Physics G4080/G6080 – Problem Set 1

Spring 2015

Due Tuesday, February 10 in class

1. In class we discussed the errors that can arise from iterating the trigonometric identity

$$\cos[(m+1)x] = 2\cos(x)\cos(mx) - \cos[(m-1)x] \tag{1}$$

In particular, we found that the floating point errors would not be uniform in the value of x used. The errors in the algorithm from the accuracy of the initial value of $\cos x$, call it ϵ_c, will produce an error in the result of

$$\frac{N\sin(Nx)}{\sin(x)}\epsilon_c \tag{2}$$

where N is the desired limit for the recursion, *i.e.* we want to calculate $\cos(Nx)$.

On the Courseworks web page, you will find a MATLAB program, `cosine_recursion.m`, which implements the recursion relation. We discussed this program in class. It does the cosine recursion in the native double precision of MATLAB, but also does the recursion where the arithmetic is limited to P decimal digits of accuracy. In addition to limiting the precision, the arithemtic is also done with truncation, where higher order digits are dropped and not rounded.

   (a) Implement the cosine recursion formula in MATLAB or another language. You can start from scratch or the program `cosine_recursion.m`. For $x = 10^{-6}$, using the recursion relation to calculate $\cos(Nx)$ for $N = 10^2, 10^3, \ldots 10^8$. Plot your results versus N, plot Eq. (2) versus N and plot their ratio. Explain what you find.

   (b) (G6080 only) The formula given in Eq. (2) does not include the errors from finite precision representations of intermediate steps in the recursion. An intermediate step with a value of y has finite precision error of ϵy. The first intermediate step is $\cos 2x$, yielding an error of $\epsilon\cos 2x$. This error can be propagated through the recursion identically to the way we propagated the original error on $\cos x$ through the recursion. Find the formula for the propagation of all the intermediate finite precision representation errors errors - this will be a sum. You can evaluate the sum numerically (or analytically if you want). How does this correct Eq. (2)?

   (c) We can use different accuracy for the initial value of $\cos x$ and the steps in the recursion. `cosine_recursion.m` does trunction of the arithmetic steps in the recursion relation using in the `floor()` function in MATLAB, which rounds towards negative infinity. By only using `floor()` on positive numbers, the `trunc()` function truncates floating point numbers towards

0. Rounding of intermediate steps can be accomplished by the `round()` function in MATLAB. Note that this function only works on integers.

   Modify the program to test the following two cases:

   i. Use native double precision for the recursion steps, but reduced precision for the initial value of $\cos x$. The error in the result should be given precisely by Eq. (2).
   ii. Use high precision for the inital value of $\cos x$ but then much reduced precision for all of the internal arithmetic in the recursion. Plot the error on the results as a function of the number of recurrence steps.
   iii. (G6080 only) Do your results match your theoretical formula from part (b) above?

2. Not all floating point operations take the same time to complete. Generally floating point adds, subtracts and multiplies are done directly in hardware. Division is done via an algorithm, which frequently has some hardware support in the form of on-processor stored tables (stored in hardware) that give a starting point for the algorithm. Trigonometric functions and exponentials are generally software based algorithms and take even longer. In this problem, you can determine the time it takes for a few different types of floating point operations. We discussed an example C program, `real_time_clock.c`, in class and it is posted on the course web page and you can start from this to time various floating point operations.

   This program can be compiled on a Linux machine via

   ```
   gcc -o real_time_clock real_time_clock.c -lrt
   ```

   On a Mac, it can be compiled with

   ```
   gcc -o real_time_clock real_time_clock.c
   ```

   When timing an operation, it is usually good to average over a considerable number of them, to avoid the time delay in getting the first result through the pipeline of the processor. If you average over too many, they will not all fit into the processors cache and you will see performance fall, due to memory access times. Loops with hundreds to thousands of iterations should satisfy these constraints.

   You should find the time for the following operations

   (a) c[i] = a[i] * b[i]
   (b) c[i] += a[i] * b[i]
   (c) c[i] += a[i] * b[2*i + 20]
   (d) c[i] = a[i] / b[i]
   (e) c[i] += a[i] / b[i]

(f) c[i] = sin(a[i])

(g) c[i] = exp(a[i])

(h) c[i] = sqrt(a[i]) (Make sure a[i] is positive.)

Determine the times with maximum compiler optimizations and with all optimizations off. For gcc -O3 (capital letter O followed by the numeral 3) selects the maximum optimization and -O0 (capital letter O followed by the numeral 0) turns off all optimization.

You should use MATLAB, or another tool, to plot your results and to help you determine the cost for each operation, as distinguished from the overhead to start and stop the clock.

3. The back recursion technique gives a convenient way to calculate Bessel functions and modified Bessel functions. Write a program to calculate $I_2(x)$ for $x = 1, 2, 3 \ldots 10$, using back recursion. How many back recursion steps are required to have 4 decimal places of accuracy for $I_2(x)$ for this range of x? How many steps are required for 8 decimal places of accuracy (here the back recursion should be done in double precision)?

   Make plots of the precision of the results for $I_2(x)$ versus the number of back recursion steps. Although you may use any language you like, I recommend that you do this in MATLAB, to become more experienced with this tool. The `cosine_recursion.m` function from Problem 1 can be used as a starting point.