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

Pia Callmer, Jonas Fey

September 2022

1 Exercise 1 - Taylor series of sin and cos

1.1 Task

The task in this exercise is to implement the taylorseries of the sine and cosine function by using

$$\sin(x) = \sum_{n=0}^{\infty} (-1)^n \cdot \frac{x^{2n+1}}{(2n+1)!}$$
$$\cos(x) = \sum_{n=0}^{\infty} (-1)^n \cdot \frac{x^{2n}}{(2n)!}.$$

To avoid overflow computations we make use of Horner's scheme to evaluate high order polynomials. For the sine function, we can divide the polynomials, such that we are not taking the expontial directly, which results in

$$\sin_n(x) \approx x \cdot \left(1 - \frac{x \cdot x}{2 \cdot 3}\left(1 - \frac{x \cdot x}{4 \cdot 5}\left(... \left(1 - \frac{x \cdot x}{(2n+1)(2n)}\right)\right)\right)\right).$$

Calculating the Taylor series of the cosine function looks similar with

$$\cos_n(x) \approx 1 - \frac{x \cdot x}{1 \cdot 2}\left(1 - \frac{x \cdot x}{3 \cdot 4}\left(... \left(1 - \frac{x \cdot x}{2n \cdot (2n-1)}\right)\right)\right).$$

Both equations can be implemented in C++ by using a for loop and evaluating the terms in brackets iteratively going from the inner most bracket to the outer one.

To compare the end results we printed out the errors between the approximation and the sine and cosine functions provided by the `<cmath>` library

$$|\sin(x) - \texttt{sinTaylor(N, x)}|, \quad |\cos(x) - \texttt{cosTaylor(N, x)}|.$$

1.2 Results

For implementation we choose $x = -1, 1, 2, 3, 4, 10$ with a different $N = 1, 3, 6, 10$. The results of the estimation error are displayed in table 1.

It can be seen that the error is lower if the Taylor approximation is evaluated close to the point it was expanded at, which is $x_0 = 0$ for this case. For values that stray further away from this x_0 the error grows exponentially as expected because a polynomial approximation is used to model a periodic function. The standard library method is able to avoid this growing error for values far away from x_0 by, most likely, using the periodic nature of sin and cos and shifting all values onto the range $[-\pi, \pi]$. This could also be implemented for the Taylor approximation implementation which would significantly reduce the error for values away from $x_0 = 0$. Furthermore, it can be seen that increasing N reduces the error for almost all cases. This is again to be expected as a higher N means a higher polynomial degree of the Taylor polynomial and thus a better approximation. However, for $x = 10$ it can be seen that increasing N actually can increase the

x	N	error $\sin(x)$	sinTaylor$(N+1, x)$	error $\cos(x)$	cosTaylor$(N+1, x)$
-1	1	0.00813765	0.166667	0.0403023	0.5
	3	2.73084e-006	0.000198413	2.45281e-005	0.00138889
	6	7.61946e-013	1.6059e-10	1.14231e-011	2.08768e-09
	10	0	1.95729e-20	0	4.11032e-19
1	1	0.00813765	0.166667	0.0403023	0.5
	3	2.73084e-006	0.000198413	2.45281e-005	0.00138889
	6	7.61946e-013	1.6059e-10	1.14231e-011	2.08769e-09
	10	0	1.95729e-20	0	4.11032e-19
2	1	0.242631	1.33333	0.583853	2
	3	0.00136092	0.0253968	0.00607539	0.0888889
	6	2.4694e-008	1.31556e-06	1.84845e-07	8.55112e-06
	10	2.22045e-016	4.10474e-14	3.83027e-015	4.3098e-13
3	1	1.64112	4.5	2.51001	4.5
	3	0.0500486	0.433929	0.147508	1.0125
	6	1.06191e-005	0.000256033	5.28659e-005	0.00110948
	10	3.5876e-012	2.0474e-10	2.74702e-011	1.43318e-09
5	1	14.8744	20.8333	11.7837	12.5
	3	4.33373	15.501	7.44338	21.7014
	6	0.0213402	0.196033	0.0632776	0.509686
	10	4.42572e-007	9.33311e-06	2.02867e-006	3.9199e-05
10	1	156.123	166.667	48.1609	50
	3	1306.92	1984.13	1020.38	1388.89
	6	549.509	1605.9	791.719	2087.68
	10	3.30511	19.5729	7.50364	41.1032

Table 1: Error estimates for the Taylor approximation of sine and cosine functions compared to the standard library. Additionally the $N+1$-th term of the corresponding Taylor series which bounds the error is shown.

error. This is again due to the fact that $x = 10$ is far away from the expansion point x_0. A higher polynomial has more oscillations and therefore the error can be unpredictable for values far away from x_0. The function is very sensitive in this area. Again, this can be fixed by shifting all values onto the range $[-\pi, \pi]$. For the error bounds it can be observed that the error is bounded by the $N+1$-th term of the corresponding Taylor series evaluated at the same point as every error in Table 1 is smaller than the $N+1$-th term.

2 Exercise 2 - Adaptive Integration

In this task the goal is to approximate the integral of a given function with the adaptive Simpson rule. The adaptive Simpson rule reads

$$I(\alpha, \beta) = \frac{\beta - \alpha}{6} \left(f(\alpha) + 4f(\frac{\alpha + \beta}{2}) + f(\beta))) \right) \tag{1}$$

By introducing a midpoint $\gamma = \frac{1}{2}(\alpha + \beta)$ and $I_2(\alpha, \beta) = I(\alpha, \gamma) + I(\gamma, \beta)$ the error can be reduced.
For the integral

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

and the tolerances `tol` $= 10^{-2}, 10^{-3}, 10^{-4}$ we get the following results in Table 2.

Looking at the error value from Table 2 for the tolerance 10^{-6} it can be seen that the adaptive Simpson integration algorithm (ASI) got the same five digits as MATLAB's integration method which indicates a good accuracy of ASI. Furthermore, increasing the tolerance also decreases the error, as expected. It can however be observed that the estimation value first decreases when decreasing the tolerance from 10^{-2} to

Tolerance	Estimation	Error compared to Matlab
0.01	2.506	0.00519
0.001	2.49986	0.00095
0.0001	2.50081	0

Table 2: Estimation with the adaptive Simpson rule with different tolerance rates compared to MATLAB's integration function. Values rounded to five digits.

10^{-4} but then increases when further decreasing the tolerance from 10^{-4} to 10^{-6}. This is likely due to the periodic nature of the sinus function. As the frequency of sinus increases exponentially of the given function, it can be expected that the function will be very sensitive for increasing x values. As the ASI algorithm takes values at certain points in the interval and evaluates them for the function, it can deliver very different values (here between 0 and 1) for just sligthly different x values if x is larger. Therefore, care has to be taken when evaluating such a function for an even higher upper bound of the interval.

Appendix – Code files

Appendix 1 – main.cpp

```
#include <iostream>
#include <cmath>

#include "ex1.h"
#include "ex2.h"

// example function given for Task 2
double func(double x){
    return (1 + sin(exp(3*x)));
}

// Main function to execute Task 1 and 2
int main(){

    // Task 1
    std::cout<<"Executing Task 1" << std::endl;

    // function to calculate the estimated error
    error();

    std::cout << std::endl;
    std::cout << "--------------------------" << std::endl;
    std::cout << std::endl;

    // Task 2
    std::cout << "Executing Task 2" << std::endl;
    std::cout << std::endl;

    // MATLAB result for integration using integral() function with
       ↪ absolute tolerance
    // of 1e-8 rounded to 5 digits
    double matlab_res = 2.50081;

    std::cout << "Using MATLAB's integration method with an absolute
       ↪ tolerance of 1e-8 results in: " << matlab_res << std::endl;

    // Print Integral estimate for different tolerances
    double tol [] = {0.01, 0.001, 0.0001};
    for (int i = 0; i < 3; i++){
        // ASI function takes left border, right, border, tolerance and a
           ↪ function which takes a double and returns a double
        double asi_res =    ASI(&func,-1, 1, tol[i]);
        // Comparing ASI result to MATLAB result
        std::cout<<"Tolerance: " << tol[i]  << ", ASI result: " << asi_res
           ↪  <<  ", Difference to MATLAB result: " << std::abs(
           ↪ matlab_res - asi_res)<< std::endl;
    }

    return 0;
}
```

Appendix 2 – ex1.h

```
#ifndef EX1_H_INCLUDED
#define EX1_H_INCLUDED
// Own build factorial function
double factorial(double );

// Taylor estimation for sin
double sinTaylor(int , double );

// Taylor estimation for cos
double cosTaylor(int , double );

// Function to calculate error between cmath sin/cos functions and Taylor
   ↪ apporximations
void error();

#endif
```

Appendix 3 – ex1.cpp

```
#include <iostream>
#include <cmath>
#include "ex1.h"


// Recursive factorial evaluation for the error estimation
double factorial(double n)
{
    if (n == 0){
       return 1;
    }
    return n * factorial(n - 1);
}


// Taylor approximation of the sine function
double sinTaylor(int N, double x)
{

    //check and return x if N = 0 to no divide through 0
    if (N == 0) {
        return x;
    }
    // Initialize temporary variable
    double temp;

    // Initialize current result variable res as the right part of the
       ↪ innermost bracket (1 - res)
    double res = (x*x)/((2*N)*(2*N+1));

    // Evaluate the result using horners scheme. Start from N and decrease
    for (int i=N; i > 1 ; i--){
        // Temp values evaluates the current innermost bracket (1-res)
        temp = 1-res;
```

```
        // Multiply temp value (current solution of the horner scheme so
           ↪ far) with the right term of the next bracket
        res =  temp*x*x/((2*i-1)*(2*i-2));

    }

    // Multiply with x as sine Taylor approximation starts with x (see
       ↪ equation in documentation)
    return x*(1-res);
}

//Taylor approximation of the cosine function
double cosTaylor(int N, double x)
{

     //check and return 1 if N = 0 to no divide through 0
    if (N == 0) {
        return 1;
    }

    // Initialize temporary variable
    double temp;

    // Initialize current result variable res as the right part of the
       ↪ innermost bracket (1 - res)
    double res = x*x/((2*N)*(2*N-1));

    // Evaluate the result using horners scheme. Start from N-1 and
       ↪ decrease
    for (int i=N-1; i >= 1; i--){
        // Temp values evaluates the current innermost bracket (1-res)
        temp = 1-res;

        // Multiply temp value (current solution of the horner scheme so
           ↪ far) with the right term of the next bracket
        res = temp*(x*x)/((2*i)*(2*i-1));

    }

    // Return evaluation of outermost bracket (1-res)
    return 1-res;
}


//Prints error for the Taylor approximations of sine and cosine for
   ↪ different N and x
void error()
{
    //set array sequences with x and N values
    double x_s[] = {-1, 1,2, 3, 5 ,10}; //
    int N_s[] = {1, 3 , 6, 10};

    int num_N = sizeof(N_s)/sizeof(N_s[0]);
```

```
    int num_x = sizeof(x_s)/sizeof(x_s[0]);

    //iterate over different x_s
    for (int j = 0; j < num_x; j++){
        std::cout << "----------------------------" << std::endl;
        std::cout << "For x=" << x_s[j] << ":" << std::endl;
        //iterate over number of Ns
        for (int i = 0; i< num_N; i++ ){

            // Calculating the n+1 term of the sinus Taylor approximation
               ↪ to check if it bound the error
            double sin_error_bound =  std::abs((1/factorial(2*N_s[i]+1))*(
               ↪ pow(x_s[j], 2*N_s[i]+1)));


            // Calculating the n+1 term of the cosinus Taylor
               ↪ approximation to check if it bound the error
            double cos_error_bound =  std::abs((1/factorial(2*N_s[i]))*(
               ↪ pow(x_s[j], 2*N_s[i])));

            // Calculate absolute error value
            double err_sin = std::abs(sin(x_s[j])-sinTaylor(N_s[i],x_s[j])
               ↪ );
            double err_cos = std::abs(cos(x_s[j])-cosTaylor(N_s[i],x_s[j])
               ↪ );

            // Printing out the errors when using the Taylor approximation
               ↪  and the estimated error bound of the n+1 term
            std::cout << "N=" << N_s[i] << ", x=" << x_s[j] << ",   error
               ↪ sin: " << err_sin << ",   n+1 term:" << sin_error_bound
               ↪ << std::endl;
            std::cout << "N=" << N_s[i] << ", x=" << x_s[j] << ",   error
               ↪ cos: " << err_cos << ",   n+1 term:" << cos_error_bound
               ↪ << std::endl;
        }
    }
}
```

Appendix 4 – ex2.h

```
#ifndef EX2_H_INCLUDED
#define EX2_H_INCLUDED


// Defining new type FunctionPointer which is used for ASI function
// Points to a function which takes a double and returns a double
typedef double (*FunctionPointer)(double);

// I function as described by the ASI algorithm
double I(FunctionPointer, double , double );


// I_2 function as described by the ASI algorithm
double I_2(FunctionPointer, double , double );
```

```
// Adaptive Simpson Integration algorithm function
double ASI(FunctionPointer, double , double  , double );


#endif // EX2_H_INCLUDED
```

Appendix 5 – ex2.cpp

```
#include <iostream>
#include <cmath>
#include "ex2.h"


// Use Simpsons rule to estimate Integral value
double I(FunctionPointer f, double a, double b){
    return ((b-a)/6)*(f(a) +4*f((a+b)/2)+ f(b));

}

// Use I2  to decrease error value for Integral approximation
double I_2(FunctionPointer f, double a, double b){
    double gamma = (a+b)/2;
    return I(f, a, gamma) + I(f, gamma,b);
}

// Adaptive Simpson Estimation
double ASI(FunctionPointer f, double a, double  b, double tol){
    double I1 =  I(f, a, b);
    double I2 = I_2(f, a, b);
    double errest = std::abs(I1-I2);
    if (errest < 15*tol){
        return I2;
    }
    return ASI(f, a, (a+b)/2, tol/2) + ASI(f, (a+b)/2, b, tol/2);
}
```