Introduction to MATLAB

Harvard Systems Biology
Summer 2009

Mike Springer michael_springer@hms.harvard.edu
Zeba Wunderlich zeba@hms.harvard.edu

1 Before we get started

1.1 Goals and Logistics

This is meant to be either a guide during a MATLAB course or a stand alone introductory packet. The goal of this course/packet is to get someone who has never programmed before to be able to read and write simple programs in 6-8 hours.

Instructions you should type into MATLAB are displayed in **`red courier font.`**
You can cut and paste commands in this packet directly into MATLAB.

Outputs that you will see from MATLAB are shown in **`black courier font.`**
The output won't always be included in this document.

1.2 Mantra

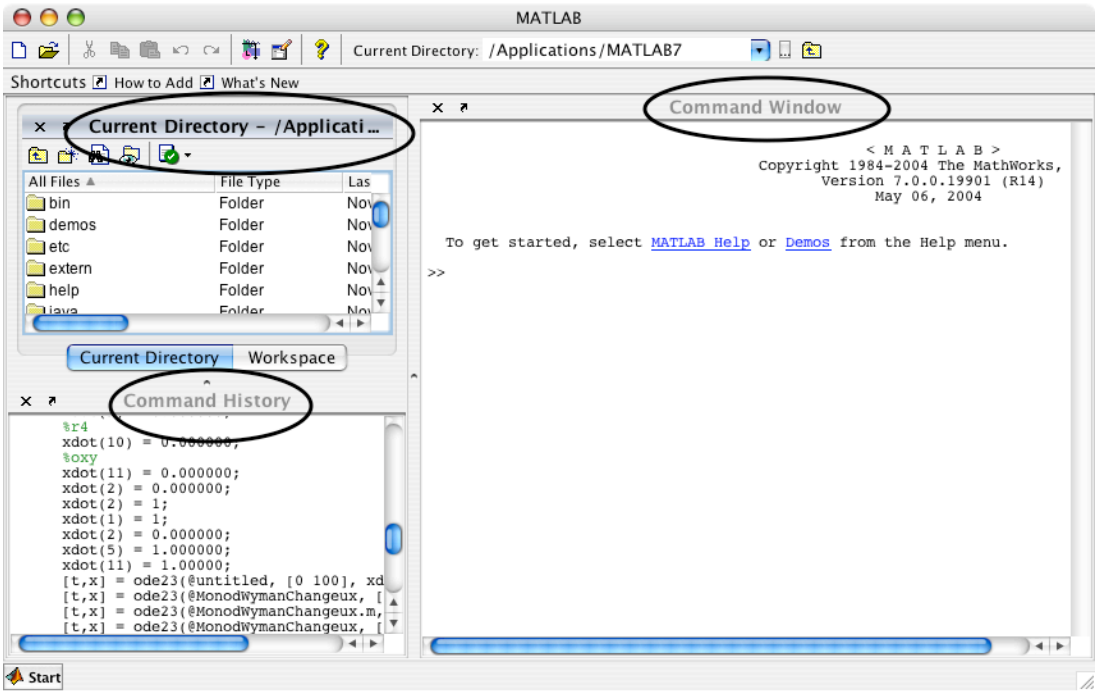
The goal is not to get through the packet as quickly as possible.
Explore and try things out. It is hard to break a computer.
Think about the commands you type in and make sure you understand the outputs.

1.3 MATLAB Help

If this is part of a course ask us any time you need help; otherwise …
There are many ways to access MATLAB help, some of which will be highlighted in the packet.

- Like most other programs, there is a help menu in the command bar at the top of the screen.
- If you know the name of the command, type '**`help <command>`**' in MATLAB
- If you don't know the name of the command, http://www.mathworks.com/access/helpdesk/help/techdoc/matlab.html

2 What you see when you open MATLAB



There are three subsections of this window:

The bottom left corner is the command history.
Any time you enter a command into MATLAB it records the command you entered.
The time and date that you log in is also recorded.
This is a useful way to repeat or view what you did on an earlier date.
You can copy and paste this into a different file, or double click a command to repeat it.

The top left corner is the current directory.
It tells you where you are in the workspace. This becomes important when you write programs that you will run later.
Alternatively if the workspace tab is highlighted, the top left will display all the variables and arrays that you have made and will indicate their current values.

The right subwindow is the command window into which you enter commands.

Often you will store all your files in one directory on you computer. You can change the directory you are in using the buttons just above the command window. We will demonstrate later.

3 Variables, Arrays & Functions

3.1 Variables

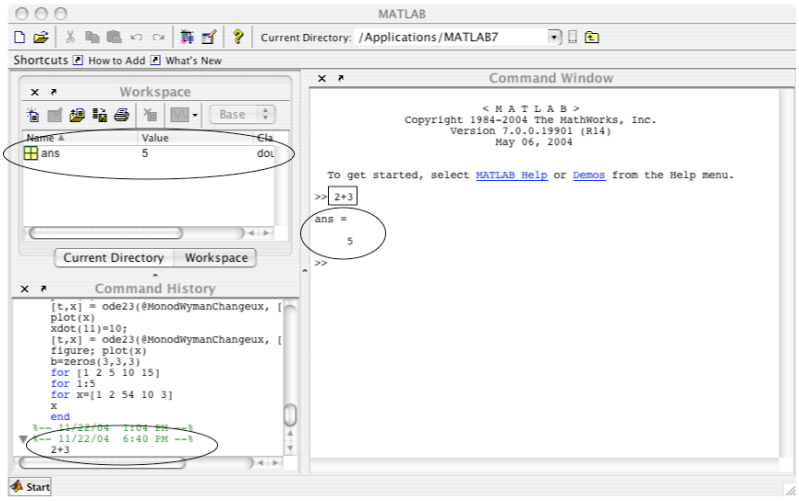
Let's start by entering a simple algebraic equation.

```
>> 2+3
```

into the command window (rectangle)
You will see that

```
ans = 5
```

appears on the screen. In the bottom left screen, the line you typed in appears. In the top left screen the variable **ans** appears (as shown below).




```
>> x=3;
```

This stores the value 3 into the variable **x**.

```
>> y=2
y = 2
```

Notice the difference of including the ";". A ";" suppresses the output of the line. You will almost always want to end your lines with semicolons. For most of this tutorial, the ";" will not be included so one can see the output for instructional purposes.

```
>> z=x+y;
>> z
z = 5

>> a=z+x*2
```

3.2 Arrays

Often, you will want to store more than one value in a single variable, e.g. data points in a time series. Arrays or matrices are incredibly useful in these applications. Due to its ability to efficiently work with large matrices, MATLAB actually stands for MATrix LABoratory. An array is just a table of numbers.

Storing multiple values in an array is useful because you can perform some operation on them simultaneously. For example, if we wanted to take the sine of a large number of values, we can store all these values in an array and take the sine with a single command. Like in Excel, individual elements or whole rows or columns can be easily extracted, plotted, or changed.

Retrieving values from an array

It is easiest to start by think of how to retrieve values from pre-existing arrays.

```
>> LoadArray
```

You have just run your first "script" or program. This command called a program (LoadArray), which we wrote to enter an array called **`MyArray`**. Look in the workspace and you will see the new array.

How big is the array (how many rows, how many columns)?

Double-click the array **`MyArray`** in the workspace. The array and all it's values now appears in a table above the command window. You can close this window by clicking on the x in the top right corner.

```
>> MyArray(1,2)
```

This returns the element that is in the first row, second column of the array **`a`**.

In general, **`a(m, n)`** retrieves the value in the m-th row, n-th column.

Guess what these commands will return before typing them in:

```
>> MyArray(4,3)

>> MyArray(4,4)

>> MyArray(4;3)
```

Defining a complete array

For the purposes of our exercises, let's define an array **`a`**.

```
>> a=[1 2]
a =
     1     2
```

Adding elements to a pre-existing array

```
>> a=[a 3]
a =
     1     2     3
```

This adds a 3 to the end of the array.

Column versus row vectors

a is a row vector – an array with one row and several columns. In some cases, we will want to work with column vector. In this context, a semi-colon indicates the start of a new row.

```
>> a=[1;2;3]
a =
     1
     2
     3
```

This creates a column array with one column and 3 rows.

You can see the variable **a** change in value in the upper left window.

To change a row vector to a column vector or vise versa, you can use the transpose operator `'`.

```
>> a'
ans =
     1     2     3
```

```
>> a=[a 4]
```

This doesn't work! Right now **a** is a 3 row x 1 column array, so we can't add another column with only 1 entry.

```
>> a=[a; 4];
```

This works, now **a** is a 4 row x 1 column array.

Mathematical manipulation of an array

```
>> a*2
a =
     2
     4
     6
     8
```

Note the 2* applies to every element of the column vector.

```
>> a=[a a*2]
a =
     1     2
     2     4
     3     6
     4     8
```

This works too! Matrix elements can be inserted and manipulated at the same time.

```
>> a=a+1
a =
     2     3
     3     5
     4     7
     5     9
```

This applies to every member of the array. Division by a constant (/) and subtraction of a constant (-) also work.

```
>> x=2;
>> a*x
```

This multiplies every element of the array by the scalar **x**.

When an array is multiplied or divided by, or added or subtracted to a scalar (a single number), each element in the array is manipulated in the same way.

Manipulating Arrays

You can also manipulate entries in arrays one at a time.

```
>> a(1,2)=a(1,2)+1

a =
     2     4
     3     5
     4     7
     5     9
```

Element retrieval and manipulation can be performed in one line.

This line took the value from the first row, second column of **a**, added 1 to this value, and then stored it back in array **a** in the first row, second column.

```
>> a(3,1)=a(3,1)+1

>> a(1,3)=a(1,3)+1
```

You receive an error message because there is no third column.

Errors messages in MATLAB are useful. Take time to read them. They will often let you quickly correct your error.

Arrays and values are interchangeable

```
>> a([1 2],1)=a(2,2)
```

This replaces the first and second row of the first column with the value in **a(2,2)**.

This is the real power of MATLAB. You can use values, an array, or even a function (which will hopefully return an array or value) interchangeably. MATLAB handles it correctly.

```
>> a(:,2)=0
```

This replaces every row in the second column with zeros.

Here the colon denotes all the rows in **a**. **a(:, 2)** is equivalent to **a([1 2 3 4], 2)**.
When you define an array you use []; when you call a position in an array you use ().

Matrix manipulation continued

```
>> a=[1 2; 3 4]
>> b=[2 4; 3 3]
>> c=a+b
```

This adds the elements of the matrices that are in identical positions, which is called an *element-wise operation*. This will only work if the dimensions of **a** and **b** match – they have the same number of rows and columns.
This is equivalent to the more laborious:

```
c(1,1)=a(1,1)+b(1,1);
c(1,2)=a(1,2)+b(1,2);
c(2,1)=a(2,1)+b(2,1);
c(2,2)=a(2,2)+b(2,2);
```

Addition and subtraction are always element-wise operations. However, division, multiplication and exponentiation (^) are not. This is because there are special definitions for these operations with matrices (linear algebra). However, to perform, e.g. element-wise multiplication, you can use the dot operator.

```
>> c=a.*b
ans =
     1     4
     9    16
```

This multiples each element of the array **a** by the equivalent element in array **b**, or:

```
c(1,1)=a(1,1)*b(1,1);
c(1,2)=a(1,2)*b(1,2);
c(2,1)=a(2,1)*b(2,1);
c(2,2)=a(2,2)*b(2,2);
```

In contrast, when we multiply two matrices together using '*' without the preceding dot, we carry out matrix multiplication.

```
>> a*a
ans =
     7    10
    15    22
```

This performed the matrix multiplication of **a** by **a**.

Entering arrays

There are many ways to enter arrays into MATLAB. When you have just a few data points it is pretty easy to just type them in. Things become much more complicated when there is lots of data.

Often you will have the data in another format and want to open in MATLAB.
There are ways to do this, which we will go into later.

Method 1: Type it in

```
>> t=[0 1 2 3 4 5 6 7 8 9]
```

This creates a variable **t**, which is a one-dimensional array (or a row vector), with 10 entries.

Method 2: Cut and paste
Open the file ExcelArray
Copy the whole array.
Now double click on MyArray in the workspace.
Now click on the top left corner of the array in the array editor: MyArray(1,1).
Paste the array (Go to Edit -> Paste, or type Ctrl+v, or right click and paste).
You have just entered a big array.

The colon operator

One can enter the array **t** above much more easily by using the colon notation:

```
>> t=[0:1:9]

t =
     0     1     2     3     4     5     6     7     8
9
```

This means make an array starting at 0 stopping when less then or equal to 9 and stepping by 1.

In general, **a:x:b** means start at **a**, and step by **x** until **b** is reached (**x** can be negative). The default step size is 1, so [0:1:9] is the same as [0:9]. If **x** is positive, the number will get bigger. If **x** is negative, it will get smaller.

More explicitly, let's take the example:
```
c = [1.1:2.3:6.8]
```

Here '**x**' = 2.3, which is greater than 0, to the series will increase. To compute **c**:

1. Start with **a** and check to see if that is less than or equal to **b**. Since 1.1 < 6.8, **c(1) = 1.1**. If **a** > **b**, the computation of **c** is complete.
2. Add **x** to **a**. In this example this results in 1.1 + 2.3 = 3.4. We will call 3.4 the new '**a**'
3. Return to Step 1.

In this example
```
c(2) = 3.4
c(3) = 5.7
```
5.7 + 2.3 = 8 > 6.8, so the computation of **c** is complete.

```
c =[1.1 3.4 5.7]
```

If **c=[-1.1:-2.3:-6.8]**, the calculation is analogous, except in step 1, the computation finishes when **a < b.** So this will return **c=[-1.1, -3.4, -5.7]**

```
>> t=[0:9]
>> t=[5:5:20]
>> t=[-2:3:8]
>> t=[0:0.5:2.75]
>> t=[8:1]
>> t=[8:-1:1]
>> t=[1.1:-2.3:6.8]
```

Try your own, and make sure you understand this.

You can also use the colon operator to extract elements from an array. For example, if you have a vector **t** with time points from an experiment and only want to look at every third time point, you can type `t(1:3:end)` and you will see every third entry of **t**.

3.3 Built-In MATLAB Functions

MATLAB has many built in functions that are useful for calculations.

```
>> sin(2*pi)
```

Functions can be performed on variables as well as on numbers.

```
>> t=2*pi;
>> g=sin(t)
```

Function call are always of the form
NAME_OF_FUNCTION(VALUE_TO_PERFORM_FUNCTION_ON)

Like variables, functions are an incredible useful organization tool and we will come back to them repeatedly.

There are a bunch of useful built in functions in MATLAB. You can often guess at the name (for example mean calculates the mean; median finds the median). This short list is meant to give you a feeling for how the functions work.

To see if a function exists, type the first couple letters and press 'tab'. A list of functions that have names starting with those letters will appear. For example, type `sin 'tab'`. This trick also works for variable names.

Try typing these in and think about what you would expect the function to do.
Here are a bunch of useful functions:

Properties of variables

```
>> length(a)          %gives you the length of the longer dimension
>> size(a)            %gives you the length of all the dimensions of the array
                      back as a n- element array)

>> length(a(:,1))     %gives you the length of the column

>> n=1
>> size(a,n)          %gives you the length of the nth dimension of a
>> length(a(1,:))     %gives you the length of the row

>> a(end)             %gives you the last element of the array
```

Analyzing arrays

```
>> max(a)             %gives the maximum number in each column
>> max(max(a))        %gives the maximum number in whole array
>> a(:)               %linearize an array – in other words make the whole array
                      into one long column
>> max(a(:))          %gives the maximum number in whole array
>> min(a)             %gives the minimum number in each column
```

Rounding numbers

```
>> a=1.5
>> floor(a)           %rounds down to the nearest integer
>> ceil(a)            %rounds up to the nearest integer
>> round(a)           %rounds to the closest integer
```

These commands work on whole arrays, too!

```
>> a=[1:1.1:10]
>> floor(a)          %rounds down to the nearest integer
>> ceil(a)           %rounds up to the nearest integer
>> round(a)          %rounds to the closest integer
```

Making arrays

```
>> n=2
>> m=3
>> a=eye(n)          %makes an array of dimension n where all value are zero
                     except on the diagonal where they are equal to 1.
>> a=eye(m,n)        %same as above except the array in m by n
>> a=zeros(m,n)      %makes an array of zeros m by n
>> a=ones(m,n)       %makes an array of ones m by n
```

3.4 Exercise 1

1. **A** is the 10 by 10 matrix:

58	20	41	21	68	45	60	8	12	23
42	37	30	64	21	4	1	45	45	23
51	78	87	32	83	2	1	44	71	4
33	68	1	96	62	31	19	35	89	7
43	46	76	72	13	1	58	15	27	64
22	56	97	41	20	38	5	67	25	19
57	79	99	74	60	68	36	69	86	84
76	5	78	26	62	9	63	72	23	17
52	60	43	43	37	3	71	47	80	17
64	5	49	93	57	61	69	55	90	99

What will **A(9:-2:3,1:6:5)** return?

2. In a single line of code:
Create **a**, a 10 by 10 array where all values are zero except along the diagonal where all values equal 5. (Try using the functions we've learned; don't just type in all the values).

3. Starting with the **a** array you just created.
You are going to change the values in the sixth and eighth rows.

In a single line of code:
To each element of the sixth row and eighth row of the array **a,** add double the value of the element in the sixth row, six column of array **a**.

4a. Save a new array **b1** as the array **a** to the power of the value in the fourth row, fifth column of array **a**.
4b. Save a new array **b2** as each element of the array **a** to the power of the value in the fourth row, fifth column of array **a**.

5a. What is the average value of each column?
b. What is the average value of each row? (Try using the built in MATLAB function)
c. What is the average value of every third column (1^{st} , 4^{th}, 7^{th}, etc.) of the array **a**?
d. What is the average value of every other row (1^{st} , 3^{rd}, 5^{th}, etc.) of the array **a**?

In less than 10 keystrokes:
6a. Make a new 20 row, 10 column array **c1** by concatenating (joining together) arrays **a** and **b**.
b. Make a new 10 row, 20 column array **c2** by concatenating (joining together) arrays **a** and **b**.

3.5 Basic Command Reference Card

- Some basic syntax
  - Defining a variable: `x = 2`
  - Defining a vector [1 3 5]: `y = [1 3 5]`
  - Defining a matrix $\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}$: `z = [1 2; 3 4]`
  - Suppressing output: `;`
  - Accessing an element in a vector: `y(2)` (*equals* 3)
  - Accessing an element in a matrix: `z(2, 1)` (*equals* 3)
  - Accessing a row/column in a matrix: `z(2, :)` (*equals* [3 4])
  - Accessing elements in a vector that satisfy some condition
    `y(y>2)` *(equals* [3 5]*)*
  - Accessing indices of elements in a vector that satisfy some condition
    `find(y>2)` *(equals* [2 3]*)*
  - Transposing a vector: `y'` *(equals* $\begin{bmatrix}1\\3\\5\end{bmatrix}$*)*
  - Mathematical operators: `+, -, *, /, ^`
  - Element-wise operations: `2.[+, -, *, /, ^]y`
    Note the dot! [1 2 3].*y = [1 6 15]
  - Special numbers (π, ∞): `pi, Inf`
  - Exponential (e^x): `exp(x)`
  - Natural logarithm (ln x): `log(x)`
  - Trigonometric functions: `cos(x), sin(x), tan(x), etc.`
  - Absolute value ($|x|$): `abs(x)`
- Some basic commands
  - Ways to make vectors:
    - Equally spaced intervals: `x = 1:0.5:3`
      (equals $\begin{bmatrix}1 & 1.5 & 2 & 2.5 & 3\end{bmatrix}$*)*
    - All zeros: `x = zeros(1, n)`
      (equals n *zeros)*
    - All ones: `x = ones(1, n)`
      (equals n *ones)*
  - Draw n uniformly distributed random numbers between 0 and 1
    `rand(1, n)`
  - Minimum of vector `min(y)`
  - Maximum of vector `max(y)`
  - Sum of vector `sum(y)`
  - Mean of vector `mean(y)`
  - Variance of vector `var(y)`
  - Size of vector `size(y)`
  - Length of vector `length(y)`

4 Plotting

4.1 Basic Plotting

MATLAB has powerful tools for plotting and visualizing data. In this example, we will plot a sine wave in a separate figure.

Let's first define the sine wave:

```
>> t=0:0.1:9
>> t=t*pi/9*4
>> g ↑  (up arrow key)
```

The up arrow key fills in the last command line starting with the letter b (very useful). If you leave off the 'b', using the up arrow will scroll through the command history.

```
>> g=sin(t)
```

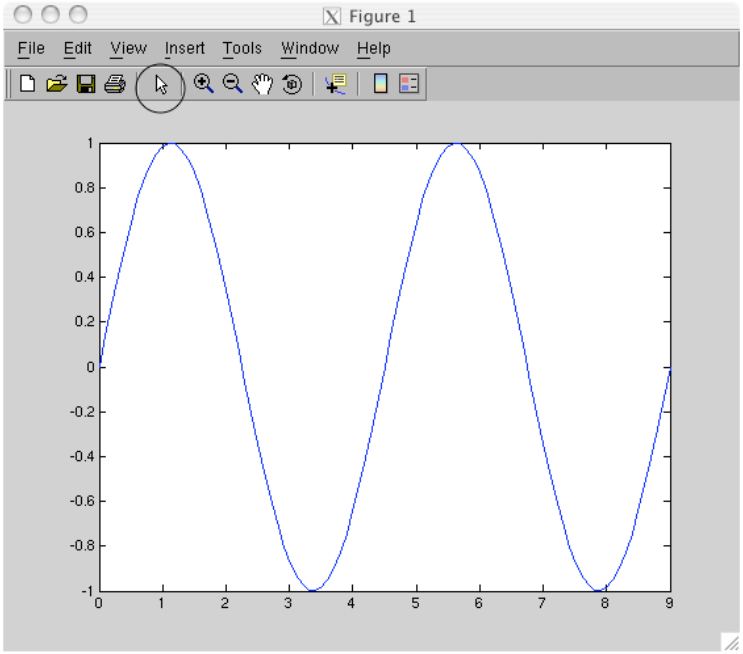
```
>> plot(t, g);
```

Here we are plotting the values of **g** (y-axis) versus the values of **t** (x-axis).

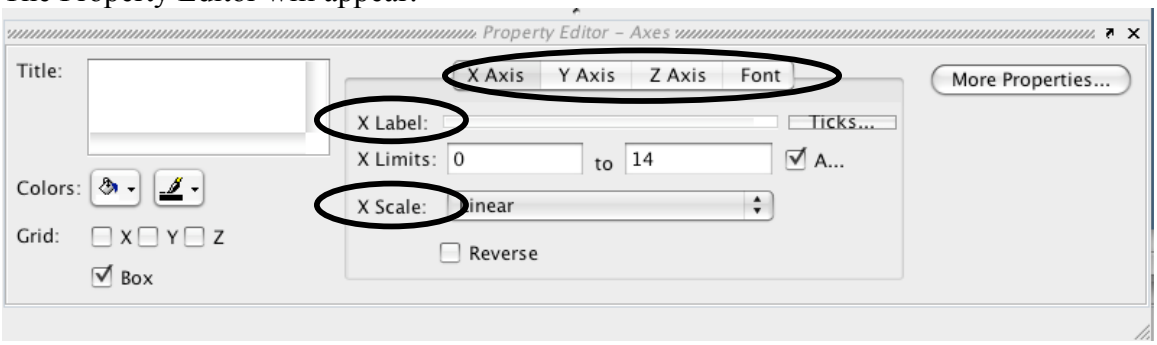
Your version of MATLAB may show something slightly different.

It is probably worth examining the different options for several minutes. Check out the menu options.

To edit the properties of the figure, choose the arrow tool and then double click in the white space of the graph (see below).



The Property Editor will appear:



The three circled areas in the window that will appear are very useful.
The tabs let you change features of each axis.
The scale and label are also circled.
To change what the line looks, like click on it. The Property Editor will now show properties of the line, and you can change things such as the line thickness

```
>> c=sin(t*pi/9*2);
>> plot(t, c);
```

This replaces the first plot.
If you had closed the plot a new plot would have opened.
If there are multiple plots open. This command would plot on the figure that you most recently viewed (not the most recently opened!)

```
>> hold on
```

This will save the first plot when plotting the second

```
>> ↑
```

keep pressing up arrow until:
`plot(t,b)`
appears on the command line

Now use the left arrow key to move the cursor and change the line to read:

```
>> plot(t,g,'gx');
```

To see what this does, read the MATLAB help on plot by typing **`help plot`**.

Making your plot easier for others to understand

```
>> legend('value of function');
>> xlabel('x axis');
>> ylabel('y axis');
>> title('my plot');
```

4.2 2D Plotting

```
>> a=0:9;
>> a=a'*a
>> plot(a);
```

This plotted each column versus their index numbers (i.e. 1:10) as a different line.

Pseudo 3D Plots

Now try:

```
>> figure; imshow(a,[]);
```

You will probably need to increase the size of the figure window to easily see what happed.

In the last line of input above, we've demonstrated one other feature of MATLAB – several commands can be entered on a single line, separated by semi-colons.

```
>> figure; imagesc(a);
>> figure; imshow(a);
```

We will discuss these commands further in the image processing packet.

More 3D Plotting

```
>> figure
>> mesh(a);
```

The **`'figure'`** command opens up a new figure window. Compare the values in the array to the 2D plot. The **`mesh`** command plots the matrix as a 2D surface. The height of each element is given by the value of the array.

```
>> a(10,3)=41;
>> a(8,10)=141;
>> figure, mesh(a);
```

How did this mesh change?

```
>> b=[0 1 2 3 4 5 6 7 8 100]
>> c=b
>> c(10)=20
>> mesh(b,c,a);

>> figure; surf(a);
```

What's the difference?

4.3 Exercise 2

1. Plot an exponential decay (if you don't know the exponential function use the MATLAB help to try and help you find it) in cyan with the points demarked by crosses. (You may wish to use the MATLAB help on the plot command.)

2. Create the following array:
a =

1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1

This should take two lines of code. (Don't just type in the values!)

Plot the array using imagesc.
Plot the array as a surface and rotate it.

3. Label the z-axis.

4.4 Plotting Command Reference Card

`plot(y)`	plot y as the y-axis, with 1,2,3,… as the x-axis
`plot(x,y)`	plot y versus x (must have same length)
`plot(x,A)`	plot columns of A versus x (must have same # rows)
`axis equal`	force the x- and y-axes of the plot to be scaled equally
`title('A Title')`	add a title A Title at the top of the plot
`xlabel('Description of x axis')`	label the x-axis as blah
`ylabel('Description of y axis')`	label the y-axis as blah
`legend('Series1','Series2')`	label 2 curves in the plot foo and bar
`grid`	include a grid in the plot
`figure`	open up a new figure window
`xlim([x1 x2])`	set the x-axis limits to [x1 x2]
`ylim([y1 y2])`	set the y-axis limits to [y1 y2]

To copy figures in MATLAB, go to Edit → Copy Figure. (Ctrl-C won't work.)

5 Writing Scripts and Functions

Sometimes you will want to execute some group of commands repetitively. Instead of typing them over and over again, you can create a script or a function. Both are created in the MATLAB editor, which is a simple text editor.

5.1 Scripts

Scripts are simply a collection of commands, saved in a file (an M-file). Scripts are a good way to write a bunch of commands that you will use over and over again but might want to modify slightly.

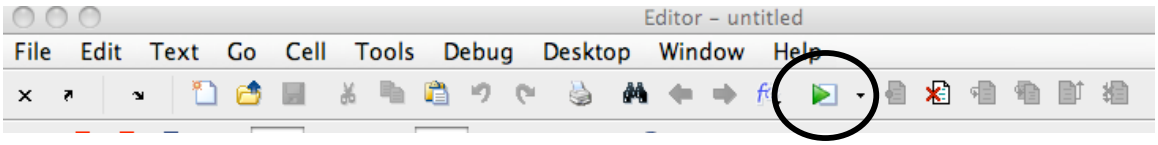
To create a new file, go to File → New → M-file
A new window will appear. You can write commands into this window.

```
a=2+4;
b=6;
c=a*b
```

To save this script go to File → Save As... Save your file on the desktop as "First.m"

One important note about writing scripts and functions: where you save the M-file matters! MATLAB only looks in certain folders to find M-files. So once you have saved your file, you must make sure its folder is included in the MATLAB path. To do this, go to MATLAB window and go to File → Set Path… and then click on "Add Folder…" Find the folder where you M-file is saved, click "Ok" and then click "Close." When MATLAB asks you if you want to save the current path for future sessions, you can click "Close." You must do this each time you save a function in a new folder.

There are now two ways to run the script. You can click on the run button (circled below):



Or you can call the M-file by it's name:

```
>> First
```

This command is case-sensitive – it must be exactly the same as the file name. It is also a bad idea to give the script the same name as a variable or built-in function. For example, if you name your script mean.m, it will run instead of the built-in mean function.

In either case, the program runs all three commands one after the other.
Go back to the "First.m" window and
change the 6 to a **100**

```
>> F↑
```

Adding comments

A good way to make your programs more readable is to add comments.
This helps other people to make sense out of what your code; but equally importantly it allows you to figure out what you did when you look at the code a month later. To mark lines in your script that you don't want to run, place a '%' in front of the line.

```
%This is a comment.
c=3
c+2
%c*c

c =
     3
ans =
     5
```

Here `c+2` will be evaluated, but `c*c` will not be evaluated.

Notice the different color of the line following the %. When the program runs, the green lines won't be executed.

Comments can also be entered in blocks:

```
%{
These lines will be
Commented out.
%}
```

This will comment out all lines between the first and last lines of the above input.

5.2 Functions

What if you want to create a set of code that you can pass an *argument* to? (For example, when we type `sin(2*pi)`, 2*pi is the *argument* to the sine command.) We can do this by writing a function. A function is an M-file that will take some input and return some output (or perform some operation).

Let's create a function.
This function will add two numbers together (similar to the "+" function).

Functions are also created by writing a set of commands in the editor and saving the file as an M-file, but the first line of the M-file for a function is special.

Open a new M-file and type:

```
function val=myfunction(a,b)
a
b
val=a+b;
```

Save this as myfunction.m (on the desktop).

You MUST save the function by the name that follows the equal sign on the first line of the M-file.

As we mentioned above, the first line of the function M-file is special. We first type "**function**" to let MATLAB know this is a function. We then type **val=myfunction(a,b)**. Here **val** is the return variable – this is the output of the function. **myfunction** is the name of the function and **a** and **b** are the function arguments. This will become clearer as we use the new function.

Open a new M-file:
This is the M-file that will call our function.

```
x=1;
y=2;
result=myfunction(x,y);
result
```

Save this as FirstFunction.m (on the desktop).

```
>> FirstFunction
```

So what happens when we type **FirstFunction**?
Lines 1 and 2 are straightforward, so let's look at line three.
How to read a line of code:

Find the equals sign.
Calculate everything to the right of the equals sign.
Store the result in the entry to the left of the equals sign.

When MATLAB hits **myfunction**, it looks for what this might be in this order:

1. Is there a variable named **myfunction**?
2. Is there a file in the current directory (or any other directory in the path) called **myfunction**?
   (The order in which the directories are searched is in the path.)
3. Is there a built-in function called **myfunction**?
4. Gives an error if no match is found.

In this case MATLAB will determine that **myfunction** is a user-written function and will pass it two arguments, the values of **x** and **y** – 1 and 2.

Now myfunction takes the value of **x** and stores it in **a**, and then takes the value of **y** and stores it in **b**. So **a** =1 and **b** = 2. This is because we wrote **myfunction(a,b)** on the first line of myfunction.m.

It then executes all the lines in myfunction, so **val** = 3.

Finally it returns the value of the variable **`val`**. This is because we wrote **`val=`** on the first line of myfunction.m. Therefore, **`return`** will be set to 3.

Understanding the first line of a function:
The first line of the code of the function will always look like this.
1. It will be the word **`function`**
2. Followed by the variable to be returned at the end of the function
3. Followed by an equals sign
4. Followed by the name of the function
5. Followed by the input arguments

Subfunctions

In order to organize your code further, you can declare subfunctions inside a function.

Comment out
```
%val=a+b;
```
from the myfucntion.m code.

At the end of the code type:
```
val=a+mysquare(b);

function c=mysquare(b)
c=b*b;
```

Resave the file.

```
>> FirstFunction
```

5.3 Exercise 3

1. Write a function that takes a 4-number row vector and returns it as 2 by 2 array.
2. Make a function that takes two 4-number row vectors converts them into 2 by 2 arrays (sounds like question 1 doesn't it?) and then performs matrix multiplication on the resulting arrays.

6 Loops and Flow Control

In both scripts and functions, you may want to perform repetitive tasks. Loops can be used to perform repetitive tasks within a program. There are also various "flow control" devices that are useful for structuring programs.

6.1 For Loop

In MATLAB, there are two types of loops, "for" loops and "while" loops. In most cases, a given task can be written in either format, but one is often more convenient than the other. We will start with the for loop, which has three components, the *loop statement*, the *loop body*, and the *end statement*.

Make an M-file

```
a=  [0 52 25 231]
for i = a                    % the loop statement
     i                       % the loop body
end                          % the end statement
```

Save as loops.m (on the desktop).

```
>> loops
```

In this case, the loop statement indicates that the *loop variable* **`i`** will be set, one-by-one with each loop *iteration* (run-through), to the next value in **`a`**. The loop body is located between the loop statement and end statement and is executed once per loop iteration. This loop will iterate four times, one for each value of **`a`**, printing out the current value of **`i`** at each iteration.

Now that our programs are getting more complicated, it is time to become familiar with the debug function, which can help you figure out what is wrong when your code is not executing as expected.

To use the debugger, after saving the M-file, click on the line just to the left of the text in the code. This should create a stop sign. When you run the program now it will stop at this stop sign. You can now step through the program line by line or run it until the next stop sign, by using the buttons in the text editor tool bar (to see what each does, mouse over the buttons). When the program is stopped you can check the values of all your variables (by mousing over the variable in the code) and even perform operations in the command window. This is very useful for debugging but also for learning.

For loop examples

Make an M-file

```
for x = 1:10
     x*x
end
```

Save as loops2.m

In this example, the loop statement is a bit different. The loop runs through the loop body, starting **x = 1** for the first iteration and incrementing **x** by one until **x** is equal to 10.

```
>> loops2
```

In these examples, there is only one line of code in the loop body; however, there is no limit on the number of lines that can appear in the loop body.

Make an M-file

```
a=zeros(10,1);

for x=1:length(a)
     a(x)=x*x;
end
a
```

Save as loops3.m

```
>> loops3
```

The construction of the loop statement above is a very useful one. It lets you run through the values of a variable (in this case **a**), using the loop variable (**x**) as the index (**a(x)**).

Make an M-file

```
y=10;
a=[];
for x=1:y
     a(end+1)=[x*x]
end
a
```

Save as loops4.m

```
>> loops4
```

a=[]; makes an empty array
a(end+1) specifies the entry after the last entry in **a**. In a empty vector, this is **a(1)**. In a 1x1 vector, this is **a(2)**, and so on.

Make an M-file

```
a=  [0 1; 52 2; 25 1; 231 3]
for i = a
     i
end
```

Save as loops1b.m

```
>> loops1b
```

The example above shows that you can loop through an array as well as a vector.

To make the code easier to read, MATLAB's editor automatically indents the loop body. The indentation doesn't affect the code's functioning, but it is useful for understanding code. If your indentation gets messed up, you can have MATLAB indent it correctly by selection the lines of code you want fixed and typing Ctrl-I (or going to Text → Smart Indent.

6.2 If Statement and Logical Operators

The '**if**' statement can be used to control the flow of the program and execute a certain set of instructions only if certain criteria are met. Below we demonstrate its use, along with a set of *logical operators*, operators that check whether certain statements are true.

The logical operator we use in the first example is "**==**", the "is equal" operator. If you write the statement **a==b**, it returns "true" if **a** and **b** are equal and "false" otherwise. Below we use this operator with the **if** statement. The "**==**" is different that the "**=**"! The statement **a=b** simply sets the variable **a** equal to the variable **b**.

We use the mod (modulus) function below. It takes the form **mod(numerator, denominator)** and returns the remainder of numerator/denominator. Type **help mod** for more information.

Make an M-file:

```
a=1:100;
b=[];

for x=1:100
    if (mod(a(x),10)==3) %if statement
        b(end+1)=a(x);   %statement body
    end                  %end statement
end
b
```

Save as loops5.m

In this first "**for if**" loop, we check each element of **a** to see whether the remainder when divided by 10 is equal to 3. If this is true, the element of **a** is inserted into **b**.

```
>> loops5
```

The **if** statement checks whether a condition is true. If it is true, it executes the commands in the statement body. The statement body is located between the **if** statement and the **end** statement. Like loop bodies, statement bodies can contain multiple commands.

In this next example we demonstrate more logical operators. The "**>**" and "**<**" are the greater than and less than operators, and the "**>=**" "**<=**" are the greater than or equals to and less than or equals to operators. They are used just like the equals operator – the statement **a > b** evaluates as "true" if **a** is greater than **b**, and false otherwise.

The "**|**" operator is the "or" operator. If either the statement before the "**|**" or after the "**|**" is true, the statement is considered true.

Make an M-file

```
c=[];
for x=1:100
    if ((a(x)>=91) | (a(x)<9))
        c(end+1)=a(x);
    end
end
c
```

Save as loops6.m

```
>> loops6
```

This example shows two more useful operators. The "**~=**" operator means "not equal." The "**&**" operator means "and". The statement before and after the "**&**" must be true for the statement to be true. With the proper use of parenthesis, multiple "and" and "or" statements can be used to make complicated logical tests.

Make an M-file

```
d=[];
for x=1:100
    if ((mod(a(x),10)~=1) & (mod(a(x),10)<2))
        d(end+1)=a(x);
    end
end
d
```

Save as loops7.m

```
>> loops7
```

6.3 Logical Operators Applied to Arrays

A brief aside – logical operators can also be applied to arrays. Define an array

```
>> m = [0:2:100]
```

Now type

```
>> m2 = m > 50
```

This returns an array of the same length as **m**, where **m2(y) = m(y) > 50**. In other words, each element of **m** is subjected to the logical test, and the result is recorded in **m2**.

A variation on this technique is to type:

```
>> m2 = m(m > 50)
```

In this case **m2** only contains the values of **m** which are greater than 50. MATLAB computes the logical vector **m > 50**, and when this is used to extract elements of **m**, only the elements which correspond to 'true' entries are returned.

The final variation is:

```
>> m2 = find(m > 50)
```

Instead of returning the values of **m** that are greater than 50, the '**find**' command returns the indices of the elements of **m** that are greater than 50.

The **find** command and logical operators are extremely useful! Imagine you have two vectors **x** and **t**. **x** has data points from a time series experiment and **t** has the times of each time point. Imagine you want to find the times when **x > 0**. You can type **t(x>0)** or **t(find(x>0))** to do this. To find out how many data point are greater than 0, you can type **sum(x>0)** or **length(find(x>0))**. The **find** command works a bit differently with arrays than vectors. Be sure to type **help find** to learn more!

6.4 Elseif and Else Statements

The "**elseif**" and "**else**" statements can be used in conjunction with "**if**" statements to create more complicated program structures.

Make an M-file

```
a=[];
b=0;
for x=1:100
     if mod(x,10)==1
          a(end+1)=x;
     elseif mod(x,10)==2
          a(end+1)=x+100;
     else
          b=b+1; %executed if (mod(x,10)~=1)&(mod(x,10)~=2)
```

```
    end
end
a
b
```

Save as loops8.m
The '**a(end+1)=x+100**' is executed if **(mod(x,10)~=1)&(mod(x,10)==2)**.
The '**b=b+1**' is executed if **(mod(x,10)~=1)&(mod(x,10)~=2)**.

```
>> loops8
```

6.5 While Loop

Like for loops, while loops are contain three parts: the loop statement, the loop body, and the end statement. In each iteration, the while loop checks whether the loop statement is true. If it is, it performs the operations in the loop body and again returns to the loop statement to see if it is true. When the statement is untrue, it proceeds to the next bit of code following the end statement.

Make an M-file:

```
x=1;
while (x<100) %loop statement
    x=x*1.3   %loop body
end           %end statement
```

Save as loops9.m

```
>> loops9
```

It is easy to make "infinite" while loops – loops that never end. To avoid this, you need to make sure that the "**while**" statement becomes false at some point. If you accidentally write a program with an infinite loop. Press Ctrl-C to stop it.

6.6 Random Number Generator

Often times, you will need to generate random numbers, e.g. if you want to simulate a stochastic process. The **rand** command creates a random number between 0 and 1. The **rand(m,n)** command generates an m by n array with random numbers. To learn more, type **help rand**. The **randn** command creates a random number normally distributed around 0 with standard deviation of 1. (Type **help rand** for more!)

```
>> A=rand(20,1);
>> A=A*100        %gives random numbers between 0 and 100
>> hist(A)        %creates a histogram of A
```

6.7 Break and Continue Commands

Break statements are a good way to get out of while loops if some set of conditions has been satisfied.

Make an M-file

```
y=round(rand(1,100));
x=0;
for i=1:length(y)
    if (y(i)==1)
        x=x+1;
    else
        break;
    end
end
```

Save as loops10.m

```
>> loops10
```

If you don't want to quit the loop entirely, but instead just want to skip to the next iteration (run-through) of the loop, use the **continue** command.

Replace '**break**' with '**continue**, save the file and run it again to compare the results. What does **x** represent in each version of loops10.m?

As we show above, you can nest MATLAB commands, e.g. **round(rand(1.100))**. This takes the output of **rand(1,100)**, a vector with 100 random numbers, and passes it to **round**. To learn more about MATLAB rounding functions type **help round**, **help floor**, **help ceiling**.

6.8 Exercise 4

1. Make a program that randomly picks a number between 1 and 100.
If the value is between 1-10 have the program output the number
If the value is between 20-30 have the program output the number plus 100
Otherwise output the value minus 100.

2. Make a program that randomly picks a number between 1 and 100.
Use a while loop to find the closest integer less than the random number.
Break out of the loop and print the value (should be equivalent to rounding down).

3. In MATLAB, loops are sometimes necessary, but if possible, code runs faster when loops are replaced by vector operations. Can you re-write the loops in this section in vector form?

6.9 Flow Control Command Reference Card

Logical operators

`==`	equal
`~=`	not equal
`>`	greater than
`<`	less than
`>=`	greater than or equal to
`<=`	less than or equal to
`|`	or
`&`	and
`xor`	exclusive or (returns true if ONLY one of the two statements is true)

Loop control statements

`for`	to start a for loop, e.g. `for i=1:10`
`while`	to start a while loop, e.g. `while (x<10)`
`end`	to enclose a loop body
`break`	exits the loop
`continue`	moves to the next loop iteration

We also learned how to use `if`, `elseif`, and `else` statements in this section.

7 Other Data Types (when you are not just dealing with numbers)

Often we are not dealing with just numbers. For example we might have microarray data where numbers are associated with lists of genes that we also need to track. Alternatively, we may be taking hundreds of images. We might need to keep track of filter set or exposure time in order to properly compare images. In short, arrays are limited for two reasons: (1) there is no way to store text; (2) it is difficult to associate text with array positions (array positions are numbers). There are some good ways of dealing with these problems in MATLAB (and in other programming languages).

```
>> clear;
%This command is useful if you have built up a whole bunch
of items in the workspace and want to start over.  But use
with caution - it erases the values of all the variables in
the workspace

>> [num,txt]=xlsread('MicroArrayData.xls');
```

(You should also take a look at the **cvsread** command.)

If you look at **num** in the workspace it is an array of numbers (doubles).
txt is something different. It is a cell array. This is denoted in the workspace pane by the fact that the icon is {} not the small spreadsheet icon of an array.

Double click **txt** in the workspace so you can look at the array.
It is an array of text!
Individual characters of text (like the letter "a" or "C") are referred to as type **char** (or character). A bunch of char in a row or an array of char is called a **string**.

7.1 Getting Friendly with Strings

```
>> String1='My First String';
```

A new variable has appeared. It also has a new icon in the workspace (this icon means **char**).

```
>> whos String1
```

whos is a useful command that gives you information about a variable. **String1** is a 1 by 15 array of characters.

Strings are not so different from numerical arrays. Try to guess the output of each of these commands before typing them in.

```
>> String1(1:2:end)
>> String1(4:9)
```

When entering text, you must put it inside single quotes. This is so MATLAB know you are entering text, not code (i.e. variable and function names).

Because they are arrays, strings can be concatenated as other arrays are:

```
>> String2=[String1,' and My Second String'];
```

7.2 Interconverting Strings and Numbers

Each character is associated with a specific number (ASCII for those of you who know what this means). Therefore, you can convert between numbers and strings:

```
>> double(String1(4:9))
>> char([65 99 101])
```

You may have noticed that the data type of characters is **`char`**, and the data type of numbers is **`double`**. Why **`double`**, you ask? **`double`** is short for "double precision," as opposed to "single precision." A single precision number take less memory to store, but has a smaller range and precision than a double precision number. The default data type for numbers in MATLAB (and many other programming languages) is double.

Converting between stings and numbers in this way isn't incredibly useful, so let's look at a more useful example.

Let's say you had a variable:

```
>> ImageNumber=10;
```

This number might be useful in the title of a figure.

```
>> figure,peaks(31)
>> title(['Image Number ',ImageNumber])
```

This won't work correctly because ImageNumber is a double, not a string, and you cannot concatenate different types of data together in this way.

```
>> title(['Image Number ',num2str(ImageNumber)])
```

The **`num2str`** command is the command that will convert a number into the corresponding text character.

Note the difference between:

```
>> num2str(1)
```

and

```
>> char(1)
```

There are a bunch of useful functions that can be used with strings. I recommend reading about **`strfind`** and **`strcmp`** (and **`regexp`** if you already can guess what that function will do).

7.3 Structures and Cell Arrays

We started our packet with basic data structures – scalar variables (e.g. **x = 2**) and arrays. As problems get more complicated, arrays and scalars may not be sufficient for your needs. For example, for a particular data point, you may want to store a variety of pieces of data in a more unified way. The choice of a good data structure can simplify one's "algorithmic life." We will review two more data structures below.

Structures

A structure is a MATLAB array that divides its contents up into fields. The fields of a structure can contain data types different from one another, and this is the main reason for their use. A few examples will illustrate how they are defined and used.

```
>> S.name = 'Jason Knapp';
>> S.age = 22;
>> S.egr_335_grade = 'A';
>> S.grade = 'senior';
```

Here, our structure is **S**, and the fields are **name**, **age**, **egr_335_grade**, and **grade**. Now that we have defined a structure, we need to know how to access the information contained within its fields. This structure contains multiple types of data: strings and a number. This would not have been possible with a single array. Furthermore, instead of having to remember what column the name or age was stored in as one would in an array, we can now refer to the name, age, etc. in a much more natural way.

```
>> fieldnames(S)
```

This lists all the fields associate with a structure. To access the values of a field we use the . operator:

```
>> S.name
ans =
Jason Knapp
>> z = S.age*2
z =
44
```

To reiterate: structures can contain data of any type, arrays, strings, other structures, and **cell** arrays. A useful function when creating structures in MATLAB is the **struct** command. Type **help struct** to lean more about it and several other commands that can be used with structures.

To add another name:

```
>> S(2).name='John Smith';
```

What happened?

```
>> S
```

```
>> S.name

>> S.age

>> S(1)

>> S(2)
```

A second entry in the structure was made, in which every field is defined (though they are mainly empty).

```
>> S(2).age='Unknown';

>> S.age
```

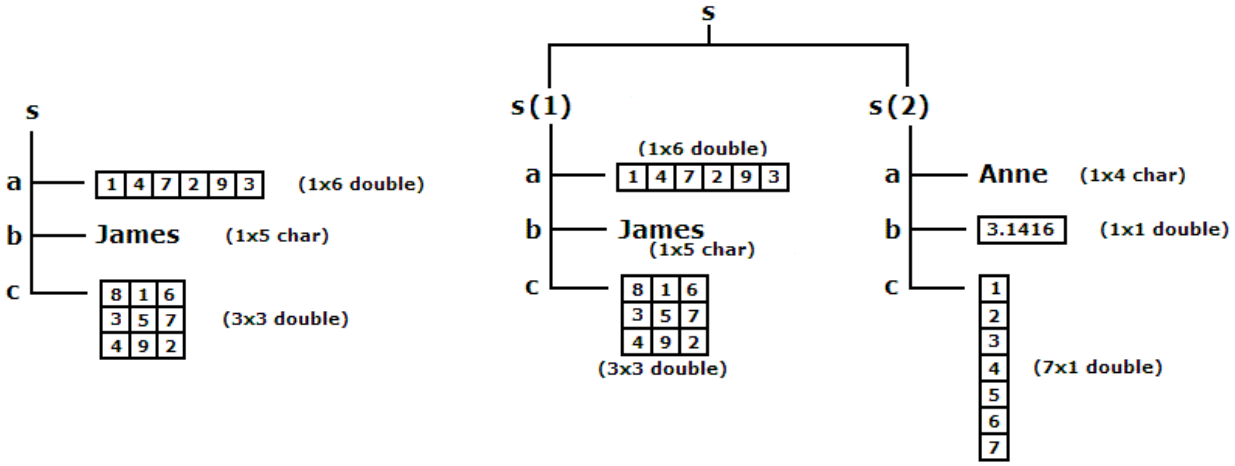
Now we have different types of data under the "age" field.

```
>> S(2).NewField=[3,4;12,4];

>> S(1)

>> S(2)
```

When a structure has a single entry, assignments like **`S.name`** are shorthand for **`S(1).name`**. However, when your structure has multiple entries, this shorthand no longer works.

Addressing entries in structures is just like accessing elements in array – things like **`S(end)`** and **`S(1:3)`** are perfectly legal. These can be combined with field specifications, e.g. **`S(1:2).name`**. In general, many of the commands available for arrays, e.g. **`size`**, **`length`**, have analogs for structures.

The figures below, taken from the MATLAB documentation, illustrates the structure **`S`**, with fields **`a`**, **`b`**, and **`c`** with one and two entries.

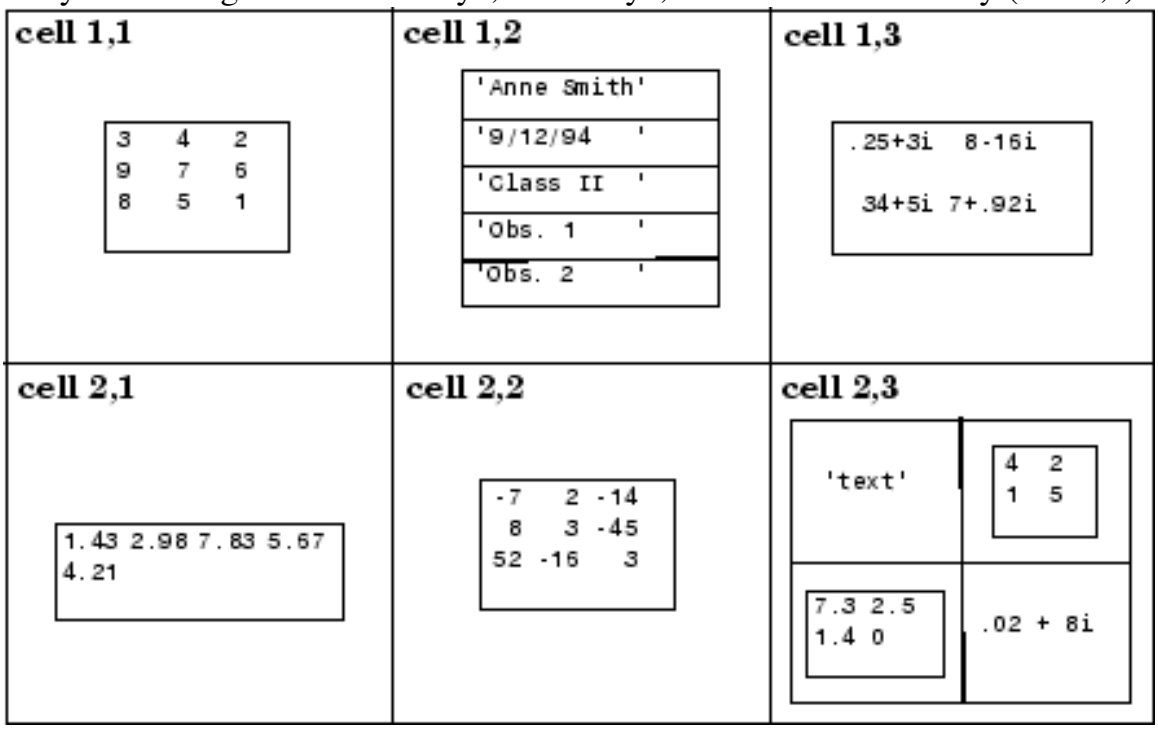


Cell arrays

`Cell` arrays are MATLAB arrays of containers. Each container can hold data of any type, but unlike structures that store their data in fields, **`cell`** arrays store their data as actual elements. An example will illustrate their use.

```
>> a{1, 1} = ones(3,1);
>> a{2, 1} = 'This is second row and first columns';
>> a{2, 2} = randn(3) ;
>> a{1, 2} = 'row 2 column 2 is a 3x3 random matrix';
```

The figure below, taken from the MATLAB documentation, shows a 2 x 3 cell array containing numerical arrays, text arrays, and a nested cell array (cell 2,3).



With **`cell`** arrays, when assigning elements use the curly brackets instead of parentheses. Here is how to access the elements of a **`cell`** array:

```
>> tmp=a{1, 1};
>> tmp2=a(1, 1);
```

Let's look at the difference in the classes:

```
>> whos tmp2
>> whos tmp
```

The {} returns what's in that cell.
The () returns the cell itself.

```
>> subcell = a(1: 2,2)  %Note the use of( ) instead of { }

subcell =
'row 2 column 2 is a 3x3 random matrix'
[3x3 double]

>> X = 3*a{2,2}

X =
0.5239
-1.7649
0.3418
-0.5601
6.5496
3.2003
2.1774
-0.4092
0.1778

>> a{1,2}

ans =
row 2 column 2 is a 3x3 random matrix
```

Now that we have learned about both cells and structures, we can use them together. Try:

```
>> f = fieldnames(S)
>> whos f
```

Note that **f** is a cell array of the field names in the structure **S**. (This allows for the fact that field names are different lengths!)

Now, if we want to inspect the values of the structure, by field name we can use this loop:

```
>> for CurrentField=f'
>>   S.(cell2mat(CurrentField))
>> end
```

We enclose **(cell2mat(CurrentField))** in parentheses so that MATLAB knows that it must evaluate this expression to get the field name. The general format is **StructureName.(VariableFieldName)**. We use the **cell2mat** command to convert the cell container into a matrix since a field name can't be type **cell**; it must be type **char**.

7.4 Parsing data and text in figures

Let's return back to our microarray data.

When we loaded it, we got two arrays: **num**, an array of doubles and **txt**, an array of cells.

If we look at **txt** we can see there is actually a header of three cells and then the info we want in the rest of column 1.

This will happen often. You will get data in one format and then need to extract it into some other useful format. This process is referred to as parsing and can take a significant amount of time to get right. It is often useful to write a program to automatically do the parsing for you.

Here is an example of a function that will parse this data.

```
function [ S ] = ParseXLSArrayData( num,txt )
%ParseXLSArrayData converts imported array data into a useful structure
%   Creates a structure S
%This structure has three fields
%The first field is the element in the header of the first text column.
%The second field is the rest of the header and is called header.
%The third field is called data and is an array of the microarray data.

S.header=txt(1,[2,3]);
S.(txt{1,1})=txt(2:end,1);
S.data=num;

end
```

```
>> S=ParseXLSArrayData(num,txt);
```

And here is a function that makes a nice figure. Take some time to study and try to understand this and the last function. It combines together a number of the concepts we've covered in this packet.

```
function PlotSpeciesComparison( S , cutoff,listnames)
%PlotSpeciesComparison
%Plots the species data in the first two column of data versus each
other.
%Adds names to the plots based on the cutoff of required diffrence
between
%the values.
%If listnames is not 0 list names on plot

figure,plot(S.data(:,1),S.data(:,2),'.','Color',0.75*[1 1 1])
%The array following color is the amount of [red green blue] in the
color
```

```
%The numbers can be between 0 and 1.  All 0 is black.  All 1 is white.
hold on
SignificantORFs=abs(S.data(:,1)-S.data(:,2))>cutoff;
%In this case the array data is assumed to be in log scale already
plot(S.data(SignificantORFs,1),S.data(SignificantORFs,2),'.','Color',0.
25*[1 1 1])
if listnames~=0

text(S.data(SignificantORFs,1),S.data(SignificantORFs,2),S.ORF(Signific
antORFs))
    xlabel(S.header(1));
    ylabel(S.header(2));
end

end
```

```
>>  PlotSpeciesComparison( S , 3,0 )

>>  PlotSpeciesComparison( S , 3,1 )
```