MatLab Textbooks

- Mastering MATLAB, Duane Hanselmann and Bruce Littlefield, Pearson (Prentice Hall), 2012.


- The MatLab notes follow the texts closely in parts.

- The figure on the next slide show the covers of the 2 texts. Naturally, the 2012 edition is the best but it doesn’t cover GUIs, whereas the earlier edition does. However, we suggest the use of GUIDE for GUI design and implementation as programming GUI directly by typing text is too involved.
x=linspace(-3,3,13);
y=1:20;
z=-5:5;
[X,Y,Z]=meshgrid(x,y,z);
V=sqrt(X.^2+Y.^2+Z.^2);
slice(X,Y,Z,11);
hold on
h=contour(X,Y,Z,V);
set(h,'showlabels',true);
xlabel('X');
ylabel('Y');
zticklabels('	');
title('Figure 2.4 - Contour Plot with Selectable Labels');
hold off
MatLab Programming Environment

- MatLab has some basic windows as shown in Figure [1]

- The **MatLab desktop** is where MatLab puts you when you launch it. It has the following subwindows:
  
  1. **Command window**: This is the main MatLab window (has prompt `>>`) where application programs are launched from or commands are typed.
  
  2. **Current directory**: This is where your current files exist.
  
  3. **Workspace**: Where all the variables you have generated are shown (with their type and size) and where you can plot these variables
The MatLab environment consists of the MatLab desktop, a figure window and a editor window. The figure and editor windows must be invoked. Note that these windows are for MatLab2012b and later, earlier versions of MatLab have a slightly different feel and touch but are fine for this course.
(click on a variable) and then, using the right mouse, select the appropriate option.

4. **Command history**: All commands typed at a MatLab prompt are recorded and can be re-executed (across multiple sessions) by selecting a command with the mouse and executing it by double clicking on it. You can also create a M-file by selecting a set of commands from this window and or a right click of the mouse to select an option from the menu.

- The **Figure window** is where the graphics results are output. The user can create as many figure windows as memory will allow. When you use MatLab commands like `imshow` (to display images) or `plot` (to display graphs) these images will show in the figure window.
• The **Editor window** is where you write, edit, create and save your MatLab programs. You can use the built-in MatLab editor or use your favourite editor (for example, `vi program.m!` or `emacs program.m!` to invoke the `vi` or `emacs` editors). The built-in editor is simple to use: click on New (or New Script) and an editor window will pop up. If you used New select script to get an empty file. Type your MatLab code, using the delete key as necessary. You can point and click to go to specific locations in the file. Click on the Run button to execute your code. Look in the command window to see the results. Click on save to save your code to a specific filename.

• **Commands** `lookfor`, `help`, `helpwin` and `hekpdesk` provide on-line help. Command `demo` provides an introductory tutorial.
A Little Taste of MatLab

• MatLab example:

```matlab
>> % MatLab prompt
>> 2+2 % Command
ans = % MatLab response
    4
```

```matlab
>> % pi*r^2 is area of circle of radius r
>> area = pi*2.15^2
area =
    14.5220
```

• A Variable is a named location in memory holding some data. Here
area is the same of a variable and area = pi*2.15^2 is an assignment statement: compute the value of the expression pi*2.15^2 on the right hand side of the equals symbol and then assign it to the variable area on the left hand side of the equals sign.

- The expression pi*2.15^2 has 2 operators (\(^2\) is multiplication and \(^^\) is exponentiation). Exponentiation has higher precedence than multiplication so it is done first. So \(2.15^2\) is evaluated first (4.6225) and then that is multiplied by \(\pi\) (a built-in MatLab constant equal to \(\pi\), which is 3.141592653589793 as a double precision number). area is a double precision number (the default MatLab data-type). A double precision number requires 2 words or 8 bytes in the computer to represent it. A single precision number, on the other hand, only requires 1 word or 4
bytes of space to represent it in the computer.

- Then the value of \( \pi \times 2.15^2 \) is computed as 14.522012041218817 and saved at the memory location for the MatLab variable allocated for area.

- Unless you specify otherwise (no semi-colon ; at the end of the statement), Matlab prints the variable value as a floating point number with 4 digits to the right of the decimal point (this is format short).

**Input/Output in MatLab**

- The fundamental data-type in MatLab is the array. A single variable is actually an array of size 1\( \times \)1! More on this later! Elements of arrays can be integers, character strings, reals (both singles and doubles), complex
numbers, structures and cells...

- Dimensioning in MatLab is automatic. The dimensions of existing matrices can be found with `size` (for vectors and matrices) and `length` (for vectors only).

- MatLab is case sensitive, so `a` and `A` are different variables. The `casesen` command can turn this off or on (but why would you want to?).

- The output of a MatLab command is displayed on the screen unless MatLab is directed otherwise. A semicolon (`;`) at the end of a MatLab command suppresses screen output (except for graphics and online help). Paged output can be obtained using `more on`. Using the command provides buffered MatLab output. The format of the numbers output is set
by typing `format type`. Seven different formats are available for the value of $10\pi$:

- `format short` 31.4159
- `format short e` 3.1416e+001
- `format long` 31.41592653589793
- `format long e` 3.141592653589793e+001
- `format short g` 31.416
- `format long g` 31.4159265358979
- `format hex` 403f6a7a2955385e
- `format rat` 3550/113
- `format bank` 31.42
The `format compact` and `format loose` commands control spacing above and below the displayed lines while `format +` displays a +, – or a blank for a positive, negative or zero number respectively. The default format is `short`.

- MatLab saves previously typed commands in a buffer. These commands can be recalled with an uparrow key `↑`. You can also recall previously typed commands by typing a few letters of the command and then the `↑` key. You can also cut and paste commands in the usual way.

**MatLab File Types**

- A **file** in named collection of data stored in a contiguous location on some storage media (i.e. disk). Files can contain ASCII characters (your Mat-
Lab or Mex program, say, ass1_2013.m or ass1_2013.cass1_2013.f77),
an executable file (a compiled version of ass1_2013.m, say, ass1_2013.p
or a MatLab figure file ass1_2013.fig) or a binary data (data stored by
MatLab, say ass1_2013.mat) or an image file, say lena.jpg.

- For example, the following MatLab code:

\[
\begin{align*}
&>> \text{imshow('lena.jpg');} \\
&>> \text{title('My copy of lena.jpg')} \\
&>> \text{print('-djpeg','x.jpg')}
\end{align*}
\]

produces the window:
• This code segment assumes `lena.jpg` is in the current directory
  `/Volumes/barron/CS2035`. A second jpeg image, `x.jpg` is created (has the title) by the print statement in the current directory. You can view it (in unix) by `xv x.jpg`.

• The are 5 main file types in MatLab:

  1. **M-files** are ASCII files and have a `.m` extension. If you can read a **M-file** you can modify and copy it.

  2. **Mat-files** are binary files with the `.mat` extension. **Mat-files** are created by MatLab when you save/read data using the `save/load` command. This special format file can be read using the `load` command.
3. **Fig-files** are binary mat files files containing sufficient information to re-create figures and have a `.fig` extension. These files are created using **Save** and/or **Save As** commands from the **file** menu. **Fig-files** can be read using `open filename.fig`.

4. **P-files** are compiled **M-files** with extension `.p`. These files are created using the **pcode** command. These files cannot be read as ASCII files (so the source is hidden) but they can be executed.

5. **Mex-files** are MatLab callable Fortran or C programs and have a `.mex` extension after they have been compiled. Thus;

```bash
mex ass1_2013.c
```

would produce `ass1_2013.mexmaci64` on a 64 bit iMac workstation.
• Consider the MatLab code segment:

```matlab
I=imread('lena.jpg');
[counts,x]=imhist(I(:));
stem(x,counts);
title('Histogram of lena.jpg');
saveas(gcf,'my_hist','fig');
openfig('my_hist.fig');
```

• `imhist` expects either a 1D vector or 2D image, but color images are actually 3D: three 2D images of the red, green and blue color planes. `I(:)` returns the values of `I` as a 1D vector of pixel values. `imhist` has no problem handling this whereas `imhist(I)` results in an error.
• `gcf` is a MatLab function that return the **handle** (a pointer) to the current figure being displayed.

• `stem` displays a stem plot of the data as lines extending from a baseline along the x-axis. `x` contains the x coordinates and `counts(x)` contains the number of times each pixel in `x` occurred in the image.

• `saveas` is a matlab function that, in this case, save the histogram as `my_hist.fig`. It could be used to save the histogram in many other formats, such as jpeg (jpg), tiff (tif), bmp, ras, png, etc.

• Both `stem` and `openfig` cause the same identical histogram to be displayed (one for `stem` and a second for `openfig`).
Figure 2

Histogram of lena.jpg
Files and Directories

- A **file** is a collection of data or information that has a name (the filename) and is stored on some computer media (disk, for example). Typically, files are stored in **directories** or **folders** in a tree-like structure.

- `cd` allows you to change directories, `ls` list the files in the current directory, `pwd` prints the current working directory (where am I), `dir` is like `ls`, `mkdir` makes a new directory, `rmdir` removes an (empty) directory.

- `what` lists the M-files in the current directory and `which x.m` specifies the full pathname where `x.m` resides.

- `path` is a MatLab command that lists all pathnames MatLab can access
on your machine.

- You can add new directories to your path. For example, to add a path

  /Volumes/barron/MATLAB

  use addpath /Volumes/barron/MATLAB. Use savepath to save
  this modification of the pathnames permanently.

Some Thing to Remember

- MatLab is platform independent, for the most part. MatLab executes the
  same way on different machines. Only commands that depend on the
  local operating system, such as an editor like vi or emacs, are different.
• **Not being in the right directory**: only files in the current directory can be accessed. Change to the appropriate directory to execute a particular MatLab file using the `cd`.

• **Not saving files in the correct directory**: by default you save files in the current directory. If you want to save a file in another directory, you need to change directories first (or include the pathname in the filename).

• **Not overwriting an existing file while editing**: If you run an M-file, do not like the results, edit the file and run it again, you may get the same results as before, as previously compiled code may execute instead. The simple cure is to clear the workspace of this function with `clear`. 
MatLab Examples

>> 2+2
ans =
     4

>> x=2+2
x =
     4

>> y=2^2+log(pi)*sin(x);

>> y; % nothing happens as ; suppresses the output
>> y
y =
3.1337

>> theta = acos(-1)
theta =
    3.1416

>> format short e
>> theta
theta =
    3.1416e+000

>> format long
>> theta
theta =
    3.14159265358979
>> 2^5/(2^5-1)
ans =
    1.0323

>> 3*(sqrt(5)-1)/(sqrt(5)+1)^2-1
ans =
    -0.6459

>> area = pi*(pi^(1/3)-1)^2
area =
    0.6781

>> exp(3)
ans =
    20.0855
>> log(exp(3))
ans =
    3

>> log10(exp(3))
ans =
    1.3029

>> log10(10^5)
ans =
    5

>> exp(pi*sqrt(163))
ans =
    2.6254e+17
>> x=log(17)/log(3)

>> x

2.5789

>> sin(pi/6)

ans =

0.5000

>> cos(pi)

ans =

-1

>> tan(pi/2) % this number is undefined?

ans =

1.6331e+16 % what a version of MatLab gives
>> (sin(pi/6))^2+(cos(pi/6))^2
ans =
    1
% cosh and sinh are hyperbolic functions

>> x=32*pi; y=(cosh(x))^2=(sinh(x))^2
>> y
y =
    0

Complex Numbers

• MatLab recognizes the letters i and j as the imaginary number $\sqrt{-1}$. A complex number has a real part and an imaginary part. For example,
2+5i has real part 2 and imaginary part 5.

- Engineers: the Fourier Transform of a signal or image gives an array of 1D or 2D complex numbers - we’ll see this later.

- Arithmetic operations are done by MatLab as follows:
  - Addition: \((a + ib) + (c + id) = (a + c) + i(b + d)\).
  - Subtraction: \((a + ib) - (c + id) = (a - c) + i(b - d)\).
  - Multiplication: \((a + ib) \times (c + id) = (ac - bd) + i(bc + ad)\).
  - Division: 
    \[
    \frac{(a+ib)}{(c+id)} = \frac{(a+ib)(c-id)}{(c+id)(c-id)} = \frac{(ac+bd)+i(bc-ad)}{c^2+d^2}.
    \]
• Also the definitions may be handy:

\[ e^{ix} = \cos x + i \sin x \]

\[ e^{-ix} = \cos x - i \sin x. \]

• The exponential function is the function \( e^x \), where \( e \) is the number (approximately 2.718281828459046) when \( x = 1 \).

• Figure A show the mathematical relation of \( e^{i\phi} \) in terms of \( \sin \) and \( \cos \) and Figure B shows a plot of this function. Note that \( e^0 \) is 1 and \( e^1 \) is 2.718281828459046 (Figures from [http://en.wikipedia.org/wiki/Exponential_function](http://en.wikipedia.org/wiki/Exponential_function)).
MatLab Complex Numbers examples:

```matlab
>> c = 2+3i
```

```matlab
c = 
    2.0000 + 3.0000i
```

```matlab
>> real(c)
```
ans =

    2

>> imag(c)
ans =

    3

>> abs(c) % magnitude of c
ans =

    3.6056

>> % MatLab computes abs as:
>> sqrt(2^2+3^2)
ans =

    3.6056
>> conj(c) % complex conjugate of a+bi is a-bi
ans =
    2.0000 - 3.0000i
>> c*conj(c) % (2+3i)*(2-3i)=4+6i-6i-9i^2=4+9=13
ans =
    13

• \(\frac{(1+3i)}{(1-3i)}\) is computed by multiplying the numerator and denominator by
(1 + 3i), the complex conjugate of (1 − 3i). The denominator now becomes real and we obtain the complex number
\[
\frac{(1+3i)(1+3i)}{(1-3i)(1+3i)} = \frac{1+6i+9i^2}{10} = \frac{-8+6i}{10} = -0.8 + 0.6i.
\]

>> (1+3i)/(1-3i)
ans =
-0.8000+0.6000i

>> exp(i*pi/4)
ans =
0.7071+0.7071i

>> format long

>> exp(1) % e^1
ans =
2.718281828459046

>> exp(pi/2) % 2.718281828459046^(pi/2)
ans =
4.810477380965351

>> exp(-pi/2) % 2.718281828459046^(-pi/2)
ans =

    0.207879576350762

>> % Complex number: use e^(ix)

>> % Here pi/2 is multiplied by i

>> exp(pi/2*i)

ans =

    0.0000+1.0000i

>> % Complex number: use e^(-ix)

>> % Here pi is divided by 2i ==> -2*pi*i/4

>> exp(pi/2i)

ans =

    0.0000-1.0000i
Arrays

- An **array** is a named collection of data (usually stored in contiguous memory locations) with individual data elements being accessed via an **index** or **indices** (in the case two or more dimensions).

- 1D arrays are also called **vectors** and 2D arrays are sometimes referred to as **tables** or **matrices** (but a matrix can also be of higher dimension).

- A variable can be thought of as a 1D array with one element (and so the index need not be specified).

- 1D array example: \( x = [2 \ 4 \ 6] \) has 3 elements. \( x(1) \) is 2, \( x(2) \) is 4 and \( x(3) \) is 6.
• 2D array example: \( y=[1 \ 2 \ 3; \ 4 \ 5 \ 6] \) has 2 rows and 3 columns of data elements. \( y(1,1) \) is 1, \( y(1,2) \) is 2, \( y(1,3) \) is 3, \( y(2,1) \) is 4, \( y(2,2) \) is 5 and \( y(2,3) \) is 6.

• MatLab has both statically and dynamically allocated arrays. Static arrays are predefined before used (has computational efficiency benefits) while dynamic arrays are defined when using them.

\[
\begin{align*}
\text{>> x} & = [1 \ 2 \ 3] \\
x & = \\
& \begin{bmatrix}
1 & 2 & 3
\end{bmatrix} \\
\text{>> y} & = [2; \ 1; \ 5] \\
y & = 1 \\
& \begin{bmatrix}
2
\end{bmatrix}
\end{align*}
\]
3

>> z = [2 1 0];

>> a = x + z

a =

   3  3  3

>> b = x + y

??? Error using ==> plus

Matrix dimensions must agree.

% You can multiply (or divide) the elements of two
% same-sized vectors term by term using array operators
% .* or ./. Note that x.*x and x.^2 both square each
% element of x.
% This is vectorization: [1 2 3] .* [2 1 0] = [2 2 0]
% Vectorization is not matrix multiplication

>> a = x .* z

a =

    2  2  0

>> x = [1 2 3 4 5];

>> x = x .^ 2

x =

    1  4  9 16 25

>> b = 2 * a % 2 * [2 2 0]

b =
4 4 0

% Create a vector \( x \) with 5 elements linearly spaced
% between 0 and 10

\[ x = \text{linspace}(0,10,5) \]

\[ x = \\
0 \hspace{1em} 2.5000 \hspace{1em} 5.0000 \hspace{1em} 7.5000 \hspace{1em} 10.0000 \]

\[ y = \sin(x) \]

\[ y = \\
0 \hspace{1em} 0.5985 \hspace{1em} -0.9589 \hspace{1em} 0.9380 \hspace{1em} -0.5440 \]

\[ \sqrt{x} \]

\[ \text{ans} = \\
0 \hspace{1em} 1.5811 \hspace{1em} 2.2361 \hspace{1em} 2.7386 \hspace{1em} 3.1623 \]
>> z=sqrt(x).*y
z =
    0  0.9463  -2.1442  2.5688  -1.7203

% We can compute x^n, where x is a vector and % n is an integer.
>> x=[1 2 3 4 5];
>> n=2;
>> x.^n
ans =
    1    4    9   16   25
We can compute \( r^n \), where scalar \( r \) is raised to the power of each element of \( n \).

Consider the series \( r^0 + r^1 + r^2 + r^3 + \ldots + r^n \).

Note that \( r^0 \) is 1 and \( r^1 \) is \( r \). To evaluate this sum for scalar \( r=0.5 \) we create a vector

\[
\begin{bmatrix}
1 & r & r^2 & r^3 & \ldots & r^n
\end{bmatrix}
\]

and then sum this vector.

In the limit, as \( n \to \infty \) this sum approaches \( \frac{1}{1-r} \), \( r < 1 \).

For \( r=0.5 \) the limit is 2.0

```
>> n=0:10

n =
   0   1   2   3   4   5   6   7   8   9  10
```
>> r=0.5;

>> x=r.^n;

>> s1=sum(x);

>> s1 % not 2

s1 =

1.9990

>> n=0:50;

>> x=r.^n;

>> s2=sum(x);

>> s2 % approximately 2

s2 =

2.0000
\[
\text{>> n=0:100;}
\]
\[
\text{>> x=r.^n;}
\]
\[
\text{>> s3=sum(x);}
\]
\[
\text{>> s3 \% much closer}
\]
\[
\text{s3 } =
\]
\[
\begin{array}{c}
2
\end{array}
\]

\[
\text{\% We can compute r^x and x^r, where both}
\]
\[
\text{\% r and x are vectors.}
\]
\[
\text{\%}
\]
\[
\text{>> r=[2 3 4]}
\]
\[
\text{r } =
\]
\[
\begin{array}{ccc}
2 & 3 & 4
\end{array}
\]


Lecture 1: CS9630a - MatLab

>> x=[1 2 3]

x =

    1     2     3

>> r.^x

ans =

    2     9    64

>> x.^r

ans =

    1     8    81

• Arrays can be used statically or dynamically.

>> % set up a 1D array (row vector)
>> % and set its 6 elements
>> a(1)=1.0;
>> a(2)=2.0;
>> a(3)=3.0;
>> a(4)=4.0;
>> a(5)=5.0;
>> a(6)=6.0;

>> a % print the elements of 1
a =
    1    2    3    4    5    6

>> size(a) % print the size of a: 1 row with 6 elements
ans =
>> % set up a 1D array by setting the last element
>> % all the elements before b(6) are set to zero
>> b(6)=6.0;

>> b

b =
     0     0     0     0     0     6

size(b)

ans =
     1     6

>> % extend b to have 16 elements by setting b(16)
>> % to 16: now b(6) and b(16) are set and all
>> % b elements have value 0
>> b(16)=16.0;

>> b

b =

    0 0 0 0 0 6 0 0 0 0 0 0 0 0 0 16

>> size(b)

ans =

    1    16
MatLab Control Statements

- MatLab has the usual control statements, such as for, while, if-then-else, switch/case, break etc. Loops are essential to the use of arrays.

For Loop Examples

- A loop allows a group of statements to be repeated a fixed, predetermined number of times (we’ll ignore the prompt symbol >> from now on):

```matlab
for x=lower_bound:step:upper_bound
    group of statements
end % for x
```

- `lower_bound:step:upper_bound` can be replaced by
lower_bound:upper_bound if step is 1.

- Note that lower_bound:step:upper_bound actually defines an array. For example, 2:2:10 is an array with values 2, 4, 6, 8, and 10. A loop:

```
for q=2:2:10
    statements
end % for q
```

would execute statements 5 times, with q being 2, 4, 6, 8 and 10.

- Loop indices can be 0 in Matlab but array indices cannot be 0 (or negative). Some more examples...
• Forward loop:

    sum=0;
    for m=1:100
        sum=sum+1/m;
    end
    >> fprintf('sum=%8.6f
',sum);
    sum=5.187378

• Reverse loop:

    sum=0;
    for n=100:-2:0
        sum=sum+1/exp(n);

end

fprintf('sum=%8.6f
',sum);

sum=1.156518

• fprintf is a print statement that prints \texttt{sum} as a floating point number using the format \texttt{\%8.6f}. This format means 6 digits are printed after the decimal point and 8 alpha numeric characters can be printed in total. Since there is a decimal point (an alpha numeric character) only 1 digits can be printed to left of the decimal point.

• Use \texttt{randperm} to get a list of numbers 1 to 10 in random order. Then a loop can work with these random number as the indices:

\[ \texttt{nums=randperm(10)} \]
nums =

6 3 7 8 5 1 2 4 9 10

for n=nums % the loop is executed 10 times,
    % with the values in nums
    x(n)=sin(n*pi/10);
end

format short
x =

0.3090 0.5878 0.8090 0.9511 1.0000
0.9511 0.8090 0.5878 0.3090 0.0000

format long
x =
Columns 1 through 5
0.309016994374947 0.587785252292473 0.809016994374947
0.951056516295154 1.000000000000000

Columns 6 through 10
0.951056516295154 0.809016994374947 0.587785252292473
0.309016994374948 0.000000000000000

- **for** loops can be nested but an **end** is required for each **for**.

```matlab
for i=1:3
    for j=4:7
        fprintf('%i=%3d j=%3d i+j=%4d\n',i,j,i+j);
    end % for j
end % for i
```
end % for i

i= 1 j= 4 i+j= 5
i= 1 j= 5 i+j= 6
i= 1 j= 6 i+j= 7
i= 1 j= 7 i+j= 8
i= 2 j= 4 i+j= 6
i= 2 j= 5 i+j= 7
i= 2 j= 6 i+j= 8
i= 2 j= 7 i+j= 9
i= 3 j= 4 i+j= 7
i= 3 j= 5 i+j= 8
i= 3 j= 6 i+j= 9
Loops are good for indexing arrays.

```matlab
i = 3; j = 7; i+j = 10

a(1) = 1;
a(2) = 2;
a(3) = 3;
a(4) = 4;
a(5) = 5;
a(6) = 6;
n = 6;
sum = 0;
for i = 1:n
```
sum=sum+a(i);
end % for i
fprintf('sum=%d n(n+1)/2=%d\n',sum,n*(n+1)/2);

prints:

sum=21 n(n+1)/2=21

That is, \( \sum_{i=1}^{n} \) is equal to \( \frac{n(n+1)}{2} \).

• Another nested loop:

```matlab
for n=1:5
for m=1:5
    A(n,m)=n^2+m^2;
end % for m
end % for n
```
end % m

disp(n) % display or print unformatted n
end % n

1

2

3

4

5

A

A =

2  5  10  17  26
5  8  13  20  29
While Loop Example

- A while loop can evaluate a group of statements an infinite number of times. The general form of a while statement is

  ```
  while expression
      group of statements
  end % while
  ```

- One example:
% Print all numbers that are powers of 2 below 10000
num = 1; i = 1;
while num < 10000
    fprintf('i=%5d num=%10d\n',i,num);
    i = i+1;
    num = 2^i;
end
i= 1 num= 1
i= 2 num= 4
i= 3 num= 8
i= 4 num= 16
i= 5 num= 32
\begin{itemize}
\item A second example: compute \textit{eps}, the smallest number that can be added to 1 such that the result is greater than 1, using the finite precision available on a computer.
\end{itemize}
% EPS is capitalized as eps is a
% built in MatLab constant

num=0; EPS=1;

while(1+EPS > 1)
    EPS=EPS/2;
    num=num+1;
end % while

% the loop expression becomes false when EPS
% becomes too small. Multiplying it by 2 once
% is the small EPS value such that EPS+1 ~= 1
EPS=EPS*2;
EPS
EPS =

2.220446049250313\times10^{-16}

eps

ans =

2.220446049250313\times10^{-16}

num

num =

53

Double precision is approximately 16 digits so we should expect eps to be near \(10^{-16}\).
If-Elseif-Else Statement Example

- The execution of 1 or more commands can be conditionally controlled on the basis of a true/false (boolean) expression. The simplest if-else-end construction is:

```matlab
if expression
    group of statements
end % if
```

- When there are 2 alternatives the if-else-end construction becomes:

```matlab
if expression
    statements1
```
else
    statements2
end % if

- When there are 3 or more alternatives the if-else-end construction becomes:

    if expression1
        statements1
    elseif expression2
        statements2
    elseif expression_nml

    ...

i=6; j=21;
if i > 5
    k=i;
    fprintf('yes\n');
elseif (i>1) & (j==20)
    k=5*i+j;
    fprintf('no\n');
else
    k=1;
fprintf('maybe');
end

prints:

yes

- EPS example with break:

EPS=1;

% lets assume at most 1000 divisions
% are needed to compute EPS
for num=1:1000
    EPS=EPS/2;

if (1+EPS) <= 1
    EPS = EPS * 2;
    break;
end % if
end % for
EPS
EPS =

2.220446049250313e-16

- EPS example with continue:

EPS = 1;

% lets assume at most 1000 divisions needed to compute
for num=1:1000
    EPS=EPS/2;
    if(1+EPS) > 1
        continue; % check the for loop condition
    end % if
    EPS=EPS*2;
    break;
end % for

EPS

EPS = 

2.220446049250313e-16

Note that the continue statement has no effect on the if-end con-
struction.

Switch-Case Statement Example

- A sequence of statements can conditionally be evaluated on the basis of repeated use of an equality test using a switch-case construction:

```matlab
switch expression
    case test_expression1
        statements1
    case test_expression2
        statements2
    ...
    otherwise statements_n
```
• expression must either be a boolean (true or false) or a character string (in which case, the case test expressions must also be character strings and equality is tested for). An example:

colour = input('colour=','s');
switch colour
    case 'red'
        c=[1 0 0];
    case 'green'
        c=[0 1 0];
    case 'blue'
        c=[0 0 1]
    otherwise
error('Invalid choice of colour')

end
Creating and Executing Script Files

- We can save a collection of MatLab commands in a file, called a *script file* (the term “script” signifies that MatLab simply read from the script found in the file) or *M-files* (the term “M-file” means the script filenames must end with the extension “.m”).

- We can write a script file, `circle.m`, with contents:

```matlab
% CIRCLE - a script file to draw a unit circle
% -----------------------------------------------

% generate 100 linearly spaced elements
theta=linspace(0,2*pi,100);
```
% x is a 100 element vector of \cos(\theta) terms
x = cos(theta);

% y is a 100 element vector of \sin(\theta) terms
y = sin(theta);

% plot x versus y elements (a circle)
plot(x, y);

% set the length scales of the 2 axes the same
axis('equal');

% label the x axis with x
xlabel('x');

% label the y axis with y
ylabel('y');
% put title on graph
title('Circle of unit radius');
print('djpeg',circle.jpg);

• In MatLab, type help circle and the comment at the beginning of the circle.m file will print:

% CIRCLE - a script file to draw a unit circle
% _______________________________________________________

• At the MatLab prompt >> type circle. Now an image of this graph appears in the figure window.
Circle of unit radius
Creating and Executing a Function File

• We can modify `circle.m` to be a function `circle_fct.m` as:

```matlab
function [x,y] = circle_fct(r,filename);

% CIRCLE Draw a circle of radius r
% Syntax: [x,y]=circle_fct(r,filename);
% or circle_fct(r,filename);
% Input: r, the radius and
% filename, the file to save the jpg image to
% Output [x,y] - the x and y coordinates of the circle points
%theta=linspace(0,2*pi,100);  % create vector theta
```
x=r*cos(theta);             % generate x coordinates
y=r*sin(theta);            % generate y coordinates
plot(x,y);                 % plot the circle
axis('equal');             % set equal scale on axes

title(['Circle of radius r=',num2str(r)]); % put title on graph
print('-djpeg',filename);  % save figure

- We can execute circlefn in 4 ways:

R=5;
filename='my_circle_fct.jpg';
[x,y]=circlefn(R, filename);
% input radius specified directly
[cx, cy] = circlefn(2.5, filename);

% you can ignore the output
circlefn(1, filename);

% the argument can be an expression
circlefn(R^2/(R+5*sin(R), filename);

• The graph for circle_fct(2.5,'my_circle_fct.jpg'); is as shown. Note the title and the $x$ and $y$ coordinates are different.
- Note that `num2str(r)` converts a numerical value in `r` to a character string. The square brackets, `[` and `]` concatenate the two strings between
Recursive Functions in MatLab

- A recursive function is a function defined in terms of itself. The recursive formula has to have a condition that stops the recursion.

- Why recursion: sometimes it is natural to write a function recursively: for example, if you are given the recursive definition of a function it is natural (and easy) to code it that way.

- Consider the recursive function, factorial(n) \([n!]\). Note the recursive calculation \(n! = n \ (n - 1)!\) and the stopping criteria \(1! = 0! = 0\).

```matlab
function factn=factorial(n);
```
% FACTORIAL: function to compute factorial
% Call Syntax: factn=factorial(n)
if n < 0 % If n is negative factorial is undefined
    error('n is negative - factorial undefined')
elseif n==0
    factn=1; % 0! is defined to be 1
else % n >= 1
    fprintf('%d ',n);
    factn=n*factorial(n-1);
end % if
end % factorial

• Note that when \( n \) is negative factorial is undefined and an error message
is printed. Also note that when n is 0 or 1 no recursive call to factorial is made.

- Some runs:

```matlab
factorial(-2)
ans =
  n is negative - factorial undefined

factorial(0)
ans =
    1

factorial(6)
ans =
   720
```
Compiling Your MatLab Code

- Suppose you have a file named `circle_fct.m`, which is interpreted when executed). You may want to save a compiled version of this file, made using `pcode circle_fct` which generates compiled code for this function in a file named `circle_fct.p`, The next time `circle_fct` is called `circle_fct.p` is used.

- Unless your function is large, it probably make no difference (speed-wise) whether you use a `m` or a `p` file. The best use of `p` files is to protect your propriety rights to your code (other people can’t see it but can use it), provided they are running your code on the same architecture it was compiled on.
Global Variables

- Syntax: `global` X defines X as global in scope.

- Typically, each MatLab function has its own local variables, which are different from those of other functions (even if they have the same names) and from the variables in your workspace.

- However, if several functions, and possibly the base workspace, all declare a particular name as global, they all share a single copy of that variable. Any assignment to that variable, in any function, is available to all the functions declaring it global.

- If the global variable does not exist, then the first time you use a global statement, that variable is initialized to the empty matrix.
• If a variable with the same name as the global variable already exists in the current workspace, MatLab prints a warning and changes the value of that variable to match the global variable’s value.

• All parameters of a MatLab function are “pass by value”, including arrays. That means large arrays have to be copied by the function and is any changes are made the array must be passed back to the calling routine, for example, \( A = \text{fct}(A) \) where \( A \) is an array. global variables allow this to be avoided (saves space and time) by having global \( A \) in function \( \text{fct} \).
Relational Operators

- We use relational operators to compare things (numbers, characters) in MatLab. There are 6 relational operators in MatLab:

  <  less than
  <= less than or equal
  >  greater than
  >= greater than or equal
  == equal
  ~= not equal

- These operators return true or false (1 or 0) depending on whether some expression is true or false.

- These operators result in a matrix or vector of the same size as the operands, where if the relationship is true for an element the correspond-
ing resultant element is 1 and 0 otherwise. Remember: scalars (single numbers) are matrices of size $1 \times 1$ or vectors of length 1. Everything in MatLab is a matrix!!!

- If $x=[1 \ 5 \ 3 \ 7]$ and $y=[0 \ 2 \ 8 \ 7]$ then:

  \[
  \begin{align*}
  k &= x < \ y \quad \% \text{ results in } k=[0 \ 0 \ 1 \ 0] \\
  k &= x \leq \ y \quad \% \text{ results in } k=[0 \ 0 \ 1 \ 1] \\
  k &= x > \ y \quad \% \text{ results in } k=[1 \ 1 \ 0 \ 0] \\
  k &= x \geq \ y \quad \% \text{ results in } k=[1 \ 1 \ 0 \ 1] \\
  k &= x == \ y \quad \% \text{ results in } k=[0 \ 0 \ 0 \ 1] \\
  k &= x \neq \ y \quad \% \text{ results if } k=[1 \ 1 \ 1 \ 0]
  \end{align*}
  \]

- Typically, these are used in conditional statements but can also be used to
do computing. For example, \( u = v \{ v \geq \sin(\pi/3) \} \) finds all elements of \( v \) such that \( v_i \geq \sin(\pi/3) \) and stores them in vector \( u \).

Logical Operations

- Two or more relational operators can be combined using logical operators. There are 4 logical operators:

  \begin{align*}
  & \& \text{logical AND} \\
  | & \text{logical OR} \\
  \sim & \text{logical complement (NOT)} \\
  \text{xor} & \text{exclusive OR}
  \end{align*}

- For \( x = [0 \ 5 \ 3 \ 7] \) and \( y = [0 \ 2 \ 8 \ 7] \) then:

  \[
  m = (x > y) \& (x > 4) \quad \% \text{ results in } m = [0 \ 1 \ 0 \ 0]
  \]
m = x|y  % results in m=[0 1 1 1]
m = ~(x|y)  % results in m=[1 0 0 0]
m = xor(x,y)  % results in m=[0 0 0 0]

• \( x((x>y) \& (x>4)) \) gives the elements of \( x \) that are greater than the corresponding element of \( y \) and greater than 4 at the same time.

• There are also some built-in logical operators:

\[
\text{all(condition)}
\]
% true is any element of a matrix satisfies the condition, i.e. any(x) returns true is any
% element of x is non-zero
any(condition)

% true if x is an existing function or variable
exist(x)

% true if x is []
isempty(x)

% true for all non-defined elements of x
isnan(x)

% returns all elements of x that are non-zero
find(x)

Character Strings

- Character strings are row vectors with one element per character. For example,

  message='Leave me alone'

  is a $1 \times 14$ vector.

- names1=['John';'Ravi';'Mary';'Xiao'] creates a column
vector with one name per row (thus names1 is a $4 \times 4$ matrix). The rows must all be the same size, so

\[
\text{names2}=['\text{Hi}';'\text{Hello}';'\text{Bonjour}']
\]

causes an error. On the other hand,

\[
\text{names2}=['\text{Hi}';'\text{Hello}';'\text{Bonjour}']
\]

works.

- We can use \texttt{char} to do pad these blank characters.

\[
\text{names2}=\text{char}('\text{Hi}','\text{Hello}','\text{Bonjour}')
\]
• To have a quote character inside a string you quote the quote. For example, `title('John''s')` to print John’s.

• Character strings can be manipulated just like matrices.

\[ c = [\text{names2}(2,:) \ \text{names1}(3,:)] \]

has Hello Mary as the output for c.

• Some built-in character functions: char, abs (convert characters to their ascii numeric values, eval (execute the string as a command), int2str, ischar, isletter, lower (convert upper case characters in string to lower case characters), upper, strcmp (string compare), strcat (string concatenate), ...
• \texttt{eval('x=5*\sin(\pi/3)')} computes $5\sin(\pi/3)$ and is equivalent to typing $x=5\sin(\pi/3)$ in the command window.

• String concatenation can be done using the left and right square brackets:
  \[
  \texttt{a=['abc' '123' 'cd' 'q45'];}
  \]
  gives the value 'abc123cdq45' to variable \texttt{a}. One good use for string concatenation is to generate file names for I/O.

\section*{Some examples}

\begin{verbatim}
if 'john' < 'mary' & 'hello' > 'bonjour'
  disp('Yabba dabba doo');
else
  disp('Sad to say');
\end{verbatim}
end

- prints Sad to say. 'john' is less than 'mary' (for the first characters, j jm). but 'hello' is greater than 'bonjour'. To compare 'hello' with 'bonjour' pad 'hello' with blank characters on the right until the result has the same number of characters as 'bonjour', i.e. 'hello '. Then compare the corresponding characters left to right until inequality is found. This comparison use the ascii values for the characters. The ascii character for '0' is 48, for 'A' is 75, for 'a' is 97, for blank is 32 (other visible special characters appear to be between 33 and 47 or between 123 and 126).

- Find all elements of $x$ greater than 3:
x = [1 2 3 4 5 6 7];
x(x>3) = 1
x = [1 2 3 1 1 1 1];

- Find the coordinates of all elements greater than 100:

x = [12 34 56 78 910 1112];
find(x>100)
ans =
   5   6

- The use of string and concatenation operators:

a = strcat('abc','123')
a =
abc123

a = strcmp('abc', 'abdefg') % a=0 is for false
a =
    0

a = strcmp('abc', 'abc') % a=1 is for true
a =
    1
Matrices and Matrix Operations

- The MatLab command $A=[1 \ 2 \ 5; \ 3 \ 9 \ 0]$ makes matrix

\[
A = \begin{bmatrix}
1 & 2 & 5 \\
3 & 9 & 0
\end{bmatrix}.
\]

- $A$ is a 2D array, with the first index specifying the row and the second index specifying the column. Thus $A(2,3)$ is the element in the $2^{nd}$ row, $3^{rd}$ column, namely 0.

- We can perform operation directly on such matrices. $\exp(A)$ computes the exponential of each element of $A$: 
exp(A)

ans =

1.0e+03 *

0.0027  0.0074  0.1484

0.0201  8.1031  0.0010

- **MatLab command** \( B = [2\times \log(x) + \sin(y); \; 5i \; 3+2i] \) makes matrix

\[
B = \begin{bmatrix}
2x & \ln x + \sin y \\
5i & 3 + 2i
\end{bmatrix}.
\]

The current \( x \) and \( y \) values are used to evaluate these expressions.
For \( x=1 \) and \( y=1 \) we obtain:

\[
B = \\
\begin{bmatrix}
2.0000 + 0.0000i & 0.8415 + 0.0000i \\
0.0000 + 5.0000i & 3.0000 + 2.0000i
\end{bmatrix}
\]

Note that the first row elements are output as complex but with 0i as the imaginary parts of these numbers.

- Vectors are 1D matrices. \( u=[1 \ 3 \ 9] \) is a row vector while \( v=[1;3;9] \) is a column vector.

\[
 u = [1 \ 2 \ 3] \\
 u =
\]
1 2 3

v =
1
2
3

- A scalar is a vector with 1 row and 1 column. In this case we can leave the brackets off. \( a = [6] \) or just \( a \) has the value 6.

- A matrix with 0 rows and 0 columns is the null/empty matrix and is indicated by empty brackets, i.e. \( X = [ ] \).

- If it is not possible to type an entire row on one line than the continuation periods ... can be used to indicate the input continues on the next line.
Thus the following 3 commands are equivalent:

\[
A = [1 \ 3 \ 9; 5 \ 10 \ 15; 0 \ 0 \ -5];
\]
\[
A = [1 \ 3 \ 9 \\
     5 \ 10 \ 15 \\
     0 \ 0 \ -5]
\]
\[
A = [1 \ 3 \ 9; 5 \ 10 \ ... \\
     15; 0 \ 0 \ -5];
\]

- Continuation periods (...) can be used anywhere white spaces can be used. For example:

\[
a = 4 + 5 + ... \\
    6 + 7
\]
• Remember, elements of a matrix can be accessed by specifying their row and column indices. Then $A(i, j)$ refers to the element at the $i^{th}$ row and $j^{th}$ column of matrix $A$.

• A range of rows and columns can be specified. For example, $A(m:n, k:l)$ specified rows $m$ to $n$ and columns $k$ to $l$. When the rows (or columns) to be specified range over all rows (or columns) we can use a : to specify this. Thus $A(:, 5:20)$ refers to columns 5 to 20 of all rows of $A$.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4; & 5 & 6 & 7 & 8; & 9 & 10 & 11 & 12; & 13 & 14 & 15 & 16 \end{bmatrix}$$
Matrix dimensions are automatically determined by MatLab. We can use `size(A)` to find the number of rows and columns of matrix A. If A is 2D then `[m, n] = size(A)` assigns the number of rows to m and the number of columns to n. For the above matrix:
size(A)
ans =

    4    4

- When a matrix is first specified, MatLab creates a matrix just big enough to accommodate it. If matrices $B$ and $C$ do not already exist then $B(2,3) = 5$; produces:

$$B = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 5 \end{bmatrix}$$

while $C = (3, 1:3) = [1 \ 2 \ 3]$ produces

$$C = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 2 & 3 \end{bmatrix}.$$  

In both cases, undefined elements of $B$ and $C$ that must exist are assigned value 0.
Matrix Manipulation

- Consider the following examples:

```matlab
% Matrices are entered row-wise. Rows are
% separated by semicolons and columns
% are separated by spaces or commas
A = [1 2 3; 4 5 6; 7 8 8]
A =
  1  2  3
  4  5  6
  7  8  8
A(2,3) % Element A(2,3) of matrix A is accessed.
```
ans =

6

% Changing any entry is easy through indexing
A(3,3) = 9

A

1 2 3
4 5 6
7 8 9

% Any submatrix of A is obtained by using
% range specifies for row and column indices
B=A(2:3,1:2)

B =
4  5
7  8

% The column by itself as a row or a column index
% specifies all rows or columns of the matrix
B=A(2:3,:)

B =

4  5  6
7  8  9

% A row or a column of a matrix is deleted by
% setting it to []
B(:,2)=[]

B =
• If $A$ is a $10 \times 10$ matrix, $B$ is a $5 \times 10$ matrix and $y$ is a 20 element row vector then

$$A[1 \ 3 \ 6 \ 9],:) = [B(1:3,:); y(1:10)]$$

replaces the $1^{st}$, $3^{rd}$ and $6^{th}$ rows of $A$ by the first 3 rows of $B$ and the $9^{th}$ row of $A$ by the first 10 elements of $y$. This statement requires that matrix sizes be compatible.
• If $Q = \begin{bmatrix} 2 & 3 & 6 & 0 & 6 \\ 0 & 0 & 20 & -4 & 3 \\ 1 & 2 & 3 & 9 & 8 \\ 2 & -5 & 5 & -5 & 6 \\ 5 & 10 & 15 & 20 & 25 \end{bmatrix}$

and $v = [1 \ 4 \ 5]$ then $Q(v; :) = \begin{bmatrix} 2 & 3 & 6 & 0 & 5 \\ 2 & -5 & 5 & -5 & 6 \\ 5 & 10 & 15 & 20 & 25 \end{bmatrix}$

and $Q(:, v) = \begin{bmatrix} 2 & 0 & 5 \\ 0 & -4 & 3 \\ 1 & 9 & 8 \\ 2 & -5 & 6 \\ 5 & 20 & 25 \end{bmatrix}$. 
• In MatLab 5 and later versions, to get the 1\(^{st}\), 4\(^{th}\) and 5\(^{th}\) rows of \(Q\) you can do:

\[ v = [1\ 0\ 0\ 1\ 1]; \quad v = \text{logical}(v); \quad Q(v,:). \]

**Reshaping Matrices**

• Matrices can be reshaped as a vector: \(b = A(:)\) strings out the elements of \(A\) row-wise as a column vector \(b\).

• If \(A\) is a \(m \times n\) matrix it can be resized into a \(p \times q\) matrix, as long as \(m \times n = p \times q\), with the command \(\text{reshape}(A,p,q)\). Thus, for a \(6 \times 6\) matrix \(A\), \(\text{reshape}(A,9,4)\) transforms \(A\) into a \(9 \times 4\) matrix and \(\text{reshape}(A,3,12)\) transforms \(A\) into a \(3 \times 12\) matrix.
Transpose of a Matrix

- The transpose of $A$ is denoted as $A'$. For a real matrix $A$, $B=A'$ yields $B = A^T$ and for a complex matrix $A$, $B=A'$ is the complex conjugate transpose $B = \bar{A}^T$.
  
- If $A = \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix}$ then $B=A'$ gives $B = \begin{bmatrix} 2 & 6 \\ 3 & 7 \end{bmatrix}$.

- If $C = \begin{bmatrix} 2 & 3+i \\ 6i & 7i \end{bmatrix}$ then $C^t=C'$ gives $C^t = \begin{bmatrix} 2 & -6i \\ 3-i & -7i \end{bmatrix}$.

- If $u=[0 \ 1 \ 2 \ \ldots \ 9]$ then $v=u(3:6)'$ gives $v$ as $\begin{bmatrix} 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$. 
Row Major versus Column Major Storage

- Most programming languages like C or Java store their arrays in row major order.

- Matlab (and Fortran) store their arrays in column major order.

- To the programmer, this makes no difference unless a large array’s indices are accessed in the opposite order to the order used to store the array (lots of paging may result, slowing down execution, but you will still get the correct result).

- Sometimes, data created in row major order by another program has to be read by MatLab (so row major order has to be converted to column major order).
• The difference between row-major order and column-major order is simply that the order of the dimensions is reversed. That is, in row-major order the rightmost indices vary faster as one steps through consecutive memory locations, while in column-major order the leftmost indices vary faster.

• In 2D, one simply needs to take the transpose of the 2D array to convert row to column or column to row major order.

• For higher dimensions things are a bit more complicated. The ordering determines which dimensions of the array are more consecutive in memory.

• Any multi-dimensional array is really a 1D array with an memory-offset
calculation based on the indices of each of its elements. You can do such calculation in a language like C but not in MatLab.

- In row-major order, the last dimension is contiguous, so that the memory-offset of this element in a $N_1 \times N_2 \times ... N_d$ array, $k = 1, ..., d$ for element $(n_1, n_2, ... n_d)$, where $n_k \in [0, N_k - 1]$ is a zero based index, is given by:

$$n_d + N_d(n_{d-1} + N_{d-1}(n_{d-2} + N_{d-2}(... N_2 n_1)...)) = \sum_{k=1}^{d} \left( \prod_{l=k+1}^{d} N_l \right) n_k$$

- In column-major order, the first dimension is contiguous, so that the memory-offset of this element in a $N_1 \times N_2 \times ... N_d$ array, $k = 1, ..., d$ for element $(n_1, n_2, ... n_d)$, where $n_k \in [0, N_k - 1]$ is a zero based index,
is given by:

\[ n_1 + N_1(n_2 + N_2(n_3 + N_3(... N_{d-1}n_d)...)) = \sum_{k=1}^{d} \left( \prod_{l=1}^{k-1} N_l \right) n_k \]

- We can use MatLab’s `reshape` and `permute` to do this conversion.

- `B=permute(A, order)` rearranges the dimensions of A so that they are in the order specified by the vector `order`. B has the same values as A but the order of the subscripts needed to access any particular element is rearranged as specified by `order`. All the elements of order must be unique. For example:

\[
A = \begin{bmatrix} 1 & 2; & 3 & 4 \end{bmatrix}
\]

A =
permute(A,[2 1])
ans = 
 1 3
 2 4

For 2D this is the same as the transpose of the original 2D matrix.

- Consider a 3D array with size size_z * size_x * size_y generated by a C program on a SUN workstation: the data is stored in row major order and Big Endian (PCs and Macs are in Little Endian as the bytes of integers and floats are reversed). For example, bytes 1 2 3 4 in Big Endian are converted to bytes 4 3 2 1 in Little Endian.
- Consider some MatLab code to do this conversion on a 3D array:

```matlab
pathname='/Volumes/barron/DATA3D'; % Some path
stemname='sin3D.'; % Some stem
% Open Volumes/barron/DATA3D/sin3D.9 in Big Endian
% mode for reading. Little Endian would require
% the 'b' be changed to 'l'

% Open /Volumes/barron/DATA3D/sin3D.9 for reading
% in Big Endian
fd=fopen([pathname '/' stemname num2str(9)],...
    'r','b');
```
% Read size_z*size_x*size_y of unsigned short data
% (2 bytes): in MatLab, use datatype 'uint16'

volume=fread(fd,size_z*size_x*size_y,'uint16');

% Reshape the array to have the opposite dimensions
% and then permute that array so that the 3rd
% dimension becomes the 1st dimension, the 2nd
% dimension remains the same and the 3rd dimension
% becomes the first ==> the data is now in column
major order with the correct dimensions
volume=permute(reshape(volume,...
        [size_x size_y size_z]),[3 2 1]);

**Initialization and Appending Rows/Columns**

- An $m \times n$ matrix, $A$, can be initialized to a zero matrix by the command $A=\text{zeros}(m,n,\text{'double'})$. The initialization reserves a contiguous block of the computer’s memory at $A$ (in column major order) with enough room for $m \times n$ double elements. This is necessary if loops are to be compiled by JIT (MatLab’s Just In Time compiler, see below) and hence executed more efficiently.

- If the rows or columns of a matrix are computed in a loop and appended to the matrix in each iteration of the loop (definitely not efficient!!!), then
you can initialize the matrix to the null matrix $A=[ ]$ and then append a row or column of any size to $A$.

- The command $A=[A \ u]$ appends column vector $u$ to $A$ while $A=[A; \ v]$ appends the row vector $v$ to the rows of the original $A$.

- If $A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$, $u=[5 \ 6 \ 7]$ and $v=\begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix}$, then

  - $A=[A; \ u]$ using the original $A$ produces $A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 5 & 6 & 7 \end{bmatrix}$, now $A$ is a $4 \times 3$ matrix,

  - $A=[A \ v]$ using the original $A$ produces $A = \begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 4 \end{bmatrix}$, making $A$ a $3 \times 4$ matrix,
- $A = [A \quad u']$ (note $A'$ is the transpose of $A$) using the original $A$
  produces $A = \begin{bmatrix} 1 & 0 & 0 & 5 \\ 0 & 1 & 0 & 6 \\ 0 & 0 & 1 & 7 \end{bmatrix}$, a $3 \times 4$ matrix

- $A = [A \quad u]$ produces an error because the original $A$ has 3 rows and 3 columns and $u$ has 1 row and 3 columns.

- $B = []; \quad B = [B; 1 \quad 2 \quad 3]$ produces array $B = [1 \quad 2 \quad 3]$ and

- $B = []; \quad \text{for } k=1:3, \quad B = [B; k \quad k+1 \quad k+2];$ end produces:
  
  $B = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix}$.

- Any row(s) or column(s) of a matrix can be deleted by setting the row or column to the null vector. Examples:

  $u = [1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9]$
u =  
   1 2 3 4 5 6 7 8 9  

length(u)  
ans =  
   9  

% deletes all elements of u except 1 to 4  

u(5:length(u))=[]  

u =  
   1 2 3 4  

length(u)  
ans =  
   4  


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

1 2 3 4 5
6 7 8 9 10
11 12 13 14 15

% delete the 2nd row of A
A(2,:) = []
A =

1 2 3 4 5
11 12 13 14 15

% deletes the 3rd through 5th columns of A
\[ A(:,3:5)=[] \]

\[ A = \]

\[
\begin{bmatrix}
1 & 2 \\
11 & 12
\end{bmatrix}
\]

**Utility Matrices**

- Some MatLab utility matrices:

  % returns a \( m \) by \( n \) matrix with ones on the main diagonal
  \texttt{eye}(m,n)

  % returns a \( m \) by \( n \) matrix of zeros
  \texttt{zeros}(m,n)

  % returns a \( m \) by \( n \) matrix of ones

ones(m,n)
%
% returns a m by n matrix of random numbers
rand(m,n)
%
% returns a m by n matrix of normally distributed numbers
randn(m,n)
%
% generate a square n by n matrix where the sum of rows,
% columns and diagonals is the same
magic(n)
%
% generates a diagonal matrix with vector v
% on the diagonal
diag(v)
%
% extracts the first diagonal of matrix A as a vector
diag(A)
% extracts the 1st upper off-diagonal vector of A
diag(A,1)

• Some functions that can be used in matrix manipulation:

  % rotate a matrix by 90 degrees
  rot90

  % flip a matrix from left to right
  fliplr

  % flip a matrix from up to down
  flipud

  % extract the lower triangular part of a matrix
  tril
% extract the upper triangular part of a matrix
triu

% change the shape of a matrix: the number of
% elements in the changed matrix must be the same
reshape

• Examples of matrix manipulation using utility matrices and functions:

% create a 3×3 identity matrix. Commands zeros, ones
% and rand work in a similar way
eye(3) % assumes a square matrix
ans =
    1 0 0
0 1 0
0 0 1

% Create matrix B using submatics: ones, zeros and the identity matrix of specified sizes
B=[ones(3) zeros(3,2); zeros(2,3) 4*eye(2)]

B =
1 1 1 0 0
1 1 1 0 0
1 1 1 0 0
1 1 1 0 0
0 0 0 4 0
0 0 0 0 4
% Pull out the diagonal of B in a row vector.
% Without the transpose operation it would be a column vector
diag(B)'

ans =
    1  1  1  4  4

% Transpose the first upper diagonal vector of B
% If you use a negative values for the 2nd argument
% you get the first lower off-diagonal vector
diag(B,1)'

ans =
% Create D by putting vector d on the main diagonal, % vector d1 on the first upper diagonal and vector d2 % on the second lower diagonal with all other elements % being zeros. Note that d, d1 and d2 must be the right % sizes, otherwise you get an error.

d=[2 4 6 8];
d1=[-3 -3 -3];
d2=[-1 -1];
D=diag(d)+diag(d1,1)+diag(d2,-2)
D =

1 1 0 0
- The general MatLab command to create a matrix of numbers over a given range with a specified increment is:

\[ v = \text{InitialValue: Increment: FinalValue} \]

- For example:

\[
% \text{produce } a=\begin{bmatrix} 0 & 10 & 20 & 30 & \ldots & 100 \end{bmatrix} \text{ (step 10)}, \\
a=0:10:100
\]
a =
    0  10  20  30  40  50  60  70  80  90  100

% produce b=[0  pi/50  2*pi/50 ...  pi/4], a linearly
% spaced vector from 0 to 2*pi spaced by step pi/50
b=0:pi/50:pi/8
b =
    0  0.0628  0.1257  0.1885  0.2513  0.3142  0.3770

% produces a=[-2 -1  0  1  2  3  4 ...  10] (step 1)
u=-2:10
u =
    -2  -1  0  1  2  3  4  5  6  7  8  9  10

- Square brackets are only required if vector concatenation is required.
$u=[1:10\ 33:-2:23]$ requires square brackets to concatenate 2 vectors $[1\ 2\ 3\ \ldots\ 10]$ and $[33\ 31\ 29\ \ldots\ 23]$

$u=[1:10\ 33:-2:23]$

\[ u = 
\begin{array}{ccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 33 & 31 & 29 & 27 & 25 & 23 \\
\end{array}
\]

- $\text{linspace}(a,b,n)$ generates a linearly spaced vector of length $n$ from $a$ to $b$. $u=\text{linspace}(0,20,5)$ generates $u=[0\ 5\ 10\ 15\ 20\ 25]$. In general, $u=\text{linspace}(a,b,n)$ is the same as $u=a:\ (b-a)/(n-1):b$. So $u=\text{linspace}(0,20,5)$ is the same as $a=0:5,20$, i.e. $(b-a)/(n-1)$ is $(20-0)/(5-1)=5.$.

- $\text{logspace}(a,b,n)$ generates a logarithmically spaced vector of length
n from $10^a$ to $10^b$. Thus $v = \text{logspace}(0, 3, 4)$ generates the vector $v = [1 \ 10 \ 100 \ 1000]$. Hence $\text{logspace}(a, b, n)$ is the same as $10.^(\text{linspace}(a, b, n))$.

\begin{verbatim}
logspace(0, 3, 4)
ans =
    1   10   100  1000

linspace(0, 3, 4)
ans =
    0   1   2   3

10.^(linspace(0, 3, 4))
ans =
    1   10   100  1000
\end{verbatim}
Matrix Multiplication

- Matrix multiplication is $C=A \times B$ where $A$ is a $m \times n$ matrix and $B$ is a $n \times k$ matrix. $C$ is a $m \times k$ matrix. So the number of columns of $A$ must equal the number of rows of $B$. $C=A \times B$ is equivalent to the multiplication performed in the nested loop below:

```matlab
% Matrix Multiplication
% Multiply A_m,k by B_k,n (#columns of A=#rows of B)
% to obtain matrix C_m,n
A=[1 2 3 4 5; 6 7 8 9 10];  % A_2,5
B=[1 2 3; 4 5 6; 7 8 9; 10 11 12; 13 14 15];  % B_5,3
[m,p]=size(A)
```
m =
   2

p =
   5

[p,n]=size(B)

p =
   5

n =
   3

for i=1:m
for j=1:n
    C(i,j)=0;
end
for k=1:p
    C(i,j)=C(i,j)+A(i,k)*B(k,j);
end
end
end
A
A =
    1   2  3   4   5
    6   7  8   9  10
B
B =
C % Matrix Multiplication by Nested Loops

\[
C = \\
\begin{bmatrix}
135 & 150 & 165 \\
310 & 350 & 390 \\
\end{bmatrix}
\]

C=A*B % MatLab Matrix Multiplication

C =
\[ A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \]

% Note what happens when the number of columns
% of the first matrix does not equal the number
% of rows of the second matrix, as in B*A

B*A

Error using  *
Inner matrix dimensions must agree.

- A+B or A–B add or subtract matrices A and B by adding or subtracting
  the corresponding elements of A or B, if A and B are the same size.
B = [4 3; 2 1];

A + B
ans =

5  5
5  5

A - B
ans =

-3  -1
1   3

・A/B is valid for same sized matrices and equals \( AB^{-1} \)
ans =

    1.5000    -2.5000
    2.5000    -3.5000

- $A^2$ is $A \times A$ makes sense if $A$ is square.

$A^2$

$A =$

    1    2
    3    4

- If $\alpha$ is a scalar then $A + \alpha$ or $A - \alpha$ add or subtracts $\alpha$ from each element of $A$.

$A = [ 1 \ 2; \ 3 \ 4 ];$
A-3

ans =

-2  -1

0    1

- Similarly, $A \times \alpha$ (or $\alpha \times A$) multiplies each element of $A$ by $\alpha$.

A*3

ans =

3    6

9    12

- Vectors are single row/column matrices. If $u$ and $v$ are $n$ sized vectors ($n \times 1$ matrices) then $u \times v$ produces an error but $u \times v'$ or $u' \times v$ produce
the outer and inner products of the two vectors.

\[ u = [1 \ 2 \ 3]; \]
\[ v = [4 \ 5 \ 6]; \]
\[ u \cdot v \]
\[ \text{Error using } * \]
\[ \text{Inner matrix dimensions must agree.} \]
\[ u \cdot v' \]
\[ \text{ans} = \]
\[ \begin{array}{c}
32 \\
\end{array} \]
\[ u' \cdot v \]
\[ \text{ans} = \]
\[ \begin{array}{ccc}
4 & 5 & 6 \\
\end{array} \]
Note that $u^T \times v$ is a singular matrix as any row or column is any other row or column scaled by a constant factor. The rows and columns are not linear independent of each other.

- In addition to normal or right division (/) there is left division (\) in MatLab. The matrix equation $A s = b$ has solution $s = A \backslash B$. Thus $A \backslash B$ is almost the same as $\text{inv}(A) \times B$ but faster and more numerically stable.
Solving Linear Systems of Equations

- Consider the solution for the unknowns, $x$, $y$ and $z$ for the following 3 equations:

\[
\begin{align*}
3x + 4y - 7z &= 16 \\
9x + 2y + 102z &= 1002 \\
-6x * 4y + z &= 14
\end{align*}
\]

This is 3 linear equations in 3 unknowns (and usually has a unique solution). These 3 equations can be written in matrix form as:

\[
\begin{pmatrix}
3 & 4 & -7 \\
9 & 2 & 102 \\
-6 & 4 & 1
\end{pmatrix}
\begin{bmatrix}
x \\
y \\
z
\end{bmatrix}
= 
\begin{bmatrix}
16 \\
1002 \\
14
\end{bmatrix}
\]
We can write this as $A\vec{s} = B$ where $\vec{s} = (x, y, z)$.

\[
A = \begin{bmatrix}
3 & 4 & -7 \\
9 & 2 & 102 \\
-6 & 4 & 1
\end{bmatrix}; \\
B = \begin{bmatrix}
16 \\
1002 \\
14
\end{bmatrix}; \\
\vec{s} = A\backslash B
\]

\[
\begin{bmatrix}
8.0862 \\
13.4175 \\
8.8470
\end{bmatrix}
\]

$\vec{s}$ satisfies all 3 equations simultaneously:

\[
\% \text{ 1st equation} \\
s(1) * 3 + s(2) * 4 + s(3) * -7 - 16
\]
ans =
  -2.8422e-14
% 2nd equation
s(1)*9+s(2)*2+s(3)*102-1002
ans =
  0
% 3rd equation
s(1)*-6+s(2)*4+s(3)*1-14
ans =
  -1.7764e-14
% The residual vector is 0 within roundoff error
r=A*s-B
ans =

1.0e-13 *

-0.2842

0

-0.1776

% The norm of the residual is computed as the length of the vector r
% Called Euclidean norm or L2 norm
% $\sqrt{r(1)^2 + r(2)^2 + r(3)^2}$

ans =

3.3516e-14
% Built-in Matlab function norm computes this

```
norm(r)
ans =
```

```
3.3516e-14
```

**Least Squares**

- Lets add two 2 rows TO $A$ (we also need to add 2 more rows to $B$). These numbers were just arbitrarily chosen! Then when we solve $As = B$ we are finding the $s$ vector that “best” fits the data. This is the **least squares** solution to an over constrained linear system of equations.

```
A = [3 4 -7; 9 2 102; -6 4 1; 2 2 2; 4 -19 6];
B = [16; 1002; 14; 20; 0 ];
```
s = A \ B

s =
6.5426
4.8799
9.1282

% Residual

r = A \ s - B

r =
-40.7504
-2.2763
-24.6078
21.1014
-11.7780

\text{norm}(r)

\text{ans} =

53.4351

We can see the residual vector and its norm are not that close to 0 now: the solution vector \( s \) is not a perfect fit to the system of equations now.

\textbf{Vectorization versus Serialization}

- In the case of scalars, \( 5/3 \) is 0.6, which is \( \frac{3}{5} \) or \( 3 \times 5^{-1} \).

- How does one compute a dot product of two vectors, which requires element by element multiplication and summing? Element by element
multiplication, division and exponentiation between 2 matrices of the same size is done by preceding the corresponding system by a period:

`. * % element by element multiplication
./ % element by element left division
.\ % element by element right division
.^ % element by element exponentiation
.' % non-conjugated transpose

This is called **vectorization** and is a powerful MatLab tool.

- Consider \texttt{A.*B}. This equivalent to

  \begin{verbatim}
  % The size of A and B must be the same in each dimension
  \end{verbatim}
for i=1:size(A,1)
for j=1:size(A,2)
    C(i,j)=A(i,j)*B(1,j)
end % for j
end % for i

This nested loop is the serialization of matrix element by element multiplication. Just In Time (JIT) compilation of this serialized code can produce code that executes almost as fast as the vectorized version of the code.
JIT - Just In Time Compiler

- MatLab is an **interpreted** language in that the program source code directly executed, statement by statement.

- **Compiled** languages require that the code first be explicitly translated into a lower-level machine language executable and then this is executed.

- Usually compiled code runs much faster than interpreted code but interpreted code is considerably more flexible. For example, MatLab lets array sizes change in loops!

- In the early days, interpretation was line by line but these days many interpreted languages, such as Java or Python, use an intermediate representation, which combines compiling and interpreting. In this case, a
compiler may output some form of bytecode or threaded code, which is then executed by a bytecode interpreter.

- JIT (or Just In Time) compilation evaluates loop code the first time the code is executed to see if it is suitable for compilation: if it is bytecode is produced, which usually runs much faster.

- MatLab code containing loops may benefit from JIT acceleration if the code has the following properties:
  
  1. The loop is a For Loop.
  2. The loop contains only logical, character, double precision variables.
  3. The loop uses arrays that are 3D or less.
4. All variables in the loop are defined prior to loop execution.

5. Memory for all variables maintain constant size and type during loop execution.

6. Loop indices are scalar quantities.

7. Only built-in MatLab functions are called.

8. Conditional statements (if-then-else and switch-case) involve only scalar comparisons.

9. Each line within the loop contain no more than 1 assignment statement.

- An example: compute \( \sin(x) \) at \( 10^7 \) points.

\[
N=1e7;
\]
% generate sin(x) at 1e7 points - vectorized solution

disp('Vectorized solution time:');
tic
x=linspace(0,2*pi,N);
y=sin(x);
elapsed_time1=toc

% generate sin(x) at 1e7 points - JIT solution

disp('JIT solution time:');
tic
i=0; % define variable i before the loop
x=zeros(1,N);
% allocate space for y and initialize as 0's
y=zeros(1,N);
for i=1:N % scalar loop index
    y(i)=sin(2*pi*(i-1)/N); % built-in fct sin
end
elapsed_time2=toc
disp('Vectorization speed up:');
elapsed_time2/elapsed_time1

produces the output:

<table>
<thead>
<tr>
<th>Machine</th>
<th>Vectorized Time</th>
<th>JIT Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUN 64bit (brown) MatLab 5 (1999)</td>
<td>5.1008</td>
<td>68.1895</td>
<td>13.3683</td>
</tr>
<tr>
<td>SUN 64bit (mccarthy) MatLab 2009b</td>
<td>1.0018</td>
<td>5.8502</td>
<td>5.8376</td>
</tr>
<tr>
<td>MAC 2.16GHz 32bit, Intel Core Duo, MatLab 2010a, 2003</td>
<td>1.1854</td>
<td>1.2942</td>
<td>1.0919</td>
</tr>
<tr>
<td>MAC 3.06GHz 64 bit, Intel dual core, MatLab 2013a, 2009</td>
<td>0.3305</td>
<td>1.4421</td>
<td>4.3334</td>
</tr>
<tr>
<td>MAC 2.70GHz 64 bit, Intel quad core, MatLab 2013a, 2012</td>
<td>0.1495</td>
<td>0.4133</td>
<td>2.7651</td>
</tr>
</tbody>
</table>
• Vectorized code is usually superior to the nested loops (even when JIT is used). JIT did not exist for MatLab 5 for sure.

• Both the JIT and vectorized solutions take about the same time on the older machine Mac notebook but on the newer machines (with later versions of MatLab) vectorization is definitely faster. Before MatLab 6.5, the for Loop would have taken order of magnitude more time to execute (see the MatLab 5 result).

• The 2009 and 2012 machines are faster because of the multi-core architectures.

• Here, JIT is not helpful because vectorizing the solution is so simple. In more substantial problems, it may be difficult or impossible to derive a
vectorized solution. In this case, JIT acceleration would be very useful!!!

More on Vectorization

• For vectors $u$ and $v$, $u \cdot v$ produces $[u_1v_1 \ u_2v_2 \ u_3v_3\ldots]$, 

\[
\begin{align*}
u &= [1 \ 2 \ 3]; \\
v &= [4 \ 5 \ 6]; \\
u \cdot v \\
\text{ans} &= \\
&\begin{bmatrix} 4 & 10 & 18 \end{bmatrix} \\
\end{align*}
\]

$u \div v$ produces $[u_1/v_1 \ u_2/v_2 \ u_3/v_3\ldots]$ while $u \ ^{\wedge} v$ produces

$[u_1^{v_1} \ u_2^{v_2} \ u_3^{v_3}\ldots]$. 
u./v
ans =
0.2500  0.4000  0.5000

u.^v
ans =
1  32  729

• For same sized matrices $A$ and $B$, $C = A.*B$ produces a matrix with elements $C_{i,j} = A_{i,j} \times B_{i,j}$. $A^2$ is matrix multiplication $A \times A$ while $A.*2$ computes $A_{i,j} \times 2$.

A=[1 2; 3 4];
A*A
ans =

    7    10
   15    22

A.*A
ans =

    1     4
    9    16

- For scalars, \(1./v\) computes \([1/v_1, 1/v_2, 1/v_3...]\) and \(\pi.^v\) computes \([\pi^{v_1}, \pi^{v_2}, \pi^{v_3}...]\).

\[\begin{array}{c}
v=[4 \ 5 \ 6];
1./v
\end{array}\]
ans =
0.2500  0.2000  0.1667

- Some more examples:

```matlab
A=[1 2 3; 4 5 6; 7 8 9];
x=A(1,:)’
x =
    1
    2
    3
% Here x is a column vector
% and x’ is a row vector
x’*x  % inner or dot product
```
ans =

14

\texttt{x}*\texttt{x}' \% \text{outer product}

ans =

1 2 3
2 4 6
3 6 9

\texttt{A}*\texttt{x} \% \text{matrix multiplied by a vector}

ans =

14
32
50
A^2 % A is square and A^2 is A*A

ans =

30 36 42
66 81 96
102 126 150

A.^2 % elements of A are raised to the power of 2

ans =

1 4 9
16 25 36
49 64 81
2D Graphs in MatLab

The plot function

- If a variable `ydata` has `n` values corresponding to `n` values of variable `xdata`, then `plot(xdata,ydata)` produces a graph with `xdata` on the horizontal axis and `ydata` on the vertical axis.

- Consider the following MatLab code using `plot`:

```matlab
% create vector x
x = 0:0.1:20;
% calculate y
y = exp(0.1*x).*sin(x);
```
% plot x vs. y
plot(x,y)
% label x axis
xlabel('Time (t) in Seconds')
% label y axis
ylabel('The Response Amplitude in mm')
% put a title
title('A Simple 2D Plot of x versus y')
% file plot0.jpg
print('plot0.jpg','-djpeg');

It generates the plot shown below:
Simple 2D plot of \( f(x) = e^{t/10} \sin(t) \) using \texttt{plot}. 
• If a figure window already existed, this plot command would draw this figure over it. To keep the original figure contents it is necessary to use a \texttt{figure} command first. A little later there is an example.
• Consider using `fplot` to plot this function:

```matlab
fplot('exp(-0.1*x).*sin(x)',[0,20])
% Note the use of latex to do the
% equation for the y axis
xlabel('x'), ylabel('f(x) = e^{x/10} \sin(x)')
title('A function plotted with fplot')
print('plot1.jpg','-djpeg')
```

The 1\textsuperscript{st} argument to `fplot` is the function to be plotted while the 2\textsuperscript{nd} argument gives the range for the x values of the function. The `print` command generates a jpg file of the figure displayed in the figure window. The figure below shows the graph produces by these commands.
Simple 2D plot of $f(x) = e^{x/10} \sin(t)$ using \texttt{fplot}.
• **ezplot** can be used to plot \( f(x) = e^{-x/10} \cos(x) \) in a very simple way (using all default values). MatLab code to do this:

```matlab
ezplot('exp(-0.1*x).*cos(x)', [0, 20])
print plot2.jpg -djpeg
```

The figure below shows the output graph.
Simple 2D plot of $f(x) = e^{x/10} \cos(t)$ using \texttt{ezplot}.
• Consider multiple graphs plotted at the same time:

```matlab
% Generate 30 values between 0 and 2*pi
x=linspace(0,2*pi,30);
y=sin(x);
z=cos(x);

% create W as a 30*2 array
% 1st column are the sine values
% 2nd column are the cosine values
W=[y;z];
plot(x,W)
title('Sine and Cosine plotted against x');
print plotA1.jpg -djpeg
```
% Use a new figure window
figure;
plot(W,x)
title('x plotted against Sine and Cosine');
print plotA2.jpg -djjpeg

- The first figure below shows the sine and cosine values (stored as a 2D array \( W \) with the 1\(^{st} \) column as the sine values and the 2\(^{nd} \) column as the cosine values) plotted against \( x \).

- The next figure below shows \( x \) plotted against these sin and cosine values (hence, the graphs are sideways!).
Simple 2D multiple plot of sin and cosine values against 30 equally spaced values between 0 and $2\pi$. 
Simple 2D multiple plot of 30 equally spaced values from 0 to $2\pi$ against their sin and cosine values.
Customizing Graphs

• Notice that in the figures for the sine and cosine, MatLab chose solid linetype and the colours blue and green for the plots.

• You can customize your plots in MatLab to use different colours, markers and linestyles by using a $3^{rd}$ argument in `plot`. This $3^{rd}$ argument is a character string with one or more characters from the Table below:
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Colour</th>
<th>Symbol</th>
<th>Marker</th>
<th>Symbol</th>
<th>Linestyle</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>Blue</td>
<td>.</td>
<td>Point</td>
<td>-</td>
<td>Solid line</td>
</tr>
<tr>
<td>g</td>
<td>Green</td>
<td>o</td>
<td>Circle</td>
<td>:</td>
<td>Dotted line</td>
</tr>
<tr>
<td>r</td>
<td>Red</td>
<td>x</td>
<td>Cross</td>
<td>:</td>
<td>Dotted line</td>
</tr>
<tr>
<td>c</td>
<td>Cyan</td>
<td>+</td>
<td>Plus sign</td>
<td>–</td>
<td>Dash-dot line</td>
</tr>
<tr>
<td>m</td>
<td>Magenta</td>
<td>*</td>
<td>Asterisk</td>
<td>none</td>
<td>no line</td>
</tr>
<tr>
<td>y</td>
<td>Yellow</td>
<td>s</td>
<td>Square</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>Black</td>
<td>d</td>
<td>Diamond</td>
<td></td>
<td></td>
</tr>
<tr>
<td>w</td>
<td>White</td>
<td>v</td>
<td>Triangle (down)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>^</td>
<td>Triangle (up)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;</td>
<td>Triangle (left)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt;</td>
<td>Triangle (right)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td></td>
<td>p</td>
<td>Pentagram</td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
<td>h</td>
<td>Hexagram</td>
<td></td>
<td></td>
</tr>
<tr>
<td>none</td>
<td></td>
<td>none</td>
<td>no marker</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Setting colours, markers and linetypes
• The following MatLab code:

```matlab
x=linspace(0,2*pi,30);
y=sin(x);
z=cos(y);
plot(x,y,'b:p',x,z,'c-');
title('Sine and Cose against x');
print plotB.jpg -djpeg
```

produces:
30 equally spaced values from 0 to $2\pi$ against their sin and cosine values. The sine plot is dotted blue with pentagram markers (’b:p’) while the cosine plot is a solid cyan line (’c-’).
**Grids, Axes and Labels**

- The `grid on` commands add grid lines to the current plot at the tick marks. By default MatLab has `grid off`.

- Normally, a graph is enclosed by solid lines called an **axes box**. Use `box off` to turn this off and `box on` to turn it on again.

- `xlabel` and `ylabel` can be used to label the $x$ and $y$ axes. The `title` command allows a title to be added above the plot.

- An example, consider the MatLab code for the plot we made earlier with some changes:

  ```matlab
  close all
  ```
x=linspace(0,2*pi,30);
y=sin(x);
z=cos(x);
W=[y;z];
plot(x,W)
box off
xlabel('X','Fontsize',16);
ylabel('Y and Z','Fontsize',16);
grid on
text(2.5,0.7,'sin x','Fontsize',14);
text(1.1,0.7,'cos x','Fontsize',14);
title({’Sine and Cosine Y and Z values’};...
’plotted against X’),’Fontsize’,18);
print plotC.jpg -djjpeg

• Note that we use `close all` to close all windows currently open in this session.

• We control the fontsize in `title`, `xlabel`, `ylabel` and `text` using attribute `FontSize` and values 18, 16 and 14.

• Note the grid printed as dotted lines. `box off` means there are no top or right solid lines on the graph.

• We print two (or more) strings in the `title` command (or in an `xlabel`, `ylabel` or `text` commands) by enclosing these strings separated by
semi-colons with curly braces. These curly braces tell MatLab each string is an element of a cell array.

- The . . . allows a MatLab command to broken at a white space and continued on the next line. This is good for preventing long command lines from wrapping around onto the next line.

- The figure below shows what is plotted:
Sine and Cosine X and Y values plotted against X values, box off, grid on, double string title, x and y labels, and text strings (all with different fontsizes).
• MatLab gives you control over the appearance of your plot axes. A few options available:

1. `axis[(xmin xmax ymin ymax)]` - control the axes limits - good when making comparisons across multiple plots.

2. `axis auto` - lets MatLab determine axis defaults.

3. `axis on` - Turn on axis labelling, tic marks and background.

4. `axis off` - Turn off axis labelling, tic marks and background.

• Another way to make the multiple sine and cosine plots would be to use `hold on`:

```matlab
x=linspace(0,2*pi,30);
y=sin(x);
```
% sin(x) as red line
plot(x, y, 'r');

z = cos(x);

hold on

% cos(x) as green line
plot(x, z, 'g');

legend('sin(x)', 'cos(x)');

title('Use of hold on');

print plotD.jpg -djjpeg

which generates the following graph:

- Without hold on the sine graph would have been overwritten by the cosine graph.
Use of hold on.
• `legend` allows each plot line to be associated with a character string. In this example, a red solid line is associated with ’sin(x)’ while a green solid line is associated with ’cos(x)’.
Subplots

- A single MatLab window can hold more than 1 plot. `subplot(n, m, p)` divides a window up into an $m \times n$ matrix and indicates the $p$'th subplot is active. For $m=2$ and $n=2$ we have a $2 \times 2$ array with 4 $p$ values:

\[
\begin{array}{cc}
1 & 2 \\
3 & 4 \\
\end{array}
\]

- A MatLab example:

```matlab
x=linspace(0,2*pi,30);
y=sin(x);
z=cos(x);
a=2*sin(x).*cos(x);
```
% eps - machine epsilon
b=sin(x)./(cos(x)+eps);

% upper left of 2 by 2 subplots
subplot(2,2,1)
plot(x,y);
axis([2 2*pi -1 1]);
title('sin(x)');

% upper right of 2 by 2 subplots
subplot(2,2,2)
plot(x,z);
axis([2 2*pi -1 1]);
title('cos(x)')
% lower left of 2 by 2 subplots
subplot(2,2,3)
plot(x,a);
axis([2 2*pi -1 1]);
title('2*sin(x)*cos(x)');

% lower right of 2 by 2 subplots
subplot(2,2,4)
plot(x,b);
axis([2 2*pi -30 30]);
title('sin(x)/(cos(x)+eps)');
print plotE.jpg -djpeg
4 subplots of $\sin(x)$, $\cos(x)$, $2 \sin(x) \cos(x)$ and $\sin(x)/(\cos(x) + \text{eps})$. 
• `eps` or `eps('double')` is is the distance from 1.0 to the next largest double-precision number, which is $2^{(-52)}$ or about 2.2204e-16 in base 10. `eps('single')` is the distance from 1.0 to the next largest single-precision number, which is $2^{-23}$ or about 1.1921e-07 in base 10. Adding `eps` prevents division by zero from happening when $\cos(x)$ is zero.

• `xlabel`, `ylabel`, `title`, `text`, `hold on`, `grid`, etc apply to the active subplot.

• A `drawnow` command forces MatLab to draw all figures computed so far (MatLab may have buffered some).
Log Plots

- We can use `semilogx` to plot a logarithmically scaled x-axis, `semilogy` to plot a logarithmically scaled y-axis and `loglog` to plot a graph with both axes logarithmically scaled.

- For example:

```matlab
x = 0:0.1:10;
semilogx(10.^x, x);
print plotF.jpg -djpeg
xlabel('Logarithmically scaled X axis');
ylabel('Lineay scaled Y axis');
title('Example of semilogx');
```
produces the graph:

Logarithmically scaled x axis example.
Stacked Area Plots

- The `area` function can be used to build stacked plots. `area(x, y)` and `plot(x, y)` are the same for vectors `x` and `y` except the area the plot is filled with colour for `area`.

- To stack areas, use `area(X, Y)`, where `Y` is a matrix and `X` is a matrix or vector whose length equals the number of rows in `Y`. If `X` is omitted, `X=1:size(Y,1)` is used.

- If `Y` consists of `n` vectors of data, then the `i^{th}` vector is added to the first `i-1` before it before being plotted. Each area is coloured differently than the other areas.

- Consider the following MatLab code:
% stacked area plot

% x axis 2011:1:2020
X=linspace(2011,2020,10);

% y1 axis 20000 to 300000
% y2 axis 155000 50 1000000

% Make column vectors
y1=linspace(50000,300000,10)';
y2=linspace(100000,1000000,10)';

Y=zeros(size(y1,1),size(y1,2) * 2);
Y(:,1)=y1;
Y(:,2)=y2;

% y2 is stacked on top of y1
area(X,Y);
colormap('summer');
set(gca,'XLim',[2011 2020])
set(gca,'YLim',[0 1800000])
set(gca,'YTick',0:200000:1400000)
set(gca,'YTickLabel',{'','200000','400000','600000','800000','1000000','1200000','1400000'})
stg=['
\fontsize{20}\color{black}\bf '...
'1,000,000 more jobs than students '...
'by 2020'];
text(2011,1600000,stg);
stg=['\fontsize{24}\color{black}\bf '...'...
    'opportunity'];

text(2012,1100000, stg);

% draw a line line with coordinates
% (x1,y1) and (x2,y2)
% use line([x1 x2],[y1 y2])
line([2014 2016],[1000000 600000],...
    'linewidth', 2);
print stacked_area_plot.jpg -djpeg

• Note the use of latex commands fontsize and color to control the size and colour of text. \bf boldfaces the text.

• The $x$ coordinates are the years 2011 to 2020 while the $y$ coordinates the
numbers 0 to 1800000. The data only goes to 1400000, the extra space is for the title.

- `gca` is the MatLab function that gets the current axis handle (pointer). Then the set commands is used to set the `XLim`, `YLim`, `YTick` and `YTickLabel` properties of the axis. We use the default `XTick` and `XTickLabel` properties. It probably wasn’t necessary to explicitly set the `XLim` property as this is probably the default value. In the case of the $y$ axis, the ticks were different and numbers were printed in scientific notation instead of as whole integers.

- The blue line as drawn using `line([x1 x2], [y1 y2])` (it is blue because the is the first colour MatLab selects).
• This code produces the following plot:

Some Computer Science propaganda: Number of computing jobs versus the number of computer science students for 2011 to 2012. From http://www.code.org/stats.
Pie Charts

- Standard pie charts can be created using \texttt{pie(a,b)}, where \texttt{a} is a vector of values and \texttt{b} is an optional argument describing a slice or slices to be pulled out from the pie chart. \texttt{pie3} gives a 3D effect.

- A standard pie chart can be produced by the following MatLab code:

  
  \begin{verbatim}
  a=[0.024 0.976];
  pie(a,{'2.4%','97.6%'});
  colormap summer
  stg='\textsize{36}\textcolor{red} Students';
  text(-0.5,-0.3,stg);
  stg='\textsize{16}\textcolor{blue}\textbf{All other Math}';
  \end{verbatim}
text(0.15,0.15,stg);
stg='\fontsize{16}\color{blue}\bf and Sciences';
text(0.15,0.05,stg);
stg='\fontsize{16}\color{blue}\bf Computer Science';
text(-1.25,1.1,stg);
line([-0.15 -0.05],[1.075 0.95],''linewidth'',2);
title({'[\fontsize{16}\color{darkgreen}\bf' ...'
'Percentage of CS students in all ' ...'
'Mathematics and science fields'],'' '})
print pie1.jpg -djjpeg
• A pie chart with a part “pulled” out (highlighted):

```matlab
a = [0.4 0.6];
explode = (a == min(a));
pie(a, explode, {'40%', '60%'});
colormap summer
% coordinates found by trial and error
stg = '\fontsize{36}\color{red}\bf Jobs';
text(0.1, 0.3, stg);
stg = '\fontsize{18}\color{red} All other Math';
text(-1.0, 0.15, stg);
stg = '\fontsize{18}\color{red} and Sciences';
text(-1.0, 0.05, stg);
```
The following pie charts are produced:
More Computer Science propaganda: (a) Percentage of computer science students versus science and mathematics students and (b) Percentage of jobs for computer science students and science and mathematics students. From http://www.code.org/stats.
Finally, following MatLab code produces:

```matlab
a=[0.5 1 1.6 1.2 0.8 2.1];
pie3(a);
title('3D pie chart');
print plotH.jpg -djpeg
s=sum(a(:));
a=cast(a/s*100,'int32')
```

produces the graph:
• `a` prints with values:

\[
a = 7 \quad 14 \quad 22 \quad 17 \quad 11 \quad 29
\]

which are exactly the percentages on the pie chart.
Filled Polygons

- Polygons can be filled with a colour using `fill`. The command `fill(x, y, 'color')` fills the polygon defined by column vectors `x` and `y` with colour `color`.

- Consider the following MatLab code:

```matlab
% 8 points on circle to make the stop sign polygon

% make the stop sign polygon

\[ t = (1:2:15)' \times \pi / 8; \]

\[ x = \sin(t); \]

\[ y = \cos(t); \]

% fill the polygon with red

fill(x, y, 'red')
```
axis square off

% print stop in white

text(0,0,'STOP',...)

'Color',[1 1 1],...

'FontSize',80,...

'FontWeight','bold',...

'HorizontAlalignment','center');

title('Stop Sign');

print stop_sign.jpg -djpeg

- The following figure is printed (with STOP in black for some reason, although it is white when run in MatLab?):
A “stop sign” filled polygon.
Plots using the Same or Different $y$ Axes

- Sometimes we may want to plot two different functions on the same $x$ axis but using different $y$ axes.

- The MatLab code to illustrate this is given as:

```matlab
x=-2*pi:pi/10:2*pi;
y=sin(x);
z=3*cos(x);
subplot(2,1,1), plot(x,y,x,z);
title('Two plots using the same $y$ axis');
subplot(2,1,2), plot(x,y,x,z);
title('Two plots using different $y$ axes');
```
and produces the following output:

Two plots on the same $y$ axis and on different axes.

Two plots using the same $y$ axis

Two plots using different $y$ axes
• Note that the 2 plots of the first graph have the same $y$ axis but the second plot has 2 different $y$ axes.
Bar and Stairs Plots

- Bar and stairs plots can be generated using `bar`, `barh` and `stairs` functions, with `bar3` and `bar3D` rendering the bar charts with a 3D effect.

- The following MatLab code:

```matlab
x=-2.9:0.2:2.9;
y=exp(-x.*x);
subplot(2,2,1)
bar(x,y)
title('2D Bar Chart');
subplot(2,2,2)
```

```matlab
```
bar3(x,y,'r');
title('3D Bar Chart');
subplot(2,2,3)
stairs(x,y)
title('Stair Chart');
subplot(2,2,4)
barh(x,y)
title('Horizontal Bar Chart');
print plotJ.jpg -djpeg

produces the graphs:
Upper left: 2D bar Chart, Upper right: 3D bar chart, lower left: stairs chart and lower right: horizontal bar chart.
Histogram Data

- Histograms counts the number of times a variable in a bin (a small range of values) occurs and presents this data as plot.

- The MatLab code:

  ```matlab
  % Specify the bins to use
  x=-2.9:0.2:2.9;
  % generate random normal data points
  % normal data is Gaussian data
  % r = randn(m,1) returns an m-by-1 matrix (a vector)
  y=randn(5000,1);
  % Draw histogram
  ```
hist(y,x)

title('Histogram of Gaussian Data')

print plotK.jpg -djpeg

produces:
Histogram of Gaussian data.
Stem Plots

- Discrete data can be plotted by using the `stem` function. The function `stem(x, z)` creates a plot of the data points in vector `z` connected to the horizontal axis at values of `x`. An optional argument can be used to specify the linestyle.

- The following MatLab code:

```matlab
% create a 30*1 matrix of random normal data
z=randn(30,1);
stem(z,'--');
title('Stem plot of random data');
print plotL.jpg -djpeg
```
plots:

Stem plot of random data.
Plots with Error Bars

- Often we would like to plot a function with its standard deviation shown as error bars.

- The following MatLab code:

```matlab
x=linspace(0,2,21);
% Use erf function to generate some Gaussian error function values for x
% ==> a smooth set of increasing values
y=erf(x);
% Generate an error array, e,
% with the same size as x
```
% Scale e by 10 to make these values small
% This simulates standard deviation measurements
e=rand(size(x))/10;
% Plot x versus y with error bars 2*e(i)
% for each point x(i),y(i) on the plot
errorbar(x,y,e);
title('Errorbar plot');
print plotM.jpg -djpeg

produces:
Errorbar plot.
• From wikipedia (for your information): \( \text{erf} \) is defined as:

\[
\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt
\]

• When the results of a series of measurements are described by a normal distribution \( i \) with standard deviation \( \sigma \) and expected mean value 0, then \( \text{erf}(\frac{a}{\sigma\sqrt{2}}) \) is the probability that the error of a single measurement lies between \( a \) and \( +a \), for positive \( a \). This is useful, for example, in determining the bit error rate of a digital communication system.
Scatter Plots

- A scatter plot of data shows the distribution of the data in a 2D area. Also know as a “bubble plot” it draws circles of varying sizes at each data point.

- The following MatLab code:

```matlab
% Generate 40 random numbers in [0,1]
x=rand(40,1);

% Generate 40 random numbers with 
% a Gaussian (normal) distribution
y=randn(40,1);

% Generate 40 areas starting at 21
```
% and uniformly increasing to 60
area=20+(1:40);

% Scatter plot the data
scatter(x,y,area);

box on

title('A scatter plot');

print plotN.jpg -djpeg

produces:
A scatter plot.
Text Formatting

- In the current versions of MatLab, all Tex (Latex) formatting commands can be used. Some of the most useful include:

  1. Superscripts and subscripts are specified by \(^\) and \(_\) respectively.
  2. \texttt{\fontname} and \texttt{fontsize} specify font type and size.
  3. Font style can be controlled by \texttt{\bf} (bold), \texttt{\it} (italic), \texttt{\sl} (slant or oblique) or \texttt{\rm} (roman).
  4. Colour can be controlled by \texttt{\color\{colorname\}} or \texttt{\color[rgb]\{r g b\}}. \texttt{r}, \texttt{g} and \texttt{b} specify the amount as red, green and blue as floating point numbers between 0 and 1. Each values is rounded to the nearest \(1/256\) value between 0 and 1; thus
there are at most 256 values for each colour or \( 256^3 = 16,777,216 \) colours in total (24 bit colour).

5. Backslash special Tex (Latex) characters to print them: \ (1 backslash), \{ and \} for curly braces, \_ for underscore and \^ for carrot.

- The next 2 tables show a large subset of symbols (including Greek symbols) that can be embedded on MatLab text strings.
<table>
<thead>
<tr>
<th>\texttt{\textbackslash{alpha}}</th>
<th>(\alpha)</th>
<th>\texttt{\textbackslash{pi}}</th>
<th>(\pi)</th>
<th>\texttt{\textbackslash{upsilon}}</th>
<th>(\upsilon)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{\textbackslash{angle}}</td>
<td>(\angle)</td>
<td>\texttt{\textbackslash{rho}}</td>
<td>(\rho)</td>
<td>\texttt{\textbackslash{phi}}</td>
<td>(\phi)</td>
</tr>
<tr>
<td>\texttt{\textbackslash{ast}}</td>
<td>(*)</td>
<td>\texttt{\textbackslash{sigma}}</td>
<td>(\sigma)</td>
<td>\texttt{\textbackslash{chi}}</td>
<td>(\chi)</td>
</tr>
<tr>
<td>\texttt{\textbackslash{beta}}</td>
<td>(\beta)</td>
<td>\texttt{\textbackslash{varsigma}}</td>
<td>(\varsigma)</td>
<td>\texttt{\textbackslash{psi}}</td>
<td>(\psi)</td>
</tr>
<tr>
<td>\texttt{\textbackslash{gamma}}</td>
<td>(\gamma)</td>
<td>\texttt{\textbackslash{tau}}</td>
<td>(\tau)</td>
<td>\texttt{\textbackslash{omega}}</td>
<td>(\omega)</td>
</tr>
<tr>
<td>\texttt{\textbackslash{delta}}</td>
<td>(\delta)</td>
<td>\texttt{\textbackslash{equiv}}</td>
<td>(\equiv)</td>
<td>\texttt{\textbackslash{Gamma}}</td>
<td>(\Gamma)</td>
</tr>
<tr>
<td>\texttt{\textbackslash{epsilon}}</td>
<td>(\epsilon)</td>
<td>\texttt{\textbackslash{Im}}</td>
<td>(\Im)</td>
<td>\texttt{\textbackslash{Delta}}</td>
<td>(\Delta)</td>
</tr>
<tr>
<td>\texttt{\textbackslash{zeta}}</td>
<td>(\zeta)</td>
<td>\texttt{\textbackslash{otimes}}</td>
<td>(\otimes)</td>
<td>\texttt{\textbackslash{Theta}}</td>
<td>(\Theta)</td>
</tr>
<tr>
<td>\texttt{\textbackslash{eta}}</td>
<td>(\eta)</td>
<td>\texttt{\textbackslash{cap}}</td>
<td>(\cap)</td>
<td>\texttt{\textbackslash{Lambda}}</td>
<td>(\Lambda)</td>
</tr>
<tr>
<td>\texttt{\textbackslash{theta}}</td>
<td>(\theta)</td>
<td>\texttt{\textbackslash{supset}}</td>
<td>(\supset)</td>
<td>\texttt{\textbackslash{Xi}}</td>
<td>(\Xi)</td>
</tr>
<tr>
<td>\texttt{\textbackslash{vartheta}}</td>
<td>(\vartheta)</td>
<td>\texttt{\textbackslash{int}}</td>
<td>(\int)</td>
<td>\texttt{\textbackslash{Pi}}</td>
<td>(\Pi)</td>
</tr>
<tr>
<td>\texttt{\textbackslash{iota}}</td>
<td>(i)</td>
<td>\texttt{\textbackslash{rfloor}}</td>
<td>(\rfloor)</td>
<td>\texttt{\textbackslash{Sigma}}</td>
<td>(\Sigma)</td>
</tr>
<tr>
<td>\texttt{\textbackslash{kappa}}</td>
<td>(\kappa)</td>
<td>\texttt{\textbackslash{lfloor}}</td>
<td>(\lfloor)</td>
<td>\texttt{\textbackslash{Upsilon}}</td>
<td>(\Upsilon)</td>
</tr>
<tr>
<td>\texttt{\textbackslash{lambda}}</td>
<td>(\lambda)</td>
<td>\texttt{\textbackslash{perp}}</td>
<td>(\perp)</td>
<td>\texttt{\textbackslash{Phi}}</td>
<td>(\Phi)</td>
</tr>
<tr>
<td>\texttt{\textbackslash{mu}}</td>
<td>(\mu)</td>
<td>\texttt{\textbackslash{wedge}}</td>
<td>(\wedge)</td>
<td>\texttt{\textbackslash{Psi}}</td>
<td>(\Psi)</td>
</tr>
<tr>
<td>\texttt{\textbackslash{nu}}</td>
<td>(\nu)</td>
<td>\texttt{\textbackslash{rceil}}</td>
<td>(\rceil)</td>
<td>\texttt{\textbackslash{Omega}}</td>
<td>(\Omega)</td>
</tr>
<tr>
<td>\texttt{\textbackslash{xi}}</td>
<td>(\xi)</td>
<td>\texttt{\textbackslash{vee}}</td>
<td>(\lor)</td>
<td>\texttt{\textbackslash{forall}}</td>
<td>(\forall)</td>
</tr>
</tbody>
</table>

Some Tex (Latex) characters.
Some more Tex (Latex) characters.
The following MatLab code:

```matlab
close all
axis([0 1 0 0.6]);
box on

text(0.0,0.52,['\fontsize{24} ' ...
   '\color[rgb]{0.9467 0.5203 0.0} ' ...
   ' CS2035 Formatting Text Examples']);
text(0.2,0.4,['f(t)=A_0 + ' ... 
   '\fontsize{20} \Sigma' ...
   '\fontsize{10} [A_n \cos ' ...
   '(n \omega_0 t)+B_n ' ...
   'sin(n \omega_0 t)]']);
text(0.2,0.3,['\fontsize{30} X=' ...
   '\fontname{courier} \fontsize{16}' ...
   '\bf x_{\alpha}+y^{2\pi}']);
text(0.2,0.2,['\fontsize{16} \nabla \times H' ...
   ' = J + \partial D/\partial t']);
text(0.2,0.1,'\color{red} \it E=M \cdot C^\rm 2');
print plot0.jpg -djjpeg
```
produces

\[ f(t) = A_0 + \sum (A_n \cos(n \omega_0 t) + B_n \sin(n \omega_0 t)) \]

\[ \chi = x_e + y^{2\pi} \]

\[ \nabla \times H = J + \frac{D}{\partial t} \]

\[ E = M \cdot C^2 \]
3D Graphs in MatLab

The plot3 function

- The plot function in the previous lecture extends into 3D via plot3. All the basic features of 2D plotting extend naturally into 3D. Pairs of arguments, for example coordinates \((x, y)\), now become triplets of arguments, \((x, y, z)\).

- The axis extends into 3D by adding the \(z\) axis limits:

\[
\text{axis}([\text{xmin} \ \text{xmax} \ \text{ymin} \ \text{ymax} \ \text{zmin} \ \text{zmax}])
\]

- There is now a zlabel for labelling the \(z\) axis.
• The `grid` command toggles a 3D grid being on or off (`grid off` is the default for `plot3`).

• The `box` command creates a 3D box around the plot (`box off` is the default for `plot3`).

• A character string can be printed in 3D using `text(x, y, z, 'string')`.

• Consider MatLab code to plot a 3D function as a single variable:

  ```matlab
  close all
  t=linspace(0,10*pi);
  plot3(sin(t),cos(t),t)
  xlabel('sin(t)');
  ylabel('cos(t)');
  ```
zlabel('t');

% Text at (0,0,0) uses those coordinates
% as the low left coordinates of its
% bounding box: adjust x and y to make
% the bullet centered exactly at (0,0,0)
text(-0.05,0.1,0,'
fontsize{36} \bullet');
text(0,0,2.5,'Origin');

grid on

box on

title('Helix 3D plot');

print plot3A.jpg -djjpeg

It produces the 3D plot:
3D plot of a Helix with the origin labelled’
Plotting 2D Scalar Arrays Using Meshgrid and Mesh

• Suppose you have a function \( z = f(x, y) \) and you wish to plot \( z \) as a function of \((x, y)\). An image is a perfect example of such a function.

• The plot of \( z \) as a function of \( x \) and \( y \) is a surface in 3D.

• The values of \( z \) mush to stored in a 2D array.

• We must store all the \( x \) and \( y \) values so that they correspond to the correct \( z \) values. We do this by creating matrices of the \( x \) and \( y \) values in the correct orientation using MatLab function `meshgrid`.

• Consider the MatLab code:

\[
x = -3:3; y = 1:5
\]
% No semicolon, so X and Y print

[X,Y]=meshgrid(x,y)

produces the output:

\[
\begin{array}{ccccccc}
X &=& -3 & -2 & -1 & 0 & 1 & 2 & 3 \\
    & & -3 & -2 & -1 & 0 & 1 & 2 & 3 \\
    & & -3 & -2 & -1 & 0 & 1 & 2 & 3 \\
    & & -3 & -2 & -1 & 0 & 1 & 2 & 3 \\
    & & -3 & -2 & -1 & 0 & 1 & 2 & 3 \\
Y &=& 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
    & & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\
    & & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\
\end{array}
\]
meshgrid duplicates $x$ for each of the 5 rows of $y$ and duplicated $y$ for the 7 columns of $x$.

Note that the $x$ axis varies from left to right and the $y$ axis varies from top to bottom.

Given $X$ and $Y$, if $z = f(x, y) = (x+y)^2$ we can plot it using the MatLab code:

```matlab
Z=(X+Y).^2
mesh(X,Y,Z)
```

% No semicolon so Z prints
xlabel('x axis');
ylabel('y axis');
zlabel('z axis');
print plot3D.jpg -djpeg

with $Z$ computed as:

\[
\begin{array}{cccccccc}
4 & 1 & 0 & 1 & 4 & 9 & 16 \\
1 & 0 & 1 & 4 & 9 & 16 & 25 \\
0 & 1 & 4 & 9 & 16 & 25 & 36 \\
1 & 4 & 9 & 16 & 25 & 36 & 49 \\
4 & 9 & 16 & 25 & 36 & 49 & 64 \\
\end{array}
\]

This produces the following plot:
3D surface plot of Z against X and Y.
The Peaks Surface

- The peaks MatLab function is a function of two variables, obtained by translating and scaling Gaussian distributions and is used commonly for demonstrating a 3D surface rendering in MatLab.

- The following MatLab code:

```matlab
% 30 by 30 array of surface values
[X,Y,Z]=peaks(30);
mesh(X,Y,Z)
xlabel('X axis');
ylabel('Y axis');
zlabel('Z axis');
```
title('Mesh plot of peaks');

print plot3E.jpg -djpeg

produces:
3D mesh plot of peaks.
Opaque and Transparent Spheres

- The areas between the mesh lines are opaque rather than transparent.

- MatLab provides a **tessellated** sphere function that draws a sphere with user given number of vertical and horizontal line (defining either triangles or quadrilaterals).

```matlab
[X,Y,Z]=sphere(10,8);
subplot(1,2,1)
mesh(X,Y,Z)
title('Opaque');
hidden on
axis square
```
axis off
subplot(1,2,2)
mesh(X,Y,Z)
title('Transparent');
hidden off
axis square
axis off
print plot3F.jpg -djpeg

The following plot is printed:
3D tessellated opaque and translated spheres
Surface Plots with Shading

- A surface plot looks like a mesh plot, except the space between the lines, called patches, are filled in.

- A surface plot is the dual of a mesh plot. In a surface plot lines are black and the patches have colour, whereas for meshes, the lines are the colour of the axes and the lines have colour.

- The default surface shading is faceted shading, where the patches are like stained glass outlined by black lines.

- The following MatLab code:

  \[ [X,Y,Z] = \text{peaks}(30); \]
surf(X,Y,Z)
xlabel('X axis');
ylabel('Y axis');
zlabel('Z axis');
title('Surface plot of peaks');
print plot3G.jpg -djpeg

produces:
3D surface plot of peaks with facet shading
• MatLab also provides for **flat** shading and **interpolated** shading.

• With flat shading, the black lines are removed and each patch has constant colour.

• With interpolated shading, the black lines are also removed but each the colour of each patch is interpolated over its area based on the colour of its vertices. The colours of the surface blend smoothly over all the patches. This may be visually pleasing but is computationally more expensive.

• Note that shading works with meshes as well, but since only lines are coloured the visual effect is minimal.

• Consider the following MatLab code:

  
  ```matlab
  [X, Y, Z] = peaks(30);
  ```
surf(X,Y,Z)
shading flat
xlabel('X axis');
ylabel('Y axis');
zlabel('Z axis');
title('Surface plot with flat shading');
print plot3H1.jpg -djpeg
% wait 5 seconds
pause(5)
shading interp
title('Surface plot with interpolated shading');
print plot3H2.jpg -djpeg
• Note that the surfaces if plotted with flat shading first, saved as plot3H1.jpg, then the shading is changed to interpolated and re-saved as plot3H2.jpg.

• These 2 plots are shown below:
Surfaces with Cutouts

- Sometimes it may be necessary to remove a part of a surface so that other underlying parts can be seen. We do this by setting the desired surface holes with NaN (Not a Number). MatLab plotting functions ignore NaN values, leaving a hole where they appear.

- MatLab code:

```matlab
[X,Y,Z]=peaks(30);
x=X(1,:); % take one column vector X as x axis
y=Y(:,1); % take one row vector of Y as y axis
% The hole is for y between 0.8 and 1.2
% and x between -0.6 and 0.5
```
% We find the coordinates satisfying these conditions
i=find(y>0.8 & y<1.2);

j=find(x>-0.6 & x<0.5);

Z(i, j)=nan;
surf(X, Y, Z)
xlabel('X axis');
ylabel('Y axis');
zlabel('Z axis');
title('Surface plot with hole');
print plot3I.jpg -djpeg

produces the plot:
3D surface plot with hole
Surface Plots with Contour

- `surfC` is a surface plot with underlying contour plot.

- The MatLab code:

```matlab
[X,Y,Z]=peaks(30);
% surf plot with contour
surf(X,Y,Z)
xlabel('X axis');
ylabel('Y axis');
zlabel('Z axis');
title('Surface plot with contours');
print plot3J.jpg -djpeg
```
produces the following plot:

3D surface plot with hole
Surface Plots with Lighting

- `surfl` produces a surface plot with lighting. `surfl` modifies the the colour of the surface to give the appearance. We can use a MatLab `colormap` to change the different set of colours to a figure. A more detail of `colormap` is delayed to later.

- The following MatLab code:

  ```matlab
  [X,Y,Z]=peaks(30) surfl(X,Y,Z);
  % surfl plots look best with interp shading
  shading interp
  % They looks better with shades of a single color
  colormap pink
  ```
xlabel('X axis');
ylabel('Y axis');
zlabel('Z axis');
title('Surface plotting with lighing');
print plot3K.jpg -djpeg

produces the 3D plot:
3D surface plot with lighting
Surface Plots with Surface Normals

- The `surfnnorm(X, Y, Z)` function compute surface normals for the surface at the data points defined at the \(X, Y\) and \(Z\) values. Both surface and surface normals are plotted.

- Note that the surface normals are unnormalized, i.e. \(\|\vec{n}\|_2 \neq 1\) for some normal \(\vec{n}\).

- The following MatLab code:

```matlab
[X,Y,Z]=peaks(30);
surfnnorm(X,Y,Z)
xlabel('X axis');
```
ylabel('Y axis');

zlabel('Z axis');

title('Surface plots with normals')

print plot3L.jpg -djpeg

produces the 3D plot:
3D surface plot with surface normals
Changing Viewports

- The default viewpoint of 3D plots is looking down at the $z = 0$ plane at an angle of $30^\circ$ and looking up at the $y = 0$ plane at angle of $-37.5^\circ$.

- The angle of orientation with respect to the $z = 0$ plane is called the *elevation* angle and the angle with respect to the $y = 0$ plane is called the *azimuth* angle. The concepts of the azimuth and elevation angles are described in the following figure.

- For 2-D plots, the default is azimuth $= 0^\circ$ and elevation $= 90^\circ$. For 3-D plots, the default is azimuth $= -37.5^\circ$ and elevation $= 30^\circ$. 
The Elevation and Azimuth Angles

- The follow MatLab code demonstrated the effect of changing these 2 angles:

```matlab
% Generate general peaks plot
[X,Y,Z]=peaks(30);
subplot(2,2,1);
surf(X,Y,Z);

% Use default view angles
view(-37.5,30);
xlabel('X axis');
ylabel('Y axis');
```
zlabel('Z axis');

title('Default view: Az=-37.5 El=30');

subplot(2,2,2);
surf(X,Y,Z);

% Add 90 to the azimuth angle
view(-37.5+90,30);
xlabel('X axis');
ylabel('Y axis');
zlabel('Z axis');
title('Az=rotated by 90+-37.5=52.55 El=30');
subplot(2,2,3);
surf(X,Y,Z);

% Add 30 to the elevation angle
view(-37.5,60);
xlabel('X axis');
ylabel('Y axis');
zlabel('Z axis');
title('Ax=-37.5 El=60');

subplot(2,2,4);
surf(X,Y,Z);

% Set the azimuth angle to 0 and
% the elevation angle to 90: this
% is a top down view
view(0,90);
xlabel('X axis');
ylabel('Y axis');
zlabel('Z axis');
title('Top Down View: Ax=0 El=90');

% plot the 4 graphs as a jpg file
print plot3M.jpg -djpeg

produces the following plot:
The effect of various elevation and azimuth angles
Contour Plots

- Contour plots show lines of constant elevation of height. A topographical map is a contour map.

- Consider the following MatLab code:

```matlab
% Use standard peaks surface
[X,Y,Z]=peaks(30);
subplot(1,2,1);
% generate 20 2D contour lines
contour(X,Y,Z,20);
axis square;
xlabel('X axis');
```
ylabel('Y axis');
title('2D contour plot');
subplot(1,2,2);
% The same contour in 3D
contour3(X,Y,Z,20);
xlabel('X axis');
ylabel('Y axis');
zlabel('Z axis');
title('3D contour Plot');
print plot3N.jpg -djpeg

The produces the plot:
2D and 3D contour plots of the peaks data
• The function `pcolour` maps height to a set of colours and present the same information as the contour plot at the same scale.

• Function `contourf` combines the a pseudo color plot with a 2D contour plot to produce a filled contour.

• The following MatLab code:

```matlab
subplot(1,2,1)
[X,Y,Z]=peaks(30);
pcolor(X,Y,Z);
shading interp % remove the grid lines on axis square
title('Pseudo Color Plot');
```
subplot(1,2,2);
% filled contour plot with 12 contours
contourf(X,Y,Z,12);
axis square
xlabel('X axis');
ylabel('Y axis');
title('Filled Contour Plot');
print plot30.jpg -djpeg

produces the plot:
2D Pseudocolour and filler contour plots
Labelled Contour Lines

- Contour lines can be labelled using the `clabel` function, which requires a matrix of lines and text strings that are return by `contour`, `contourf` and `contour3`.

- The following MatLab code:

```matlab
[X,Y,Z]=peaks(30);
C=contour(X,Y,Z,12);
clabel(C);
title('Contour plot with labels');
print plot3P.jpg -djpeg
```
produces the plot:

![2D Contour Plot With Labels](image-url)
Ribbon Plots

- MatLab provides a way to plot “ribbon” flows of functions. Each ribbon plot the contours of some dimension i, say $T$ as separate ribbons.

- The function $\text{ribbon}(Y,Z)$ plots the columns of $Y$ against $Z$.

- The width of a ribbon can be specified as a $3^{rd}$ argument of $\text{ribbon}$, where the default width being 0.75.

- Consider the following MatLab code: which produces the plot:

```matlab
% X and Y default to the size
% of the array produced for Z.
[X,Y,Z]=peaks(30);
```
subplot(1,2,1)
ribbon(Y,Z,0.75);
xlabel('X axis');
ylabel('Y axis');
zlabel('Z axis');
title('Ribbon plot of Y versus Z peaks (0.75)');
subplot(1,2,2)
ribbon(Y,Z,1.5);
xlabel('X axis');
ylabel('Y axis');
zlabel('Z axis');
title('Ribbon plot of Y versus Z peaks (1.5)');
print plot3Q.jpg -djpeg

which produces the 2 plots:
3D ribbon plots with width 0.75 and 1.5 respectively
2D Quiver Fields

- The function `quiver(x, y, dz, dy)` draws a vector field \((dx, dy)\) at points \((x, y)\).

- Consider the following MatLab code:

```matlab
[X,Y,Z]=peaks(20);
% Compute the gradient if Z with respect % to X and Y for values 0.5,0.5
[dx,dy]=gradient(Z,0.5,0.5);
quiver(X,Y,dx,dy);
title('2D quiver plot');
print plot3R.jpg -djpeg
```
The code produces the plot:
3D Quiver Fields

- 3D quiver plots \texttt{quiver3(x,y,z,dx,dy,dz)} display the vectors (dx, dy, dz) at the 3D points (x, y, z).

- The MatLab code:

\begin{verbatim}
[X,Y,Z]=peaks(30);
[Ny,Ny,Nz]=surfnorm(X,Y,Z);
% We won't display the 3D surface
quiver3(X,Y,Z,Nx,Ny,Nz);
title('3D quiver plot');
print plot3S.jpg -djpeg
\end{verbatim}
produces the 3D vector field plot:
The 3D version of stem plots extends naturally from 2D. The command
\[ \text{stem3}(X,Y,Z,C,'filled') \]
plots data points in \((X,Y,Z)\) with lines extending to the \(x - y\) plane. The optional \(C\) argument specifies the marker type of the colour and the optional ’filled’ argument causes the marker to be filled in, \(\text{stem3}(Z)\) points the values in \(Z\) and automatically generates the \(X\) and \(Y\) values.
• Consider the MatLab code:

\[
Z = \text{rand}(5.5) % \text{rand}(5) \text{ gives the sane values}
\]

\[
\text{stem3}(Z, 'ro', 'filled');
\]

\[
\text{grid on}
\]

\[
\text{title('3D stem plot of random data');}
\]

\[
\text{print plot3T.jpg -djpeg}
\]

This code produces the following plot:
3D quiver field
Volume Visualization - Slice Plots

- In addition to mesh, surface and contours plots, MatLab offers volume visualization capabilities.

- As an example of slice images, consider the following MatLab code:

```matlab
x=linspace(-3,3,13);
y=1:20;
z=-5:5;
[X,Y,Z]=meshgrid(x,y,z);
V=sqrt(X.^2+cos(Y).^2+Z.^2);
slice(X,Y,Z,V,[0 3],[5 15],[-3 5]);
xlabel('X axis');
```
ylabel('Y axis');

zlabel('Z axis');

title('Slice plot through a volume');

print plot3V.jpg -djpeg

The following figure is plotted:
3D slice plot through a volume. Note the slices at X=0 and X=3, the slices at Y=5 and Y=15 and the slices at Z=-3 and Z=5.
• Slices do not have to be planes. They can be surfaces. Consider:

```matlab
x=linspace(-3,3,13);
y=1:20;
z=-5:5;

[xs,ys]=meshgrid(x,y);

% Generate a surface
zs=sin(-xs+ys/2);
V=sqrt(X.^2+cos(Y).^2+Z.^2);
slice(X,Y,Z,V,xs,ys,zs);
xlabel('X axis');
ylabel('Y axis');
zlabel('Z axis');
```
title('Slice plot using a surface');

print plot3W.jpg -djpeg

This code defines a surface using \( xs, ys \) and \( zs \). The following figure is plotted:
3D slice plot using a surface
Slice Plot for MRI Cardiac Data

- Consider a 4D volume of MRI cardiac gated data. Suppose we wish to visualize volume 9 (of a 20 volume sequence of 1 heart beat). We can use `slice` to view orthogonal slices of the data. Programs to move these slices while viewing the data can be written in MatLab (and probably have),

- Consider the following MatLab code:

```matlab
% The variable name used by save was vol9
% A volume is 75 slices of 256 by 256
% unsigned short data: grayvalues in
% the range 0-4095, i.e 12 bit data,
% stored as 16 bit (2 byte) data
load MRI_DATA_9.mat
fprintf('9th volume of MRI data loaded\n')

sample_xy=8;
sample_z=4;
```
%% Size of volume: 75*256*256 shorts
[end_z,end_x,end_y]=size(vol9);

start_z=1;
start_x=1;
start_y=1;
fprintf('start_z=%d start_x=%d start_y=%d\n',...
    start_z,start_x,start_y);
fprintf('end_z=%d end_x=%d end_y=%d\n',...
    end_z,end_x,end_y);

vol9=cast(squeeze(vol9),’double’);
% subsample the data to allow MatLab to display
% Data now 19 by 32 by 32
vol9=vol9(start_z:sample_z:end_z,....
    start_x:sample_xy:end_x,....
    start_y:sample_xy:end_y);

% must switch the first 2 arguments
% of meshgrid in 3D
[Z,X,Y]=meshgrid(1:size(vol9,2),...
    1:size(vol9,1),...
    1:size(vol9,3));

% The data has been subsampled so
% subsample the z, x and y slice
% numbers to be displayed as well
% Swap slice numbers as in meshgrid
% Original slice numbers 36,128,128
% become 9,16,16
slice_number_z=36/sample_z;
slice_number_x=128/sample_xy;
slice_number_y=128/sample_xy;

% Generate xy, xz and yz images
% for these slice numbers
% 1. Color the images with the summer colormap
% using grs2rgb available at
% http://www.mathworks.com/matlabcentral/
% fileexchange/13312-grayscale-to-rgb-converter/
% content/grs2rgb.m
% 2. Enlargen the images by a factor
% of 5 using imresize
imz=squeeze(vol9(slice_number_z,:,:));
imx=squeeze(vol9(:,slice_number_x,:));
imy=squeeze(vol9(:,:,slice_number_y));
slice_z=imresize(grs2rgb(imz,...
    colormap(summer(4096))),5);
slice_x=imresize(grs2rgb(imx,...
    colormap(summer(4096))),5);
slice_y=imresize(grs2rgb(imy,...
    colormap(summer(4096))),5);
imwrite(slice_z,'slice_z.jpg');
imwrite(slice_x,'slice_x.jpg');
imwrite(slice_y,'slice_y.jpg');
imshow(slice_z,[]);
title(['vol9(' num2str(slice_number_z) ',,:,:)]);

figure
imshow(slice_x,[]);
title(['vol9(:,,' num2str(slice_number_x) ',:)']);

figure
imshow(slice_y,[]);
title(['vol9(:,:,:,' num2str(slice_number_x) ')']);

% Generate 3D slice image
% Exchange Z and X or slice_number_x
% and slice_number_z to be consistent
figure
slice(Z,X,Y,vol9,slice_number_x,...
     slice_number_z,slice_number_y);
% could use colormap(gray(4096));
colormap(summer(4096));
axis tight
shading interp
view(-37.5,30.0);
% Swap axis labels as well
zlabel('Y axis');
xlabel('X axis');
ylabel('Z axis');
title({['Slice plots of 3 orthogonal planes'...
     ' in the 9th volume of a subsampled']...}
MRI cardiac dataset];
['Original data size:' num2str(end_z)...
' by ' num2str(end_x)...
' by ' num2str(end_y) ' ...
'Subsampled data size:' num2str(size(vol9,1))...
' by ' num2str(size(vol9,2))...
' by ' num2str(size(vol9,3))];
['Subsampling in z by ' num2str(sample_z) ', '...
'subsampling in x and y by ' num2str(sample_xy)];
['Displaying:' ...
num2str(slice_number_z) 'th z slice ' ...
num2str(slice_number_x) 'th x slice ' ...
num2str(slice_number_x) 'th y slice '];

print plot3U.jpg -djpeg

- Note that MatLab often cannot nicely display volume data because there is too much data. In the above code, we handled this problem by subsampling the data (by 4 in z and by 8 in x and y). Matlab does have a function reduce volume which reduces the data (in some unknown way).
• The $xy$, $xz$ and $yz$ subsampled images are shown below:

The $xy$ ($z = 36$), $xz$ ($y = 128$) and $yz$ ($x = 128$) subsampled MRI images
Finally, the following 3D slice plot is produced:
Easy Plotting

- There may be times when you want to let MatLab specify the data points for a 3D plot.

- MatLab has functions `ezcontour3`, `ezcontourf`, `ezmesh`, `ezmeshc`, `ezplot3`, `ezsurf` and `ezsurfc` that plot the data as their non-`ez` counterparts but usually take string expressions as arguments.

- The following MatLab code:

```matlab
fstr=['3*(1-x).^2.*exp(-(x.^2)-(y+1).^2)' ... 
      '-10*(x/5-x.^3-y.^5).*exp(-x.^2-y.^2)' ... 
      '-1/3*exp(-(x+1).^2-y.^2)'];
```
subplot(2,2,1)
ezmesh(fstr);
title('Mesh of peaks');

subplot(2,2,2)
ezsurf(fstr);
title('Surf of peaks');

subplot(2,2,3)
ezcontour(fstr);
title('Contour of peaks');
subplot(2,2,4)
ezcontourf(fstr);
title('Contoutf of peaks');

print plot3X.jpg -djpeg

produces the plot:
Four easy plots