Program construction in C++ for Scientific Computing

Ilian Häggmark
mail ilianh@kth.se

Andreas Karlsson
mail andreas.a.karlsson@ki.se

September 29, 2016

1 Project 1

1.1 Task 1 - Taylor Expansion with Horner's Scheme

The Taylor expansions for sine and cosines can be written as in equation 1, 2.

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

$$\cos(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!} \tag{2}$$

For an efficient implementation we can calculate the sums using Horner's scheme as described in equation 3, 4. This has the advantage that only `n` additions and `n` multiplications must be performed, compared with the $(\mathtt{n}^2+\mathtt{n})/2$ multiplications that needs to be performed in the original form. The calculation can therefor be carried out faster, and will also be more precise.

$$\mathrm{p}(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3 + \cdots + c_n x^n \tag{3}$$

$$\mathrm{p}(x) = c_0 + x(c_1 + x(c_2 + \ldots (c_{n-1} + x c_n))) \tag{4}$$

We calculated $(-1)^n$ using a modulus operation, which saved us a more expensive pow() call. After first calculating the factorial explicitly we were able to implement it as a multiplying factor within Horner's scheme.

sinus	N	x=-1	x=1	x=2	x=3	x=5	x=10
sin	1	0.158529	0.158529	1.0907	2.85888	5.95892	10.544
cos	1	0.459698	0.459698	1.41615	1.98999	0.716338	1.83907
sin	10	0	0	4.08562e-14	2.01152e-10	8.89053e-06	16.2678
cos	10	0	0	4.2738e-13	1.40571e-09	3.71704e-05	33.5995
sin	100	0	0	0	1.38778e-16	1.44329e-15	3.8658e-13
cos	100	0	0	5.55112e-17	2.22045e-16	3.38618e-15	1.66422e-13

Table 1: Absolute difference between the cmath implementation and Horner's scheme over a number of, x, angles and N terms for the Taylor expansion. 1.

We compared our implementation with the sin and cos functions implemented in the cmath header. The resulting absolute value of the difference between the implementations can be seen in table 1.

Based on the results in table 1 we conclude that for small values of x (e.g. one) fewer terms (e.g. ten) in the tailor expansion will produce a very precise approximation. However for larger values of x (e.g. 10) more terms are need.

Task 2 - Adaptive Simpson Integration (ASI)

The source code for task 2 contains two main functions, ASI and recursiveASI. The latter is more intuitive but left here only as a comparison.

The ASI funtion takes four parameters. A function pointer, that points to the function giving the integrand, two doubles one for the lower and one for the upper integration limit, and finally a double that sets the error tolerance. The Adaptive Simpson Integration is implemented using a while-loop that runs as long as the approximate error is bigger than the given tolerance. Inside the while is a for-loop that calculates the integral of an integrand f, from a to b by splitting the integration interval into n subintervals, calculating the subintervals with the integral approximation, i.e.

$$I(a,b) = \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right)$$

and then summing them up. After this the approximate error is calculated and an if-statement is checked to see if the error is small enough and the loop can be broken. If the error is too big the number of intervals, n is doubled and the loop starts anew.

The subintervals are calculated by another for-loop and stored in a vector:

$$\text{intervals} = [a, a+h, a+2h, a+3h, ..., b], \qquad h = \frac{b-a}{\text{n}}$$

where a is the lower limit of the integral, b is the upper limit, n is the number of subintervals and h is the subinterval length.

The result of the two ASI implementation is shown in the table below.

tol	ASI (recursive)	error	ASI (while-loop)	error
0.01	2.505996	0.005187	2.4995921	0.001217
0.001	2.499856	0.000952	2.5007683	0.0000408
0.0001	2.500809	0.0000005	2.5007683	0.0000408

Figure 1: Results and errors for three (3) tolerance values and two (2) ASI implementations, recursive and while-loop.

Using a recursive method is not recommended due to the large amount of calls. A notable difference between between ASI and recursiveASI, apart from the fact that the recursive is much easier to understand, is that ASI distributes its integration intervals equally, while the recursive puts a finer interval division on intervals that have high error (i.e. where the function changes rapidly). This allows the recursive method to give better result with equal amount of intervals. This more "intelligent" way of increasing the number of intervals could also be done in the while-loop implementation.