MTH6150: Numerical Computing in C and C++

Coursework 2020

Sajid Ahmed
170140500

Question 1. Self-consistent iteration.

The transformation between the two coordinate systems, x = x(r), is given by x = r + ln |r − 1| . (1) We seek the inverse transformation, r = r(x). Write a C++ function rofx that takes x as argument, solves Eq. (1) for r via self-consistent iteration, and returns the value of r. Run your code for a value of x > 2. In how many iterations did it converge? What is the final value of r? Substitute r into Eq. (1): is it satisfied? [10]

Firstly, I defined all the modules that I would need for the entire coursework. I also defined values EP as epsilon and pi. **[CODE]**

```
#include <iostream>
#include <cmath>
#include <algorithm>
#include <vector>
#include <random>
#include <iomanip>
#include <ctime>
#include <math.h>
#define EP 0.000000000001
#define pi 3.14159265358979323846
using namespace std;
```

Here I defined the function rofx with void type. **[CODE]**

```
double rofx(const double x) {
	int counter = 0; //This counter is to count the number of
iterations the programme loops
	const double GM = 0.5;
	double r0 = 0.8 * x;
	double r1 = x - 2 * GM * abs(log(r0 / (2 * GM) - 1));
	while (abs(r1 - r0) >= EP) { //Defining the while statement to
produce a loop that stops once it is greater than or equal to epsilon
		r0 = r1;
		r1 = x - 2 * GM * abs(log(r0 / (2 * GM) - 1));
		counter++; //If the loop is successful, it is finished off
by adding one to the counter representing one iteration step complete
	}
	cout << "The root (in terms of r) is: " << r1 << endl;
//Printed on screen is the output r
	cout << "The number of iteration steps taken were: " << counter <<
endl; //Printed on screen is the number of iterations
	double eq1 = r1 + log(r1 - 1); //This is the transformation
equation
	if (x == round(eq1)) { //I rounded the value of eq1 because it
tails off to a very long decimal and when I create the if statement the
round(eq1) could never be equal to the integer x, thus rounding eq1
would round to the nearest integer.
            cout << "Equation 1 (unrounded) is: " << eq1 << endl;
            cout << "Does it satisfy equation 1? Yes" << endl;
      }
      else {
            cout << "Equation 1 is: " << eq1 << endl;
            cout << "Does it satisfy equation 1? No" << endl;
      }
      cout << endl;
      return r1; //Here the function returns r1 which is the converged
value that we require
}
int main() {
      rofx(5.0);
      rofx(6.0);
}
```

```
[OUTPUT]
The root (in terms of r) is: 3.92627
The number of iteration steps taken were: 24
Equation 1 (unrounded) is: 5
Does it satisfy equation 1? Yes

The root (in terms of r) is: 4.69344
The number of iteration steps taken were: 20
Equation 1 (unrounded) is: 6
Does it satisfy equation 1? Yes
```

The number I used in this example for x was 5. The root of the equation was outputted onto the screen and it took 24 iteration steps. The root also satisfies the equation as it matched the original value of x, is 5. I tried another value greater than 2, which was 6. The root of the equation was 4.69 and it took 20 iteration steps to converge. It also satisfies the equation by substituting 4.69 into equation 1.

Question 2. Inner products.

(a) Write a function that takes as input two vectors and returns their inner product as a double number. Demonstrate that your program works by computing the inner product of two real Euclidean 3-vectors, u={2,7,2} and v={3,1,4}. [5]

(b) Write code for a function object that has a member variable m of type int, a suitable constructor, and a member function of the form double operator()(const vector u, const vector v) const {. Does the quantity I2(~u,~v)2 equal the inner product ~u · ~v that you obtained above? [5]

Here I created a function outside the main called “inner_product” that takes the two vectors u and v. Then it enters a for loop which takes the first element in each vector and multiplies them together and saves that value in a running total called S. It does this for a loop until it has reached the end of the vector size u (which is also the same size as v). **[CODE]**

```
double inner_product(const vector<double>& u, const vector<double>& v) {
      double S = 0.0; //S is the value storing the product of two
elements
      for (int i = 0; i < u.size(); i++)
            S += u[i] * v[i]; //Multiplying each element of u and v
together
      return S; //Returns the cumulative total of the dot products for
each element in vector u and v.
}
```

For part b, We needed to create a function object with a constructor. I used the function object “struct” as it seemed the best fitting with the function “operator”. “Operator” takes two vectors and a constant value “m” that follows the formula displayed on the coursework. The struct allows me to call the function norm for the following set of questions.

```
struct normal {
      double operator()(const vector<double>& u, const vector<double>&
v, const int m) {
            double S = 0.0;
            double count = 0.0; //Count is the running total of the
inner product between the two elements but to the power of m^2
            for (int i = 0; i < u.size(); i++) {
                  S = abs(u[i] * v[i]);
                  count += pow(S, (double(m) / double(2)));
            }
            return pow(count, (double(1) / double(m))); //The one over m
creates the root function
      }
};
```

Here I am defining the operator from above called “norm”. It follows the exact same code as the operator.

```
double norm(const vector<double>& u, const vector<double>& v, const int
m) {
      double S = 0.0;
      double count = 0.0;
      for (int i = 0; i < u.size(); i++) {
            S = abs(u[i] * v[i]);
            count += pow(S, (double(m) / double(2)));
      }
      return pow(count, (double(1) / double(m)));
}

int main() {
      vector<double> u{ 2.0,7.0,2.0 };
      vector<double> v{ 3.0,1.0,4.0 };
      cout << "The inner product between vector u and v are: " <<
inner_product(u, v) << endl;
      cout << "The weighted norm of L1 is: " << norm(u, v, 1) << endl;
      cout << "The weighted norm of L2 is: " << norm(u, v, 2) << endl;
      cout << "The weighted norm of L2 is: " << norm(u, v, 2)* norm(u,
v, 2) << endl;
}
[OUTPUT]
The inner product between vector u and v are: 21
The weighted norm of L1 is: 7.92367
The weighted norm of L2 is: 4.58258
The weighted norm of L2 squared is: 21
```

The inner product between the two vectors was 21 which is equal to the L2 norm squared.

Question 3. Finite differences.

(a) Write a C++ program that uses finite difference methods to numerically evaluate the first derivative of a function f(x) whose values on a fixed grid of points are specified f(xi), i = 0, 1, ..., N. Output the values ei of this vector on the screen and tabulate (or plot) them in your report. [8]

(b) For the same choice of f(x), demonstrate 2nd-order convergence, by showing that, as N increases, the mean error hei or the root mean square error he2i1/2 decrease proportionally to Δx2 ∝ N −2 . Is the quantity N2hei (or N2he2i1/2) approximately constant?[7]

[CODE]

```
void question3(const double N) { //Takes any value of N
    //Defining all variables
    const double n = N + 1;
    double i = 0.0;
    //Defining the three vectors
    vector<double> x(n);
    vector<double> f(n);
    vector<double> df(n);
    const double changex = 2.0 / N;
    vector<double> calculate(n);
    while (i < n) {
        x[i] = double((2.0 * i - N)) / double(N); //Filling in the
vector x[i] with all the grid points
        f[i] = exp(-x[i] * x[i]);//Filing in vector of function x[i]
        calculate[i] = -2.0 * x[i] * exp(-x[i]*x[i]); //The actual
derivative of e^(-x*x)
        i++;
    }
    //Creating a loop to assess which formula it should use to
calculate the approximation of f'[i]
    int counter = 0;
    for (int j = 0; j < n; j++) {
        if (j == 0) {
            //Use the left function
            df[0] = double((-3 * f[0] + 4 * f[1] - f[2])) /
double(2 * changex);
            counter++;
        }
        else if (j == N) {
            //Use the right function
            df[N] = double(f[N - 2] - 4 * f[N - 1] + 3 * f[N]) /
double(2 * changex);
			counter++;
		}
		else {
			df[j] = double(f[j + 1] - f[j - 1]) / double(2 * 
changex);
			counter++;
		}
		//cout << counter << endl; This counter was used to 
double-check that all the elements in the vector were filled correctly.
	}
	//Initialising vector for the difference e[i]
	int s = 0;
	double sum = 0;
	vector<double> Ei(n);
	//Tabular Form
	cout << "i\t" << "x[i]\t" << "f[i]\t\t" << "fa'[i]\t\t" << 
"fn'[i]\t\t" << "Difference\n";
	while (s < n) {
		cout << s << " \t";
		cout << df[s] - calculate[s] << " \n";
		sum += (df[s] - calculate[s]);	//Approximate f'[i] - 
Exact f'[i] and sums them up
		Ei[s] = df[s] - calculate[s]; //Not in absolute form
		s++;
	}
	cout << N << " \t";
	cout << norm(Ei, Ei, 2) << "\t"; //This is the L2 error norm
	cout << N*N* norm(Ei, Ei, 2) << "\n";} //This is the N2L2 error 
norm.
int main() {
	question3(15.0);
	question3(31.0);
	question3(63.0);
	question3(127.0);
	question3(1023.0);
}
```

[OUTPUT]

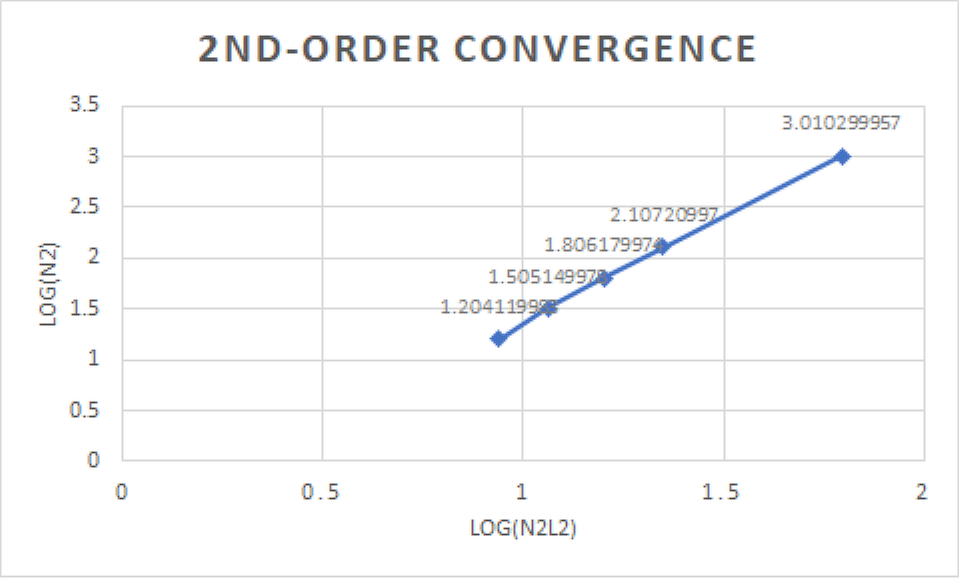
In the cpp file, I have unhid all the lines of codes so it would show all the different vector elements for 15,31,63,127 and 1023. However, I have just taken the essential items that are required for the question. I have outputted on screen in a tabular form of each element in the error vector and the value it corresponds to. The errors are very small and don't fluctuate at N =15.

```
i       Difference
0       0.0130597
1       -0.00723978
2       -0.00972013
3       -0.0112386
4       -0.0113227
5       -0.00973973
6       -0.0065929
7       -0.00233214
8       0.00233214
9       0.0065929
10      0.00973973
11      0.0113227
12      0.0112386
13      0.00972013
14      0.00723978
15      -0.0130597
```

For part b, we were asked to show the error norm approaches to zero. Firstly I calculated each N^2L_2 for N = 15,31,63,127,1023.

```
N       L2              N^2L2
15      0.0378346       8.51278
N       L2              N^2L2
31      0.0117958       11.3358
N       L2              N^2L2
63      0.00397026      15.758
N       L2              N^2L2
127     0.00137394      22.1602
N       L2              N^2L2
1023    5.96855e-05     62.4626
```

Then, I drew a log-log plot of the log(N^2) against log($N^2 L_2$) . The slope is equal to 2. This shows that the quantity log($N^2 L_2$) is approximately independent of N as it increases constantly as the value of N increases.



Question 4. Stellar equilibrium.

(a) Solve the above 1st-order system numerically, with a 4th-order Runge-Kutta method, using N + 1 = 101 equidistant points in x ∈ [0, π]. Output the values x0, x10, x20, ..., x100 and h(x0), h(x10), h(x20), ..., h(x100) to the screen and tabulate them in your report. [10]

(b) Compute the difference and output the error values e(x0), e(x10), e(x20), ..., e(x100) to the screen and tabulate them in your report. [5]

(c) Compute the error norms: l1(e,e), l2(e,e) [5]

Using the “RK4 code” in qmplus[1], I used the double output to return h and z. **[CODE]**

```
auto step_rk4(const double t, const double y, const double z, const
double h,
      double f(const double, const double, const double),
      double g(const double, const double, const double)) {
      //This struct is used to return two function values instead of one
      struct retVals {
            double d1, d2;
      };
      const double k1 = h * f(y, z, t);
      const double l1 = h * g(y, z, t);
      const double k2 = h * f(y + k1 / 2.0, z + l1 / 2.0, t + h / 2.0);
      const double l2 = h * g(y + k1 / 2.0, z + l1 / 2.0, t + h / 2.0);
      const double k3 = h * f(y + k2 / 2.0, z + l2 / 2.0, t + h / 2.0);
      const double l3 = h * g(y + k2 / 2.0, z + l2 / 2.0, t + h / 2.0);
      const double k4 = h * f(y + k3, z + l3, t + h);
      const double l4 = h * g(y + k3, z + l3, t + h);
      return retVals{ y + (k1 + 2.0 * k2 + 2.0 * k3 + k4) / 6.0, z + (l1
+ 2.0 * l2 + 2.0 * l3 + l4) / 6.0 };
}
```

Here I am creating two functions that “fz” and “fh” that correspond to the two 1st-order difference equations.

```
double fz(const double z, const double h, const double x) {
      if (x == 0) //This if statement allows me to establish the two
conditions set in the question
            return -1.0 / 3.0;
      else
      return -(2/x)*z - h;
}
```

```
double fh(const double z, const double h, const double x) {
	return z;
}


int main() {
	const double t0 = 0.0;	//initial time
	const double Z0 = 0.0;	//initial conditions
	const double H0 = 1.0;
	const double x0 = 0.0;
	const int N = 100;	//number of points is N+1; number of intervals is N
	const double h = pi / double(N);	//time step
	vector<double> Z(N + 2, Z0); //vector S; size is N+1; initial element value is S0
	vector<double> H(N + 2, H0); //vector I; size is N+1; initial element value is I0
	vector<double> x(N + 2, x0); //vector x; size is N+1; initial element value is x0
	vector<double> e(N + 1);//vector e; size is N+1;
	double t = t0;

	for (int i = 0; i <= N; i++) {
		auto [Znext, Hnext] = step_rk4(t, Z[i], H[i], h, fz, fh);
		Z[i + 1] = Znext; //This updates the next z value to be the value RK4 calculated
		H[i + 1] = Hnext;
		x[i + 1] = x[i] + h; //Adding one time step to the x vector

		if (i == 0) { //At i=0, the error terms would give "nan" as 1/0 cannot be calculated
			double hexact = 1.0; //So at 0, the value just corresponds to 1
			e[i] = H[i] - hexact;
		}
		else {
			double hexact = (1.0 / x[i]) * sin(x[i]);
			e[i] = H[i] - hexact;
		}

		t += h;
	}
	vector<int> nn{ 0,10,20,30,40,50,60,70,80,90,100 }; //This is a vector that can output the values between 0,10,...,100
	int s = 0;
	cout << "N \t\t\tX[i] \t\t\th[i] \t\t\tz[i] \tdiff" << endl;
```

```
      while (s < nn.size()) {
            double j = nn[s];
            cout << j << "\t\t\t" << x[j] << "\t\t\t" << H[j] <<
"\t\t\t" << Z[j] << "\t" << e[j] << endl;
            s++;
      }
      cout << "The weighted l1 norm is: " << setprecision(16) << norm(e,
e, 1) << endl;
      cout << "The weighted l2 norm is: " << setprecision(16) << norm(e,
e, 2) << endl;
}
[OUTPUT]
N         X[i]                h[i]             z[i]             diff
0            0                  1                0                0
10       0.314159           0.983632       -0.10369         -1.41424e-09
20       0.628319           0.935489       -0.201287        -1.97501e-09
30       0.942478           0.858394       -0.287124        -2.11239e-09
40       1.25664            0.756827       -0.356356        -2.1285e-09
50       1.5708             0.63662        -0.405285        -2.11063e-09
60       1.88496            0.504551       -0.431611        -2.10787e-09
70       2.19911            0.367883       -0.43457         -2.15355e-09
80       2.51327            0.233872       -0.414952        -2.26776e-09
90       2.82743            0.109292       -0.375022        -2.45619e-09
100      3.14159          -2.70896e-09     -0.31831         -2.70897e-09

The weighted l1 norm is: 2.097642159230597e-07
The weighted l2 norm is: 2.166072863254038e-08
```

I tabulated the outputs from 0,10,...,100 for each x,h,z and e value. The error norm of L1 is 2.1e-07 and error norm of L2 is 2.2e-08.

Question 5. Numerical integration.

(a) Use the composite trapezium rule to compute the integral I, using N + 1 = 64 equidistant points in x ∈ [−1, 1]. Output to the screen (and list in your report) your numerical result.
(b) Use the composite Hermite integration rule to compute the integral I, using N + 1 = 64 equidistant points in x ∈ [−1, 1]. Output to the screen (and list in your report) your numerical result.
(c) Use the Clenshaw-Curtis quadrature rule to compute the integral I. Output to the screen (and list in your report) your numerical result.
(d) Compute the integral I using a hit and miss Monte Carlo method with N = 10000 samples. Output to the screen (and list in your report) your numerical result.

[CODE]

```
struct f_I { //Creating the function I that will allow a value of x as
input and it returns the corresponding y value for the function. Using a
struct as it can be used to call the operator anywhere within the work.
      double operator()(const double x) {
            return 1.0 / (1.0 + 25.0 * x * x);
      }
};
double f_I(const double x) {
      return 1.0 / (1.0 + 25.0 * x * x);
}
double summation(const double theta) { //This is the summation for the
Clenshaw-Curtis quadrature rule that takes a theta value and creates a
sum.
      double sum = 0.0;
      for (double je = 1.0; je <= 31.0; je++) {
            sum += double(2.0 * cos(2.0 * je * theta)) / double(4.0 * je
* je - 1.0);
      }
      return sum;
}
int main() {
      //Define variables
      const double N = 63.0;
      const double n = 64.0;
      vector<double> x(n);
      const double a = -1.0;
      const double b = 1.0;
      //Fill vector x with all equidistant points
      double s = 1.0;
      x[0] = -1.0; //Setting initial element of x to be -1
```

```
    while (s <= N) {
        x[s] = x[s-1] + double(b - a) / double(N);
        s++;
    }
    //Now establishing vector w[i]
    vector<double> w(n);
    double i = 0.0;
    int counter = 0;
    double changex = (double(b - a) / double(N));
    while (i <= N) {
        if (i==0.0) {
            w[i] = changex / 2;
            counter++;
        }
        else if (i == N) {
            w[i] = changex / 2;
            counter++;
        }
        else{
            w[i] = changex;
            counter++;
        }
        i++;
    }
    //Establishing the vector f[x[i]]
    vector<double> f(n);
    int fcount = 0;
    while (fcount<=N) {
        f[fcount] = f_I(x[fcount]);
        fcount++;
    }
    //Introducing the dot product of wi and fi
    double integral = inner_product(w,f);
    //Hermite integration using the above
    double dfdxa = 25.0 / 338.0;
    double dfdxb = -25.0 / 338.0;
    double hermite = integral + ((changex * changex) / 12) * (dfdxa -
dfdxb);

    //Clenshaw-Curtis Quadrature Rule
    int icounter = 0;
    //Fill a vector of theta i and x i
    vector<double> theta(n);
    vector<double> xc(n);
    while (icounter <=N) {
            theta[icounter] = double(icounter * pi) / double(N);
            xc[icounter] = -cos(theta[icounter]);
            icounter++;
      }
      //Fill a vector of f with the new xi from Clenshaw-Curtis
      vector<double> fc(n);
      int fccount = 0;
      while (fccount <= N) {
            fc[fccount] = f_I(xc[fccount]); //Plugging values of x =
-cos(theta) into the I function and storing it in fc vectors.
            fccount++;
      }
      //Fill a vector of w i with new values
      vector<double> wc(n);
      double p = 0.0;
      double jeep = 0.0;
      while (p <= N) {
            if (p == 0.0) {
                  wc[p] = 1.0/(N*N);
                  cout << wc[p] << endl;
            }
            else if (p == N) {
                  wc[p] = 1.0/(N*N);
            } //The previous two if statements state if i =0 or N, then
the 1/n*n formula should be used
            else {
                  wc[p] = (2.0 / N) * (1.0 - summation(theta[p]));
            } //If i is not 0 or N, then it should use this formula. For
each value of p (the counter from 0 to N), it calculates the summation
from 1 to (N-1)/2 for that specific theta(p) value.
            p++;
      }
      //Now the integration part for Clenshaw-Curtis
      double Clenshaw = inner_product(wc, fc);
      //Hit and Miss Monte Carlo Method
      //Defining variables
      const double N1 = 10000.0;
      double count = 0.0;
      double in_area = 0.0;
      //Generating a random uniform sample
      const int seed = 31;
      mt19937_64 mtrand(seed); //mtrand is the name of the variable and
mt19937 is the variable type that allows me to create a random integer.
      uniform_real_distribution<double> unif(0.0, 1.0); //This creates a
uniform distribution value.
```

```
    while (count<=N1) {
        //Generate random point with two variable x and y
        const double x_cord = - 1.0 + 2.0 * unif(mtrand); //A new
uniform value is created at each loop to create a point within the box.
        const double y_cord = 1.0 * unif(mtrand);
        if (y_cord<f_I(x_cord)){
            In_area++; //If the randomly generated y value is less
than the curve’s y value (f_I(x_cord)) then it adds one to the counter
that is within the box area.
        } count++; }
    //Area of the box
    double area_box = 2.0;
    //Area of integral
    double Hit = (in_area / count) * area_box;
    double Iexact = (2.0 / 5.0) * (atan(5));
    cout << "The exact value of integration is: " << Iexact << endl;
    cout << "The composite trapezium rule gave the integration of: "
<< integral << endl;
    cout << "The difference between the two is: " << setprecision(16)
<< integral - Iexact << endl;
    cout << "The hermite trapezium rule gave the integration of: " <<
hermite << endl;
    cout << "The difference between the two is: " << setprecision(16)
<< hermite - Iexact << endl;
    cout << "The Clenshaw-Curtis trapezium rule gave the integration
of: " << Clenshaw << endl;
    cout << "The difference between the two is: " << setprecision(16)
<< Clenshaw - Iexact << endl;
    cout << "The Hit and Miss Monte Carlo gave the integration of: "
<< Hit << endl;
    cout << "The difference between the two is: " << setprecision(16)
<< Hit - Iexact << endl;
}
```

[OUTPUT]

```
The exact value of integration is: 0.54936
The composite trapezium rule gave the integration of: 0.549348
The difference between the two is: -1.242147856816977e-05
The hermite trapezium rule gave the integration of: 0.5493603089992857
The difference between the two is: 2.221279271630294e-09
The Clenshaw-Curtis trapezium rule gave the integration of:
0.5493603067567434
The difference between the two is: -2.1262991367621e-11
The Hit and Miss Monte Carlo gave the integration of: 0.5515448455154485
The difference between the two is: 0.0021845387374420
```

Question 6. Harmonic Oscillator.

(a) Use a 2nd-order Runge-Kutta (RK2) midpoint method to evolve the system, with initial conditions q(0) = 0, p(0) = √2 and time-step Δt = 0.1. Stop the evolution at time t = 100. Describe how your code works. [8]
(b) Output to the screen (and tabulate in your report) the values of the position q(t), momentum p(t), energy E(t) = 1 2 (p2 + q2) and the difference e(t) = E(t) − E(0) for t = 0, t = 1, t = 10 and t = 100. Is E(t) constant numerically? [7]

[CODE]

```
//Define dp/dt = f(pi,ti)
double dpdt(const double q,const double t) {
        return (-q); //Takes a value q and t and returns -q
    }
//Define dq/dt = f(qi,ti)
double dqdt(const double p,const double t) {
    return (p); //Takes a value p and t and returns p
}
int main() {
    //Define variables
    const double T = 1001.0; //This is the total time step between 0 and 100
    vector<double> ti(T);
    ti[0] = 0.0; //Setting initial element of time to be 0
    const double h = 0.1; //This is the change in t, basically our step size
    for (int i = 1; i < ti.size(); i++) {
        ti[i] = ti[i-1] + 0.1; //For every loop, take the previous time step and add the change in t = 0.1
    }
    //Define initial conditions for the two ODES
    vector<double> Pi(T);
    vector<double> Qi(T);
    vector<double> fPi(T);
    vector<double> fQi(T);
    vector<double> Pi1(T);
    vector<double> Qi1(T);
    vector<double> fPi1(T);
    vector<double> fQi1(T);
    Pi[0] = pow(2, 0.5); //First element of Pi is sqrt 2
    Qi[0] = 0; //First element of Qi is 0
    //Doing Eulers step once
    fPi[0] = dpdt(Qi[0], ti[0]); //The first derivative of P. It takes the first element of Qi and Time step, then plugs it into the dpdt function and outputs -q
```

```
      fQi[0] = dqdt(Pi[0], ti[0]); //Same thing, this is the first
derivative of Q. Takes values p and t, and returns the p value.
      Pi1[0] = Pi[0] + fPi[0] * h;
      Qi1[0] = Qi[0] + fQi[0] * h; //This is the second point that is
created by adding the derivative multiplied by the time step and added
on from the initial value of P and Q.
      fPi1[0] = dpdt(Qi1[0], ti[0]);
      fQi1[0] = dqdt(Pi1[0], ti[0]); //Once again, we calculate the
first derivative from the new points Pi1 and Qi1. This sets us up to do
RK2 as RK2 requires the new points to work out the initial p and q
values in the next time step (i.e 0.1)
      //Doing RK2
      int rk2counter = 1;
      while (rk2counter < T) { //Creating a while loop that starts from
1 and ends at 1000
            Pi[rk2counter] = Pi[rk2counter - 1] + ((h * (fPi[rk2counter
- 1] + fPi1[rk2counter - 1])) / 2);
            Qi[rk2counter] = Qi[rk2counter - 1] + ((h * (fQi[rk2counter
- 1] + fQi1[rk2counter - 1])) / 2); //This is the main calculation to
work out the new initial p and q at the new time step. In the case of
the first loop (at loop 1), it takes the initial p and creates an
average between the first derivative fPi and the new first derivative
fPi1. Then, it is added onto the initial p and q values from the
previous time step.
            fPi[rk2counter] = dpdt(Qi[rk2counter], ti[rk2counter]);
            fQi[rk2counter] = dqdt(Pi[rk2counter], ti[rk2counter]);
            Pi1[rk2counter] = Pi[rk2counter] + (fPi[rk2counter] * h);
            Qi1[rk2counter] = Qi[rk2counter] + (fQi[rk2counter] * h);
            fPi1[rk2counter] = dpdt(Qi1[rk2counter], ti[rk2counter]);
            fQi1[rk2counter] = dqdt(Pi1[rk2counter], ti[rk2counter]);
//And the process is repeated by calculating two derivative points. One
of the initial time step values (at loop 1) of p and q and then the
output is stored in the vectors and used in the second loop.
            rk2counter++;
      }
      vector<double> nn{ 0,1,10,100 }; //Creating a vector with the
desired time steps
      for (int j = 0; j < nn.size(); j++) {
            const int T = nn[j]; //Selecting each value from that vector
to be the current time step
            float E_t = 0.5 * (Pi[T] * Pi[T] + Qi[T] * Qi[T]);
            const float  E_0 = 0.5 * (Pi[0] * Pi[0] + Qi[0] * Qi[0]);
//Here I am just using the formulas given in the coursework sheet.
            cout << "For t = " << T << endl;
            cout << "The output for position q(T) is: " << Qi[T] <<
endl;
            cout << "The output for momentum p(T) is: " << Pi[T] <<
endl;
            cout << "The output for energy E(T) is: " << E_t << endl;
            cout << "The output for difference E(T) - E(0) is: " << E_t
- E_0 << endl;
            cout << "\n\n";
      }
}

[OUTPUT]
For t = 0
The output for position q(T) is: 0
The output for momentum p(T) is: 1.41421
The output for energy E(T) is: 1
The output for difference E(T) - E(0) is: 0

For t = 1
The output for position q(T) is: 0.141421
The output for momentum p(T) is: 1.40714
The output for energy E(T) is: 1.00003
The output for difference E(T) - E(0) is: 2.5034e-05

For t = 10
The output for position q(T) is: 1.19144
The output for momentum p(T) is: 0.76222
The output for energy E(T) is: 1.00025
The output for difference E(T) - E(0) is: 0.000249982

For t = 100
The output for position q(T) is: -0.789959
The output for momentum p(T) is: -1.17515
The output for energy E(T) is: 1.0025
The output for difference E(T) - E(0) is: 0.00250304
```

We can see the difference for E(T) - E(0) increases as time increases. Moreover, the difference between each time step apart from 0 to 1, is a multiplication by 10. E(T) looks to be constant at 1 regardless of the time step.

References

1: Qmplus.qmul.ac.uk. 2020. *MTH6150: Numerical Computing In C And C++*. [online] Available at: <https://qmplus.qmul.ac.uk/mod/resource/view.php?id=1112975> [Accessed 16 May 2020].