1 LAB2 (NEWTON METHOD)

1.1 SOLVING EQUATIONS NUMERICALLY

For the next few lectures we will focus on the problem of solving an equation:

$$f(x) = 0 \quad (3.1)$$

As you learned in calculus, the final step in many optimization problems is to solve an equation of this form where f is the derivative of a function, F, that you want to maximize or minimize. In real engineering problems the functions, f, you wish to find roots for can come from a large variety of sources, including formulas, solutions of differential equations, experiments, or simulations.

1.2 NEWTON ITERATIONS

We will denote an actual solution of equation (3.1) by x∗. There are three methods which you may have discussed in Calculus: the bisection method, the secant method and Newton's method. All three depend on beginning close (in some sense) to an actual solution x∗.

Recall Newton's method. You should know that the basis for Newton's method is approximation of a function by its linearization at a point, i.e.

$$f(x) = f(x_0) + f'(x_0)(\mathrm{x} - x_0) \quad (3.2)$$

Since we wish to find x so that f(x) = 0, set the left hand side (f(x)) of this approximation equal to 0 and solve for x to obtain:

$$x \approx x_0 - \frac{f(x_0)}{f'(x_0)} \qquad (3.3)$$

We begin the method with the initial guess x_0, which we hope is close to x∗. Then, we define a sequence of points {x_0,x1,x2,x3,...} from the formula:

$$x_{i+1} = x_i - \frac{f(x_0)}{f'(x_0)} \qquad (3.4)$$

which comes from (3.3). If f(x) is reasonably well-behaved near x∗ and x_0 is close enough to x∗, then it is a fact that the sequence will converge to x∗ and will do it very quickly.

1.3 THE LOOP: FOR ... END

To do Newton's method, we need to repeat the calculation in (3.4) several times. This is accomplished in a program using a loop, which means a section of a program which is repeated. The simplest way to accomplish this is to count the number of times through. In MATLAB, a **for ... end** statement makes a loop as in the following simple function program:

```
function S = mysum(n)
    % Gives the sum of the first n integers
    % Input: n
    % Output: the sum
    S = 0;                % start at zero
    % The loop:
    for i = 1:n           % do n times
        S = S + i;        % add the current integer
    end                   % end of the loop
end
```

Figure 1: Sum of n numbers using for loop

Using code snippet of function, shown in Figure 1, print sum of 100 integers in command window. Use **mysum** function.

The result will be the sum of the first 100 integers. All **for ... end** loops have the same format, it begins with for, followed by an index (i) and a range of numbers (1:n). Then come the commands that are to be repeated. Last comes the end command.

Loops are one of the main ways that computers are made to do calculations that humans cannot. Any calculation that involves a repeated process is easily done by a loop.

Now let's do a program that does n steps (iterations) of Newton's method. We will need to input the function, its derivative, the initial guess, and the number of steps. The output will be the final value of x, i.e. xn. If we are only interested in the final approximation, not the intermediate steps, which is usually the case in the real world, then we can use a single variable x in the program and change it at each step:

```
function x = mynewton(f,f1,x0,n)
    % Solves f(x) = 0 by doing n steps of Newton's method starting at x0.
    % Inputs: f -- the function
    %         f1 -- it's derivative
    %         x0 -- starting guess, a number
    %         n -- the number of steps to do
    % Output: x -- the approximate solution
    x = x0;                       % set x equal to the initial guess x0
    for i = 1:n                   % Do n times
        x = x - f(x)/f1(x)        % Newton's formula, prints x too
    end
end
```

Figure 2: Newton's method in MATLAB

Make sure you use **compact** and **long** formats discussed in lab1. For testing, define two user defined anonymous functions. One contains any functions like (x^3 - 5) and second equal to its derivative (3x^2). Use Newton method (hit and trial) and find lowest value of **n**, for which program converges (stops changing).

1.4 CONVERGENCE

Newton's method converges rapidly when f0(x∗) is nonzero and finite, and x0 is close enough to x∗ that the linear approximation (3.2) is valid. Let us look at what can go wrong. For f(x) = $x^{\frac{1}{3}}$ we have x∗ = 0 but f0(x∗) = ∞. If you try

$f(x)$ = @(x) $x^{\frac{1}{3}}$

$f'(x)$ = @(x) $\frac{1}{3}x^{\frac{-2}{3}}$

x = mynewton(f,f1 ,0.1,10)

then x explodes.

For f(x) = x^2 we have x∗ = 0 but f0(x∗) = 0. If you try

f = @(x) x^2

f1 = @(x) 2*x

x = mynewton(f,f1,1,10)

then x does converge to 0, but not that rapidly.

If x0 is not close enough to x∗ that the linear approximation (3.2) is valid, then the iteration (3.4) gives some x1 that may or may not be any better than x0. If we keep iterating, then either

• xn will eventually get close to x∗ and the method will then converge (rapidly), or

• the iterations will not approach x∗.

1.5 Measuring error and the Residual

If we are trying to find a numerical solution of an equation f(x) = 0, then there are a few different ways we can measure the error of our approximation. The most direct way to measure the error would be as {Error at step n} = en = xn −x∗ where xn is the n-th approximation and x∗ is the true value. However, we usually do not know the value of x∗, or we wouldn't be trying to approximate it. This makes it impossible to know the error directly, and so we must be cleverer.

One possible strategy, that often works, is to run a program until the approximation xn stops changing. The problem with this is that it sometimes doesn't work. Just because the program stops changing does not necessarily mean that xn is close to the true solution.

For Newton's method we have the following principle: At each step the number of significant digits roughly doubles. While this is an important statement about the error (since it means Newton's method converges really quickly), it is somewhat hard to use in a program.

Rather than measure how close xn is to x∗, in this and many other situations it is much more practical to measure how close the equation is to being satisfied, in other words, how close yn = f(xn) is to 0. We will use the quantity rn = f(xn)−0, called the residual, in many different situations. Most of the time we only care about the size of rn, so we use the absolute value of the residual as a measure of how close the solution is to solving the problem: |rn| = |f(xn)|.

1.6 The if ... end statement

If we have a certain tolerance for |rn| = |f(xn)|, then we can incorporate that into our Newton method program using an if ... end statement:

```
function x = mynewton(f,f1,x0,n,tol)
    % Solves f(x) = 0 by doing n steps of Newton's method starting at x0.
    % Inputs: f -- the function
    %         f1 -- it's derivative
    %         x0 -- starting guess, a number
    %         tol -- desired tolerance, prints a warning if |f(x)|>tol
    % Output: x -- the approximate solution
    x = x0;                    % set x equal to the initial guess x0
    for i = 1:n                % Do n times
        x = x - f(x)/f1(x)   % Newton's formula
    end
    r = abs(f(x))
    if r > tol
        warning('The desired accuracy was not attained')
    end
end
```

In this program if checks if abs(y) > tol is true or not. If it is true, then it does everything between there and end. If not true, then it skips ahead to end.

In the command window define a function and its derivative:

f = @(x) x^3-5

f1 = @(x) 3*x^2

Then, use the program with n = 3, tol = .01, and x0 = 2. Next, change tol to 10^−10 and repeat.

1.7 THE LOOP: WHILE ... END

While the previous program will tell us if it worked or not, we still must input n, the number of steps to take. Even for a well-behaved problem, if we make n too small then the tolerance will not be attained, and we will have to go back and increase it, or, if we make n too big, then the program will take more steps than necessary. One way to control the number of steps taken is to iterate until the residual |rn| = |f(x)| = |y| is small enough. In MATLAB, this is easily accomplished with a **while ... end** loop.

```
function x = mynewtontol(f,f1,x0,tol)
    % Solves f(x) = 0 using Newton's method until |f(x)| < tol.
    % Inputs: f -- the function
    %         f1 -- it's derivative
    %         x0 -- starting guess, a number
    %         tol -- desired tolerance, runs until |f(x)|<tol
    % Output: x -- the approximate solution
    x = x0;                    % set x equal to the initial guess x0
    y = f(x);
    while abs(y) > tol       % Do until the tolerence is reached.
        x = x - y/f1(x)      % Newton's formula
        y = f(x)
    end
end
```

The statement **while ... end** is a loop, like **for ... end**, but instead of going through the loop a fixed number of times it keeps going as long as the statement abs(y) > **tol** is true.

One obvious drawback of the program is that abs(y) might never get smaller than **tol**. If this happens, the program would continue to run over and over until we stop it. Try this by setting the tolerance to a really small number:

tol = 10^(-100)

then run the program again for f(x) = x3 −5. (You can use Ctrl-c to stop the program when it is stuck.) One way to avoid an infinite loop is add a counter variable, say i and a maximum number of iterations to the programs. Using the while statement, this can be accomplished as:

```
function x = mynewtontol(f,f1,x0,tol)
    % Solves f(x) = 0 using Newton's method until |f(x)| < tol.
    %     Safety stop after 1000 iterations
    % Inputs: f -- the function
    %         f1 -- it's derivative
    %         x0 -- starting guess, a number
    %         tol -- desired tolerance, runs until |f(x)|<tol
    % Output: x -- the approximate solution
    x = x0;    % set x equal to the initial guess x0.
    i=0;       % set counter to zero
    y = f(x);
    while abs(y) > tol & i < 1000
        % Do until the tolerence is reached or max iter.
        x = x - y/f1(x)     % Newton's formula
        y = f(x)
        i = i+1;            % increment counter
    end
end
```

1.8 Part A – Exercise (To Be Completed in Lab)

1. Enter: format long. Use **mynewton** on the function f(x) = x^5 −7, with x0 = 2. By trial and error, what is the lowest value of n for which the program converges (stops changing). Compute the error, which is how close the program's answer is to the true value. Compute the residual, which is the program's answer plugged into f. Are the error and residual zero?
2. Write a program that calculates value of π using following series. Calculate error from actual pi available in MATLAB (Absolute and relative)
$$\frac{\pi}{8} = \frac{1}{1 \times 3} + \frac{1}{5 \times 7} + \frac{1}{9 \times 11} + \cdots$$
3. Determine the largest relative error in a calculation of a cross-sectional area of a wire from a measurement of its diameter D, where D = .825 $\pm$ 0.002 cm
$Area = \pi\frac{D^2}{4}$

1.9 Home Assignment

1. In Calculus we learn that a geometric series has an exact sum

$$\sum_{i=0}^{\infty} r^i = \frac{1}{1-r},$$

provided that $|r| < 1$. For instance, if $r = .5$ then the sum is exactly 2. Below is a script program that lacks one line as written. Put in the missing command and then use the program to verify the result above. How many steps does it take? How close is the answer to 2?

```
% Computes a geometric series until it seems to converge
format long
format compact
r = .5;
Snew = 0;             % start sum at 0
Sold = -1;            % set Sold to trick while the first time
i = 0;                % count iterations
while Snew > Sold     % is the sum still changing?
   Sold = Snew;       % save previous value to compare to
   Snew = Snew + r^i;
   i=i+1;
Snew                  % prints the final value.
i                     % prints the # of iterations.
```

Add a line at the end of the program to compute the relative error of **Snew** versus the exact value. Run the script for $r = 0.9, 0.99, 0.999, 0.9999, 0.99999$, and 0.999999. In a table, report the number of iterations needed and the relative error for each r.

2. Suppose a ball is dropped from a height of 2 meters onto a hard surface and the coefficient of compensation of the collision is .9 (see Wikipedia for an explanation). Write a well-commented script program to calculate the total distance the ball has travelled when it hits the surface for the n-th time. Enter: format long. By trial and error approximate how large n must be so that total distance stops changing. Turn in the program and a brief summary of the results.
3. For f(x) = x3 −4, perform 3 iterations of Newton's method with starting point x0 = 2. (On paper, but use a calculator.) Calculate the solution (x∗ = 41/3) on a calculator and find the errors and percentage errors of x0, x1, x2 and x3. Use enough digits so that you do not falsely conclude the error is zero. Put the results in a table.
4. Modify your program from exercise 2 to compute the total distance travelled by the ball while its bounces are at least 1 millimetre high. Use a while loop to decide when to stop summing; do not use a for loop or trial and error. Turn in your modified program and a summary of the results.