Project 2 Report

October 5, 2016

Ben Maxey
MATH 3316

1 Project 2 - Linear and Nonlinear Solvers

1.1 Vandermonde Matrix

In this section, we were tasked with solving linear systems whose coefficients are given by a Vandermonde matrix. An (nxn) Vandermonde matrix A is defined as $A(i,j) = v_{i-1}^{j-1}$, for $i = 1, 2, ..., n$, $j = 1, 2, ..., n$, and v is given by the vector $\mathbf{v} = [v_0, v_1, ..., v_{n-1}]^T$. In this project, $n = 5, 9, 17, 33, 65$, and **v** is a column vector with n equally-spaced entries from 0 to 1. For each value of n, a random column vector with n entries was generated using the Matrix class's built-in Random() function. These vectors are the known solutions (x) for each value of n, and the right hand side vector (b) was set as the product of A and x. After solving each system, the error vectors were calculated by subtracting the calculated solutions from the previously generated right hand side vectors, and the residual vectors were calculated by multiplying each original coefficient matrix by the error vectors. Finally, the norms of each error and residual vector were calculated using the Matrix class's built-in Norm() function. To solve each system, I implemented a function that uses Gaussian elimination with partial pivoting and backward substitution to determine x given A and b. The function is given below.

```
In [ ]: # %load vandermonde.cpp

        Matrix solve(Matrix A, Matrix b)
        {
                Matrix x(b.Rows(), 1);

                //Gaussian elimination with partial pivoting.
                for (int k = 0; k < A.Cols() - 1; k++)
                {
                        //Find largest entry in current column.
                        int m = k;
                        for (int i = k + 1; i < A.Cols(); i++)
                        {
                                if (abs(A(i, k) > abs(A(m,k))))
                                        m = i;
                        }
                        //Swap row containing max entry with row k.
                        for (int i = k; i < A.Cols(); i++)
                        {
                                double temp = A(m, i);
                                A(m, i) = A(k, i);
                                A(k, i) = temp;
                        }

                        double temp = b(m, 0);
                        b(m, 0) = b(k, 0);
                        b(k, 0) = temp;

                        //Carry out elimination.
                        for (int i = k + 1; i < A.Cols(); i++)
                        {
                                A(i, k) /= A(k, k);

                                for (int j = k + 1; j < A.Cols(); j++)
                                        A(i, j) -= A(i, k) * A(k, j);

                                b(i, 0) -= A(i, k) * b(k, 0);
                        }
                }

                //Backward substitution.
                for (int i = A.Cols() - 1; i >= 0; i--)
                {
                        double sum = 0;

                        for (int k = i + 1; k < A.Cols(); k++)
                                sum += A(i, k) * x(k, 0);

                        x(i, 0) = (b(i, 0) - sum) / A(i, i);
                }

                return x;
        }
```

This function passes both matrices by value so that the original matrices remain unaltered and can be reused in error calculation later. In the elimination section of the function, for step k, the largest entry in column k is selected. The row containing this entry is swapped with row k. Then, regular elimination is carried out for column k. After the elimination stage A is upper triangular, so backward substitution can be used to find b. The remainder of vandermonde.cpp is shown below.

```
In [ ]: # %load vandermonde.cpp
```

```
#include <cmath>

#include "matrix.hpp"

//************************************************************************
//        Function definitions.
//************************************************************************

//Generates a Vandermonde matrix of order (size).
Matrix generate(size_t size)
{
        Matrix v = Linspace(0, 1, 1, size);
        Matrix A(size, size);

        for (int i = 0; i < size; i++)
        {
                for (int j = 0; j < size; j++)
                        A(i, j) = pow(v(i), j);
        }

        return A;
}

/* solve() function defined here */

//Generates all matrices and vectors, solves, then calculates error and
//residual vectors.
int main(int argc, char* argv[])
{
        Matrix A5 = generate(5);
        Matrix A9 = generate(9);
        Matrix A17 = generate(17);
        Matrix A33 = generate(33);
        Matrix A65 = generate(65);

        Matrix x5 = Random(5, 1);
        Matrix x9 = Random(9, 1);
        Matrix x17 = Random(17, 1);
        Matrix x33 = Random(33, 1);
        Matrix x65 = Random(65, 1);

        Matrix b5 = A5 * x5;
        Matrix b9 = A9 * x9;
        Matrix b17 = A17 * x17;
        Matrix b33 = A33 * x33;
        Matrix b65 = A65 * x65;

        Matrix sol5 = solve(A5, b5);
```

```
                Matrix sol9 = solve(A9, b9);
                Matrix sol17 = solve(A17, b17);
                Matrix sol33 = solve(A33, b33);
                Matrix sol65 = solve(A65, b65);

                Matrix err5 = x5 - sol5;
                Matrix err9 = x9 - sol9;
                Matrix err17 = x17 - sol17;
                Matrix err33 = x33 - sol33;
                Matrix err65 = x65 - sol65;

                Matrix res5 = A5 * err5;
                Matrix res9 = A9 * err9;
                Matrix res17 = A17 * err17;
                Matrix res33 = A33 * err33;
                Matrix res65 = A65 * err65;

                std::cout << "Error norm with n = 5: " << Norm(err5) << std::endl;
                std::cout << "Error norm with n = 9: " << Norm(err9) << std::endl;
                std::cout << "Error norm with n = 17: " << Norm(err17) << std::endl
                std::cout << "Error norm with n = 33: " << Norm(err33) << std::endl
                std::cout << "Error norm with n = 65: " << Norm(err65) << std::endl

                std::cout << "Resid norm with n = 5: " << Norm(res5) << std::endl;
                std::cout << "Resid norm with n = 9: " << Norm(res9) << std::endl;
                std::cout << "Resid norm with n = 17: " << Norm(res17) << std::endl
                std::cout << "Resid norm with n = 33: " << Norm(res33) << std::endl
                std::cout << "Resid norm with n = 65: " << Norm(res65) << std::endl
        }
```

The error of each solve was calculated by subtracting the solutions from the known x vectors. The residual vectors were calculated by multiplying each coefficient matrix with its corresponding error vector. The results of this program were redirected into a text file, "vandermonde.txt," which is shown below.

```
In [ ]: # %load vandermonde.txt
        Error norm with n = 5: 2.82112e-14
        Error norm with n = 9: 5.01012e-11
        Error norm with n = 17: 0.00194135
        Error norm with n = 33: 411.436
        Error norm with n = 65: 2339.96
        Resid norm with n = 5: 5.67799e-16
        Resid norm with n = 9: 6.3029e-16
        Resid norm with n = 17: 2.8805e-15
        Resid norm with n = 33: 7.08161e-14
        Resid norm with n = 65: 8.38081e-13
```

The Vandermonde matrix is notoriously ill-conditioned. A measure of the conditioning of a matrix is the "condition number," which is calculated as follows: $k(A) = ||A||_2||A^{-1}||_2$. After solving each system, I calculated the condition number of each Vandermonde matrix using the following code. The results are given below the code.

```
In [ ]: Matrix A5i = Inverse(A5);
        Matrix A9i = Inverse(A9);
        Matrix A17i = Inverse(A17);
        Matrix A33i = Inverse(A33);
        Matrix A65i = Inverse(A65);

        double k5 = Norm(A5) * Norm(A5i);
        double k9 = Norm(A9) * Norm(A9i);
        double k17 = Norm(A17) * Norm(A17i);
        double k33 = Norm(A33) * Norm(A33i);
        double k65 = Norm(A65) * Norm(A65i);

        std::cout << "Condition number with n = 5: " << k5 << std::endl;
        std::cout << "Condition number with n = 9: " << k9 << std::endl;
        std::cout << "Condition number with n = 17: " << k17 << std::endl;
        std::cout << "Condition number with n = 33: " << k33 << std::endl;
        std::cout << "Condition number with n = 65: " << k65 << std::endl;

        //******************************************************************
        //******************************************************************

        Condition number with n = 5: 749.246
        Condition number with n = 9: 2.23975e+06
        Condition number with n = 17: 2.77633e+13
        Condition number with n = 33: 107.027
        Condition number with n = 65: 234.662
```

It might seem odd that for $n = 33$ and $n = 65$ the condition number is so low; however, these condition numbers were never actually calculated. After calling the Inverse() function, errors were thrown for A_{33} and A_{65} because they were singular. Thus, the condition number of both A_{33} and A_{65} can be taken as infinity. The condition number is directly proportional to the upper bound on the error. Thus, because the condition number grows exponentially as n increases, the error in solving these systems also grows exponentially as n increases. Although the residual vector norms are negligible compared to the error vectors, they cannot be used as an indication of the solving method's accuracy. The residual vector can be interpreted as the perturbation in b required for the calculated solutions ($sol5$, $sol9$, etc.) to be the true solutions of the system. Because the system is ill-conditioned, small perturbations in b can cause large changes in the solutions (i.e. small residual vectors can still imply large absolute errors for ill-conditioned systems).

1.2 Newton's Method

In this section, I implemented Newton's method for approximating nonlinear functions and tested the accuracy of this method. The newton() function approximates a zero of a function given the function, its derivative, an initial guess, a maximum number of iterations, a maximum error tolerance, and a flag that determines whether details of each iteration are printed. The newton()

function uses the fixed-point iteration $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$ to approximate a zero of $f(x)$ to a given tolerance. The iteration ceases when the difference between x_{n+1} and x_n is less than the given tolerance. If "show_iterates" is true, the function will print the iteration count, the current guess, the solution update, and current residual to the console.

```
In [ ]: # %load newton.cpp

        #include "newton.hpp"

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

                for (int i = 0; i < maxit; i++)
                {
                        double fp = df(x);
                        double d = fx/fp;
                        x -= d;
                        fx = f(x);

                        if (d < 0)
                                d = 0 - d;

                        if (show_iterates)
                        {
                                std::cout << "----------------------------------" <
                                                  << "Iteration " << i + 1 << ": "
                                                  << "\tCurrent guess: x = " << x <
                                                  << "\tSolution update: |h| = " <<
                                                  << "\tCurrent residual: f(x) = "
                                                  << std::endl;
                        }

                        if (d < tol)
                        {
                                if (show_iterates)
                                {
                                        std::cout << "\tConverged at x = " << x <<
                                        std::cout << "----------------------------
                                }
                                return x;
                        }
                }
        }
```

The function used to test the newton() method is $f(x) = x^2(x-3)(x+2)$, for which the zeros are $x = 3$, $x = -2$, and $x = 0$. $f(x)$ is realized by the definition of a subclass of the Fcn class, which was provided to us. The function will be run for three different initial guesses (-3, 1, 2) and three different tolerances (10^{-1}, 10^{-5}, 10^{-9}), for a total of 9 runs. The maximum number of iterations in each run is set at 50. The test_newton program is shown below.

```
In [ ]: # %load test_newton.cpp

        #include <iostream>
        #include <cmath>

        #include "newton.hpp"
        #include "fcn.hpp"

        //*************************************************************************
        //        Subclass definitions.
        //*************************************************************************

        //f(x) = x^2(x - 3)(x + 2) = x^4 - x^3 - 6x^2
        class Func : public Fcn
        {
        public:
                double operator() (double x)
                {
                        return pow(x, 2) * (x - 3) * (x + 2);
                }
        };

        //f'(x) = 4x^3 - 3x^2 - 12x
        class Deriv : public Fcn
        {
        public:
                double operator() (double x)
                {
                        return 4 * pow(x, 3) - 3 * pow(x, 2) - 12 * x;
                }
        };

        //*************************************************************************
        //        Function definition.
        //*************************************************************************

        int main(int argc, char* argv[])
        {
                Func fx;
                Deriv fp;

                double z1 = newton(fx, fp, -3, 50, pow(10, -1), true);
                double z2 = newton(fx, fp, 1, 50, pow(10, -1), true);
                double z3 = newton(fx, fp, 2, 50, pow(10, -1), true);
                double z4 = newton(fx, fp, -3, 50, pow(10, -5), true);
                double z5 = newton(fx, fp, 1, 50, pow(10, -5), true);
                double z6 = newton(fx, fp, 2, 50, pow(10, -5), true);
                double z7 = newton(fx, fp, -3, 50, pow(10, -9), true);
                double z8 = newton(fx, fp, 1, 50, pow(10, -9), true);
                double z9 = newton(fx, fp, 2, 50, pow(10, -9), true);
        }
```

Because they are several hundred lines long, the results from test_newton.cpp are not included in this report, but a brief description of the results is warranted. Test 1 converged to -2.00045 in 4 iterations, Test 2 converged to 0.057983 in 4 iterations, Test 3 converged to -2 in 2 iterations, Test 4 converged to -2 in 6 iterations, Test 5 converged to 7.1416e-6 in 17 iterations, Test 6 converged to -2 in 2 iterations, Test 7 converged to -2 in 7 iterations, Test 8 converged to 8.71761e-10 in 30 iterations, and Test 9 converged to -2 in 2 iterations. The full details of the results can be seen by running test_newton after compiling the included source files.

First, it is worth noting that Tests 3, 6, and 9 all converge in 2 iterations because the exact zero is found in two iterations. Choosing $x = 2$ as the initial guess results in the second iteration returning $x_2 = -2$. Thus, if 2 is the initial guess, changing the tolerance does not influence the number of iterations required for convergence. In the case of Tests 1, 4, and 7, the number of iterations required for convergence is dependent on the error tolerance as the exact zero is never found. Because the error decreases quadratically for each iteration, one would expect that if the tolerance decreases from 10^{-1} to 10^{-5}, the number of iterations should increase by approximately 2 or 3:

$$|r - x_n| < 10^{-1}$$
$$|r - x_{n+1}| < C_1(10^{-1})^2 = C_1(10^{-2})$$
$$|r - x_{n+2}| < C_2(10^{-4})$$
$$|r - x_{n+3}| < C_3(10^{-8})$$
$$|r - x_{n+4}| < C_4(10^{-16})$$

Thus, it makes complete sense that Test 4 took only 2 more iterations than Test 1 and that Test 7 took only 1 more iteration than Test 4. For Tests 2, 5, and 8, the rate of convergence is linear because $x = 0$ is a double root. The root $x = 3$ was never found in these tests because the initial guesses used did not lead to this root. A different choice of initial guess would be required for newton() to find $x = 3$.

1.3 Application: Kepler's Equation

An application of nonlinear solvers is Kepler's equation describing an object's position in an elliptical orbit: $\epsilon sin(w) - w = t$, where $\epsilon = \sqrt{1 - \frac{b^2}{a^2}}$ is the object's orbital eccentricity, t is proportional to time, and w is the angle of the object of the object around its elliptical orbit. For this project, a is set as 2.0 and b is set as 1.25. The ultimate goal of this problem is to be able to describe the x and y position of an object at any time t. The first step is to determine a residual function that can be used to solve for w for each value of t. This can be accomplished simply by subtracting t from both sides of the equation given above: $f(w) = \epsilon sin(w) - w - t$, $f'(w) = \epsilon cos(w) - 1$. An interesting problem arose in dealing with t. Because Fcn objects can only take one parameter in the () operator, I set t as a member variable of the derived class, Func. Each time a Func object was instantiated, a value for t was passed to it in its constructor. A new Func object was instantiated for each value of t. Once I had obtained $f(w)$ and $f'(w)$, solving for w was simple. The newton() function written in Part 2 was called for each $t = 0, 0.001, ..., 10$ with a tolerance of 10^{-5} and a maximum of 6 iterations. The calculated angles were stored in the vector w. The radial position of the object could then be determined from the equation $r(w) = \frac{ab}{\sqrt{(bcos(w))^2+(asin(w))^2}}$. Each r was stored in a vector. Finally, the x and y positions were calculated by evaluating the functions $x(t) = rcos(w)$ and $y(t) = rsin(w)$ for all w. The code for this part is shown below.

```
In [ ]: # %load kepler.cpp

        #include <iostream>
        #include <cmath>

        #include "matrix.hpp"
        #include "newton.hpp"
        #include "fcn.hpp"

        //**************************************************************************
        //        Subclass definitions.
        //**************************************************************************

        //f(x) = [(1 - b^2/a^2)^(1/2)]sin(x) - x - t
        //b = 1.25, a = 2, t = time
        class Func : public Fcn
        {
        private:
                double t;
                double e;
        public:
                Func()
                {
                        t = 0;
                        e = sqrt(1 - (1.25 * 1.25)/(4));
                }

                Func(double time)
                {
                        t = time;
                        e = sqrt(1 - (1.25 * 1.25)/(4));
                }

                double operator() (double x)
                {
                        return e * sin(x) - x - t;
                }
        };
```

```
//f'(x) = [(1 - b^2/a^2)^(1/2)]cos(x) - 1
class Deriv : public Fcn
{
private:
        double e = sqrt(1 - (1.25 * 1.25)/(4));
public:
        double operator() (double x)
        {
                return e * cos(x) - 1;
        }
};

//r(x) = ab/{[(bcos(x))^2 + (asin(x))^2]^(1/2)}
//b = 1.25, a = 2
class Radial : public Fcn
{
public:
        double operator() (double x)
        {
                double denominator = sqrt(pow(1.25 * cos(x), 2) + pow(2 * s
                return 2.5 / denominator;
        }
};

//************************************************************************
//        Function definitions.
//************************************************************************

//Uses newton() function to determine a zero of Kepler's equation for each
//the given times.  The initial guess for each solve is the result of the
//previous solve.
Matrix solve(int initialGuess, Matrix& times)
{
        Matrix output(1, times.Cols());
        double initial = initialGuess;

        for (int i = 0; i < times.Cols(); i++)
        {
                double time = times(i);

                Func f(time);
                Deriv df;

                output(i) = initial = newton(f, df, initial, 6, pow(10, -5)
        }

        return output;
}
```

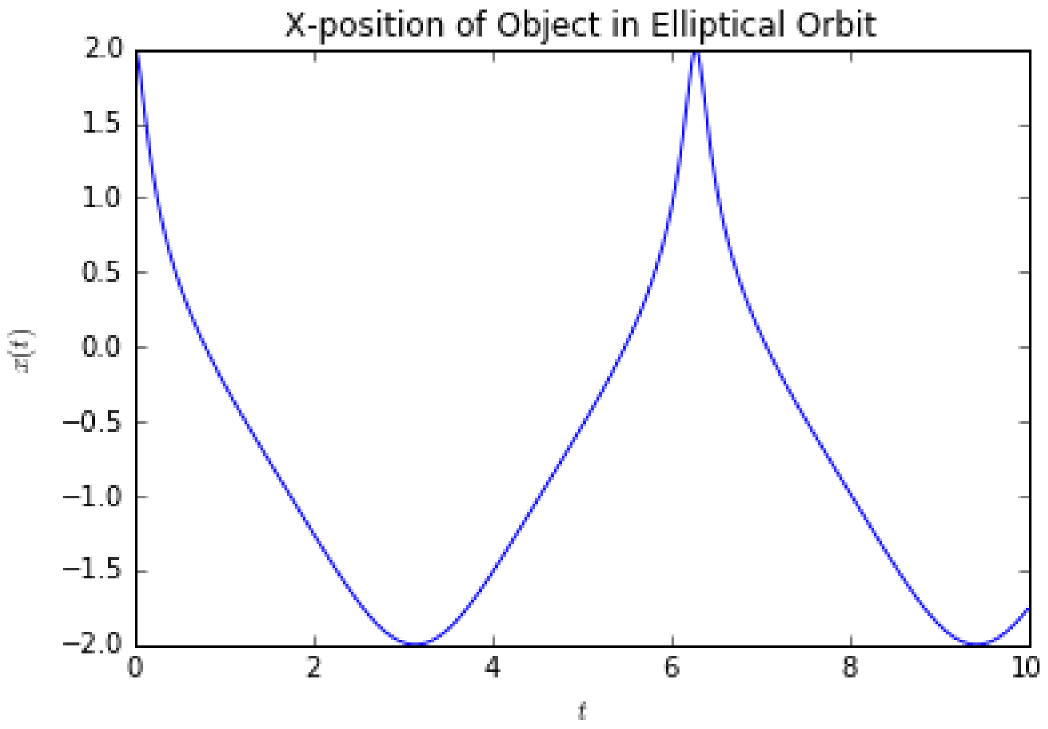
```
//Calculates the x value of the object at each angle.
Matrix computeX(Matrix& angles)
{
        Matrix output(1, angles.Cols());
        Radial r;

        for (int i = 0; i < angles.Cols(); i++)
                output(i) = r(angles(i)) * cos(angles(i));

        return output;
}

//Calculates the y value of the object at each angle.
Matrix computeY(Matrix& angles)
{
        Matrix output(1, angles.Cols());
        Radial r;

        for (int i = 0; i < angles.Cols(); i++)
                output(i) = r(angles(i)) * sin(angles(i));

        return output;
}

//Generates times from 0 to 10, then determines angles and positions.
int main(int argc, char** argv)
{
        Matrix times = Linspace(0, 10, 1, pow(10,4) + 1);
        times.Write("t.txt");

        Matrix w = solve(0, times);
        w.Write("w.txt");

        Matrix x = computeX(w);
        x.Write("x.txt");

        Matrix y = computeY(w);
        y.Write("y.txt");
}
```

The results for this part are reasonably straightforward. The graphs of $x(t)$ versus t, $y(t)$ versus t, and $y(t)$ versus $x(t)$ are given below. The x values range from -2 to 2, and the y values range from -1.25 to 1.25, which is consistent with the values of b and a. This consistency suggests that the results are accurate.

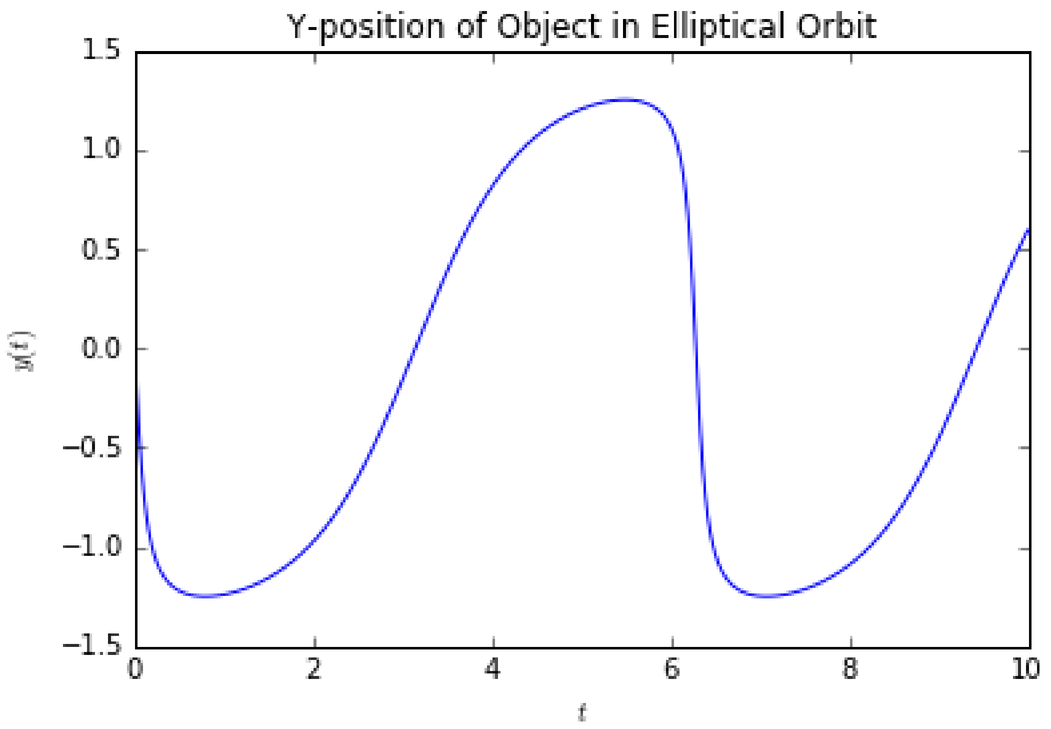
```
In [1]: %pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

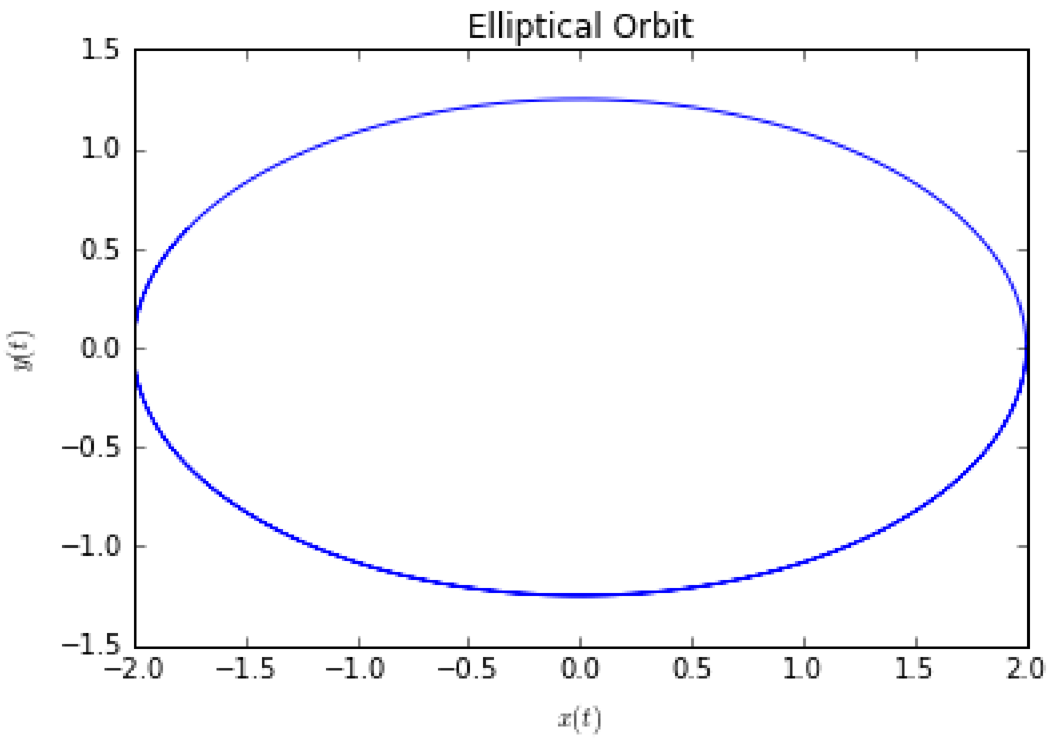
```
In [2]: t = loadtxt('t.txt')
        x = loadtxt('x.txt')
        y = loadtxt('y.txt')

In [3]: plot(t,x)
        xlabel('$t$')
        ylabel('$x(t)$')
        title('X-position of Object in Elliptical Orbit')

Out[3]: <matplotlib.text.Text at 0x7f56a12aa668>
```



```
In [4]: plot(t,y)
        xlabel('$t$')
        ylabel('$y(t)$')
        title('Y-position of Object in Elliptical Orbit')

Out[4]: <matplotlib.text.Text at 0x7f569e681860>
```



```
In [5]: plot(x,y)
        xlabel('$x(t)$')
        ylabel('$y(t)$')
        title('Elliptical Orbit')

Out[5]: <matplotlib.text.Text at 0x7f569e5f27b8>
```



1.4 Appendix

1.4.1 Makefile

```
In [ ]: # %load Makefile

        STD = -std=c++11

        all: vandermonde test_newton kepler

        vandermonde: vandermonde.o matrix.o
                g++ $^ -o $@ $(STD)

        test_newton: test_newton.o newton.o
                g++ $^ -o $@ $(STD)

        kepler: kepler.o newton.o matrix.o
                g++ $^ -o $@ $(STD)

        vandermonde.o: vandermonde.cpp
                g++ -c vandermonde.cpp $(STD)

        matrix.o: matrix.cpp
```

```
	g++ -c matrix.cpp $(STD)

test_newton.o: test_newton.cpp
	g++ -c test_newton.cpp $(STD)

newton.o: newton.cpp
	g++ -c newton.cpp $(STD)

kepler.o: kepler.cpp
	g++ -c kepler.cpp $(STD)

clean:
	rm -f *o
	rm -f *txt
```