Signal and System Theory II, SS 2011

Prof. J. Lygeros

Automatic Control Laboratory
ETH Zurich, ETL I28
8092 Zurich

Introduction to MATLAB (Based on Matlab Manual)

1.1 What is MATLAB?

The name MATLAB stands for "MATrix LABoratory" and was originally designed as a tool for doing numerical computations with matrices and vectors. It has since grown into a high-performance language for technical computing. MATLAB, integrating computation, visualization, and programming in an easy-to-use environment, allows easy matrix manipulation, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs in other languages. Typical uses include:

- Math and Computation
- Modeling and Simulation
- Data Analysis and Visualization
- Application Development
- Graphical User Interface development

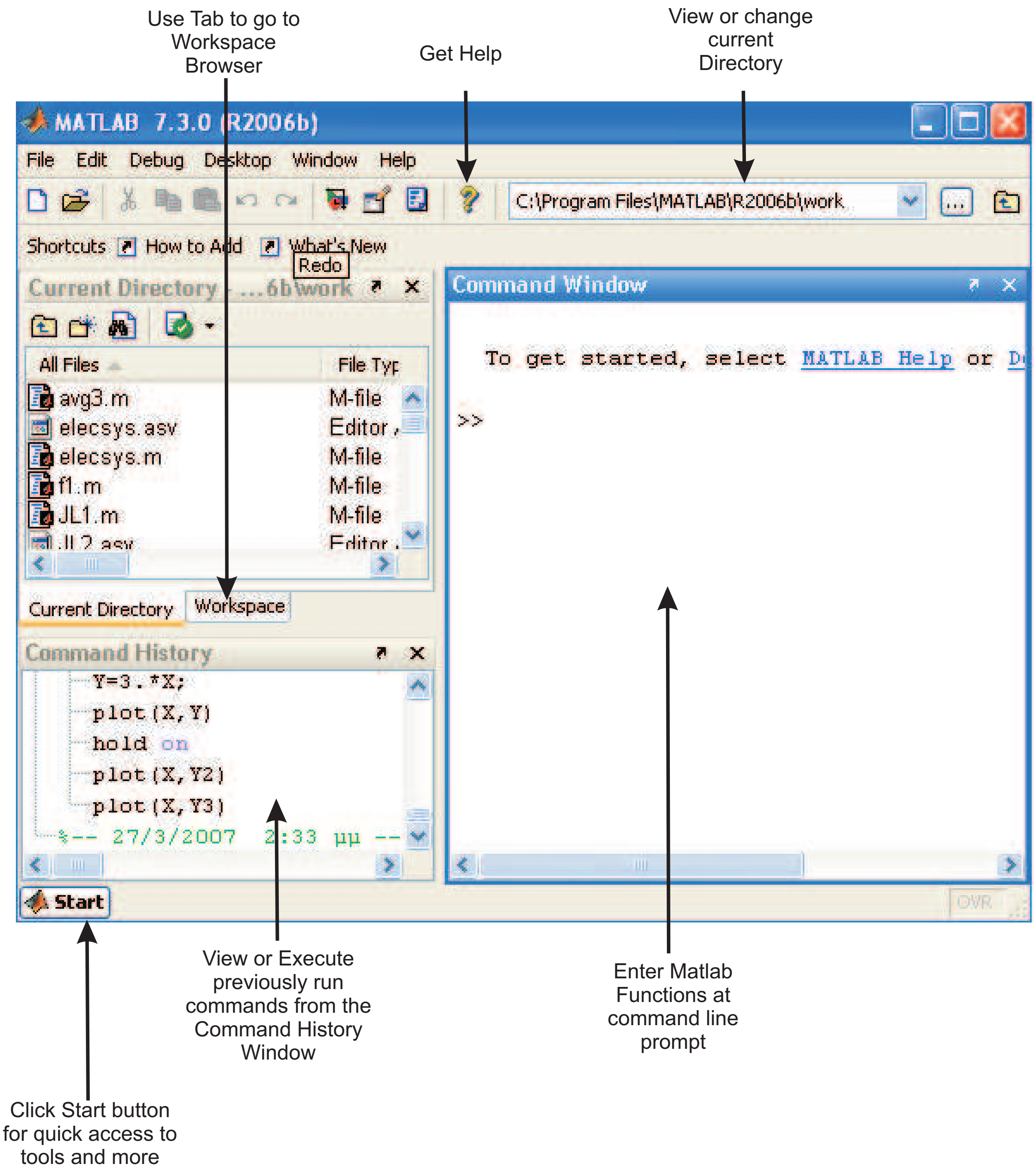
1.2 Getting Started

1.2.1 Window Layout

The first time you start MATLAB, the desktop appears with the default layout, as shown in Figure 1. The following tools are managed by the MATLAB desktop:

- **Command Window:** Run MATLAB statements.
- **Current Directory Browser:** To search for, view, open, find, and make changes to MATLAB related directories and files, use the MATLAB Current Directory browser.
- **Command History:** Displays a log of the functions you entered in the Command Window, copy them, execute them, and more.
- **Workspace Browser:** Shows the name of each variable, its value, and the Min and Max calculations, which MATLAB computes using the min and max functions, and updates automatically.

In case that the desktop does not appear with the default layout, you can change it by the menu `Desktop` → `Desktop Layout` → `Default`.


Use Tab to go to Workspace Browser
Get Help
View or change current Directory
MATLAB 7.3.0 (R2006b)
File Edit Debug Desktop Window Help
C:\Program Files\MATLAB\R2006b\work
Shortcuts How to Add What's New
Redo
Current Directory - ...6b\work
Command Window
All Files
File Typ
avg3.m M-file
elecsys.asv Editor
elecsys.m M-file
f1.m M-file
JL1.m M-file
JL2.asv Editor
Current Directory
Workspace
Command History
Start
OVR
View or Execute previously run commands from the Command History Window
Enter Matlab Functions at command line prompt
Click Start button for quick access to tools and more

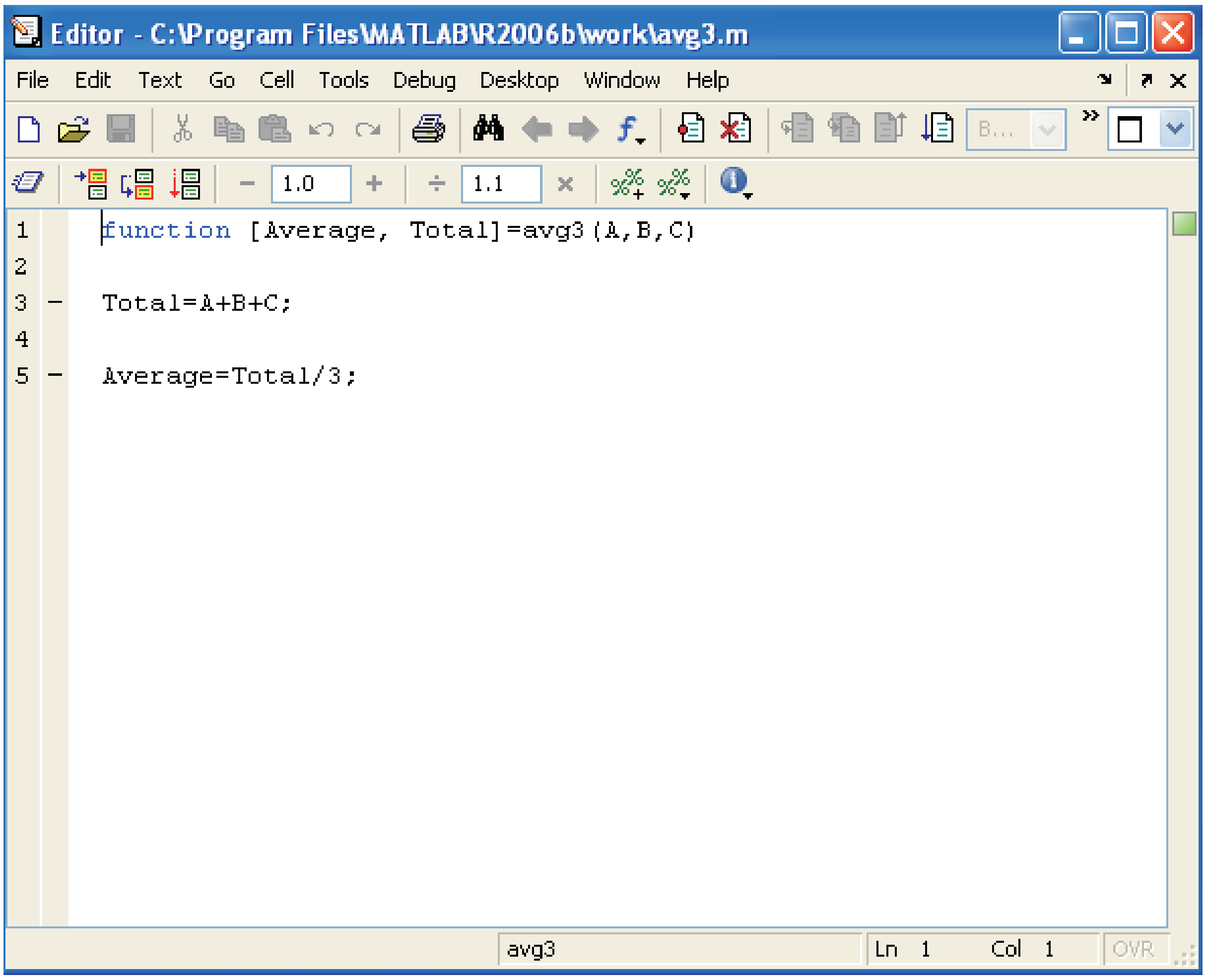
```
  To get started, select MATLAB Help or D

>>
```

```
Y=3.*X;
plot(X,Y)
hold on
plot(X,Y2)
plot(X,Y3)
%-- 27/3/2007  2:33 μμ --
```

Figure 1: Matlab Window (default layout)

1.2.2 Editor

MATLAB editor (Figure 2) can be used to create and debug M – files, which are programs you write to run MATLAB functions. A M – file is a text file that contains a sequence of MATLAB commands; the commands contained in a script file can be run, in order, in the MATLAB command window simply by typing the name of the file at the command prompt. M – files are very useful when you use a sequence of commands over and over again, in many different MATLAB sessions and you do not want to manually type these commands at the command prompt whenever you want to use them.


Editor - C:\Program Files\MATLAB\R2006b\work\avg3.m
File Edit Text Go Cell Tools Debug Desktop Window Help
1.0
1.1
avg3
Ln 1 Col 1
OVR

```
function [Average, Total]=avg3(A,B,C)

Total=A+B+C;

Average=Total/3;
```

Figure 2: Matlab Editor

You can run a script or a function that does not require an input argument directly from the Editor/Debugger either by pressing $F5$ or selecting Run from the Debug menu. If you want to run a part of the script then you can highlight the part of the script you want to evaluate and then press $F9$ to run it. In all cases the results are shown in Command Window.

1.2.3 MATLAB Help

MATLAB has an extensive help system built into it, containing detailed documentation and help information on all of the commands and functions of MATLAB. There are two different ways to obtain help for MATLAB:

Command Line

- **HELP:** HELP FUN displays a description of and syntax for the function FUN. (e.g. help plot)
- **DOC:** DOC FUN displays the HTML documentation for the MATLAB function FUN. (e.g. doc help)

Help Browser Another source of help is the MATLAB help browser. You can invoke the MATLAB help browser by typing, helpbrowser, at the MATLAB command prompt, clicking on the help button, or by selecting *Start* → *MATLAB* → *Help* from the MATLAB desktop.

1.3 Variables

The simplest way to use MATLAB is for arithmetic operations. The basic arithmetic operators are $+, -, /, *$ and $\wedge$ (power). These operators can be used in conjunction with brackets (). As with all programming languages special care should be given on how a mathematic expression is written. For example, the result of the expression $5 + 10/2 * 3$ is 20 and corresponds in the expression $5 + (10/2) * 3$ and not in the expression $5 + 10/(2 * 3)$. Generally, Matlab works according to the priorities:

(1) quantities in brackets

(2) powers

(3) $*, /$ working left to right

(4) $+, -$ working left to right

For example:

$$\begin{aligned}
3 + 5/2 * 4 - 2\hat{}3 + (5 * 2) &= 3 + 5/2 * 4 - 2\hat{}3 + 10 \\
&= 3 + 5/2 * 4 - 8 + 10 \\
&= 3 + 2.5 * 4 - 8 + 10 \\
&= 3 + 10 - 8 + 10 \\
&= 15
\end{aligned}$$

MATLAB always stores the result of a calculation in a varable named *ans*, but it is possible to use our own names to store numbers:

```
>> x=5+2^2

x =

     9
```

and then we can use these variables in other calculations:

```
>> y=2*x

y =

    18
```

These are examples of **assignment statements**: values are assigned to variables. **Each variable must be assigned a value before it may be used on the right of an assignment statement**.

One often does not want to see the result of intermediate calculations. This can be done by terminating the assignment statement or expression with semi–colon:

```
>> x=5+2^2;
>> y=2*x;
>> z=x^2+y^2

z =

   405
```

You can also assign pieces of text to variables, not just numbers. You do this using single quotes (not double quotes — single quotes and double quotes have different uses in MATLAB) around the text you want to assign to a variable. For example:

```
>> w = 'Goodmorning';
>> w

w =

Goodmorning
```

1.3.1 Variable Names

There are some specific rules for what you can name your variables, so you have to be careful.

- Only use primary alphabetic characters (i.e., "A-Z"), numbers, and the underscore character in your variable names.
- You cannot have any spaces in your variable names, so, for example, using "this is a variable" as a variable name is not allowed (in general, you can use the underscore character wherever you would use space to string words together in your variable name).
- MATLAB is **case sensitive**. What this means for variables is that the same text, with different mixes of capital and small case letters, will not be the same variables in MATLAB. For example, "VaRIAbLe", "variable", "VARIABLE" and "variablE" would all be considered distinct variables in MATLAB.

1.4 Matlab Functions

1.4.1 Simple Built – in Functions

- Trigonometric Functions:`sin, cos, tan` (their arguments should be in radians).
- Inverse Trigonometric Functions:`asin, acos, atan` (the result is in radians).
- Square root function:`sqrt`.
- Exponential function:`exp`.

- Natural logarithm:textttlog.
- Logarithm base 10:textttlog10.

1.4.2 M – file functions

You add new functions to the MATLAB vocabulary by expressing them in terms of existing functions. The existing commands and functions that compose the new function reside in an M-file. A line at the top of a function M-file contains the syntax definition. The name of a function, as defined in the first line of the M-file, should be the same as the name of the file without the .m extension.

The main steps to follow when defining a Matlab function are:

(1) Decide on a name for the function, making sure that it does not conflict with a name that is already used by Matlab.

(2) The first line of the file must have the format:

function [list of outputs] = FunctionName(list of inputs)

(3) Include the code that defines the function.

A simple example of a function m – file is shown below:

```
function [Average, Total]=avg3(A,B,C)
Total=A+B+C;
Average=Total/3;
```

If you write in command window:

```
>> [K,L]=avg3(3,5,7)
```

the result is:

```
K =

     5


L =

    15
```

1.5 Matrices

In MATLAB, and in linear algebra, numeric objects can be categorized simply as matrix: Both scalars and vectors can be considered a special type of matrix. For example a scalar is a matrix with a row and column dimension of one (1-by-1 matrix). And a vector is a matrix with one row and n columns, or n rows and one column. Most calculations in MATLAB are done with “matrices”. Hence the name MATrix LABoratory.

1.5.1 Creating Matrices in MATLAB

In MATLAB matrices are defined inside a pair of square brackets ([]). Punctuation marks of a comma (,), and semicolon (;) are used as a row separator and column separator, respectfully (you can also use a space as a row separator, and a carriage return (the enter key) as a column separator as well.). Use the examples below to check how vectors and matrices can be created in MATLAB.

```
>> A=[1,4,7]

>> B=[1;4;7]

>> C=[1 2 3;3 2 1;4 5 6;6 5 4]
```

You can also combine different vectors and matrices together to define a new matrix. For example:

```
>> D=[A A]

>> E=[B B]

>> F=[C C]

>> G=[C;C]
```

1.5.2 Colon Operator

The colon operator allows you to create an incremental vector of regularly spaced points by specifying *startvalue* : *increment* : *stopvalue*. Instead of an incremental value you can also specify a decrement as well. Check the examples:

`>> A=[0:10:200]` and `>> B=[100:-10:-100]`

1.5.3 Indexing Into a Matrix

Once a vector or a matrix is created you might needed to extract only a subset of the data, and this is done through indexing. Each element of a matrix is indexed according to which row and column it belongs to. The entry in the ith row and jth column is denoted mathematically by $A_{i,j}$ and in Matlab by $A(i, j)$. So for the matrix:

`>> C=[1 2 3;4 5 6;7 8 9;10 11 12]`

the element in 3rd row and the 2nd column is:

```
>> C(3,2)

ans =

     8
```

You can also extract any continuous subset of a matrix, by referring to the row range and column range you want. In the following examples we extract i) the 3rd column, ii) the 2nd and 3rd columns, iii) the 4th row, and iv) the central 2×2 matrix.

```
>> C(:,3)

>> C(:,2:3)

>> C(4,:)

>> C(2:3,2:3)
```

1.5.4 Matrix Operations

- Addition: $>> C = A + B$
- Subtraction: $>> C = A - B$
- Multiplication: $>> C = A * B$
- Element-by-element Multiplication: $C = A. * B$
- Transposing: $>> C = A'$

Note: Matrices must have compatible dimensions.

1.5.5 Matrix Functions

- The **lu** function expresses a matrix A as the product of two essentially triangular matrices, one of them a permutation of a lower triangular matrix and the other an upper triangular matrix. The factorization is often called the LU, or sometimes the LR, factorization. $[L, U] = lu(A)$ returns an upper triangular matrix in U and a permuted lower triangular matrix in L such that $A = L * U$. Return value L is a product of lower triangular and permutation matrices.

- The **qr** function performs the orthogonal-triangular decomposition of a matrix. This factorization is useful for both square and rectangular matrices. It expresses the matrix as the product of a real complex unitary matrix and an upper triangular matrix. $[Q, R] = qr(A)$ produces an upper triangular matrix R of the same dimension as A and a unitary matrix Q so that $A = Q * R$. For sparse matrices, Q is often nearly full. If $[m\ n] = size(A)$, then Q is $m \times m$ and R is $m \times n$.

- $[V, D] = eig(A)$ produces matrices of eigenvalues (D) and eigenvectors (V) of matrix A, so that $A * V = V * D$. Matrix D is the canonical form of A a diagonal matrix with A's eigenvalues on the main diagonal. Matrix V is the modal matrix its columns are the eigenvectors of A.

- The **svd** command computes the matrix singular value decomposition. $s = svd(X)$ returns a vector of singular values. $[U, S, V] = svd(X)$ produces a diagonal matrix S of the same dimension as X, with nonnegative diagonal elements in decreasing order, and unitary matrices U and V so that $X = U * S * V'$.

1.6 Loops

There are occasions that we want to repeat a segment of code a number of different times. In such cases it is convenient to use loop structures. In MATLAB there are three loop structures:

- For Loop

  The general syntax is:

```
for variable = expression
    statement
    ...
    statement
end
```

The columns of the expression are stored one at a time in the variable while the following statements, up to the end, are executed.In practice, the expression is almost always of the form scalar:scalar , in which case its columns are simply scalars.The scope of the for statement is always terminated with a matching end.

- While Loop

  The general format is

```
while expression
    statement
    ...
    statement
end
```

  while repeats statements an indefinite number of times. The statements are executed while the real part of expression has all nonzero elements. Expression is usually of the form

  expression relation-operator expression

  where relation-operator is $==, <, >, <=, >=$ or $\;\;=$.

- If. . .then. . .else

  MATLAB evaluates the expression and, if the evaluation yields logical 1 (true) or a nonzero result, executes one or more MATLAB commands denoted here as statements.When you are nesting ifs, each if must be paired with a matching end.When using elseif and/or else within an if statement, the general form of the statement is

```
if expression1
    statements1
elseif expression2
    statements2
else
    statements3
end
```

1.7 System Representation

1.7.1 State Space

The function:
$sys = ss(A, B, C, D)$
creates the continuous-time state-space model:

$$\begin{aligned}\dot{x} &= Ax + Bu \\ y &= Cx + Du\end{aligned}$$

The output *sys* is an SS model that stores the model data.

1.7.2 Transfer Function

The function:
$sys = tf(num, den)$
creates a continuous-time transfer function with numerator and denominator specified by *poly*; with coefficients stored in the vectors *num* and *den*. The output *sys* is a TF object storing the transfer function data.

1.7.3 Convertions

- $tf2ss$ converts the parameters of a transfer function representation of a given system to those of an equivalent state-space representation.

  $[A, B, C, D] = tf2ss(b, a)$

  returns the A, B, C, and D matrices of a state space representation for the single-input transfer function

  $$H(s) = \frac{B(s)}{A(s)} = C(sI - A)^{-1}B + D \tag{1}$$

- $ss2tf$ converts a state-space representation of a given system to an equivalent transfer function representation.

  $[b, a] = ss2tf(A, B, C, D, iu)$ returns the transfer function

  $$H(s) = \frac{B(s)}{A(s)} = C(sI - A)^{-1}B + D \tag{2}$$

  of the system

  $$\begin{aligned}\dot{x} &= Ax + Bu \\ y &= Cx + Du\end{aligned}$$

  from the iu-th input.

1.8 Model Simulation

The function *lsim* simulates the (time) response of continuous or discrete time linear systems (LTI) to arbitrary inputs. When invoked without left-hand arguments, *lsim* plots the response on the screen. Thus,
$lsim(sys, u, t)$
produces a plot of the time response of the LTI model *sys* to the input time history t, u. The vector t specifies the time samples for the simulation and consists of regularly spaced time samples.

$t = 0 : dt : Tfinal$

The matrix u must have as many rows as time samples ($length(t)$) and as many columns as system inputs. Each row $u(i,:)$ specifies the input value(s) at the time sample $t(i)$.

When invoked with left-hand arguments,

$[y,t] = lsim(sys,u,t)$

$[y,t,x] = lsim(sys,u,t) \rightarrow$ for state-space models only

$[y,t,x] = lsim(sys,u,t,x0) \rightarrow$ with initial state x_0

return the output response y, the time vector t used for simulation, and the state trajectories x (for state-space models only).

There are also Matlab functions for specific type of inputs:

- Step input: $step(sys,t)$ or $[y,t,x] = step(sys)$
- Impulse input: $impulse(sys,t)$ or $[y,t,x] = impulse(sys)$

1.8.1 Ordinary Differential Equations Solver

In some cases, a model is described by a system of differential equations instead of a State Space or a Transfer Function. For such cases an *odesolver* can be used to calculate the system response. The most common syntax is:

$[T,Y] = solver(odefun, tspan, y0)$

(where solver is one of ode45, ode23, ode113, ode15s, ode23s, ode23t, or ode23tb.)

with

$tspan = [t0\ tf]$

and integrates the system of differential equations $y' = f(x,t)$ from time t_0 to t_f with initial conditions y_0. $odefun$ is a function handle. Function $f = odefun(t,y)$, for a scalar t and a column vector y, must return a column vector f corresponding to $f(x,t)$. Each row in the solution array Y corresponds to a time returned in column vector T.

An *odesolver* can be used for example for the carrier pendulum shown in Figure 3. The code for this system and the Matlab Function (odefun) for the carrier pendulum differential equations are shown below.

```
clear all
t_init=0;T_s=0.1;t_fin=40;options=[]; %simulation interval & sampling
row_num=4; %state vector dimension
x_system=zeros(row_num,ceil((t_fin-t_init)/T_s)); %system response
cnt=1; %counter
x_ode=zeros(row_num,1);x_ode(3)=50*pi/180;
x_system(:,1)=x_ode; %initial conditions specifications
for t=t_init:T_s:t_fin-T_s...
cnt=cnt+1;... %counter increment
F_l=0;... %input specification
[t_ode,x_ode]=ode45('carrierpendulum',[t
t+T_s],[x_ode],options,F_l);...
szx_ode=size(x_ode);x_ode=x_ode(szx_ode(1),:)';x_system(:,cnt)=x_ode;...
end figure(1) t=[t_init:T_s:t_fin];
subplot(2,2,1),plot(t,x_system(1,:)),title('carrier position')
subplot(2,2,2),plot(t,x_system(2,:)),title('carrier velocity')
subplot(2,2,3),plot(t,x_system(3,:)),title('pendulum angle')
```

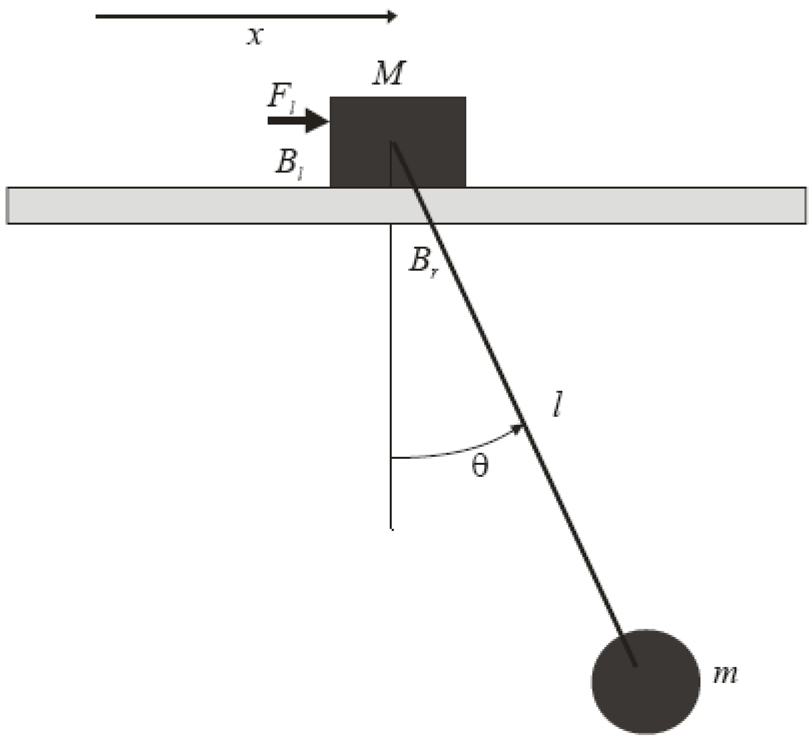


Figure 3: Carrier Pendulum

```
subplot(2,2,4),plot(t,x_system(4,:)),title('pendulum angular
velocity')

function xprim = carrierpendulum(t,x,options,F_l)
M=1;m=1;l=1;B_l=0.3;B_r=0.3;g=10; xprim=zeros(size(x));
xprim(1)=x(2);
xprim(2)=(F_l-B_l*x(2)+m*l*sin(x(3))*(x(4))^2 + ...
m*g*cos(x(3))*sin(x(3))+B_r/l*cos(x(3))*x(4))/...
((M+m)-m*(cos(x(3)))^2);
xprim(3)=x(4);
xprim(4)=(-m*g*l*sin(x(3))-B_r*x(4)-m*l*cos(x(3))*xprim(2))/...
(m*l^2);
```

1.9 Plots

The most basic plotting command in MATLAB is the plot command. The plot command, when called with two same-sized vectors X and Y, makes a two-dimensional line plot for each point in X and its corresponding point in Y: the numbers in X are on the abscissa (x-axis) and the numbers in Y are on the ordinate (y-axis). In other words, it will draw points at $(X(1), Y(1))$, $(X(2), Y(2))$, $(X(3), Y(3))$, etc., and then connect all these points together with lines. For example:

```
>> X=[1 3 4 6 8 12 18];
>> Y=3.*X;
>> plot(X,Y)
```

1.9.1 Multiple Plots and Subplots

Another thing you might want to do is superimpose multiple plots in the same figure window, to compare the plots for example. This can be done using the hold command. Normally, when you type a plot command, any previous figure window is simply erased, and replaced by the results of the new plot. However, if you type “hold on” at the command prompt, all line plots created after that will be superimposed in the same figure window and axes. Likewise the command “hold off” will stop this behavior, and revert to the default (i.e., new plot will replace the previous plot). For example:

```
>> X=[1 3 4 6 8 12 18];
>> Y1=3.*X;
>> Y2=4.*X+5;
>> Y3=2.*X-3;
>> plot(X,Y1);
>> hold on
>> plot(X,Y2);
>> plot(X,Y3);
```

Still another thing you might want to do is to have multiple plots in the same window, but each in a separate part of the window (i.e., each with their own axes). You can do this using the **subplot** command. If you type subplot(M,N,P) at the command prompt, MATLAB will divide the plot window into a bunch of rectangles — there will be M rows and N columns of rectangles — and MATLAB will place the result of the next ”plot” command in the Pth rectangle (where the first rectangle is in the upper left). For example:

```
>> subplot(3,1,1)
>> plot(X,Y1)
>> subplot(3,1,2)
>> plot(X,Y2)
>> subplot(3,1,3)
>> plot(X,Y3)
```