Report Project1
SF2565: Program construction in C++ for Scientific Computing

Marcus Klasson

September 23, 2019

Task 1: Taylor Series

We begin by focusing on the Taylor series of $\sin(x)$ and inspect the fractional terms T at step n and $n-1$:

$$T_n = \frac{x^{2n+1}}{(2n+1)!}, \qquad T_{n-1} = \frac{x^{2(n-1)+1}}{(2(n-1)+1)!} = \frac{x^{2n-1}}{(2n-1)!}.$$

We can write T_n with T_{n-1} as

$$T_n = \frac{x^{2n+1}}{(2n+1)!} = \frac{x^{2n-1}}{(2n-1)!}\frac{x^2}{2n\cdot(2n+1)} = T_{n-1}\frac{x^2}{2n\cdot(2n+1)},$$

which gives us a sequential computation of the T_n that does not require us to explicitly compute the factorial for each term in the sum. In the function `sinTaylor`, we therefore compute the Taylor series of $\sin(x)$ with

$$\sin(x) = T_0 + \sum_{n=1}^{\infty}(-1)^n T_{n-1}\frac{x^2}{2n\cdot(2n+1)},$$

where $T_0 = x$. This approach corresponds to Horner's scheme on $\sin(x)$ with equation

$$\begin{aligned}\sin(x) =& \sum_{n=0}^{\infty} a_n x^{2n+1} \\ =& a_0 x + a_1 x^3 + a_2 x^5 + \cdots + a_{n-1}x^{2(n-1)+1} + a_n x^{2n+1} \\ =& x(a_0 + x^2(a_1 + x^2(a_2 + \cdots + x^2(a_{n-1} + a_n x^2)))),\end{aligned}$$

where $a_0 = 1$ and $a_n = (-1)^n/(2n+1)!$. We can use the same approach for $\cos(x)$ where the fractional terms S_n and S_{n-1} are given by

$$S_n = \frac{x^{2n}}{(2n)!}, \qquad S_{n-1} = \frac{x^{2(n-1)}}{(2(n-1))!} = \frac{x^{2n-2}}{(2n-2)!}.$$

We write S_n with S_{n-1} as

$$S_n = \frac{x^{2n}}{(2n)!} = \frac{x^{2n-2}}{(2n-2)!}\frac{x^2}{(2n-1)\cdot 2n} = S_{n-1}\frac{x^2}{(2n-1)\cdot 2n}.$$

The function `cosTaylor` is then computed using the equation

$$\cos(x) = S_0 + \sum_{n=1}^{\infty}(-1)^n S_{n-1}\frac{x^2}{(2n-1)\cdot 2n},$$

where $S_0 = 1$.

The results suffered from overflow when x was greater than π. Since x is periodic with 2π and that sine and cosine have function values between -1 and 1, we can scale the values of x such that they are within an interval where we can compute the Taylor series without overflow. This is called argument reduction and I decided to reduce x into a variable y which is in the interval $[-\pi, \pi]$, while keeping $\sin(x) = \sin(y)$ and $\cos(x) = \cos(y)$. I defined the reduction as $y = x2\pi k$, where k is the nearest integer to $x/2\pi$[1]. This argument reduction was implemented in the function `argumentReduction`. The precision of the argument reduction depends on how many digits there are in the constant for π in the `cmath` library named `M_PI`, i.e. the number of digits in the `M_PI` must be equal or greater than the number of digits in x. Since $x = 10$ is the largest value we will use for evaluating $\sin(x)$ and $\cos(x)$, we do not have to worry about the precision in the reduction.

We need to verify that the absolute errors are bounded by the $(N + 1)$'st term in the Taylor series for different numbers of terms. We display the outputs of the main program `taylor.cpp` for $n = 1, 2, 5, 10$ and 100 in the .pdf named `Printouts_Sourcecode_MarcusKlasson.pdf`. When $n = 5$ the absolute error is bounded by the $n + 1$'st term denoted `bound` for all x in the printouts. The errors decrease when increasing n, which is consistent with that the approximation should be more accurate when using more terms in the sum. For $x = 3, 5$ and 10 when $n = 1$ and 2, the errors are larger than `bound`, which could potentially be avoided if using an argument reduction to a smaller interval than $[-\pi, \pi]$, e.g. $[-\pi/4, \pi/4]$.

Task 2: Adaptive Integration

In this task, it seemed appropriate to create a function pointer as an argument to the adaptive integration function, such that the integration can be performed on any kind of function $f(x)$. I followed the example in the course material how to define a function pointer using `typedef`:

```
typedef double (*FunctionPointer)(double);
```

This statement means that FunctionPointer is a pointer to a function that takes a parameter of type `double` and returns a `double`.

I defined a function called `simpsonRule` that evaluates $I(\alpha, \beta)$ given in the assignment, which is used when computing $I(\alpha, \beta)$ and $I_2(\alpha, \beta) = I(\alpha, \gamma) + I(\gamma, \beta)$. The output of the main program `asi.cpp` is the following:

```
Adaptive Simpson Integration of f(x) = 1 + sin(exp(3*x)):
 I           tolerance
 2.548323    1e-01
 2.505996    1e-02
 2.499857    1e-03
```

Using Matlab's integration function `integral` results in $I \approx 2.500809$. This means that we can improve the error of the approximation by lowering the tolerance and establish more subintervals where we perform Simpson's rule. When decreasing the tolerance to 10^{-8}, the output is the same as the answer from Matlab:

```
Adaptive Simpson Integration of f(x) = 1 + sin(exp(3*x)):
 I           tolerance
 2.500809    1e-07
```

[1] `https://www.csee.umbc.edu/~phatak/645/supl/Ng-ArgReduction.pdf`