description

The **tolower()** function converts an uppercase letter to a corresponding lowercase letter.

Example

```
/* This is an example of the function tolower(wint_t). This will convert the
   wide character into its lowercase representation.
*/
#include <wctype.h>
#include <wchar.h>

int main() {
    wint_t wc;
    wc = 'A';

    printf("Before: ");
    fputwc(wc, stdout);
    printf("\n");
    printf("After: ");
    fputwc(tolower(wc), stdout);
}
```

Output

```
Before: A
After: a
```

See Also

towupper

Synopsis

```
#include <towupper.h>
wint_t towupper(wint_t wc);
```

Purpose

Converts lowercase letter to corresponding uppercase letter.

Return Value

If the argument is a wide character for which **iswlower()** is true and there are one or more corresponding wide characters, as specified by the current locale, for which **iswupper()** is true, the **towupper()** function returns one of the corresponding characters; otherwise, the argument is returned unchanged.

Parameters

wc Integer argument.

Description

The **towupper()** function converts a lowercase letter to a corresponding uppercase letter.

Example

```
/* This is an example of the function towupper(wint_t). This function converts
   lowercase letters to uppercase.
*/
#include <wctype.h>
#include <wchar.h>

int main() {
    wint_t wc;
    wc = 'i';

    printf("Before: ");
    fputwc( wc, stdout);
    printf("\n");
    printf("After: ");
    fputwc( towupper(wc), stdout);
    printf("\n");
}
```

Output

```
Before: i
After: I
```

See Also

wctrans

Synopsis

```
#include <wctype.h>
wctrans_t wctrans(const char *property);
```

Purpose

Describes mapping between wide characters.

Return Value

If *property* identifies a valid mapping of wide characters according to the **LC_CTYPE** category of the current locale, the **wctrans()** function returns a nonzero value that is valid as the second argument to the **towctrans()** function; otherwise, it returns zero.

Parameters

**property* Character argument.

Description

The **wctype()** function constructs a value with type **wctrans_t**() that describes a mapping between wide characters identified by the string argument **property**.

The strings listed in the description of the **towctrans()** function shall be valid in all locales as *property* arguments to the **wctrans()** function.

Example

```
/* This is an example of the function wctrans(const char*). This returns a
   non-negative number if the string is a valid wctrans.
*/
#include <wctype.h>

int main() {
    char *prop;
    prop = "tolower";
    if(wctrans(prop) > 0)
    printf("'%s' is a valid wctrans.\n", prop);
    else
        printf("'%s' is not a valid wctrans.\n", prop);
}
```

Output

```
'tolower' is a valid wctrans.
```

See Also

wctype

Synopsis

```
#include <wctype.h>
wctype_t wctype(const char *property);
```

Purpose

Describes a class of wide characters.

Returns

If *property* identifies a valid class of wide characters according to the **LC_CTYPE** category of the current locale, the **wctype()** function returns a nonzero value that is valid as the second argument to the **iswctype()** function; otherwise, it returns zero.

Parameters

**property* Character argument.

Description

The **wctype()** function constructs a value with type **wctype_t()** that describes a class of wide characters identified by the string argument *property*.

The strings listed in the description of the **iswctype()** function shall be valid in all locales as *property* arguments to the **wctype()** function.

Example

```
/* This is an example of the function wctype(const char*). This returns a
   non-negative number if the string is a valid wctype.
*/
#include <wctype.h>

int main() {
    char *prop;
    prop = "xdigit";
    if(wctype(prop) > 0)
        printf("'%s' is a valid wctype.\n", prop);
    else
        printf("'%s' is not a valid wctype.\n", prop);
}
```

Output

```
'xdigit' is a valid wctype.
```

See Also

iswctype()

Appendix A

Functions Not Supported in Specific Platforms

Most functions in Ch are supported across different platforms. Functions which are not supported under different operating systems are listed below.

A.1 HPUX

The following functions are implemented with return error code only under the HPUX operating system.

Function	Header File: windows/winsock.h	Return Value
HANDLE WSAAPI WSAAsyncGetHostByAddr(HWND <i>hwnd</i> , unsigned int <i>wMsg</i> , const char FAR * <i>addr</i> , int <i>len</i> , int <i>type</i> , char FAR * <i>buf</i> , int <i>buflen</i>)		0
HANDLE WSAAPI WSAAsyncGetHostByName(HWND <i>hwnd</i> , unsigned int <i>wMsg</i> , const char FAR * <i>name</i> , char FAR * <i>buf</i> , int <i>buflen</i>)		0
HANDLE WSAAPI WSAAsyncGetProtoByName(HWND <i>hwnd</i> , unsigned int <i>wMsg</i> , const char FAR * <i>name</i> , char FAR * <i>buf</i> , int <i>buflen</i>)		0
HANDLE WSAAPI WSAAsyncGetProtoByNumber(HWND <i>hwnd</i> , unsigned int <i>wMsg</i> , int <i>number</i> , char FAR * <i>buf</i> , int <i>buflen</i>)		0
HANDLE WSAAPI WSAAsyncGetServByName(HWND <i>hwnd</i> , unsigned int <i>wMsg</i> , const char FAR * <i>name</i> , const char FAR * <i>proto</i> , char FAR * <i>buf</i> , int <i>buflen</i>)		0
HANDLE WSAAPI WSAAsyncGetServByPort(HWND <i>hwnd</i> , unsigned int <i>wMsg</i> , int <i>port</i> , const char FAR * <i>proto</i> , char FAR * <i>buf</i> , int <i>buflen</i>)		0
int WSAAPI WSAAsyncSelect(SOCKET <i>s</i> , HWND <i>hwnd</i> , unsigned int <i>wMsg</i> , long <i>lEvent</i>)		- 1
int WSAAPI WSACancelAsyncRequest(HANDLE <i>AsyncTaskHandle</i>)		- 1
int WSAAPI WSACancelBlockingCall(void)		- 1
int WSAAPI WSACleanup(void)		0
void WSAAPI WSAGetLastError(void)		N/A
BOOL WSAAPI WSAIsblocking(void)		False
BOOL WSAAPI WSASetBlockingHook(FARPROC <i>lpBlockFunc</i>)		NULL
void WSAAPI WSASetLastError(int <i>iError</i>)		N/A
int WSAAPI WSAStartup(WORD <i>wVersionRequested</i> , LPWSADATA <i>lpWSADATA</i>)		0
int WSAAPI WSAUnhookBlockingHook(void)		- 1
int WSAAPI ioctlsocket(SOCKET <i>s</i> , long <i>cmd</i> , u_long * <i>argp</i>)		- 1

Chapter A: Appendix A: Functions Not Supported in Specific Platforms

Function	Header File: netdb.h	Return Value
struct hostent *gethostbyname_r(const char *name, struct hostent *result, char *buffer, int *buflen, int *h_errnop)		NULL
struct hostent *gethostbyaddr_r(const char *addr, int length, int type, struct hostent *result, char *buffer, int buflen, int *h_errnop)		NULL
struct hostent *gethostent_r(struct hostent *result, char *buffer, int buflen, int *h_errnop)		NULL
struct servent *getservbyname_r(const char *name, const char *proto, struct servent *result, char *buffer, int buflen)		NULL
struct servent *getservbyport_r(int port, const char *proto, struct servent *result, char *buffer, int buflen)		NULL
struct servent *getservent_r(struct hostent *result, char *buffer, int buflen)		NULL
struct netent *getnetbyname_r(const char *name, struct netent *result, char *buffer, int buflen)		NULL
struct netent *getnetbyaddr_r(long net, int type, struct netent *result, char *buffer, int buflen)		NULL
struct netent *getnetent_r(struct netent *result, char *buffer, int buflen)		NULL
struct protoent *getprotobyname_r(const char *name, struct protoent *result, char *buffer, int buflen)		NULL
struct protoent *getprotobynumber_r(int proto, struct protoent *result, char *buffer, int buflen)		NULL
struct protoent *getprotoent_r(struct protoent *result, char *buffer, int buflen)		NULL

Function	Header File: math.h	Return Value
double gamma_r(double x, int *signagamp)		- 1
double scalbn(double x, int *n)		- 1
int isnormal(<i>real-floating</i> x)		0
int signbit(<i>real-floating</i> x)		- 1

Function	Header File:unistd.h	Return Value
char getlogin_r(char *name, int namelen)		NULL
char *ttyname_r(int fildes, char *buf, int len)		NULL
int setegid(uid_t egid)		0
int seteuid(uid_t egid)		0

Function	Header File: grp.h	Return Value
struct group *getgrnam_r(const char *name, struct group *result, char *buffer, int buflen)		NULL
struct group *getgrgid_r(gid_t gid, struct group *result, char *buffer, int buflen)		NULL
struct group *fgetgrent_r(FILE *f, struct group *result, char *buffer, int buflen)		NULL
struct group *fgetgrent(FILE *f)		NULL

Function	Header File: mqueue.h	Return Value
----------	-----------------------	--------------

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned int *msg_prio)		0
<hr/>		
Function	Header File: sys/procset.h	Return Value
int sigsendset(procset_t *psp, int sig)		- 1
<hr/>		
Function	Header File: libintl.h	Return Value
char *gettext(const char *msgid)		NULL
char *dgettext(const char *domainname, const char *msgid)		NULL
char *textdomain(const char *domainname)		NULL
char *bindtextdomain(const char *domainname)		NULL
<hr/>		
Function	Header File: time.h	Return Value
int asctime(char *s, const char *format, const struct tm *timeptr)		0
int cftime(char *s, const char *format, const time_t clock)		0
<hr/>		
Function	Header File: wchar.h	Return Value
wint_t btowc(int c)		WEOF
int fwide(FILE *stream, int c)		0
int mbsinit(const mbstate_t *ps)		0
size_t mbrlen(const char *str, size_t length, mbstate_t ps)		0
size_t mbrtowc(wchar_t *str1, const char *str2, size_t length, mbstate_t ps)		- 1
size_t mbsrtowcs(wchar_t *str1, const char **str2, size_t length, mbstate_t *ps)		- 1
size_t wctomb(char *str1, wchar_t s, mbstate_t *ap)		- 1
size_t wcsrtombs(char *str1, const char **str2, size_t length, mbstate_t *ps)		- 1
wchar_t wcsstr(const wchar_t *str1, const wchar_t *str2)		NULL
int wctob(wint_t wc)		EOP
wchar_t wmemchr(const wchar_t *str1, wchar_t c, size_t count)		NULL
int wmemcmp(const wchar_t *str1, const wchar_t *str2, size_t count)		NULL
wchar_t *wmemmove(wchar_t *str1, const wchar_t *str2, size_t count)		NULL
wchar_t *wmemset(wchar_t *str1, wchar_t str2, size_t count)		NULL
wchar_t *wmemcpy(wchar_t *str1, const wchar_t *str2, size_t count)		NULL
<hr/>		
Function	Header File: wctype.h	Return Value
wint_t towctrans(wint_t wc, wctrans_t desc)		- 1
wctrans_t wctrans(const char *prop)		0

A.2 Linux

The following functions are implemented with return error code only under the Linux operating system.

Chapter A: Appendix A: Functions Not Supported in Specific Platforms

Function	Header File: windows/winsock.h	Return Value
HANDLE WSAAPI WSAAsyncGetHostByAddr(HWND <i>hwnd</i> , unsigned int <i>wMsg</i> , const char FAR <i>*addr</i> , int <i>len</i> , int <i>type</i> , char FAR <i>*buf</i> , int <i>buflen</i>)		0
HANDLE WSAAPI WSAAsyncGetHostByName(HWND <i>hwnd</i> , unsigned int <i>wMsg</i> , const char FAR <i>*name</i> , char FAR <i>*buf</i> , int <i>buflen</i>)		0
HANDLE WSAAPI WSAAsyncGetProtoByName(HWND <i>hwnd</i> , unsigned int <i>wMsg</i> , const char FAR <i>*name</i> , char FAR <i>*buf</i> , int <i>buflen</i>)		0
HANDLE WSAAPI WSAAsyncGetProtoByNumber(HWND <i>hwnd</i> , unsigned int <i>wMsg</i> , int <i>number</i> , char FAR <i>*buf</i> , int <i>buflen</i>)		0
HANDLE WSAAPI WSAAsyncGetServByName(HWND <i>hwnd</i> , unsigned int <i>wMsg</i> , const char FAR <i>*name</i> , const char FAR <i>*proto</i> , char FAR <i>*buf</i> , int <i>buflen</i>)		0
HANDLE WSAAPI WSAAsyncGetServByPort(HWND <i>hwnd</i> , unsigned int <i>wMsg</i> , int <i>port</i> , const char FAR <i>*proto</i> , char FAR <i>*buf</i> , int <i>buflen</i>)		0
int WSAAPI WSAAsyncSelect(SOCKET <i>s</i> , HWND <i>hwnd</i> , unsigned int <i>wMsg</i> , long <i>lEvent</i>)		- 1
int WSAAPI WSACancelAsyncRequest(HANDLE <i>AsyncTaskHandle</i>)		- 1
int WSAAPI WSACancelBlockingCall(void)		- 1
int WSAAPI WSACleanup(void)		0
void WSAAPI WSAGetLastError(void)		N/A
BOOL WSAAPI WSAIsblocking(void)		False
BOOL WSAAPI WSASetBlockingHook(FARPROC <i>lpBlockFunc</i>)		NULL
void WSAAPI WSASetLastError(int <i>iError</i>)		N/A
int WSAAPI WSAStartup(WORD <i>wVersionRequested</i> , LPWSADATA <i>lpWSAData</i>)		0
int WSAAPI WSAUnhookBlockingHook(void)		- 1
int WSAAPI ioctlsocket(SOCKET <i>s</i> , long <i>cmd</i> , u_long <i>*argp</i>)		- 1

Function	Header File: aio.h	Return Value
int aio_read(struct aiocb <i>*aiocbp</i>)		- 1
int aio_write(struct aiocb <i>*aiocbp</i>)		- 1
int lio_listio(int <i>mode</i> , struct aiocb <i>*list</i> [], int <i>nent</i> , struct sigevent <i>*sig</i>)		- 1
int aio_error(const struct aiocb <i>*aiocbp</i>)		- 1
int aio_return(struct aiocb <i>*aiocbp</i>)		- 1
int aio_cancel(int <i>fildevs</i> , struct aiocb <i>*aiocbp</i>)		- 1
int aio_suspend(const struct aiocb <i>*aiocbp</i> [], int <i>nent</i> , const struct timespec <i>*timeout</i>)		- 1
int aio_fsync(int <i>op</i> , struct aiocb <i>*aiocbp</i>)		- 1

Function	Header File: time.h	Return Value
char <i>*ctime_r</i> (const time_t <i>*clock</i> , char <i>*buf</i> , int <i>buflen</i>)		NULL
char <i>*ctime</i> (char <i>*s</i> , char <i>*format</i> , const time_t <i>*clock</i>)		0
struct tm <i>*localtime_r</i> (const time_t <i>*clock</i> , struct tm <i>*res</i>)		NULL
struct tm <i>*gmtime_r</i> (const time_t <i>*clock</i> , struct tm <i>*res</i>)		NULL
char <i>*asctime_r</i> (const struct tm <i>*tm</i> , char <i>*buf</i> , int <i>buflen</i>)		NULL
int clock_settime(clockid_t <i>clock_id</i> , const struct timespec <i>*tp</i>)		- 1
int clock_gettime(clockid_t <i>clock_id</i> , const struct timespec <i>*tp</i>)		- 1
int clock_getres(clockid_t <i>clock_id</i> , const struct timespec <i>*res</i>)		- 1

Chapter A: Appendix A: Functions Not Supported in Specific Platforms

int timer_create(clockid_t clock_id, struct sigevent *evp, timer_t *timerid)	- 1
int timer_delete(timer_t timerid)	- 1
int timer_settime(timer_t timerid, int flags, const struct itimerspec *value, struct itimerspec *ovalue)	- 1
int timer_gettime(timer_t timerid, struct timespec *value)	- 1
int timer_getoverrun(timer_t timerid)	- 1
int nanosleep(const struct timespec *rqtp, struct timespec *rmtpt)	- 1
char ascftime(char *s, const char *format, const struct tm *timeptr)	0

Function	Header File: libgen.h	Return Value
char* regcomp(const char *string1, /* char*string2 */ ..., /* (char*)0 */)		NULL
char* regex(const char *re, const char *subject, /* char*ret0 */ ...)		NULL

Function	header File: math.h	Return Value
int fpclassify(real-floating x)		- 1
int isnormal(real-floating x)		0
double gamma_r(double x, int *signgamp)		NaN
int signbit(real-floating x)		- 1

Function	Header File: signal.h	Return Value
int sighold(int sig)		- 1
int sigrelse(int sig)		- 1
int sigignore(int sig)		- 1
int sigaltstack(const stack_t *ss, stack_t oss)		- 1

Function	Header File: stropts.h	Return Value
int fattach(int fildes, const char *path)		- 1
int fdetach(const char *path)		- 1
int getmsg(int fildes, struct strbuf *ctptr, struct strbuf *dataptr, long *flagsp)		- 1
int getpmsg(int fildes, struct strbuf *ctptr, struct strbuf *dataptr, int *band, long *flagsp)		- 1
int isastream(int fildes)		- 1
int putmsg(int fildes, struct strbuf *ctptr, struct strbuf *dataptr, long flags)		- 1
int putpmsg(int fildes, struct strbuf *ctptr, struct strbuf *dataptr, int band, long flags)		- 1

Function	Header File: sys/mman.h	Retrun Value
int msync(void * addr, size_t len, int flags)		- 1
int mlock(void *addr, size_t len)		- 1
int munlock(void *addr, size_t len)		- 1
int shm_open(const char *name, int oflag, mode_t mode)		- 1

int shm_unlink(const char *name) – 1

Function	Header File: sys/procset.h	Return Value
int sigsend(idtype_t idtype, id_t id, int sig)		– 1
int sigsendset(procset_t *psp, int sig)		– 1

Function	Header File: sys/wait.h	Return Value
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options)		– 1

Function	Header File: poll.h	Return Value
int poll(struct pollfd *fds, unsigned int nfds, int timeout)		– 1

Function	Header File: unistd.h	Return Value
int fchroot(int fildes)		– 1
char *gettxt(const char *msgid, const char *dflt_str)		NULL
pid_t getsid(pid_t pid)		– 1
int mincore(caddr_t addr, size_t len, char *vec) unsigned int scale)		0

A.3 Solaris

The following functions are implemented with return error code only under the Solaros operating system.

Function	Header File: windows/winsock.h	Return Value
HANDLE WSAAPI WSAAsyncGetHostByAddr(HWND hwnd, unsigned int wMsg, const char FAR *addr, int len, int type, char FAR *buf, int buflen)		0
HANDLE WSAAPI WSAAsyncGetHostByName(HWND hwnd, unsigned int wMsg, const char FAR *name, char FAR *buf, int buflen)		0
HANDLE WSAAPI WSAAsyncGetProtoByName(HWND hwnd, unsigned int wMsg, const char FAR *name, char FAR *buf, int buflen)		0
HANDLE WSAAPI WSAAsyncGetProtoByNumber(HWND hwnd, unsigned int wMsg, int number, char FAR *buf, int buflen)		0
HANDLE WSAAPI WSAAsyncGetServByName(HWND hwnd, unsigned int wMsg, char FAR *buf, int buflen)		0
HANDLE WSAAPI WSAAsyncGetServByPort(HWND hwnd, unsigned int wMsg, int port, const char FAR *proto, char FAR *buf, int buflen)		0
int WSAAPI WSAAsyncSelect(SOCKET s, HWND hwnd, unsigned int wMsg, long lEvent)		– 1
int WSAAPI WSACancelAsyncRequest(HANDLE AsyncTaskHandle)		– 1
int WSAAPI WSACancelBlockingCall(void)		– 1
int WSAAPI WSACleanup(void)		0

void WSAAPI WSAGetLastError(void)	N/A
BOOL WSAAPI WSAIsblocking(void)	False
BOOL WSAAPI WSASetBlockingHook(FARPROC lpBlockFunc)	NULL
void WSAAPI WSASetLastError(int iError)	N/A
int WSAAPI WSAStartup(WORD wVersionRequested, LPWSADATA lpWSADATA)	0
int WSAAPI WSAUnhookBlockingHook(void)	- 1
int WSAAPI ioctlsocket(SOCKET s, long cmd, u_long *argp)	- 1

Function	Header File: math.h	Return Value
int fpclassify(<i>real-floating</i> x)		- 1
int isnormal(double x)		0
int signbit(double x)		0

Function	Header File: wchar.h	Return Value
wint_t btowc(int c)		WEOF
int fwide(FILE *stream, int c)		0
int mbsinit(const mbstate_t *ps)		0
size_t mbrlen(const char *str, size_t length, mbstate_t ps)		0
size_t mbrtowc(wchar_t *str1, const char *str2, size_t length, mbstate_t ps)		- 1
size_t mbsrtowcs(wchar_t *str1, const char **str2, size_t length, mbstate_t *ps)		- 1
size_t wctomb(char *str1, wchar_t s, mbstate_t *ap)		- 1
size_t wcsrtombs(char *str1, const char **str2, size_t length, mbstate_t *ps)		- 1
wchar_t *wcsstr(const wchar_t *str1, const wchar_t *str2)		NULL
int wctob(wint_t wc)		EOP
wchar_t wmemchr(const wchar_t *str1, wchar_t c, size_t count)		NULL
int wmemcmp(const wchar_t *str1, const wchar_t *str2, size_t count)		NULL
wchar_t wmemmove(wchar_t *str1, const wchar_t *str2, size_t count)		NULL
wchar_t wmemset(wchar_t *str1, wchar_t str2, size_t count)		NULL
wchar_t wmemcpy(wchar_t *str1, const wchar_t *str2, size_t count)		NULL

A.4 Windows

The following functions are implemented with return error code only under the Windows 95/98/NT operating systems.

Function	Header File: arpa/inet.h	Return Value
unsigned long inet_network(const char *cp)		- 1
struct in_addr inet_makeaddr(int net, int lna)		- 1
unsigned long inet_lnaof(struct in_addr in)		- 1
unsigned long inet_netof(struct in_addr in)		- 1

Function	Header File: libintl.h	Return Value
char *gettext(const char *msgid)		NULL

Chapter A: Appendix A: Functions Not Supported in Specific Platforms

char *dgettext(const char *domainname, const char *msgid)	NULL
char *textdomain(const char *domainname)	NULL
char *bindtextdomain(const char *domainname)	NULL

Function	Header File: aio.h	Return Value
int aio_read(struct aiocb *aiocbp)		- 1
int aio_write(struct aiocb *aiocbp)		- 1
int lio_listio(int mode, struct aiocb *list [], int nent, struct sigevent *sig)		- 1
int aio_error(const struct aiocb *aiocbp)		- 1
int aio_return(struct aiocb *aiocbp)		- 1
int aio_cancel(int fildes, struct aiocb *aiocbp)		- 1
int aio_suspend(const struct aiocb *aiocbp[], int nent, const struct timespec *timeout)		- 1
int aio_fsync(int op, struct aiocb *aiocbp)		- 1

Function	Header File: glob.h	Return Value
int glob(const char *pattern, int flags, int (*errfunc)(const char *epath, int eerrno) glob_t *pglob)		-1
void globfree(glob_t *pglob)		N/A

Function	Header File: grp.h	Return Value
int initgroups(const char *name, gid_t basegid)		- 1
void setgrent(void)		N/A
struct group *getgrent(void)		NULL
int getgrent_r(struct group *result, char *buffer, int buflen, FILE **grfp)		- 1

Function	Header File: libgen.h	Return Value
char* regcmp(const char *string1, /* char*string2 */ , ... /* (char*)0 */)		NULL
char* regex(const char *re, const char *subject, /* char*ret0 */ ...)		NULL

Function	Header File: math.h	Return Value
double erf(double x)		NaN
double erfc(double x)		NaN
double cbrt(double x)		NaN
double rint(double x)		NaN
double scalbn(double x, int n)		NaN
double remainder(float x, float x)		NaN
double ilogb(double x)		NaN
double gamma(double x)		NaN
double lgamma(double x)		NaN
double gamma_r(double x, int *signgamp)		NaN
double lgamma_r(double x, int *signgamp)		NaN
int fpclassify(real-floating x)		- 1

Chapter A: Appendix A: Functions Not Supported in Specific Platforms

int isnormal (<i>real-floating x</i>)	0
int signbit (<i>real-floating x</i>)	- 1

Function	Header File: netdb.h	Return Value
struct hostent *gethostent(void)		NULL
struct netent *getnetbyname(const char *name)		NULL
struct netent *getnetbyaddr(long net, int type)		NULL
struct netent *getnetent(void)		NULL
struct servent *getservent(void)		NULL
struct protoent *getprotoent(void)		NULL
int sethostent (int stayopen)		- 1
int endhostent (void)		- 1
int endnetent (void)		- 1
int setnetent (int stayopen)		- 1
int setservent (int stayopen)		- 1
int endservent (void)		- 1
int setprotoent (int stayopen)		- 1
int endprotoent (void)		- 1
int rcmd (char **ahost, unsigned short inport, char *luser, char *ruser, char *cmd, int *fd2p)		- 1
int rresvport (int *port)		- 1
int ruserok (char *rhost, int suser, char *ruser, char *luser)		- 1
int rexec (char **ahost, unsigned short inport, char *user, char *passwd, char *cmd, int *fd2p)		- 1
struct hostent *gethostbyname_r(const char *name, struct hostent *result, char *buffer, int *buflen, int *h_errnop)		NULL
struct hostent *gethostbyaddr_r(const char *addr, int length, int type, struct hostent *result, char *buffer, int buflen, int *h_errnop)		NULL
struct hostent *gethostent_r(struct hostent *result, char *buffer, int buflen, int *h_errnop)		NULL
struct servent *getservbyname_r(const char *name, const char *proto, struct servent *result, char *buffer, int *buflen)		NULL
struct servent *getservbyport_r(int port, const char *proto, struct servent *result, char *buffer, int buflen)		NULL
struct servent *getservent_r(struct hostent *result, char *buffer, int buflen)		NULL
struct netent *getnetbyname_r(const char *name, struct netent *result, char *buffer, int buflen)		NULL
struct netent *getnetbyaddr_r(long net, int type, struct netent *result, char *buffer, int buflen)		NULL
struct netent *getnetent_r(struct netent *result, char *buffer, int buflen)		NULL
struct protoent *getprotobyname_r(const char *name, struct protent *result, char *buffer, int buflen)		NULL
struct protoent *getprotobynumber_r(int proto, struct protent *result, char *buffer, int buflen)		NULL
struct protoent *getprotoent_r(struct protent *result, char *buffer, int buflen)		NULL

Function	Header File: poll.h	Return Value
int poll (struct pollfd *fds, unsigned int nfds, int timeout)		- 1

Function	Header File: pwd.h	Return Value
----------	--------------------	--------------

Chapter A: Appendix A: Functions Not Supported in Specific Platforms

struct passwd *getpwent()	NULL
struct passwd *getpwnam(const char *name)	NULL
struct passwd *getpwuid(uid_t uid)	NULL
struct passwd *putpwent(const struct passwd *p, FILE *f)	- 1

Function	Header File: regex.h	Return Value
char *regcomp(regex_t *preg, const char *pattern, int cflags)		- 1
int regexec(const regex_t *preg, const char *string, size_t nmatch, regmatch_t pmatch [])		- 1
regerror(int errcode, const regex_t *preg, char *errbuf, size_t eflags)		- 1
void regfree(regex_t *preg)		N/A

Function	Header File: signal.h	Return Value
int kill(pid_t pid, int sig)		- 1
int sigaction(int sig, const struct sigaction *act, struct sigaction *oact)		- 1
int sigemptyset(sigset_t *set)		- 1
int sigfillset(sigset_t *set)		- 1
int sigaddset(sigset_t *set, int signo)		- 1
int sigdelset(sigset_t *set, int signo)		- 1
int sigismember(const sigset_t *set, int signo)		- 1
int sigpending(sigset_t *set)		- 1
int sigprocmask(int how, const sigset_t *set, sigset_t *oset)		- 1
int sigsuspend(const sigset_t *set)		- 1
int gsignal(int sig)		- 1
int sighold(int sig)		- 1
int sigrelse(int sig)		- 1
int sigignore(int sig)		- 1
int sigpause(int sig)		- 1
int sigaltstack(const stack_t *ss, stack_t *oss)		- 1

Function	Header File: stdio.h	Return Value
FILE *fdopen(int fildes, const char *type)		NULL

Function	Header File: stdlib.h	Return Value
char * initState(unsigned int seed, char * state, size_t size)		NULL
long random()		0
char * setstate(const char state)		NULL
void srandom(unsigned int *seed)		N/A
int mkstemp(char * tempplate)		-1

Function	Header File: stropts.h	Return Value
int fattach(int fildes, const char *path)		- 1
int fdetach(const char *path)		- 1
int getmsg(int fildes, struct strbuf *cptr, struct strbuf *dataptr, long *flagsp)		- 1

Chapter A: Appendix A: Functions Not Supported in Specific Platforms

int getpmsg(int fildes, struct strbuf *ctptr, struct strbuf *dataptr, int *band, long *flagsp)	- 1
int isastream(int fildes)	- 1
int putmsg(int fildes, struct strbuf *ctptr, struct strbuf *dataptr, long flags)	- 1
int putpmsg(int fildes, struct strbuf *ctptr, struct strbuf *dataptr, int band, long flags)	- 1

Function	Header File: sys/stat.h	Return Value
int mknod(const char *path, mode_t mode, dev_t dev)		- 1
int fchmod(int fildes, mode_t mode)		- 1
int mkfifo(const char *path, mode_t mode)		- 1

Function	Header File: sys/acct.h	Return Value
int acct(const char *path)		- 1

Function	Header File: sys/fcntl.h	Return Value
int fcntl(int fildes, int cmd, .../* arg */)		- 1

Function	Header File: sys/msg.h	Return Value
int msgctl(int msqid, int cmd, struct msqid_ds *buf)		- 1
int msgget(key_t key, int msgflg)		- 1

Function	Header File: sys/mman.h	Return Value
void *mmap(void * addr, size_t len, int prot, int flags, int fildes, off_t off)		(void *) - 1
int mprotect(void *addr, size_t len, int port)		- 1
int munmap(void *addr, size_t len)		- 1

Function	Header File: sys/procset.h	Return Value
int sigsend(idtype_t idtype, id_t id, int sig)		- 1
int sigsendset(procset_t *psp, int sig)		- 1

Function	Header File: sys/resource.h	Return Value
int getpriority(int which, int who)		- 1
int setpriority(int which, int who, int prio)		- 1
int getrlimit(int resource, struct rlimit *rlp)		- 1
int setrlimit(int resource, const struct rlimit *rlp)		- 1

Chapter A: Appendix A: Functions Not Supported in Specific Platforms

Function	Header File: sys/socket.h	Return Value
int recvmsg(int <i>s</i> , struct msghdr * <i>msg</i> , int <i>flags</i>)		- 1
int sendmsg(int <i>s</i> , struct msghdr * <i>msg</i> , int <i>flags</i>)		- 1
int socketpair(int <i>domain</i> , int <i>type</i> , int <i>protocol</i> , int <i>sv</i>)		- 1

Function	Header File: sys/stat.h	Return Value
int mknod(const char * <i>path</i> , mode_t <i>mode</i> , dev_t <i>dev</i>)		- 1
int fchmod(int <i>fildev</i> , mode_t <i>mode</i>)		- 1
int mkfifo(const char * <i>path</i> , mode_t <i>mode</i>)		- 1

Function	Header File: sys/times.h	Return Value
clock_t times(struct tms * <i>buffer</i>)		- 1

Function	Header File: sys/uio.h	Return Value
int writev(int <i>fildev</i> , const struct iovec * <i>iov</i> , int <i>iovcnt</i>)		- 1

Function	Header File: sys/utsname.h	Return Value
int uname(struct utsname * <i>name</i>)		- 1

Function	Header File: sys/wait.h	Return Value
pid_t wait(int * <i>stat_loc</i>)		0
pid_t waitpid(pid_t <i>pid</i> , int * <i>stat_loc</i> , int <i>options</i>)		- 1

Function	Header File: syslog.h	Return Value
void syslog(int <i>priority</i> , const char * <i>message</i> , ...)		N/A
void openlog(const char * <i>ident</i> , int <i>logopt</i> , int <i>facility</i>)		N/A
void closelog(void)		N/A
int setlogmask(int <i>maskpri</i>)		- 1

Function	Header File: termios.h	Return Value
int cfgetispeed(struct termios * <i>termios_p</i>)		NULL
speed_t cfgetospeed(struct termios * <i>termios_p</i>)		NULL
int cfsetispeed(struct termios * <i>termios_p</i> , speed_t <i>speed</i>)		- 1
int cfsetospeed(struct termios * <i>termios_p</i> , speed_t <i>speed</i>)		- 1
int ltcdrain(int <i>fildev</i>)		- 1
int tcflow(int <i>fildev</i> , int <i>action</i>)		- 1

Chapter A: Appendix A: Functions Not Supported in Specific Platforms

int tcflush(int <i>fildev</i>, int <i>queue_selector</i>)	- 1
int tegetattr(int <i>fildev</i>, struct <i>termios</i> *<i>termios_p</i>)	- 1
int tcsendbreak(int <i>fildev</i>, int <i>duration</i>)	- 1
int tcsetattr(int <i>fildev</i>, int <i>optional_actions</i>, const struct <i>termios</i> *<i>termios_p</i>)	- 1

Function	Header File: time.h	Return Value
int ascftime(char *<i>s</i>, const char *<i>format</i>, const struct <i>tm</i> *<i>timeptr</i>)		0
int cftime(char *<i>s</i>, char *<i>format</i>, const time_t *<i>clock</i>)		0
char *ctime_r(const time_t *<i>clock</i>, char *<i>buf</i>, int <i>buflen</i>)		NULL
struct <i>tm</i> *localtime_r(const time_t *<i>clock</i>, struct <i>tm</i> *<i>res</i>)		NULL
struct <i>tm</i> *gmtime_r(const time_t *<i>clock</i>, struct <i>tm</i> *<i>res</i>)		NULL
char *asctime_r(const struct <i>tm</i> *<i>tm</i>, char *<i>buf</i>, int <i>buflen</i>)		NULL
int clock_settime(clockid_t <i>clock_id</i>, const struct <i>timespec</i> *<i>tp</i>)		- 1
int clock_gettime(clockid_t <i>clock_id</i>, const struct <i>timespec</i> *<i>tp</i>)		- 1
int clock_getres(clockid_t <i>clock_id</i>, const struct <i>timespec</i> *<i>res</i>)		- 1
int timer_create(clockid_t <i>clock_id</i>, struct <i>sigevent</i> *<i>evp</i>, timer_t *<i>timerid</i>)		- 1
int timer_delete(timer_t <i>timerid</i>)		- 1
int timer_settime(timer_t <i>timerid</i>, int <i>flags</i>, const struct <i>itimerspec</i> *<i>value</i>, struct <i>itimerspec</i> *<i>ovalue</i>)		- 1
int timer_gettime(timer_t <i>timerid</i>, struct <i>timespec</i> *<i>value</i>)		- 1
int timer_getoverrun(timer_t <i>timerid</i>)		- 1
int nanosleep(const struct <i>timespec</i> *<i>rqtp</i>, struct <i>timespec</i> *<i>rmtpt</i>)		- 1

Function	Header File:unistd.h	Return Value
int acct(const char *<i>path</i>)		- 1
unsigned alarm(unsigned int <i>sec</i>)		0
int chown(const char *<i>path</i>, uid_t <i>owner</i>, gid_t <i>group</i>)		- 1
int chroot(const char *<i>path</i>)		- 1
char *ctermid(char *<i>s</i>)		NULL
char *cuserid(char *<i>s</i>)		NULL
int fchdir(int <i>fildev</i>)		- 1
int fchown(int <i>fildev</i>, uid_t <i>owner</i>, gid_t <i>group</i>)		- 1
int fchroot(int <i>fildev</i>)		- 1
int fsync(int <i>fildev</i>)		- 1
int ftruncate(int <i>fildev</i>, off_t <i>length</i>)		- 1
uid_t getuid(void)		- 1
uid_t geteuid(void)		- 1
gid_t getgid(void)		- 1
gid_t getegid(void)		- 1
int getgroups(int <i>gidsetsize</i>, gid_t *<i>grouplist</i>)		- 1
gethostid(void)		0
char *getlogin(void)		NULL
char *getlogin_r(char *<i>name</i>, int <i>namelen</i>)		NULL
pid_t getpgrp(void)		NULL
pid_t getppid(void)		NULL
pid_t getpgid(pid_t <i>pid</i>)		- 1
char *gettxt(const char *<i>msgid</i>, const char *<i>dflt_str</i>)		NULL
pid_t getsid(pid_t <i>pid</i>)		NULL
pid_t setsid(void)		NULL
int link(const char *<i>existing</i>, const char *<i>newfile</i>)		- 1
int lchown(const char *<i>path</i>, uid_t <i>owner</i>, gid_t <i>group</i>)		- 1
int mincore(caddr_t <i>addr</i>, size_t <i>len</i>, char *<i>vec</i>)		- 1
int nice(int <i>incr</i>)		- 1

Chapter A: Appendix A: Functions Not Supported in Specific Platforms

long pathconf(const char *path, int name)	- 1
int pause(void)	- 1
void profil(unsigned short *buff, unsigned int bufsiz, unsigned int offset, unsigned int scale)	N/A
int ptrace(int request, pid_t pid, int addr, int data)	- 1
int readlink(const char *path, void *buf, size_t bufsiz)	- 1
void *sbrk(int incr)	(void *) - 1
int setuid(uid_t uid)	- 1
int setegid(gid_t egid)	- 1
int seteuid(uid_t euid)	- 1
int setgid(gid_t gid)	- 1
int setgroups(int ngroups, const gid_t *grouplist)	- 1
int setpgid(pid_t pid, pid_t pgid)	- 1
pid_t setpgrp(void)	- 1
int stime(const time_t *tp)	- 1
int symlink(const char *name1, const char *name2)	- 1
void sync(void)	N/A
long sysconf(int name)	- 1
pid_t tcgetpgrp(int fildes)	NULL
int tcsetpgrp(int fildes, pid_t pgid)	- 1
int truncate(const char *path, off_t length)	- 1
char ttyname(int fildes)	NULL
char ttyname_r(int fildes, char *buf, int len)	NULL
void vhangup(void)	N/A

Function	Header File: wchar.h	Return Value
int fwide(FILE *stream, int c)		0
int mbsinit(const mbstate_t *ps)		0
wchar_t wmemchr(const wchar_t *str1, wchar_t c, size_t count)		NULL
int wmemcmp(const wchar_t *str1, const wchar_t *str2, size_t count)		NULL
int wwidth(const wchar_t *c)		- 1
int wcswidth(const wchar_t *str1, size_t width)		- 1

Function	Header File: wctype.h	Return Value
wint_t towctrans(win_t wc, wctrans_t desc)		- 1
wctrans_t wctrans(const char *prop)		0
wctype_t wctype(const char *prop)		0

Index

`_IOFBF`, 878
`_IOLBF`, 878
`_IONBF`, 878
`__bool_true_false_are_defined`, 871
`_wfpopen()`, 1028, **1030**
`lldiv_t`, 933

`abort()`, 934, **936**
`abs()`, 934, **937**
`acct()`, 1126, 1128
`acos()`, 364, **367**
`acosh()`, 364, **368**
`aio.h`, ii
`aio_cancel()`, 1119, 1123
`aio_error()`, 1119, 1123
`aio_fsync()`, 1119, 1123
`aio_read()`, 1119, 1123
`aio_return()`, 1119, 1123
`aio_suspend()`, 1119, 1123
`aio_write()`, 1119, 1123
`alarm()`, 1128
`arc()`, 3, *see* CPlot
`arcSector()`, 3, *see* CPlot
`arcSegment()`, 3, *see* CPlot
`array.h`, ii
`arraycopy()`, 850, **851**
`arrow()`, 3, *see* CPlot
`asctime()`, 1118, 1120, 1128
`asctime()`, 1013, **1014**
`asctime_r()`, 1119, 1128
`asin()`, 364, **369**
`asinh()`, 364, **370**
`assert()`, 1, **2**
`assert.h`, ii
`atan()`, **371**
`atan2()`, 364, **372**
`atanh()`, **374**
`atexit()`, 934, **938**
`atoc()`, 934, **939**
`atof()`, 934, **940**
`atoi()`, 934, **941**
`atol()`, 934, **941**
`atoll()`, 934, **941**
`autoScale()`, 4, *see* CPlot
`axes()`, 4, *see* CPlot
`axis()`, 4, *see* CPlot
`axisRange()`, 4, *see* CPlot

`backgroundColor()`, 4, *see* CPlot
`balance()`, 436, **441**
`barGraph()`, 4, *see* CPlot
`barGraphCategory()`, 4, *see* CPlot
`barGraphColor()`, 4, *see* CPlot
`barGraphLegend()`, 4, *see* CPlot
`barGraphValue()`, 4, *see* CPlot
`barSizd()`, 4
`barSize()`, *see* CPlot
`bffdot()`, *see* CPlot
`bindtextdomain()`, 1118, 1123
`boo`, 871
`border()`, 4, *see* CPlot
`borderOffsets()`, 4, *see* CPlot
`boundingBoxOrigin()`, 4
`boxAndWhisker()`, 4, *see* CPlot
`boxBorder()`, 4, *see* CPlot
`boxFill()`, 4, *see* CPlot
`boxWidth()`, 4, *see* CPlot
`bsearch()`, 934, **942**
`btowc`, **1031**
`btowc()`, 1028, 1118, 1122
`BUFSIZ`, 878

`cabs()`, 310, **312**
`cacos()`, 310, **313**
`cacosh()`, 310, **314**
`calloc()`, 934, **944**
`carg()`, 310, **315**
`casin()`, 310, **316**
`casinh()`, 310, **317**
`catan()`, 310, **318**
`catanh()`, 310, **319**
`Cauchy`, 436, **784**
`cbirt()`, 364, **375**, 1123
`ccompanionmatrix()`, 436, **444**
`ccos()`, 310, **320**
`ccosh()`, 310, **321**
`cdeterminant()`, 436, **445**
`cdiagonal()`, 436, **447**
`cdiagonalmatrix()`, 436, **450**
`cdivisor()`, 436, **452**

- cdivisornum(), 436, **453**
- ceil(), 364, **376**
- cexp(), 310, **322**
- cfevalarray(), 436, **454**
- cfgetispeed(), 1127
- cfgetospeed(), 1127
- cfsetispeed(), 1127
- cfsetospeed(), 1127
- ctime(), 1118, 1119, 1128
- cfunm(), 436, **456**
- CH_CARRAY, 849
- CH_CARRAYPTR, 849
- CH_CARRAYPTRTYPE, 857
- CH_CARRAYTYPE, 857
- CH_CARRAYVLA, 849
- CH_CARRAYVLATYPE, 857
- CH_CHARRAY, 849
- CH_CHARRAYPTRTYPE, 849, 857
- CH_CHARRAYTYPE, 857
- CH_CHARRAYVLATYPE, 849, 857
- CH_UNDEFINETYPE, 857
- changeViewAngle(), 4, *see* CPlot
- CHAR_BIT, 352
- CHAR_MAX, 352
- CHAR_MIN, 352
- charpolycoef(), 436, **458**
- Chebyshev, 436
- ChebyshevVandermonde, **785**
- chinfo, 300
- chinfo(), 300, **302**
- choldecomp(), 436, **460**
- Chow, 436, **787**
- chown(), 1128
- chplot.h, ii
- chroot(), 1128
- chshell.h, ii
- ChType_t, 848
- cimag(), 310, **323**
- cinverse(), 436, **464**
- circle(), 4, *see* CPlot
- circleClosed(), 4, *see* CPlot
- circleOpen(), 4, *see* CPlot
- Circul, 436, **788**
- clearerr(), 877, **879**
- Clement, 436, **789**
- clinsolve(), 436, **466**
- clock(), 1013, **1016**
- clock_getres(), 1119, 1128
- clock_gettime(), 1119, 1128
- clock_settime(), 1119, 1128
- CLOCKS_PER_SEC, 1012
- clog(), 310, **324**
- closelog(), 1127
- cmean(), 436, **468**
- colorBox(), 4, *see* CPlot
- combination(), 436, **470**
- companionmatrix(), 436, **471**
- complex.h, ii
- complexsolve(), 436, **472**
- complexsolvePP(), 436, **478**
- complexsolvePR(), 436, **480**
- complexsolveRP(), 436, **483**
- complexsolveRR(), 436, **485**
- complexsolveRRz(), 436, **488**
- condnum(), 436, **490**
- conj(), 310, **325**
- constrain(), **492**
- contourLabel(), 4, *see* CPlot
- contourLevels(), 4, *see* CPlot
- contourMode(), 4, *see* CPlot
- conv(), 436, **493**
- conv2(), 436, **498**
- coordSystem(), 4, *see* CPlot
- copyright, i
- copysign(), 364, **377**
- corrcoef(), 436, **501**
- correlation2(), 436, **503**
- cos(), 364, **378**
- cosh(), 364, **379**
- covariance(), 436, **505**
- cpio.h, ii
- CPlot, **3**
 - ~CPlot, 3
 - arc(), 3, **10**
 - arcSector(), 3, **11**
 - arcSegment(), 3, **12**
 - arrow(), 3, **16**
 - autoScale(), 4, **20**
 - axes(), 4, **28**
 - axis(), 4, **21**
 - axisRange(), 4, **24**
 - backgroundColor(), 4, **30**
 - barGraph(), 4, **31**
 - barGraphCategory(), 4, **35**
 - barGraphColor(), 4, **36**
 - barGraphLegend(), 4, **36**
 - barGraphValue(), 4, **37**
 - barSize(), 4, **38**
 - bffdot(), **97**
 - border(), 4, **38**
 - borderOffsets(), 4, **41**
 - boundingBoxOrigin(), 4
 - boxAndWhisker(), 4, **42**
 - boxBorder(), 4, **44**
 - boxFill(), 4, **46**
 - boxWidth(), 4, **48**
 - changeViewAngle(), 4, **49**
 - circle(), 4, **51**

circleClosed(), 4, **54**
circleOpen(), 4, **55**
colorBox(), 4, **56**
contourLabel(), 4, **58**
contourLevels(), 4, **60**
contourMode(), 4, **62**
coordSystem(), 4, **65**
CPlot(), 3
data(), 4, **70**
data2D(), 4, **73**
data2DCurve(), 4, **77**
data3D(), 4, **78**
data3DCurve(), 4, **85**
data3DSurface(), 4, **87**
dataFile(), 4, **89**
dataSetNum(), 4, **93**
deleteData(), 4, **94**
deletePlots(), 4, **95**
dimension(), 4, **95**
displayTime(), 4, **96**
dot(), 4
ellipse(), 4, **98**
enhanceText(), 4, **100**
expr(), 4, **104**
exprPoint(), 4, **105**
exprxy(), 4, **107**
fillColor(), 4, **108**
fillCurve(), 4, **109**
fillOpacity(), 5, **113**
fillPattern(), 5, **114**
fillStyle(), 5, **116**
font(), 5, **119**
fontScale(), 5, **120**
func2D(), 5, **121**
func2DPoint(), 5, **122**
func3D(), 5, **124**
funcp2D(), 5, **126**
funcp3D(), 5, **127**
getGrid(), 5, **128**
getLabel(), 5, **129**
getOutputType(), 5, **130**
getSubplot(), 5, **131**
getTitle(), 5, **135**
grid(), 5, **136**
histogram(), 5, **139**
histogramColor(), 5, **141**
histogramValue(), 5, **142**
inequal(), 5, **142**
inequalAnd(), 5, **145**
inequalOffset(), 5, **149**
isUsed(), 5, **150**
label(), 5, **150**
legend(), 5, **152**
legendLocation(), 5, **155**
legendOption(), 5, **157**
line(), 5, **160**
line1D(), 5, **162**
line1DOffset(), 5, **164**
lineStyle(), 5, **164**
lineType(), 5, **166**
margins(), 5, **168**
mathCoord(), 5, **169**
noFill(), 5, **172**
noStrokeColor(), 5, **173**
numberLine(), 5, **173**
numberLineScatter(), 5, **175**
numberLineScattern(), 5, **176**
origin(), 5, **177**
outputType(), 5, **178**
plotting(), 5, **216**
plotType(), 5, **188**
point(), 5, **217**
pointStyle(), 5, **220**
pointType(), 5, **222**
polarPlot(), 5, **223**
polygon(), 5, **226**
quad(), 6, **229**
rectangle(), 6, **230**
rectanglexy(), 6, **233**
regularPolygon(), 6, **234**
removeHiddenLine(), 6, **236**
scaleType(), 6, **238**
scatter(), 6, **241**
scattern(), 6, **240**
showMesh(), 6, **243**
size(), 6, **245**
size3D(), 6, **246**
sizeOutput(), 6, **247**
sizeRatio(), 6, **248**
smooth(), 6, **250**
star(), 6, **252**
strokeColor(), 6, **253**
strokeWidth(), 6, **256**
subplot(), 6, **258**
text(), 6, **259**
textColor(), 6, **260**
tics(), 6, **261**
ticsDay(), 6, **262**
ticsDirection(), 6, **263**
ticsFormat(), 6, **265**
ticsLabel(), 6, **266**
ticsLevel(), 6, **268**
ticsLocation(), 6, **270**
ticsMirror(), 6, **272**
ticsMonth(), 6, **273**
ticsPosition(), 6, **274**
ticsRange(), 6, **275**
title(), 6, **276**

- triangle(), 6, **278**
- cpolyeval(), 436, **508**
- cpow(), 310, **326**
- cproduct(), 436, **510**
- cquadratic(), 437, **512**
- creal(), 310, **327**
- cross(), 436, **514**
- crypt.h, ii
- csin(), **328**
- csinh(), 310, **329**
- csqrt(), **330**
- csum(), 436, **515**
- ctan(), 310, **331**
- ctanh(), 310, **332**
- ctermid(), 1128
- ctime(), 1013, **1017**
- ctime_r(), 1119, 1128
- ctrace(), 436, **517**
- ctriangularmatrix(), 436, **518**
- ctype.h, ii
- cumprod(), 436, **521**
- cumsum(), 436, **523**
- curvefit(), 436, **525**
- cuserid(), 1128

- data(), 4, *see* CPlot
- data2D(), 4, *see* CPlot
- data2DCurve(), 4, *see* CPlot
- data3D(), 4, *see* CPlot
- data3DCurve(), 4, *see* CPlot
- data3DSurface(), 4, *see* CPlot
- dataFile(), 4, *see* CPlot
- dataSetNum(), 4, *see* CPlot
- DBL_DIG, 349
- DBL_EPSILON, 350
- DBL_MANT_DIG, 349
- DBL_MAX, 350
- DBL_MAX_10_EXP, 350
- DBL_MAX_EXP, 350
- DBL_MIN, 351
- DBL_MIN_10_EXP, 350
- DBL_MIN_EXP, 350
- deconv(), 436, **529**
- def2rad(), **380, 422**
- deg2rad(), 364
- deleteData(), 4, *see* CPlot
- deletePlots(), 4, *see* CPlot
- DenavitHartenberg, 436, **790**
- DenavitHartenberg2, 436, **793**
- derivative(), 436, **532**
- derivatives(), 436, **534**
- determinant(), 436, **537**
- dgettext(), 1118, 1123
- diagonal(), 436, **539**
- diagonalmatrix(), 436, **542**
- difference(), 436, **544**
- difftime(), 1013, **1018**
- dimension(), 4, *see* CPlot
- dirent.h, ii
- displayTime(), 4, *see* CPlot
- div(), 934, **945**
- div_t, 933
- divisor(), 436, **545**
- divisornum(), 436, **546**
- dlfcn.h, ii
- dot(), 4, 436, **547**
- Dramadah, 436, **794**

- EDOM, 348
- eigen(), 436, **548**
- eigensystem(), 436, **553**
- EILSEQ, 348
- ellipse(), 4, *see* CPlot
- endhostent(), 1124
- endnetent(), 1124
- endprotoent(), 1124
- endservent(), 1124
- enhanceText(), 4, *see* CPlot
- EOF, 878
- ERANGE, 348
- erf(), 364, **381, 1123**
- erfc(), 364, **382, 1123**
- errno.h, ii
- exit(), 934, **946**
- EXIT_FAILURE, 933
- EXIT_SUCCESS, 933
- exp(), 364, **383**
- exp2(), 364, **384**
- expfit(), 436, **556**
- expm(), 436, **554**
- expm1(), 364, **385**
- expr(), 4, *see* CPlot
- exprPoint(), 4, *see* CPlot
- exprxy(), 4, *see* CPlot

- fabs(), 364, **386**
- factor(), 436, **559**
- factorial(), 436, **560**
- factornum(), 436, **561**
- false, 871
- fat tach(), 1125
- fattach(), 1120
- fchdir(), 1128
- fchmod(), 1126, 1127
- fchown(), 1128
- fchroot(), 1121, 1128
- fclose(), 877, **880**
- fcntl(), 1126

fcntl.h, ii
fdetach(), 1120, 1125
fdim(), 364, **387**
fdopen(), 1125
fenv.h, ii
feof(), 877, **881**
ferror(), 877, **882**
fevalarray(), 436, **562**
fflush(), 877, **883**
fft(), 436, **564**
fgetc(), 877, **884**
fgetgrent(), 1117
fgetgrent_r(), 1117
fgetpos(), 877, **885**
fgets(), 877, **886**
fgetwc(), 1028, **1032**
fgetws(), 1028, **1033**
Fiedler, 436, **796**
FILE, 878
FILENAME_MAX, 878
fillColor(), 4, *see* CPlot
fillCurve(), 4, *see* CPlot
fillOpacity(), 5, *see* CPlot
fillPattern(), 5, *see* CPlot
fillStyle(), 5, *see* CPlot
filpud(), 436
filter(), 436, **569**
filter2(), 436
findvalue(), 436, **576**
fliplr(), 436, **578**
flipud(), **580**
float.h, ii
floor(), 364, **388**
FLT_DIG, 349
FLT_EPSILON, 350
FLT_MANT_DIG, 349
FLT_MAX, 350
FLT_MAX_10_EXP, 350
FLT_MAX_EXP, 350
FLT_MIN, 351
FLT_MIN_10_EXP, 350
FLT_MIN_EXP, 350
FLT_RADIX, 349
fma(), 364, **389**
fmax(), 364, **390**
fmin(), 364, **391**
fminimum(), 436, **582**
fminimums(), 436, **585**
fmod(), 364, **392**
font(), 5, *see* CPlot
fontScale(), 5, *see* CPlot
fopen(), 877, **887**
FOPEN_MAX, 878
fpclassify(), **393**, 1120, 1122, 1123
fplotxy(), 8, **280**
fplotxyz(), 8, **282**
fpos_t, 878
fprintf(), 877, **888**
fputc(), 877, **892**
fputs(), 877, **893**
fputwc(), 1028, **1034**
fputws(), 1028, **1036**
Frank, 436, **797**
fread(), 877, **894**
free(), 934, **947**
freopen(), 877, **895**
frexp(), 364, **394**
fscanf(), 877, **896**
fseek(), 877, **899**
fsetpos(), 877, **901**
fsolve(), 436, **588**
fsync(), 1128
ftell(), 877, **902**
ftruncate(), 1128
func2D(), 5, *see* CPlot
func2DPoint(), 5, *see* CPlot
func3D(), 5, *see* CPlot
funcp2D(), 5, *see* CPlot
funcp3D(), 5, *see* CPlot
funm(), 436, **591**
fwide(), 1028, **1037**, 1118, 1122, 1129
fwrite(), 877, **904**
fzero(), 436, **593**
fzero1(), 436, **595**
fzero2(), 436, **596**

gamma(), 1123
gamma_r(), 1117, 1120, 1123
gcd(), 436, **598**, **599**
Gear, 436, **798**
getc(), 877, **906**
getchar(), 877, **905**
getegid(), 1128
getenv(), 934, **948**
geteuid(), 1128
getgid(), 1128
getgrent(), 1123
getgrent_r(), 1123
getgrgid_r(), 1117
getGrid(), 5, *see* CPlot
getgrnam_r(), 1117
getgroups(), 1128
gethostbyaddr_r(), 1117, 1124
gethostbyname_r(), 1117, 1124
gethostent(), 1124
gethostent_r(), 1117, 1124
gethostid(), 1128
getLabel(), 5, *see* CPlot

getlogin(), 1128
getlogin_r(), 1128
getlogin_r(), 1117
getmsg(), 1120, 1125
getnetbyaddr(), 1124
getnetbyaddr_r(), 1117, 1124
getnetbyname(), 1124
getnetbyname_r(), 1117, 1124
getnetent(), 1124
getnetent_r(), 1117, 1124
getnum(), 436, **601**
getOutputType(), 5, *see* CPlot
getpgid(), 1128
getpgrp(), 1128
getpmsg(), 1120, 1126
getppid(), 1128
getpriority(), 1126
getprotobyname_r(), 1117, 1124
getprotobyname_r(), 1117, 1124
getprotoent(), 1124
getprotoent_r(), 1117, 1124
getpwent(), 1125
getpwnam(), 1125
getpwuid(), 1125
getrlimit(), 1126
gets(), 877, **907**
getservbyname_r(), 1117, 1124
getservbyport_r(), 1117, 1124
getservent(), 1124
getservent_r(), 1117, 1124
getsid(), 1121, 1128
getSubplot(), 5, *see* CPlot
gettext(), 1118, 1122
getTitle(), 5, *see* CPlot
gettxt(), 1121, 1128
getuid(), 1128
getwc(), 1028, **1038**
getwchar(), 1028, **1039**
glob(), 1123
glob.h, ii
globfree(), 1123
gmtime(), 1013, **1019**
gmtime_r(), 1119, 1128
grid(), 5, *see* CPlot
grp.h, ii
gsignal(), 1125

Hadamard, 436, **799**
Hankel, 436, **801**
hessdecomp(), 436, **602**
Hilbert, 436, **802**
histogram(), 5, *see* CPlot, 436, **605**
histogramColor(), 5, *see* CPlot
histogramValue(), 5, *see* CPlot

householdermatrix(), 436, **608**
HUGE_VAL, 365
hypot(), 364, **395**

identitymatrix(), 436, **610**
ifft(), 436, **611**
ilogb(), 364, **396**, 1123
inequal(), 5, *see* CPlot
inequalAnd(), 5, *see* CPlot
inequalOffset(), 5, *see* CPlot
inet.h, ii
inet_lnaof(), 1122
inet_makeaddr(), 1122
inet_netof(), 1122
inet_network(), 1122
INFINITY, 365
initgroups(), 1123
initstate(), 1125
INT_MAX, 353
INT_MIN, 353
integral1(), 436, **613**
integral2(), 436, **618**
integral3(), 436, **620**
integralp1(), 436, **616**
integration2(), 436, **622**
integration3(), 436, **624**
interp1(), 436, **627**
interp2(), 436, **629**
inttypes.h, ii
inverse(), 436, **632**
InverseHilbert, 436, **803**
ioctllock(), 1116, 1119, 1122
iostream.h, ii
isalnum(), 334, **335**
isalpha(), 334, **336**
isastream(), 1120, 1126
iscnan(), 310, **333**
iscntrl(), 334, **337**
iscnum(), 934, **949**
isdigit(), 334, **338**
isenv(), 934, **950**
isfinite(), 365, **397**
isgraph(), 334, **339**
isgreater(), 365, **398**
isgreaterequal(), 365, **399**
isinf(), 365, **400**
iskey, 300
iskey(), 300, **304**
isless(), 365, **401**
islessequal(), 365, **402**
islessgreater(), 365, **403**
islower(), 334, **340**
isnan(), 365, **404**
isnormal(), 365, **405**, 1117, 1120, 1122, 1124

isnum(), 934, **951**
iso646.h, ii
isprime(), 436, **634**
isprint(), 334, **341**
ispunct(), 334, **342**
isspace(), 334, **343**
isstudent, 300
isstudent(), 300, **306**
isunordered(), 365, **406**
isupper(), 334, **344**
isUsed(), 5, *see* CPlot
isvar, 300
isvar(), 300, **307**
iswalnum(), 1096, **1098**
iswalphalpha(), 1096, **1099**
iswcntrl(), 1096, **1100**
iswctype(), 1096, **1101**
iswdigit(), 1096, **1103**
iswgraph(), 1096, **1104**
iswlower(), 1096, **1105**
iswnum(), 934, **952**
iswprint(), 1096, **1106**
iswpunct(), 1096, **1107**
iswspace(), 1096, **1108**
iswupper(), 1096, **1109**
iswxdigit(), 1096, **1110**
isxdigit(), 334, **345**

jump_buf, 837

kill(), 1125

L_tmpnam, 878
label(), 5, *see* CPlot
labs, 934
labs(), **937**
LC_ALL, 355
LC_COLLATE, 355
LC_CTYPE, 355
LC_MONETARY, 355
LC_NUMERIC, 355
LC_TIME, 355
lchown(), 1128
lcm(), 436, **635**
lcmarray(), **636**
lconv, 355
LDBL_DIG, 349
LDBL_EPSILON, 350
LDBL_MANT_DIG, 349
LDBL_MAX, 350
LDBL_MAX_10_EXP, 350
LDBL_MAX_EXP, 350
LDBL_MIN, 351
LDBL_MIN_10_EXP, 350
LDBL_MIN_EXP, 350

ldexp(), 364, **407**
ldiv_t, 933
legend(), 5, *see* CPlot
legendLocation(), 5, *see* CPlot
legendOption(), 5, *see* CPlot
lgamma(), 364, **408**, 1123
lgamma_r(), 1123
libintl.h, ii
limits.h, ii
lindata(), 436, **637**
line(), 5, *see* CPlot
line1D(), 5, *see* CPlot
line1DOffset(), 5, *see* CPlot
linearcorrcoef(), 436, **639**
linearfit(), 436, **640**
lineStyle(), 5, *see* CPlot
lineType(), 5, *see* CPlot
link(), 1128
linrange(), 436, **643**
linsolve(), 436, **644**
linsolvesi(), 436, **646**
linsolvestd(), 436, **648**
linspace(), 436, **650**
lio_listio(), 1119, 1123
llabs(), 934, **937**
LLONG_MAX, 353
LLONG_MIN, 353
llsqcovsolve(), 436, **653**
llsqnonnegsolve(), 436, **655**
llsqsolve(), 436, **657**
local_timer(), 1119
locale.h, ii
localeconv(), 355, **356**
localtime(), 1013, **1020**
localtime_r(), 1128
log(), 364, **409**
log10(), 364, **410**
log1p(), 364, **411**
log2(), 364, **412**
logb(), 364, **413**
logdata(), 436, **662**
logm(), 436, **660**
logrange(), 436, **663**
logspace(), 436, **664**
LONG_MAX, 353
LONG_MIN, 353
longjmp(), 837, **841**
lrint(), 364, **414**
lround(), 364, **415**
ludcomp(), 436, **666**

Magic, 436, **803**
malloc(), 934, **953**
malloc.h, ii

- margins(), 5, *see* CPlot
- math.h, ii
- mathCoord(), 5, *see* CPlot
- maxloc(), 436, **670**
- maxv(), 436, **671**
- MB_CUR_MAX, 933
- MB_LEN_MAX, 352
- mblen(), 934, **954**
- mbrlen(), 1028, **1040**, 1118, 1122
- mbrtow(), 1122
- mbrtowc(), 1028, **1041**, 1118
- mbsinit(), 1028, **1043**, 1118, 1122, 1129
- mbsrtowcs(), 1028, **1044**, 1118, 1122
- mbstate_t, 1027
- mbstowcs(), 934, **955**
- mbtowc(), 934, **957**
- mean(), 436, **672**
- median(), 436, **675**
- memchr(), 973, **975**
- memcmp(), 973, **976**
- memcpy(), 973, **977**
- memmove(), 973, **978**
- memset(), 973, **979**
- mincore(), 1121, 1128
- minloc(), 436, **677**
- minv(), 436, **678**
- mkfifo(), 1126, 1127
- mknod(), 1126, 1127
- mkstemp(), 1125
- mktime(), 1013, **1021**
- mlock(), 1120
- mmap(), 1126
- mode(), 436, **679**
- modf(), 364, **416**
- mprotect(), 1126
- mq_receive(), 1118
- mqueue.h, ii
- msgctl(), 1126
- msgget(), 1126
- msync(), 1120
- munlock(), 1120
- munmap(), 1126

- NAN, 365
- nan(), 364, **417**
- nanosleep(), 1120, 1128
- nearbyint(), 364, **418**
- netconfig.h, ii
- netdb.h, ii
- netdir.h, ii
- netinet/in.h, ii
- new.h, ii
- nextafter(), 365, **419**
- nexttoward(), 365, **420**

- nice(), 1128
- noFill(), 5, *see* CPlot
- norm(), 436, **681**
- noStrokeColor(), 5, *see* CPlot
- nullspace(), 436, **685**
- numberLine(), 5, *see* CPlot
- numberLineScatter(), 5, *see* CPlot
- numberLineScatterm(), 5, *see* CPlot
- numeric.h, ii

- oderk(), 436, **688**
- oderungekutta(), 436, **697**
- odesolve(), 436, **699**
- offsetof(), 872
- openlog(), 1127
- origin(), 5, *see* CPlot
- orthonormalbase(), 436, **701**
- outputType(), 5, *see* CPlot

- Pascal, 436, **805**
- pathconf(), 1129
- pause(), 1129
- perror(), 877, **908**
- pinverse(), 436, **704**
- PLOT_ANGLE_DEG, 7, 66, 223
- PLOT_ANGLE_RAD, 7, 66, 223
- PLOT_AXIS_X, 7, 20, 22, 24, 28, 129, 151, 239, 261, 262, 265, 267, 271–273, 275, 276
- PLOT_AXIS_X2, 7, 20, 22, 24, 28, 129, 151, 239, 261, 262, 265, 267, 271–273, 275, 276
- PLOT_AXIS_XY, 7, 20, 22, 24, 28, 151, 239, 261, 262, 265, 267, 271, 272, 274–276
- PLOT_AXIS_XYZ, 7, 20, 24, 28, 151, 239, 261, 262, 265, 267, 272, 274–276
- PLOT_AXIS_Y, 7, 20, 22, 24, 28, 129, 151, 239, 261, 262, 265, 267, 271–273, 275, 276
- PLOT_AXIS_Y2, 7, 20, 22, 24, 28, 129, 151, 239, 261, 262, 265, 267, 271–273, 275, 276
- PLOT_AXIS_Z, 7, 20, 24, 28, 129, 151, 239, 261, 262, 265, 267, 272, 273, 275, 276
- PLOT_BORDER_ALL, 7
- PLOT_BORDER_BOTTOM, 7
- PLOT_BORDER_LEFT, 7
- PLOT_BORDER_RIGHT, 7
- PLOT_BORDER_TOP, 7
- PLOT_BOXFILL_EMPTY, 7, 47
- PLOT_BOXFILL_PATTERN, 7, 47
- PLOT_BOXFILL_SOLID, 7, 47
- PLOT_BOXFILL_TRANSPARENT_PATTERN, 7, 47
- PLOT_BOXFILL_TRANSPARENT_SOLID, 7
- PLOT_BOXFILL_TRANSPARENT_SOLID, 47
- PLOT_CONTOUR_BASE, 7, 63
- PLOT_CONTOUR_SURFACE, 7, 63
- PLOT_COORD_CARTESIAN, 7, 66

PLOT_COORD_CYLINDRICAL, 7, 66
 PLOT_COORD_SPHERICAL, 7, 66
 PLOT_FILLSTYLE_EMPTY, 7, 116
 PLOT_FILLSTYLE_PATTERN, 7, 117
 PLOT_FILLSTYLE_SOLID, 7, 116
 PLOT_FILLSTYLE_TRANSPARENT_PATTERN, 7, 117
 PLOT_FILLSTYLE_TRANSPARENT_SOLID, 7
 PLOT_FILLSTYLE_TRANSPARENT_SOLID, 117
 PLOT_OFF, 7, 21, 22, 39, 57, 58, 136, 236, 243, 261, 272
 PLOT_ON, 7, 21, 22, 39, 56, 58, 136, 236, 243, 261, 272
 PLOT_OUTPUTTYPE_DISPLAY, 7, 130, 178
 PLOT_OUTPUTTYPE_FILE, 7, 130, 178
 PLOT_OUTPUTTYPE_STREAM, 7, 130, 178
 PLOT_PLOTTYPE_BOXERRORBARS, 7, 44, 47, 48, 117, 190
 PLOT_PLOTTYPE_BOXES, 7, 44, 47, 48, 117, 190, 216
 PLOT_PLOTTYPE_BOXXYERRORBARS, 7, 44, 47, 48, 117, 191
 PLOT_PLOTTYPE_CANDLESTICKS, 7, 44, 47, 48, 117, 191, 208
 PLOT_PLOTTYPE_DOTS, 7, 190
 PLOT_PLOTTYPE_FILLED_CURVES, 7, 44, 47, 117, 190, 212, 213, 215
 PLOT_PLOTTYPE_FINANCEBARS, 7, 30, 191, 209
 PLOT_PLOTTYPE_FSTEPS, 7, 190
 PLOT_PLOTTYPE_HISTEPS, 7, 190
 PLOT_PLOTTYPE_IMPULSES, 8, 189
 PLOT_PLOTTYPE_LINES, 8, 63, 189
 PLOT_PLOTTYPE_LINESPOINTS, 8, 190
 PLOT_PLOTTYPE_POINTS, 8, 190
 PLOT_PLOTTYPE_STEPS, 8, 190
 PLOT_PLOTTYPE_SURFACES, 8, 63, 190
 PLOT_PLOTTYPE_VECTORS, 8, 208
 PLOT_PLOTTYPE_Vectors, 190, 207
 PLOT_PLOTTYPE_XERRORBARS, 8, 38, 192
 PLOT_PLOTTYPE_XERRORLINES, 8, 192
 PLOT_PLOTTYPE_XYERRORBARS, 8, 38, 192
 PLOT_PLOTTYPE_XYERRORLINES, 8, 192
 PLOT_PLOTTYPE_YERRORBARS, 8, 38, 192, 211
 PLOT_PLOTTYPE_YERRORLINES, 8, 192
 PLOT_SCALETYPETYPE_LINEAR, 8, 239
 PLOT_SCALETYPETYPE_LOG, 8, 239
 PLOT_TEXT_CENTER, 8, 259
 PLOT_TEXT_LEFT, 8, 259
 PLOT_TEXT_RIGHT, 8, 259
 PLOT_TICS_IN, 8, 264
 PLOT_TICS_OUT, 8, 264
 plotting(), 5, *see* CPlot
 plotType(), 5, *see* CPlot
 plotxy(), 8, **284**
 plotxyf(), 8, **290**
 plotxyz(), 8, **292**
 plotxyzf(), 8, **297**
 png, 184
 pngcairo, 184
 point(), 5, *see* CPlot
 pointStyle(), 5, *see* CPlot
 pointType(), 5, *see* CPlot
 polarPlot(), 5, *see* CPlot
 poll(), 1121, 1124
 poll.h, ii
 polycoef(), **709**
 polyder(), 436, **712**
 polyder2(), 436, **714**
 polyeval(), 436, **717**
 polyevalarray(), 436, **719**
 polyevalm(), 436, **721**
 polyfit(), 436, **723**
 polygon(), 5, *see* CPlot
 polygonperi(), 436, **727**
 pow(), 365, **421**
 printf(), 877, **910**
 product(), 436, **728**
 profil(), 1129
 pthread.h, ii
 ptrace(), 1129
 ptrdiff_t, 872
 putc(), 877, **912**
 putchar(), 877, **911**
 putmsg(), 1120, 1126
 putpmsg(), 1120, 1126
 putpwent(), 1125
 puts(), 877, **913**
 putwc(), 1028, **1046**
 putwchar(), 1028, **1047**
 pwd.h, ii
 qrdecomp(), 436, **730**
 qrdelete(), 436, **736**
 qrinsert(), 436, **740**
 qsort(), 934, **959**
 quad(), 6, *see* CPlot
 quadratic(), 439, **743**
 quadraticfit(), 436, **745**
 quartiles(), 436, 439, **748**
 rad2deg(), 365
 raise(), **846**
 rand(), 934, **961**
 RAND_MAX, 933
 randcolor(), 436, **750**
 randdouble(), 436, **751**
 randint(), 436, **752**
 randintfreq(), 436, **753**
 random(), 1125
 rank(), 436, **754**
 rcmd(), 1124
 rcondnum(), 436, **756**

re_comp.h, ii
 readline.h, ii
 readlink(), 1129
 realloc(), 934, **962**
 rectangle(), 6, *see* CPlot
 rectanglexy(), 6, *see* CPlot
 recvmsg(), 1127
 regcmp(), 1120, 1123
 regcomp(), 1125
 regerror(), 1125
 regex(), 1120, 1123
 regex.h, ii
 regexec(), 1125
 regfree(), 1125
 regularPolygon(), 6, *see* CPlot
 regularpolygon(), 436, **758**
 remainder(), 365, **423**, 1123
 removenv(), 934, **964**
 remove(), 877, **914**
 removeHiddenLine(), 6, *see* CPlot
 remquo(), 365, **424**
 rename(), 877, **915**
 residue(), 436, **760**
 rewind(), 877, **917**
 rexec(), 1124
 rgb2hex(), 436
 rgbTwohex(), **765**
 rint(), **425**
 rint(), 365, 1123
 roots(), 436, **766**
 Rosser, 436, **806**
 rot90(), 436, **768**
 rotatepoint(), 436, **770**
 rotatepolygon(), 436, **771**
 rotmatrix(), 436, **773**
 round(), 365, **426**
 rresvport(), 1124
 rsf2csf(), 436, **774**
 ruserok(), 1124

 sbrk(), 1129
 scalbn(), 365, 1117, 1123
 scalbn() & scalbln(), **427**
 scaleType(), 6, *see* CPlot
 scanf(), 877, **918**
 scatter(), 6, *see* CPlot
 scattern(), 6, *see* CPlot
 SCHAR_MAX, 352
 SCHAR_MIN, 352
 sched.h, ii
 schurdecomp(), 436, **777**
 SEEK_CUR, 878
 SEEK_END, 878
 SEEK_SET, 878

 semaphore.h, ii
 sendmsg(), 1127
 setbuf(), 877, **920**
 setegid(), 1117, 1129
 seteuid(), 1117, 1129
 setgid(), 1129
 setgrent(), 1123
 setgroups(), 1129
 sethostent(), 1124
 setjmp(), 837, **839**
 setjmp.h, ii
 setlocale(), 355, **361**
 setlogmask(), 1127
 setnetent(), 1124
 setpgid(), 1129
 setpgrp(), 1129
 setpriority(), 1126
 setprotoent(), 1124
 setrlimit(), 1126
 setservent(), 1124
 setsid(), 1128
 setstate(), 1125
 setuid(), 1129
 setvbuf(), 877, **921**
 shmopen(), 1120
 shmunlink(), 1121
 showMesh(), 6, *see* CPlot
 SHRT_MAX, 352
 SHRT_MIN, 352
 sig_atomic_t, 843
 SIG_DFL, 843
 SIG_ERR, 843
 SIG_IGN, 843
 sigaction(), 1125
 sigaddset(), 1125
 sigaltstack(), 1120, 1125
 sigdelset(), 1125
 sigemptyset(), 1125
 sigfillset(), 1125
 sighold(), 1120, 1125
 sigignore(), 1120, 1125
 sigismember(), 1125
 sign(), 436, **780**
 signal(), 843, **844**
 signal.h, ii
 signbit(), 365, **428**, 1117, 1120, 1122, 1124
 sigpause(), 1125
 sigpending(), 1125
 sigprocmask(), 1125
 sigrelse(), 1120, 1125
 sigsend(), 1121, 1126
 sigsendset(), 1118, 1121, 1126
 sigsuspend(), 1125
 sin(), 365, **429**

sinh(), 365, **430**
size(), 6, *see* CPlot
size3D(), 6, *see* CPlot
size_t, 872, 878
sizeofelement, 300
sizeofelement(), 300, **308**
sizeOutput(), 6, *see* CPlot
sizeRatio(), 6, *see* CPlot
smooth(), 6, *see* CPlot
snprintf(), 877, **922**
socketpair(), 1127
sort(), 436, **781**
specialmatrix(), 436, **784**
sprintf(), 877, **923**
sqrt(), 365, **431**
sqrtm(), 436, **811**
srand(), 934, **965**
srandom(), 1125
sscanf(), 877, **924**
star(), 6, *see* CPlot
std(), 436, **813**
stdarg.h, ii
stdbool.h, ii
stddef.h, ii
stderr, 878
stdio.h, ii
stdlib.h, ii
stdn, 878
stdout, 878
stime(), 1129
str2ascii(), **980**
str2mat(), **981**
stradd(), 973, **983**
strcasecmp, 973
strcasecmp(), **984**
strcat(), 973, **985**
strchr(), 973, **986**
strcmp(), 973, **987**
strcoll(), 973, **988**
strconcat(), 973, **989**
strcpy(), 973, **990**
strcspn(), 973, **991**
strdup(), 973, **992**
strerror(), 973, **993**
strftime(), 1013, **1022**
strgetc(), 973, **994**
string.h, ii
strjoin(), 973, **995**
strlen(), 973, **996**
strncasecmp(), 973, **997**
strncat(), 973, **998**
strncmp(), 973, **999**
strncpy(), 973, **1000**
strokeColor(), 6, *see* CPlot
strokeWidth(), 6, *see* CPlot
stropts.h, ii
strpbrk(), 973, **1001**
strputc(), 973, **1002**
strrchr(), 974, **1003**
strrep(), 974, **1004**
strspn(), 974, **1005**
strstr(), 974, **1006**
strtoascii(), 973
strtod(), 934, **966**
strtof(), 934, **966**
strtok(), 974, **1009**
strtok_r, 974, **1007**
strtol(), 934, **968**
strtold(), 934, **966**
strtoll(), 934, **968**
strtomat(), 973
strtoul(), 934, **968**
strtoull(), 934, **968**
strxfrm(), 974, **1011**
subplot(), 6, *see* CPlot
sum(), 436, **815**
svd(), 436, **817**
symlink(), 1129
sync(), 1129
sysconf(), 1129
syslog(), 1127
syslog.h, ii
system(), 934, **970**

tan(), 365, **432**
tanh(), 365, **433**
tar.h, ii
tcdrain(), 1127
tcflow(), 1127
tcflush(), 1128
tcgetattr(), 1128
tcgetpgrp(), 1129
tcsendbreak(), 1128
tcsetattr(), 1128
tcsetpgrp(), 1129
termios.h, ii
text(), 6, *see* CPlot
textColor(), 6, *see* CPlot
textdomain(), 1118, 1123
tgamma(), 365, **434**
tgmath.h, ii
tics(), 6, *see* CPlot
ticsDay(), 6, *see* CPlot
ticsDirection(), 6, *see* CPlot
ticsFormat(), 6, *see* CPlot
ticsLabel(), 6, *see* CPlot
ticsLevel(), 6, *see* CPlot
ticsLocation(), 6, *see* CPlot

ticsMirror(), 6, *see* CPlot
ticsMonth(), 6, *see* CPlot
ticsPosition(), 6, *see* CPlot
ticsRange(), 6, *see* CPlot
time(), 1013, **1026**
time.h, ii
timer_create(), 1120, 1128
timer_delete(), 1120, 1128
timer_getoverrun(), 1120, 1128
timer_gettime(), 1120, 1128
timer_settime(), 1120, 1128
times(), 1127
title(), 6, *see* CPlot
tiuser.h, ii
tm, 1012, 1027
TMP_MAX, 878
tmpfile(), 877, **925**
tmpnam(), 877, **926**
Toeplitz, 436, **807**
tolower(), 334, **346**
toupper(), 334, **347**
towctrans(), 1096, **1111**, 1118, 1129
towlower(), 1096, **1112**
toupper(), 1096, **1113**
trace(), 436, **824**
triangle(), 6, *see* CPlot
triangularmatrix(), 436, **826**
true, 871
trunc(), 365, **435**
truncate(), 1129
ttyname(), 1129
ttyname_r(), 1117, 1129
typographical conventions, ii

UCHAR_MAX, 352
UINT_MAX, 353
ULLONG_MAX, 353
ULONG_MAX, 353
uname(), 1127
ungetc(), 877, **927**
ungetwc(), 1028, **1048**
unistd.h, ii
unwrap(), 436, **829**
unwrapdeg(), 436, **831**
urand(), 436, **832**
USHRT_MAX, 352
utime.h, ii

va_arg(), 847, **853**
va_arraydim(), 847, **854**
va_arrayextent(), 847, **855**
va_arraynum(), 847, **856**
va_arraytype(), 847, **857**
va_copy(), 847, **859**
va_count(), 848, **861**
va_datatype(), 848, **862**
va_dim(), 848, **863**
va_elementtype(), 848, **864**
va_end(), 848, **865**
va_extent(), 848, **866**
va_list(), 848
VA_NOARG, 847
va_start(), 848, **867**
va_tagname(), 848, **869**
vandermatrix(), 436
Vandermonde, 436, **809**
vertex(), 440, **833**
vfprintf(), 877, **929**
vhangup(), 1129
vprintf(), 877, **930**
vsprintf(), 878, **931**
vsprintf(), 878, **932**

wait(), 1127
wait.h, ii
waitid(), 1121
waitpid(), 1127
wchar.h, ii
WCHAR_MAX, 1027
WCHAR_MIN, 1027
wchar_t, 872
wrtomb(), 1028, **1050**, 1118, 1122
wscat(), 1028, **1052**
wchr(), 1028, **1053**
wscmp(), 1028, **1055**
wscoll(), 1028, **1057**
wscpy(), 1028, **1059**
wscspn(), 1028, **1060**
wcsftime(), 1028, **1061**
wcslen(), 1028, **1063**
wscncat(), 1028, **1064**
wscncmp(), 1028, **1065**
wscncpy(), 1028, **1067**
wscpbrk(), 1028, **1068**
wscrchr(), 1028, **1070**
wscrombs(), 1028, **1072**, 1118, 1122
wcsspn(), 1028, **1074**
wcsstr(), 1028, **1075**, 1118, 1122
wcstod(), 1028, **1077**
wcstok(), 1028, **1079**
wcstol(), 1028, **1081**
wcstombs(), 934, **971**
wcsswidth(), 1129
wcsxfrm(), 1028, **1083**
wctob(), 1028, **1084**, 1118, 1122
wctomb(), 934, **972**
wctrans(), 1096, **1114**, 1118, 1129
wctrans_t, 1095

wctype(), 1096, **1115**, 1129
wctype.h, ii
wctype_t, 1095
wcwidth(), 1129
WEOF, 1027, 1095
Wilkinson, 436, **809**
wint_t, 1027, 1095
wmemchr(), 1028, **1085**, 1118, 1122, 1129
wmemcmp(), 1028, **1087**, 1118, 1122, 1129
wmemcpy(), 1028, **1089**, 1118, 1122
wmemmove(), 1028, **1091**, 1118, 1122
wmemset(), 1028, **1093**, 1118, 1122
writev(), 1127
WSAAsyncGetHostByAddr(), 1116, 1119, 1121
WSAAsyncGetHostByName(), 1116, 1119, 1121
WSAAsyncGetProtoByName(), 1116, 1119, 1121
WSAAsyncGetProtoByNumber(), 1116, 1119, 1121
WSAAsyncGetServByName(), 1116, 1119, 1121
WSAAsyncGetServByPort(), 1116, 1119, 1121
WSAAsyncSelect(), 1116, 1119, 1121
WSACancelAsyncRequest(), 1116, 1119, 1121
WSACancelBlockingCall(), 1116, 1119, 1121
WSACleanup(), 1116, 1119, 1121
WSAGetLastError(), 1116, 1119, 1122
WSAIsblocking(), 1116, 1119, 1122
WSASetBlockingHook(), 1116, 1119, 1122
WSASetLastError(), 1116, 1119, 1122
WSAStartup(), 1116, 1119, 1122
WSAUnhookBlockingHook(), 1116, 1119, 1122

xcorr(), 436, **834**