Project 1
SF2568 Program construction in C++ for Scientific Computing

Caroline Eriksson Sophie Malmliden

September 2019

Task 1

The goal of this exercise is to write two functions, sinTaylor(N, x) and cosTaylor(N, x), that calculate the sum of the first N terms in the Taylor series of the sine function as well as the cosine function. Further we are to compare the results with the sine and cosine functions included in the C standard library and show how the error is affected by the input value x and the number of terms from the sums.

The functions sinTaylor(N, x) and cosTaylor(N, x) are created by implementing the polynomial evaluation using Horner's scheme. Consider the Taylor series of the sine and cosine function, the first six terms explicitly printed:

$$sin(x) = \sum_{n=0}^{\infty}(-1)^n \frac{x^{2n+1}}{(2n+1)!} = \frac{x}{1} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} \quad ... \quad = \tag{1}$$

The Taylor series of the sine function using Horner's Method:

$$= \frac{x}{1}\left(1 - \frac{x^2}{2\cdot 3}\left(1 - \frac{x^2}{4\cdot 5}\left(1 - \frac{x^2}{6\cdot 7}\left(1 - \frac{x^2}{8\cdot 9} \quad ... \quad \right)\right)\right)\right) \tag{2}$$

$$cos(x) = \sum_{n=0}^{\infty}(-1)^n \frac{x^{2n}}{(2n)!} = 1 - \frac{x^2}{2} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \frac{x^{10}}{10!} \quad ... \quad = \tag{3}$$

The Taylor series of the cosine function using Horner's Method:

$$= 1 - \frac{x^2}{2}\left(1 - \frac{x^2}{3\cdot 4}\left(1 - \frac{x^2}{5\cdot 6}\left(1 - \frac{x^2}{7\cdot 8}\left(1 - \frac{x^2}{9\cdot 10} \quad ... \quad \right)\right)\right)\right) \tag{4}$$

This method is implemented in the code for creating the numerical functions sinTaylor(N, x) and cosTaylor(N, x). For further details and comments view the code in *Appendix 1.*

The error for the sinTaylor(N, x) and cosTaylor(N, x) functions is calculated for six different values of x; -1, 1, 2, 3, 5, 10 with from one to 25 number of terms from the Taylor series of the functions, which is displayed in the tables; *Table of error for sinTaylor(N, x)* and *Table of error for cosTaylor(N, x)* below. Each column correspond to one x value and each row correspond to one N-value. For further details and comments view the code in *Appendix 1.*

Table of error for sinTaylor(N, x)

	-1	1	2	3	5	10
0	0.158529	0.158529	1.0907	2.85888	5.95892	10.544
1	0.00813765	0.00813765	0.242631	1.64112	14.8744	156.123
2	0.000195682	0.000195682	0.0240359	0.38388	11.1673	677.211
3	2.73084e-06	2.73084e-06	0.00136092	0.0500486	4.33373	1306.92
4	2.48923e-08	2.48923e-08	5.00159e-05	0.00419249	1.04855	1448.82
5	1.59828e-10	1.59828e-10	1.29086e-06	0.000245414	0.174693	1056.4
6	7.61946e-13	7.61946e-13	2.4694e-08	1.06191e-05	0.0213402	549.509
7	2.77556e-15	2.77556e-15	3.64234e-10	3.53719e-07	0.00199707	215.207
8	0	0	4.26925e-12	9.35338e-09	0.000147906	65.9385
9	0	0	4.06342e-14	2.01152e-10	8.89053e-06	16.2678
10	0	0	4.44089e-16	3.5876e-12	4.42572e-07	3.30511
11	0	0	0	5.3485e-14	1.85497e-08	0.563058
12	0	0	0	4.71845e-16	6.6372e-10	0.0816369
13	0	0	0	4.71845e-16	2.05128e-11	0.0102
14	0	0	0	4.71845e-16	5.53668e-13	0.00110996
15	0	0	0	4.71845e-16	1.80966e-14	0.00010617
16	0	0	0	4.71845e-16	1.44329e-15	8.99724e-06
17	0	0	0	4.71845e-16	1.44329e-15	6.80353e-07
18	0	0	0	4.71845e-16	1.44329e-15	4.61929e-08
19	0	0	0	4.71845e-16	1.44329e-15	2.83155e-09
20	0	0	0	4.71845e-16	1.44329e-15	1.58004e-10
21	0	0	0	4.71845e-16	1.44329e-15	7.88059e-12
22	0	0	0	4.71845e-16	1.44329e-15	7.23643e-13
23	0	0	0	4.71845e-16	1.44329e-15	4.46088e-13
24	0	0	0	4.71845e-16	1.44329e-15	4.46088e-13

Table of error for cosTaylor(N, x)						
	-1	1	2	3	5	10
0	0.459698	0.459698	1.41615	1.98999	0.716338	1.83907
1	0.0403023	0.0403023	0.583853	2.51001	11.7837	48.1609
2	0.00136436	0.00136436	0.0828135	0.864992	14.258	368.506
3	2.45281e-05	2.45281e-05	0.00607539	0.147508	7.44338	1020.38
4	2.73497e-07	2.73497e-07	0.000273821	0.0152157	2.24474	1459.78
5	2.07625e-09	2.07625e-09	8.36627e-06	0.00105661	0.446409	1295.96
6	1.14231e-11	1.14231e-11	1.84845e-07	5.28659e-05	0.0632776	791.719
7	4.77396e-14	4.77396e-14	3.09176e-09	1.9983e-06	0.00673425	355.355
8	1.11022e-16	1.11022e-16	4.05174e-11	5.91063e-08	0.000558655	122.593
9	0	0	4.2738e-13	1.40571e-09	3.71704e-05	33.5995
10	0	0	3.60822e-15	2.74698e-11	2.02867e-06	7.50364
11	0	0	1.66533e-16	4.48752e-13	9.24909e-08	1.39316
12	0	0	1.66533e-16	5.77316e-15	3.57612e-09	0.218582
13	0	0	1.66533e-16	2.22045e-16	1.18764e-10	0.0293776
14	0	0	1.66533e-16	2.22045e-16	3.42298e-12	0.00342131
15	0	0	1.66533e-16	2.22045e-16	8.53206e-14	0.000348676
16	0	0	1.66533e-16	2.22045e-16	8.32667e-16	3.1363e-05
17	0	0	1.66533e-16	2.22045e-16	8.32667e-16	2.5086e-06
18	0	0	1.66533e-16	2.22045e-16	8.32667e-16	1.79616e-07
19	0	0	1.66533e-16	2.22045e-16	8.32667e-16	1.15802e-08
20	0	0	1.66533e-16	2.22045e-16	8.32667e-16	6.75559e-10
21	0	0	1.66533e-16	2.22045e-16	8.32667e-16	3.59103e-11
22	0	0	1.66533e-16	2.22045e-16	8.32667e-16	1.23779e-12
23	0	0	1.66533e-16	2.22045e-16	8.32667e-16	1.12133e-14
24	0	0	1.66533e-16	2.22045e-16	8.32667e-16	1.12133e-14

For sinTaylor(N, x), the higher value for the input variable x, the bigger error is achieved, with very few exceptions. For x equal to -1, 1, 2 and 3, the error decrease for each added term from the Taylor series. But when x is 5 and 10, the error initially increases for some added terms, to later again decrease for every added term. One can also conclude that for the values of x equal to -1, 1 and 2, the error will fully converge to 0, but for x equal to 3, 5 and 10, the error will only converge to a number close to 0.

For cosTaylor(N, x), similar results are achieved, the higher value for the input variable x, the bigger error is achieved, with very few exceptions. For x equal to -1, 1 and 2, the error decreases for each added term from the Taylor series. But when x is 3, 5 and 10, the error initially increases for some added terms, to later again decrease for every added term. One can also conclude that for the values of x equal to -1 and 1, the error will fully converge to 0, but for x equal to 2, 3, 5 and 10, the error will only converge to a number close to 0.

Task 2

The goal of this exercise is to compute an approximation to the integral below in equation 5 using prescribed tolerance and the adaptive Simpson's method for numerical integration. Approximations of the integral with the adaptive Simpson's method for the tolerances $10^{-2}, 10^{-3}, 10^{-4}$ are to be computed and compared with the result provided by Matlab's integration functions.

$$\int_{-1}^{1} (1 + sin(e^{3x}))dx \tag{5}$$

Denote the function inside the integral in equation 5 with $f(x)$. The adaptive Simpson's method is a recursive numerical integration method, which uses an estimate of the error we get from calculating a definite integral using Simpson's rule. If the error exceeds the tolerance, the algorithm calls for subdividing the interval of integration in two and applying adaptive Simpson's method to each subinterval in a recursive manner. The definite integral calculated with Simpson's rule can be written as in equation 6 below. The estimate of the error can be written as in equation 7 below, with help of the integral approximation $I2(a, b) := I1(a, (a + b)/2) + I1((a + b)/2, b)$.

$$I1(a, b) = \frac{b - a}{6}(f(a) + 4f((a + b)/2) + f(b)) \tag{6}$$

$$errest = \frac{1}{15}(I2(a, b) - I1(a, b)) \tag{7}$$

For further details and comments view the code in Appendix 2.

In the table below, the results of this exercise are presented. From the numerical Adaptive Simpson's Integration method, the Integral given is approximated with the tolerances; $10^{-2}, 10^{-3}, 10^{-4}$ and compared to the result of the integration function with the tolerance 10^{-8} from Matlab. The errors presented in the table are calculated with the function value from Matlab as reference.

Table: Integral values and errors for ASI method

Tolerances	10^{-2}	10^{-3}	10^{-4}	$10^{-8}(Matlab.Int.func.)$
Integral value	2.506	2.49986	2.50081	2.500809110336167
Error	0.00519088966	0.00094911034	0.00000088966	

As the table presents; the Adaptive Simpson's method yields a more precise integral value and thus a smaller error the lower the tolerance is, which is to be expected.

Appendix 1: C++ code for Task 1

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

// declaration of functions before use
long double sinTaylor(double, double);
long double cosTaylor(double, double);

// Creating function sinTaylor(N, x) that calculates the sum of
// the first N terms in the Taylor series of the sine function
// for the variable x.
long double sinTaylor(double N, double x)
{
        long double result;

        if (N == 0) //the first term in the sum is calculated separately
        {
                result = x; //the first term is calculated here; x/1
        }
        else //for more terms from the sum, the calculations will be different
        {
                //The sum for N = 1, will be A defined right below times x.
                //For higher N's A right below will be the content of the
                //innermost paranthesis of the polynomialization of the sum.
                long double A = 1 - (x / (2 * N)) * (x / (2 * N + 1));
                //for higher values of N more calculations are necessary
                for (double i = N-1; i > 0; --i)
                {
                        // multiplying the innermost paranthesis A with
                        // the closest fraction from the polynomial
                        long double B = A * (x / (2 * i)) * (x / (2 * i + 1));
                        //A right below form the two innermost parantheses
                        A = 1 - B;
                        // this loop keeps repeating to fill up all
                        // paranthesis of the polynomial
                }
                //now only the outer x/1 is missing, multiply here
                result = x * A;
        }
        return result;
}

// Creating function cosTaylor(N, x) that calculates the sum of
// the first N terms in the Taylor series of the cosine function
// for the variable x.
long double cosTaylor(double N, double x)
{
        long double result;

        if (N == 0)  //the first term in the sum is calculated separately
        {
                result = 1; //the first term is calculated here; 1
        }
        else //for more terms from the sum, the calculations will be different
        {
                //The sum for N = 1, will be A defined right below.
                //For higher N's A right below will be the content of the
                //innermost paranthesis of the polynomialization of the sum.
                long double A = 1 - (x / (2 * N) * (x / (2 * N - 1)));

                //for higher values of N more calculations are necessary
                for (double i = N - 1; i > 0; --i)
                {
                        // multiplying the innermost paranthesis A with
                        // the closest fraction from the polynomial
                        long double B = A * (x / (2 * i)) * (x / (2 * i - 1));
                        //A right below form the two innermost parantheses
                        A = 1 - B;
                        // this loop keeps repeating to fill up all
                        // paranthesis of the polynomial
                }
                //A will now be the full polynomial
                result = A;
        }
        return result;
}

int main()
{
        //declaration of variable before use, here choose input
        //variable for the approximations
        double x = 2;
        //initialization of vector before use, later use for
        //filling up with errors from sine approximation
        vector<long double> s;
        //initialization of vector before use, later use for
        //filling up with errors from cosine approximation
        vector<long double> c;

        for (double N = 0; N <= 24; ++N)
        {
                //definition of variable, error between value of
                //approximative sine function and sine function
                //from C standard library
                long double e_s = abs(sin(x) - sinTaylor(N, x));
                //definition of variable, error between value of
                //approximative cosine function and cosine function
                //from C standard library
                long double e_c = abs(cos(x) - cosTaylor(N, x));
                //filling the vectors with the errors of the
                //approximative sine and cosine functions for one
                //input value x and 1 to 25 number of terms from
                //the Taylor series.
                s.push_back(e_s);
                c.push_back(e_c);
                //printing out the errors of the approximations
                cout << e_s << endl;
                // cout << e_c << endl;
        }
        //printing out the exact and approximate values and corresponding errors
        //cout << sin(x) << " " << sinTaylor(N,x) << " " << e_s << endl;
        //cout << cos(x) << " " << cosTaylor(N, x) << " " << e_c << endl;
        return 1;
}
```

Appendix 2: C++ code for Task 2

```
#include <iostream>
#include <cmath>
#include <math.h>
using namespace std;

//declaration of function
double f(double x)
{
        //definition of function
        double f = 1 + sin(exp(3 * x));
        return f;
}
//declaration of function
double I1(double a, double b)
{
        //definition of function, integral using Simpson's rule
        double I_1 = ((b - a) / 6) * (f(a) + 4 * f((a + b) / 2) + f(b));
        return I_1;
}
//declaration of function
double I2(double a, double b)
{
        //definition of variable, midpoint
        double gamma = (a + b) * 0.5;
        //definition of function, helping calculate error
        double I_2 = I1(a, gamma) + I1(gamma, b);
        return I_2;
}

//declaration of function, adaptive Simpson's Integration
double ASI(double a, double b, double tol)
{
        //compute I1 integral between values a and b
        double I_1 = I1(a, b);
        //compute I2 integral for estimating error
        double I_2 = I2(a, b);
        //calculate error estimation
        double errest = abs(I_1 - I_2);

        if (errest < 15 * tol) //decide if error is too big
        {
                //end loop if error estimate satisfies tolerance
                return I_2;
        }
        else
        {
                //keep looping if error estimate does not satisfy
                //condition, algorithm will subdivide interval of
                //integration in two and apply ASI recursively
                return ASI(a, (a + b) / 2, tol / 2) + ASI((a + b) / 2, b, tol / 2);
        }

}
```

```
int main()
{
        // declaration of variable before use, lower integration limit
        double a = -1;
        // declaration of variable before use, higher integration limit
        double b = 1;
        // declaration of variable before use, tolerance
        double tol = pow(10, -4);

        // declaration of function, final approximation of integral
        double I = ASI(a, b, tol);
        // printing result
        cout << I << endl;

        return 1;
}
```