# Project 2 - Nonlinear Solvers

**Taylor Ellington**

October 8, 2015

In this project, we will be examining the algorithms behind the regular Newton's method and a variant of Newton's method that utilizes the finite-difference.

## 1 Newton's Method

To start off, I implemented the algorithm for Newton's method that we covered in classnewton.cpp

```
#pragma once

#include "Fcn.cpp"
#include <iostream>
#include <cmath>

using namespace std;

double  newton(Fcn& f, Fcn& df, double x, int maxit, double tol, bool show_iterates){

    double H;
    double X_K = x;
    double X_K_1;

    for(int i = 0; i < maxit; i++){
        H = f(X_K) / df(X_K);
        X_K_1 = X_K - H;

        if(show_iterates)
        cout << "\t" << i << " - Current Guess: " << X_K <<
            " Solution update magniutde: " << fabs(H) << " Residual: " << f(X_K) << endl;

        if( fabs( X_K_1  - X_K ) < tol){
            if(show_iterates)
                cout << "\t Tolerance Reached" << endl;

            break;
        }

        if(  i == maxit-1 && show_iterates == true )
            cout << "\t Max iterations reached" << endl;

        X_K = X_K_1;
    }

    return X_K;

}
```

This implementation accepts two Fcn, function objects, an initial guess double, an int representing the maximum number of iterations the method can loop through, a double to denote the desired tolerance, and a control flag for verbose output

Then to test this code, I built a program that supplied inital conditions with a variety of paramaters. To get a feel for how Newton's method behaved, I tested the root finding problem $f(x) = x(x-3)(x+1) = 0$. I ran nine tests total, checking every purmutaion of three initial guesses and three sets of tolerances. Only the number of iterations, control flag, and functions $f(x)$ and $f'(x)$ did not vary."' #include "newton.cpp" #include "Fcn.cpp" #include

using namespace std;

class F : public Fcn { double operator() (double x){ return x * ( x - 3) * (x + 1); } };

class FD : public Fcn { double operator() (double x){ return 3*x*x - 4 * x - 3; } };

int main(int argc, char * argv[]){

```
F f;
FD fd;
double x[] = {-2.0, 1.0, 2.0};
double tol[] = {1e-1, 1e-5, 1e-9};
int maxit =  15;
bool show_iterates =  true;

cout << "test" << endl;

//run every permutation
for(int i = 0; i < 3; i++){
    for( int j = 0; j < 3; j++){
        cout << "Experement with X: " << x[i] << ",  and tolerance: "<< tol[j] << endl;
        cout << "\tresult:" <<  newton(f, fd, x[i], maxit, tol[j], show_iterates) << endl;
    }
}

    return 0;
```

}"'The immediately observable trend in the results of this experiment, is that as the tolerance decreases the number of iterations increases, approaching the maximum number of iterations set in code. Interestingly enough, the initial guesses we feed the algorithm are conditioned such that we never hit our maximum. As we vary our initial guess, we end up finding different roots, as this function has three roots, and each guess puts the algorithm near a different root.

## 2 Finite-Difference Newton Method

This section of the project necessitates a slightly different approach to Newton's method. Instead of using the derivative, we will instead use the forward finite difference as part of our correcting term $h$. The implementation is similar to before, however, instead of a second Fcn term, we accept a double to represent the increment term $alpha$

```
#include "Fcn.cpp"
#include <iostream>
#include <cmath>
```

```
using namespace std;

double ffd(Fcn& f, double x, double alpha){

    return (f(x-alpha) - f(x) ) / alpha;
}


double  fd_newton(Fcn& f, double x, int maxit, double tol, double alpha,  bool show_iterate

    double H;
    double X_K = x;
    double X_K_1;

    for(int i = 0; i < maxit; i++){
        H = 0 - (f(X_K) / ffd(f, X_K, alpha));
        X_K_1 = X_K - H;

        if(show_iterates)
        cout << "\t" << i << " - Current Guess: " << X_K <<
            " Solution update magniutde: " << fabs(H) << " Residual: " << f(X_K) << endl;

        if( fabs( X_K_1  - X_K ) < tol){
            if(show_iterates)
                cout << "\t Tolerance Reached" << endl;

            break;
        }

        if(  i == maxit-1 && show_iterates == true )
            cout << "\t Max iterations reached" << endl;

        X_K = X_K_1;

    }

    return X_K;

}
```

The program that tests this function, is also similar to before. However, this time we will run twenty seven tests, there will be three initial guesses, three tolerance values, and now we will add three differencing parameters alpha.

```
#include "fd_newton.cpp"
#include "Fcn.cpp"
#include <iostream>

using namespace std;

class F : public Fcn
{

    double operator() (double x){
       return  x * ( x - 3) * (x + 1);
    }

};


int main(int argc, char * argv[]){



    F f;
    double x[] = {-2.0, 1.0, 2.0};
    double tol[] = {1e-1, 1e-5, 1e-9};
    double alpha[] = {pow(2, -4), pow(2, -26), pow(2, -50)};
    int maxit =  20;
    bool show_iterates =  true;

    cout << "test" << endl;

    //run every permutation
    for(int i = 0; i < 3; i++){
        for( int j = 0; j < 3; j++){
            for(int k = 0; k < 3; k++){
                cout << "Experement with X: " << x[i] <<" Alpha: "<< alpha[k] << ",  and to
                cout << "\t==result:" <<  fd_newton(f, x[i], maxit, tol[j], alpha[k], show_
            }
        }
    }

        return 0;

}
```

My first impression is that Newton's method requires fewer iterations than the finite difference Newton's method. However finite difference offers us a solution to a hypothetical problem where the derivative of $f(x)$ is impossible or very costly to compute. The second thing I notice is that a smaller $alpha$ value is correlated to a smaller number of iterations.

## 3 Application

Now we will use the newton method from part one to solve out the positions of an object around a eliptical orbit. This will be acomplished in the following steps

1. For every point in our range $\{0, 0.001, \ldots\ 10\}$ use Newton's method to solve for the position of the object
2. convert the radial position into cartesian coordinates
3. Save everything to disk

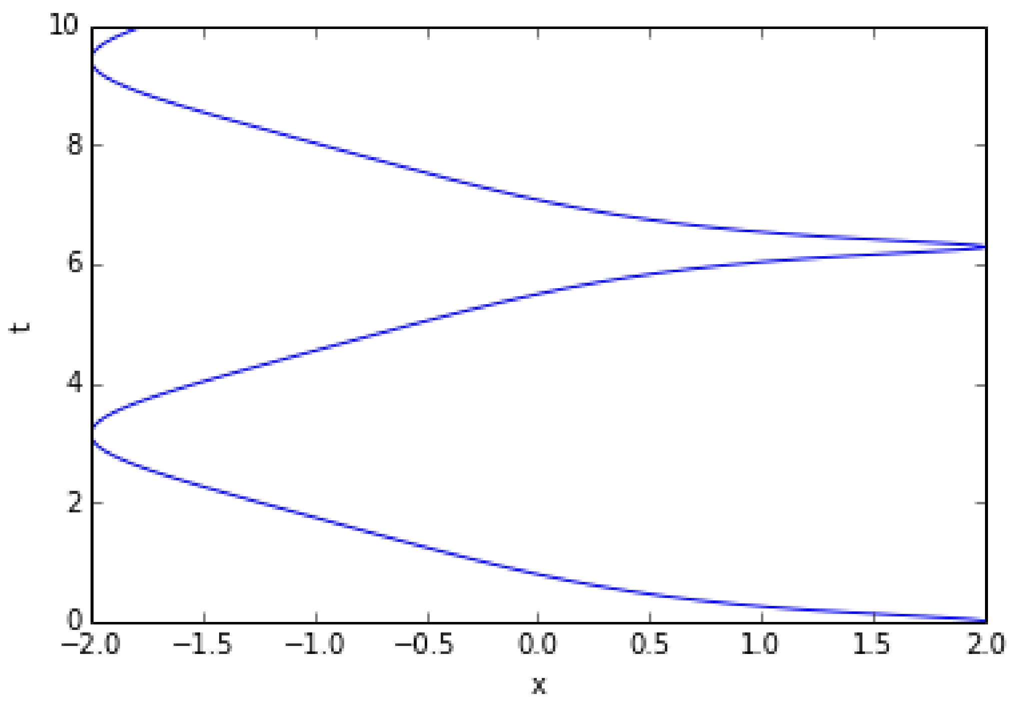
```
#include "matrix.hpp"
#include "Fcn.cpp"
#include "newton.cpp"
#include <math.h>
#include <stdio.h>
class F: public  Fcn{

    double t = 0; // supress warnings
    double e = 0;
public:
    double operator() (double x){
       return  e *sin(x) - x - t;
    }
    void  setT(double t){
        this->t = t;
    }
    void solveE(double a, double b){
        e = sqrt(  1 - ( (b*b) / (a*a) )  );
    }

};

class FD: public Fcn{

    double t = 0; // supress warnings
    double e = 0;
public:
    double operator() (double x){
       return  e * cos(x) - 1;
    }
    void  setT(double t){
        this->t = t;
    }
    void solveE(double a, double b){
        e = sqrt(  1 - ( (b*b) / (a*a) )  );
    }

};

double radialPosition( double a, double b, double w){

    return (a * b) / sqrt ( (b*cos(w) )*(b*cos(w)) + ( a* sin(w))*(a* sin(w))  );
}

int main(int argc, char * argv[]){
    double time_min = 0;
    double time_max  = 10;

    F f;
    FD fd;

    f.solveE(2.0, 1.25);
    fd.solveE(2.0, 1.25);

    Matrix t = Linspace(time_min, time_max, 1, 10000);
    Matrix x(t.Size() );
    Matrix y(t.Size() );

    double w = 0;
    double rw = 0;

    for(int i = 0; i < t.Size() ; i++){
        f.setT( t(i) );
        fd.setT( t(i) );

        w = newton(f,fd, w, 6, 1e-5, false);
        rw = radialPosition(2.0, 1.25, w);
        x(i) = rw * cos(w);
        y(i) = rw * sin(w);

    }

    t.Write("t.txt");
    x.Write("x.txt");
    y.Write("y.txt");


    return 0;
}
```

One of the complications I encountered, was that our implementation of Newton's method had no space to accept a changing value for t, it was only set to accept the assigned set of values. Rather than create a new version of Newton's method, or commit the programming faux pas of global variables I found a way to sneak my t value in. I added a bit of state and accompanying methods to the Fcn classes for $f(x)$ and $f'(x)$. The method `setT(double t)` exists entirely so that I can pass in the right value of t each time the program iterates.

Once the data is saved, we load it up into python and produce several visualizatons, including the object's path along the x and y axis with respect to time and the complete elipse that represents the orbit of our object.

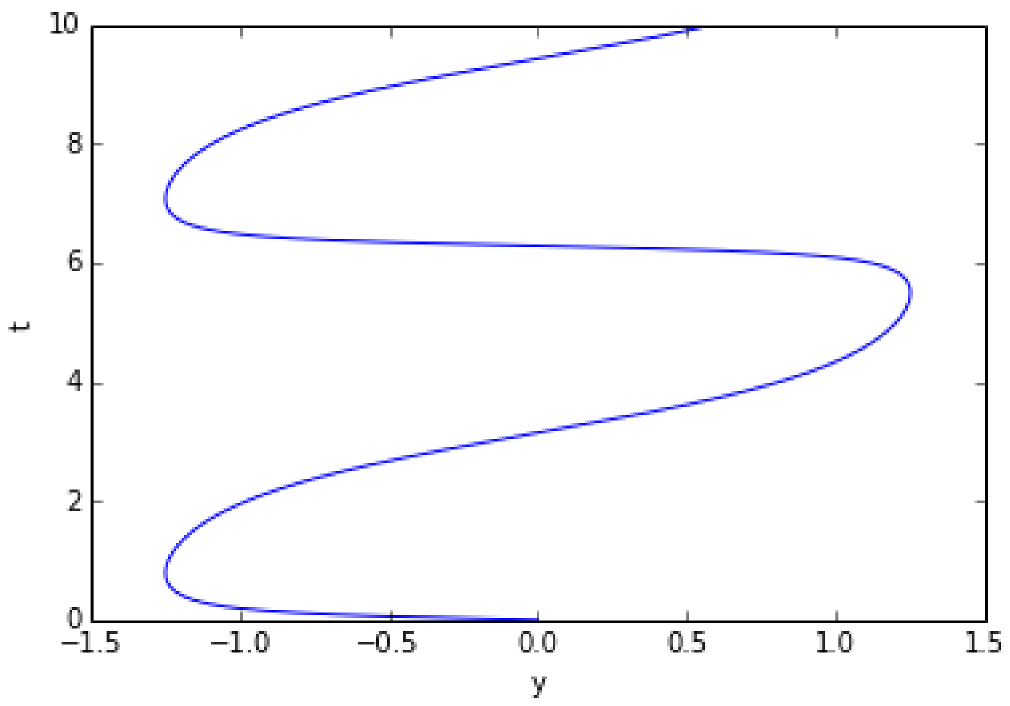
In [1]:
```
import numpy as np
t = np.loadtxt("t.txt")
y = np.loadtxt("y.txt")
x = np.loadtxt("x.txt")
```

In [2]:
```
%matplotlib inline
import matplotlib.pyplot as plt

plt.plot(x, t)
plt.xlabel("x")
plt.ylabel("t")

plt.show()
```

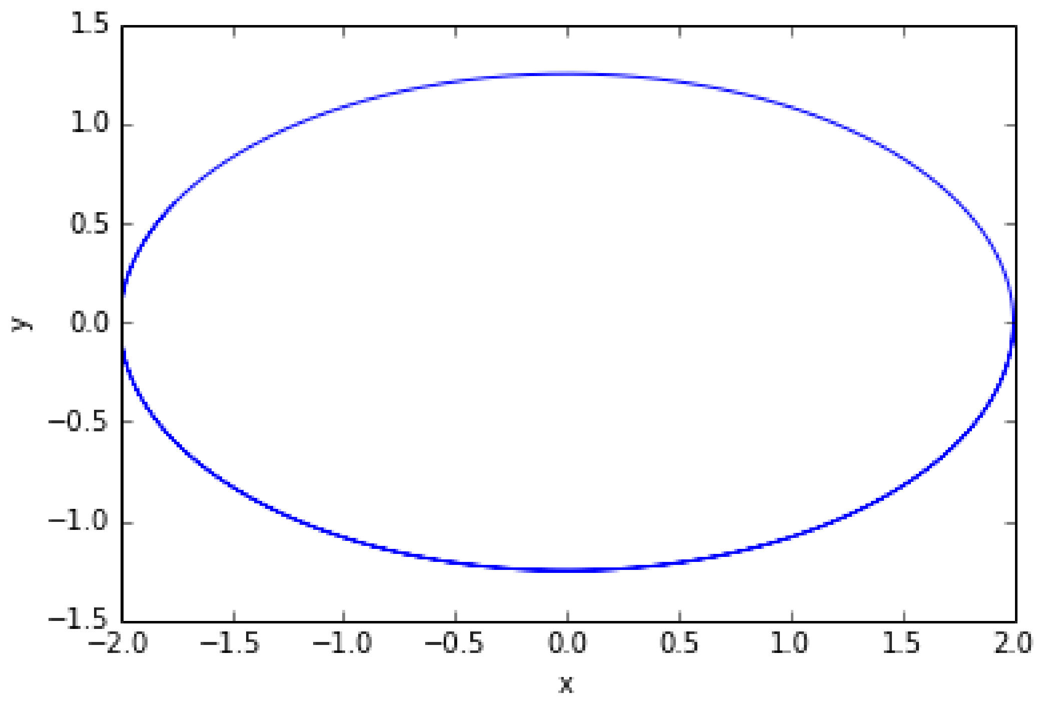


In [3]:

```
%matplotlib inline
import matplotlib.pyplot as plt

plt.plot(y, t,)
plt.xlabel("y")
plt.ylabel("t")

plt.show()
```



In [8]:
```
%matplotlib inline
import matplotlib.pyplot as plt

plt.plot(x, y)
plt.xlabel("x")
plt.ylabel("y")

plt.show()
```



## 4 Makefile

```
################################################################################
# Project 2 Makefile
#
# Taylor Ellington
# Scientific HPC
# Fall 2015
################################################################################

# compiler & flags
CXX = g++
CXXFLAGS = -O -std=c++11
###################################

All : test_newton.exe test_fd_newton.exe kepler.exe

test_newton.exe :test_newton.cpp Fcn.o
	$(CXX)$(CXXFLAGS) $^ -o$@

test_fd_newton.exe : test_fd_newton.cpp Fcn.o
	$(CXX)$(CXXFLAGS) $^ -o$@
```

```
kepler.exe : kepler.cpp matrix.o
    $(CXX)$(CXXFLAGS) $^ -o$@

newton.o : newton.cpp
    $(CXX)$(CXXFLAGS) -c $< -o$@

fd_newton.o : fd_newton.cpp
    $(CXX)$(CXXFLAGS) -c $< -o$@

Fcn.o : Fcn.cpp
    $(CXX)$(CXXFLAGS) -c $< -o$@

matrix.o : matrix.cpp matrix.hpp
    $(CXX)$(CXXFLAGS) -c $< -o$@




clean :
    rm *.exe
    rm *.o
    rm *.txt
```

```
In []:
```