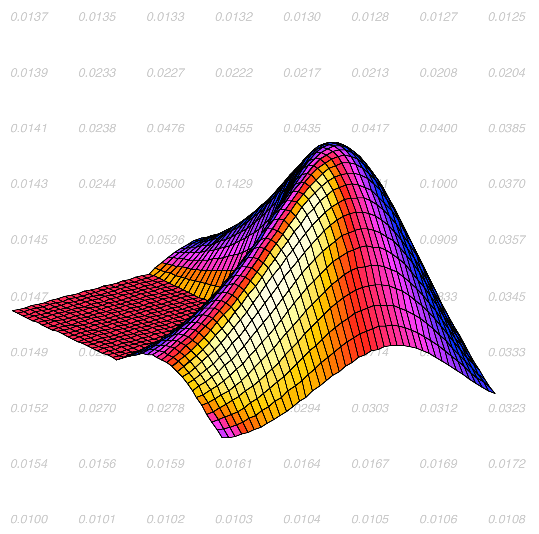
Linear Algebra with Application to
Engineering Computations

CME200
M. Gerritsen
Autumn 2006
Handout 3

A brief introduction to MATLAB



September 2006

MATLAB, short for MATrix LABoratory is a programming package specifically designed for quick and easy scientific calculations and I/O. It has literally hundreds of built-in functions for a wide variety of computations and many toolboxes designed for specific research disciplines, including statistics, optimization, solution of partial differential equations, data analysis.

For CME200, you need a solid knowledge of basic MATLAB commands and several more advanced features including two- and three-dimensional graphics, solution of algebraic equations, solution of ordinary differential equations, calculations with matrices and solutions of linear systems of equations. Most of what you need is discussed here, but most importantly, after this tutorial you should be able to find your way around the MATLAB help function and browser functions to find any additional features you may need or want to use.

General directions
This is a hands-on tutorial. MATLAB commands for you to type are printed in **bold letters**. Bold letters are also used to make MATLAB expressions that are in lower case more visible when found in a sentence.

Where to find MATLAB?
For those who own a PC, the cheap student edition of MATLAB is available at the bookstore. *We recommend you buy a copy of this version because you will likely use it throughout your Stanford career and beyond.*
If you do not have your own PC, you will find that MATLAB is available on a wide variety of platforms around campus.

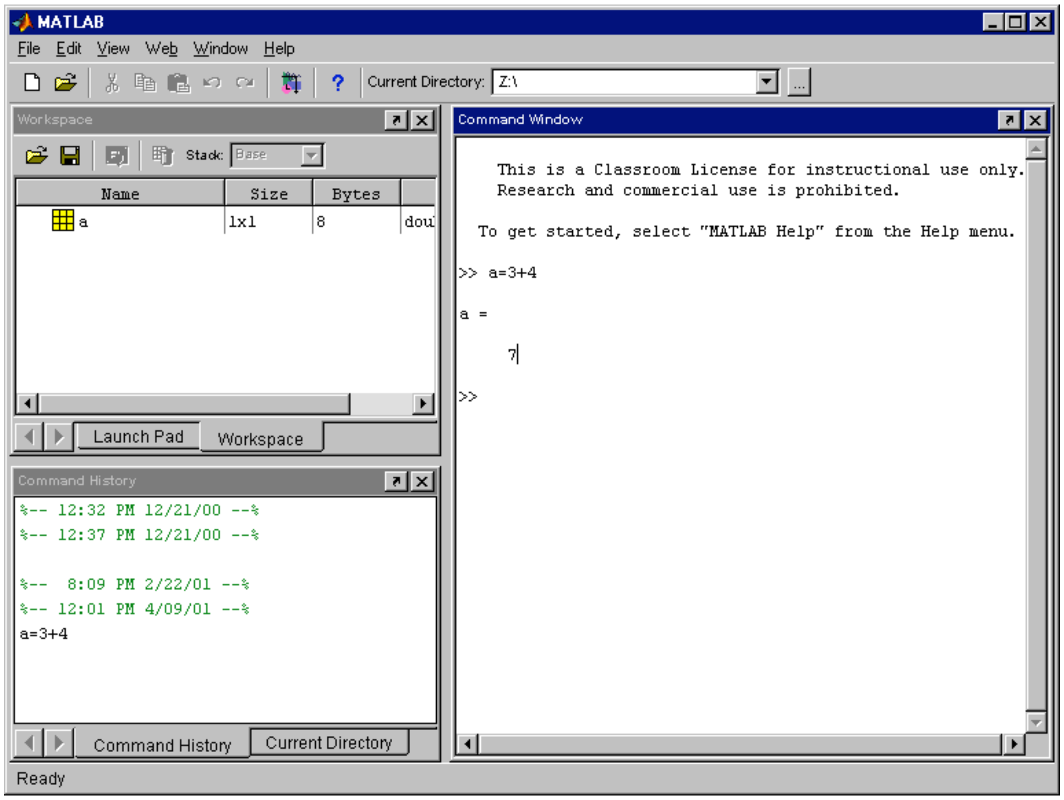
This tutorial assumes you can open the MATLAB GUI. If you are running on a UNIX machine, you can also run MATLAB in any xterm window, but you will miss the advanced interface options that makes the new versions of MATLAB such a pleasure to deal with.

This MATLAB tutorial is based on version 6x (from 2001). The latest version of MATLAB has a slightly different interface, and additional features. However, the basic commands and workings of MATLAB discussed in this tutorial are still the same.

• Section 1 – The Basics

1) Start MATLAB by double clicking on the MATLAB icon in the applications folder, or wherever it is. Note that on some campus machines MATLAB is listed as an Optional software under the applications folder. If that is the case, you must download the complete MATLAB folder onto the hard drive from the server. Read the Optional software instructions available there. This download may take up to 30 minutes.

The MATLAB window should come up on your screen. It looks like:



Name	Size	Bytes	
a	1x1	8	dou

```
   This is a Classroom License for instructional use only.
   Research and commercial use is prohibited.

 To get started, select "MATLAB Help" from the Help menu.

>> a=3+4

a =

     7

>>
```

```
%-- 12:32 PM 12/21/00 --%
%-- 12:37 PM 12/21/00 --%

%--  8:09 PM 2/22/01 --%
%-- 12:01 PM 4/09/01 --%
a=3+4
```

This is the window in which you interact with MATLAB. The main window on the right is called the *Command Window*. You can see the command prompt in this window, which looks like >>. If this prompt is visible MATLAB is ready for you to enter a command. In the figure, you can see that we typed in the command **a=3+4**.
In the top left corner you can view the *Launch Pad window* and the *Workspace window*. Swap from one to the other by clicking on the appropriate tag. The *Workspace window* will show you all variables that you are using in your current MATLAB session. In this example, the workspace contains the variable 'a'. When you first start up MATLAB, the workspace is empty. More about workspaces later.
In the bottom left corner you can see the *Command History window*, which simply gives a chronological list of all MATLAB commands that you used, and the *Current Directory window* which shows you the contents and location of the directory you are currently working in.

You can change the layout of the MATLAB window:

2) To change the layout of the MATLAB window, select *View*, then *Desktop Layout*. Six different layout styles can be chosen.

Note that MATLAB is case-sensitive, so

3) make sure that the 'Caps Lock' is switched off!

During the MATLAB sessions you will create files to store programs or workspaces.

4) Create an appropriate folder to store this lab's files.

5) Go to the Current Directory window. Find the *Browse for folder* button on the menu (the one with the 3 dots …). Select the folder you just created so that MATLAB will automatically save files in this folder. Note that the current directory (or folder: folder and directory mean the same thing) is also displayed in the top right corner next to the main menu.

If you like to see a demo of MATLAB you can type **demo** after the prompt.

6) Now, let's try a simple command. Next to the prompt in the Command Window type **a=3+4** and then press 'enter' to activate this command. On the screen you should see

```
a=

          7
```

Notice how MATLAB carries out this command immediately, and gives you the prompt >> for your next command. Here, a variable called 'a' is created by MATLAB and assigned the value of 7. MATLAB stores the variable a in its *workspace* until you exit MATLAB or tell MATLAB to delete the variable.

7) Go to the Workspace window and check that the variable a is in your workspace.

You will see in this window that a is stored in 8 bytes, that it is a double and that it has size 1x1. This size seems a bit odd and we will explain it later.

8) In the Workspace window, double click on the name 'a'. A small window will pop up displaying a's value. This is a handy feature.

We can change the value of the variable:

9) Try **a = 9**, followed by 'enter' to change the value of a to 9. Then type **b=sqrt(a)** to find the square root of a. Don't forget to press 'enter' to enter the command (we will not always remind you to hit 'enter' but it is necessary to tell MATLAB to carry out the command). Now you should see

```
b=

          3
```

10) Go back to the Workspace window and check that b is indeed in your workspace. Also check by double-clicking on a that a's value has changed.

• Section 2 - The MATLAB on-line help facility

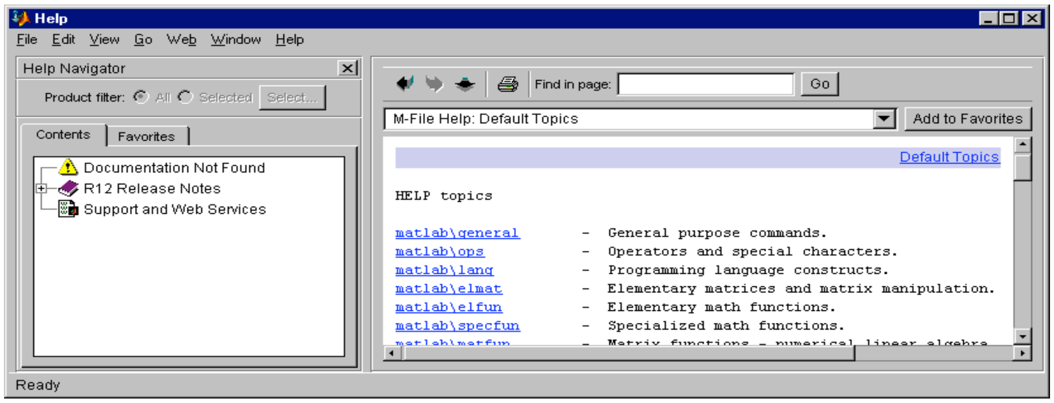
MATLAB has handy on-line help facilities. There are several ways to get help. You can go to Help on the menu (or the ? on the menu) and select any of the available help facilities listed there. There are simpler help commands as well that work in all versions of MATLAB.

1) Type the command **help** in the Command Window to find a long list of all different categories for which there are MATLAB commands. Each of the listed categories contains more detailed information about available MATLAB functions. For example, 'ops' (listed in the top under 'matlab:ops') contains information about MATLAB operators such as addition and subtraction.

2) Type **help ops** to find a list of MATLAB operators. You can see from this list, for example, that more information about addition can be found in **help arith** .

Alternatively you can launch the help window:

3) Type **helpwin** to launch the help window shown below.



```
HELP topics

matlab\general      -  General purpose commands.
matlab\ops          -  Operators and special characters.
matlab\lang         -  Programming language constructs.
matlab\elmat        -  Elementary matrices and matrix manipulation.
matlab\elfun        -  Elementary math functions.
matlab\specfun      -  Specialized math functions.
matlab\matfun       -  Matrix functions - numerical linear algebra.
```

In this window, you can get help on operators, for example, by double clicking on 'ops'.

Another very useful MATLAB command is the **lookfor** command. You can use this to search for commands related to a keyword.

4) Find out which MATLAB commands are available for plotting using the command **lookfor plot** in the command window. It may take a bit of time to get all commands on the screen. You can stop the search at any time by typing Ctrl-c (the control and the c key simultaneously).

Important
In the MATLAB **help** descriptions, the MATLAB commands and functions are often given in capital letters. But: <u>always type commands in lower case</u>. MATLAB is case sensitive and will generally <u>not</u> recognize commands typed in capital letters! Note that because of this case sensitivity the variables 'A' and 'a', for example, are different. Here, we use capital letters like 'A', 'B' for matrices and lower case letters for scalars or vectors.

• Section 3 - Data representation in MATLAB

As we discussed in the lectures, MATLAB stands for 'MATrix LABoratory'. This title is appropriate because the structure for the storage of <u>all</u> data in MATLAB is a matrix. The MATLAB matrix-variables may have any number of rows and columns. Scalars like the variables a and b that you worked with above are also stored as matrix variables with 1 row and 1 column. This is the reason that the variable a in your workspace is shown as a 1x1 matrix!
So beware, a matrix-variable can be <u>any</u> variable in MATLAB, that is, it could be a scalar, a vector or a matrix of any size. If we refer to scalars, vectors or matrices specifically we mean just that: scalars, vectors or matrices.

Entering variables

An mxn ('*m* by *n*') MATLAB matrix-variable (or simply variable) has *m* rows and *n* columns. For example, the variable **A** = $\begin{bmatrix} \mathbf{1} & \mathbf{2} & \mathbf{3} \\ \mathbf{4} & \mathbf{5} & \mathbf{6} \end{bmatrix}$ is a 2x3 *matrix*. The numbers 1 through 6 are called the *elements* of the matrix. The element on row 1 and column 2 has the value 2. The element on row 2 and column 3 has the value 6.

1) Create the matrix A by typing **A = [1 2 3;4 5 6]** followed by the enter. Make sure that you separate the elements 1,2 and 3, and 4, 5 and 6 with a space. If you don't MATLAB will think you created the vector A with first element 123 (one hundred and twenty-three) and second element 456. Also, use *square* brackets instead of parentheses.

Instead of a space, you can also use a comma to separate the elements. The semi-colon (;) in this context is used to separate the elements of one row from another, but you can also use a line break to separate rows as shown below.

2) Remove the matrix A by typing **clear A**. Then recreate A using
>> **A = [1,2,3** *(now hit enter for the line break)*
4,5,6]

Error messages

If your command is invalid MATLAB gives you explanatory error messages (luckily!). Read them carefully. Normally MATLAB points to the exact position of where things went wrong.

3) Try to change A to $\begin{bmatrix} \mathbf{11} & \mathbf{22} \\ \mathbf{33} & \mathbf{44} \end{bmatrix}$ using the incorrect command **A = [11 22 33;44]**

The command has failed. MATLAB tells you that you do not have the same number of columns in each row, i.e. the same number of elements in each row. You put the semi-colon in the wrong place.

Row and column vectors
A row vector has one row and any number of columns. Similarly a column vector has one column and any number of rows.

4) Type **v=[1, 2, 3]** . Now use the **size** function to check that v has 1 row and 3 columns by typing **n= size(v)** .

size is an example of a MATLAB function. The round brackets enclose the variable to be operated on. The first number returned by the **size** function gives the number of rows in the variable and the second number gives the number of columns.

5) Enter the column vector **c** = $\begin{bmatrix} \mathbf{1} \\ \mathbf{2} \\ \mathbf{3} \end{bmatrix}$ using **c = [1; 2; 3]** and check that is has 3 rows and 1 column (note the use of semicolons to separate the rows).

Changing matrices
Just as with scalars, you can change a matrix simply by defining it to be something else. We already unsuccessfully tried this when we tried to change A. Lets get it right this time.

6) Type **A=[11 22;33 44]** to make A become the 2x2 matrix given earlier.

The 2x3 matrix that A used to be has now been lost. The variable A still exists of course in MATLABs workspace but now has a *different size and different values for its elements.*

• Section 4 - Scrolling

Suppose you want to repeat or edit an earlier command. If you dislike typing it all back in, then there is good news for you: MATLAB lets you search through your previous commands using the up-arrow and down-arrow keys. This is a very convenient facility, which can save a considerable amount of time.

1) Suppose you regret the last command, and that you want A still to be the 2x3 matrix you originally defined. Press the up-arrow key until the earlier command
A=[1 2 3;4 5 6] appears in the Command Window (the down-arrow key can be used if you go back too far). Now press enter. The matrix A is back to what it was originally.

You can speed up the scroll if you remember the first letters of the command you are interested in. For example, if you quickly want to jump back to the command **v = [1, 2, 3]** just type **v** = and then press the up-arrow-key.

MATLAB will display the most recent command starting with 'v ='. Alternatively you can use 'Copy' and 'Paste' under 'Edit'.

In MATLAB 6 you can also use the Command History window to jump to a previous command:

2) Go to the Command History window. Jump back to the command **v=[1,2,3]** by double-clicking on this line in the Command History window. The command will be carried out immediately.

• Section 5 - Basic arithmetic operations in MATLAB

MATLAB allows you to do operations on variables as single entities like in matrix-matrix multiplication and also operations on the individual elements of the matrix.

1) Verify on paper that the product of the matrices $\mathbf{A} = \begin{bmatrix} \mathbf{1} & \mathbf{2} \\ \mathbf{0} & \mathbf{1} \end{bmatrix}$ and $\mathbf{B} = \begin{bmatrix} \mathbf{4} & \mathbf{2} \\ \mathbf{3} & \mathbf{5} \end{bmatrix}$ is equal to $\mathbf{AB} = \begin{bmatrix} \mathbf{10} & \mathbf{12} \\ \mathbf{3} & \mathbf{5} \end{bmatrix}$.

Remember that the multiplication AB is possible because the number of columns in A is equal to the number of rows in B. Matrix multiplication is generally not commutative (commutative means AB = BA), so it is no surprise to find that $\mathbf{BA} = \begin{bmatrix} \mathbf{4} & \mathbf{10} \\ \mathbf{3} & \mathbf{11} \end{bmatrix}$ is a different matrix.

2) Create the matrices A and B in MATLAB.

3) Type **A*B** to perform the first multiplication. Because you did not give the result of this expression a name MATLAB calls it 'ans' (which is short for answer). Note that multiplication of matrices in MATLAB as in A*B is the *matrix multiplication (product)* operation.

4) Type **B*A** to perform the second matrix multiplication. The matrix 'ans' has now changed to be the result of this expression. In MATLAB 'ans' is the result of the last unnamed MATLAB command, so you should not use 'ans' as the name of one of your variables!

5) Type **A+B** to add the two matrices. Type **3*A** to multiply A by 3.

So you see that MATLAB can be used as a simple 'pocket calculator' for matrices. You can quickly and easily multiply, add or subtract them. It is a very handy tool to check matrix computations that you must carry out for other courses.

• Section 6 - Keeping in touch with your variables

By now you might have lost track of what variables you created. Yet they are still there as we mentioned before: Unless you explicitly delete them, change them or quit MATLAB, they do not go away, but are kept in MATLABs workspace. You can see a list of all your variables in the Workspace window. Alternatively you can use commands in the Command window to do the same:

1) Type **who** to see all your matrices.

2) Type **whos** to see a more complete description, including sizes, of all your stored data. From this description check that v is a row vector and c is a column vector

Deleting matrices

3) Remove the matrix B and the vector v by typing **clear B v** . Alternatively, you can remove B and v by selecting them in the Workspace window and then clicking the 'Delete Selection' button (with the wee trashcan).

Remember that you don't have to clear matrices to use the same names again. You can just set up a new matrix with the same name.

4) Change the matrix A by typing **A=2**.

5) Check that A is now a *scalar* - the previous A matrix is lost. Again, we changed the size and the value of the elements of the matrix-variable A. The size is now 1x1.

6) More drastically, remove *all* your variables by just typing **clear** without any names following.

7) Check that there are is nothing left in your workspace. Careful with this command!

• Section 7 - Generating variables made easy

Ready-made matrices and vectors

MATLAB provides functions to create several basic matrices automatically without having to type or read in each of the elements. The most important functions are

zeros	**zeros(m,n)** creates an mxn matrix whose elements are equal to zero
ones	**ones(m,n)** creates an mxn matrix whose elements are equal to one.
eye	**eye(m,n)** creates an mxn identity matrix
rand	**rand(m,n)** creates an mxn matrix whose elements are all random number between 0 and 1

1) Create a 5x5 matrix with random numbers. Now add to it a diagonal matrix with the diagonal elements equal to 2. You can create such a diagonal matrix using the command **2*eye(5,5)** .

The colon operator as a way of generating row vectors

The colon (:) is used a lot in MATLAB. One of its functions is to generate sequences of equally spaced numbers in a row vector.

2) Type **h=10:2:20**. You should see that you have created a row vector h with elements starting at 10 and increasing in steps of 2 up to 20. We could also have created this vector with **h=[10 12 14 16 18 20]**. Note that we did NOT need brackets in **h=10:2:20**

If the step size is negative the sequence decreases. So **i=9:-1:5** sets i to be [9 8 7 6 5].

3) Type **r=50:55**. You should have the vector [50 51 52 53 54 55]. In other words, if the step size is omitted it is assumed to be +1.

Suppression of output

Up till now you have always seen the results of each command on the screen. This is often not required or even desirable; the output might clutter up the screen, or be long and take a long time to print. MATLAB suppresses the output of a command if you finish the command with a semi-colon **;**. This is therefore the second use of the semicolon that you have come across. What was the first one again?

4) Set up the row vector with the first 500 whole numbers by just typing **r=1:500** and see how long it takes to echo on the screen.

5) Now let's create a second such vector by typing **rr=1:500;** noting that we have added a semicolon.

This last command executes quickly: the printing on the screen was taking up all the time.

Building larger matrices from smaller ones

6) Delete all your variables using **clear**.

7) Create the matrices $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and $B = \begin{bmatrix} -8 & -7 & -6 \\ -5 & -4 & -3 \end{bmatrix}$

8) Now type **C=[A B]** or **C=[A,B]** and you should see that MATLAB put A and B side-by-side and called the resulting matrix C.

9) Type **D=[A B;B A]** and you should see that MATLAB placed the matrix [B A] below the matrix [A B] and called it D.

10) Now type **D=[A;B]**.

This gives an error message, which tells you that this operation cannot be done. The number of columns in A and B are not equal. The matrix

$$D = \begin{bmatrix} 1 & 2 & ? \\ 3 & 4 & ? \\ -8 & -7 & -6 \\ -5 & -4 & -3 \end{bmatrix}$$

is a nonsense matrix. No new matrix D has been created and D is still equal to [A B;B A].

Changing values of individual elements

Individual elements in a variable can be identified using index numbers. For example the element 3 in the matrix A above can be referred to as A(2,1) because it is the first element of the second row of A. Note that the row and column indices are separated by a comma.

11) Change this element to have the value of 1 by typing **A(2,1)=1**

Note how MATLAB prints the whole matrix (unless you include a ; at the end), and that A(2,1) has changed.

Similarly sections of matrices (sub-matrices) can be identified using the colon operator.

12) Type **A2 =A(1:2,1)** to extract the first column of A and store it in A2.

We could also have achieved this by typing **A2 = A(:,1)** . The colon operator identifies *all* of the first column (taking the first element in ALL of the rows). Similarly **A(2,:)** identifies the whole of the second row of A. Note that MATLAB will object if you use negative (or zero!) indices for matrices.

13) Type **A(-1,1)=23** for example.

However if you use an index which is above the maximum dimension the matrix is enlarged and all the intervening elements are set to zero.

14) Type **A(4,4)=2**.

See how the extra elements of the matrix have been created, but only A(4,4) has been set to 2, the other extra elements are set to zero by default.

We can use indexing to shift elements of a variable.

15) Swap the first and second columns of B by typing **B = [B(:,2) , B(:,1), B(:,3)]** . This overwrites the old B. So read this as : B becomes the matrix with first column equal to the second column of the old B, second column equal to the first column of the old B and third column equal to the third column of the old B. Why couldn't you do it like this: **B(:,1) = B(:,2)** followed by **B(:,2) = B(:,1)** ?

Transposing matrices and vectors

The operator ' (single quote) finds the transpose of a variable (note that the transpose of a scalar is just the scalar itself). Transposing a matrix or vector swaps rows and columns while retaining their order. For example if A is the matrix $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$, then **B = A'** gives the matrix $B = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$

A column vector is the transpose of a row vector.

16) Set up a column vector of the first 10 whole numbers by writing **(1:10)'**

• Section 8 – Saving and retrieving your work

All variables that you created in this MATLAB session are stored in MATLAB's workspace. Upon exiting MATLAB this workspace will be destroyed. So you must save your workspace if you want to use it in later MATLAB sessions.

1) Select *Save Workspace As* under *File* on the menu. A save window will come up. Check that the correct folder is given. Type in the name you wish to use for the workspace file, for example 'lab6' and save the file. MATLAB automatically selects the '.mat' extension for the workspace files.

VERY IMPORTANT:
These .mat files are **BINARY** files which means that you CAN NOT EDIT OR READ THEM like you can with ascii files. SO DO NOT ATTEMPT TO DO THAT. Reading the saved information MUST be done using the MATLAB *Import Data* option discussed below.

2) Type **clear** to delete your workspace. Type **who** to check that your workspace is empty (or check your Workspace window). Retrieve the saved workspace using *Import Data* under *File*. Type **who** again to check that all your variables are back in the workspace (or check the Workspace window). So, use the *Import Data* option to load mat files back into memory.

This method of saving saves all the variables in your workspace. You can select any number of variables if you save from within the Command Window instead.

3) Create two random variables by typing **x=rand;** and **y=rand;** To save these two variables in a .mat file called 'savexy.mat' type **save savexy x y** .

The contents of your current directory is shown in the Current Directory window. But you can also find out its contents using a command:

4) The command **dir** gives you the list of all files that are in your current directory. The command **what** gives you the files in the current folder relevant to MATLAB only. Try them both.

Practice

1) Set up the matrix $\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$. Interchange the 2nd and 3rd rows *using one line of code*.

2) Create a 10x10 matrix with random numbers between 0 and 10. Now, make all elements in the first row and first column equal to 1.

3) We would like to create the row vector $\begin{bmatrix} 8 & 6 & 4 & 2 & 0 & \dots & 0 & 0 & 10 \end{bmatrix}$ with a total number of elements equal to 200 (that means there are 195 zeros in the vector). Think of two ways to create this variable without typing in all the numbers.

• Section 9- Script files

In this section we show you how to store commands in a file. Such MATLAB command files are known as *script files*. They are stored with the .m extension. In MATLAB files with the .m extension are also called M-files. All script files are M-files, but not all M-files are script files as you will see later (for example, MATLAB functions are also stored in .m files)

1) From the menu choose *File* then *New* then *M-file*.

You are now in the *MATLAB Editor/Debugger*.

2) Type the following lines in the new file window. Note that the file is still called 'Untitled'.

```
% Add two matrices and display the result
A = [1, 1; 2, 2]
B = [2, 2; 3, 3]
D = A + B
disp('The result is ');
disp(D);
```

The first line is a comment written to explain what your program does. MATLAB ignores everything on a line after a %-character. Note that comments are displayed in a different colour so you can directly recognize them. It is very important to write adequate comments so that other people who need to use your program can understand it.

You may have noticed that when closing brackets the two matching brackets are identified (usually by flashing or underscoring them). This is useful in cases where you have several sets of brackets: you can easily check if all brackets are matched up.

The **disp** command is used to display output strings on your screen, as in **disp('the result is ')** and also to display variables as in **disp(D)** . The text that will be displayed is shown in a different colour. Note that we can display strings or variables but not at the same time. So the command **disp('the result is ',D);** is <u>not accepted</u>.

We will now save this file.

3) Choose *File* then *Save As* from the menu. The current directory (which should be the directory you just created) is automatically selected. Save the file as addmat.m .

4) Go back to the command window and type **addmat** to run the file (note that you should not type the .m extension!). If MATLAB gives an error saying that it can not find addmat, make sure that the directory in which the addmat.m file is stored and MATLAB’s current directory are one and the same.

Running a program step-by-step, one command at a time

MATLAB runs the addmat program without breaks or pauses. Often it is convenient to run a program step by step to carefully check that all the lines of code that you wrote are doing what you expect them to do. This is a good way to find errors in your code, or in other words to <u>debug</u> your code.

5) Go back to the MATLAB Editor/Debugger window. Click on the first command line of the addmat.m (the line **A = [1, 1;2, 2];**). Now select *Breakpoints* from the menu and then the line containing *Set /Clear Breakpoint.*

This will force MATLAB to break the execution of addmat at this line. You should see a red dot appear left of this command line.

6) Select *Run* under *Debug*. A yellow arrow appears at the first command line in the Editor/Debugger window indicating that MATLAB stopped at that line and is waiting for you to tell it to continue. Select *Debug* and *Step* to step to the next command line in the script file. Back in the Command

window you see the first line being executed and A is shown as is expected. Step through the rest of the program and check the results in the command window.

The matrices A and B are written to screen because we did not add any semi-colons to the end of the command lines. To view the contents of A and B you could of course also use the Workspace window as we did last week.

Under *Debug* and *Breakpoints* on the menu there are more options than the single step option we just explored. For example, you can ask MATLAB to stop when it encounters an error *(Stop if error* under *Breakpoints*) or when it sees that you are not doing something quite right (*Stop if warning* under *Breakpoints*). Also, you can create a breakpoint further down in your file and use *Continue* under *Debug* to run the program until it hits the next break point. Explore the MATLAB debugger in these labs: it is a very useful tool.

Also have a look at *Text* in the Editor/debugger window. This gives you handy ways to comment out parts of your code, uncomment or improve your layout using indentation.

• Section 10 - Logical expressions

Relational operators

MATLAB has 6 relational operators to make comparisons between variables. These are

<	*is less than*
<=	*is less than or equal to*
>	*is greater than*
>=	*is greater than or equal to*
==	*is equal to*
~=	*is not equal to*

Note that the *is equal to* operator consists of two equal signs and not a single = sign as you might expect. We'll play around with these relational operators for a little while.

1) Go back to the Command Window. Set up the scalars q=10, w=10, and e=20.

In MATLAB a logical expression has two possible values that are not 'true' or 'false' but *numeric*, i.e. 1 if the expression is true and 0 if it is false.

2) Type **w< e** to see that the result is given the value of 1 because w is indeed less than e. Now type **q==e**. The == operator checks if two variables have the same value. Therefore the answer is 0 in this case.

Relational operations are performed *after* arithmetic operations. For example **q == w-e** results in 0. Parentheses can be used to override the natural order of precedence. So **(q==w) - e** results in -19.

The relational operators can be used for all elements of vectors simultaneously.

3) Create the vectors x and y by typing **x=[-1, 3, 9]** and **y=[-5, 5, 9]** . Now type **z= (x < y)** . The i-th element of z is 1 if x(i) < y(i), otherwise it is 0. So, the answer is the vector with elements 0, 1, and 0.

Logical operators

MATLAB uses three logical operators. These are

and	**&**
or	**|**
not	~

For **&** (and) to give a true result both expressions either side of the **&** must be true. The logical expression **e> 0** is true and the logical expression **q<0** is false. So the logical expression (**e>0) & (q<0**) is false.

For **|** (or) to give a true result only one of the expressions either side of the **|** needs to be true.

4) Type **(e>0) | (q<0)** . Is the result as expected?

The ~ (not) operator changes a logical expression from 0 to 1 and vice versa. So **result = ~(q<0)** would be 1. The operator ~ has a high priority. So, to avoid mistakes, put whatever expression you want to negate in parentheses, as we did here.

The operations **&** and **|** are performed after the relational operations,<>== etc.

Just as with relational operators we can use logical operators for vectors or matrices.

5) Try **z = ~(x < y)** . Is the answer as you expected?

• Section 11 - The 'if' statements

The if-statement starts with **if** followed by a condition, usually a relational operation, and is finished with an **end;** command. For example (taken from a soccer game simulator)

```
if  score ~= 0
        disp('GGOOAAALLL – the score is ');
        disp(score);
end;
```

The body of this if-statement consists of two lines and both will be carried out if the score is not equal to zero. We can include as many lines in a body as we want.

We could also write a message if the score is zero. Instead of writing a separate if-statement for this case we combine it with the previous one in

```
if  score ~= 0
        disp('GGOOAAALLL – the score is');
        disp(score);
else
        disp('darn – kick too soft');
end;
```

If the score is not zero the first body will be carried out (the one immediately following the first condition statement). Otherwise the command in the body after the else-statement is carried out.

Another example:

```
if  score == 1
        disp('FFFIIIRRRSSSTTT   GGGOOAAALLL');
elseif score > 1
        disp('We are on a roll');
else
        disp('darn – kick too soft');
end;
```

The **elseif** statement allows us to include more options.

Yet another example where now we check if a penalty kick will enter the goal (our keeper has a reaction time of 0.5 seconds):

```
if  time < 0.5
        % compute the height of the ball as it reaches the goal
        height =  initialS*time*sin(angle) – 0.5*g*time^2;
```

```
                if height < goalheight
                        score = score + 1;
                end;
        end;
```

From this example we see that it is possible to have nested if-statements, that is if- statements inside other if-statements. Now we get two **end** statements, one for each **if**. It is VERY IMPORTANT TO USE INDENTATION!!! Otherwise it is hard to figure out which **end** belongs to which **if**. Note that you can use the *Text* menu item in the editor/debugger window to add fancy indentations.

Practice

1) Create two random numbers x and y. Write an if-statement that displays x if x is smaller than y. Now, write an if-else statement to display the minimum of x and y. Create a third random number z. Write an if-statement that displays the value of z if z is larger than x or y.

2) The height h(t) and speed S(t) of a projectile (such as a kicked soccer ball, or a rocket) launched with an initial speed S_0 at an angle A to the horizontal are given by

$$h(t) = S_0 t \sin(A) - 0.5gt^2$$

$$S(t) = \sqrt{S_0{}^2 - 2S_0 gt\sin(A) + g^2t^2}$$

where g is the acceleration due to gravity. The projectile will strike the ground when h(t)=0 which gives the time to hit $t_{hit} = 2(S_0 / g)\sin(A)$.

- Write a MATLAB program that computes the hit time for a given initial speed and angle, and displays the hit time and the speed of the rocket when it hits the ground.
- Follow the programming steps as discussed in the lecture.
- Store the program in a script file called rocket.m and run it.

Note: you can compute square roots using the MATLAB function **sqrt** . Do not forget to include comments and a header, use meaningful variable names and indentation when appropriate.

• Section 12 - The 'for' loop

Let’s start with a very simple example. We will create a vector with 3 random angles between 0 and π, compute the cosine of each angle, and display it.

The program is

```
angle = pi*rand(3,1);

for k=1:3
        cosangle = cos(angle(k));
        disp(cosangle);
end;
```

Note

- k is called the loop-variable.
- the loop is ended as required by the **end;** statement
- the body is the code between the **for**-statement and the corresponding **end;**
- the body is indented to make it easy to see what’s happening inside the loop

Now, let’s save the computed cosines and store them in a vector called cosines. To do this we can add the line **cosines(k) = cosangle;** at the end of the body of the for-loop and remove **disp(cosangle);**

1) Try that.

In this for-loop we access the k-th element of the vector called angle in each pass and store the result of our cosine computation in the k-th element of the vector called cosines.

We could shorten the program a bit by storing the compute cosine value in the cosines vector immediately, and replace the body of the for-loop with the one line **cosines(k) = cos(angle(k));** .

In the above examples the loop increment is 1 (the loop increment is the amount with which the loopvariable is incremented each time MATLAB starts a new pass). But it does not have to be 1. For example, we can loop over the odd elements of angle by writing **for k=1:2:5** instead. The loop variable k is now incremented with 2 at each pass and takes on the values 1, 3 and 5. The second and fourth element of the cosines vector have not been given a value by the program. MATLAB automatically sets them to zero.

Let’s look at another example. Suppose we want to find the maximum element of a vector x with 4 elements that are random numbers between 0 and 1. We could use the MATLAB function **max** of course, but here we write our own code.
We store the maximum value in the variable called maxval. At the start of our code we initialize maxval to 0. Then we access all elements of the vector x one-by-one and each time check if the element is larger than maxval. If so, we set maxval to this new value. Accessing the elements can be done in a for-loop:

```
x = rand(4,1);
maxval = 0;

for k=1:4
    if (x(k) > maxval)
          maxval = x(k);
    end;
end;

disp(maxval);
```

Note that both the if-statement and the for-loop must have a corresponding end statement. We used indentation and therefore it is easy to see which end belongs to which statement.

Do you understand why we initialized the variable maxval to 0 at the start of our program?

• Section 13 – Nested for-loops

Just as we can have if-statements inside for-loops we can have for-loops inside for-loops. We compute the exponential of each element of a 2x2 matrix A.

```
A = rand(2,2);

for k=1:2
        for m=1:2
                exponential = exp(A(k,m));
                display(exponential);
        end;
end;
```

What happens? Let's go through the program step-by-step. You can do this in the debugger as well. We will call the k-loop the outer loop and the m-loop the inner loop.

1. MATLAB enters the outer loop and sets k to 1.
2. At the next line, MATLAB reaches the inner for-loop and sets m to 1.
3. It computes the exponential of A(1,1) and displays it.
4. It reaches the **end;** statement corresponding to the inner-loop.
5. Since it has not completed the inner loop yet, MATLAB jumps back to the **for**-statement of the inner loop and sets m to 2.
6. It computes the exponential of A(1,2) and displays it.
7. It reaches the **end;** statement of the inner-loop.
8. It realizes that it has completely carried out the inner loop and continues to the next line in the program which is the **end;** statement of the outer-loop.
9. Since it has not completed the outer loop yet, MATLAB jumps back to the **for**-statement of the outer loop and sets k to 2.
10. It goes to the inner loop and passes through it twice as above for k=1, computing the exponential of A(2,1) and displaying it, and then computing A(2,2) and displaying it.
11. It then reaches the **end;** statement of the outer loop once more, realizes it is done and stops.

• Section 14 – While loops

In a for-loop we repeat the statements inside the body a fixed number of times. But often we want to repeat some calculation not for a fixed number of times but until some condition is satisfied. Let's look at a very simple example.

```
x = 7;
while x > 0
        x = x-4;
        disp(x);
end;

disp('Done');
```

How does it work?
1. The variable x is equal to 7 initially.
2. Since x > 0 MATLAB enters the body of the while loop and subtracts 4 from x, which is now equal to
3. MATLAB displays the new value of x.
3. MATLAB reaches the **end;** statement and jumps back to the while-condition (**while x>0**).
4. Again x > 0 so MATLAB goes into the while-loop, sets x to –1 and displays it.
5. MATLAB reaches the **end;** statement and jumps back to the while-condition.
6. Now x is smaller than 0, so MATLAB will not enter the body of the while-loop but jumps to the final display statement and writes "Done" to screen.

In this example we must set x to a value (initialize) before we start the while-loop. Otherwise the condition in the while-statement can not be checked. So, MAKE SURE YOU INITIALIZE.

Here, x should change value as we are looping and at some point reach a negative value. Otherwise we will never get out of the while-loop. So, MAKE SURE THAT THE WHILE-LOOP TERMINATES.

Another example. Suppose that we are gambling on a 1-dollar slot machine. We start off with 10 dollars and will keep playing until we run out of money. It's a very simple machine: we have 25 % chance of winning 2 dollars each time we play. We can simulate a run by drawing a random number between 0 and 1 using the function **rand** . If the number is below 0.25 we win, otherwise we loose. We will keep track of how many times we can play. In MATLAB code

```
money = 10;
numberplay = 0;

while money > 0
        money = money – 1;
        % keep score
        numberplay = numberplay + 1;
        % draw a number
        draw = rand;
        if  draw < 0.25
                money = money + 2;
        end;
end;

disp(numberplay);
```

Below is a third example:

```
money = 10;
while money ~= 0
        money = money-3;
end;
```

The program does not terminate. Do you see why?

• Section 15 - Advanced matrix and vector operations

In many engineering applications we need to solve matrix-vector equations. This is very easy in MATLAB, as MATLAB was especially designed for matrices. In this section we look at some matrix operations and then discuss how to solve the matrix-vector equations. We will run this section from the command window. It is not necessary to store the commands in an script file.

We create the matrices $A = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}, B = \begin{bmatrix} 4 & 2 \\ 3 & 5 \end{bmatrix}$.

We can 'divide' the matrices by typing **G=A/B**. This is called right division and finds the matrix G such that A=G*B, or in other words: *G equals A times the inverse of B.*

1) Perform the right division by typing **G=A/B**, and check that GB is the same matrix as A by typing **G*B-A** .

MATLAB also allows left division entered by typing **H=A\B** which calculates H such that A*H=B, or in other words, *H equals the inverse of A times B*. So, in matrices slashes refer to inverses:
If the slash is a backslash (\) then the inverse is taken of the preceding matrix. If it is a normal slash (/) then the inverse is taken of the following matrix!

2) Perform the left division by typing **H=A\B** and check that AH is the same matrix as B by typing **A*H-B** .

Now suppose that we want to solve the system of equations represented by the matrix-vector equation $Ax = b$, where x and b are two vectors with n elements and A is an nxn matrix. For example, A is given as above, $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ is the vector with unknowns and $b = \begin{bmatrix} 5 \\ 2 \end{bmatrix}$. The matrix-vector equation then represents the system of equations $\begin{matrix} x_1 + 2x_2 = 5 \\ x_2 = 2 \end{matrix}$. The answer is given by $x = A^{-1}b$. Using left division we can compute x with **x = A\b**.

3) Create the vector b and then type **x = A\b** and check that the answer is given by $x_2 = 2, x_1 = 1$. Compare the result of the command **A*x** with b. They should be the same. Now try to type **x=A\b'** using a row vector instead of a column vector. You will get an error message.

Often people are tempted to use the command **x = b/A** instead. This command is accepted if b is a row vector but gives completely the wrong answer. The correct command **x = A\b** only works for a column vector b. So a good 'trick' is to make sure b is always a column vector!

In class, we will discuss the LU decomposition. The above command **x=A\b** actually computes the LU decomposition of A first behind the scenes and then performs the back and forward substitutions that we'll talk about in class. The L and U matrices are not stored for you. To explicitly generate them, you can use the **lu** function.

4) Find the L and U matrices of A by typing **[L,U] = lu(A)** . If needed, MATLAB will perform pivoting. To get the permutation matrix, you can use **[L,U,P] = lu(A)** .

Note that it is of course not necessary to use the names L, U or P in the left hand side. You can use whatever notation you like such as **[Aap, Noot, Mies]** = **lu(A)**, but do remember that the order in which the solutions are given is fixed, so if you type **[P,U,L]** = **lu(P)**, the L matrix will be stored in your variable P, and P in your variable L.

Other useful MATLAB matrix functions for this course are **eig** that computes eigenvalues and eigenvectors of a matrix, **orth** that computes an orthogonal basis for the range of a matrix, and **svd** that returns the singular value decomposition. Do browse for useful functions as the course progresses.

Practice

1) Create two random vectors x and y, each with 5 elements. Write a for-loop to add x(1) to y(1), x(2) to y(2), etc. Each time, store the computed value in a variable called sumelements.

2) Write a while-loop that finds the index of the first element of x that is larger than the corresponding element of y. For example, if x = [0.1, 0.11, 0.05, 0.8, 0.91] and y = [0.83, 0.64, 0.09, 0.42, 0.5], you should find that the required index is 4, because x(4) > y(4) but the first three elements of x are all smaller than the corresponding elements of y.

• Section 16 – Writing your own MATLAB functions

Let's look at a MATLAB function, for example **cos**. If we type **cosangle = cos(0)** we give the function a value (0) and it returns a value for the cosine which is stored in cosangle. Another example: when calling the function **disp**, *you give it* a string or a variable as in **disp(cosangle)** and *it returns* text to the screen.

<u>In all cases, *you give* the function information, and *it returns* a result.</u>

We will call the information you give the function the 'function input' and the returned results the 'function output'.

MATLAB has a large number of built-in functions, and the number is constantly increasing with new releases. But, MATLAB may not always supply what you want or need. You can construct your own functions in MATLAB and in this section we will tell you how.

The first MATLAB function you will write is a simple function to convert temperatures given in degrees Fahrenheit into degrees Celsius, according to the formula:

$$t_{celsius} = (t_{fahrenheit} - 32) * 5/9$$

1) Open a new M-file. Type

```
function tc = fahtocel(tf)
    % converts function input 'tf' of temperatures in Fahrenheit to
    % function output 'tc'of temperatures in Celsius
    temp = tf-32;
    tc = temp*5/9;
return;
```

The word **function** on the first line tells MATLAB that you are about to define a function. The next words **tc = fahtocel(tf)** tell MATLAB that:
- the function is called 'fahtocel'
- the function returns a function output variable called 'tc'
- the function is given a function input variable called 'tf'.

The **return** statement indicates the end of the function and asks MATLAB to return to the place from which the function was called. You can call functions from the command window of course, or from script files, or even from other functions.

Note that we indented the body of the function for clarity. This is not required but it does make it easier to read the file.

Function M-files must <u>always</u> start with the **function** statement and the name of the function M-file must <u>always</u> be the same as the name of the function. So here, we must save this M-file as 'fahtocel.m'.

2) Save the M-file as 'fahtocel.m'. MATLAB will suggest the name fahtocel.m to you automatically and all you have to do is press the save button.

The function is now known to MATLAB and we can use it like any MATLAB function.

3) The boiling point of water in Fahrenheit is 212°. This is 100° Celsius. To show this type **boilcel = fahtocel(212)** in the Command Window.

We can also pass a variable name to our function instead of a number.

4) For example, type

```
tempf=1000;
tempc=fahtocel(tempf)
```

to get the required answer of 537.78° (rounded).

Could we use the function to compute temperatures in Celsius for a vector or matrix of temperatures given in Fahrenheit, all at the same time? Let's try.

5) Create the vector [212 200 350] by typing **tempf = [212 200 350]** . Now type **fahvec = fahtocel(tempf)** .

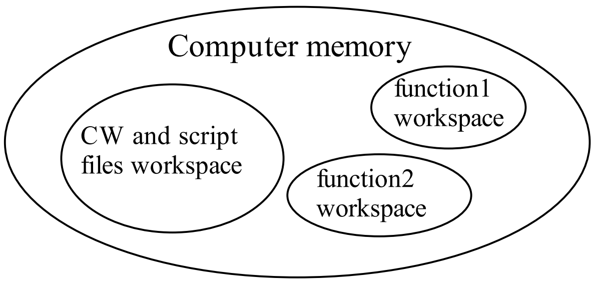
The resulting vector 'fahvec' contains the conversions of all the elements of the 'tempf' vector. So here, we can use the function for vector inputs as well. It will also work for matrices.

6) Type **tempf = [212, 200; 100, 300];** followed by **fahmat = fahtocel(tempf)** .

As you can see, this function also accepts matrices.

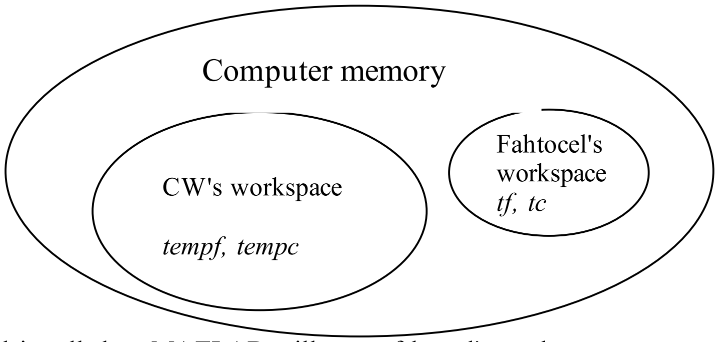
• Section 17 - How do functions work?

Each function has its own workspace in memory. This workspace is different from the workspace used by the Command Window and script files. In the discussion below we will use the abbreviation CW for the command window and talk about 'CW's workspace' instead of 'CW and script files workspace' for brevity.



1) In the command window, type **tempf = 212;** followed by **tempc=fahtocel(tempf)**

The variables tempc and tempf are used in the command window so they live in CW's workspace.



The function fahtocel is called so MATLAB will go to fahtocel's workspace to execute the function M-file. The function is defined as **function tc = fahtocel(tf)** . So, in fahtocel's workspace we have the variables tc and tf. Comparing the call to the function **tempc = fahtocel(tempf)** with the function definition **function tc = fahtocel(tf)** we see that tempf in CW's workspace corresponds to tf in the function's workspace, and tempc in CW's workspace corresponds to tc in the function's workspace.
CW can not read or access tf and tc. Fahtocel can not read or access tempf and tempc.

2) To check that the variable tc is really not known in the command window, type **tc** in the command window. MATLAB warns you that this variable is not known.

MATLAB copies the value of tempf , which is 212, to tf in fahtocel's workspace.
The function computes tc, which is 100. MATLAB copies the value of tc to tempc in CW's workspace. This is illustrated below:

$$\begin{tikzcd} \mathit{tempf} \arrow[r, dash] & 212 \arrow[r] & \mathit{tf} \\ \mathit{tempc} & 100 \arrow[l] & \mathit{tc} \arrow[l, dash] \end{tikzcd}$$

Because the function workspace and the normal workspace are completely separate the same variable names could be used in each of the workspaces to stand for two different variables.

3) To test this, type **temp = [1, 2];** in the command window to create a variable called temp in the command window's workspace. Then type **tempc = fahtocel(tempf)** to run the program again followed by **disp(temp)** .

Your output shows that the variable named temp is still the same vector [1, 2] in the command window even though the function used a variable with the same name. This is nice: we cannot mess up variables in the command window if we accidentally use the same variable names in a function.

• Section 18 - Functions with more than one input variable

So far, we have only seen functions with one function input variable and one function output variable. Functions can have more than one input variable or output variable however.

As an example we will write a function to find the volume of a cuboid with the lengths of the three sides as the input variables.

1) Type the following in a new M-file and save the function.

```
function vol = cuboid(len,br,dep)
  % Function to return the volume of a cuboid given
  % the length len, breadth br and depth dep
  vol = len*br*dep;
return;
```

This function accepts <u>three</u> input arguments and returns one output argument.

We will now call this function from a script file instead of the command window.

2) Open a new M-file and type

```
% script file for cuboid
length=5;
breadth=4;
depth=3;
% call the function cuboid
volume = cuboid(length,breadth,depth);
disp(volume);
```

When you call the function, you must make sure that the input variables are given in the correct order. Here, the first input variable is length, the second breadth and the third depth. This corresponds to the order in which they are written in the function itself.

Note that in the function the variables are known as len, br and dep. So length in CW's workspace corresponds to len in cuboid's workspace. The *values* of length and len are the same. Similarly, breadth corresponds to br and depth to dep. The output variable vol in cuboid's workspace corresponds to the variable volume in CW's workspace.

3) Save this script file as cub.m, and run this script file from the command window. You should see that the volume is 60.

4) Delete the **,depth** from the last line of cub.m so that the line finishes with **cuboid(length,breadth)** , save the file, and try to run it again.

MATLAB objects because it knows that the cuboid function needs three input arguments. You provided only two, so it gives an error message suggesting that it cannot find the third argument which is known by the name 'dep' within the function.

• Section 19 - Functions with more than one output variable

Suppose that we also want to compute the surface area of the cuboid. We do not have to construct a second function, but can compute both volume and surface area in the cuboid function.

1) Change your cuboid function to

```
function [vol, surf] = cuboid(len,br,dep)
  % Function to return the volume and surface area of a cuboid given
  % the length len, breadth br and depth dep
  vol = len*br*dep;
  surf = 2*(len*br + len*dep + br*dep);
return;
```

The function now accepts three input arguments and returns two output arguments. The output arguments are always written in square brackets, separated by commas.

2) Could we compute volumes and surfaces for more than one cuboid at the same time? Let's try to change the cub.m file to

```
% script file for cuboid
length =[ 5, 10];
breadth = [4, 10];
depth = [3, 8];
% call the function
[volume, surface] = cuboid(length,breadth,depth)
```

Run it.

MATLAB will complain that matrix dimensions in the multiplication operation of cuboid do not agree. We can solve this by using the "element-by-element" operations instead. These are created simply by inserting a dot before the operation symbol. For example, **vol = len.*br.*dep;** will compute a matrix vol with two elements, the first giving the volume for the first cuboid with length 5, breadth 4 and depth 3 and the second element giving the volume for the second cuboid with length 10, breadth 10 and depth 8.

Note that by using **[volume, surface] = cuboid(length,breadth,depth)** we tell MATLAB to store the output of the cuboid function in the variables 'volume' and 'surface'. Again, the order in which you write volume and surface should correspond to the order in which they are written in the function description. Here, vol in cuboid's workspace corresponds to volume in CW's workspace and surf in cuboid's workspace corresponds to surface in CW's workspace.

• Section 20 - Functions with no input or no output variables

Functions do not HAVE to have input or output variables. Suppose that we want the cuboid function to compute the volume and surface and display them on screen, but we are not interested in keeping the output. Then we do not have to specify any output variables in the function. The cuboid function then looks like

```
function cuboid(len,br,dep)
  %  Function to return the volume and surface area of a cuboid given
  %  the length len, breadth br and depth dep
  vol = len*br*dep;
  surf = 2*(len*br + len*dep + br*dep);
```

```
    disp('The volume is ');
    disp(vol);
    disp('The surface area is ');
    disp(surf);
  return;
```

We simply left out everything to the left of the function name in the function statement.
The cub.m file needs to be changed now too. In the previous file, we asked for the volume and surface to be sent back from the function. Now we cannot, because the function does not have any output. So we change the line **[volume, surface] = cuboid(length,breadth,depth)** to **cuboid(length,breadth,depth);**

1) Change the cuboid function and cub file as above and run the program.

An example of a function that does not take any input variables is discussed in the assignment.

Practice

1) Create a function that computes the volume of a sphere with a certain radius. Call the function 'volsphere'. Now compute the volumes of the spheres with radii 0.1, 0.2, 0.3, through to 1.2 using a for-loop. Display the computed volumes. Store the program in a script file called volumes.m

2) Change the function and script file of task 1 so that you also compute and display the surface areas of the spheres.

• Section 21 - Introduction to MATLAB graphics

MATLAB has a large number of functions associated with graphical output. If you'd like to explore the possibilities use **help plot** or **help plot3** for 3-dimensional plots, or run the MATLAB demo (by typing **demo**) and look at the information on visualization and graphics.
We start with basic plotting routines and look at some fancy graphics to get a taste of MATLAB's abilities.

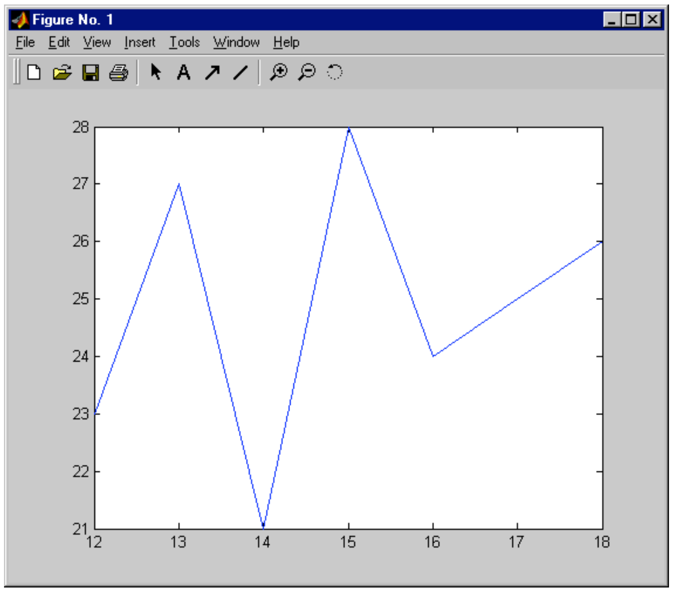
Suppose that the maximum and minimum temperature (in degrees Celsius) recorded from 12 to 18 December are 23, 27, 21, 28, 24, 25, 26, and 11, 10, 15, 15, 14, 15, 12 respectively.

1) Set up the vector 'date' to have elements from 12 to 18, and the vectors 'maxtemp' and 'mintemp' to contain the temperature data given above. It does not matter whether the vectors are row or column vectors.

All the vectors are of the same length so it is possible to plot one against the other.

2) Create a plot of the maximum temperatures by typing **plot(date, maxtemp)**

The x-axis variable is listed first. This graph will be created in a window called Figure No. 1. If this window is not visible select *Figure No. 1* under *Window* on the menu. Your graph should look like the one below.



3) Now type **plot(date,mintemp)** .

The previous plot disappears and the new plot is displayed.

We can also plot both graphs in one figure:

4) Type **plot(date,maxtemp);** followed by **hold on;** Then type **plot(date, mintemp);** followed by **hold off;**

hold on tells MATLAB to keep the old plot and add the new graph to it. **hold off** turns the hold-feature off again. We could also have used **plot(date, maxtemp, date, mintemp);** which tells MATLAB to plot maxtemp against date and then mintemp against date in the same graph. The colours of the graphs are then different.

The plot could be made to look a lot better.

5) Select *Insert* from the figure window and then *Title*. A text box will appear in the figure window. Type '12-18 December'. Now add the label 'date' to the x-axis and 'temp (Celsius)' to the y-axis using the appropriate selections under *Insert*.

Note: in older versions of MATLAB a title can be created using the MATLAB command **title** and labels can be added using the **xlabel** and **ylabel** commands.

The set of axes chosen is just large enough to contain all the data points, but this doesn't necessarily produce the most pleasing display.

6) Select *Edit* and then *Axes properties*. Un-select *Auto* for X (click in the *Auto* box to remove the tick). Then change the limits of 12 and 18 for the x-axis to 11.5 and 18.5 respectively. Now change the limits for the y-axis to 9 and 30. If the *Immediately apply* box at the bottom of the Axes Properties window is not ticked you must click OK to see the changes.

This plot is better but some might say that discrete data points should not be joined with lines because, for example, 'maximum temperature on the twelfth-and-a-half day in December' is not a meaningful statement.

7) Click on the graph corresponding to the maximum temperature. Then select *Current Object Properties* under *Edit* in the figure window. Select *No line (none)* under *Line Style*. Under *Marker Properties*, select a marker of your choice, eg. Six-pointed star. Choose a nice colour.

8) Repeat for the graph corresponding to the minimum temperature with a different colour and marker.

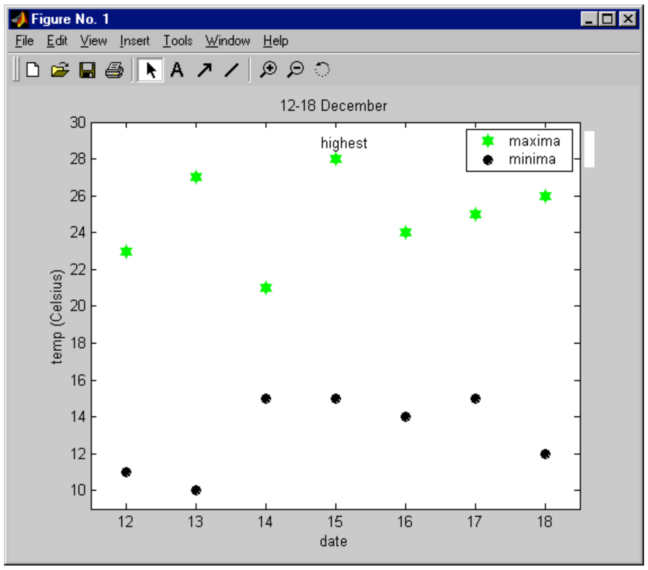
If you want to change two or more graphs at the same time (and give them the same properties), you click on the first graph, hold down the shift key while clicking on the other graphs and then follow the above steps.

9) To add a legend, select *Insert* then *Legend*. You can change the description of the graphs in the legend by double clicking on the text in the figure window. Change the descriptions to 'maxima' for the maximum temperature graph and 'minima' for the minimum temperature graph.

Finally we add some text inside the plot.

10) Select *Insert*, then *Text* (or click on the A button on the menu). Move your mouse to the plot window. Position the cursor near the highest point in the graph and click. Type 'highest' in the field.

The final graph is shown below



• Section 22 – Three-dimensional plots and a movie

We will first create a three-dimensional plot of a sample function often used in MATLAB demos. This is the function called 'peaks'.

1) Close the current figure window. You can do that by typing **close** or clicking on the x in the upper right corner of the figure window. Type **sampleplot = peaks(25);** This creates the matrix sampleplot whose elements are values of the peaks function. Type **surf(sampleplot);** to create a 3-D surface plot of these function values.

A three-dimensional plot is shown in the figure window. The axes are the x-, y- and z-axis. The z-axis is in the vertical. The colours vary with the value of z. The colour scheme used can easily be changed. Let's liven this picture up a bit:

2) Let's change the background colour of the plot. Go to *Edit* then *Figure Properties*. Click on *Style*. Select yellow for background colour. Select *Title* and set title to 'groovy plot' for example. This title will show up next to the Figure No. 1 text. Close the Figure Properties window. Click on the plotted surface. Select *Current Object Properties* under *Edit*. Go to *Color* and change the colormap. Play around to get a feel for what you can do. For example, the lighting possibilities of 3D objects are pretty impressive.

3) We can view the object from different angles. Go to *Tools* and then select *Rotate 3D*, or click on the rotation button (far right next to the magnifying glasses). Click anywhere in the figure and hold the cursor down. Now move the cursor around a bit. As you can see you can control the viewing point that way.

There are many many more features. Have a play! For example, try out the camera.

Let's look at another example.

4) Type **sphere** . This command will create a three dimensional picture of a sphere with radius 1. It is drawn (per default) using 400 points on the sphere.

Depending on the size of your plot window, the sphere may look flattened. Change the window size to a square to improve the look.

Now let's make a little movie. The program below is just an example to play with. You do not have to remember how to make movies since it's a little complicated. But if you ever want to make a movie yourself you can use this program as a start.

5) Create the following program:

```
% clear the figure window
clf;
% create a sphere using 10 points
[x,y,z]=sphere(10);
% create a first plot
surf(x,y,z);
% initialize a movie with 19 frames
M=moviein(19);

% fill the frames
for j=1:19,
        % compute the x-position of the sphere in this frame
        posx = j-1;
        % compute the z-position of the sphere in this frame
        % no worries about the maths here: we make the sphere follow
        % a parabolic path to simulate a bouncing ball
        posz = -(rem(j-1,6)-3)^2+9;
        % plot the sphere in the frame
        surf(x+posx,y,z+posz);
        shading interp;
        axis([0,18,-6,6,0,10]);

        % store the frame
        M(:,j)=getframe;
        disp('collecting frame ');
        disp(j);
end;
```

This program stores 19 pictures in a movie variable called M. Each time a frame is collected, the sphere is moved to a new position determined by posx and posz (the y value stays the same). The sphere follows a parabola (a bit like a bouncing ball).

6) If you have a figure window open close it using the **close** command. Run the program. Now type **clf**; followed by **movie(M);** to see the movie.

The movie you stored will be played. You may see the movie twice. The first time is when MATLAB stores the movie into memory. We can also run the movie backwards and forwards and at a slower rate by typing, for example, **movie(M, -2, 4)** . The argument –2 tells MATLAB to run the movie twice from front to back and back to front. The argument 4 tells MATLAB to run the movie at 4 frames per second.

• Section 23 – Fast MATLAB loops: the dot-operator

Sometimes for-loops can be sped up in MATLAB. Let's say that we want to square all 10 elements of a vector called 'v'. We store the squares in a vector called 'sqv'. We could write:

```
for k=1:10
        sqv(k) = v(k)*v(k);
end;
```

But we can do better in MATLAB. MATLAB provides a fast way to perform an operation on all elements of a vector (or matrix): the 'element-by-element method'. Instead of the for-loop we can write **sqv = v.*v;**

1) In the Command Window type **v=rand(10,1);** to create a random vector v with 10 elements, followed by **sqv=v.*v** . Check that sqv contains the correct values.

The element-by-element method always uses a dot in front of the operation. It performs the operation behind the dot on all individual elements of the vectors (or matrices). Another example

2) Type **w = sqv./v;** Each element of sqv is now divided by the corresponding element of v so w should be the same as v. Check this.

It is usually much more efficient to use the element-by-element method than the for-loop.

• Section 24 – A different way to look at MATLAB for-loops

So far we have used standard for-loops only. An example is

```
for k=1:10
        sq(k) = k*k;
end;
```

Here, we loop over the integers 1 through to 10, compute their squares and store them in the vector sq. You will encounter such for-loops in other programming languages as well. The syntax in other programming languages will be very similar to the syntax used in MATLAB.

In MATLAB however, we can look at this for-loop in a different way. We know that in MATLAB the command **1:10** creates the vector (1, 2, 3, …, 10). So, **for k=1:10** can be interpreted to loop k over all elements in the vector (1, 2, 3, …., 10).
Could we replace the vector 1:10 by another vector? For example,

```
for k=[1, 3, 10]
        sq(k) = k*k;
end;
```

1) Write the above code in a script file. Type **clear** to clear your workspace. Run the script file. Display sq.

You will see that this code computes the squares of 1, 3 and 10 and stores them in the first, third and tenth element of the vector sq respectively. So the command **for k=[1, 3, 10]** sets k equal to 1 in the first pass of the loop, equal to 3 in the second pass and equal to 10 in the third pass. MATLAB loops k over all elements of the vector [1, 3, 10]. Intermediate elements in sq are set to 0 by MATLAB.

2) Replace **[1, 3, 10]** by **[4, 2, 9]** in the script file, type **clear sq** and run the script file.

Now the squares of 4, 2 and 9 are computed and stored in the fourth, second and ninth element of sq respectively. We see that the elements in the vector used in the for-loop do not have to be in increasing order.

As a last example, suppose that we want to compute the squares of the elements of the vector [.1, -2, 5.7] and display them. We could write this in two different ways. First, the traditional way:

```
v = [.1, -2, 5.7];
for m=1:3
        square = v(m)*v(m);
        disp(square);
end;
```

Now we write it in the special MATLAB way by looping directly over the vector [.1, -2, 5.7]:

```
for k=[.1, -2, 5.7]
        square=k*k;
        disp(square);
end;
```

3) Write this last loop in a script file. Call the script file specialloop.m and run it. Check the answers.

Note that loops can only be written this way in MATLAB.

• Section 25 - Function-functions

A group of predefined MATLAB functions go by the curious title of *function-functions*. These refer to functions that have another function as an input argument. For example, the function-function **fminbnd** can be used to find the position of the minimum of a function.
We will use **fminbnd** to find the positions of the minima of the function $f(x) = 2x^4 + 3x^3 - 12x^2 - 7x + 2$. The function **fminbnd** takes three input arguments. The first input argument is the name of the function we want to minimize. The second and third arguments are the x-values between which MATLAB should look for a minimum.

This function f(x) is of course not readily available within MATLAB. That means that we have to write our own function for f(x). Since 'f' is not a descriptive name, we rename it to 'quartic'.

1) Without looking at the example below try writing the function quartic yourself.
Now check your function and save it. It should look like

```
function y=quartic(x)
```

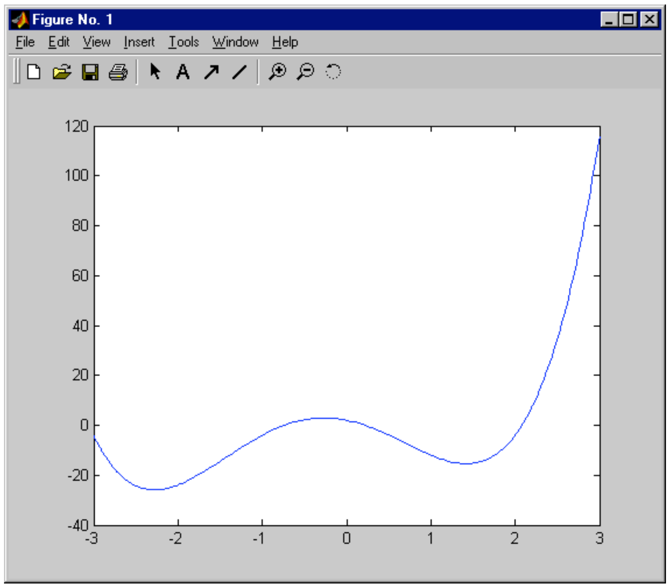
```
        y= 2*x.^4 + 3*x.^3 - 12*x.^2 - 7*x +2;
return;
```

Here, x is the input variable and y is the returned output. We have used the dot power (.^) so that we can use vectors as input variables as well.

A quartic polynomial may have two local minima. Let's make a plot of quartic to find out where the minima are located. This is easy using the special function plot command **fplot** .

2) In the Command Window type **fplot('quartic',[-3, 3]);** View the plot.

The function is plotted in the interval [-3, 3]. We see that the function has two minima in this interval, and that these are approximately given by x = -2.3 and x = 1.4.



We will search for the first minimum in the interval [-3,-2] and for the second in the interval [1,2].

3) Type **xmin1=fminbnd('quartic',-3,-2)** to see that the minimum is at -2.2747

When we call the **fminbnd** function with these arguments, MATLAB evaluates **quartic(x)** for many x-values in the interval between -3 and -2, and picks out the x-value (within some predefined error tolerance) which corresponds to a local minimum between these limits.

4) Type **ymin1=quartic(xmin1)** to see that the function at this local minimum has a value of -25.9320

5) Find out what happens if you give **fminbnd** the interval boundaries -3 and 2. Which of the minima is computed?

So, MATLAB only returns one value and will not warn you that there are more. You have to do some research (for example by plotting the function) to find out how many minima there are and where they are located approximately. We will compute the second minimum in the assignment.

Other function-functions are, for example, **fzero** which finds the value of *x* at which the function value is equal to zero, and **quad** which can be used to compute integrals.

The last example

For our last example we will solve a simple ordinary differential equation (ODE) in MATLAB. You have learned about the so-called ODEs in Mathematical Modelling 1. Remember the water tank problem? Water flows out of the bottom of a tank of muddy water through a small tap at a rate proportional to the volume. With time, the tap slowly clogs up and the rate of flow of water is inversely proportional to the time. Experiments show that the water in this situation flows at a rate which can be modelled by

$$V' = -\frac{0.5V}{t},$$

where V is the volume in litres and t is the time in hours starting at an initial time of 10 hours. The initial volume in the tank at t = 10 is equal to 24000.

In MATLAB it is reasonably simple to do this computation using functions and a built-in function-function called **ode23**. This function solves ODEs using a second order Runge-Kutta method with automatic time-step selection. MATLAB has several different Runge-Kutta solvers. Another solver is **ode45**, which is a fourth-order method.

How do we set this computation up in MATLAB?

First of all, we create a function that computes the right hand side of the above equation, in other words a function that tells us what V' (the rate of change of the volume) is at any given time.

6) Create the following function. Save it under the appropriate name.

```
function vprime = watertank(t,volume)

        vprime = -0.5*volume/t;
return;
```

This function can be used by **ode23** . All functions used by **ode23** must be of the form:

```
function derivative = name(t,value)
```

We can use any name we like for the derivative (above we use vprime), and the value (above we use volume) and the name of the function (watertank in the example), but we can not change the order in which t and value are given inside the brackets.

Solving the equation is now simple.

7) In the command window type **timespan=[10,100];** followed by
[allt, allvolumes] = ode23('watertank',timespan, 24000)

MATLAB now computes volumes at a lot of different times t between 10 and 100, with an initial value of 24000 for the volume. So the first argument of the ode23 function is the name of the function in which you stored the derivative. The second argument is the timespan, and the third the initial value of the volume.
The computed volumes are stored in the vector 'allvolumes' and the times at which the volumes are computed are stored in 'allt'.

8) To see a plot of the volumes as a function of the time type **plot(allt,allvolumes)**

In this example MATLAB decides at which times between 10 and 100 it computes the volumes. MATLAB chooses these times based on the required accuracy of the computations. The default accuracy is usually equal to 0.001 . You can change this accuracy but we do not want to go into that now.
We can force MATLAB to compute the volumes at specific times by storing more than two values in the 'timespan' vector. For example, suppose we are only interested in the volumes at times 10, 20, 30, …, 100. Then we can set **timespan=10:10:100;** and run ode23 again.

9) Try this.