MATLAB Basics

WDRP - Simple Discrete Models in Biology and MATLAB

Contents

1 Defining Variables and Operations

MATLAB allows storage of values and strings into *variables* by using the "=" sign.

```
>> x = pi;
>> r = 'Hello World';
>> x

x =

    3.1416
```

Note that variables don't need to be single letters like we usually think with "x" and "y" in math.

```
>> SinVal = sin(pi);
>> Password = 'OneTwoThreeFourFive';
```

Often times, it is actually better to use these longer variable names, as it makes it's purpose clear.

Moreover, variables can be "self referenced", that is, we can write something like the following

```
>> SinVal = SinVal + 1

SinVal =

    1.0000
```

Of course, when it comes to a mathematical equation, we can't have something like $x = x + 1$, otherwise $0 = 1$. But to a computer, there is an order to how it performs operations. So, the sum comes first, and THEN it assigns the value back to SinVal.

1.1 Mathematical Operations

MATLAB allows us to perform pretty much all the usual math operations you can think of. Moreover, they use the exact kinds of symbol that you would expect. See the example on the next page.

Remark. Notice that there is an "ans =" whenever there is no semicolon. This is because if you want to suppress an output, you must enter a semicolon at the end of that line. It is good practice to use semicolons at every line and instead display the output using a *print* command, which we will explain later.

```
>> 1+1

ans =

     2

>> 3*5

ans =

    15

>> 5-x;
>> SinVal/2;
>> 2^5

ans =

    32
```

We also have the "mod" operation. This takes two numbers, the first is divided by the second and then the output is the remainder. You actually don't even need to use integers, decimals and fractions work as well.

```
>> mod(3,2)

ans =

     1

>> mod(2,1.5)

ans =

    0.5000
```

1.2 Function Handles

Function handles are exactly like the usual functions in math, such as $f(x) = e^x$. These are essentially used to "store operations" so you don't have to keep trying to write the same computations with different numbers. To do this,, you are still defining a variable in the usual way, but before you write the expression with the operations, you have to indicate to the computer what your variable is by using $@(x)$ where x is the variable you will use. Of course, you are not limited to only using x as a variable.

```
>> f = @(n) n^2+n-1;
>> Gfunc = @(t) t*sin(pi*t/2);
>> f(2)-Gfunc(pi)

ans =

    8.0642
```

You also do not have to limit yourself to a single variable, you can use multiple variables.

```
>> h = @(x1,x2) x1^2+x1*x2-x2^2;
>> h(2,1)

ans =

     5
```

2 Arrays

It can be argued that the reason we are doing coding/MATLAB at all is because of arrays. Arrays may be as fundamental to the work we will do as the simple operations we've seen.

So what is an array? An array is a collection of rows and columns of entries. These entries are most often numbers, but just like how we could store strings in variables, we can do the same for the entries of an array.

To make an array, there are many ways. Commonly, they all use brackets "[ ]". Entries are distinguished by adding a space between them, as in the examples below.

```
>> t = [1 2 3 4 5];
>> r = [[1 2 3]; [4 5 6]]

r =

     1     2     3
     4     5     6
```

Below, the first command

[min : step : max]

gives a single row array with entries starting at the "min" ending at the "max" and the other entries are the in between spaced by the "steps". The command

arrayfun(function, array)

takes in a function handle and an array, and outputs the value of the function on each entry of the given array in an array the same "shape" as the given one. If instead of using v, we used r, the output would have 2 rows and 3 columns (the same as r).

```
>> v = [1:1:10]

v =

     1     2     3     4     5     6     7     8     9    10

>> u = arrayfun(f,v);
>> u

u =

     1     5    11    19    29    41    55    71    89   109
```

The importance of arrays is both for storing data and for performing computations; although we will pretty much only do the prior since we won't use any linear algebra. But to use this stored data we will want to be able to do some operations on these arrays

2.1 Operations on Arrays

We can do a lot of similar operations to arrays as we did with numbers for example

```
>> 3*v

ans =

     3     6     9    12    15    18    21    24    27    30

>> v.^2

ans =

     1     4     9    16    25    36    49    64    81   100

>> u+v;
>> u./v

ans =

  Columns 1 through 7

    1.0000    2.5000    3.6667    4.7500    5.8000    6.8333    7.8571

  Columns 8 through 10

    8.8750    9.8889   10.9000
```

Its important that when squaring, like in the second example, we use a period before the operation. This lets the computer know to do this operation entry-wise.

Similarly, fo r division, we must put a period so that we divide entry-wise. In particular, the first entry of u is divided by the first entry of v and so on. Otherwise, we get something super unusual. Try it!

2.1.1 Concatenation

We can also put together arrays in other ways than computationally. In particular, by "concatenation" both by rows and columns.

Below, w is v stacked on top of u, while r is v followed by u, in the same row. The last one just tacks on 11 to the end of v.

```
>> w = [v; u]

w =

     1     2     3     4     5     6     7     8     9    10
     1     5    11    19    29    41    55    71    89   109

>> s = [v u];
>> ss = [v 11]

ss =

     1     2     3     4     5     6     7     8     9    10    11
```

This begs the question, what happens if the rows or columns don't really "match". Recall that w is v stacked on u, so there are two rows in w. If we try to tack on an 11 to w, we will get an error.

```
>> www = [w; 11];
Error using vertcat
Dimensions of arrays being concatenated are not consistent.

>> ww = [w 11];
Error using horzcat
Dimensions of arrays being concatenated are not consistent.
```

These both give errors. So we are only allowed to concatenate when the dimensions (ie. rows and columns) of the two arrays match. So something like the following works.

```
>> wwww = [w [11; 12]]

wwww =

     1     2     3     4     5     6     7     8     9    10    11
     1     5    11    19    29    41    55    71    89   109    12
```

2.1.2 Extraction

Now, we know how to put two arrays together/ add entries to make a bigger array. So, what about taking out entries / smaller arrays? To pinpoint a specific entry we need to know its "location" in the array, namely the row and column (in that order). Remember that v had the numbers from 1 to 10 in the first row, so we can do something like

```
>> v(1,1)

ans =

     1

>> v(1,3)

ans =

     3
```

And for w which has v in the top row and u in the second we can do

```
>> w1 = w(2,4);
>> s1 = s(4,2);
Index in position 1 exceeds array bounds. Index must not exceed 1.
```

Notice that the second one didn't work because we tried calling an entry from the fourth row, when r only has one row. We can also just give a single number and that will also call an entry.

```
>> s2 = s(4);
>> w2 = w(5);
>> w3 = w(6);
>> w4 = w(7);
```

Giving only one number gives that entry by counting the rows, starting with the first column. This means it starts at the top left and goes down through the columns of the array. Moreover, we can only put numbers in up to # of rows $\times$ # of cols.

In addition, rather than extracting a single entry, we may extract a collection of entries. Here is the setup for an array a

$$a([\text{*array of rows*}], [\text{*array of columns*}])$$

So for example,

```
>> r = [[1 2 3];[4 5 6];[7 8 9]]
r13 = r(1:3,[1, 3])

r =

     1     2     3
     4     5     6
     7     8     9


r13 =

     1     3
     4     6
     7     9
```

Notice, for the rows, we can just use $1:3$ which creates the array [1 2 3]. We used this before to define the array v, but before we indicated the step size. If the step size is not indicated it defaults to a step size of 1.

2.1.3 Insertion

We can extract a particular entry of an array, how about *inserting* an entry into an array. Now, this is closer to replacement than insertion since we can't just add in a new entry that didn't already exist. To do this, we use extraction in some loose sense, and reassign the extracted entry to a new one. Here is a general set up for an array a.

$$a(i,j) = \text{*new entry*}$$

where i, j are the row and column of the entry. Also, just like before with extraction, we can insert multiple entries at once using the setup

$$a([\text{*array of rows*}], [\text{*array of columns*}]) = \text{*Array of new entries*}$$

Here is an example:

```
>> r = zeros(3,3)

r =

     0     0     0
     0     0     0
     0     0     0

>> r(1,2) = 1

r =

     0     1     0
     0     0     0
     0     0     0

>> r(1:3,[1, 3]) = ones(3,2)

r =

     1     1     1
     1     0     1
     1     0     1
```

3 Some Useful Operations

In case we forget how many rows, columns or numbers of entries an array has we can use the following commands

3.1 Array commands

1. length(*array*) gives the number of entries in an array
2. size(*array*) gives the number of rows and number of columns of an array (as an array ie. it gives [rows cols]).
3. zeros(i,j) creates an array of all zeros with i rows and j columns
4. ones(i,j) creates an array of all ones with i rows and j columns

Another helpful command is the following, which gives a random whole number between the maximum and minimum listed.

3.2 Random integer

To generate a random integer between a minimum value and a maximum value use

randi([min, max])

Additionally, we can create arrays of random integer entries by also specifying the size of the array as follows

randi([min, max], [i,j])

where i is the number of rows and j is the number of columns.

3.3 Standard functions

The last ones are some functions, sin we have already seen.

1. exp(-) is the exponential function e^x
2. log(-) is the logarithmic function
3. sin(-)
4. cos(-)

3.4 User Input

Something else we can do is ask a user for input, whether it's a number, a string, an array or even a function handle.

input('string to be displayed goes here')

so for example

```
>> EvenNum = input('Please input an even number: ');
Please input an even number: 3
>> EvenNum

EvenNum =

     3
```

the command window will wait until input from the user has been given. This is useful when you want to just call a script and not have to play with the numbers in the code itself.

3.5 Printing

Now we can finally say how to properly display outputs from our scripts. To do this we use

fprintf('String goes here')

```
>> fprintf('Hello World \n');
Hello World
```

the \ n is important for ending the line of text so that the >> appears on the next line instead of right after the displayed text. Try it without the \ n and see what happens!

We can also display numbers by using %f in the string part and then indicating the variable/numbers to be displays. For example

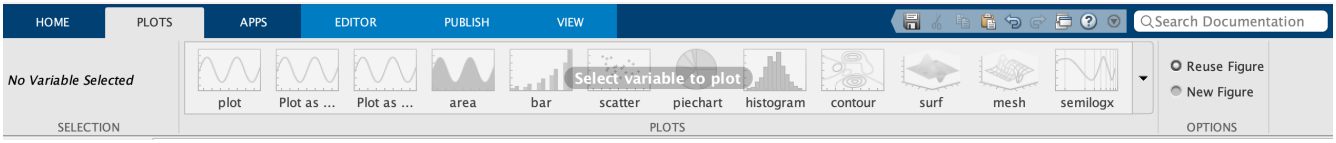
```
>> fprintf('We can use it to show numbers like %f \n', pi);
We can use it to show numbers like 3.141593
>> fprintf('Or we can show strings, like %s \n', Password);
Or we can show strings, like OneTwoThreeFourFive
```

Additionally, we can specify how many numbers and decimal places to display, for example, %3.4f would display 3 numbers before the decimal place and 4 decimal places.

Remark. Normally ; suppresses outputs, but fprintf will still display what you want it to even if you put a ;.

4 Plotting

Graphical representations of data are extremely useful. Luckily, MATLAB can make plots, and the program itself makes it easy to use different kinds. At the top of the MATLAB window there is a tab for "PLOTS". You'll see that it has lots of different options for types of plots. It asks you to first pick a variable to plot, all of the available variables can be found to the right in the small window labeled "Workspace".



Feel free to play around with these different tools, but we will only discuss the basic plot function in this section.

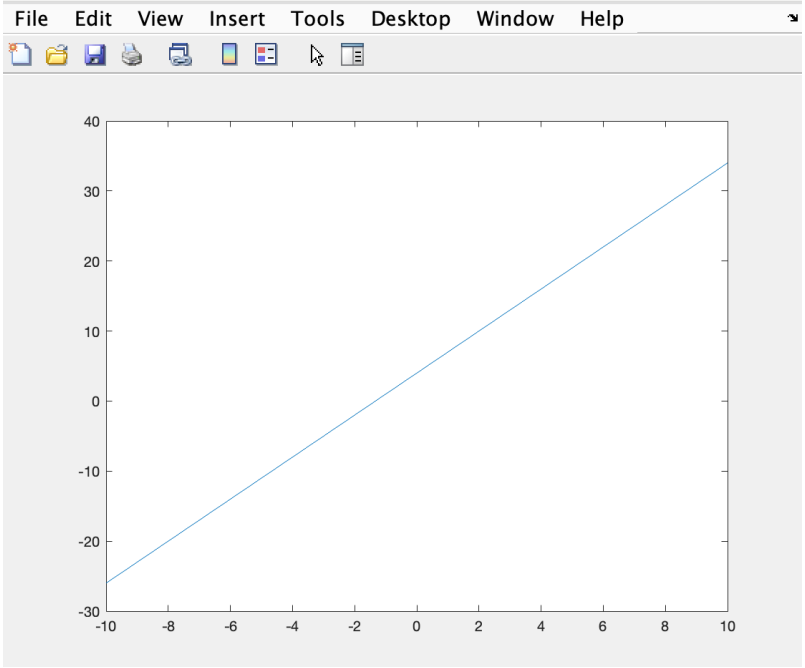
So plotting follows this setup

plot(X,Y, '*color* *line style* *marker style*','MarkerSize', *marker size*)

The X is an array containing the x coordinates of the points you want to plot. The Y is then the y coordinates of the points. For example,

```
>> u = -10:1:10;
v = 3*u+4;

plot(u,v)
```



Notice the plot still worked without including all the other stuff. It defaults to this blue with no markers in case you don't specify. If we do want to specify, we include them in the order as above, all together in the apostrophes. You can see a couple of examples on the next page.

To see what kinds of line specs are available, you can view them on the MATLAB website. The same page has plenty of examples and extra details if you're interested in reading further.

Some Key things to notice about the examples. In the first, we were able to plot two curves using only one plot command. In general we can actually use

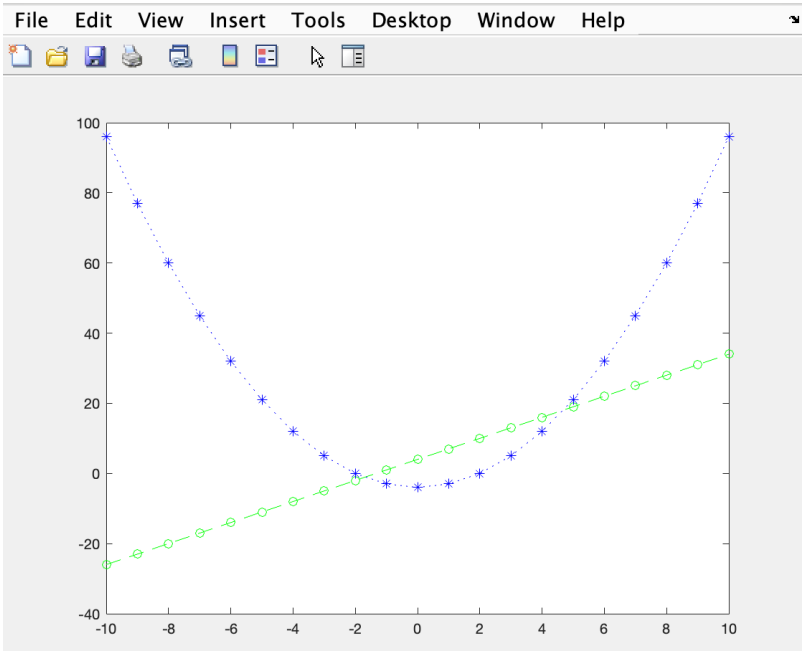
plot(X1,Y1,'LineSpec1',X2,Y2,'LineSpec2',...Xn,Yn,'LineSpecn')

But, if you want to change the marker size I believe you can't use a single plot command. That is why the second example has two separate plot commands. Moreover, in the first script of the second example we used "hold on". This tells the computer to put both plots on the same figure.

Alternatively, we can also write "figure" before each plot command and get two different figures. This is the second script.

```
>> u = -10:1:10;
v = 3*u+4;
w = u.^2-4;

plot(u,v,'g--o',u,w,'blue:*');
```

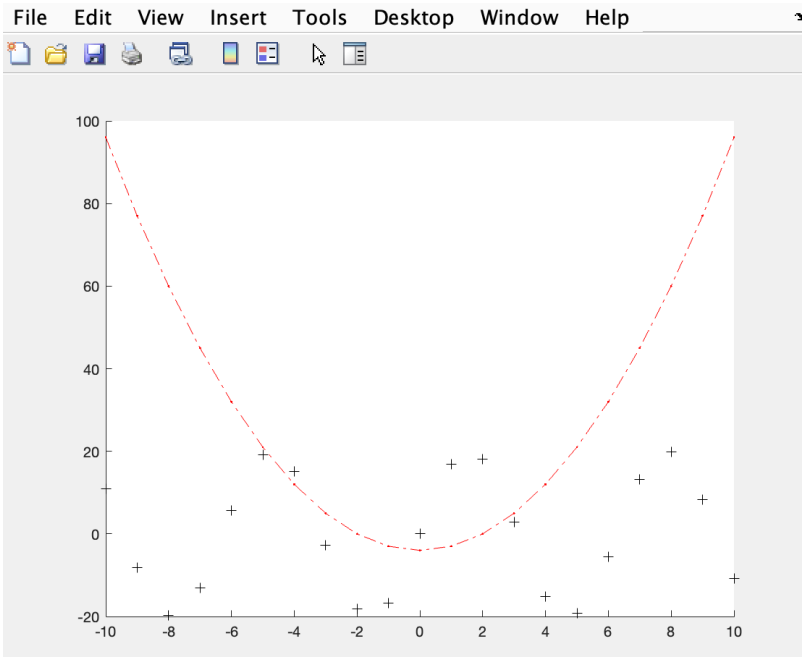



```
>> u = -10:1:10;
v = 20*sin(u);
w = u.^2-4;

hold on
plot(u,v,'Black+','MarkerSize',5)
plot(u,w,'r-..','MarkerSize',0.5);
```

```
>> u = -10:1:10;
v = 20*sin(u);
w = u.^2-4;

figure
plot(u,v,'Black+','MarkerSize',5)
figure
plot(u,w,'r-..','MarkerSize',0.5);
```



5 Logical Operators

Although computers don't actually "reason" like humans do, logic is mathematical and we can tell the computer how to calculate "reasoning". This is done by logic operators. Before we can actually get to the operators, we should understand how the computer calculates "reasoning". That is, by truth tables.

5.1 Truth Tables

In math we assign a "truth value" to any given statement. For example, the statement "two is even" has a truth value T, while the statement "one is prime" has a truth value of F. So, given any statement, its truth value is either T or F. So what are the possible combinations for two statements? Given two statements P and Q we have the following

P	Q
T	T
T	F
F	T
F	F

5.1.1 AND

Now, we can ask "What is the truth value of both P AND Q?". For example, what is the truth value of the following statements?

"two is even AND two is odd"

Although the first statement is true, the second is false. This means the $\wedge$ of the two statements is also false.

Remark. As a quick note, most of the time you'll see the notation $\wedge$ as a replacement for the word "AND", which we will stick with that going forward.

"three is odd AND three is prime"

In this case, both statements are true, so the $\wedge$ of the two statements is true.

Here is the truth table for the AND of two statements.

P	Q	P $\wedge$ Q
T	T	T
T	F	F
F	T	F
F	F	F

Thus, the truth value of P $\wedge$ Q is F if at least one has truth value F. It only has truth value T if both P and Q are true.

5.1.2 OR

Now, we ask "what is the truth value of either P OR Q?". We denote "OR = $\vee$". Consider

"Two is even or two is odd"

Indeed, two is even, so we are correct in saying, it's either even or it's odd. So we get the following truth table

P	Q	P $\vee$ Q
T	T	T
T	F	T
F	T	T
F	F	F

So $\vee$ is sort of the opposite of $\wedge$, as $P \vee Q$ has truth value T whenever at least one of them has truth value T and has truth value F only when both have truth value F.

5.1.3 NOT

The last question we can ask is not so much a question, but more like altering a given statement. "What is NOT P?". We denote "NOT = ". Consider

"two is not even"

Since the truth value of the statement "two is even" is T, the truth value of the is the opposite, F. So negation just "flips" a statements truth value.

5.1.4 Thinking like a Computer

So, how do computers "think" when it comes to logic? The computer keeps the truth value of T as 1, and F as 0. So the previous tables look more like this to a computer

P	Q	P $\vee$ Q	P $\wedge$ Q	$\sim$ P
1	1	1	1	0
1	0	1	0	0
0	1	1	0	1
0	0	0	0	1

Computers then take your statements, like $3 > 4$, and assigns a truth value and can use that to perform calculations.

5.2 (In)Equality Operators

To check equality we can ask the computer using ==. So, for example,

```
>> 1 == 0
ans =
  logical
   0
>> 0 == 0
ans =
  logical
   1
```

We can also ask if two things are not equal

```
>> 0 ~= 0
ans =
  logical
   0
>> 0 ~= 1
ans =
  logical
   1
```

5.3 Comparison Operators

Checking equality is a strong condition, so sometimes we need something a bit weaker, namely greater than (>) and less than (<).

```
>> 0 > 1
ans =
  logical
   0
>> 0 < 1
ans =
  logical
   1
```

there is also greater(>=)/less(<=) than with equality

```
>> 0 >= 1

ans =

  logical

   0

>> 0 <= 1

ans =

  logical

   1
```

5.4 And/Or Operators

Now that we can extract truth values from statements, we can perform operations like $\wedge$ (&&) and $\vee$ (||). For example,

```
>> 0 < 1 || 0 > 1

ans =

  logical

   1

>> 0 < 1 && 0 > 1

ans =

  logical

   0
```