An Introduction to Matlab: Part 6

This lecture does not use all of the ideas from our previous lectures. It is simply intended as a tutorial for those of you who would like to attempt to use Matlab for programming. The final result is a nice little Newton's Method for a specific function f. You should know how to create and execute functions and scripts, and have a familiarity with vectors and component wise arithmetic. I am also assuming you have some familiarity with programming in general. This is not meant to be a tutorial on *how to program* but rather how to use Matlab to program. This tutorial is also a bit more example oriented, again, since it is geared toward people who can program already. So, we are covering only:

- Programming in Matlab

Programming in Matlab

Most everything that one would expect to work in a programming language such as C, C++ or Fortran works in Matlab. We'll work through some examples, describing some differences between Matlab and other languages. A huge advantage in Matlab is that we have access to a gigantic library of mathematical functions. To access this many functions in other languages often requires the use of third party programs, which can sometimes be a bit difficult to compile and use. Writing working Matlab code tends to take much less time than in C or C++, but code tends to be slower. Many people (including myself) use Matlab to confirm ideas or algorithms. If you then need to have the algorithm do something very fast, you program it in C or C++ after confirming the algorithm works in Matlab.

1. *Data Types:* First, there are generally no data types in Matlab separating floats, integers, doubles, unsigned integers or anything like that. If I have $a = 1$ and $b = 4$ then a/b will be .25 and not 0 (like if these were both integers and I was in C). Matlab is going to convert our variables to whatever it thinks we need, and it's going to be fairly smart about it. Be aware of this fact, since you may need to use *floor* or *ceil* to get things as integers.
2. *Hello World!:* A nice way to start learning a programming language is to write a program which displays *Hello World!*
   (a) Open Matlab. If you already have it open, type *clear all;* in the *Command Window*. Open up the Matlab editor. This can be accomplished by typing *edit* in the *Command Window* or by using the menus.
   (b) We're going to create a script file, so all we need to do is type the series of commands we want to have executed. When you opened the editor, you should have had an empty document open, likely titled *untitled1.m*. Save this file wherever you want as *HelloWorld.m* by going to File->SaveAs.
   (c) Now we'll type the command that displays *Hello World!*. It is called fprintf. For those of your who know C, the command is very familiar. We type
   fprintf('Hello World!\n')
   Here, the text we wish to display is enclosed in single quotes, and the \n is a command character that tells fprintf to print a new line.
   (d) *fprintf* would also be the command you use to write out to a file. If you ever need to do this, consult the Matlab documentation.
3. *Loops:* Anyone who programs knows that you're going to have to eventually learn to use a loop to be effective.
   (a) Let us create a new .m file, save it as *myprogram.m*. Once you have saved the file, hit the run button in the editor. You will likely have to click *Change Directory* when prompted by Matlab (we do this simply to make sure Matlab switches to the same directory as your program).
   (b) We're going to make *myprogram* a function, which takes two arguments, *x0* and *N*, and returns the variable *x*. We do this by typing the first line of the function to be
   function x=myprogram(x0,N)
   (c) Next, we're going to define a function f in Matlab. One way to do this is to use what is called an *inline* function. For you next line, type
   *f = inline('x.^2 - 2*x - 8');*
   An inline function will assume that I am defining a function of x (or x, y if I use two variables). So essentially this make f a function which takes x as an argument and returns $y = x^2 - 2x - 8$.

(d) We're going to perform an iteration on f for N steps, at a starting value of $x0$. The idea is that we're going to compute $x_{n+1} = x_n - f(x_n)$ over an over again, starting at x_1 and going to x_{N+1}. The function will print out each value of x. To do a loop, we use the function *for*. The construct for *for* consists of a variable name and a vector. The variable name tells the loop which variable is going to change, and the vector is going to indicate how it will change. In order to end a loop, we type *end*. Everything that we want to be looped, we put between the *for* and the *end*. So, for your program, the next lines are:

```
x(1)=x0;
for n=1:N
  x(n+1) = x(n)-f(x(n));
end
```

(e) Run this function from the *Command Window* . Try *N=4* and some different values of x (try at least $x = -2, 0$ and 4). To run for $x = 0$ and $N = 3$ we would type

```
x=myprogram(0,3)
```

(f) What happens for $x = -2$ and $x = 4$? Can you explain why this is so?

4. *if-then (if -else)statements:* Another crucial component to any programming is the if/then statement.

(a) We're going to use the same exact program as above, but just modify a few lines. We're going to implement Newton's method for $f(x) = x^2 - 2x - 8$ with a check that quits the loop if Newton's method has converged. First, in order to use Newton's method, we have to define the derivative of $f(x)$. After you define f, but before the loop, define fp to be an inline function that is the derivative of f.

(b) Recall what Newton's method does. We iterate $x_{n+1} = x_n - \dfrac{f(x_n)}{f'(x_n)}$. So replace your previous iteration, *x(n+1) = x(n)-f(x(n));* , with

```
x(n+1) = x(n)-f(x(n))/fp(x(n));
```

(c) Try rerunning the program with $N = 10$ and $x = -3$. Try it with $x = 10$. You'll notice that long before we hit $N = 10$, it appears we've reached the stable point of the iteration (the root of the polynomial). We want to tell the loop to quit if this is the case. We're going to use an if-then statement to do this. The construct in Matlab actually uses *if, elseif, else* and *end*. It would look like

```
if(something)
   some statement here
elseif(something else)
   some other statement here
else
   yet another statement here
end
```

For our program, we want an *if* statement within the loop that quits if $f(x(n))$ is small, and continues otherwise. Inside the loop, and after the Newton iteration, we put the statement

```
if(abs(f(x(n+1))) < 1e-6)
   fprintf('Newtons Method converged in %d iterations\n',n);
   break;
end
```

The *break* command quits the innermost *for* loop. A *return* command instead of *break* would quit the entire function. A *continue* command would quit the current iteration of the loop and cycle to the next iteration (ie, quit at that point and index n).

(d) Rerun your program with $N = 10$ and try out many different values of x. What happens?

5. *while loops:* We explore how to create a while loop, which is useful when we do not know how many iterations it will take something to converge.

(a) We're going to modify the previous program. You should have a section that looks a whole lot like:

```
for n=1:N
   x(n+1) = x(n)-f(x(n))/fp(x(n));
   if(abs(f(x(n+1))) < 1e-6)
      fprintf('Newtons Method converged in %d iterations\n',n);
      break;
   end
end
```

Instead of creating a *for* loop, we can use a *while*. What do we really want? We want this to keep running while *abs(f(x(n+1))) > 1e-6*. If we hit the maximum number of iterations, N , then we'll quit. How can we do this?

```
n=1;
while(abs(f(x(n))) > 1e-6)
   x(n+1) = x(n)-f(x(n))/fp(x(n));
   if(n == N)
      fprintf('Newtons Method did not converge in %d iterations\n',n);
      break;
   end
   n = n+1;
end
fprintf('Newtons Method converged in %d iterations\n',n-1);
```

The *while* statement says to keep doing whatever is between *while* and *end* until the condition is not satisfied anymore. So this will run until $|f(x_n) \leq 1e-6|$. Next, we have our Newton iteration. Then we check to see if we have hit the maximum number of iterations. If so, we quit and tell the user that Newton's Method did not converge. After this *if* statement, we increase n by 1, so that it will be different the next time through the while loop. You can see that with the *if* statement, this has the same behavior as the previous program. Without the *if* statement, this would run until convergence and never stop if you hit some maximum number of iterations.

(b) Rerun your new program using the same values as the previous program. You should get the same results.

6. *other tips/tricks:* We discuss some other tips and tricks to programming in Matlab.

(a) *functions within functions*: You can create a function file, and have functions defined in that file that ONLY that function can use. In other words, let say I have my Newton's Method code. Instead of defining $f(x)$ by an inline function, I can define it as a separate function, known only by my Newton's Method code. This is useful when you need to create some subroutines that only your function needs to use. For example, we can have:

```
function x=myprogram(x,N)
.... All the Newton's Method code here with f(x) and fp(x) being used, but not defined as inline functions

function y=f(x)
y=x.^2-2*x-8;

function y=fp(x)
y=2*x-2;
```

Now try running your code. You should get the same answer. Type *f(1)* at the Command Window. Type *fp(1)*. Neither of these work... why not?? Because *f* and *fp* exist ONLY as part of *myprogram*, not as external functions.

(b) *loops within loops:* You can, of course, have loops within loops (within loops...etc). This is sometimes nice when you are using matrices, but you should use them ONLY when necessary. For example, if I have two matrices of size 3×3, given by A and B, I can find $C = AB$ by

```
for i=1:3
   for j=1:3
```

```
    C(i,j)=0;
    for k=1:3
      C(i,j)=C(i,j)+A(i,k)*B(k,j);
    end
  end
end
```

I would AVOID MULTIPLE LOOPS as much as possible. There is no need to program matrix multiplication, since $A * B$ works perfectly fine (and is much faster) in Matlab. Use built in Matlab functions and component wise arithmetic when possible. They are always going to be significantly faster than what you write on your own.

(c) *cell arrays:* Every now and then you'll want to create an 'array' which has different data types (this isn't really an array, but more of a container). In other words, I'd like to create some structure that has a matrix, a number, and a string in it. How can I do this? The answer is cell arrays. They allow you to create whatever object you want in any location of the cell. The objects of the array are indexed by {} braces instead of [] like with regular arrays. An example of a cell array is:

```
c= {[1 2; 3 4] 4 'Hello World!'};
```

Type this in Matlab, then look at *c{1}*, *c{2}* and *c{3}* .