14:440:127– Introduction to Computers for Engineers

Notes for Lecture 02

Rutgers University, Spring 2010 Instructor- Blase E. Ur

1 Built-In Math Functions

Matlab includes many built-in functions for math operations. Here are a number of the most important ones. For these functions, you are specifying input arguments (i.e. 5 is the input argument in the first example). If you have multiple arguments, separate them by commas, as in the second example:

```
sqrt(5)    % square root of 5
nthroot(27,3)  %cube(3rd) root of 27
sin(pi)       % sine of pi radians
cos(pi/2)       % cosine of pi/2
asin(1)       % arcsine of 1
sind(75)      % sine of 75 degrees
abs(-5)     % absolute value of -5
log(5)   % natural logarithm (base e) of 5
log10(5)  % logarithm (base 10) of 5
exp(5)   %  e^5
```

Note that you can use these basic functions for more complicated examples i.e. to calculate $\log_5 25$, just do a change of base with the command `log(25)/log(5)`.

```
round(5.3) % round 5.3 (.5 or greater rounds up)
fix(5.3)  % round towards 0
floor(5.3) % round towards -inf
ceil(5.3) % round towards +inf
```

```
rem(15,2)    % remainder of 15/2
mod(15,2)    % similar to rem
    % but can give negative (congruent) answers

sign(x)  % 1 for x>0, 0 for x=0, -1 for x<0

factor(15)  % returns a vector of the prime factors of 15
gcd(15,20)  % the greatest common divisor
lcm(3,7)  % least common multiple
factorial(15)  % 15!
primes(100)   % lists all primes <= 100
isprime(101)  % 1 (true) or 0 (false) 101 is prime
```

1.1 Function Examples

```
sqrt(9)    % ans=3
nthroot(81,4)  %ans=3
rem(20,3)    % ans = 2
mod(20,3)    % ans = 2
rem(20,-3)    % ans = 2
mod(20,-3)    % ans = -1
primes(10)   % ans = 2 3 5 7
isprime(23549)  % ans = 1
```

```
round(5.3) % ans = 5
fix(5.3)  % ans = 5
floor(5.3) % ans = 5
ceil(5.3) % ans = 6

round(5.6) % ans = 6
fix(5.6)  % ans = 5
floor(5.6) % ans = 5
ceil(5.6) % ans = 6

round(-5.6) % ans = -6
fix(-5.6)  % ans = -5
floor(-5.6) % ans = -6
ceil(-5.6) % ans = -5
```

1.2 Help

If you type *help* followed by the name of a Matlab command, you'll get that command's help file. i.e. *help round*

This is very useful if you want to see how a function works or what type of input it expects.

2 Comments

Whenever you begin a line or part of a line with a percent sign (%), everything to the right of the percent sign on that line is ignored by Matlab. This is known as a comment. In order to document your Matlab code or remind yourself to do things, you can and should insert comments i.e.

```
stuff = stuff * 10; % adding a zero
% remember to take the absolute value
```

Comments are also a very useful tool for debugging (fixing problems with) your programs. When you want to eliminate a few lines of code temporarily, you can 'comment them out' rather than cutting and pasting them into and out of your program. You can also make multiple lines part of a comment:

```
{%
both this line
and this line are a comment
%}
```

3 Inputs and Outputs

3.1 Input

So far in this class, the main variability in the programs you've written has come from you, the programmers, by changing the value of variables you've set. In the real world, you'll usually want to get input from the user, the person who is interacting with your program. For example, if you designed and built an ATM in Matlab, you'd ask the customer how much money they'd like to withdraw.

Getting user input in Matlab uses the *input* function. The only argument (input) you need to provide to this function is a string of characters (text surrounded by single quotes) that will be displayed to the user of the program, telling them what to do. Whatever the user types is returned to your Matlab program as the result of this function. Most of the time, you'll want to store this result in a variable. For instance, the following code solves our ATM problem. It displays the text 'Please enter how much to withdraw' to the screen, waits for the user to type something and hit enter, and stores the user's answer in a variable called *money*.

```
money = input('Please enter how much to withdraw: ');
```

If you want the user to be able to type text rather than numbers, include a second input argument to our *input* function: `'s'`:

```
name = input('Type your name: ','s');
```

Otherwise, if the user types in letters, Matlab will assume that they are referring to a variable rather than entering text.

3.2 Output Formatting- disp, fprintf

In the last lecture, we saw the difference between the *disp* command (just display the value of a variable) and leaving out the semicolon (display the result of the line).

disp(x) would display: '5' whereas x would display 'x = 5' and *x;* would display nothing at all.

For more advanced formatting, you can use the *fprintf* formatted print function. The key idea of *fprintf* is that you can specify carefully formatted, static text but insert values from variables in its midst. *fprintf* requires two arguments: 1) a 'formatting string' that specifies the formatting and where variables should be inserted and 2) the variables you want to insert, separated by commas

The formatting string is contained in single quotes. When you want to reference a variable, you can just say `%f` (fixed point/floating point/decimal) or `%s` (a string of multiple characters). In other words, use `%f` if you want to insert a number or `%s` if you're inserting a string of characters.

Let's say you had a variable x that stores a number and a variable *name* that stores a string of characters. If we wanted to print out both in one line, here's what we could do:

```
name = 'Jones';
x = 14.7;
fprintf('Mr. %s has %f Matlab books.\n', name, x)
```

Note that the `%s` corresponds to the first variable listed (*name*) and `%f` corresponds to the second variable listed (x). Therefore, the order in which you list the variables is quite important! Also, notice the `\n` at the end of the format string. This is called the newline character and it tells Matlab to skip to the next line, which Matlab *does not* do automatically when using *fprintf*. The newline `\n` character works just like a standard letter, which means that `'H\nE\nL\nL\nO'` displays each letter on a separate line.

In sum, the preceding example displays 'Mr. Jones has 14.700000 Matlab books.' To specify how many digits to display after the decimal, you will slightly modify the "`%f`" placeholder to something like '`%.1f`', which indicates that 1 digit must be displayed following the decimal point. Note that integers have 0 digits following the decimal and thus could be displayed with '`%.0f`', which uses the *round* function to shorten a number:

```
name = 'Jones';
x = 14.7;
fprintf('Mr. %s has %.0f Matlab books.\n', name, x)
```

This example would display 'Mr. Jones has 15 Matlab books.'

In a slightly advanced example, note that if you use a format string and then have a vector or matrix stored in the corresponding variable, Matlab will effectively repeat the format string for each value in the vector or matrix.

4 Conditional Statements

A very important part of computer programming is the ability to make decisions based on whether things are True or False. To do this, you'll use 'If Statements,' which are a crucial piece of nearly every programming language. The idea of an *IF STATEMENT* is that you specify some condition, which is a Matlab statement that evaluates to either true or false. If and only if that condition evaluates to true, Matlab will execute some statements. If the condition evaluates to false, those statements will be skipped.

4.1 If Statement

In Matlab, If Statements are implemented as follows, replacing *CONDITION* with an expression that evaluates to True or False and *MATLAB STATEMENTS* with one or more lines of Matlab code:

```
if(CONDITION)
     MATLAB STATEMENTS
end
```

As an example, let's consider:

```
age = input('Enter your age\n');
if(age>=21)
     disp('You can enter the bar')
end
```

In this example, if age is greater than or equal to 21, then Matlab will display 'You can enter the bar.' If *age* is under 21, then Matlab doesn't display anything.

4.2 If-Else Statements

Sometimes, you'll want to take one action if the condition is true and a different action if the condition is false. Matlab lets you use If-Else Statements to say 'If the condition is true, let's do only the first set of actions; however, if the condition is false, let's execute only the second set of actions.'

```
if(CONDITION)
     MATLAB STATEMENTS
else
     OTHER MATLAB STATEMENTS
end
```

As an example, let's consider:

```
age = input('Enter your age\n');
if(age>=21)
     disp('You can enter the bar')
else
     disp('Sorry, you have to stand outside');
end
```

In this example, if the variable *age* contains a number greater than or equal to 21, then Matlab will display 'You can enter the bar.' If *age* is under 21, then Matlab displays 'Sorry, you have to stand outside.' It's important to note that only one set of actions is ever taken.

4.3 If-Elseif-...-Else Statements

Sometimes, you'll want your program to choose one from many courses of action. In these instances, use If-Elseif-Else statements.

```
if(condition)
     statements1
elseif(condition2)
     statements2
...
else
     statements3
end
```

Matlab will first evaluate the truth of the *If* condition. If that condition is true, it executes statements1 and then skips down to the end. If that condition is not true, but condition2 is indeed true, Matlab executes statements2 and then skips down to the end. If Matlab goes through the *if* condition and all of the *elseif* conditions and hasn't found anything that's true, it will execute the statements under *else* (statements3).

Note that you can have as many *elseif* statements as you want. Furthermore, the *else* statement is optional.

As an example, let's consider the following example that splits people by their age:

```
if(age>=65)
     disp('You are a senior citizen')
elseif(age>=25)
     disp('You are a grown-up')
elseif(age>=18)
     disp('You are a college student')
elseif(age>=5)
     disp('You are in school')
else
     disp('You are a little kid');
end
```

It's important to note that only one option is ever chosen– the first that's true. If you're evaluating your code by hand, begin at the *if* statement. The **first** true condition will have its statement(s) executed, and all of the others will be ignored.

As a result, the order of the statements matters very much! What would happen if the first statement in the example above were if(age>=5)? Well, it would display 'You are in school' for everyone over age 5, overriding the messages about senior citizens, grown-ups, and college students.

4.4 More examples of Conditionals

In this first example, we'll use an *if statement* and Matlab's built-in *isnumeric* and *error* functions to stop a program if the expected type of input isn't provided. Checking user input is an important and often neglected part of programming:

```
age = input('Enter your age\n');
if(~isnumeric(age))  % if age isn't a number
     error('age should have been a number, loser');
end
if(age>=21)
     disp('You can enter the bar')
else
     disp('Sorry, you have to stand outside');
end
```

The following example uses the OR operator extensively. Think about what would have happened if the order were changed (i.e. if the second condition came before the first).

```
if(x>=1000 | x<=-1000)
     disp('number has 4+ digits')
elseif(x>=100 | x<=-100)
     disp('number has 3 digits')
elseif(x>=10 | x<=-10)
     disp('number has 2 digits')
elseif(x==0)
     disp('number is 0')
else
     disp('number has 1 digit');
end
```

Of course, there's a better way to solve the above problem, creatively using the functions you learned earlier in this lecture:

```
x = abs(x);  % make x positive
if(x==0)
     fprintf('number is 0\n')
else
     d = floor(log10(x))+1;  % calculate the log, base 10
          % round down, and add 1 (so that 10,100, etc work right)
     fprintf('number has %.0f digits before the decimal\n',d)
end
```

```
if(grade>100 | grade<0)
     disp('Did you enter the grade correctly?');
elseif(grade>70)
     disp('You passed')
else
     disp('You failed');
end
```

This next example shows the AND operator in action:

```
if(age>=17 & age<=25)
     disp('You can drive, but can''t rent a car');
end
```

Notice in this example that we typed two single quotes inside of a string of characters. Normally, the single quote ends the string, but two single quotes right next to each other lets you include the apostrophe inside the string.

This example shows both the AND as well as the OR operators in action. Let's assume the variable *beendrinking* contains a 1 (true) if the driver has been drinking, and a 0 otherwise. Similarly, let's assume the variable *AboveTheLaw* contains a 1 if the driver just doesn't care about ages or drunk driving laws. If the driver is 17 or older and sober, OR if he is Above The Law, he will drive:

```
if( (age>=17 & beendrinking==0) | AboveTheLaw==1 )
     disp('He is going to drive');
end
```

Note that there are many ways to write the above statement. For instance, saying `AboveTheLaw==1` is superfluous. You could have merely said `AboveTheLaw`, which would have evaluated to False if it contained 0 (false), and evaluated to True if it contained 1 (true). Similarly, you could have just typed `~beendrinking`.... "not beendrinking". If beendrinking were 0, `~beendrinking` would have been true. For all other values of beendrinking (including 1), the *not* operator would have made the statement false.

4.5 Switch/Case

Switch case (sometimes known as select case) is an alternative to *If Statements* when you have a variable that only takes on a certain number of discrete values. In all honesty, you never need to use a *switch case* statement if you know *if statements* well, but you should recognize the statement when reading other programmers' code. In particular, it's usually easier to read a *switch case* statement than an *if statement* if you're dealing with character strings. The syntax is as follows:

```
switch VARIABLE
case OPTION1
     STATEMENTS
case OPTION2
     STATEMENTS
otherwise
     STATEMENTS
end
```

You must replace 'VARIABLE' with the name of the variable you wish to test. Replace OPTION1, OPTION2, etc. with values (you don't need to put x==5, you JUST type 5). Matlab will look up the value of the variable that follows the word 'switch' and go down the list of possible cases until it finds the first one that is true. Once it finds

one statement that is true, it doesn't even look at any of the others. If you want to include more than one value, put all of the possibilities in squigly braces; if the variable in question contains ANY of the values in squigly braces, that option will be chosen. The optional 'OTHERWISE' statement is equivalent to 'else' in an *if statement*– if you get down to the *otherwise* statement, you will execute those statements.

Here's an example of *switch case* in action. Note that 'exciting' will be displayed:

```
destination= 'ethiopia';
switch destination
case 'florida'
     disp('boring!')
case 'italy'
     disp('a little less boring')
case {'turkey' , 'ethiopia' }
     disp('exciting')
otherwise
     disp('i don''t know about that country')
end
```

5 Vectors and Matrices

The name Matlab comes from MATrix LABoratory because of its ease at working with matrices. A **matrix** is essentially a grid or array of numbers, with some number of rows and columns of numbers. (In Matlab, the convention is that we always refer to the number of rows first, followed by the number of columns). A **vector** is merely a matrix with only one row or one column.

5.1 Creating Vectors or Matrices

Creating matrices or vectors (matrices with only 1 row OR 1 column) is quite easy in Matlab. You simply enclose a list of elements in square brackets. Here's our first row vector (1 row, 4 columns):

```
x = [ 1 5 8 9 ]
```

To instead create column vectors, you need to indicate to Matlab to 'skip to the next row' in the vector. You use the semicolon (;) to do this:

```
y = [ 1; 5; 8; 9 ]
```

As you might guess, creating a Matrix is essentially the same as a vector. Don't forget that the semicolon skips to the next row:

```
z = [ 1 2 3; 4 5 6 ]
```

This creates a 2x3 (2 rows by 3 columns) Matrix. *Whenever we talk about the size of Matrices, the number of rows always comes first.*

5.2 Special Ways to Create Vectors/Matrices

Matlab presents a number of more efficient ways to create certain types of vectors and matrices.

5.2.1 Colon

For cases where you need to create a vector of, say, the integers from 1 to 1000, you can do so by using the colon operator, which in Matlab essentially means "from X to Y" when you write X:Y.

```
a = 1:1000
```

The code above creates the vector [1 2 3 ... 1000], containing all of the integers from 1 to 1,000.

By default, the colon uses steps of 1 when going from one number to the next. However, adding a number in the middle of the start and end points specifies the 'step size,' or 'how much to skip' when going between elements. You'll want to do this when trying to space points very close together, or decreasing between points rather than increasing:

```
% variable = start : skip : end
a = 1:0.1:1000
b = 0:-1:-100
```

The "a=" line above creates a vector containing [1 1.1 1.2 1.3... 999.8 999.9 1000], the numbers from 1 to 1,000 skipping by 0.1. Similarly, the "b=" line above creates a vector starting at 0 and going down (skipping by -1 each time) to -100: [0 -1 -2 ... -99 -100].

5.2.2 Linspace

Whereas the colon operator is very useful for creating vectors when you know how much space to skip in between each point, sometimes you'll want to say something like 'give me 200 points between 5 and 22.' To accomplish this task, you use the linspace operator, which spaces points evenly (linearly) between a start number and end number:

```
% variable = linspace(start,stop,# of points)
c = linspace(5,22,200)
```

This example creates a vector of 200 evenly spaced points, where the first element is 5 and the final element is 22.

5.2.3 From Other Matrices

When creating a Matrix using the square brackets ([ ]), other vectors or matrices can be included inside the brackets. For instance, given a vector *c = [1 2 3]*, you could write *d = [c ; c]*, and you'd get *d* equal to the following 2x3 vector:

```
d = [ c ; c ]
      d = [ 1 2 3 ]
          [ 1 2 3 ]
```

Note that this isn't a matrix with other matrices inside of it, but rather one big matrix of numbers. Matlab smashes together all of the inner matrices to make one matrix.

5.2.4 Ones, Zeros, Eye

Three functions exist to quickly create special matrices:

ones(3,4) creates a 3x4 matrix filled entirely with the number 1. If you only pass the *ones* function a single parameter i.e. *ones(3)*, a 3x3 square matrix filled with the number 1 is created. If you multiply the matrix created by *ones* by a scalar (single) number, you can now create a matrix filled with any single number. For instance, *5.5*ones(4,3)* creates a 4x3 matrix containing 5.5 as each element.

Similarly, the *zeros* function creates a matrix filled with the number 0, i.e. *zeros(10,9)*.

The *eye* function creates the identity matrix (normally called i– get the joke? Matlab's joke is more unfunny than my jokes.) Recall that the identity matrix contains 1 along the main diagonal (from top left to bottom right), and zero elsewhere. For instance, *f = eye(3,4)* creates:

```
f = [ 1 0 0 0 ]
    [ 0 1 0 0 ]
    [ 0 0 1 0 ]
```

5.3 Determining the Length/Size of a Vector/Matrix

To find the size of a matrix, you can use the *size* function. It returns a 1x2 vector containing the number of rows, and then the number of columns. You can either store this 1x2 vector in a single variable (as in the first example below) or store the number of rows and number of columns in separate variables (as in the second example). To accomplish the latter, Matlab follows an interesting convention. If you know that a function you call will return 2 outputs, you can

set multiple variables at once by enclosing 2 variable names in square brackets on the left-hand-side of the equals sign:

```
% assume M is a 2x3 matrix
z = size(M);  % z will be a 1x2 vector of the answers
[r c] = size(M);  % specify output variables separately
```

z would contain the vector [2 3], r would contain the value 2, and c would contain the value 3.

A vector only has one meaningful dimension, although you don't always know whether it has only 1 row or instead only 1 column. Additionally, sometimes you only care about the largest dimension of a matrix rather than both the number of rows and columns. In this case, you can use the *length* function, which returns the LARGER of the number of rows and number of columns, and thus the length of a vector:

```
% assume X is a vector
l = length(X)
```

5.4 Accessing Vector/Matrix Elements

5.4.1 Row, Column

For a vector or for a matrix, you can access an element by indicating the row location and the column location, along with the name of the variable that stores the vector or matrix. i.e. to access the element in the 2nd row and 3rd column of the matrix stored in the variable *z*, you'd type:

```
z(2,3)
```

You can use this technique to either display a value i.e. *disp(z(2,3))* or to set a value i.e. *z(2,3)=100* sets the element in Row 2, Column 3 of *z* to 100 but leaves the rest of the matrix unchanged..

5.4.2 Single Number

Alternately, you can just use a single number to identify the n'th element of either a vector or a matrix. For a vector *v*, typing *v(4)* would allow you to access (to display or change the value of) the 4th element of v.

Matrices are numbered first going all the way down the first column, and then going all the way down the second column, and so on. Thus, in a 4x4 matrix *M*, M(5) will refer to the element in the 1st row and 2nd column.

5.4.3 Colon Operator

In order to select multiple elements from a vector or matrix, you can use the colon operator (:). As you saw earlier, x = 1:100 means 'create a vector with the elements 1 through 100, incrementing by one by default, and store the result in x.' On the other hand, you can also write the following, which 'selects elements 1 through 100 of the vector or matrix stored in x':

```
x(1:100)
```

Thus, if you typed *disp(x(1:100))*, you would display the first 100 elements of *x*.

This same convention also works when you're selecting elements from a matrix by identifying the rows and the columns. The following example selects all of the elements in rows 1 through 2 and columns 3 through 4:

```
z = [ 7 2 6 3 ; 2 5 3 2 ; 3 6 4 2 ]
     z = [ 7 2 6 3 ]
         [ 2 5 3 2 ]
         [ 3 6 4 2 ]

z(1:2,3:4)
     ans = [ 6 3 ]
           [ 3 2 ]
```

Sometimes, you want to have Matlab 'select all of the rows (or columns)' without needing to think about how many elements are in that row or column. If you use the colon operator by itself, it means 'all of the.' In the following example, we output all rows and only the 2nd column of *y*:

```
z = [ 7 2 6 3 ; 2 5 3 2 ; 3 6 4 2 ]
     z = [ 7 2 6 3 ]
         [ 2 5 3 2 ]
         [ 3 6 4 2 ]

z(:,2)
     ans = [ 2 ]
           [ 5 ]
           [ 6 ]
```

If you have a matrix that needs to be turned into a vector, you can simply index the matrix with a single colon or use the (more complicated) *reshape* command. Using the single colon operator adds elements from the matrix into the vector going down the first column, then going down the second column, and so on:

```
z = [ 7 2 6 3 ; 2 5 3 2 ; 3 6 4 2 ]
     z = [ 7 2 6 3 ]
         [ 2 5 3 2 ]
         [ 3 6 4 2 ]

z2 = z(:)
     z2 = [7 2 3 2 5 6 6 3 4 3 2 2]
     % z2 is actually a 12x1 column vector!
```

5.5 Strings

Character strings in Matlab are simply vectors of single characters. Thus, any operation we perform on vectors can be used on a string:

```
a = 'I love Matlab';
a(3:6) = 'hate'
     ans = 'I hate Matlab'
a((end-2):end) = []   % deletes those values
     % by setting them to the 'empty vector'
     ans = 'I hate Mat'
fliplr(a)
     ans = 'taM etah I'
```

5.6 Element by Element Math Operations on Matrices

Mathematical operations on Matrices can be a bit tricky since there are multiple ways in which operations such as multiplication can be defined mathematically on matrices. However, other operations such as addition and subtraction are much simpler since they operate 'element by element.'

5.6.1 Add, Subtract

In order to add or subtract two vectors or matrices, they must have the same dimensions (number of rows, number of columns). The element in the first row and first column of the first matrix is added to the corresponding element in the second matrix, and that becomes the element in the first row and first column of the result:

```
a = [ 7 2 ; 6 3 ];
b = [ 1 3 ; 1 5 ];
a + b
    ans = [ 8 5 ]
          [ 7 8 ]
```

5.6.2 Element by Element (Dot) Multiply, Exponentiate

The multiplicative analogue of addition and subtraction, which work 'element by element,' is called 'dot multiplication' and is defined using a period followed by a multiplication sign– *.*

```
a = [ 7 2 ; 6 3 ];
b = [ 1 3 ; 1 5 ];
a .* b
    ans = [ 7  6 ]
          [ 6 15 ]
```

Similarly, to raise each individual element in a matrix to a particular power, which indeed is just repeated multiplication, also uses a dot operator– .^

```
b = [ 1 3 ; 1 5 ];
b.^2
    ans = [ 1  9 ]
          [ 1 25 ]
```

5.7 Functions for Matrices and Vectors

5.7.1 Sum, Prod

Matlab includes functions that add or multiply all of the numbers in a vector. As you might guess, inputting a vector to the *sum* function adds all of the elements of the vector, whereas the *prod* function multiplies all of the elements.

```
sum( [ 1 5 7 ] )
                ans = 13
prod( [ 2 5 7 ] )
                ans = 70
```

Note that applying these functions to a matrix results in summing or multiplying *EACH COLUMN* of the matrix and outputs a vector containing the sum or product of each column. Thus, to sum all of the elements of a matrix *M*, you first run *sum(M)* to create a vector of the sums of each column, and then sum that result: *sum(sum(M))*.

```
M = [ 1 5 7 ; 6 1 2 ]
   ans = [ 1 5 7 ]
         [ 6 1 2 ]
sum(M)
   ans = [ 7 6 9 ]
sum(sum(M))
    ans = 22
```

5.7.2 Max, Min

Matlab also provides functions to find the maximum and minimum values of a vector or matrix. These are the creatively named *max* and *min* functions. For vectors, they return a single scalar with the absolute maximum number. For matrices, they return the maximum or minimum number *in each column*. Thus, as with the *sum* and *prod* functions, to find the absolute largest number in a matrix *M*, you would use *max(max(M))*.

The *max* and *min* functions can also be used to identify the location of the maximum or minimum number inside a vector (or, with a little bit more trouble, a matrix). If you write a statement like *[a,b] = max(V)*, where *a* and *b* are names of variables and *V* is a vector, then *a* will contain the largest value and *b* will contain the location of that largest value inside the vector V. Note that [a,b] is essentially just a vector of two variables. Also note that if the largest value is found more than one time inside the vector, *b* will contain the first location where it is found.

```
V = [ 5 7 6 1 2 ];
[a,b] = min(V)
     a = 1   % The smallest element is 1
     b = 4   % 1 is the 4th element of V
```

What if you wanted to find the row and column in which the maximum or minimum is located? Here, you can be creative using a few different methods, one of which is shown:

```
V = [ 5 7; 6 1; 2 3];
[bigC,rowsC] = max(V)
     bigC = [6 7]   % The largest element IN EACH COLUMN
     rowsC = [2 1]   % The ROWS that contain the largest elements
[big,column] = min(smallC)
     big = 7   %  The largest overall element
     column = 2   % The COLUMN containing the overall largest element
row = rowsC(column)  % Column 2 contains the largest overall element.
          % Check which ROW in Column 2 contains the largest element.
     row = 1
```

5.7.3 Mean, Median

The *mean* and *median* functions operate on vectors just as you'd expect from statistics. The *mean* calculates the average, the sum of the elements of a vector divided by the number of elements. The *median* identifies the middle number if the vector were to be ordered from smallest to largest element.

As with the other matrix functions, calculating the mean or median of a matrix calculates the mean or median of each column. Thus, to find the mean of a matrix M, use *mean(mean(M))*. However, this wouldn't work for the median– think about why! Instead, use *mean(M(:))*, which first converts the matrix M to a vector (by making a vector containing ALL of the elements) and then finding the median.

5.7.4 Sort, Sortrows

You can also sort the elements of a vector in ascending (smallest to largest) order by using the *sort* function. To instead sort a vector V from largest to smallest, instead use *flipud(sort(V))* for column (vertical) vectors and *fliplr(sort(V))* for row (horizontal) vectors. For matrices, note that the *sort* function sorts each column individually.

There is a similar function called *sortrows*, but be careful– it doesn't sort the rows individually and is thus not an analogue of the *sort* function. Instead, it merely performs a sort in which the rows are kept together and then ordered beginning with the elements in the first column, and then the second, and so on. If the rows actually just contain words (i.e. you have characters as each element), *sortrows* would *alphabetize* the words.

6 Filtering And Choosing Certain Elements

Being able to pick particular elements out of a matrix or vector is very useful. The first key in filtering matrices and vectors is to understand how Matlab interprets logical operators involving a scalar (single value) when applied to matrices. For a matrix M, when given an operation like $M > 5$, Matlab returns a matrix the same size as M, but filled with 1's and 0's (true and false) indicating whether or not that statement was true for each element of the matrix.

```
A = [ 1 5 ; 7 9 ]
     A = [ 1 5 ]
         [ 7 9 ]
A>=5
     ans = [ 0 1 ]
           [ 1 1 ]
```

6.1 Logical Indexing

Interestingly (and rather uniquely among program languages), Matlab allows you to use these logical (true and false) matrices to index (or choose) elements from a matrix or vector. Given a logical matrix as the index, Matlab returns a vector of only those elements for which the corresponding element in the logical matrix was true:

```
A = [ 1 5 ; 7 9 ]
     A = [ 1 5 ]
         [ 7 9 ]
A(A>=5)
     ans = [ 7; 5; 9 ]

A(A>=5) = 99
     A = [  1 99 ]
         [ 99 99 ]
```

6.2 Find

Whereas logical operators by themselves return a 1's/0's matrix (true and false) and whereas using logical indexing returns only the elements of a matrix for which the condition is true, the *find* function returns a **vector of the LOCATIONS**, indexed with a single number, of the elements in a matrix that meet a certain condition. In other words, find will return a vector containing the indices of all matrix elements for which that conditional is true.

```
A = [ 6 3 8 2 1 ];
locations = find(A>5)
                locations = [ 1 3]
```

This means that the 1st and 3rd elements of A are greater than 5. To return vectors of the rows and columns, respectively, of the elements meeting a certain condition, simply say something like `[rows cols] = find(V==5)`

6.3 More Examples

Let's say we had a matrix M and wanted to sum every number over 100. The following code would do that:

```
sum(M(M>100))
```

Note that although the 'M(M...' part of this example seems redundant, it certainly is not. Moving from the innermost parentheses outwards, 'M¿100' returns a logical (true and false / 1 and 0) matrix. Thus, 'M(M¿100)' returns all elements of M that are greater than 100. This is a vector, so we just sum that vector to get the answer!

Here's one more example of how you can use the outputs of different functions to feed into each other. Let's say we had a matrix *M = [285 146 137 ; 69 267 6 ; 182 229 246]* and we wanted to find the largest number between 130 and 150. We could use the following code:

```
M = [ 285 146 137 ; 69 267 6 ; 182 229 246 ];
max(M(M>=130 & M<=150))
```

Note that `M(M>=130 & M<=150)` will be a vector containing [146 137], the two numbers fitting the criteria of being between 130 and 150, inclusive. Thus, we're just finding the max of the vector [146 137].

You've probably noticed by now that Matlab gives you lots of basic tools; however, you often have to combine them creatively to get the answer. Let's say I asked you for the sum of all the prime numbers from 103 to 1003, inclusive. Well, you know that the *sum(primes(x))* gives you the sum of the primes from 1 to x. Well, why not just sum all the primes from 1 to 1003, and subtract the sum of all the primes from 1 to 102... that leaves you with the sum of the primes from 103 to 1003:

```
% sum the primes from 103 to 1003
sum(primes(1003) - sum(primes(102))  % 102 is 1 less than 103
```

Now, what if I wanted you to just create a vector of the primes from 103 to 1003? Well, we can create a vector of all the primes from 1 to 1003 and use logical indexing:

```
% create a vector V of the primes from 103 to 1003
a = primes(1003);
V = a(a>=103);

% Here's an alternate, less elegant, solution
a = primes(1003);
V = a((length(primes(102))+1):end);
```