Project 2

Stefan Popov

Building the project.

This project comes with a provided Makefile, as specified in the instructions. To build all executables for this project, simply use the command

make

in Terminal or Command Prompt.

The executable

kepler.exe

would need to be manually executed to create the required files for this project.

To redirect the other files' output to a file, simply use the commands

./vandermonde.exe > "vandermonde.txt"

and

./test_newton.exe > "test_newton.txt"

in the Terminal or Command Prompt.

To delete all output files simply use the command

make clean

To delete all output files and executables, type in

make realclean

1. Vandermonde Matrix

For the first part of this project, we were to write a C++ program vandermonde.cpp that would solve a linear system $ Ax=b $ with the Vandermonde matrix A of order n as the coefficient matrix for each $ n = 5, 9, 17, 33, 65 $.

The Vandermonde matrix A, which is a well-known ill-conditioned matrix, was to be generated by the vector v with n equally-spaced entries between 0 and 1 (including 0 and 1).

A random vector x with n random entries was generated using a built-in function of the provided Matrix class.The vector b was then constructed, so that x would be the solution to $Ax=b$.

The general linear solver in the Matrix class was used to find the solution $\hat{x}$ of $Ax=b$.

We were then to find the 2-norm of the error vector and residual vector.

First of all to create the Vandermonde matrix a function $createVandermonde$ was defined that would take in a vector of the Linspace from 0 and 1 split into n equally-spaced entries between 0 and 1.

The function definition is as follows :

In [ ]:

```
Matrix createVandermonde(Matrix& v){
    vector< vector<double> > vandermonde;
 vector<double> temp;
    for(float i=0.0; i<(float)v.Cols(); i+=1.0){
        for(int j=0; j<v.Cols(); j++){
         temp.push_back(powf((float)v[j][0],i));
        }
        vandermonde.push_back(temp);
  temp.clear();
 }
    Matrix v1(vandermonde);
    return v1;
}
```

Here, we would create a vector of vector of doubles called vandermonde and a vector of double called temp. We would then iterate over the number of columns v has and in the inner loop we would do the same, but with another variable (called j).

We would then append the j-th element of the vector raised to the power of i to the temporary vector. At the end of the inner loop, we would append temp to the vandermonde vector of vectors and clear temp. At function exit, we would create a Matrix object using the vector of vectors of doubles and

return it.

Another helper method is defined that would return a vector from the Matrix passed.

In [ ]:

```
vector<double> getVector(Matrix x){
 vector<double> returnVec;
    for(int i=0; i<x.Rows(); i++){
  returnVec.push_back(x[0][i]);
 }
 return returnVec;

}
```

In our main method, we would declare our n and vandermonde Matrices, as well as the random vector x and the vectors v used for creating the Vandermonde matrices.

In [ ]:

```
int main(){


 double n[5]= {5,9,17,33,65};

    Matrix v1 = Linspace(0.0, 1.0, 1, (size_t)n[0]);
    Matrix v2 = Linspace(0.0, 1.0, 1, (size_t)n[1]);
    Matrix v3 = Linspace(0.0, 1.0, 1, (size_t)n[2]);
    Matrix v4 = Linspace(0.0, 1.0, 1, (size_t)n[3]);
    Matrix v5 = Linspace(0.0, 1.0, 1, (size_t)n[4]);

    Matrix x1 = Random((size_t)n[0],1);
    Matrix x2 = Random((size_t)n[1],1);
    Matrix x3 = Random((size_t)n[2],1);
    Matrix x4 = Random((size_t)n[3],1);
    Matrix x5 = Random((size_t)n[4],1);

    Matrix a1 = createVandermonde(v1);
    Matrix a2 = createVandermonde(v2);
    Matrix a3 = createVandermonde(v3);
    Matrix a4 = createVandermonde(v4);
    Matrix a5 = createVandermonde(v5);
```

We would then set up b, so that x is the solution of $Ax=b$.

In [ ]:

```
    Matrix b1 = a1*x1;
    Matrix b2 = a2*x2;
    Matrix b3 = a3*x3;
    Matrix b4 = a4*x4;
    Matrix b5 = a5*x5;
```

Afterwards, the $\hat{x}$ is determined using the provided Solve() method in the Matrix class.

In [ ]:

```
    Matrix x1Hat = Solve(a1,b1);
    Matrix x2Hat = Solve(a2,b2);
    Matrix x3Hat = Solve(a3,b3);
    Matrix x4Hat = Solve(a4,b4);
    Matrix x5Hat = Solve(a5,b5);
```

The error and residual vectors were obtained using the getVector function :

In [ ]:

```
 vector<double> error1Vector = getVector(x1Hat - x1);
 vector<double> error2Vector = getVector(x2Hat - x2);
 vector<double> error3Vector = getVector(x3Hat - x3);
 vector<double> error4Vector = getVector(x4Hat - x4);
 vector<double> error5Vector = getVector(x5Hat - x5);

 vector<double> residual1 = getVector(createVandermonde(v1)*x1Hat -
createVandermonde(v1)*x1);
 vector<double> residual2 = getVector(createVandermonde(v2)*x2Hat -
createVandermonde(v2)*x2);
 vector<double> residual3 = getVector(createVandermonde(v3)*x3Hat -
createVandermonde(v3)*x3);
 vector<double> residual4 = getVector(createVandermonde(v4)*x4Hat -
createVandermonde(v4)*x4);
 vector<double> residual5 = getVector(createVandermonde(v5)*x5Hat -
createVandermonde(v5)*x5);
```

Finally, the 2-norm of the error and residual vectors is printed to the screen.

In [ ]:

```
    cout<<Norm(error1Vector)<<" "<<Norm(residual1)<<endl;
    cout<<Norm(error2Vector)<<" "<<Norm(residual2)<<endl;
    cout<<Norm(error3Vector)<<" "<<Norm(residual3)<<endl;
    cout<<Norm(error4Vector)<<" "<<Norm(residual4)<<endl;
    cout<<Norm(error5Vector)<<" "<<Norm(residual5)<<endl;
```

The output during one of the runnings was as follows :

In [ ]:

```
2.82112e-14 0
1.61079e-11 1.08921e-15
6.83844e-06 1.60119e-15
3.54709 25.394
5.01578 50.2465
```

This shows us that as the vector that builds the Vandermonde matrix increases in size, the 2-norm of the error and residual vectors for that solution increase. This is logical, since it is expected for the 2-norm of the error vector to increase as we increase the size of the Vandermonde matrix. For the Vandermonde matrix of size 5, we achieved a 2-norm of the residual vector and it steadily increased as we increased the size of the vector we passed in as an argument to create the above mentioned Vandermonde matrix.

2. Newton's Method

For the second part of this assignment, we were supposed to create a new C++ file named newton.cpp that contains a function to perform the Newton method to approximate the root of a provided function to a specified tolerance.

The solver function was supposed to have the calling syntax

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

where f defines the nonlinear residual function to solve, df is the derivative of that function, x is the initial guess, maxit is the number of allowed iterations, tol is the desired solution tolerance, and show iterates is a flag to enable/disable printing of iteration information to the screen during the Newton solve.

At each iteration, if show iterates is “true” the method we wrote should output the current iteration index, the current solution guess, x, the absolute value of the solution update, |h|, and the absolute value of the current residual, |f(x)|.

After that we were to create a C++ program test_newton.cpp to verify that the newton function works as desired on the root-finding problem $$f(x) = x^2(x−3)(x+2) = 0.$$

For the tests, we were to start with initial guesses of $x_0 = {−3, 1, 2}$, and solve the problem to tolerances of $ ε = {10^{−1}, 10^{−5}, 10^{−9}} $. All of these tests were to set show iterates to true and allow a maximum of 50 iterations.

The newton() function was written following the pseudocode provided in the book and was as follows:

In [ ]:

```
double newton(Fcn& f, Fcn& df, double x, int maxit, double tol, bool show_it
erates){
 double x1=0.0;
 if(show_iterates){
  cout<<"Iteration \t x \t f(x1)"<<endl;
 }
 for ( int i=1; i< maxit; i++){
  double y0 = f(x);
  double yp = df(x);
  x1= x-y0/yp;

  if(show_iterates){
   cout<<i<<"\t"<<x1<<"\t"<<f(x1)<<endl;
  }
  if(abs(f(x1))<tol)
   break;
  else{
   x=x1;
  }
 }
 return x1;
}
```

In the test_newton.cpp file, I started by declaring $f(x)$ and $f'(x)$:

In [ ]:

```
class f: public Fcn{
public:
  double operator()(double x) {   // function evaluation
    return x*x*(x-3)*(x+2);
  }
};

class derf: public Fcn{
public:
  double operator()(double x) {   // function evaluation
    return x*(4*x*x - 3*x - 12);
  }
};
```

In the main method, the function and its derivative were declared, as well as the tolerance array and the array of initial guesses.Then using a loop, the newton method was called for each pair.

In [ ]:

```
int main(){
 f func;
 derf derfunc;
 double tol[3]= { pow(10.0,-1.0),pow(10.0,-5.0),pow(10.0,-9.0)};
 double x0[3]= {-2,1,2};
 for (int i=0; i<3;i++){
  for(int j=0;j<3;j++){
   cout<<"Calling newton for x0= "<<x0[i]<<" with tolerance tol= "<<tol[j]<
<endl;
            cout<<"The result yielded was : "<<newton(func,derfunc,x0[i],50
,tol[j],true)<<endl<<endl;
        }
 }
}
```

The yielded output from the main method was :

In [ ]:

```
Calling newton for x0= -3 with tolerance tol= 0.1
Iteration   x   f(x1)
1 -2.45455 14.9375
2 -2.14186 3.3464
3 -2.01957 0.400736
The result yielded was : -2.01957

Calling newton for x0= -3 with tolerance tol= 1e-05
Iteration   x   f(x1)
1 -2.45455 14.9375
2 -2.14186 3.3464
3 -2.01957 0.400736
The result yielded was : -2.01957

Calling newton for x0= -3 with tolerance tol= 1e-09
Iteration   x   f(x1)
1 -2.45455 14.9375
2 -2.14186 3.3464
```

```
2  2.14186  3.5464
3 -2.01957 0.400736
The result yielded was : -2.01957

Calling newton for x0= 1 with tolerance tol= 0.1
Iteration   x   f(x1)
1 0.454545 -1.2909
2 0.228022 -0.321116
The result yielded was : 0.228022

Calling newton for x0= 1 with tolerance tol= 1e-05
Iteration   x   f(x1)
1 0.454545 -1.2909
2 0.228022 -0.321116
The result yielded was : 0.228022

Calling newton for x0= 1 with tolerance tol= 1e-09
Iteration   x   f(x1)
1 0.454545 -1.2909
2 0.228022 -0.321116
The result yielded was : 0.228022

Calling newton for x0= 2 with tolerance tol= 0.1
Iteration   x   f(x1)
1 -2 -0
The result yielded was : -2

Calling newton for x0= 2 with tolerance tol= 1e-05
Iteration   x   f(x1)
1 -2 -0
The result yielded was : -2

Calling newton for x0= 2 with tolerance tol= 1e-09
Iteration   x   f(x1)
1 -2 -0
The result yielded was : -2
```

As we can see from the output, as the initial guess increases, we get a decrease in the number of iterations needed, as we approach to -2, which is a root, since it takes 1 iterations to return the value. We do not get any change in iterations as tolerance increases, since we get as close to the root just from a 10^{-1} tolerance on the iterations.

3. Application

For the last part of this project, we were to solve the Kepler equation

$$ \epsilon sin(\omega) - \omega = t$$ where $$ \epsilon = \sqrt{1 - \frac{b^2}{a^2}} $$

t is proportional to time and $ \omega $ is the angle of the object around its elliptical orbit.

The parameters a and b were set to 2.0 and 1.25, respectively.

In a C++ file called kepler.cpp, we were to solve the Kepler equation for a derived $f(\omega)$ using Newton's method, which was previously written for Part 1 of this project, for each t in $[0, 0.001, ... ,10]$.

$f(\omega)$ was gotten by setting Kepler equation to 0. From there, $f'(\omega)$ was then derived by taking the derivative of $f(\omega)$ with respect to ω.

The classes for both of those functions were declared in the kepler.cpp as a derived class of the provided Fcn abstract class. Here, a and b and t were declared as private data members, since the solver for the function only accepts one argument - ω, which is called x here for simplicity. In the fcn class, there is an additional setT method that is called upon each iteration later in main.

In [ ]:

```
class fcn: public Fcn{
public:
   double operator()(double x) {   // function evaluation
     return x - epsilon*sin(x)  - t;
   }
   void setT(double t){
    this->t=t;
   }
 private:

 const double a = 2.0;
 const double b = 1.25;
 double epsilon = sqrt(1-(b*b)/(a*a));
 double t;
};

class derfcn: public Fcn{
public:
 double operator()(double x) {   // function evaluation
     return 1 - epsilon*cos(x) ;

private:
 const double a = 2.0;
 const double b = 1.25;
 double epsilon = sqrt(1-(b*b)/(a*a));
};
```

Part A.

In part A, we would iterate through the Matrix Linspace to solve Newton's method for the Kepler equation for each t. Each solve allowed 6 iterations, did not print output information to the screen and was set to a tolerance of 10^{-5}.

The initial guess for each ω was the previous t, with 0 being the very first ω. This was easily achieved using the provided matrix class and declaring all variables, as well as initializing the Linspace.

The for loop iterated over the values of t and solved the newton equation for that t and stored it in a vector, which would be used later in part B. The value of ω was reset at each iteration.

In [ ]:

```
int main(){
 //declare function and derivative
 fcn f;
 derfcn df;

 //declare tolerance, iterations coutn and iterates
 const double tolerance = pow(10.0,-5.0);
 const bool show_iterates = false;
 const int iterations = 6;

 //declare a, b and derive epsilon
 const double a = 2.0;
 const double b = 1.25;
 double epsilon = sqrt(1-(b*b)/(a*a));

 //declare the initial w and the Linspace of t
 double w = 0.0;
 Matrix ts = Linspace(0,10,1,(size_t)10000);

 //declare a vector of double to store results from the newton method
 vector<double> ws;

 //loop over all values of t, reset t in the function class and push the
    //result of Newton's method to the vector of results. Reset w
 for( int i =0 ; i<ts.Cols();i++){
  f.setT(ts[0][i]);
  ws.push_back(newton(f,df,w,  iterations, tolerance,show_iterates));
  w = ts[0][i];
 }
```

Part B.

In part B, we were to use the computed radial position to compute the cartesian Coordinates for an object at angle ω using the formula

$$r(\omega) = \frac{ab} { \sqrt{(bcos(\omega))^2 + (asin(\omega)^2}}$$

with $x(t) = rcos(\omega)$ and $y(t) = rsin(\omega)$. This was done by iterating over the values of ω from part A.

In [ ]:

```
 //convert x and y coordinates to radial
 vector<double> cartesianxs;
 vector<double> cartesianys;

 double r;

 //iterate over all values of w, compute r and the cartesian coordinates an
d
 for( int i = 0 ; i < ws.size(); i++){

  r = (a*b) / (sqrt( pow(b * cos(ws[i]),2) + pow(a * sin(ws[i]),2) ));


  cartesianxs.push_back(r*cos(ws[i]));
  cartesianys.push_back(r*sin(ws[i]));
```

```
}
```

Part C.

For the last part we were to save our results in a Matrix and write to a file. This was done easily using the provided functionality of the Matrix class.

```
//create matrix from vector Matrix xs(cartesianxs); Matrix ys(cartesianys); //write to file xs.Write("x.txt"); ys.Write("y.txt"); ts.Write("t.txt");
```

The first plot required from the above is $x(t)$ vs. t.

In [40]:

```
%pylab inline
t = loadtxt('t.txt')
x = loadtxt('x.txt')
y = loadtxt('y.txt')
```

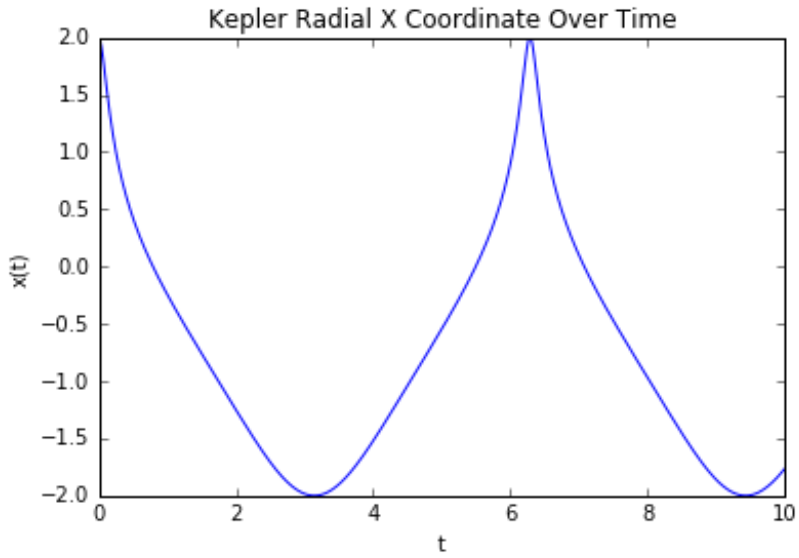
```
Populating the interactive namespace from numpy and matplotlib
```

In [41]:

```
plot(t,x)
xlabel('t')
ylabel('x(t)')
title('Kepler Radial X Coordinate Over Time')
```

Out[41]:

```
<matplotlib.text.Text at 0x7f850ec7ec88>
```



These are the expected results for x, since we can see that it has a period of 6 and looks like a sinusoid in shape, because it repeats. Since we used $r*cos(\omega)$ to get this x-coordinate, we are to expect this specific behaviour, as we are dealing with radial positions, since ω is an angle of the elliptical orbit of the Kepler function.

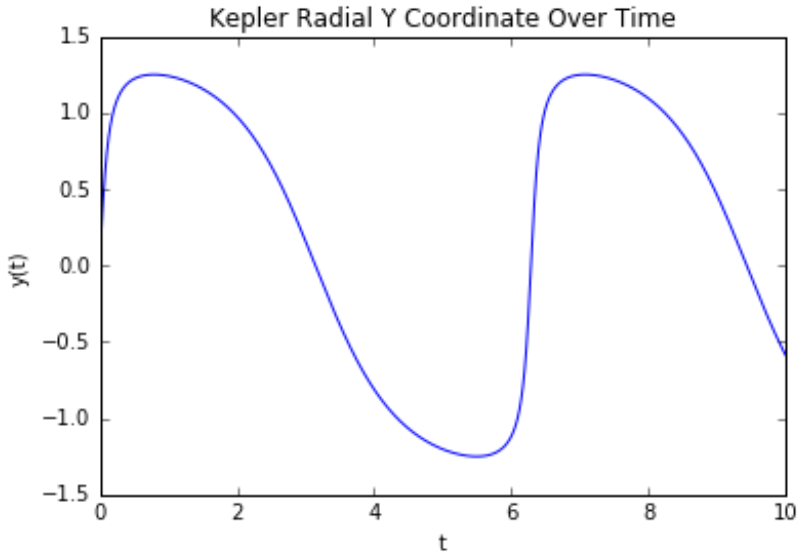
In [42]:

```
plot(t,y)
xlabel('t')
ylabel('y(t)')
```

```
ylabel('y(t)')
title('Kepler Radial Y Coordinate Over Time')
```

Out[42]:

```
<matplotlib.text.Text at 0x7f850ec22c18>
```



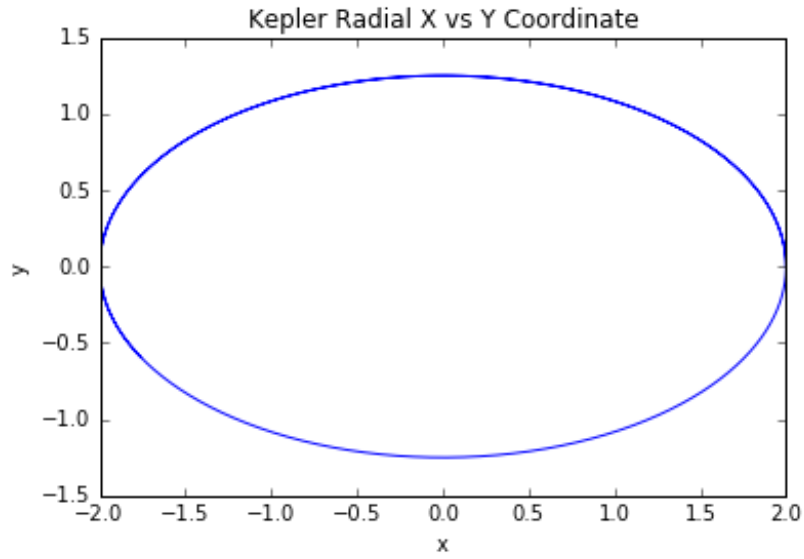
Similarly, y gives expected values and has the same period as x. The same behaviour is observed, since we are again dealing with radial positions, since ω is an angle of the elliptical orbit of the Kepler function.

In [43]:

```
plot(x,y)
xlabel('x')
ylabel('y')
title('Kepler Radial X vs Y Coordinate')
```

Out[43]:

```
<matplotlib.text.Text at 0x7f850ec01e48>
```



The X vs Y plot shows us that for each value of x there are two corresponding values of y, that have the same magnitude, but different sign. This is why we get an oval as a result from the plot. We recreate the original elliptical orbit that the Kepler function would create in the given time interval that we specified in our input. This is to represent the Kepler function on the time interval from $[0,10]$

and initial ω of 0.0.

4. Makefile

The following Makefile was used to ease the compilation of all the files in the project.

In [ ]:

```
# compiler & flags
CXX = g++
CXXFLAGS = -O -std=c++11
#CXXFLAGS = -O0 -g -std=c++11

# makefile targets
all : vandermonde.exe test_newton.exe kepler.exe

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

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

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

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

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

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

clean :
 \rm -f *.o *.out a_data *.txt

realclean : clean
 \rm -f *.exe *~
```