Question 1

```
#include <iostream>
#include <cmath>
#include <iomanip>
using namespace std;

double rofx(const double x) {
    //I have specified the return type of my function rofx to be double before
defining it

    double eps = pow(double(10), double(-12));
    //Initialising a tolerance parameter of 10^-12 using the pow function
located in the <cmath> library
    //The value of epsilon is an extremely small number, therefore, I will
take this number to be equivalent to zero and use it accordingly in my for
loop

    double r_n = double(0.8) * x;
    //Initialised the variable r_n to be my initial guess of [r(n)]

    double r_n1 = x - double(log(abs(r_n - 1.0)));
    //Initialised the variable r_n1 to be equal to x – ln|r(n) – 1|

    double iterations = 0.0;
    while (double(abs(r_n1 - r_n)) >= eps) {
        //The while loop will keep running until the condition (abs(r_n1 -
r_n) >= eps) is met
        //When the distance between rn+1 and rn is less than 10^-12, we can
assume that the value of r has converged as the distance between r(n) and
r(n+1) is less than the tolerance level

        r_n = r_n1;
        //After one iteration the value of r(n) will become r(n+1) and after
the second iteration will be r(n+2) and so on

        r_n1 = x - double(log(abs((r_n - 1))));
        //After one iteration the value of r(n+1) will become r(n+2) and after
the second iteration will be r(n+3) and so on

        iterations++;
        //Initialised a variable ‘iterations’ before my while loop which
increments its value by 1 each time the while loop runs to count how many
times the while loop has run
    }

    cout << "The function rofx converged in " << iterations << " iterations. "
<< endl;

    const double error = (r_n1 + double(log(abs(r_n1 - 1)))) - x;
```

```
    //Initialised a variable called error to compute the difference between x
and r(n+1) + ln|rn – 1|

    cout << "The equation x = r + ln|r - 1| is satisfied when substituting
rn+1 into the equation as we get " << error << endl;

    return r_n1;
}


int main() {
    double x;
    cout << "Input a coordinate value for x which is strictly greater than 2 "
<< endl;
    cin >> x;
    cout << "The final value of r is: " << setprecision(16) << rofx(x) <<
endl;
}
```

Question 1

The function rofx takes x as an argument and solves x = r + ln |r – 1| via self-consistent iteration. The self-consistent iteration method returns the value of r up to the specified tolerance ε by using a while loop that continues to run until |r(n+1) – r(n)| < ε. As the value of x must remain fixed, I defined x to be constant. The value of r(n) is updated within the while loop after each iteration with the value of r(n+1), and the value of r(n+1) is updated to r(n+2) by the equation x – ln |rn – 1|. As well as this, when substituting the converged value of r into the equation x = r + ln |r – 1| the value of x is returned; this demonstrates the functionality of the function.

If I run my code my compiler will first ask to input a coordinate value for x which is strictly greater than 2.

Let’s say I input the value:

2.4

My code will run and my console will output these messages.

“The function rofx converged in 143 iterations.

The equation x = r + ln|r - 1| is satisfied when substituting rn+1 into the equation as we get -7.19425e-13

The final value of r is: 2.209660367668291”

Question 2

```
#include <iostream>
#include <cmath>
#include <vector>
#include <iomanip>
#define eps 0.000001
//Defined eps as such so that the compiler will replace references to these
constants with the defined value at compile time
//https://www.arduino.cc/reference/en/language/structure/further-
syntax/define/
using namespace std;

double inner_product(const vector<double>& u, const vector<double>& v) {
      //The function inner_product takes u and v as arguments and u and v are
passed as references so the compiler doesn’t need to copy the vectors.

      double sum1;
      sum1 = 0.0;

      if (u.size() == v.size())
            //I have made an if statement to ensure that the sizes of the
vectors are the same, as the dot product of two vectors can't be computed if
the sizes of the vectors are different

            for (int i = 0; i < u.size(); i++) {
                  //As the size of vectors u and v in our example are 3 the
conditions for sum are held as we are counting from 0 to 3 not inclusive
                  //cout << u[i] << " * " << v[i] << " = " << u[i] * v[i] <<
endl;
                  sum1 += double(u[i]) * double(v[i]);
                  //At each iteration of the for loop the value of the sum is
increased by the value of u[i] * v[i]. Therefore, on the final for loop the
value of the sum will be equal to the inner product
                  //cout << sum1 << endl;
            }
      else {
            cout << "size of vectors aren't equal, therefore, cannot compute."
<< endl;
      }
      return sum1;
}

class Weighted_norm {
private:
      int m;
      //I have declared my member variable m to be private so that it can only
be accessed via member functions and not individually
public:
      Weighted_norm() : m(1) {}
      //As the m’th root of the weighted norm expression can't be zero I have
set the value of m to be 1 when a value for m has not been specified
```

```
	Weighted_norm(int a) : m(a) {}
	//Set this constructor as such so that the weighted norm accepts ‘a’ as
a parameter for the value of m

	double operator()(const vector<double> u, const
		vector<double> v) const {
		//This member function takes two vectors as arguments and as the
function won't change the member data, the member function can be declared as
a const function
		double l = 0.0;
		double sum2 = 0.0;

		if (u.size() == v.size()) {
			//I have made an if statement to ensure that the sizes of the
vectors are the same, as the dot product can't be computed if the sizes of the
vectors are different
			for (int i = 0; i < u.size(); i++) {
				sum2 += pow(abs(double(u[i]) * double(v[i])), double(m)
/ 2.0);
				//At each iteration of the for loop the value of the
sum is increased by the value of |u[i] * v[i]| ^ m/2
				//As I am doing a division of m/2, both these numbers
are integers, therefore, they must be converted into doubles
				}
			l = pow(sum2, 1.0 / double(m));
			//I have declared the variable ‘l’ to calculate the m’th root
of the sum computed in the previous for loop. As I am doing a division of 1/m,
both these numbers are integers, therefore, they must be converted into
doubles
		}
		else {
			cout << "size of vectors aren't equal, therefore, cannot
compute." << endl;
		}
		return l;
	}
};


int main() {
	vector<double> u = { 2, 7, 2 };
	vector<double> v = { 3, 1, 4 };
	cout << "The inner product of vectors u and v is " << inner_product(u,
v) << endl;

	const int m1 = 1;
	//Value we are assigning our member variable m to be
	//const Weighted_norm m1_weighted_norm;
	//If we didn’t declare m, then by default the value of m would be 1 as I
set the constructor as such

	const Weighted_norm m1_weighted_norm(m1);
	//Before we use the object, Weighted_norm, we first have to declare an
```

```
instance of the object, which in our case is m1_weighted_norm
	//Accessing our weighted norm object and using our parameter m1 as the
input for m
	double l1 = m1_weighted_norm(u, v);
	//Assigning vectors u and v to be the arguments in the member function
	cout << "The weighted norm for l1(u,v) is " << l1 << endl;

	const int m2 = 2;
	const Weighted_norm m2_weighted_norm(m2);
	double l2 = m2_weighted_norm(u, v);
	cout << "The weighted norm for l2(u,v) is " << l2 << endl;

	const double error = (l2 * l2) - inner_product(u, v);
	//l2 is a double variable and therefore doesn't need to be converted

	cout << "The quantity l2(u,v)^2 equals the inner product of u*v as the
error value is " << error << endl;

}
```

Question 2(a)

The function inner_product takes as input two vectors $\vec{u} = \{u_0, u_1, \ldots, u_N\} \epsilon \mathbb{R}^{N+1}$ and $\vec{v} = \{v_0, v_1, \ldots, v_N\} \epsilon \mathbb{R}^{N+1}$. It returns their inner product by computing the sum $\sum_{i=0}^{N} u_i v_i$. I accomplished this by creating a for loop to compute the sum for each u_i and v_i. The codes functionality was demonstrated by computing the inner products of two real Euclidean 3-vectors, $\vec{u} = \{2, 7, 2\}$ and $\vec{v} = \{3, 1, 4\}$.

My compiler computes the inner product of these two vectors and outputs:

"The inner product of vectors u and v is 21"

Question 2(b)

The object weighted norm for $l_m(\vec{u}, \vec{v})$ is computed by first calculating the sum $\sum_{i=0}^{N} |u_i v_i|^{\frac{m}{2}}$ and then using this value to compute the m'th root $\sqrt[m]{\sum_{i=0}^{N} |u_i v_i|^{\frac{m}{2}}}$. I completed this using object-oriented programming and declared my member variable to be m. Additionally, two suitable constructors were assigned with the same name as the variable type, to define default values for m. A member function called operator(), which computes the weighted norm of $l_m(\vec{u}, \vec{v})$, was declared. Finally, I demonstrated my codes functionality by computing the weighted norms of two real Euclidean 3-vectors, $\vec{u} = \{2, 7, 2\}$ and $\vec{v} = \{3, 1, 4\}$. My compiler computes the weighted norm of these two vectors and outputs:

"The weighted norm for l1(u,v) is 7.92367

The weighted norm for l2(u,v) is 4.58258

The quantity l2(u,v)^2 equals the inner product of u*v as the error value is 0”

Question 3(a)

```
#include <iostream>
#include <cmath>
#include <vector>
#include <iomanip>
using namespace std;

double function_exp(const double x) {
	//I have declared a function for f(x) which returns e^-(x^2) to test my programs functionality
	return exp(-(x * x));
}

double f_prime_exp(const double x) {
	// I have declared a function for f'(x) which returns -2x * e^-(x^2) to test my programs functionality
	return -2.0 * double(x) * double(exp(-(x * x)));
}

vector<double> error(const int N, double func(const double), double diff_func(const double)) {
	//I have created a vector function to return the error vector, this is so that I can use this function for Question 3b
	//I have also defined it in this manner so that 'vector<double> error' can be used for any function such as e^-x^2 (as given in the question)
	//Also, because of the layout of my function I can output any vector x, fx, f'x numeric, f'x analytic and error vector easily for any function and any number of points (N+1)
	//From all the methods I experimented with I found this method to be the neatest and most efficient

	double delta_x = 2.0 / double(N);
	//The value for delta x, given in the question

	vector<double> x(N + 1);
	for (int i = 0; i < x.size(); i++) {
		x[i] = (2.0 * double(i) - double(N)) / double(N);
		//cout << "Values of xi for each element in the vector \t" << i << "\t" << x[i] << endl;
	}
	//The values of xi are assumed to be located at the grid-points (2i-N) / N, for each i the value is stored in the i'th entry of the vector x
	//cout << endl;

	vector<double> fx(N + 1);
	for (int i = 0; i < fx.size(); i++) {
		fx[i] = func(x[i]);
		//cout << "Values of fx for each element in the vector \t" << i
```

```
<< "\t" << fx[i] << endl;
	}
	//This vector stores the values of f(xi) by inputting the values xi, computed in the previous vector, into the argument func
	//cout << endl;

	vector<double> f_prime_numeric(N + 1);
	for (int i = 0; i < f_prime_numeric.size(); i++) {
		if (i == 0) {
			f_prime_numeric[0] = ((-3.0 * fx[0]) + (4.0 * fx[1]) - (fx[2])) / (2.0 * delta_x);
		}

		else if (i > 0 && i <= (N - 1)) {
			f_prime_numeric[i] = (fx[i + 1] - fx[i - 1]) / (2.0 * delta_x);
		}

		else {
			f_prime_numeric[N] = ((fx[N - 2]) - (4.0 * fx[N - 1]) + (3.0 * fx[N])) / (2.0 * delta_x);
		}
		//cout << "Values of f'x numeric for each element in the vector \t" << i << "\t" << f_prime_numeric[i] << endl;
	}
	//This vector stores the numerical values of f'(x) using the finite differences method covered in the interpolating lecture
	//cout << endl;

	vector<double> f_prime_analytic(N + 1);
	for (int i = 0; i < f_prime_analytic.size(); i++) {
		f_prime_analytic[i] = diff_func(x[i]);
		//cout << "Values of f'x analytic for each element in the vector \t" << i << "\t" << f_prime_analytic[i] << endl;
	}
	//This vector stores the analytic values of f'(x) by inputting the values of xi, into the differential function argument
//cout << endl;

	vector<double> error_vector(N + 1);
	for (int i = 0; i < error_vector.size(); i++) {
		error_vector[i] = f_prime_numeric[i] - f_prime_analytic[i];
		cout << setprecision(12) << "Error values for each element in the vector \t " << i << "\t" << error_vector[i] << endl;
	}
	//This vector stores the error values, which is f'numerical(xi) - f'analytical(xi)
	return error_vector;
	//Returns the error vector computed in the previous step
	//This is the only method I found that will allow me to output vectors, as the void function is not recognised as a vector when trying to input the error vector into the weighted norm object defined for question 3(b)
}
```

```
int main() {

	//This is the vector I have defined to output the answers for Question 3(a) in the neatest manner possible
	//I have also done it as such so that if I wanted to output the vectors within the function for more points, then I can easily do so by changing the limit within the for loop
	//Simply run this vector for however many values of N and for whatever function you would like to test it with
	vector<double> Question_3a(1);
	for (int j = 16; j < 18; j *= 2) {
		error(j - 1, function_exp, f_prime_exp);
		cout << endl;
	}

}
```

Question 3(a)

I defined a vector function to numerically evaluate the first derivative of a function f(x) using the finite differences method. Defining a vector function allowed me to return a vector, hence allowing me to use this vector function for 3(b). Initially, I stored the values of xi using a vector<double> and the grid-points for xi were located at xi = (2i – N)/N on the interval [-1,1]. Following this, I stored the values of f(xi) and f'(xi) in vectors. When storing the values of f'(xi) I computed the values of f'(x) at each i using the finite differences method. I used an 'if' statement for when i = 0, an 'else if' statement for when i > 0 and i ≤ N – 1, and an 'else' statement for when i = N. When computing the derivatives for f(x), $O(\Delta x^2)$ did not need to be calculated as this is the margin of error; this will be calculated in the next question. When defining my function, I did not specify a specific function, therefore, this function can be used to work out the first derivative of any function, for however many points N + 1.

In order to check whether my program worked, I tested my program by evaluating the derivatives of $f(x) = e^{-x^2}$, with N + 1 = 16 points.

I also computed the difference between my numerical derivatives and the known analytical derivatives at each grid point. The error values for each i are tabulated below.

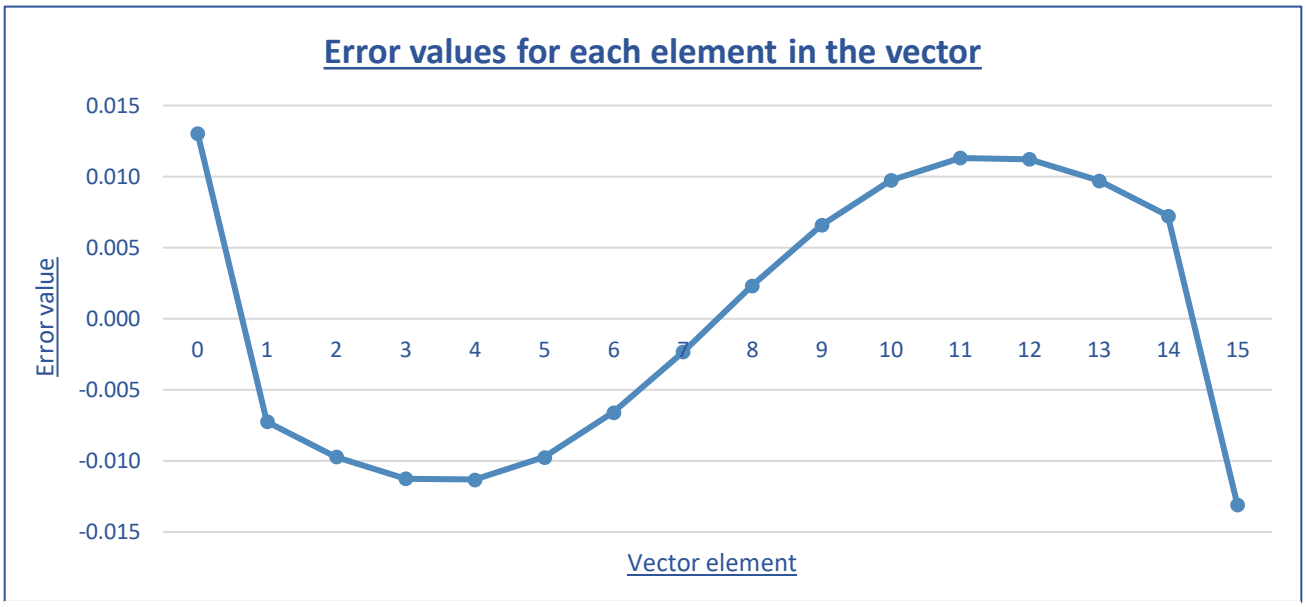

Figure 1 - Error values for each element in the vector

Vector elements	Error values
0	0.0130597102043
1	-0.00723977639065
2	-0.00972012545961
3	-0.0112385876945
4	-0.011322663707
5	-0.0097397329676
6	-0.00659289815322
7	-0.00233213692403
8	0.00233213692403
9	0.00659289815322
10	0.0097397329676
11	0.011322663707
12	0.0112385876945
13	0.00972012545961
14	0.00723977639065
15	-0.0130597102043

Table 1 – Error values for each element in the vector

Question 3(b)

```
#include <iostream>
#include <cmath>
#include <vector>
#include <iomanip>
using namespace std;
```

```
double function_exp(const double x) {
	return exp(-(x * x));
}

double f_prime_exp(const double x) {
	return -2.0 * x * exp(-(x * x));
}

vector<double> error(const int N, double func(const double), double
diff_func(const double)) {

	double delta_x = 2.0 / double(N);

	vector<double> x(N + 1);
	for (int i = 0; i < x.size(); i++) {
		x[i] = (2.0 * double(i) - double(N)) / double(N);
	}

	vector<double> fx(N + 1);
	for (double i = 0; i < fx.size(); i++) {
		fx[i] = func(x[i]);
	}

	vector<double> f_prime_numeric(N + 1);
	for (int i = 0; i < f_prime_numeric.size(); i++) {
		if (i == 0) {
			f_prime_numeric[0] = ((-3.0 * fx[0]) + (4.0 * fx[1]) -
(fx[2])) / (2.0 * delta_x);
		}

		else if (i > 0 && i <= (N - 1)) {
			f_prime_numeric[i] = (fx[i + 1] - fx[i - 1]) / (2.0 *
delta_x);
		}

		else {
			f_prime_numeric[N] = ((fx[N - 2]) - (4.0 * fx[N - 1]) + (3.0
* fx[N])) / (2.0 * delta_x);
		}
	}

	vector<double> f_prime_analytic(N + 1);
	for (int i = 0; i < f_prime_analytic.size(); i++) {
		f_prime_analytic[i] = diff_func(x[i]);
	}

	vector<double> error_vector(N + 1);
	for (int i = 0; i < error_vector.size(); i++) {
		error_vector[i] = f_prime_numeric[i] - f_prime_analytic[i];
	}
	//This vector stores the error values, which is f'numerical(xi) -
f'analytical(xi)
```

```
	return error_vector;
	//Returns the error vector computed in the previous step
}

//This is the class from Question 2(b) to work out the weighted norm of two
vectors
//I have simply renamed the object to Mean_error and declared the object
before int main() and defined it under int main() as this functions
functionality was already demonstrated in the previous question
class Mean_error {
private:
	int m;
public:
	Mean_error() : m(1) {}
	Mean_error(int a) : m(a) {}
	double operator()(const vector<double> u, const
		vector<double> v) const;
};

int main() {

	//This is the vector I have defined to output the answers for Question
3(b) in the neatest manner possible
	//I have also done it as such so that if I wanted to output the mean
error for more points (N) then I can easily do so by changing the limit within
the for loop

	vector<double> Question_3b(5);
	//This for loop will allow me to output the mean error <e> for N + 1 =
16, 32, 64, 128, 256.
	for (int j = 16; j < 257; j *= 2) {
		const Mean_error N_mean_error(1);
		//In order to work out the mean error, the value for m in weighted
norm m will be 1
		double error_N = N_mean_error(error(j - 1, function_exp,
f_prime_exp), error(j - 1, function_exp, f_prime_exp));
		//The mean error is to be computed for vectors ei
		error_N /= ((double(j) - 1.0) + 1.0);
		//Once the sum is computed this value has to be divided by N + 1
		cout << "The mean error for N + 1 = " << j << " is " <<
setprecision(12) << error_N << endl;
		cout << "N^2<e> is " << setprecision(12) << double(j - 1.0) *
double(j - 1.0) * error_N << endl;
		cout << endl;
	}

	cout << "As we can see from the values that have been outputted; as the
value of N increases the mean error decreases, therefore as <e> decreases
1/N^2 also decreases proportionally to <e> " << endl;
	cout << " As we can see from the values of N^2<e> that have been output
the value of N^2<e> is approximately constant " << endl;

}
```

```
double Mean_error::operator()(const vector<double> u, const
	vector<double> v) const {
	double l = 0.0;
	double sum2 = 0.0;

	if (u.size() == v.size()) {
		for (int i = 0; i < u.size(); i++) {
			sum2 += pow(abs(double(u[i]) * double(v[i])), double(m) /
2.0);
		}
		l = pow(sum2, 1.0 / double(m));
	}
	else {
		cout << "size of vectors aren't equal, therefore, cannot compute."
<< endl;
	}
	return l;
};
```

Question 3(b)

To demonstrate second order convergence for $f(x) = e^{-x^2}$ I defined a vector 'Question_3b' that contains a for loop which computes the value of the mean error <e> for different values of N. The for loop computes <e> by using the object Weighted_norm from question 2 and assigning the vector e_i, computed in Question 3a, as an argument in the member function. After computing the weighted norm, I divided the weighted norm by N + 1 to work out the mean error. Furthermore, to test whether the mean error decreased proportionally to $\Delta x^2 \propto N^{-2}$, I computed the value of $N^2\langle e \rangle$ for different values of N. I further checked whether this value was approximately constant.

I have tabulated my results below:

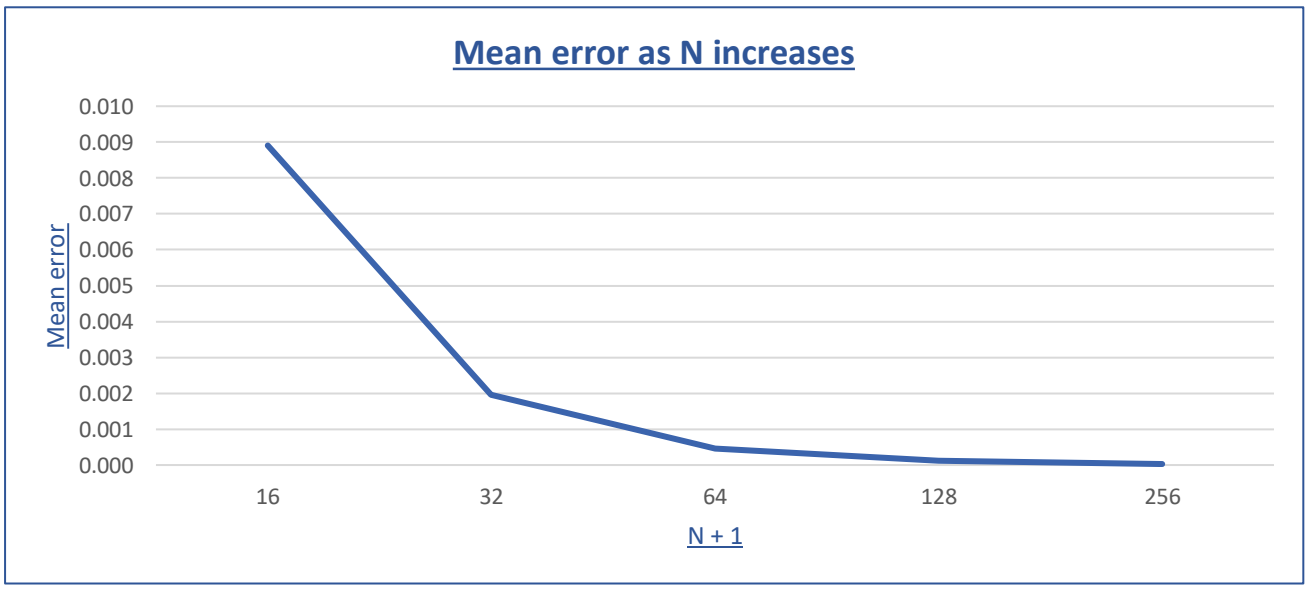

Figure 2 - Mean error as N increases

N + 1	Mean error
16	0.00890570393762
32	0.00196225396942
64	0.00046559367990
128	0.00011372189986
256	0.00002812177445

Table 3 - Mean error as N increases

It is evident from Figure 2 and Table 2 that as the size of N increases, the mean error decreases, therefore as <e> decreases, 1/N^2 also decreases proportionally to <e>.

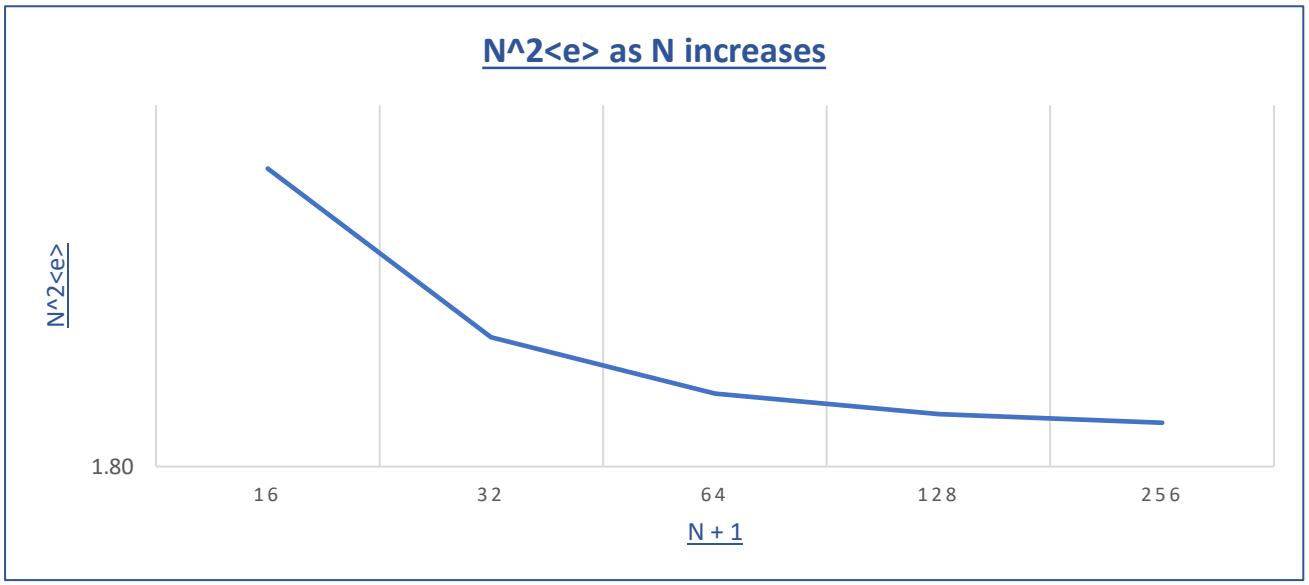

Figure 3 – N^2<e> as N increases

N + 1	N^2<e>
16	2.00378338596
32	1.88572606461
64	1.84794131554
128	1.83422052286
256	1.82861838347

Table 3 – N^2<e> as N increases

It is also apparent from both the log-log plot and the tabulated results (Figure 3 and Table 3) that as N increases, $N^2\langle e\rangle$ is approximately constant, since the value of $N^2\langle e\rangle$ gets closer and closer to 1.8 and doesn't change its value drastically. From the log-log plot we can see the value of $N^2\langle e\rangle$ is asymptotically bounded by ≈ 1.8. Therefore, we can conclude that the mean error <e> decreased proportionally to $\Delta x^2 \propto N^{-2}$.

Question 4

```
#include <iostream>
#include <cmath>
#include <vector>
#include <iomanip>
#include <tuple>
using namespace std;

tuple<double, double> step_rk4 (const double x, const double z, const double
h, const double step,
    double z_prime(const double, const double, const double),
    double h_prime(const double, const double, const double)) {
    //This function is declared to be a tuple so that there can exist two
return values, which are accessed by their positioning in the tuple
    //http://www.cplusplus.com/reference/tuple/

    const double k1 = step * z_prime(z, h, x);
    const double l1 = step * h_prime(z, h, x);
    const double k2 = step * z_prime(z + k1 / 2.0, h + l1 / 2.0, x + step /
2.0);
    const double l2 = step * h_prime(z + k1 / 2.0, h + l1 / 2.0, x + step /
2.0);
    const double k3 = step * z_prime(z + k2 / 2.0, h + l2 / 2.0, x + step /
2.0);
    const double l3 = step * h_prime(z + k2 / 2.0, h + l2 / 2.0, x + step /
2.0);
    const double k4 = step * z_prime(z + k3, h + l3, x + step);
    const double l4 = step * h_prime(z + k3, h + l3, x + step);

    return { z + (k1 + 2.0 * k2 + 2.0 * k3 + k4) / 6.0, h + (l1 + 2.0 * l2 +
2.0 * l3 + l4) / 6.0 };
    //This code multiplies each k and l by 'step' so that the code is neater
    //The first element in the tuple increments the value of z in each time
step by (k1 + 2.0 * k2 + 2.0 * k3 + k4) / 6.0
    //The second element in the tuple increments the value of h in each time
step by (l1 + 2.0 * l2 + 2.0 * l3 + l4) / 6.0
    //This method of computing RK4 is defined in lectures and each k1,k2,k3,k4
are 4 slope estimates between xi and xi+1 and the value of z and h are
incremented by the average of the 4 slope estimates in the manner specified by
the RK4 method to give a more accurate approximation to the curve
    //This is the general method for computing the next step of the fourth
order Runge-Kutta method in a system of ODE's and I have left it in this
manner so that no matter what system of ODE's need to be computed they can
easily be computed by simply defining them and inputting them into the step-
RK4 function
}

double z_prime(const double z, const double h, const double x) {
    if (x == 0) {
        //As the equation is singular at x = 0, we use L'hopital's rule to
work out the value of (2/x)h'(x) at x = 0. Therefore, after differentiating
```

```
both the numerator and denominator we get 2h''(x)/1 and after rearranging our
second order ODE we calculate the value of h''(0) to be equal to -1/3
        return -1.0 / 3.0;
    }
    else {
        return ((-2.0 / x) * z) - h;
    }
}

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

class Error_norm {
private:
    int m;
public:
    Error_norm() : m(1) {}
    Error_norm(int a) : m(a) {}
    double operator()(const vector<double> u, const
        vector<double> v) const;
};

int main() {

    const double x0 = 0.0;
    //Initial value of x
    const double xN = 3.14159265358979;
    //Final value of x
    const int N = 100;
    //Number of points
    const double step = (xN - x0) / double(N);

    const double z0 = 0.0;
    //Initial condition for z
    vector<double> z(N + 2, z0);
    //Declared a double vector z which has an initial element z0 and a size of
N+2

    const double h0 = 1.0;
    //Initial condition for h
    vector<double> h(N + 2, h0);
    //Declared a double vector h which has an initial element h0 and a size of
N+2

    vector<double> x(N + 2, x0);
    //Declared a double vector x which has an initial element x0 and a size of
N+2

    for (int i = 0; i <= N; i++) {
        auto [znext, hnext] = step_rk4(x[i], z[i], h[i], step, z_prime,
h_prime);
        //Given our initial value of x, we compute the value of z[i + 1] and
```

```
h[i + 1] in the next time step of the RK4 method, defined above
        z[i + 1] = znext;
        h[i + 1] = hnext;
        //Our initial value of z0 and h0 were defined initially, therefore the
RK4 method computes the values of the next z and h
        x[i + 1] = (double(i) + 1.0) * (xN - x0) / double(N);
        //Have defined the x[i + 1] in this manner because this will compute x
by multiplying i+1 with the equidistant formula so even if the function were
to be changed, the elements of the vector x would not need to be changed
        //Once one time step has been computed and we have worked out the
value of z[i + 1] and h[i + 1] we then update our value of x so that the RK4
method can work out the values of the next z and h
    }

    vector<double> h_exact(N + 2, 1.0);
    //As sin(x)/x at x = 0 is indeterminate we use L'Hopital's rule to compute
its value at x = 0. Therefore, by differentiating both the numerator and
denominator we get cos(x)/1 and at x = 0 this has a value of one.
    //For this reason I have declared my vector with double variables to have
values of 1.0 and these values will be updated with my for loop as required
    for (int i = 1; i <= N; i++) {
        //I have defined my for loop to be from 1 to N as the initial value of
h_exact is 1.0 and already defined
        h_exact[i] = sin(x[i]) / x[i];
        //sin(x) is a double function by default and I have declared my
vectors to be double so for this reason I don't need to convert them to
doubles in the fraction
        //cout << "h_exact " << i << "\t" << h_exact[i] << endl;
    }

    cout << "i" << "\t" << setw(23) << left << "x" << setw(20) << left << " h
" << setw(20) << left << "\te" << endl;

    vector<double> e(N + 2);
    for (int i = 0; i <= N; i++){
        e[i] = h[i] - h_exact[i];
        if (i % 10 == 0) {
            //prints values of x, h(x), e(x) for t = 0, 10, .., 100 by setting
an if statement to only return those values of i which when divided by 10
leaves a remainder of 0
            cout << i << "\t" << setprecision(10) << setw(20) << left << x[i]
<< "\t" << setw(20) << left << h[i] << "\t" << setw(20) << left << e[i] <<
endl;
        }
    }
    cout << endl;

    Error_norm error1(1);
    double error_norm_l1 = error1(e, e);
    cout << "error norm l1 = "<< error_norm_l1 << endl;
    cout << endl;

    Error_norm error2(2);
```

```
    double error_norm_l2 = error2(e, e);
    cout << "error norm l2 = " << error_norm_l2 << endl;
    //Have explained the uses of classes in Question 2

}

double Error_norm::operator()(const vector<double> u, const
    vector<double> v) const {
    double l = 0.0;
    double sum2 = 0.0;

    if (u.size() == v.size()) {
        for (int i = 0; i < u.size(); i++) {
            sum2 += pow(abs(double(u[i]) * double(v[i])), double(m) / 2.0);
        }
        l = pow(sum2, 1.0 / double(m));
    }
    else {
        cout << "size of vectors aren't equal, therefore, cannot compute." <<
endl;
    }
    return l;
};
```

Question 4(a)

I defined a tuple function to return the values of z_{i+1} and h_{i+1} in order to solve a first order system numerically using the fourth order Runge-Kutta method. To compute the values of z_{i+1} and h_{i+1} I first computed the 4 slope estimates k1, k2, k3, k4 (and l1, l2, l3, l4) and incremented the value of z and h by the average of the 4 slope estimates in the manner specified by the Runge-Kutta method. These estimates increment the value of z and h based on their positioning in the tuple. This step is repeated N times using N equidistant points between 0 and π. Finally, I outputted the values of x_0, x_{10}, x_{20}, …, x_{100} and $h(x_0)$, $h(x_{10})$, $h(x_{20})$, ..., $h(x_{100})$. The code I have written works for all system of ODEs.

I have tabulated the values of x and h(x) below:

Values of x		Values of h(x)	
x_0 =	0	$h(x_0)$ =	1
x_{10} =	0.3141592654	$h(x_{10})$ =	0.9836316417
x_{20} =	0.6283185307	$h(x_{20})$ =	0.9354892818
x_{30} =	0.9424777961	$h(x_{30})$ =	0.8583936892
x_{40} =	1.256637061	$h(x_{40})$ =	0.7568267265
x_{50} =	1.570796327	$h(x_{50})$ =	0.6366197703
x_{60} =	1.884955592	$h(x_{60})$ =	0.5045511503
x_{70} =	2.199114858	$h(x_{70})$ =	0.3678830084
x_{80} =	2.513274123	$h(x_{80})$ =	0.2338723187

x_{90} =	2.827433388	h(x_{90}) =	0.1092924023
x_{100} =	3.141592654	h(x_{100}) =	-0.00000000271

Table 4 – Values of x and values of h(x)

Question 4(b)

To compute the difference between my numerical solution of h(x) and the exact solution of h(x) I defined a vector to store my values of $h_{numerical}(x)$ and $h_{exact}(x)$ and defined a new vector 'e' which stored the values of the error.

I have tabulated my results below:

Error values	
e(x0) =	0
e(x10) =	-0.00000000141424095
e(x20) =	-0.00000000197501293
e(x30) =	-0.00000000211239326
e(x40) =	-0.00000000212849616
e(x50) =	-0.00000000211063267
e(x60) =	-0.00000000210787454
e(x70) =	-0.00000000215355156
e(x80) =	-0.00000000226776131
e(x90) =	-0.00000000245618431
e(x100) =	-0.00000000270896458

Table 5 – Error values

Question 4(c)

Using the Weighted_norm object I defined in question 2b, I computed the error norms.

For $l_1(\vec{e},\vec{e}) = \sum_{i=0}^{N}|e_i|$, I used the parameter m = 1 and the error value vector as arguments in the member function.

For $l_2(\vec{e},\vec{e}) = \sqrt{\sum_{i=0}^{N}|e_i|^2}$, I used the parameter m = 2 and the error value vector as arguments in the member function.

In turn, this computed the error values l1 and l2.

"error norm l1 = 2.097641994e-07

error norm l2 = 2.166072657e-08"

Question 5

The exact value of the integral I = $\int_{-1}^{1} \frac{1}{1+25x^2}$ dx is 0.549360306778.

I will use this value for all parts of question 5 to determine how accurate my numerical integration approximations are.

```
#include <iostream>
#include <cmath>
#include <vector>
#include <iomanip>
using namespace std;

double integral(const double x) {
	return 1.0 / (1.0 + (25.0 * (x * x)));
	//Have defined a function to return the f(x) of the integral provided in
our question
}

const double I_exact = (2.0 / 5.0) * double(atan(5.0));
//This is the exact value of the definite integral when integrated, to be used
when comparing the exact answer of the integration with the methods computed
numerically

double inner_product(const vector<double>& u, const vector<double>& v);

double Trapezium_rule(const double a, const double b, const double N, double
func(const double)) {

	const double delta_x = (b - a) / double(N);
	//Declaring a variable that computes delta x. To be used when working
out the weights
	//cout << "delta x has value of " << delta_x << endl;

	vector<double> x(N + 1);
	//Declaring a vector x of size N+1 to hold entries of double numbers for
the values of xi
	for (int i = 0; i < x.size(); i++) {
		x[i] = double(a) + double(i) * (double(b) - double(a)) / double(N);
		//I am computing x[i] using the equidistant nodes formula mentioned
in lectures { xi = a + i*(b-a)/N, i = 0,1,....,N
	}

	vector<double> w(N + 1);
	//Declaring a vector w of size N+1 to hold entries of double numbers for
the weights wi
	for (int i = 0; i < w.size(); i++) {
		if (i == 0 || i == N) {
			w[i] = delta_x / 2.0;
		}
```

```
            else {
                  w[i] = delta_x;
            }
            //I have made an if and else statement to compute the weights wi
for when i = 0 or i = N, and 1 <= i <= N-1
      }

      //cout << "i" << "\t" << setw(15) << left << "x" << setw(15) << left <<
" w " << setw(18) << left << "\tfx" << endl;

      vector<double> fx(N + 1.0);
      for (int i = 0; i < fx.size(); i++) {
            fx[i] = func(x[i]);
            //This vector computes the value of fx[i] by inputting the values
of x[i] into the function
            //cout << i << "\t" << setprecision(8) << setw(15) << left << x[i]
<< "\t" << setw(15) << left << w[i] << "\t" << setw(15) << left << fx[i] <<
endl;
      }

      const double I_trapezium = inner_product(w, fx);
      //To compute the integral I using the trapezium rule, I use the function
defined in Question 2a and compute the inner product of vectors wi and fi

      cout << "The value of the integral computed using the composite
trapezium rule is " << setprecision(12) << I_trapezium << endl;

      //cout << "The exact value of the integral is " << setprecision(12) <<
I_exact << endl;

      cout << "The difference between the numerical integral and the exact
integral is " << "";

      return I_trapezium - I_exact;
      //The function returns the difference between the Integral computed
using the trapezium rule and the exact integral
}


int main() {
      const double a = -1.0;
      //Lower bound of the definite integral
      const double b = 1.0;
      //Upper bound of the definite integral
      const double N = 63.0;
      //Total number of equidistant points

      cout << Trapezium_rule(a, b, N, integral);
      //The trapezium rule accepts arguments: lower bound of the integral,
upper bound of the integral, total number of points and the f(x) of the
integral being computed
      cout << endl;
```

```
}

double inner_product(const vector<double>& u, const vector<double>& v) {
	double sum1;
	sum1 = 0.0;

	if (u.size() == v.size())
		for (int i = 0; i < u.size(); i++) {
			sum1 += double(u[i]) * double(v[i]);
		}
	else {
		cout << "size of vectors aren't equal, therefore, cannot compute."
<< endl;
	}
	return sum1;
}
//This is the function inner_product. This codes functionality was
demonstrated in question 2a
```

Question 5(a)

I defined a function that computed the definite integral of any f(x) using the composite trapezium rule. I first defined a vector x using N + 1 = 64 equidistant points in x ∈ [-1, 1], which stored the values of the grid-points x_i. I then computed the values of $f(x_i)$ by inputting the values of x_i as arguments into the function. Following this, I stored the values of w_i in the vector 'w'. Finally, I computed the inner product of the vector w_i with the vector f_i ($\sum_{i=0}^{N} w_i f_i$), the result of which gave the value of the definite integral computed using the trapezium rule.

The code gives the following results:

The value of the integral computed using the composite trapezium rule is:

0.549347885299

The difference between the numerical integral and the exact integral is:

-0.0000124214785682 (-1.24214785682e-05)

Question 5(b)

```
#include <iostream>
#include <cmath>
#include <vector>
#include <iomanip>
```

```
using namespace std;

double integral(const double x);

double inner_product(const vector<double>& u, const vector<double>& v);

const double I_exact = (2.0 / 5.0) * double(atan(5.0));

double f_prime_minus(const double a, const double b) {
	//To use the composite Hermite integration rule we need to compute
[f'(a) - f'(b)] for any given function, therefore, I have defined a function
to do exactly this
	vector<double> f_prime_vec(2);
	//I have defined a vector f_prime_vec to store the values of f'(a) and
f'(b)
	double minus_primes = 0;
	for (int i = 0; i < f_prime_vec.size(); i++) {
		//The derivative of f'(x) = -50x / (1+25x^2)^2
		f_prime_vec[0] = (-50.0 * double(a) / (pow(1.0 + 25.0 * double((a *
a)), 2.0)));
		//The derivative of f(x) at x = a is f'(a) = -50(a) / (1+25(a)^2)^2
		f_prime_vec[1] = (-50.0 * double(b) / (pow(1.0 + 25.0 * double((b *
b)), 2.0)));
		//The derivative of f(x) at x = b is f'(b) = -50(b) / (1+25(b)^2)^2
		minus_primes = f_prime_vec[0] - f_prime_vec[1];
		//Finally, I compute the value of f'(a) - f'(b) by subtracting
f_prime_vec[0] and f_prime_vec[1]
	}
	return minus_primes;
}

double Hermite_integ(const double a, const double b, const double N, double
func(const double), double diff_func(const double, const double)) {

	const double delta_x = (double(b) - double(a)) / double(N);
	//cout << "delta x has value of " << delta_x << endl;

	vector<double> x(N + 1);
	for (int i = 0; i < x.size(); i++) {
		x[i] = double(a) + double(i) * (double(b) - double(a)) / double(N);
	}

	vector<double> w(N + 1.0);
	for (int i = 0; i < w.size(); i++) {
		if (i == 0 || i == N) {
			w[i] = double(delta_x) / double(2.0);
		}
		else {
			w[i] = double(delta_x);
		}
	}
```

```
	//cout << "i" << "\t" << setw(15) << left << "x" << setw(15) << left << 
" w " << setw(18) << left << "\tfx" << endl;

	vector<double> fx(N + 1);
	for (int i = 0; i < fx.size(); i++) {
		fx[i] = func(x[i]);
		//cout << i << "\t" << setprecision(8) << setw(15) << left << x[i] 
<< "\t" << setw(15) << left << w[i] << "\t" << setw(15) << left << fx[i] << 
endl;
	}

	const double I_hermite = inner_product(w, fx) + ((delta_x * delta_x) / 
12.0) * double(diff_func(a, b));
	//To compute the integral I using the Hermite integration rule, I use 
the function defined in Question 2a and compute the inner product of vectors 
wi and fi and add delta_x^2/12*[f'(a) - f'(b)]

	cout << "The value of the integral computed using the composite Hermite 
integration rule is " << setprecision(12) << I_hermite << endl;

	//cout << "The exact value of the integral is " << setprecision(12) << 
I_exact << endl;

	cout << "The difference between numerical integral and the exact 
integral is " << "";

	return I_hermite - I_exact;
	//The function returns the difference between the Integral computed 
using the Hermite integration rule and the exact integral
}



int main() {

	const double a = -1.0;
	//Lower bound of the definite integral
	const double b = 1.0;
	//Upper bound of the definite integral
	const double N = 63.0;
	//Total number of equidistant points

	cout << Hermite_integ(a, b, N, integral, f_prime_minus);
	//The Hermite integration rule accepts arguments: lower bound of the 
integral, upper bound of the integral, total number of points, the f(x) of the 
integral being computed and the difference between f'(a) and f'(b)
	cout << endl;
}

double integral(const double x) {
	return 1.0 / (1.0 + (25.0 * (x * x)));
}
```

```
double inner_product(const vector<double>& u, const vector<double>& v) {
	double sum1;
	sum1 = 0.0;

	if (u.size() == v.size())
		for (int i = 0; i < u.size(); i++) {
			sum1 += double(u[i]) * double(v[i]);
		}
	else {
		cout << "size of vectors aren't equal, therefore, cannot compute."
<< endl;
	}
	return sum1;
}
```

Question 5(b)

I defined a function that computed the definite integral of any f(x) using the composite Hermite integration rule. Firstly, I defined a vector x using N + 1 = 64 equidistant points in x ∈ [-1, 1], which stored the values of the grid-points x_i. The values of $f(x_i)$ were then computed by inputting the values of x_i as arguments into the function. Following this, I stored the values of w_i in the vector 'w'. Finally, the inner product of the vector w_i with the vector f_i ($\sum_{i=0}^{N} w_i f_i$) was calculated - this result was added to $\frac{\Delta x^2}{12}[f'(a) - f'(b)]$. This returned the value of the definite integral computed using the Hermite integration rule.

The code gives the following results:

```
The value of the integral computed using the composite Hermite integration
rule is:
0.549360308999
```

```
The difference between the numerical integral and the exact integral is:
0.00000000222127927163 (2.22127927163e-09)
```

Question 5(c)

```
#include <iostream>
#include <cmath>
#include <vector>
#include <iomanip>
```

```
using namespace std;

double integral(const double x);

const double I_exact = (2.0 / 5.0) * atan(5);

double inner_product(const vector<double>& u, const vector<double>& v);

double Clenshaw_curtis(const double a, const double b, const double N, double
func(const double)) {

	const double delta_x = (b - a) / double(N);
	//cout << "delta x has value of " << delta_x << endl;

	vector<double> theta(N + 1);
	//Declaring a vector theta of size N+1 to hold entries of double
numbers for the values of theta_i
	for (int i = 0; i < theta.size(); i++) {
		const double pi = 3.14159265358979;
		theta[i] = (double(i) * pi) / double(N);
		//This for loop computes the value of theta at each i using the
formula provided in the question
	}

	vector<double> x(N + 1);
	for (int i = 0; i < x.size(); i++) {
		x[i] = double(-cos(theta[i]));
	}

	vector<double> w(N + 1);
	//Declaring a vector 'w' of size N+1 to hold entries of double numbers
for the values of the weights wi

	for (int i = 0; i < w.size(); i++) {
		if (i == 0 || i == N) {
			w[i] = 1.0 / (double(N) * double(N));
		}
		else {
			double sum = 0.0;
			for (int k = 1; k <= ((double(N) - 1.0) / 2.0); k++) {
				sum += (2.0 * cos(2.0 * double(k) * theta[i])) / (4.0
* (double(k) * double(k)) - 1.0);
				//To compute the weights wi for 1 <= i <= N - 1 I
first compute the value of the sum (2cos(2k(thetai))/4(k^2)-1 from k = 1 to
(N - 1)/2
				w[i] = (2.0 / double(N) * (1.0 - sum));
				//After computing the sum, I work out the value of wi
by doing 2/N*(1 - sum)
				//I have made an if and else statement to compute the
weights wi for when i = 0 or i = N, and 1 <= i <= N-1
			}
		}
```

```
	}

	//cout << "i" << "\t" << setw(15) << left  << "theta" << setw(15) << left << "x" << setw(15) << left << "  w " << setw(18) << left << "\tfx" << endl;


	vector<double> fx(N + 1);
	for (int i = 0; i < fx.size(); i++) {
		fx[i] = func(x[i]);
		//cout << i << "\t" << setprecision(8) << setw(15) << left << theta[i] << setw(15) << left << x[i] << "\t" << setw(15) << left << w[i] << "\t" << setw(15) << left << fx[i] << endl;

	}

	const double I_clenshawcurtis = inner_product(w, fx);
	//To compute the integral I using the Clenshaw-Curtis quadrature rule, I use the function defined in Question 2a and compute the inner product of vectors wi and fi


	cout << "The value of the integral computed using the Clenshaw-Curtis quadrature rule is " << setprecision(12) << I_clenshawcurtis << endl;

	//cout << "The exact value of the integral is " << setprecision(12) << I_exact << endl;

	cout << "The difference between the numerical integral and the exact integral is " << "";

	return I_clenshawcurtis - I_exact;
	//The function returns the difference between the Integral computed using the Clenshaw-Curtis quadrature rule and the exact integral

}


int main() {
	const double a = -1.0;
	const double b = 1.0;
	const double N = 63.0;

	cout << Clenshaw_curtis(a, b, N, integral);
	//The Clenshaw-Curtis quadrature rule accepts arguments: lower bound of the integral, upper bound of the integral, total number of points and the f(x) of the integral being computed
	cout << endl;
}

double inner_product(const vector<double>& u, const vector<double>& v) {
	double sum1;
```

```
	sum1 = 0.0;

	if (u.size() == v.size())
		for (int i = 0; i < u.size(); i++) {
			sum1 += double(u[i]) * double(v[i]);
		}
	else {
		cout << "size of vectors aren't equal, therefore, cannot
compute." << endl;
	}
	return sum1;
}

double integral(const double x) {
	return 1.0 / (1.0 + (25.0 * (x * x)));
}
```

Question 5(c)

I defined a function that found the definite integral of any f(x) using the Clenshaw-Curtis quadrature rule. I first defined a vector x on a grid of N + 1 = 64 points where x_i = -cos(θ_i) and $\theta_i = \frac{i\pi}{N}$, $i = 0, 1, \ldots, N$; the vector then stored these values. I then computed the values of f(x_i) by inputting the values of x_i as arguments into the function. Following this, I stored the values of w_i in the vector 'w'. Finally, I computed the inner product of the vector w_i with the vector f_i ($\sum_{i=0}^{N} w_i f_i$); the result of this gave the value of the definite integral, which was computed using the Clenshaw-Curtis quadrature rule.

The code gives the following results:

The value of the integral computed using the Clenshaw-Curtis quadrature rule is:

0.549360306757

The difference between the numerical integral and the exact integral is:

-0.000000000212625472784 (-2.12625472784e-11)

Question 5(d)

```
#include <iostream>
#include <cmath>
#include <vector>
#include <iomanip>
#include <random>
#include <algorithm>
using namespace std;

double integral(const double x);

const double I_exact = (2.0 / 5.0) * atan(5);

double max_c(const double a, const double b, double func(const double)) {
	//I have defined this function as such, to return the largest value of
f(x).
	//Rather than computing the value analytically I have adopted this
method because it can be used to calculate the maximum value any function can
have

	const double arbitrary_num = 1000.0;
	//Declaring a variable arbitrary_num to declare the size of my vector as
well as to compute the maximum f(x) using arbitrary_num of points (1001)

	vector<double> x(arbitrary_num);
	for (int i = 0; i < x.size(); i++) {
		x[i] = a + i * (b - a) / double(arbitrary_num);
		//Calculating points for x[i] using the equidistant method
	}

	vector<double> fx(arbitrary_num);
	for (int i = 0; i < fx.size(); i++) {
		fx[i] = func(x[i]);
		//This will compute all of the values of the function f(x) by
inputting the values of xi
	}
	return *max_element(fx.begin(), fx.end());
	//This returns the maximum value of my vector fx and hence the maximum
value of any function
	//I have used the asterisk (*) before the max_element() function as this
returns an iterator.
	//https://stackoverflow.com/questions/9874802/how-can-i-get-the-max-or-
min-value-in-a-vector
}
double Monte_carlo(const double a, const double b, double func(const double))
{

	double area_rec = max_c(a, b, func) * (b - a);
	//Declared the variable area_rec to compute the area of the rectangle
between a <= x <= b and 0 <= fx <= c
```

```
	const int seed = 93;
	//I am setting my seed as 93 to initialise the sequence of uniform
random numbers
	mt19937_64 rand(seed);
	//I am using the Mersenne Twister algorithm to generate my uniform
random numbers because it is better than the linear congruential method
	//mt19937_64 is the object that will generate the uniform random numbers
	//I have declared my variable to be called rand and have supplied my
seed to it
	//Combining the two will generate uniform random numbers
	uniform_real_distribution<double> X(a, b);
	//I am generating uniform random numbers for X between the values of a
and b
	uniform_real_distribution<double> W(0, max_c(a, b, func));
	//I am generating uniform random numbers for W between 0 and the maximum
of f(x) (a <= x <= b)

	const int Monte_carlo_N = 10000;
	//The number of times I want my for loop to run and hence also the
amount of uniform random variables I want to generate
	int M = 0;
	for (int i = 0; i < Monte_carlo_N; i++) {
		const double x = X(rand);
		//This will generate uniform random numbers for x between a and b
		const double w = W(rand);
		//This will generate uniform random numbers for w between 0 and
c_max
		//Therefore, these combined will generate points randomly within
the region specified (in a rectangle where -1 < x < 1 and 0 < y < 1)

		if (w <= func(x))
			//The if statement will ensure that all of the points that
are under the function curve will be accounted for by the counter M as M will
increase in value by 1 if and only if W <= f(X)
			M++;
	}

	const double I_MC = (double(M) * area_rec) / double(Monte_carlo_N);

	cout << setprecision(16) << "The value of the integral computed using
the hit and miss Monte Carlo method with N = 10000 is " << I_MC << endl;

	//cout << "The exact value of the integral is " << setprecision(12) <<
I_exact << endl;

	cout << "The difference between the numerical integral and the exact
integral is " << "";

	return I_MC - I_exact;
}
int main() {
	const double a = -1.0;
	const double b = 1.0;
```

```
	const double N = 63.0;

	cout << (setprecision(16)) << Monte_carlo(a, b, integral) << endl;

	//cout << max_c(a, b, integral);
}

double integral(const double x) {
	return 1.0 / (1.0 + (25.0 * (x * x)));
}
```

Question 5(d)

I defined a function that computed the definite integral of any f(x) using the hit and miss Monte Carlo method with N = 10000. I first defined a function that computed the maximum value (c) of a function f(x) so that my code can be used to work out the integral of any function. Following this, I computed the area of the rectangle between a <= x <= b and 0 <= fx <= c. Once this was completed, I generated points randomly for X between a and b, and generated points randomly for W between 0 and c. Within this region of the rectangle I set up a counter (M) so that whenever a randomly generated point fell below the curve, the counter (M) increased in value by 1. This process was repeated for however many values the for loop was defined for (in this case, it was N = 10000), and hence counted what proportion of points had $W \leq f(X)$.

Consequently, the proportion M/N approximates the ratio of the area under f and the area of the rectangle. Therefore, by rearranging this formula, the integral estimate I_H = M*area_rec/N can be computed.

The code gives the following results:

The value of the integral computed using the hit and miss Monte Carlo method with N = 10000 is:

0.5492

The difference between the numerical integral and the exact integral is:

-0.0001603067780063805

Question 6

```
#include <iostream>
#include <cmath>
#include <vector>
#include <iomanip>
#include <tuple>
```

```
using namespace std;

tuple<double, double> step_rk2(const double t, const double q, const double p,
const double step,
    double q_prime(const double, const double, const double),
    double p_prime(const double, const double, const double)) {
    //This function is declared to be a tuple so that there can exist two
return values which are accessed by their positioning in the tuple
    //http://www.cplusplus.com/reference/tuple/

    const double k1 = step * q_prime(q, p, t);
    const double l1 = step * p_prime(q, p, t);
    const double k2 = step * q_prime(q + k1 / 2.0, p + l1 / 2.0, t + step /
2.0);
    const double l2 = step * p_prime(q + k1 / 2.0, p + l1 / 2.0, t + step /
2.0);

    return { q + k2, p + l2 };
    //This code multiplies each k and l by 'step' so that the code is neater
    //The first element in the tuple increments the value of q in each time
step by k2
    //The second element in the tuple increments the value of p in each time
step by l2
    //This method of computing Runge-Kutta 2 midpoint method is defined in
lectures and we approximate p and q at the midpoint of the interval to give us
the slope k2 and l2, we then use this slope k2 and l2 to compute the next
value of q and p
    //This is the general method for computing the next step of the Runge-
Kutta 2 midpoint method in a system of ODE's and I have left it in this manner
so that no matter what system of ODE's need to be computed they can easily be
computed by simply defining them and inputting them into the step-RK2 function
}

double q_prime(const double q, const double p, const double t) {
    return p;
}

double p_prime(const double q, const double p, const double t) {
    return -q;
}

int main() {
    const double t0 = 0.0;
    //Initial time
    const double tN = 100.0;
    //Final time
    const int N = 1000;
    //Number of points is N+1; therefore, the number of intervals is N
    const double step = (tN - t0) / double(N);
    //Time step is 0.1, therefore, minus the difference between the initial
time and final time and divide by N

    const double q0 = 0;
```

```
    //Initial condition for q0
    vector<double> q(N + 2, q0);
    //Declared a vector q which has an initial element q0 and a size of N+2

    const double p0 = sqrt(2);
    //Initial condition for p0
    vector<double> p(N + 2, p0);
    //Declared a vector p which has an initial element p0 and a size of N+2

    vector<double> t(N + 2, t0);
    //Declared a vector t which has an initial element t0 and a size of N+2

    for (int i = 0; i <= N; i++) {
        auto [qnext, pnext] = step_rk2(t[i], q[i], p[i], step, q_prime,
p_prime);
        q[i + 1] = qnext;
        p[i + 1] = pnext;
        //Our initial value of q0 and p0 were defined initially, therefore,
the RK2 midpoint method computes the values of the next q and p
        t[i + 1] = (double(i) + 1.0) * (double(tN) - double(t0)) / double(N);
        //Have defined the t[i + 1] in this manner because this will compute t
by multiplying i+1 with the equidistant formula so even if the function were
to be changed this would not need to be changed
        //Once one time step has been computed and we have worked out the
value of q[i + 1] and p[i + 1] we then update our value of t so that the RK2
midpoint method can work out the next values of q and p
    }

    vector<double> E(N + 2);
    for (int i = 0; i <= N; i++) {
        E[i] = (1.0 / 2.0) * ((p[i] * p[i]) + (q[i] * q[i]));
    }

    vector<double> e(N + 2);
    for (int i = 0; i <= N; i++) {
        e[i] = E[i] - E[0];
    }

    cout << setw(7) << left << "t" << setw(18) << left << "Position q(t)" <<
setw(18) << left << "Momentum p(t) " << setw(18) << left << "Energy E(t)" <<
setw(18) << left << "Difference e(t)" << setw(18) << endl;
    cout << setprecision(7) << setw(7) << left << 0 << setw(18) << left <<
q[0] << setw(18) << left << p[0] << setw(18) << left << E[0] << setw(18) <<
left << e[0] << setw(18) << endl;

    for (int j = 10; j <= N; j *= 10) {
        cout << setprecision(12) << setw(7) << left << t[j] << setw(18) <<
left << q[j] << setw(18) << left << p[j] << setw(18) << left << E[j] <<
setw(18) << left << e[j] << setw(18) << endl;
    }
}
```

Question 6

I defined a tuple function to return the values of q_{i+1} and p_{i+1} in order to solve a first-order system numerically using the second order Runge-Kutta midpoint method. To compute the values of q_{i+1} and p_{i+1} I first computed the slope estimates k2 and l2 and incremented the value of q and p by the slope k2 and l2. The positioning of q and p in the tuple determine which value is incremented. This step was repeated N times using N equidistant points between 0 and 100. As the final time is t = 100 and the initial time is t = 0, the number of N could be computed as $\Delta t = 0.1$, therefore, N was equal to 1000. Finally, I output the values of the position q(t), momentum p(t), energy E(t) and the difference e(t) = E(t) – E(0) for t = 0, t = 1, t = 10 and t = 100. The code I have written works for all system of ODEs.

I have tabulated the values of q(t), p(t), E(t) and e(t) below:

t	Position q(t)	Momentum p(t)	Energy E(t)	Difference e(t)
0	0	1.414214	1	0
1	1.191436624660	0.762219670224	1.00025002813	0.000250028126875
10	-0.789959298054	-1.17514701207	1.00250309628	0.00250309627809
100	-0.510884580084	1.33776924246	1.02531480012	0.0253148001188

Table 6 – Table of values for q(t), p(t), E(t) and e(t)

Between t = 0 and t = 100, E(t) is constant numerically, however, it bears noting that as t increases, the value of E(t) also increases. Therefore, if this trend continues (even if the increase is by an infinitesimal amount each time), when t = 10000, the difference will be 2.5. Thus, E(t) will no longer remain constant.

Bibliography:

1. http://www.cplusplus.com/reference/algorithm/max_element/
2. http://www.cplusplus.com/doc/tutorial/operators/
3. https://www.programiz.com/cpp-programming/library-function/cmath/exp
4. https://www.geeksforgeeks.org/constructors-c/
5. https://stackoverflow.com/questions/24999861/determining-if-two-numbers-are-almost-equal-and-outputting-the-result
6. https://www.arduino.cc/reference/en/language/structure/further-syntax/define/
7. https://stackoverflow.com/questions/15704565/efficient-way-to-return-a-stdvector-in-c
8. https://stackoverflow.com/questions/9874802/how-can-i-get-the-max-or-min-value-in-a-vector
9. https://stackoverflow.com/questions/2728190/how-are-iterators-and-pointers-related
10. http://www.cplusplus.com/reference/algorithm/max_element/

11. https://www.slideshare.net/batuhanyil23/es272-ch7
12. http://www.cplusplus.com/reference/tuple/