Optimization in Matlab (manually)

In these notes, we look specifically at the Matlab code that we will write to perform basic optimization. To do that, first we look at how Matlab deals with functions.

How to deal with Functions

As in mathematics, in programs, a function is an operation that takes in values (arguments), performs some operations on those values, and outputs values. Unlike in mathematics, in a program, a function can output more than one "y-"value.

There are several ways of writing functions in Matlab- We'll look at two of them.

Function in an M-file

We've already seen how to write a short Matlab function using a text editor. For example, let's write a function (call it "MyFunc") that will take in two values, say a and b and will output the distance between the numbers, as well as the center between them. Here is what we would type:

```
function [distance,center]=MyFunc(a,b)
% It is always good to include comments about the inputs and
%   outputs of the function- you might forget!
% function [distance,center]=MyFunc(a,b)
%  Input:  Two numbers a and b
%  Output:  distance is the length between the numbers.
%           center is the center point between the numbers.

distance=abs(b-a);
center=(a+b)/2;

end  %This is optional, but a good idea.
```

Now save this file as `MyFunc.m`, and in the directory or search path for this function, we can call it. For example, in the command window I can type:

```
[C,D]=MyFunc(3,6);
```

The value stored in C will be 3 and the value stored in D will be 4.5.

An Anonymous Function

If you have a function with only one output, you might write it as an anonymous function. Here's a quick example, with inputs a and b and output being the average:

```
MyAvg = @(a,b) (a+b)/2;
```

Now typing `MyAvg(3,6)` should return the value 4.5. Even though we cannot formally have an anonymous function have more than one output, we can get around that.

Here's an example, where we define the anonymous function as our previously written function, `MyFunc.m`:

```
MyFunc2 = @(x,y) MyFunc(x,y);
```

Now in the command window, we can type: `[C,D]=MyFunc2(3,5);`, and you'll see that $C = 2$ and $D = 4$.

Passing a Function to a Function

In a numerical algorithm like the bisection method, we would want to pass a function into the algorithm so that the bisection algorithm can work for any function.

Here's an example. We'll define a function f, and we'll pass it into a function defined by an $M-$filel, then we'll call that function and see what happens.

First, use a text editor or Matlab's built-in editor to type our main function (like the bisection algorithm). Here's a short program that should take in a function f, and one value of x, then output $f(x)$ and $f(2)$:

```
function [y1,y2]=SampleFunc(f,x)
y1=f(x);  % This is regular function notation!
y2=f(2);  % This is just to get a 2d output
end
```

Save this as `SampleFunc.m`. Now define a function f in the command window or script file:

```
f = @(x) x^2-1;
```

To actually call our function, we would now type:

```
[C,D]=SampleFunc(f,0)
```

Then $C = -1$ and $D = 3$.

When the argument function is an m-file

When the function that is being passed into another function is written as an m-file (rather than as an anonymous function, we'll call it using the `@` symbol.

For example, if I have a function file saved as `parab.m` and I want to use the bisection method on it (`bisect.m`, defined below), I would either type this in in the command line or in a script file:

```
yOut=bisect(@parab,1,3,1e(-6)); %Note: 1e(-6) is 10^(-6)
```

Now we're ready to work with functions in our programs. First up will be the bisection algorithm. The inputs should be a function, together with two numbers a, b so that the zero of the function is in the interval $[a, b]$.

The Bisection Algorithm

```
function xc=bisect(f,a,b,tol)
% Bisection Method, xc=bisect(f,a,b,tol)
%  Computes an approximation to f(x)=0 given that the
%  root is bracketed in [a,b] with f(a)f(b)<0.  Will run
%  until TOL is reached, and will output the solution xc.
%
% EXAMPLE:  f=@(x) x^3+x-1;
%          xc=bisect(f,0,1,5e-5);
% Output:
%         Finished after 14 iterates
%         xc=0.6823


%Error check and initialization:
fa=f(a);  fb=f(b);
if sign(fa)*sign(fb)>=0
    error('Root is not be bracketed');
end

iter=0;
while (b-a)/2>tol
    iter=iter+1;
    c=(a+b)/2;
    fc=f(c);
    if fc==0  %This means that (a+b)/2 is the root-
        break     %Break out of the while loop and
                  %continue execution
    end
    if sign(fc)*sign(fa)<0  %New interval is [a,c] (reset b)
        b=c; fb=fc;
    else
        a=c; fa=fc;   %New interval is [c,b] (reset a)
    end
end
fprintf('Finished after %d iterates\n',iter);
xc=(a+b)/2;
end
```

The nice thing about the bisection method is that it is easy to implement. It takes a lot of iterations to converge, however. That leads us to Newton's Method. We'll have two forms - one for the "calc 1" version, and the other will be the multidimensional version.

Newton's Method part I

There's an example in the code to run. First, we'll need an m-file to hold the function that we're working with. Remember that we need to output both the function value and the value of the derivative. The following file would be saved as `MyFunc.m`.

```
function [y,dy]=MyFunc(x)
y=x^3+x-1;
dy=3*x^2+1;
end
```

Now save the code below as `NewtonMethod.m`

```
function out=NewtonMethod(F,x0,numits,tol)

for k=1:numits
   [g,gprime]=F(x0);
   if gprime==0
      error('Derivative is zero');
   end
   xnew=x0-g/gprime;

   d=abs(xnew-x0);
   if d<tol
      out=xnew;
      break
   end
   x0=xnew;
end
fprintf('Newton used %d iterations\n',k);
out=xnew;
end
```

And, as shown in the code, we can run an example by typing the following into the command window:

```
yout=NewtonMethod(@MyFunc,0,100,5e-5)
```

Multidimensional Newton's Method

In multidimensional Newton's Method, we'll assume that we have a function f for which we're trying to determine the roots of the gradient,

$$\nabla f(\mathbf{x}) = \vec{0}$$

In that case, the function file for f should actually output three items:

- $f(\mathbf{x})$ (which is a real number)
- $\nabla f(\mathbf{x})$ (which is a vector)
- $Hf(\mathbf{x})$, or the Hessian of f, which is a matrix of all of the second derivatives of f:

$$(Hf(\mathbf{x})_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}(\mathbf{x})$$

Here's the code for that=

```
function out=MultiNewton(F,x0,numits,tol)

for k=1:numits
   [g,gradg,hessg]=F(x0);
   if cond(hessg)>1000000
       error('The Hessian Matrix is not invertible');
   end
   xnew=x0-inv(hessg)*gradg;

   d=norm(xnew-x0);
   if d<tol
      out=xnew;
      break
   end
   x0=xnew;
end
fprintf('Newton used %d iterations\n',k);
out=xnew;
```

Here's a quick example. First, the function file:

```
function [y,dy,hy]=testfunc(x)
%  A test function for Newton's Method:
%  The input is the VECTOR x (with elements x,y below)
%
%     y = (1/4)x^4-(1/2)x^2+(1/2)y^2
%    dy = Gradient = [x^3-x; y]  (The gradient will output as a COLUMN)
%    hy = Hessian  = [3x^2-1, 0;0,1]

y=(1/4)*x(1)^4-(1/2)*x(1)^(2)+(1/2)*x(2)^2;
dy=[x(1)^3-x(1); x(2)];
hy=[3*x(1)^2-1, 0;0,1];
```

Now the function call would be something like

```
yout=MultiNewton(@testfunc,[-3;2],100,1e-6);
```

And the output will be: "Newton used 8 iterations", and `yout` would be $(-1, 0)$.