1 Getting Matlab

```
Matlab is freely available to all BU students.
```

2 Using the Command Line

2.1 Simple Math

When you open Matlab, the first thing you'll usually see is the command line, which looks something like this:

```
>>
```

You can type commands here, and Matlab will execute them. For example, try

```
>> 1+2

ans = 3
```

If you put a semicolon (`;`) at the end of the line, Matlab will not print the output to the screen: (this can be useful if the output takes up a lot of space, like if it's a large matrix)

```
>> 1+2;
```

Matlab has many built-in math operators, a few are (`+ - * / ^`) for add, subtract, multiply, divide, and raise-to-power. Matlab follows all the normal order-of-operations rules, and you can use parenthesis to tell it exactly what order to do things.

2.2 Creating Variables

Just like in any programming language, you can store numbers in variables. Creating a variable is very easy: say we want to store the number 5 in a variable named `myVar`. The code for this is

```
>> myVar = 5;
```

Then we can use `myVar` just like a number in Matlab commands:

```
>> myVar + 1

ans = 6

>> myVar^2

ans = 25
```

3 Working With Vectors

3.1 Creating Vectors

Matlab's true power is in its ability to do computations involving vectors and matrices. There are a number of ways to create vectors; the most straightforward is to use square brackets:

```
>> lost = [4, 8, 15, 16, 23, 42];
```

In the above, we used commas (`,`) to separate the elements of the vector. In Matlab, this gives us a **row** vector. If we use semicolons (`;`) instead, we get a **column** vector:

```
>> fib = [1; 1; 2; 3; 5; 8];
```

If you want to **transpose** a column vector to a row vector (or vice-versa), use an apostrophe (`'`):

```
>> fib'

ans = 1  1  2  3  5  8
```

Note that this did not transpose the `fib` vector in place, it merely printed a transposed version of it. In memory, `fib` is still stored as a column vector. **This is true of most Matlab operations: most operations on vectors return a *c*opy of the vector with the operation performed.**

You can also create a **matrix** in the same way by combining commas and semicolons:

```
>> myMatrix = [1, 2; 3, 4]

ans =
     1     2
     3     4
```

You can multiply vectors and matrices together as long as their inner dimensions match; so since `fib` is 6x1 and `lost` is 1x6, you can multiply them together like

```
>> lost * fib

ans = 541
```

However, you can't execute `lost*lost`, because the rules of matrix algebra don't allow you to multiply a 1x6 with a 1x6. If you want to multiply them together **elementwise**, i.e., multiply the first two elements together, then the second two, and so on, you have to use the special (`.`) symbol:

```
>> lost.*lost

ans = 16 64 225 256 529 1764
```

The (`.`) symbol executes an operation on *each* element of the vector or matrix, so if we wanted to calculate the inverse of every element of `myMatrix`, we would write

```
>> 1./myMatrix

ans =
    1.0000    0.5000
    0.3333    0.2500
```

3.2 The colon (:) operator

One quick way to make some simple vectors is to use the colon. There are two ways to use colon. The first is like this:

```
>> sequence = 1:5

ans = 1  2  3  4  5
```

In this usage, the colon operator creates a vector that is a sequence of integers from 1 to 5. You can also use the colon to tell Matlab how much space to put between each element of the vector:

```
>> anotherSequence = 0:0.2:1

ans = 0.0000  0.2000  0.4000  0.6000  0.8000  1.0000
```

Using it this way, it says "create a vector starting at 0, with elements spaced 0.2 apart, all the way up to 1". This can be useful for many things!

3.3 Indexing Vectors

If you want to select a particular element from a vector, use parenthesis (note: in Matlab, vectors are indexed starting with the number 1. This is unusual in programming languages; most other languages number the first element 0). To get the 6th element from `fib`, type

```
>> fib(6)

ans = 8
```

You use the same syntax to modify an element of a vector. To change element `1` of the `lost` vector to `0`, I write

```
>> lost(1) = 0

ans = 0  8  15  16  23  42
```

Suppose you want the last element of a vector, but you don't know how long it is. Then you can use the `end` keyword:

```
>> lost(end)

ans = 42
```

To select a range of elements in a vector, you can use the colon operator. For example, if we want elements `2` through `4` from `lost`, type

```
>> lost(2:4)

ans = 8  15  16
```

If you want elements `1`, `3`, and `5` from `lost`, type

```
>> lost([1,3,5])

ans = 0  15  23
```

3.4 Indexing Matrices

You can get sub-vectors and elements from matrices in much the same way you can from vectors. Just remember that Matlab uses the standard row-index-first, then column-index-second convention. So if we want the upper-left element (1st row, 1st column) from the `myMatrix` object we created earlier, we type

```
>> myMatrix(1,1)

ans = 1
```

If we want the bottom-left element (2nd row, 1st column), type

```
>> myMatrix(2,1)

ans = 3
```

If you want to pull the 1st row out of a matrix, you use the colon (`:`) operator to tell Matlab to give you the 1st row and "all columns":

```
>> myMatrix(1,:)

ans = 1  2
```

Likewise, if you want the 2nd column, you tell Matlab to give you "all rows" and the 2nd column:

```
>> myMatrix(:,2)

ans = 2

      4
```

4 m-Files

The command line is great if you want to get an answer to a simple math problem quickly, but usually you'll want to run a bunch of commands all in a row using a script file. To create a script file, go to the file menu in Matlab and choose new>Script. This will open up a blank window for you to write a script file. Since Matlab saves it with the extension `*.m`, it's called an "m-file."

An m-file is just a list of commands; when you execute the file, Matlab just executes the commands in the order you put them in the file. You can run the file either by pressing the green arrow button at the top of the editor window or by typing the filename (without the `.m`) into the command line.

5 Important Functions

5.1 If/Else Statements

Often you'll want to run code only when a particular statement is true. Suppose we are pumping water into a tank and we want to turn the pump off when the tank level is over 100. We have a variable `tankLevel` that tells us the tank level, and a variable `pumpSpeed` that we can change to

tell our pump how fast to run. The following code would shut the pump off (set the speed to 0) when the level is over 100, runs the pump slowly (a speed of 1) if the tank is between 90 and 100, and runs the pump fast (a speed of 2) if the tank is below 90:

```
if tankLevel > 100
      pumpSpeed = 0;
elseif tankLevel > 90
      pumpSpeed = 1;
else
      pumpSpeed = 2;
end
```

Note that in an `if`-statement, you don't actually need an `elseif` or even an `else`, but you *always* need an `end`. You can also put as many `elseif`-statements as you want. For example, in the above statement, we could have set a different speed when the level is between 80 and 90.

You can use many different logical operators in an `if`-statement:

- `(~a)` means "not `a`"
- `(a && b)` means "`a` and `b`"
- `(a || b)` means "`a` or `b`"
- `(a == b)` means "`a` equal to `b`"
- `(a ~= b)` means "`a` not equal to `b`"
- `(a >= b)` means "`a` greater than or equal to `b`"
- `(a <= b)` means "`a` less than or equal to `b`"

5.2 For Loops

Sometimes, you want to repeat the same code a particular number of times. If you know how many times to repeat, use a `for`-loop. For example, suppose you want to add up all the numbers from 1 to 10. One way to do this would be like so:

```
result = 0;
for i = 1:10
      result = result + i;
end
```

First, we create the variable `result` and set it to 0. This will be the place we store our sum as we compute it. Next, the `for`-loop starts. When we write "`for i = 1:10`," we're saying "create a temporary variable called `i`. The first time the loop runs, let `i = 1`; the second time, let `i = 2`, and so on all the way up to `i = 10`. Then, each time the loop repeats, we add `i` to `result`, and in this way we add together all the integers from 1 to 10.

5.3 While Loops

Sometimes you want to repeat the same code many times, but you don't know exactly *how many* times. In this case, you may want to try a `while`-loop. This type of loop will run over and over until a particular condition is met.

For example, suppose you wanted to find out how many powers of 2 it takes to reach 1 million. One way to do it would be like this:

```
count = 0;

powers = 0;

while (2^powers) < 1000000

      count = count + 1;

      powers = powers + 1;

end
```

First, we set our `count` and `power` variables to 0. Each time the loop runs, we'll increase them by 1. When we write "`while (2^powers) < 1000000`," we're saying "run this loop if 2 to the power of `powers` is less than 1 million." Then we increase `powers` each time the loop runs, so eventually the loop will stop.

Be careful with this kind of loop, because you could create an infinite loop by mistake. In this example, if we forgot to write code that makes `powers` increase with each loop, we'd be stuck with `(2^powers) = 0` for all time and the loop would never terminate. If you get stuck in an infinite loop, Matlab may not tell you - your code will just sit there and never stop. You can always stop it manually by pressing `ctrl-c`.

6 Writing Your Own Functions

Often you'll want to write code that you can use over and over in a modular way. In Matlab, we do this with functions. Here is a simple function that takes in a number and returns that number plus one:

```
function result = increment(a)

result = a+1;

end
```

A function declaration always starts with the keyword `function`

followed by `<output> = <functionName>(<input>)`.

Typically, you give each function its own m-file, and keep it in the same directory as the rest of your m-files. Make sure that the filename is the same as the function name, so `increment.m` would have to be the name of the file containing our `increment` function. Then you can call the function either from the command line or from a script file in the same directory simply by executing `increment(a)`.

7 To learn more, or What to do when you're stuck

Matlab has a tutorial program at `https://matlabacademy.mathworks.com`.

Other than that, here are some powerful tools you can use if you get stuck:

1. `help`. Suppose you know that there is a Matlab function called `plot`, but you don't know how to use it. At the command line, just type `help plot` and Matlab will give you the contents of the help file. (of course, if you don't know the name of a command, Google is usually pretty helpful.)
2. Debug Mode. You can use a built-in tool called debug mode to look inside your code while it's operating. You may have to play around with it to figure out how your particular version of Matlab works, but essentially, debug mode lets you tell a script or function to pause when it gets to a particular line of code.
3. Check out `http://www.mathworks.com/products/matlab/videos.html` for videos explaining everything about Matlab.
4. If you have any other questions, just type `why` into the command line.

8 Exercises (totally optional, not graded in any way)

1. Write a script that computes the first 100 Fibonacci numbers. The Fibonacci numbers are computed like this: the next number is the sum of the previous two numbers. So if the first two numbers are $\{1, 1\}$, then the third number is $2 = 1 + 1$, the fourth number is $3 = 1 + 2$, the fifth is $5 = 2 + 3$, and so on: $\{1, 1, 2, 3, 5, 8, 13, 21, \ldots\}$. Formally, we say that for each i, $f_{i+1} = f_i + f_{i-1}$.
2. Write a function called `myFactorial(n)` that takes in one argument, n, and returns $n!$ (that is, n-factorial). Recall that $n!$ is the product of all the integers up to n; so $5! = 1{\cdot}2{\cdot}3{\cdot}4{\cdot}5 = 120$.

You can use the built-in MATLAB function `factorial(n)` to check your work, but don't call `factorial(n)` in your function.

3. Run the command `testVec = rand(1000,1);`. This will create a vector with 1000 random numbers (between 0 and 1). Write a function that takes in this vector and returns the smallest element. You can use the built-in function `min()` to check your work, but don't call `min()` in your function.

4. Now, run the command `testMat = rand(1000,1000);`. This creates a 1000x1000 matrix full of random numbers. Write a function that takes in this matrix and returns the smallest element.

5. Write a function like in Exercise 4, but this time make it so it can find the smallest element in a matrix of *any* size. Hint: look up help on the built-in function `size()`.

6. To test yourself more, check out the exercises at
`http://www.facstaff.bucknell.edu/maneval/help211/exercises.html`
Some of these are quite advanced.