Session 2 activities worksheet

Due: January 17 *Regrades until:* February 14

Online handouts: gnuplot mini-manual; gnuplot example; listings of several codes.

Your goals (in order of priority; you will finish some in PS#1):

- Look at a round-off error demonstration with quadratic equations
- Discuss a special-case problem to help understand summing in different orders
- Debug sample codes; make some plots with gnuplot
- Assess different methods of calculating spherical Bessel functions, including with GSL

Please work in pairs and briefly answer questions asked here on this sheet. The instructors will bounce around the room and answer questions (don't be shy to ask about anything!).

You should go to your working directory, download session02.zip from Carmen and unzip it in your directory just as you did for the first session.

A) Round-Off Errors with Quadratic Equations

This section assumes you have read the background notes for Session 2, which describe round-off errors and the quadratic equation.

1. Review the discussion about round-off errors and the quadratic equation in the Session 2 notes with your partner. Make sure you go over the pseudo-code for the test case and understand it.
2. Look at the code quadratic_equation_1a.cpp in an editor. It represents the pseudo-code as it has just been typed in, mistakes and all. Note that it is not indented consistently. *Why might we care about formatting?*

   → We care about formatting since it makes our code cleaner and easier to read. This allows us to debug a little better and allows other to read our code better as well. It also helps reduce errors through consistency.

3. Now try compiling with "make -f make_quadratic_equation_1a". You should get a bunch of warnings and errors. Each one comes with the line number of the error. Take each one in turn and write below what you think the error is (you'll need to identify the line numbers in your editor; ask an instructor if you don't know how to turn them on). Each warning is a clue to a mistake in the code (such as a typo)! [One hint: "setprecision" is defined in the header file "iomanip".] If you get stuck, quadratic_equation_1.cpp is the corrected version (without the "a" after the "1"). *List the errors here:*

→

1. The first error comes from defining the "main" function without the return type specified, which should be "int main".

2. The second error has to do with a typo. The writer typed "out" when it should be "cout".

3. "disc" was initialized without a type, so it should have been "float disc" or "double disc".

4. "setprecision" is in the "iomanip" header, so that needed to be included via "#include <iomanip>".

5/6. The last two errors are warnings that state that "x1p" and "x2p" are not actually used after declaring them. Since warnings are being treated as errors, we fix this by updating the incorrect logic in the code and update the second print statement to output "x1p" and "x2p" instead of "x1" and "x2".

4. Correct these errors and see that it compiles and runs. Try a=1, b=2, c=0.1 and see that the roots differ. *Which two (of x1, x1p, x2, x2p) are the most accurate? Explain why the others are not as accurate.* (Hint: check the Session 2 notes.)

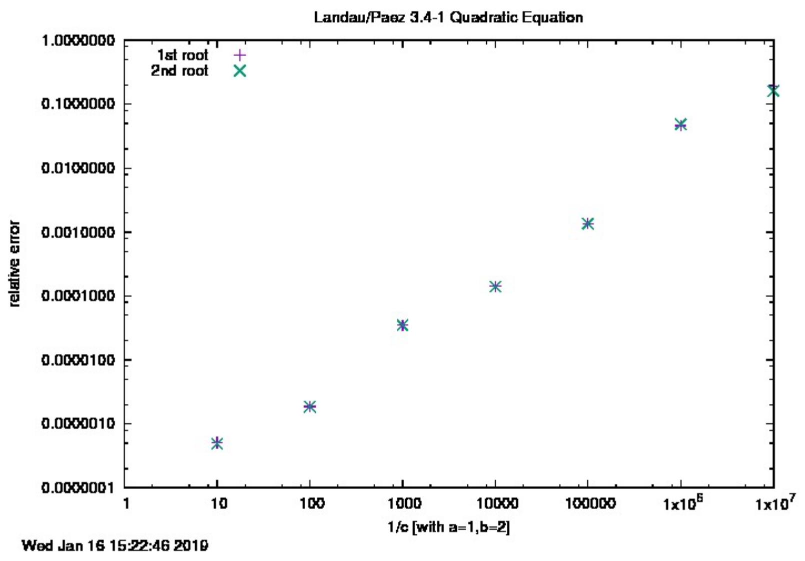
   → From the Session 2 notes, this is the case for b>0, so we should use "xgs1p" and "x2" as the most accurate results of the roots. This has to do with subtractive cancellation, where with "x1" and "x2p" the numerator and/or denominator can cancel out to 0 by the cancellation given via the quadratic formula, thus causing these results to be less accurate and they can possibly have "bad" results.

5. Now look at quadratic_equation_2.cpp, which is the second pass at the code. Check the comments for a description of changes. List below up to three C++ lines from the code that you don't understand completely (it may be most of them or it may be none!):

   → The math and looping I understand, assuming I'm following the math properly. The only part I was a little confused about are the lines involving "fplot". I've used ofstreams before but not with a .dat file. I understand roughly what it's doing, but I was confused if there really was any difference between using ofstream with a .dat file and using ofstream with another file type. It seems like to me that it doesn't matter.

6. Compile and run quadratic_equation_2.cpp, and enter a=1, b=2, c=.1 again. The output is now more detailed and an output file named "quadratic_eq.dat" is created. We'll discuss outputting to a file more later; for now, this is an example you can follow to do it on your own (there were also examples from the Session 1 codes).

7. Follow the "Plotting Data from a File with Gnuplot" handout to duplicate this plot.



Try some of the extra commands, including output to a postscript file. Make a note on the worksheet with the name of the file and include it with your submitted session at the end.
→ Name of the file is “quadratic_eq.ps”

8. *Is the plot qualitatively and quantitatively consistent with the analysis in the notes? Explain in a few sentences. [Hints: What does a straight line mean? What does the slope tell you?]*

→ Yes, this plot is consistent with the analysis in the notes. The straight line implies that the error increases linearly as 1/c increases, which is exactly what we see through equation 2.18 in the notes. Based on the log-log plot, we would anticipate that the slope roughly equals 1 (which very roughly approximates machine precision), and that an extrapolation to c=1 should get closer to 0, which appears to be true for this plot. The slope found isn’t exactly the machine precision (or 2*machine precision) we expect, but it is roughly in the ballpark of what we expect.

B) Summing in Different Orders (see Session 2 notes)

Here we consider another example of subtractive cancellation and how computer math is not the same as formal math. Imagine we need to sum the numbers 1/n from n=1 to n=N, where N is a large number. We're interested in whether it makes any difference whether we sum this as:

1 + 1/2 + 1/3 + ... + 1/N (summing up)

or as:

1/N + 1/(N-1) + ... + 1/2 + 1 (summing down)
In formal mathematics, of course, the answer are identical, but not on a computer!

To help think about the sum up vs. sum down problem, it's useful to think first about a simpler version. Suppose you want to numerically add 10^7 small numbers a = 5×10^{-8} to 1 (so the exact answer is 1.5). For the first three questions, *predict* the answer before running the code.

1. *If you add 1 + a + a + ... in single precision, what do you expect to get for the total?* (Hint: recall the discussion of "machine precision.")

   → For single precision, I believe the machine precision is larger than 5e-8. Therefore, for adding 1+a the first time should yield 1, and you should iteratively yield 1 repeatedly, so adding by this method should just return 1.

2. *If, instead, you add a + a + ... + 1 in single precision, do you expect a different answer? Which will be closer to the exact answer?*

   → By this method, we can keep iteratively adding a, even though a is smaller than machine precision, we should be able to get an answer as we will add a repeatedly. Then when we add 1, the earlier added values should be above machine precision, so we can safely add 1 to it and get closer to the exact answer of 1.5.

3. *If you repeat the exercise in double precision, what do you expect will happen?*

   → In double precision, both methods should work as 5e-8 is much larger than the machine precision required of a double, so both should yield effectively the same answer. We expect the double precision to be more accurate than the single precision result too.

4. Write a brief "pseudocode" here for a program that would compare these two ways of summing. Then look at the file order_of_summation1a.cpp in an editor; *does it look like an implementation of your pseudocode?*

```
SET N = 10,000,000
SET a = 5.0e-8
// Single precision
SET sum_up_float = 1.0 (as float)
SET sum_down_float = 0.0 (as float)
FOR i = 1 to N:
        sum_up_float = sum_up_float + a
        sum_down_float = sum_down_float + a
sum_down_float = sum_down_float + 1.0
// Double precision
SET sum_up_double = 1.0 (as double)
SET sum_down_double = 0.0 (as double)
FOR j = 1 to N:
        sum_up_double = sum_up_double + a
        sum_down_double = sum_down_double + a
sum_down_double = sum_down_double + 1.0
// Output
PRINT "SINGLE PRECISION:"
PRINT "Summing Up: ", sum_up_float // Likely 1.0
PRINT "Summing Down: ", sum_down_float // Likely 1.5
PRINT "DOUBLE PRECISION:"
PRINT "Summing Up: ", sum_up_double // Likely 1.5
PRINT "Summing Down: ", sum_down_double // Likely 1.5
```

→ This implementation is like the actual code (variable names aside). The only difference is that the code file did it all in one loop, which would be better and is possible to get to that result from my pseudocode.

5. In a terminal window, use the makefile make_order_of_summation1a to try and compile it ("make -f make_order_of_summation1a"). Identify and fix the errors, until it compiles and runs correctly (verify the results against the comments in the file). Correct your answers to 1,2,3 if necessary (and understand them!).
   → Answers are good! I don't know if that was supposed to be a purposeful error, but I think I found an issue with the double precision result, which was added incorrectly, which I fixed.
6. Run indent on the file ("indent order_of_summation1a.cpp") and check that it is now nicely formatted.
7. *Challenge: Predict the results for 10^8 additions (instead of 10^7):*

   → For adding 1e8 to 1, our anticipated result with our a value should be 6. We predict that for single-precision, adding 1+a should get us 1.0 while the other method should get us 2.0. This is because for single-precision, the values will be added until we get to 1.0. Once we get to 1.0, the machine precision increases (meaning less precise), such that the value of a is smaller than the machine precision. Thus, every addition afterwards still returns 1.0, and then at the end we add 1.0 once again to get 2.0. For both double-precision results, we should get close to 6.0.

Now change the code, and see if your prediction is correct (or that you can explain it after the fact).

→ Our prediction matches.

C) Bessel 1: Making Another Plot with Gnuplot

The next three tasks (Bessel 1-3) build on the discussion of spherical Bessel functions in the Session 2 notes. Skim this discussion before going further.

1. Copy make_area to make_bessel and open make_bessel in an editor. (If you are in the session_02 subdirectory, you can use "cp ../session_01/make_area ." to create a new copy where you are or "cp ../session_01/make_area make_bessel" to make a copy directly; do you understand what all the "."'s mean? If not, ask!) Follow the instructions in the makefile to convert it to make_bessel. Run the makefile using "make -f make_bessel".
2. If you run bessel.cpp, you'll generate a data file "bessel.dat". By looking at the code, figure out what the columns in the file mean. *What Bessel function is being output? What are the columns?*

   → From the code, it looks like a 10th order spherical Bessel function. The columns are the x-value, the value of the function from the down recursion method, and the value of the function from the up recursion method, as defined in the C++ file.

3. Make a plot of the third column as a function of the first column using gnuplot. (Use the example in the handout as a guide. You might also find the "GNUPLOT Manual and Tutorial" useful.) Try plotting BOTH the second and third columns vs. the first column on the same plot (i.e., two curves). *Include your plot in the submission and note the filename here.*

   → Filename is "bessel.ps", included in the zip file.

D) Bessel 2: Error Assessment (complete in PS#1)

1. Modify the output statements in bessel.cpp to increase the number of digits in the output (look back at previous codes for clues how to do this).
2. Modify the code bessel.cpp (give it another name) to compare upward and downward recursion methods by adding a new column to the output that calculates the *relative* difference between the results for each x. We'll define the relative difference of a and b as |a-b|/(|a|+|b|). Google to find out how to take an absolute value in C++.

3. Make an *appropriate* plot of the error vs. x and interpret the different regions of the graph. Try with and without log scales.

   →Using log scale on the plot allowed us to see the results better. From this, we can see that the relative error is the worst (most disagreement between the cases) for small x (x<1) and larger x (x>30). This means there are three clear regions of the graph. The first region for x<1 is a high error regime, likely due to the initial conditions of the recursive method, as the function tries to approximate the Bessel function. Then there is an accurate regime between x=1 and x=30 where the relative error gets smaller than machine precision for float. This doesn't mean necessarily that the function result is correct, but that both methods agree. Then as x continues to increase past 30, the error increases once again, meaning the two methods disagree once again. The plot is "relative_error.ps" and is included in the zip file.

E) Bessel 3: Using the GSL Scientific Library (complete in PS#1)

Make sure you have completed the GSL task from Session 1 before doing this.

1. Modify bessel.cpp to also calculate the spherical Bessel function using the GSL routine gsl_sf_bessel_jl for comparison. The GSL documentation gives the usage of this function as:
   Function: double **gsl_sf_bessel_jl** (*int l, double x*)
   "This routine computes the regular spherical Bessel function of order l, j_l(x), for l >= 0 and x >= 0."
   How does the accuracy of the downward recursion routine in bessel.cpp compare to that of the GSL function? (How can you tell?)

   → The accuracy of the downward recursion routine is best for x<30. We can tell by making a log scale plot of the relative error like we did in (D), assuming that the GSL library is returning accurate results. In doing so, we see that the relative error is less than float machine precision for x<30. However, as x increases past 30, the error increases, meaning that the downward recursion method does a worse job at being accurate for these larger x values. The plot made is called "bessel_gsl.ps" and is included in the zip file.

F) Reflection

Write down which of the six learning goals on the syllabus this worksheet addressed for you and how:

→ This session helped me learn more about the tools of computational physics, including machine precision calculations, the use of GSL, and the use of gnuplot. Through the error analysis in the notes and the computer codes verifying that with the quadratic formula, I now also better understand the limitations of computational solutions and numerical approaches in general.