Newton's Method

Background

It is a common task to find the roots of some equation by setting the equation equal to zero and then solving for the variable x. For linear equations like $y = x + 4$, this is a fairly trivial task. For equations like $y = x^2 - 3x + 4$, one must either factor the equation or use the quadratic formula to find the solutions. And for equations like $y = x\sin(x^2) + 5$, there is no mechanical way to determine the solutions.

Newton's Method is a way of finding approximations of the solutions without actually solving the equations. All that is required is knowledge of basic algebra, finding derivatives and Octave.

Newton's Method works by starting with a guess of a solution and then using that as input into a simple formula, which generates a rough approximation of one of the solutions of the equation. This rough approximation can be fed back into the formula to get a better approximation. The process can be repeated as many times as desired to get a very close approximation of the solution.

There is a catch, though. First, if an equation has multiple solutions (and most equations that would require Newton's Method will have multiple solutions), Newton's Method will only give one of the solutions for a given guess. To find the others, one must run Newton's Method with different guesses closer to the other solutions. This would require, for example, graphing the equation to see about where the solutions are, and then using those as guesses.

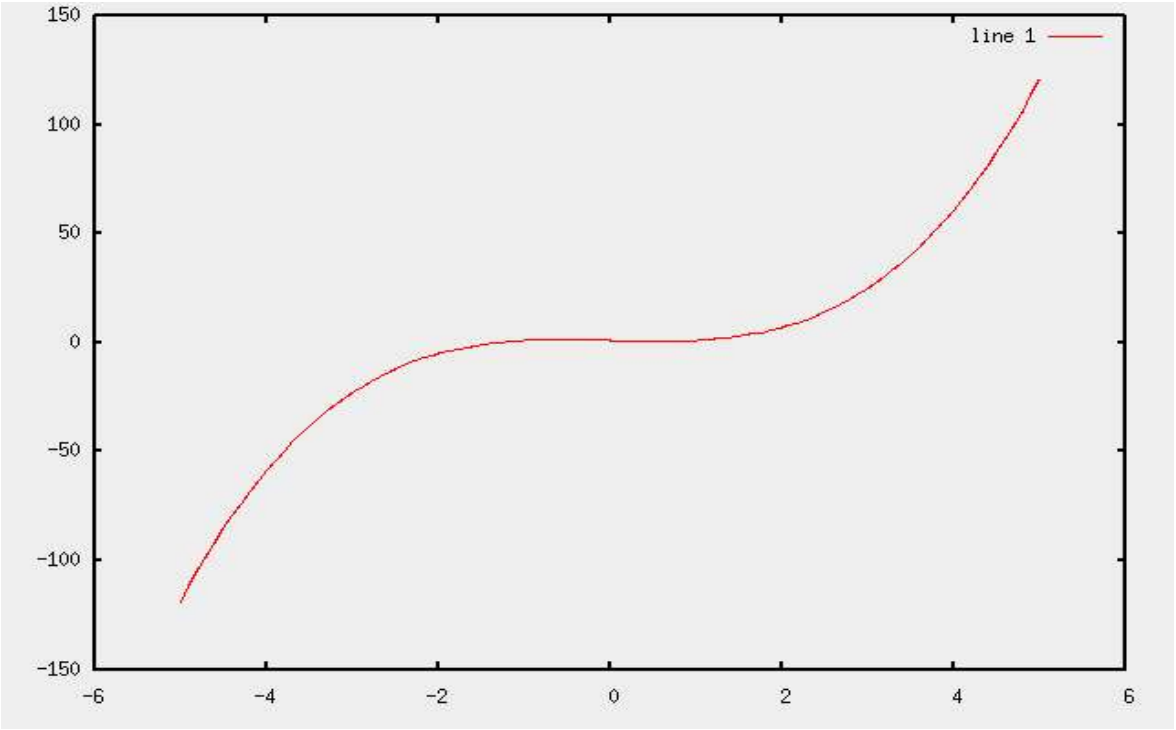
Second, Newton's Method may not generate a solution, even if there is a solution to be found. For example, the equation $y = x^3$, even though it has a solution (zero), Newton's Method will not yield a solution. How do we know that it doesn't yield a solution? If successive iterations of Newton's Method keep producing new numbers rather than numbers that are closer and closer to some value, then Newton's Method is not *converging* and that means that even if one were to continue interating forever, one would never reach a stable solution.

So how does Newton's Method work? It consists of a single formula shown below:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_{n+1})}$$

For some function $f(x)$ and its derivative $f'(x)$.

Let's run through a basic example using the equation $y = x^3 - x + 1$. A good first step is to graph the function to get an idea of where the solution(s) may be.



We can see that the line crosses the *x*-axis at around -1.5, so we will use that as our guess. Plugging this into the formula given above, we get:

$$x_0 = -1.5$$

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = x_0 - \frac{{x_0}^3 - x_0 + 1}{3{x_0}^2 - 1} = -1.5 - \frac{(-1.5)^3 - (-1.5) + 1}{3(-1.5)^2 - 1} = -1.5 - \frac{-0.875}{5.75} = -1.347826$$

Now we use -1.347826 as input to the formula again in order to get a better answer:

$$x_2 = -1.347826 - \frac{(-1.347826)^3 - (-1.347826) + 1}{3(-1.347826)^2 - 1} = -1.325200$$

$$x_3 = -1.325200 - \frac{(-1.325200)^3 - (-1.325200) + 1}{3(-1.325200)^2 - 1} = -1.324718$$

$$x_4 = -1.324718 - \frac{(-1.324718)^3 - (-1.324718) + 1}{3(-1.324718)^2 - 1} = -1.324717$$

$$x_5 = -1.324717 - \frac{(-1.324717)^3 - (-1.324717) + 1}{3(-1.324717)^2 - 1} = -1.324717$$

Notice that by the last iteration, the result of the formula was the same as the input (out to six digits). Thus, we know the approximation has gotten very close to the real answer. If we instead saw the numbers changing wildly after every iteration without getting closer to anything, then we would know that our guess was bad (or that the equation has no solution).

Objectives

1. Octave
   a) Functions
   b) For-loops
2. General
   a) Basic calculus
   b) Solving non-linear equations
   c) Approximations

Octave Constructs

Functions

Octave functions work like functions in mathematics and functions in traditional programming languages. A function takes some number of inputs (arguments), performs some calculations and then returns some result. Let's consider a simple function, which would be defined by the following code in Octave:

```
function y = f(x)
      y = x + 2;
endfunction
```

The first line describes the function, giving information about the inputs to the function, the return value and the name of the function. In this case, the function is named "f", takes a single argument "x" and returns a single value "y". The second line performs a (very simple) calculation using x and stores the result in y. By storing the result in y, the result of that calculation is returned to the caller of the function. This differs from traditional programming languages where there is a return-statement that explicitly sets the return value and exits the function.

For-loops

A for-loop basically contains a block of code that is executed multiple times. A for-loop also contains a special variable called an iterator, which is modified by the loop itself on every iteration of the loop. It is fairly common to have a for-loop run a piece of code n times with the iterator starting at 1 and being incremented every time through the loop until it finally reaches n. Octave makes for-loops fairly straightforward. The general structure of a for-loop is as follows:

```
for i=a:b
      <code>
endfor
```

This fragment executes code $b - a$ times, incrementing the variable i on every loop iteration (i starts with the value a and ends with the value b). The variable i can be used within the loop body.

Step-by-Step

1) Define the function to be solved. This can be done from the Octave command line or in a .m file. In either case, using the function given in the introduction, the Octave commands would look like this:

```
function y = f(x)
y = x .^ 3 – x + 1;
endfunction
```

2) Define the derivative of the function to be solved. This step is required because Octave does not yet have any facilities for doing symbolic differentiation.

```
function y = fprime(x)
y = 3 .* (x .^ 2) – 1;
endfunction
```

3) Plot a graph of the function to see where the roots are and if there even are any roots.

```
x = linspace(-10,10,100);
y = f(x);
plot(x,y)
```

4) Using the graph, make a guess as to where one of the solutions is.

```
x(1) = <guess>;
```

5) Run Newton's Method using a for-loop.

```
for i = 1:10
x(i+1) = x(i) – f(x(i)) / fprime(x(i+1));
endfor
```