Numerical Computing with C and C++

Zubaid Amadxarif

May 2023

The following solutions were written and tested on the CLion compiler used on a Windows 11 operating system. The outputs reflect the output of the code and can be seen when running the code on any such compiler.

Question 1 - *Self-consistent iteration*

The following is the solution of question 1:

```
int main() {
   // define x0, x1, tolerance and max_i as specified in problem sheet
   double x0 = 0;
   double x1;
   double tolerance = 1e-12;
   int max_i = 1e6;
   int i;

   // for loop to iteratively update x0 and update x1 to cos(x0)
   for (i = 0; i < max_i; ++i) {
       x1 = cos(x0);

       // check if the difference between x1 and x0 is less than the tolerance and
exit if true
       if (abs(x1 - x0) < tolerance) {
           break;
       }

       x0 = x1;
   }
   // calculate final error
   double final_error = x1 - cos(x1);

   // set output precision to 16 digits
   cout << setprecision(16);

   // print final value, number of iterations and final error
   cout << "Final value x_n+1: " << x1 << endl;
   cout << "Number of iterations: " << i << endl;
   cout << "Final error: " << final_error << endl;

   return 0;   // exit program
}
```

The following is the output generated:

```
Final value x_n+1: 0.7390851332147726
Number of iterations: 69
Final error: -6.493694471032541e-13

Process finished with exit code 0
```

The provided code applies the fixed-point iteration method to approximate the fixed point of the cosine function. Starting with an initial guess of `x0 = 0`, the code iteratively updates `x1` by calculating the cosine of `x0`. The iteration continues until the difference between `x1` and `x0` falls below a specified tolerance. After the iterations, the code determines the final error by computing the difference between `x1` and `cos(x1)`. The output includes the final value of `x_n+1`, which is approximately `0.7390851332147726`, indicating the estimated fixed point. The code took 69 iterations to converge to this result. Additionally, the final error is approximately `-6.493694471032541e-13`, indicating a high level of accuracy in the approximation. The code successfully applies the fixed-point iteration method to find the fixed point of the cosine function. It provides an accurate estimation of the fixed point and demonstrates the effectiveness of the iteration method in achieving convergence.

Question 2 - *Inner Products*

part a) The following is the code for the solution:

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

// create a function that calculate the inner product of two vectors of type
valarray<long double>
long double inner_product(valarray<long double> u, valarray<long double> v)
{
   return (u*v).sum();
}

int main() {
   // define the vector u, inner product and difference in the question
   valarray<long double> u(0.1, 1e6);
   long double final_value = inner_product(u,u);
   long double difference = final_value - 10000;

   // print out the inner product and different
   cout << setprecision(20) << "The inner product of u and u is: " << final_value
<< endl;
   cout << setprecision(20) << "The difference of u*u - 10^4 is: " << difference <<
endl;

   return 0;
}
```

The following is the output:

```
The inner product of u and u is: 9999.9999999998775184
The difference of u*u - 10^4 is: -1.2248158043348666979e-10

Process finished with exit code 0
```

The code above defines the function `inner_product` which uses the `(u*v).sum()`, and takes two valarrays as input. The valarray `u` is then defined as required in the questions and the inner product is calculated and assigned to a `long double`. The difference is also calculated as required and then these are outputted to the system - the inner product is then `9999.9999999998775184`, and the difference is `-1.2248158043348666979e-10` which is a very small difference.

part b) The following is the code for the solution of the Kahan Sum.

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

// KahanSumInnerProduct function taken from KahanSum function in week 9 content and
modified for two vectors
long double KahanSumInnerProduct(valarray<long double> u, valarray<long double> v){
   long double sum = 0.; // initialise sum to zero
   long double c = 0.; // initialise the Kahan sum error term to zero
   long double y;
   long double t;
   for(int i = 0; i < u.size(); ++i){
       y = u[i] * v[i] - c; // adjust the current value by subtracting the Kahan
sum error term
       t = sum + y;
       c = (t - sum) - y;
       sum = t;
   }
   return sum; // return the final Kahan sum
}



int main() {
   // define the vector u, inner product using Kahan sum, and the difference in the
question
   valarray<long double> u(0.1, 1e6);
   long double final_value = KahanSumInnerProduct(u,u);
   long double difference = final_value - 10000;

   // print it all out
   cout << setprecision(20) << "The inner product of u and u is: " << final_value
<< endl;
   cout << setprecision(20) << "The difference of u*u - 10^4 is: " << difference <<
endl;

   return 0;

}
```

The following is the output:

```
The inner product of u and u is: 10000.00000000000111
```

```
The difference of u*u - 10^4 is: 1.1102230246251565404e-12

Process finished with exit code 0
```

The code above uses the Kahan Sum provided in Week 9 from the Module Page and it has been modified to work with two vectors instead of one which looks at both `u[i]` and `v[i]`. The main function contains the same as the code in part a, where we define the final_value, difference and the `valarray u`. The output is provided and the inner product is close to 10000, with the difference is `1.11022e-12` which is very small.

part c) The following is the code:

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

using namespace std;

// Define the Norm class
class Norm {
public:
   int m;

   // Constructor to initialize m
   Norm(int m) : m(m) {}

   // Overload the operator() to calculate the weighted norm of a valarray
   long double operator()(const valarray<double>& u) const {
       // initialise sum to store the weighted sum and iterate through valarray
elements
       double sum = 0.0;
       for (size_t i = 0; i < u.size(); ++i) {
           sum += pow(u[i], m);
       }
       // Return the m-th root of the sum as the norm
       return pow(sum, 1.0 / m);
   }
};

int main() {
   Norm l2norm(2); // Create an instance of Norm with m = 2 (L2 norm)

   valarray<double> u(0.1, 1e6); // Initialize a valarray u with size 1000000 and
each element set to 0.1
```

```
    // Calculate the L2 norm and L2 norm squared of u using the function object
    double norm_l2 = l2norm(u);
    double norm_l2_squared = pow(norm_l2, 2);

    // Output the L2 norm squared
    cout << setprecision(20) << "L2 norm squared: " << norm_l2_squared << endl;

    return 0;
}
```

The following is the output of the code:

```
L2 norm squared: 10000.000000171858119

Process finished with exit code 0
```

The Norm class is defined with an `int` member variable `m` and an overloaded `operator()` function. The constructor initialises the member variable m with the provided argument. The `operator()` function takes a `valarray<double> u` as input and calculates the weighted norm of u using the formula `sum(pow(u[i], m))^(1/m)`. It returns the calculated weighted norm. An instance of the Norm class is created with m set to 2, representing the L2 norm and the valarray vector u is initialised of the same size as before. The L2 Norm is firstly calculated followed by the L2 Norm squared which is outputted as very close to 10000, which shows that all three methods of question 2 are valid methods of calculating the inner product of two vectors.

Question 3 - *Finite Differences*

part a) The following is the code for the solution:

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

using namespace std;

long double f(long double x) {
   return sin(3 * x);
}

long double df_analytical(long double x) {
   return 3 * cos(3 * x);
}

int main() {
   // define all the constants
   const int n = 31;
   const long double a = -1;
   const long double b = 1;
   const long double X = (b - a) / n;

   // define all valarrays from question
   valarray<long double> x(n + 1);
   valarray<long double> fx(n + 1);
   valarray<long double> df_numerical(n + 1);
   valarray<long double> e(n + 1);

   // calculate x[i] and fx[i]
   for (int i = 0; i <= n; i++) {
       x[i] = a + i * X;
       fx[i] = f(x[i]);
   }

   // numerical derivative for i = 0
   df_numerical[0] = (-3 * fx[0] + 4 * fx[1] - fx[2]) / (2 * X);

   // numerical derivative for i = 1 to (N-1)
   for (int i = 1; i < n; i++) {
       df_numerical[i] = (fx[i + 1] - fx[i - 1]) / (2 * X);
   }

   // numerical derivative for i = N
   df_numerical[n] = (fx[n - 2] - 4 * fx[n - 1] + 3 * fx[n]) / (2 * X);
```

```
   // difference between numerical and analytical derivatives
   for (int i = 0; i <= n; i++) {
       e[i] = df_numerical[i] - df_analytical(x[i]);
   }

   //  the errors e[i] are printed
   cout << "The following are the error values between numerical and analytical
derivatives:\n";
   for (int i = 0; i <= n; i++) {
       cout << setprecision(20) << "e[" << i << "] = " << e[i] << endl;
   }

   return 0;
}
```

The following is the output of the code:

```
The following are the error values between numerical and analytical
derivatives:
e[0] = -0.035839139032681525347
e[1] = 0.01765529658054848061
e[2] = 0.016142928006157757862
e[3] = 0.014027715017131953037
e[4] = 0.011388648508785184903
e[5] = 0.008324282256727464387 8
e[6] = 0.004949052506895372472 9
e[7] = 0.001389004447746682529 1
e[8] = -0.002222914843988063696 7
e[9] = -0.005751821202914310360 6
e[10] = -0.009065930513570635927
e[11] = -0.012041480080202145403
e[12] = -0.014567350443189778673
e[13] = -0.016549215044189315334
e[14] = -0.017913062774136786137
e[15] = -0.018607961857452324974
e[16] = -0.018607961857452326275
e[17] = -0.017913062774136786354
e[18] = -0.016549215044189313165
e[19] = -0.014567350443189778456
e[20] = -0.012041480080202147246
e[21] = -0.009065930513570636577 5
e[22] = -0.005751821202914309384 8
e[23] = -0.002222914843988062856 5
e[24] = 0.001389004447746682298 7
```

```
e[25] = 0.0049490525068953710635
e[26] = 0.008324282256727463954 2
e[27] = 0.011388648508785186204
e[28] = 0.014027715017131953904
e[29] = 0.016142928006157757862
e[30] = 0.017655296580548479092
e[31] = -0.035839139032681530768

Process finished with exit code 0
```

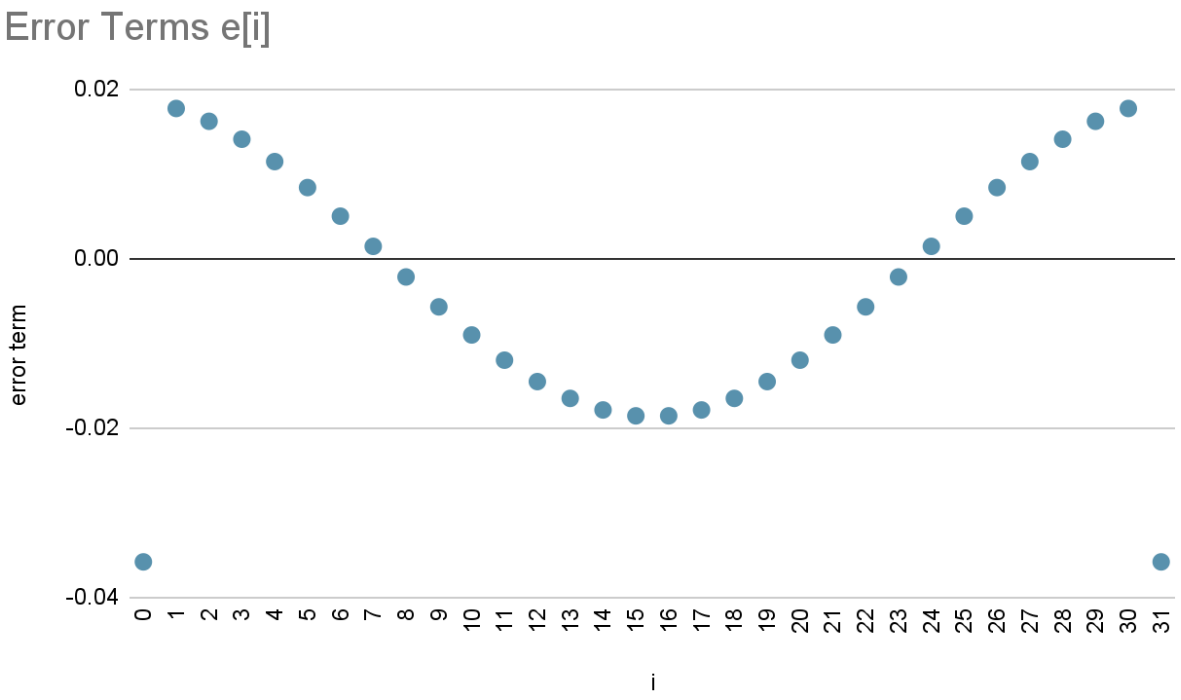

Figure 1 - Plot of error terms (numerical differential - analytical differential)

The code above implements and calculates the first derivative of the given function `f(x) = sin(3x)` using the finite differencing method.

The code begins by defining both `f(x) = sin(3x)` and the differential `f'(x) = 3cos(3x)` to be used for the error terms. The constants `n, a, b` and `delta_X` are defined as constants to determine the number of grid points, the interval of the grid and the step size respectively.

The variables `x[i], f(x[i]),` and `f'(x[i])` and `e[i]` are defined as valarrays and `x[i]` and `f(x[i])` are calculated as a loop iterates through all the grid points and calculator the points using the `f(x)` that was already defined.

The code then moves on and calculates the numerical derivatives using the given second-order differencing equations in the problem sheet for the boundary points `i = 0, i = N` and then

all the points in between (`i = 1, 2, .. , N - 1`). Another loop follows this and the error values are calculated as the difference between the `numerical` and `analytical` derivatives. These errors are outputted as can be seen from the output above.
We can see from the plot of these error points that there seems to be a symmetrical pattern with the smallest error values for `i = 7` and `i = 24`.

part b) The following is the code for the solution:

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

using namespace std;

class Norm {
public:
    int m;
    Norm(int m) : m(m) {}
    long double operator()(const valarray<long double>& u) const {
        double sum = 0.0;
        for (size_t i = 0; i < u.size(); ++i) {
            sum += pow(abs(u[i]), m);
        }
        return pow(sum, 1.0 / m);
    }
};

// define f(x) as sin(3x)
long double f(long double x) {
    return sin(3 * x);
}

// use for the error term calculation
long double df_analytical(long double x) {
    return 3 * cos(3 * x);
}


int main() {
    // instance of class created to make a l1_norm
    Norm l1_norm(1);

    // set table width
    cout << setw(10) << "N" << setw(30)  << "N^2(e)" << endl;

    // for loop running for n values 16 to 128 from question
```

```
    for (int j = 16; j <= 128; j *= 2){

        // a, b, n and delta x are defined
        double a = -1;
        double b = 1;
        int n = j  - 1;
        double X = (b - a) / n;

        // follows the same as 3a to obtain the error terms
        valarray<long double> x(n + 1);
        valarray<long double> fx(n + 1);
        valarray<long double> df_numerical(n + 1);
        valarray<long double> e(n + 1);

        for (int i = 0; i <= n; i++) {
            x[i] = a + i * X;
            fx[i] = f(x[i]);
        }

        df_numerical[0] = (-3 * fx[0] + 4 * fx[1] - fx[2]) / (2 * X);

        for (int i = 1; i < n; i++) {
            df_numerical[i] = (fx[i + 1] - fx[i - 1]) / (2 * X);
        }

        df_numerical[n] = (fx[n - 2] - 4 * fx[n - 1] + 3 * fx[n]) / (2 * X);

        for (int i = 0; i <= n; i++) {
            e[i] = df_numerical[i] - df_analytical(x[i]);
        }

        // (e) is calculated using the formula is the problem sheet and then
outputted in a table
        long double E = 1.0 / (n + 1) * l1_norm(e);
        cout << setprecision(20) << setw(10) << n << setw(30)  << n * n * E <<
endl;
    }
    return 0;
}
```

The following is the output of the code:

```
         N                         N^2(e)
        15          13.372657624371884003
        31          12.399050424673247106
        63          11.786666179203728316
       127          11.479774927671680358
```

```
Process finished with exit code 0
```

The given code snippet demonstrates 2nd-order convergence by calculating the `mean error (e)` and `N^2(e)` for a numerical differentiation problem using the central difference method and outputs the results in a tabular format showing that the mean error decreases proportionally as N increases.

The code begins by bringing the `Norm` class from question 2C, defining `f(x)` and then the derivative of `f(x)` which is used to calculate the error terms.

The main function begins by creating the `L1` norm instance of the `Norm` class followed by a loop from `16 to 128` for `N` as required in the problem sheet. The code then follows the same as part a and calculates `x, f(x)` and numerical derivative values for each `N`. The valarray `e[i]` is created for each `N`. The mean error `(E)` is then calculated using the formula provided in the problem which is the `L1-norm` divided by `(n + 1)` . The values are then outputted showing the 2nd-order convergence.

We can see from the table that the `mean error E` decreases proportionally as `N` increases confirming the 2nd-order convergence.

Question 4 - *Numerical Integration*

part a) The following is the code for the solution:

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

using namespace std;

long double f(long double x) {
    return sin(1 / (x + 0.5));
}

// inner_product function from question 2a
long double inner_product(valarray<long double> u, valarray<long double> v) {
    return (u * v).sum();
}

long double composite_trapezium_rule(const valarray<long double> &function_values,
const valarray<long double> &weights) {
    // Use the inner_product function instead of directly calculating the dot
product
    return inner_product(function_values, weights);
}

int main() {
    int N = 127; // N + 1 = 128
    long double a = 0;
    long double b = 10;
    long double delta_x = (b - a) / N;

    valarray<long double> gridpoints(N + 1);
    valarray<long double> function_values(N + 1);
    valarray<long double> weights(N + 1);

    // Populate the gridpoints, function_values, and weights arrays
    for (int i = 0; i <= N; ++i) {
        gridpoints[i] = a + i * delta_x;
        function_values[i] = f(gridpoints[i]);
        if (i == 0 || i == N) {
            weights[i] = delta_x * 0.5;
        } else {
            weights[i] = delta_x;
        }
    }

    // Calculate the approximate integral using the composite trapezium rule
```

```
    long double approx_integral = composite_trapezium_rule(function_values,
weights);

    // Output the results
    cout << setprecision(20) << "Approximated integral (trapezium): " <<
approx_integral << endl;
    cout << setprecision(20) << "Difference: " << approx_integral -
2.74324739415100920 << endl;

    return 0;
}
```

The following is the output of the code:

```
Approximated integral (trapezium): 2.7423926434484638586
Difference: -0.00085475070254551876 32

Process finished with exit code 0
```

The code above performs numerical integration using the composite trapezium rule provided in the problem sheet. The code begins by defining `f(x)`, and brings the `inner_product` function created in question 2a. The `composite_trapezium_rule` function is defined taking the parameters `function_values` and `weights`. This uses the `inner_product` function to calculate the approximate integral.
The main function is then initialised with the `N, a, b` and `delta_x` being defined representing the number of subdivisions, interval boundaries and the step size. A loop fills the valarrays by calculating the grid points, evaluating the function at those points, and assigning weights based on the trapezium rule and the approximate integral is calculated using the `composite_trampezium_rule`. The results are then outputted including the approximated integral and the difference between the $I_{trapezium}$ and I_{exact}. Which are `2.7423926434484638586` and `-0.00085475070254551876 32`. As the difference is very small this is a very good approximation.

part b) The following is the code for the solution:

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

using namespace std;

long double f(long double x) {
   return sin(1 / (x + 0.5));
}
```

```
long double df_exact(long double x){
   return (-1 / ((x + 0.5) * (x + 0.5))) * cos(1/ (x + 0.5));
}

// inner_product function from question 2a
long double inner_product(valarray<long double> u, valarray<long double> v) {
   return (u * v).sum();
}

long double composite_hermite_rule(const valarray<long double> &function_values,
const valarray<long double> &weights) {
   // Use the inner_product function instead of directly calculating the dot
product
   return inner_product(function_values, weights);
}

int main() {
   int N = 127; // N + 1 = 128
   long double a = 0;
   long double b = 10;
   long double delta_x = (b - a) / N;

   valarray<long double> gridpoints(N + 1);
   valarray<long double> function_values(N + 1);
   valarray<long double> weights(N + 1);

   // Populate the gridpoints, function_values, and weights arrays
   for (int i = 0; i <= N; ++i) {
       gridpoints[i] = a + i * delta_x;
       function_values[i] = f(gridpoints[i]);
       if (i == 0 || i == N) {
           weights[i] = delta_x * 0.5;
       } else {
           weights[i] = delta_x;
       }
   }

   // Calculate the approximate integral using the composite trapezium rule
   long double composite = composite_hermite_rule(function_values, weights);
   // RHS of hermite rule
   long double hermite_approx = ((delta_x * delta_x) / 12) * (df_exact(a) -
df_exact(b));
   // Output the results
   cout << setprecision(20) << "Approximated integral (hermite): " << composite +
hermite_approx << endl;
   cout << setprecision(20) << "Difference: " << composite + hermite_approx -
```

```
2.74324739415100920 << endl;

   return 0;
}
```

The following is the output of the code:

```
Approximated integral (hermite): 2.7432573470552861041
Difference: 9.9529042767267702357e-06

Process finished with exit code 0
```

The code above implements the composite Hermite rule to approximate the integral of a function `f(x) = sin(3x)`.
The code firstly defines `f(x)` which takes `x` as an input and also defines `df_exact(x)` which returns the exact derivative of `f(x)` as needed in the approximation. The code uses the inner_product function that was calculated in question 2a. The code follows the same as 4a where the number of grid points `N`, the interval `[a,b]` and `delta_x` are defined. This then follows the same until after the loop where the inner product of `w[i]` and `f[i]` are calculated and added to the `delta_x` squared divided by `12` which is multiplied by `f'(a) - f'(b)`. The approximated integral is then outputted along with the difference as can be seen above. The approximate is also very close to the approximate calculated in part a, with the error being relatively small.

part c) The following is the code for the solution:

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

using namespace std;

// inner_product function from question 2a
long double inner_product(valarray<long double> u, valarray<long double> v) {
   return (u * v).sum();
}

// Function f(x) = sin(1 / (x + 0.5))
long double f(long double x) {
   return sin(1.0 / (x + 0.5));
}

// Compute the modified Clenshaw-Curtis weights and perform the quadrature
```

```
long double clenshaw_curtis_quadrature(int n, double a, double b, long double
(*func)(long double)) {
   // define w, theta, x, f as valarrays
   valarray<long double> weights(n + 1);
   valarray<long double> theta(n + 1);
   double range = (b - a) * 0.5;
   double h = (b - a) / 2.0;
   valarray<long double> x(n + 1);
   valarray<long double> function_values(n + 1);

   // Loop through each point to calculate the weights and function values
   for (int i = 0; i <= n; ++i) {
       if (i == 0 || i == n) {
           weights[i] = range / (n * n);
       } else {
           theta[i] = i * M_PI / n;
           double cos_sum = 0.0;

           // Calculate the sum of cosines for the current point
           for (int k = 1; k <= n / 2; ++k) {
               cos_sum += 2.0 * cos(2.0 * k * theta[i]) / (4.0 * k * k - 1);
           }

           // Calculate the weight for the current point
           weights[i] = range * 2.0 / n * (1.0 - cos_sum);
       }
       // calculate x values and function values at x
       x[i] = a + h * (1.0 - cos(i * M_PI / n));
       function_values[i] = func(x[i]);
   }

   // Calculate the approximate integral using the Clenshaw-Curtis quadrature and
inner product function
   return inner_product(function_values, weights);
}

int main() {
   // define n, a and b
   int n = 127;
   double a = 0.0;
   double b = 10.0;

   // Calculate the approximated integral
   long double approx_integral = clenshaw_curtis_quadrature(n, a, b, f);

   // output the integral and the difference
   cout << setprecision(20) << "Approximated integral (Clenshaw-Curtis): " <<
```

```
approx_integral << endl;
   cout << setprecision(20) << "Difference: " << approx_integral -
2.74324739415100920 << endl;

   return 0;
}
```

The following is the output for the code:

```
Approximated integral (Clenshaw-Curtis): 2.7432473941510092236
Difference: -1.5373986805844452874e-16

Process finished with exit code 0
```

The code above implements the Clenshaw-Curtis quadrature weights and uses it to compute the integral of a given function over the specified interval.
The main function inside this code is the `clenshaw_curtis_quadrature` which takes as input the number of points n, the endpoints of the integration interval a and b, and a function pointer to the function `f(x)` being integrated. It first defines the valarrays that will hold the weights, theta values, x values and function values. It then loops through each point calculating the weights and function values at that point. For the weights, it first sets the weights at the endpoints to a fixed value and calculates the weights at the remaining points using the function specified. Once the weights and function values are calculated, it uses the `inner_product` function from question 2a to calculate the approximate integral using the Clenshaw-Curtis quadrature formula. Finally, in the main function, it calls `clenshaw_curtis_quadrature` with a specified number of points `n`, the endpoints of the integration interval a and b, and the function `f(x) = sin(1/(x+0.5))`. It then outputs the approximated integral and the difference between the approximated integral and the known exact value as can be seen above. The approximations are very close to the answers in part a and b, with the error also being very small.

part d) The following is the code for the solution:

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

using namespace std;

// define f(x) from the problem sheet
double f(const double x) {
   return sin(1 /(x + 0.5));
}
```

```
// Uniform distribution function from Exercise 19 in module page (modified)
class Uniform {
   mt19937_64 mt;
   uniform_real_distribution<double> u; // Use uniform_real_distribution instead of
uniform_int_distribution
public:
   Uniform(const unsigned int s) : mt(s) {}
   double operator()() {
       return u(mt);
   }
};


// Mean Value function from Exercise 19 in the module page (modified)
double mean_value(const double a, const double b, const int N, double
(*func)(double), Uniform& u) {
   double S = 0.0;
   for (int i = 0; i<N; i++) {
       const double x = a + (b-a)*u();
       S += func(x);
   }
   return (b - a)*S / double(N);
}


int main() {
   // bounds and uniform distribution with seed is defined
   const int a = 0;
   const int b = 10;
   const int seed = 1234;
   Uniform unif(seed);

   // loop to calculate and print the Mean Value Monte Carlo method and error
values with the N values specified
   for (int i = 1000; i <= 100000; i *= 10) {
       long double montecarlo = mean_value(a, b, i, f, unif);
       cout << setprecision(20) << "The Monte Carlo Mean value integration method
for  N = " << i << " is equal to: " << montecarlo << endl;
       cout << setprecision(20) << "The error value for the same value of N is: "
<< montecarlo - 2.74324739415100920 << endl;
       cout << " " << endl;
   }
   return 0;
}
```

The following is the output for the code:

```
The Monte Carlo Mean value integration method for  N = 1000 is equal to:
2.857785013439380073
The error value for the same value of N is: 0.11453761928837069561

The Monte Carlo Mean value integration method for  N = 10000 is equal to:
2.7245346262576712881
The error value for the same value of N is: -0.018712767893338089209

The Monte Carlo Mean value integration method for  N = 100000 is equal to:
2.7457079339471714974
The error value for the same value of N is: 0.0024605397961621200409

Process finished with exit code 0
```

The code above implements the Mean Value Monte Carlo integration method to estimate the value of the integral function `f(x) = sin (1 + (x + 0.5)).`
The code begins by defining `f(x)` and then by defining the Uniform class to generate random numbers required for the Monte Carlo method. This specific function is from the module page and has been modified to use the `uniform_real_distribution` instead of the integer version. The Mean Value Monte Carlo function is then defined to compute the Monte Carlo estimate of the integral with parameters `a, b, N, func, unif` which are the bounds, the size of `N`, the function for integration and the uniform distribution.
After this, the main function defines these values to be passed and a for loop is created to compute and output the integral I with `N = 1000, N = 10000 and N = 100000` samples. The difference `I_MonteCarlo - I_exact` is also outputted to the screen and we can see that as `N` increases, the error value decreases indicating a better estimate as `N` gets larger.

Question 5 - *Stellar Equilibrium*

part a) The following is the code for the solution:

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

valarray<long double> F(const long double t, const valarray<long double>
&u) {
    // define h, m, h' and m' from problem sheet
    long double h = u[0];
    long double m = u[1];
    long double h_dash;
    long double m_dash;

    // set functions based on given t
    if (t == 0) {
        h_dash = 0.0;  // h_dash[x] = 0 when x = 0
        m_dash = 0.0;  // m_dash[x] = 0 at x = 0
    } else if (t > 0) {
        h_dash = (-1/(t*t))*m;  // h_dash[x] = -m[x] / x^2 when x > 0
        m_dash = h*t*t;  // m_dash[x] = h[x] * x^2
    }

    // return valarray with h' and m' as parameters
    valarray<long double> f = {h_dash, m_dash};
    return f;
}


int main() {

    return 0;
}
```

The following is the output for the code:

```
return 0
```

The code above defines the function `F` as needed in the rest of question parts. It defines `h` and `m` as the first and second elements of the `u valarray` and also initiates `h'[x]` and `m'[x]`. Based on the conditions when `t = 0` and `t > 0`, we use the defined first order differential to define this. The `valarray f` is returned with `h'` and `m'` as parameters. As part a only asked for a function, there is no specific output.

part b & c) The following is the code for the solution:

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

valarray<long double> F(const long double t, const valarray<long double> &u) {
   // define h, m, h' and m' from problem sheet
   long double h = u[0];
   long double m = u[1];
   long double h_dash;
   long double m_dash;

   // set functions based on given t
   if (t == 0) {
       h_dash = 0.0;  // h_dash[x] = 0 when x = 0
       m_dash = 0.0;  // m_dash[x] = 0 at x = 0
   } else if (t > 0) {
       h_dash = (-1/(t*t))*m;  // h_dash[x] = -m[x] / x^2 when x > 0
       m_dash = h*t*t;  // m_dash[x] = h[x] * x^2
   }

   // return valarray with h' and m' as parameters
   valarray<long double> f = {h_dash, m_dash};
   return f;
}

// 4th-order Runge-Kutta method implementation with the parameters stated in
problem sheet
valarray<long double> RK4(const long double t, const long double dt, const
valarray<long double> &val, valarray<long double> f(const long double, const
valarray<long double> &)) {
   unsigned long long m = val.size();
   valarray<long double> k1(m), k2(m), k3(m), k4(m);
   k1 = dt * f(t, val);
   k2 = dt * f(t + 0.5 * dt, val + 0.5 * k1);
   k3 = dt * f(t + 0.5 * dt, val + 0.5 * k2);
   k4 = dt * f(t + dt, val + k3);
```

```
    return val + (k1 + 2.0*(k2 + k3) + k4)/6.0;
}


int main() {
    // initialise begin and final time
    long double ti = 0.;
    long double tf = M_PI;

    // define the exact and numerical values of h
    long double h_exact;
    long double h_numerical;

    // define n and step size
    int n = 150;
    long double dt = (tf-ti)/(n);

    // define valarrays to store time values, solution values and error values
    valarray<long double> T(n+2);
    valarray<valarray<long double>> U(n+2);
    valarray<long double> e(n+2);

    // initial condition from problem sheet
    U[0] = {1.0, 0.0};

    // initialise table with x, h[x] and error values
    cout << "x" << setw(30)
         << fixed << setprecision(20) << "h(x)"
         << setw(30) << fixed << setprecision(20)
         << "error: e[i]" << endl;

    // loop to calculate and output current time value, the RK4 solution, and error
value
    // where error value is h_numerical - h_exact.
    for (int i = 0; i <= n; i++) {
        T[i] = ti + i * dt;
        U[i + 1] = RK4(T[i], dt, U[i], F);
        h_numerical = U[i][0];
        if (T[i] == 0.0) {
            h_exact = 1;
            e[i] = h_numerical - h_exact;
        } else {
            h_exact = sin(T[i]) / T[i];
            e[i] = h_numerical - h_exact;
        }
        for (int j = 0; j <= n; j += 10) {
            if (i == j) {
```

```
                cout << setprecision(20) << T[i] << setw(30) << fixed
                     << setprecision(20) << U[i][0] << setw(30) << fixed
                     << setprecision(20) << e[i] << endl;
                break;
            }
        }
    }
    cout << endl;
    return 0;
```

The following is the output of the code:

```
X                           h(x)                         error: e[i]
0.00000000000000000000      1.00000000000000000000       0.00000000000000000000
0.20943951023931954107      0.99268533627883450093      -0.00001986332837225930
0.41887902047863908214      0.97099276194445305408      -0.00001944728730985513
0.62831853071795862320      0.93547054588383160535      -0.00001873790480743276
0.83775804095727816428      0.88704602516288019417      -0.00001776789630359322
1.04719755119659770537      0.82697677899451009200      -0.00001656413817799498
1.25663706143591724640      0.75681157059346761915      -0.00001515804718937755
1.46607657167523678754      0.67834245289833402907      -0.00001358564872926664
1.67551608191455632857      0.59354964706712187592      -0.00001188666155851545
1.88495559215387586960      0.50454104909263803019      -0.00001010333446665437
2.09439510239319541074      0.41348839238374606941      -0.00000827918259800329
2.30383461263251495188      0.32256219433188526799      -0.00000645768963614476
2.51327412287183449280      0.23386763992167696293      -0.00000468102548283603
2.72271363311115403394      0.14938350490430895750      -0.00000298882365460805
2.93215314335047357508      0.07090609719885867996      -0.00000141705879898657
3.14159265358979311600      0.00000000293958249709       0.00000000293958245811


Process finished with exit code 0
```

The code above implements a numerical method for solving a system of first-order ODEs. The method specified is the 4th-order Runge-Kutta method as specified in the problem sheet.
The code here begins by defining the function F, and then defining the RK4 function with parameters specified.
The `RK4` function starts by determining the size `m` of the state vector `val`, and initialising four valarray objects `k1, k2, k3,` and `k4` to store the intermediate slopes. `k1` is calculated as the product of dt and the derivative of the state variables at the current time and state, `f(t, val)`. `k2` is calculated as the product of dt and the derivative of the state variables at the midpoint of the time step `(t + 0.5 * dt)` and the midpoint of the state `(val + 0.5 * k1)`. `k3` is similar to `k2`, but uses `k2` instead of `k1` to estimate the state at the midpoint. `k4` is calculated as the product of `dt` and the derivative of the state variables at the end of the time step `(t +`

`dt)` and the estimated end state `(val + k3)`. The next state is calculated as the current state plus a weighted average of the four slopes. The weights for `k1` and `k4` are `1/6` each, while the weights for `k2` and `k3` are `1/3` each.
The main function of the code sets up an initial and final time, `ti` and `tf`, and calculates a time step `dt` based on a number of steps `n`. The code then initialises valarrays to store the time values `(T)`, the solution values `(U)`, and error values `(e)`.
The initial condition for `h` and `m` is set as `U[0] = {1.0, 0.0}` and the code then enters a loop where for each time step it calculates the next state using the `RK4` function, and the error between the numerical solution for `h` and the exact solution. The numerical solution `h_numerical` is calculated by the RK4 method while the exact solution `h_exact` is given by the sin function specified from the problem sheet. The error `e[i]` is the difference between these two values.
Finally, the code outputs the current time value, the numerical solution, and the error every 10th step in a tabular setting.
The error gets smaller and smaller as N tends to infinity and this is expected. `e[150]` is equal to `h(x)` when `n = 150`, and as `N` increases the error will tend to `0`.

part d) The following is the output of the code:

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

// same as part bc
valarray<long double> F(const long double t, const valarray<long double> &u) {
    long double h = u[0];
    long double m = u[1];
    long double h_dash;
    long double m_dash;
    if (t == 0) {
        h_dash = 0.0;
        m_dash = 0.0;
    } else if (t > 0) {
        h_dash = (-1/(t*t))*m;
        m_dash = h*t*t;
    }
    valarray<long double> f = {h_dash, m_dash};
    return f;
}


// Norm class from question 2c
class Norm {
```

```
public:
    int m;
    Norm(int m) : m(m) {}
    long double operator()(const valarray<double>& u) const {
        // initialise sum to store the weighted sum and interate through valarray
elements
        double sum = 0.0;
        for (size_t i = 0; i < u.size(); ++i) {
            sum += pow(u[i], m);
        }
        return pow(sum, 1.0 / m);
    }
};

// same as part bc
valarray<long double> RK4(const long double t, const long double dt, const
valarray<long double> &val, valarray<long double> f(const long double, const
valarray<long double> &)) {
    unsigned long long m = val.size();
    valarray<long double> k1(m), k2(m), k3(m), k4(m);
    k1 = dt * f(t, val);
    k2 = dt * f(t + 0.5 * dt, val + 0.5 * k1);
    k3 = dt * f(t + 0.5 * dt, val + 0.5 * k2);
    k4 = dt * f(t + dt, val + k3);
    return val + (k1 + 2.0*(k2 + k3) + k4)/6.0;
}


int main() {
    // same as part bc
    long double ti = 0.;
    long double tf = M_PI;
    long double h_exact;
    long double h_numerical;
    int n = 150;
    long double dt = (tf-ti)/(n);
    valarray<long double> T(n+2);
    valarray<valarray<long double>> U(n+2);
    valarray<double> e(n + 2);
    U[0] = {1.0, 0.0};
    for (int i = 0; i <= n; i++) {
        T[i] = ti + i * dt;
        U[i + 1] = RK4(T[i], dt, U[i], F);
        h_numerical = U[i][0];
        if (T[i] == 0.0) {
            h_exact = 1;
            e[i] = h_numerical - h_exact;
```

```
        } else {
            h_exact = sin(T[i]) / T[i];
            e[i] = h_numerical - h_exact;
        }
    }

    // create the l1 and l2 norm instances from Norm class
    Norm l1norm(1);
    long double norm_l1 = l1norm(abs(e));
    Norm l2norm(2);
    long double norm_l2 = l2norm(abs(e));

    // output results
    cout << setprecision(20);
    cout << "L1 Norm: " << norm_l1 << endl;
    cout << "L2 Norm: " << norm_l2 << endl;
    return 0;
    }
```

The following is the output of the code:

```
L1 Norm: 0.0017577184344788304281
L2 Norm: 0.0001638307934484686931 2

Process finished with exit code 0
```

The code above calculates and outputs the `L1` and `L2` Norms of the error values.
The code begins exactly the same as part b&c, where `F` is defined, the `Norm` class is brought from question 2c, `RK4` is defined, and the main function calculates the first-order ODEs using the `RK4` function. The error values are calculated for each `i`, and assigned to the valarray as specified. Instances of the Norm class are initiated as `L1Norm` and `L2Norm`. The absolute value of the error is taken and passed to both the `L1Norm` and `L2Norm` as needed. This is then outputted as can be seen above.
The `L1 norm` is `0.00176`, which is relatively small, suggesting that the overall absolute error in the numerical solution is quite low. The `L2 norm` is around `0.000164`, which is even smaller than the `L1 norm`. The `L2 norm` tends to emphasise larger errors because they contribute more to the sum when squared. Thus, this small `L2 norm` suggests that there are no particularly large errors in the numerical solution, or in other words, there are no significant outliers in the error.