# MATLAB Tutorial Sessions

MATLAB is a formidable mathematics analysis package. An introduction to the most basic commands that are needed for our use is provided. No attempts are made to be comprehensive. We do not want a big, intimidating manual. The beauty of MATLAB is that we need to know only a tiny bit to get going and be productive. Once we get started, we can pick up new skills quickly with MATLAB's excellent on-line help features. We can learn only with hands-on work; these notes are written as a "walk-through" tutorial – you are expected to enter the commands into MATLAB as you read along.

## Session 1. Important Basic Functions

For each session, the most important functions are put in a table at the beginning of the section for easy reference or review. The first one is on the basic commands and plotting. Try the commands as you read. You do not have to enter any text after the "%" sign. Any text that follows "%" is considered a comment and can be ignored. To save some paper, the results generated by MATLAB are omitted. If you need to see that for help, they are provided on the *Web Support*. There is also where any new MATLAB changes and upgrades are posted. Features in our tutorial sessions are based on MATLAB Version 6.1 and Control System Toolbox 5.1.

Important basic functions

| | |
|---|---|
| *General functions:* | |
| `cd` | Change subdirectory |
| `demo (intro)` | Launch the demo (introduction) |
| `dir (what)` | List of files in current directory (or only M-files) |
| `help, helpwin` | Help! Help window |
| `load` | Load workspace |
| `lookfor` | Keyword search |
| `print` | Print graph; can use pull-down menu |
| `quit` | Quit! |
| `save` | Save workspace |
| `who, whos` | List of variables in workspace |

| | |
|---|---|
| *Calculation functions:* | |
| `conv` | Convolution function to multiply polynomials |
| `size, length` | Size of an array, length of a vector |
| *Plotting functions:* | |
| `axis` | Override axis default of plot |
| `grid` | Add grid to plot |
| `hold` | Hold a figure to add more plots (curves) |
| `legend` | Add legend to plot |
| `plot` | Make plots |
| `text (gtext)` | Add text (graphical control) to plot |
| `title` | Add title to plot |
| `xlabel, ylabel` | Add axis labels to plot |

## M1.1. Some Basic MATLAB Commands

The following features are covered in this session:

- using help
- creating vectors, matrices, and polynomials
- simple matrix operations
- multiplying two polynomials with `conv()`

To begin, we can explore MATLAB by using its demonstrations. If you are new to MATLAB, it is highly recommended that you take a look at the introduction:

```
intro             % launch the introduction
demo              % launch the demo program
```

It is important to know that the MATLAB on-line help is excellent, and there are different ways to get that:

```
help              % old-fashioned help inside the Command Window
helpbrowser       % launch the help browser window; also available
                  % from the Help pull-down menu and toolbar
```

We should make a habit of using the on-line help. The user interface of the help browser, which also works as a Web browser, is extremely intuitive, and it is highly recommended. When the `help` command is mentioned, that is just a general comment; it does not mean that you have to use the old-style help. To use help in the Command Window, turn the page display mode on first. Here's an example of seeking help on the print command with the old-style help:

```
more on           % turn the page mode on
help print
lookfor print     % general key-word search
which print       % list the path name of print.m
```

The help features and the Command Window interface tend to evolve quickly. For that reason, the *Web Support* is used to provide additional hints and tidbits so that we can quickly post the latest MATLAB changes. For now, a few more basic commands are introduced:

```
who              % list the variables that are currently defined
whos             % whos is a more detailed version of who
dir              % list the files in the current subdirectory
what             % list only the M-files
cd               % change the subdirectory
pwd              % list the present working directory
```

For fun, we can try

```
why
fix(clock)
```

MATLAB is most at home in dealing with arrays, which we refer to as matrices and vectors. They are all created by enclosing a set of numbers in brackets, [ ]. First, we define a row **vector** by entering, in the MATLAB Command Window,

```
x = [1 2 3 4 5 6 7 8 9 10]
```

If we add a semicolon at the end of a command, as in

```
x = [1 2 3 4 5 6 7 8 9 10];
```

we can suppress the display of the result. We can check what we have later by entering the name of the variable. To generate a column vector, we insert semicolons between numbers (a more specific example is given with a matrix below). The easier route is to take the transpose of $x$:

```
x = x'
```

Keep in mind that in MATLAB variables are case sensitive. Small letter `x` and capital `X` are two different variables.

We can also generate the row vector with the colon operator:

```
x = 1:10         % same as 1:1:10
y = 0:0.1:2      % just another example
```

The colon operator is very useful when we make longer vectors for plotting or calculations. With this syntax, the increment is squeezed between the beginning and the ending values of the vector and they are separated by colons. If the increment value is missing, the default increment is 1. Remember to add a semicolon at the end of each statement to suppress the display of a long string of numbers. This is skipped in the illustration just so you may see what is generated. When we do calculations based on vectors, MATLAB will vectorize the computation, which is much faster than if we write a loop construct as in the FOR loop in C or the DO loop in FORTRAN.

To create a **matrix**, we use a semicolon to separate the rows:

```
a = [1 2 3 ; 4 5 6 ; 7 8 9]
```

In place of the semicolons, we can also simply hit the return key as we generate the matrix.

There are circumstances in which we need the size of an array or the length of a vector. They can be found easily:

```
size(y)        % find the size of an array
length(y)      % find the length of a vector
```

In MATLAB, **polynomials** are stored exactly the same as vectors. Functions in MATLAB will interpret them properly if we follow the convention that a vector stores the coefficients of a polynomial in descending order – it begins with the highest-order term and *always* ends with a constant, even if it is zero. Some examples:

```
p1 =[1 -5 4]    % defines p1(s) = s^2  - 5*s + 4
p2 =[1 0 4]     % defines p2(s) = s^2 + 4
p3 =[1 -5 0]    % defines p3(s) = s^2 - 5*s
```

We can multiply two polynomials together easily with the convolution function `conv()`. For example, to expand $(s^2 - 5s + 4)(s^2 + 4)$, we can use

```
conv(p1,p2)    % this multiplies p1 by p2
```

or

```
conv([1 -5 4], [1 0 4])
```

MATLAB supports every imaginable way that we can manipulate vectors and matrices. We need to know only a few of them, and we will pick up these necessary ones along the way. For now, we'll do a couple of simple operations. With the vector `x` and matrix `a` that we've defined above, we can perform simple operations such as

```
y1 = 2*x        % multiplies x by a constant
y2 = sqrt(x)    % takes the square root of each element in x
b  = sqrt(a)    % takes the square root of each element in a
y3 = y1 + y2    % adds the two vectors
c  = a*b        % multiplies the two matrices
```

Note that all functions in MATLAB, such as `sqrt()`, are smart enough that they accept scalars, vectors, and, where appropriate, matrices.[1]

When we operate on an *element-by-element* basis, we need to add a period before the operator. Examples based on the two square matrices `a` and `b`:

```
d  = a.^3       % takes the cube of each element
a3 = a^3        % versus the cube of the matrix
e  = a.*b       % multiplies each element a(i,j)*b(i,j)
f  = a*b        % versus matrix multiplication a*b
```

Of course, we can solve the matrix equation $\mathbf{Ax} = \mathbf{b}$ easily. For example, we can try

```
A = [ 4 -2 -10; 2 10 -12; -4 -6 16];
b = [-10; 32; -16];
x = A\b        % Bingo!
```

[1] In computer science, this is referred to as polymorphism. The fact that mathematical operators can work on different data types is called overloading.

Let's check the solution by inverting the matrix[2] A:

```
C = inv(A);
x = C*b
```

We can find the eigenvalues and eigenvectors of A easily:

```
[X,D] = eig(A)
```

Finally, we do a simple polynomial fit illustration. Let's say we have a set of $(x, y)$ data:

```
x = [ 0  1  2  4  6  10];
y = [ 1  7  23  109  307  1231];
```

To make a third-order polynomial fit of $y = y(x)$, all we need to enter is

```
c = polyfit(x,y,3)    % should obtain c = [1 2 3 1]
```

The returned vector c contains the coefficients of the polynomial. In this example, the result should be $y = x^3 + 2x^2 + 3x + 1$. We can check and see how good the fit is. In the following statements, we generate a vector xfit so that we can draw a curve. Then we calculate the corresponding yfit values and plot the data with a symbol and the fit as a line:

```
xfit=1:0.5:10;
yfit=xfit.^3 + 2*xfit.^2 + 3*xfit +1;
plot(x,y,'o', xfit,yfit)              % explanation on plotting
title('3rd order polynomial fit')     % is in the next subsection
legend('data','3rd order fit')
```

Speaking of plotting, this is what we get into next.

## M1.2. Some Simple Plotting

The following features are covered in this session:

- Using the plot() function
- Adding titles, labels, and legends

Let's create a few vectors first:

```
x = 0:0.5:10;
y1= 2*x;
y2= sqrt(x);
```

Now we plot y1 versus x and y2 versus x together:

```
plot(x,y1, x,y2)
```

[2] If you have taken a course on numerical methods, you would be pleased to know that MATLAB can do $LU$ decomposition:

```
[L,U] = lu(A);
```

We have a limited selection of line patterns or symbols. For example, we can try[3]

```
plot(x,y1,'-.')
hold              % or use "hold on"
plot(x,y2,'--')
hold              % or use "hold off"
```

We can find the list of pattern selections with on-line help. The command `hold` allows us to add more plots to the same figure, and `hold` works as a toggle. That is why we do not have to state "on" and "off" explicitly.

We can add a title and axis labels too:

```
title('A boring plot')
xlabel('The x-axis label'), ylabel('The y-axis label')
```

We can issue multiple commands on the same line separated by commas. What makes MATLAB easy to learn is that we can add goodies one after another. We do not have to worry about complex command syntax. We can also do logarithmic plots. Try entering `help semilogx, semilogy`, or `loglog`. We'll skip them because they are not crucial for our immediate needs.

We can add a grid and a legend with

```
grid
legend('y1','y2')
```

A box with the figure legend will appear in the Graph Window. Use the mouse to drag the box to where you want it to be. We can also add text to annotate the plot with

```
text(1,9,'My two curves')    % starting at the point (1,9)
```

The text entry can be interactive with the use of

```
gtext('My two curves')
```

Now click on the Graph Window, and a crosshair will appear. Move it to where you want the legend to begin and click. Presto! Repeat for additional annotations.

In rare cases, we may not like the default axis scaling. To override what MATLAB does, we can define our own minimum and maximum of each axis with

```
axis([0 15 0 30])    % the syntax is [xmin  xmax  ymin  ymax]
```

We need the brackets inside because the argument to the axis function is an array.

### Plotting for Fun

We do not need to do three-dimensional (3-D) plots, but then it's too much fun not to do at least a couple of examples. However, we will need to use a few functions that we do not need otherwise, so do not worry about the details of these functions that we will not use

[3] To do multiple plots, we can also use

```
plot(x,y1,'-.', x,y2,'--')
```

again. We will get a pretty 3-D picture:

```
[x,y]=meshgrid(-10:0.5:10,  -10:0.5:10);

% meshgrid transforms the specified domain
% where -10 < x < 10, and -10 < y < 10
% into a grid of (x,y) values for evaluating z

r=sqrt(x.^2 + y.^2) + eps;  % We add the machine epsilon eps
z=sin(r)./r;                % so 1/r won't blow up
mesh(z)
title('The Sinc Sombrero')
```

So you say wow! But MATLAB can do much more and fancier than that. We try one more example with Bessel functions, which you can come across in heat and mass transfer problems with cylindrical geometry:

```
% Here we do a 3-D mesh plot of Jo(sqrt(x^2+y^2))
% The x and y grids remain the same as in the previous plot

r=sqrt(x.^2+y.^2);
z=bessel(0,r);
mesh(z)
```

## M1.3. Making M-files and Saving the Workspace

The following features are covered in this session:

- Executing repeated commands in a script, the so-called M-file
- Saving a session

For tasks that we have to repeat again and again, it makes sense to save them in some kind of a script and execute them. In MATLAB, these scripts are called M-files. The name came from the use of macros in the old days of computing. We can use M-files to write unstructured scripts or user-defined functions. MATLAB now refers to both as programs. You may want to keep in mind that a scripting interpretive language is not the same as a compiled language like C.

For our needs, a simple script suffices in most circumstances. To use an M-file[4]

(1) save all the repetitious MATLAB statements in a text file with the `".m"` extension,
(2) execute the statements in that file by entering the file name *without* the `".m"` extension.

[4] There is another easy way to "cheat." On UNIX/Linux workstations, open up a new text editor and enter your frequently used statements there. In Windows, you can use the really nice MATLAB Editor. You can copy and paste multiple commands back and forth between the text editor window and the MATLAB window easily. If you want to save the commands, you certainly can add comments and annotations. You can consider this text file as a "free-format notebook" without having to launch the Microsoft Word Notebook for MATLAB.

Here is one simple example. We need to plot x versus y repeatedly and want to automate the task of generating the plots. The necessary statements are

```
% ———————————— M-file script: plotxy.m ————————————
% A very simple script to plot x vs y and add the labels
% ...the kind of things we don't want to repeat typing
%   again and again...

plot(x,y)
grid
xlabel('Time [min]')
ylabel('Step Response')
title('PID Controller Simulation')

% End of plotxy.m. An "end" statement is not needed.
```

Save these statements in a file named, say, `plotxy.m`. Anything after the "`%`" sign is regarded as a comment, which you do not have to enter if you just want to repeat this exercise. After we have defined or updated the values of `x` and `y` in the Command Window, all we need is to enter `"plotxy"` at the prompt and MATLAB will do the rest. The key is to note that the M-file has *no* "read" or "input" for `x` and `y`. All statements in an M-file are simply executed in the Command Window.

If you have an M-file, MATLAB may not find it unless it is located within its search path. Let's see where MATLAB looks first. On UNIX/Linux machines, MATLAB by default looks in the subdirectory from which it is launched. A good habit is to keep all your work in one subdirectory and change to that specific subdirectory before you launch MATLAB. On Windows machines, MATLAB looks for your files in the Work folder buried deep inside the Program Files folder. A good chance is that you want to put your files in more convenient locations. To coax MATLAB to find them, you need to change the directory or the search path. So the next question is how to do that, and the answer applies to both UNIX/Linux and Windows machines. The formal way is to learn to use the `"cd"` and `"path"` commands. The easy way is to use point-and-click features that can be found under pull-down menus, on toolbars, or in subwindows. Because these graphical interface features tend to change with each MATLAB upgrade, please refer to the *Web Support*, from which you can find updates of new changes and additional help.

If we want to take a coffee break and save all the current variables that we are working with, enter

```
save
```

before we `quit` MATLAB. When we launch MATLAB again, we type

```
load
```

and everything will be restored. Do not save the workspace if you are going to be away any longer because the old workspace is not very useful if you have all these variables floating around and you forget what they mean.

As a final comment, we can use `load` and `save` to import and export arrays of data. Because we do not really need this feature in what we do here, this explanation is deferred to the *Web Support*.

## Session 2. Partial-Fraction and Transfer Functions

This tutorial is to complement our development in Chap. 2. You may want to go over the tutorial quickly before you read the text and come back later a second time for the details.

Partial-fraction and transfer functions

| | |
|---|---|
| `poly` | Construct a polynomial from its roots |
| `residue` | Partial-fraction expansion |
| `roots` | Find the roots to a polynomial |
| `tf2zp` | Transfer function to zero-pole form conversion |
| `zp2tf` | Zero-pole form to transfer function conversion |
| *Object-oriented functions:* | |
| `tf` | Create a transfer function object |
| `get` | List the object properties |
| `pole` | Find the poles of a transfer function |
| `zpk` | Create a transfer function in pole-zero-gain form |

### M2.1. Partial Fractions

The following features are covered in this session:

- Finding the roots of a polynomial with `roots()`
- Generating a polynomial from its roots with `poly()`
- Doing partial fractions with `residue()`

Of secondary importance:

- Transfer function to zero-pole form, `tf2zp()`
- Zero-pole form to transfer function, `zp2tf()`

Let's first define a polynomial:

```
p = [1 5 4]      % makes p(s) = s^2 + 5*s + 4
```

We can find the roots of $p(s) = 0$ with the function `roots()`:

```
poles = roots(p)
```

MATLAB should return $-4$ and $-1$. That means the polynomial can be factored as $p(s) = (s + 4)(s + 1)$.[5]

[5] MATLAB has the function `fzero()` to find a root of a given function.

We can go backwards. Given the roots (or pole positions), we can get the polynomial with

```
p2 = poly(poles)
```

MATLAB returns the results in a column vector. Most functions in MATLAB take either row or column vectors, and we usually do not have to worry about transposing them.

We can do partial fractions with the `residue()` function. Say we have a transfer function

$$G(s) = \frac{q(s)}{p(s)} = \frac{1}{s^2 + 5s + 4},$$

where $q(s) = 1$ and $p(s)$ remains `[1 5 4]`, as previously defined. We can enter

```
q = 1;
residue(q,p)
```

MATLAB returns the numbers $-0.3333$ and $0.3333$. That is because the function can be factored as

$$\frac{1}{s^2 + 5s + 4} = \frac{-1/3}{s + 4} + \frac{1/3}{s + 1}.$$

How can we be sure that it is the $-0.3333$ coefficient that goes with the root at $-4$? We can use the syntax

```
[a,b,k]=residue(q,p)
```

MATLAB will return the coefficients in `a`, the corresponding poles in `b`, and whatever is left over in `k`, which should be nothing in this case. Note that `[]` denotes an empty matrix or vector.

Let's try another transfer function with poles at 0, $-1$, $-2$, and $-3$:

$$G(s) = \frac{1}{s(s+1)(s+2)(s+3)}.$$

To find the partial fractions, this is what we can do[6]:

```
poles=[0 -1 -2 -3];
p=poly(poles);
q=1;
[a,b,k]=residue(q,p)
```

One more example. Find the partial fractions of the nasty-looking function

$$G(s) = \frac{s^2 + 4s + 3}{s^4 - 7s^3 + 11s^2 + 7s - 12}.$$

[6] If we need to write complex-conjugate roots, make sure there are no spaces within a complex number. For example, enter `[-3+4*j   -3-4*j]`. Either `i` or `j` can be used to denote $\sqrt{-1}$.

```
q=[1  4  3];
zeros=roots (q)        % should return -3, -1
p=[1  -7  11  7  -12];
poles=roots (p)        % should return 4, 3, 1, -1
[a,b,k]=residue (q,p)
```

See that MATLAB returns the expansion:

$$\frac{s^2+4s+3}{s^4-7s^3+11s^2+7s-12}=\frac{2.33}{s-4}-\frac{3}{s-3}+\frac{0.67}{s-1}.$$

Note that the coefficient associated with the pole at $-1$ is zero. That is because it is canceled by the zero at $-1$. In other words, the $(s+1)$ terms cancel out. It is nice to know that the program can do this all by itself. We do not need to know the roots to use `residue()`, but it is a good habit to get a better idea of what we are working with.

A transfer function can be written in terms of its poles and zeros. For example,

$$F(s)=\frac{6s^2-12}{(s^3+s^2-4s-4)}=\frac{6(s-\sqrt{2})(s+\sqrt{2})}{(s+1)(s+2)(s-2)}.$$

The RHS is called the pole-zero form (or zero-pole form). MATLAB provides two functions, `tf2zp()` and `zp2tf()`, to do the conversion. For instance,

```
q=[6  0  -12];
p=[1  1  -4  -4];
[zeros, poles, k]=tf2zp(q,p)
```

Of course, we can go backward with

```
[q,p]=zp2tf(zeros,poles,k)
```

*Note*: The factor `k` is 6 here, and in the MATLAB manual it is referred to as the "gain." This factor is really the ratio of the leading coefficients of the two polynomials $q(s)$ and $p(s)$. Make sure you understand that the `k` here is *not* the steady-state gain, which is the ratio of the *last* constant coefficients. (In this example, the steady-state gain is $-12/-4=3$.) MATLAB actually has a function called `dcgain` to do this.

One more simple example:

```
zero= -2;              % generate a transfer function
poles=[-4  -3  -1];    % with given poles and zeros
k=1;
[q,p]=zp2tf(zero,poles,k)
```

Double check that we can recover the poles and zeros with

```
[zero,poles,k]=tf2zp(q,p)
```

We can also check with

```
roots(q)
roots(p)
```

Try `zp2tf` or `tf2zp` on your car's license plate!

## M2.2. Object-Oriented Transfer Functions

The following features are covered in this session:

- Defining a transfer function object with `tf()` or `zpk()`
- Determining the poles with `pole()`
- Using overloaded operators

MATLAB is object oriented. Linear-time-invariant (LTI) models are handled as objects. Functions use these objects as arguments. In classical control, LTI objects include transfer functions in polynomial form or in pole-zero form. The LTI-oriented syntax allows us to better organize our problem solving; we no longer have to work with individual polynomials that we can identify only as numerators and denominators.

We will use this syntax extensively starting in Session 3. Here, we see how the object-oriented syntax can make the functions `tf2zp()` and `zp2tf()` redundant and obsolete.

To define a transfer function object, we use `tf()`, which takes the numerator and denominator polynomials as arguments. For example, we define $G(s) = [s/(s^2 - 5s + 4)]$ with

```
G1 = tf([1 0], [1 -5 4])
```

We define $G(s) = [(6s^2 - 12)/(s^3 + s^2 - 4s - 4)]$ with

```
G2 = tf([6 0 -12], [1 1 -4 -4])
```

We can also use the zero-pole-gain function `zpk()` which takes as arguments the zeros, poles, and gain factor of a transfer function. Recall the comments after `zp2tf()`. This gain factor is not the steady-state (or dc) gain.

For example, we define $G(s) = \{4/[s(s+1)(s+2)(s+3)]\}$ with

```
G3 = zpk([],[0 -1 -2 -3], 4)   % the [] means there is no zero
```

The `tf()` and `zpk()` functions also serve to perform model conversion from one form to another. We can find the polynomial form of `G3` with

```
tf(G3)
```

and the pole-zero form of `G2` with

```
zpk(G2)
```

The function `pole()` finds the poles of a transfer function. For example, try

```
pole(G1)
pole(G2)
```

You can check that the results are identical to the use of `roots()` on the denominator of a transfer function.

We may not need to use them, but it is good to know that there are functions that help us extract the polynomials or poles and zeros back from an object. For example,

```
[q,p]=tfdata(G1,'v')        % option 'v' for row vectors
[z,p,k]=zpkdata(G3,'v')
```

The addition and multiplication operators are overloaded, and we can use them to manipulate or synthesize transfer functions. This capability will come in handy when we analyze control systems. For now, let's consider one simple example. Say we are given

$$G_1 = \frac{1}{s+1}, \qquad G_2 = \frac{2}{s+2}.$$

We can find $G_1 + G_2$ and $G_1 G_2$ easily with

```
G1=tf(1,[1 1]);
G2=tf(2,[1 2]);
G1+G2           % or we can use zpk(G1+G2)
G1*G2           % or we can use zpk(G1*G2)
```

This example is simple enough to see that the answers returned by MATLAB are correct.

With object-oriented programming, an object can hold many properties. We find the associated properties with

```
get(G1)
```

Among the MATLAB result entries, we may find the properties `InputName`, `OutputName`, and `Notes`. We can set them with [7]

```
G1.InputName = 'Flow Rate';
G1.OutputName = 'Level';
G1.Notes = 'My first MATLAB function';
```

You will see the difference if you enter, from now on,

```
G1
get (G1)
```

MATLAB can use symbolic algebra to do the Laplace transform. Because this skill is not crucial to solving control problems, we skip it here. You can find a brief tutorial on the *Web Support*, and you are encouraged to work through it if you want to know what symbolic algebra means.

## Session 3. Time-Response Simulation

This tutorial is to complement our development in Chap. 3. You may want to go over the tutorial quickly before you read the text and come back later a second time for the details.

[7] We are using the typical structure syntax, but MATLAB also supports the `set()` function to perform the same task.

Time-response simulation functions

| | |
|---|---|
| `damp` | Find damping factor and natural frequency |
| `impulse` | Impulse response |
| `lsim` | Response to arbitrary inputs |
| `step` | Unit-step response |
| `pade` | Time-delay Padé approximation |
| `ltiview` | Launch the graphics viewer for LTI objects |

## M3.1. Step- and Impulse-Response Simulations

The following features are covered in this session:

- Using `step()` and `impulse()`
- Time response to any given input, `lsim()`
- Dead-time approximation, `pade()`

Instead of spacing out in the Laplace domain, we can (as we are taught) guess how the process behaves from the pole positions of the transfer function. However, wouldn't it be nice if we could actually trace the time profile without having to do the reverse Laplace transform ourselves? Especially the response with respect to step and impulse inputs? Plots of time-domain dynamic calculations are extremely instructive and a useful learning tool.[8]

The task of time-domain calculation is easy with MATLAB. Let's say we have

$$\frac{Y(s)}{X(s)} = \frac{1}{s^2 + 0.4s + 1},$$

and we want to plot $y(t)$ for a given input $x(t)$. We can easily do

```
q=1;
p=[1 0.4 1];  % poles at -0.2±0.98j
G=tf(q,p)

step(G)       % plots y(t) for unit step input, X(s)=1/s

impulse(G)    % plots y(t) for impulse input, X(s)=1
```

What a piece of cake! Not only does MATLAB perform the calculation, but it automatically makes the plot with a properly chosen time axis. Nice! [9] As a habit, find out more about a function with `help` as in

```
help step     % better yet, use helpwin or helpbrowser
```

[8] If you are interested, see the *Web Support* for using the `Runge—Kutta` integration of differential equations.

[9] How could we guess what the time axis should be? It is not that difficult if we understand how to identify the dominant pole, the significance behind doing partial fractions, and that the time to reach 99% of the final time response is approximately five time constants.

The functions also handle multiple transfer functions. Let's make a second transfer function in pole-zero form:

$$H(s) = \frac{2}{(s+2)(s^2+2s+2)};$$

```
H=zpk([], [-2 -1+j -1-j], 2)
```

We can compare the unit-step responses of the two transfer functions with

```
step(G,H)
```

We can, of course, choose our own axis, or rather, time vector. Putting both the unit-step and impulse-response plots together may also help us understand their differences:

```
t=0:0.5:40;  % don't forget the semicolon!
ys=step(G,t);
yi=impulse(G,t);
plot(t,ys,t,yi)
```

*Note*: In the text, the importance of relating pole positions of a transfer function to the actual time-domain response was emphasized. We should get into the habit of finding what the poles are. The time-response plots are teaching tools that reaffirm our confidence in doing analysis in the Laplace domain. Therefore we should find the roots of the denominator. We can also use the `damp()` function to find the damping ratio and the natural frequency.

```
pole(G) % same result with roots(p)
damp(G) % same result with damp(p)
```

One more example. Consider the transfer function

$$\frac{Y(s)}{X(s)} = G(s) = \frac{2s+1}{(4s+1)(s+1)}.$$

We want to plot $y(t)$ if we have a sinusoidal input $x(t) = \sin(t)$. Here we need the function `lsim()`, a general simulation function that takes any given input vector:

```
q=[2 1];                    % a zero at -1/2
p=conv([4 1],[1 1]);        % poles at -1/4 and -1
G=tf(q,p)                   % (can use zpk instead)
t=0:0.5:30;
u=sin(t);
y=lsim(G,u,t);              % response to a sine function input
plot(t,y,t,u,'-.'), grid
```

Keep this exercise in mind. This result is very useful in understanding what is called frequency response in Chap. 8. We can repeat the simulation with higher frequencies. We can also add what we are familiar with:

```
hold
ys=step(G,t);
yi=impulse(G,t);
```

```
plot(t,ys,t,yi)
hold off
```

For fun, try one more calculation with the addition of random noise:

```
u=sin(t)+rand(size(t));
y=lsim(G,u,t);
plot(t,y,'r',t,u,'b'), grid   % Color lines red and blue
```

For useful applications, `lsim()` is what we need to simulate a response to, say, a **rectangular pulse**. This is one simple example that uses the same transfer function and time vector that we have just defined:

```
t=0:0.5:30;                   % t = [0 .5 1 1.5 2 2.5 3 … ]
u=zeros(size(t));             % make a vector with zeros
u(3:7)=1;                     % make a rectangular pulse from t=1
                              % to t=3
y=lsim(G,u,t);
yi=impulse(G,t);              % compare the result with impulse
                              % response
plot(t,u, t,y, t,yi,'-.');
```

Now we switch gears and look into the dead-time transfer function approximation. To do a Padé approximation, we can use the MATLAB function [10]

```
[q,p]=pade(Td,n)
```

where `Td` is the dead time, `n` is the order of the approximation, and the results are returned in `q(s)/p(s)`. For example, with `Td` $= 0.2$ and `n` $= 1$, entering

```
[q,p]=pade(0.2,1)             % first-order approximation
```

will return

```
q = -1 s + 10
p =  1 s + 10
```

We expected $q(s) = -0.1s + 1$ and $p(s) = 0.1s + 1$. Obviously MATLAB normalizes the polynomials with the leading coefficients. On second thought, the Padé approximation is so simple that there is no reason why we cannot do it ourselves as in a textbook. For the first-order approximation, we have

```
Td=0.2;
q = [-Td/2  1];
p = [ Td/2  1];
```

[10] When we use `pade()` without the left-hand argument `[q,p]`, the function automatically plots the step and the phase responses and compares them with the exact responses of the time delay. A Padé approximation has unit gain at all frequencies. These points will not make sense until we get to frequency-response analysis in Chap. 8. For now, keep the `[q,p]` on the LHS of the command.

We can write our own simple-minded M-file to do the approximation. You may now try

```
[q,p]=pade(0.2,2)     % second-order approximation
```

and compare the results of this second-order approximation with the textbook formula.

## M3.2. LTI Viewer

The following feature is covered in this session:

- Graphics viewer for LTI objects, `ltiview`[11]

We can use the LTI Viewer to do all the plots, not only step and impulse responses, but also more general time-response and frequency-response plots in later chapters. If we know how to execute individual plot statements, it is arguable whether we really need the LTI Viewer. Nonetheless, that would be a personal choice. Here the basic idea and some simple instructions are provided.

To launch the LTI Viewer, enter in the MATLAB Command Window

```
ltiview
```

A blank LTI window will pop up. The first task would be to poke into features supported under the File and Tools pull-down menus and see what we can achieve by point and click. There is also a Help pull-down menu, which activates the Help Window.

The LTI Viewer runs in its own workspace, which is separate from the MATLAB workspace. The Viewer also works with only LTI objects generated by functions such as `tf()` and `zpk()`, and after Chap. 4, state-space objects, `ss()`. So let's generate a couple of objects in the MATLAB Command Window first:

```
G=tf(1,[1 0.4 1])
H=zpk([], [-2 -1+j -1-j], 2)
```

Now, go to the LTI Viewer window and select **Import** under the File pull-down menu. A dialog box will pop out to help import the transfer function objects. By default, a unit-step response will be generated. Click on the axis with the *right mouse button* to retrieve a pop-up menu that will provide options for other plot types, for toggling the object to be plotted, and for other features. With a step-response plot, the Characteristics feature of the pop-up menu can identify the peak time, rise time, and settling time of an underdamped response.

The LTI Viewer was designed to do comparative plots, either comparing different transfer functions or comparing the time-domain and (later in Chap. 8) frequency-response properties of a transfer function. Therefore a more likely (and quicker) case is to enter, for example,

```
ltiview('step',G,H)
```

The transfer functions `G` and `H` will be imported automatically when the LTI Viewer is launched, and the unit-step response plots of the two functions will be generated.

[11] The description is based on Version 5.1 of the MATLAB control toolbox. If changes are introduced in newer versions, they will be presented on the *Web Support*.

Another useful case is, for example,

```
ltiview({'step';'bode'},G)
```

In this case, the LTI Viewer will display both the unit-step response plot and the Bode plot for the transfer function `G`. We will learn about Bode plots in Chap. 8, so don't panic yet. Just keep this possibility in mind until we get there.

## Session 4. State-Space Functions

This tutorial is to complement our development in Chap. 4. You may want to go over the tutorial quickly before you read the text and come back later a second time for the details.

State-space functions

| | |
|---|---|
| `canon` | Canonical state-space realization |
| `eig` | Eigenvalues and eigenvectors |
| `ss2ss` | Transformation of state-space systems |
| `ss2tf` | Conversion from state-space to transfer function |
| `tf2ss` | Conversion from transfer function to state-space |
| `printsys` | Slightly prettier looking display of model equations |
| `ltiview` | Launch the graphics viewer for LTI objects |
| `ss` | Create state-space object |

### M4.1. Conversion between Transfer Function and State-Space

The following features are covered in this session:

- Using `ss2tf()` and `tf2ss()`
- Generating object-oriented models with `ss()`

We need to revisit Example 4.1 with a numerical calculation. Let's use the values $\zeta = 0.5$ and $\omega_n = 1.5$ Hz to establish the transfer function and find the poles:

```
z=0.5;
wn=1.5;               % Should find
q=wn*wn;              % q=2.25
p=[1 2*z*wn  wn*wn]   % p=[1  1.5  2.25]
roots(p)              % -0.75 ± 1.3j
```

From the results in Example 4.1, we expect to find

$$\mathbf{A} = \begin{bmatrix} 0 & 1 \\ -2.25 & -1.5 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 0 \\ 2.25 \end{bmatrix}, \quad \mathbf{C} = [1 \quad 0], \quad \mathbf{D} = 0.$$

Now let's try our hands with MATLAB by using its transfer function to state-space conversion function:

```
[a,b,c,d]=tf2ss(q,p)
```

MATLAB returns with

$$\mathbf{a} = \begin{bmatrix} -1.5 & -2.25 \\ 1 & 0 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \mathbf{c} = [\,0 \quad 2.25], \quad \mathbf{d} = 0,$$

which are not the same as those in Example 4.1. You wonder what's going on? Before you kick the computer, a closer look should reveal that MATLAB probably uses a slightly different convention. Indeed, MATLAB first "split" the transfer function into product form:

$$\frac{Y}{U} = \frac{X_2}{U}\frac{Y}{X_2} = \frac{1}{\left(s^2 + 2\zeta\omega_n s + \omega_n^2\right)}\omega_n^2 = \frac{1}{(s^2 + 1.5s + 2.25)}2.25.$$

From $X_2/U = 1/(s^2 + 2\zeta\omega_n s + \omega_n^2)$ and with the state variables defined as

$$x_1 = \frac{\mathrm{d}x_2}{\mathrm{d}t}, \qquad x_2 = x_2 \text{ (i.e., same)},$$

we should obtain the matrices **a** and **b** that MATLAB returns. From $Y/X_2 = \omega_n^2$, it should be immediately obvious how MATLAB obtains the array **c**.

In addition, we should be aware that the indexing of state variables in MATLAB is in *reverse order* of textbook examples. Despite these differences, the inherent properties of the model remain identical. The most important of all is to check the eigenvalues:

```
eig(a)    % should be identical to the poles
```

A conversion from state-space back to a transfer function should recover the transfer function:

```
[q2,p2]=ss2tf(a,b,c,d,1)     % same as q/p as defined earlier
```

The last argument in `ss2tf()` denotes the *i*th input, which must be 1 for our SISO model. To make sure we cover all bases, we can set up our own state-space model as in Example 4.1,

```
a=[0 1; -2.25 -1.5]; b=[0; 2.25]; c=[1 0]; d=0;
```

and check the results with

```
eig(a)                       % still the same!
[qs,ps]=ss2tf(a,b,c,d,1)
```

The important message is that there is no unique state-space representation, but all model matrices should have the same eigenvalues. In addition, the number of state variables is the same as the order of the process or system.

The fact that the algorithm used by MATLAB does not return a normalized output matrix **C** can create problems when we do feedback calculations in Chap. 9. The easy solution is to rescale the model equations. The output equation can be written as

$$y = [\alpha \quad 0]\mathbf{x} = [1 \quad 0]\bar{\mathbf{x}},$$

where $\bar{\mathbf{x}} = \alpha\mathbf{x}$. Substitution for $\mathbf{x}$ by $\bar{\mathbf{x}}$ in $\mathrm{d}\mathbf{x}/\mathrm{d}t = \mathbf{A}\mathbf{x} + \mathbf{B}u$ will lead to

$$\frac{\mathrm{d}\bar{\mathbf{x}}}{\mathrm{d}t} = \mathbf{A}\bar{\mathbf{x}} + \alpha\mathbf{B}u = \mathbf{A}\bar{\mathbf{x}} + \bar{\mathbf{B}}u,$$

where $\bar{\mathbf{B}} = \alpha\mathbf{B}$. In other words, we just need to change $\mathbf{C}$ to the normalized vector and multiply $\mathbf{B}$ by the scaling factor. We can see that this is correct from the numerical results of Example 4.1. (Again, keep in mind that the MATLAB indexing is in reverse order of textbook examples.) We will use this idea in Chap. 9.

We now repeat the same exercise to show how we can create object-oriented state-space LTI models. In later chapters, all control toolbox functions take these objects as arguments. We first repeat the statements above to regenerate the state matrices `a`, `b`, `c`, and `d`. Then we use `ss()` to generate the equivalent LTI object.

```
q=2.25;
p=[1 1.5 2.25];
[a,b,c,d]=tf2ss(q,p);
sys_obj=ss(a,b,c,d)
```

We should see that the LTI object is identical to the state-space model. We can retrieve and operate on individual properties of an object. For example, to find the eigenvalues of the matrix `a` inside `sys_obj`, we use

```
eig(sys_obj.a)     % find eigenvalue of state matrix a
```

We can obtain the transfer function, as analogous to using `ss2tf()`, with

```
tf(sys_obj)
```

Now you may wonder if we can generate the state-space model directly from a transfer function. The answer is, of course, yes. We can use

```
sys2=ss(tf(q,p))
eig(sys2.a)        % should be identical to the poles
```

MATLAB will return with matrices that look different from those given previously:

$$\mathtt{a} = \begin{bmatrix} -1.5 & -1.125 \\ 2 & 0 \end{bmatrix}, \quad \mathtt{b} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \mathtt{c} = [\,0 \quad 1.125], \quad \mathtt{d} = 0.$$

With what we know now, we bet `ss()` uses a different scaling in its algorithm. This time, MATLAB factors the transfer function into this product form:

$$\frac{Y}{U} = \frac{X_2}{U}\frac{Y}{X_2} = \frac{2}{(s^2 + 1.5s + 2.25)}1.125.$$

From $X_2/U = 2/(s^2 + 1.5s + 2.25)$ and with the state variables defined as

$$x_1 = \frac{1}{2}\frac{\mathrm{d}x_2}{\mathrm{d}t}\left(\text{i.e., } \frac{\mathrm{d}x_2}{\mathrm{d}t} = 2x_1\right), \qquad x_2 = x_2,$$

we should obtain the new state matrices. Again, the key result is that the state matrix `a` has the same eigenvalue.

This exercise underscores one more time that there is no unique way to define state variables. Because our objective here is to understand the association between transfer function and state-space models, the introduction continues with the `ss2tf()` and the `tf2ss()` functions.

Two minor tidbits before we move on. First, the `printsys()` function displays the model matrices or polynomials in a slightly more readable format. Sample usage:

```
printsys(a,b,c,d)
printsys(q,p,'s')
```

Second, with a second-order transfer function, we can generate the textbook state-space matrices, given a natural frequency `wn` and damping ratio `z`:

```
[a,b,c,d]=ord2(wn,z)    % good for only q=1
```

If we examine the values of `b` and `c`, the result is restricted to a unity numerator in the transfer function.

## M4.2. Time-Response Simulation

To begin with, we can launch the LTI Viewer with

```
ltiview
```

as explained in MATLAB Session 3. The graphics interface is designed well enough so that no further explanation is needed.

The use of `step()` and `impulse()` on state-space models is straightforward as well. Here just a simple example is provided. Let's go back to the numbers that we have chosen for Example 4.1 and define

```
a=[0 1; -2.25 -1.5]; b=[0; 2.25]; c=[1 0]; d=0;
sys=ss(a,b,c,d);
```

The `step()` function also accepts state-space representation, and generating the unit-step response is no more difficult than using a transfer function:

```
step(sys)
```

Now we repeat the calculation in the transfer function form and overlay the plot on top of the last one:

```
G=tf(2.25,[1 1.5 2.25]);
hold
step(G,'x')
hold off
```

Sure enough, the results are identical. We would be in big trouble if it were not! In fact, we should get the identical result with other state-space representations of the model. (You may try this yourself with the other set of `a,b,c,d` returned by `tf2ss()` when we first went through Example 4.1.)

Many other MATLAB functions, for example, `impulse()`, `lsim()`, etc., take both transfer function and state-space arguments (what can be called polymorphic). There is very little reason to do the conversion back to the transfer function once you can live in state-space with peace.

## M4.3. Transformations

The following features are covered in this session:

- Similarity and canonical transforms
- Using functions `canon()` and `ss2ss()`

First a similarity transform is demonstrated. For a nonsingular matrix **A** with distinct eigenvalues, we can find a nonsingular (modal) matrix **P** such that the matrix **A** can be transformed into a diagonal made up of its eigenvalues. This is one useful technique in decoupling a set of differential equations.

Consider the matrix **A** from Example 4.6. We check to see if the rank is indeed 3, and compute the eigenvalues for reference later:

```
A=[0 1 0;  0 -1 -2;  1 0 -10];
rank(A)
eig(A)    % -0.29,  -0.69,  -10.02
```

We now enter

```
[P,L] = eig(A)     % L is a diagonal matrix of eigenvalues
                   % P is the modal matrix whose columns are the
                   % corresponding eigenvectors
a = inv(P)*A*P     % Check that the results are correct
```

Indeed, we should find `a` to be the diagonal matrix with the eigenvalues.

The second route is to diagonalize the entire system. With Example 4.6, we further define

```
B=[0;  2;  0];
C=[1  0  0];
D=[0];
S=ss(A,B,C,D);      % Generates the system object
SD=canon(S)
```

The `canon()` function by default will return the diagonalized system and, in this case, in the system object `SD`. For example, we should find `SD.a` to be identical to the matrix `L` that we obtained a few steps back.

The third alternative to generate the diagonalized form is to use the state-space to state-space transform function. The transform is based on the modal matrix that we obtained earlier:

```
SD=ss2ss(S,inv(P))
```

To find the observable canonical form of Example 4.6, we use

```
SO=canon (S,'companion')
```

In the returned system `SO`, we should find `SO.a` and `SO.b` to be

$$\mathbf{A}_{\text{ob}} = \begin{bmatrix} 0 & 0 & -2 \\ 1 & 0 & -10 \\ 0 & 1 & -11 \end{bmatrix}, \qquad \mathbf{B}_{\text{ob}} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}.$$

*Optional reading:*

The rest of this section requires material on the *Web Support* and is better read together with Chap. 9. Using the supplementary notes on canonical transformation, we find that the observable canonical form is the transpose of the controllable canonical form. In the observable canonical form, the coefficients of the characteristic polynomial (in reverse sign) are in the last column. The characteristic polynomial is, in this case,

$$P(s) = s^3 + 11s^2 + 10s + 2.$$

We can check that with

```
roots([1 11 10 2])    % Check the roots
poly(A)               % Check the characteristic polynomial of A
```

We can find the canonical forms ourselves. To evaluate the observable canonical form $\mathbf{A}_{\text{ob}}$, we define a new transformation matrix based on the controllability matrix:

```
P=[B A*B A^2*B];
inv(P)*A*P            % Should be Aob as found by canon()
inv(P)*B              % Shoud be Bob (Bob!)
```

To find the controllable canonical form,

$$\mathbf{A}_{\text{ctr}} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -2 & -10 & -11 \end{bmatrix}, \quad \mathbf{B}_{\text{ctr}} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix},$$

we use the following statements based on the *Web Support* supplementary notes. Be very careful when constructing the matrix **M**:

```
poly(A);              %To confirm that it is [1 11 10 2]
M=[10 11 1; 11 1 0; 1 0 0];
T=P*M;
inv(T)*A*T
inv(T)*B
```

We now repeat the same ideas one more time with Example 4.9. We first make the transfer function and the state-space objects:

```
G=zpk([],[-1 -2 -3],1);
S=ss(G);
```

As a habit, we check the eigenvalues:

```
eig(S)                % Should be identical to eig(G)
```

To find the modal matrix, we use

```
[P,L]=eig(S.a)
inv(P)*S.a*P        % Just a check of L
```

The observable canonical form is

```
SD=canon(S)
```

The component `SD.a` is, of course, the diagonalized matrix `L` with eigenvalues. We can check that `SD.b` and `SD.c` are respectively computed from

```
inv(P)*S.b     % Identical to SD.b
S.c*P          % Identical to SD.c
```

Finally, the observable canonical form is

```
SO=canon(S, 'companion')
```

The matrix `SO.a` is

$$\mathbf{A}_{\text{ob}} = \begin{bmatrix} 0 & 0 & -6 \\ 1 & 0 & -11 \\ 0 & 1 & -6 \end{bmatrix},$$

meaning that

$$P(s) = s^3 + 6s^2 + 11s + 6,$$

which is the characteristic polynomial

```
poly([-1 -2 -3])
```

as expected from the original transfer function.

## Session 5. Feedback Simulation Functions

This tutorial is to complement our development in Chaps. 5 and 6. You may want to go over the tutorial quickly before you read the text and come back later a second time for the details.

Feedback simulation functions

| | |
|---|---|
| `feedback` | Generate feedback-system transfer function object |
| `simulink` | Launch Simulink |

## M5.1. Simulink

### Comments with Respect to Launching Simulink

Simulink is a user-friendly simulation tool with an icon-driven graphics interface that runs within MATLAB. The introduction here is more conceptual than functional for two reasons. One, the Simulink interface design is very intuitive and you may not need help at all! Second, for a thorough introduction, we need to reproduce many of the graphics windows. To conserve

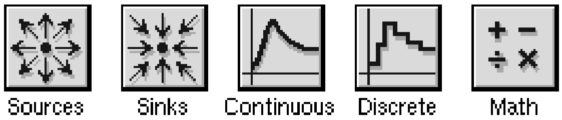


**Figure M5.1.**

paper (and trees), these print-intensive and detailed explanations have been moved to the *Web Support*. Furthermore, the Helpbrowser of MATLAB is extremely thorough and should serve as our main guide for further applications.

To launch Simulink, enter in the Command Window

```
simulink
```

and MATLAB will launch the Simulink Block Library window with pull-down menus. A few sample block library icons are shown in Fig. M5.1. Each icon represents a toolbox and contains within it a set of models, which will make themselves available if we double-click on the toolbox icons. For example, we can find within the Sources toolbox (Fig. M5.1) a model for generating a step input function and within the Sinks toolbox a model for graphing results. Within the Continuous toolbox are the important models for generating transfer functions and state-space models (Fig. M5.2).

All we need is to drag and drop the icons that we need from the toolboxes into a blank model window. If this window is not there, open a new one with the File pull-down menu. From here on, putting a feedback loop together to do a simulation is largely a point-and-click activity. An example of what Simulink can generate is shown in Fig. M5.3.

Simulink is easy to learn, fun, and instructive, especially with more complex MIMO systems. For systems with time delays, Simulink can handle the problem much better than the classical control toolbox. Simulink also has ready-made objects to simulate a PID controller.

A few quick pointers:

- These are some of the features that we use most often within the Simulink Block Library:

| | |
|---|---|
| *Sources:* | Step input; clock for simulation time |
| *Sinks:* | Plotting tools; output to MATLAB workspace or a file |
| *Continuous:* | Transfer functions in polynomial or pole-zero form; state-space models; transport delay |
| *Math:* | Sum; gain or gain slider |
| *Nonlinear:* | Saturation; dead zone |
| *Blocksets:* | From the Blocksets and Toolboxes, choose "Simulink Extras," and then "Additional Linear." In there are the PID and the PID with approximate derivative controllers. |

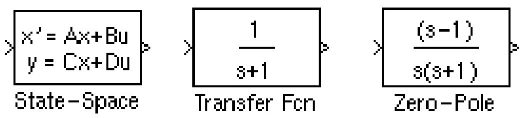


**Figure M5.2.**

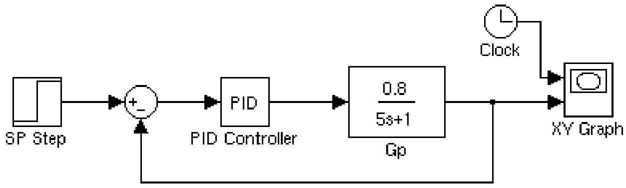


**Figure M5.3.**

- All Simulink simulation block diagrams are saved as ASCII files with the "mdl" extension.
- Before we start a simulation, choose Parameters under the Simulation pull-down menu to select the time of simulation. If we are using the XY Graph, we need to double-click its icon and edit its parameters to make the time information consistent.
- Simulink shares the main MATLAB workspace. When we enter information into, say, the transfer function block, we can use a variable symbol instead of a number. We then define the variable and assign values to it in the MATLAB Command Window. This allows for a much quicker route for doing parametric studies than does changing the numbers within the Simulink icons and dialog boxes.
- We can build our own controllers, but two simple ones are available: an ideal PID and a PID with approximate derivative action.

For curious minds: The first time you use the PID controllers, drag the icon onto a new simulation window, select the icon, and then *Look under mask* under the Edit pull-down menu. You will see how the controllers are put together. The simple PID controller is

$$G_c(s) = K_c + \frac{K_I}{s} + K_D s,$$

and the PID with approximate derivative controller is

$$G_c(s) = K_c + \frac{K_I}{s} + \frac{K_D s + 1}{s/N + 1}.$$

We also see the transfer functions used by each icon when we double-click on it and open up the parameter entry dialog window. Therefore, in terms of notation, we have $K_I = K_c/\tau_I$, $K_D = K_c\tau_D$, and $N = 1/\alpha\tau_D$.

## M5.2. Control Toolbox Functions

The following feature is covered in this session:

- Synthesizing a closed-loop transfer function with `feedback()`

The closed-loop transfer function of a servo problem with proper handling of units is Eq. (5.11) in text:

$$\frac{C}{R} = \frac{K_m G_c G_p}{1 + G_m G_c G_p}.$$

It can be synthesized with the MATLAB function `feedback()`. As an illustration, we use a simple first-order function for $G_p$ and $G_m$ and a PI controller for $G_c$. When all is done, we test the dynamic response with a unit-step change in the reference. To make the reading easier, we break the task up into steps. Generally, we would put the transfer function statements inside an M-file and define the values of the gains and time constants outside in the workspace.

*Step 1*: Define the transfer functions in the forward path. The values of all gains and time constants are arbitrarily selected:

```
km=2;                      % Gc is a PI controller
kc=10;
taui=100;
Gc=tf(km*kc*[taui 1], [taui 0]);

kp=1;
taup=5;
Gp=tf(kp, [taup 1]);       % Gp is the process function
```

In the definition of the controller $G_c$, we have included the measurement gain $K_m$, which usually is in the feedback path and the reference (Fig. 5.4). This is a strategy that helps to eliminate the mistake of forgetting about $K_m$ in the reference. One way to spot whether you have made a mistake is if the system calculation has an offset when in theory you know that it should not.

*Step 2*: Define the feedback path function. Let's presume that our measurement function is first order too. The measurement gain has been taken out and implemented in Step 1:

```
taum=1;                   % Gm is the measurement function
Gm=tf(1, [taum 1]);       % Its s.s. gain km is in Gc
```

*Step 3*: Define the closed-loop function:

```
Gcl=feedback(Gc*Gp,Gm);    % Gcl is the closed-loop function C/R
```

*Comments*:

- By default, `feedback()` uses negative feedback.
- With unity feedback, i.e., $G_m = K_m = 1$, we would simply use

  ```
  Gcl=feedback(Gc*Gp,1)
  ```

  to generate the closed-loop function.
- We could generate a closed-loop function with, for example, `Gc*Gp/(1 + Gc*Gp)`, but this is not recommended. In this case, MATLAB simply multiplies everything together with no reduction and the resulting function is very unclean.

*Step 4*: We can now check (if we want to) the closed-loop poles and do the dynamic simulation for a unit-step change in $R$:

```
disp('The closed-loop poles & s.s. gain:')
pole(Gcl)
dcgain(Gcl)

step(Gcl)   % Of course, we can customize the plotting
```

This is the general idea. You can now put it to use by writing M-files for different kinds of processes and controllers.

When we have a really simple problem, we should not even need to use `feedback()`. Yes, we can derive the closed-loop transfer functions ourselves. For example, if we have a proportional controller with $G_c = K_c$ and a first-order process, all we need are the following statements, which follow Example 5.1 and Eq. (E5.1) in text:

```
kc=1;
kp=0.8;
taup=10;
Gcl=tf(kc*kp,[taup 1+kc*kp]);
pole(Gcl)
step(Gcl);   % Again for unit-step change in R
```

Try a proportional controller with a second-order process as derived in Example 5.2 in text. This is another simple problem for which we do not really need `feedback()`.

We now finish up with what we left behind in Session 4. Let's revisit Example 4.6. For checking our results later, we first find the poles of the closed-loop transfer function with

```
q=2*[1 10];
p=[1 11 10 2];
roots(p)     % -0.29, -0.69, and -10.02
```

Next, we define each of the transfer functions in the example:

```
G1=tf(1,[1 0]);
G2=tf(2,[1 1]);
H=tf(1,[1 10]);
```

Note that the numbering and notation are entirely arbitrary. We now generate the closed-loop transfer function and check that it has the same closed-loop poles:

```
Gcl=feedback(G1*G2,H);
pole(Gcl)
```

We can also easily obtain a state-space representation and see that the eigenvalues of the state matrix are identical to the closed-loop poles:

```
ssm=ss(Gcl);
eig(ssm.a)
```

For fun, we can recover the closed-loop transfer function `Gcl` with:

```
tf(ssm)
```

One final check with our own derivation. We define the coefficient matrices with Eqs. (E4.23) and (E4.24) and then do the conversion:

```
a=[0 1 0; 0 -1 -2; 1 0 -10];
b=[0; 2; 0];
c=[1 0 0];
d=0;
eig(a)                          % should return the same
[q3,p3]=ss2tf(a,b,c,d,1)        % eigenvalues and transfer
                                % function
```

If this is not enough to convince you that everything is consistent, try `step()` on the transfer function and different forms of the state-space model. You should see the same unit-step response.

## Session 6. Root-Locus Functions

This tutorial is to complement our development in Chap. 7. You may want to go over the tutorial quickly before you read the text and come back later a second time for the details.

Root-locus functions

| | |
|---|---|
| `rlocus` | Root-locus plot |
| `rlocfind` | Find the closed-loop gain graphically |
| `sgrid` | Draw the damping and natural frequency lines |
| `sisotool` | Launch the SISO system design graphics interface |

### M6.1. Root-Locus Plots

The following features are covered in this session:

- Root-locus calculation and plots, `rlocus()`
- Frequency and damping factor grid, `sgrid()`
- Obtaining gain of chosen closed-loop pole, `rlocfind()`

In simple terms, we want to solve for $s$ in the closed-loop equation

$$1 + G_0(s) = 1 + kG(s) = 0,$$

where we further write $G_0 = kG(s)$ and $G(s)$ is the ratio of two polynomials, $G(s) = q(s)/p(s)$. In the simplest case, we can think of the equation as a unity feedback system with only a proportional controller (i.e., $k = K_c$) and $G(s)$ as the process function. We are interested in finding the roots for different values of the parameter $k$. We can either tabulate the results or we can plot the solutions $s$ in the complex plane – the result is the root-locus plot.

Let's pick an arbitrary function such that $q(s) = 1$ and $p(s) = s^3 + 6s^2 + 11s + 6$. We can generate the root-locus plot of the system with:

```
p=[1 6 11 6];
roots(p)                 % Check the poles
G=tf(1,p);
rlocus(G)                % Bingo!
```

For the case in which $q(s) = s + 1$, we use

```
G=tf([1 1],p);            % Try an open-loop zero at -1
rlocus(G)                 % to cancel the open-loop pole at -1
```

MATLAB automatically selects a reasonable vector for $k$, calculates the roots, and plots them. The function `rlocus()` also adds the open-loop zeros and poles of $G(s)$ to the plot.

Let's try two more examples with the following two closed-loop characteristic equations:

$$1 + K\frac{1}{(s+1)(s+3)} = 0, \qquad 1 + K\frac{1}{(s+1)(s+2)(s+3)} = 0;$$

```
G=zpk([],[-1 -3],1)        % The second-order example
rlocus(G)

G=zpk([],[-1 -2 -3],1)     % The third-order example
rlocus(G)
```

The point of the last two calculations is that a simple second-order system may become extremely underdamped, but it never becomes unstable.

*Reminder*: We supply the polynomials $q(s)$ and $p(s)$ in $G(s)$, but do not lose sight that MATLAB really solves for $s$ in the equation $1 + kq(s)/p(s) = 0$.

In the initial learning stage, it can be a bad habit to rely on MATLAB too much. Hence the following two exercises take the slow way in making root-locus plots, which, it is hoped, may make us more aware of how the loci relate to pole and zero positions. The first thing, of course, is to identify the open-loop poles:

```
q=[2/3 1];          % Redefine q(s) and p(s)
p=[1 6 11 6];
poles=roots(p)'     % display poles and zeros as row vectors
zeros=roots(q)'
G=tf(q,p);
k=0:0.5:100;        % define our own gains; may need
                    % 0:0.1:10 to better see the break-off point
rlocus(G,k);        % MATLAB will plot the roots with '+'
```

Until we have more experience, it will take some trial and error to pick a good range and increment for $k$, but then that is the whole idea of trying it ourselves. This manual approach makes us better appreciate the placements and changes of closed-loop poles as we vary the proportional gain.[12]

We may also want to override the MATLAB default format and use little dots:

```
r=rlocus(G,k);     % Save loci to array "r" first
plot(r,'.')        % Now use plot() to do the dots
hold               % hold the plot to add goodies
pzmap(G)           % pzmap() draws the open-loop poles
hold off           % and zeros
```

Be careful to read where the loci are on the real axis because `pzmap()` also traces the axis with little dots, which can be confusing.

We may want to find the ultimate gain when the loci cross the imaginary axis. Again there are many ways to do it. The easiest method is to estimate with the MATLAB function `rlocfind()`, which is introduced next.

[12] The gain vector generated automatically by MATLAB is not always instructive if we want to observe the region close to the imaginary axis. We can use "tricks," like making two gain vectors with different increments, concatenating them, and using the result in `rlocus()`. However, we should not get bogged down with fine details here. Certainly for day-to-day routine calculations, we can omit the gain vector and let MATLAB generate it for us.

There are two very useful MATLAB features. First, we can overlay onto the root-locus plot lines of the constant damping factor and the natural frequency. These lines help us pick the controller gain if the design specification is in terms of the frequency or the damping ratio,

```
sgrid             % use the default grid
```

or, better yet,

```
sgrid(zeta,wn)    % plot only lines with given damping ratio
                  % and natural frequency
```

*Example:*

```
sgrid(0.7,1)      % add the approx. 45° line for zeta=0.7 and
                  % the unit circle (frequency=1)
```

The second feature is the function `rlocfind()`, which allows us to find the gain associated with a closed-loop pole. We can enter

```
[ck,cpole]=rlocfind(G)
```

or

```
rlocfind(G)       % this simple form will not return
                  % the values of the closed-loop poles
```

MATLAB will wait for us to click on a point (the chosen closed-loop pole) in the root-locus plot and then return the closed-loop gain (`ck`) and the corresponding closed-loop poles (`cpole`). MATLAB does the calculation with the root-locus magnitude rule, which is explained on the *Web Support*.

What if we click a point not exactly on a root locus? When we select a point $s^*$, MATLAB calculates the value $k^* = -p(s^*)/q(s^*)$, which will be a real positive number only if $s^*$ satisfies the closed-loop equation. Otherwise, $k^*$ is either complex or negative if the pole is a real number. In this case, MATLAB calculates the magnitude of $k^*$, uses it as the gain, and computes the corresponding closed-loop poles. Thus we find that the chosen points are always right on the root loci, no matter where we click.

We may also want to use the `zoom` feature of MATLAB to zoom in and out of a plot to get a better picture of, say, the break-off point of two loci. Make sure you enter "`zoom off`" when you are done.

## M6.2. SISO System Design Graphics Interface

The following feature is covered in this session:

- Graphics user interface for designing SISO systems, `sisotool` [13]

The control toolbox supports an extremely nice SISO system design tool that is ideal for experimentation. This graphics interface is even more intuitive and self-explanatory than that of Simulink. The same approach is taken as that of our introduction to Simulink, and

[13] The description is based on Version 5.1 of the MATLAB control toolbox. If changes are introduced in newer versions, they will be presented on the *Web Support*.

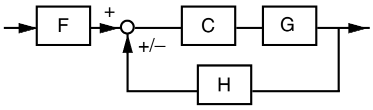


**Figure M6.1.**

the not-so-necessary and print-intensive window display and instructions have been moved to the *Web Support*. Only a very brief conceptual introduction is provided here.

To launch the SISO system design tool, enter in the MATLAB Command Window

```
sisotool             % default view
```

or

```
sisotool('rlocus')    % root-locus view only
```

A graphics window with pull-down menus and tool buttons will pop out, slowly. The default view displays both the root-locus and the Bode editors. Because we have not learned Bode plots yet, the second option with `rlocus` is less intimidating for the moment. Here are some pointers on the usage of the tool:

- The SISO design tool supports a flexible block diagram, as shown in Fig. M6.1. The feedback can be either positive or negative. Similar to the LTI Viewer, the tool runs in its own functional space. We have to import the transfer functions under the File pull-down menu. By default, the transfer functions $F$, $C$, $G$, and $H$ are all assigned the value "1," so we have to import at least a transfer function for $G$ to do meaningful calculations.
- The default compensator $C$ in Fig. M6.1 is a proportional controller, but it can be changed to become a PI, PD, or PID controller. The change can be accomplished many ways. One is to retrieve the compensator-editing window by clicking on the $C$ block or by using the Compensator pull-down menu. We can also use the set of button on the toolbar to add or move open-loop poles and zeros associated with the controller.
- Once a root-locus plot is generated, we can interactively change the locations of the closed-loop poles and the tool will compute the corresponding controller gain for us.
- For a given system and chosen closed-loop poles displayed in the root-locus plot, we can generate its corresponding time-response and frequency-response plots with features under the Tools pull-down menu.

In the next section, you can use the SISO design tool if you prefer, but the explanation is given with commands. It is easier to get the message across with commands, and in the beginner's learning stage, entering your own command can give you a better mental imprint of the purpose of the exercise.

## M6.3. Root-Locus Plots of PID Control Systems

The following feature is covered in this session:

- Making root-locus plots that model situations of PID control systems

Here are some useful suggestions regarding root-locus plots of control systems. In the following exercises, we consider only the simple unity feedback closed-loop characteristic equation:

$$1 + G_c G_p = 0.$$

We ignore the values of any gains. We focus on only the probable open-loop pole and zero positions introduced by a process or by a controller, or, in other words, the shape of the root-locus plots.

Let's begin with a first-order process $G_p = 1/(s+1)$. The root-locus plot of a system with this simple process and a proportional controller, $G_c = K_c$, is generated as follows:

```
Gp=tf(1,[1 1]);               % open-loop pole at -1
subplot(221), rlocus(Gp)      % Gc = Kc
```

To implement an ideal PD controller, we will have an additional open-loop zero. Two (of infinite) possibilities are

```
taud=2;                       % open-loop zero at -1/2
Gc=tf([taud 1],1);
subplot(222), rlocus(Gc*Gp)
```

and

```
taud=1/2;                     % open-loop zero at -2
Gc=tf([taud 1],1);
subplot(223), rlocus(Gc*Gp)
```

What are the corresponding derivative time constants? Which one would you prefer?

We next turn to a PI controller. We first make a new figure and repeat proportional control for comparison:

```
figure(2)
subplot(221), rlocus(Gp)    % Gc = Kc
```

Integral control will add an open-loop pole at the origin. Again, we have two regions where we can put the open-loop zero:

```
taui=2;                       % open-loop zero at -1/2
Gc=tf([taui 1],[taui 0]);
subplot(222), rlocus(Gc*Gp)
```

and

```
taui=1/2;                     % open-loop zero at -2
Gc=tf([taui 1],[taui 0]);
subplot(223), rlocus(Gc*Gp)
```

Once again, what are the corresponding integral time constants? Which one would you prefer?

Finally, let's take a look at the probable root loci of a system with an ideal PID controller, which introduces one open-loop pole at the origin and two open-loop zeros. For illustration, we will not use the integral and derivative time constants explicitly, but refer to only the two zeros that the controller may introduce. We will also use `zpk()` to generate the transfer functions:

```
figure(3)
subplot(221), rlocus(Gp)        % redo Gc = Kc

op_pole=[0];                    % open-loop pole at 0

op_zero=[-0.3 -0.8];            % both zeros larger than -1
Gc=zpk(op_zero,op_pole,1);
subplot(222),rlocus(Gc*Gp)

op_zero=[-1.5 -3];              % both zeros less than -1
Gc=zpk(op_zero,op_pole,1);
subplot(223),rlocus(Gc*Gp)

op_zero=[-0.5 -1.8];            % one zero in each region
Gc=zpk(op_zero,op_pole,1);
subplot(224),rlocus(Gc*Gp)
```

Yes, you know the question is coming. Which case would you prefer? We can use the rule of thumb that the derivative time constant is usually approximately one fourth the value of the integral time constant, meaning that the zero farther away from the origin is the one associated with the derivative time constant.

Note that the system remains stable in all cases, as it should for a simple first- or second-order system. One final question: Based on the design guidelines by which the system should respond faster than the process and the system should be slightly underdamped, what are the ranges of derivative and integral time constants that you would select for the PD, PI, and PID controllers? And in what region are the desired closed-loop poles?

We'll finish with implementing the P, PI, and PD controllers on a second-order overdamped process. As in the previous exercise, try to calculate the derivative or integral time constants and take a minute to observe the plots and see what may lead to better controller designs.

Let's consider an overdamped process with two open-loop poles at $-1$ and $-2$ (time constants at 1 and 0.5 time units). A system with a proportional controller would have a root-locus plot as follows. We stay with `tf()`, but you can always use `zpk()`.

```
figure(1)
p=poly([-1 -2]);                % open-loop poles -1, -2
Gp=tf(1,p);
subplot(221),rlocus(Gp)         % proportional control
```

To implement an ideal PD controller, we now have three possible regions in which to put the zero:

```
taud=2;                         % open-loop zero at -1/2
Gc=tf([taud 1],1);
subplot(222), rlocus(Gc*Gp)

taud=2/3;                       % open-loop zero at -1.5
Gc=tf([taud 1],1);
subplot(223), rlocus(Gc*Gp)
```

```
taud=1/3;                         % open-loop zero at -3
Gc=tf([taud 1],1);
subplot(224), rlocus(Gc*Gp)
```

We will put the PI controller plots on a new figure:

```
figure(2)
subplot(221),rlocus(Gp)          % redo proportional control
```

The major regions in which to place the zero are the same, but the interpretation as to the choice of the integral time constant is very different. We now repeat, adding the open-loop zeros:

```
taui=2;                           % open-loop zero at -1/2
Gc=tf([taui 1],[taui 0]);
subplot(222), rlocus(Gc*Gp)

taui=2/3;                         % open-loop zero at -1.5
Gc=tf([taui 1],[taui 0]);
subplot(223), rlocus(Gc*Gp)

taui=1/3;                         % open-loop zero at -3
Gc=tf([taui 1],[taui 0]);
subplot(224), rlocus(Gc*Gp)
```

You may want to try some sample calculations using a PID controller. One way of thinking: We need to add a second open-loop zero. We can limit the number of cases if we assume that the value of the derivative time constant is usually smaller than the integral time constant.

## Session 7. Frequency-Response Functions

This tutorial is to complement our development in Chap. 8. You may want to go over the tutorial quickly before you read the text and come back later a second time for the details.

Frequency-response functions

| | |
|---|---|
| `bode` | Bode plots |
| `freqresp` | Frequency response of a transfer function |
| `logspace` | Logarithmically spaced vector |
| `margin` | Gain margin and crossover frequency interpolation |
| `nichols, ngrid` | Nichols plots |
| `nyquist` | Nyquist plots |
| `sisotool` | Launch the SISO system design graphics interface |

## M7.1. Nyquist and Nichols Plots

The following feature is covered in this session:

- Nyquist plots, `nyquist()`

The SISO system design tool `sisotool`, as explained in Session 6, can be used to do frequency-response plots. Now we want to use the default view, so we just need to enter

```
sisotool
```

Hints to make better use of the design tool are on the *Web Support*. We use commands here because they give us a better idea behind the calculations. This section is brief as our main tool will be Bode plots, which will be explained in the next section.

Let's say we have a simple open-loop transfer function $G_0$ of the closed-loop characteristic equation,

$$1 + G_0 = 0,$$

and we want to find the proportional gain that will give us an unstable system. For this simple exercise, we take $G_0(s) = KG(s)$:

```
p=poly([-1; -2; -3]);     % Open-loop poles at -1, -2, -3
G=tf(10,p);               % Arbitrary value K=10
nyquist(G);               % Bingo!
```

We'll see two curves. By default, MATLAB also maps and plots the image of the negative imaginary axis. That can make the plot too busy and confusing, at least for a beginner. So we'll stay away from the default in the following exercises:

```
[re,im]=nyquist(G);
plot(re(1,:),im(1,:))     % Only the positive Im-axis image
```

Of course, we can define our own frequency vector:

```
w=logspace(-1,1);       % Generate numbers between [10^-1, 10^1]
[re,im]=nyquist(G,w);
plot(re(1,:),im(1,:))
```

The function `logspace()` generates a vector with numbers nicely spaced on the logarithmic scale. Its use is optional. The default of the function gives 50 points and is usually adequate. For a smoother curve, use more points. For example, this command will use 150 points: `logspace(-1,1,150)`.

```
hold                     % to add the (-1,0) point and the axes
                         % on the plot
x=-1; y=0;
xh=[-2 2]; yh=[0 0];     % the axes
xv=[0 0];   yv=[-2 1];
plot(x,y,'o',xh,yh,'-',xv,yv,'-')
```

We can increase the gain $K$ and repeat the calculation with, for example, two more trials[14]:

```
G=tf(50,p);              % try again
[re,im]=nyquist(G,w);
plot(re(1,:),im(1,:))

G=tf(60,p);              % and again
[re,im]=nyquist(G,w);
plot(re(1,:),im(1,:))
hold off
```

We do not use the Nichols plot (log magnitude versus phase) much, but it is nice to know that we can do it just as easily:

```
p=poly([-1; -2; -3]);
G=tf(10,p);
nichols(G)
ngrid
zoom                     % need to zoom into the meaningful region
```

The plot with default settings is quite useless unless we use `ngrid` to superimpose the closed-loop gain and phase grid lines. Instead of zooming in, we can reset the axes with

```
axis([-360 0 -40 20])
```

## M7.2. Magnitude and Phase-Angle (Bode) Plots

The following features are covered in this session:

- Bode plot calculation, `bode()`
- Finding the gain and phase margins, `margin()`
- Bode plots for transfer functions with dead time

We begin with one simple example. Let's say we want to analyze the closed-loop characteristic equation

$$1 + \frac{1}{s^2 + 0.4s + 1} = 0.$$

We generate the Bode plot with

```
G=tf(1,[1 0.4 1]);
bode(G)                 % Done!
```

The MATLAB default plot is perfect! That is, except when we may not want decibels as the unit for the magnitude. We have two options. One, learn to live with decibels, the

[14] All functions like `nyquist()`, `bode()`, etc., can take on multiple LTI objects, as in

```
nyquist(G1,G2,G3)
```

but only when we do not use LHS arguments.

convention in the control industry, or two, we do our own plots. This is a task that we need to know when we analyze systems with dead time. This is how we can generate our own plots:

```
w=logspace(-1,1);
[mag,phase]=bode(G,w);
mag=mag(1,:);                   % required since MATLAB v.5
phase=phase(1,:);

subplot(211), loglog(w,mag)
              ylabel('Magnitude'), grid
subplot(212), semilogx(w,phase)
              ylabel('Phase, deg'), grid
              xlabel('Frequency (rad/time)')
```

As an option, we can omit the subplot command and put the magnitude and phase plots in individual figures.

This is how we can make a Bode plot with decibels as the scale for the magnitude.

```
dB=20*log10(mag);     % converts magnitude to decibels
```

Now we do the plotting. Note that the decibel unit is already a logarithmic scale:

```
subplot(211),     semilogx(w,dB)   % Use semilogx for dB
                  ylabel('Magnitude(dB)')
                  grid
subplot(212),     semilogx(w,phase)
                  ylabel('Phase angle (degree)')
                  xlabel('Frequency, w')
                  grid
```

We most often use radians per second as the unit for frequency. In the case in which cycles per second or hertz are needed, the conversion is

```
f=w/(2*pi);            % Converts w [rad/s] to [Hz]
```

After using the `subplot()` command and before doing any other plots, we should make it a habit to reset the window with

```
clf                    % clear figure
```

We now find the gain margin with its crossover frequency (`Gm, Wcg`) and phase margin with its crossover frequency (`Pm, Wcp`) with either one of the following options:

```
[Gm,Pm, Wcg,Wcp]=margin(mag,phase,w)     % option 1
```

where `mag` and `phase` are calculated with the function `bode()` beforehand. We can skip the `bode()` step and use the transfer function directly as the argument,

```
[Gm,Pm, Wcg,Wcp]=margin(G)     % option 2
```

or simply

```
margin(G)         % option 3, Gm in dB
```

In the last option without any LHS arguments, MATLAB will do the Bode plot, and display the margin calculations on the plot.
Two important comments:

(1) With `G=tf(1,[1 0.4 1])`, i.e., a simple second-order system, it is always stable. The gain margin calculation is meaningless. Nevertheless, MATLAB returns a set of results anyway. Again, a computer is not foolproof. All `margin()` does is an interpolation calculation.
(2) If you use option 1 or 2 above, `margin()` returns the linear-scale gain margin in the variable `Gm`. With option 3, however, the gain margin displayed in the plot is in decibels, you need to convert it back with $10^{\mathrm{dB}/20}$.

To handle dead time, all we need is a simple modification using the fact that the time delay transfer function has magnitude 1 and phase angle $-t_d\omega$. We need one single statement to "tag on" the lag that is due to dead time, and we do it after the `bode()` function call.

So let's start with the second-order function, which is always stable:

```
G=tf(1,[1 0.4 1]);
freq=logspace(-1,1);          % freq is in rad/time
[mag,phase]=bode(G,freq);
mag=mag(1,:);
phase=phase(1,:);
```

Now let's say we also have dead time:

```
tdead=0.2;     % [time unit]
```

The following statement is the only addition needed to introduce the phase lag that is due to dead time:

```
phase = phase - ((180/pi)*tdead*freq);     % phase is in degrees
```

We can now proceed with the plotting and phase/gain margin interpolation:

```
subplot(211),     loglog(freq,mag)
                  ylabel('Magnitude'),title('Bode Plot')
                  grid
subplot(212),     semilogx(freq,phase)
                  ylabel('Phase(degree)'),xlabel('Frequency')
                  grid

% now using new phase variable that includes dead-time phase lag
[Gm,Pm,Wcg,Wcp]=margin(mag,phase,freq)
```

The whole idea of handling dead time applies to other types of frequency-domain plots, but the Bode plot is the easiest to learn from.

There is no magic in the functions `nyquist()` or `bode()`. We could have done all our calculations by using the more basic `freqresp()` function. What it does essentially is make the $s = j\omega$ substitution numerically in a given transfer function $G(s)$. A sample usage is

```
w=logspace(-1,1);
gjw=freqresp(G,w);     % does the s=jw calculation for each
                       % value in w
```

After that, we can use the result to do frequency-response analysis. If you are interested in the details, they are provided in the Session 7 Supplement on the *Web Support*.