MTH6150: Numerical Computing in C & C++

FINAL PROJECT

Waqil Rahman

190484682

Contents

Preface:

In order to avoid repetition of explanations within this report and comments on the .cpp files I have written below a series of explanations. These explanations are to explain parts of code which are 'basic' but is needed to understand in order to maintain a high level of accuracy when the each code is being run.

static_cast<long double>('expression') – This particular bit of code allows to avoid accuracy loss when the expression takes a value in the form of int and needs to be redefined as a long double. This is because if we were carry out integer division we would lose the decimal point values as it wont be stored.

using namespace std; - This removes the need to include std:: before every cout and cin making it much more easier to read and review ones code.

Const – This specifies that a variable's value is constant and tells the compiler to prevent the programmer or the code from modifying it.

#include <iostream> … - This list of pre-processor directives are needed to enable a programme to run as it would process each of those labelled libraries to call on functions stored in them in order to allow the programme to run without errors. An example of this is using #include <cmath> in order to allow the programme to understand the code exp(10); .

Question 1:

Code:

```
#include <iostream>
#include <cmath>
#include <iomanip>
#define EP 1E-15
//this defines the epsilon
using namespace std;

int main() {

    long double x0 = 1.0;
    //initial guess is defined

    for (int i = 0; x0 < 2.0; ++i) {
    //for loop will continue until the value of x is more than 2(this won't occur due to the nature of the function
        long double x1 = exp(-x0);
        //This outputs the result of the function
        long double iterations = i + 1.0 ;
        //This long double stores number of iterations until the end of the loop or if the loop breaks

        if (abs(x1-x0) <= EP){
        //This if statement sets the conditions for the final value of x
            cout << setprecision(18) << "The final value of x: " << x0 << endl;
            //This outputs to 18 digits the final value of x
            cout << "The number of iterations: " << iterations << endl;
            //This outputs the number iterations occurred stored from the double
            long double final_value = x0 - exp(-x0);
            //This calculates the error
            cout << "The error in the transcendental equation is: " << final_value << endl;
            break;
            //This breaks the entire for loop
        }
        x0 = x1;
        //This re-assignment of doubles allows for the recursive iteration to loop with the new value
    }

    return 0;
}
```

Output:

```
The final value of x: 0.567143290409784506
The number of iterations: 61
The error in the transcendental equation is: 9.92370248475982208e-16
Program ended with exit code: 0
```

A) From the code output, we know the answer is 0.567143290409784506 to 18 significant points which is stored in long double x0. I know this value is near to the solution as $x_{inifinity}$ is 0.567143 which is similar to my solution.

B) Since we implemented the long double variable 'iterations' which would increase by 1.0 every time the '*if*' loop is executed. When the loop is broken from the *break* function the final value of the number of iterations is 61(as we output the long double 'iterations') which is shown in the code output.

C) We calculate the transcendental error by using the equation mentioned in the question by inputting our final value of x. The error is: 9.932470248475982208e-16, this is what I expected as I had programmed the 'if' loop to break when |x_new – x_old| is less than or equal to the stated epsilon. In this case, it will always be less than epsilon as it would be very improbable for x to be equal to 1e-15.

Question 2:

Code:

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

long double dot(const valarray<long double> a,const valarray<long double> b){
    return (a*b).sum();
    //This valarray calculation returns the sum of A*B
}
//This answers q2a

//The function cdot is defined in week10 notes - compensated summation.cpp - KahanSum function
long double cdot(const valarray<long double> v){
    long double sum = 0.;
    long double c = 0.;
    long double y;
    long double t;
    for(int i = 0; i < v.size(); ++i){
            y = v[i] - c;
            t = sum + y;
            c = (t - sum) - y;
            sum = t;
    }
    return sum;
}

class normal_cal{
//using class allows to extract different components out the function
    int m;
public:
    normal_cal(int M) : m(M){ }
    double operator()(const valarray<long double> a) const {
    //This initialises the valarray into the class and uses function object to call on the valarray
        long double normal = 0.;
        //We let normal be zero as we are going to add to it
        for (int i = 0; i <= a.size(); ++i){
        //the for loop will go through the entirety of the valarray
            normal += pow(abs(a[i]),m);
            //This double computes summation part of the normal
        }return pow(static_cast<long double>(normal),0.5);
        //The function will now return the final value of the normal
    }
};

int main(){
    cout<< setprecision(20);
    //We use the setprecision function to see the accuracy of the programme outputs
    long double n = pow(10.0,6.0);
    //Defined from question to 10^6 from library cmath
    valarray<long double>a(n);
    //We initialise valarray a with size n
    for (double i = 1; i < a.size(); ++i){
        //This creates the valarray of the Euclidean n-valarray
        a[i] = 1 /static_cast<long double>(i);
    }
```

```
    cout << "Product of dot: " << dot(a,a)<<endl;
    //Using the dot function we sum A and A together
    long double pi = 3.1415926535897932385;
    cout << "Difference in dot and the actual answer: " << dot(a,a) - (pow(pi,2.0)/6.0) << endl;
    //This answers q2b

    valarray<long double>x(0.1,n);
    //This is the constant Euclidean constant n-valarray for the dot function
    valarray<long double>x_new(0.01,n);
    //This is the constant Euclidean constant n-valarray for the dot function
    //We have to use 0.01 as that is c^2
    long double nc2 = 10000.0;
    // 10000 = (0.1^2)*(10^6)
    // We use the numerical exact value instead to avoid accuracy loss when the programme calculates
the large number

    cout << endl;
    cout << "Product of dot: " << dot(x,x) << endl;
    //Using the dot function we sum x and x together to find the sum of nc^2 using the dot function
    cout << "Product of cdot: " << cdot(x_new) << endl;
    //Using the dot function we sum x_new to find the sum of nc^2 using the KahanSum method
    cout << "Difference in cdot and the actual answer: " << cdot(x_new) - nc2 << endl;
    //This answers q2c

    normal_cal norm(2);
    //This uses the function object and defines the int m = 2
    long double normal = norm(a);
    //This uses the function object and defines the valarray object valarray a
    cout << endl << "The result of norm l2('valarray'a) is: " << normal << endl;
    //This answers q2d

    return 0;
}
```

Output:

```
Product of dot: 1.6449330668477264373
Difference in dot and the actual answer: -1.0000004998708938303e-06

Product of dot: 9999.9999999998775184
Product of cdot: 10000.000000000000208
Difference in cdot and the actual answer: 2.0783375020982930437e-13

The result of norm l2('valarray' a) is: 1.2825494403132093879
Program ended with exit code: 0
```

A) I have defined the dot function before main() and the code explaining it is simply understood as it uses valarray multiplication with .sum() to return the final value.

B) I first create the valarray A within main() with size n as specified in the question. Then using the dot function I had outputted the final result and the exact value difference which is shown in the output. We get for dot: 1.6449330668477264373, with the difference being: -1.0000004998708938303e-06.

C) Since the cdot function already exists in the KahanSum function in week 10 notes (Compensated Summatation.cpp) I have just adjusted the valarray x to have the value c squared already instead of having to create two input values for cdot which reduces accuracy loss in the output. We also can see the difference in values from dot and cdot and the exact value. The result of cdot is :

10000.000000000000208, with the difference between cdot and the exact answer: 2.0783375020982930437e-13.

D) Using the class normal_cal we can implement the double operator()… into the function in order to create a function object. The result of finding the $l_m(A)$ is: 1.2825494403132093879 . I had expected this result as mathematically the exact result should be the square root of $pi^2/6$ which is 1.28254… (Source: https://www.symbolab.com/solver/step-by-step/%5Csqrt%7B%5Cfrac%7B%5Cpi%5E%7B2%7D%7D%7B6%7D%7D?or=input). Additionally $l_m(A)^2$ = dot(A,A) (Source: https://www.symbolab.com/solver/step-by-step/1.2825494403132093879%5E%7B2%7D?or=input) if we compute it within the .cpp file which is implied from the previous statement of square rooting the exact value.

Question 3

Code:

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

long double normal_calculation(int m, valarray<long double>a){
    //We have turned the normal_cal function object into a function so it's easier to use to avoid longer
calculation times and less re-defining of valarrays when the function is called several times
    long double norm = 0;
    for (int i = 0; i <= a.size(); ++i){
        norm += pow(abs(a[i]),m);
    }
    long double normal = pow(norm,1/static_cast<long double>(m));
    //This takes the normal formula from 2d in the form of a function instead of function object
    return normal;
}

void fanalytical(int N){
    valarray<long double> gridpoint(N+1);
    valarray<long double> fx(N+1);
    valarray<long double> fddx(N+1);
    //This initialises the valarray to the correct size for given N

    long double delta_x2 = pow(2.0 / static_cast <long double>(N),2.0);
    //We calculate the long double here which is required to calculate other doubles

    for (int i = 0; i < N+1; ++i) {
        gridpoint[i] = (2.0 * double(i) - (N)) / (N); //xi formula
        fx[i] = exp(-16.0 * (pow(gridpoint[i], 2.0))); //f(xi) formula
        fddx[i] = -32.0 * (-32.0 * fx[i] * pow(gridpoint[i], 2.0) + fx[i]);
    }//This series of for loops inputs the correct values into each respective valarray

    valarray <long double> f__(N+1);
    valarray <long double> ei(N+1);
    for (int i =0;i<N+1;++i){
        if (i == 0) {
            f__[0] = (fx[i + 2] - 2.0 * fx[i + 1] + fx[i]) /static_cast <long double>(delta_x2);
            ei[0] = fddx[0] - f__[0];
        } else if (i == N) {
            f__[N] = (fx[N] - 2.0 * fx[N - 1] + fx[N - 2]) /static_cast <long double>(delta_x2);
            ei[i] = fddx[N] - f__[N];
        } else {
            f__[i] = (fx[i + 1] - 2.0 * fx[i] + fx[i - 1]) / static_cast <long double>(delta_x2);
            ei[i] = fddx[i] - f__[i];
        }
    }

    // Write header,in week 9 lab code
    #define SP << setw(30) << setprecision(10) <<    // save some repetition when writing
    cout << " i:" SP "xi:" SP "f(xi):" SP "fddx(xi):" SP "ei:" << endl;
    for (int i =0;i<N+1;++i){
        cout << i SP gridpoint[i] SP fx[i] SP f__[i] SP ei[i] << endl;
    } //This generates the headers when tabulating the outputs
}

long double e(int N){
```

```
    valarray<long double> gridpoint(N+1);
    valarray<long double> fx(N+1);
    valarray<long double> fddx(N+1);

    long double delta_x2 = pow(2.0 / static_cast <long double>(N),2.0);
    //We calculate the long double here which is required to calculate other doubles

    for (int i = 0; i < N+1; ++i) {
        gridpoint[i] = (2.0 * double(i) - (N)) / (N); //xi formula
        fx[i] = exp(-16.0 * (pow(gridpoint[i], 2.0))); //f(xi) formula
        fddx[i] = -32.0 * (-32.0 * fx[i] * pow(gridpoint[i], 2.0) + fx[i]);
    }//This series of for loops inputs the correct values into each respective valarray

    cout << setprecision(10);
    valarray <long double> f__(N+1);
    valarray <long double> ei(N+1);
    for (int i =0;i<N+1;++i){
        if (i == 0) {
            f__[0] = (fx[i + 2] - 2.0 * fx[i + 1] + fx[i]) /(delta_x2);
            ei[0] = fddx[0] - f__[0];
        }  else if (i == N) {
            f__[N] = (fx[N] - 2.0 * fx[N - 1] + fx[N - 2]) /(delta_x2);
            ei[i] = fddx[N] - f__[N];
        } else {
            f__[i] = (fx[i + 1] - 2.0 * fx[i] + fx[i - 1]) /(delta_x2);
            ei[i] = fddx[i] - f__[i];
        }
    } //The above is already explained but we need the previous values to calculate the error
    cout << setprecision(15);
    long double error = (static_cast <long double>(N*N)/(static_cast<long
double>(N)+1))*normal_calculation(1,ei);
    //This long double calculates the final value error from the value of n inputted
    cout << "The result of (N^2)<e>: " << error << " N = "<< N << endl;
    return error;
}


int main() {

    fanalytical(63);

    int i =15;
    while (i<2048){
        e(i);
        i = (i*2) + 1;
    }
    return 0;
}
```

Output:

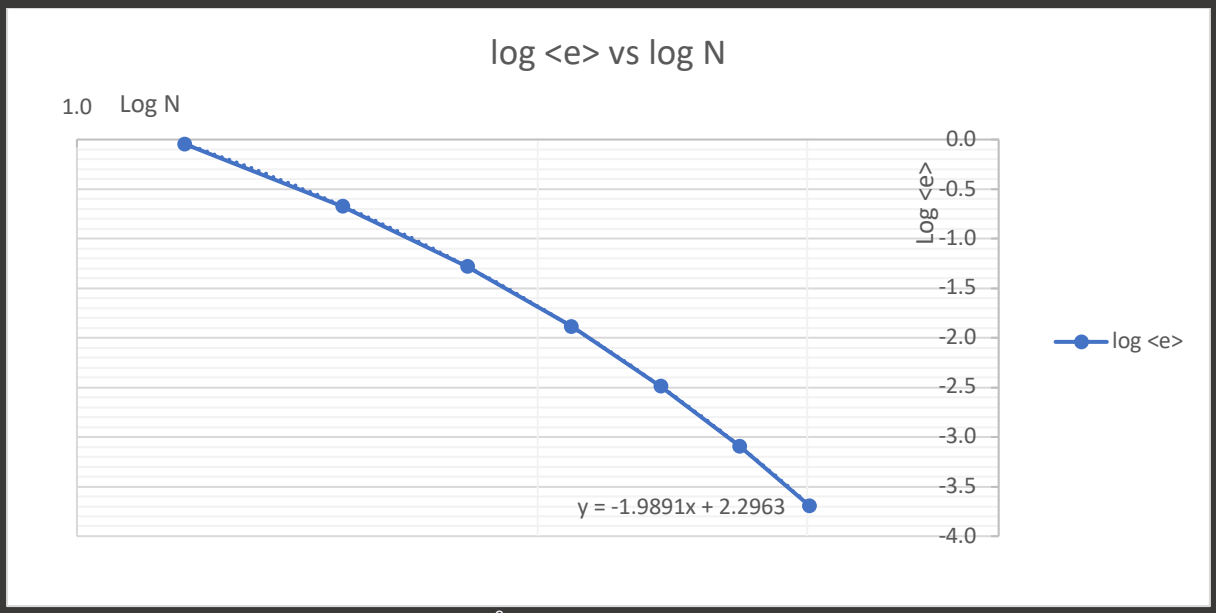
i:	xi:	f(xi):	fddx(xi):	ei:
0	-1.0000000000E+00	1.1253517470E-07	3.0325028050E-04	-1.9161538720E-04
1	-9.6825396830E-01	3.0582525730E-07	3.0325028050E-04	-1.9439510260E-05
2	-9.3650793650E-01	8.0473416630E-07	7.4090850990E-04	-4.3931269510E-05
3	-9.0476190480E-01	2.0503384740E-06	1.7485617630E-03	-9.5496127630E-05
4	-8.7301587300E-01	5.0581617430E-06	3.9853066720E-03	-1.9953322570E-04
5	-8.4126984130E-01	1.2082419050E-05	8.7701453610E-03	-4.0040497600E-04
6	-8.0952380950E-01	2.7945321210E-05	1.8629553820E-02	-7.7090936670E-04
7	-7.7777777780E-01	6.2583283910E-05	3.8187293640E-02	-1.4223185560E-03
8	-7.4603174600E-01	1.3570680330E-04	7.5510389870E-02	-2.5108658700E-03
9	-7.1428571430E-01	2.8493048890E-04	1.4397690970E-01	-4.2330422030E-03
10	-6.8253968250E-01	5.7925562040E-04	2.6459093940E-01	-6.7983992840E-03
11	-6.5079365080E-01	1.1402382870E-03	4.6839726220E-01	-1.0366788600E-02
12	-6.1904761900E-01	2.1732766480E-03	7.9822669370E-01	-1.4940336510E-02
13	-5.8730158730E-01	4.0107762770E-03	1.3084792130E+00	-2.0212618550E-02
14	-5.5555555560E-01	7.1669750380E-03	2.0611753540E+00	-2.5400518920E-02
15	-5.2380952380E-01	1.2400448030E-02	3.1163492080E+00	-2.9115670420E-02
16	-4.9206349210E-01	2.0774610570E-02	4.5153801800E+00	-2.9362383040E-02
17	-4.6031746030E-01	3.3699420810E-02	6.2573972800E+00	-2.3760098410E-02
18	-4.2857142860E-01	5.2930501930E-02	8.2715389850E+00	-1.0059826070E-02
19	-3.9682539680E-01	8.0497727160E-02	1.0391247510E+01	1.3058370220E-02
20	-3.6507936510E-01	1.1853736110E-01	1.2339938960E+01	4.5049998080E-02
21	-3.3333333330E-01	1.6901331540E-01	1.3738815290E+01	8.2718057280E-02
22	-3.0158730160E-01	2.3333539250E-01	1.4145669560E+01	1.1994722350E-01
23	-2.6984126980E-01	3.1191362430E-01	1.3127321060E+01	1.4830119450E-01
24	-2.3809523810E-01	4.0372170860E-01	1.0358281690E+01	1.5862304310E-01
25	-2.0634920630E-01	5.0596897830E-01	5.7267759290E+00	1.4343306840E-01
26	-1.7460317460E-01	6.1398775300E-01	-5.7970953320E-01	9.9533673940E-02
27	-1.4285714290E-01	7.2142229040E-01	-8.0392287710E+00	2.9969057300E-02
28	-1.1111111110E-01	8.2075480830E-01	-1.5832696630E+01	-5.5495213470E-02
29	-7.9365079370E-02	9.0413096780E-01	-2.2958712730E+01	-1.4184488810E-01
30	-4.7619047620E-02	9.6436909490E-01	-2.8408464310E+01	-2.1208606360E-01
31	-1.5873015870E-02	9.9597687240E-01	-3.1362817260E+01	-2.5148112580E-01
32	1.5873015870E-02	9.9597687240E-01	-3.1362817260E+01	-2.5148112580E-01
33	4.7619047620E-02	9.6436909490E-01	-2.8408464310E+01	-2.1208606360E-01
34	7.9365079370E-02	9.0413096780E-01	-2.2958712730E+01	-1.4184488810E-01
35	1.1111111110E-01	8.2075480830E-01	-1.5832696630E+01	-5.5495213470E-02
36	1.4285714290E-01	7.2142229040E-01	-8.0392287710E+00	2.9969057300E-02
37	1.7460317460E-01	6.1398775300E-01	-5.7970953320E-01	9.9533673940E-02
38	2.0634920630E-01	5.0596897830E-01	5.7267759290E+00	1.4343306840E-01

39	2.3809523810E-01	4.0372170860E-01	1.0358281690E+01	1.5862304310E-01
40	2.6984126980E-01	3.1191362430E-01	1.3127321060E+01	1.4830119450E-01
41	3.0158730160E-01	2.3333539250E-01	1.4145669560E+01	1.1994722350E-01
42	3.3333333330E-01	1.6901331540E-01	1.3738815290E+01	8.2718057280E-02
43	3.6507936510E-01	1.1853736110E-01	1.2339938960E+01	4.5049998080E-02
44	3.9682539680E-01	8.0497727160E-02	1.0391247510E+01	1.3058370220E-02
45	4.2857142860E-01	5.2930501930E-02	8.2715389850E+00	-1.0059826070E-02
46	4.6031746030E-01	3.3699420810E-02	6.2573972800E+00	-2.3760098410E-02
47	4.9206349210E-01	2.0774610570E-02	4.5153801800E+00	-2.9362383040E-02
48	5.2380952380E-01	1.2400448030E-02	3.1163492080E+00	-2.9115670420E-02
49	5.5555555560E-01	7.1669750380E-03	2.0611753540E+00	-2.5400518920E-02
50	5.8730158730E-01	4.0107762770E-03	1.3084792130E+00	-2.0212618550E-02
51	6.1904761900E-01	2.1732766480E-03	7.9822669370E-01	-1.4940336510E-02
52	6.5079365080E-01	1.1402382870E-03	4.6839726220E-01	-1.0366788600E-02
53	6.8253968250E-01	5.7925562040E-04	2.6459093940E-01	-6.7983992840E-03
54	7.1428571430E-01	2.8493048890E-04	1.4397690970E-01	-4.2330422030E-03
55	7.4603174600E-01	1.3570680330E-04	7.5510389870E-02	-2.5108658700E-03
56	7.7777777780E-01	6.2583283910E-05	3.8187293640E-02	-1.4223185560E-03
57	8.0952380950E-01	2.7945321210E-05	1.8629553820E-02	-7.7090936670E-04
58	8.4126984130E-01	1.2082419050E-05	8.7701453610E-03	-4.0040497600E-04
59	8.7301587300E-01	5.0581617430E-06	3.9853066720E-03	-1.9953322570E-04
60	9.0476190480E-01	2.0503384740E-06	1.7485617630E-03	-9.5496127630E-05
61	9.3650793650E-01	8.0473416630E-07	7.4090850990E-04	-4.3931269510E-05
62	9.6825396830E-01	3.0582525730E-07	3.0325028050E-04	-1.9439510260E-05
63	1.0000000000E+00	1.1253517470E-07	3.0325028050E-04	-1.9161538720E-04

(The above table was created by exporting the output into a .txt file then importing it into a delimited .txt import on excel which has been copied and pasted into word – This is to ensure easy viewing of data. Additionally, the cells have been formatted to be scientific to 9 decimal places/10 significant points.)

The result of (N^2)<e>: 202.712732660724 N = 15
The result of (N^2)<e>: 204.148768632211 N = 31
The result of (N^2)<e>: 208.55174670147 N = 63
The result of (N^2)<e>: 209.897396090865 N = 127
The result of (N^2)<e>: 210.919058979752 N = 255
The result of (N^2)<e>: 211.350172340754 N = 511
The result of (N^2)<e>: 211.600845591134 N = 1023
The result of (N^2)<e>: 211.702824050687 N = 2047

Process finished with exit code 0



(I had used the values generated from (N^2)<e> to create a graph showing a line graph of log<e> vs. log N and using the trendline function on excel I had outputted the function of the graph show by y. After creating this graph on excel I had copied and pasted it into Word so we can view the data easily.)

A) The error is what I expected as the value of error increases until it reaches N=32 and N=31 which is the max value of the error in that column. This is because it is the approximately the midpoint of N so the error would go back to decreasing after N=32 as N=63. We are visually able to verify this by viewing the table generated in output and created by excel.

B) I have altered the code from 2d to use a function instead of function object as it is easier for new users to understand and manipulate values. Additionally, I have outputted the values of N^2<e> which we can see the values approaching a constant near 211.8… as the value of N increase which proves the dependence is linear. Moreover, I have used the raw values from output and inputted it into Excel to create the above graph. We can see the gradient of y is -1.9891 ≈ -2 which verifies the linear dependency.

Question 4

Code:

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

long double dot(const valarray<long double> a,const valarray<long double> b){
    return (a*b).sum();
} //This function is recalled from question 2a

long double monteCarloEstimate(long double lowBound,long double upBound,long double iterations){
//Function to execute Monte Carlo integration on predefined function
    long double totalSum = 0.0;
    const int s = 31;
    //seed value
    mt19937_64 mtrand(s);
    //We are generating a seed with a real random value
    uniform_real_distribution<long double>unif(lowBound,upBound);
    //This generates random values unifromally between the lower and upper bound

    for (int i=0;i<=iterations;++i){
        //Select a random number within the limits of integration
        long double xi_randNum = unif(mtrand);
        //Add the f(x) value to the running sum using the correct function
        totalSum += pow(((4.0-(xi_randNum))*(xi_randNum)),0.5);

    }
    long double estimate = ((upBound-lowBound)*(totalSum))/static_cast<long double>(iterations);
    //This calculates the final value of the Integration estimate using this method
    return estimate;
}

int main() {

    long double n = 63.0;
    //This is the value of n defined in the question
    long double x_delta = static_cast<long double>(4/n);
    //We calculate delta x using the formula given, we use static_cast as this a fraction to avoid
accuracy loss

    valarray<long double> gridpoint(n+1);
    for (int i = 0; i <gridpoint.size(); ++i) {
        gridpoint[i] = (double(i)*4.0)/n;
    } //This calculates the gridpoints for 4a/4b and inputs them into a valarray

    valarray<long double> fi(n+1);
    for (int i =0; i < fi.size(); ++i){
        fi[i] = pow(((4.0-gridpoint[i])*gridpoint[i]),0.5);
    } //This calculates the f(i) points for 4a/4b and inputs them into a valarray

    valarray <long double>w_trapezium(n+1);
    for (int i = 0; i<w_trapezium.size(); ++i){
        long double multi = static_cast<long double>(x_delta/2.0);
        if (i == 0){
            w_trapezium[i] = (multi);
            //This inputs the values into the valarray for when i is 0
```

```
        }
        else if (i == n){
            w_trapezium[i] = (multi);
            //This inputs the values into the valarray for when i is 63
        }
        else {
            w_trapezium[i] = 2.0 * (multi);
            //This inputs the values into the valarray
        }
    } //This for loop inputs in the correct values for the weights creates the values inputted into the
valarray

    valarray<long double>w_simpson(n+1);
    for (int i = 0; i<w_simpson.size(); ++i){
        long double multi = static_cast<long double>(x_delta/48.0);
        if (i == 0) {
            w_simpson[i] = 17.0 *(multi);
            //This inputs the values into the valarray for when i is 0
        }
        else if (i == 1) {
            w_simpson[i] = 59.0 * (multi);
            //This inputs the values into the valarray for when i is 1
        }
        else if (i == 2) {
            w_simpson[i] = 43.0 * (multi);
            //This inputs the values into the valarray for when i is 2
        }
        else if (i == 3) {
            w_simpson[i] = 49.0 * (multi);
            //This inputs the values into the valarray for when i is 3
        }
        else if (i == (n-3)) {
            w_simpson[i] = 49.0 * (multi);
            //This inputs the values into the valarray for when i is 60
        }
        else if (i == (n-2)) {
            w_simpson[i] = 43.0 * (multi);
            //This inputs the values into the valarray for when i is 61
        }
        else if (i == (n-1)) {
            w_simpson[i] = 59.0 * (multi);
            //This inputs the values into the valarray for when i is 62
        }
        else if (i == n){
            w_simpson[i] = 17.0 *(multi);
            //This inputs the values into the valarray for when i is 63
        }
        else {
            w_simpson[i] = 48.0 * (multi);
            //This inputs the values into the valarray
        }
    }//This for loop inputs in the correct values for the weights creates the values inputted into the
valarray

    long double pi = 3.1415926535897932385;

    valarray <long double> theta(n+1);
    for (int i = 0; i<theta.size(); ++i){
        theta[i] = i*static_cast<long double>(pi/n);
    } //This calculates the theta values for 4c and inputs them into a valarray
```

```
    valarray<long double> gridpoint_CC(n+1);
    for (int i = 0; i <gridpoint_CC.size(); ++i) {
        gridpoint_CC[i] = ((4.0+(static_cast <double>(-4)*cos(theta[i])))/2.0);
    } //This calculates the gridpoints for 4c and inputs them into a valarray

    valarray<long double> fi_CC(n+1);
    for (int i =0; i < fi_CC.size(); ++i){
        fi_CC[i] = pow(((4.0-gridpoint_CC[i])*gridpoint_CC[i]),0.5);
    } //This calculates the f(i) for 4c and inputs them into a valarray


    valarray<long double>w_CC(n+1);
    for (int i = 0; i<w_CC.size(); ++i){
        if(i==0){
            w_CC[i] = static_cast<double>(2.0)/pow(n,2.0);
            //This inputs the values into the valarray for when i is 0
        }
        else if(i==n){
            w_CC[i] = static_cast<double>(2.0)/pow(n,2.0);
            //This inputs the values into the valarray for when i is 63
        }
        else {
            long double summation = 0;
            for (double k = 1.0; k <32;++k){
                summation += (static_cast<double>(2.0)*cos(2.0*k*theta[i]))/(4.0*pow(k,2.0)-1.0);
                //This generates the summation part of the CC equation
            }
            w_CC[i] = 2.0 * ((static_cast<double>(2.0)*(1.0 - summation))/n);
            //This inputs the values for the valarray for the correct values of i
        }
    } //This for loop inputs in the correct values for the weights creates the values inputted into the
valarray

    long double i_real = 2.0*pi;
    long double i_trapezium = dot(w_trapezium,fi);
    long double i_simpson = dot(w_simpson,fi);
    long double i_CC = dot(w_CC,fi_CC);
    //We use the function from 2a to calculate the porduct of two valarrays
    long double i_MC = monteCarloEstimate(0.0, 4.0, 10000.0);
    //This calls the function defined to calculate the MC estimate for 4d

    //The below outputs all the results for question 4
    cout << setprecision(10);
    cout << "i_real: " << i_real << endl << endl;
    cout << "i_trapezium: " << i_trapezium << endl;
    cout << "i_trapezium - i_real = " << i_trapezium - i_real << endl << endl;
    cout << "i_simpson: " << i_simpson << endl;
    cout << "i_simpson - i_real = " << i_simpson - i_real << endl << endl;
    cout << "i_CC: " << i_CC << endl;
    cout << "i_trapezium - i_real = " << i_CC - i_real << endl << endl;
    cout << "i_MC: " <<  i_MC << endl;
    cout << "i_MC - i_real = " << i_MC - i_real << endl;

    return 0;
}
```

Output:
i_real: 6.283185307

i_trapezium: 6.269894766
i_trapezium - i_real = -0.01329054152

i_simpson: 6.277420371
i_simpson - i_real = -0.005764936515

i_CC: 6.283171404
i_trapezium - i_real = -1.390312456e-05

i_MC: 6.279742749
i_MC - i_real = -0.003442558292
Program ended with exit code: 0

A) Using the dot function from 2a, we get the answer for 4a in the first two rows of the output past i_real which gives our I_{exact}. The error is what I expected as this method is not the most accurate method. This is shown by $I_{trapezium}$ = 6.269894766, with $I_{trapezium}$ - I_{exact} = -0.01329054152.

B) Using the dot function from 2a, we get the answer for 4b in the second two rows of the output past i_real. The error is what I expected as this method is not the most accurate method but it is more accurate compared to the trapezium method. This is shown by $I_{Simpson}$ = 6.277420371, with $I_{Simpson}$ - I_{exact} = -0.005764936515. This result verifies our expectation as the error is smaller than the error in trapezium.

C) Using the dot function from 2a, we get the answer for 4c in the third two rows of the output past i_real. The error is what I expected as this method is the most accurate method. This is shown by $I_{ClenshawCurtis}$ = 6.283171404, with $I_{ClenshawCurtis}$ - I_{exact} = -1.390312456e-05. This result verifies our expectation as the error is smaller than the error in Extended Simpson method.

D) Using the dot function from 2a, we get the answer for 4d in the last two rows of the output past i_real. The error is what I expected as this method is a more accurate method compared to the Extended Simpson method. However, the accuracy is highly dependent on the seed value implemented into the random number generator in the function 'monteCarloEstimate'. $I_{MonteCarlo}$ = 6.279742749, with $I_{MonteCarlo}$ - I_{exact} = -0.003442558292 .

Question 5a, 5b

Code:

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

valarray<long double> F(const long double &t, const valarray<long double> &u){
    /* du/dt = f(t,u) = {  u0
                        3(u1)^2 + (u1)
     since a =0,c=1(given) we get the above
    */
    valarray<long double> f(u.size());
    // u[0] = dq/dt, u[1] = dp/dt
    //Insert the relevant equations into the f valarray
    f[0] = u[1];
    f[1] = 3.0*pow(static_cast <long double>(u[0]),2.0) + u[0];
    return f;
} //This function will return the correct values of dp/dt and dq/dt by using a valarray<valarray<... in
main()

valarray<long double> RK2(const long double &t, const valarray<long double> &u, const long double
dt, valarray<long double> f(const long double&, const valarray<long double>&)){
    unsigned long long m = u.size();
    //This allows for us to get the value of the size of the valarray we are using to stay consistent
    valarray<long double> k1(m), k2(m);
    //This initialises valarrays of the correct size
    k1 = dt * f(t, u);
    //Since we call on function f and don't declare any specific valarray we are able to manipulate the
function to our use different valarrays
    k2 = dt * f(t + 0.5 * dt, u + 0.5 * k1);
    return u + k2;
} //The above formula is used the lecture notes - Page 5 of ODEs

int main() {

    long double t_initial = 0.;
    long double t_final = 10.0;
    int n = 100000;
    long double t_delta = (t_final-t_initial)/static_cast<long double>(n);
    valarray<long double> T(n+2);   //[0],T[1],...,T[n],T[n+1]
    //Given from question

    valarray<valarray<long double>> U(n+2);
    valarray<long double> E(n+2);
    valarray<long double> e(n+2);

    //set initial data such that U(0) = {q(0),p(0)} = {-0.5,0}
    U[0] = {-0.5, 0.};

    for(int j=0; j<=n; j++){
        T[j] = t_initial + static_cast<long double>(j)*t_delta;
        //given in question
        U[j+1] = RK2(T[j], U[j], t_delta, F);
        //We use the +1 as we already have been given the initial conditions, we do not want to overwrite
it - we also find the 0th value of each valarray without the need for 2 'for' loops
        E[j] = static_cast<long double>(0.5 * U[j][1]*U[j][1]) - static_cast<long
double>(U[j][0]*U[j][0]*U[j][0]) - static_cast<long double>(0.5 *U[j][0]*U[j][0]);
```

```
        //E(t) = 0.5(p^2) + (V(q)) = 0.5(p*p) - (q*q*q) -(0.5*q*q)
        e[j] = U[j][0] - static_cast<long double>(-0.5/(cosh(0.5*T[j])*cosh(0.5*T[j])));
        //error = q(t) - (-0.5sech^2(0.5*t) = q(t) - (-0.5/cosh^2(0.5*t))
        //We have to use cosh identity instead of sech as only cosh is cmath library
    }

    // Write header
#define SP << setw(26) << setprecision(10) <<    // save some repetition when writing
    cout << "t" SP "q(t)" SP "p(t)" SP "E(t)" SP "e(t)" << endl;

    int i = 0;
    while (i <= n){
        cout << T[i] SP U[i][0] SP U[i][1] SP E[i] SP e[i] << endl;
        i = i + 10000;
        //We use a while function and this equation to output the only relevant time intervals for
i=0,i=1...i=10
    }
    return 0;

}
```

Output:

t	q(t)	p(t)	E(t)	e(t)
0	-5.0000000000E-01	0.0000000000E+00	0.0000000000E+00	0.0000000000E+00
1	-3.9322386600E-01	1.8171549550E-01	-5.2467299170E-12	5.2732825060E-10
2	-2.0998717020E-01	1.5992500170E-01	-2.0739801250E-11	6.0979269530E-10
3	-9.0353319320E-02	8.1783148900E-02	-2.5493658060E-11	1.3709330730E-10
4	-3.5325412760E-02	3.4054671400E-02	-2.5990213230E-11	-3.3179652270E-10
5	-1.3296114370E-02	1.3118134550E-02	-2.6023080210E-11	-1.0259183440E-09
6	-4.9330212880E-03	4.9086209670E-03	-2.6024900570E-11	-2.7049412680E-09
7	-1.8204495770E-03	1.8171181970E-03	-2.6024994840E-11	-7.2171302700E-09
8	-6.7049481720E-04	6.7000626130E-04	-2.6024999610E-11	-1.9475678440E-08
9	-2.4681149980E-04	2.4664508290E-04	-2.6024999850E-11	-5.2800230780E-08
10	-9.0935003930E-05	9.0640063020E-05	-2.6024999860E-11	-1.4338845530E-07

(The above table was created by exporting the output into a .txt file then importing it into a delimited .txt import on excel which has been copied and pasted into word – This is to ensure easy viewing of data. Additionally, the cells have been formatted to be scientific to 10 decimal places.)

A) Analysing the values in the E(t) column we see that energy is being put out of the system as the values are negative. This would be the case as this system of Korteweg-De Vries (KDV) equation is used to model the behaviour of waves in shallow waters and since waves are passing through the system would not conserve any energy. Gravitational potential energy and kinetic energy would be put out by the system past a certain point as the energy would move with the wave. Over time, we notice the energy being outputted of the system decreasing as in any accurate model energy is lost to the surroundings which would be out of the system. Therefore, energy is not conserved efficiently as waves will lose energy in the form of sound and heat which is not factored into this system. Moreover, since the ODE system is non-linear and non-periodic there would be no sufficient energy conserved.

B) The error in the $q_{numerical} - q_{exact}$ is what I expected as time increases the error would decrease. This is what I would expect as the numerical answer would only get more accurate over tine as it is trying to model the exact value given the conditions in the system.

Question 5c

Code:

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

valarray<long double> F(const long double &t, const valarray<long double> &u){
    /* du/dt = f(t,u) = {  u0
                        3(u1)^2 + (u1)
     since a =0,c=1(given) we get the above
    */
    valarray<long double> f(u.size());
    // u[0] = dq/dt, u[1] = dp/dt
    //Insert the relevant equations into the f valarray
    f[0] = u[1];
    f[1] = 3.0*pow(static_cast <long double>(u[0]),2.0) + u[0];
    return f;
} //This function will return the correct values of dp/dt and dq/dt by using a valarray<valarray<... in
main()

//We use the pre-defined function Hermit-2 as it uses the trapezium rule without having to alter the
function F values
valarray<long double> H2(const long double &t1, const valarray<long double> &u1, const long double
dt, valarray<long double> f(const long double&, const valarray<long double>&)){
    long double t2 = t1 + dt; //This increases the time by delta_t starting from t_initial
    unsigned int imax = 10; //This is the maximum number of self-consistent iterations (10) that will be
used
    valarray<long double> u2 = u1; //This sets initial guess for the implicit ODE
    for(int i = 0; i<=imax; ++i) {
        //This 'for' loop carries out the self-consistent iterations
        u2 = u1 + 0.5 * dt * (f(t1, u1) + f(t2, u2));  //trapezium rule is now applied to the new values
    }
    return u2;
}

int main(){

    long double t_initial = 0.;
    long double t_final = 10.0;
    int n = 100000;
    long double t_delta = (t_final-t_initial)/static_cast<long double>(n);
    valarray<long double> T(n+2);   //[0],T[1],...,T[n],T[n+1]
    valarray<valarray<long double>> U(n+2);
    valarray<long double>E(n+2);
    valarray<long double> e(n+2);
    //Given from question

    // Write header
#define SP << setw(26) << setprecision(10) <<    // save some repetition when writing
    cout << "t" SP "q(t)" SP "p(t)" SP "E(t)" SP "e(t)" << endl;

    //set initial data such that U(0) = {q(0),p(0)} = {-0.5,0}
    U[0] = {-0.5, 0.};

    for(int j=0; j<=n; j++){
        T[j] = t_initial + static_cast<long double>(j)*t_delta;
        //given in question
```

```
        U[j+1] = H2(T[j], U[j], t_delta, F);
        //We use the +1 as we already have been given the initial conditions, we do not want to overwrite
it - we also find the 0th value of each valarray without the need for 2 'for' loops
        E[j] = static_cast<long double>(0.5 * U[j][1]*U[j][1]) - static_cast<long
double>(U[j][0]*U[j][0]*U[j][0]) - static_cast<long double>(0.5 *U[j][0]*U[j][0]);
        //E(t) = 0.5(p^2) + (V(q)) = 0.5(p*p) - (q*q*q) -(0.5*q*q)
        e[j] = U[j][0] - static_cast<long double>(-0.5/(cosh(0.5*T[j])*cosh(0.5*T[j])));
        //error = q(t) - (-0.5sech^2(0.5*t) = q(t) - (-0.5/cosh^2(0.5*t))
        //We have to use cosh identity instead of sech as only cosh is cmath library
    }

    int i = 0;
    while (i <= n){
        cout << T[i] SP U[i][0] SP U[i][1] SP E[i] SP e[i] << endl;
        i = i + 10000;
        //We use a while function and this equation to output the only relevant time intervals for
i=0,i=1...i=10
    }
    return 0;
}
```

Output:

t	q(t)	p(t)	E(t)	e(t)
0	-5.0000000000E-01	0.0000000000E+00	0.0000000000E+00	0.0000000000E+00
1	-3.9322386670E-01	1.8171549530E-01	1.0518493570E-11	-2.4935283190E-10
2	-2.0998717100E-01	1.5992500250E-01	4.1511998940E-11	-1.9208806810E-10
3	-9.0353319240E-02	8.1783149770E-02	5.1020583580E-11	2.2312316660E-10
4	-3.5325411610E-02	3.4054672630E-02	5.2013756370E-11	8.1606356960E-10
5	-1.3296111260E-02	1.3118137480E-02	5.2079493920E-11	2.0833425880E-09
6	-4.9330131700E-03	4.9086288420E-03	5.2083134830E-11	5.4129834200E-09
7	-1.8204279210E-03	1.8171396040E-03	5.2083323390E-11	1.4438908260E-08
8	-6.7043636860E-04	6.7006446620E-04	5.2083332920E-11	3.8972944540E-08
9	-2.4665303330E-04	2.4680331010E-04	5.2083333390E-11	1.0566625470E-07
10	-9.0504655690E-05	9.1070174610E-05	5.2083333420E-11	2.8695978550E-07

(The above table was created by exporting the output into a .txt file then importing it into a delimited .txt import on excel which has been copied and pasted into word – This is to ensure easy viewing of data. Additionally, the cells have been formatted to be scientific to 10 decimal places.)

C) We know the code is using the implicit system nested with a trapezium rule to calculate each respective value. As a result, we can state the same reasons as to why energy is conserved as it is the integration of the 1st Order ODE which would see how energy is moving through the waves at a given displacement point (q_t). Hence, we get positive values for E(t) within this case and how it increases over time as we analyse the energy conserved at different displacement values. Moreover, the reason as to why the error values are smaller than from 5a,b is because we are using an implicit method of 1st order, while in part a I was using an explicit method of 2n order. If the implicit method was 2nd order we would expect a smaller error but not in 1st order. However, for it being implicit the error is smaller with comparison to what I would have expected with an explicit method of 1st order.