# Matlab Introduction for AM105b

DEAS IT: Academic Computing Support

Based upon the original notes by Suvendra Nath Dutta, DEAS/IT,
as revised by Anthony A. Harkin, DEAS, January 2004.

29th January 2004

## Contents

**1 Introduction to Matlab©** **3**
- 1.1 What Is MATLAB? . . . . . . . . . . . . . . . . . . . . . . . . . . 3
- 1.2 The MATLAB System . . . . . . . . . . . . . . . . . . . . . . . . . 3
- 1.3 Getting access to Matlab . . . . . . . . . . . . . . . . . . . . . . . 4
- 1.4 Starting and stopping MATLAB . . . . . . . . . . . . . . . . . . . . 4
- 1.5 Basic MATLAB syntax . . . . . . . . . . . . . . . . . . . . . . . . 6
- 1.6 Saving and loading data . . . . . . . . . . . . . . . . . . . . . . . . 7
- 1.7 Where to get help . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

**2 Matrices and vectors** **9**
- 2.1 Transpose of matrices and vectors . . . . . . . . . . . . . . . . . . 10
- 2.2 Creating vectors . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
- 2.3 Creating matrices . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
- 2.4 Basic matrix operations . . . . . . . . . . . . . . . . . . . . . . . . 13
- 2.5 Indexing into a matrix . . . . . . . . . . . . . . . . . . . . . . . . . 14

**3 Graphics** **16**
- 3.1 2-D plots . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
- 3.2 3-D plots . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

**4 Programming with MATLAB** **20**
- 4.1 Using m-files . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
- 4.2 Scripts . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
- 4.3 Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
- 4.4 Program flow control . . . . . . . . . . . . . . . . . . . . . . . . . 22

**5 MATLAB Examples** **24**
5.1 Solution of linear system . . . . . . . . . . . . . . . . . . . . . . . . 24
5.2 Solution of linear differential system . . . . . . . . . . . . . . . . . 25
5.3 Fourier series analysis . . . . . . . . . . . . . . . . . . . . . . . . 26
5.4 Taylor series expansion . . . . . . . . . . . . . . . . . . . . . . . . 30

**6 MATLAB Symbolic Math Toolbox** **33**
6.1 Symbolic computing . . . . . . . . . . . . . . . . . . . . . . . . . . 33
6.2 Symbolic calculus . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
6.3 Symbolic simplification of expressions . . . . . . . . . . . . . . . . 35
6.4 Symbolic solution of equations . . . . . . . . . . . . . . . . . . . . 36

# 1 Introduction to Matlab©

## 1.1 What Is MATLAB?

MATLAB is a high-performance language for technical computing. It integrates computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation. Typical uses include

- Math and computation
- Algorithm development
- Data acquisition
- Modeling, simulation, and prototyping
- Data analysis, exploration, and visualization
- Scientific and engineering graphics
- Application development, including graphical user interface building

MATLAB is an interactive system whose basic data element is an array that does not require dimensioning. This allows you to solve many technical computing problems, especially those with matrix and vector formulations, in a fraction of the time it would take to write a program in a scalar non-interactive language such as C or Fortran.

The name MATLAB stands for matrix laboratory . MATLAB was originally written to provide easy access to matrix software developed by the LINPACK and EISPACK projects. Today, MATLAB engines incorporate the LAPACK and BLAS libraries, and several *Toolboxes* that allow for real-life engineering problem solving through an intuitive interface.

## 1.2 The MATLAB System

The MATLAB system consists of several different components all of which can be used individually or together to solve a problem. The first and most apparent piece is the *Development Environment*. This is the set of tools that let you do all the basic functions like entering commands, view and save data etc. The second element in MATLAB is the *Mathematical Function Library.*This contains the various mathematical functions ranging from the elementary (like *sum*, *sine* etc.) to the complicated (like Bessel Functions, etc.). The third important tool within MATLAB is the graphics package that comes with it. This allows users to graph both data and functions in 2D and 3D. When using MATLAB in the interactive mode,these three would probably the most used components of MATLAB.

In addition to these three components MATLAB has the *MATLAB language*. This is a high-level matrix/array language with flow control, functions, data

structures etc. This component is particularly useful when writing scripts and functions to run in a non-interactive mode.

The final component in MATLAB is the *Application Programming Interface*, otherwise known as the API. This allows users to extend MATLAB by writing specialized functions and methods in other high-level languages like *C/C++* or *Fortran*. We will not be using this component in this workshop.

## 1.3 Getting access to Matlab

Matlab is available on any public lab PC or Mac in the Science Center or in the Houses, although its useful Symbolic Math Toolbox is only installed on the PCs at present. Additionally, Matlab may be installed onto your personal PC or Mac (OS 10.2 or higher) using the fas software downloads, located at

`http://www.fas.harvard.edu/computing/download`

Installing onto a PC from this site is straightforward. However, Mac users, before attempting to download or install anything from that site, should study the instructions `Matlab_install_instr_Mac_OS_X.pdf` in the AM105b Matlab folder. (The instructions were prepared by TF Mike Weidman of DEAS, in consultation with Thad Sze of the DEAS IT group. You can send an e-mail to Mike at `mweidman@fas.harvard.edu` if there are questions about the procedure.)

## 1.4 Starting and stopping MATLAB

- On Windows platforms, to start MATLAB, double-click the MATLAB shortcut icon on your Windows desktop.
- On UNIX platforms, to start MATLAB, type `matlab` at the operating system prompt.
- On Mac OS X, start MATLAB by double-clicking the LaunchMATLAB icon in the bin folder, within the Matlab folder in Applications.

When you start MATLAB, the MATLAB desktop appears, containing tools (graphical user interfaces) for managing files, variables, and applications associated with MATLAB. The first time MATLAB starts, the desktop appears as shown in the Figure 1. You can change the way your desktop looks by opening, closing, moving, and resizing the tools in it. You can change the directory in which MATLAB starts, define startup options including running a script upon startup, and reduce startup time in some situations.

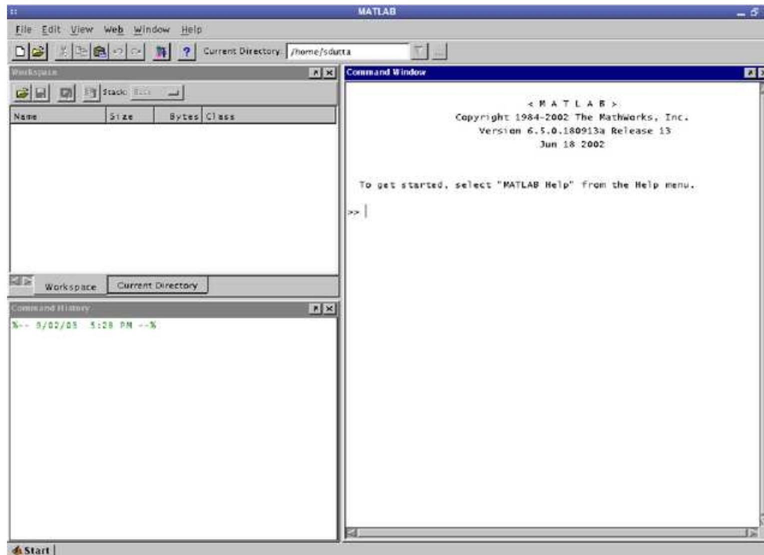


Figure 1: The MATLAB desktop

To end your MATLAB session, select Exit MATLAB from the File menu in the desktop, or type quit in the Command Window. To execute specified functions each time MATLAB quits, such as saving the workspace, you can create and run a finish .m script.

On Unix platforms typing `matlab -h` will give a listing of command line options that allow control over how MATLAB is opened. On a Mac OS X use:

```
/Applications/MATLAB6p5/bin/matlab -h
```

The ones of greatest use are `-nodesktop -nojvm -nosplash`. If you are using MATLAB by connecting to a remote unix machine with either a poor connection or no X windows support, this will launch a bare bones MATLAB environment that runs a lot faster than the one with the full graphical user interface support. Note that on a Mac OS X machine it is possible to get this brief version of MATLAB by typing

```
/Applications/MATLAB6p5/bin/matlab -nodesktop -nosplash -nojvm
```

(this assumes that MATLAB was installed in the default location of the disk).

**NOTE**: MATLAB on Mac OS X will start X11 before starting MATLAB. It is **very** important that the X11 application stay running for the entire duration MATLAB is running. If the X11 application is closed before MATLAB is quit, you will not be able to run MATLAB any further and will have to shut it down.

## 1.5 Basic MATLAB syntax

MATLAB requires that all variable names be assigned numerical values prior to being used. Typing the name (say `x`), then the equal to sign (`=`), followed by the numerical value (viz. `5`) and finally `Enter` assigns the numerical value to the variable (in our example `5` is assigned to `x`).

For example:

```
>> p=7.1
p =
    7.1000
```

A semicolon at the end of the expression typed by the user suppresses the system's echoing of entered data

```
>> p=7.1;
>>
```

It is also possible to combine expressions with a semicolon sign or a comma sign. Depending on whether a semicolon or a comma is used different things are echoed by the system.

```
>> p=7.1; x=4.92;
>> p=7.1, x=4.92;
p =
    7.1000
>> p=7.1, x=4.92,
p =
    7.1000
x =
    4.9200
>> p=7.1; x=4.92,
x =
    4.9200
```

The arithmetic operators in MATLAB are addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`) and exponentiation (`^`). For example the equation below:

$$t = \left(\frac{1}{1+px}\right)^k$$

is written in MATLAB as:

```
t = (1/(1+p*x))^k
```

Some useful keys to remember are:

- The ↑ key scrolls through previously typed commands. To recall a particular entry from the history, type the first few letters of the entry and then press the ↑ key.
- The ← and → keys allow you to edit the previously typed command

- The `ESC` key clears the command line.
- `Ctrl+C` quits the current operation and returns control to the command line.

## 1.6 Saving and loading data

If you need to shutdown MATLAB in the middle of work, it is possible to save your work with the `save` keyword. This writes out a file called `matlab.mat` to the current directory. When MATLAB is restarted, you can load your work back with the `load` keyword. This loads all the variables you defined in your last session into the current session, and you can continue your work.

```
>> p=7.1
p =
    7.1000
>> save
Saving to: matlab.mat
>> quit
```

Restart MATLAB and then say the following:

```
>> load
Loading from: matlab.mat
>> p
p =
    7.1000
```

Note that you now have `p` defined in the new session. Note that this will not work if you do not have write permission in your current directory:

```
>> cd /etc
>> save
??? Error using ==> save
Unable to write file matlab.mat: permission denied.
```

## 1.7 Where to get help

MATLAB comes with an enormous amount of help. You can type `help` at the command line. Typing `help` followed by some keyword or function will give detailed help on that function. If you are not running MATLAB with the `-nodesktop` option you can view a large set of demos by typing `demo`.

There is a lot of material online at the web site of Mathworks (`www.mathworks.com`) (the makers of MATLAB).

Useful resources for learning about Matlab:

- The Matlab help files.

- G. Jensen, Using Matlab in Calculus, Prentice Hall, 2000. (Ordered for sale at the Coop; on reserve at Cabot).
- D. J. Higham and N. J. Higham, Matlab Guide, Society of Industrial and Applied Mathematics (SIAM), 2000. (On reserve at Cabot.)
- D. Hanselman and B. R. Littlefield, Mastering Matlab 6, Prentice Hall, 2000.
- Search on the web for "matlab tutorials"; you'll find several good sites; e.g.

  `http://web.ew.usna.edu/~mecheng/DESIGN/CAD/MATLAB/usna.html`

  (When appropriate, you can copy and paste from web pages into Matlab to speed your way through tutorials.)

## 2 Matrices and vectors

An array **A** of $m$ rows and $n$ columns is called a matrix of order $(m \times n)$. The elements of **A** are referred to as $A_{ij}$ where $i$ is the row number and $j$ is the column number. The simplest way of entering the matrix in MATLAB is by entering it explicitly.

To enter the matrix, simply type in the Command Window

```
>> A = [16 3 2 13; 5 10 11 8; 9 6 7 12; 4 15 14 1]
A =
    16     3     2    13
     5    10    11     8
     9     6     7    12
     4    15    14     1
```

The order of the matrix **A** is determined with:

```
>> size(A)
ans =
     4     4
```

Note that the function `size` returns two values. It is possible to assign these values to variables as follows:

```
>> [m, n] = size(A)
m =
     4
n =
     4
```

**Note** that to enter the matrix as a list of its elements you only have to follow a few basic conventions:

- Separate the elements of a row with spaces or commas.
- Use a semicolon, ;, to indicate the end of each row.
- Surround the entire list of elements with square brackets, [ ].

It is possible to mix spaces and commas when declaring a matrix as shown below

```
>> A = [16, 3, 2, 13; 5, 10 11, 8; 9, 6 7, 12; 4, 15, 14 1]
A =
    16     3     2    13
     5    10    11     8
     9     6     7    12
     4    15    14     1
```

But this can get very hard to read.

Vectors are just a special case of matrices. If $m = 1$, then $A$ is a column vector. Similarly if $n = 1$ then $A$ is a row vector.

The distinction between row and column vectors are important because of the rules of multiplying vectors and matrices. For example, suppose you have a matrix **A**, a column vector **c** and a row vector **r**. Only the following operations are allowed: **A**.**c** and **r**.**A**. This can be seen in MATLAB as follows:

```
>> c=[3 2 1 4]';
>> r=[3 2 1 4];
>> r*A
ans =
    83    95    91    71
>> A*c
ans =
   108
    78
    94
    60
>> A*r
??? Error using ==> *
Inner matrix dimensions must agree.
>> c*A
??? Error using ==> *
Inner matrix dimensions must agree.
```

## 2.1 Transpose of matrices and vectors

A transpose of a matrix is defined as follows:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

In a general case the elements of the transpose $\mathbf{A}^T$ of the matrix **A** with elements $A_{ij}$ is simply the matrix with elements $A_{ji}$.

In MATLAB the `'` operator takes the transpose of a matrix or a vector. Transposing a row vector turns it into a column vector and vice-versa. For example we could take our column vector `c` from above and transpose it to get a row vector.

```
>> cr=c';
>> cr*A
ans =
    83    95    91    71
```

## 2.2 Creating vectors

There several ways of creating vectors that can be very useful. The simplest and probably most commonly used method create a vector uses the colon notation

```
x = s:d:f
```

where `s` is the start of vector, `d` is the increment (or decrement) between the elements of the vector and `f` is the last element of the vector. Obviously this can be used when the elements of the vector are equispaced. For example:

```
>> x=0:0.3:pi;
>> x'
ans =
         0
    0.3000
    0.6000
    0.9000
    1.2000
    1.5000
    1.8000
    2.1000
    2.4000
    2.7000
    3.0000
```

Size of the vector can be got from `length(x)`.

```
>> length(x)
ans =
    11
```

If `d` is ignored MATLAB assumes an increment of 1.

```
>> x=0:pi
x =
     0     1     2     3
```

On the other hand to specify $n$ equally spaced intervals use the following:

```
>> x=linspace(0, pi, 7)
x =
         0    0.5236    1.0472    1.5708    2.0944    2.6180    3.1416
```

In this case the increment or decrement is (final - start)/(n-1).

To specify equal spacing in logarithm space use the following:

```
>> logspace(1,2,7)
ans =
   10.0000   14.6780   21.5443   31.6228   46.4159   68.1292  100.0000
```

in this case MATLAB creates the vector, $[10^s 10^{s+d} \dots 10^f]$, where $d$ is $d = (f-s)/(n-1)$. Note that if $f$ is $\pi$ then the elements of the vector are numbers between $10^s$ and $\pi$. In this case the interval $d$ is $(log_{10}(\pi) - s)/(n-1)$.

Of course it is possible to explicitly write out the matrices as we have seen before. It is also possible to create vectors from matrices as will be shown later.

## 2.3 Creating matrices

The easiest way of creating matrices is as described before, by listing members explicitly.

```
>> A = [16 3 2 13; 5 10 11 8; 9 6 7 12; 4 15 14 1]
A =
    16     3     2    13
     5    10    11     8
     9     6     7    12
     4    15    14     1
```

It is also possible to create a matrix from a group of row vectors. For example

```
>> v_1 = [1 2 3];
>> v_2 = [4 5 6];
>> v_3 = [7 8 9];
>> A = [v_1; v_2; v_3]
A =
     1     2     3
     4     5     6
     7     8     9
```

The order of `A` is $3 \times$ `length(v_1)`.

```
>> size(A)
ans =
     3     3
>> length(v_1)
ans =
     3
```

In addition there are a few utility routines to create matrices:

- `zeroes(m, n)`: a matrix with all zeros of order $m \times n$.
- `ones(m, n)`: a matrix with all ones.
- `eye(m, n)`: the identity matrix (ones along the diagonal, zeros everywhere else).
- `rand(m, n)`: uniformly distributed random elements.
- `randn(m, n)` : normally distributed random elements.
- `magic(m)`: a square matrix whose elements have the same sum, along the row, column and diagonal. An example

  ```
  >> magic(3)
  ans =
       8     1     6
       3     5     7
       4     9     2
  ```

- `pascal(m)`: a pascal matrix. An example would be:

```
>> pascal(3)
ans =
     1     1     1
     1     2     3
     1     3     6
```

## 2.4 Basic matrix operations

You have already seen the transpose operator ' before. In addition there are the following list of operations possible on a matrix:

- `^`: exponentiation
- `*`: multiplication
- `/`: division
- `\`: left division. The operation `A\B` is effectively the same as `INV(A)*B`, although left division is calculated differently.
- `+`: addition
- `-`: subtraction

One very important to thing to note is the automatic promotion of scalars. For example when adding a $m \times n$ order matrix $\mathbf{A}$ to a scalar $x$, the scalar is promoted to a matrix of order $m \times n$ with every element equal to the original scalar .

```
>> w = [1 2; 3 4] + 5
w =
     6     7
     8     9
```

There are also a set of operations that apply to the matrices on a element by element basis. These are called array operations. Examples are:

- `.'` : array transpose
- `.^` : array power
- `.*` : array multiplication
- `./` : array division

It is very important to distinguish between these. In the example below with two $2 \times 2$ matrices, a matrix multiplication `*` and an array multiplication `.*` result in complete different matrices.

```
>> A=[1 2; 3 4];
>> B=[5 6; 7 8];
>> A*B
ans =
    19    22
    43    50
>> A.*B
ans =
     5    12
    21    32
```

## 2.5 Indexing into a matrix

Indices in MATLAB follow the "fortra" notation of starting at 1 and going up to the order of the matrix. So we have the following:

```
>> A=rand(2)
A =
    0.9501    0.6068
    0.2311    0.4860
>> A(2,2)
ans =
    0.4860
```

It is also possible to use a single index, which goes top to bottom (column first) and then left to right (row second).

```
>> A(4)
ans =
    0.4860
```

In other words it is possible to refer to the element $A_{ij}$ as `A(i, j)` or as `A((i-1)*m+j)`, where `m` is the no. of rows of the matrix.

A very powerful operator in indexing into a MATLAB matrix is the `:` operator. For example:

```
>> A(:,end)
ans =
    0.6068
    0.4860
```

gives the last column of the matrix. Or

```
>> A(1:2,1:1)
ans =
    0.9501
    0.2311
```

gives the first (`1:1`) column both (`1:2`) rows. It can now be seen that it is possible to create vectors from the rows and columns of a matrix as follows:

```
>> r=A(1:1, 1:2)
r =
    0.9501    0.6068
>> c=A(1:2, 1:1)
c =
    0.9501
    0.2311
```

MATLAB has a lot more information about matrices and the kind of operations you can do with them. To read that information click on the `Help` link at the top of the desktop (on Mac OS X it is on the top of the screen). Then select the `Contents` view. Click on the words `MATLAB`. If you see a small "+" sign to the left of MATLAB click it to open the documentation tree. Then click on the "+" sign to the left of `Mathematics` and click on `Matrices and Linear Algebra`.

# 3 Graphics

## 3.1 2-D plots

The basic 2-D plotting routine in MATLAB is `plot(xdata, ydata, 'color_linestyle_marker')`. For example:

```
>> x=-5:0.1:5;
>> sqr=x.^2;
>> pl1=plot(x, sqr, 'r:s');
```

produces the Figure 2.

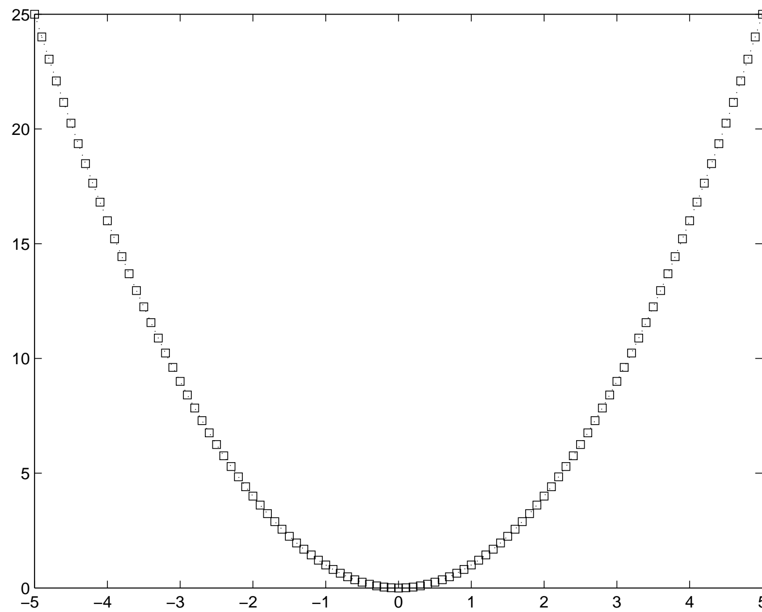


Figure 2: A simple 2D plot

To plot a second plot on top of an existing plot, use hold on. This is demonstrated in Figure 3. Obviously, hold off forces the next plot to show up on a different window.

```
>> cub=x.^3;
>> hold on
>> pl2=plot(x, cub, 'k-o');
```

MATLAB allows the annotation of the plots with a few keywords.

```
>> title('Demo plot');
>> xlabel('X Axis');
>> ylabel('Y Axis');
>> legend([pl1, pl2], 'x^2', 'x^3');
```

produces Figure 4

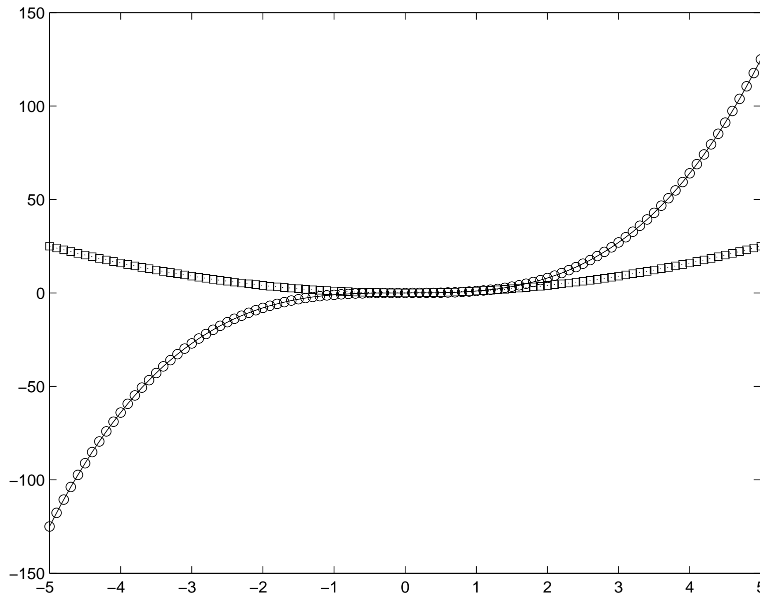


Figure 3: A 2D plot displaying overlay

## 3.2 3-D plots

It is possible to draw 3-D line plots exactly the same way as 2-D plots using `plot3(x, y, z);` where `x`, `y` and `z` are vectors of same length. For example the following:

```
>> z=0:0.1:40;
>> x=cos(z);
>> y=sin(z);
>> pl=plot3(x, y, z);
```

produces Figure 5.

A far more powerful set of 3D plotting functions are those that create surfaces, contours and so on. The basic surface plotting routines are surf and mesh. If we have a surface defined by `z = f(x, y)` then the surface plot is generated by `surf(x, y, z)`. For example the following code:

```
>> xx1=linspace(-3, 3, 15);
>> xx2=linspace(-3, 13, 17);
>> [x1, x2] = meshgrid(xx1, xx2);
>> z=x1.^4+3*x1.^2-2*x1+6-2*x2.*x1.^2+x2.^2-2*x2;
>> pl=surf(x1, x2, z);
```

results in Figure 6.

The possibilities of complex plots are quite enormous. To see the capabilities of MATLAB look at the graphics demos. To do this click on `Help` at the top

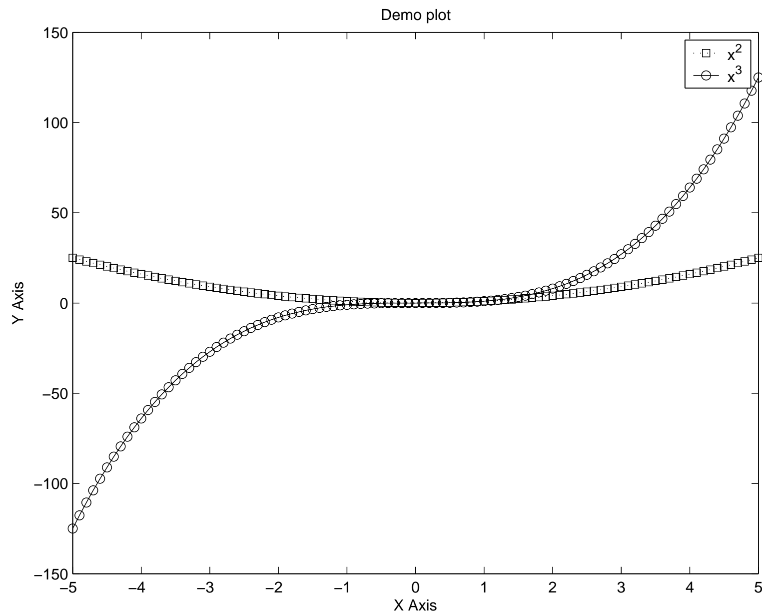


Figure 4: A 2D plot with annotations

of the desktop, as usual. Then click on the word `Demos` on the top left. Then click the "+" sign to the left of `MATLAB`. Then click the "+" sign to the left of `Graphics`. Try any one of the demos listed. Particularly attractive ones are `Teapot`, `Viewing a Penny` and `Earth's Topography`.

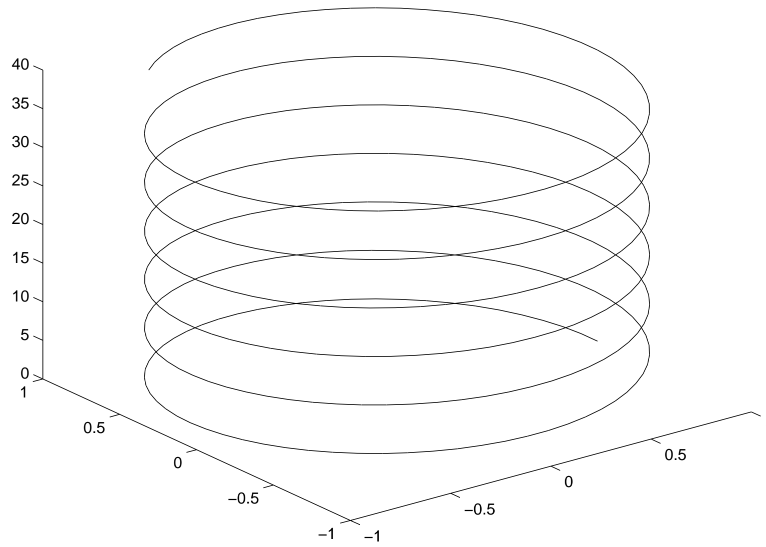


Figure 5: A simple 3D plot

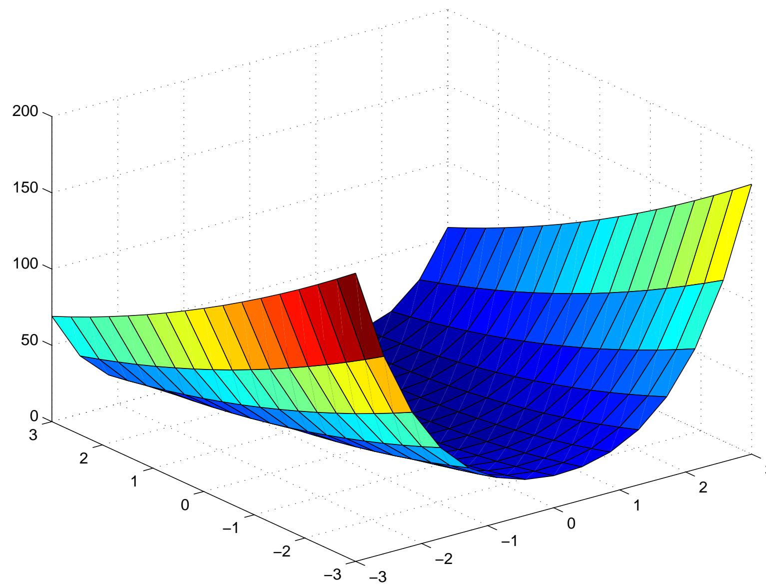


Figure 6: A 3D surface plot

# 4 Programming with MATLAB

## 4.1 Using m-files

MATLAB provides a full programming language that enables you to write a series of MATLAB statements into a file and then execute them with a single command. You write your program in an ordinary text file, giving the file a name of filename.m . The term you use for filename becomes the new command that MATLAB associates with the program. The file extension of .m makes this a MATLAB M-file.

M-files can be scripts that simply execute a series of MATLAB statements, or they can be functions that also accept arguments and produce output. You create M-files using a text editor, then use them as you would any other MATLAB function or command.

The process looks as displayed in Figure 7.

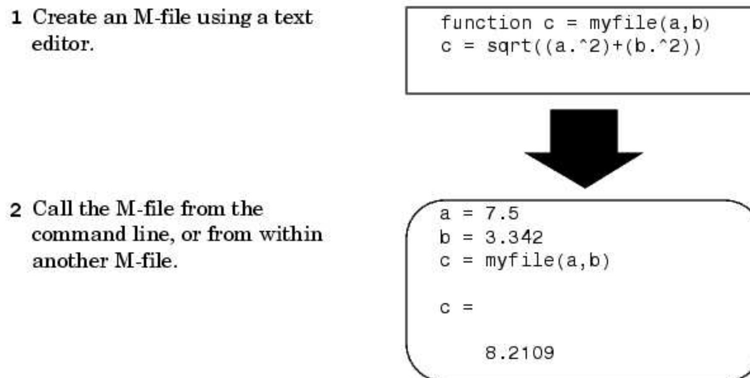


Figure 7: Steps in using a m-file

What goes in a M-file?

```
function f = fact(n) % Function definition line
% FACT Factorial. % H1 line
% FACT(N) returns the factorial of N, H! % Help text
% usually denoted by N!
% Put simply, FACT(N) is PROD(1:N).
f = prod(1:n);       % Function body
return
```

This function has some elements that are common to all MATLAB functions:

- A function definition line. This line defines the function name, and the number and order of input and output arguments.
- An H1 line. H1 stands for "help 1" line. MATLAB displays the H1 line for a function when you use `lookfor` or request help on an entire directory.

- Help text. MATLAB displays the help text entry together with the H1 line when you request help on a specific function.
- The function body. This part of the function contains code that performs the actual computations and assigns values to any output arguments.

## 4.2 Scripts

Scripts are the simplest kind of M-file because they have no input or output arguments. They're useful for automating series of MATLAB commands, such as computations that you have to perform repeatedly from the command line. Scripts operate on existing data in the workspace, or they can create new data on which to operate. Any variables that scripts create remain in the workspace after the script finishes so you can use them for further computations.

The following demonstrates a simple script `m`-file. These statements calculate $\rho$ for several trigonometric functions of $\theta$, then create a series of polar plots.

```
% An M-file script to produce       % Comment lines
% "flower petal" plots
theta = -pi:0.01:pi;                % Computations
rho(1,:) = 2 *sin(5 *theta).^2;
rho(2,:) = cos(10 *theta).^3;
rho(3,:) = sin(theta).^2;
rho(4,:) = 5 *cos(3.5 *theta).^3;
for k = 1:4
    polar(theta,rho(k,:))           % Graphics output
    pause
end
```

Try entering these commands in an M-file called petals.m. This file is now a MATLAB script. Typing petals at the MATLAB command line executes the statements in the script. In this case it will cycle through four plots. The `pause` button will cause MATLAB to wait after drawing on figure for any key to be pressed. After the script displays a plot, press Return to move to the next plot. There are no input or output arguments; petals creates the variables it needs in the MATLAB workspace. When execution completes, the variables (`i`,`theta`, and `rho`) remain in the workspace. To see a listing of them, enter `whos` at the command prompt. You can also see the variables listed in the workspace window if you have that open. Note that if you click on the variable listed in the workspace you open the *Array editor* which displays and allows you to edit the variable array.

## 4.3 Functions

Functions are M-files that accept input arguments and return output arguments. They operate on variables within their own workspace. This is separate from the workspace you access at the MATLAB command prompt. This will be explained in more detailed in the next section.

The average function shown below is a simple M-file that calculates the average of the elements in a vector.

```
function y = myaverage(x)
% myaverage Mean of vector elements.
% myaverage(X), where X is a vector, is the mean of vector elements.
% Nonvector input results in an error.
[m,n] = size(x);
if (~((m == 1) | (n == 1)) | (m == 1 & n == 1))
        error('Input must be a vector')
end
y = sum(x)/length(x);      % Actual computation
return
```

Enter these commands in an M-file called average.m . The average function accepts a single input argument and returns a single output argument. To call the average function, enter

```
>> x=1:99;
>> myaverage(x)
ans =
    50
```

## 4.4 Program flow control

MATLAB has four basic flow control structures in programming: while, if, for, and switch. Each of these control elements must have a matching end keyword downstream in the program. Logic control structures are:

```
if/elseif/else
switch/case/otherwise
```

Iterative loop structures are:

```
for
while
```

An example of the if, elseif programming is as follows:

```
if i==j
    A(i, j) = 2; % called only when i is equal to j
elseif abs(i-j)==1
    A(i, j) = -1; % called only when i and j differ by 1
else
    A(i, j) = 0; % all other situations
end
```

The above assigns a tri-diagonal matrix to A. Similarly, an example of switch is:

```
switch algorithm   % switch depending on the value of the variable "algorithm"
case 'ode23'
   str = '2nd/3rd order';
case {'ode15s', 'ode23s'}
   str = 'stiff system';
otherwise
   str = 'other algorithm';
end
```

Note that, unlike most other languages, there is no need for a break statement. Also `switch` is more efficient than `if` when comparing string arguments.

A simple iterative loop using for is:

```
n=10;
for i=1:n
   for j=1:n
      a(i, j) = 1/(i+j-1);
   end
end
```

Because MATLAB is designed to work with matrices it is possible to dramatically speed up a loop. It can become more readable in the process as well, when done correctly. Following displays the traditional way of writing a loop over a order $m \times n$ matrix:

```
mass = rand(5, 10000); length = rand(5, 10000);
width = rand(5, 10000); height = rand(5, 10000);
[m, n] = size(mass);
for i=1:m
   for j=1:n
      density(i, j) = mass(i, j) / (length(i, j)*width(i, j)*height(i, j));
   end
end
```

Using MATLAB "vector" notation the above piece of code becomes:

```
density = mass ./ (length .* width .* height);
```

# 5 MATLAB Examples

## 5.1 Solution of linear system

We will now focus on some examples to demonstrate how to work with MATLAB. First we look at the solution of a system of linear algebraic equations.

Consider a system of $n$ equations with $n$ unknowns $x_k$, $k$ = 1, 2, ..., n:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_1 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

We can rewrite this in matrix notation as: $\mathbf{Ax} = \mathbf{b}$ where,

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1n} \\ A_{21} & A_{22} & \dots & A_{2n} \\ & \vdots & & \\ A_{n1} & A_{n2} & \dots & A_{nn} \end{pmatrix}$$

and

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \& \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

The symbolic solution is:

$$\begin{aligned} \mathbf{A}^{-1}\mathbf{Ax} &= \mathbf{A}^{-1}\mathbf{b} \quad \text{dividing both sides by } \mathbf{A}^{-1} \\ \mathbf{x} &= \mathbf{A}^{-1}\mathbf{b} \end{aligned}$$

since $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$ and $\mathbf{Ix} = \mathbf{x}$, where $\mathbf{I}$ is the identity matrix:

$$\mathbf{I} = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ & \vdots & & \\ 0 & 0 & \dots & 1 \end{pmatrix}$$

In MATLAB `A\b` is equivalent to `inv(A)b` (where `inv(A)` calculates the inverse of the matrix **A**) but it is calculated without inverting **A**. This results in a significantly reduced computational effort and time. So in MATLAB the way to solve this set of equations is as simple as `x = A\b`!

An example of a linear system is the following set of equations:

$$\begin{aligned} 8x_1 + x_2 &= 7 \\ 3x_1 + 5x_2 &= 4 \end{aligned}$$

In this case the matrix **A** is:

$$\begin{pmatrix} 8 & 1 \\ 3 & 5 \end{pmatrix}$$

Clearly the determinant of **A** is 37, and so the inverse of **A** is:

$$\frac{1}{37}\begin{pmatrix} 5 & -1 \\ -3 & 8 \end{pmatrix} \rightarrow \begin{pmatrix} 0.1351 & -0.0270 \\ -0.0811 & 0.2162 \end{pmatrix}$$

The solution then becomes:

$$\frac{1}{37}\begin{pmatrix} 5 & -1 \\ -3 & 8 \end{pmatrix}\begin{pmatrix} 7 \\ 4 \end{pmatrix} \rightarrow \frac{1}{37}\begin{pmatrix} 31 \\ 11 \end{pmatrix} \rightarrow \begin{pmatrix} 0.8378 \\ 0.2973 \end{pmatrix}$$

If we solve the same problem in MATLAB

```
>> A=[8 1; 3 5];
>> b=[7 4]';
>> x=A\b
x =
    0.8378
    0.2973
```

we can confirm that we get the same answer with considerably greater facility of use.

It can be confirmed that `x` is the solution of the equations, by multiplying it with `A`

```
>> z=A*x
z =
     7
     4
```

which is identical to `b`. Look at the section on Matrices for details on matrix operations.

## 5.2 Solution of linear differential system

MATLAB has a number of functions to solve first order linear differential equations. One in particular is `ode45` which solves *non-stiff* differential equations (non-stiff means the differential equations have solutions that have a single timescale). The function `ode45` solves a differential equation of the form:

$$\frac{dy_i}{dt} = f_i(y_1, y_2, \ldots, y_n) \quad i = 1, 2, \ldots, n$$

over the interval $t_0 \leq t \leq t_f$ subject to the initial conditions $y_j(t_0) = a_j, j = 1, 2, ..., n$, where $a_j$ are constants. The usage of the `ode45` are as follows:

```
[t, y] = ode45(@FunctionName, [t0 tf], [a1 a2 ... an]', ...
        options, p1, p2, ...)
```

In the above [`t, y`] denotes that `ode45` returns two results. The first`t` is a column vector of the times in the range [`t0 tf`] that are determined by `ode45` and the second output `y` is the matrix of solutions such that the rows are the solutions at any given time $\mathtt{t}_i$ in the corresponding row of the first output `t`. Also, `@FunctionName` is the handle for the name of the function file *FunctionName* (ignoring the *.m* at the end of the file) that represents the array of functions which form the right hand side of the equations. Its form must be: `function yprime=FunctionName(t, g, p1, p2, ...)` where `t` is the independant variable, `g` is the vector representing $y_j$, and `p1, p2,` etc. are parameters.

Consider the following second order ordinary differential equation, which could represent a forced damped oscillator.

$$\frac{d^2y}{dt^2} + 2\xi\frac{dy}{dt} + y = h(t)$$

Let us now make the substitution,

$$y_1 = y$$
$$y_2 = \frac{dy}{dt}$$

Then the second order equation can be replaced by two first order equations.

Assume that $\xi = 0.15$ and that we start at time $t_0 = 0$ and end at $t_f = 35$. At $t_0$ the displacement and the velocity are both zero, viz. $y_1(t_0) = 0$ and $y_2(t_0) = 0$. Finally we assume $h(t) = 1$.

First create the function which returns the array of right hand side functions (in this case a two element column vector.

```
function ForcingFunction(t, w, xi)
% ForcingFunction - return the right hand side of the linear differential system % H1 line
% ForcingFunction takes in the time t, vector w, and the constant xi. The
% vector w gives the values of the dependant variable at the current time.
y = [w(2); -2*xi*w(2)-w(1)+1];
```

save this as a file ForcingFunction.m. This file may be created using MATLAB's own editor as displayed in the Figure 8.

Then run the following commands:

```
>> [tt, yy] = ode45(@ForcingFunction, [0 35], [0 0]', [], 0.15);
>> plot(tt, yy(:, 1))
>> xlabel('Time');
>> ylabel('y(Time)');
```

You should get the Figure 9 displaying the displacement $y(t)$ of the oscillator with time $t$. As expected, the asymptotic value is the forcing value of 1, and the oscillator displays a transient with a damping related to the constant.

## 5.3 Fourier series analysis

Any real valued periodic function $f(x)$can be represented as an infinite sum of a Fourier series, a sum of sin and cos functions:

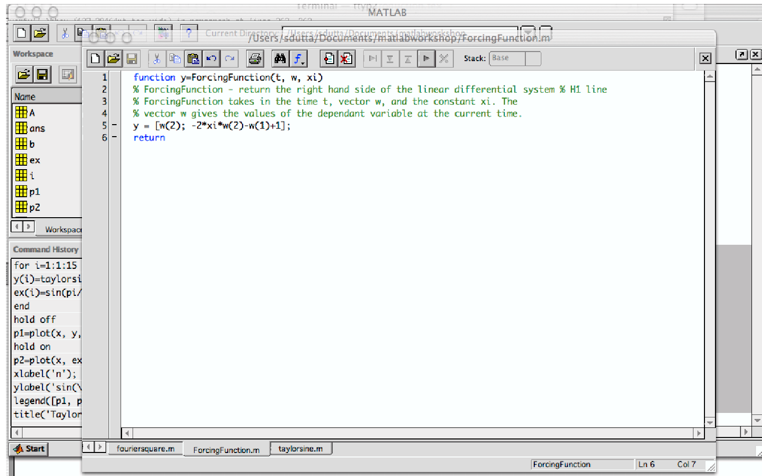


Figure 8: Matlab's built in editor

$$f(x) = \sum_{n=0}^{\infty} a_n \sin(nx) + \sum_{n=0}^{\infty} b_n \cos(nx)$$

We can use the orthogonality properties of sin and cos to get the following relations:

$$\begin{aligned}
a_0 &= \frac{1}{L}\int_{-L}^{L} f(x')dx' \\
a_n &= \frac{1}{L}\int_{-L}^{L} f(x')\cos\left(\frac{n\pi x'}{L}\right)dx' \\
b_n &= \frac{1}{L}\int_{-L}^{L} f(x')\sin\left(\frac{n\pi x'}{L}\right)dx'
\end{aligned}$$

where $2L$is the periodicity of the function. Now let us look at the square wave function. The function is defined as:

$$f(x) = \begin{cases} h & 0 < x < \pi \\ 0 & \pi < x < 2\pi \end{cases}$$

Then using the above relations for the constants $a_n$and $b_n$we have:

$$f(x) = \frac{h}{2} + \frac{2h}{\pi}\left(\frac{\sin(x)}{1} + \frac{\sin(3x)}{3} + \frac{\sin(5x)}{5} + \cdots\right)$$

The square wave function can be representated in MATLAB using the following function myquare.m:

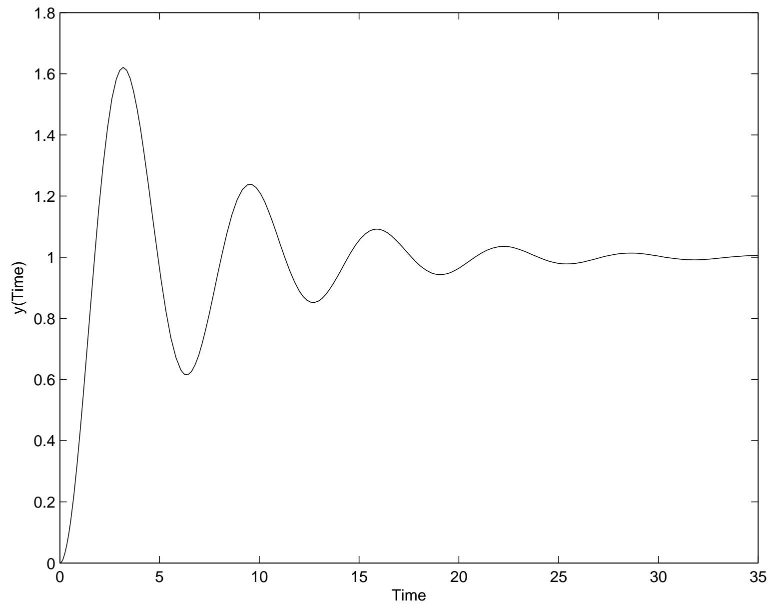


Figure 9: Solution of the differential system

```
function f=mysquare(x, h)
% mysquare -- a square hat function % H1 line
% Given a vector x, mysquare returns a vector f
% which is the square hat function of height h
% and periodicity of 2*pi
xmod = mod(x, 2*pi);
f = h*(xmod <= pi);
return
```

Note the use of the *vector* `if` statement above. The code to calculate the function *f(x)* using the fourier sum is fouriersquare.m:

```
function f=fouriersquare(x, h, n)
% fourierseries - Fourier series fit for a square hat function % H1 line
% fourierseries takes in the vector x, the height h and the no.
% n of terms in the expansion and returns a fourier fit to the
% square hat function
f = 0.5*h*ones(size(x));  % zeroth order term
sum = zeros(size(x));
for i=1:2:n % include only the odd terms
    sum = sum + sin(i*x)/i;
end
f = f + 2*h*sum/pi;
return
```

Using the following set of commands we can view the quality of fit in Figure

10.

```
>> x=0:0.1:5*pi;
>> p1=plot(x, mysquare(x, 1), 'k');
>> hold on
>> p2=plot(x, fouriersquare(x, 1, 10), 'r');
>> p3=plot(x, fouriersquare(x, 1, 50), 'b');
>> p4=plot(x, fouriersquare(x, 1, 200), 'g');
>> legend([p1, p2, p3, p4], 'Exact', 'n=10', 'n=50', 'n=200');
xlabel('\theta')
>> ylabel('f(\theta)')
>> title('Fourier fitting  square wave')
```

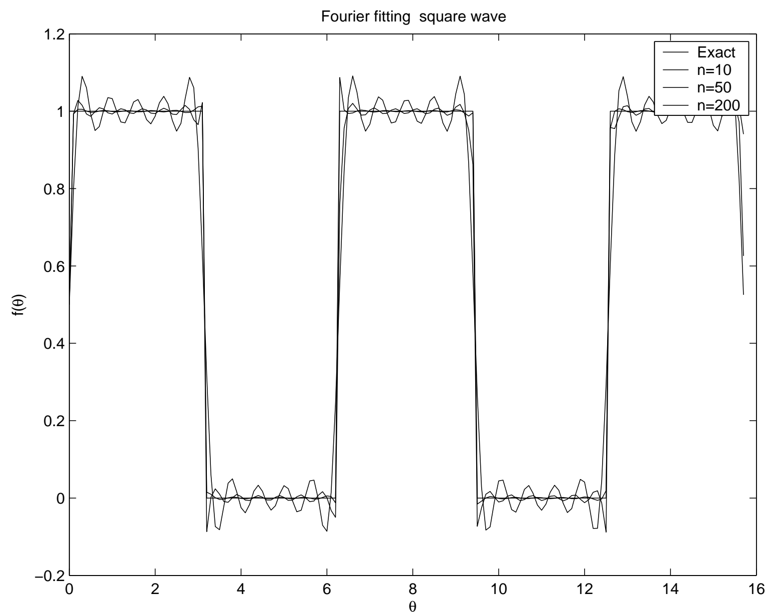


Figure 10: Fourier series fit of the square wave

As can be seen the fit gets closer as the number of terms increases, but there remains even in the sums with very high numbers of Fourier terms a *ringing* near the edges of the square wave. This is a well known phenomenon and can be understood as a natural problem occuring from trying to fit a function with a discontinuous derivative with a sum of trignometric functions all of whose higher derivative are continuous.

## 5.4 Taylor series expansion

Most well-behaved functions (viz. possessing derivatives of any order) can be expressed as a polynomial series expansion about some location:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$$

In the above $f^{(n)}$ is the $n^{\text{th}}$ order derivative of $f(x)$, i.e.:

$$f^{(n)}(x) = \frac{d^n f(x)}{dx^n}$$

and $n!$ is of course the factorial $n(n-1).(n-2)\ldots 2.1$ of $n$. This is known as the Taylor's series expansion.

Consider for example the $\sin(x)$. This can be expanded about $x = 0$ as:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

This result can be easily derived if one recalls that all even derivatives of $\sin(x)$ will some power of $-1 \times \sin(x)$ and so must vanish at $x = 0$. And all odd derivatives of $\sin(x)$ will be some power of $-1 \times \cos(x)$ which will be equal to 1 (or $-1$) at $x = 0$.

We can then use Matlab to create a function to calculate the series for a given number of terms $n$. The function `taylorsine` (in the file `taylorsine.m`) is

```
function y=taylorsine(x, n)
% taylorsine - Calculates the taylor series approximation to the sine function % H1 line
% taylorsine takes in the value of x and number n of terms to sum to and
% returns the value of the fit y
sum = 0;
for m=1:2:n; % pick each odd term to n
    sign=(-1)^((m-1)/2); % sign of each term
    yterm = sign*x.^m/factorial(m);
    sum = sum + yterm;
end
y=sum;
return
```

Then the following series of commands on MATLAB

```
>> x=1:1:15;
>> format long
>> for i=1:1:15
y(i)=taylorsine(pi/4,i);
ex(i)=sin(pi/4);
end
>> hold off
```

```
>> p1=plot(x, y, 'ro');
>> hold on
>> p2=plot(x, ex, 'k+');
>> xlabel('n');
>> ylabel('sin(\pi/4)');
>> legend([p1, p2], 'Taylor series', 'Exact');
>> title('Taylor series approximation');
```

produces the Figure 11 which displays the rapidity with which the approximation approaches the actual value of $\sin(\pi/4)$ $(= 1/\sqrt{2})$.

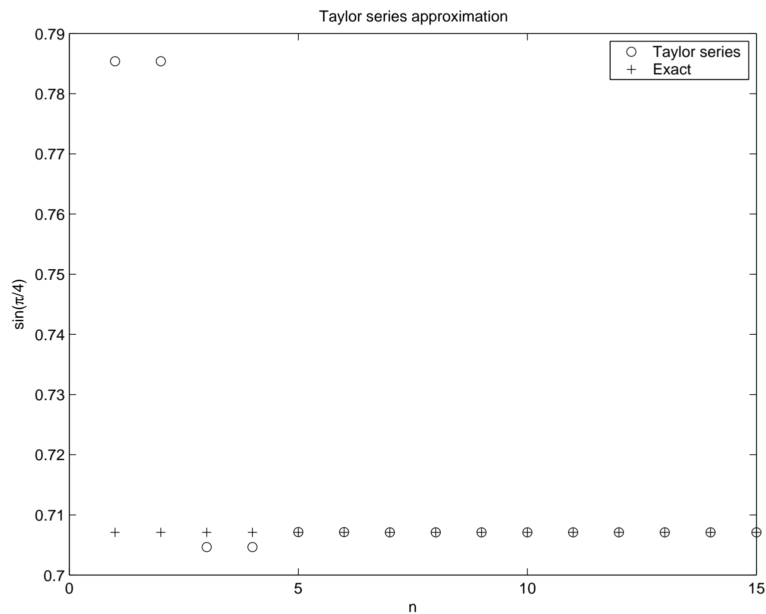


Figure 11: Taylor's approximation

This surprising result can be visualized better if we plot the sin function and the Taylor's series to different terms. Use the following code:

```
>> x=-pi:0.1:pi;
>> y=sin(x);
>> y1=x;
>> y2=x-x.^3/factorial(3);
>> y5=taylorsine(x, 5);
>> y15=taylorsine(x, 15);
>> hold off;
>> p=plot(x, y, 'k-');
>> hold on
>> p1=plot(x, y1, 'k.');
>> p2=plot(x, y2, 'k+');
>> p5=plot(x, y5, 'k*');
```

```
>> p15=plot(x, y15, 'kx');
>> xlabel('\theta');
>> ylabel('sin(\theta)');
>> legend([p, p1, p2, p5, p15], 'Exact', '1 term', '2 terms', '5 terms', ...
'15 terms');
>> title('Taylors series fit of sin(\theta)');
```

Note that we used the function `taylorsine` defined above to calculate the series fits for 5 terms and 15 terms. This results in the Figure 12.

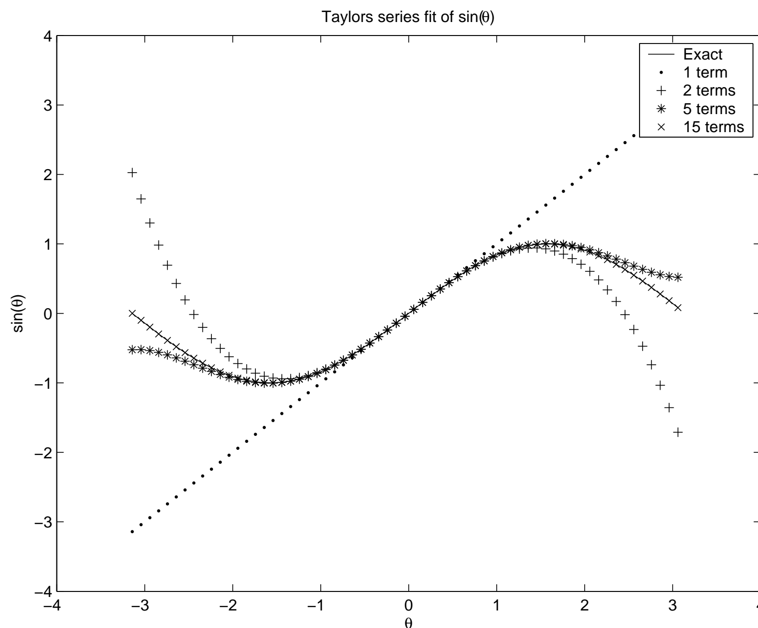


Figure 12: Taylor series expansion of Sin($\theta$)

As noticed before upto about 45° the linear fit is quite good! After that the linear fit rises too quickly. Adding the second term (which is negative) bring downs the contribution of the linear term, but it does so too fast! Adding the next term brings up the fit to above the exact curve and so on until at about 15 terms the fit looks quite good between the ranges $-\pi < \theta < \pi$.

# 6 MATLAB Symbolic Math Toolbox

## 6.1 Symbolic computing

Matlab's Symbolic Math Toolbox includes a full set of functions to solve various types of symbolic operations, including:

- Calculus
- Linear algebra
- Integral transforms
- Simplification of symbolic expressions
- Symbolic equation solving

Complete documentation on Matlab's symbolic math toolbox can be found at

`www.mathworks.com/access/helpdesk/help/toolbox/symbolic/symbolic.shtml`

**Symbolic variables:** To perform symbolic manipulations in Matlab, you must first create symbolic variables to work with by using the `syms` command. For example, the following code will create the symbolic variables $x, y, a$ and $b$, and then create and add two symbolic functions.

```
>> syms x y a b
>> f = x^2 + sin(y)
f =
 x^2+sin(y)
>> g = 2*a + b
g =
 2*a+b
>> f+g
ans =
 x^2+sin(y)+2*a+b
```

## 6.2 Symbolic calculus

The symbolic toolbox can perform several calculus operations.

**Ordinary differentiation:** The `diff` command performs differentiation.

```
>> syms x
>> g = exp(x)*cos(x);
>> diff(g)
ans =
 exp(x)*cos(x)-exp(x)*sin(x)
```

To take the second derivative of $g$, you could use `diff(diff(g))`, or use the syntax `diff(g,2)`.

```
>> diff(g,2)
ans =
 -2*exp(x)*sin(x)
```

**Partial differentiation:** The `diff` command can also perform partial differentiation.

```
>> syms s t
>> f = sin(s*t);
>> diff(f,t)        % partial derivative of f with respect to t
ans =
 cos(s*t)*s
>> diff(f,s)        % partial derivative of f with respect to s
ans =
 cos(s*t)*t
```

**Indefinite integration:** The `int` command performs symbolic integration.

```
>> syms x
>> f = 1/x;
>> int(f)
ans =
 log(x)
```

**Definite integration:** The `int` command can also evaluate definite integrals.

```
>> syms x
>> f = sin(x)^2;
>> int(f,0,2*pi)     % integrate f from 0 to 2*pi
ans =
 pi
```

**Taylor series:** The `taylor` command can find the Taylor series of a function.

```
>> syms x
>> f = 1/(5+4*cos(x));
>> T = taylor(f,8)       % finds the first 7 terms in the series
T =
 1/9+2/81*x^2+5/1458*x^4+49/131220*x^6
```

**Plotting:** The `ezplot` command makes a simple plot of a symbolic function.

```
>> syms x
>> f = 1/(5+4*cos(x));
>> ezplot(f)
```

### 6.3 Symbolic simplification of expressions

There are symbolic commands to simplify complicated expressions.

**Expand:** The `expand` command expands out expressions.

```
>> syms x
>> f = (x-1)*(x-2)*(x-3);
>> g = expand(f)
g =
 x^3-6*x^2+11*x-6
```

The `expand` command can also handle trigonometric expressions.

```
>> syms x y
>> f = cos(x+y);
>> expand(f)
ans =
 cos(x)*cos(y)-sin(x)*sin(y)
```

**Factor:** The `factor` command factors polynomials.

```
>> syms x
>> f = x^6+1;
>> factor(f)
ans =
 (x^2+1)*(x^4-x^2+1)
```

**Simplify:** The `simplify` command tries to find the simplest form of the expression.

```
>> syms x
>> f = (1-x^2)/(1-x);
>> simplify(f)
ans =
 x+1
```

**Simple:** The `simple` command tries to find the shortest form of the expression.

```
>> syms x
>> f = cos(x)^2-sin(x)^2;
>> g = simple(f)
g =
 cos(2*x)
```

**Subs:** The `subs` command substitutes a number or variable into an expression.

```
>> syms x
>> f = x^2;
>> subs(f,x,3)
ans =
     9
```

## 6.4 Symbolic solution of equations

The symbolic toolbox can solve both algebraic and differential equations.

**Algebraic equations:** The `solve` command tries to find solutions to algebraic equations.

```
>> syms x
>> f = x^3 + 1;
>> solve(f)      % solve f = 0
ans =
 [               -1]
 [ 1/2-1/2*i*3^(1/2)]
 [ 1/2+1/2*i*3^(1/2)]
```

The `solve` command can also handle trigonometric equations.

```
>> syms x
>> s = solve('cos(2*x)+sin(x)=1')
s =
 [      0]
 [     pi]
 [ 1/6*pi]
 [ 5/6*pi]
```

**Differential equations:** The `dsolve` command tries to find solutions to differential equations.

```
>> y = dsolve('Dy=1+y^2')
y =
 tan(t+C1)
```

An initial condition can be specified if desired.

```
>> y = dsolve('Dy=1+y^2','y(0)=1')
y =
 tan(t+1/4*pi)
```

A second order ODE with two initial conditions can be solved for and simplified.

```
>> y = dsolve('D2y=cos(2*x)-y','y(0)=1','Dy(0)=0', 'x')
y =
 (1/2*sin(x)+1/6*sin(3*x))*sin(x)+(1/6*cos(3*x)-1/2*cos(x))*cos(x)+4/3*cos(x)
>> simplify(y)
ans =
 -2/3*cos(x)^2+4/3*cos(x)+1/3
```