# MATLAB workshop

DEAS IT: Academic Computing Support

23rd September 2004

Spring 2004

## Contents

# 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 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 Applications.

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.

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.

## 1.4 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.

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.

Additionally you can use the keyword diary to save your commands into a file. If you type \verb+diary+
beginning of your session MATLAB will create a file called {\it diary} in your working directory and s
the commands you type in it. This file can be useful to then convert into a script or a function later
```

## 1.5 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 (`http://www.mathworks.com`) (the makers of MATLAB).

In addition you can email for assistance at `matlab_fall04@deas.harvard.edu`.

# 2 MATLAB Desktop

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.

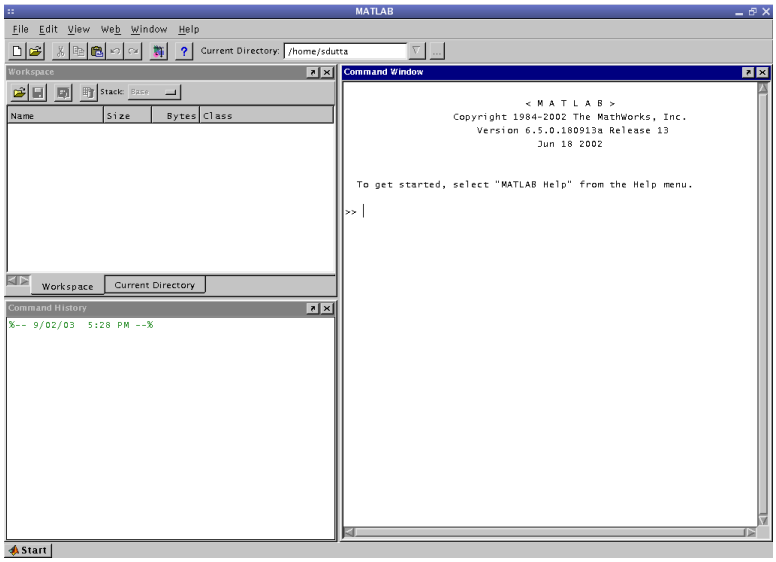


Figure 1: The MATLAB desktop

You can change the way your desktop looks by opening, closing, moving, and resizing the tools in it. Use the View menu (see Figure 2) to open or close the tools.

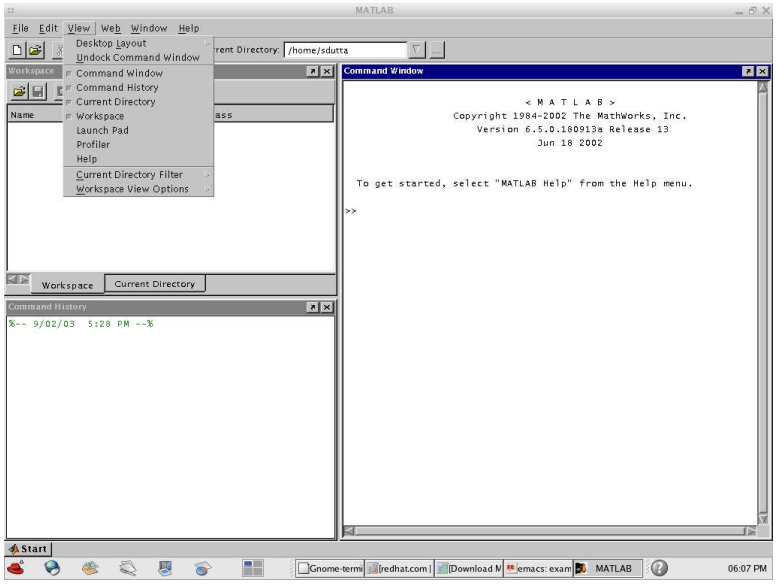


Figure 2: The MATLAB desktop: View menu

You can also move tools outside the desktop or move them back into the desktop (docking), see the Figure 3.

Note that these screenshots are from a linux machine. They may look different in a Mac OS X machine or a Windows machine.

In particular on a Mac OS X machine the `File`, `View`, etc. menu is not at the top of the MATLAB desktop but (like all other Mac OS X applications) at the top of the screen, as is shown in the Figure 4.

On a Windows XP machine the MATLAB desktop looks as in Figure 5.

You can specify certain characteristics for the desktop tools by selecting Preferences from the File menu. For example, you can specify the font characteristics for Command Window text. For more information, click the Help button in the Preferences dialog box.

In addition to the `File` menu at the top MATLAB also has a `Start` button at the bottom left corner, that has shortcuts to some commonly used activities as shown in Figure 6

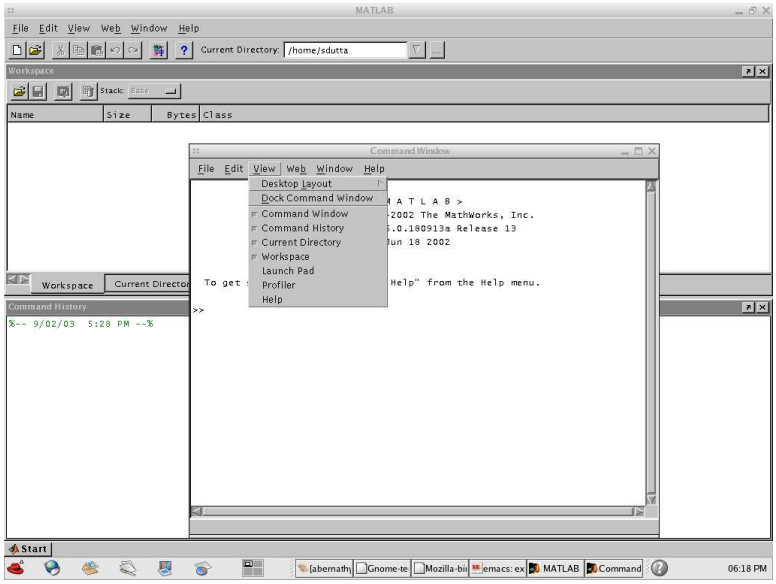


Figure 3: The MATLAB desktop: Docking menu

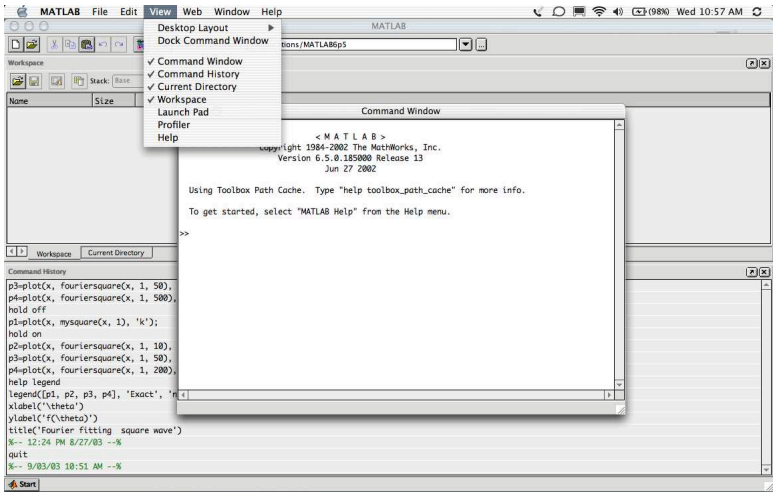


Figure 4: The MATLAB desktop: Mac OS X

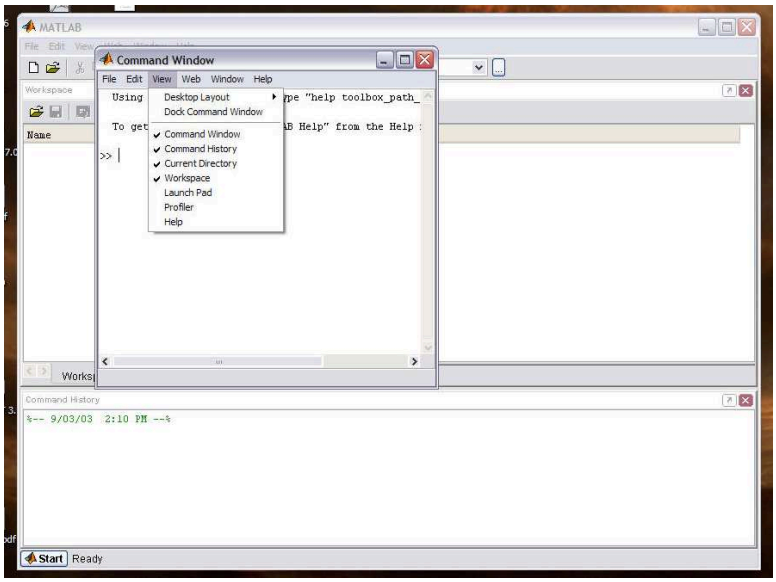


Figure 5: The MATLAB desktop: Windows XP

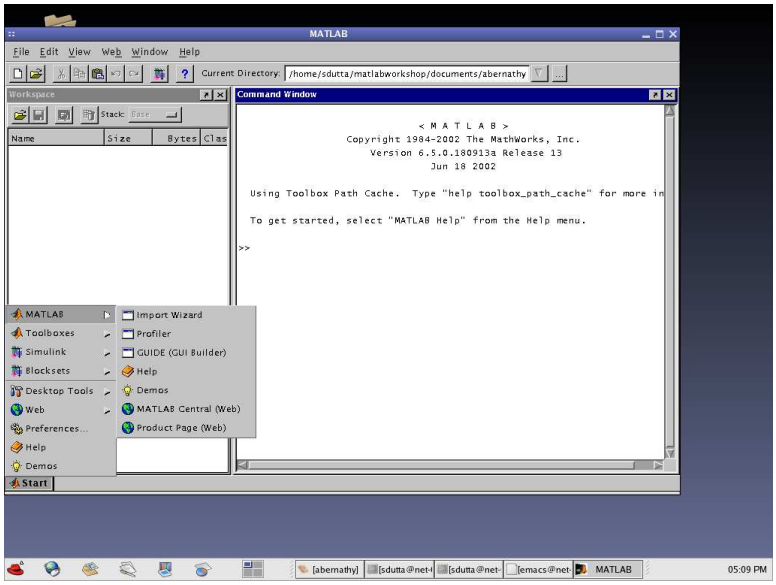


Figure 6: The MATLAB desktop: Start button

# 3 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.
```

## 3.1 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
```

## 3.2 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, $[10^s 10^{s+d} \ldots 10^f]$, 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.

## 3.3 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
```

## 3.4 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
```

## 3.5 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`.

# 4 Graphics

## 4.1 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 7.

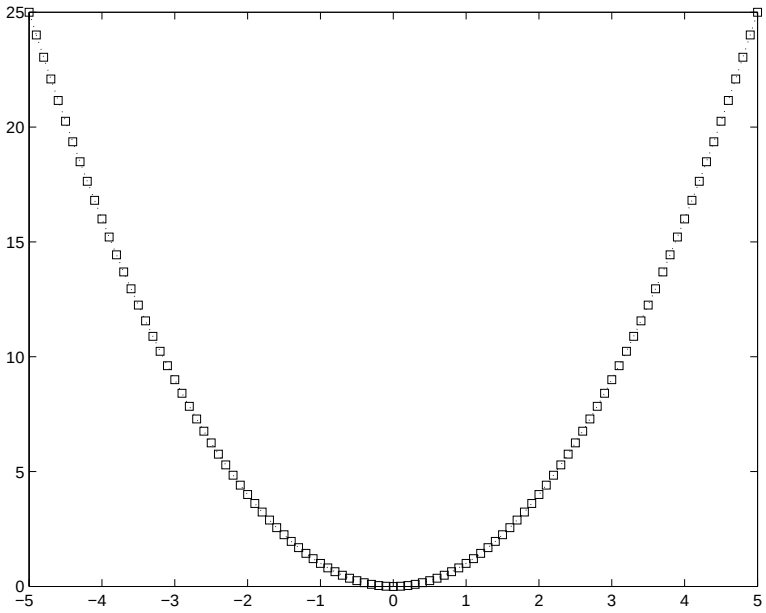


Figure 7: A simple 2D plot

To plot a second plot on top of an existing plot, use hold on. This is demonstrated in Figure 8. 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 9

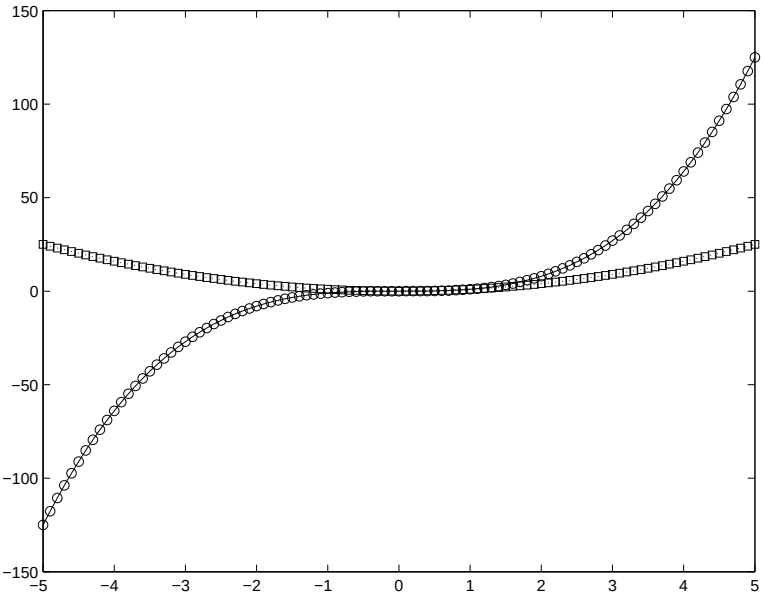


Figure 8: A 2D plot displaying overlay

## 4.2 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 10.

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 11.

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

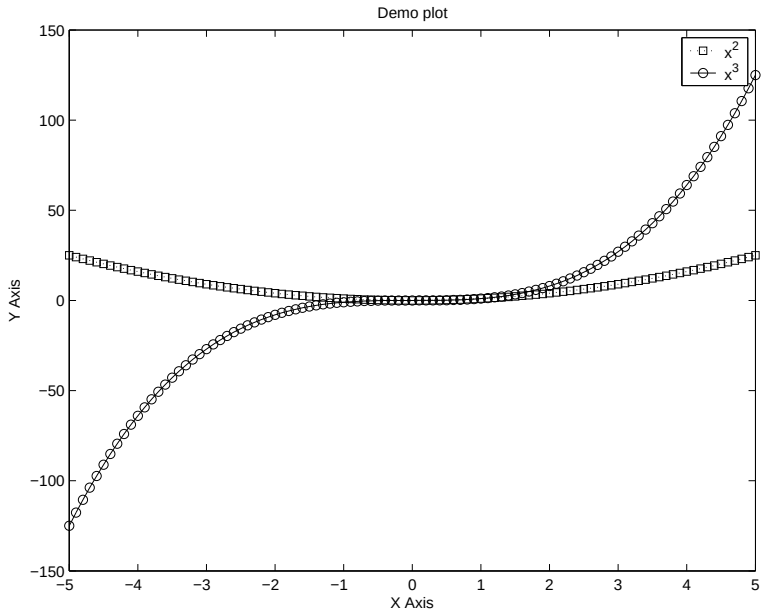


Figure 9: A 2D plot with annotations

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`.

## 4.3 Tables

Another important related need when presenting data is to produce tables. Frequently summary data can be most easily read when presented in the form of a table. This allows the most comparable numbers to be visible next to each other for quick numerical comparison. This can be easily accomplished by using the file I/O routines.

Consider for example finding the roots of a function, that is the values where the functions is equal to zero. Let us take as an example the function,

$$f(x) = x - 12x^{1/3} + 12$$

Lets use a fixed point method to find the roots. This is an iterative method where:

$$x_{k+1} = g(x_k) = x_k + f(x_k)$$

Then there will be a $x^*$ such that $x^* = g(x^*)$ which gives $f(x^*) = 0$. For the

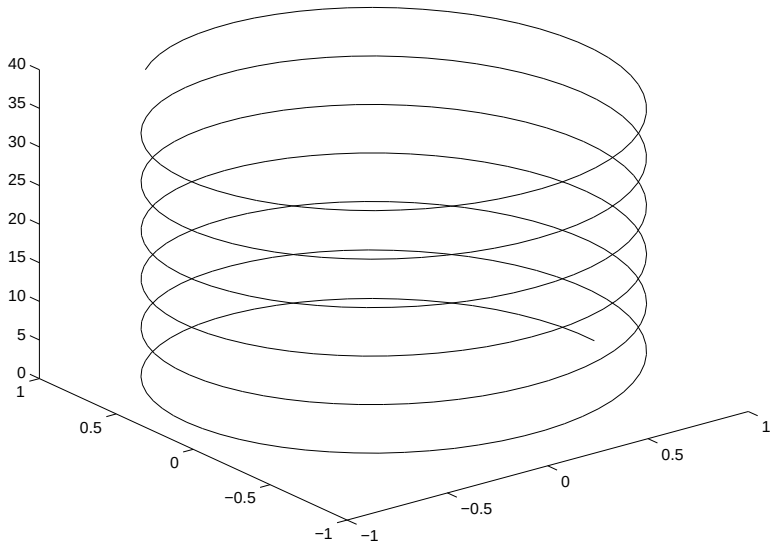


Figure 10: A simple 3D plot

function of interest let us pick three functions $g(x)$:

$$\begin{aligned} g(x_k) &= 12x_k^{1/3} - 12 \\ g(x_k) &= \left(\frac{x_k + 12}{12}\right)^3 \\ g(x_k) &= \frac{8x_k^{1/3} - 12}{1 - 4x_k^{-2/3}} \end{aligned}$$

The first function can be got by setting $f(x) = 0$ and separating the first $x$ term. The second function is by separating out the $x^{\frac{1}{3}}$ term. The last function is just a good guess.

If we plot the function $f(x)$ we will see easily see where the zeros are. We can do this using the following commands:

```
>> x=linspace(0.1, 30, 50);        % generate 50 equispaced points between 0.1 and 30
>> f=x-12*x.^(1/3)+12; % calculate the function value at each x (note the array op.)
>> z=zeros(length(x), 1); % a row vector of same size as x full of zeros
>> plot(x, f, 'b-', x, z, 'k-'); % plot x vs. f (blue line) and x vs. z (black line)
>> xlabel('x');
>> ylabel('f(x)');
```

This produces the Figure 12.

As can be seen, the function $f(x)$ has two roots, one between 1&2 and another one between 21&22. The root finding algorithm is very simple. Given a choice of $g(x)$,

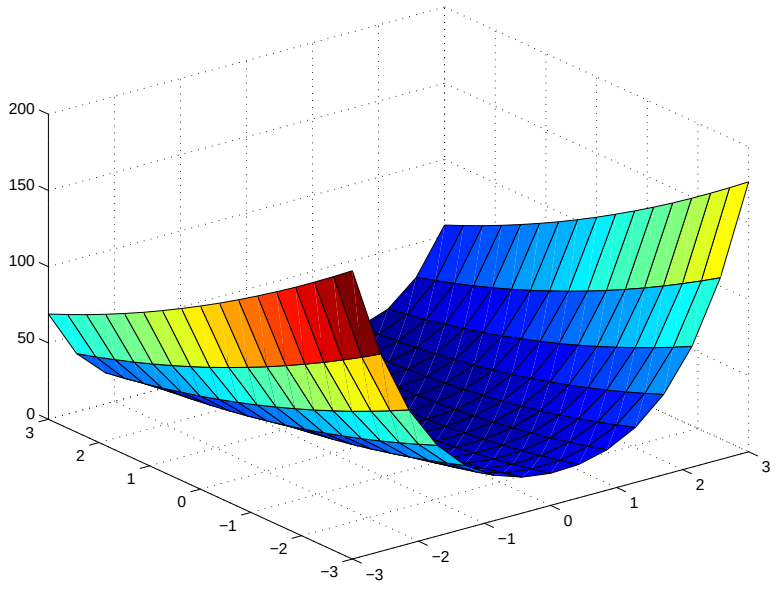


Figure 11: A 3D surface plot

```
>> xguess=1; % start with an initial guess of 1
>> g=xguess;
>> f=g-12*g.^(1/3)+12;
>> err=norm(f);        % (\sum_k f(x_k)^2)^(1/2)
>> tol=1.e-5;
>> while (err>tol) % iterate until error is small
g=12*g.^(1/3)-12;
f=g-12*g.^(1/3)+12;
err=norm(f);
disp(sprintf('Error = %f', err));
end
Error = 12.000000
Error = 27.473142
Error = 18.106115
Error = 8.829625
Error = 4.256514
Error = 2.108523
Error = 1.067871
Error = 0.548209
Error = 0.283578
Error = 0.147292
Error = 0.076671
Error = 0.039956
Error = 0.020835
Error = 0.010868
Error = 0.005670
```

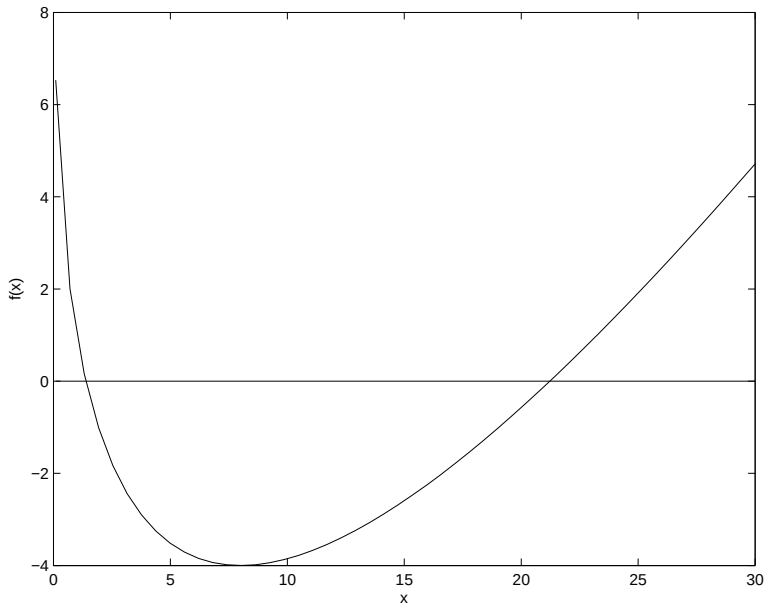


Figure 12: The roots of the function $f(x)$

```
Error = 0.002958
Error = 0.001543
Error = 0.000805
Error = 0.000420
Error = 0.000219
Error = 0.000114
Error = 0.000060
Error = 0.000031
Error = 0.000016
Error = 0.000008
>> g
g =
  21.2248 + 0.0000i
```

What we need though is to compare the results of the three functions and check on the convergence as we iterate. This is where a tabular representation of data can be particularly useful. We will use the `m-file fixed_point2.m`.

```
function fixed_point2=fixed_point2(xguess);
% function: fixed_point
%
% find succesive approximations to f(x)= x-12*x^(1/3)+12
%
% Input: starting point for x in iterative scheme
% Output: table for different iteration functions g
```

```
%
n=10;                   % no. of iterations
% open a file
fid = fopen('succ_approx.txt', 'wt');
% create a string to print it, both on the screen and into a file
line = sprintf('%4s %15s %15s %15s %15s %15s %15s', 'k',  'g1', 'f(g1)', ...
  'g2', 'f(g2)', 'g3', 'f(g3)');
disp(line);                     % print to screen
fprintf(fid, '%s\n', line);     % print to file
% start the iteration
g1=xguess;
g2=xguess;
g3=xguess;
line = sprintf('%4.0f %15.10f %15.10f %15.10f %15.10f %15.10f %15.10f', ...
  0, g1,fx(g1), g2, fx(g2), g3, fx(g3));
disp(line);
fprintf(fid, '%s\n', line);
for k=1:n
    g1=12*g1.^(1/3)-12;
    g2=((g2+12)/12).^3;
    g3=(8*g3^(1/3)-12)/(1-4*g3^(-2/3));
    line = sprintf('%4.0f %15.10f %15.10f %15.10f %15.10f %15.10f %15.10f', ...
      k, g1,fx(g1), g2, fx(g2), g3, fx(g3));
    disp(line);
    fprintf(fid, '%s\n', line);
end
fclose(fid);
end

function fx=fx(x)
% calculate the function given x
fx=x-12*x.^(1/3)+12;
end
```

We can now call the function `fixed_point` from MATLAB prompt with an argument of beginning guess for the root.

```
>> fixed_point2(1)
```

| k | g1 | f(g1) | g2 | f(g2) | g3 | f(g3) |
|---|---|---|---|---|---|---|
| 0 | 1.0000000000 | 1.0000000000 | 1.0000000000 | 1.0000000000 | 1.0000000000 | 1.0000000000 |
| 1 | 0.0000000000 | 12.0000000000 | 1.2714120370 | 0.2714120370 | 1.3333333333 | 0.1256243378 |
| 2 | -12.0000000000 | -13.7365709106 | 1.3527192196 | 0.0813071826 | 1.3879068816 | 0.0024040866 |
| 3 | 1.7365709106 | -16.5904536662 | 1.3777341143 | 0.0250148947 | 1.3889923491 | 0.0000009093 |
| 4 | 18.3270245768 | -3.5731664525 | 1.3854917428 | 0.0077576285 | 1.3889927600 | 0.0000000000 |
| 5 | 21.9001910293 | -0.2023939369 | 1.3879034437 | 0.0024117008 | 1.3889927600 | 0.0000000000 |
| 6 | 22.1025849663 | 0.2916692093 | 1.3886537660 | 0.0007503223 | 1.3889927600 | 0.0000000000 |
| 7 | 21.8109157570 | 0.2477684464 | 1.3888872595 | 0.0002334935 | 1.3889927600 | 0.0000000000 |
| 8 | 21.5631473106 | 0.1532914641 | 1.3889599259 | 0.0000726664 | 1.3889927600 | 0.0000000000 |
| 9 | 21.4098558464 | 0.0862246673 | 1.3889825412 | 0.0000226153 | 1.3889927600 | 0.0000000000 |
| 10 | 21.3236311792 | 0.0466467994 | 1.3889895796 | 0.0000070384 | 1.3889927600 | 0.0000000000 |

In addition a file `succ_approx.txt` is created which looks like the following:

```
>> type succ_approx.txt
```

| k | g1 | f(g1) | g2 | f(g2) | g3 | f(g3) |
|---|---|---|---|---|---|---|
| 0 | 1.0000000000 | 1.0000000000 | 1.0000000000 | 1.0000000000 | 1.0000000000 | 1.0000000000 |
| 1 | 0.0000000000 | 12.0000000000 | 1.2714120370 | 0.2714120370 | 1.3333333333 | 0.1256243378 |
| 2 | -12.0000000000 | -13.7365709106 | 1.3527192196 | 0.0813071826 | 1.3879068816 | 0.0024040866 |
| 3 | 1.7365709106 | -16.5904536662 | 1.3777341143 | 0.0250148947 | 1.3889923491 | 0.0000009093 |
| 4 | 18.3270245768 | -3.5731664525 | 1.3854917428 | 0.0077576285 | 1.3889927600 | 0.0000000000 |
| 5 | 21.9001910293 | -0.2023939369 | 1.3879034437 | 0.0024117008 | 1.3889927600 | 0.0000000000 |
| 6 | 22.1025849663 | 0.2916692093 | 1.3886537660 | 0.0007503223 | 1.3889927600 | 0.0000000000 |
| 7 | 21.8109157570 | 0.2477684464 | 1.3888872595 | 0.0002334935 | 1.3889927600 | 0.0000000000 |
| 8 | 21.5631473106 | 0.1532914641 | 1.3889599259 | 0.0000726664 | 1.3889927600 | 0.0000000000 |
| 9 | 21.4098558464 | 0.0862246673 | 1.3889825412 | 0.0000226153 | 1.3889927600 | 0.0000000000 |
| 10 | 21.3236311792 | 0.0466467994 | 1.3889895796 | 0.0000070384 | 1.3889927600 | 0.0000000000 |

It is now easy to see from the table that the function `g3` converges very rapidly to the lower root while `g1` and `g2` converge much slower to the higher and lower roots respectively.

# 5 Programming with MATLAB

## 5.1 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 13.

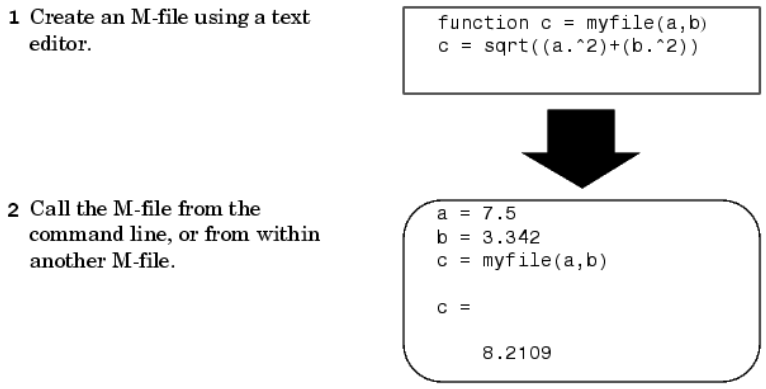


Figure 13: Steps in using a m-file

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.

## 5.2 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.

## 5.3 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
```

## 5.4 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);
```

# 6 MATLAB workspace and File I/O

## 6.1 MATLAB workspace

Note that as you work in the MATLAB command window, MATLAB remembers your commands. You can always recall your previous commands using the up arrow key. This is true of all the variables created through these commands as well. For example:

```
>> x=-5:0.1:5;
>> sqr=x.^2;
>> sqrnorm = norm(sqr);
>> sqrnorm
sqrnorm =
  114.6008
>> pl1=plot(x, sqr, 'r:s');
>> sqrnorm
sqrnorm =
  114.6008
```

The variable sqrnorm remained in the workspace. The keyword `who` recalls the list of variables in the workspace.

```
>> who

Your variables are:

A      i      pl     rho    x      xx1    z
ans    j      pl1    sqr    x1     xx2
cub    k      pl2    theta  x2     y
```

The keyword `whos` gives more detailed information about the workspace.

```
>> whos
  Name          Size                    Bytes  Class

  pl1           1x1                         8  double array
  sqr           1x101                     808  double array
  sqrnorm       1x1                         8  double array
  x             1x101                     808  double array
Grand total is 204 elements using 1632 bytes
```

The keyword `clear` removes the variable from workspace. The command `clear` all by itself clears the workspace of all variables. Use this command with caution as there is no undo!

```
>> clear z
>> z
??? Undefined function or variable 'z'.
```

## 6.2 Function workspace

Script files (i.e., m-files with no functions) share its workspace with the base workspace. However functions have their own workspace. So variables defined within the function has no value outside the function. This is usually referred to as encapsulation in programming parlance. This is a good thing that encourages well designed programs that are readable and manageable. However, in case, it is necessary for variables within a function workspace be available in the base workspace, it is possible to use the global keyword. A related concept is the persistent keyword which allows variables in a function workspace be available in two different invocations of the function (although still not available in the base workspace).

## 6.3 Native data files

The existing workspace data can be saved to a file using save. To save only some variables to a specified file use save filename var1 var2. The code fragment below demonstrates saving the workspace to a file called workspace (this produces a binary file called workspace.mat in the working directory). You can then quit MATLAB and restart it. Obviously now the workspace is empty. However it is possible to load in the saved workspace using load as shown below.

```
>> save workspace
%-- 8/12/03 11:00 AM --%   <-- quit MATLAB
>> load workspace          <-- new MATLAB session
>> sqrnorm=norm(sqr)           <-- sqr read in from saved workspace
sqrnorm =

  114.6008
```

## 6.4 Data import and export

In addition to the native files format, MATLAB provides a large set of file I/O functions.

```
>> help fileformats

  Readable file formats.

   Data formats                     Command    Returns
    MAT  - MATLAB workspace         load       Variables in file.
    CSV  - Comma separated numbers  csvread    Double array.
    DAT  - Formatted text           importdata Double array.
    DLM  - Delimited text           dlmread    Double array.
    TAB  - Tab separated text       dlmread    Double array.

   Spreadsheet formats
    XLS  - Excel worksheet          xlsread    Double array and cell array.
    WK1  - Lotus 123 worksheet      wk1read    Double array and cell array.
```

```
Scientific data formats
 CDF  - Common Data Format               cdfread    Cell array of CDF records
 FITS - Flexible Image Transport System  fitsread   Primary or extension table data
 HDF  - Hierarchical Data Format         hdfread    HDF or HDF-EOS data set

Movie formats
 AVI  - Movie                    aviread    MATLAB movie.

Image formats
 TIFF - TIFF image               imread     Truecolor, grayscale or indexed image(s).
 PNG  - PNG image                imread     Truecolor, grayscale or indexed image.
 HDF  - HDF image                imread     Truecolor or indexed image(s).
 BMP  - BMP image                imread     Truecolor or indexed image.
 JPEG - JPEG image               imread     Truecolor or grayscale image.
 GIF  - GIF image                imread     Indexed image.
 PCX  - PCX image                imread     Indexed image.
 XWD  - XWD image                imread     Indexed image.
 CUR  - Cursor image             imread     Indexed image.
 ICO  - Icon image               imread     Indexed image.
 RAS  - Sun raster image         imread     Truecolor or indexed.
 PBM  - PBM image                imread     Grayscale image.
 PGM  - PGM image                imread     Grayscale image.
 PPM  - PPM image                imread     Truecolor image.

Audio formats
 AU   - NeXT/Sun sound           auread     Sound data and sample rate.
 SND  - NeXT/Sun sound           auread     Sound data and sample rate.
 WAV  - Microsoft Wave sound     wavread    Sound data and sample rate.
```

It is also possible to do low-level file I/O using the standard, file open, write, read and file close functions. The code fragment below demonstrates a trivial use of these functions.

```
>> fid = fopen ('square_mat.txt', 'wt');
>> fprintf(fid, '%s\n', 'This is a square matrix');
>> fprintf(fid, '%i\t%i\t%i\n', [1 2 3; 4 5 6; 7 8 9]');
>> fclose(fid);
```

This produces a file square_mat.txt in the current working directory which contains:

```
This is a square matrix
1       2       3
4       5       6
7       8       9
```

# 7 Ordinary Differential Equations

## 7.1 Second order homogeneous linear equation with constant coefficients

Consider the second order linear equation:

$$\frac{d^2y}{dx^2} + a\frac{dy}{dx} + by = 0$$

where $a$ and $b$ are constants. This is second order because the largest derivative of the dependant variable $y$ is 2. It is homogeneous because the right hand side is 0.

If we choose a solution of the form, $y = \exp(kx)$, where $k = \text{const}$ then,

$$\frac{dy}{dx} = k\exp(kx), \qquad \frac{d^2y}{dx^2} = k^2\exp(kx)$$

and for this to be a solution of the differential equation the *auxiliary equation* $k^2 + ak + b = 0$ must be true. This equation (in general) will have two complex roots, $k_1$ and $k_2$.

$$\begin{aligned} k_1 &= -a/2 + 1/2\sqrt{a^2 - 4b} \\ k_2 &= -a/2 - 1/2\sqrt{a^2 - 4b} \end{aligned}$$

The general solution of the equation then is:

$$y(x) = C_1\exp(k_1x) + C_2\exp(k_2x)$$

Let us now add the initial conditions:

$$y(0) = d; \quad \left.\frac{dy}{dx}\right|_{x=0} = g$$

Then, $C_1 + C_2 = d$ and $k_1C_1 + k_2C_2 = g$. Solving the simultaneous equation:

$$\begin{aligned} C_1 + C_2 &= d \\ C_1k_1 + C_2k_2 &= g \end{aligned}$$

we get the constants

$$\begin{aligned} C_1 &= (g - dk_2)/(k_1 - k_2) \\ C_2 &= (dk_1 - g)/(k_1 - k_2) \end{aligned}$$

In MATLAB we can use the symbolic package as following:

```
>> syms a b c d g k1 k2 y       % declare variables as symbolic
>> k1=-(a/2)+1/2*sqrt(a^2-4*b);
>> k2=-(a/2)-1/2*sqrt(a^2-4*b);
>> y=(g-d*k2)/(k1-k2)*exp(k1*x)+(d*k1-g)/(k1-k2)*exp(k2*x);
```

In the special case:

$$a = 1 \quad b = 1 \quad d = 1 \quad g = 0$$

we get:

```
>> a=1;
>> b=1;
>> d=1;
>> g=0;
>> pretty(eval(y))

                1/2                          1/2
  (1/2 - 1/6 I 3   ) exp((- 1/2 + 1/2 I 3   ) x)

                           1/2                          1/2
         + (1/2 + 1/6 I 3   ) exp((- 1/2 - 1/2 I 3   ) x)
>> x=0:0.1:6*pi;
>> g=-10;
>> y1=eval(y);
>> g=0;
>> y2=eval(y);
>> g=10;
>> y3=eval(y);
>> plot(x, y1, 'k-', x, y2, 'k--', x, y3, 'k-.');
>> legend('g=-10', 'g=0', 'g=10');
>> xlabel('x');
>> ylabel('y');
>> title('Damped oscillator')
```

This produces the Figure 14 displaying the evolution of the function $y$. This is exactly as expected, with the displacement $y$ starting off at 1 at $x = 0$ and the damping term ($\propto dy/dx$) bringing the displacement very rapidly down to zero. Depending on the initial value of $dy/dx$ at $x = 0$, the transient behaves differently. In all three cases though this is a overdamped system, where the damping rapidly brings the system to a stop.

It is also possible to use MATLAB's own differential equation solver `dsolve` to get the above solution almost trivially.

```
>> S=dsolve('D2y+a*Dy+b*y=0', 'y(0)=d, Dy(0)=g');
>> t=x;
>> plot(x, eval(S), 'k-', x, y3, 'kx');
```

Note the line `t=x;` has to be there because `dsolve` returns its results in terms of `t` and not `x`. As can be seen from Figure 15 the solution we had derived and the result of `dsolve` are identical.

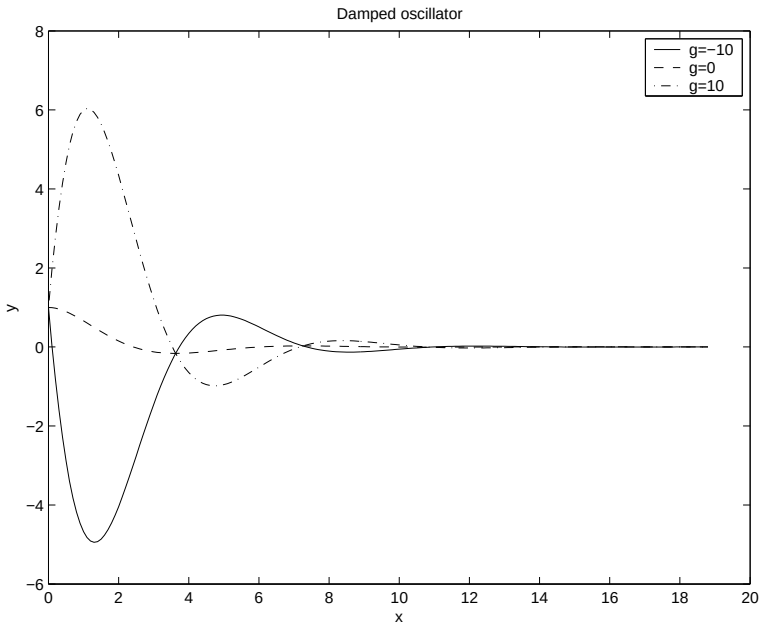


Figure 14: Solution of the ordinary 2nd order homogeneous differential equation

## 7.2 Non-homogeneous 2nd order differential equations

We can now use `dsolve` to solve for a non-homogeneous 2nd order differential equation. In particular we want to look at the equation:

$$\frac{d^2y}{dx^2} + a\frac{dy}{dx} + by = c$$

with the same initial conditions as before:

$$y(0) = d; \quad \left.\frac{dy}{dx}\right|_{x=0} = g$$

Then again using the symbolic math package in MATLAB:

```
>> syms a b c d g y
>> y=dsolve('D2y+a*Dy+b*y=c', 'y(0)=d, Dy(0)=g', 'x');
>> y
y =
1/2*exp((-1/2*a+1/2*(a^2-4*b)^(1/2))*x)*(-a*c+a*d*b-(a^2-4*b)^(1/2)*c+
(a^2-4*b)^(1/2)*d*b+2*g*b)/(a^2-4*b)^(1/2)/b+
1/2*exp((-1/2*a-1/2*(a^2-4*b)^(1/2))*x)*(a*c-a*d*b-(a^2-4*b)^(1/2)*c+
(a^2-4*b)^(1/2)*d*b-2*g*b)/(a^2-4*b)^(1/2)/b+1/b*c
```

Note the last argument `x` to the function `dsolve`. This tells MATLAB the solution `y` should in terms of the explicit variable `x` instead of the default `t`.

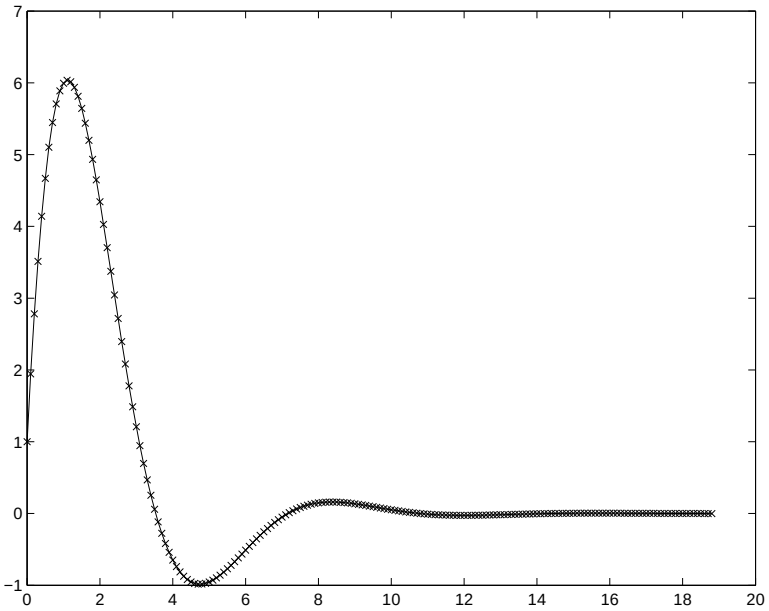


Figure 15: Comparison of `dsolve` and the traditional method of solving

If we now look at the same special case:

```
>> a=1;
>> b=1;
>> c=1;
>> d=1;
>> x=0:0.1:6*pi;
>> g=-1;
>> y1=eval(y);
>> g=0;
>> y2=eval(y);
>> g=1;
>> y3=eval(y);
>> plot(x, y1, 'k-', x, y2, 'k--', x, y3, 'k-.');
```

This gives us the Figure 16. As can be seen it is similar to the homogeneous case. The exception is that because of the forcing constant on the right hand side, the solution asymptotes to that constant rather than 0.

A simple way of quickly visualizing a solution from `dsolve` is the `ezplot` function. The script below demonstrates the use of the `ezplot` function.

```
>> syms a b c d g y x
>> y=dsolve('D2y+a*Dy+b*y=c', 'y(0)=d, Dy(0)=g', 'x');
>> a=1;b=1;c=1;d=1;g=1;
```

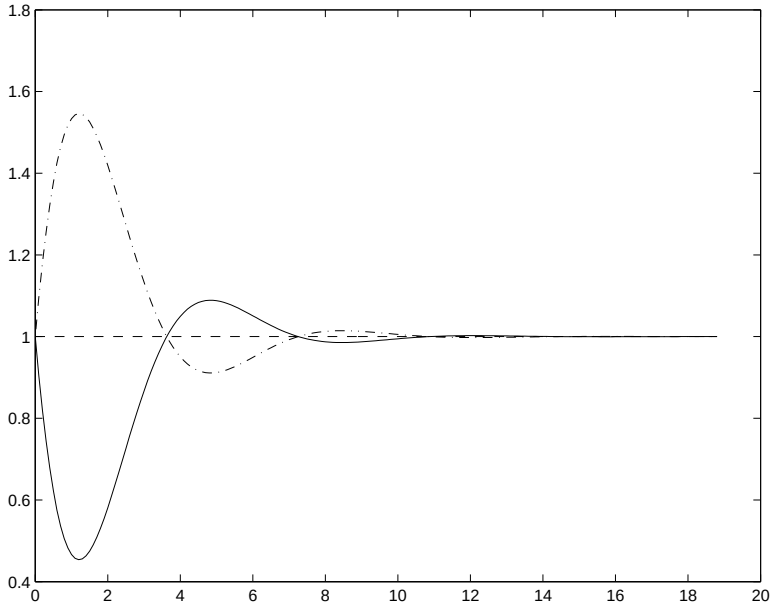


Figure 16: Solution of an inhomogeneous equation

```
>> ezplot(eval(y), [0, 6*pi]);
>> axis([-0.5 20. 0.8 1.6]);
```

This produces Figure 17.

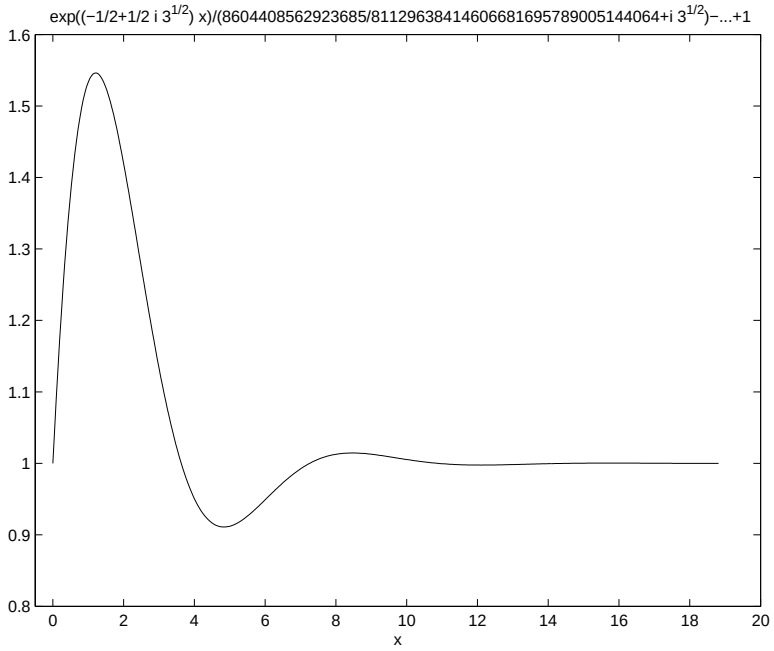


Figure 17: Using the `ezplot` function

# 8 Sparse Matrices

## 8.1 Storage of data

Matrices and linear algebra are frequently a way in which to analyze complex problems that can be framed in terms of partial differential equations. These equations can then be discretized, using some scheme so the system can be represented by discrete fields (instead of continuous fields). Discretizations of partial differential equations almost always lead to *sparse* matrices. These are matrices where most of the elements are zeros. Although it is still possible to use the same algorithms for sparse as dense matrices (matrices with few zero elements) it is very inefficient to do so. In particular some problems are so large that to write out the entire matrix would require prohibitive amounts of memory. It is therefore critical to take advantage of the sparseness of the matrix.

The first advantage that can be taken is in the storage of data. Normally a matrix in MATLAB is stored by writing out every element. A sparse matrix stores its data using three vectors. Only non-zero elements are stored. The length of the three vectors is called `nzmax`. This must be greater than the number of non-zero elements in the matrix. The first vector is just the list of (`nnz`) non-zero elements. The second vector is the row index of the non-zero elements. There are `nzmax` elements in this vector. The third vector is start and end column index of non-zero elements in each row.

For example consider the matrix:

$$\begin{bmatrix} 1 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 6 & 0 \\ 0 & 3 & 2 & 0 \end{bmatrix}$$

```
>> A=[1 0 3 0; 0 4 0 0; 0 0 6 0; 0 3 2 0];
>> A1=sparse(A);
>> A1
A1 =
   (1,1)        1
   (2,2)        4
   (4,2)        3
   (1,3)        3
   (3,3)        6
   (4,3)        2
>> whos
  Name      Size                    Bytes  Class

  A         4x4                       128  double array
  A1        4x4                        92  double array (sparse)

  Grand total is 22 elements using 220 bytes
>> nzmax(A1)
```

```
ans =
     6
>> nnz(A1)
ans =
     6
```

The reason for a possible difference between `nzmax` and `nnz` is efficiancy. It is more efficient to allocate more space initially and allow the matrix to expand to fill it than to allocate exactly what is needed. However the tradeoff with the extra memory needed against the improved performance changes at some large matrix size. In our tiny example `nzmax` and `nnz` are the same.

Note that if the matrix is complex, then there is a fourth vector which is the list of imaginary numbers to store the matrix data.

## 8.2 Creating sparse matrices

We saw in the previous section `sparse` is one way of creating sparse matrices from dense ones. But clearly this is not a very efficient way of creating sparse matrices.

Another usage of `sparse` is to create the matrix from its member vectors. For example for our our example matrix:

```
>> A2=sparse([1 2 4 1 3 4], [1 2 2 3 3 3], [1 4 3 3 6 2], 4, 4);
>> A2
A2 =

   (1,1)        1
   (2,2)        4
   (4,2)        3
   (1,3)        3
   (3,3)        6
   (4,3)        2
>> A3=sparse([1 2 4 1 3 4], [1 2 2 3 3 3], [1 4 3 3 6 2], 4, 4, 10);
>> A3
A3 =

   (1,1)        1
   (2,2)        4
   (4,2)        3
   (1,3)        3
   (3,3)        6
   (4,3)        2
>> nzmax(A2)
ans =
     6
>> nzmax(A3)
ans =
```

```
    10
>> nnz(A2)
ans =
     6
>> nnz(A3)
ans =
     6
```

The second usage of `sparse` sets `nzmax` to 10 instead of the default 6 (the number of non-zero elements). This would be useful if we were to grow the number of non-zero elements in `A3`.

Another way of creating sparse matrices is to use the `spdiags` which uses the *Compressed-Diagonal storage mode*. A matrix of order $n$ has $2n-1$ diagonals.

$$A = \begin{array}{c} \\ d_{-1} \\ d_{-2} \\ d_{-3} \\ \vdots \\ d_{1-n} \end{array} \begin{array}{c} \begin{array}{ccccccc} d_0 & d_1 & d_2 & d_3 & \cdots & d_{n-1} \end{array} \\ \left[\begin{array}{ccccc} a_{11} & a_{12} & a_{13} & & \\ a_{21} & a_{22} & a_{23} & \cdots & a_{1n} \\ a_{31} & a_{32} & a_{33} & & \\ & \vdots & & \ddots & \vdots \\ & a_{n1} & & \cdots & a_{nn} \end{array}\right] \end{array}$$

The matrix $A$ can then stored using two matrices $B$ and $d$, such that each diagonal (including its zero elements) of $A$ that has at least one non-zero element is a column of $B$. These columns are padded such that:

- Each superdiagonal ($k > 0$) which has $n - k$ elements, is padded with $k$ leading zeros.
- The main diagonal (which has $n$ elements), isn't padded.
- Each subdiagonal ($k < 0$), which has $n - |k|$ elements, is padded with $k$ trailing zeros.

And $d$ is a vector with the list of the diagonal numbers corresponding to the columns of $B$.

For example for a matrix $A$,

$$A = \begin{bmatrix} 1 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 6 & 0 \\ 0 & 3 & 2 & 0 \end{bmatrix}$$

we have

$$B = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 3 & 0 & 4 & 0 \\ 0 & 2 & 6 & 3 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

and

$$d = \begin{bmatrix} -2 & -1 & 0 & 2 \end{bmatrix}$$

So we can construct our matrix $A$ using:

```
>> B=[0 0 1 0; 3 0 4 0; 0 2 6 3; 0 0 0 0];
>> d=[-2 -1 0 2];
>> A2=spdiags(B, d, 4, 4)

A2 =

   (1,1)        1
   (2,2)        4
   (4,2)        3
   (1,3)        3
   (3,3)        6
   (4,3)        2
```

Some of the built in functions to create dense matrices, like, `rand`, `eye` etc. have their sparse equivalents.

$$\begin{aligned} rand(m,n) &\Rightarrow sprand(m,n) \\ eye(m,n) &\Rightarrow speye(m,n) \\ zeros(m,n) &\Rightarrow sparse(m,n) \end{aligned}$$

There is no equivalent (for obvious reasons) of `ones` that produces a sparse matrix. However there is a function `spones` which takes a sparse matrix as an argument and preserves its structure, but replaces each non-zero element with 1.

In addition there is the command `spconvert` that will use load to read in data (in row, column index and value format) and convert it to a sparse matrix.

## 8.3 Viewing sparse matrices

MATLAB has a script `spy` to graphically view sparse matrices. For example `spy(A2)` produces the Figure 18. This isn't a very interesting picture. If we use `spy` to look at more interesting matrices we can see patterns emerge.

For example Figure 19 shows a matrix that comes with MATLAB.

A particularly useful function is `find`. This returns the list of indices and value of all non-zero elements of a matrix, regardless whether the matrix a dense or sparse. For example:

```
>> [i, j, v]=find(A2)
i =

     1
     2
     4
     1
     3
     4
j =
```

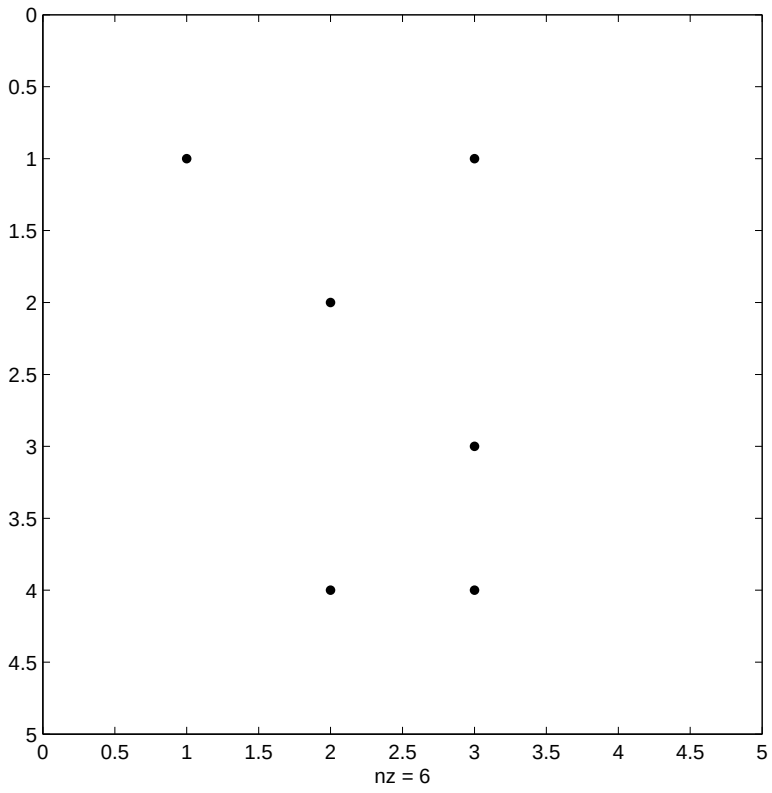


Figure 18: Viewing a sparse matrix graphically

```
     1
     2
     2
     3
     3
     3

v =

     1
     4
     3
     3
     6
     2
```

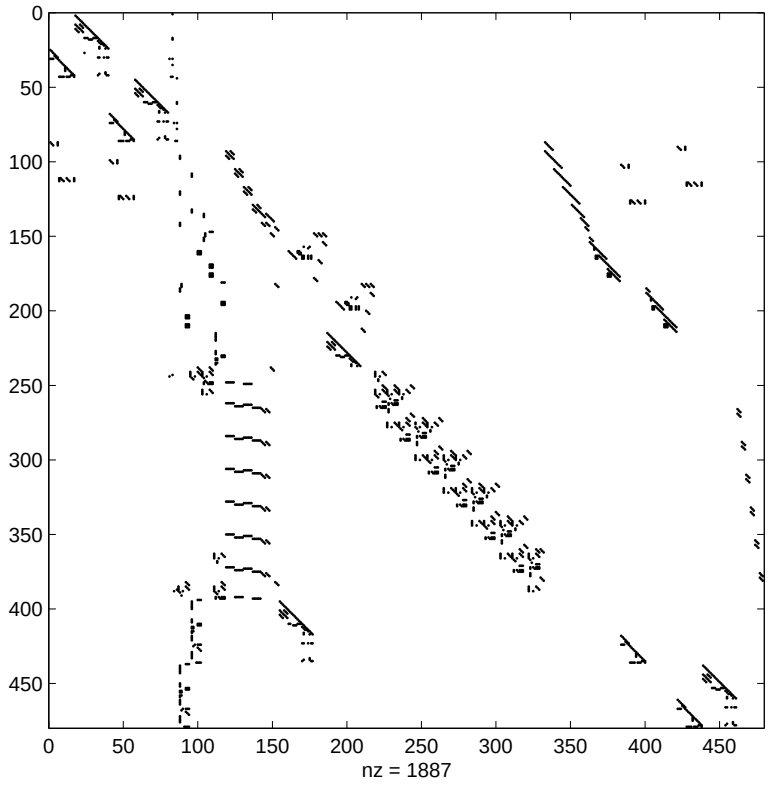


Figure 19: Viewing a large sparse matrix graphically

## 8.4 Sparse matrix computations

A sparse matrix may stop being a sparse matrix when operated on if it is not possible to preserve sparsity in that operation. This can potentially produce a very large matrix inadvertantly, so it is important to keep this in mind when operating on a sparse matrix. Functions that act on matrices and return a vector or a scalar return a dense matrix regardless if the original matrix was a sparse or dense. Note, the original matrix is unaffected.

Most unary operations (operations not involving a second matrix) preserve the sparsity of the matrix. For example `max(A)` is a sparse matrix if `A` is a sparse matrix. Obviously, the functions `sparse` and `full` change the sparsity of the matrix.

Binary operations (operations between two matrices) preserves the sparsity of the matrix if possible. So `A.*B` is sparse if either `A` or `B` are sparse. But `A+B` is dense if one of the two `A` or `B` are dense.

## 8.5 Reordering of matrices

The simple way of reordering a matrix is to use the permutation vector. That is the vector that lists the new ordering of the rows or columns of the matrix. Using our matrix `A2` from before:

```
>> full(A2)
ans =
     1     0     3     0
     0     4     0     0
     0     0     6     0
     0     3     2     0
>> p=[4 2 3 1];
>> A2(p,:)
ans =
   (4,1)        1
   (1,2)        3
   (2,2)        4
   (1,3)        2
   (3,3)        6
   (4,3)        3
>> full(A2(p,:))
ans =
     0     3     2     0
     0     4     0     0
     0     0     6     0
     1     0     3     0
```

It is also possible to use a permutation matrix to represent the permutation. It is possible to convince oneself that the permutation matrix `P` is simply `I(p,:)` where `I` is the identity matrix.

```
>> I=eye(4,4);
>> P=I(p,:);
>> full(P*A2)
ans =
     0     3     2     0
     0     4     0     0
     0     0     6     0
     1     0     3     0
```

A particularly useful function is `colperm` which returns a permutation vector such that on permutation the matrix has its column ordered in increasing number of non-zero elements.

Some other ordering functions are `symrcm`, `symamd` and `symmmd` that work on symmetric matrices and `colamd` and `colmmd` that work on non-symmetric matrices.

# 9 Numerical solutions of Ordinary Differential Equations

Consider a second order differential equation:

$$\frac{d^2y}{dx^2} + a(x)\frac{dy}{dx} + b(x)y = c(x)$$

where $a$ and $b$ are functions of $x$. This is second order because the largest derivative of the dependant variable $y$ is 2.

This can be written as two coupled first order differential equations:

$$\begin{aligned} \frac{dy}{dx} &= z \\ \frac{dz}{dx} + a(x)z + b(x)y &= c(x) \end{aligned}$$

We can then focus, momentarily, on a single first order ODE.

$$\frac{dy}{dx} = f(x, y)$$

To the simplest approximation a solution would be the Euler's approximation:

$$y_{n+1} = y_n + hf(x_n, y_n)$$

This is exact if $y(x)$ is linear in $x$. In most other cases however this yields pretty poor results. In addition this method can become unstable, i.e., small errors would accumulate and result in a final $y_n$ very different from the actual $y(x_n)$. An improvement to it is the Runge-Kutta $2^{\text{nd}}$ order method:

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1) \\ y_{n+1} &= y_n + k_2 + O(h^3) \end{aligned}$$

where $O(h^3)$ refers to terms of order $x^3$. Effectively, this introduces an intermediat step in the Euler's method in order to refine our guess of the change of $y$ with $x$.

It is possible to generalize this to $n$ orders as:

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + a_2h, y_n + b_{21}k_1) \\ &\cdots \\ k_m &= hf(x_n + a_nh, y_n + b_{m1}k_1 + b_{m2}k_2 + \cdots + b_{m(m-1)}k_{m-1}) \\ y_{n+1} &= y_n + c_1k_1 + c_2k_2 + \cdots + c_mk_m + O(h^m) \end{aligned} \tag{1}$$

A particularly useful aspect of this method is that a different choice of constants $c_i$ results in a different order approximation to y:

$$y^*_{n+1} = y_n + c^*_1 k_1 + c^*_2 k_2 + \cdots + c^*_m k_m + O(h^{m-1})$$

It is thus possible to get an error estimate for $y_{n+1}$ without extra evaluations of the function $f(x, y)$:

$$\Delta \equiv y_{n+1} - y^*_{n+1} = \sigma^m_{i=1}(c_i - c^*_i)k_i$$

The popular versions of these Runge-Kutta approximate solutions are, (4)5 and (2)3. MATLAB implements both these forms, as `ode45` and `ode23` respectively. In either of these functions MATLAB uses the error estimate to modify the step size.

Note that both these are for non-stiff ODE's, where non-stiff means the differential equations have solutions that have a single "timescale" (or at least multiple "timescales" are are not very different from each other). Here "timescale" refers to the scale of the independant variable $x$ over which $y$ changes significantly.

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 `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 y=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,

$$\begin{aligned} y_1 &= y \\ y_2 &= \frac{dy}{dt} \end{aligned}$$

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

$$\begin{aligned} \frac{dy_1}{dt} &= y_2 \\ \frac{dy_2}{dt} &= -2\xi y_2 - y_1 + h(t) \end{aligned}$$

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 $h(t) = 1$.

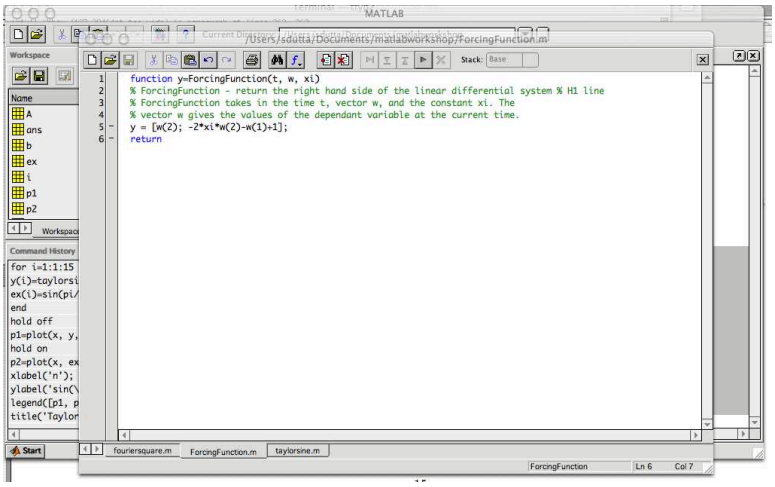


Figure 20: Matlab's built in editor

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

```
function y=ForcingFunction(t, w, xi)
% ForcingFunction - return the right hand side of the system
% 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];
return
```

save this as a file ForcingFunction.m. This file may be created using MATLAB's own editor as displayed in the Figure 20.

Then run the following commands:

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

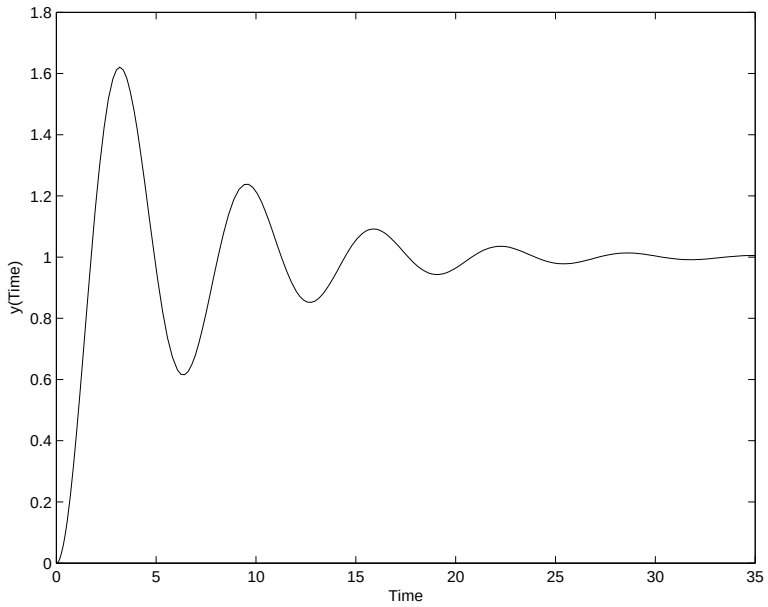


Figure 21: Solution of the differential system

You should get the Figure 21 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.

# 10 SIMULINK

SIMULINK is an extension to MATLAB which uses a icon-driven interface for the construction of a block diagram representation of a process. A block diagram is simply a graphical representation of a process (which is composed of an input, the system, and an output).

Typically, the MATLAB m-file ode45 is used to solve sets of linear and nonlinear ordinary differential equations. The "traditional" numerical methods approach is used, e.g. supply the equations to be solved in a function file, and use a general purpose equation solver (linear or nonlinear algebraic, linear or nonlinear differential equation, etc.) which "calls" the supplied function file to obtain the solution. One of the reasons why MATLAB is relatively easy to use is that the "equation solvers" are supplied for us, and we access these through a command line interface (CLI). However, SIMULINK uses a graphical user interface (GUI) for solving process simulations. Instead of writing MATLAB code, we simply connect the necessary "icons" together to construct the block diagram. The "icons" represent possible inputs to the system, parts of the systems, or outputs of the system. SIMULINK allows the user to easily simulate systems of linear and nonlinear ordinary differential equations. Many of the features of SIMULINK are user-friendly due to the icon-driven interface, yet it is important to spend some time experimenting with SIMULINK and its many features.

## 10.1 Getting Started in Simulink

SIMULINK is an icon-driven state of the art dynamic simulation package that allows the user to specify a block diagram representation of a dynamic process. Assorted sections of the block diagram are represented by icons which are available via various "windows" that the user opens (through double clicking on the icon). The block diagram is composed of icons representing different sections of the process (inputs, state-space models, transfer functions, outputs, etc.) and connections between the icons (which are made by "drawing" a line connecting the icons). Once the block diagram is "built", one has to specify the parameters in the various blocks, for example the gain of a transfer function. Once these parameters are specified, then the user has to set the integration method (of the dynamic equations), stepsize, start and end times of the integration, etc. in the simulation menu of the block diagram window.

In order to use SIMULINK start a MATLAB session (click on the MATLAB button). Once MATLAB has started up, type simulink (SMALL LETTERS!) at the MATLAB prompt (>>) followed by a carriage return (press the return key). A SIMULINK window should appear shortly, with the several icons: Sources, Sinks, Discrete, etc. This is shown in the Figure 22.

Next, go to the file menu in this window and choose New in order to begin building the block diagram representation of the system of interest.

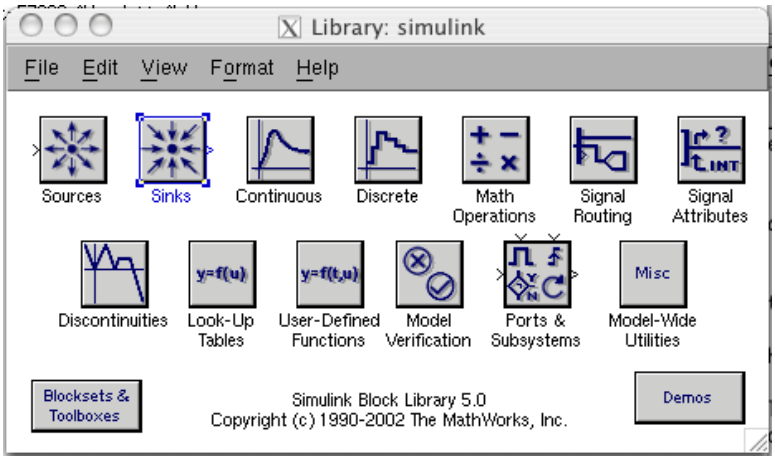


Figure 22: The SIMULINK library

## 10.2 Block Diagram Construction

As mentioned previously, the block diagram representation of the system is made up of various type of icons. Basically, one has to specify the model of the system (state space, discrete, transfer functions, nonlinear ODE's, etc), the input (source) to the system, and where the output (sink) of the simulation of the system will go. Open up the Sources, Sinks, and Linear windows by clicking on the appropriate icons. Note the different types of sources (step function, sinusoidal, white noise, etc.), sinks (scope, file, workspace), and linear systems (transfer function, state space model, etc.).

Let us illustrate this by trying to model a simple harmonic oscillator (a pendulum). The equation of motion of the pendulum is:

$$ml^2\frac{d^2\theta}{dt^2} + \gamma\frac{d\theta}{dt} + mgl\sin\theta = Af(\omega_D t)$$

In order to represent this as a block diagram, re-write the equation of motion in the following way:

$$\begin{aligned}\frac{d^2\theta}{dt^2} &= Af(\omega_D t) - \frac{\gamma}{ml^2}\frac{d\theta}{dt} - \frac{g}{l}\sin\theta \\ \theta &= \int\int\left[Af(\omega_D t) - \frac{\gamma}{ml^2}\frac{d\theta}{dt} - \frac{g}{l}\sin\theta\right]\end{aligned}$$

In the above $\int$ is the integration operator. For our example we will consider the case where the forcing function $Af(\omega_D t) = \sin(t)$, i.e., $A = 1$, $\omega_D = 1$ and $f(\omega_D t) = \sin(\omega_D t)$. In addition we will take the case where $m = 1$, $\gamma = 10$ and $l = 1$. In S.I. $g = 9.81 kgm/s^2$.

Let us now try to construct this system using Simulink. Following the instructions given before start a new block diagram.

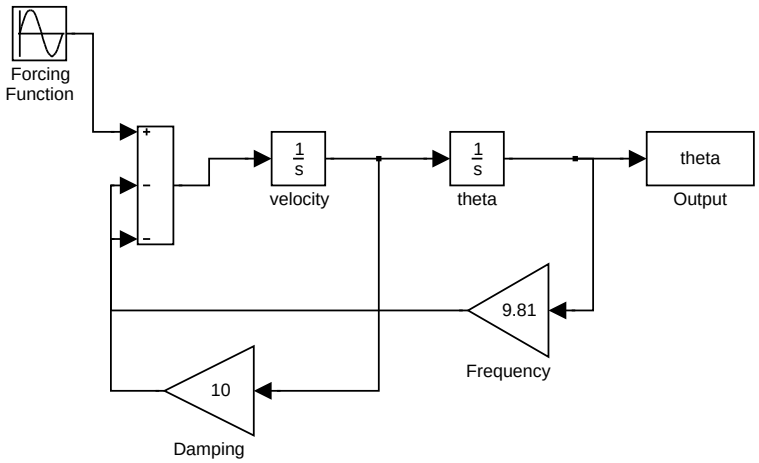


Figure 23: The SIMULINK model of a simple pendulum

1. On the MATLAB prompt (`>>`) type `simulink`.
2. Once the `Library: simulink` window is up click on `File` and then `New` and select `Model`.
3. Double click on `Sources`. This opens a list of sources.
4. Click on the `Sine wave` block and drag it to the new model window.
5. Double click on the `Sine wave` block. A new window pops up displaying the parameters controlling the block. Make sure that the `Amplitude` is 1 and the `Frequency` is set to 1 (according to the parameters of our problem). This is the forcing function on the pendulum.
6. Next double click on the `Continuous` icon in the library window. Drag an integrator to your model. Repeat this, as our solution has two integrations.
7. Double click on the `Math Operations` icon and drag two `Gain` and one `Sum` icon to the model.
8. Double click the `Sum` block and change the shape to rectangular. In the signs box enter `+|-|-`. This means there will be three inputs and one output. The second two inputs are to be subtracted from the first input.
9. Double click on the `Gain` blocks and put in the correct multiplicative factors.
10. Finally double click on the `Sinks` icon and drag a `To workspace` icon to the model.
11. All the required blocks are now in, and we can connect them into a system.

12. To connect blocks click on the source block, and click on the destination block while pressing the `ctrl` key. In order to branch bring the mouse to the path that will be branched. Then keeping the `ctrl` key pressed click and drag the mouse to the destination block.

13. Connect the blocks as shown in the Figure 23.

14. Double click on the `To workspace` block and name the variable (say, `theta`). Choose the save format as `Array`.

15. Now click on the `Simulation` tab on the top of the model window and select `Simulation parameters`. Select the `Solver` tab and set the `Stop time` to be 30. Select the `Workspace I/O` tab and uncheck everything but the `Time` button. Leave the variable name as `tout`. Uncheck the `Limit data point to last:` button.

Now you can run the system, but clicking `Simulation` and selecting `Start`. Now if you go back to the MATLAB prompt you will find that the new variables `tout` and `theta` are now in the workspace. Plotting these two gives us the evolution of the pendulum over time.

## 10.3 General Simulink Tips

The following are general tips and should be used often:

1. In order to save your work, select Save from the file menu and give the file that you want to save a name (or choose an old name if you are "writing over" an old version), and click the ok button (using the left-most mouse button). Realize that you have a choice of the "folder" that the file is saved in.

2. The results of a simulation can be sent to the MATLAB window by the use of the to workspace icon from the Sinks window. Open the to workspace icon and select the variable name that you want the results stored in the MATLAB workspace.

3. If your simulation has n state (or output) variables and you want to save them as different names, then you have to use a special connection called a `Demux` (as in demultiplexer) icon which is found in the `Signal Routing` window. Basically, it takes a vector input and converts it into several scalar lines. You can set the number of outputs (scalar lines) by double clicking on the icon and changing the number of outputs. A `Mux` icon takes several scalar inputs and multiplexes those in a vector (useful sometimes in transferring the results of a simulation to the MATLAB workspace, for example).

4. You can add steady state constants to variables using a `Constant` icon in the `Sources` window. To do this for a scalar output variable, just enter the value of the steady-state into the `Constant` icon and add this to the

scalar output using the `Sum` icon. For a vector output, you must first "break-up" the vector into scalar outputs using the `Demux` icon and then add the steady-state value to each scalar output.

5. Parameters can be "passed" to SIMULINK from the MATLAB window by using the parameter in a SIMULINK block or parameter box and defining the parameter in the MATLAB window. For example, say that one wants to run the simulation with many different damping constants $\gamma$, just rewrite the constant in the `Gain` block as `gamma`. Then make sure that you define the variable `gamma` in MATLAB to some constant. Now run the simulation. If you do not take the last step and `gamma` is undefined, the simulation will stop with an error on the `Gain` block using `gamma`.

6. In order to print the block diagram, first save the block diagram. Then click `File` and select `Print`.

## 10.4 More information

For more information on SIMULINK, please click `Help` in the top of the model window and select `Using Simulink`