Sahan Chowdhury

MTH6150 Numerical Computing with C & C++ Final exam Report

1)
Code:

```
#include <iostream>
// Input/output stream
#include <cmath>
// Enables the program to carry out mathematical functions
#include <iomanip>
//Enables to set the output number to a specific number of decimal places
using namespace std;

int main()
{
    long double X0 = 0;
    long double X = X0;
    long double N;
    long double TOL = 1e-12;
    int numiteration = 0;
// We are declaring a starting conditions for the code which is X0 = 0 and
the tolerance
//N will store the iterated value
    for (;;)
    {N = cos(X),numiteration++;
        if (abs(N - X) < TOL) break;
        X = cos(X);}
// Perform iterations until the condition absolute value (N - exact value)
is less than the prescribed tolerance


    long double finalerror = N - cos(N);
    //calculating the final error

    cout << "Final value: " << setprecision(16) << N << endl;
    // Prints out the final value of x obtained to 16 decimal places
    cout << "Number of iterations: " << setprecision(16)<< numiteration <<
endl;
    // Prints out the number of iterations performed to 16 decimal places
    cout << "Final error: " <<setprecision(16) << abs(finalerror) << endl;}
// Prints out the final error in the transcendental equation to 16 decimal
places which is calculated by the absolute value of the difference between
N-minus the exact value of cos(N)
```

- The code above calculates the value of cos(x) using iteration. It loops until the absolute difference between N and the exact value is less than prescribed tolerance of e=10^-12. As initial conditions stated the code starts with the value of x0=0 and is updated by each iteration by the cosine function. Iterations are performed until the for-loop condition ‘absolute value < tolerance’ is met, this is when the code will terminate and it will calculate the final value, number of iterations preformed and the final error, which will all be outputted to 16 decimal places, as required by the question.

Output:

Final value: 0.7390851332147725
Number of iterations: 70
Final error: 6.495914917081791e-13

Process finished with exit code 0

- From our final value we notice that it is identical to that of the exact value which was when X tends to infinity it equals 0.739085. and from our final value we notice the difference between the exact result and our final value, is very insignificant.

2)

a.

Code

```
#include <iostream>
// Include the input/output stream library
#include <valarray>
// Include the valarray library for numerical arrays
#include <cmath>
// Includes the library for mathematical functions
#include <iomanip>
//allows to set output to a desired number of decimal places


using namespace std;

// Multiplies the corresponding elements of u and v and returns their sum
long double inner_product(const valarray<long double> u, const
valarray<long double> v)
{
    return (u * v).sum();
}

int main()
{int N = 1000000;
    // Define the size of the valarray (N = 10^6)

    valarray<long double> u(0.1, N);
    // Initialize the valarray with a constant value of only 0.1 with 10^6
elements

    long double dot_product = inner_product(u, u);
    // Calculate the dot product of u with itself

    cout << "Dot product: " << setprecision(20)<< dot_product << endl;
    // Prints the dot product

    cout << "Difference dot product - 10^4: " << setprecision(20) <<
dot_product - 10000 << endl;}
    // Print the difference from 10^4
```

- The code above calculated the dot product of valaray u. We have created the valaray to consist of 1miilion element all which are 0.1. We multiplied the valaray by itself and then summing the elements. Finally, the code also calculated the difference from the expected value which is 10^4

Output

Dot product: 9999.9999999998775184
Difference dot product - 10^4: -1.2248158043348666979e-10

Our output shows that the calculated dot product of the valaray is close to 10^4, which is close to our expected value, hence the difference is very small. Our output is displayed to 20 decimal places as required.

b.

Code:

```
#include <iostream>
// Input/output stream
#include <valarray>
// Enables to create valarray to store
#include <iomanip>
// set output to desired number of decimal places

using namespace std;

long double dot_product(valarray<long double> u, valarray<long double> v)
    {long double sum = 0.0;
    // Set the sum to zero
    long double compensation = 0.0;
    // Set the compensation value to zero
    for (int i = 0; i < u.size(); ++i) {
        // Iterate through the elements of the valarrays
        long double product = u[i] * v[i] - compensation;
        // Calculate the product of the two vectors
        long double temp = sum + product;
        // Add the product to the current sum
        compensation = (temp - sum) - product;
        // Update the compensation value
        sum = temp;}
    // Update the sum with the compensated value

    return sum;}



int main() {
    const int N = 1000000;
    // Define the size of the valarray
    valarray<long double> u(0.1, N);
    //Create valarray with a constant value of 0.1 for 10^6 elements

    long double result = dot_product(u, u);
    // Calculate the dot product of the valarray with itself
    long double expectedValue = 1e4;
    //Expected value of dot product
    // Calculate the difference between the dot product and the expected
value

    cout << "Dot product: " << setprecision(20) << result << endl;
    // Print the dot product
    cout << "Difference: " << setprecision(20) << result - expectedValue <<
endl;}
    // Print the difference by calculating difference between result and
expected value
```

-The above code calculates the dot product of the valaray similar as the code prior, however in this case it uses the Kahan compensated summation to compute the sum. The code includes a for loop which iterates through each element in the valaray and calculates the product and subtracts the compensation value. The compensation value includes the loss off accuracy to make sure that all other iterations maintain accuracy.

Output

Dot product: 10000.00000000000111
Difference: 1.1102230246251565404e-12

-As we can see the dot product is just above 10000, which is our expected value. We can see the slight difference is very small too. If we compare to the previous code difference its much smaller. Hence, we can conclude the Kahan compensated summation is much more precise.

c.

Code

```
#include <iostream>
// Including the input/output stream library.
#include <valarray>
// To create valarray to store values
#include <cmath>
// Includes library for mathematical operations.
#include <iomanip>
// Change output precision to 20 decimal places

using namespace std;
// Using the std namespace for standard library objects and functions.

double innerProduct(const valarray<double>& u, const valarray<double>& v) {
    // Function to calculate the inner product of two valarrays.
    // Takes two valarrays by reference and returns their inner product.

    return (u * v).sum();}
    // Returns the sum of element-wise multiplication of valarray u and v.


class WeightedNorm
    {public:
    int m;
    // Member variable m of type int.

    WeightedNorm(int m) : m(m) {}
    // Constructor that initializes the member variable m with the provided
value.

    double operator()(const valarray<double> u) const {
        // function to calculate the weighted norm of a valarray.
        // Takes a valarray by reference and returns the weighted norm.

        double sum = 0.0;
```

```
        // Variable to store the sum of weighted elements.

        for (int i = 0; i < u.size(); i++) {
            // Loop through the elements of the valarray u.

            sum += pow(abs(u[i]), m);
            // Calculate the weighted sum by taking the absolute value of
each element raised to the power of m.
        }

        return pow(sum, 1.0 / m);}};
// Return the m-th root of the sum to calculate the weighted norm as
equation 2 requires



int main()
    {const int N = 1000000;
        // Constant variable N set to 1000000.
    valarray<double> u(0.1, N);
    // Create a valarray u with N elements, each initialized to 0.1.



    double dotProduct = innerProduct(u, u);
    // Calculate the dot product of valarray u with itself using the
innerProduct function.
    // Store the result in dotProduct.

    double expectedValue = 10000;
    // Define the expected value for comparison.

    WeightedNorm L(2);
    // Create an instance of the WeightedNorm class with m = 2.

    double norm = L(u);
    // Store the result in norm.

    cout << "norm L2: " << setprecision(20)<< norm << endl;
    // Output the calculated L2 norm of valarray u.

    cout << "Square of L2 norm: " << setprecision(20)<< pow(norm, 2.0) <<
endl;
    // Output the square of the L2 norm.

    cout << "Inner product using Part (a): " << setprecision(20)<<
dotProduct << endl;}
    // Output the dot product obtained using Part (a) of the code.
```

- The code above calculates the weighted norm of a valaray, and output includes L2 norm, square of L2 norm and inner product using part a. We calculate the weighted norm following with equation (2) from the question.

Output

norm L2: 100.00000000085928775
Square of L2 norm: 10000.000000171858119
Inner product using Part (a): 10000.0000001718563

Process finished with exit code 0

As we can see from the output, the square and inner product are almost identical with very slight differences. This suggest the calculations are accurate.

3)

a.

Code:

```
#include <iostream>
// Include the input/output stream library
#include <valarray>
// Include the valarray library
#include <cmath>
// Include the math library
#include <iomanip>
// Include the iomanip library for setprecision

using namespace std;
// Use the standard namespace

int N = 31;
// Define N (N+1=33)
long double A = -1.0;
// Set lower bound
long double B = 1.0;
// Set upper bound
long double GRID_SIZE = (B - A) / N;
// Calculate the grid size

int main() {
    // Main function
    valarray<long double> gridPoints(N + 1);
    // array to store grid points
    valarray<long double> functionValues(N + 1);
    // array to store function values
    valarray<long double> derivatives(N + 1);
    // array to store derivatives

    for (int i = 0; i <= N; i++) {
        // Loop over the number of points
        gridPoints[i] = A + i * GRID_SIZE;
        // Calculate and store the grid points
        functionValues[i] = sin(3 * gridPoints[i]); }
        // Calculate and store the function values


    derivatives[0] = (-3.0L * functionValues[0] + 4.0L * functionValues[1]
- functionValues[2]) / (2.0L * GRID_SIZE);
    // Calculate and store the derivative at the first point

    for (int i = 1; i < N; i++) {
        // Loop over the remaining points
        derivatives[i] = (functionValues[i + 1] - functionValues[i - 1]) /
(2.0L * GRID_SIZE); }
        // Calculate and store the derivatives


    derivatives[N] = (functionValues[N - 2] - 4.0L * functionValues[N - 1]
+ 3.0L * functionValues[N]) / (2.0L * GRID_SIZE);
```

```
    // Calculate and store the derivative at the last point

    valarray<long double> analyticalDerivatives(N + 1);
    // Create an array to store analytical derivatives

    for (int i = 0; i <= N; i++) {
        // Loop over the number of points
        analyticalDerivatives[i] = 3.0L * cos(3 * gridPoints[i]);}
    // Calculate and store the analytical derivatives


    valarray<long double> errors(N + 1);
    // Create an array to store the errors

    errors = derivatives - analyticalDerivatives;
    // Calculate the errors

    cout << setprecision(20);
    // Set precision to 20

    for (int i = 0; i <= N; i++) {
        // Loop over the number of points
        cout << "Grid-point[" << i << "] , Error[" << i << "] = " <<
errors[i] << endl; }}
        // Output the grid points and errors
```

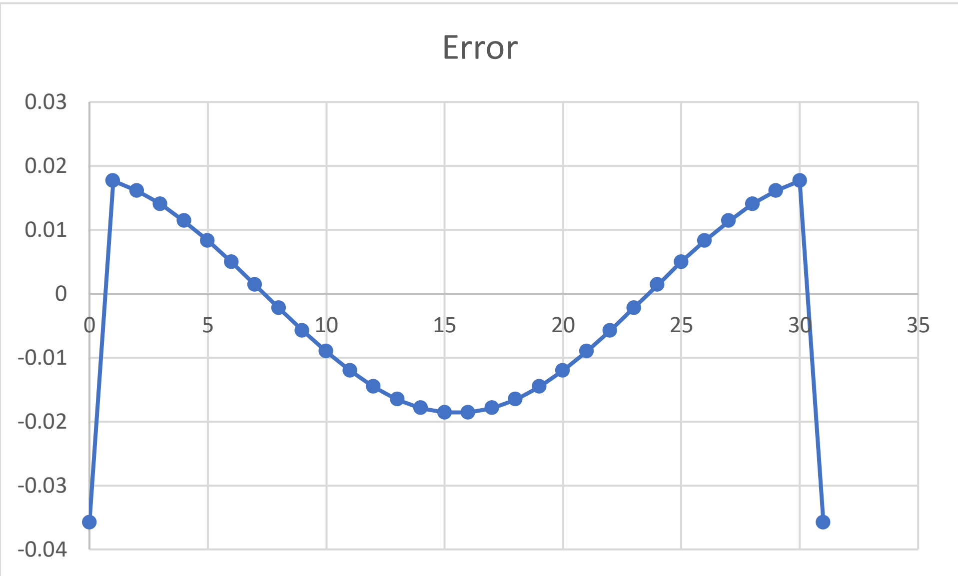
-The code above is a code that performs numerical differentiation using a finite difference method and compares these values to the array called analyticalDerivatives. It calculates the derivatives of the first grid point using a forward difference formula and the last grid point using a backward difference formula. The analyticalDerivatives are calculated by using the original function at each grid points. Then finally the errors are calculated by calculating the difference between the estimated derivates and analyticalDerivatives.

Output:

Grid-point[0] , Error[0] = -0.035839139032676214924
Grid-point[1] , Error[1] = 0.017655296580547652496
Grid-point[2] , Error[2] = 0.016142928006154466658
Grid-point[3] , Error[3] = 0.014027715017131292541
Grid-point[4] , Error[4] = 0.011388648508786181068
Grid-point[5] , Error[5] = 0.0083242822567279883827
Grid-point[6] , Error[6] = 0.0049490525068955903976
Grid-point[7] , Error[7] = 0.0013890044477458673988
Grid-point[8] , Error[8] = -0.0022229148439878998467
Grid-point[9] , Error[9] = -0.0057518212029136206453
Grid-point[10] , Error[10] = -0.0090659305135705192669
Grid-point[11] , Error[11] = -0.012041480080202449088
Grid-point[12] , Error[12] = -0.01456735044318957506
Grid-point[13] , Error[13] = -0.016549215044189491408
Grid-point[14] , Error[14] = -0.017913062774137067379
Grid-point[15] , Error[15] = -0.018607961857452171017
Grid-point[16] , Error[16] = -0.018607961857452615106
Grid-point[17] , Error[17] = -0.017913062774137067379

Grid-point[18] , Error[18] = -0.016549215044189491408
Grid-point[19] , Error[19] = -0.014567350443190019149
Grid-point[20] , Error[20] = -0.012041480080202227043
Grid-point[21] , Error[21] = -0.0090659305135705192669
Grid-point[22] , Error[22] = -0.0057518212029150639353
Grid-point[23] , Error[23] = -0.0022229148439876778021
Grid-point[24] , Error[24] = 0.0013890044477475882445
Grid-point[25] , Error[25] = 0.0049490525068944801745
Grid-point[26] , Error[26] = 0.0083242822567271002043
Grid-point[27] , Error[27] = 0.011388648508786181068
Grid-point[28] , Error[28] = 0.014027715017134845255
Grid-point[29] , Error[29] = 0.016142928006157131193
Grid-point[30] , Error[30] = 0.017655296580544543872
Grid-point[31] , Error[31] = -0.035839139032686428976

Process finished with exit code 0



-The output shows there is a variation in terms of the errors, some are over calculated, and some are under calculated, hence the interchange of positive and negative values. A positive error indicates the calculated derivate was higher than the actual (analyticalDerivatives) and a negative suggest the opposite. From the scatter plot we can see the graph is symmetrical.

b.

Code:

```
#include <iostream>
// Input/output stream
#include <valarray>
```

```cpp
// include the valarray library
#include <cmath>
// include the math library
#include <iomanip>
// Include the iomanip library for set precision

using namespace std;
// Using the standard namespace

// Function to calculate f(x) = sin(3x)
long double calculateFunction(long double x) {
    return sin(3 * x);
}

int main()
    {int nValues[] = {15, 31, 63, 127};
    // We want for N=15,31,63,127

    for (int i = 0; i < sizeof(nValues) / sizeof(nValues[0]); i++) {
        // Iterate over each n value
        int N = nValues[i];

        long double A = -1.0;
        //Lower bound
        long double B = 1.0;
        //Upper bound
        const long double gridSpacing = (B - A) / N;
        // Grid spacing
        valarray<long double> gridPoints(N + 1), functionValues(N + 1),
derivativeValues(N + 1);
        // Valarray to store the grid points
        for (int i = 0; i <= N; i++) {
            gridPoints[i] = A + i * gridSpacing;
            // Compute the current grid point
            functionValues[i] = calculateFunction(gridPoints[i]);}
            // Compute the function value at the current grid point


        // Compute f'(x) using 2nd order finite differencing
        derivativeValues[0] = (-3.0 * functionValues[0] + 4.0 *
functionValues[1] - functionValues[2]) / (2.0 * gridSpacing);

        for (int i = 1; i < N; i++) {
            derivativeValues[i] = (functionValues[i + 1] - functionValues[i
- 1]) / (2.0 * gridSpacing);}
        derivativeValues[N] = (3.0 * functionValues[N] - 4.0 *
functionValues[N - 1] + functionValues[N - 2]) / (2.0 * gridSpacing);

        // Compute the error at each grid point and the mean error
        long double meanError = 0.0;
        for (int i = 0; i <= N; i++)
        {long double error = derivativeValues[i] - 3.0 * cos(3.0 *
gridPoints[i]);
            meanError += abs(error);}

        meanError /= (N + 1);
        // Compute the mean error

        // Compute N^2 * mean error by squaring the mean error
        long double nSquaredMeanError = pow(N, 2.0) * meanError;
```

```
        // Output N , and n squared mean error to 20 decimal places
        cout << "N = " << N << ", N^2 * mean error = " <<
setprecision(20)<<  nSquaredMeanError << endl;}}
```

-The code above uses 2nd order finite differencing to carry out differentiation, to estimate the derivative of sin(3x). After calculating derivatives, the code computes errors at each grid point. In this code N was specifically assigned to 15,31,63,127.

Output:

N = 15, N^2 * mean error = 13.37265762437181138
N = 31, N^2 * mean error = 12.399050424673701443
N = 63, N^2 * mean error = 11.786666179205209204
N = 127, N^2 * mean error = 11.479774927671847706

Process finished with exit code 0

- From the output we can see an obvious trend that as number of grid points increases, the error decreases, as a result the more grid points the more accurate.

4)

a.

Code;

```
#include <iostream>
// Includes the input/output stream library
#include <valarray>
// Includes library for handling arrays
#include <cmath>
// Includes library for mathematical functions
#include <iomanip>
// output adjustment

using namespace std;
// Uses the standard namespace

// Define the function f(x)
long double f(long double x)
{return sin(1.0 / (x + 0.5));}


int main()
    {int N = 127;  // Number of equidistant points
    long double a = 0.0L, b = 10.0L;  // upper and lower bound for
integration
    long double dx = (b - a) / N;  // Grid spacing

    valarray<long double> x(N + 1), w(N + 1), y(N + 1);
    //Valarray to store Grid points, weights, function values

    x[0] = a;
    //the first point is equal to the lower bound
    x[N] = b;
    //the last point is equal to the upper bound
    w[0] = w[N] = dx / 2.0L;
    // Set the weights for the endpoints

    // Fill in the remaining grid points and weights
    for (int i = 1; i < N; i++) {
        x[i] = a + i * dx;
        w[i] = dx;}


    // Compute the function values at each grid point
    for (int i = 0; i < N + 1; i++) {
        y[i] = f(x[i]);}

    long double Itrapezium = (w * y).sum();
    // Perform numerical integration using the composite trapezium rule

    long double Iexact = 2.74324739415100920L;
    // Exact value

    // Output the results
    cout << "Numerical result (composite trapezium rule): " <<
setprecision(20) << Itrapezium << endl;
```

```
    cout << "Difference: " << setprecision(20) << Itrapezium - Iexact <<
endl;}
```

- This code above, perform numerical integration using the composite trapezium rule. It calculates an estimate of the given function in the question with the required number of equidistant points. The code then takes each individual calculated integral and sums it all together, to compute the numerical approximation of the integral

Output:

Numerical result (composite trapezium rule): 2.7423926434484635628
Difference: -0.00085475070254581453355

Process finished with exit code 0

- From the output we can see that the numerical result is very similar of that to the exact value. The difference is very small, since we are using 20 decimal places it highlights the slight difference in values. The method is only a good approximation, and not an exact value.

b.

Code:

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

using namespace std;

long double f(long double x) {
    return sin(1.0 / (x + 0.5));
}

long double f_prime(long double x) {
    return -cos(1.0 / (x + 0.5)) / pow(x + 0.5, 2);
}

int main() {
    int N = 127;
    // Number of intervals
    long double a = 0.0L, b = 10.0L;
    // Lower and upper bounds for integration
    long double dx = (b - a) / N;
    // Grid spacing

    valarray<long double> x(N + 1), w(N + 1), y(N + 1);
    // Arrays to store grid points, weights, and function values

    x[0] = a;
    // Set the first element of 'x' as 'a'
```

```
    x[N] = b;
    // Set the last element of 'x' as 'b'
    w[0] = w[N] = dx / 2.0L;
    // Set the first and last elements of 'w' as half of 'dx'

    for (int i = 1; i < N; i++) {
        x[i] = a + i * dx;
        // Calculate the i-th element of 'x'
        w[i] = dx;
        // Set the i-th element of 'w' as 'dx'
    }

    for (int i = 0; i < N + 1; i++) {
        y[i] = f(x[i]);
        // Evaluate the function at each grid point and store the values in
'y'
    }

    long double Ihermite = (w * y).sum() + (pow(dx, 2) / 12.0L) *
(f_prime(a) - f_prime(b));
    // Calculate the integral using the composite Hermite rule and store it
in 'Ihermite'

    long double Iexact = 2.7432473941510092OL;   // Exact value of the
integral

    cout << "Numerical result (composite Hermite rule): " <<
setprecision(20) << Ihermite << endl;
    // Output the numerical result of the integral to 20dp
    cout << "Difference: " << setprecision(20) << abs(Ihermite - Iexact) <<
endl;}
    // Output the difference between the numerical and exact values of the
integral to 20dp
```

-The code above performs numerical integration using the composite Hermite Rule, and compares the result to the exact value, similarly to the previous code.

Output:

Numerical result (composite Hermite rule): 2.7432573470552856776
Difference: 9.9529042763002451011e-06

Process finished with exit code 0

- From the output of this code we see its close to the actual value. When we compare it to the composite trapezium method, we see its much more accurate and closer to the exact value, as a result we can say, the Hermite rule is a better approximation.

c.

Code:

```
#include <iostream>
// Include the input/output stream library
#include <valarray>
// Include the library for arrays
#include <cmath>
// Include the library for mathematical functions
#include <iomanip>
// Include the iomanip library for setting precision

using namespace std;
// Use the standard namespace

long double f(long double x)
    {return sin(1.0 / (x + 0.5));}

// Define a function named 'f' that returns a long double

long double innerProduct(valarray<long double> u, valarray<long double> v)
    {return (u * v).sum();}

// Define a function named 'innerProduct' that takes two valarray<long
double> parameters 'u' and 'v' and returns a long double

int main()
    {int N = 127; // Ensure N is odd
    // Declare and initialize an integer variable 'N' with the value 127

    long double a = 0.0, b = 10.0;
    // Declare and initialize two long double variables 'a' and 'b' with
the values 0.0 and 10.0 respectively

    valarray<long double> theta(N + 1), x(N + 1), w(N + 1), y(N + 1);
    // Declare four valarray<long double> variables 'theta', 'x', 'w', and
'y' with size N+1

    for (int i = 0; i <= N; i++)
        // Loop from 0 to N (inclusive)
        {theta[i] = i * 3.14159265358979323846 / N;
        // Calculate the i-th element of 'theta'
        x[i] = ((a + b) - (b - a) * cos(theta[i])) / 2.0;
        // Calculate the i-th element of 'x' using the Clenshaw-Curtis
formula

        if (i == 0 || i == N)
            w[i] = (b - a) / 2.0 * (1.0 / (N * N));
        else
            {long double sum = 0.0;
            // Declare and initialize a long double variable 'sum' with the
value 0.0

            for (int k = 1; k <= (N - 1) / 2; k++)
                // Loop from 1 to (N - 1) / 2
                sum += (2 * cos(2 * k * theta[i])) / ((4 * k * k) - 1);
            // Accumulate the sum according to the Clenshaw-Curtis weights
formula
```

```
            w[i] = (b - a) / 2.0 * ((2.0 / N) * (1.0 - sum));}}
            // Calculate the i-th element of 'w' using the Clenshaw-Curtis
weights formula

    for (int i = 0; i <= N; i++)
        // Loop from 0 to N (inclusive)
    {y[i] = f(x[i]);}
        // Evaluate the function 'f' at each grid point and store the
values in 'y'


    long double I_clenshawCurtis = innerProduct(w, y);
    // Calculate the inner product of 'w' and 'y' and store the result in
'I_clenshawCurtis'

    long double I_exact = 2.74324739415100920;
    // Declare and initialize 'I_exact' with the exact value of the
integral

    cout << "Numerical result (Clenshaw-Curtis quadrature rule): " <<
setprecision(20) << I_clenshawCurtis << endl;
    // Output the numerical result of the integral using the Clenshaw-
Curtis quadrature rule with 20 decimal point precision

    cout << "Difference: " << setprecision(20) << I_clenshawCurtis- I_exact
<< endl;}
    // Output the difference between the exact value and the numerical
result of the integral with 20 decimal point precision
```

-The code above performs numerical integration using Clenshaw-curtis quadrature rule and approximates integral of a function over an interval by dividing the interval into a set of points and assigning weights to each point. The code calculates the weights and evaluates the function at the points using the Clenshaw-Curtis formula. It compares the numerical result of integration to the exact value, and outputs the result to 20 decimal places.

Output;

Numerical result (Clenshaw-Curtis quadrature rule): 2.7432473941510067128
Difference: -2.664535259100375697e-15

Process finished with exit code 0

-From the output we see that since the difference is almost near 0, it gives an indication that the numerical result is mostly identical to the exact result. This proves that the Clenshaw-Curtis quadrature provides accurate approximation of the integral.

d.

Code:

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

using namespace std;
//declaring constants
long double a = 0.0;
long double b = 10.0;
long double Iexact = 2.743247394151009 20;

double f(double x) {
    return sin(1.0 / (x + 0.5));
}

//Monte-Carlo integration
double monte_carlo(int N) {
    unsigned seed = chrono::system_clock::now().time_since_epoch().count();
    default_random_engine generator(seed);
    uniform_real_distribution<double> distribution(a, b);
    double sum = 0.0;
    for (int i = 0; i < N; i++) {
        double x = distribution(generator);
        sum += f(x);
    }
    double average = sum / (double)N;
    return (b - a) * average;}
    //Calculating avg value of sum

int main() {
    //Different values of N
    int N1 = 1000;
    int N2 = 10000;
    int N3 = 100000;


    //computing difference and numerical results for Monte Carlo
integration
    long double ImonteCarlo1 = monte_carlo(N1);
    long double diff1000 = ImonteCarlo1 - Iexact;
    long double ImonteCarlo2 = monte_carlo(N2);
    long double diff10000 = ImonteCarlo2 - Iexact;
    long double ImonteCarlo3 = monte_carlo(N3);
    long double diff1000000 = ImonteCarlo3 - Iexact;

    //Output
    cout << "Numerical result using Mean Value Monte Carlo method with N =
" << setprecision(20) << N1 << ": " << ImonteCarlo1 << endl;
    cout << "Difference from exact result: " << setprecision(20)<< diff1000
<< endl;
    cout << "Numerical result using Mean Value Monte Carlo method with N =
" << setprecision(20)<< N2 << ": " << ImonteCarlo2 << endl;
    cout << "Difference from exact result: " << setprecision(20)<<
diff10000 << endl;
    cout << "Numerical result using Mean Value Monte Carlo method with N =
```

```
" << setprecision(20)<< N3 << ": " << ImonteCarlo3 << endl;
    cout << "Difference from exact result: " << setprecision(20)<<
diff1000000 << endl;}
```

-The code above uses Monte-Carlo method of integration, with different values of 'N',(1000,10000,100000). The Monte-Carlo method generated random values and evaluates with the function, and from there calculates the average. It then displays the numerical result and the difference to 20 dp.

Output:

Numerical result using Mean Value Monte Carlo method with N = 1000:
2.6858743231961854647
Difference from exact result: -0.057373070954823912615
Numerical result using Mean Value Monte Carlo method with N = 10000:
2.7782473098784024046
Difference from exact result: 0.034999915727393027254
Numerical result using Mean Value Monte Carlo method with N = 100000:
2.7543853734233083586
Difference from exact result: 0.011137979272298981215

Process finished with exit code 0

5)

a.

b.

c.

d.