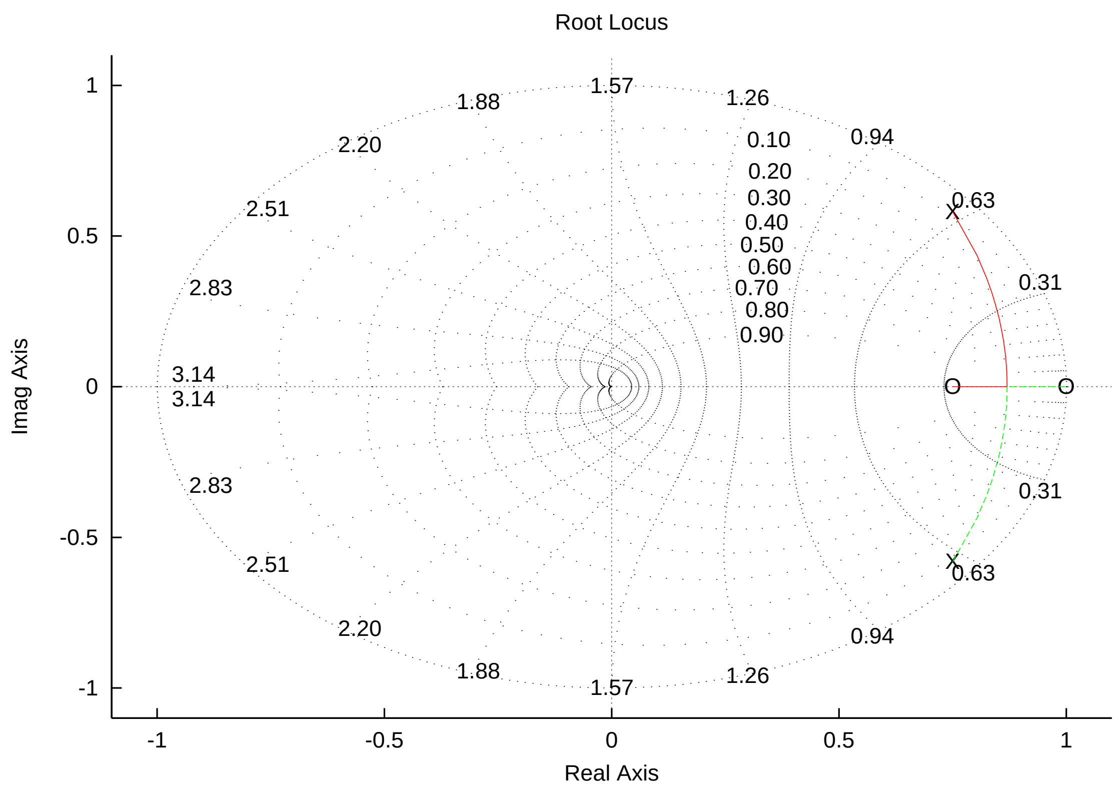
SoftIntegration®

Ch Control System Toolkit

Version 2.5

User's 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

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->Ch6.0->Ch** indicates that you first select **Start**. Then select submenu **Programs** by pointing the mouse on **Programs**, followed by **Ch6.0**. 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 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 Ch 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.

Table of Contents

Chapter 1

Getting Started with Ch Control System Toolkit

Note: The source code for all examples described in this document are available in CHHOME/toolkit/demos/control. CHHOME is the directory where Ch is installed. It is recommended that you try these examples while reading this document.

1.1 Introduction

The Ch language environment is a superset of C interpreter. It allows software developers to use one language, anywhere and everywhere, for any programming and numerical computing tasks. More information about Ch can be found at http://www.softintegration.com. As a component of the Ch language environment, Ch Control System Toolkit provides a class named **CControl** with member functions for design, analysis, and modeling of control systems.

It is well known that control systems can be modeled in transfer functions, zero-pole-gain, and state-space form. With Ch Control System Toolkit, either continuous-time or discrete-time linear time invariant (LTI) control systems can be created in these forms. Once a system model is created, it can be manipulated, converted, analyzed and even be used to design another system in both time and frequency domains. Ch Control System Toolkit supports most classical and modern control techniques. In this document, the techniques for control system modeling, design and analysis using Ch Control System Toolkit will be described. Details of member functions of class **CControl** will be presented in Appendix A. These member functions in the framework of C/C++ are similar to functions in MATLAB Control System Toolbox in simplicity and power. Their syntaxes are comparatively presented in Appendix B. Complete application examples using Ch Control System Toolkit and MATLAB Control System Toolbox are described in Appendix C.

1.2 Features

Written in Ch, Ch Control System Toolkit has following salient features.

1. **C/C++ Compatible**
   Different from other similar control software packages, programs written in Ch Control System Toolkit can work with existing C/C++ programs and libraries seamlessly.

2. **Object-Oriented**
   Implemented as a class, Ch Control System Toolkit is object-oriented.

3. **Embeddable**
   With Embedded Ch, Ch programs using Control System Toolkit can be embedded in other C/C++ application programs.

4. **Web-Based**
   Using the Web-based control design and analysis system in the Ch Control System Toolkit, design and analysis of control systems can be performed through the Web using a Web browser over the internet without any software installation, system configuration and programming.

5. **Model Building**
   Support multiple model representations such as state-space models, transfer function, and zero-pole-gain models both for continuous-time and discrete-time linear time invariant (LTI) systems. When a system is created using one of these three models, the Ch Control System Toolkit automatically converts it to other two models internally and keeps the information of these three models for user to retrieve.

6. **Continuous/Discrete Systems Conversions**
   Provide the capability of converting from continuous to discrete, discrete to continuous, and discrete to discrete system.

7. **System Interconnecting and Reduction**
   Constructs arbitrary complex system models given in a block diagram form by using interconnecting member functions.

8. **Time Domain Analysis**
   Compute the time responses to arbitrary inputs and initial conditions for both SISO and MIMO systems. The typical inputs are step and impulse inputs. The output responses can be displayed by either a plot or a data set.

9. **Frequency Domain Analysis**
   Generate commonly used frequency response plots of Bode, Nyquist and Nichols plots. Calculates the bandwidth, DC gain, gain and phase margins of an SISO system for system dynamics and stability analysis.

10. **Root Locus Design**
    Calculate and plots the root locus of an SISO system.

11. **Controllability and Observability**
    Construct the controllability and observability matrices. Determines the controllability and observability of systems. Calculates the controllability and observability grammians of a state-space model.

12. **State-Space Design**
    Find the control-law and estimator that will allow the user to select a set of pole locations for a satisfactory system dynamic response.

13. **Optimal Control System Design and Equation Solvers**
    Calculate LQ-optimal gain for both continuous and discrete systems. Solves continuous and discrete-time Lyapunov equations.

14. **Plotting Utilities**
    Provide many plotting functions to allow output visually displayed or exported as external files with a variety of different file formats including postscript file, PNG, LaTeX, etc. They can also readily be copied and pasted in other applications such as Word.

1.3 Getting Started

To help users to get familiar with Ch Control System Toolkit, two sample programs will be used to illustrate basic features and applications of Ch Control System Toolkit.

In the first example, the system shown in Figure 1.1 consists of a plant and a feedback controller with transfer functions

$$G(s) = \frac{3}{s(s+2)}, H(s) = \frac{2}{3}.$$

The closed-loop transfer function of the system is

$$T(s) = \frac{G(s)}{1 + G(s)H(s)} = \frac{3}{s^2 + 2s + 2}$$

The step response of the closed-loop system will be plotted in this example.

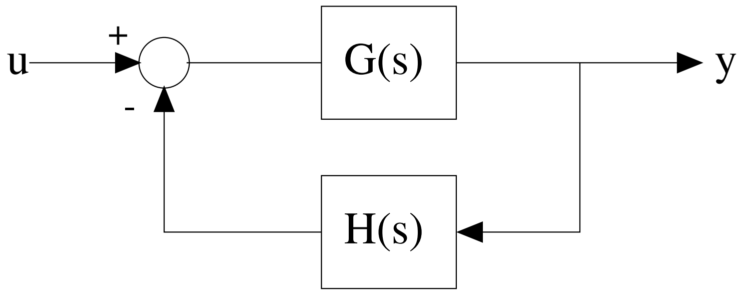


Figure 1.1: Block diagram of the system.

The Ch program for solving this problem is shown in Program 1.1. The first line of program

```
#include <control.h>
```

includes the header file **control.h** which defines the class **CControl**, macros, and prototypes of member functions. Like a C/C++ program, a Ch program will start to execute at the **main**() function after the program is parsed. The next two lines

```
double num[1] = {3};
double den[3] = {1, 2, 2};
```

define two arrays num and den to store the coefficients of the numerator and denominator of the polynomials of the transfer function. Line

```
class CPlot plot;
```

defines a class **CPlot** for creating and manipulating two and three dimensional plotting. The **CPlot** class is defined in **chplot.h** which is included in **control.h** header file. Line

```
class CControl sys;
```

```
#include <control.h>

int main() {
  double num[1] = {3};
  double den[3] = {1, 2, 2};
  class CPlot plot;
  class CControl sys;

  sys.model("tf", num, den);
  sys.step(&plot, NULL, NULL, NULL);

  return 0;
}
```

Program 1.1: A simple example of using Ch Control System Toolkit (sexample1.ch)

instantiates a CControl class. The member function **CControl::model**() constructs a transfer function model of the system as follows.

```
sys.model("tf", num, den);
```

The type of models created by member function **CControl::model**() is specified by the first argument. For example, string `"tf"` indicates transfer function model. The second and third arguments specify the coefficients of the numerator and denominator polynomials of the transfer function. Ch Control System Toolkit supports transfer function (TF), zero-pole-gain (ZPK), state-space (SS), and other LTI models. The details of model types supported by Ch Control System Toolkit and corresponding first argument passed to the member function **CControl::model**() will be described in Chapter 2. The keyword **class** is optional. Line

```
sys.step(&plot, NULL, NULL, NULL);
```

computes and plots the step response of the system. Member function **step**() has four arguments. The first argument is a pointer to an existing object of class **CPlot**. The other three arguments are arrays of reference containing the output of the step response, time vector, and state trajectories. If the output data are not required, these three arguments can be set to NULL. The step response of the system when Program 1.1 is executed is shown in Figure 1.2.

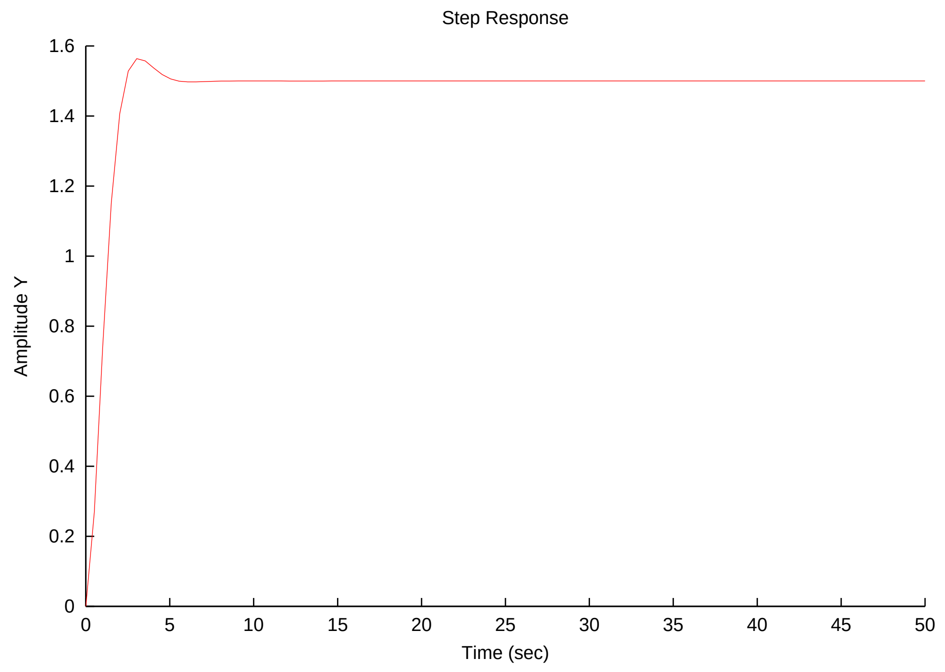


Figure 1.2: Step response of the system.

In the second sample program shown in Program 1.2, the member function **feedback**() is called to construct a closed-loop system in Figure 1.1. The step response of the closed-loop system is also plotted by member function **step**().

Before using member function **feedback**(), the two LTI models for both plant and feedback controller have to be constructed. This is accomplished by the following two lines.

```
sys1.model("tf", num1, den1);
sys2.model("tf", num2, den2);
```

Line

```
 sys3 = sys1.feedback(&sys2);
```

makes a feedback connection between plant $sys1$ and feedback controller $sys2$ and returns a pointer to class CControl as a new system. The output from Program 1.2 is the same as Figure 1.2.

```
#include <control.h>

int main() {
  double num1[1] = {3};
  double den1[3] = {1, 2, 0};
  double num2[1] = {2};
  double den2[1] = {3};
  CPlot plot;
  CControl sys1, sys2, *sys3;

  sys1.model("tf", num1, den1);
  sys2.model("tf", num2, den2);
  sys3 = sys1.feedback(&sys2);
  sys3->step(&plot, NULL, NULL, NULL);

  return 0;
}
```

Program 1.2: Another example of using Ch Control System Toolkit (sexample2.ch)

Chapter 2

LTI Modeling

2.1 LTI Model Types

For analysis of a control system using the Ch Control System Toolkit, the model of the system needs to be constructed first. There are many important properties of control systems. The Ch Control System Toolkit is mainly concerned with linear time-invariant (LTI) system whose input-output relations are linear and characteristics are invariant with time. The LTI systems can be specified by linear, time invariant equations in transfer function (TF), zero-pole-gain (ZPK) or state-space (SS) models. Ch Control System Toolkit supports all these three LTI models. In addition, some models of special purposes are supported by Ch Control System Toolkit is shown in Table 2.1. For a system represented in one of these LTI models, the member function **CControl::model**() with proper arguments can be used to create the model of the system. The first argument of the function indicates the type of the model. The possible values of this argument and corresponding model types are listed in Table 2.1. The critical system parameters, such as zeros, poles and gains, can be retrieved from the existing model by the member functions listed in Table 2.2. These functions will be described in details in the remaining sections.

Table 2.1: Types of model created by **CControl::model**().

The first argument	**model type**
"tf"	transfer function model
"zpk"	zero-pole-gain model
"ss"	state-space model
"rss"	random state-space model
"drss"	random state-space discrete-time model

Table 2.2: Member functions getting system parameters.

Member functions	**Description**
CControl::tfdata()	get the numerators and denominators in TF model.
CControl::zpkdata()	get zeros, poles and gains in ZPK model.
CControl::ssdata()	get system, input/output and direct transmission matrix in SS model.
CControl::printSystem()	print information of system.
CControl::printss()	print state space matrices.
CControl::printtf()	print the numerator and denominator of transfer function.
CControl::printzpk()	print the zeros, poles and gain of system.

```
// tfsys1.ch
// example of creating transfer function model

#include <control.h>
int main() {
  array double num[2] = {1, 3}, den[3] = {1, 2, 9};
  CControl sys;

  sys.model("tf", num, den);
  sys.printSystem();

  return 0;
}
```

Program 2.1: Example of creating transfer function model using Ch Control System Toolkit (`tfsys1.ch`).

2.1.1 Transfer Function Models

An LTI control system can be represented by a transfer function as

$$\mathrm{H}(s) = \frac{\mathrm{num}(s)}{\mathrm{den}(s)}$$

The numerator $num(s)$ and denominator $den(s)$ are both polynomials of the Laplace variable s which may be complex number. In Ch Control System Toolkit, the user can create transfer function model by declaring a **CControl** class and calling the member function **CControl::model**() as follows.

```
CControl sys;
sys.model("tf", num, den);
```

where the CControl class is defined in header file **control.h**. The first argument `"tf"` of a string type indicates that the model to be created is a transfer function model. The other two arguments num and den are two one-dimensional arrays containing the coefficients of the polynomials $num(s)$ and $den(s)$, respectively. Program 2.1 is an example of creating transfer function model using Ch Control System Toolkit. In this program, the transfer function of the control system to be analyzed or controlled is

$$\mathrm{H}(s) = \frac{s+3}{s^2+2s+9}$$

The corresponding arrays passed to the member function **CControl::model**() are $\{1, 3\}$ and $\{1, 2, 9\}$. Because class **CControl** is defined within the header file **control.h**, it has to be included before any member function provided by Ch Control System Toolkit is invoked. The member function **CControl::printSystem**() can be used to print not only the coefficients of the transfer function, but also some other critical parameters of the system, including input, output, and direct transmission matrices in the state space model, zeros, poles and gain in the zero-pole-gain model, as well as delays, sample times. The output from executing Program 2.1 is shown in Figure 2.1.

To get the numerator and denominator of an existing system instantiated as sys of the class CControl, the user can use the member function **CControl::tfdata**() as follows,

```
sys.tfdata(num, den);
```

where num and den are Ch computational arrays of double type. More information about Ch computational array can be found in *Ch User's Guide*. Program 2.2 is an example of getting numerator and denominator

```
Continuous-time System
State-space arguments:
A =
 -2.000000  -9.000000
  1.000000   0.000000
B =
  1.000000
  0.000000
C =
  1.000000   3.000000
D =
  0.000000

Transfer function parameters:
Numerator:
  0.000000   1.000000   3.000000
Denominator:
  1.000000   2.000000   9.000000

Zero-Pole-Gain parameters:
Zero:
complex( -3.000000,  0.000000)
Pole:
complex( -1.000000,  2.828427) complex( -1.000000, -2.828427)
Gain:   1.000000

input delay is   0.000000
output delay is   0.000000
input/output delay is   0.000000
```

Figure 2.1: The output from executing Program 2.1.

```
// tfdata.ch
// example of getting numerator and denominator of
// a transfer function model

#include <control.h>
int main() {
  array double num1[2] = {1, 3}, den1[3] = {1, 2, 9};
  CControl sys;

  sys.model("tf", num1, den1);
  int nnum = sys.size('n');
  int nden = sys.size('d');
  array double num[nnum], den[nden];
  sys.tfdata(num, den);
  printf("num = %f\nden = %f\n", num, den);

  return 0;
}
```

Program 2.2: Example of getting numerator and denominator of the transfer function model (`tfdata.ch`).

```
num = 0.000000 1.000000 3.000000

den = 1.000000 2.000000 9.000000
```

Figure 2.2: The output from executing Program 2.2.

of the transfer function model created in Program 2.1. In order to declare the deferred-shape arrays num and den with proper sizes, the member function **CControl::size**() is called to retrieve orders of the numerator and denominator by passing arguments $'n'$ and $'d'$, respectively. The details of member function **CControl::size**() will be described in section 2.2. The output from executing Program 2.2 is shown in Figure 2.2.

2.1.2 Zero-Pole-Gain Models

The zero-pole-gain model is another commonly used model for control system design. In this model, the rational transfer function is represented as a ratio of two polynomials in factored zero-pole model as follows.

$$\mathrm{H}(s) = k\frac{(s-z_1)\ldots(s-z_m)}{(s-p_1)\ldots(s-p_n)}$$

where k is called the transfer function gain which is a real scalar calue. The roots of the numerator, $z_1, \ldots, z_m$, and the roots of the denominator, $p_1, \ldots, p_n$, are called zeros and poles of the system, respectively. They present in real or complex conjugate pairs. The zeros correspond to the signal transmission-blocking properties of the system, whereas the poles determine the stability properties and unforced behavior of the system. In Ch Control System Toolkit, the user can create a zero-pole-gain mode as follows.

```
sys.model("zpk", z, p, k);
```

where the first argument `"zpk"` indicates that the model to be created is a zero-pole-gain model. The other arguments, z, p and k, contain the zeros, poles and gain of the system. z and p are one-dimensional

```
// zpksys1.ch
// example of creating zero-pole-gain model

#include <control.h>
int main() {
  array double complex zero[1]={complex(-3, 0)},
                       pole[2]={complex(-1.000000, 2.828427),
                                complex(-1.000000, -2.828427)};
  double k=1;
  CControl sys;

  sys.model("zpk", zero, pole, k);
  sys.printSystem();

  return 0;
}
```

Program 2.3: Example of creating zero-pole-gain model using Ch Control System Toolkit (`zpksys1.ch`).

arrays and k is of type double. Program 2.3 is an example of creating zero-pole-gain model using **CControl::model**() member function. In this program, the system sys has one zero, -3, and two poles, $-1 + 2.828427i$ and $-1 - 2.828427i$. The system gain is 1. The transfer function can be written as follows,

$$\mathrm{H}(s) = \frac{s+3}{(s+1-2.828427i)(s+1+2.828427i)}$$

The arguments passed to the member function **CControl::model**() are arrays of $\{3\}$ for the zero and $\{complex(-1.0, 2.828427), complex(-1.0, -2.828427)\}$ for the poles, and 1 for the gain of the system. The output from executing Program 2.3 is shown in Figure 2.3.

To get the zeros, poles and gain of an existing system sys, the user can use the member function **CControl::zpkdata**() as follows,

```
sys.zpkdata(zero, pole, k);
```

where $zero$ and $pole$ are Ch computational array of complex type, and k is a pointer to double. More information about complex number in Ch can be found in *Ch User's Guide*.

2.1.3 State-Space Models

In Ch Control System Toolkit, both single-input/single-output (SISO) systems and multiple-input/multiple-output (MIMO) systems can be modeled in the state-space model. Typically the state-space model of a system consists of two first-order differential vector equations in the form of

$$\begin{aligned} \dot{X} &= AX + BU \\ Y &= CX + DU \end{aligned}$$

For an n_xth order system with n_u inputs and n_y outputs, the vector X, which contains the states of the system, has the shape of ($n_x \times 1$). Vectors U and Y, which are input and output of the system, respectively , have shapes of ($n_u \times 1$) and ($n_y \times 1$). The quantity A is an ($n_x \times n_x$) system matrix, B is an ($n_x \times n_u$) input matrix, C is an ($n_y \times n_x$) output matrix, D is an ($n_y \times n_u$) direct transmission matrix. In Ch Control System Toolkit, the user can create a state-space mode as follows,

```
sys.model("ss", A, B, C, D);
```

```
Continuous-time System
State-space arguments:
A =
 -2.000000  -8.999999
  1.000000   0.000000
B =
  1.000000
  0.000000
C =
  1.000000   3.000000
D =
  0.000000

Transfer function parameters:
Numerator:
  0.000000   1.000000   3.000000
Denominator:
  1.000000   2.000000   8.999999

Zero-Pole-Gain parameters:
Zero:
complex( -3.000000,  0.000000)
Pole:
complex( -1.000000,  2.828427) complex( -1.000000, -2.828427)
Gain:   1.000000

input delay is   0.000000
output delay is   0.000000
input/output delay is   0.000000
```

Figure 2.3: The output from executing Program 2.3.

```
// sssys1.ch
// example of creating SISO state space model

#include <control.h>

#define NUMX 2        // number of states
#define NUMU 1        // number of inputs
#define NUMY 1        // number of outputs

int main() {
    array double A[NUMX][NUMX] = {{-2, -9},
                                  {1,  0}},
                 B[NUMX][NUMU] = {1, 0},
                 C[NUMY][NUMX] = {1, 3},
                 D[NUMY][NUMU] = {0};
    CControl sys;

    sys.model("ss", A, B, C, D);
    sys.printSystem();

    return 0;
}
```

Program 2.4: Example of creating SISO state-space model using Ch Control System Toolkit (`sssys1.ch`).

Where the first argument `"ss"` indicates that a state-space model will be created. The subsequent four arguments are system matrix, input matrix, output matrix and direct transmission matrix. They are all represented as two-dimensional computational arrays in Ch programs. Two examples of creating state-space model using member function **CControl::model**() will be presented. Program 2.4 named `sssys1.ch` creates an SISO model with 2 states. The output from executing Program 2.4 is shown in Figure 2.4. The other example `sssys2.ch` shown in Program 2.5 creates an MIMO model with 2 inputs, 2 outputs and 4 states. The output from executing `sssys2.ch` is shown in Figure 2.5.

If a matrix in state-space model is absent, the value of a $NULL$ pointer can be used to replace that matrix. For example, to represent the following system *sys_ac*,

$$\begin{aligned}\dot{X} &= AX \\ Y &= CX\end{aligned}$$

the user can call member function **CControl::model**() as follows,

```
    sys_ac.model("ss", A, NULL, C, NULL);
```

In the header file **control.h**, all possible state-space model are defined as macros listed in Table 2.3. The macro CHTKT_CONTROL_SYSTEMTYPE_TFO indicates the system, such as a single derivative unit, which cannot be represented using a state-space model. The model represented by macro CHTKT_CONTROL_SYSTEMTYPE_AB is designed for a situation where only the system matrix A and input matrix B are involved in the calculation such as for **CControl::ctrb**() member function. Similarly, The CHTKT_CONTROL_SYSTEMTYPE_AC model can be used for a situation where only the system matrix A and output matrix C are involved such as for **CControl::obsv**() member function. **CControl::getSystemType()** returns one of these macros according to the system type. For the above system, the following expression

```
    sys_ac.getSystemType()==CHTKT_CONTROL_SYSTEMTYPE_AC
```

```
Continuous-time System
State-space arguments:
A =
 -2.000000  -9.000000
  1.000000   0.000000
B =
  1.000000
  0.000000
C =
  1.000000   3.000000
D =
  0.000000

Transfer function parameters:
Numerator:
  0.000000   1.000000   3.000000
Denominator:
  1.000000   2.000000   9.000000

Zero-Pole-Gain parameters:
Zero:
complex( -3.000000,  0.000000)
Pole:
complex( -1.000000,  2.828427) complex( -1.000000, -2.828427)
Gain:   1.000000

input delay is   0.000000
output delay is   0.000000
input/output delay is   0.000000
```

Figure 2.4: The output from executing Program 2.4.

```
// sssys2.ch
// example of creating MIMO state space model

#include <control.h>

#define NUMX 4        // number of states
#define NUMU 2        // number of inputs
#define NUMY 2        // number of outputs

int main() {
    array double A[NUMX][NUMX] = {{1,  2,  3,  4},
                                  {5,  6,  7,  8},
                                  {9,  10, 11, 12},
                                  {13, 14, 15, 16}},
                 B[NUMX][NUMU] = {1, 0,
                                  0, 1,
                                  2, 3,
                                  4, 5},
                 C[NUMY][NUMX] = {1, 2, 3, 4,
                                  5, 6, 7, 8},
                 D[NUMY][NUMU] = {0, 0, 0, 0};
    CControl sys;

    sys.model("ss", A, B, C, D);
    sys.printss();

    return 0;
}
```

Program 2.5: Example of creating MIMO state-space model using Ch Control System Toolkit (`sssys2.ch`).

```
State-space arguments:
A =
  1.000000   2.000000   3.000000   4.000000
  5.000000   6.000000   7.000000   8.000000
  9.000000  10.000000  11.000000  12.000000
 13.000000  14.000000  15.000000  16.000000
B =
  1.000000   0.000000
  0.000000   1.000000
  2.000000   3.000000
  4.000000   5.000000
C =
  1.000000   2.000000   3.000000   4.000000
  5.000000   6.000000   7.000000   8.000000
D =
  0.000000   0.000000
  0.000000   0.000000
```

Figure 2.5: The output from executing Program 2.5.

is true.

Table 2.3: Macros of system types.

Macros	**System Type**
CHTKT_CONTROL_SYSTEMTYPE_INVALID	invalid system
CHTKT_CONTROL_SYSTEMTYPE_EMPTY	empty system
CHTKT_CONTROL_SYSTEMTYPE_TFO	transfer function model only
CHTKT_CONTROL_SYSTEMTYPE_ABCD	$\dot{X} = AX + BU; Y = CX + DU$
CHTKT_CONTROL_SYSTEMTYPE_D	$Y = DU$
CHTKT_CONTROL_SYSTEMTYPE_AC	$\dot{X} = AX; Y = CX$
CHTKT_CONTROL_SYSTEMTYPE_AB	$\dot{X} = AX + BU$
CHTKT_CONTROL_SYSTEMTYPE_A	$\dot{X} = AX$

In some cases, the user wants to create an state-space system, but doesn't care about the characteristics of the system of CControl class itself. A random system can be created by passing the string `"rss"` as the first argument to the member function **CControl::model**(). For example, statement

```
sys.model("rss");
```

creates a model of SISO, first order system with random system matrix, input matrix, output matrix and direct transmission matrix. The declaration statement

```
sys.model("rss", nx, nu, ny);
```

constructs a model of (nx)th order random system with nu inputs and ny outputs.

To get the system matrix, input matrix, output matrix and direct transmission matrix of an existing system sys, the user can use the member function **CControl::ssdata**() as follows.

```
sys.ssdata(A, B, C, D);
```

where A, B, C and D are Ch computational array of double type.

2.1.4 Discrete-Time Models

Ch Control System Toolkit can handle not only continuous-time models, but also discrete-time models. The user can easily create a model for discrete-time system by passing an extra argument ts representing sample time to the member function **CControl::model**() as shown below

```
sys.model("tf", num, den, 0.5);
```

The line above initializes a model of discrete-time system with sample time of 0.5. If the value of argument ts equals to 0, the model will be treated as a continuous-time model. Discrete-time system can also be modeled using transfer function, state-space, and zero-pole-gain. Program 2.6 is an example of creating discrete-time transfer function model using Ch Control System Toolkit.

To create a discrete-time model using the default sample time, the default values of sample time for different member functions might be different. For example, the default value of 1 will be used for step and impulse functions. A negative value of -1 can be used to replace the sample time, for example

```
sys.model("ss", A, B, C, D, -1);
```

The member function **CControl::isDiscrete**() can be used to check if a system is a discrete-time system. It returns 1 for discrete-time system, 0 for continuous-time system.

```
// tfsys3.ch
// example of creating discrete-time transfer function model

#include <control.h>
int main() {
  array double num[2] = {1, 3}, den[3] = {1, 2, 9};
  CControl sys;

  sys.model("tf", num, den, .5);
  sys.printSystem();

  return 0;
}
```

Program 2.6: Example of creating discrete-time model (`tfsys2.ch`).

```
Discrete-time System with sample time of 0.500000s.
State-space arguments:
A =
 -2.000000  -9.000000
  1.000000   0.000000
B =
  1.000000
  0.000000
C =
  1.000000   3.000000
D =
  0.000000

Transfer function parameters:
Numerator:
  0.000000   1.000000   3.000000
Denominator:
  1.000000   2.000000   9.000000

Zero-Pole-Gain parameters:
Zero:
complex( -3.000000,  0.000000)
Pole:
complex( -1.000000,  2.828427) complex( -1.000000, -2.828427)
Gain:   1.000000

input delay is   0.000000
output delay is   0.000000
input/output delay is   0.000000
```

Figure 2.6: The output from executing Program 2.6.

```
// setdelay.ch
// example of setting delays

#include <control.h>
int main() {
  array double num[2] = {1, 3}, den[3] = {1, 2, 9};
  CControl sys;

  sys.model("tf", num, den);
  sys.setDelay(1, .1);
  sys.setDelay(2, .2);
  sys.setDelay(3, .3);
  sys.printSystem();

  return 0;
}
```

Program 2.7: Example of setting delays (`setdelay.ch`).

2.1.5 Models with Delays

Given an SISO state-space model as follows

$$\begin{aligned}\dot{x}(t) &= Ax(t) + Bu(t-\tau)\\ y(t) &= Cx(t-\theta)\end{aligned}$$

where τ is called input delay which is between the input u and the state vector x, and θ is called output delay which is between the x and the output y. In addition, delays called input/output delays which is between u and y can be found in transfer function models.

The member function **CControl::setDelay**() allows users to specify these delays after an LTI model is created. The syntax of this function is shown as follows.

```
    sys.setDelay(type, delay);
```

where $type$ is an integer which indicates the type of the delay to be set. The description of the possible values of $type$ are listed in Table 2.4. The second argument $delay$ is the value of the selected delay. The user can use member function **CControl::hasdelay**() to determine if the system has delays. Program 2.7 is an example which set input, output and input/output delays in the transfer function model.

Table 2.4: Meanings of different values of the 1st argument of member function **CControl::setDelay**().

Value	**Description**
1	set the input delay in the system
2	set the output delay in the system
3	set the input/output delay in the system

2.1.6 Empty Models

Sometimes, the user needs to call some member functions to perform some calculations which do not depend on the characteristics of the system itself. In this case, the member function **CControl::model**() does not need to be called to build the control model. For example, the user can just call the member function **CControl::lyap**() to solve a Lyapunov equation as follows.

```
Continuous-time System
State-space arguments:
A =
 -2.000000  -9.000000
  1.000000   0.000000
B =
  1.000000
  0.000000
C =
  1.000000   3.000000
D =
  0.000000

Transfer function parameters:
Numerator:
  0.000000   1.000000   3.000000
Denominator:
  1.000000   2.000000   9.000000

Zero-Pole-Gain parameters:
Zero:
complex( -3.000000,  0.000000)
Pole:
complex( -1.000000,  2.828427) complex( -1.000000, -2.828427)
Gain:   1.000000

input delay is   0.100000
output delay is   0.200000
input/output delay is   0.300000
```

Figure 2.7: The output from executing Program 2.7.

```
CControl sys;
sys.lyap(x, a, b, q);
...
```

The solution of the Lyapunov equation depends only on the arguments of a, b and q. The member function **CControl::lyap**() will be described in detail in section 4.4.

2.2 General Model Characteristics

Each control system has its own characteristics such as system order, input/output number. Ch Control System Toolkit provides member functions to setting and retrieve the values related to its characteristics. They are listed in the Table 2.5. For system with delays, including input delays, output delays, and I/O delays, the member function **CControl::hasdelay()** returns 1, otherwise 0. The member function **CControl::isDiscrete()** returns 1 if the system is a discrete-time system, otherwise 0. Program 2.8 is an example of getting some system characteristics by calling member functions **CControl::getSystemType()**, **CControl::hasdelay()**, and **CControl::isDiscrete()**.

Table 2.5: General model characteristics and related member functions.

Function	Description
CControl::getSystemType()	get system type
CControl::getTs()	get sample time of discrete-time system
CControl::setTs()	set sample time of discrete-time system
CControl::hasdelay()	determine if the system has any delay
CControl::setDelay()	set delay in the system
CControl::isDiscrete()	determine if the system is discrete
CControl::size()	get system parameters' sizes

Table 2.6: Meanings of different values of the argument of member function **CControl** ::**size** ().

Value	Description
x	get the order of the system
y	get the number of the output of the system
u	get the number of the input of the system
n	get the order of the numerator of the transfer function
d	get the order of the denominator of the transfer function
z	get the number of the zeros of the system
p	get the number of the poles of the system

As it has been used in Program 2.2, the member function **CControl::size()** returns dimensions of output/input/system matrix for state-space models, orders of numerator and denominator for transfer function models, or numbers of zeros and poles for zero-pole-gain models, according to different values of the argument passed. Table 2.6 describes meanings of these different values.

2.3 Model and System Conversion

As it has been revealed by the member function **CControl::printSystem()**, systems created in the Programs 2.1 - 2.4 are identical internally. When a system is created using one of three models of state-space,

```
// syschar.ch
// example of getting system characteristics

#include <control.h>

#define NUMX 2        // number of states
#define NUMU 1        // number of inputs
#define NUMY 1        // number of outputs

void syschar(CControl *pSys);
int main() {
    array double A[NUMX][NUMX] = {{-2, -9},
                                  {1,  0}},
                 B[NUMX][NUMU] = {1, 0},
                 C[NUMY][NUMX] = {1, 3},
                 D[NUMY][NUMU] = {0};
    CControl sys1;
    CControl sys2;
    CControl sys3;

    sys1.model("ss", A, B, C, D);
    sys1.setDelay(1, .5);
    syschar(&sys1);

    sys2.model("ss", A, NULL, NULL, NULL, -1);
    syschar(&sys2);

    syschar(&sys3);

    return 0;
}

void syschar(CControl *pSys){
   switch(pSys->getSystemType()) {
      case CHTKT_CONTROL_SYSTEMTYPE_ABCD:
         printf("\nThe system type is : dot_x=Ax+Bu; y=Cx+Du\n");
         break;
      case CHTKT_CONTROL_SYSTEMTYPE_D:
         printf("\nThe system type is : y=Du\n");
         break;
      case CHTKT_CONTROL_SYSTEMTYPE_AC:
         printf("\nThe system type is : dot_x=Ax; y=Cx\n");
         break;
      case CHTKT_CONTROL_SYSTEMTYPE_AB:
         printf("\nThe system type is : dot_x=Ax+Bu\n");
         break;
      case CHTKT_CONTROL_SYSTEMTYPE_A:
         printf("\nThe system type is : dot_x=Ax\n");
         break;
      case CHTKT_CONTROL_SYSTEMTYPE_EMPTY:
         printf("\nThe system type is : empty system.\n");
         break;
      case CHTKT_CONTROL_SYSTEMTYPE_TFO:
         printf("\nThe system only has transfer function model.\n");
         break;
      case CHTKT_CONTROL_SYSTEMTYPE_INVALID:
         printf("\nThe system type is invalid.\n");
         break;
      default:
         printf("\nError: Incorrect mode specified.\n");
   }

   if(pSys->isDiscrete()) {
         printf("The system is discrete-time system.\n");
```

```
The system type is : dot_x=Ax+Bu; y=Cx+Du
The system has delays.

The system type is : dot_x=Ax
The system is discrete-time system.

The system type is : empty system.
```

Figure 2.8: The output from executing Program 2.8.

transfer function and zero-pole-gain models, the Ch Control System Toolkit automatically converts it to other two models internally. The user can retrieve any parameter of these three models by calling member functions **CControl::tfdata()**, **CControl::ssdata()**, and **CControl::zpkdata()**. For MIMO systems, only state-space models are available in Ch Control System Toolkits in its current implementation.

To improve the conditioning of the system matrix A, sometimes the user needs to perform similarity transformation on the state vector X, and produces an equivalent state-space model. Assume that the original state-space model is

$$\begin{aligned}\dot{X} &= AX + BU\\ Y &= CX + DU\end{aligned}$$

and the similarity transformation is $\bar{X} = TX$, the equivalent state-space model is

$$\begin{aligned}\dot{\bar{X}} &= TAT^{-1}\bar{X} + TBU\\ Y &= CT^{-1}\bar{X} + DU\end{aligned}$$

Typically, this generated system has the same input/output characteristics as the original one. The member function **CControl::ss2ss()** can perform such transformation. Its syntax is

```
    syst = sys.ss2ss(T);
```

where the state coordinate transformation T is a computation array of double type. The function returns a pointer of **CControl** class. Program 2.9 transforms the system sys to its equivalent state-space model $syst$. They have different state-space models, but the same transfer functions.

Member functions **CControl::c2d()** and **CControl::d2c()** can explicitly convert systems from continuous-time models to discrete-time models and vice versa. Member function **CControl::d2d()** converts discrete-time models to discrete-time models which have different sample time. Program 2.10 is an example of system conversions between continuous-time models and discrete-time models.

For systems that have delays, member function **CControl::delay2z()** replaces all delays with a phase shift.

Table 2.7: Member functions performing system conversions.

Member functions	**Description**
CControl::ss2ss()	state coordinate transformation for state-space models
CControl::c2d()	convert systems from continuous-time models to discrete-time models
CControl::d2c()	convert systems from discrete-time models to continuous-time models
CControl::d2d()	convert discrete-time models to discrete-time models with different sample time
CControl::delay2z()	replace all delays with a phase shift

```
// ss2ss.ch
// example of coordinate transformation for state-space models

#include <control.h>

#define NUMX 2        // number of states
#define NUMU 1        // number of inputs
#define NUMY 1        // number of outputs

int main() {
    array double A[NUMX][NUMX] = {{-2, -9},
                                  {1,  0}},
                 B[NUMX][NUMU] = {1, 0},
                 C[NUMY][NUMX] = {1, 3},
                 D[NUMY][NUMU] = {0};

    array double T[NUMX][NUMX] = {{-4.12, -17.4},
                                  {4,     0}};
    CControl sys;
    CControl *sysT;

    sys.model("ss", A, B, C, D);
    sysT = sys.ss2ss(T);
    sysT->printSystem();

    return 0;
}
```

Program 2.9: Example of state coordinate transformation (`ss2ss.ch`).

```
Continuous-time System
State-space arguments:
A =
 -2.131034  -4.484966
  2.068966   0.131034
B =
 -4.120000
  4.000000
C =
 -0.172414   0.072414
D =
  0.000000

Transfer function parameters:
Numerator:
  0.000000   1.000000   3.000000
Denominator:
  1.000000   2.000000   9.000000

Zero-Pole-Gain parameters:
Zero:
complex( -3.000000,  0.000000)
Pole:
complex( -1.000000,  2.828427) complex( -1.000000, -2.828427)
Gain:   1.000000

input delay is   0.000000
output delay is   0.000000
input/output delay is   0.000000
```

Figure 2.9: The output from executing Program 2.9.

```
// sssys1.ch
// example of creating SISO state space model

#include <control.h>

#define NUMX 2        // number of states
#define NUMU 1        // number of inputs
#define NUMY 1        // number of outputs

int main() {
    array double A[NUMX][NUMX] = {{-2, -9},
                                  {1,  0}},
                 B[NUMX][NUMU] = {1, 0},
                 C[NUMY][NUMX] = {1, 3},
                 D[NUMY][NUMU] = {0};
    CControl *sysd1, *sysd2;
    CControl sys;

    sys.model("ss", A, B, C, D);
    sys.printSystem();

    sysd1 = sys.c2d(.5, "zoh");
    sysd1->printSystem();

    sysd2 = sysd1->d2d(1);
    sysd2->printSystem();

    return 0;
}
```

Program 2.10: Example of system conversions (`modconv.ch`).

```
Continuous-time System
State-space arguments:
A =
 -2.000000  -9.000000
  1.000000   0.000000
B =
  1.000000
  0.000000
C =
  1.000000   3.000000
D =
  0.000000

Transfer function parameters:
Numerator:
  0.000000   1.000000   3.000000
Denominator:
  1.000000   2.000000   9.000000

Zero-Pole-Gain parameters:
Zero:
complex( -3.000000,  0.000000)
Pole:
complex( -1.000000,  2.828427) complex( -1.000000, -2.828427)
Gain:   1.000000

input delay is   0.000000
output delay is   0.000000
input/output delay is   0.000000

Discrete-time System with sample time of 0.500000s.
State-space arguments:
A =
 -0.117233  -1.906357
  0.211817   0.306402
B =
  0.211817
  0.077066
C =
  1.000000   3.000000
D =
  0.000000

Transfer function parameters:
Numerator:
  0.000000   0.443017  -0.050113
Denominator:
  1.000000  -0.189169   0.367879

Zero-Pole-Gain parameters:
Zero:
complex(  0.113118,  0.000000)
Pole:
complex(  0.094585,  0.599110) complex(  0.094585, -0.599110)
Gain:   0.443017
```

Figure 2.10: The output from executing Program 2.10.

```
input delay is   0.000000
output delay is   0.000000
input/output delay is   0.000000

Discrete-time System with sample time of 1.000000s.
State-space arguments:
A =
 -0.390056  -0.360624
  0.040069  -0.309918
B =
  0.040069
  0.145546
C =
  1.000000   3.000000
D =
  0.000000

Transfer function parameters:
Numerator:
  0.000000   0.476709   0.135061
Denominator:
  1.000000   0.699974   0.135335

Zero-Pole-Gain parameters:
Zero:
complex( -0.283320,  0.000000)
Pole:
complex( -0.349987,  0.113333) complex( -0.349987, -0.113333)
Gain:   0.476709

input delay is   0.000000
output delay is   0.000000
input/output delay is   0.000000
```

Figure 2.10: The output from executing Program 2.10 (Contd.).

Chapter 3

Manipulating LTI Models

3.1 Interconnecting and Reduction

If the individual objects are available as subsystems in a control system, it is possible to use Ch Control System Toolkit to create a new object of the interconnected systems. Ch Control System Toolkit provides three member functions **CControl::series**(), **CControl::parallel**(), and **CControl::feedback**() for typical forms of interconnection of series, parallel, and feedback, respectively.

Table 3.1: Member functions interconnecting and reducing system models.

Member functions	**Description**
CControl::append()	group LTI models by appending their inputs and outputs
CControl::augstate()	append the state vector to the output vector
CControl::connect()	derive state-space model from block diagram description
CControl::series()	series connection of LTI models
CControl::parallel()	parallel connection of LTI models
CControl::feedback()	feedback connection of two LTI models
CControl::minreal()	minimal realization of the system

The series connection of control systems is illustrated in Figure 3.1, where system sys consists of two subsystems $sys1$ and $sys2$. They are connected in series. The input of $sys1$ and the output of $sys2$ are the input and output of the sys. The first output of $sys1$ is connected as the second input of $sys2$. To build the system sys, the member function **CControl::series**() can be called as follows.

```
array int output1[1] = {1};
array int input2[1] = {2};
sys = sys1.series(sys2, output1, input2);
```

Arrays $output1$ and $input2$ contain indices of the outputs of the first subsystem and inputs of the second subsystem which will be connected with each other. The index starts with 1. Program 3.1 makes a series interconnection of these two subsystems $sys1$ and $sys2$ with the output of the program in Figure 3.2.

Another form of interconnction is parallel which is illustrated in Figure 3.3, where system sys consists of two subsystems $sys1$ and $sys2$ that are connected in parallel. The interconnected system sys has three inputs $u1$, $u2$, $u3$ and three outputs $y1$, $y2$, $y3$. Each of its two subsystems has two inputs and two outputs. Two of their inputs $u12$ and $u21$ are connected with each other and treated as the second input of sys, and so do the outputs. Other two inputs, $u11$ and $u22$, and two outputs, $y11$ and $y22$ are treated as inputs and outputs of sys. To build the system sys, the member function **CControl::parallel**() can be called as follows.

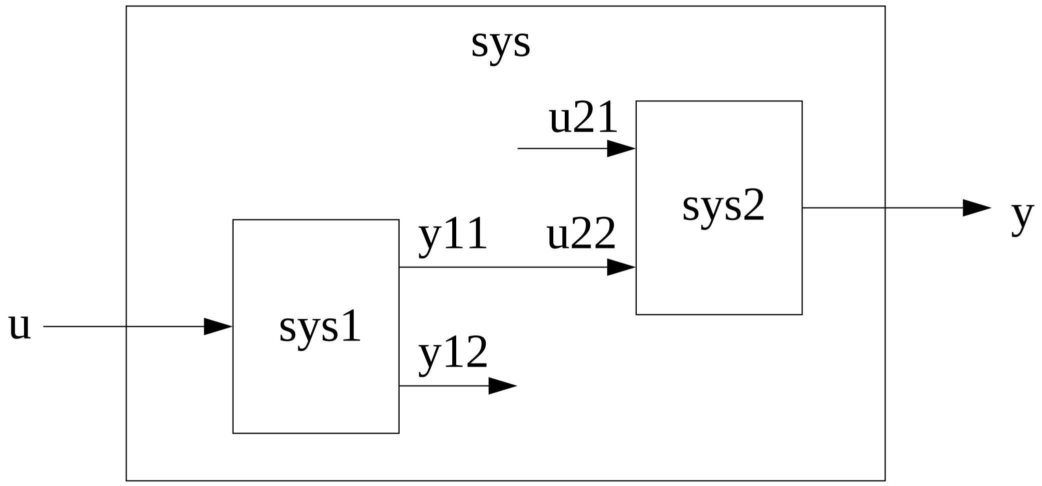


Figure 3.1: Series connection.

```
array int input1[1] = {2}, input2[1] = {1};
array int output1[1] = {2}, output2[1] = {1};
sys = sys1.parallel(sys2, input1, input2, output1, output2);
```

The index arrays $input1$ and $input2$ specify which inputs of $sys1$ and which inputs of $sys2$ are connected. Similarly, the index arrays $output1$ and $output2$ specify which outputs of $sys1$ and which outputs of $sys2$ are summed. Program 3.2 makes a parallel interconnection of two same subsystems $sys1$ and $sys2$ with output of the the program in Figure 3.4. Each of them has 3 inputs and 2 outputs, and the input 2 of system $sys1$ is connected to the input 1 of system $sys2$, and the output 2 of $sys1$ is summed with the output 1 of $sys2$ as shown in Figure 3.3.

The third form of the interconnection is feedback which is illustrated in Figure 3.5, where system sys is a closed-loop model with a feedback loop. The output $y2$ of the subsystem $sys1$ is fed out as the input to $sys2$. The output of $sys2$ is summed with the second input of sys. To build the system sys, the member function **CControl::feedback**() can be called as follows.

```
array int feedin[1] = {2}, feedout[1] = {2};
sys = sys1.feedback(sys2, feedin, feedout);
```

the array $feedin$ contains indices of the input vector of $sys1$ and specifies which inputs are used in the feedback loop. Similarly, array $feedout$ specifies which outputs of $sys1$ are used for feedback. The resulting model sys has the same inputs and outputs as $sys1$. Program 3.3 makes a feedback interconnection of these two subsystems $sys1$ and $sys2$. The output of the program is shown in Figure 3.7. For an SISO control system shown in Figure 3.6, the arrays $feedin$ and $feedout$ could be omitted in the member function **CControl::feedback**() as shown below.

```
sys = sys1.feedback(sys2);
```

Program 3.4 makes such a feedback interconnection with the output of the program in Figure 3.8.

Other member functions provided by Ch Control System Toolkit for interconnecting systems are listed in Table 3.1. Two commonly used member functions are **CControl::append**() and **CControl::connect**(). These two member functions can be used to construct models for complex systems given in block diagram forms. Different from **CControl::series**() function, the **connect**() function can specify the external inputs and outputs of the connected system by arrays $input$ and $output$. In Figure 3.9, the first output of $sys1$

```
// series.ch
// example of series connection

#include <control.h>

#define NUMX1 4    // number of states of system 1
#define NUMU1 1    // number of inputs of system 1
#define NUMY1 2    // number of outputs of system 1
#define NUMX2 4    // number of states of system 2
#define NUMU2 2    // number of inputs of system 2
#define NUMY2 1    // number of outputs of system 2

int main() {
    array double A1[NUMX1][NUMX1] = {{-4.12, -17.4, -30.8, -60},
                                     {1,     0,     0,     0},
                                     {0,     1,     0,     0},
                                     {0,     0,     1,     0}},
                 B1[NUMX1][NUMU1] = {1, 0, 0, 0},
                 C1[NUMY1][NUMX1] = {4, 8.4, 30.78, 60,
                                     3, 7.4, 29.78, 50},
                 D1[NUMY1][NUMU1] = {1, 2};
    array double A2[NUMX2][NUMX2] = {{-4.12, -17.4, -30.8, -60},
                                     {1,     0,     0,     0},
                                     {0,     1,     0,     0},
                                     {0,     0,     1,     0}},
                 B2[NUMX2][NUMU2] = {1, 0,
                                     2, 0,
                                     0, 1,
                                     0, 1},
                 C2[NUMY2][NUMX2] = {4, 8.4, 30.78, 60},
                 D2[NUMY2][NUMU2] = {1, 2};
    array int input2[1] = {2},
              output1[1] = {1};
    CControl sys1;
    CControl sys2;
    CControl *sys;

    sys1.model("ss", A1, B1, C1, D1);
    sys2.model("ss", A2, B2, C2, D2);
    sys = sys1.series(&sys2, output1, input2);
    sys->printss();

    return 0;
}
```

Program 3.1: Example of series interconnection (`series.ch`).

```
State-space arguments:
A =
 -4.120000 -17.400000 -30.800000 -60.000000   0.000000   0.000000   0.000000
  0.000000
  1.000000   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000
  0.000000
  0.000000   1.000000   0.000000   0.000000   4.000000   8.400000  30.780000
 60.000000
  0.000000   0.000000   1.000000   0.000000   4.000000   8.400000  30.780000
 60.000000
  0.000000   0.000000   0.000000   0.000000  -4.120000 -17.400000 -30.800000
-60.000000
  0.000000   0.000000   0.000000   0.000000   1.000000   0.000000   0.000000
  0.000000
  0.000000   0.000000   0.000000   0.000000   0.000000   1.000000   0.000000
  0.000000
  0.000000   0.000000   0.000000   0.000000   0.000000   0.000000   1.000000
  0.000000
B =
  0.000000
  0.000000
  1.000000
  1.000000
  1.000000
  0.000000
  0.000000
  0.000000
C =
  4.000000   8.400000  30.780000  60.000000   8.000000  16.800000  61.560000
120.000000
D =
  2.000000
```

Figure 3.2: The output from executing Program 3.1.

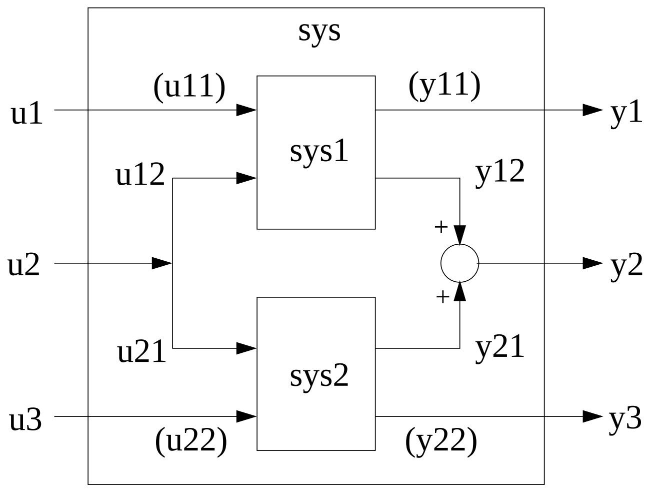


Figure 3.3: Parallel connection.

```
// parallel.ch
// example of parallel connection

#include <control.h>

#define NUMX 4    // number of states
#define NUMU 3    // number of inputs
#define NUMY 2    // number of outputs

int main() {
    array double A[NUMX][NUMX] = {{-4.12, -17.4, -30.8, -60},
                                   {1,     0,     0,     0},
                                   {0,     1,     0,     0},
                                   {0,     0,     1,     0}},
                 B[NUMX][NUMU] = {1, 0, 0,
                                  2, 0, 0,
                                  0, 0, 1,
                                  0, 1, 0},
                 C[NUMY][NUMX] = {4, 8.4, 30.78, 60,
                                  3, 7.4, 29.78, 50},
                 D[NUMY][NUMU] = {1, 2, 3,
                                  4, 5, 6};
    array int input1[1] = {2}, input2[1] = {1};
    array int output1[1] = {2}, output2[1] = {1};
    CControl sys1;
    CControl sys2;
    CControl *sys3;

    sys1.model("ss", A, B, C, D);
    sys2.model("ss", A, B, C, D);
    sys3 = sys1.parallel(&sys2, input1, input2, output1, output2);
    sys3->printss();

    return 0;
}
```

Program 3.2: Example of parallel interconnection (`parallel.ch`).

```
State-space arguments:
A =
 -4.120000 -17.400000 -30.800000 -60.000000   0.000000   0.000000   0.000000
  0.000000
  1.000000   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000
  0.000000
  0.000000   1.000000   0.000000   0.000000   0.000000   0.000000   0.000000
  0.000000
  0.000000   0.000000   1.000000   0.000000   0.000000   0.000000   0.000000
  0.000000
  0.000000   0.000000   0.000000   0.000000  -4.120000 -17.400000 -30.800000
-60.000000
  0.000000   0.000000   0.000000   0.000000   1.000000   0.000000   0.000000
  0.000000
  0.000000   0.000000   0.000000   0.000000   0.000000   1.000000   0.000000
  0.000000
  0.000000   0.000000   0.000000   0.000000   0.000000   0.000000   1.000000
  0.000000
B =
  1.000000   0.000000   0.000000   0.000000   0.000000
  2.000000   0.000000   0.000000   0.000000   0.000000
  0.000000   1.000000   0.000000   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   2.000000   0.000000   0.000000
  0.000000   0.000000   0.000000   0.000000   1.000000
  0.000000   0.000000   0.000000   1.000000   0.000000
C =
  4.000000   8.400000  30.780000  60.000000   0.000000   0.000000   0.000000
  0.000000
  3.000000   7.400000  29.780000  50.000000   4.000000   8.400000  30.780000
 60.000000
  0.000000   0.000000   0.000000   0.000000   3.000000   7.400000  29.780000
 50.000000
D =
  1.000000   3.000000   2.000000   0.000000   0.000000
  4.000000   6.000000   6.000000   2.000000   3.000000
  0.000000   0.000000   4.000000   5.000000   6.000000
```

Figure 3.4: The output from executing Program 3.2.

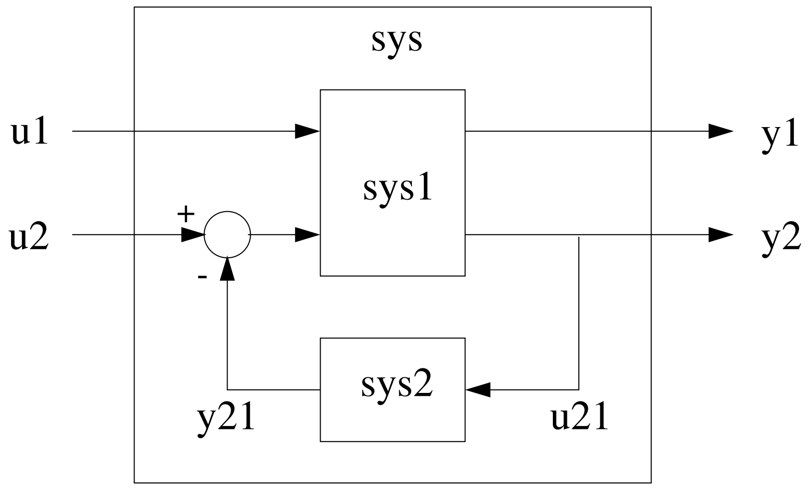


Figure 3.5: Feedback connection of a MIMO control system.

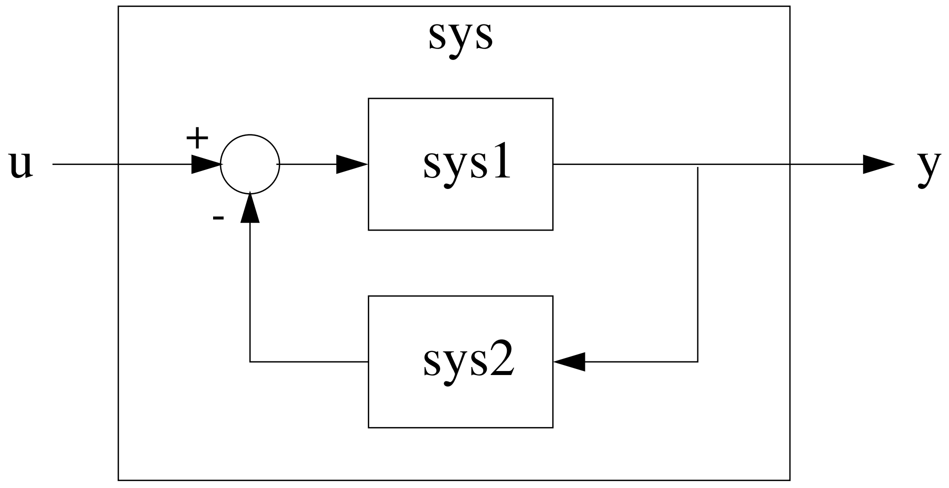


Figure 3.6: Feedback connection of an SISO control system.

```
#include <control.h>

#define NUMX1 2        // number of states of system 1
#define NUMU1 2        // number of inputs of system 1
#define NUMY1 2        // number of outputs of system 1
#define NUMX2 2        // number of states of system 2
#define NUMU2 1        // number of inputs of system 2
#define NUMY2 1        // number of outputs of system 2

int main() {
    array double A1[NUMX1][NUMX1] = {{-2, -3},
                                     {1,   0}},
                 B1[NUMX1][NUMU1] = {1, 0, 0, 1},
                 C1[NUMY1][NUMX1] = {1, -5, 2, 3},
                 D1[NUMY1][NUMU1] = {2, 1, 3, 2};
    CControl sys1;
    CControl sys2;
    CControl *sys3;

    sys1.model("ss", A1, B1, C1, D1);

    array double A2[NUMX2][NUMX2] = {-10, 1, 1, 1},
                 B2[NUMX2][NUMU2] = {1, 1},
                 C2[NUMY2][NUMX2] = {-40, 1},
                 D2[NUMY2][NUMU2] = {5};
    sys2.model("ss", A2, B2, C2, D2);

    array int feedin[1] = {2},
              feedout[1] = {2};

    sys3 = sys1.feedback(&sys2, feedin, feedout);
    sys3->printss();
    return 0;
}
```

Program 3.3: Example of MIMO feedback interconnection (`feedback_mimo.ch`).

```
State-space arguments:
A =
 -2.000000  -3.000000   0.000000   0.000000
  0.090909  -1.363636   3.636364  -0.090909
  0.181818   0.272727  -2.727273   0.818182
  0.181818   0.272727   8.272727   0.818182
B =
  1.000000   0.000000
 -1.363636   0.090909
  0.272727   0.181818
  0.272727   0.181818
C =
  0.090909  -6.363636   3.636364  -0.090909
  0.181818   0.272727   7.272727  -0.181818
D =
  0.636364   0.090909
  0.272727   0.181818
```

Figure 3.7: The output from executing Program 3.3.

```
#include <control.h>

#define NUMX1 2        // number of states of system 1
#define NUMU1 1        // number of inputs of system 1
#define NUMY1 1        // number of outputs of system 1
#define NUMX2 2        // number of states of system 2
#define NUMU2 1        // number of inputs of system 2
#define NUMY2 1        // number of outputs of system 2

int main() {
    array double A1[NUMX1][NUMX1] = {{-2, -3},
                                     {1,   0}},
                 B1[NUMX1][NUMU1] = {1, 0},
                 C1[NUMY1][NUMX1] = {1, -5},
                 D1[NUMY1][NUMU1] = {2};
    CControl sys1;
    CControl sys2;
    CControl *sys3;

    sys1.model("ss", A1, B1, C1, D1);

    array double A2[NUMX2][NUMX2] = {-10, 1, 1, 1},
                 B2[NUMX2][NUMU2] = {1, 1},
                 C2[NUMY2][NUMX2] = {-40, 1},
                 D2[NUMY2][NUMU2] = {5};
    sys2.model("ss", A2, B2, C2, D2);

    sys3 = sys1.feedback(&sys2); /* array int feedin[1] = {1}, feedout[1] = {1}
                                   sys3 = sys1.feedback(&sys2, feedin, feedout);
                                */
    sys3->printss();

    return 0;
}
```

Program 3.4: Example of SISO feedback interconnection (`feedback_siso.ch`).

```
State-space arguments:
A =
 -2.454545  -0.727273   3.636364  -0.090909
  1.000000   0.000000   0.000000   0.000000
  0.090909  -0.454545  -2.727273   0.818182
  0.090909  -0.454545   8.272727   0.818182
B =
  0.090909
  0.000000
  0.181818
  0.181818
C =
  0.090909  -0.454545   7.272727  -0.181818
D =
  0.181818
```

Figure 3.8: The output from executing Program 3.4.

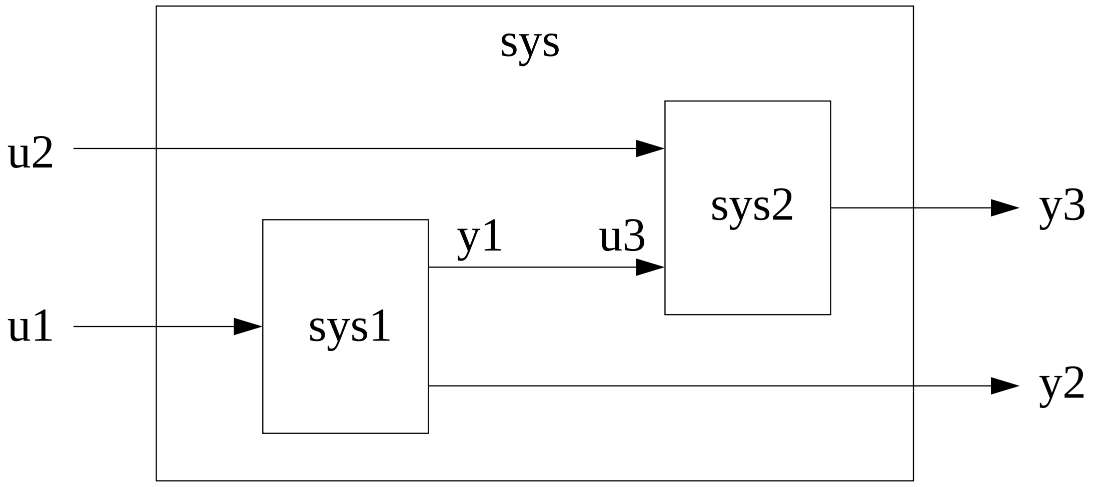


Figure 3.9: Modeling of block diagram interconnections.

is connected to the second input of $sys2$. There are three inputs u1, u2, and u3 for two subsystems $sys1$ and $sys2$. The input u1 of $sys1$ and the first input u2 of $sys2$ are the inputs of sys. This is reflected by the value $\{1, 2\}$ of array $input$ with two elements. There are also three outputs y1, y2, and y3 for these two subsystems. The second output y2 of $sys1$ and the output y3 of $sys2$ are the outputs of sys. Therefor, array $output$ contains also two elements with values of 2 and 3 corresponding to y2 and y3, respectively. To build the system sys, member functions **CControl::append**() and **CControl::connect**() can be called as follows.

```
array int q[1][2] = {3, 1};
array int input[2] = {1, 2};
array int output[2] = {2, 3};
sys = sys1.append(sys2);
sysc = sys->connect(q, input, output);
```

Member function **append**() group two LTI models $sys1$ and $sys2$ by appending their inputs and outputs. It returns a block-diagonal, unconnected LTI model sys. The array q contains input and output numbers to indicate how the blocks on the diagram are connected. Each row of array q specifies a summing input from output of the unconnected system. In this case, the summing input is simply input u3 with output from y1. The first element of each row is the number of summing input which is u3 and the subsequent elements are output numbers which is y1 to sum up and form this summing input. There is only one summing input and this summing input gets its input only from one output. As a result, the array q is a one dimensional array with two elements. Its values of 3 and 1 corresponding to u3 and y1, respectively. The arrays $input$ and $output$ specify which inputs and outputs of the unconnected system are used as external inputs and outputs of the connected system. Program 3.5 constructs a state-space model $sysc$ for the above system. The system matrices of the connected system are shown in Figure 3.10.

The member function **CControl::minreal**() eliminates uncontrollable or unobservable states of a system, the resulting system has the minimal order and the same response characteristics as the original system. Details and application examples of these member functions can be found in Appendix A.

3.2 Controllability and Observability

The controllability and observability are important structural properties of a control system. If there exists an input array $U(t)$ that will take the states of the system from any initial states X_0 to any desired final states

```
// connect.ch
// example of connect

#include <control.h>

#define NUMX1 4    // number of states of system 1
#define NUMU1 1    // number of inputs of system 1
#define NUMY1 2    // number of outputs of system 1
#define NUMX2 4    // number of states of system 2
#define NUMU2 2    // number of inputs of system 2
#define NUMY2 1    // number of outputs of system 2

int main() {
    array double A1[NUMX1][NUMX1] = {{-4.12, -17.4, -30.8, -60},
                                     {1,     0,     0,     0},
                                     {0,     1,     0,     0},
                                     {0,     0,     1,     0}},
                 B1[NUMX1][NUMU1] = {1, 0, 0, 0},
                 C1[NUMY1][NUMX1] = {4, 8.4, 30.78, 60,
                                     3, 7.4, 29.78, 50},
                 D1[NUMY1][NUMU1] = {1, 2};
    array double A2[NUMX2][NUMX2] = {{-4.12, -17.4, -30.8, -60},
                                     {1,     0,     0,     0},
                                     {0,     1,     0,     0},
                                     {0,     0,     1,     0}},
                 B2[NUMX2][NUMU2] = {1, 0,
                                     2, 0,
                                     0, 1,
                                     0, 1},
                 C2[NUMY2][NUMX2] = {4, 8.4, 30.78, 60},
                 D2[NUMY2][NUMU2] = {1, 2};

    array int q[1][2] = {3, 1};
    array int input[2] = {1, 2},
              output[2] = {2, 3};
    CControl sys1;
    CControl sys2;
    CControl *sys, *sysc;

    sys1.model("ss", A1, B1, C1, D1);
    sys2.model("ss", A2, B2, C2, D2);
    sys = sys1.append(&sys2);
    sysc = sys->connect(q, input, output);
    sysc->printss();

    return 0;
}
```

Program 3.5: Example of using connect function (`connect.ch`).

```
State-space arguments:
A =
 -4.120000 -17.400000 -30.800000 -60.000000   0.000000   0.000000   0.000000
  0.000000
  1.000000   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000
  0.000000
  0.000000   1.000000   0.000000   0.000000   0.000000   0.000000   0.000000
  0.000000
  0.000000   0.000000   1.000000   0.000000   0.000000   0.000000   0.000000
  0.000000
  0.000000   0.000000   0.000000   0.000000  -4.120000 -17.400000 -30.800000
-60.000000
  0.000000   0.000000   0.000000   0.000000   1.000000   0.000000   0.000000
  0.000000
  4.000000   8.400000  30.780000  60.000000   0.000000   1.000000   0.000000
  0.000000
  4.000000   8.400000  30.780000  60.000000   0.000000   0.000000   1.000000
  0.000000
B =
  1.000000   0.000000
  0.000000   0.000000
  0.000000   0.000000
  0.000000   0.000000
  0.000000   1.000000
  0.000000   2.000000
  1.000000   0.000000
  1.000000   0.000000
C =
  3.000000   7.400000  29.780000  50.000000   0.000000   0.000000   0.000000
  0.000000
  8.000000  16.800000  61.560000 120.000000   4.000000   8.400000  30.780000
 60.000000
D =
  2.000000   0.000000
  2.000000   1.000000
```

Figure 3.10: The output from executing Program 3.5.

```
// ctrb.ch
// example of calculating the controllability matrix
#include <control.h>

#define NUMX 2    // number of states
#define NUMU 1    // number of inputs

int main() {
    int num;
    array double A[NUMX][NUMX] = {{-2, -9},
                                  {1,  0}},
                 B[NUMX][NUMU] = {1, 0};
    array double co[NUMX][NUMX*NUMU];
    CControl sys;

    sys.model("ss", A, B, NULL, NULL);
    sys.ctrb(co);
    num = rank(co);
    if (num == min(NUMX, NUMX*NUMU))
      printf("The system is controllable.\n");
    else
      printf("The system is not controllable.\n");
    printf("co = %f\n", co);

    return 0;
}
```

Program 3.6: Example of calculating controllability matrix(`ctrb.ch`).

```
The system is controllable.
co = 1.000000 -2.000000
0.000000 1.000000
```

Figure 3.11: The output from executing Program 3.6.

X_f in a finite time interval, the system is controllable. Normally, the controllability matrix can be easily used to determine the controllability. For n_xth order system, if the controllability matrix which is defined as

$$\begin{bmatrix} B & AB & A^2B & \dots & A^{(n-1)}B \end{bmatrix}$$

has full rank n_x, the system is controllable. In Ch Control System Toolkit, the member function **CControl::ctrb**() calculates the controllability matrix of the system. For example, assume that system sys has nx states and nu inputs, the controllability matrix co can be computed as follows

```
    array double co[nx][nx*nu];
    sys.ctrb(co);
```

Program 3.6 is an example of calculating controllability matrix and determine its rank by numerical function **rank**(). Since only system matrix A and input matrix B are involved in the calculation, the other two matrices can be omitted in the member function **CControl::model**().

Table 3.2: Member functions computing controllability/observability matrix.

Member functions	**Description**
CControl::ctrb()	compute controllability matrix of systems
CControl::ctrbf()	compute controllability staircase form
CControl::obsv()	compute observability matrix of systems
CControl::obsvf()	compute observability staircase form
CControl::gram()	controllability and observability gramians

Similar, for any initial states X_0, if there is a finite time t_f such that X_0 can be determined from $U(t)$ and $Y(t)$ for $0 \leq t \leq t_f$, the system is observable. For n_xth order system, if the observability matrix which is defined as

$$\left[\begin{array}{c} C \\ CA \\ CA^2 \\ \ldots \\ CA^{(n-1)} \end{array} \right]$$

has full rank n_x, the system is observable. In Ch Control System Toolkit, the member function **CControl::obsv**() calculates the observability matrix of the system. For example, assume that system sys has nx states and ny outputs, the observability matrix ob can be calculated as follows

```
    array double ob[nx*ny][nx];
    sys.obsv(ob);
```

Program 3.7 is an example of calculating the observability matrix and its rank. Since only system matrix A and output matrix C are involved in the calculation, the other two matrices can be omitted in the member function **CControl::model**().

If the controllability matrix has rank $r < nx$, then there exists a similarity transformation T which can change system (A, B, C) into a staircase form $(\bar{A}, \bar{B}, \bar{C})$, in which the uncontrollable modes are in the upper left corner, such as

$$\bar{A} = TAT^T, \bar{B} = TB, \bar{C} = CT^T$$

and

$$\bar{A} = \left[\begin{array}{cc} A_{uc} & 0 \\ A_{21} & A_c \end{array} \right], \bar{B} = \left[\begin{array}{c} 0 \\ B_c \end{array} \right], \bar{C} = \left[\begin{array}{cc} C_{uc} & C_c \end{array} \right]$$

where (A_c, B_c) is controllable, and $C_c(sI - A_c)^{-1}B_c = C(sI - A)^{-1}B$. The member function **CControl::ctrbf**() can do this transformation. Similar, the member function **CControl::obsvf**() can compute the observability staircase form of a system. Details of these member functions can be found in Appendix A.

3.3 Analysis in Time Domain

The time responses reveal the time-domain transient behavior of LTI models for particular classes of inputs and disturbances. The user can determine such system characteristics as rise time, setting time, overshoot and peak time from the time response. Ch Control System Toolkit provides a set of member functions to compute time responses to arbitrary inputs, as well as initial condition response. These functions includes **CControl::step**() for calculating step response, **CControl::stepinfo**() for obtaining characteristic information for step response, **CControl::impulse**() for impulse response, **CControl::lsim**() for response to an arbitrary input, and **CControl::initial**() for response based on the initial condition as shown in Table 3.3.

```
// obsv.ch
// example of calculating observablity matrix

#include <control.h>

#define NUMX 2        // number of states
#define NUMY 1        // number of outputs

int main() {
    int num;
    array double A[NUMX][NUMX] = {{-2, -9},
                                  {1,  0}},
                 C[NUMY][NUMX] = {1, 3};
    array double ob[NUMX*NUMY][NUMX];
    CControl sys;

    sys.model("ss", A, NULL, C, NULL);
    sys.obsv(ob);
    num = rank(ob);
    if(num == min(NUMX*NUMY, NUMX))
      printf("The system is observable.\n");
    else
      printf("The system is not observable.\n");
    printf("ob = %f\n", ob);
    return 0;
}
```

Program 3.7: Example of calculating observability matrix(`obsv.ch`).

```
The system is observable.
ob = 1.000000 3.000000
1.000000 -9.000000
```

Figure 3.12: The output from executing Program 3.7.

All these functions can plot results by using Ch **CPlot** class. For example, the syntax of function **CControl::step**() is

```
sys.step(plot, yout, tout, xout, tf);
```

The first argument $plot$ is an object of Ch **CPlot** class. Arrays $yout$, $tout$ and $xout$ are output response, the time vector used for simulation, and the state trajectories, respectively. The last argument tf specifies the final time for simulation. Please refer to Appendix A for syntax of other functions that are listed in Table 3.3. Program 3.8 is an example of plotting time domain response of system sys by calling these functions. Customizing plot using the class plotting **CPlot** will be described in Chapter 5. Figures 3.13 - 3.16 are plots from executing Program 3.8.

Table 3.3: Member functions computing time responses.

Member functions	**Description**
CControl::step()	step response
CControl::impulse()	impulse response
CControl::lsim()	response to arbitrary inputs
CControl::initial()	initial condition response

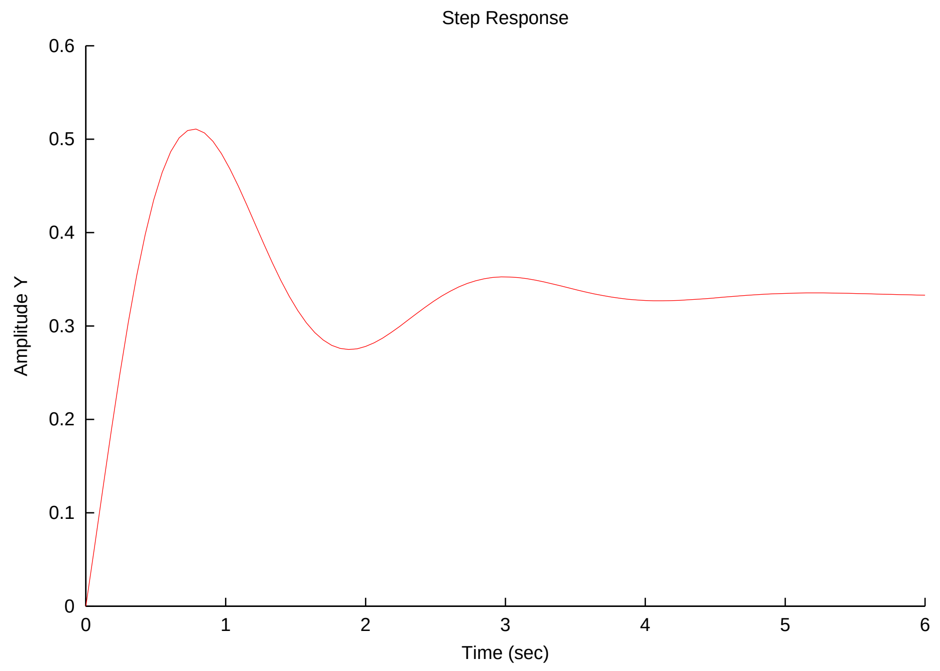


Figure 3.13: Step response.

```
// timedom.ch
// example of time response

#include <control.h>

#define NUMX 2        // number of states
#define NUMU 1        // number of inputs
#define NUMY 1        // number of outputs

int main() {
    int i, j, k = 0, flag;
    double iu[100];
    array double A[NUMX][NUMX] = {{-2, -9},
                                  {1,  0}},
                 B[NUMX][NUMU] = {1, 0},
                 C[NUMY][NUMX] = {1, 3},
                 D[NUMY][NUMU] = {0};
    array double x0[2] = {1.0, 0.0};
    CPlot plot1, plot2, plot3, plot4;
    CControl sys;

    sys.model("ss", A, B, C, D);

    // step
    sys.step(&plot1, NULL, NULL, NULL, 6);

    // impulse
    sys.impulse(&plot2, NULL, NULL, NULL, 6);

    // square wave
    flag = 1;  // generate the square wave as input
    for (i = 0; i < 5; i++) {
      flag = 1-flag;
      for (j = 0; j < 20; j++) {
        iu[k] = flag;
        k++;
      }
    }
    sys.lsim(&plot3, NULL, NULL, NULL, iu, 6);

    // initial condition response
    sys.initial(&plot4, NULL, NULL, NULL, x0);

    return 0;
}
```

Program 3.8: Example of time domain response plotting (`timedom.ch`).

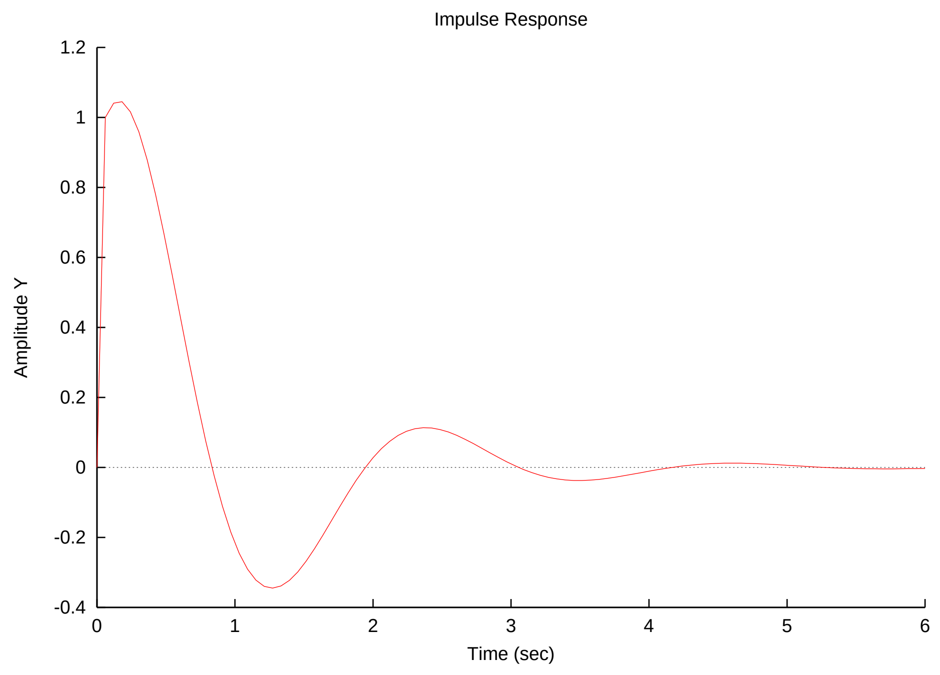


Figure 3.14: Impulse response.

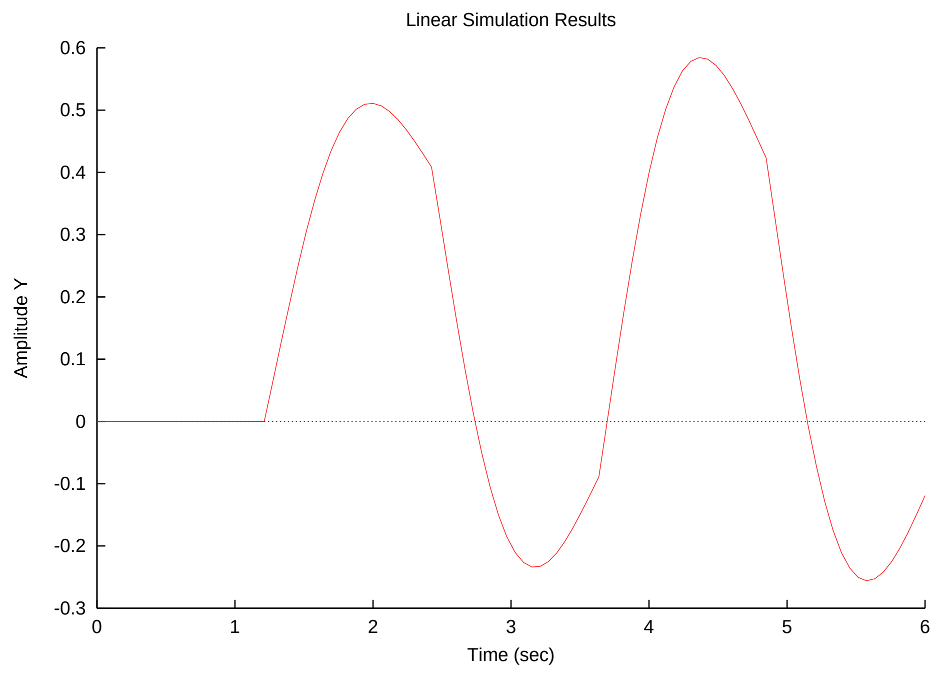


Figure 3.15: Response to square wave input.

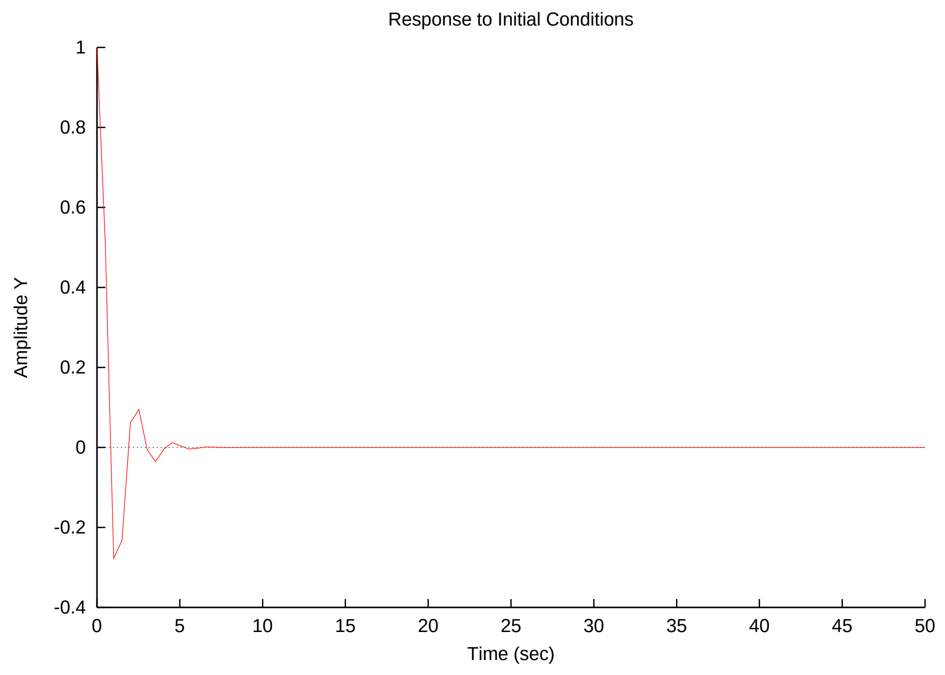


Figure 3.16: Initial condition response.

3.4 Analysis in Frequency Domain

The frequency response is a linear system's response to sinusoidal input. It not only can help the user understand how a system response to a sinusoidal input, but also help determine the stability of a closed -loop system. Ch Control System Toolkit provides member functions **CControl::bode**(), **CControl::nichols**(), and **CControl::nyquist**() to generate commonly used frequency response plots of Bode, Nichols and Nyquist plots as shown in Table 3.3, respectively. These plots are displayed by Ch **CPlot** class. For example, the syntax of function **CControl::bode**() for a Bode plot is as follows.

```
sys.bode(plot, mag, phase, wout, wmin, wmax);
```

The first argument $plot$ is an object of Ch **CPlot** class. Arrays mag and $phase$ contain the returned magnitude (in decibels) and phase (in degrees) of the frequency response at the frequencies $wout$ (in rad/sec). The last two arguments, $wmin$ and $wmax$, explicitly specify frequency range to be used for the plot. Please refer to Appendix A for syntax of other functions that are listed in Table 3.4. Program 3.9 is an example of plotting the frequency domain response of system sys by calling member functions **CControl::bode**(), **CControl::nichols**(), **CControl::nyquist**(). Figures 3.17 - 3.19 are plots from executing Program 3.9.

```
// freqdom.ch
// example of frequency response

#include <control.h>

#define NUMX 2        // number of states
#define NUMU 1        // number of inputs
#define NUMY 1        // number of outputs

int main() {
    array double A[NUMX][NUMX] = {{-2, -9},
                                   {1,  0}},
                 B[NUMX][NUMU] = {1, 0},
                 C[NUMY][NUMX] = {1, 3},
                 D[NUMY][NUMU] = {0};
    CPlot plot5, plot6, plot7;
    CControl sys;

    sys.model("ss", A, B, C, D);
    sys.bode(&plot5, NULL, NULL, NULL);
    sys.nichols(&plot6, NULL, NULL, NULL);
    sys.nyquist(&plot7, NULL, NULL, NULL);

    return 0;
}
```

Program 3.9: Example of frequency domain response-plotting (`freqdom.ch`).

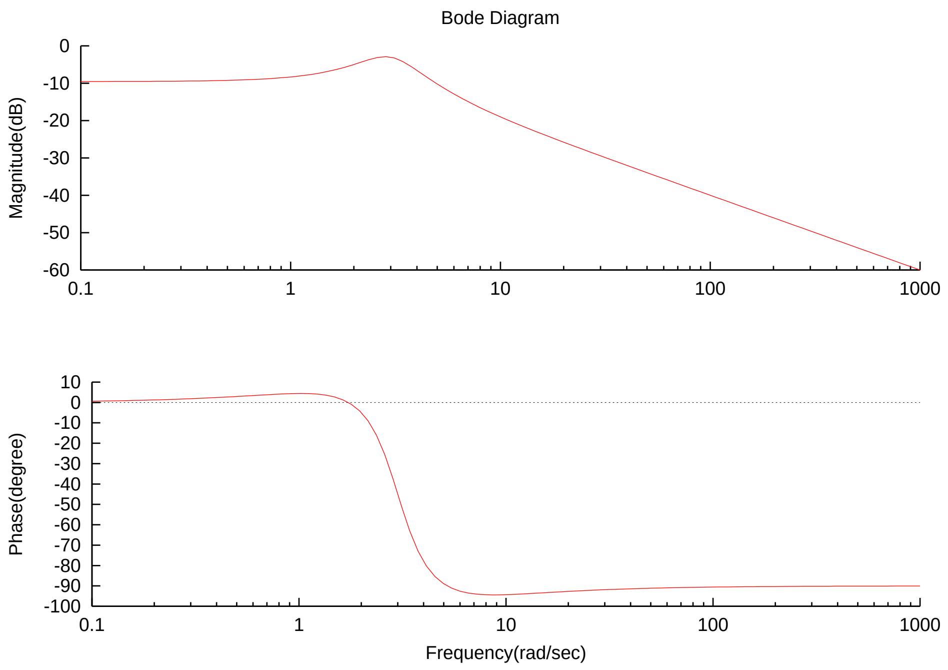


Figure 3.17: Bode plot.

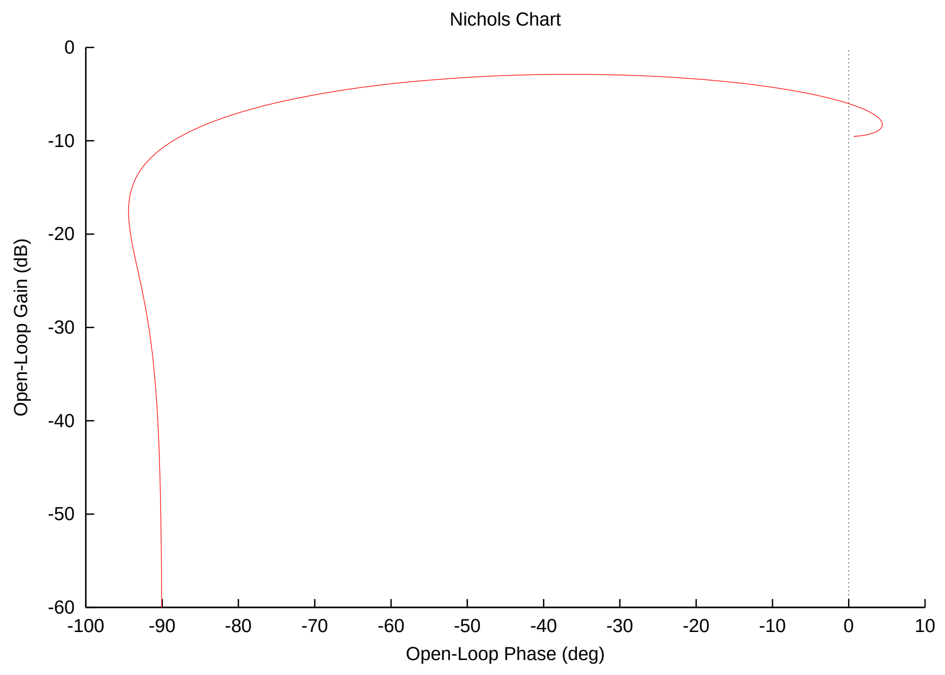


Figure 3.18: Nichols plot.

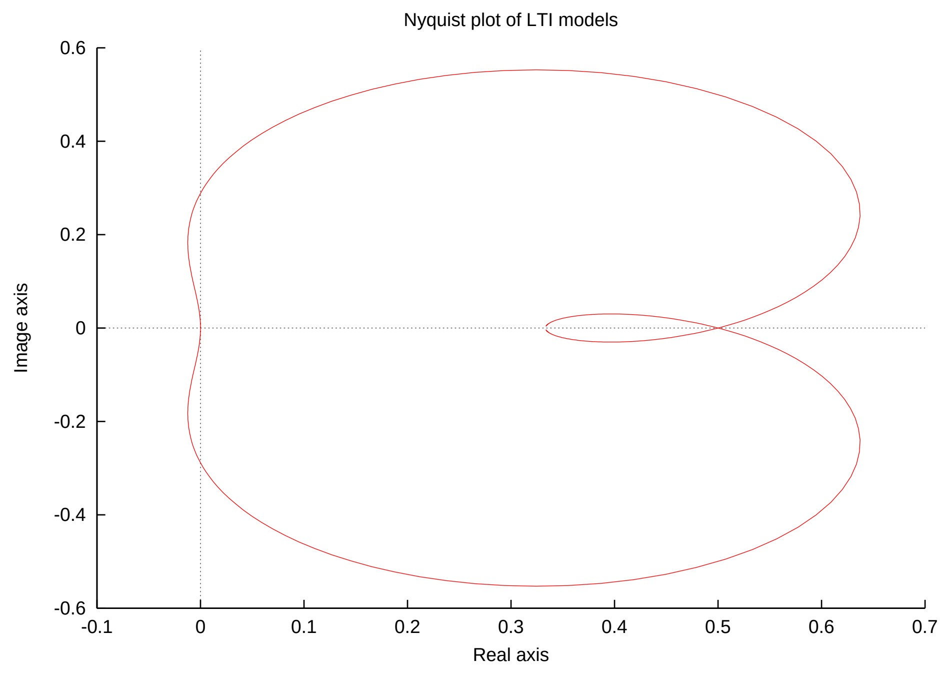


Figure 3.19: Nyquist plot.

Besides these three commonly used member functions, other member functions listed in Table 3.4 can be used to analyze model dynamics such as the gain margin, phase margin, DC gain, bandwidth, disturbance rejection, and stability. Please refer to Appendix A for syntaxes of these functions.

Table 3.4: Member functions computing frequency responses.

Member functions	**Description**
CControl::bandwidth()	bandwidth of SISO models
CControl::bode()	Bode plot
CControl::bodemag()	plot Bode magnitude
CControl::damp()	natural frequency and damping
CControl::dcgain()	low-frequency gain
CControl::nichols()	Nichols plot
CControl::nyquist()	Nyquist plot
CControl::margin()	gain and phase margins

Chapter 4

System Design Using Ch Control System Toolkit

In this chapter, how to design and optimize a control system using the Ch Control System Toolkit will be described. Particularly, the root-locus design method, pole placement, Linear-quadratic regulator design, and solution of Lyapunov equations will be presented. The member functions in Ch Control System Toolkit for design and optimization of control systems are listed in Tables 4.1 - 4.4.

4.1 Root Locus Design

Table 4.1: Member functions for root locus design.

Member functions	Description
CControl::rlocus()	compute and plot the root locus of SISO systems
CControl::pzmap()	compute the pole-zero map of an LTI model

Normally, roots of the characteristic equation, which actually are the closed-loop poles, will change as one of the system's parameters varies over a continuous range with default values. The paths of the changing roots make a plot called the root locus. The root locus is most commonly used to study the effect of loop gain variations.

Assume that the control system shown in Fig 4.1 consists of the plant $p(s)$ and the scalar gain k. The closed-loop poles are the roots of following equation

$$q(s) = 1 + kp(s)$$

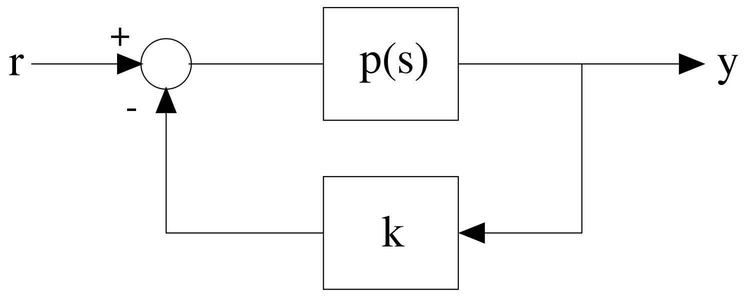


Figure 4.1: System with feedback gain K.

```
// rlocus.ch
// example of root locus

#include <control.h>

int main() {
  array double num[2] = {1, 3}, den[3] = {1, 2, 9};
  CControl sys;
  int nx;
  CPlot plot8;

  sys.model("tf", num, den);
  nx = sys.size('x');

  array double complex r[nx][100];
  array double kout[100];
  sys.rlocus(&plot8, r, kout);

  return 0;
}
```

Program 4.1: Example of root locus plotting (`rlocus.ch`).

The root locus is the trajectories of the roots when the feedback gain k varies from 0 to positive infinity. The member function to plot root locus is **CControl::rlocus**() whose syntax is shown as follows,

```
sys.rlocus(plot, r, kout, k);
```

where k is an array which contains user-selected gains for which the roots are calculated. If k is omitted, the function will select gains adaptively, and return these gains in $kout$. The computational array r is used to contain the complex root locations for gains in $kout$. Program 4.1 is an example of root locus. The plot of the output for Program 4.1 is shown in Figure 4.2. The zero is indicated by a small circle, whereas a pole is represented by a cross in a root locus diagram.

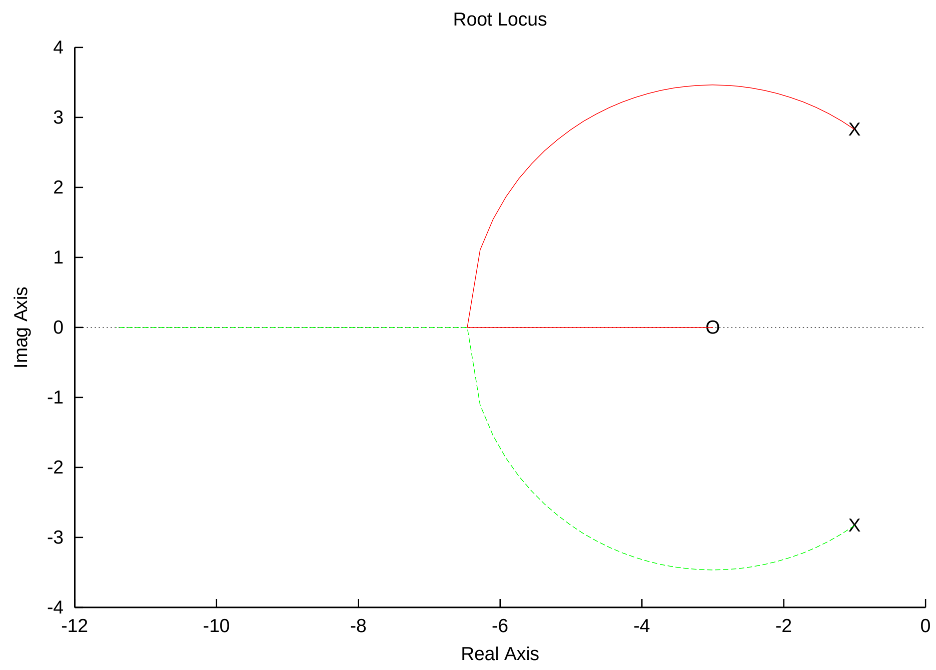


Figure 4.2: Root locus.

4.2 Pole Placement

The closed-loop response characteristics, such as rise time, settling time, and transient oscillatins, depend on the set of closed-loop poles of the system. For a system presented by the state-space model

$$\dot{X} = AX + BU$$
$$Y = CX + DU$$

with the linear state feedback regulator control law defined as $U = -KX$, the closed loop poles are eigenvalues of matrix $(A - BK)$. To make the system more stable and have the desired time response characteristics, the user can first select desired pole locations, and then find the gain matrix K which moves these poles to the desired locations. This technique is known as pole placement. The member functions which can be used in pole placement are listed in Table 4.2. The member function **CControl::pole**() can compute the poles of an LTI system. For example, statements

```
int np = sys.size('p');
array double complex p[np];
sys.pole(p);
```

retrieve poles of sys in variable p.

The member function **CControl::acker**() is typically used for pole placement of SISO system. The syntax is

```
sys.acker(k, p);
```

where p is an array of desired poles and k is the corresponding gain matrix which makes the close loop poles match the matrix p. Program 4.2 is an example of pole placement of second order SISO system using

```
// acker.ch
#include <control.h>

#define NUMX 2    // number of states
#define NUMU 1    // number of inputs
#define NUMY 1    // number of outputs

int main() {
     array double A[NUMX][NUMX] = {{-2, -9},
                                   {1,  0}},
                  B[NUMX][NUMU] = {1, 0},
                  C[NUMY][NUMX] = {1, 3},
                  D[NUMY][NUMU] = {0};
     array double k[NUMX];
     array double complex p[NUMX] = {-1, -2};
     int i;
     CControl sys;

     sys.model("ss", A, B, C, NULL);
     sys.acker(k, p);
     for (i = 0; i < NUMX; i ++) {  // print out the result
       printf("k[%d] = %f\n", i, k[i]);
     }

     return 0;
}
```

Program 4.2: Example of pole placement using function acker() (`acker.ch`).

```
k[0] = 1.000000
k[1] = -7.000000
```

Figure 4.3: The output from executing Program 4.2.

```
// place.ch
#include <control.h>

#define NUMX 4    // number of states
#define NUMU 2    // number of inputs
#define NUMY 2    // number of outputs

int main() {

     array double A[NUMX][NUMX] = {{-2,  -9,  0,  0},
                                   {1,  0,  0,  0},
                                   {0, 0, -2, -9},
                                   {0, 0, 1, 0}},
                  B[NUMX][NUMU] = {1, 0,
                                   0, 0,
                                   0, 1,
                                   0, 0},
                  C[NUMY][NUMX] = {1, 3, 0, 0,
                                   0, 0, 1, 3},
                  D[NUMY][NUMU] = {0, 0, 0, 0};
     array double complex k[NUMU][NUMX];
     array double complex p[NUMX] = {-1, -2, -1, -2};
     CControl sys;

     sys.model("ss", A, B, C, D);
     sys.place(k, p);

     // print out the result
     printf("k = \n%f\n", k);
     return 0;
}
```

Program 4.3: Example of pole placement for MIMO systems using function place() (`place.ch`).

CControl::acker(). In this example, the desired poles are -2 and -1. The corresponding gain matrix is shown in Figure 4.3

For MIMO systems, Ch toolkit provides another member function **CControl::place**(). It has a similar syntax to **CControl::acker**() as shown below.

```
sys.place(k, p);
```

where p is an array of desired poles and k is the corresponding gain matrix which makes the closed-loop poles match the matrix p. Program 4.3 is an example of pole placement of a second order two-input-two-output system using **CControl::place**(). In this example, the desired poles are -2, -1, -2, and -1. The corresponding gain matrix is shown in Figure 4.4.

Table 4.2: Member functions for pole placement.

Member functions	**Description**
CControl::pole()	compute poles of LTI systems
CControl::acker()	pole placement gain selection using Ackermann's formula for SI systems
CControl::place()	pole placement design

```
k =
complex(1.000000,0.000000) complex(-7.000000,0.000000) 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(-7.000000,0.000000)
```

Figure 4.4: The output from executing Program 4.3.

4.3 LQG Design

Given the state dynamics equation

$$\begin{aligned}\dot{\mathbf{x}} &= A\mathbf{x} + B\mathbf{u} \\ \mathbf{y} &= C\mathbf{x} + D\mathbf{u}\end{aligned}$$

and a cost function

$$J(\mathbf{u}) = \int \mathbf{x}^{\mathbf{T}} Q\mathbf{x} + \mathbf{u}^{\mathbf{T}} R\mathbf{u} + 2\mathbf{x}^{\mathbf{T}} N\mathbf{u}\,\mathrm{d}t,$$

the purpose of Linear Quadratic Gaussian (LQG) control scheme is to find an optimal gain matrix K, so that the state-feedback law

$$\mathbf{u} = -K\mathbf{x}$$

minimizes the above cost function $J(\mathbf{u})$. The gain K is called the LQ optimal gain. To obtain the LQ optimal gain, the user need to solve an algebraic Riccati equation shown below

$$SA + A^T S - (SB + N)R^{-1}(B^T S + N^T) + Q = 0$$

Then, K can be derived from the solution S of the above equation by

$$K = R^{-1}(B^T S + N^T)$$

Ch Control System Toolkit provides two functions, **CControl::lqr**() and **CControl::dlqr**(), for designing linear-quadratic (LQ) state-feedback regulator for continuous-time and discrete-time plants, respectively. The syntax of **CControl::lqr**() is shown as follows,

```
    sys.lqr(k, s, e, Q, R, N);
```

where k is the returned optimal gain, s is the solution of the algebraic Riccati equation, and e is the closed-loop eigenvalues of matrix($A - BK$). Q, R, and N are weighting matrices of the cost function $J(u)$. If the matrix N is set to zero, it can be absent. Program 4.4 is an example of LQ regulator design for continuous-time systems using **CControl::lqr**(). In this example, the weighting matrix N is absent. The results of executing program `lqr.ch` is shown in Figure 4.5.

Another function **CControl::dlqr**() is for discrete-time systems. It has the same syntax as that of function **CControl::lqr**(),

```
    sys.dlqr(k, s, e, Q, R, N);
```

As an example, for a discrete-time system with sampling time of 0.1 and following state space equation,

$$\dot{\mathbf{x}} = A\mathbf{x} + B\mathbf{u}$$

```
// lqr.ch

#include <control.h>

#define NUMX 2    // number of states
#define NUMU 1    // number of inputs

int main() {
    array double A[NUMX][NUMX] = {{-1, 0},
                                  {0,  -2}},
                 B[NUMX][NUMU] = {1, 1};
    CPlot plot;
    array double k[1][2], s[2][2];
    array double complex e[2];
    array double Q[2][2] = {1, 0, 0, 0};
    array double R[1][1] = {1};
    CControl sys;

    sys.model("ss", A, B, NULL, NULL);
    sys.lqr(k, s, e, Q, R);
    printf("k = %f\ns = %f\ne = %f\n", k, s, e);

    return 0;
}
```

Program 4.4: Example of LQ regulator design using function lqr() (`lqr.ch`).

```
k = 0.414214 0.000000

s = 0.414214 -0.000000
-0.000000 -0.000000

e = complex(-2.000000,0.000000) complex(-1.414214,0.000000)
```

Figure 4.5: The output from executing Program 4.4.

```
// dlqr.ch

#include <control.h>

#define NUMX 2    // number of states
#define NUMU 1    // number of inputs

int main() {
    array double A[NUMX][NUMX] = {{-1, 0},
                                  {0,  -2}},
                 B[NUMX][NUMU] = {1, 1};
    array double k[NUMU][NUMX], s[NUMX][NUMX];
    array double complex e[NUMX];

    array double Q[NUMX][NUMX] = {1, 0, 0, 0};
    array double R[NUMU][NUMU] = {1};
    CControl sys;

    sys.model("ss", A, B, NULL, NULL, .1);
    sys.dlqr(k, s, e, Q, R);
    printf("k = %f\ns = %f\ne = %f\n", k, s, e);

    return 0;
}
```

Program 4.5: Example of LQ regulator design for discrete-time systems using function dlqr() (`dlqr.ch`).

```
k = 0.309017 -2.427051

s = 4.618034 -7.854102
-7.854102 20.562306

e = complex(-0.381966,0.000000) complex(-0.500000,0.000000)
```

Figure 4.6: The output from executing Program 4.5.

where

$$A = \begin{bmatrix} -1 & 0 \\ 0 & -2 \end{bmatrix}, B = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

program 4.5 designs the LQ regulator for this discrete-time system using **CControl::dlqr**(). In this example, the parameters of the cost function

$$J(\mathbf{u}) = \int \mathbf{x}^{\mathbf{T}} Q \mathbf{x} + \mathbf{u}^{\mathbf{T}} R \mathbf{u} + 2\mathbf{x}^{\mathbf{T}} N \mathbf{u}\, \mathrm{d}t,$$

are as follows.

$$Q = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, R = 1, N = 0$$

The results of executing program `dlqr.ch` is shown in Figure 4.6.

Table 4.3: Member functions for LQG designing.

Member functions	**Description**
CControl::lqr()	Linear-quadratic regulator design for continuous-time systems
CControl::dlqr()	Linear-quadratic regulator design for discrete-time systems

4.4 Lyapunov Equation Solvers

In control engineering, Lyapunov matrix equation is often used to solve problems related to the system stability. The general form of the Lyapunov matrix equation (Sylvester equation) can be represented as follows,

$$AX + XB = -Q$$

where matrices A, B and Q must have compatible dimensions. If the matrix B in the generalized Lyapunov matrix equation is the transpose of the matrix A, the equation is called the special form of the Lyapunov matrix equation

$$AX + XA^T = -Q$$

where A and Q are square matrices of same sizes. The member function **CControl::lyap()** in Ch Control System Toolkit can be used to solve these two forms of continuous-time Lyapunov equations. For a general form, its syntax is

```
sys.lyap(X, A, B, Q);
```

For a special form, the syntax is

```
sys.lyap(X, A, Q);
```

where array X is the solution of the Lyapunov matrix equation, arrays A, B and Q are parameters of the equation. Because the solution of a Lyapunov equation is only dependent on the parameters of A, B and Q, rather than the characteristics of the system sys itself, the user can simply create an empty system. Program 4.6 is an example of solving continuous-time Lyapunov equations using **CControl::lyap**(). In this example, the Lyapunov equation is in the general form. The results of executing program `lyap.ch` is shown in Figure 4.7.

To solve a discrete Lyapunov equation

$$AXA^T - X = -Q,$$

the member function **CControl::dlyap()** can be used. It has the syntax as follows,

```
sys.dlyap(X, A, Q);
```

where array X is the solution of the discrete Lyapunov matrix equation, A and Q are parameters of the equation.

Table 4.4: Member functions for solving Lyapunov equations.

Member functions	**Description**
CControl::lyap()	solve continuous-time Lyapunov equations
CControl::dlyap()	solve discrete-time Lyapunov equations

Besides member functions introduced in this chapter, Ch itself already includes many numerical functions, such as **roots()** and **norm()** for system design. Please refer to *Ch User's Guide* for details of these functions.

```
#include <control.h>

int main() {
    array double complex a[3][3] = {2, 3, 4,
                            5, 6, 7,
                            8, 9, 10};
    array double complex b[3][3] = {2, 4, 5,
                            11, 13, 15,
                            1, 2, 3};
    array double complex c[3][3] = {2.3000, 4.5000, 2,
                            3.4000, 6.7000, 5.0000,
                            2.8700, 9.3200, 1};
    array double complex x[3][3];
    CControl sys;

    sys.lyap(x, a, b, c);
    printf("x = \n%f\n", x);

    return 0;
}
```

Program 4.6: Example of solving continuous-time Lyapunov equations (`lyap.ch`).

```
x =
complex(-2.009236,0.000000) complex(-0.260711,0.000000) complex(4.408764,0.000000)

complex(3.094459,0.000000) complex(-0.368029,0.000000) complex(-5.158256,0.000000)

complex(-1.271845,0.000000) complex(-0.066459,0.000000) complex(1.346946,0.000000)
```

Figure 4.7: The output from executing Program 4.6.

Chapter 5

Customizing Response Plot Properties

5.1 Ch Plot Properties

Ch has a class **CPlot** for high-level creation and manipulation of two and three dimensional plotting. In Ch Control System Toolkit, all member functions with plotting features internally invoke member functions of class **CPlot** to generate 2D/3D plots. To allow the user to customize a plot, an object of class **CPlot** is required to be instantiated in the user's application program. Then, the user can set properties such as title and axis labels by calling member functions of class **CPlot**. Finally, the address (a pointer) of this object needs to be passed as an argument to the member functions with plotting features of Ch Control System Toolkit.

For example, the following statements

```
CPlot plot;
...
sys.step(&plot, NULL, NULL, NULL);
```

can generate a step response plot with default title "Step Response" as shown in Figure 3.13. To display a user-defined title, the function **CPlot::title**() can be used. For example, statements below

```
CPlot plot;
...
plot.title("My Title");
sys.step(&plot, NULL, NULL, NULL);
```

will change the default title to "My Title". Not only can the user change an existing property in the plot, but also modify the default properties of a plot using member functions of class **CPlot** . For example, member function **CPlot::border**() can be used to add the border to a plot and member function **CPlot::outputType**() can be used to change the output type as shown below .

```
CPlot plot;
...
plot.title("My Title");
plot.border(PLOT_BORDER_ALL, PLOT_ON);
plot.outputType(PLOT_OUTPUTTYPE_FILE, "postscript eps color", "fig9.eps");
sys.step(&plot, NULL, NULL, NULL);
```

where the macro **PLOT_BORDER_ALL** indicates that the border is put on all sides of the plot, and **PLOT_ON** enables the drawing of the box around the plot. The output of the program will be exported

```
// plot_border.ch
// example of frequency response plot with border

#include <control.h>

#define NUMX 2    // number of states
#define NUMU 1    // number of inputs
#define NUMY 1    // number of outputs

int main() {
    array double A[NUMX][NUMX] = {{-2, -9},
                                  {1,  0}},
                 B[NUMX][NUMU] = {1, 0},
                 C[NUMY][NUMX] = {1, 3},
                 D[NUMY][NUMU] = {0};
    CPlot plot9;
    CControl sys;

    sys.model("ss", A, B, C, D);
    plot9.title("My Title");
    plot9.border(PLOT_BORDER_ALL, PLOT_ON);
    plot9.outputType(PLOT_OUTPUTTYPE_FILE, "postscript eps color", "fig9.eps");
    sys.step(&plot9, NULL, NULL, NULL, 6);

    return 0;
}
```

Program 5.1: Example of creating transfer function model using Ch Control System Toolkit (`plot_border.ch`).

to an extern color postscript file `fig9.eps`, instead of displayed by default. The example of adding user-defined title and border is shown in Program 5.1 with output displayed in Figure 5.1 exported as a color postscript file. More information on how to customize a plot using member functions of class **CPlot** can be found in *Ch User's Guide.*

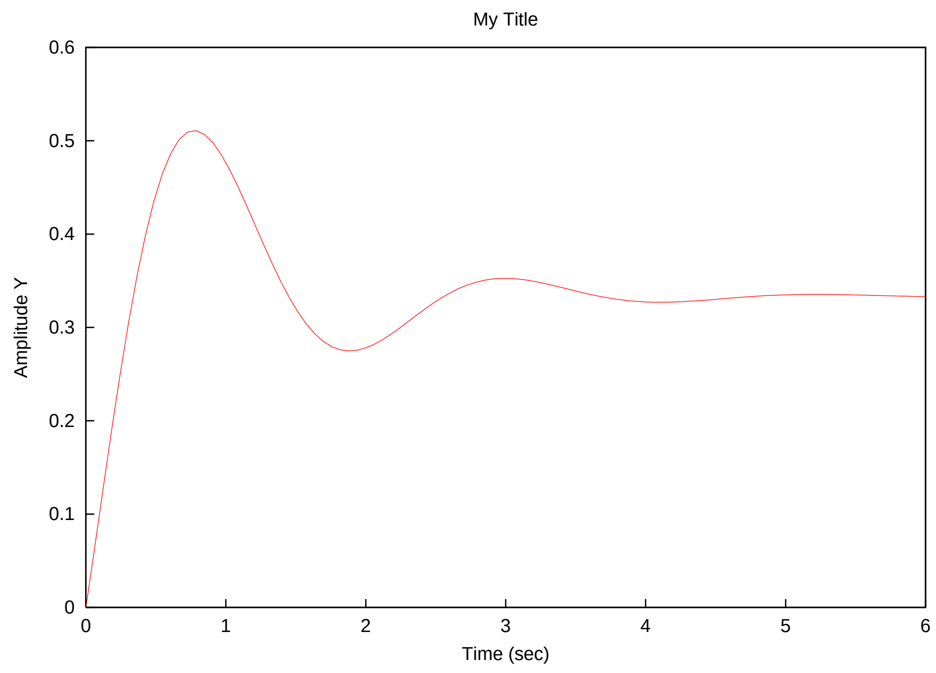


Figure 5.1: Step response plot with user-defined title and border.

Detailed description of each member function of class **CPlot** can be found in the chapter about plotting in *Ch User's Guide*.

5.2 Plotting with Grids

Proper grids can help the user understand some special plots, such as Nyquist plot, Nichols plot, and root locus or pole-zero maps in z-plane and s-plane. In Ch Control System Toolkit, member functions that can add special grids to a plot are listed in Table 5.1.

Table 5.1: Member functions adding grid lines to plots.

Member functions	**Description**
CControl::grid()	add regular/Nyquist grid lines
CControl::ngrid()	add Nichols chart grid lines
CControl::sgrid()	generate an s-plane grid
CControl::zgrid()	generate a z-plane grid

The member function **CControl::grid**() can add grid lines to Nyquist plots, regular plots such as step response plots and Bode plots. Its syntax is shown as follows,

```
sys.grid(flag);
```

where $flag$ is an integer. The function turns on the grid when the argument $flag$ equals 1, and turns off the grid when it equals 0. Figure 5.2 is an example of regular grid which is added to the Bode plot shown in Figure 3.17. The original program without grid lines is shown in Program 3.9.

When the member function **CControl::grid**() is applied to Nyquist plot, the grid lines are circles instead of regular straight lines as illustrated in Figure 5.3. For SISO systems, when $G(j\omega)$ is the complex transfer function, there is a corresponding unity-feedback closed-loop transfer function

$$T(j\omega) = \frac{\mathrm{G}(j\omega)}{1+\mathrm{G}(j\omega)}$$

Assume that $M(\omega)$ and $a(\omega)$ are magnitude and phase of the closed-loop transfer function, we have equations as shown below,

$$M(\omega) = |T(j\omega)|, a(\omega) = \angle T(j\omega)$$

In Nyquist plot, the contours of constant value of $M(\omega)$ and $a(\omega)$ are circles. These circles are referred to as the M and N circles, respectively. Figure 5.3 is an example of adding M and N circles to Nyquist plots shown in Figure 3.19 using member function **CControl::grid**().

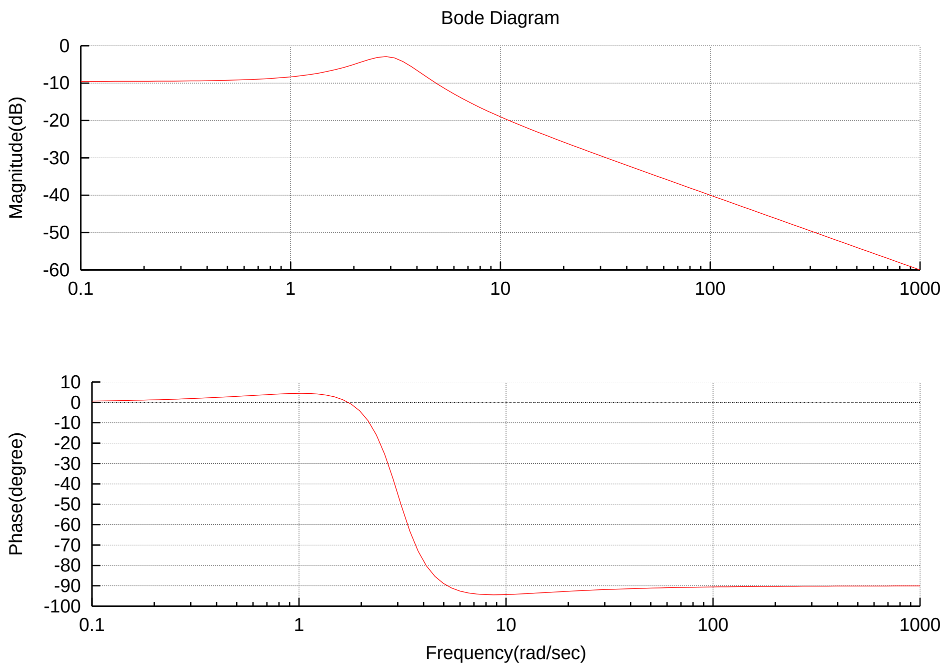


Figure 5.2: Bode plot with regular grid lines.

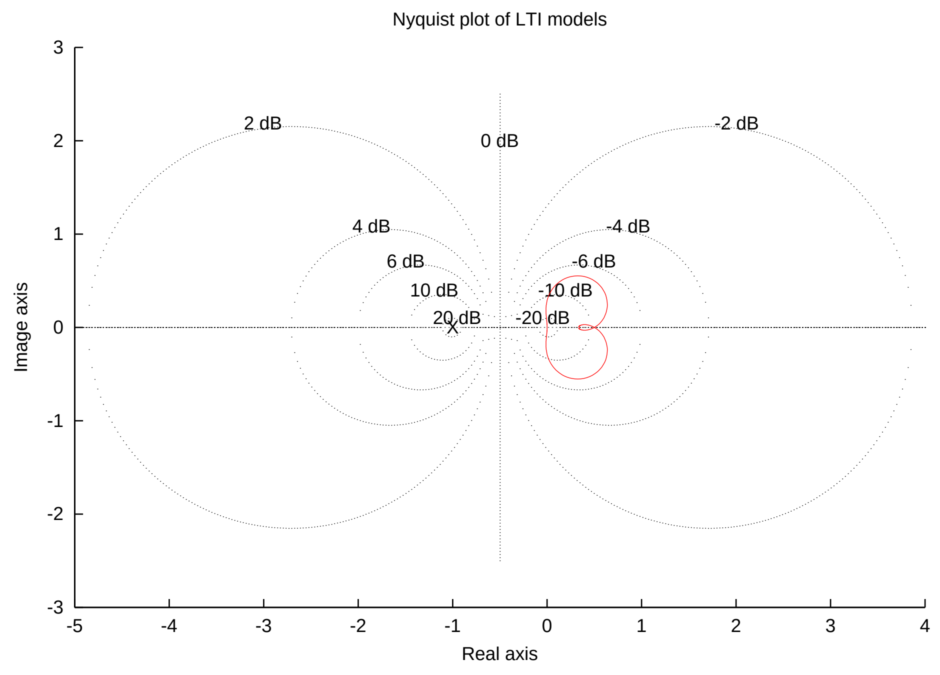


Figure 5.3: Nyquist plot with grid lines.

The member function **CControl::ngrid**() adds grid lines to a Nichols chart. Its syntax is shown as follows,

```
sys.ngrid(flag);
```

where $flag$ is an integer. The function turns on the grid when the argument $flag$ equals 1, and turns off the grid when it equals 0. Similar to a Nyquist plot, a Nichols chart consists of the contours of constant closed-loop magnitude and phase. But these contours are not circles. Figure 5.4 is an example of superimposing Nichols chart grid lines over the Nichols frequency response shown in Figure 3.18, so that the user can determine the bandwidth, gain margin, phase margin, and so on by observing where the Nichols plot crosses some special grid lines.

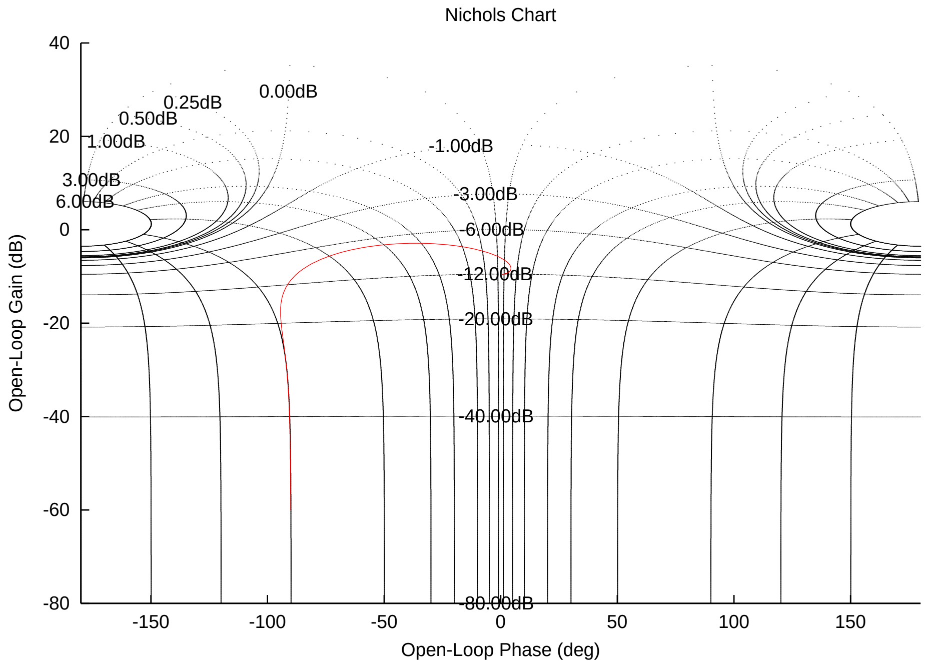


Figure 5.4: Nichols frequency response with Nichols chart grid lines.

The member function **CControl::sgrid**() adds s-plane grid of constant damping factors and natural frequencies to plots of pole-zero map or root locus. The syntax of this member function is as follows.

```
sys.sgrid(flag);
```

If $flag$ equals 1, a grid of constant damping factors from 0 to 1 in steps of 0.1 and natural frequencies from 0 to 10 rad/sec in steps of 1 rad/sec will be added to a plot. If the user needs to replace these default steps, the following syntax can be used,

```
sys.sgrid(flag, z, w);
```

where arrays z and w contain the user-defined damping factors and natural frequencies. Figure 5.5 is the root locus plot shown in Figure 4.2 with s-plane grid lines.

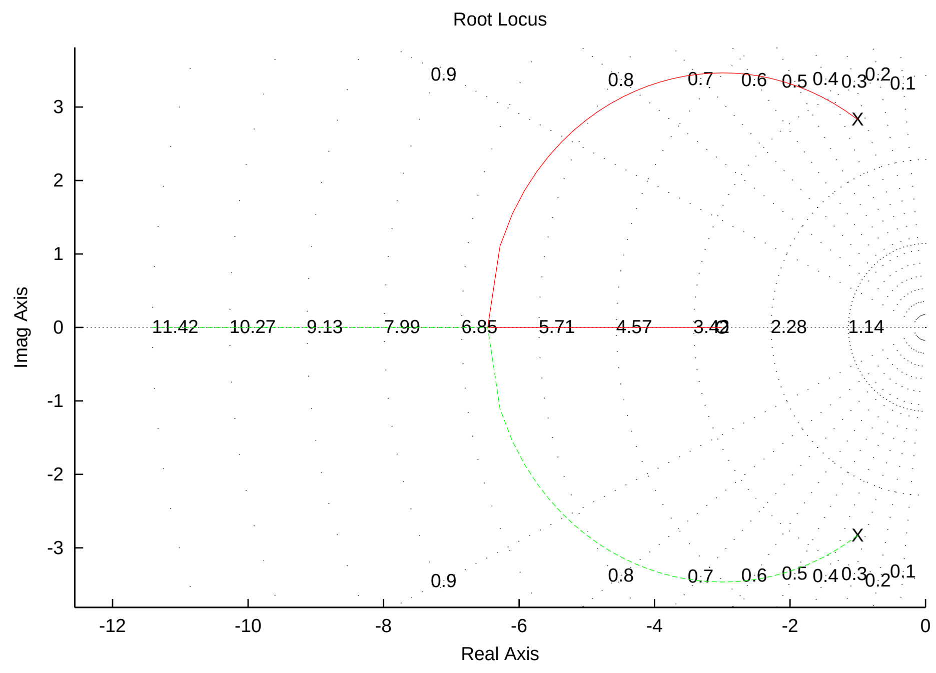


Figure 5.5: Root locus plot with s-plane grid of constant damping factors and natural frequencies.

If the object is a discrete-time system and the root locus is in the z-plane. The member function **CControl::zgrid**() can be used to add z-plane grid of constant damping factors and natural frequencies. The default grid of constant damping factors is from 0 to 1 in steps of 0.1 and natural frequencies from 0 to π in steps of $\pi/10$. The user can replace these default steps by passing extra arguments to the member function. The syntax of this function is shown as follows. The statement

```
sys.zgrid(flag);
```

adds grid to z-plane with default steps, whereas the

```
sys.zgrid(flag, z, w);
```

statement uses the user-defined steps indicated by arrays z and w.

Figure 5.6 illustrates the grid added to a discrete z-plane root locus diagram. The source code is shown in Program 5.2.

```
// rlocus.ch
// example of root locus

#include <control.h>

int main() {
  array double num[3] = {2, -3.5, 1.5}, den[3] = {1, -1.5, 0.9};
  int nx;
  CPlot plot12;
  CControl sys;

  sys.model("tf", num, den, -1);
  nx = sys.size('x');

  array double complex r[nx][100];
  array double kout[100];
  sys.zgrid(1);
  sys.rlocus(&plot12, r, kout);

  return 0;
}
```

Program 5.2: Example of root locus plot with z-plane grid of constant damping factors and natural frequencies. (`plot_zgrid.ch`).

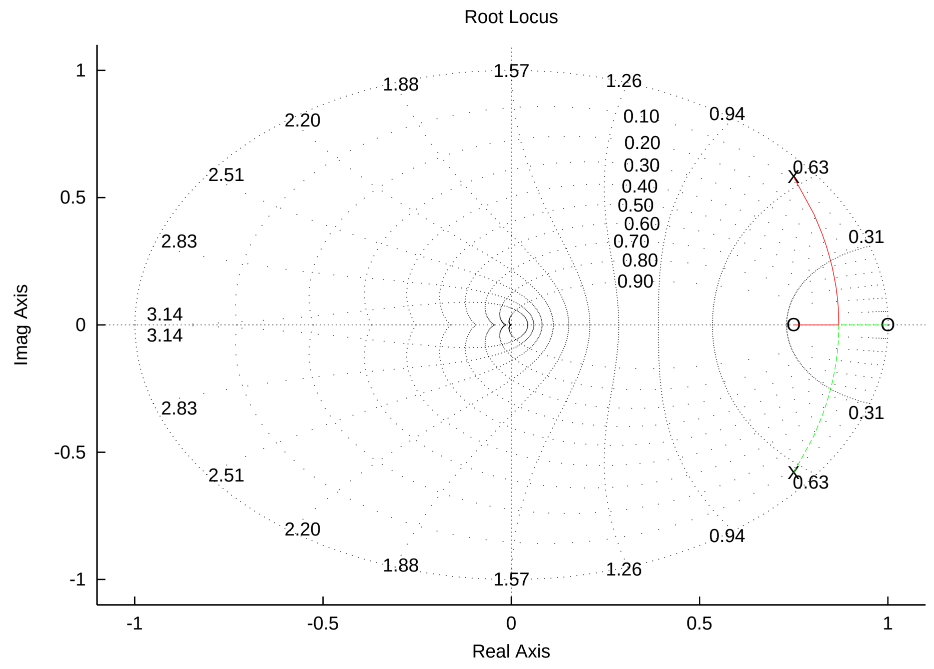


Figure 5.6: The output from executing Program 5.2.

Chapter 6

Web-Based Control Design and Analysis System

Using web-based control design and analysis system (WCDAS) in Ch Control System Toolkit, design and analysis of control systems can be performed using a web browser over the internet without any software installation on the client machine and without tedious programming. The user can specify a simulation method, system type of continuous-time or discrete-time, system model of state-space, transfer function, or zero-pole-gain, and other system parameters in a web browser. These data are then transferred to the web server for computation using Ch and Ch Control System Toolkit through Common Gateway Interface (CGI). The text or graphical results are sent back and displayed on the client web browser. This web-based system is especially useful for teaching and student learning of control theorem. In this chapter, the web-based control design and analysis system in Ch Control System Toolkit will be described.

6.1 System Requirement and Installation

To use Web-based control and design system, it only requires a Web brower in a computer connected to the internet. However, to install and setup a Web-based control and design system, it requires the following modules. which are available for downloading from the Web.

1. A Web server.
2. Ch Professional Edition.
3. Ch CGI Toolkit.
4. Ch Control System Toolkit.

The html files for Web-based control and design system are located in `$CHHOME/toolkit/demos/CGI/chhtml/toolkit/control` You need to copy or symbolically link html files to a documentation directory in a Web server, such as /usr/local/apache_1.3.9/htdocs/chhtml/toolkit/control for Apache web server version 1.3.9, by the folllowing command in Ch.

```
cd your_web_doc_home_dir
mkdir -p chhtml/toolkit
cd chhtml/toolkit
ln -s $CHHOME/toolkit/demos/CGI/chhtml/toolkit/control control
```

or

```
cp -r $CHHOME/toolkit/demos/CGI/chhtml/toolkit/control control
```

The Ch CGI programs for Web-based control and design system are located in `$CHHOME/toolkit/demos/CGI/chcgi/toolkit/control`. You need to copy or symbolically link the chcgi directory CHHOME/toolkit/demos/CGI/chcgi/toolkit/control to the Web server cgi-bin directory such as /usr/local/apache_1.3.9/htdocs/chcgi/toolkit/control for Apache web server version 1.3.9, by the folllowing command in Ch.

```
cd your_web_cgi-bin_dir
mkdir -p chcgi/toolkit
cd chcgi/toolkit
ln -s $CHHOME/toolkit/demos/CGI/chcgi/toolkit/control control
```

or

```
cp -r $CHHOME/toolkit/demos/CGI/chcgi/toolkit/control control
```

After sectup, Ch Web-based control and design system can be accessed at http://www.your_web_server.com/chhtml/toolkit/control/index.html.

6.2 Features of Web-Based Control Design and Analysis System

Taking the advantage of the Ch language environment, the web-based control design and analysis system has the following salient features:

Powerful. The WCDAS provides most commonly used functions in control system design and analysis such as time-domain response, frequency-domain response, system analysis, system design, model conversion and system conversion, etc. Most functions can be applied to both continuous-time and discrete-time LTI systems, which can be modeled in state-space equation, transfer function or zero-pole-gain representations. They also support multiple-input, and multiple-output (MIMO) systems. The functionality of the WCDAS is outlined in Table 6.1. Figure 6.1 gives the partial view of the index page of WCDAS.

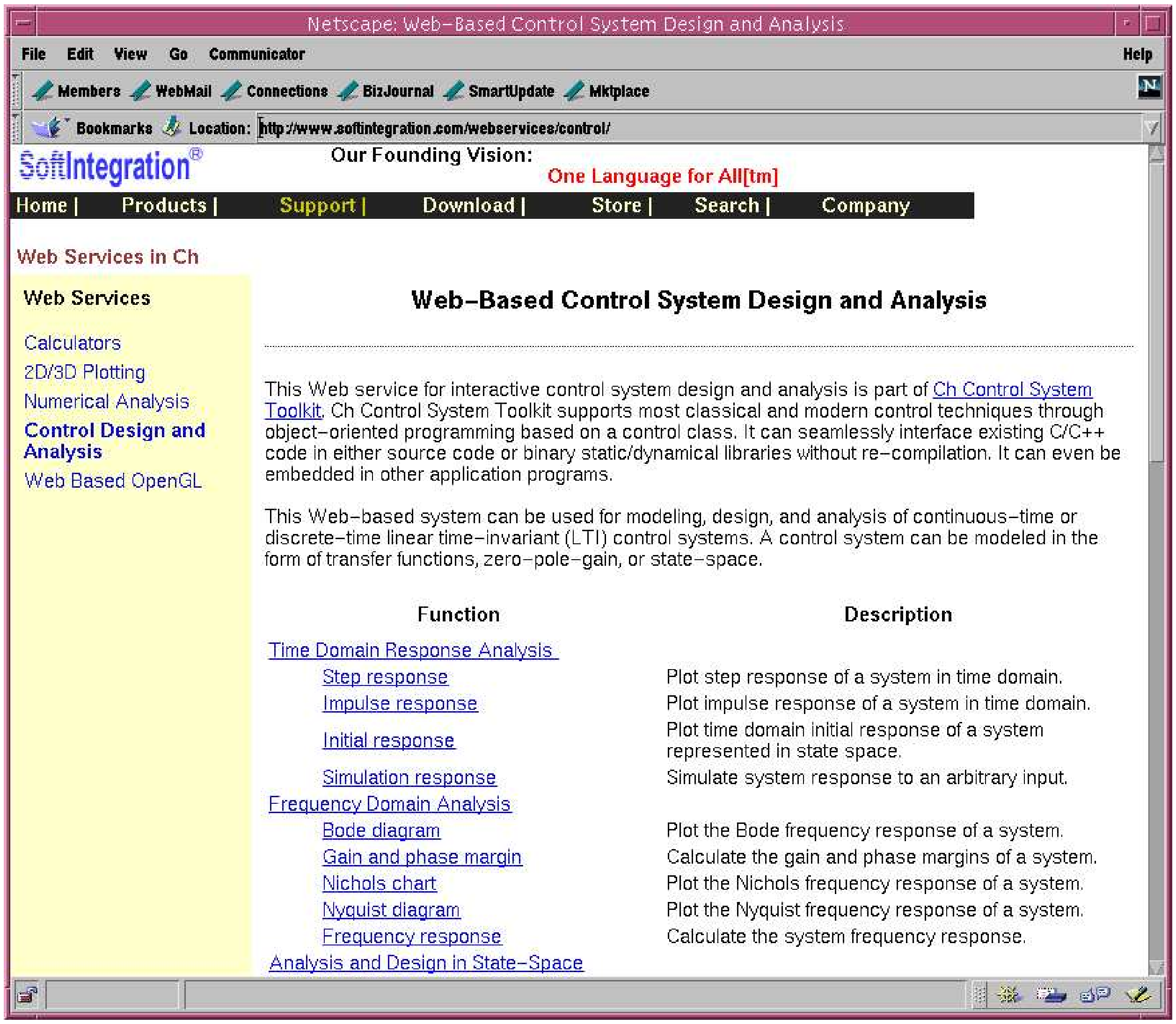
Interactive. All the functions in WCDAS are interactive and all parameters such as system model, system type, and system data can be modified on-line to solve practical engineering problems. The user's input is checked for consistency of control models, vector and matrix dimension matching, data validation, etc. with informative messages to guide the user to solve control problems.

Ease to use. A unique feature of the WCDAS is its ability for the user to design, analyze and verify control strategies over the internet without any software installation, system configuration and programming. The user can focus on the control system problems and get the results on the fly without tedious programming. The WCDAS provides a working sample case for each function to illustrate its use. By following the sample cases, entering the system parameters in the form and clicking buttons to select the different choices, the user can gain valuable experience of the control system design and analysis, and readily solve more complicated problems.

Easy to maintain. From the system developer's point of view, the WCDAS is easy to maintain. Each function is arranged in a series of files without across function calls. A function can be added or modified from the web site with ease.

Table 6.1: Functions of web-based control design and analysis system.

1.Time Domain Response Analysis	**5. Model Reduction and Dynamics**
Step response	Bandwidth
Impulse response	Pole-zero map
Initial response	Damping factors and natural frequencies
Simulation response	DC gain
2. Frequency Domain Analysis	Sort poles
Bode diagram	Minimal realization
Gain and phase margin	Pzcancel
Nichols chart	**6. Model Conversion**
Nyquist diagram	State-space model
Frequency response	Transfer function model
3. Analysis and Design in State-Space	ZPK model
Controllability analysis	**7. System Conversion**
Controllability staircase	Coordinate transformation
Grammian	Continuous-time to discrete-time
LQE design	Discrete-time to continuous-time
LQG design	Discrete-time to discrete-time
Lyapunov equation solvers	Delay2z
Observability analysis	**8. System Interconnection**
Observability staircase	Series
Pole placement	Parallel
4. Root Locus Design	Feedback
Root locus	Append
	Connect


Netscape: Web-Based Control System Design and Analysis
File Edit View Go Communicator Help
Members WebMail Connections BizJournal SmartUpdate Mktplace
Bookmarks Location: http://www.softintegration.com/webservices/control/
SoftIntegration®
Our Founding Vision:
One Language for All[tm]
Home | Products | Support | Download | Store | Search | Company
Web Services in Ch
Web Services
Calculators
2D/3D Plotting
Numerical Analysis
Control Design and Analysis
Web Based OpenGL

Web–Based Control System Design and Analysis

This Web service for interactive control system design and analysis is part of Ch Control System Toolkit. Ch Control System Toolkit supports most classical and modern control techniques through object–oriented programming based on a control class. It can seamlessly interface existing C/C++ code in either source code or binary static/dynamical libraries without re–compilation. It can even be embedded in other application programs.

This Web–based system can be used for modeling, design, and analysis of continuous–time or discrete–time linear time–invariant (LTI) control systems. A control system can be modeled in the form of transfer functions, zero–pole–gain, or state–space.

Function	Description
Time Domain Response Analysis	
Step response	Plot step response of a system in time domain.
Impulse response	Plot impulse response of a system in time domain.
Initial response	Plot time domain initial response of a system represented in state space.
Simulation response	Simulate system response to an arbitrary input.
Frequency Domain Analysis	
Bode diagram	Plot the Bode frequency response of a system.
Gain and phase margin	Calculate the gain and phase margins of a system.
Nichols chart	Plot the Nichols frequency response of a system.
Nyquist diagram	Plot the Nyquist frequency response of a system.
Frequency response	Calculate the system frequency response.
Analysis and Design in State–Space	

Figure 6.1: Index page of web-based control design and analysis system.

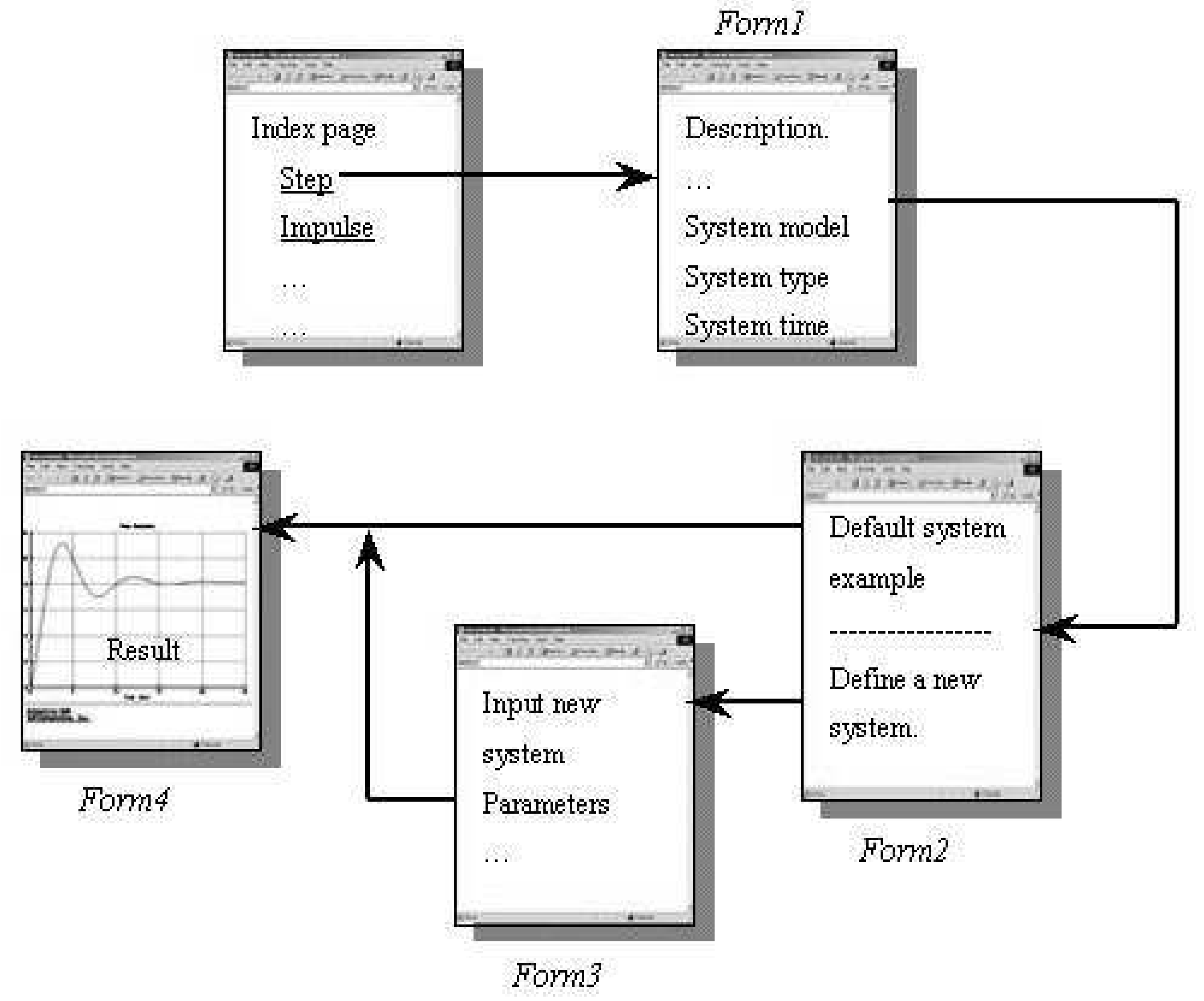


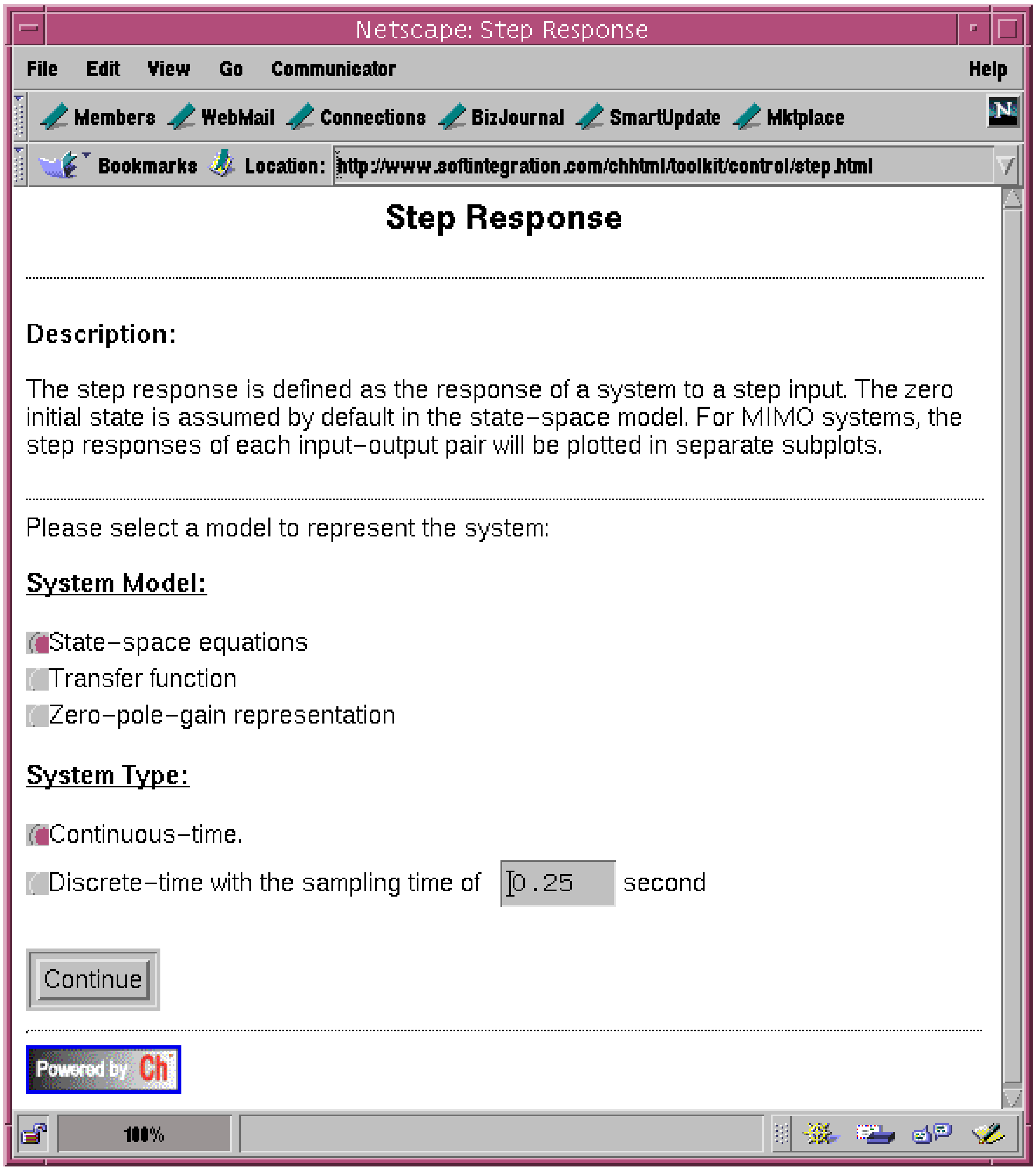
Figure 6.2: The overview of using a function in the WCDAS.

6.3 User Interface of Web-Based Control Design and Analysis System

In WCDAS, multiple web pages are provided for each function. In this section, the function of step response will be used as an example to illustrate the user interface of the system. The system overview of using a function in the WCDAS, such as step response function, is shown in Figure 6.2.

Clicking *Step response* link on the index page shown in Figure 6.1, the new page will be brought up as shown in Figure 6.3, which is indicated as Form 1 in Figure 6.2. As the first page for function of step response, function description is presented at the top of the page. The user first selects a system model from state-space, transfer function or zero-pole-gain representations. Then selects the system type of continuous-time or discrete-time. For a discrete-time system, the user has to input a sampling time in unit of second. In Figure 6.3, continuous system in state-space equation has been selected. We label this web page as Form 1.

Clicking on *Continue* button in Form 1, we will get the second page shown in Figure 6.4. This page consists of two parts. A default example, corresponding to the system type and system model chosen by the user in the previous page Form 1, is given in the upper part. For the function of step response, the default


Netscape: Step Response
File Edit View Go Communicator Help
Members WebMail Connections BizJournal SmartUpdate Mktplace
Bookmarks Location: http://www.softintegration.com/chhtml/toolkit/control/step.html
Continue
Powered by Ch
100%

Step Response

Description:

The step response is defined as the response of a system to a step input. The zero initial state is assumed by default in the state-space model. For MIMO systems, the step responses of each input-output pair will be plotted in separate subplots.

Please select a model to represent the system:

System Model:

[✓ State-space equations]
Transfer function
Zero-pole-gain representation

System Type:

[✓ Continuous-time.]
Discrete-time with the sampling time of 0.25 second

Figure 6.3: System model and system type selection page (Form 1).

control system in state-space equation is as follows.

$$\dot{x}_1 = -0.5x_1 - 0.8x_2 + u \tag{6.1}$$
$$\dot{x}_2 = 0.8x_1 \tag{6.2}$$
$$y = 2x_1 + 6.5x_2 \tag{6.3}$$

In standard state-space equation of

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} \tag{6.4}$$
$$\mathbf{y} = \mathbf{C}\mathbf{x} + \mathbf{D}\mathbf{u} \tag{6.5}$$

values for matrices **A**, **B**,**C** and **D** are

$$\mathbf{A} = \begin{bmatrix} -0.5 & -0.8 \\ 0.8 & 0 \end{bmatrix}, \mathbf{B} = \begin{bmatrix} 1.0 \\ 0 \end{bmatrix}, \mathbf{C} = \begin{bmatrix} 2 & 6.5 \end{bmatrix}, \mathbf{D} = \begin{bmatrix} 0 \end{bmatrix}. \tag{6.6}$$

This example illustrates how to fill out the input boxes. The user can just click *Run Example* button in this part to get the result.

The user can also define a new control system in the lower part of the Figure 6.4. The dimension of system matrices for state-space equations, the order of polynomials of numerator and denominator for transfer function, the number of zeros and poles for ZPK representations can be specified by the user. For example, the lower part in Figure 6.4 asks the user to specify the dimension of system matrices because we chose the system model of state-space representation in the previous page shown in Figure 6.3. We label this web page as Form 2.

If the user clicks *Submit* button in the lower part of Form 2, a new page will be brought up as shown in Figure 6.5. This page asks the user to input each entry of the system matrices for a state-space model, the coefficients of polynomials of numerator and denominator for a transfer function model, or the zeros, poles, and gain for a zpk representation. We label this new page as Form 3. In Figure 6.5, we entered values for system matrices **A,B,C** and **D** in state-space equation. The dimensions of these matrices are defined in the lower part of Form 2 in Figure 6.4.

The user can obtain the final result by two ways. One is clicking*Run Example* button in Form 2 for sample system, and the other is *Run* button in Form 3 for user-defined systems. We label the result page as Form 4. The time duration of 16 seconds and plotting grid are selected at the bottom of Form 3 in Figure 6.5. Figure 6.6 shows the step response of the system defined in Figure 6.5.

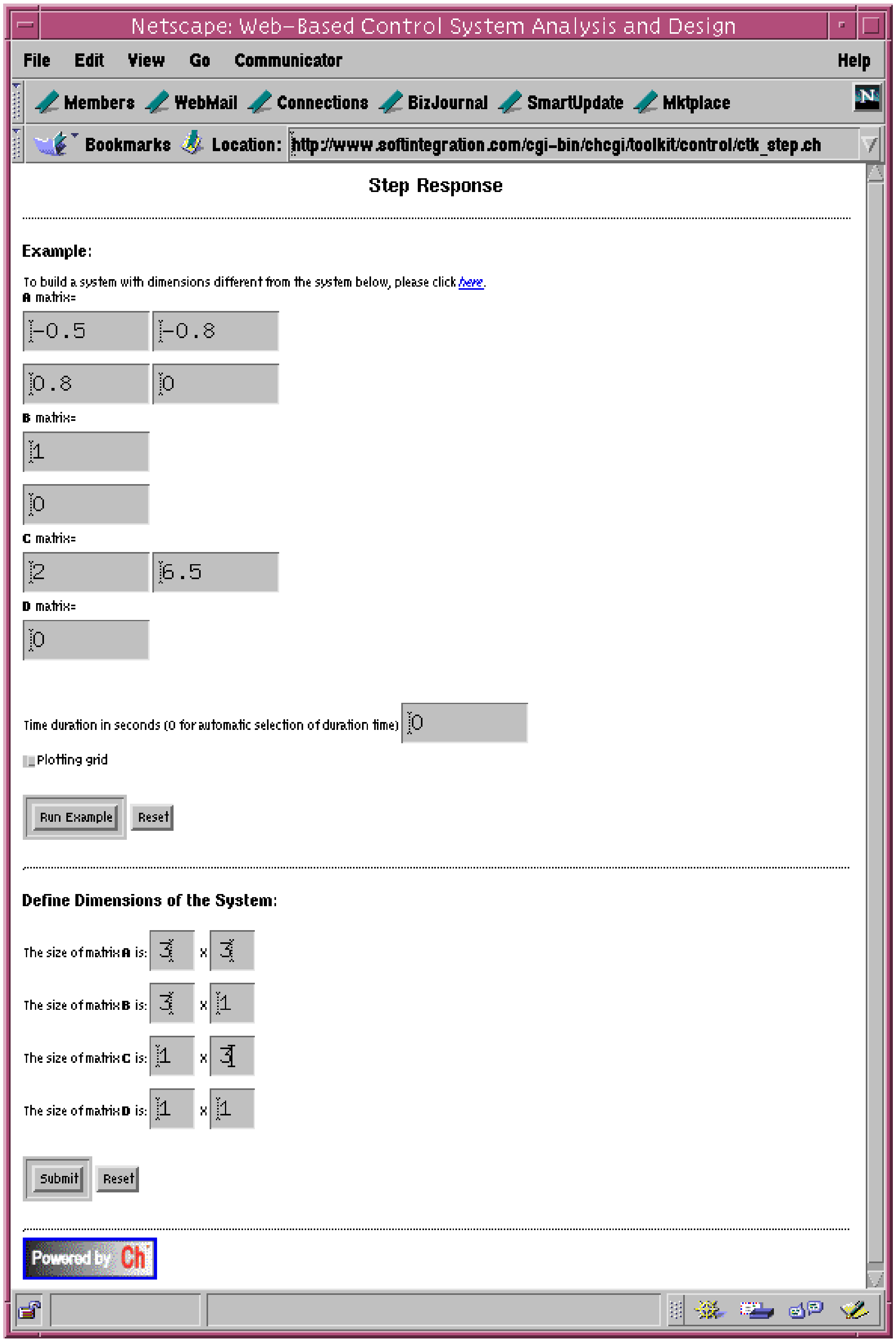
6.4 An Application Example

In this section, a sample application will be used to illustrate how control problems are solved using this web-based control design and analysis system. For a continuous-time linear time-invariant control system represented in state-space equations (6.4) and (6.5), given

$$\mathbf{A} = \begin{bmatrix} -0.5 & -0.8 & 0.53 \\ 0.8 & 0.12 & 0.7 \\ -0.53 & -0.7 & -1.1 \end{bmatrix}, \mathbf{B} = \begin{bmatrix} 1.0 \\ 0.8 \\ -1.5 \end{bmatrix}, \mathbf{C} = \begin{bmatrix} 2 & 6.5 & -2 \end{bmatrix}, \mathbf{D} = \begin{bmatrix} 1.2 \end{bmatrix}. \tag{6.7}$$

plot the step response of the system.

The procedure to obtain the result is as follows. First, click *Step response* link in the index page shown in Figure 6.1 to bring up Form 1 shown in Figure 6.3. Then, select the system model of state-space equation and system type of continuous-time, click *Continue* button then Form 2 appears as shown in Figure 6.4. We



http://www.softintegration.com/cgi-bin/chcgi/toolkit/control/ctk_step.ch

Step Response

Example:

To build a system with dimensions different from the system below, please click here.

A matrix=

-0.5	-0.8
0.8	0

B matrix=

1
0

C matrix=

2	6.5

D matrix=

0

Time duration in seconds (0 for automatic selection of duration time) 0

Plotting grid

Run Example Reset

Define Dimensions of the System:

The size of matrix A is: 3 x 3

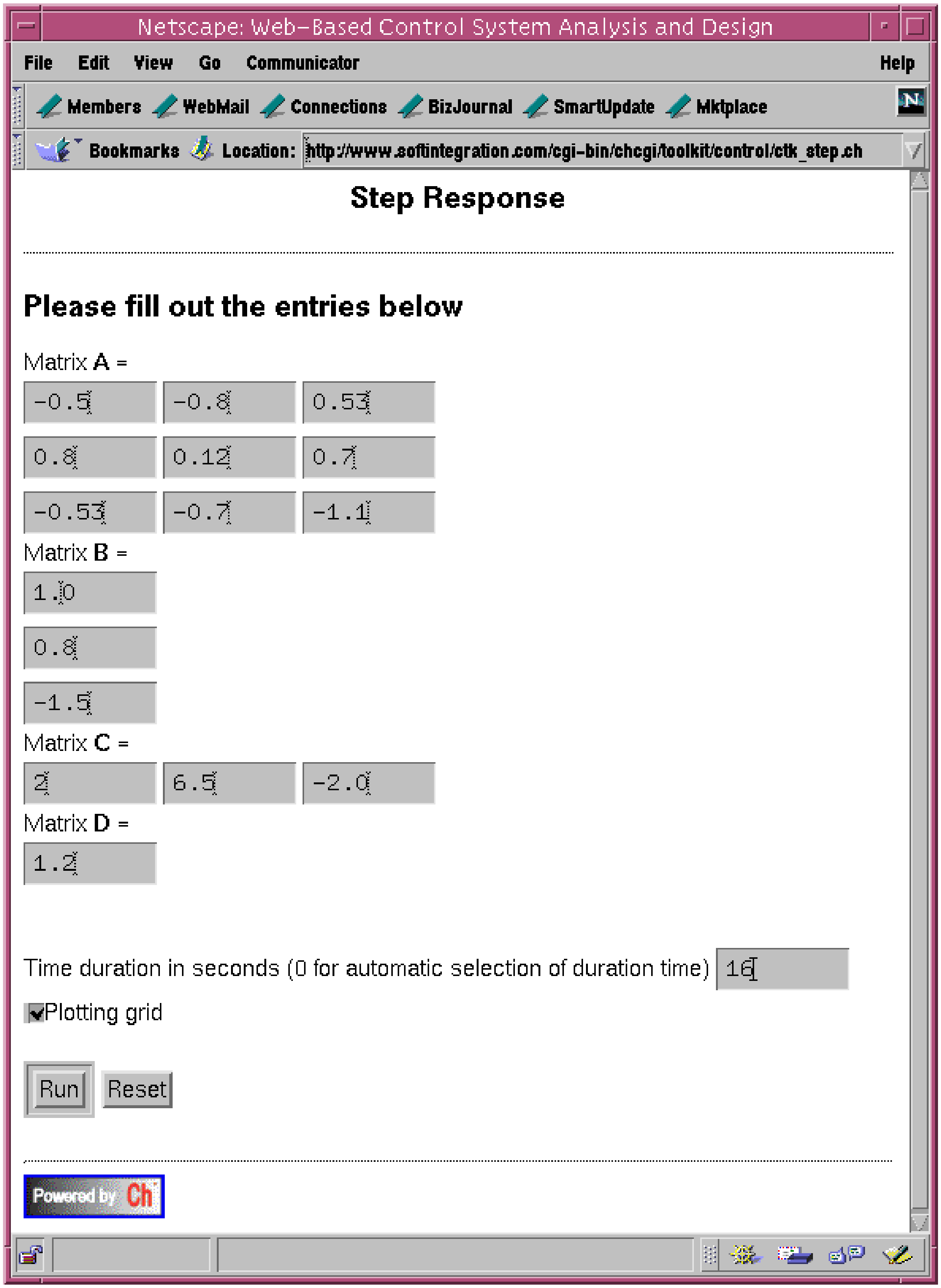
The size of matrix B is: 3 x 1

The size of matrix C is: 1 x 3

The size of matrix D is: 1 x 1

Submit Reset

Figure 6.4: The default system example and new system definition page (Form 2).



http://www.softintegration.com/cgi-bin/chcgi/toolkit/control/ctk_step.ch

Step Response

Please fill out the entries below

Matrix **A** =

-0.5	-0.8	0.53
0.8	0.12	0.7
-0.53	-0.7	-1.1

Matrix **B** =

1.0
0.8
-1.5

Matrix **C** =

2	6.5	-2.0

Matrix **D** =

1.2

Time duration in seconds (0 for automatic selection of duration time) 16

✓ Plotting grid

Figure 6.5: The user defined system input page (Form 3).

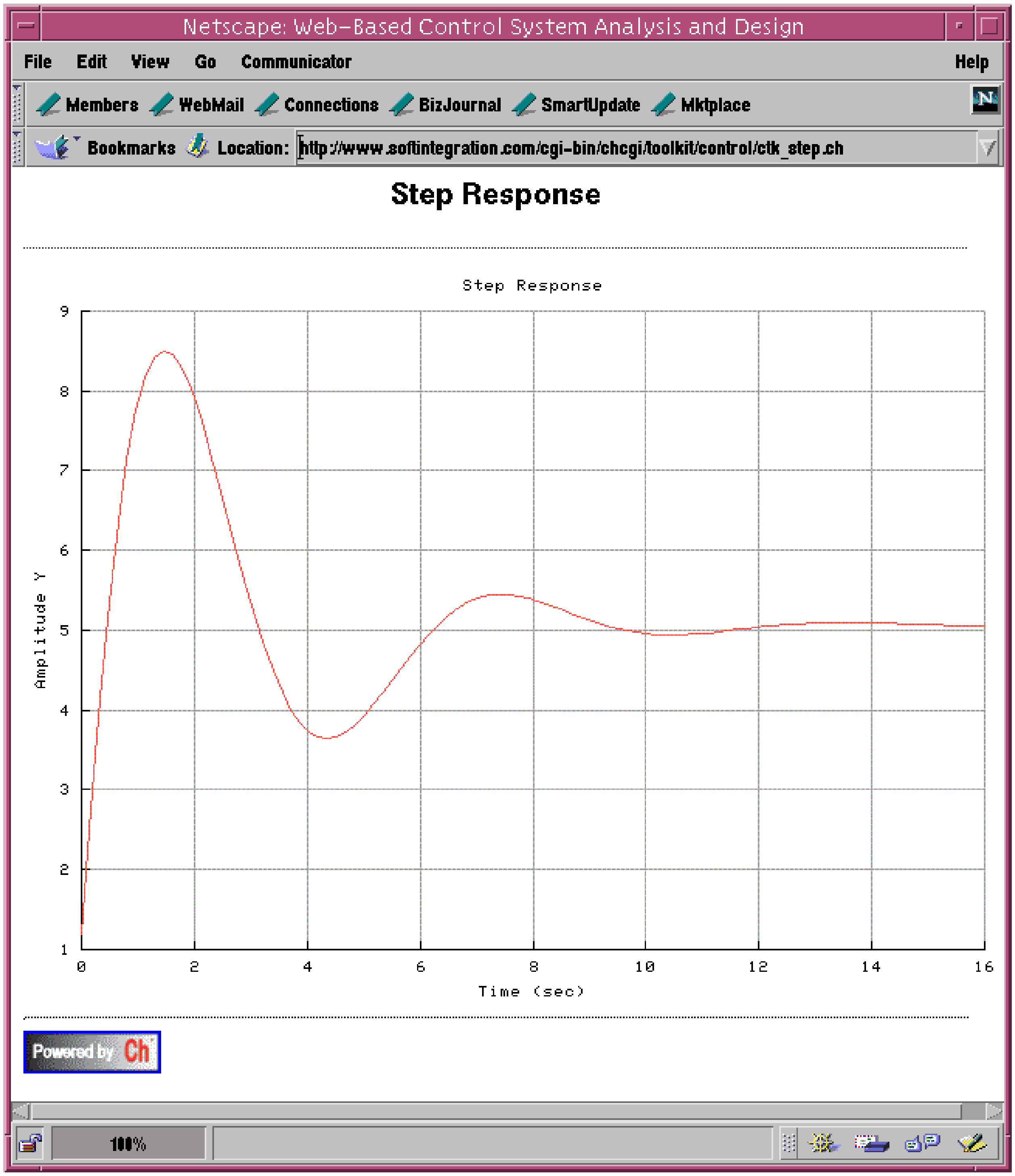


Figure 6.6: The result page of the step response (Form 4).

need to build a new system with matrix dimensions of **A** of 3x3, **B** of 3x1, **C** of 1x3 and **D** of 1x1. Next, click *Submit* button in Form 2 in Figure 6.4, Form 3 will be brought up as shown in Figure 6.5. Fill out each element with given data. Finally, click *Run* button to display the result as shown in Figure 6.6.

Appendix A

Reference of Member Functions of CControl Class

The Ch Control System Toolkit provides class **CControl** and a set of member functions for control system design, analysis, and modeling. The header file **control.h** contains the definition of the the class **CControl** , macros and prototypes of member functions that are alphabetically listed in Table A.1.

Table A.1: Member functions of **CControl** class.

Functions	**Description**	**C**[1]	**D**[2]	**M**[3]
acker()	pole placement gain selection using Ackermann's formula for SI systems	X	X	X[4]
append()	group LTI models by appending their inputs and outputs	X	X[5]	X
augstate()	append the state vector to the output vector	X	X	X
bandwidth()	bandwidth of SISO models	X		N
bode()	Bode plot	X	X	
bodemag()	plot Bode magnitude	X		
c2d()	convert systems from continuous-time models to discrete-time models	X	N	X
canon()	canonical state-space realizations	X	X	X
compensatorZeroPoleGain()	compute a compensator's zero, pole, and gain	X		
connect()	derive state-space model from block diagram description	X	X	X
ctrb()	compute controllability matrix of systems	X	X	X
ctrbf()	compute controllability staircase form	X	X	X
d2c()	convert systems from discrete-time models to continuous-time models	N	X	X
d2d()	convert discrete-time models to discrete-time models with different sample time	N	X	X
damp()	compute natural frequency and damping factors	X		
dcgain()	compute low-frequency gain	X	X	X

[1]Continuous-time systems supporting. 'X' indicates that the member function supports continuous-time systems; Blank indicates that the feature will be added; 'N' indicates that the function does not apply to continuous-time systems.

[2]Discrete-time systems supporting. 'X' indicates that the member function supports discrete-time systems; Blank indicates that the feature will be added; 'N' indicates that the function does not apply to discrete-time systems.

[3]MIMO systems supporting. 'X' indicates that the member function supports MIMO systems; Blank indicates that the feature will be added; 'N' indicates that the function does not apply to MIMO systems. All member functions support SISO systems.

[4]Only apply to single input system.

[5]All discrete-time systems should have the same sample time.

Table A.1: Member functions of **CControl** .

Functions	**Description**	**C**	**D**	**M**
delay2z()	replace all delays with a phase shift	N	X	
dlqr()	Linear-quadratic regulator design for discrete-time systems	N	X	X
dlyap()	solve discrete-time Lyapunov equations	N	X	X
dsort()	sort the poles of discrete-time systems	N	X	
esort()	sort the poles of continuous-time systems	X	N	
estim()	produce a state/output estimator	X		X
feedback()	feedback connection of two LTI models	X	X	X
freqresp()	frequency response function	X		
getDominantPole()	compute the dominant pole based on the percent overshoot or damping ratio	X		
getSystemType()	get system type	X	X	X
gram()	controllability and observability gramians	X	X	X
grid()	add regular/Nyquist grid lines	X	X	X
hasdelay()	determine if the system has any delay	X	X	
impulse()	impulse response	X	X	X
initial()	initial condition response	X	X	X
isDiscrete()	determine if the system is discrete	X	X	X
lqe()	Kalman estimator design for continuous-time systems	X	N	X
lqr()	Linear-quadratic regulator design for continuous-time systems	X	N	X
lsim()	response to arbitrary inputs	X	X	X
lyap()	solve continuous-time Lyapunov equations	X	N	X
margin()	compute gain and phase margins	X		N
minreal()	minimal realization of an LTI model	X	X	X
model()	create a model for an LTI system	X	X	X
ngrid()	add Nichols chart grid lines	X		N
nichols()	Nichols plot	X		
nyquist()	Nyquist plot	X		
obsv()	compute observability matrix of the system	X	X	X
obsvf()	compute observability staircase form	X	X	X
parallel()	parallel connection of LTI models	X	X	X
place()	pole placement design	X	X	X
pole()	compute poles of LTI systems	X	X	
printSystem()	print information of system	X	X	X
printss()	print system, input, output, and direct transmission matrices for state space models	X	X	X
printtf()	print numerator and denominator of a transfer function model	X	X	
printzpk()	print zeros, poles and gain of a zero-pole-gain model	X	X	
pzcancel()	pole-zero cancellation in transfer function or zero-pole-gain models	X	X	
pzmap()	compute the pole-zero map of an LTI model	X	X	
rlocfind()	find feedback gains for a given set of roots	X	X	N
rlocus()	compute and plot the root locus of an SISO system	X	X	N
series()	series connection of LTI models	X	X	X
setDelay()	set delay in the system	X	X	

Table A.1: Member functions of **CControl** .

Functions	**Description**	**C**	**D**	**M**
setTs()	set sample time of discrete-time system	N	X	X
sgrid()	generate an s-plane grid	X	N	X
size()	get system parameters' sizes	X	X	X
ss2ss()	state coordinate transformation for state-space models	X	X	X
ssdata()	get system, input/output and direct transmission matrix in SS model	X	X	X
step()	step response	X	X	X
stepinfo()	characteristics information for step response	X	X	X
tfdata()	get the numerators and denominators in TF model	X	X	
tzero()	return the transmission zeros of the LTI system	X	X	
zgrid()	generate a z-plane grid	N	X	X
zpkdata()	get zeros, poles and gains in ZPK model	X	X	

CControl ::acker

Synopsis
int **acker** (array double &k, array double complex p[:]);

Purpose
Pole placement design for single-input systems.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters

k Array of reference containing the gains that make the closed-loop poles match these specified in the argument p.

p An assumed-shape array of double complex which contains desired closed-loop pole locations.

Description
The function **acker** () uses Ackermann's formula to calculate a gain vector k such that the state feedback $u = -kx$ places the closed-loop poles, which are the eigenvalues of matrix ($A - Bk$), at the locations specified by p. Here, A and B are system and input matrices, respectively.

Note: acker () is limited to controllable single-input systems. The member function **place** () is recommented for pole placement design.

Example

```
#include <control.h>

#define NUMX 2        // number of states
#define NUMU 1        // number of inputs
#define NUMY 1        // number of outputs

int main() {
     array double A[NUMX][NUMX] = {-0.5572,   -0.7814,
                                    0.7814,    0};
     array double B[NUMX][NUMU] = {1, 0};
     array double C[NUMY][NUMX] = {1.9691,  6.4493};
     array double k[NUMX];
     array double complex p[NUMX] = {1, 2};
     int i;
     CControl sys;

     sys.model("ss", A, B, C, NULL);
     sys.acker(k, p);

     for (i = 0; i < NUMX; i ++) {  // print out the result
       printf("k[%d] = %f\n", i, k[i]);
     }
     return 0;
}
```

The output is shown as follows.

```
k[0] = -3.557200
k[1] = 1.778109
```

See Also
CControl::lqr(), **CControl::rlocus**(), **CControl::place**().

CControl ::append

Synopsis
CControl* **append** (. . . /* **CControl** *$sys2$, . . . */);

Purpose
Group LTI models by appending their inputs and outputs.

Return Value
Upon successful completion, a non-NULL pointer to class **CControl** representing the generated system is returned. Otherwise, a NULL pointer is returned.

Parameters
. . . Variable number of pointers to class **CControl** representing systems to be appended to this system.

Description
The function **append** () appends the inputs and outputs of the LTI models sys_1, . . ., and sys_n, which are represented in state space models as (A_1, B_1, C_1, D_1), . . ., and (A_n, B_n, C_n, D_n), to form the resulting model sys. The matrices of state space model (A, B, C, D) of sys is shown as below.

$$A = \begin{bmatrix} A_1 & 0 & \dots & 0 \\ 0 & A_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & A_n \end{bmatrix}, B = \begin{bmatrix} B_1 & 0 & \dots & 0 \\ 0 & B_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & B_n \end{bmatrix},$$

$$C = \begin{bmatrix} C_1 & 0 & \dots & 0 \\ 0 & C_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & C_n \end{bmatrix}, D = \begin{bmatrix} D_1 & 0 & \dots & 0 \\ 0 & D_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & D_n \end{bmatrix}$$

If the systems are created in other models rather than state space, they will be changed to state space models by Ch Control System Toolkit automatically.

Example

```
#include <control.h>

#define NUMU2 1        // number of inputs
#define NUMY2 1        // number of outputs
#define NUMX3 1        // number of states
#define NUMU3 1        // number of inputs
#define NUMY3 1        // number of outputs
```

```
int main() {
    array double num[1] = {1}, den[2] = {1, 0};
    array double D2[NUMY2][NUMU2] = {10};
    array double A[NUMX3][NUMX3] = {1},
                 B[NUMX3][NUMU3] = {2},
                 C[NUMY3][NUMX3] = {3},
                 D[NUMY3][NUMU3] = {4};
    CControl sys1, sys2, sys3, *sys;

    sys1.model("tf", num, den);
    sys2.model("ss", NULL, NULL, NULL, D2);
    sys3.model("ss", A, B, C, D);

    sys = sys1.append(&sys2, &sys3);
    sys->printSystem();
    return 0;
}
```

The output is shown as follows.

```
Continuous-time System
State-space arguments:
A =
 -0.000000   0.000000
  0.000000   1.000000
B =
  1.000000   0.000000   0.000000
  0.000000   0.000000   2.000000
C =
  1.000000   0.000000
  0.000000   0.000000
  0.000000   3.000000
D =
  0.000000   0.000000   0.000000
  0.000000  10.000000   0.000000
  0.000000   0.000000   4.000000
```

See Also
CControl::connect(), **CControl::parallel**(), **CControl::series**().

CControl ::augstate

Synopsis
CControl* **augstate** ();

Purpose
Append the state vector to the output vector.

Return Value
Upon successful completion, a non-NULL pointer to class **CControl** representing the generated system is returned. Otherwise, a NULL pointer is returned.

Parameters
None.

Description

The function **augstate** () appends the state vector to the output vector of the system. Assume that the system sys has state space model

$$\begin{aligned} \dot{X} &= AX + BU \\ Y &= CX + DU \end{aligned}$$

the function call `sys.augstate` returns a system with state space model of

$$\dot{X} = AX + BU$$
$$\begin{bmatrix} Y \\ X \end{bmatrix} = \begin{bmatrix} C \\ I \end{bmatrix} X + \begin{bmatrix} D \\ 0 \end{bmatrix} U$$

If the systems are created in other models rather than in state space, they will be changed to state space models by Ch Control System Toolkit automatically.

Example

```
#include <control.h>

#define NUMX 4        // number of states
#define NUMU 2        // number of inputs
#define NUMY 2        // number of outputs

int main() {
    array double A[NUMX][NUMX] = {{-4.12, -17.4, -30.8, -60},
                            {1,     0,     0,     0},
                            {0,     1,     0,     0},
                            {0,     0,     1,     0}},
                 B[NUMX][NUMU] = {1.5, 0, 0, 2, 1, 2, 3, 4},
                 C[NUMY][NUMX] = {4, 8.4, 30.78, 60, 3, 4, 5, 6 },
                 D[NUMY][NUMU] = {1, 2, 3, 4};
    CPlot plot;
    CControl sys, *asys;

    sys.model("ss", A, B, C, D);
    asys = sys.augstate();
    asys->printSystem();
    return 0;
}
```

The output is shown as follows.

```
Continuous-time System
State-space arguments:
A =
 -4.120000 -17.400000 -30.800000 -60.000000
  1.000000   0.000000   0.000000   0.000000
  0.000000   1.000000   0.000000   0.000000
  0.000000   0.000000   1.000000   0.000000
B =
  1.500000   0.000000
  0.000000   2.000000
  1.000000   2.000000
  3.000000   4.000000
```

```
C =
  4.000000   8.400000  30.780000  60.000000
  3.000000   4.000000   5.000000   6.000000
  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
D =
  1.000000   2.000000
  3.000000   4.000000
  0.000000   0.000000
  0.000000   0.000000
  0.000000   0.000000
  0.000000   0.000000
```

See Also
CControl::feedback(), **CControl::parallel**(), **CControl::series**().

CControl ::bandwidth

Synopsis
double* **bandwidth** (. . . /* double dbdrop */);

Purpose
Calculate the bandwidth of the SISO model.

Return Value
A double value representing the bandwidth of the system is returned. It is expressed in radians per second.

Parameters
. . . The following argument is optional.

dbdrop A double value which specifies the critical gain drop in decibel.

Description
The function **bandwidth** () calculates the bandwidth which is defined as the first frequency where the gain drops below the critical gain drop specified by argument *dbdrop* or 70.79 percent (-3 dB) by default of system steady-state gain.

Example

```
#include <control.h>

int main() {
   array double complex zero[2] = {-10, -20.01};
   array double complex pole[3] = {-5, -9.9, -20.1};
   double k = 1;
   double fb;
   CControl sys;

   sys.model("zpk", zero, pole, k);

   fb = sys.bandwidth();
   printf("Bandwidth of the system corresponding to default critical gain drop is %f\n", fb);

   fb = sys.bandwidth(-4);
   printf("Bandwidth of the system corresponding to critical gain drop of -4 is %f\n", fb);

   return 0;
}
```

The output is shown as follows.

```
Bandwidth of the system corresponding to default critical gain drop is 4.969758
Bandwidth of the system corresponding to critical gain drop of -4 is 6.120673
```

See Also
CControl::dcgain()

CControl ::bode

Synopsis
int* **bode** (class **CPlot** *$plot$, array double mag[&], array double $phase$[&], array double $wout$[&], $\ldots$/* array double w[&] or double $wmin$, double $wmax$ */);

Purpose
Compute the Bode frequency response of system.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters
$plot$ Pointer to an existing object of class **CPlot** .

mag Array of reference containing magnitudes of the frequency response at the frequencies in array $wout$. The magnitudes are expressed in decibels.

$phase$ Array of reference containing phases of the frequency response at the frequencies $wout$. The phases are expressed in degrees.

$wout$ Array of reference containing output frequencies which are expressed in rad/sec. If user-defined frequencies are absent, the automatically generated frequencies will be contained.

... Variable length argument list expecting one argument, w, or two arguments, $wmin$ and $wmax$.

w Array of reference containing user-defined frequencies at which frequency response is calculated.

$wmin$ **and** $wmax$ Two double value specifying bounds of a particular frequency interval [wmin,wmax].

Description
The function **bode** () computes and plots the magnitude and phase of the frequency response of LTI models. If the first argument is not a NULL pointer, a Bode plot will be produced. If the user explicitly specifies the frequency range or frequency points, they will be used in calculations and plots. Otherwise, the default frequency vector w is generated by the following statement automatically.

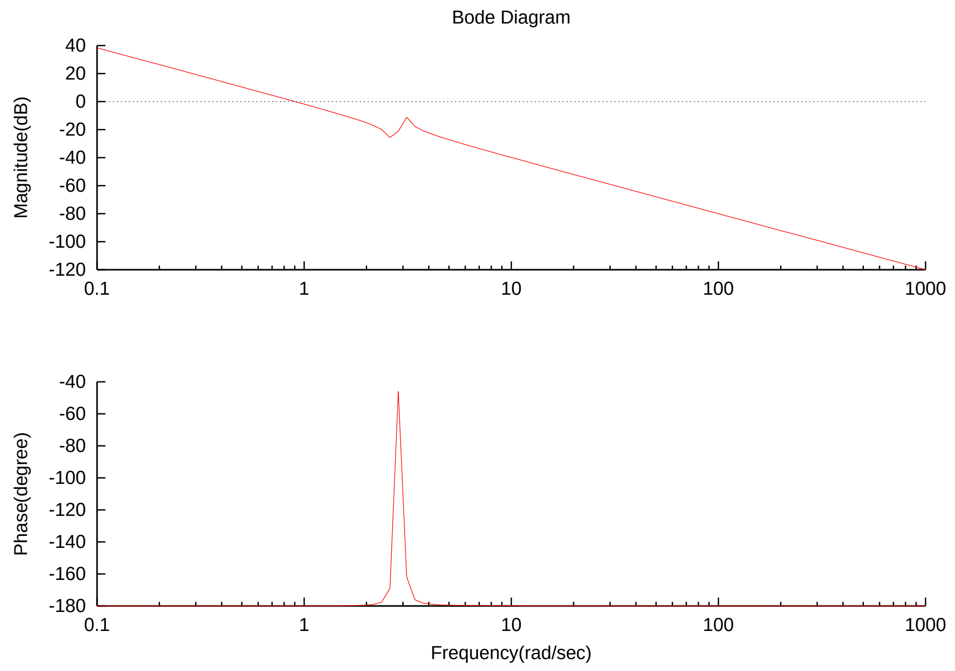
```
  logspace(w, log10(0.1), log10(1000))
```

w is a logarithmically spaced frequency vector and expressed in $radians/sec$. This function is applicable only to SISO cases in the current implementation.

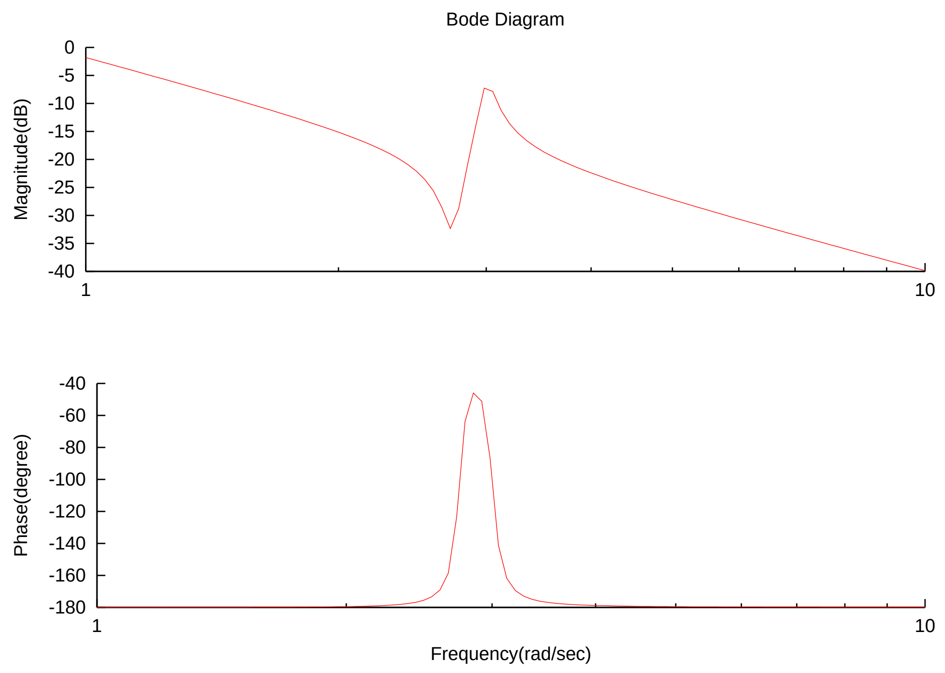
Example
In this example, the Bode plots are produced on both default and user-defined frequency range. The user can retrieve output information from arrays mag, $phase$, and $wout$.

```
#include <control.h>

int main() {
   array double num[3] = {1, 0.1, 7.5}, den[5] = {1, 0.12, 9, 0, 0};
   CPlot plot1, plot2;
   array double mag[100], phase[100], wout[100];
   CControl sys;

   sys.model("tf", num, den);
   sys.bode(&plot1, NULL, NULL, NULL);
   sys.bode(&plot2, mag, phase, wout, 1, 10);
   return 0;
}
```

The plot on the default frequency range [.1, 1000] is shown as follows.



The plot on the user-defined range [1, 10] is shown as follows.



See Also
CControl::freqresp(), **CControl::nichols**(), **CControl::nyquist**().

CControl ::bodemag

Synopsis
int* **bodemag** (class **CPlot** **plot*, array double *mag*[&], array double *wout*[&], . . . /* array double *w*[&] or double *wmin*, double *wmax* */);

Purpose
Compute the Bode frequency response of system.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters
plot Pointer to an existing object of class **CPlot** .

mag Array of reference containing magnitudes of the frequency response at the frequencies in array *wout*. The magnitudes are expressed in decibels.

wout Array of reference containing output frequencies which are expressed in rad/sec. If user-defined frequencies are absent, the automatically generated frequencies will be contained.

. . . Variable length argument list expecting one argument, *w*, or two arguments, *wmin* and *wmax*.

w Array of reference containing user-defined frequencies at which frequency response is calculated.

wmin **and** *wmax* Two double value specifying bounds of a particular frequency interval [wmin,wmax].

Description

The function **bodemag** () computes and plots the magnitude of the frequency response of LTI models. If the first argument is not a NULL pointer, a Bode plot will be produced. If the user explicitly specifies the frequency range or frequency points, they will be used in calculations and plots, otherwise the default frequency vector w is generated by the following statement automatically.

```
logspace(w, log10(0.1), log10(1000))
```

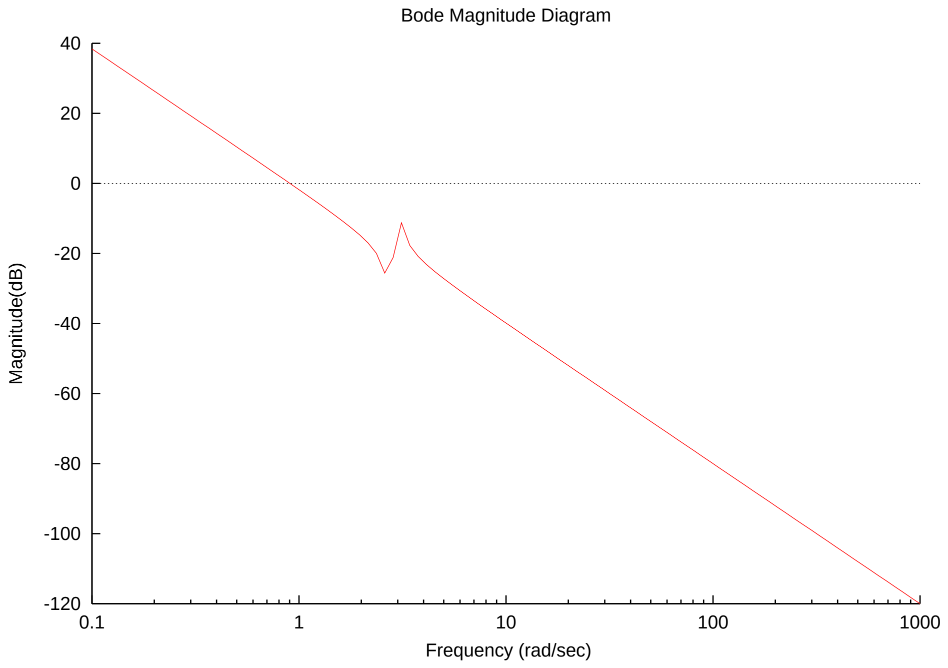
w is a logarithmically spaced frequency vector and expressed in $radians/sec$. This function is applicable to SISO system only in the current implementation.

Example

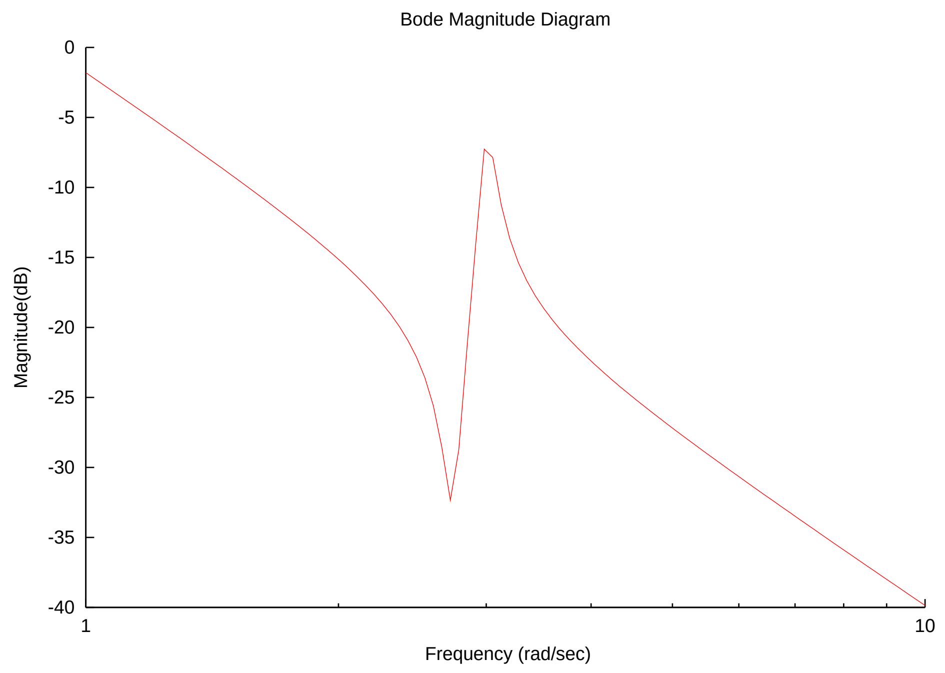
In this complete example, the Bode plots are produced on both default and user-defined frequency range. The user can retrieve output information from arrays mag, and $wout$.

```
#include <control.h>

int main() {
   array double num[3] = {1, 0.1, 7.5}, den[5] = {1, 0.12, 9, 0, 0};
   CPlot plot1, plot2;
   array double mag[100], wout[100];
   CControl sys;

   sys.model("tf", num, den);
   sys.bodemag(&plot1, NULL, NULL);
   sys.bodemag(&plot2, mag, wout, 1, 10);

   return 0;
}
```

The plot on the default frequency range [.1, 1000] is shown as follows.



The plot on the user-defined range [1, 10] is shown as follows.



See Also
CControl::bode()

CControl ::c2d

Synopsis
CControl* **c2d** (double ts, . . . /* string_t method */);

Purpose
Converts a state space model from continuous-time models to discrete-time.

Return Value
Upon successful completion, a non-NULL pointer to class **CControl** representing the resulting discrete-time generated system is returned. Otherwise, a NULL pointer is returned.

Parameters
ts Sample time.

. . . The following argument is optional.

$method$ Selected discretization method among `"zoh"`, `"foh"`, `"tustin"`, `"prewarp"`, and `"matched"`. `"zoh"` standing for `"Zero-order hold"` is the default and the only method available for now.

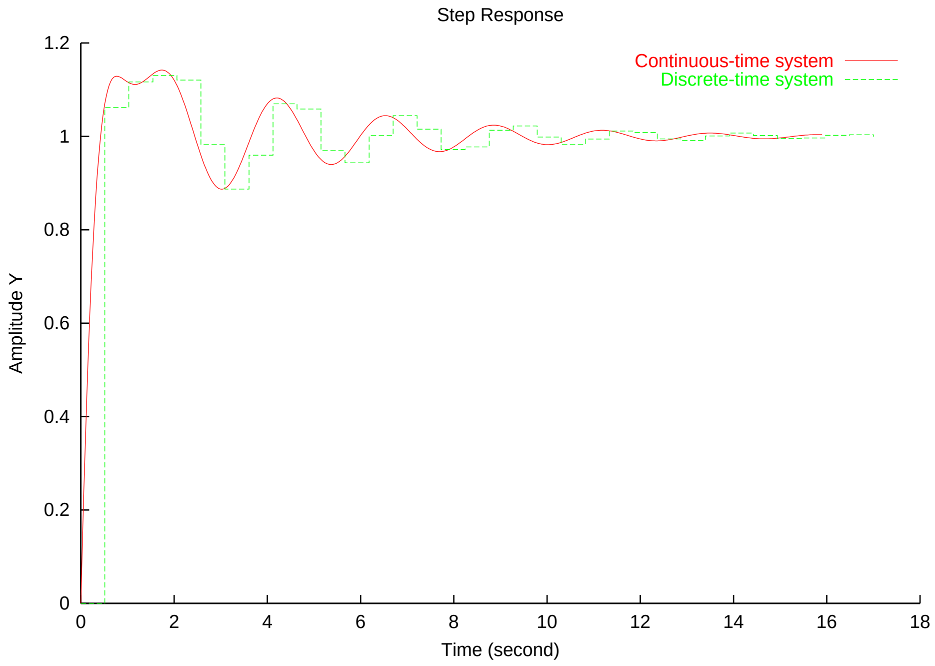
Description
The function **c2d** () discretizes the continuous-time LTI model `"sys"` using specified methods such as "zero-order hold" on the inputs with the sample time of ts seconds.

Example
The following example produces a discrete-time system $sysd$ from the continuous one sys using "zero-order hold" as the discretization method and 0.5 second as the sample time. The difference between the

continuous and discretized system are illustrated by the comparison of their step responses in the following figure.

```
#include <control.h>

#define NUMX 4    // number of states
#define NUMU 1    // number of inputs
#define NUMY 1    // number of outputs
#define TS   0.5  // time sampling of discrete system
#define N1   500  // number of time samples for continuous-time system
#define N2   34   // number of time samples for discrete-time system
  // 17 seconds for ts = 0.5

int main() {
    array double A[NUMX][NUMX] = {{-4.12, -17.4, -30.8, -60},
                                  {1,     0,     0,     0},
                                  {0,     1,     0,     0},
                                  {0,     0,     1,     0}},
                 B[NUMX][NUMU] = {1, 0, 0, 0},
                 C[NUMY][NUMX] = {4, 8.4, 30.78, 60};
    CControl sys, *sysd;
    array double y1[N1], t1[N1];
    array double y2[N2], t2[N2];
    CPlot plot;

    sys.model("ss", A, B, C, NULL);
    sys.step(NULL, y1, t1, NULL);
    plot.data2D(t1, y1);

    sysd = sys.c2d(TS, "zoh");
    sysd->step(NULL, y2, t2, NULL);
    plot.data2D(t2, y2);

    plot.plotType(PLOT_PLOTTYPE_STEPS, 1);
    plot.label(PLOT_AXIS_X, "Time (second)");
    plot.label(PLOT_AXIS_Y,"Amplitude Y");
    plot.title("Step Response");
    plot.legend("Continuous-time system", 0);
    plot.legend("Discrete-time system", 1);
    plot.plotting();

    return 0;
}
```

The step responses of both systems are shown as follows.



See Also
CControl::d2c(), **CControl::d2d**(), **CControl::ss2ss**().

CControl ::canon

Synopsis
CControl* canon (array double &T, . . . /* string_t *type* */);

Purpose
Compute canonical state-space realizations.

Return Value
Upon successful completion, a non-NULL pointer to class **CControl** expressed in canonical state-space model is returned. Otherwise, a NULL pointer is returned.

Parameters
T State coordinate transformation.

. . . The following argument is optional.

type Type of canonical forms. Two types, `"modal"` and `"companion"`, are available.

Description
The function **canon** () returns a canonical state-space model of the original LTI system. The system matrix A_c of the canonical model can be in modal or companion form. In modal form, the real eigenvalues of A_c appear on the diagonal, and the complex conjugate eigenvalues appear in 2×2 blocks on the diagonal. Assume matrix A has eigenvalues $(\lambda, \sigma \pm \omega)$, it has the modal form as

$$A_c = \begin{bmatrix} \lambda & 0 & 0 \\ 0 & \sigma & \omega \\ 0 & -\omega & \sigma \end{bmatrix}$$

In companion form, coefficients of the characteristic polynomial of the system appears explicitly in the rightmost column of the system matrix A_c. For a system with characteristic polynomial

$$p(s) = s^n + a_1 s^{n-1} + \ldots + a_{n-1}s + a_n$$

the system matrix A_c in companion form is

$$\begin{bmatrix} 0 & 0 & \ldots & \ldots & 0 & -a_n \\ 1 & 0 & 0 & \ldots & 0 & -a_{n-1} \\ 0 & 1 & 0 & \ldots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \ldots & 1 & 0 & -a_2 \\ 0 & 0 & \ldots & \ldots & 1 & -a_1 \end{bmatrix}$$

The state coordinate transformation T satisfies the equation

$$X_c = TX$$

where X_c is canonical state vector, whereas X is the state vector of the original system.

Example

```
#include <control.h>

#define NUMX 4    // number of states
#define NUMU 1    // number of inputs
#define NUMY 1    // number of outputs

int main() {
    array double A[NUMX][NUMX] = {{-4.12, -17.4, -30.8, -60},
                                  {1,     0,     0,     0},
                                  {0,     1,     0,     0},
                                  {0,     0,     1,     0}},
                 B[NUMX][NUMU] = {1, 0, 0, 0},
                 C[NUMY][NUMX] = {4, 8.4, 30.78, 60};
    array double T[NUMX][NUMX];
    CControl sys, *sysc1, *sysc2;

    sys.model("ss", A, B, C, NULL);
    sysc1 = sys.canon(T, "modal");
    printf(">>>>>> State transformation for modal form :\n%f\n", T);
    sysc1->printss();
    sysc2 = sys.canon(T, "companion");
    printf(">>>>>> State transformation for companion form :\n%f\n", T);
    sysc2->printss();

    return 0;
}
```

The output is shown as follows.

```
>>>>>> State transformation for modal form :
1.335048 3.507940 11.328091 20.707772
0.183734 -2.528629 -0.024470 -19.374270
-0.269257 -3.533022 -11.409088 -20.855833
```

```
-0.922560 -2.830358 -5.755593 3.946991


State-space arguments:
A =
 -1.797086   2.213723   0.000000   0.000000
 -2.213723  -1.797086   0.000000   0.000000
  0.000000   0.000000  -0.262914   2.703862
  0.000000   0.000000  -2.703862  -0.262914
B =
  1.335048
  0.183734
 -0.269257
 -0.922560
C =
  3.434442   0.219592   0.433539   0.551465
D =
  0.000000
>>>>>> State transformation for companion form :
1.000000 4.120000 17.400000 30.800000
0.000000 1.000000 4.120000 17.400000
0.000000 0.000000 1.000000 4.120000
0.000000 0.000000 0.000000 1.000000


State-space arguments:
A =
  0.000000   0.000000   0.000000 -60.000000
  1.000000   0.000000   0.000000 -30.800000
  0.000000   1.000000   0.000000 -17.400000
  0.000000   0.000000   1.000000  -4.120000
B =
  1.000000
  0.000000
  0.000000
  0.000000
C =
  4.000000  -8.080000  -5.530400 100.177248
D =
  0.000000
```

See Also

CControl::ctrb(), **CControl::ctrbf**().

CControl ::compensatorZeroPoleGain

Synopsis
int **compensatorZeroPoleGain**(array double complex $compsys\_dp$[:],
array double complex $comp\_zero$[:],
array double complex $comp\_pole$[:],
double *$comp\_gain$);

Purpose
Compute a compensator's zero, pole, and gain based on the desired dominant pole.

Return Value
Upon successful completion, zero is returned. Otherwise, -1 is returned.

Parameters
$compsys\_dp$ Computational array of double complex containing the desired dominant pole.

$comp\_zero$ Computational array of double complex containing the compensator's zero.

$comp\_pole$ Computational array of double complex containing the compensator's pole.

$comp\_gain$ Pointer to double value denoting the compensator's gain.

Description
The function **compensatorZeroPoleGain()** computes and returns the zero, pole, and gain of a compensator based on the desired dominant pole. One of the approaches in the design of a compensator is to specify the dominant pole that can be possessed by the compensated system. This function allows a user to specify a single dominant pole or a dominant pole pair, and a real value for the compensator's zero/pole. Based on these input data, the compensator's pole/zero and gain will be calculated and returned. Note that currently we only deal with compensators with one real pole and one real zero. This is the most common situation in the design of a compensator. Therefore the currently acceptable size of the second and third arrays of this function is one. Since assumed-size computational arrays of double complex are adopted as the second and third arguments of this function, this function can be extended to solve for compensators with multiple zeros and poles of real or complex values.

Example 1
In this example, the compensator's zero and gain are calculated and returned based on the specified dominant pole and compensator's pole.

```
#include <control.h>

int main() {
  // Define the plant
  double plant_k = 4;
  array double complex plant_p[] = {0, -2};

  // Specify a pole for the dominant pole pair
  array double complex dominant_p[] ={complex(-2, 2*sqrt(3))};

  // Specify the compensator's pole/zero
  array double complex comp_p[1] = {-20}, comp_z[1];
  // Declare a double variable for the compensator’s gain
  double comp_k;

  // Declare a CControl object for the plant
  CControl plant;

  // Calculate the compensator’s pole/zero and gain
  plant.model("zpk", NULL, plant_p, plant_k);
  plant.compensatorZeroPoleGain(dominant_p, comp_z, comp_p, &comp_k);

  // Display compensator’s information
  printf("\nCompensator:\n"
         "    Pole: %f"
         "    Zero: %f"
         "    Gain: %f\n\n", real(comp_p), real(comp_z), comp_k);
  return 0;
}
```

The output of Example 1 is shown as follows.

```
Compensator:
    Pole: -20.000000
    Zero: -6.000000
    Gain: 12.000000
```

Example 2
In this example, the compensator’s pole and gain are calculated and returned based on the specified dominant pole and compensator’s zero.

```
#include <control.h>

int main() {
  // Define the plant
  double plant_k = 4;
  array double complex plant_p[] = {0, -2};

  // Specify a pole for the dominant pole pair
  array double complex dominant_p[] ={complex(-2, 2*sqrt(3))};

  // Specify the compensator’s pole/zero
  array double complex comp_p[1], comp_z[1] = {-3};

  // Declare a double variable for the compensator’s gain
  double comp_k;

  // Declare a CControl object for the plant
  CControl plant;

  // Calculate the compensator’s pole/zero and gain
  plant.model("zpk", NULL, plant_p, plant_k);
  plant.compensatorZeroPoleGain(dominant_p, comp_z, comp_p, &comp_k);

  // Display compensator’s information
  printf("\nCompensator:\n"
         "    Pole: %f"
         "    Zero: %f"
         "    Gain: %f\n\n", real(comp_p), real(comp_z), comp_k);
  return 0;
}
```

The output of Example 2 is shown as follows.

```
Compensator:
    Pole: -5.600000
    Zero: -3.000000
    Gain: 4.800000
```

See Also
CControl::getDominantPole().

CControl ::connect

Synopsis
CControl* **connect** (array int q[:][:], array int $input$[:], array int $output$[:]);

Purpose
Construct state-space model from a unconnected block-diagonal model.

Return Value
Upon successful completion, a non-NULL pointer to class **CControl** representing the generated system is returned. Otherwise, a NULL pointer is returned.

Parameters

q Array of interger which contains input and output numbers to indicate how the blocks on the diagram are connected. It has a row for each connected input of the system, where the first element is the input number. The subsequent elements specify where the corresponding input gets its summing inputs. Negative elements indicate minus inputs.

$input$ Array of interger which contains the indices indicating which of the inputs in the unconnected system are used as external inputs in the connected system.

$output$ Array of interger which contains the indices indicating which of the outputs in the unconnected system are used as external outputs in the connected system.

Description
The function **connect** () constructs state-space model of a large scale system by connecting inputs and outputs of a set of small subsystems. Given all subsystems that can be expressed in state-space, transfer function, or zero-pole-gain models, the user need to use member function **CControl** ::**append** () first to append these subsystem together and construct an unconnected block-diagonal state-space model. The function **connect** () then can be used to make the selected inputs and outputs connected. Note that the count of the inputs and outputs in Ch Control System Toolkit begins with 1.

Example
This example constructs the state-space model of the system shown in Figure A.1. The system consists of

three subsystems. The transfer functions of sys1 and sys3 are

$$\begin{aligned} G_1(s) &= \frac{6(s+3)}{(s+2)(s+5)} \\ G_3(s) &= \frac{1}{s^2+2s+1} \end{aligned}$$

The system matrices of sys2 are

$$A = \begin{bmatrix} -9.0201 & 17.7791 \\ -1.6943 & 3.2138 \end{bmatrix}, B = \begin{bmatrix} -0.5112 & 0.5362 \\ -0.002 & -1.8471 \end{bmatrix}$$

$$C = \begin{bmatrix} -3.2897 & 2.4544 \\ -13.5009 & 18.0745 \end{bmatrix}, D = \begin{bmatrix} -0.5476 & -0.1410 \\ -0.6459 & 0.2958 \end{bmatrix}$$

The inputs u1 and u2 are external inputs of the system. The outputs y2 and y3 are external outputs of the system. The input 3 is connected to the outputs 1, and 4, where the input from output 4 is negative, the corresponding row of q is 3, 1, -4. The internal input 4 is from output 3.

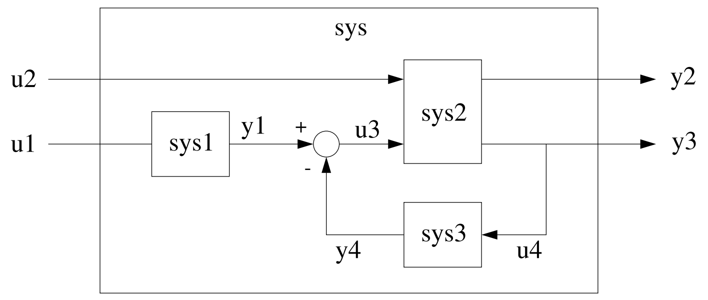


Figure A.1: Block diagram of the system.

```
#include <control.h>

#define NUMX 2    // number of states
#define NUMU 2    // number of inputs
#define NUMY 2    // number of outputs

int main() {
    array double complex z[1] = {-3}, p[2] = {-2, -5};
    double k = 6;
    array double A[NUMX][NUMX] = {{-9.0201,  17.7791},
                                  {-1.6943,  3.2138}},
                 B[NUMX][NUMU] = {{-.5112, .5362},
                                  {-.002,  -1.8471}},
                 C[NUMY][NUMX] = {{-3.2897,  2.4544},
                                  {-13.5009,  18.0745}},
                 D[NUMY][NUMU] = {{-.5476,  -.1410},
                                  {-.6459,  .2958}};
    array double num[1] = {1}, den[3] = {1, 2, 1};
    array int q[2][3] = {{3, 1, -4},
                         {4, 3, 0}},
              input[2] = {1, 2},
              output[2] = {2, 3};
    CControl sys1, sys2, sys3, *sys, *sysc;
    sys1.model("zpk", z, p, k);
    sys2.model("ss", A, B, C, D);
    sys3.model("tf", num, den);
    sys = sys1.append(&sys2, &sys3);
    sysc = sys->connect(q, input, output);
    sysc->printss();

    return 0;
}
```

The output is shown as follows.

```
State-space arguments:
A =
 -7.000000 -10.000000   0.000000   0.000000   0.000000   0.000000
  1.000000   0.000000   0.000000   0.000000   0.000000   0.000000
  3.217200   9.651600  -9.020100  17.779100   0.000000  -0.536200
-11.082600 -33.247800  -1.694300   3.213800   0.000000   1.847100
  1.774800   5.324400 -13.500900  18.074500  -2.000000  -1.295800
  0.000000   0.000000   0.000000   0.000000   1.000000   0.000000
B =
  1.000000   0.000000
  0.000000   0.000000
  0.000000  -0.511200
  0.000000  -0.002000
  0.000000  -0.645900
  0.000000   0.000000
C =
 -0.846000  -2.538000  -3.289700   2.454400   0.000000   0.141000
  1.774800   5.324400 -13.500900  18.074500   0.000000  -0.295800
D =
  0.000000  -0.547600
  0.000000  -0.645900
```

See Also
CControl::append(), **CControl::feedback**(), **CControl::connect**(), **CControl::parallel**(), **CControl::series**().

CControl ::ctrb

Synopsis
int ***ctrb** (array double &*co*);

Purpose
Compute the controllability matrix.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters
co Output controllability matrix.

Description
The function **ctrb** () computes the controllability matrix from system matrix and input matrix. Assume that

the system has an $n \times n$ system matrix A and an $n \times m$ input matrix B, **ctrb** (A,B) returns the $n \times nm$ controllability matrix C_o

$$\begin{bmatrix} B & AB & A^2B & \dots & A^{(n-1)}B \end{bmatrix}$$

The system is controllable if C_o has full rank n.

Example

```
#include <control.h>

#define NUMX 2        // number of states
#define NUMU 2        // number of inputs
#define NUMY 1        // number of outputs

int main() {
     array double A[NUMX][NUMX] = {-0.5572,   -0.7814,
                                    0.7814,    0};
     array double B[NUMX][NUMU] = {1, 0,
                                   0, 2 };
     array double co[NUMX][NUMX*NUMU];
     CControl sys;
     int num;

     sys.model("ss", A, B, NULL, NULL);
     sys.ctrb(co);
     num = rank(co);
     if(num == min(NUMX, NUMX*NUMU))
       printf("The system is controllable.\n");
     else
       printf("The system is not controllable.\n");
     printf("co = %f\n", co);
     return 0;
}
```

The output is shown as follows.

```
The system is controllable.
co = 1.000000 0.000000 -0.557200 -1.562800
0.000000 2.000000 0.781400 0.000000
```

See Also
CControl::obsv(), **CControl::ctrbf**().

CControl ::ctrbf

Synopsis
int ***ctrbf** (array double &$abar$, array double &$bbar$, array double &$cbar$, array double &t, array int &k, array double &a, array double &b, array double &c, . . . /* double tol */);

Purpose
Compute the controllability staircase form.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters

$abar$ Computational array of double containing the system matrix in controllability staircase form.

$bbar$ Computational array of double containing the input matrix in controllability staircase form.

$cbar$ Computational array of double containing the output matrix in controllability staircase form.

t Computational array of double containing similarity transformation matrix from the original form to controllability staircase form. The dimension of t is $n_x \times n_x$, where n_x is the number of state variables of original system.

k Computational array of int containing the number of controllable states identified at each iteration of the algorithm. The array k has n_x elements. The number of controllable states equals the sum of k.

a Computational array of double containing the system matrix in original system.

b Computational array of double containing the input matrix in original system.

c Computational array of double containing the output matrix in original system.

... The following argument is optional.

tol A double value used to calculate the controllable/uncontrollable subspaces.

Description
The function **ctrbf** () decomposes the original state-space system represented by (A, B, C) into the controllability staircase form ($\bar{A}$, $\bar{B}$, $\bar{C}$) which satisfies

$$\bar{A} = TAT^T, \bar{B} = TB, \bar{C} = CT^T$$

and

$$\bar{A} = \begin{bmatrix} A_{uc} & 0 \\ A_{21} & A_c \end{bmatrix}, \bar{B} = \begin{bmatrix} 0 \\ B_c \end{bmatrix}, \bar{C} = \begin{bmatrix} C_{uc} & C_c \end{bmatrix}$$

where (A_c, B_c) is controllable subspace, and $A_u c$ is uncontrollable states. The dimension of $\bar{A}$, $\bar{B}$, and $\bar{C}$ are the same as those of A, B, and C, respectively. The input/output characteristics of the original system and its controllability staircase form are exactly the same; that is, $C_c(sI - A_c)^{-1}B_c = C(sI - A)^{-1}B$. For the user's convenience, an object of empty model can be created to call this member function.

Example

```
#include <control.h>

#define NUMX 2    // number of states
#define NUMU 2    // number of inputs
#define NUMY 2    // number of outputs

int main() {
    array double A[NUMX][NUMX] = {1,1,4,-2},
                 B[NUMX][NUMU] = {1, -1, 1, -1},
                 C[NUMY][NUMX] = {1, 0, 0 ,1};
    array double Abar[NUMX][NUMX], Bbar[NUMX][NUMU], Cbar[NUMY][NUMX],
                 t[NUMX][NUMX];
    array int k[NUMX];
    CControl sys;

    sys.ctrbf(Abar, Bbar, Cbar, t, k, A, B, C);
    printf("Abar = %f\nBbar = %f\nCbar = %f\nt = %f\nk = %d\n", Abar, Bbar, Cbar, t, k);
    printf("tAt’ = %f\ntB = %f\nCt’ = %f\n", t*A*transpose(t), t*B, C*transpose(t));
    return 0;
}
```

The output is shown as follows.

```
Abar = -3.000000 0.000000
3.000000 2.000000

Bbar = 0.000000 0.000000
-1.414214 1.414214

Cbar = -0.707107 -0.707107
0.707107 -0.707107

t = -0.707107 0.707107
-0.707107 -0.707107

k = 1 0

tAt’ = -3.000000 0.000000
3.000000 2.000000

tB = 0.000000 0.000000
-1.414214 1.414214

Ct’ = -0.707107 -0.707107
0.707107 -0.707107
```

See Also
CControl::ctrb().

CControl ::d2c

Synopsis
CControl* **d2c** (. . . /* string_t method */);

Purpose
Converts a state space model from the discrete-time system to the continuous-time system.

Return Value
Upon successful completion, a non-NULL pointer to class **CControl** representing the resulting continuous-time generated system is returned. Otherwise, a NULL pointer is returned.

Parameters
. . . The following argument is optional.

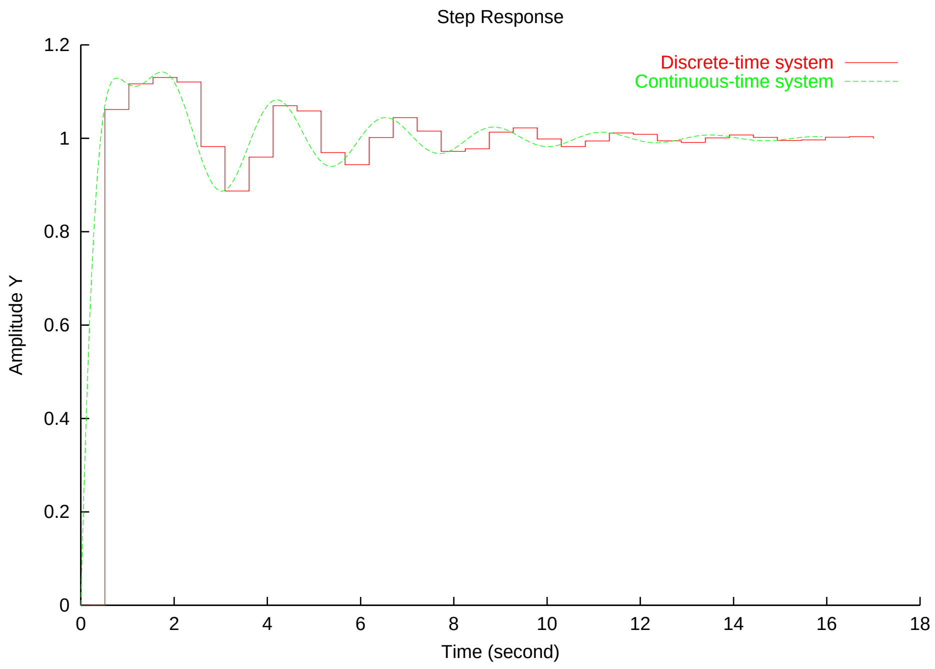
$method$ Selected conversion method among `"zoh"`, `"tustin"`, `"prewarp"`, and `"matched"`. `"zoh"` standing for `"Zero-order hold"` is the default and the only method available in the current implementation.

Description
The function **d2c** () converts discrete-time models to continuous-time models using specified conversion methods such as "zero-order hold".

Example
The following example produces a continuous-time system $sysc$ from the discrete-time one $sysd$ using "zero-order hold" as the conversion method. The differences between these two systems are illustrated by the comparison of their step responses in the generated figure.

```
#include <control.h>

#define NUMX 4    // number of states
#define NUMU 1    // number of inputs
#define NUMY 1    // number of outputs
#define N1   34   // number of time samples for discrete-time system
                  // 17 seconds for ts = 0.5
#define N2   500  // number of time samples for continuous-time system

int main() {
    array double A[NUMX][NUMX] =
                    { -0.491495,  -3.142023,  -4.676251,  -3.624473,
                      0.060408,  -0.242614,  -2.090926,  -2.815688,
                      0.046928,   0.253752,   0.573935,  -0.645539,
                      0.010759,   0.091255,   0.440958,   0.905312,
                    },
                 B[NUMX][NUMU] = {  0.060408,
                                    0.046928,
                                    0.010759,
                                    0.001578,
                                 },
                 C[NUMY][NUMX] = {4, 8.4, 30.78, 60};

    array double y1[N1], t1[N1];
    array double y2[N2], t2[N2];
    CPlot plot;
    CControl *sysc;

    CControl sysd;
    sysd.model("ss", A, B, C, NULL, 0.5);
    sysd.step(NULL, y1, t1, NULL);
    plot.data2D(t1, y1);

    sysc = sysd.d2c();
    sysc->step(NULL, y2, t2, NULL);
    plot.data2D(t2, y2);

    plot.plotType(PLOT_PLOTTYPE_STEPS, 0);
    plot.label(PLOT_AXIS_X, "Time (second)");
    plot.label(PLOT_AXIS_Y,"Amplitude Y");
    plot.title("Step Response");
    plot.legend("Discrete-time system", 0);
    plot.legend("Continuous-time system", 1);
    plot.plotting();

    return 0;
}
```

The step responses of both systems are shown as follows.



See Also
CControl::c2d(), **CControl::d2d**(), **CControl::ss2ss**().

CControl ::d2d

Synopsis
CControl* d2d (double ts);

Purpose
Resample a discrete-time LTI model.

Return Value
Upon successful completion, a non-NULL pointer to class **CControl** representing the resulting system is returned. Otherwise, a NULL pointer is returned.

Parameters
ts Sampling time expressed in second for generating the new discrete-time system.

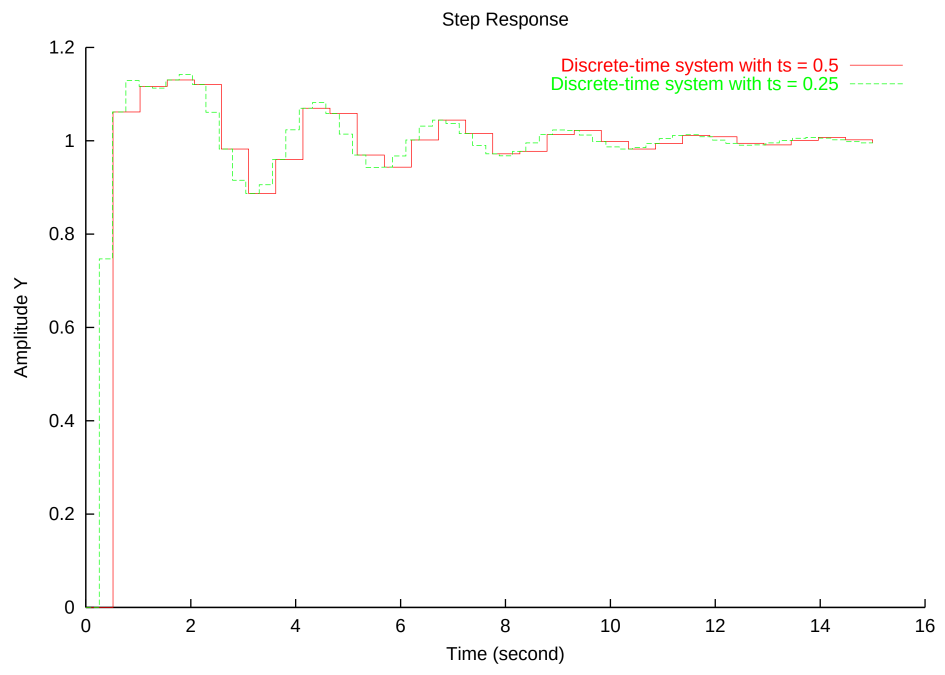
Description
The function **d2d** () generates a discrete-time system with different sampling time.

Example
The following example produces a continuous-time system $sysd2$ with sampling time $0.25s$ from the $sysd$ with sampling time $0.5s$. The difference between these two systems are illustrated by the comparison of

their step responses in the generated figure.

```
#include <control.h>

#define NUMX 4   // number of states
#define NUMU 1   // number of inputs
#define NUMY 1   // number of outputs
#define N1   30  // number of time samples for discrete-time system with ts=0.5
                 // 15 seconds for ts = 0.5
#define N2   60  // number of time samples for discrete-time system with ts=0.25
                 // 15 seconds for ts = 0.25


int main() {
    array double A[NUMX][NUMX] =
                    { -0.491495,  -3.142023,  -4.676251,  -3.624473,
                      0.060408,  -0.242614,  -2.090926,  -2.815688,
                      0.046928,   0.253752,   0.573935,  -0.645539,
                      0.010759,   0.091255,   0.440958,   0.905312,
                    },
                 B[NUMX][NUMU] = {  0.060408,
                                    0.046928,
                                    0.010759,
                                    0.001578,
                                 },
                 C[NUMY][NUMX] = {4, 8.4, 30.78, 60};

    array double y1[N1], t1[N1];
    array double y2[N2], t2[N2];
    CPlot plot;
    CControl sysd, *sysd2;

    sysd.model("ss", A, B, C, NULL, 0.5);
    sysd.step(NULL, y1, t1, NULL);
    plot.data2D(t1, y1);

    sysd2 = sysd.d2d(0.25);
    sysd2->step(NULL, y2, t2, NULL);
    plot.data2D(t2, y2);

    plot.plotType(PLOT_PLOTTYPE_STEPS, 0);
    plot.plotType(PLOT_PLOTTYPE_STEPS, 1);
    plot.label(PLOT_AXIS_X, "Time (second)");
    plot.label(PLOT_AXIS_Y,"Amplitude Y");
    plot.title("Step Response");
    plot.legend("Discrete-time system with ts = 0.5", 0);
    plot.legend("Discrete-time system with ts = 0.25", 1);
    plot.plotting();

    return 0;
}
```

The step responses of both systems are shown as follows.



See Also
CControl::c2d(), **CControl::d2c**(), **CControl::ss2ss**().

CControl ::damp

Synopsis
int **damp** (. . . /* array double &wn, array double &z, array double complex &p */);

Purpose
Compute the damping factors and natural frequencies of system poles.

Return Value
Upon successful completion, 0 is returned. Otherwise, -1 is returned.

Parameters
. . . Variable length argument list

wn Array of reference containing the natural frequencies of system poles.

z Array of reference containing the damping factors of system poles.

p Array of reference containing system poles.

Description
The function **damp** () computes the damping factors and natural frequencies of system poles. When invoked without arguments, the function outputs the system poles, along with their damping factors and natural frequencies through standard output stream stdout. The poles are displayed in ascending order. If the function is called with two arguments, the natural frequencies and damping factors of system poles are passed by arrays wn and z to the calling function. The function can also pass system poles by the third argument. The function applies to SISO cases only in the current implementation.

Example
This example dispalys the poles, natural frequencies, and damping factors of the following system

$$\frac{2s^2+5s+1}{s^2+2s+3}$$

```
#include <control.h>

int main() {
   array double num[3] = {2, 5, 1}, den[3] = {1, 2, 3};
   CControl sys;
   int np;

   sys.model("tf", num, den);
   np = sys.size('p');

   array double complex p[np];
   array double z[np];
   array double wn[np];

   sys.damp();  /* sys.damp(p, z, wn); */
   return 0;
}
```

The output of this example is shown as follows.

```
Damping
0.5774 0.5774

Natural Frequency (rad/s)
1.732 1.732

Eigenvalue
complex(-1,1.414) complex(-1,-1.414)
```

See Also
CControl::esort(), **CControl::pole**(), **CControl::pzmap**().

CControl ::dcgain

Synopsis
array double **dcgain** ()[:][:];

Purpose
Compute low frequency (DC) gain of the system.

Return Value
Upon successful completion, a computation array representing the array of the DC gain is returned. Otherwise, a NULL pointer is returned.

Parameters
None.

Description
The function **dcgain** () computes low frequency (DC) gain k of the system using formula $K = D - CA^{-1}B$ for continuous-time system or $K = D + C(I - A)^{-1}B$ for discrete-time system.

Example
In this example, DC gains of continuous-time system sys and discrete-time system $sysd$ are calculated.

```
#include <control.h>

#define NUMX 4    // number of states
#define NUMU 2    // number of the inputs
#define NUMY 2    // number of the outputs
#define TF   15   // final time

int main() {
    array double A[NUMX][NUMX] = {{-4.12, -17.4, -30.8, -60},
                                  {1,     0,     0,     0},
                                  {0,     1,     0,     0},
                                  {0,     0,     1,     0}},
                 B[NUMX][NUMU] = {1, 0, 0, 1, .5, .3, 0, .11},
                 C[NUMY][NUMX] = {4, 8.4, 30.78, 60,
                                  3, 2, 1, 5};
    CPlot plot;
    array double K[NUMY][NUMU];
    CControl sys, sysd;

    sys.model("ss", A, B, C, NULL);
    sysd.model("ss", A, B, C, NULL, -1);
    K = sys.dcgain();
    printf("K = %f\n", K);
    K = sysd.dcgain();
    printf("K = %f\n", K);

    return 0;
}
```

The output is shown as follows.

```
K = 5.500000 2.822200
-0.191667 -2.649333

K = 4.962962 3.683898
-1.309919 -3.437857
```

See Also
CControl::bandwidth().

CControl ::delay2z

Synopsis
CControl* delay2z ();

Purpose
Generate a discrete-time system in which all delays in the original system are replaced by poles at z=0.

Return Value
Upon successful completion, a non-NULL pointer to class **CControl** representing the generated system is returned. Otherwise, a NULL pointer is returned.

Parameters
None.

Description
The function **delay2z** () replaces all delays in the original system by poles at z=0.

Example
In this example, the system $sysd1$ has input delay $2s$.

```
#include <control.h>

int main() {
   array double complex zero[2]={2, 3},
                        pole[2]={1, -1};
   double k = 3;

   CControl *sysd2;
   CControl sysd1;
   sysd1.model("zpk", zero, pole, k, -1);
   sysd1.setDelay(1, 2);
   sysd1.printSystem();

   sysd2 = sysd1.delay2z();
   sysd2->printSystem();
   return 0;
}
```

The output is shown as follows.

```
Discrete-time System with unspecified sample time
State-space arguments:
A =
 -0.000000   1.000000
  1.000000   0.000000
B =
  1.000000
  0.000000
C =
-15.000000  21.000000
D =
  3.000000

Transfer function parameters:
Numerator:
  3.000000 -15.000000  18.000000
Denominator:
  1.000000   0.000000  -1.000000

Zero-Pole-Gain parameters:
Zero:
complex(  2.000000,  0.000000) complex(  3.000000,  0.000000)
Pole:
```

```
complex(  1.000000,  0.000000) complex( -1.000000,  0.000000)
Gain:   3.000000

input delay is   2.000000
output delay is   0.000000
input/output delay is   0.000000

Discrete-time System with unspecified sample time
State-space arguments:
A =
 -0.000000   1.000000  -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   0.000000
B =
  1.000000
  0.000000
  0.000000
  0.000000
C =
  0.000000   3.000000 -15.000000  18.000000
D =
  0.000000

Transfer function parameters:
Numerator:
  0.000000   0.000000   3.000000 -15.000000  18.000000
Denominator:
  1.000000   0.000000  -1.000000   0.000000   0.000000

Zero-Pole-Gain parameters:
Zero:
complex(  2.000000,  0.000000) complex(  3.000000,  0.000000)
Pole:
complex(  0.000000,  0.000000) complex(  0.000000,  0.000000) complex(  1.000000,  0.000000)
complex( -1.000000,  0.000000)
Gain:   3.000000

input delay is   0.000000
output delay is   0.000000
input/output delay is   0.000000
```

See Also
CControl::hasdelay().

CControl ::dlqr

Synopsis
int **dlqr** (array double &k, array double &s, array double complex &e, array double &q, array double &r, . . . /* array double &n */);

Purpose
Build linear-quadratic (LQ) state-feedback regulator for discrete-time plant.

Return Value

Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters

k Array of reference containing the output optimal gain matrix K for state-feedback law

$$u[n] = -Kx[n]$$

s Array of reference containing the solution of discrete-time Riccati equation.

$$A^TSA - S - (A^TSB + N)(B^TSB + R)^{-1}(B^TSA + N^T) + Q = 0$$

e Array of reference containing the closed-loop eigenvalues.

q Array of reference containing parameter Q in the following cost function J

$$J(u) = \sum_{n=1}^{\infty}(x[n]^TQx[n] + u[n]^TRu[n] + 2x[n]^TNu[n])$$

r Array of reference containing parameter R in the above cost function J.

... The following argument is optional.

n Array of reference containing parameter N in the above cost function J. It is set to zero when omitted.

Description

The function **dlqr** () can be used to design linear-quadratic (LQ) state-feedback regulator for discrete-time plants shown as follows,

$$\begin{aligned} x[n+1] &= Ax[n] + Bu[n] \\ y[n] &= Cx[n] + Du[n] \end{aligned}$$

and a cost function

$$J(u) = \sum_{n=1}^{\infty}(x[n]^TQx[n] + u[n]^TRu[n] + 2x[n]^TNu[n])$$

The purpose of Linear Quadratic Gaussian (LQG) control scheme is to find an optimal gain matrix K, so that the state-feedback law

$$u[n] = -Kx[n]$$

minimizes the above cost function $J(u)$. The gain K is called the LQ optimal gain. To obtain the LQ optimal gain, the user needs to solve a discrete-time algebraic Riccati equation below

$$A^TSA - S - (A^TSB + N)(B^TSB + R)^{-1}(B^TSA + N^T) + Q = 0$$

Then, K can be derived from the solution S of the above equation by

$$K = (B^TSB + R)^{-1}(B^TSA + N^T)$$

Example

```
#include <control.h>

#define NUMX 2    // number of states
#define NUMU 1    // number of inputs

int main() {
    array double A[NUMX][NUMX] = {{-1,     0},
                                  {0,     -2}},
                 B[NUMX][NUMU] = {1, 1};
    CPlot plot;
    array double k[NUMU][NUMX], s[NUMX][NUMX];
    array double complex e[NUMX];
    array double Q[NUMX][NUMX] = {1, 1, 1, 1};
    array double R[NUMU][NUMU] = {5};
    array double N[NUMX][NUMU] = {10, 20};
    CControl sys;

    sys.model("ss", A, B, NULL, NULL, .1);
    sys.dlqr(k, s, e, Q, R, N);
    printf("k = %f\ns = %f\ne = %f\n", k, s, e);

    return 0;
}
```

The output is shown as follows.

```
k = 0.078506 -1.848855

s = 21.812674 -24.550627
-24.550627 184.544003

e = complex(-0.350526,0.000000) complex(-0.879124,0.000000)
```

See Also
CControl::lqr().

CControl ::dlyap

Synopsis
int **dlyap** (array double complex x[&][&], array double complex a[&][&], array double complex c[&][&]);

Purpose
Solve the discrete Lyapunov equation.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters

x Array of reference containing the solution of the Lyapunov equation.

a Array of reference containing the parameter A of Lyapunov matrix equation.

c Array of reference containing the parameter C of Lyapunov matrix equation.

Description
The function **dlyap** () solves the following discrete-time Lyapunov equation,

$$A^T XA - X + Q = 0$$

where X is the solution, and parameters A and Q are $n \times n$ matrices. For the user's convenience, an object of empty model can be created to call this member function.

Example

```
#include <control.h>

int main() {
     array double complex x[2][2];
     array double a[2][2] = { -0.557200, -0.781400,
                              0.781400, 0.000000},
                  q[2][2] = {1.000000, 0.000000,
                             0.000000, 4.000000};
     CControl sys;

     sys.dlyap(x, a, q);
     printf("x = %f\n", x);
     return 0;
}
```

The output is shown as follows.

```
x = complex(6.234801,0.000000) complex(-1.685479,0.000000)
complex(-1.685479,0.000000) complex(7.806882,0.000000)
```

See Also
CControl::lyap().

CControl ::dsort

Synopsis
array double complex **dsort** ()[:];

Purpose
Sort the poles of discrete-time systems.

Return Value
Upon successful completion, a one-dimensional computational array containing the sorted poles is returned.

Parameters
None.

Description
The function **dsort** () sorts the poles of a discrete-time system in descending order by magnitude that makes unstable poles appear first.

Example

```
#include <control.h>

int main() {
   array double complex zero[3]={complex(-0.05, 2.738156),
                                 complex(-0.05, -2.738156),
                                 -2.0},
   pole[6]={ 0.0001, -9.5,
     complex(1.797086, -2.213723), complex(1.797086, 2.213723),
     complex(-0.262914, 2.703862), complex(-0.262914, -2.703862)};
   double k=4;
   array double complex pp[6];
   CControl sys;

   sys.model("zpk", zero, pole, k, -1);
   pp = sys.dsort();
   printf("pp = %f\n", pp);
   return 0;
}
```

The output is shown as follows.

```
pp = complex(-9.500000,0.000000) complex(1.797086,2.213723) complex(1.797086,-2.213723)
complex(-0.262914,2.703862) complex(-0.262914,-2.703862) complex(0.000100,0.000000)
```

See Also
CControl::dsort(), **CControl::pzmap**().

CControl ::esort

Synopsis
array double complex **esort** ()[:];

Purpose
Sort the poles of continuous-time systems.

Return Value
Upon successful completion, a one-dimensional computational array containing the sorted poles is returned.

Parameters
None.

Description
The function **esort** () sorts the poles of a continuous-time system in descending order by real part that makes unstable poles appear first.

Example

```
#include <control.h>
int main() {
   array double complex zero[3]={complex(-0.05, 2.738156),
                                 complex(-0.05, -2.738156),
                                -2.0},
   pole[6]={ 0.0001,
     complex(1.797086, -2.213723),
     1.797086,
     complex(1.797086, 2.213723),
     complex(-0.262914, 2.703862), complex(-0.262914, -2.703862)};
   double k=4;
   array double complex pp[6];
   CControl sys;

   sys.model("zpk", zero, pole, k);
   pp = sys.esort();
   printf("pp = %f\n", pp);
   return 0;
}
```

The output is shown as follows.

```
pp = complex(1.797086,2.213723) complex(1.797086,-2.213723) complex(1.797086,0.000000)
complex(0.000100,0.000000) complex(-0.262914,2.703862) complex(-0.262914,-2.703862)
```

See Also
CControl::dsort(), **CControl::pzmap**().

CControl ::estim

Synopsis
CControl* **estim** (array double &$gain$, . . . /* array int $sensors$[:], array int $known$[:] */);

Purpose
Produce a state/output estimator.

Return Value
Upon successful completion, a non-NULL pointer to class **CControl** representing the estimator is returned. Otherwise, a NULL pointer is returned.

Parameters
$gain$ Computational array containing the estimator gain.

. . . Following arguments are optional.

$sensors$ Computational array containing indices specify which outputs are measured.

$known$ Computational array containing indices specify which inputs are known.

Description
The function **estim** () produces a state/output estimator based on a state-space model and given estimator gain. If the optional arguments are absent, all inputs of the system are assumed stochastic, and all outputs are measured. For the system expressed in the following model,

$$\dot{X} = AX + BU$$
$$Y = CX + DU$$

the estimated states $\hat{X}$ and outputs $\hat{Y}$ are given from

$$\dot{\hat{X}} = A\hat{X} + L(Y - C\hat{X})$$
$$\begin{bmatrix} \hat{Y} \\ \hat{X} \end{bmatrix} = \begin{bmatrix} C \\ I \end{bmatrix} X$$

If the user specifies the measured outputs Y and known inputs U by passing additional argument $sensors$ and $known$, the state-space model of the plant can be expressed as follows,

$$\begin{aligned} \dot{X} &= AX + B_1W + B_2U \\ \begin{bmatrix} Z \\ Y \end{bmatrix} &= \begin{bmatrix} C_1 \\ C_2 \end{bmatrix} X + \begin{bmatrix} D_{11} \\ D_{21} \end{bmatrix} W + \begin{bmatrix} D_{12} \\ D_{22} \end{bmatrix} U \end{aligned}$$

where W are stochastic inputs and Z are nonmeasured outputs. In this case, the estimated states $\hat{X}$ and outputs $\hat{Y}$ are given from

$$\dot{\hat{X}} = A\hat{X} + L(Y - C_2\hat{X} - D_{22}U)$$
$$\begin{bmatrix} \hat{Y} \\ \hat{X} \end{bmatrix} = \begin{bmatrix} C_2 \\ I \end{bmatrix} X + \begin{bmatrix} D_{22} \\ 0 \end{bmatrix} U$$

Example

```
#include <control.h>

#define NUMX 4    // number of states
#define NUMU 3    // number of inputs
#define NUMY 3    // number of outputs

int main() {
    array double A[NUMX][NUMX] = {{-4.12, -17.4, -30.8, -60},
                                   {1,     0,     0,     0},
                                   {0,     1,     0,     0},
                                   {0,     0,     1,     0}},
                 B[NUMX][NUMU] = {1, 0, 0,
                                   0, 1, 0,
                                   0, 0, 1,
                                   1, 2, 3},
                 C[NUMY][NUMX] = {4, 8.4, 30.78, 33,
                                   60, 2.1, 5, 78,
                                   5.5, 6.7, 111, 21},
                 D[NUMY][NUMU] = {1, 2, 3,
                                   4, 5, 6,
                                   7, 8, 9};
    array int sensors[2] = {2, 0},
              known[2] = {1, 2};
    array double l[NUMX][NUMY] = {2, 1, 3,
                                   55, 8, 10,
                                   33, 43, 8,
                                   101, 9, 30};
    CControl sys, *est;
    sys.model("ss", A, B, C, D);
    est = sys.estim(l, sensors, known);
    est->printSystem();
    return 0;
}
```

The output is shown as follows.

```
Continuous-time System
State-space arguments:
A =
-19.120000 -39.200000 -283.580000 -135.000000
-235.500000 -482.100000 -2025.900000 -1878.000000
-84.000000 -136.600000 -1195.800000 -498.000000
-353.500000 -582.300000 -4985.540000 -2112.000000
B =
-18.000000 -21.000000   2.000000   1.000000
-133.000000 -192.000000   3.000000  55.000000
-84.000000 -101.000000   8.000000  10.000000
-348.000000 -423.000000  33.000000  43.000000
C =
  5.500000   6.700000 111.000000  21.000000
  4.000000   8.400000  30.780000  33.000000
  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
D =
  8.000000   9.000000   0.000000   0.000000
  2.000000   3.000000   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
```

See Also
CControl::place().

CControl ::feedback

Synopsis
CControl* **feedback** (**CControl** *sys2, . . . /* array int feedin[:], array int feedout[:], int sign */);

Purpose
Feedback connection of two LTI models.

Return Value
Upon successful completion, a non-NULL pointer to class **CControl** representing the generated system is returned. Otherwise, a NULL pointer is returned.

Parameters
$sys2$ Pointer to the class **CControl** representing the system to be connected in the feedback path.

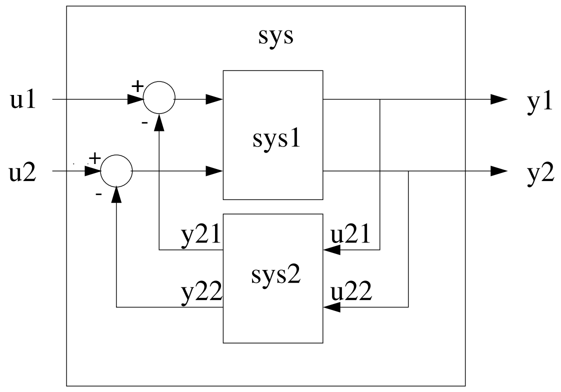
. . . The following arguments are optional.

$feedin$ One-dimensional computational array containing indices specifying which inputs are involved in the feedback loop. For SISO systems, the default value is $\{1\}$. For MIMO systems with m inputs, the default value of the array is {1, 2, 3, ..., m}.

$feedout$ One-dimensional computational array containing indices specifying which outputs are involved in the feedback loop. For SISO systems, the default value is $\{1\}$. For MIMO systems with n outputs, the default value of the array is {1, 2, 3, ..., n}.

$sign$ Integer indicating every element of input is a positive feedback or a negative one. The value 1 stands for a positive feedback, whereas value -1 a negative feedback. The default value of this argument is -1.

Description
The function **feedback** () builds a system with feedback interconnection like the following figure,



The resulting system has the same inputs and outputs as $sys1$. The arguments $feedin$ and $feedout$ contain indices specifying which inputs and outputs are involved in the feedback loop, respectively. Note that the count of the feedin and feedout in Ch Control System Toolkit begins with 1.
Example

```
#include <control.h>

#define NUMX 4    // number of states
#define NUMU 2    // number of inputs
#define NUMY 2    // number of outputs

int main() {
    array double A[NUMX][NUMX] = {{-4.12, -17.4, -30.8, -60},
                                  {1,      0,     0,     0},
                                  {0,      1,     0,     0},
                                  {0,      0,     1,     0}},
                 B[NUMX][NUMU] = {1, 0,
                                  2, 0,
                                  0, 0,
                                  0, 1},
                 C[NUMY][NUMX] = {4, 8.4, 30.78, 60,
                                  3, 7.4, 29.78, 50},
                 D[NUMY][NUMU] = {1, 2,
                                  4, 5};
    array int feedin[2] = {1, 2}, feedout[2] = {1, 2};
    CControl sys1, sys2, *sys3;
    sys1.model("ss", A, B, C, D);
    sys2.model("ss", A, B, C, D);
    sys3 = sys1.feedback(&sys2, feedin, feedout);
    sys3->printss();

    return 0;
}
```

The output is shown as follows.

```
State-space arguments:
A =
 -3.504615 -16.276923 -27.094615 -51.538462  -1.923077  -3.784615 -13.253077
-27.692308
  2.230769   2.246154   7.410769  16.923077  -3.846154  -7.569231 -26.506154
-55.384615
  0.000000   1.000000   0.000000   0.000000   0.000000   0.000000   0.000000
  0.000000
 -1.346154  -2.869231  -9.616154 -20.384615   1.269231   2.453846   8.479231
 18.076923
  1.923077   3.784615  13.253077  27.692308  -3.504615 -16.276923 -27.094615
-51.538462
  3.846154   7.569231  26.506154  55.384615   2.230769   2.246154   7.410769
 16.923077
  0.000000   0.000000   0.000000   0.000000   0.000000   1.000000   0.000000
  0.000000
 -1.269231  -2.453846  -8.479231 -18.076923  -1.346154  -2.869231  -9.616154
-20.384615
B =
  0.653846  -0.230769
  1.307692  -0.461538
  0.000000   0.000000
 -0.461538   0.192308
 -0.269231   0.153846
 -0.538462   0.307692
  0.000000   0.000000
  0.307692   0.038462
C =
  1.923077   3.784615  13.253077  27.692308   0.615385   1.123077   3.705385
  8.461538
 -1.269231  -2.453846  -8.479231 -18.076923  -1.346154  -2.869231 -10.616154
-20.384615
D =
 -0.269231   0.153846
  0.307692   0.038462
```

See Also
CControl::connect(), **CControl::parallel**(), **CControl::series**().

CControl ::freqresp

Synopsis
array double complex **freqresp** (array double w[&])[:];

Purpose

Frequency response function.

Return Value

Upon successful completion, a computational array containing frequency response at the specified frequency points is returned.

Parameters

w Computational array containing frequency points at which the system responses are evaluated.

Description

The function **freqresp** () computes frequency response of a system at given frequency points. The response at a frequency ω of system (A, B, C, D) is given by

$$H(j\omega) = D + C(j\omega I - A)^{-1}B$$

Example

```
#include <control.h>

int main() {
   array double num[3] = {1, 0.1, 7.5}, den[5] = {1, 0.12, 9, 0, 0};
   CControl sys;
   array double w[10];
   array double complex h[10];
   int i;

   sys.model("tf", num, den);
   logspace(w, -1, 1);
   h = sys.freqresp(w);
   for(i = 0; i < 10; i++) {
     printf("w = %f  ", w[i]);
     printf("h(w) = %f\n", h[i]);
   }
   return 0;
}
```

The output is shown as follows.

```
w = 0.100000  h(w) = complex(-83.314792,-0.000025)
w = 0.166810  h(w) = complex(-29.929872,-0.000041)
w = 0.278256  h(w) = complex(-10.744235,-0.000070)
w = 0.464159  h(w) = complex(-3.849019,-0.000120)
w = 0.774264  h(w) = complex(-1.370246,-0.000219)
w = 1.291550  h(w) = complex(-0.476849,-0.000480)
w = 2.154435  h(w) = complex(-0.141430,-0.002260)
w = 3.593814  h(w) = complex(-0.106578,-0.004632)
w = 5.994843  h(w) = complex(-0.029371,-0.000165)
w = 10.000000  h(w) = complex(-0.010165,-0.000024)
```

See Also

CControl::bode(), **CControl::nyquist**(), **CControl::nichols**().

CControl ::getDominantPole

Synopsis
int **getDominantPole**(double *$percent\_overshoot$,
double *$damping\_ratio$,
array double complex $dominant\_pole$[:],
double *$extra\_gain$,
double *$natural\_frequency$);

Purpose
Based on the desired percent overshoot/damping ratio, compute the corresponding dominant pole/dominant pole pair, extra gain, natural frequency, and damping ratio/percent overshoot.

Return Value
Upon successful completion, zero is returned. Otherwise, -1 is returned.

Parameters
$percent\_overshoot$ Pointer to double value denoting the desired percent overshoot.

$damping\_ratio$ Pointer to double value denoting the desired damping ratio.

$dominant\_pole$ Computational array of double complex containing the dominant pole pair or a single pole of the
dominant pole pair.

$extra\_gain$ Pointer to double value denoting the extra gain that can be added to the system to yield the dominant
pole.

$natural\_frequency$ Pointer to double value denoting the natural frequency with respect to the dominant pole.

Description
In the design and analysis of control systems, oftentimes we need to utilize the root locus to calculate the dominant pole, extra gain, and natural frequency of a system at which point the system exhibits the desired percent overshoot or damping ratio. Based on the specified percent overshoot/damping ratio, the function **getDominantPole()** computes and returns the corresponding dominant pole/dominant pole pair of the system, extra gain that should be added to yield the dominant pole, natural frequency of the system, and damping ratio/percent overshoot. Note that for the pair of the first and second arguments, only one argument can point to a non-zero value. The remaining argument of this pair should point to NULL or a variable where no value is assigned. If any of the third to fifth arguments is not desired, it can be specified as NULL, so that the corresponding value will not be calculated and returned.

Example 1
In this example, the dominant pole, extra gain, natural frequency, and damping ratio of a system are calculated and returned when the percent overshoot of the system is specified.

```
#include <control.h>

int main() {
  // Define the plant
  double plant_k = 4;
  array double complex plant_p[] = {0, -2};

  // Specify the percent overshoot/damping ratio
  double percent_os = 30, damping_ratio;

  // Declare necessary variables
  array double complex dominant_p[1];
  double extra_gain, natural_freq;

  // Declare a CControl object for the plant
  CControl plant;

  // Calculate the dominant pole, extra gain,
  // natural frequency, and damping ratio
  plant.model("zpk", NULL, plant_p, plant_k);
  plant.getDominantPole(&percent_os, &damping_ratio, dominant_p, &extra_gain, &natural_freq);

  // Display the ouput data
  printf("\nDamping ratio: %f\n"
         "Dominant pole: %f"
         "Extra gain: %f\n"
         "Natural frequency: %f\n\n", damping_ratio, dominant_p, extra_gain, natural_freq);
  return 0;
}
```

The output of Example 1 is shown as follows.

```

Damping ratio: 0.357857
Dominant pole: complex(-1.000000,2.609318)
Extra gain: 1.952135
Natural frequency: 2.794376
```

Example 2
In this example, only the dominant pole and extra gain of a system are calculated and returned when the damping ratio of the system is specified. The arguments for those unwanted values are specified as NULL.

```
#include <control.h>

int main() {
  // Define the plant
  double plant_k = 4;
  array double complex plant_p[] = {0, -2};

  // Specify the percent overshoot/damping ratio
  double damping_ratio = 0.6;

  // Declare necessary variables
  array double complex dominant_p[1];
  double extra_gain;

  // Declare a CControl object for the plant
  CControl plant;

  // Calculate the dominant pole, extra gain,
  // natural frequency, and damping ratio
  plant.model("zpk", NULL, plant_p, plant_k);
  plant.getDominantPole(NULL, &damping_ratio, dominant_p, &extra_gain, NULL);

  // Display the ouput data
  printf("\nDominant pole: %f"
         "Extra gain: %f\n\n", dominant_p, extra_gain);
  return 0;
}
```

The output of Example 2 is shown as follows.

```
Dominant pole: complex(-1.000000,1.333335)
Extra gain: 0.694446
```

See Also
CControl::compensatorZeroPoleGain().

CControl ::getSystemType

Synopsis
int **getSystemType** ();

Purpose
Get system type.

Return Value
Upon successful completion, a positive integer which stands for one of the system types is returned. Otherwise, CHTKT_CONTROL_SYSTEMTYPE_INVALID is returned.

Parameters
None.

Description
The function **getSystemType** () returns a system type which is defined as a macro in file **control.h**. All possible system types are listed in Table A.2.

Table A.2: Macros of system types.

Macros	**System Type**
CHTKT_CONTROL_SYSTEMTYPE_INVALID	invalid system
CHTKT_CONTROL_SYSTEMTYPE_EMPTY	empty system
CHTKT_CONTROL_SYSTEMTYPE_TFO	transfer function model only
CHTKT_CONTROL_SYSTEMTYPE_ABCD	$\dot{X} = AX + BU; Y = CX + DU$
CHTKT_CONTROL_SYSTEMTYPE_D	$Y = DU$
CHTKT_CONTROL_SYSTEMTYPE_AC	$\dot{X} = AX; Y = CX$
CHTKT_CONTROL_SYSTEMTYPE_AB	$\dot{X} = AX + BU$
CHTKT_CONTROL_SYSTEMTYPE_A	$\dot{X} = AX$

Example

```
#include <control.h>

#define NUMX 2        // number of states
#define NUMU 1        // number of inputs
#define NUMY 1        // number of outputs
void sysprint(CControl *pSys);

int main() {
     array double A[NUMX][NUMX] = {-0.5572,   -0.7814,
                                    0.7814,    0};
     array double B[NUMX][NUMU] = {1, 0};
     array double C[NUMY][NUMX] = {1.9691,  6.4493};
     array double D[NUMY][NUMU] = {5};
     CControl sys1, sys2, sys3;

     sys1.model("ss", A, B, C, D);
     sysprint(&sys1);
     sys2.model("ss", A, B, NULL, NULL);
     sysprint(&sys2);
     sys3.model("ss", NULL, NULL, NULL, D);
     sysprint(&sys3);

     return 0;
}

void sysprint(CControl *pSys){
   switch(pSys->getSystemType()) {
      case CHTKT_CONTROL_SYSTEMTYPE_ABCD:
         printf("\nThe system type is : dot_x=Ax+Bu; y=Cx+Du\n");
         break;
      case CHTKT_CONTROL_SYSTEMTYPE_D:
         printf("\nThe system type is : y=Du\n");
         break;
      case CHTKT_CONTROL_SYSTEMTYPE_AC:
         printf("\nThe system type is : dot_x=Ax; y=Cx\n");
         break;
      case CHTKT_CONTROL_SYSTEMTYPE_AB:
         printf("\nThe system type is : dot_x=Ax+Bu\n");
         break;
      case CHTKT_CONTROL_SYSTEMTYPE_A:
         printf("\nThe system type is : dot_x=Ax\n");
         break;
      case CHTKT_CONTROL_SYSTEMTYPE_EMPTY:
         printf("\nThe system type is : empty system.\n");
         break;
      case CHTKT_CONTROL_SYSTEMTYPE_TFO:
         printf("\nThe system only has transfer function model.\n");
         break;
      case CHTKT_CONTROL_SYSTEMTYPE_INVALID:
         printf("\nThe system type is invalid.\n");
         break;
      default:
         printf("\nError: Incorrect mode specified.\n");
   }

   if(pSys->isDiscrete()) {
        printf("The system is discrete-time system.\n");
   }

   if(pSys->hasdelay()) {
        printf("The system has delays.\n");
   }
}
```

The output is shown as follows.

```
The system type is : dot_x=Ax+Bu; y=Cx+Du

The system type is : dot_x=Ax+Bu

The system type is : y=Du
```

See Also
CControl::hasdelay(), **CControl::isDiscrete**().

CControl ::gram

Synopsis
int **gram** (array double complex w[:][:], char $type$);

Purpose
Compute the controllability or observability grammians of a state-space model.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters

w Computational array containing the output grammian.

$type$ Character indicates which grammian is computed. Character ’c’ results in the controllability grammian, and ’o’ results in the observability grammian.

Description
The function **gram** () computes the controllability or observability grammians of the system (A, B, C, D). If the system is continuous, its controllability grammian W_c is defined by

$$W_c = \int_0^\infty e^{A\tau} B B^T e^{A^T\tau} \, \mathrm{d}\tau$$

and the observability grammian W_o by

$$W_o = \int_0^\infty e^{A\tau} C^T C e^{A^T\tau} \, \mathrm{d}\tau$$

If the system is discrete, the controllability grammian and observability grammian are

$$W_c = \sum_{k=0}^{\infty} A^k B B^T (A^T)^k$$

and

$$W_o = \sum_{k=0}^{\infty} (A^T)^k B^T C A^k$$

respectively. To obtain these grammians, some Lyapunov equations are solved internally by Ch Control System Toolkit.

Example

```
#include <control.h>

#define NUMX 2        // number of states
#define NUMU 2        // number of inputs
#define NUMY 1        // number of outputs

int main() {
     array double A[NUMX][NUMX] = {-0.5572,   -0.7814,
                                    0.7814,    0};
     array double B[NUMX][NUMU] = {1, 0,
                                   0, 2 };
     array double C[NUMY][NUMX] = {1.9691,  6.4493};
     array double complex w [NUMX][NUMX];
     CControl sys1, sys2;

     sys1.model("ss", A, B, C, NULL); // continuous time
     sys2.model("ss", A, B, C, NULL, -1); // discrete time

     sys1.gram(w, ’c’);
     printf("The controllability grammian of continuous-time system is\nwc = %f\n", w);
     sys1.gram(w, ’o’);
     printf("The observability grammian of continuous-time system is\nwo = %f\n", w);

     sys2.gram(w, ’c’);
     printf("The controllability grammian of discrete-time system is\nwc = %f\n", w);
     sys2.gram(w, ’o’);
     printf("The observability grammian of discrete-time system is\nwo = %f\n", w);

     return 0;
}
```

The output is shown as follows.

```
The controllability grammian of continuous-time system is
wc = complex(4.486719,0.000000) complex(-2.559509,0.000000)
complex(-2.559509,0.000000) complex(6.311851,0.000000)

The observability grammian grammian of continuous-time system is
wo = complex(40.802966,0.000000) complex(26.614711,0.000000)
complex(26.614711,0.000000) complex(43.529355,0.000000)

The controllability grammian of discrete-time system is
wc = complex(6.234801,0.000000) complex(-1.685479,0.000000)
complex(-1.685479,0.000000) complex(7.806882,0.000000)

The observability grammian grammian of discrete-time system is
wo = complex(40.584885,0.000000) complex(18.856377,0.000000)
complex(18.856377,0.000000) complex(66.374031,0.000000)
```

See Also
CControl::ctrb(), **CControl::obsv**(), **CControl::lyap**(), **CControl::dlyap**().

CControl ::grid

Synopsis
int **grid** (int $flag$);

Purpose
Generate grid lines for regular plots or Nyquist plots.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters

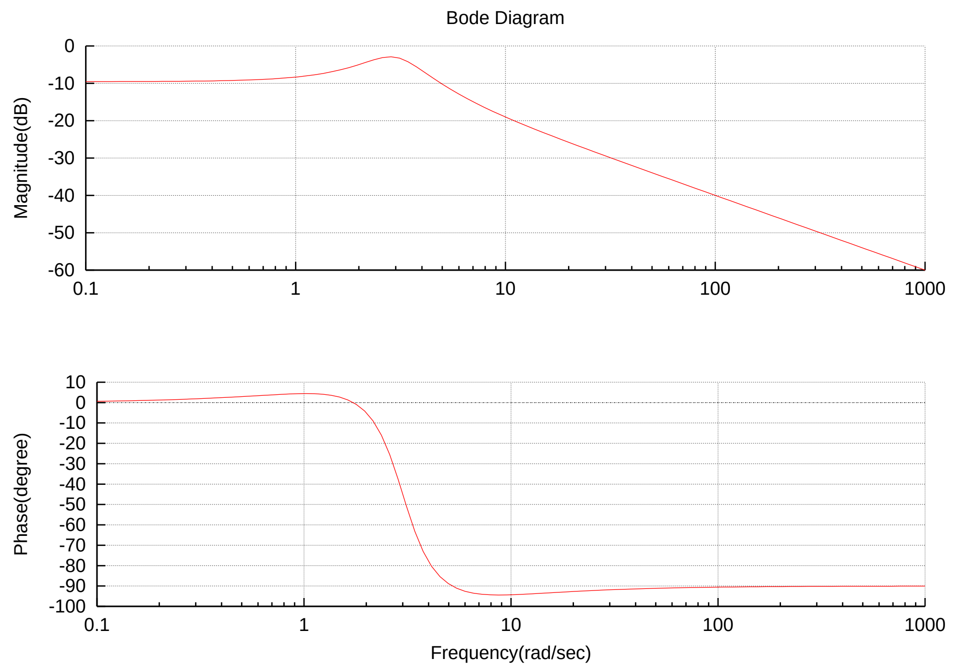
$flag$ Integer to turn on/off the grid. The value 1 turns on the grid, whereas 0 turns off the grid.

Description
The function **grid** () adds or removes grid lines for regular plots or Nyquist plots.

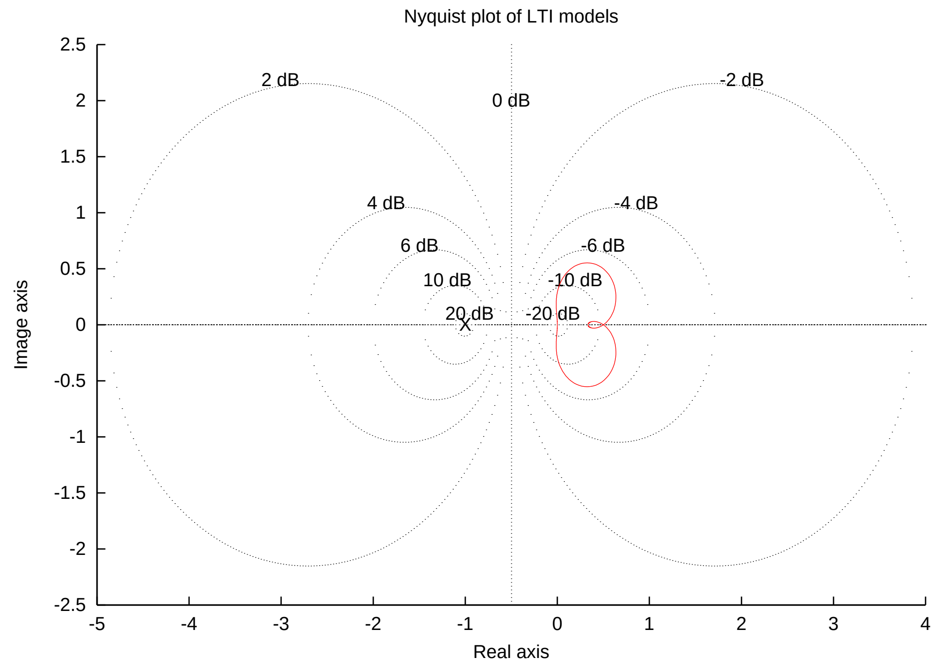
Example
The following example adds grid lines on both Bode plot and Nyquist plot.

```
#include <control.h>

#define NUMX 2    // number of states
#define NUMU 1    // number of inputs
#define NUMY 1    // number of outputs

int main() {
    array double A[NUMX][NUMX] = {{-2, -9},
                                  {1,  0}},
                 B[NUMX][NUMU] = {1, 0},
                 C[NUMY][NUMX] = {1, 3},
                 D[NUMY][NUMU] = {0};
    CPlot plot1, plot2;
    CControl sys;

    sys.model("ss", A, B, C, D);
    sys.grid(1);
    sys.bode(&plot1, NULL, NULL, NULL);
    sys.nyquist(&plot2, NULL, NULL, NULL);

    return 0;
}
```

The bode plot with grid is shown as follows,



The nyquist plot with grid is shown as follows,



See Also
CControl::sgrid(), **CControl::ngrid**(), **CControl::zgrid**().

CControl ::hasdelay

Synopsis
int **hasdelay** ();

Purpose

Determine if a system has delays.

Return Value
If system has delays, 1 is returned. Otherwise, 0 is returned.

Parameters
None.

Description
The function **hasdelay** () determines if a system mode has input delay, output delay, or input/output delay. Another functions **CControl** ::**setDelay** () can be used to set delays in a system.
Example

```
#include <control.h>

int main() {
  array double num[2] = {1, 3}, den[3] = {1, 2, 9};
  CControl sys;

  sys.model("tf", num, den);
  sys.setDelay(1, .1);
  sys.setDelay(2, .2);
  sys.setDelay(3, .3);
  if(sys.hasdelay()) {
    printf("The system has delays.\n");
  }
  return 0;
}
```

The output is shown as follows.

```
The system has delays.
```

See Also
CControl::setDelay().

CControl ::impulse

Synopsis
int **impulse** (class **CPlot** *$plot$, array double &$yout$, array double &$tout$, array double &$xout$, . . . /* double tf */);

Purpose
Calculate and plot impulse response of continuous time linear systems.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters
$plot$ Pointer to an existing object of class **CPlot** .

$yout$ Array of reference containing the output impulse response sequence.

$tout$ Array of reference containing time vector used for simulation.

$xout$ Array of reference containing state trajectories.

... The following argument is optional.

tf Double value specifying the final time of the simulation.

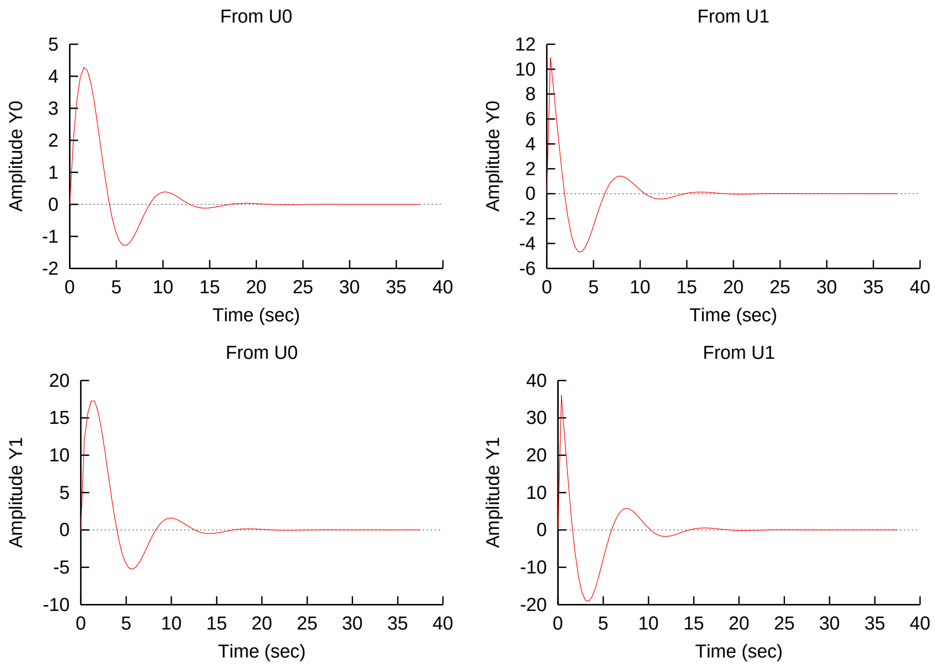
Description
The function **impulse** () computes and plots the impulse response of continuous LTI system. The impulse response is defined as the response of system to a Dirac input $\delta(t)$. The initial state is assumed to be zero. If system output array $yout$ is not specified, the default value 100 of macro CHTKT_CONTROL_SAMPLE_POINTS, which is defined in file control.h, is used as the number of time samples. When $yout$ is specified, the number of time samples is the same as the extent of its most right dimension. If the system is a multi-input system, the output is the collection of impulse responses for each input channel, and data of every input/output channel will be displayed in a seperate subplot. If the optional argument tf is absent, the function will determine the duration of simulation automatically on the transient behavior of the response.

Example
This example generates impulse response of a two-input-two-output system.

```
#include <control.h>

#define N    300      // number of time samples
#define TF   20       // final time
#define NUMX 2        // number of states
#define NUMU 2        // number of inputs
#define NUMY 2        // number of outputs

int main() {
     array double A[NUMX][NUMX] = { -0.5572,   -0.7814,
                                     0.7814,    0};
     array double B[NUMX][NUMU] = { 1, -1,    // 2i
                                    0,  2 };
     array double C[NUMY][NUMX] = { 1.9691,  6.4493,  // 2o
                                    12.0,  24.0 };
     CPlot plot;
     array double y[NUMU][NUMY][N];

     CControl sys;
     sys.model("ss", A, B, C, NULL);
     // To plot only
     sys.impulse(&plot, NULL, NULL, NULL);

     // To get the response in y with plotting
     // sys.impulse(&plot, y, NULL, NULL);

     // user-defined TF
     // sys.impulse(&plot, NULL, NULL, NULL, TF);

     // To get the response in y with plotting
     // sys.impulse(&plot, y, NULL, NULL, TF);

     // getting the response in y without plotting
     // sys.impulse(NULL, y, NULL, NULL, TF);
```

```
    /* print the response y */
    /*
       int i, j, k;
       for (i = 0; i < NUMU; i ++) { // for each input
           printf("\nResponse from U%d\n", i);
           for (j = 0; j < NUMY; j++) {  // for each output
               printf("to Y%d \n", j);
               for (k = 0; k < N; k += 30)  // for every 30 time samples
                   printf("y[%d][%d][%d] = %f\n", i, j, k, y[i][j][k]);
           }
       }
    */
    return 0;
}
```

The output of four seperate subplots is shown in the following figure.



See Also
CControl::step(), **CControl::initial**(), **CControl::lsim**().

CControl ::initial

Synopsis
int **initial** (class **CPlot** **plot*, array double &*yout*, array double &*tout*, array double &*xout*, array double &*x*0, . . . /* double *tf* */);

Purpose
Free response to the initial condition.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters

$plot$ Pointer to an existing object of class **CPlot** .

$yout$ Array of reference containing the output initial response sequence.

$tout$ Array of reference containing time vector used for simulation.

$xout$ Array of reference containing state trajectories.

$x0$ Array of reference containing the initial state.

. . . The following argument is optional.

tf Double value specifying the final time of the simulation.

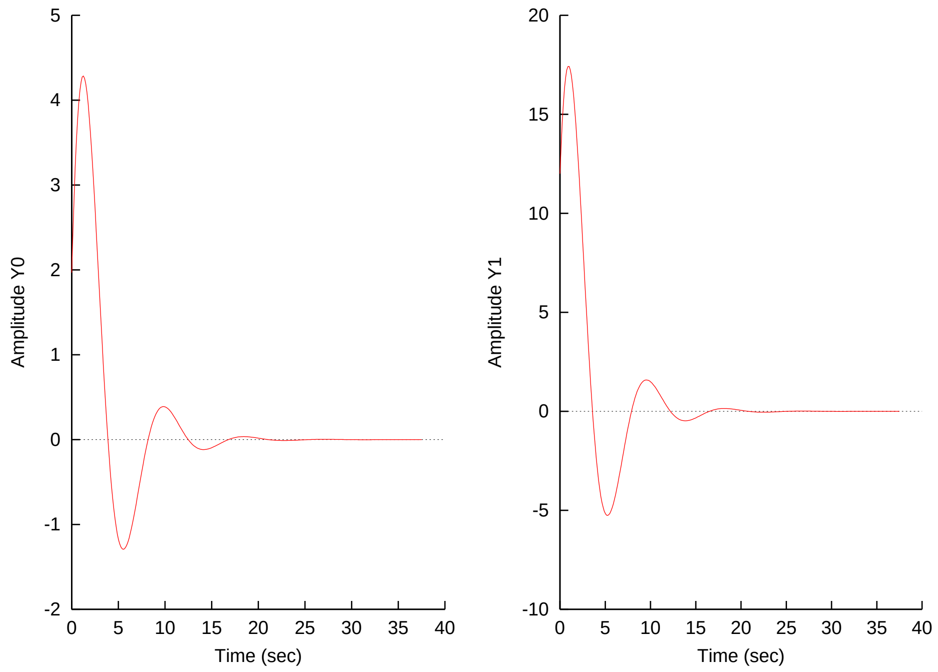
Description
The function **initial** () computes and plots the unforced response to the initial condition X_0 of continuous LTI system. The system input is assumed to be zero. If system output array $yout$ is not specified, the default value 100 of macro CHTKT_CONTROL_SAMPLE_POINTS, which is defined in file control.h, is used as the number of time samples. When $yout$ is specified, the number of time samples is the same as the extent of its most right dimension. If the system is a multi-input system, the output is the collection of initial responses for each output, and data from every output will be displayed in a seperate subplot. If the optional argument tf is absent, the function will determine the duration of simulation automatically on the transient behavior of the response.

Example
This example generates initial response of a two-input-two-output second order system.

```
#include <control.h>

#define N    300      // number of time samples
#define TF   50       // final time
#define NUMX 2        // number of states
#define NUMU 2        // number of inputs
#define NUMY 2        // number of outputs

int main() {
     array double A[NUMX][NUMX] = { -0.5572,   -0.7814,
                                     0.7814,    0};
     array double B[NUMX][NUMU] = { 1, -1,    // 2i
                                    0,  2 };
     array double C[NUMY][NUMX] = { 1.9691,  6.4493,  // 2o
                                    12.0,  24.0 };
     array double x0[NUMX] = {1.0, 0.0};
     CPlot plot;
     array double y[NUMY][N], tout[N], xout[NUMX][N];
     CControl sys;

     sys.model("ss", A, B, C, NULL);

     // To plot only
     //sys.initial(&plot, NULL, NULL, NULL, x0);

     // To obtain the response in y with plotting
     // sys.initial(&plot, y, NULL, NULL, x0);

     // user-defined TF
```

```
    // sys.initial(&plot, NULL, NULL, NULL, x0, TF);

    // To obtain the response in y with plotting
    // sys.initial(&plot, y, NULL, NULL, x0, TF);

    // To obtain the response in y without plotting
    // sys.initial(NULL, y, NULL, NULL, x0);

    // To obtain the response in y without plotting
    // sys.initial(NULL, y, NULL, NULL, x0, TF);

    // to obtain the response in y without plotting
    // sys.initial(&plot, y, tout, xout, x0, TF);

    sys.initial(&plot, y, tout, xout, x0); // obtaining the response in y with plotting
    /* print the response y and state trajectories */
       int j, k;
       for (j = 0; j < NUMY; j++) {  // for each output
           printf("Response of output No. %d \n", j);
           for (k = 0; k < N; k += 30)  // for every 30 time samples
               printf("%f: y[%d][%d] = %f\n", tout[k], j, k, y[j][k]);
       }
       printf("state trajectories\n");
       for (k = 0; k < N; k += 30)  // for every 30 time samples
         for (j = 0; j < NUMX; j++) {  // for each output
            printf("%f: x[%d][%d] = %f\n", tout[k], j, k, xout[j][k]);
       }

    return 0;
}
```

The plot consists of two subplots is shown in the following figure.



The output of response sequence and state trajectories at corresponding time is shown as follows.

```
Response of output No. 0
0.000000: y[0][0] = 1.969100
```

```
3.762542: y[0][30] = 0.192155
7.525084: y[0][60] = -0.366336
11.287625: y[0][90] = 0.213473
15.050167: y[0][120] = -0.093138
18.812709: y[0][150] = 0.034044
22.575251: y[0][180] = -0.010587
26.337793: y[0][210] = 0.002668
30.100334: y[0][240] = -0.000426
33.862876: y[0][270] = -0.000052
Response of output No. 1
0.000000: y[1][0] = 12.000000
3.762542: y[1][30] = -1.037237
7.525084: y[1][60] = -0.803365
11.287625: y[1][90] = 0.647387
15.050167: y[1][120] = -0.320254
18.812709: y[1][150] = 0.127707
22.575251: y[1][180] = -0.043295
26.337793: y[1][210] = 0.012326
30.100334: y[1][240] = -0.002657
33.862876: y[1][270] = 0.000205
state trajectories
0.000000: x[0][0] = 1.000000
0.000000: x[1][0] = 0.000000
3.762542: x[0][30] = -0.375040
3.762542: x[1][30] = 0.144302
7.525084: x[0][60] = 0.119832
7.525084: x[1][60] = -0.093390
11.287625: x[0][90] = -0.031466
11.287625: x[1][90] = 0.042707
15.050167: x[0][120] = 0.005638
15.050167: x[1][120] = -0.016163
18.812709: x[0][150] = 0.000218
18.812709: x[1][150] = 0.005212
22.575251: x[0][180] = -0.000834
22.575251: x[1][180] = -0.001387
26.337793: x[0][210] = 0.000513
26.337793: x[1][210] = 0.000257
30.100334: x[0][240] = -0.000229
30.100334: x[1][240] = 0.000004
33.862876: x[0][270] = 0.000085
33.862876: x[1][270] = -0.000034
```

See Also
CControl::step(), **CControl::impulse**(), **CControl::lsim**().

CControl ::**isDiscrete**

Synopsis
int **isDiscrete** (. . . /* int isdisc */);

Purpose
Determine if the system is discrete time.

Return Value
If the system is discrete-time system, value 1 is returned. Otherwise, value 0 is returned.

Parameters
. . . The following argument is optional

$isdisc$ An integer to set the system to continuous time by value 0 or discrete time by value 1.

Description
The function **isDiscrete** () determine if the system is discrete-time system. The user can change the system to continuous time by passing the argument $isdisc$ with value 0, or to discrete time with value 1.

Example

```
#include <control.h>
int main() {
   array double complex zero[2]={2, 3},
                        pole[2]={1, -1};
   double k = 3;

   CControl sys;
   sys.model("zpk", zero, pole, k, -1);

   if(sys.isDiscrete()) {
     printf("This is a discrete system.\n");
   }
   return 0;
}
```

The output is shown as follows.

```
This is a discrete system.
```

See Also
CControl::delay2z(), **CControl::c2d**(), **CControl::d2c**().

CControl ::lqe

Synopsis
int **lqe** (array double &l, array double &p, array double complex &e, array double &g, array double &q, array double &r, . . . /* array double &n */);

Purpose
Kalman estimator design for continuous-time systems.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters
l Array of reference containing the observer gain matrix L

p Array of reference containing the solution P of the following Riccati equation.

$$AP + PA^T - (PC^T + GN)R^{-1}(CP + N^TG^T) + GQG^T = 0$$

e Array of reference containing the closed-loop eigenvalues, that is eigenvalues of matrix ($A - LC$).

g Array of reference containing the system noise matrix G

q Array of reference containing parameter Q in the following cost function J

$$J(u) = \int_0^\infty X^TQX + U^TRU + 2X^TNU\,\mathrm{d}x$$

r Array of reference containing parameter R in the above cost function J.

. . . The following argument is optional.

n Array of reference containing parameter N in the above cost function J. It is the noise cross-correlation, and set to zero when omitted.

Description

The function **lqe** () calculates observer gain matrix of the Kalman filter for the continuous-time systems

$$\begin{aligned}\dot{X} &= AX + BU + GW \\ \dot{Y} &= CX + DU + V\end{aligned}$$

with process noise and measurement noise covariances shown as

$$E\{W\} = E\{V\} = 0, E\{WW^T\} = Q, E\{VV^T\} = R, E\{WV^T\} = N$$

The function returns the observer gain matrix L such that the stationary Kalman filter

$$\dot{\hat{X}} = A\hat{X} + BU + L(Y - C\hat{X} - DU)$$

produces an optimal state estimate $\hat{X}$ of X using the sensor measurements Y.

Example

```
#include <control.h>

#define NUMX 2    // number of states
#define NUMU 1    // number of the inputs
#define NUMY 1    // number of the outputs

int main() {
    array double A[NUMX][NUMX] = {{-1,     0},
                                  {0,     -2}},
                 C[NUMY][NUMX] = {1, 1};
    array double l[NUMX][NUMY], p[NUMX][NUMX];
    array double complex e[NUMX];
    array double G[NUMX][1] = {0, 1};
    array double Q[1][1] = {1};
    array double R[NUMY][NUMY] = {1};
    array double N[1][NUMY] = {10};
    CControl sys;

    sys.model("ss", A, NULL, C, NULL);
    sys.lqe(l, p, e, G, Q, R, N);
    printf("l = %f\np = %f\ne = %f\n", l, p, e);
    return 0;
}
```

The output is shown as follows.

```
l = 0.000000
4.708204

p = -0.000000 -0.000000
-0.000000 -5.291796

e = complex(-6.708204,0.000000) complex(-1.000000,0.000000)
```

See Also
CControl::care().

CControl ::lqr

Synopsis
int **lqr** (array double &k, array double &s, array double complex &e, array double &q, array double &r, .../* array double &n */);

Purpose
Build linear-quadratic (LQ) state-feedback regulator for continuous-time plant.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters
k Reference of computational array containing the output optimal gain matrix K for state-feedback law

$$U = -KX$$

s Reference of computational array containing the solution S of continuous-time Riccati equation.

$$SA + A^TS - (SB + N)R^{-1}(B^TS + N^T) + Q = 0$$

e Reference of computational array containing the closed-loop eigenvalues, that is eigenvalues of matrix($A - BK$).

q Reference of computational array containing the parameter Q in the following cost function J

$$J(U) = \int_0^\infty (X^TQX + U^TRU + 2X^TNU)\,\mathrm{d}x$$

r Reference of computational array containing the parameter R in the above cost function J.

... The following argument is optional.

n Reference of computational array containing the parameter N in the above cost function J. It is set to zero when omitted.

Description
The function **lqr** () can be used to design linear-quadratic (LQ) state-feedback regulator for continuous-time plants shown as follows,

$$\begin{aligned}\dot{X} &= AX + BU \\ Y &= CX + DU\end{aligned}$$

and a cost function

$$J(U) = \int_0^\infty (X^T QX + U^T RU + 2X^T NU)\,\mathrm{d}x$$

The purpose of Linear Quadratic Gaussian (LQG) control scheme is to find an optimal gain matrix K, so that the state-feedback law

$$U = -KX$$

minimizes the above cost function $J(U)$. The gain K is called the LQ optimal gain. To obtain the LQ optimal gain, the user needs to solve a continuous-time algebraic Riccati equation shown below

$$SA + A^T S - (SB + N)R^{-1}(B^T S + N^T) + Q = 0$$

Then, K can be derived from the solution S of above equation by

$$K = R^{-1}(B^T S + N^T)$$

Example

```
#include <control.h>

#define NUMX 2    // number of states
#define NUMU 1    // number of inputs

int main() {
    array double A[NUMX][NUMX] = {{-1,     0},
                                  {0,     -2}},
                 B[NUMX][NUMU] = {1, 1};
    array double k[NUMU][NUMX], s[NUMX][NUMX];
    array double complex e[NUMX];
    array double Q[NUMX][NUMX] = {1, 1, 1, 1};
    array double R[NUMU][NUMU] = {5};
    array double N[NUMX][NUMU] = {10, 20};
    CControl sys;

    sys.model("ss", A, B, NULL, NULL);
//    sys.lqr(k, s, e, Q, R);
    sys.lqr(k, s, e, Q, R, N);
    printf("k = %f\ns = %f\ne = %f\n", k, s, e);

    return 0;
}
```

The output is shown as follows.

```
k = 0.975947 2.196276

s = -1.881182 -3.239082
-3.239082 -5.779537

e = complex(-4.923478,0.000000) complex(-1.248745,0.000000)
```

See Also
CControl::dlqr().

CControl ::lsim

Synopsis
int **lsim** (class **CPlot** *$plot$, array double &$yout$, array double &$tout$, array double &$xout$, array double &u, $\ldots$/* double tf, array double $x0[\&]$ */);

Purpose
Simulation of response to an arbitrary input of the linear systems.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters
$plot$ Pointer to an existing object of class **CPlot** .

$yout$ Array of reference containing the output response sequence.

$tout$ Array of reference containing the time vector used for simulation.

$xout$ Array of reference containing the state trajectories.

u Array of reference containing input sequence.

... The following arguments are optional.

tf Double value specifying the final time of the simulation.

$x0$ Array of reference containing the initial state.

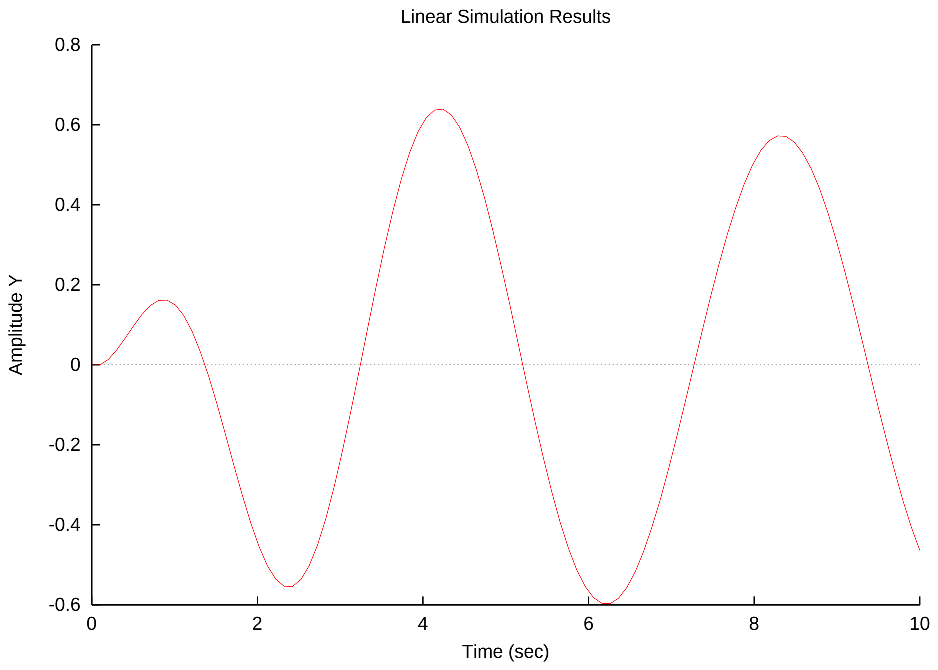
Description
The function **lsim** () computes and plots the response to an arbitrary input u of continuous-time linear systems with the initial condition $x0$. The initial condition is assumed to be zero by default. If the system is a multi-output system, response of each output channel will be displayed in a seperate subplot. If the optional argument tf is absent, the function will determine the duration of simulation automatically on the transient behavior of the response.

Example
This example simulates the response to a sinual wave input.

```
#include <control.h>

#define N 100
```

```
int main() {
   int i;
   array double num[2]={1, -1},
                den[3]={1, 1, 5};
   CPlot plot;
   double times[N],iu[N];
   CControl sys;

   sys.model("tf", num, den);
   linspace(times,0,12); // generate the sinual wave as input
   for (i=0; i<N; i++)
      iu[i] = sin(2*3.14/5*times[i]);
   sys.lsim(&plot, NULL, NULL, NULL, iu, 10);
   return 0;
}
```

The plot of system response to the sinual wave input is shown in the following figure.



See Also
CControl::step(), **CControl::impulse**(), **CControl::initial**().

CControl ::lyap

Synopsis
int **lyap** (array double complex x[:][:], array double complex a[:][:], .../* array double complex b[:][:], array double complex c[:][:] */);

Purpose
Solve the continuous Lyapunov equation.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters

x Array of reference containing the solution of the Lyapunov equation.

a Array of reference containing the parameter A of the Lyapunov matrix equation.

... One argument c or two arguments b and c can be chosen.

b Array of reference containing the parameter B of the Lyapunov matrix equation.

c Array of reference containing the parameter C of the Lyapunov matrix equation.

Description

The function **lyap** () solves the following special continuous-time Lyapunov equation

$$AX - XA^T = -C$$

where X is the solution, and parameters A and C are $n \times n$ matrices. If the argument b is given, the general form of the Lyapunov equation (also called Sylvester equation) is solved.

$$AX - XB = -C$$

Example

```
#include <control.h>

int main() {
    array double complex a[3][3] = {2, 3, 4,
                                    5, 6, 7,
                                    8, 9, 10};
    array double complex b[3][3] = {2, 4, 5,
                                    11, 13, 15,
                                    1, 2, 3};
    array double complex c[3][3] = {2.3000, 4.5000, 2,
                                    3.4000, 6.7000, 5.0000,
                                    2.8700, 9.3200, 1};
    array double complex x[3][3];
    CControl sys;

    sys.lyap(x, a, b, c);
    printf("x = \n%f\n", x);

    return 0;
}
```

The output is shown as follows.

```
x =
complex(-2.009236,0.000000) complex(-0.260711,0.000000) complex(4.408764,0.000000)

complex(3.094459,0.000000) complex(-0.368029,0.000000) complex(-5.158256,0.000000)

complex(-1.271845,0.000000) complex(-0.066459,0.000000) complex(1.346946,0.000000)
```

See Also
CControl::dlyap().

CControl ::margin

Synopsis
int **margin** (class **CPlot** *$plot$, double *gm, double *pm, double *wcg, double *wcp, $\ldots$/* array double mag[&], array double phase[&], array double w[&] */);

Purpose
Computes the gain and phase margins.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters
$plot$ Pointer to an existing object of class **CPlot** .

gm Pointer to double value containing the resulting gain margin.

pm Pointer to double value containing the resulting phase margin.

wcg Pointer to double value containing the associated crossover frequency of the gain margin.

wcp Pointer to double value containing the associated crossover frequency of the phase margin.

... The following arguments are optional.

mag Array of reference containing the magnitude of frequency response.

$phase$ Array of reference containing the phase of frequency response.

w Array of reference containing the frequency points corresponding to mag and $phase$.

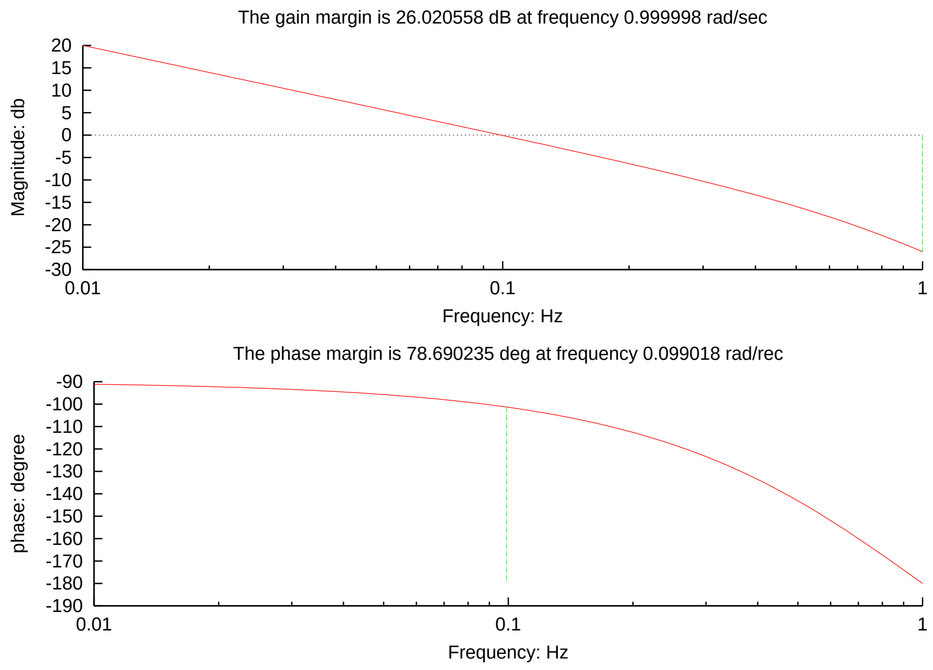
Description
The function **margin** () calculates and plots the minimum gain margin, phase margin, and corresponding crossover frequencies of SISO open-loop models. The gain margin is the factor by which the gain K can be raised before the system becomes unstable. It can be calculated from $\frac{1}{|G(j\omega)|}$, where $|G(j\omega)|$ is the magnitude at frequency ω which makes the phase $\angle G(j\omega)$ equal $-180°$. Similarly, the phase margin is another measure which is used to indicate the stability margin in a system. It can be calculated from $\angle G(j\omega) - 180°$ at frequency ω which makes the magnitude $|G(j\omega)|$ equal 1.0. The frequency ω is then called crossover frequency. It is generally found that gain margins of three or more combined with phase margins between 30 and 60 degrees result in reasonable trade-offs between bandwidth and stability.

Example

```
#include <control.h>

int main() {
   array double num[1]={.1},
                den[4]={1,2,1,0};
   double gm, pm, wcg, wcp;
```

```
    CPlot plot;
    CControl sys;

    sys.model("tf", num, den);
    sys.margin(&plot, &gm, &pm, &wcg, &wcp);
    printf("The gain margin is %f at frequency %f rad/sec\n", gm, wcg);
    printf("The phase margin is %f at frequency %f rad/sec\n", pm, wcp);
    return 0;
}
```

The Bode diagram displaying gain margin and phase margin is shown as follows



The output is shown as follows

```
The gain margin is 26.020558 at frequency 0.999998 rad/sec
The phase margin is 78.690235 at frequency 0.099018 rad/sec
```

See Also
CControl::bode(), **CControl::freqresp**().

CControl ::minreal

Synopsis
int ***minreal** (array double &u, .../* double tol */);

Purpose
Produce a minimal relization of an LTI model.

Return Value
Upon successful completion, a pointer to the corresponding minimal relization of a given LTI system is returned. Otherwise, a NULL pointer is returned.

Parameters
u An orthogonal matrix such that (u*A*u', u*B, C*u') is a Kalman decomposition of (A, B, C).

. . . The following argument is optional.

tol A double value used for the elimination of uncontrollable/unobservable state variables.

Description
The function **minreal** () eliminates uncontrollable and unobservable state variables from the state-space representation of an LTI system. The resulting state-space equation is controllable/observable and has the minimal state varibles. The minimal realization of the system has the same dynamic response characteristics as the original system. The output matrix u is an orthogonal matrix and (u*A*u', u*B, C*u') is a Kalman decomposion of (A, B, C). The input value of tol is used in the state elimination process.
Example

```
#include <control.h>

#define NUMX 3        // number of states
#define NUMU 1        // number of inputs
#define NUMY 1        // number of outputs

int main() {
     array double A[NUMX][NUMX] = { 0, 0, 1,
                                    1, 0, 0,
                                    0, 1, 0};
     array double B[NUMX][NUMU] = { 1, 0, 0};
     array double C[NUMY][NUMX] = { 1, -1, 0};
     CControl sys, *sys2;

     sys.model("ss", A, B, C, NULL);
     sys2 = sys.minreal(NULL);
     printf("The minimal relization of the system.\n");
     sys2->printss();
     return 0;
}
```

The output is shown as follows.

```
The minimal relization of the system.

State-space arguments:
A =
 -0.500000  -0.866025
  0.866025  -0.500000
B =
  0.408248
 -0.707107
C =
  0.000000  -1.414214
D =
  0.000000
```

See Also
CControl::pzcancel().

CControl ::model

Synopsis
int **model** (string_t *type*, . . . /* [array double *num*[&], array double *den*[&], [double *ts*]],

[array double &a, array double &b, array double &c, array double &d, [double ts]],
[array double complex z[&], array double complex p[&], double k, [double ts]],
[int nx, int ny, int nu] */);

Syntax
model(`"tf"`, *num*, *den*)
model(`"tf"`, *num*, *den*, *ts*)
model(`"ss"`, *a*, *b*, *c*, *d*)
model(`"ss"`, *a*, *b*, *c*, *d*, *ts*)
model(`"zpk"`, *z*, *p*, *k*)
model(`"zpk"`, *z*, *p*, *k*, *ts*)
model(`"empty"`)
model(`"rss"`)
model(`"rss"`, *nx*)
model(`"rss"`, *nx*, *ny*)
model(`"rss"`, *nx*, *ny*, *nu*)
model(`"drss"`)
model(`"drss"`, *nx*)
model(`"drss"`, *nx*, *ny*)
model(`"drss"`, *nx*, *ny*, *nu*)

Purpose
Create a continuous-time or discrete-time model for an LTI system.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters

$type$ String type value indicating the resulting model type.

... The following arguments are optional.

num Array of double containing the coefficients of the polynomial which represents the numerator of the transfer function.

den Array of double containing the coefficients of the polynomial which represents the denominator of the transfer function.

ts Double type value specifying the sample time of the discrete-time system.

a Array of reference containing the value of system matrix.

b Array of reference containing the value of input matrix.

c Array of reference containing the value of output matrix.

d Array of reference containing the value of transmission matrix.

z Array of double containing the zeros of the system.

p Array of double containing the poles of the system.

k Double type value containing the gain of the system.

nx Integer type value containing the number of states of the system.

ny Integer type value containing the number of outputs of the system.

nu Integer type value containing the number of inputs of the system.

Description
The function **model** () is used to create continuous-time or discrete-time models for an LTI system. The first argument of the function indicates the type of the model. To create a continuous-time transfer function model, the value of first argument is string `"tf"`. The other two arguments num and den are two one-dimensional arrays containing the coefficients of the polynomials which represent the numerator and denominator of the transfer function, respectively. To create a continuous-time state-space model, a string `"ss"` should be passed to the first argument of the function. The remaining four arguments a, b, c, and d are arrays of reference containing the value of system matrix, input matrix, output matrix, and transmission matrix, respectively. To create a continuous-time zero-pole-gain model, the value of first argument is string `"zpk"`. The arguments z and p are two one-dimensional arrays containing the zeros and poles of the system. The argument k represents the gain of the system. To create a discreate-time models of above three types, an extra argument ts which specifies the sample time of the discrete-time system should be passed to the function. If the value of ts equals to 0, the model will be treated as a continuous-time model. To create a random state-space model or a discrete-time random state-space model with default number of states, outputs, and inputs, only the string `"rss"` or `"drss"` is needed to be passed to the first argument of the function. To specify the number of states, outputs, and inputs, the values of nx, ny, and nu should be passed to the function.

Example

```
#include <control.h>

int main() {
  array double num[2] = {1, 3}, den[3] = {1, 2, 9};
  CControl sys;

  sys.model("tf", num, den);
  sys.printSystem();

  return 0;
}
```

The output is shown as follows

```
Continuous-time System
State-space arguments:
A =
 -2.000000  -9.000000
  1.000000   0.000000
B =
  1.000000
  0.000000
C =
  1.000000   3.000000
D =
  0.000000
```

```
Transfer function parameters:
Numerator:
  0.000000   1.000000   3.000000
Denominator:
  1.000000   2.000000   9.000000

Zero-Pole-Gain parameters:
Zero:
complex( -3.000000,  0.000000)
Pole:
complex( -1.000000,  2.828427) complex( -1.000000, -2.828427)
Gain:   1.000000

input delay is   0.000000
output delay is   0.000000
input/output delay is   0.000000
```

See Also
CControl::tfdata(), **CControl::ssdata**(), **CControl::zpkdata**().

CControl ::ngrid

Synopsis
int **ngrid** (int $flag$);

Purpose
Generate grid lines for Nichols plots.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters

$flag$ Integer to turn on/off the ngrid. The value 1 turns on the ngrid, whereas 0 turns off the ngrid.

Description
The function **ngrid** () adds or removes grid lines for Nichols charts produced by function **CControl** ::**nichols** (). For SISO systems, when $G(j\omega)$ is the complex transfer function, there is a corresponding unity-feedback closed-loop transfer function

$$T(j\omega) = \frac{\mathrm{G}(j\omega)}{1 + \mathrm{G}(j\omega)}$$

Assume that $M(\omega)$ and $a(\omega)$ are magnitude and phase of that closed-loop transfer function, we have equations as shown below,

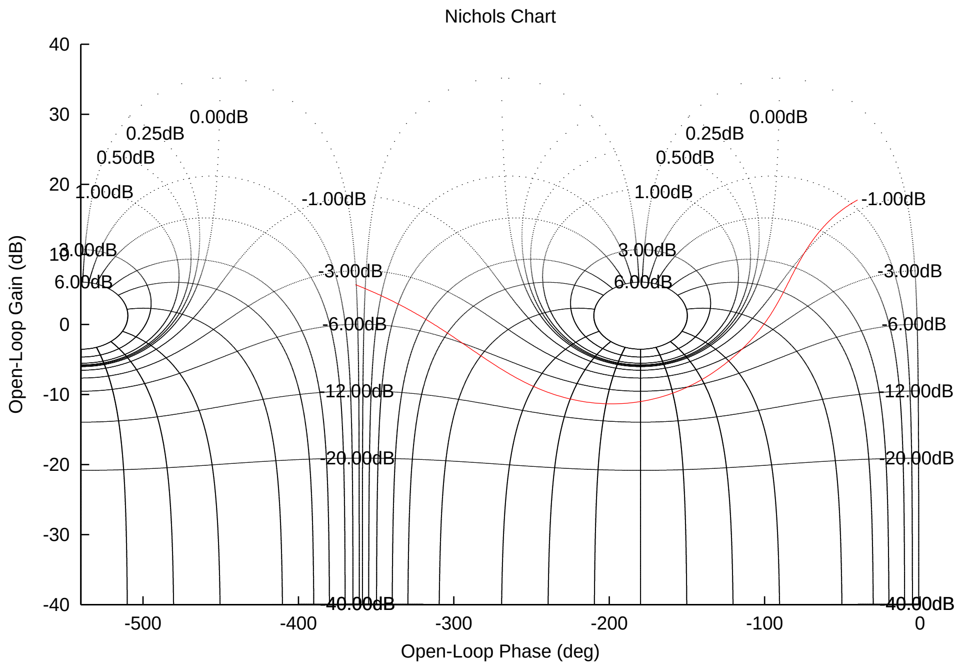
$$M(\omega) = |T(j\omega)|, a(\omega) = \angle T(j\omega)$$

In Nichols chart plotted by function **CControl** ::**nichols** (), the contours of constant value of $M(\omega)$ and $a(\omega)$ are added as grid lines.

Example
The following example adds grid lines on Nichols plot.

```
#include <control.h>

int main() {
   array double num[5] = {-4, 48, -18, 250, 600},
                den[5] = {1, 30, 282, 525, 60};
   CPlot plot;
   CControl sys;
   array double w[500];

   sys.model("tf", num, den);
   logspace(w, -1, 1);
   sys.ngrid(1);
   sys.nichols(&plot, NULL, NULL, NULL, w);
   return 0;
}
```

The Nichols plot with grid lines is shown as follow.



See Also
CControl::sgrid(), **CControl::grid**(), **CControl::zgrid**(), **CControl::nichols**().

CControl ::nichols

Synopsis
int* **nichols** (class **CPlot** *$plot$, array double mag[&], array double $phase$[&], array double $wout$[&], . . . /* array double w[&] or double $wmin$, double $wmax$ */);

Purpose
Compute the Nichols frequency response of the system.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters

$plot$ Pointer to an existing object of class **CPlot** .

mag Array of reference containing magnitudes of the frequency response at the frequencies in array $wout$. The magnitudes are expressed in decibels.

$phase$ Array of reference containing phases of the frequency response at the frequencies $wout$. The phases are expressed in degrees.

$wout$ Array of reference containing output frequencies which are expressed in rad/sec. If the user-defined frequencies are absent, the automatically generated frequencies will be contained.

... Variable length argument list expecting one argument, w, or two arguments, $wmin$ and $wmax$.

w Array of reference containing user-defined frequencies at which frequency response is calculated.

$wmin$ **and** $wmax$ Two double value specifying bounds of a particular frequency interval [wmin,wmax].

Description
The function **nichols** () computes the magnitude and phase of the frequency response of LTI models and plots them in Nichols coordinates. If the first argument is not a NULL pointer, a Nichols plot will be produced. If the user explicitly specifies the frequency range or frequency points, they will be used in calculations and plots. Otherwise, the default frequency vector w is generated by the following statement automatically.

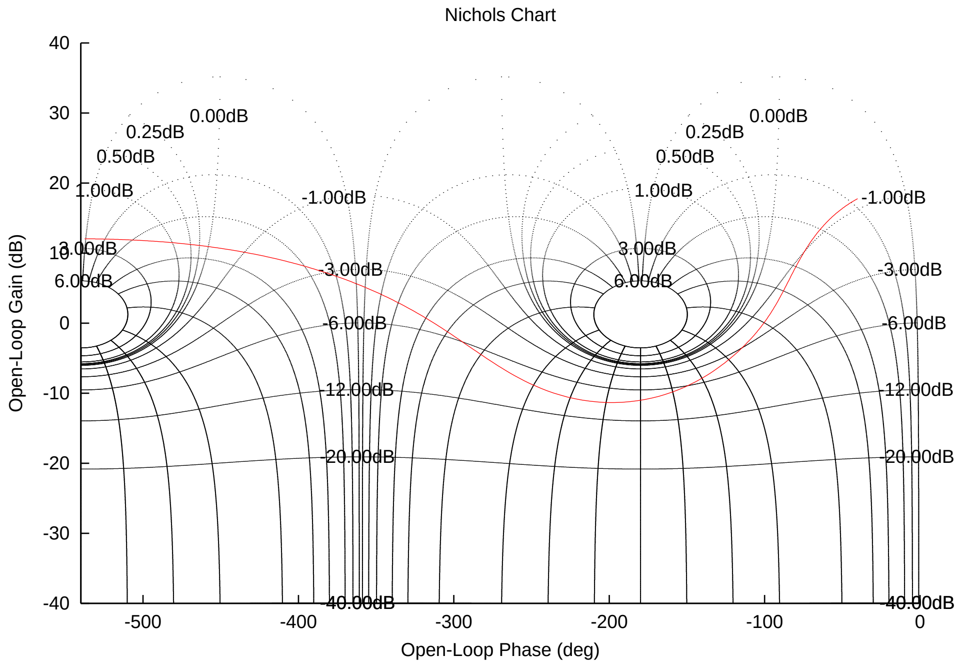
```
logspace(w, log10(0.1), log10(1000))
```

w is a logarithmically spaced frequency vector and expressed in $radians/sec$. This function only applys to SISO cases in its current implementation. It is recommended to call function **CControl** ::**ngrid** () to add the grid lines before this function is called.

Example

```
#include <control.h>

int main() {
   array double num[5] = {-4, 48, -18, 250, 600},
                den[5] = {1, 30, 282, 525, 60};
   CPlot plot;
   CControl sys;

   sys.model("tf", num, den);
   sys.ngrid(1);
   sys.nichols(&plot, NULL, NULL, NULL);

   return 0;
}
```

The Nichols chart with grid lines is shown as follows.



See Also
CControl::ngrid(), **CControl::bode**(), **CControl::nyquist**().

CControl ::nyquist

Synopsis
int **nyquist** (class **CPlot** *$plot$, array double re[&], array double im[&], array double $wout$[&], .../* array double w[&] or double $wmin$, double $wmax$ */);

Purpose
The function **nyquist** calculates the Nyquist frequency response of the system.

Return Value
Upon successful completion, zero is returned. Otherwise, minus one is returned.

Parameters
$plot$ Pointer to an existing object of class **CPlot** .

re Array of reference containing the real part of the frequency response at the frequencies corresponding to array $wout$.

im Array of reference containing the imaginary part of the frequency response at the frequencies corresponding to $wout$.

$wout$ Array of reference containing output frequencies which are expressed in rad/sec. If the user-defined frequencies are absent, the automatically generated frequencies will be contained.

... Variable length argument list expecting one argument, w, or two arguments, $wmin$ and $wmax$.

w Array of reference containing user-defined frequencies for frequency response calculation.

$wmin$ **and** $wmax$ Two double values specifying boundary of a particular frequency interval for frequency response calculation.

Description
The function **nyquist** () calculates the Nyquist frequency response of LTI models. If the first argument is not a NULL pointer, **nyquist** function produces a Nyquist plot on the terminal. The arguments w or $wmin$ and $wmax$ explicitly specify the desired frequency points or frequency range to be used for the plot. If the user does not specify the w, the default frequency vector w is generated by the following statement automatically.

```
logspace(w, log10(wmin), log10(wmax))
```

where wmin = 0.1, wmax = 1000. w is a logarithmically spaced frequency vector and expressed in $radians/sec$. This function only applys to SISO cases in the current implementation.

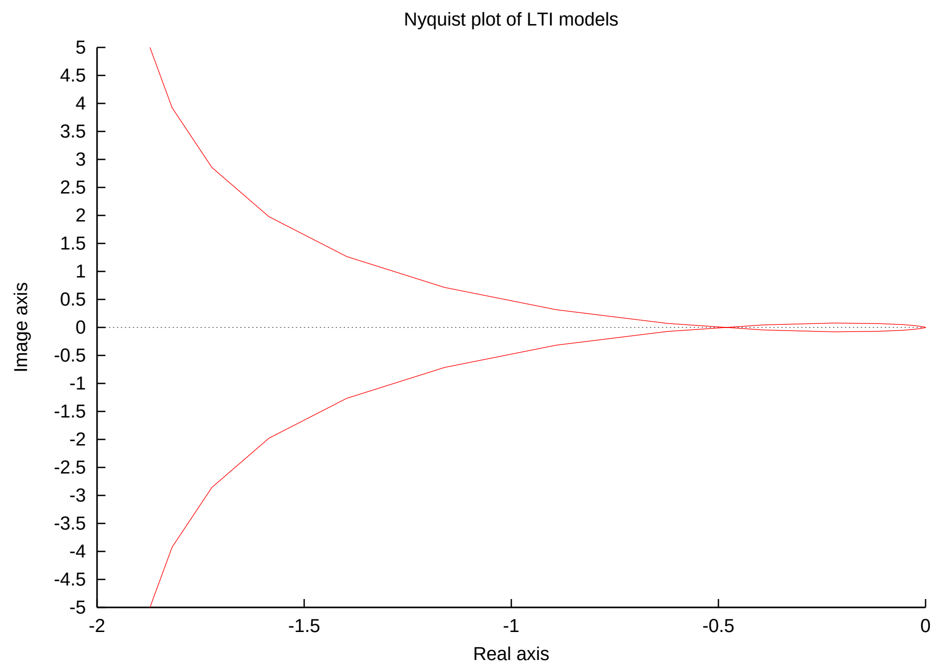
Example
This example plots the Nyquist plot of the system

$$\frac{1}{s(s^2+2s+1)}$$

on the frequency interval [-.00001 100000].

```
#include <control.h>

int main() {
   array double num[1] = {1}, den[4] = {1, 2, 1, 0};
   array double w[100];
   CPlot plot;
   CControl sys;

   sys.model("tf", num, den);
   plot.axisRange(PLOT_AXIS_Y, -5, 5, 0.5);
   logspace(w, -5, 5);
   sys.nyquist(&plot, NULL, NULL, NULL, w);
   return 0;
}
```

The Nyquist plot of this example is shown as follows.



See Also
CControl::freqresp(), **CControl::nichols**(), **CControl::bode**().

CControl ::obsv

Synopsis
int ***obsv** (array double &ob);

Purpose
Form the observability matrix.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters
ob Output observability matrix.

Description
The function **obsv** () computes the observability matrix from the system matrix and output matrix. Assume that the system has an $n \times n$ system matrix A and a $p \times n$ output matrix C, **obsv** (A, C) returns the $np \times n$ observability matrix O_b

$$\begin{bmatrix} C \\ CA \\ CA^2 \\ \dots \\ CA^{(n-1)} \end{bmatrix}$$

The system is observable if O_b has full rank n.

Example

```
#include <control.h>

#define NUMX 2        // number of states
#define NUMY 1        // number of outputs

int main() {
     array double A[NUMX][NUMX] = {-0.5572,   -0.7814,
                                    0.7814,    0};
     array double C[NUMY][NUMX] = {1.9691,  6.4493};
     array double ob[NUMX*NUMY][NUMX];
     CControl sys;
     int num;

     sys.model("ss", A, NULL, C, NULL);
     sys.obsv(ob);
     num = rank(ob);
     if(num == min(NUMX*NUMY, NUMX))
       printf("The system is observable.\n");
     else
       printf("The system is not observable.\n");
     printf("ob = %f\n", ob);
     return 0;
}
```

The output is shown as follows.

```
The system is observable.
ob = 1.969100 6.449300
3.942301 -1.538655
```

See Also
CControl::ctrb(), **CControl::ctrbf**(), **CControl::obsvf**().

CControl ::obsvf

Synopsis
int ***obsvf** (array double &$abar$, array double &$bbar$, array double &$cbar$, array double &t, array int &k, array double &a, array double &b, array double &c, . . . /* double tol */);

Purpose
Compute the observability staircase form.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters
$abar$ Computational array of double containing the system matrix in observability staircase form.

$bbar$ Computational array of double containing the input matrix in observability staircase form.

$cbar$ Computational array of double containing the output matrix in observability staircase form.

t Computational array of double containing the similarity transformation matrix from the original form to observability staircase form. The dimension of t is $n_x \times n_x$, where n_x is the number of state variables of original system.

k Computational array of int containing the number of observable states identified at each iteration of the algorithm. The array k has n_x elements. The number of observable states equals to the sum of k.

a Computational array of double containing the system matrix of original system.

b Computational array of double containing the input matrix of original system.

c Computational array of double containing the output matrix of original system.

... The following argument is optional.

tol A double value used to calculate the observable/unobservable subspaces.

Description
The function **obsvf** () decomposes the original unobservable state-space system represented by (A, B, C) into the observability staircase form ($\bar{A}$, $\bar{B}$, $\bar{C}$) which satisfies

$$\bar{A} = TAT^T, \bar{B} = TB, \bar{C} = CT^T$$

and

$$\bar{A} = \begin{bmatrix} A_{no} & A_{12} \\ 0 & A_o \end{bmatrix}, \bar{B} = \begin{bmatrix} B_{no} \\ B_o \end{bmatrix}, \bar{C} = \begin{bmatrix} 0 & C_o \end{bmatrix}$$

where T is a transformation matrix. The dimension of $\bar{A}$, $\bar{B}$, and $\bar{C}$ are the same as those of A, B, and C, respectively. If the rank of observability matrix of original state-space equation is equal to $n2$, the $n2$-dimensional subequations

$$\begin{aligned} \dot{\bar{\mathbf{x_o}}} &= \mathbf{A_o}\bar{\mathbf{x_o}} + \mathbf{B_o}\mathbf{u} \\ \mathbf{y} &= \mathbf{C_o}\bar{\mathbf{x_o}} \end{aligned}$$

is observable and has the same transfer function as the original system. For the user's convenience, an object of empty model can be created to call this member function.

Example

```
#include <control.h>

#define NUMX 3        // number of states
#define NUMU 1        // number of inputs
#define NUMY 1        // number of outputs

int main() {
     array double A[NUMX][NUMX] = {-2, 1, 2,
                                    1, 0, 0,
                                    0, 1, 0};
     array double B[NUMX][NUMU] = {1, 0, 0};
     array double C[NUMY][NUMX] = {0, 1, -1};
     array double Abar[NUMX][NUMX], Bbar[NUMX][NUMU], Cbar[NUMY][NUMX], t[NUMX][NUMX];
     array int k[NUMX];
     CControl sys;

     sys.model("ss", A, B, C, NULL);
     sys.obsvf(Abar, Bbar, Cbar, t, k, A, B, C);
     printf("Abar = \n%f\nBbar = \n%f\nCbar = \n%f\nt = \n%f\nk = \n%d\n", Abar, Bbar, Cbar, t, k);

     return 0;
}
```

The output is shown as follows.

```
Abar =
1.000000 1.414214 0.000000
0.000000 -2.500000 -0.866025
0.000000 0.866025 -0.500000

Bbar =
0.577350
-0.816497
0.000000

Cbar =
0.000000 0.000000 -1.414214

t =
0.577350 0.577350 0.577350
-0.816497 0.408248 0.408248
0.000000 -0.707107 0.707107

k =
1 1 0
```

See Also
CControl::obsv(), **CControl::ctrb**(), **CControl::ctrbf**().

CControl ::parallel

Synopsis
CControl* **parallel** (**CControl** *$sys2$, $\ldots$/* array int $input1$[:], array int $input2$[:], array int $output1$[:], array int $output2$[:] */);

Purpose
Parallel connection of two LTI models.

Return Value
Upon successful completion, a non-NULL pointer to class **CControl** representing the generated system is returned. Otherwise, a NULL pointer is returned.

Parameters
$sys2$ Pointer to the class **CControl** representing the system to be connected in the parallel form.

... The following arguments are optional.

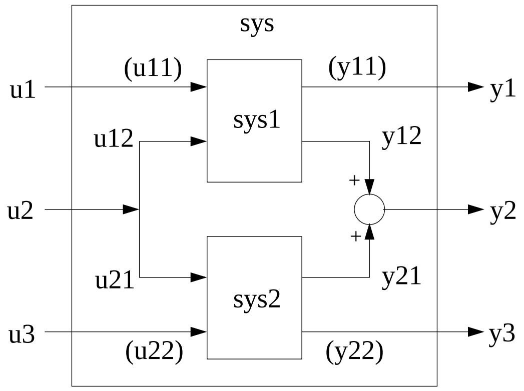
$input1$ One-dimensional computational array containing indices specifying which inputs of $sys1$ are connected to the inputs of $sys2$. If $sys1$ has m1 inputs, the default value of the array is {1, 2, ..., m1}.

$input2$ One-dimensional computational array containing indices specifying which inputs of $sys2$ are connected to the inputs of $sys1$. If $sys1$ has m1 inputs and $sys2$ has m2 inputs, the default value of the array is {m1+1, m1+2, ..., m1+m2}.

$output1$ One-dimensional computational array containing indices specifying which outputs of $sys1$ are summed with outputs of $sys2$. If $sys1$ has n1 outputs, the default value of the array is {1, 2, ..., n1}.

$output2$ One-dimensional computational array containing indices specifying which outputs of $sys2$ are summed with outputs of $sys1$. If $sys1$ has n1 outputs and $sys2$ has n2 outputs, the default value of the array is {n1+1, n1+2, ..., n1+n2}.

Description
The function **parallel** () builds a system with parallel interconnection shown in the figure.



The resulting system has the inputs of $[u_1, u_2, u_3]$ and outputs $[y_1, y_2, y_3]$. Note that the count of the inputs and outputs in Ch Control System Toolkit begins with 1.

Example
In this example, two identical three-input-two-output systems are connected in parallel form. The first and second inputs of $sys1$ are connect with the second and third inputs of $sys2$, respectively. On the other hand, the first and second outputs of $sys1$ are summed with the second and first inputs of $sys2$, respectively. Therefore, the resulting system $sys3$ has four inputs and two outputs.

```
#include <control.h>

#define NUMX 4    // number of states
#define NUMU 3    // number of inputs
#define NUMY 2    // number of outputs

int main() {
    array double A[NUMX][NUMX] = {{-4.12, -17.4, -30.8, -60},
                                  {1,     0,     0,     0},
                                  {0,     1,     0,     0},
                                  {0,     0,     1,     0}},
                 B[NUMX][NUMU] = {1, 0, 0,
                                  2, 0, 0,
                                  0, 0, 1,
                                  0, 1, 0},
                 C[NUMY][NUMX] = {4, 8.4, 30.78, 60,
                                  3, 7.4, 29.78, 50},
                 D[NUMY][NUMU] = {1, 2, 3,
                                  4, 5, 6};
    array int input1[2] = {1, 2}, input2[2] = {2, 3},
              output1[2] = {1, 2}, output2[2] = {2, 1};
    CControl sys1, sys2, *sys3;

    sys1.model("ss", A, B, C, D);
    sys2.model("ss", A, B, C, D);
    sys3 = sys1.parallel(&sys2, input1, input2, output1, output2);
    sys3->printSystem();

    return 0;
}
```

The output is shown as follows.

```
Continuous-time System
State-space arguments:
A =
 -4.120000 -17.400000 -30.800000 -60.000000   0.000000   0.000000   0.000000
  0.000000
  1.000000   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000
  0.000000
  0.000000   1.000000   0.000000   0.000000   0.000000   0.000000   0.000000
  0.000000
  0.000000   0.000000   1.000000   0.000000   0.000000   0.000000   0.000000
  0.000000
  0.000000   0.000000   0.000000   0.000000  -4.120000 -17.400000 -30.800000
-60.000000
  0.000000   0.000000   0.000000   0.000000   1.000000   0.000000   0.000000
  0.000000
  0.000000   0.000000   0.000000   0.000000   0.000000   1.000000   0.000000
  0.000000
  0.000000   0.000000   0.000000   0.000000   0.000000   0.000000   1.000000
  0.000000
B =
  0.000000   1.000000   0.000000   0.000000
  0.000000   2.000000   0.000000   0.000000
  1.000000   0.000000   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   2.000000
  0.000000   0.000000   1.000000   0.000000
  0.000000   1.000000   0.000000   0.000000
C =
  4.000000   8.400000  30.780000  60.000000   3.000000   7.400000  29.780000
 50.000000
  3.000000   7.400000  29.780000  50.000000   4.000000   8.400000  30.780000
 60.000000
D =
  3.000000   6.000000   8.000000   4.000000
  6.000000   6.000000   8.000000   1.000000
```

See Also
CControl::connect(), **CControl::feedback**(), **CControl::series**().

CControl ::place

Synopsis
int **place** (array double complex &k, array double complex p[:]);

Purpose
Pole placement design.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters
k Array of reference containing the output gain vector.

p Computational array containing the desired closed-loop pole locations.

Description
The function **place** () returns the gain vector k which makes closed loop poles match the user-specified vector p. In other words, for system (A, B, C, D), the eigenvalues of matrix($A - Bk$) will match p. This function is designed for both SISO and MIMO systems.

Example

In this example, the desired poles are -2, -1, -2, and -1.

```
#include <control.h>

#define NUMX 4    // number of states
#define NUMU 2    // number of inputs
#define NUMY 2    // number of outputs

int main() {

     array double A[NUMX][NUMX] = {{-2,  -9,  0,  0},
                                   {1,  0,  0,  0},
                                   {0, 0, -2, -9},
                                   {0, 0, 1, 0}},
                  B[NUMX][NUMU] = {1, 0,
                                   0, 0,
                                   0, 1,
                                   0, 0},
                  C[NUMY][NUMX] = {1, 3, 0, 0,
                                   0, 0, 1, 3},
                  D[NUMY][NUMU] = {0, 0, 0, 0};
     array double complex k[NUMU][NUMX];
     array double complex p[NUMX] = {-1, -2, -1, -2};
     CControl sys;

     sys.model("ss", A, B, C, D);
     sys.place(k, p);
     printf("k = %f\n", k);
     return 0;
}
```

The output is shown as follows.

```
k = complex(1.000000,0.000000) complex(-7.000000,0.000000) 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(-7.000000,0.000000)
```

See Also
CControl::acker(), **CControl::lqr**(), **CControl::rlocus**().

CControl ::pole

Synopsis
int **pole** (array double complex &p);

Purpose
Retrieve poles of LTI system.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters
p Computational array containing the system poles.

Description
The function **pole** () returns the poles p of the system. The poles are the eigenvalues of the system matrix A of the system.

Example

```
#include <control.h>

int main() {
   array double complex zero[3]={complex(-0.05, 2.738156),
                                 complex(-0.05, -2.738156),
                                 -2.0},
   pole[4]={
     complex(-1.797086, 2.213723), complex(-1.797086, -2.213723),
     complex(-0.262914, 2.703862), complex(-0.262914, -2.703862)};
   double k=4;
   int np;
   CControl sys;

   sys.model("zpk", zero, pole, k);
   np = sys.size('p');
   array double complex p[np];
   sys.pole(p);
   printf("The poles of the system are\n%f\n", p);
   return 0;
}
```

The output is shown as follows.

```
complex(-1.797086,2.213723) complex(-1.797086,-2.213723) complex(-0.262914,2.703862)
complex(-0.262914,-2.703862)
```

See Also
CControl::damp(), **CControl::getDominantPole**(), **CControl::pzmap**().

CControl ::printSystem

Synopsis
void **printSystem** ();

Purpose
Print information of the system.

Return Value
None.

Parameters None.

Description
The function **printSystem** () prints informaton of systems, including input, output, and direct transmission matrices for state space models, numerator, and denominator for transfer function model, zeros, poles and gain for zero-pole-gain models, as well as delays, sample times. Ch Control System Toolkit will automatically try to create all state-space, transfer function and zero-pole-gain models for a SISO system as it is created. Some systems could only have one or two valid models. Some matrices could be displayed as $NULL$ pointers for special types such as input matrix B for type **CHTKT_CONTROL_SYSTEMTYPE_AC**. For MIMO systems, this function only display the state space model in current implementation.

Example

```
#include <control.h>

#define NUMX 2    // number of states

int main() {
   array double complex zero[3]={complex(-0.05, 2.738156),
                                 complex(-0.05, -2.738156),
                                 -2.0},
   pole[4]={ complex(-1.797086, 2.213723),
             complex(-1.797086, -2.213723),
             complex(-0.262914, 2.703862),
             complex(-0.262914, -2.703862)};
   double k=4;
   array double A[NUMX][NUMX] = {1, 2, 3, 4};
   CControl sys1, sys2;

   sys1.model("zpk", zero, pole, k);
   printf("\nInformation about sys1\n");
   sys1.printSystem();

   sys2.model("ss", A, NULL, NULL, NULL);
   printf("\nInformation about sys2\n");
   sys2.printSystem();

   return 0;
}
```

The output is shown as follows.

```
Information about sys1
Continuous-time System
```

```
State-space arguments:
A =
 -4.120000 -17.399997 -30.799994 -59.999994
  1.000000   0.000000   0.000000   0.000000
  0.000000   1.000000   0.000000   0.000000
  0.000000   0.000000   1.000000   0.000000
B =
  1.000000
  0.000000
  0.000000
  0.000000
C =
  4.000000   8.400000  30.799993  59.999986
D =
  0.000000

Transfer function parameters:
Numerator:
  0.000000   4.000000   8.400000  30.799993  59.999986
Denominator:
  1.000000   4.120000  17.399997  30.799994  59.999994

Zero-Pole-Gain parameters:
Zero:
complex( -0.050000,  2.738156) complex( -0.050000, -2.738156) complex( -2.000000,  0.000000)

Pole:
complex( -1.797086,  2.213723) complex( -1.797086, -2.213723) complex( -0.262914,  2.703862)
complex( -0.262914, -2.703862)
Gain:   4.000000

input delay is   0.000000
output delay is   0.000000
input/output delay is   0.000000


Information about sys2
Continuous-time System
State-space arguments:
A =
  1.000000   2.000000
  3.000000   4.000000
B =
NULL
C =
NULL
D =
NULL
```

See Also
CControl::printss(), **CControl::printtf**(), **CControl::printzpk**().

CControl ::printss

Synopsis
void **printss** ();

Purpose
Print system, input, output, and direct transmission matrices for state space models.

Return Value
None.

Parameters None.

Description
The function **printss** () prints system, input, output, and direct transmission matrices for state space models. Ch Control Toolkit will automatically try to create state-space model for each SISO system as it is created. Some matrices will be displayed as $NULL$ pointers for special types such as **CHTKT_CONTROL_SYSTEMTYPE_AC**.

Example

```
#include <control.h>

#define NUMX 2    // number of states

int main() {
   array double complex zero[3]={complex(-0.05, 2.738156),
                                 complex(-0.05, -2.738156),
                                 -2.0},
   pole[4]={ complex(-1.797086, 2.213723),
             complex(-1.797086, -2.213723),
             complex(-0.262914, 2.703862),
             complex(-0.262914, -2.703862)};
   double k=4;
   array double A[NUMX][NUMX] = {1, 2, 3, 4};
   CControl sys1, sys2;

   sys1.model("zpk", zero, pole, k);
   printf("Matrices of sys1");
   sys1.printss();

   sys2.model("ss", A, NULL, NULL, NULL);
   printf("\nMatrices of sys2");
   sys2.printss();

   return 0;
}
```

The output is shown as follows.

```
Matrices of sys1
State-space arguments:
A =
 -4.120000 -17.399997 -30.799994 -59.999994
  1.000000   0.000000   0.000000   0.000000
  0.000000   1.000000   0.000000   0.000000
  0.000000   0.000000   1.000000   0.000000
B =
  1.000000
  0.000000
  0.000000
  0.000000
C =
  4.000000   8.400000  30.799993  59.999986
D =
  0.000000

Matrices of sys2
State-space arguments:
A =
  1.000000   2.000000
  3.000000   4.000000
B =
NULL
C =
NULL
D =
NULL
```

See Also
CControl::printSystem(), **CControl::printtf**(), **CControl::printzpk**().

CControl ::printtf

Synopsis
void **printtf** ();

Purpose
Print numerator and denominator of a transfer function model.

Return Value
None.

Parameters
None.

Description
The function **printtf** () print numerator and denominator of a transfer function model. Ch Control Toolkit will automatically try to create transfer function model for each SISO system as it is created. In the current implementation, this function is applicable to SISO systems only.

Example

In this example, the system $sys2$ cannot be converted to transfer function model from state-space model, since it is not a SISO system.

```
#include <control.h>

#define NUMX 2    // number of states

int main() {
   array double complex zero[3]={complex(-0.05, 2.738156),
                                 complex(-0.05, -2.738156),
                                 -2.0},
   pole[4]={ complex(-1.797086, 2.213723),
             complex(-1.797086, -2.213723),
             complex(-0.262914, 2.703862),
             complex(-0.262914, -2.703862)};
   double k=4;
   array double A[NUMX][NUMX] = {1, 2, 3, 4};
   CControl sys1, sys2;

   sys1.model("zpk", zero, pole, k);
   printf("Matrices of sys1\n");
   sys1.printtf();

   sys2.model("ss", A, NULL, NULL, NULL);
   printf("\nMatrices of sys2\n");
   sys2.printtf();

   return 0;
}
```

The output is shown as follows.

```
Error: CControl::printtf() system is not a SISO
Matrices of sys1
Transfer function parameters:
Numerator: 4.000000*s*s*s+8.400000*s*s+30.799993*s+59.999986
Denominator: 1.000000*s*s*s*s+4.120000*s*s*s+17.399997*s*s+30.799994*s+59.999994

Matrices of sys2
```

See Also
CControl::printss(), **CControl::printSystem**(), **CControl::printzpk**().

CControl ::printzpk

Synopsis
void **printzpk** ();

Purpose
Print zeros, poles and gain of a zero-pole-gain model.

Return Value
None.

Parameters
None.

Description
The function **printzpk** () print zeros, poles and gain of a zero-pole-gain model. Ch Control Toolkit will automatically try to create zero-pole-gain model for each SISO system as it is created. In the current implementation, this function is applicable to SISO systems only.

Example

In this example, the system $sys2$ cannot be converted to zero-pole-gain model from state-space model, since it is not a SISO system.

```
#include <control.h>

#define NUMX 2    // number of states

int main() {
   array double complex zero[3]={complex(-0.05, 2.738156),
                                 complex(-0.05, -2.738156),
                                 -2.0},
   pole[4]={ complex(-1.797086, 2.213723),
             complex(-1.797086, -2.213723),
             complex(-0.262914, 2.703862),
             complex(-0.262914, -2.703862)};
   double k=4;
   array double A[NUMX][NUMX] = {1, 2, 3, 4};
   CControl sys1, sys2;

   sys1.model("zpk", zero, pole, k);
   printf("Matrices of sys1\n");
   sys1.printzpk();

   sys2.model("ss", A, NULL, NULL, NULL);
   printf("\nMatrices of sys2\n");
   sys2.printzpk();

   return 0;
}
```

The output is shown as follows.

```
Error: CControl::printzpk() system is not a SISO
Matrices of sys1

Zero-Pole-Gain parameters:

Zero: complex(-0.050000,2.738156) complex(-0.050000,-2.738156) complex(-2.000000,0.000000)


Pole: complex(-1.797086,2.213723) complex(-1.797086,-2.213723) complex(-0.262914,2.703862)
complex(-0.262914,-2.703862)

Gain: 4.000000

Matrices of sys2
```

See Also
CControl::printss(), **CControl::printSystem**(), **CControl::printtf**().

CControl ::pzcancel

Synopsis
CControl* **pzcancel** (.../* double tol */);

Purpose
Cancel the pole-zero pairs with the same value of a system.

Return Value
Upon successful completion, a pointer to the new model after cancelling the pole-zero pairs of a given LTI system is returned. Otherwise, a NULL pointer is returned.

Parameters
... The following argument is optional.

tol A value of double type specifies the tolerance in pole-zero cancellation.

Description
The function **pzcancel** () cancels the pole-zero pairs with the same value in the transfer function or zero-pole-gain model of a system. The new model has the same dynamic response characteristics as the original system. The input value of tol specifies the tolerence in the pole-zero cancellation process.
Example

```
#include <control.h>

int main() {
     array double complex zero[2] = {5, -2},
                          pole[4] = {complex(-1, 1), complex(-1, -1), -1, 5};
     double k = 3;
     CControl sys, *sys2;

     sys.model("zpk", zero, pole, k);
     sys2 = sys.pzcancel();
     printf("The system after pole-zero cancellation.\n");
     sys2->printSystem();
     return 0;
}
```

The output is shown as follows.

```
The system after pole-zero cancellation.
Continuous-time System
State-space arguments:
A =
 -3.000000  -4.000000  -2.000000
  1.000000   0.000000   0.000000
  0.000000   1.000000   0.000000
B =
  1.000000
  0.000000
  0.000000
C =
  0.000000   3.000000   6.000000
D =
  0.000000

Transfer function parameters:
Numerator:
  0.000000   0.000000   3.000000   6.000000
Denominator:
  1.000000   3.000000   4.000000   2.000000

Zero-Pole-Gain parameters:
Zero:
```

```
complex( -2.000000,  0.000000)
Pole:
complex( -1.000000,  1.000000) complex( -1.000000, -1.000000) complex( -1.000000,  0.000000)

Gain:   3.000000

input delay is   0.000000
output delay is   0.000000
input/output delay is   0.000000
```

See Also
CControl::minreal().

CControl ::pzmap

Synopsis
int **pzmap** (class **CPlot** *$plot$, array double complex &p, array double complex &z);

Purpose
Produce the pole-zero map of an LTI model.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters
$plot$ Pointer to an existing object of class **CPlot** .

p Array of reference containing output system poles.

z Array of reference containing output transmission zeros.

Description
The function **pzmap** () computes and plots the pole-zero map. The poles are marked as x's and the zeros o's. It is recommended to use function **CControl** ::**sgrid** () to plot lines of constant damping ratio and natural frequency in the s-plane. This function supports continuous-time SISO systems only in current implementation.

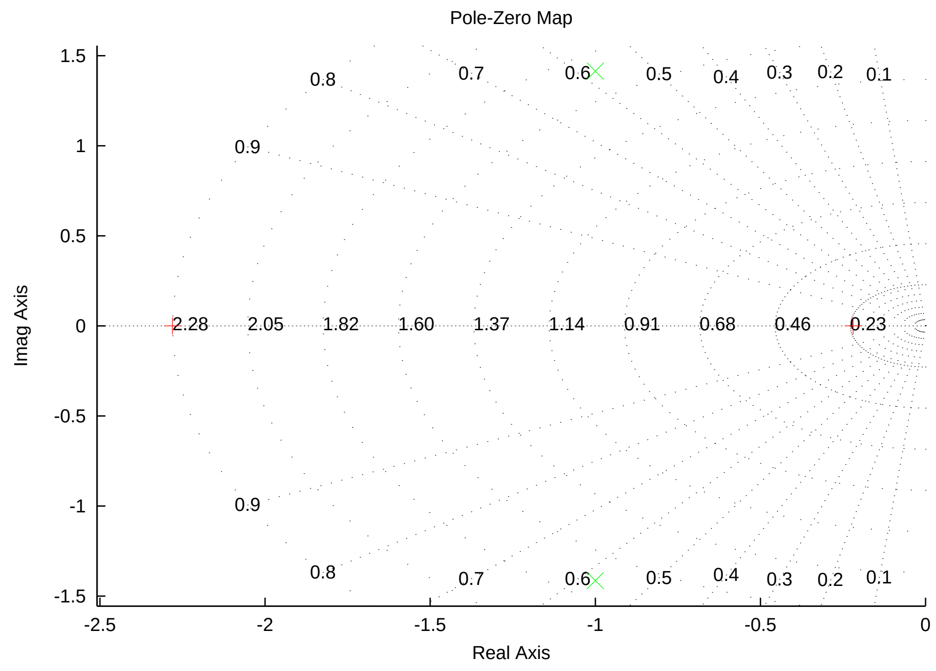
Example

```
#include <control.h>

int main() {
   array double num[3] = {2, 5, 1}, den[3] = {1, 2, 3};
   CPlot plot;
   CControl sys;
   int np, nz;

   sys.model("tf", num, den);
   np = sys.size('p');
   nz = sys.size('z');
   array double complex p[np], z[nz];

   sys.sgrid(1);
   sys.pzmap(&plot, p, z);
   printf("p = %fz = %f", p, z);

   return 0;
}
```

The poles and zeros are shown as follows.

```
p = complex(-1.000000,1.414214) complex(-1.000000,-1.414214)
z = complex(-0.219224,0.000000) complex(-2.280776,0.000000)
```

The pole-zero map plot with grid lines is shown in the following figure.



See Also
CControl::sgrid(), **CControl::pole**(), **CControl::damp**().

CControl ::rlocfind

Synopsis
int **rlocfind** (array double &k, array double &$poles$, . . . /* array double complex p[:] */);

Purpose
Find feedback gains for a given set of roots.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters
k Array of reference containing the output feedback gain.

$poles$ Array of reference containing system poles corresponding to the feedback gain k.

. . . The following argument is optional.

p Array of reference containing user-specified poles. If this argument is absent, interactive gain computation will perform.

Description
The function **rlocfind** () computes root locus gain k to make the system poles move to the user-desired poles p as close as possible.

Example

In this example, the user-desired poles are $-1.0 + 1.4142i; -1 + i$.

```
#include <control.h>

int main() {
   array double num[3] = {2, 5, 1}, den[3] = {1, 2, 3};
   array double complex p[2] = {complex(-1.0000, 1.4142), complex(-1, 1)};
   CPlot plot;
   int np;
   CControl sys;

   sys.model("tf", num, den);
   np = sys.size('p');
   array double k[2];
   array double complex poles[2][np];

   sys.rlocfind(k, poles, p);
   printf("user specified p = %f\n", p);
   printf("k = %f\npoles = %f\n", k, poles);

   return 0;
}
```

The poles and zeros are shown as follows.

```
user specified p = complex(-1.000000,1.414200) complex(-1.000000,1.000000)

k = 0.000006 0.242536

poles = complex(-1.000003,1.414200) complex(-1.000003,-1.414200)
complex(-1.081658,1.006696) complex(-1.081658,-1.006696)
```

See Also
CControl::pole(), **CControl::pzmap**().

CControl ::rlocus

Synopsis
int **rlocus** (class CPlot *$plot$, array double complex &r, array double &$kout$, . . . /* array double &k */);

Purpose
Compute and plot root locus of SISO.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters
$plot$ Pointer to an existing object of class **CPlot** .

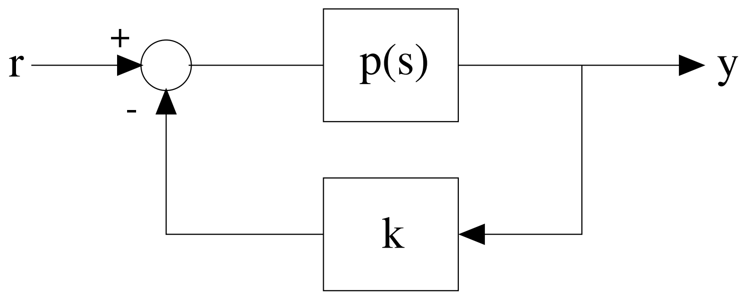
r Array of reference containing the complex root locations corresponding to gains in argument $kout$.

$kout$ Array of reference containing selected gains by users or by default.

... The following argument is optional.

k Array of reference containing user-specified gains.

Description
The function **rlocus** () computes and plots rlocus of SISO. Normally, roots of the characteristic equation will change as one of the system's parameters varies over a continuous range of value. Assume that the control system shown in the figure,



consists of the plant $p(s)$ and the scalar gain k. The root locus is the trajectories of the roots of following equation

$$q(s) = 1 + kp(s)$$

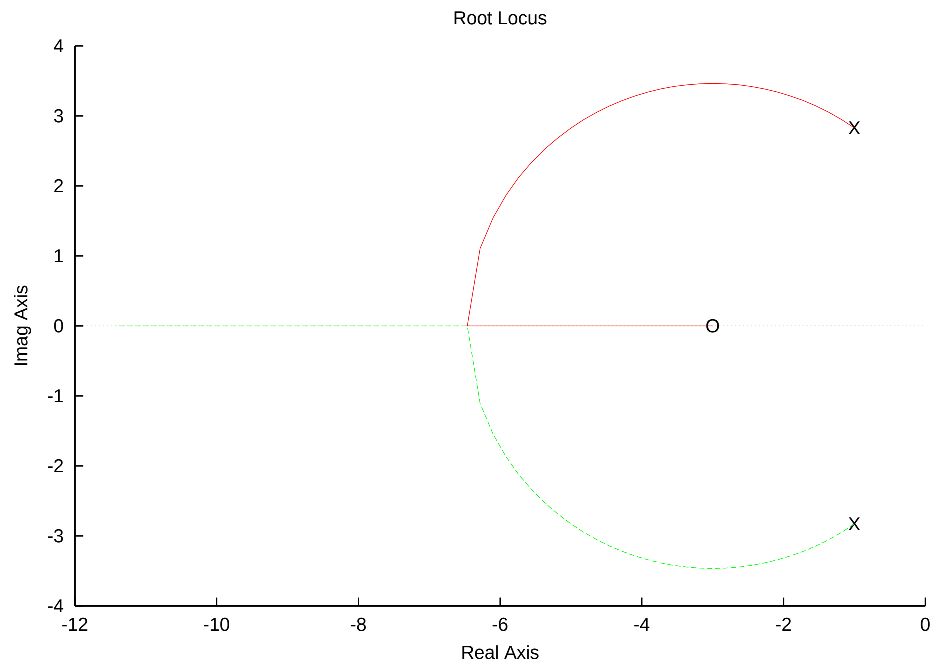
when the feedback gain k varies from 0 to positive infinity. If the user does not specify gains for which the roots are calculated, the function will select gains adaptively, and return these gains in $kout$. The number of branches of the root locus equal the number of the poles. Each branche begins at one of the roots (poles of the system) and moves off to infinity or zeros of the system. The poles are marked as x's and the zeros o's.

Example

In this example, the system sys has two poles and one zero.

```
#include <control.h>

int main() {
  array double num[2] = {1, 3}, den[3] = {1, 2, 9};
  CPlot plot;
  CControl sys;
  int nx;

  sys.model("tf", num, den);
  nx = sys.size('x');
  array double complex r[nx][100];
  array double kout[100];
  sys.rlocus(&plot, r, kout);

  return 0;
}
```

The plot of rlocus of system sys is shown in the following figure.



See Also
CControl::pole(), **CControl::pzmap**().

CControl ::series

Synopsis
CControl* **series** (**CControl** *$sys2$, . . . /* array int $output1$[:], array int $input2$[:] */);

Purpose
Series connection of two models.

Return Value
Upon successful completion, a non-NULL pointer to class **CControl** representing the generated system is returned. Otherwise, a NULL pointer is returned.

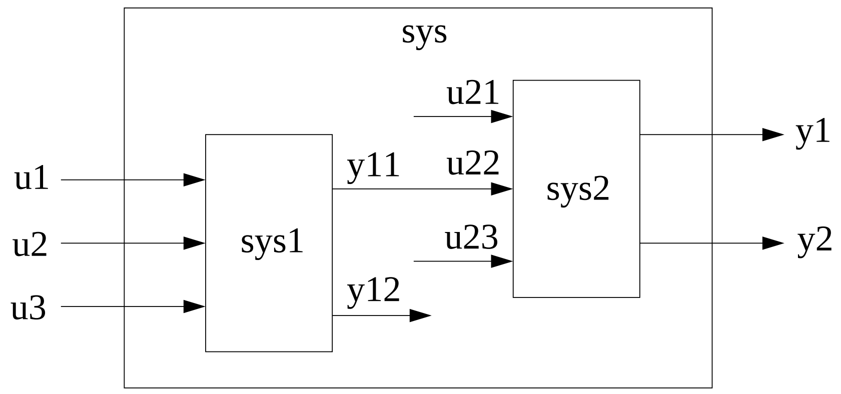
Parameters
$sys2$ Pointer to the class **CControl** representing the system to be connected in the series form.

. . . The following arguments are optional.

$output1$ One-dimensional computational array containing indices specifying which outputs of $sys1$ are connected to the inputs of $sys2$. If $sys1$ has n outputs, the default value of the array is {1, 2, ..., n}.

$input2$ One-dimensional computational array containing indices specifying which inputs of $sys2$ are connected to the outputs of $sys1$. If $sys2$ has m inputs, the default value of the array is {1, 2, ..., m}.

Description
The function **series** () builds a system with series interconnection shown in the figure,



Note that the count of the inputs and outputs in Ch Control System Toolkit begins with 1. The resulting system has the same inputs as $sys1$ and outputs as $sys2$.

Example
In this example, two identical three-input-two-output systems, $sys1$ and $sys2$, are connected in series form. The first output of system $sys1$ are connect to the second inputs of $sys2$. Therefore, the resulting system sys has three inputs and two outputs.

```
#include <control.h>

#define NUMX 4    // number of states
#define NUMU 3    // number of inputs
#define NUMY 2    // number of outputs

int main() {
    array double A[NUMX][NUMX] = {{-4.12, -17.4, -30.8, -60},
                                  {1,     0,     0,     0},
                                  {0,     1,     0,     0},
                                  {0,     0,     1,     0}},
                 B[NUMX][NUMU] = {1, 0, 0,
                                  2, 0, 0,
                                  0, 0, 1,
                                  0, 1, 0},
                 C[NUMY][NUMX] = {4, 8.4, 30.78, 60,
                                  3, 7.4, 29.78, 50},
                 D[NUMY][NUMU] = {1, 2, 3,
                                  4, 5, 6};
    array int output1[1] = {1}, input2[1] = {2};
    CControl sys, sys2, *sys3;

    sys.model("ss", A, B, C, D);
    sys2.model("ss", A, B, C, D);
    sys3 = sys.series(&sys2, output1, input2);
    sys3->printSystem();
    return 0;
}
```

The output is shown as follows.

```
Continuous-time System
State-space arguments:
A =
 -4.120000 -17.400000 -30.800000 -60.000000   0.000000   0.000000   0.000000
  0.000000
  1.000000   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000
  0.000000
  0.000000   1.000000   0.000000   0.000000   0.000000   0.000000   0.000000
  0.000000
  0.000000   0.000000   1.000000   0.000000   4.000000   8.400000  30.780000
 60.000000
  0.000000   0.000000   0.000000   0.000000  -4.120000 -17.400000 -30.800000
-60.000000
  0.000000   0.000000   0.000000   0.000000   1.000000   0.000000   0.000000
  0.000000
  0.000000   0.000000   0.000000   0.000000   0.000000   1.000000   0.000000
  0.000000
  0.000000   0.000000   0.000000   0.000000   0.000000   0.000000   1.000000
  0.000000
B =
  0.000000   0.000000   0.000000
  0.000000   0.000000   0.000000
  0.000000   0.000000   0.000000
  1.000000   2.000000   3.000000
  1.000000   0.000000   0.000000
  2.000000   0.000000   0.000000
  0.000000   0.000000   1.000000
  0.000000   1.000000   0.000000
C =
  4.000000   8.400000  30.780000  60.000000   8.000000  16.800000  61.560000
120.000000
  3.000000   7.400000  29.780000  50.000000  20.000000  42.000000 153.900000
300.000000
D =
  2.000000   4.000000   6.000000
  5.000000  10.000000  15.000000
```

See Also
CControl::connect(), **CControl::feedback**(), **CControl::parallel**().

CControl ::setDelay

Synopsis
int **setDelay** (int type, double delay);

Purpose
Set system delays.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters
type Integer indicating which delay to be set. Value 1 indicates input delay, 2 indicates output delay, and 3 indicates intput/output delay.

delay Double value specifying the delay value.

Description
The function **setDelay** () sets system delays, including input delay, output delay and input/output delay. If any delay has been set, the return value of function **CControl** ::**hasdelay** is 1, otherwise 0.

Example
Refer to function **CControl** ::**hasdelay** ().

See Also
CControl::hasdelay(), **CControl::delay2z**().

CControl ::setTs

Synopsis
int **setTs** (double ts);

Purpose
Set the sample time for the discrete-time system.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters
ts Double value specifying sample time.

Description
The function **setTs** () sets or changes sample times for discrete-time systems. If the system is a continuous-time system, it will be changed to discrete-time model after setting a nonzero ts. On the other hand, the system will be treated as a continuous-time system if the value of ts equals to 0. This function does not make any model conversion, except for the sample time, all other system parameters are unchanged. Function **c2d** () can be used to convert models between continuous-time systems and discrete-time systems, whereas **d2d** () convert models between discrete-time systems with different sample times.

Example

```
#include <control.h>

int main() {
  array double num[2] = {1, 3}, den[3] = {1, 2, 9};
  CControl sys;

  sys.model("tf", num, den, -1); // no sample time specified
  sys.printSystem();

  printf("Set a new sample time.\n");
  sys.setTs(.5);
  sys.printSystem();

  return 0;
}
```

The output is shown as follows.

```
Discrete-time System with unspecified sample time
State-space arguments:
A =
 -2.000000  -9.000000
  1.000000   0.000000
B =
  1.000000
  0.000000
C =
  1.000000   3.000000
D =
  0.000000

Transfer function parameters:
Numerator:
  0.000000   1.000000   3.000000
Denominator:
  1.000000   2.000000   9.000000

Zero-Pole-Gain parameters:
Zero:
complex( -3.000000,  0.000000)
Pole:
complex( -1.000000,  2.828427) complex( -1.000000, -2.828427)
Gain:   1.000000

input delay is   0.000000
output delay is   0.000000
input/output delay is   0.000000

Set a new sample time.
Discrete-time System with sample time of 0.500000s.
State-space arguments:
A =
 -2.000000  -9.000000
  1.000000   0.000000
B =
  1.000000
  0.000000
C =
  1.000000   3.000000
D =
  0.000000

Transfer function parameters:
Numerator:
  0.000000   1.000000   3.000000
Denominator:
  1.000000   2.000000   9.000000

Zero-Pole-Gain parameters:
Zero:
complex( -3.000000,  0.000000)
Pole:
complex( -1.000000,  2.828427) complex( -1.000000, -2.828427)
Gain:   1.000000

input delay is   0.000000
output delay is   0.000000
```

```
input/output delay is   0.000000
```

See Also
CControl::c2d(), **CControl::d2d**().

CControl ::sgrid

Synopsis
int **sgrid** (int $flag$, $\ldots$/* array double z[:], array double w[:] */);

Purpose
Generate an s-plane grid of constant damping factors and natural frequencies.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters

$flag$ Integer to turn on/off the sgrid. The value 1 turns on the sgrid, whereas 1 turns off the sgrid.

... The following two argument are optional.

z Computational array containing damping factors at which the grid lines are plotted.

w Computational array containing natural frequencies at which the grid lines are plotted.

Description
The function **sgrid** () generates an s-plane grid of constant damping factors and natural frequencies over the plots produced by function **CControl** ::**rlocus** () and **CControl** ::**pzmap** (). If the optional arguments z and w are absent, a grid of constant damping factors from 0 to 1 in steps of 0.1 and natural frequencies from 0 to 10 rad/sec in steps of 1 rad/sec will be plotted.

Example
Refer to function **CControl** ::**pzmap** ().

See Also
CControl::pzmap(), **CControl::rlocus**(), **CControl::zgrid**(), **CControl::ngrid**().

CControl ::size

Synopsis
int **size** (char property);

Purpose
Get the size of a specified property of a system.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters

property Character indicating which property whose size will be returned. Table A.3 lists all possible values of the argument.

Table A.3: Meanings of different values of the argument of member function **CControl ::size** ().

Value	**Description**
x	get the order of the system
y	get the number of the output of the system
u	get the number of the input of the system
n	get the order of the numerator of the transfer function
d	get the order of the denominator of the transfer function
z	get the number of the zeros of the system
p	get the number of the poles of the system

Description
The function **size** () returns the size of a system property specified in the argument *property*. This function is useful to determine how large a computational array, which is used to contain an output system property, should be declared at runtime.

Example

In this program, the length of array p is determined by the return value of function **size** ().

```
#include <control.h>

int main() {
   array double complex zero[3]={complex(-0.05, 2.738156),
                                 complex(-0.05, -2.738156),
                                 -2.0},
   pole[4]={
     complex(-1.797086, 2.213723), complex(-1.797086, -2.213723),
     complex(-0.262914, 2.703862), complex(-0.262914, -2.703862)};
   double k=4;
   int np;
   CControl sys;

   sys.model("zpk", zero, pole, k);
   np = sys.size('p'); // obtain the number of the poles
   array double complex p[np]; // declare an array with proper length

   sys.pole(p);
   printf("%f\n", p);
   return 0;
}
```

The output is shown as follows.

```
complex(-1.797086,2.213723) complex(-1.797086,-2.213723) complex(-0.262914,2.703862)
complex(-0.262914,-2.703862)
```

See Also
None.

CControl ::ss2ss

Synopsis
CControl* ss2ss (array double &T);

Purpose
State coordinate transformation for state-space models.

Return Value
Upon successful completion, a non-NULL pointer to class **CControl** representing the object of the transformatted system is return. Otherwise, a NULL pointer is returned.

Parameters
T Computational array containing the state coordinate transformation.

Description
The function **ss2ss** () performs a state coordinate transformation $\bar{X} = TX$ for state-space models. Assume the system sys has the following state equations,

$$\begin{aligned}\dot{X} &= AX + BU \\ \dot{Y} &= CX + DU\end{aligned}$$

This function with argument T will make a similarity transformation on the state vector X, and returns the following model with equations,

$$\begin{aligned}\dot{\bar{X}} &= TAT^{-1}\bar{X} + TBU \\ \dot{Y} &= CT^{-1}\bar{X} + DU\end{aligned}$$

Example

```
#include <control.h>

#define NUMX 4    // number of states
#define NUMU 1    // number of inputs
#define NUMY 1    // number of outputs

int main() {
    array double A[NUMX][NUMX] = {{-4.12, -17.4, -30.8, -60},
                                  {1,     0,     0,     0},
                                  {0,     1,     0,     0},
                                  {0,     0,     1,     0}},
                 B[NUMX][NUMU] = {1, 0, 0, 0},
                 C[NUMY][NUMX] = {4, 8.4, 30.78, 60};
    array double T[NUMX][NUMX] = {{-4.12, -17.4, -30.8, -60},
                                  {2,     0,     0,     10},
                                  {0,     3,     9,     0},
                                  {0,     0,     4,     0}};
    double D = 0;
    CControl sys, *sysT;

    sys.model("ss", A, B, C, NULL);
    sysT = sys.ss2ss(T);
    sysT->printss();
    return 0;
}
```

The output is shown as follows.

```
State-space arguments:
A =
 -6.328122 -13.248731 -23.073773  19.913452
  2.000000   0.000000   0.000000   2.500000
  0.380711   2.284264   5.208122  -8.786802
  0.000000   0.000000   1.333333  -3.000000
B =
 -4.120000
  2.000000
  0.000000
  0.000000
C =
 -1.015228  -0.091371  -3.088325   6.826472
D =
  0.000000
```

See Also
CControl::c2d(), **CControl::d2c**(), **CControl::d2d**().

CControl ::**ssdata**

Synopsis
int **ssdata** (array double &A, array double &B, array double &C, array double &D);

Purpose
Retrieve matrices of the state-space model.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters

A Array of reference containing the output of the system matrix A.

B Array of reference containing the output of the input matrix B.

C Array of reference containing the output of the output matrix C.

D Array of reference containing the output of the direct transmission matrix D.

Description
The function **ssdata** () retrieves system, input, output, direct transmission matrices from a state-space model. If the matrix is not available in some special system type, all its elements will be 0. The **CControl** ::**size** ()

can be used to determine sizes of these matrices.

Example

In this program, the system is created in transfer function model first.

```
#include <control.h>
int main() {
   array double num[4] = {4, 8.4, 30.8, 60}, den[5] = {1, 4.12, 17.4, 30.8, 60};
   CPlot plot;
   int nx, ny, nu;
   CControl sys;

   sys.model("tf", num, den); // get a tf model
   nx = sys.size('x');  // get the order of system
   ny = sys.size('y');  // get the number of output of the system
   nu = sys.size('u');  // get the number of input of the system
   array double A[nx][nx], B[nx][nu], C[ny][nx], D[ny][nu];

   sys.ssdata(A, B, C, D);
   printf("A = %f\nB = %f\nC = %f\nD = %f\n", A, B, C, D);
   return 0;
}
```

The output is shown as follows.

```
A = -4.120000 -17.400000 -30.800000 -60.000000
1.000000 0.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000

B = 1.000000
0.000000
0.000000
0.000000

C = 4.000000 8.400000 30.800000 60.000000

D = 0.000000
```

See Also
CControl::tfdata(), **CControl::zpkdata**().

CControl ::step

Synopsis
int **step** (class **CPlot** *$plot$, array double &$yout$, array double &$tout$, array double &$xout$, $\ldots$/* double tf */);

Purpose
Calculate and plot step response of the system.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters
$plot$ Pointer to an existing object of class **CPlot** .

$yout$ Array of reference containing the output of the step response sequence.

$tout$ Array of reference containing the time vector used for simulation.

$xout$ Array of reference containing the state trajectories.

... The following argument is optional.

tf Double value specifying the final time of the simulation.

Description
The function **step** () computes and plots the step response of LTI system. The step response is defined as the response of system to a unit step input with the zero initial state. If system output array $yout$ is not specified, the default value 100 of macro CHTKT_CONTROL_SAMPLE_POINTS, which is defined in file control.h, is used as the number of time samples. When $yout$ is specified, the number of time samples is the same as the extent of its most right dimension. If the system is a multi-input system, the output is the collection of step responses for each input channel, and data of every input/output channel will be displayed in a seperate subplot. If the optional argument tf is absent, the function will determine the duration of simulation automatically on the transient behavior of the response.

Example
The example outputs the step responses of a two-input-two-output system.

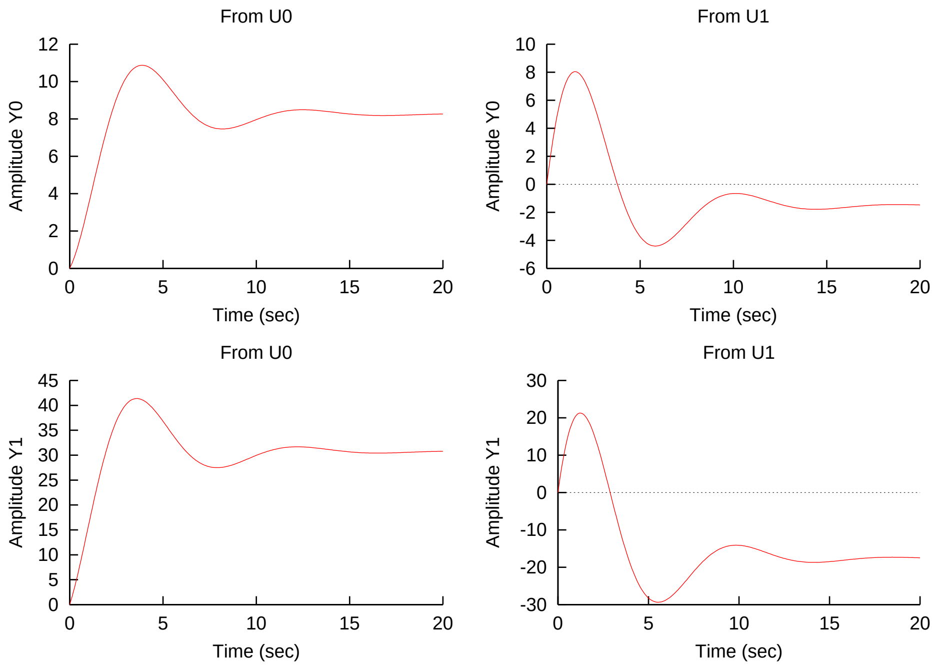
```
#include <control.h>

#define N    300      // number of time samples
#define TF   20       // final time
#define NUMX 2        // number of states
#define NUMU 2        // number of inputs
#define NUMY 2        // number of outputs

int main() {
     array double A[NUMX][NUMX] = { -0.5572,   -0.7814,
                                     0.7814,    0};
     array double B[NUMX][NUMU] = { 1, -1,    // 2 inputs
                                    0,  2 };
     array double C[NUMY][NUMX] = { 1.9691,  6.4493,  // 2 outputs
                                    12.0,  24.0 };
     CPlot plot;
     array double y[NUMU][NUMY][N];
     CControl sys;

     sys.model("ss", A, B, C, NULL);

     // plot and get the response in y with plotting
     sys.step(&plot, y, NULL, NULL, TF);

     /* print the response y */
        int i, j, k;
        for (i = 0; i < NUMU; i ++) { // for each input
            printf("\nResponse from U%d \n", i);
            for (j = 0; j < NUMY; j++) {  // for each output
                printf("to Y%d \n", j);
                for (k = 0; k < N; k += 30)  // for every 30 time samples
                    printf("y[%d][%d][%d] = %f\n", i, j, k, y[i][j][k]);
            }
        }
     return 0;
}
```

The output is shown as follows.

```

Response from U0
to Y0
y[0][0][0] = 0.000000
y[0][0][30] = 7.497842
y[0][0][60] = 10.860255
y[0][0][90] = 8.815381
y[0][0][120] = 7.469260
y[0][0][150] = 7.975130
y[0][0][180] = 8.476264
y[0][0][210] = 8.371425
y[0][0][240] = 8.194947
y[0][0][270] = 8.207903
to Y1
y[0][1][0] = 0.000000
y[0][1][30] = 31.540688
y[0][1][60] = 40.854114
y[0][1][90] = 31.668591
y[0][1][120] = 27.514701
y[0][1][150] = 30.015671
y[0][1][180] = 31.675605
y[0][1][210] = 31.058543
y[0][1][240] = 30.441397
y[0][1][270] = 30.568572

Response from U1
to Y0
y[1][0][0] = 0.000000
y[1][0][30] = 7.411369
y[1][0][60] = -0.941325
y[1][0][90] = -4.372861
y[1][0][120] = -2.056892
y[1][0][150] = -0.655433
y[1][0][180] = -1.243238
y[1][0][210] = -1.772357
y[1][0][240] = -1.644115
y[1][0][270] = -1.455661
to Y1
y[1][1][0] = 0.000000
y[1][1][30] = 15.299495
y[1][1][60] = -19.409946
y[1][1][90] = -28.603363
y[1][1][120] = -18.367507
y[1][1][150] = -14.125990
y[1][1][180] = -16.959721
y[1][1][210] = -18.688490
y[1][1][240] = -17.970965
```

```
y[1][1][270] = -17.319184
```

The plot with four seperate subplots is shown below.



See Also
CControl::step(), **CControl::impulse**(), **CControl::initial**(), **CControl::lsim**().

CControl ::stepinfo

Synopsis
int **stepinfo**(double *$risetime$, double *$settlingtime$, double *$overshoot$, double *$peakvalue$, double *$peaktime$, double *$undershoot$, double *$settlingmin$, double *$settlingmax$, $\ldots$ /* double *$ts\_percentage$, double *$tr\_lowerlimit$, double *$tr\_upperlimit$ */);

Purpose
Compute the step response characteristics of an LTI system.

Return Value
Upon successful completion, zero is returned. Otherwise, -1 is returned.

Parameters
$risetime$ Pointer to double value containing the rise time of the response.

$settlingtime$ Pointer to double value containing the settling time of the response.

$overshoot$ Pointer to double value containing the percent overshoot of the response.

$peakvalue$ Pointer to double value containing the peak absolute value of the response.

$peaktime$ Pointer to double value containing the time when the peak absolute value is reached.

$undershoot$ Pointer to double value containing the percent undershoot of the response.

$settlingmin$ Pointer to double value containing the minimum response value once the response has risen.

$settlingmax$ Pointer to double value containing the maximum response value once the response has risen.

... The following arguments are optional.

$ts\_percentage$ Pointer to double value containing the settling time percentage.

$tr\_lowerlimit$ Pointer to double value containing the rise time lower limit.

$tr\_upperlimit$ Pointer to double value containing the rise time upper limit.

Description
The function **stepinfo()** computes and returns the step response characteristics for an LTI system. The returned performance indicators include (1) rise time, (2) settling time, (3) percent overshoot, (4) peak absolute value of the response, (5) time at which peak absolute value is reached, (6) percent undershoot, (7) minimum response value once the response has risen, and (8) maxmimum response value once the response has risen. If any of the eight required arguments is specified as NULL, the corresponding performance indicator will not be returned. Note that the three optional arguments, $ts\_percentage$, $tr\_lowerlimit$, and $tr\_upperlimit$, have to be ALL ABSENT or ALL SPECIFIED. If the optional arguments are all absent, the function will use their default values. The default values are 0.02, 0.1, and 0.9 for $ts\_percentage$, $tr\_lowerlimit$, and $tr\_upperlimit$, respectively. If the optional arguments are all specified, the function will use the specified values correspondingly. However, if only some the optional arguments need to be assigned values, the remaining optional arguments can be specified as NULL.

Example 1
This example outputs all the step response characteristics of a system without specifying the three optional arguments – settling time percentage, rise time lower limit, and rise time upper limit. The default values of these three parameters are used for calculations.

```
#include "control.h"

int main() {
  array double num[] = {1, 5},
               den[] = {1, 2, 5, 7, 2};
  double tr, ts, os, pv, tp, us, smin, smax;
  CControl sys;

  sys.model("tf", num, den);
  sys.stepinfo(&tr, &ts, &os, &pv, &tp, &us, &smin, &smax);

  printf("Rise time: %f\n", tr);
  printf("Settling time: %f\n", ts);
  printf("Percent overshoot: %f\n", os);
  printf("Peak value: %f\n", pv);
  printf("Peak time: %f\n", tp);
  printf("Percent undershoot: %f\n", us);
  printf("Settling Min.: %f\n", smin);
  printf("Settling Max.: %f\n", smax);

  return 0;
}
```

The output of Example 1 is shown as follows.

```
Rise time: 4.686024
Settling time: 13.945610
Percent overshoot: 0.798522
Peak value: 2.520273
Peak time: 15.267267
Percent undershoot: 0.000000
Settling Min.: 2.189444
Settling Max.: 2.520273
```

Example 2
This example outputs all the step response characteristics of a system with specifying all the three optional arguments in such a way that only the lower and uppper limits of the rise time are specified.

```
#include "control.h"

int main() {
  array double num[] = {1, 5},
               den[] = {1, 2, 5, 7, 2};
  double tr, ts, os, pv, tp, us, smin, smax, trll = 0.05, trul = 0.95;
  CControl sys;

  sys.model("tf", num, den);
  sys.stepinfo(&tr, &ts, &os, &pv, &tp, &us, &smin, &smax, NULL, &trll, &trul);

  printf("Rise time: %f\n", tr);
  printf("Settling time: %f\n", ts);
  printf("Percent overshoot: %f\n", os);
  printf("Peak value: %f\n", pv);
  printf("Peak time: %f\n", tp);
  printf("Percent undershoot: %f\n", us);
  printf("Settling Min.: %f\n", smin);
  printf("Settling Max.: %f\n", smax);

  return 0;
}
```

The output of Example 2 is shown as follows.

```
Rise time: 7.446058
Settling time: 13.945610
Percent overshoot: 0.798522
Peak value: 2.520273
Peak time: 15.267267
Percent undershoot: 0.000000
Settling Min.: 2.373753
Settling Max.: 2.520273
```

Example 3
This example is the same as Example 1 except that it only outputs some the step response characteristics of a system.

```
#include "control.h"

int main() {
  array double num[] = {1, 5},
               den[] = {1, 2, 5, 7, 2};
  double tr, ts, os, pv, tp;
  CControl sys;

  sys.model("tf", num, den);
  sys.stepinfo(&tr, &ts, &os, &pv, &tp, NULL, NULL, NULL);

  printf("Rise time: %f\n", tr);
  printf("Settling time: %f\n", ts);
  printf("Percent overshoot: %f\n", os);
  printf("Peak value: %f\n", pv);
  printf("Peak time: %f\n", tp);

  return 0;
}
```

The output of Example 3 is shown as follows.

```
Rise time: 4.686024
Settling time: 13.945610
Percent overshoot: 0.798522
Peak value: 2.520273
Peak time: 15.267267
```

See Also
CControl::step().

CControl ::tfdata

Synopsis
int **tfdata** (array double &num, array double &den);

Purpose
Retrieve coefficients of the numerator and denominator of the transfer function model.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters
num Array of reference containing the coefficients of the numerator.

den Array of reference containing the coefficients of the denominator.

Description
The function **tfdata** () retrieves coefficients of the numerator and denominator of the SISO transfer function model.
The **CControl** ::**size** () can be used to determine the number of these coefficients.

Example

In this program, the system is created in state-space model first.

```
#include <control.h>

#define NUMX 4    // number of states
#define NUMU 1    // number of inputs
#define NUMY 1    // number of outputs

int main() {
    array double A[NUMX][NUMX] = {{-4.12, -17.4, -30.8, -60},
                                  {1,     0,     0,     0},
                                  {0,     1,     0,     0},
                                  {0,     0,     1,     0}},
                 B[NUMX][NUMU] = {1, 0, 0, 0},
                 C[NUMY][NUMX] = {4, 8.4, 30.8, 60};
    int nnum, nden;
    CControl sys;

    sys.model("ss", A, B, C, NULL);

    nnum = sys.size('n');
    nden = sys.size('d');
    array double num[nnum], den[nden];

    sys.tfdata(num, den);
    printf("num = %f\nden = %f\n", num, den);
    return 0;
}
```

The output is shown as follows.

```
num = 0.000000 4.000000 8.400000 30.800000 60.000000

den = 1.000000 4.120000 17.400000 30.800000 60.000000
```

See Also
CControl::ssdata(), **CControl::zpkdata**().

CControl ::tzero

Synopsis
int **tzero** (array double complex &$zeros$, . . . /* double *$gain$ */);

Purpose
Return the transmission zeros of the LTI system.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters

$zeros$ Array of reference containing the output transmission zeros of the system.

$dots$ The following argument is optional.

$gain$ Pointer to double value indicating the gain of the system.

Description
The function **tzero** () returns the transmission zeros and gain of the LTI system. This function is applicable to SISO system in the current implementation.

Example

```
#include <control.h>

#define NUMX 4    // number of states
#define NUMU 1    // number of inputs
#define NUMY 1    // number of outputs
#define TF   15   // final time

int main() {
    array double A[NUMX][NUMX] = {{-4.12, -17.4, -30.8, -60},
                                  {1,     0,     0,     0},
                                  {0,     1,     0,     0},
                                  {0,     0,     1,     0}},
                 B[NUMX][NUMU] = {1, 0, 0, 0},
                 C[NUMY][NUMX] = {4, 8.4, 30.78, 60};
    CControl sys;
    double gain;
    int nz;

    sys.model("ss", A, B, C, NULL);
    nz = sys.size('z');
    array double complex zeros[nz];

    sys.tzero(zeros, &gain);
    printf("zeros = %f\ngain = %f\n", zeros, gain);
    return 0;
}
```

The output is shown as follows.

```
zeros = complex(-0.049557,2.737558) complex(-0.049557,-2.737558) complex(-2.000885,0.000000)


gain = 4.000000
```

See Also
CControl::pole(), **CControl::pzmap**().

CControl ::zgrid

Synopsis
int **zgrid** (int $flag$, $\ldots$/* array double z[:], array double w[:] */);

Purpose
Generate a z-plane grid of the constant damping factors and natural frequencies.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters

$flag$ Integer to turn on/off the zgrid. The value 1 turns on the zgrid, whereas 0 turns off the zgrid.

... The following two arguments are optional.

z Computational array containing the damping factors at which the grid lines are plotted.

w Computational array containing the natural frequencies at which the grid lines are plotted.

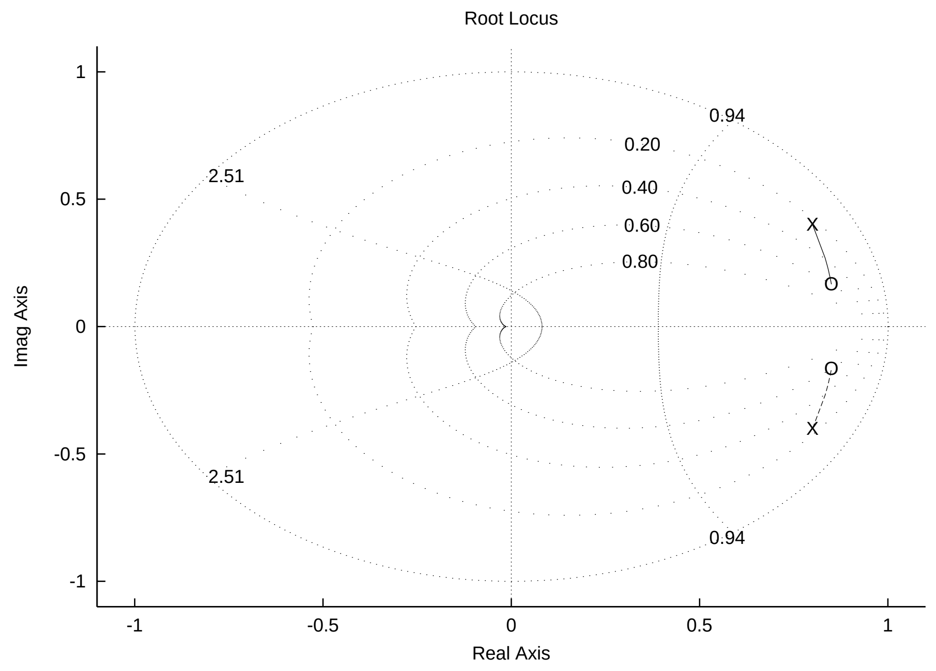
Description
The function **zgrid** () generates grid lines of constant damping factors and natural frequencies over the discrete z-plane plots produced by function **CControl** ::**rlocus** () and **CControl** ::**pzmap** (). If the optional arguments z and w are absent, a grid of constant damping factors from 0 to 1 in steps of 0.1 and natural frequencies from 0 to π in steps of $\frac{pi}{10}$ will be plotted.

Example
The following example adds grid lines on both rlocus plot and pzmap plot of system sys.

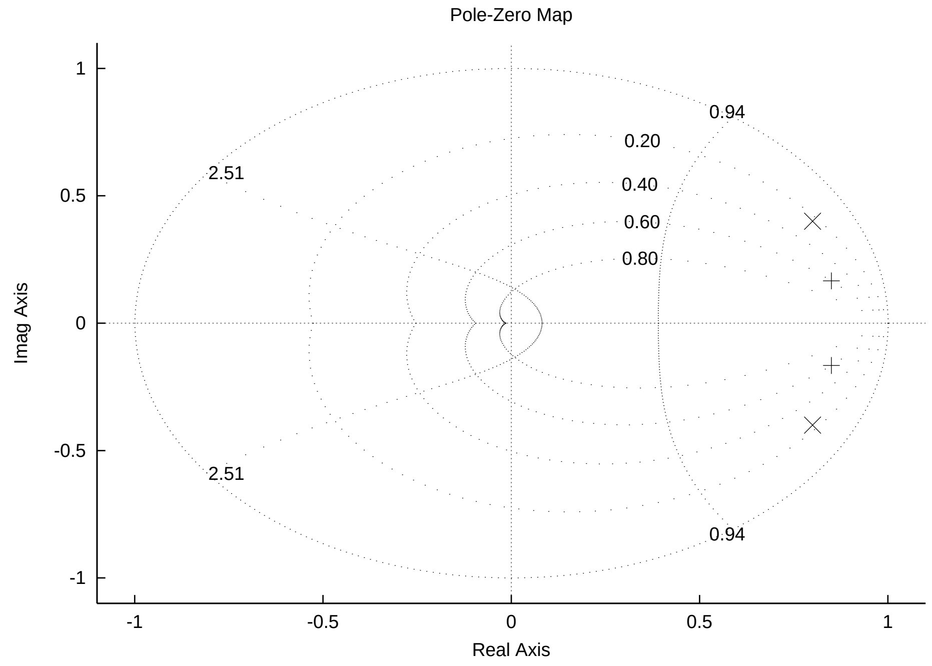
```
#include <control.h>

int main() {
   int nx;
   CPlot plot1, plot2;
   array double num[3] = {2, -3.4, 1.5}, den[3] = {1, -1.6, 0.8};
   CControl sys;

   sys.model("tf", num, den, -1);
   nx = sys.size('x');
   array double z[4] = {.2, .4, .6, .8}, w[2] = {M_PI*3/10, M_PI*4/5};
   sys.zgrid(1, z, w); // turn on zgrid

   array double complex r[nx][30];
```

```
    array double kout[30];
    sys.rlocus(&plot1, r, kout);

    int np = sys.size('p');
    int nz = sys.size('z');
    array double complex poles[np];
    array double complex zeros[nz];
    sys.pzmap(&plot2, poles, zeros);

    return 0;
}
```

The rlocus plot with grid lines is shown in the following figure.



The pzmap plot with grid lines is shown in the following figure.



See Also
CControl::pzmap(), **CControl::rlocus**(), **CControl::sgrid**(), **CControl::ngrid**().

CControl ::zpkdata

Synopsis
int **zpkdata** (array double complex &z, array double complex &p, double *k);

Purpose
Retrieve zeros, poles and gain of a SISO zero-pole-gain model.

Return Value
Upon successful completion, zero is returned. Otherwise, a value of non-zero is returned.

Parameters
z Array of reference containing zeros of the system.

p Array of reference containing poles of the system.

k Pointer to double value indicating the gain of the system.

Description
The function **zpkdata** () retrieves zeros, poles and gain of SISO zero-pole-gain model.
The **CControl** ::**size** () can be used to determine the number of zeros and poles.

Example

In this program, the system is created in state-space model first.

```
#include <control.h>

#define NUMX 4    // number of states
#define NUMU 1    // number of inputs
#define NUMY 1    // number of outputs

int main() {
    array double A[NUMX][NUMX] = {{-4.12, -17.4, -30.8, -60},
                                  {1,     0,     0,     0},
                                  {0,     1,     0,     0},
                                  {0,     0,     1,     0}},
                 B[NUMX][NUMU] = {1, 0, 0, 0},
                 C[NUMY][NUMX] = {4, 8.4, 30.78, 60};

    CControl sys;
    int nz, np;
    double k;

    sys.model("ss", A, B, C, NULL);
    nz = sys.size('z');
    np = sys.size('p');
    array double complex z[nz], p[np];

    sys.zpkdata(z, p, &k);
    printf("z = %f\n p = %f\n k = %f\n", z, p, k);
    return 0;
}
```

The output is shown as follows.

```
z = complex(-0.049557,2.737558) complex(-0.049557,-2.737558) complex(-2.000885,0.000000)


 p = complex(-1.797086,2.213723) complex(-1.797086,-2.213723) complex(-0.262914,2.703862)
complex(-0.262914,-2.703862)

 k = 4.000000
```

See Also
CControl::ssdata(), **CControl::tfdata**().

Appendix B

Syntax Comparison of Ch Control System Toolkit with MATLAB Control System Toolbox

Almost all features in MATLAB Control System Toolbox are available in Ch and Ch Control System Toolkit. To ease the porting of code written in MATLAB to Ch and comparison study, the syntax comparison of Ch Control System Toolkit with MATLAB Control System Toolbox is presented in this appendix.

Table B.1: Syntax Comparison of Ch Control Toolkit with MATLAB Control Toolbox.

MATLAB Control Toolbox	**Ch Control Toolkit**
k = acker(A,b,p);	sys.model(”ss”, A, b, NULL, NULL);
	k = sys.acker(A, b, p);
sys = append(sys1,sys2,...,sysN);	sys = sys1.append(sys2, ..., sysN);
asys = augstate(sys);	asys = sys.augstate();
fb = bandwidth(sys);	fb = sys.bandwidth();
bodemag(sys);	sys.bodemag(plot, NULL, NULL);
bodemag(sys,{wmin, wmax});	sys.bodemag(plot, NULL, NULL, wmin, wmax);
w={wmin, wmax}; bodemag(sys, w);	sys.bodemag(plot, NULL, NULL, wmin, wmax);
bode(sys);	sys.bode(plot, NULL, NULL, NULL);
bode(sys,{wmin, wmax});	sys.bode(plot, NULL, NULL, NULL, wmin, wmax);

Table B.1: Syntax comparison of Ch Control Toolkit with MATLAB Control Toolbox (continued).

MATLAB Control Toolbox	**Ch Control Toolkit**
w={wmin, wmax}; bode(sys, w[1]);	sys.bode(plot, NULL, NULL, NULL, wmin, wmax);
w=[w0,w1,w2, ..., wn]; bode(sys, w);	double w[]=w0,w1,w2, ..., wn;
	sys.bode(plot, NULL, NULL, NULL, w);
w=logspace(wmin, wmax); bode(sys, w);	logspace(w,wmin,wmax);
	sys.bode(plot, NULL, NULL, NULL, w);
w=logspace(wmin, wmax, n); bode(sys, w);	array double w[n]; logspace(w, wmin, wmax);
	sys.bode(plot, NULL, NULL, NULL, w);
[mag,phase,wout]=bode(sys);	sys.bode(NULL, mag, phase, wout);
[mag,phase,wout]=bode(sys, {wmin, wmax});	sys.bode(NULL, mag, phase, wout, wmin, wmax);
[mag,phase]=bode(sys, w);	sys.bode(NULL, mag, phase, NULL, w);
sysd = c2d(sys, Ts, method);	sysd = sys.c2d(Ts, method);
csys = canon(sys);	csys = sys.canon(NULL);
[csys, T]; = canon(sys)	csys = sys.canon(T);
csys = canon(sys, ’type’);	csys = sys.canon(NULL, ’type’);
[csys, T] = canon(sys, ’type’);	csys = sys.canon(T, ’type’);
sysc = connect(sys,[]);	sysc = sys.connect(NULL, NULL, NULL);
sysc = connect(sys,Q);	sysc = sys.connect(Q, NULL, NULL);
sysc = connect(sys,Q,inputs,outputs);	sysc = sys.connect(Q, inputs, outputs);
co = ctrb(sys);	sys.ctrb(co);
co = ctrb(A, B);	sys.model(”ss”, A, B, NULL, NULL);
	sys.ctrb(co);
[Abar,Bbar,Cbar,T,k] = ctrbf(A,B,C);	sys.model(”ss”, A, B, C, NULL);
	sys.ctrbf(Abar, Bbar, Cbar, T, k, A, B, C);
[Abar,Bbar,Cbar,T,k] = ctrbf(A,B,C,tol);	sys.model(”ss”, A, B, C, NULL);
	sys.ctrbf(Abar, Bbar, Cbar, T, k, A, B, C, tol);
sysc = d2c(sysd);	sysc = sysd.d2c();
sysc = d2c(sysd,method);	sysc = sysd.d2c(method);
sys1 = d2d(sys,ts);	sys1 = sys.d2d(ts);
damp(sys);	sys.damp();
[wn,z] = damp(sys);	sys.damp(wn, z);
[wn,z,p] = damp(sys);	sys.damp(wn, z, p);
K = dcgain(sys);	K = sys.dcgain();
[k,s,e] = dlqr(a,b,q,r);	sys.model(”ss”, a, b, NULL, NULL);
	sys.dlqr(k, s, e, q, r);
[k,s,e] = dlqr(a,b,q,r,n);	sys.model(”ss”, a, b, NULL, NULL);
	sys.dlqr(k, s, e, q, r, n);

[1]The argument w in MATLAB could be either {wmin, wmax} to focus on the particular frequency interval [wmin, wmax], or a vector of desired frequencies to use particular frequency points. In Ch, they have different prototypes.

Table B.1: Syntax comparison of Ch Control Toolkit with MATLAB Control Toolbox (continued).

MATLAB Control Toolbox	**Ch Control Toolkit**
x = dlyap(a, q);	sys.dlyap(x, a, q);
sys = drss(n);	sys.model("drss", n);
sys = drss(n,p);	sys.model("drss", n, p);
sys = drss(n,p,m);	sys.model("drss", n, p, m);
s = dsort(p);	sys.model("zpk", z, p, k, -1); s = sys.dsort();
s = esort(p);	sys.model("zpk", z, p, k); s = sys.esort();
est = estim(sys,l);	est = sys.estim(l);
est = estim(sys,l,sensors,known);	est = sys.estim(l, sensors, known);
sys = feedback(sys1,sys2);	sys = sys1.feedback(sys2);
sys = feedback(sys1,sys2,sign);	sys = sys1.feedback(sys2, sign);
sys = feedback(sys1,sys2, feedin, feedout, sign);	sys = sys1.feedback(sys2, feedin, feedout, sign);
grid on	sys.grid(1);
grid off	sys.grid(0);
grid	sys.grid(0); or sys.grid(1);
impulse(sys);	sys.impulse(plot, NULL, NULL, NULL);
impulse(sys, tf);	sys.impulse(plot, NULL, NULL, NULL, tf);
yout = impulse(sys);	sys.impulse(NULL, yout, NULL, NULL);
[y, tout] = impulse(sys);	sys.impulse(NULL, yout, tout, NULL);
[y, tout, xout] = impulse(sys);	sys.impulse(NULL, yout, tout, xout);
yout = impulse(sys, tf);	sys.impulse(NULL, yout, NULL, NULL, tf);

yout = impulse(sys, 0:dt:tf);	double y[tf/dt+1];
	sys.impulse(NULL, yout, NULL, NULL, tf);

initial(sys, x0);	sys.initial(plot, NULL, NULL, NULL, x0);
initial(sys, x0, 0:dt:tf);	sys.initial(plot, NULL, NULL, NULL, x0, tf);
yout = initial(sys, x0);	sys.initial(NULL, yout, NULL, NULL, x0);
yout = initial(sys, x0, 0:dt:tf);	sys.initial(NULL, yout, NULL, NULL, x0, tf);
[yout, tout] = initial(sys, x0);	sys.initial(NULL, yout, tout, NULL, x0);
[yout, tout] = initial(sys, x0, 0:dt:tf);	sys.initial(NULL, yout, tout, NULL, x0, tf);
[yout, tout, xout] = initial(sys, x0);	sys.initial(NULL, yout, tout, xout, x0);
[yout, tout, xout] = initial(sys, x0, 0:dt:tf);	sys.initial(NULL, yout, tout, xout, x0, tf);

r = input('message to user');	char s[30]; double r;
	printf("message to user");
	scanf("%s", s); or getline(_stdio, s, 30);
	r = streval(s);

r = input('message to user', 's');	char r[30];
	printf("message to user");
	scanf("%s", r) or getline(_stdio, r, 30);

[l,p,e] = lqe(a,g,c,q,r);	sys.model("ss", a, NULL, c, NULL); sys.lqe(l, p, e, g, q, r);
[l,p,e] = lqe(a,g,c,q,r,n);	sys.model("ss", a, NULL, c, NULL); sys.lqe(l, p, e, g, q, r, n);

Table B.1: Syntax comparison of Ch Control Toolkit with MATLAB Control Toolbox (continued).

MATLAB Control Toolbox	**Ch Control Toolkit**
[k,s,e] = lqr(a,b,q,r);	sys.model(”ss”, a, b, NULL, NULL);
	sys.lqr(k, s, e, q, r);
[k,s,e] = lqr(a,b,q,r,n);	sys.model(”ss”, a, b, NULL, NULL);
	sys.lqr(k, s, e, q, r, n);
lsim(sys, u, 0:dt:tf);	array double u[tf/dt+1];
	sys.lsim(plot, NULL, NULL, NULL, u, tf);
lsim(sys, u, t);	sys.lsim(plot, NULL, NULL, NULL, u, tf);
lsim(sys, u, t, x0);	sys.lsim(plot, NULL, NULL, NULL, u, tf, x0);
y = lsim(sys, u, t, x0);	sys.lsim(plot, y, NULL, NULL, u, tf, x0);
[y, tout] = lsim(sys, u, t);	sys.lsim(plot, y, tout, NULL, u, tf);
[y, tout] = lsim(sys, u, t, x0);	sys.lsim(plot, y, tout, NULL, u, tf, x0);
[y, tout, xout] = lsim(sys, u, t);	sys.lsim(plot, y, tout, xout, u, tf);
[y, tout, xout] = lsim(sys, u, t, x0);	sys.lsim(plot, y, tout, xout, u, tf, x0);
margin(sys);	sys.margin(plot, NULL, NULL, NULL, NULL);
[gm,pm,wcg,wcp] = margin(sys);	sys.margin(NULL, gm, pm, wcg, wcp);
[Gm,Pm,Wcg,Wcp] = margin(mag,phase,w);	sys.margin(NULL, gm, pm, wcg, wcp, mag, phase, w);
[Am,Bm,Cm,Dm] = minreal(A,B,C,D);	sys.model(”ss”, A, B, C, D);
	sys2 = sys.minreal();
	sys2->ssdata(Am, Bm, Cm, Dm);
sysr = minreal(sys);	sys.model(”ss”, A, B, C, NULL);
	sys2 = sys.minreal(NULL);
	sys2 = sys.pzcancel();
sysr = minreal(sys, tol);	sys.model(”ss”, A, B, C, NULL);
	sys2 = sys.minreal(NULL, tol);
	sys2 = sys.pzcancel(tol);
[sysr, u] = minreal(sys, tol);	sys.model(”ss”, A, B, C, NULL);
	sys2 = sys.minreal(u, tol);
wc = gram(sys,’c’);	sys.model(”ss”, A, B, NULL, NULL);
	sys.gram(wc, ’c’);
wo = gram(sys,’o’);	sys.model(”ss”, A, NULL, C, NULL);
	sys.gram(wo, ’o’);
ngrid	sys.ngrid(1);
ngrid(’new’)	sys.ngrid(1);
nichols(H); ngrid	sys.ngrid(1); sys.nichols();
ngrid(’new’); nichols(H);	sys.ngrid(1); sys.nichols();
nichols(sys);	sys.nichols(plot, NULL, NULL, NULL);
nichols(sys,{wmin, wmax});	sys.nichols(plot, NULL, NULL, NULL, wmin, wmax);
w={wmin, wmax}; nichols(sys, w);	sys.nichols(plot, NULL, NULL, NULL, wmin, wmax);
w=[w0,w1,w2, ..., wn]; nichols(sys, w);	double w[]=w0,w1,w2, ..., wn;
	sys.nichols(plot, NULL, NULL, NULL, w);
w=logspace(wmin, wmax); nichols(sys, w);	logspace(w,wmin,wmax);
	sys.nichols(plot, NULL, NULL, NULL, w);
w=logspace(wmin, wmax, n); nichols(sys, w)	array double w[n]; logspace(w, wmin, wmax);
	sys.nichols(plot, NULL, NULL, NULL, w);

Table B.1: Syntax comparison of Ch Control Toolkit with MATLAB Control Toolbox (continued).

MATLAB Control Toolbox	**Ch Control Toolkit**
[mag,phase,wout]=nichols(sys);	sys.nichols(NULL, mag, phase, wout)
[mag,phase,wout]=nichols(sys, {wmin, wmax});	sys.nichols(NULL, mag, phase, wout, wmin, wmax);
[mag,phase]=nichols(sys, w);	sys.nichols(NULL, mag, phase, NULL, w);
nyquist(sys);	sys.nyquist(plot, NULL, NULL, NULL);
nyquist(sys,w);	sys.nyquist(plot, NULL, NULL, NULL, w);
[re,im,wout] = nyquist(sys);	sys.nyquist(NULL, re, im, wout)
[re,im] = nyquist(sys,w);	sys.nyquist(NULL, w, re, NULL, w);
ob = obsv(sys);	sys.obsv(ob);
ob = obsv(A, C);	sys.model(”ss”, A, NULL, C, NULL);
	sys.obsv(ob);
[Abar,Bbar,Cbar,T,k] = obsvf(A,B,C);	sys.model(”ss”, A, B, C, NULL);
	sys.obsvf(Abar, Bbar, Cbar, T, k, A, B, C);
[Abar,Bbar,Cbar,T,k] = obsvf(A,B,C,tol);	sys.model(”ss”, A, B, C, NULL);
	sys.obsvf(Abar, Bbar, Cbar, T, k, A, B, C, tol);
sys = parallel(sys1, sys2);	sys = sys1.parallel(sys2);
sys = parallel(sys1, sys2, inp1, inp2, out1, out2);	sys = sys1.parallel(sys2, input1, input2, output1, output2);
k = place(A,B,p);	sys.model(”ss”, A, B, NULL, NULL);
	sys.place(k, p);
p = pole(sys);	sys.pole(p);
N/A	sys.printss();
N/A	sys.printSystem();
N/A	sys.printtf();
N/A	sys.printzpk();
pzmap(sys);	sys.pzmap(plot, NULL, NULL);
[p,z]=pzmap(sys);	sys.pzmap(NULL, p, z);
N/A	sys.pzmap(plot, p, z);
rlocus(sys);	sys.rlocus(plot, NULL, NULL);
rlocus(sys,k);	sys.rlocus(plot, NULL, NULL, k);
[r,kout] = rlocus(sys);	sys.rlocus(NULL, r, kout);
[r] = rlocus(sys, k);	sys.rlocus(NULL, r, NULL, k);
N/A	sys.rlocus(plot, r, kout);
N/A	sys.rlocus(plot, r, NULL, k);
rlocfind(sys, p);	sys.rlocfind(NULL, NULL, p);
[k, poles] = rlocfind(sys, p);	sys.rlocfind(k, poles, p);
sys = rss(n);	sys.model(”rss”, n);
sys = rss(n,p);	sys.model(”rss”, n, p);
sys = rss(n,p,m);	sys.model(”rss”, n, p, m);
sys = series(sys1, sys2);	sys = sys1.series(sys2);
sys = series(sys1, sys2, out1, inp2);	sys = sys1.series(sys2, output1, input2);

Table B.1: Syntax comparison of Ch Control Toolkit with MATLAB Control Toolbox (continued).

MATLAB Control Toolbox	**Ch Control Toolkit**
sgrid	sys.sgrid(1)[2];
sgrid('new')	sys.sgrid(1);
sgrid(z, wn)	sys.sgrid(1, z, wn);
sgrid(z, wn, 'new');	sys.sgrid(1, z, wn);
pzmap(H); sgrid	sys.sgrid(1); sys.pzmap();
sgrid('new'); pzmap(H);	sys.sgrid(1); sys.pzmap()[3];
pzmap(H); sgrid(z, wn)	sys.sgrid(1, z, wn); sys.pzmap();
sgrid(z, wn, 'new'); pzmap(H)	sys.sgrid(1, z, wn); sys.pzmap();
nx = size(sys, 'order');	nx = sys.size('x');
ny = size(sys, 1);	ny = sys.size('y');
nu = size(sys, 2);	nu = sys.size('u');
N/A	nn = sys.size('n');
N/A	nd = sys.size('d');
N/A	nz = sys.size('z');
N/A	np = sys.size('p');
N/A	nk = sys.size('k');
[A,B,C,D] = ssdata(sys);	int nx, ny, nu;
	nx = sys.size('x');
	ny = sys.size('y');
	nu = sys.size('u');
	array double A[nx][nx], B[nx][nu],
	C[ny][nx], D[ny][nu];
	sys.ssdata(A, B, C, D);
step(sys);	sys.step(plot, NULL, NULL, NULL);
step(sys, tf);	sys.step(plot, NULL, NULL, NULL, tf);
[yout, tout] = step(sys);	sys.step(NULL, yout, tout, NULL);
[yout, tout, xout] = step(sys);	sys.step(NULL, yout, tout, xout);
yout = step(sys, tf);	sys.step(NULL, yout, NULL, NULL, tf);
yout = step(sys, 0:dt:tf);	double y[tf/dt+1];
	sys.step(NULL, yout, NULL, NULL, tf);
N/A	sys.step(plot, yout, NULL, NULL);
N/A	sys.step(plot, yout, NULL, NULL, tf);
step(sys);	sys.step(plot, NULL);
step(sys, tf);	sys.step(plot, NULL, tf);
y = step(sys);	sys.step(NULL, y);
y = step(sys, tf);	sys.step(NULL, y, tf);
y = step(sys, 0:dt:tf);	double y[tf/dt+1]; sys.step(NULL, y, tf);

[2]Only apply to functions pzmap() and rlocus().

[3]z and wn can be intelligently generated.

Table B.1: Syntax comparison of Ch Control Toolkit with MATLAB Control Toolbox (continued).

MATLAB Control Toolbox	**Ch Control Toolkit**
N/A	sys.step(plot, y);
N/A	sys.step(plot, y, tf);
S = stepinfo(sys);	sys.stepinfo(risetime, settlingtime, overshoot, peakvalue, peaktime, undershoot, settlingmin, settlingmax);
S = stepinfo(sys, ’SettlingTimeThreadshold’, ts_percentage);	sys.stepinfo(risetime, settlingtime, overshoot, peakvalue, peaktime, undershoot, settlingmin, settlingmax, ts_percentage, NULL, NULL);
rt[1] = tr_lowerlimit	
rt[2] = tr_upperlimit	
S = stepinfo(sys, ’RiseTimeLimits’, rt);	sys.stepinfo(risetime, settlingtime, overshoot, peakvalue, peaktime, undershoot, settlingmin, settlingmax, NULL, tr_lowerlimit, tr_upperlimit);
sys = ss(A, B, C, D);	sys.model(”ss”, A, B, C, D);
sysT = ss2ss(sys,T);	sysT = sys.ss2ss(T);
[num,den] = ss2tf(A,B,C,D);	sys.model(”ss”, A, B, C, D);
	sys.tfdata(num, den);
[z, p, k] = ss2zp(A,B,C,D);	sys.model(”ss”, A, B, C, D);
	sys.zpkdata(z, p, k);
sys = tf(num, den);	sys.model(”tf”, num, den);
sys = tf(num, den, Ts);	sys.model(”tf”, num, den, Ts);
[num, den] = tfdata(sys);	int nnum, nden;
	nnum = sys.size(’n’);
	nden = sys.size(’d’);
	array double num[nnum], den[nden];
	sys.tfdata(num, den);
[A,B,C,D] = tf2ss(num,den);	sys.model(”tf”, num, den);
	sys.ssdata(A,B,C,D);
[z, p, k] = tf2zp(num,den);	sys.model(”tf”, num, den);
	sys.zpkdata(z, p, k);

[3]Only apply to functions pzmap() and rlocus().

Table B.1: Syntax comparison of Ch Control Toolkit with MATLAB Control Toolbox (continued).

MATLAB Control Toolbox	**Ch Control Toolkit**
z = tzero(sys);	nz = sys.size('z');
	array double complex z[nz];
	sys.tzero(z);
[z,gain] = tzero(sys);	nz = sys.size('z');
	array double complex z[nz];
	double gain;
	sys.tzero(z, &gain);
z = tzero(A,B,C,D);	sys.model("ss", A, B, C, D);
	nz = sys.size('z');
	array double complex z[nz];
	double gain;
	sys.tzero(z, &gain);
zgrid;	sys.zgrid(1)[4];
zgrid('new');	sys.zgrid(1);
zgrid(z, wn);	zgrid(1, z, wn);
zgrid(z, wn, 'new');	zgrid(1, z, wn);
pzmap(H); zgrid	zgrid(1); sys.pzmap();
zgrid('new'); pzmap(H);	sys.zgrid(1); sys.pzmap();
pzmap(H); zgrid(z, wn)	zgrid(1, z, wn); sys.pzmap();
zgrid(z, wn, 'new'); pzmap(H)	zgrid(1, z, wn); sys.pzmap();
[A,B,C,D] = zp2ss(z, p, k);	sys.model("zpk", z, p, k);
	sys.ssdata(A,B,C,D);
[num, den] = zp2tf(z, p, k);	sys.model("zpk", z, p, k);
	sys.tfdata(num, den);
sys = zpk(z, p, k)	sys.model("zpk", z, p, k);
[z,p,k] = zpkdata(sys);	int nz, np; double k;
	nz = sys.size('z'); np = sys.size('p');
	array double complex z[nz], p[np];
	sys.zpkdata(z, p, &k);
N/A	sys.compensatorZeroPoleGain(compsys_dp, comp_zero, comp_pole, comp_gain);
N/A	sys.getDominantPole(percent_overshoot, damping_ratio, dominant_pole, extra_gain, natural_frequency);

Appendix C

Application Examples Using Ch Control System Toolkit

In this Appendix, applications of Ch Control System Toolkit are illustrated by pratical examples. For comparison, all problems in these examples are solved by both Ch programs and MATLAB m files using MATLAB Control System Toolbox.

C.1 Calculating Transfer Function

The system shown in Fig C.1 consists of three subsystems with transfer functions $G_1 = \frac{1}{(s+1)}$, $G_2 = \frac{1}{s}$, $G_3 = 2$. Find the transfer function of this system.

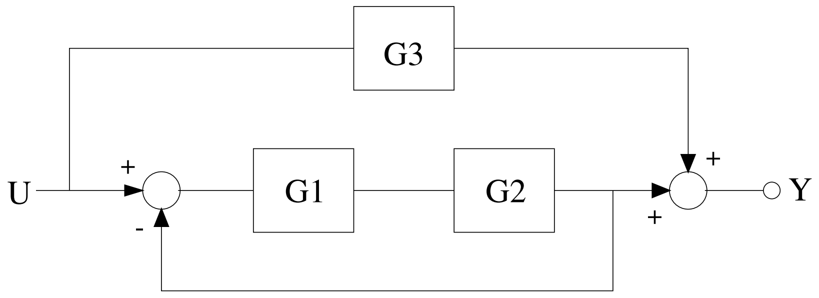


Figure C.1: Block diagram of the system.

Answer
The transfer function of this system is

$$G(s) = \frac{2.00s^2 + 2.00s + 3.00}{1.00s^2 + 1.00s + 1.00}$$

Program in Ch

```
#include <control.h>
int main() {
    /* given G1=1/(s+1), G2=1/s, G3=2 */
    double num1[1] = {1}, den1[2] = {1, 1};
    class CControl sysG1;
    sysG1.model("tf", num1, den1);

    double num2[1] = {1}, den2[2] = {1, 0};
    class CControl sysG2;
    sysG2.model("tf", num2, den2);

    double num3[1] = {2}, den3[1] = {1};
    class CControl sysG3;
    sysG3.model("tf", num3, den3);

    /* series connection for G1 and G2 */
    class CControl *sysG1G2;
    sysG1G2 = sysG1.series(&sysG2);

    /* unit feedback */
    double numf[1] = {1}, denf[1] = {1};
    class CControl sysF;
    sysF.model("tf", numf, denf);

    class CControl *sysG1G2F;
    /* or sysG1G2F = sysG1G2->feedback(&sysF, -1);*/
    sysG1G2F = sysG1G2->feedback(&sysF);

    /* parallel connection */
    class CControl *sysT;
    sysT = sysG1G2F->parallel(&sysG3);
    sysT->printtf();

    return 0;
}
```

Output in Ch

```
Transfer function parameters:
Numerator: 2.000000*s*s+2.000000*s+3.000000
Denominator: 1.000000*s*s+1.000000*s+1.000000
```

Program in MATLAB

```
% given G1=1/(s+1), G2=1/s, G3=2
num1 = 1;
den1 = [1 1];
sysG1 = tf(num1,den1);

num2 = 1;
den2 = [1 0];
sysG2 = tf(num2,den2);

num3 = 2;
den3 = 1;
sysG3 = tf(num3,den3);

% series connection
sysG1G2 = sysG1*sysG2;
```

```
% unit feedback
numf = 1;
denf = 1;
sysF = tf(numf, denf);
% or sysG1G2F = feedback(sysG1G2,sysF,-1);
sysG1G2F = feedback(sysG1G2,sysF);

% parallel connection
[sysT] = parallel(sysG1G2F,sysG3)
```

C.2 Model Conversion from State Space to Transfer Function and Zero-Pole-Gain

Give the system

$$\begin{aligned}\dot{X} &= AX + BU \\ Y &= CX + DU\end{aligned}$$

where

$$A = \begin{bmatrix} -4.12 & -17.4 & -30.8 & -60 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, B = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, C = \begin{bmatrix} 4 & 8.4 & 30.78 & 60 \end{bmatrix}, D = 0$$

find the transfer function, zeros and poles of the system.

Answer
The transfer function of the system is

$$\frac{4s^3 + 8.4s^2 + 30.8s + 60}{s^4 + 4.12s^3 + 17.4s^2 + 30.8s + 60}$$

The zeros, poles and gain of the system are

$$\begin{aligned}zeros &= -0.05 + 2.738156i, -0.05 - 2.738156i, -2.0 \\ poles &= -1.797086 + 2.213723i, -1.797086 - 2.213723i, -0.262914 + 2.703862i, -0.262914 - 2.703862i \\ k &= 4\end{aligned}$$

Program in Ch

```
#include <control.h>

#define NUMX 4    // number of states
#define NUMU 1    // number of inputs
#define NUMY 1    // number of outputs

int main() {
    double A[NUMX][NUMX] = {{-4.12, -17.4, -30.8, -60},
                            {1,     0,     0,     0},
                            {0,     1,     0,     0},
                            {0,     0,     1,     0}},
           B[NUMX][NUMU] = {1, 0, 0, 0},
           C[NUMY][NUMX] = {4, 8.4, 30.78, 60},
           D[NUMY][NUMU] = {0};
    class CControl sys;
    int nnum, nden;

    sys.model("ss", A, B, C, D);
    nnum = sys.size('n');
    nden = sys.size('d');
    array double num[nnum], den[nden];
    printf("\nThe system is created with SS model\n");
    printf("Transfer function parameters obtained by printtf() member function.\n");
    sys.printtf();
    printf("\n\n");
/* get the coefficients of numerator and denominator */
    sys.tfdata(num, den);
    printf("Transfer function parameters obtained by tfdata() member function.\n");
    printf("num = %f\nden = %f\n", num, den);

    int nz, np;
    nz = sys.size('z');
    np = sys.size('p');
    array double complex z[nz], p[np];
    double k;

    printf("ZPK information obtained by printzpk() member function.\n");
    sys.printzpk();
    printf("\n\n");
/* get the zeros and poles of the system */
    sys.zpkdata(z, p, &k);
    printf("ZPK information obtained by zpkdata() member function.\n");
    printf("z = %f\n p = %f\n k = %f\n", z, p, k);

    return 0;
}
```

Output in Ch

```

The system is created with SS model
Transfer function parameters obtained by printtf() member function.
Transfer function parameters:
Numerator: 4.000000*s*s*s+8.400000*s*s+30.780000*s+60.000000
Denominator: 1.000000*s*s*s*s+4.120000*s*s*s+17.400000*s*s+30.800000*s+60.000000


Transfer function parameters obtained by tfdata() member function.
num = 0.000000 4.000000 8.400000 30.780000 60.000000

den = 1.000000 4.120000 17.400000 30.800000 60.000000

ZPK information obtained by printzpk() member function.

Zero-Pole-Gain parameters:

Zero: complex(-0.049557,2.737558) complex(-0.049557,-2.737558) complex(-2.000885,0.000000)


Pole: complex(-1.797086,2.213723) complex(-1.797086,-2.213723) complex(-0.262914,2.703862)
```

```
complex(-0.262914,-2.703862)

Gain: 4.000000


ZPK information obtained by zpkdata() member function.
z = complex(-0.049557,2.737558) complex(-0.049557,-2.737558) complex(-2.000885,0.000000)


 p = complex(-1.797086,2.213723) complex(-1.797086,-2.213723) complex(-0.262914,2.703862)
complex(-0.262914,-2.703862)

 k = 4.000000
```

Program in MATLAB

```
A = [-4.12 -17.4 -30.8 -60; 1 0 0 0; 0 1 0 0; 0 0 1 0];
B = [1; 0; 0; 0];
C = [4 8.4 30.78 60];
D = 0;

sys = ss(A, B, C, D);
[num,den] = tfdata(sys, 'v')
[z,p,k] = zpkdata(sys,'v')
```

C.3 Model Conversion from Zero-Pole-Gain to State Space and Transfer Function

Create the system with given zeros, poles and gain,

$$\begin{aligned} zeros &= -0.05 + 2.738156i, -0.05 - 2.738156i, -2.0 \\ poles &= -1.797086 + 2.213723i, -1.797086 - 2.213723i, -0.262914 + 2.703862i, -0.262914 - 2.703862i \\ k &= 4 \end{aligned}$$

find system matrices and the transfer function of the system.

Answer
The system matrices are

$$A = \left[\begin{array}{llll} -4.12 & -17.4 & -30.8 & -60 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{array}\right], B = \left[\begin{array}{l} 1 \\ 0 \\ 0 \\ 0 \end{array}\right], C = \left[\begin{array}{llll} 4 & 8.4 & 30.78 & 60 \end{array}\right], D = 0$$

The transfer function of the system is

$$\frac{4s^3 + 8.4s^2 + 30.8s + 60}{s^4 + 4.12s^3 + 17.4s^2 + 30.8s + 60}$$

Program in Ch

```
#include <control.h>

int main() {
   array double complex zero[3]={complex(-0.05, 2.738156),
                                 complex(-0.05, -2.738156),
                                 -2.0},
   pole[4]={ complex(-1.797086, 2.213723),
             complex(-1.797086, -2.213723),
             complex(-0.262914, 2.703862),
             complex(-0.262914, -2.703862)};
   double k1=4;
   class CControl sys1;

   sys1.model("zpk", zero, pole, k1);
   printf("\nThe system is created with ZPK model\n");
   printf("System matrices obtained by printss() member function.\n");
   sys1.printss();
   printf("\n\n");

   int nx, ny, nu;
   nx = sys1.size('x');  // obtain the number of states of the system
   ny = sys1.size('y');  // obtain the number of outputs of the system
   nu = sys1.size('u');  // obtain the number of inputs of the system
   array double A1[nx][nx], B1[nx][nu], C1[ny][nx], D1[ny][nu];

   sys1.ssdata(A1, B1, C1, D1);
   printf("System matrices obtained by ssdata() member function.\n");
   printf("A = \n%f\nB = \n%f\nC = \n%f\nD = \n%f\n", A1, B1, C1, D1);

   printf("Transfer function parameters obtained by printtf() member function.\n");
   sys1.printtf();
   printf("\n\n");
   int nnum1, nden1;
   nnum1 = sys1.size('n');
   nden1 = sys1.size('d');
   array double num1[nnum1], den1[nden1];

   sys1.tfdata(num1, den1);
   printf("Transfer function parameters obtained by tfdata() member function.\n");
   printf("num1 = %f\nden1 = %f\n", num1, den1);

   return 0;
}
```

Output in Ch

```
The system is created with ZPK model
System matrices obtained by printss() member function.

State-space arguments:
A =
 -4.120000 -17.399997 -30.799994 -59.999994
  1.000000   0.000000   0.000000   0.000000
  0.000000   1.000000   0.000000   0.000000
  0.000000   0.000000   1.000000   0.000000
B =
  1.000000
  0.000000
  0.000000
  0.000000
C =
  4.000000   8.400000  30.799993  59.999986
D =
  0.000000


System matrices obtained by ssdata() member function.
A =
-4.120000 -17.399997 -30.799994 -59.999994
1.000000 0.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000

B =
1.000000
0.000000
0.000000
0.000000

C =
4.000000 8.400000 30.799993 59.999986

D =
0.000000

Transfer function parameters obtained by printtf() member function.
Transfer function parameters:
Numerator: 4.000000*s*s*s+8.400000*s*s+30.799993*s+59.999986
Denominator: 1.000000*s*s*s*s+4.120000*s*s*s+17.399997*s*s+30.799994*s+59.999994


Transfer function parameters obtained by tfdata() member function.
num1 = 0.000000 4.000000 8.400000 30.799993 59.999986

den1 = 1.000000 4.120000 17.399997 30.799994 59.999994
```

Program in MATLAB

```
z1 = [-0.05+2.738156i -0.05-2.738156i -2.0];
p1 = [-1.797086+2.213723i -1.797086-2.213723i -0.262914+2.703862i -0.262914-2.703862i];
k1 = 4;
sys1 = zpk(z1, p1, k1);
[a1,b1,c1,d1] = ssdata(sys1)
[num1,den1] = tfdata(sys1, 'v')
```

C.4 Model Conversion from Transfer Function to State Space and Zero-Pole-Gain

Create the system with a given transfer function

$$\frac{4s^3 + 8.4s^2 + 30.8s + 60}{s^4 + 4.12s^3 + 17.4s^2 + 30.8s + 60}$$

find the system matrices, zeros, poles and gain of the system.

Answer

The system matrices are

$$A = \begin{bmatrix} -4.12 & -17.4 & -30.8 & -60 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, B = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, C = \begin{bmatrix} 4 & 8.4 & 30.78 & 60 \end{bmatrix}, D = 0$$

The zeros, poles and gain of the system are

$$\begin{aligned} zeros &= -0.05 + 2.738156i, -0.05 - 2.738156i, -2.0 \\ poles &= -1.797086 + 2.213723i, -1.797086 - 2.213723i, -0.262914 + 2.703862i, -0.262914 - 2.703862i \\ k &= 4 \end{aligned}$$

Program in Ch

```
#include <control.h>

int main() {
   array double num2[4] = {4, 8.4, 30.8, 60},
                den2[5] = {1, 4.12, 17.4, 30.8, 60};
   class CControl sys2;

   sys2.model("tf", num2, den2); // create a tf model
   printf("\nThe system is created with TF model\n");
   printf("System matrices obtained by printss() member function.\n");
   sys2.printss();
   printf("\n\n");

   int nx2, ny2, nu2;
   nx2 = sys2.size('x');  // obtain the number of states of the system
   ny2 = sys2.size('y');  // obtain the number of outputs of the system
   nu2 = sys2.size('u');  // obtain the number of inputs of the system
   array double A2[nx2][nx2], B2[nx2][nu2], C2[ny2][nx2], D2[ny2][nu2];

   sys2.ssdata(A2, B2, C2, D2);
   printf("System matrices obtained by ssdata() member function.\n");
   printf("A = \n%f\nB = \n%f\nC = \n%f\nD = \n%f\n", A2, B2, C2, D2);

   printf("ZPK information obtained by printzpk() member function.\n");
   sys2.printzpk();
   printf("\n\n");

   int nz2, np2;
   double k2;
   nz2 = sys2.size('z');
   np2 = sys2.size('p');
   array double complex z2[nz2], p2[np2];
   sys2.zpkdata(z2, p2, &k2);
   printf("ZPK information obtained by zpkdata() member function.\n");
   printf("z2 = %f\n p2 = %f\n k2 = %f\n", z2, p2, k2);

   return 0;
}
```

Output in Ch

```
The system is created with TF model
System matrices obtained by printss() member function.

State-space arguments:
A =
 -4.120000 -17.400000 -30.800000 -60.000000
  1.000000   0.000000   0.000000   0.000000
  0.000000   1.000000   0.000000   0.000000
  0.000000   0.000000   1.000000   0.000000
B =
  1.000000
  0.000000
  0.000000
  0.000000
C =
  4.000000   8.400000  30.800000  60.000000
D =
  0.000000


System matrices obtained by ssdata() member function.
A =
-4.120000 -17.400000 -30.800000 -60.000000
1.000000 0.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000

B =
1.000000
0.000000
0.000000
0.000000

C =
4.000000 8.400000 30.800000 60.000000

D =
0.000000

ZPK information obtained by printzpk() member function.

Zero-Pole-Gain parameters:

Zero: complex(-0.050000,2.738156) complex(-0.050000,-2.738156) complex(-2.000000,0.000000)


Pole: complex(-1.797086,2.213723) complex(-1.797086,-2.213723) complex(-0.262914,2.703862)
complex(-0.262914,-2.703862)

Gain: 4.000000


ZPK information obtained by zpkdata() member function.
z2 = complex(-0.050000,2.738156) complex(-0.050000,-2.738156) complex(-2.000000,0.000000)
```

```
 p2 = complex(-1.797086,2.213723) complex(-1.797086,-2.213723) complex(-0.262914,2.703862)
complex(-0.262914,-2.703862)

 k2 = 4.000000
```

Program in MATLAB

```
num2 = [4 8.4 30.8 60];
den2 = [1 4.12 17.4 30.8 60];
sys2 = tf(num2, den2);
[a2,b2,c2,d2] = ssdata(sys2)
[z2,p2,k2] = zpkdata(sys2,'v')
```

C.5 Time Domain Response Using Transfer Function Model

For the system with the following closed-loop transfer function $T(s) = \frac{2s+4}{s^3+5s^2+6s+4}$.

1. Plot the step response of system.
2. Plot the impulse response of system.

Program in Ch

```
#include <control.h>

int main() {
    class CPlot plotstep, plotimpulse;
    double num[2] = {2, 4};
    double den[4] = {1, 5, 6, 4};
    class CControl sys;

    sys.model("tf", num, den);
    sys.step(&plotstep, NULL, NULL, NULL);
    sys.impulse(&plotimpulse, NULL, NULL, NULL);
    return 0;
}
```

Output in Ch

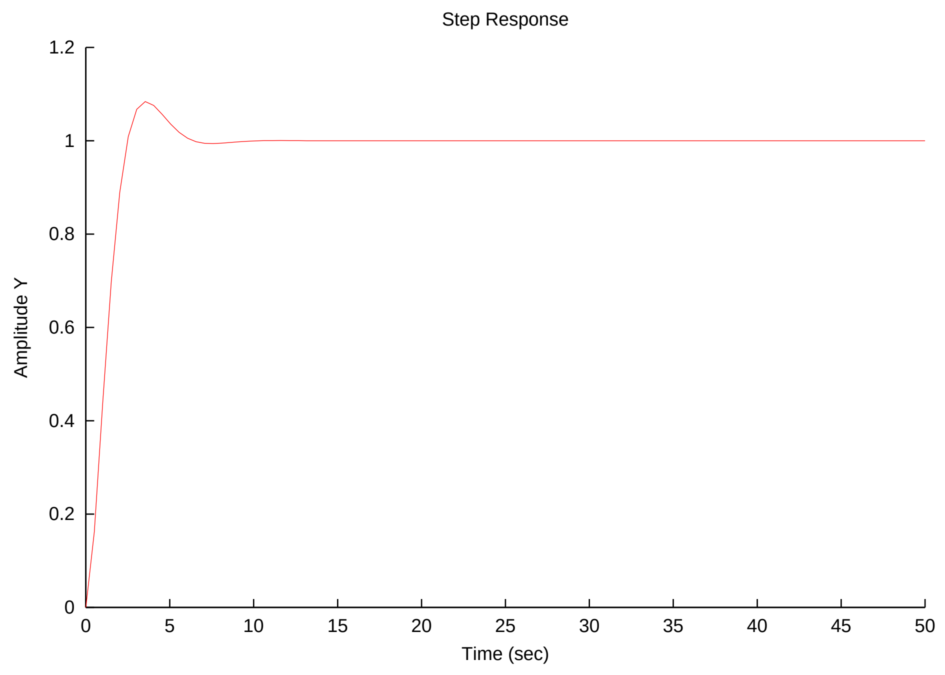


Figure C.2: Step response of the system in Ch.

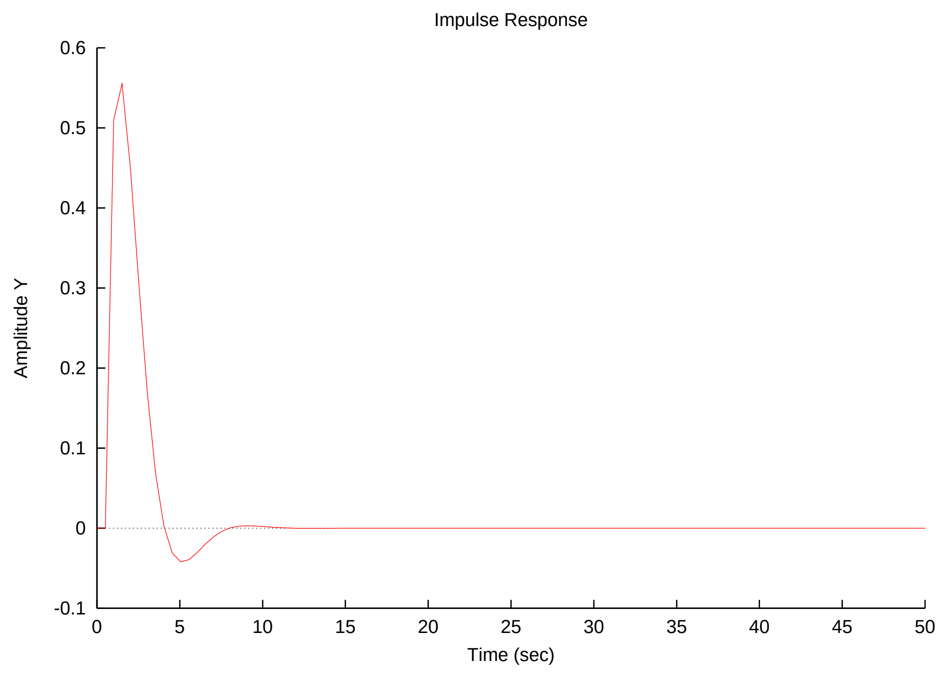


Figure C.3: Impulse response of the system in Ch.

Program in MATLAB

```
num = [2 4];
den = [1 5 6 4];
```

```
sys = tf(num,den);
step(sys);
impulse(sys);
```

C.6 Root Locus of System

Sketch the root locus shown in Fig C.4 with respect to k.

$$G(s)H(s) = \frac{k(s+3)}{s^2+3s+10}$$

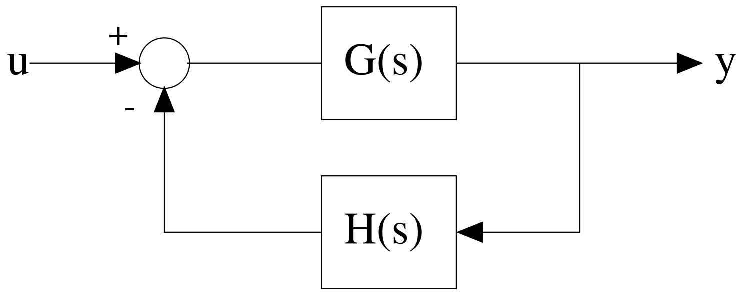


Figure C.4: Block diagram of the system.

Program in Ch

```
#include <control.h>

int main() {
  double num[2] = {1, 3};
  double den[3] = {1, 3, 10};
  class CPlot plot;
  class CControl sys;

  sys.model("tf", num, den);
  sys.sgrid(1);
  sys.rlocus(&plot, NULL, NULL);
  return 0;
}
```

Output in Ch

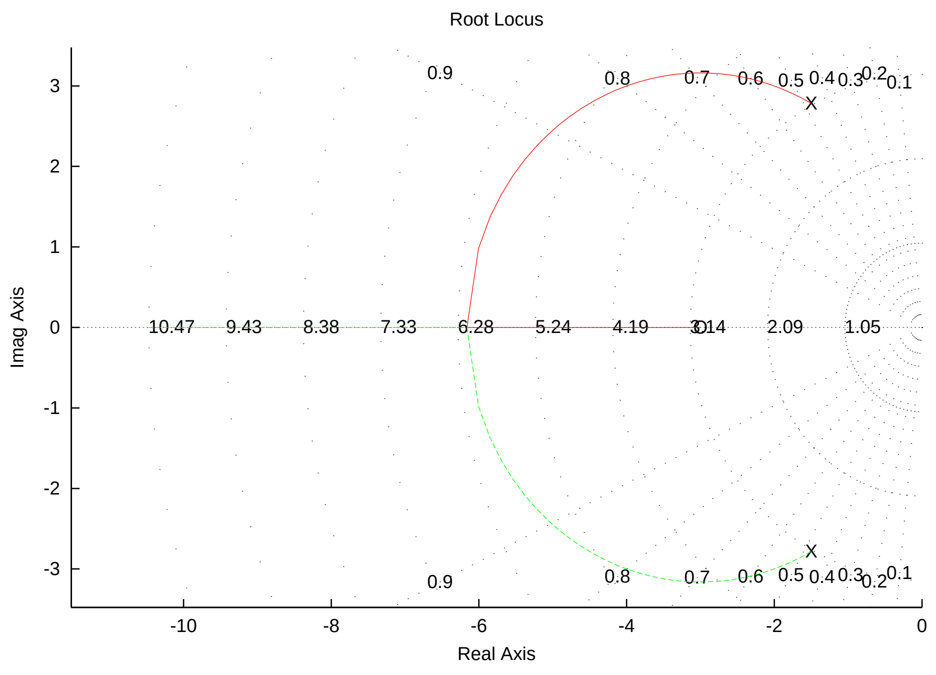


Figure C.5: Root locus of the system in Ch.

Program in MATLAB

```
num = [1 3];
den = [1 3 10];

sys = tf(num,den);
rlocus(sys)
sgrid
```

C.7 Frequency Domain Analysis

For the system with following open-loop transfer function $\dfrac{(s+2)^2}{s^2(s+20)(s^2+6s+25)}$

1. Sketch the Bode plot magnitude and phase,

2. Sketch the Nyquist plot.

Program in Ch

```
#include <control.h>

#define M1 2
#define M2 2
#define M 3 // M = M1+M2-1
#define N1 4
#define N2 3
#define N 6 // N = N1+N2-1
```

```
int main() {
   double num1[M1] = {1, 2};
   double num2[M2] = {1, 2};
   double den1[N1] = {1, 20, 0, 0};  //sˆ2(s+20) = sˆ3+20sˆ2
   double den2[N2] = {1, 6, 25};
   double num[M];
   double den[N];
   class CPlot plotbode, plotnyquist;
   class CControl sys;

   conv(num, num1, num2);
   conv(den, den1, den2);
   sys.model("tf", num, den);
   sys.bode(&plotbode, NULL,  NULL, NULL);
   sys.nyquist(&plotnyquist, NULL, NULL, NULL);

   return 0;
}
```

Output in Ch

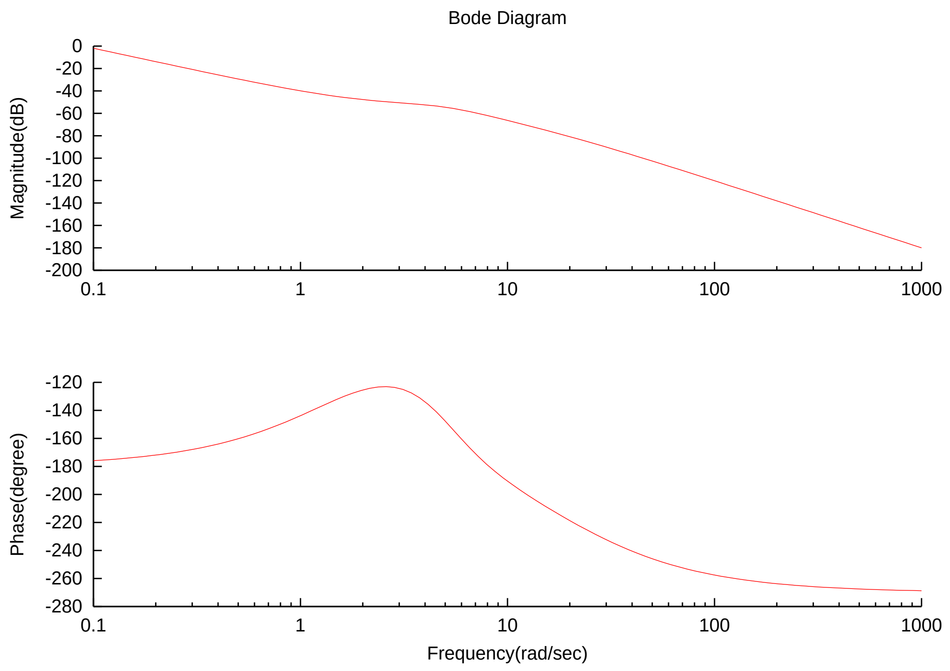


Figure C.6: Bode plot of the system in Ch.

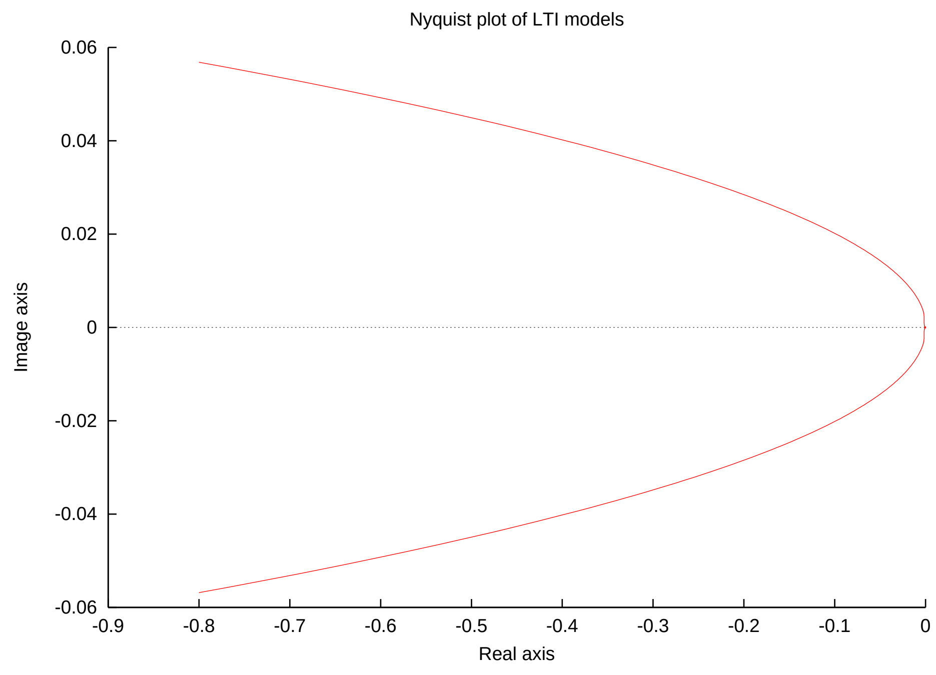


Figure C.7: Nyquist plot of the system in Ch.

Program in MATLAB

```
num1 = [1 2];
num2 = [1 2];
den1 = [1 20 0 0];
den2 = [1 6 25];

num = conv(num1,num2);
den = conv(den1,den2);
sys = tf(num, den);
bode(sys);
nyquist(sys);
```

C.8 Controllability and Observability

The linearized equations of motion for a satellite are

$$\begin{aligned}\dot{X} &= AX + BU \\ Y &= CX + DU\end{aligned}$$

where

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0.0024 \\ 0 & 0 & 0 & 1 \\ 0 & 0.0024 & 0 & 0 \end{bmatrix}, B = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}, C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, D = 0$$

determine the controllability and observability of the system.

Answer

The controllability matrix is

$$co = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0.002400 & -0.000006 & 0 \\ 1 & 0 & 0 & 0.002400 & -0.000006 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -0.002400 & 0 & 0 & -0.000006 \\ 0 & 1 & -0.002400 & 0 & 0 & -0.000006 & 0 & 0 \end{bmatrix}$$

The observability matrix is

$$ob = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0.002400 \\ 0 & -0.002400 & 0 & 0 \\ 0 & -0.000006 & 0 & 0 \\ 0 & 0 & 0 & -0.000006 \end{bmatrix}$$

Program in Ch

```
#include <control.h>

#define NUMX 4        // number of states
#define NUMU 2        // number of inputs
#define NUMY 2        // number of outputs

int main() {
    array double A[NUMX][NUMX] = {{0, 1, 0, 0},
                                  {0, 0, 0, 0.0024},
                                  {0, 0, 0, 1},
                                  {0, -0.0024, 0, 0}};
    array double B[NUMX][NUMU] = {0, 0, 1, 0, 0, 0, 0, 1};
    array double C[NUMY][NUMX] = {1, 0, 0, 0, 0, 0, 1, 0};
    array double co[NUMX][NUMX*NUMU];
    array double ob[NUMX*NUMY][NUMX];
    int num;
    class CControl sys;

    sys.model("ss", A, B, C, NULL);
    sys.ctrb(co);
    num = rank(co);
    if(num == NUMX)
      printf("The system is controllable.\n");
    else
      printf("The system is not controllable.\n");
    printf("co = \n%f\n", co);

    sys.obsv(ob);
    num = rank(ob);
    if(num == NUMX)
      printf("The system is observable.\n");
    else
      printf("The system is not observable.\n");
    printf("ob = \n%f\n", ob);

    return 0;
}
```

Output in Ch

```
The system is controllable.
co =
0.000000 0.000000 1.000000 0.000000 0.000000 0.002400 -0.000006 0.000000
1.000000 0.000000 0.000000 0.002400 -0.000006 0.000000 0.000000 -0.000000
0.000000 0.000000 0.000000 1.000000 -0.002400 0.000000 0.000000 -0.000006
0.000000 1.000000 -0.002400 0.000000 0.000000 -0.000006 0.000000 0.000000

The system is observable.
ob =
1.000000 0.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 0.000000 1.000000
0.000000 0.000000 0.000000 0.002400
0.000000 -0.002400 0.000000 0.000000
0.000000 -0.000006 0.000000 0.000000
0.000000 0.000000 0.000000 -0.000006
```

Program in MATLAB

```
A = [0 1 0 0; 0 0 0 0.0024; 0 0 0 1; 0 -0.0024 0 0];
B = [0 0; 1 0; 0 0; 0 1];
C = [1 0 0 0; 0 0 1 0];

co = ctrb(A, B)
num = rank(co);
if num == 4
  'The system is controllable.'
else
  'The system is not controllable.'
end

ob = obsv(A, C)
num = rank(ob);
if num == 4
  'The system is observable.'
else
  'The system is not observable.'
end
```

C.9 Pole Placement

Consider the system

$$\begin{aligned}\dot{X} &= AX + BU \\ Y &= CX + DU\end{aligned}$$

where

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -0.5 & 0 & 0 & 0 \end{bmatrix}, B = \begin{bmatrix} 0 \\ 1 \\ 0 \\ -1 \end{bmatrix}, C = \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}, D = 0$$

using acker() and place() functions to find the feedback gain K that place the closed-loop poles at s = -1.5, 0, -1+j, -1-j.

Answer

$$K = \begin{bmatrix} 6 & 9.5 & 0 & 6 \end{bmatrix}$$

Program in Ch

```
#include <control.h>

#define NUMX 4        // number of states
#define NUMU 1        // number of inputs
#define NUMY 1        // number of outputs

int main() {
     double A[NUMX][NUMX] = { 0, 1, 0, 0,
                              1, 0, 0, 0,
                              0, 0, 0, 1,
                             -0.5, 0, 0, 0};
     double B[NUMX][NUMU] = { 0, 1, 0, -1};
     array double K[NUMX];
     double complex p[NUMX] = {-1.5, 0, complex(-1,1), complex(-1,-1)};
     class CControl sys0;

     sys0.model("ss", A, B, NULL, NULL);
     sys0.acker(K, p);
     printf("K obtained by acker function = %f\n", K);

     sys0.place(K, p);
     printf("K obtained by place function = %f\n", K);
     return 0;
}
```

Output in Ch

```
K obtained by acker function = 6.000000 9.500000 0.000000 6.000000

K obtained by place function = 6.000000 9.500000 0.000000 6.000000
```

Program in MATLAB

```
A = [0 1 0 0; 1 0 0 0; 0 0 0 1; -0.5 0 0 0];
B = [0; 1; 0; -1];
p = [-1.5; 0; -1+j; -1-j];
K1 = acker(A, B, p)
K2 = place(A, B, p)
```

C.10 Time Domain Response Using State Space Model

For the system in section C.9

1. Find the feedback gain K that places the closed-loop poles at s = -1, -1, -1+j, -1-j.
2. Plot the response of the closed-loop system to an initial condition of x1 = 0.175.

3. Plot the response of the closed-loop system to a unit-step reference input.
4. Plot the impulse response of the closed-loop system.

Program in Ch

```
#include <control.h>

#define NUMX 4        // number of states
#define NUMU 1        // number of inputs
#define NUMY 1        // number of outputs

int main() {
     array double A[NUMX][NUMX] = { 0, 1, 0, 0,
                                    1, 0, 0, 0,
                                    0, 0, 0, 1,
                                    -0.5, 0, 0, 0};
     array double B[NUMX][NUMU] = { 0, 1, 0, -1};
     array double C[NUMY][NUMX] = { 0, 0, 1, 0};
     array double D[1][1] = {0};
     array double K[1][NUMX];
     array double complex p[NUMX] = {-1, -1, complex(-1,1), complex(-1,-1)};
     array double A1[NUMX][NUMX];
     array double x0[NUMX] = {0.175, 0, 0, 0};
     class CPlot plotinitial, plotstep, plotimpulse;
     class CControl sys0;
     class CControl sys;

     sys0.model("ss", A, B, C, D);
     sys0.acker(K, p);
     printf("K = %f\n", K);
     A1 = A - B*K;
     sys.model("ss", A1, B, C, D);

     sys.initial(&plotinitial, NULL, NULL, NULL, x0);
     sys.step(&plotstep, NULL, NULL, NULL);
     sys.impulse(&plotimpulse, NULL, NULL, NULL);

     return 0;
}
```

Output in Ch

```
K = 12.000000 16.000000 4.000000 12.000000
```

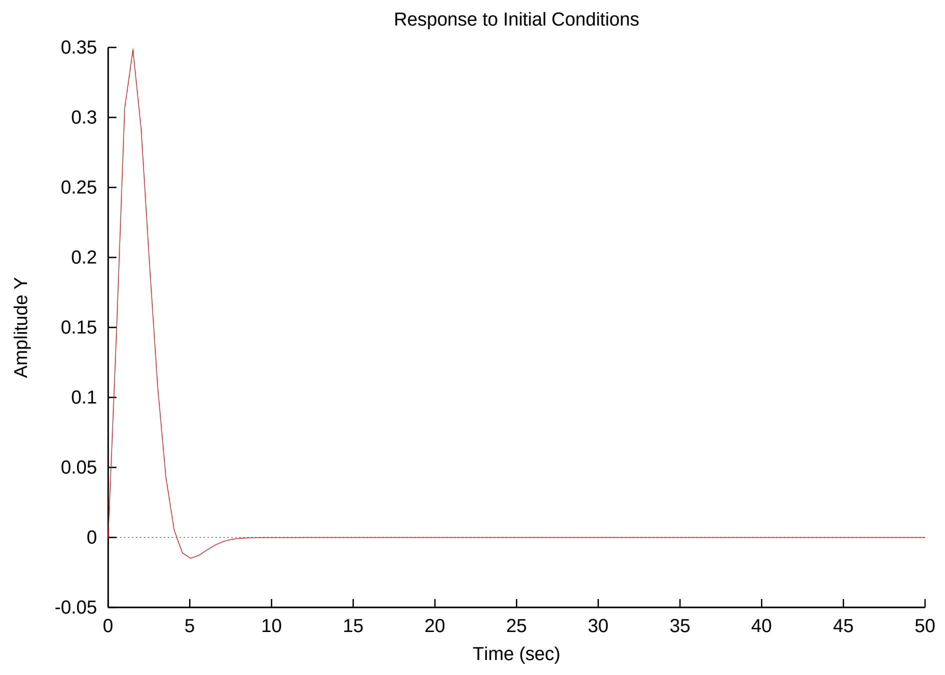


Figure C.8: Initial response of the system in Ch.

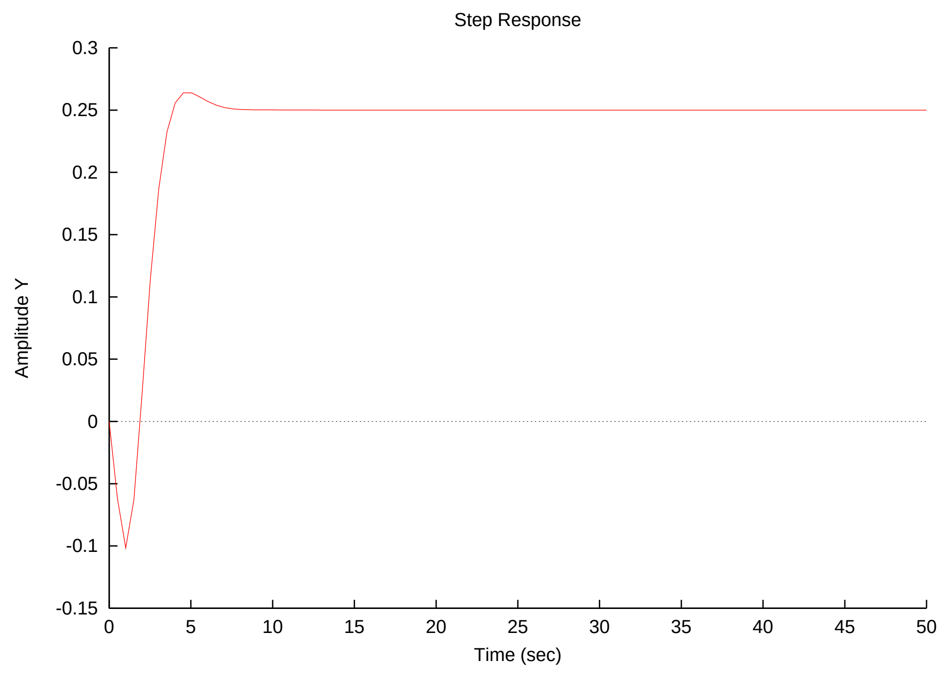


Figure C.9: Step response of the system in Ch.

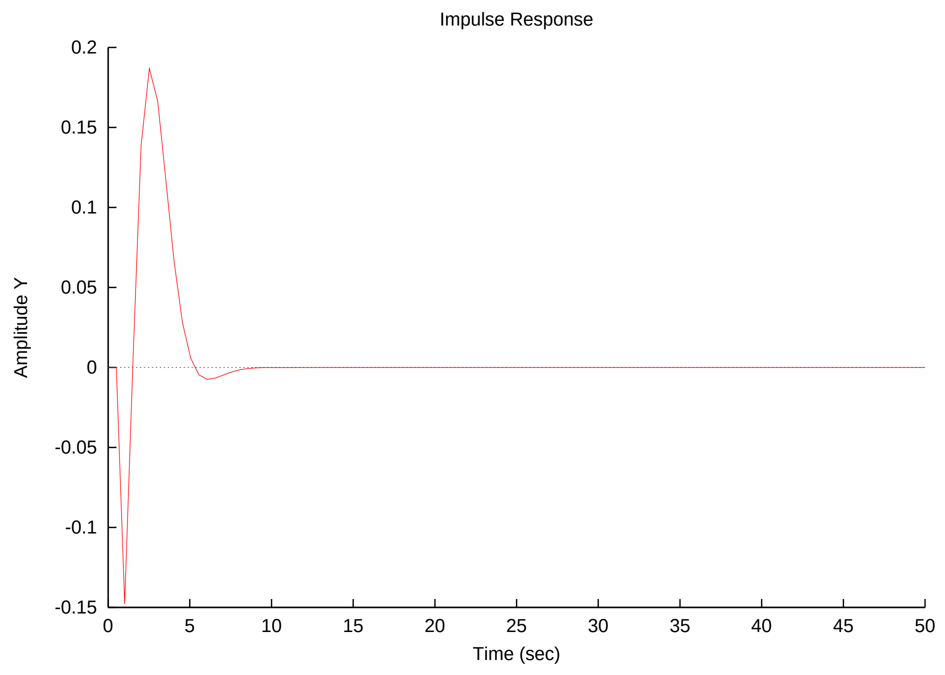


Figure C.10: Impulse response of the system in Ch.

Program in MATLAB

```
A = [0 1 0 0; 1 0 0 0; 0 0 0 1; -0.5 0 0 0];
B = [0; 1; 0; -1];
C = [0 0 1 0];
D = 0;
p = [-1; -1; -1+j; -1-j];
x0 = [0.175 0 0 0];

K = acker(A, B, p)
A1 = A-B*K;

sys = ss(A1,B,C,D);
initial(sys, x0);
step(sys);
impulse(sys);
```

Index