Chapter 1
Introduction to MATLAB

MATLAB™ is an interactive environment for scientific and engineering calculations, simulations, and data visualization. MATLAB™ provides a powerful platform to solve mathematical and engineering problems related to matrix algebra, differential equations, etc. The basic set of MATLAB™ operations can be extended by *toolboxes*. A toolbox is a function library developed to support calculations in a specific subject area. Such special subject areas include signal processing (*Signal Processing Toolbox*), control engineering (*Control System Toolbox*), image processing (*Image Processing Toolbox*), identification (*Identification Toolbox*), the application of neural networks (*Neural Network Toolbox*), etc. The graphical interface of SIMULINK™ provides possibilities for modelling and simulating processes.

MATLAB™ works as an interpreter: it executes the commands row by row. This mode generally results in slow operation. Programs written in MATLAB™ can be accelerated by using matrix operations. In this case there is no need to write cycles: the inner code of MATLAB™ realizes these operations. As matrix operations in MATLAB™ are executed in optimized machine code, the runtime of these programs is similar to that written in other programming languages (e.g. C++); in the case of big matrices, the runtime is even shorter.

The commands can be MATLAB™ functions or so-called *script* files. An *m*–file is a simple text file containing a sequence of MATLAB™ commands and it has the *.m* extension. This series of commands can be executed by writing the name of the file (without the extension). From the MATLAB™ *m*-files, C and C++ files or function libraries (″.lib″, ″.dll″) can also be created using the MATLAB™ translator.

L. Keviczky et al., *Control Engineering: MATLAB Exercises*,
Advanced Textbooks in Control and Signal Processing,
https://doi.org/10.1007/978-981-10-8321-1_1

1.1 Basic Operation of MATLAB™

The goal of this introduction is to enable the newcomer to use MATLAB™ as quickly as possible. However, for detailed descriptions the user should consult the MATLAB™ manuals. They can be found in electronic form in the '*matlab/help*' directory. Also, on-line help is at the MATLAB™ user's disposal.

```
helpdesk
```

The *help* command displays information about any command. For example:

```
help sqrt
```

A script file of MATLAB™ commands can be created. This is a text file with the extension "*.m*". This script file can be used as a new command (without the extension).

Variable names: The maximum length is 31 characters (letters, numbers and underscore). The first character must be a letter. Lower and upper cases are distinguished. Every variable is treated as a matrix. A scalar variable is a 1 by 1 matrix.

1.1.1 Data Entry

If data entry or any other statement/operation is not terminated by a semicolon, the result of the statement will always be displayed. MATLAB™ can use several types of variables. The type declaration is automatic.

Integer:

```
k=2
```

If the command is ended by a semicolon, then the result is not shown on the display, e.g.:

```
J=-4;
```

Real:

```
s=3.6
F2=-12.6e-5
```

Complex:

```
z=3+4*i
r=5*exp(i*pi/3)
```

Although **`i=sqrt(-1)`** is predefined, you may want to denote the unit imaginary vector by another variable. You are allowed to do so, e.g. simply type

```
j=sqrt(-1)
```

Vectors:

```
x=[1, 2, 3]  % row vector, its elements are separated by commas or spaces
q=[4; 5; 6]  % column vector, its elements are separated by semicolons
```

A column vector can be formed from a raw vector by transposition

```
v=[4, 5, 6]'  % the same as q
```

Remark be careful when using the transpose operation! For complex variables it results in the complex conjugate:

```
i'
      0 - 1.000i
```

Matrices:

```
A=[7, 8, 9; 5, 6, 7]
```

Here **A** is a 2 x 3 matrix, MATRIX = [row1; row2; …; rowN];

Special vectors and matrices:

```
u=1:3;       % generates u = [1 2 3] as a row vector, » u = start : stop
w=1:2:10;    % generates w = [1 3 5 7 9],          » w = start : increment : stop
E=eye(4)
      E=
          1   0   0   0
          0   1   0   0
          0   0   1   0
          0   0   0   1
B=eye(3,4)
      B=
                 1   0   0   0
                 0   1   0   0
                 0   0   1   0
C=zeros(2,4)
      C=
                 0   0   0   0
                 0   0   0   0
```

```
D=ones(3,5)
     D=
               1   1   1   1   1
               1   1   1   1   1
               1   1   1   1   1
```

Variable values:

Typing the name of a variable displays its value:

```
A
   A=
         7  8  9
         5  6  7
A(2, 3)
    ans =
         7
```

The first index is the row number and the second index is the column number. The answer is stored in the `ans` variable.

Changing one single value in **v** results in the printing of the entire vector **v,** unless printing is suppressed by a semicolon:

```
v(2)= -6
   v=
         4
        -6
         6
```

Subscripting: A colon (:) can be used to access multiple elements of a matrix. It can be used in several ways for accessing and setting matrix elements.

Start index : end index—means a part of the matrix

: a colon in the index means all the elements in a row or in a column

For vectors:	v=[v(1) v(2) . . . v(N)]
For matrices:	M=[M(1,1)...M(1,m); M(2,1)...M(2,m); ... ; M(n,1)...M(n,m)]

Assume B is an 8 × 8 matrix, Then

B(1:5,3)	is a column vector, **[B(1,3); B(2,3); B(3,3); B(4,3); B(5,3)]**
B(2:3,4:5)	is a matrix **[B(2,4) B(2,5); B(3,4) B(3,5)]**
B(:,3)	assigns all the elements of the third column of **B**
B(2,:)	assigns the second row of **B**
B(1:3,:)	assigns the first three rows of **B**
A(2,1:2)	% second row of matrix A, with its first and second elements
A(:,2)	% all the elements in the second column

1.1.2 Workspace

The used variables are stored in a memory area called the *workspace*. The workspace can be displayed by the following commands:

```
who
whos            % displays also the size of the variables
```

The size of the variables can be displayed by the commands length and size.

For vectors:

```
lng=length(v)
 lng=
       3
```

For matrices and vectors:

```
[m,n]=size(A)
 m=
    2  % number of rows

 n=

    3  % number of columns
```

The workspace can be saved, loaded and cleared:

save	% saves the workspace to the default *matlab.mat* file.
save filename.mat	% saves the workspace to filename.mat file.
clear	% clears the workspace, deletes all the variables.
load	% loads the default matlab.mat file from the workspace.
load filename.mat	% loads the filename.mat file from the workspace.

1.1.3 Arithmetic Operations

Addition and subtraction:

```
A=[1  2;  3  4];
B=A';
C=A+B;
          C=
               2   5
               5   8
D=A-B
          D=
               0  -1
               1   0
x=[-1  0  2]';
y=x-1                    % Observe that all entries are affected!
     y=
          -2
          -1
           1
```

Multiplication:

Vector by scalar:

```
2*x
      ans=
            -2
             0
             4
```

Matrix by scalar:

```
3*A
      ans=
        3    6
        9   12
```

Inner (scalar) product:

```
s=x'*y
    s=
        4
    y'*x
    ans=
        4
```

Outer product:

```
M=x*y'
     M=
           2    1      -1
           0    0       0
          -4   -2       2
y*x'
     ans=
           2     0      -4
           1     0      -2
          -1     0       2
```

Matrix by vector:

```
b=M*x
     b=
       -4
        0
        8
```

Division:

```
C/2
```

For matrices: B/A corresponds to $B^{*}A^{-1}$; A\B corresponds to $A^{-1}{*}B$.

Powers: A^p, where A is a square matrix and p is a real constant, e.g.: the inverse of A:
A^(-1), or equivalently one can use the command inv(A).

1.1.4 Manipulations of Complex Numbers

```
c=4+2i
    c =
        4.0000 + 2.0000i
real(c)
    ans =
        4
imag(c)
    ans =
        2
abs(c)
    ans =
        4.4721
angle(c)  % the result is in radian
    ans =
        0.4636
```

To get the phase in degrees

```
angle(c)*180/pi
     ans =
        26.5651
```

1.1.5 Array Operations Element-by-Element

Element by element (.*) operations on arithmetic arrays constitute an important class of operations. To indicate an array operation to be executed elementwise the operator should be preceded by a point: a.*b = [a(1)*b(1), a(2)*b(2), ..., a(n)*b(n)]. The sizes of the variables must be the same.

Example

```
a=[2  4  6]
b=[5  3  1]
a.*b
     ans=
          10   12   6
```

The command can be used for different operations, e.g. for division and powers: ./, .^

1.1.6 Elementary Mathematical Functions

(Use the on-line help for details and additional items)

abs	absolute value or magnitude of a complex number
sqrt	square root
real	real part
imag	imaginary part
conj	complex conjugate
round	round to nearest integer
fix	round towards zero
floor	round towards -infinity
ceil	round towards +infinity
sign	signum function
rem	remainder
sin	sine
cos	cosine
tan	tangent
asin	arcsine
acos	arccosine
atan	arctangent
atan2	four quadrant arctangent
sinh	hyperbolic sine
cosh	hyperbolic cosine
tanh	hyperbolic tangent
exp	exponential base *e*
log	natural logarithm
log10	log base 10
bessel	BESSEL function
rat	rational approximation
expm	matrix exponential
logm	matrix logarithm
sqrtm	matrix square root

For example:

```
help  sqrt
g=sqrt(2)
```

1.1.7 Cell Array Data Type

A *cell array* is a matrix whose elements are also matrices. A *cell array* can be given e.g. as follows:

```
ca={1, [1,2],[1,2,3]}
        ca = [1]   [1x2 double]   [1x3 double]
ca{2}
         ans = 1    2
```

1.1.8 Graphics Output

The most basic graphics command is plot.

```
plot(2,3)                % plots the point given by the coordinates x=2, y=3 .
```

Multiple points can be plotted by storing the coordinate values in vectors.

```
x=[1,2,3]
y=[0,2,1]
plot(x,y)                % the points are connected with a line
plot(x,y,'*')            % only the points are plotted
```

This method can plot quite sophisticated curves, too.

Example

```
t=0:0.05:4*pi;
y=sin(t);
plot(t,y)
title('Sine function'),
xlabel('Time'),ylabel('sin(t)'),grid on;
```

where the title, xlabel, ylabel and grid commands are optional.
Plotting more curves in the same coordinate system:

```
y1=3*sin(2*t);
plot(t,y,'r',t,y1,'b'); % r - red, b - blue
```

The typeface and colour for plotting (optional) can be given as follows: » plot(t, y, '@#'), where '@' means line type:

–	solid
– –	dashed
:	dotted
.	point
+	plus
*	star
o	circle
x	x-mark

and '#' means colour as follows:

r	red
g	green
b	blue
w	white
y	yellow

1.1.9 Polynomials

To define a polynomial, e.g. $\mathcal{P}(x) = 2x^5 - 3x^4 + 5x^3 - x^2 - 10x$ simply introduce a vector containing the coefficients of the polynomial:

```
p=[2 -3 5 -1 -10 0]
```

The roots of the polynomial, i.e. the solutions of the equation $\mathcal{P}(x) = 0$ can be calculated by the command roots.

```
xi=roots(p)
   xi =
        0
        0.4756 + 1.7910i
        0.4756 - 1.7910i
        1.5119
       -0.9630
```

It can be seen that this polynomial has three real and two complex roots. Complex roots always appear in conjugate pairs.

From the roots $p_1, p_2, \ldots, p_n$ of a polynomial, the coefficients of the polynomial $\mathcal{P}(x) = (x - p_1)(x - p_2)\ldots(x - p_n)$ can be calculated by

```
p1=poly(xi)
      p1 =
            1.0000 -1.5000 2.5000 -0.5000 -5.0000 0
```

The command poly results in a polynomial with a leading coefficient of 1, therefore to get the original polynomial we have to multiply by 2.

```
2*p1
```

It can be seen that we have obtained the original polynomial.

Let us graph the polynomial $\mathcal{P}(x)$ in the region [−1.5,2].

```
x=[-1.5:0.01:2]
y=p(1)*x.^5+p(2)*x.^4+p(3)*x.^3+p(4)*x.^2+p(5)*x+p(6);
plot(x,y),grid
```

In Fig. 1.1, it can be seen that the crosspoints of the function with the x axis coincide with the real roots.

Consider now the following matrix:

```
M=[3 5; 7 -1]
```

The eigenvalues of $\boldsymbol{M}$ can be computed by the command eig.

```
e=eig(M)
     e =
           7.2450
          -5.2450
```

Taking the above values as the roots of a polynomial, that polynomial can be calculated by the command poly

```
poly(e)
     ans =
          1 -2 -38
```

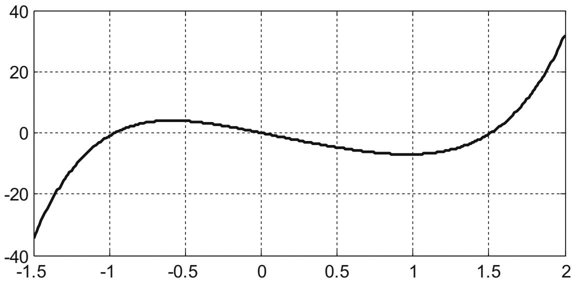

Fig. 1.1 Plotting a polynomial

The above polynomial $x^2 - 2x - 38$ is the characteristic polynomial of $\boldsymbol{M}$, which is defined as $\det(x\boldsymbol{I} - \boldsymbol{M})$. The characteristic polynomial can be directly calculated from $\boldsymbol{M}$:

```
poly(M)
    ans =
          1 -2 -38
```

As can be seen, sometimes commands can be called in different ways. MATLAB™ help of the particular command will provide all the possibilities for how to call it.

1.1.10 Writing MATLAB™ Programs

In the simplest mode of using MATLAB™, we write the commands to be executed in the *command window* of MATLAB™. For solving more complex tasks, this is a long procedure difficult to implement. MATLAB™ provides several possibilities for writing programs. There are two ways to produce programs: in the form of a *script* file or in the form of a *function* file. These programs are simple text files which contain MATLAB™ commands as rows. The extension of the files is *.m*, therefore they are called m-files. The m-files can be written in any text editor, but it is preferable to use the text editor of MATLAB, as it provides several helps for formatting and finding and correcting errors.

A script file program contains MATLAB commands. It can be run in several ways. We can write the name of the file without an extension in the command window, then by the command *Run* from the *menu* or by pressing button F5 (which also saves the modified file) we can run the file. The variables used in the script file appear in the global *workspace*, so their values can be seen from the command window. As an example, let us write a simple script file and run it. Let us create it: File–>New-Script (or m-file).

```
a=2
bscript=2*a+1
```

Let us save the file with the name *myscript.m*. Let us run it. If we run the m-file from the MATLAB™ command window, then in the *Current Folder* window we have to set the place, where has the m-file is to be saved (the *path* can also be set). We can see the result on the screen. The command *whos* ckecks that the variable bscript has been created.

A function can be created using the *function* m-file. The function may have one or more input and output parameters, and it uses local variables. The first row of the m-file contains the key word *function*. Let us create a function m-file.

```
function y=myfunction(x)
bfunction=3*x
y=bfunction+2
```

Let us save the file with the name *myfunction.m*. Call the function from the MATLAB™ command editor:

```
myfunction(4)
```

Using the command *whos* we can check that the global *workspace* does not contain the local variable bfunction. The functions can be embedded in each other in a file, but only the upper function block can be reached from outside.

From the menu, a number of debug devices are available to facilitate programming (Breakpoint, Step, Continue).

1.2 Introduction to the MATLAB™ *Control System Toolbox*

The *Control System Toolbox* extends the toolset of MATLAB™ so as to carry out the analysis, modeling, and design of control systems. The toolbox provides a repertory of algorithms and functions for these purposes, written mainly in the m-file format.

1.2.1 The Use of Functions of the Control System Toolbox

Consider a single-input, single-output (*SISO*), continuous-time, linear, time invariant (*LTI*) system defined by its transfer function (Fig. 1.2.):

Using MATLAB™, we can calculate the step response of a system with the transfer function $H(s) = \frac{2}{s^2+2s+4} = \frac{num}{den}$. The step response is defined as the output $y(t)$ of the system applying a unit step function input $u(t) = 1(t)$ assuming zero initial conditions.

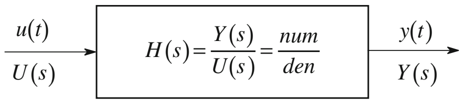

Fig. 1.2 An LTI system defined by its transfer function

The transfer function can be defined in MATLAB™ by its numerator and denominator as polynomials: $num = 2$, $den = s^2 + 2s + 4$. The polynomials are given by their coefficients put in a vector in descending order of s:

```
num=2
den=[1 2 4]
```

The step response can be displayed directly by the MATLAB™ step command (Fig. 1.3.):

```
step(num,den);
```

Note that it is equivalent to use the compact form

```
step(2,[1 2 4]);
```

The time scale is automatically selected by MATLAB™.

Expanding the above command by a left-hand side argument it is possible to store the values of the step response function (the output signal, the state variables and the time vector) in an array:

```
[y,x,t]=step(num,den)
```

or more simply if only the values of the output signal are requested:

```
y=step(num,den)
```

It has to be mentioned that when calling MATLAB™ functions the number of arguments in both sides may vary. The left-hand side output variables in the first activation of the step function are the output variables. y is the output of the step response, t gives the time points where it has been calculated, while x provides the so-called inner or state variables. Let us observe that in this case the output signal is not plotted. The values stored in a variable can be displayed by typing the name of the variable.

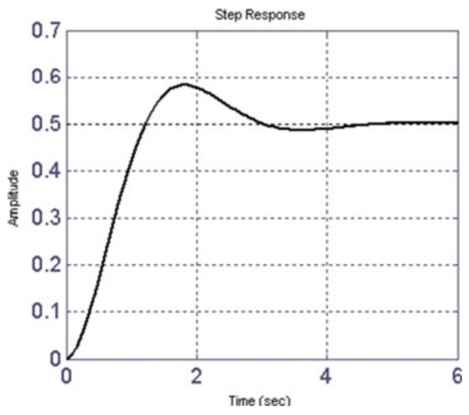

Fig. 1.3 Step response

```
y
```

The result is a column vector whose elements are the calculated values at the sampled points of the step response function. It can be seen from the figure that MATLAB™ has chosen the time interval $0 \le t \le 6$ based on the system's dynamical properties (zeros, poles). The sampling time applied by MATLAB™ can be calculated from the time interval and the size of the vector y:

```
n=length(y)
      n = 109
T=6/n
      T = 0.055
```

The calculated sampling time is thus $T{=}6/109{=}\ 0.055$ s.

The **help** command shows further possible forms of using the step command:

```
help step
```

It can be seen, then, that there are other ways to use the **step** command. E.g. if the time interval $0 \le t \le 10$ and the sampling time $T = 0.1$ are explicitly selected by

```
t=0:0.1:10
```

the following form can be employed:

```
y=step(num,den,t)
```

The output vector can now be displayed with the plot command:

```
plot(t,y);
```

or adding the grid option to support the easy reading of the plot

```
plot(t,y),grid on;
```

As far as the visualization is concerned, the plot command uses linear interpolation between the calculated samples. To avoid this interpolation, the command

```
plot(t,y,'.');
```

displays only the calculated samples.

The obtained y vector can be used for further calculations. The maximum of the step response (more precisely the largest calculated sample) can be determined by command max:

```
ym=max(y)
    ym = 0.5815
```

The steady state value of the step response is obtained by the dcgain command:

```
ys=dcgain(num,den)
    ys = 0.5
```

and the percentage overshoot of the output is

```
yovrsht=(ym-ys)/ys*100
    yovrsht = 16.2971
```

1.2.2 LTI Model Structures (sys Forms)

In order to simplify the commands the Control System Toolbox can also use data-structures. There are three basic forms to describe linear time-invariant (*LTI*) systems in MATLAB™:

Transfer function form: $H_{\text{tf}}(s) = \dfrac{s^m + b_{m-1}s^{m-1} + \ldots + b_2 s^2 + b_1 s + b_0}{a_n s^n + a_{n-1}s^{n-1} + \ldots + a_2 s^2 + a_1 s + a_0} = \dfrac{2}{s^2+3s+2}$

Zero-pole-gain form: $H_{\text{zpk}}(s) = k\dfrac{(s-z_1)(s-z_2)\ldots(s-z_m)}{(s-p_1)(s-p_2)\ldots(s-p_n)} = \dfrac{2}{(s+1)(s+2)}$

State space form: $\begin{array}{l}\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{b}u \\ y = \mathbf{c}^{\mathrm{T}}\mathbf{x} + du\end{array}$; $\mathbf{A} = \begin{bmatrix} -3 & -1 \\ 2 & 0 \end{bmatrix}$; $\mathbf{b} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$; $\mathbf{c}^{\mathrm{T}} = [0 \quad 1]$; $d = 0$

Using the MATLAB commands tf, zpk and ss, the *LTI* system can be given in an *LTI* data-structure.

Defining the LTI sys structure

Let the transfer function of the system be $H(s) = \frac{2}{s^2+3s+2} = \frac{2}{(s+1)(s+2)}$.

The transfer function form is given as

```
num=2
den=[1, 3, 2]
H=tf(num,den)
```

The transfer function is:

```
          2
-------------
s^2 + 3 s + 2
```

or directly

```
Htf=tf(2,[1, 3, 2])
```

Defining the zero-pole-gain form:

```
Hzpk=zpk([],[-1, -2],2)
```

The zero/pole/gain form is:

```
     2
----------
(s+1)(s+2)
```

The state space form can be given by

```
A=[-3, -1; 2, 0]; B=[1; 0]; C=[0, 1]; D=0;
Hss=ss(A,B,C,D)
```

The models can be converted into each other:

```
H=zpk(H)
H=ss(H)
H=tf(H)
```

The *LTI* models possess several properties. These properties can be obtained using the command get.

```
get(Htf)
get(Hzpk)
get(Hss)
```

The *LTI sys* model parameters can be obtained by using the commands tfdata, zpkdata, ssdata. The *LTI* data structure can be used also in case of *Multi-Input Multi-Output* (*MIMO*) systems, therefore it stores some parameters in *cell array* form. The parameters can be accessed in vector format if we put the flag 'v' in the command.

```
[num,den]=tfdata(H,'v')
    num = 0     0     2
    den = 1     3     2
[z,p,k]=zpkdata(H,'v')
[A,B,C,D]=ssdata(H)        % here flag 'v' is omitted.
```

Symbolic data entry

The transfer function can be defined even more simply by symbolic data entry. Let us give the s variable of the LAPLACE transform by a special command:

```
s=zpk('s')
H=1/(s^2+3*s+2)
```

The transfer function appears in zero-pole-gain form.
If the variable s is given in the form

```
s=tf('s')
```

then the transfer functions defined with this variable will be obtained in tf form, i.e. in polynomial-polynomial form.

Arithmetic operations can be applied to data given in *LTI sys* structures as well. The most frequently used operations are: +, -, *, /, \, ', inv, ^. For example the resulting transfer function of a closed loop system can be calculated by the following symbolic relation:

```
Hcl=H/(1+H)
```

The possible simplifications are executed by the command minreal.

```
Hcl=minreal(Hcl
```

Among the *LTI sys* structures a hierarchical sequence order is defined: tf ->zpk ->ss. If in a command or in a calculation the operands are *LTI* models of different forms, then the result is always in the form which is higher in the hierarchy. For example, the result of

```
Htf*Hzpk
```

is obtained in the zpk form.

1.2.3 Time Domain Analysis

The *Control System Toolbox* contains several commands that provide basic tools for time domain analysis. Let us analyse the system

$$H(s) = \frac{2}{s^2 + 2s + 4}$$

```
H=2/(s^2+2*s+4)
```

Define the time vector to be

```
t=0:0.1:10;
```

Step response: All previously discussed versions of the `step` command can be used. Additionally the following forms can also be applied:

```
step(H);
[y,t,x]=step(H);
```

Let us remark that when using the *LTI sys* structure the order of the output parameters differ from the order when using the (`num, den`) polynomial form:

```
[y,x,t]=step(num,den);
```

Impulse response: The impulse response is the response of the system to a DIRAC delta input.

```
impulse(H);
yi=impulse(H,t);
plot(t,yi)
```

The system's behaviour can also be analysed for nonzero *initial conditions*. Nonzero initial conditions can only be taken into account if state space models are used. Accordingly, to apply the `initial` command, the system has to be transformed into a state space representation.

```
H=ss(H)
x0=[1, -2]
[y,t,x]=initial(H,x0);
plot(t,y),grid on
```

Note that these commands yield `x` as a matrix having as many columns as dictated by the number of the state variables (two in this case), and as many rows as dictated by the time instants (109 in this case). Just to check:

```
size(x)
   ans = 109   2
```

The state trajectory can also be calculated and plotted. The first column of `x` contains the first state variable, while the second state variable will show up in the second column. The notation ':' means that all elements of a vector are chosen. The state trajectory plots a state variable versus the other one.

```
x1=x(:,1); x2=x(:,2);
plot(x1,x2)
```

Output response to an arbitrary input: The output can be calculated as a response for any input signal.

Let us determine the output signal if the input is the following sinusoidal signal: $u(t) = 2\sin(3t)$

```
usin=2*sin(3*t);
ysin=lsim(H,usin,t);
```

Plot the input and the output in the same diagram (input: red, output: blue).

```
plot(t,usin,'r',t,ysin,'b'), grid;
```

1.2.4 Frequency Domain Analysis

The system's behaviour can also be analysed in the frequency domain.

The Bode diagram can be calculated by the `bode` command. There are several ways to use this command. The gain and phase shift of the system can be calculated at a given frequency. Let us calculate the gain and the phase shift of system H at the frequency $\omega = 5$:

```
w=5;
[gain,phase]=bode(H,w);
```

The result is `gain` $=$ `0.0860`, `phase` $=$ `−154.5367`.

These calculations can be repeated for several frequencies. The command `bode` can be activated to calculate the absolute value and the phase angle of the frequency function at several frequencies by one call. The Bode diagram can be displayed (see Fig. 1.4) by

```
bode(H),grid
```

In this case, MATLAB™ automatically calculates a frequency vector based on the system dynamics.

The frequency scale is logarithmic, as in this case a big frequency range can be taken into account. The calculations can be repeated for a selected frequency range. A logarithmic frequency vector can be generated by the `logspace` command

```
w=logspace(-1,1,200);
```

This command creates 200 logarithmically equidistant frequency points between $10^{-1} = 0.1$ and $10^{1} = 10$.

The values of the Bode diagram can be calculated at these frequency points as

```
[gain,phase]=bode(H,w);
```

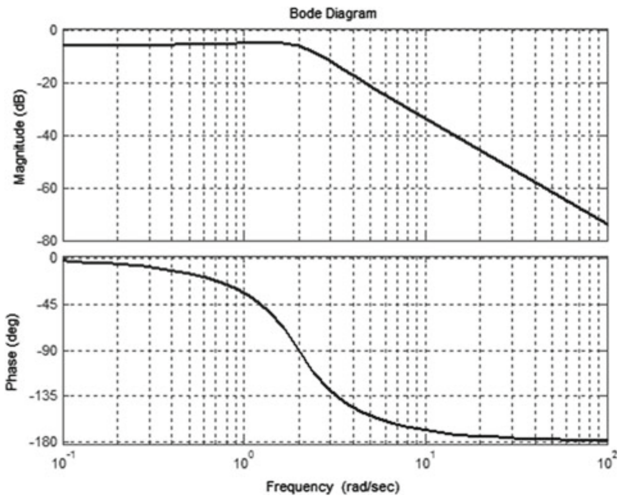

Fig. 1.4 BODE diagram

Since the *LTI* structure can be used also in the case of *MIMO* systems, the parameters *gain* and *phase* are given in 3 dimensional array format. This can be transformed to vector form by the operation (:). Let us compare the following two commands:

```
gain
gain(:)
```

NYQUIST diagram: At a given frequency the gain and the phase angle provide a vector (a point) in the complex plane. These vectors are plotted in the complex plane and their points are connected while the frequency is changing in a given range (Fig. 1.5).

The NYQUIST diagram is produced by the command

```
nyquist(H);
```

The command margin evaluates the main characteristics of the frequency function. It is an important tool to check the stability margins of a system.

```
margin(H);
```

(Frequency functions will be analyzed in more detail in Sect. 2.3.)

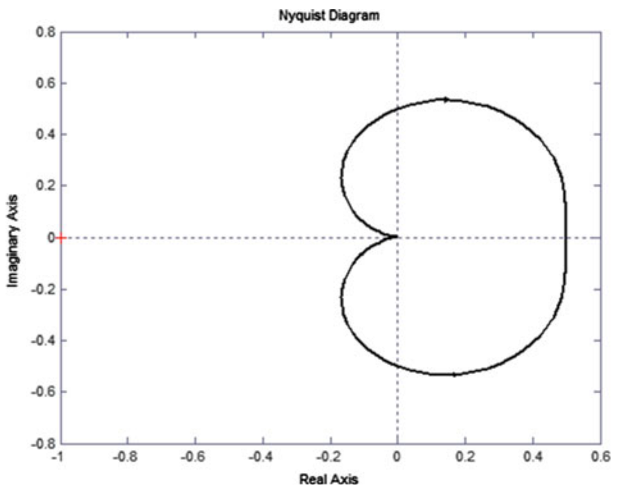

Fig. 1.5 Nyquist diagram

1.2.5 Zeros, Poles

The roots of the denominator of the transfer function are the poles of the system.

```
[num,den]=tfdata(H,'v');
poles=roots(den);
```

The roots of the numerator of the transfer function are the zeros of the system.

```
zeros=roots(num);
```

The zeros and the poles can be immediately obtained from the `zpk` model:

```
[z,p,k]=zpkdata(H,'v');
```

The zeros and the poles can be plotted in the complex plane:

```
subplot(111);
pzmap(H);
```

The `damp` command lists all the poles and (in the case of complex pole-pairs) the natural frequencies and damping factors:

```
damp(H);
```

The gain of the system (its steady state value in the case of a step input) can be calculated

```
K=dcgain(H);
```

1.2.6 LTI *Viewer*

A linear system can be analysed in detail by *LTI Viewer*, which is a graphical user interface for analysing the system response in the time domain and in the frequency domain. The systems can be analysed from the menu or using the right mouse button:

```
ltiview
```

or

```
ltiview('bode',H);
```

To demonstrate the application of *LTI Viewer*, we will first analyse a so called *first-order lag element* which can be described by a first-order differential equation. Its transfer function, differential equation, and step response can be obtained by the following relations.

First-order lag element:
The transfer function is the ratio of the LAPLACE transforms of the output and the input signals.

$$\frac{Y(s)}{U(s)} = H(s) = \frac{A}{1+sT}, \text{ or } (1+sT)Y(s) = AU(s).$$

Hence the differential equation is

$$y(t) + T\dot{y}(t) = Au(t)$$

and its solution for a unit step input is

$$y(t) = A\left(1 - e^{-t/T}\right)1(t).$$

The step response can be obtained in several ways using MATLAB™.
Possibilities for simulation include the following:

a. by solving the differential equation:

```
T=10; A=5; t=0:0.1:50; y1=A*(1-exp(-t/T)); plot(t,y1);
grid;
```

b. on the basis of the transfer function:

```
y2=step(A,[T 1],t); plot(t,y1,t,y2)
```

c. using the *LTI* description:

```
s=tf('s'); P1=A/(1+T*s); y3=step(P1,t);
plot(t,y1,t,y2,t,y3)
```

d. using the block orientated SIMULINK™ program (see Sect. 1.3.).

It can be seen that the curves of the step responses calculated in the three different ways coincide.

With *LTI Viewer* a system can be imported and then analysed with its different characteristic functions. Let us consider the previous *LTI* model:

```
P1=A/(1+T*s)
```

The transfer function is

```
     5
--------
10 s + 1
```

```
ltiview
```

Select File/Import
Import from Workspace
Select P1
OK
RightClick, Select ‘Plot Types’:

Step
Impulse
Bode
Nyquist
Pole/Zero

…

Checking the points of the curves: Left Click on the curve

Analysing several systems in parallel:

```
P2=5/(1+20*s), P3=5/(1+50*s);
```

Back to the *LTI* viewer:
Import P1, P2, P3
Right Click, Select Systems: automatic order of the colours: blue, green, red

The system dynamics can be seen for the different time constants in the time- and the frequency domain and on the complex plane.

Let us now analyse the behaviour and characteristic functions of the so called second-order oscillating element, which can be described by a second-order differential equation.

```
P4=1/(s^2+s+1), P5=1/(s^2+0.5*s+1);ltiview
```

Let us add a zero to the second-order system and analyse its effect on the characteristic functions of the system.

```
s=zpk('s');
P6=8/6*(s+6)/(s+2)/(s+4),P7=8/3*(s+3)/(s+2)/(s+4);
ltiview
P8=8*(s+1)/(s+2)/(s+4);P9=-8/3*(s-3)/(s+2)/(s+4);
ltiview
```

1.3 SIMULINK™

SIMULINK™ is a graphics software package supporting block-oriented system analysis. SIMULINK™ has two phases, *model definition* and *model analysis*. First a model has to be defined, then it can be analysed by running a simulation. SIMULINK™ represents dynamical systems with block diagrams. Defining a system is much like drawing a block diagram. Instead of drawing the individual blocks, blocks are copied from libraries of blocks. The standard block library is organized into several subsystems, grouping blocks according to their behaviour. Blocks can be copied from these or any other libraries or models into your model. The SIMULINK™ block library can be opened from the MATLAB™ command window by entering the command `simulink`. This command displays a new window containing icons for the subsystem blocks. To construct your model, select **New** from the **File** menu of SIMULINK to open a new empty window in which you can build your model. Open one or more libraries and drag some blocks into your active window, then release the button. To connect two blocks use the left mouse button to click on either the output or input port of one block, drag to the other block's input or output port to draw a connecting line, and then release the button. By clicking on the block with the right button you can duplicate it. The blocks can be increased, decreased, and rotated. Open the blocks by double clicking to change some of their internal parameters. Save the system by selecting **Save** from the **File** menu. Figure 1.6. shows the SIMULINK™ diagram of a control system.

Run a simulation by selecting *Start* from the *Simulation* menu or by clicking on the *Run* icon (►). Simulation parameters can also be changed. You can monitor the behavior of your system with a *Scope* or you can use the *To Workspace* block to

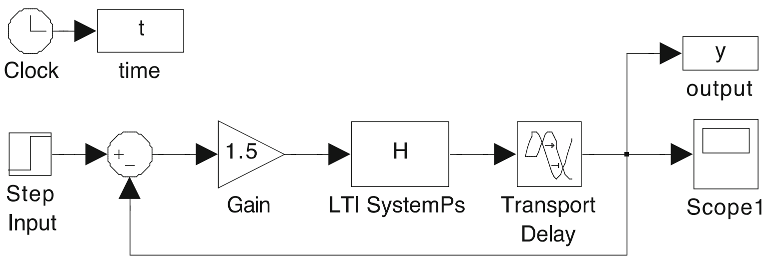

Fig. 1.6 SIMULINK™ diagram of a control system

send data to the MATLAB™ workspace and perform MATLAB™ functions (e.g. plot) on the results. Parameters of the blocks can be referred also by variables defined in MATLAB™. Simulation of SIMULINK™ models involves the numerical integration of sets of ordinary differential equations. SIMULINK™ provides a number of integration algorithms for the simulation of such equations. The appropriate choice of method and the careful selection of simulation parameters are important considerations for obtaining accurate results. To get yourself familiarized with the flavour of the options offered by SIMULINK™ consider the following example.

Create a new file and copy various blocks (Fig. 1.6). The block parameters should then be changed to the required value. Change the *Simulation –>Parameters–>Stop* time parameter to 50 from the menu. SIMULINK™ uses the variables defined in the MATLAB™ workspace.

$H(s)$: *Control System Toolbox –>LTI system* : H
Creating difference: *Simulink–>Math–>Sum*: +–
Dead-time, delay: *Simulink–>Continuous–>Transport Delay*: 1
Gain: *Simulink–>Math–>Gain*: 1.5
Step input: *Simulink–>Sources–>Step*
Scope: *Simulink–>Sinks–>Scope*
Clock: *Simulink–>Sources–>ClockOutput, time*: *Simulink–>Sinks–>To Workspace*: t, y

The result can be analysed directly by the *Scope* block or it can be sent back to the MATLAB™ workspace by the *To Workspace* output block. The results can be further processed and displayed graphically. Change the *Gain* parameter between 0.5 and 2. Determine the critical value of the gain, where steady oscillations do appear in the control system.

The results of the simulation can be sent to the MATLAB™ workspace through the *Scope* block as well. Let us set the parameters of the graphical window of the *Scope* as follows:

Under the '*properties*' menu

Data history: *Save data to workspace –>Variable name*: ty (tu for the control signal)

Matrix format

So the values of the time vector t and the output vector y can be obtained easily after the simulation, and then some properties (as e.g. the overshoot, settling time, maximum value of the control signal, etc.) can be determined.

```
t=ty(:,1)
y=ty(:,2)
plot(t,y),grid on
```