grade: check - (upgrade by revising the questions w/blue comments)

cd /cygdrive/c/users/'Evan Zobel.000'/Desktop/phy480

PHY480/905: Activities 2 Evan Zobel

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

Your goals (in order of priority):

- If you have not turned in your Activity 1 worksheet, please do so ASAP via GitHub (ask if you need help setting up GitHub and adding me as a collaborator!)
- 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

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 (see Activities 1 for creating one), download session02.zip from the course homepage and unzip it in your directory.

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-- discuss any questions you have w/your breakout room colleagues and/or the instructors. 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?*
    - Proper indenting and formatting make the code MUCH more readable. It can be hard to tell where errors are when even the working code is formatted improperly. Keeping things consistent helps.

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 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*:

- error: ‘out’ was not declared in this scope

```
43 | out << "a = " << a << ", b = " << b << ", c = " << c;
   | ^~~
```

- error: ‘disc’ was not declared in this scope

```
45 | disc = pow (b * b - 4. * a * c, 0.5); // definition of discriminant
   | ^~~~
```

- error: ‘setprecision’ was not declared in this scope

```
53 | cout << fixed << setprecision (16) << x1 << "    " << x2;
   |                  ^~~~~~~~~~~~
```

- error: unused variable ‘x1p’ [-Werror=unused-variable]

```
48 | float x1p = -2. * c / (b + disc);     // first root, new formula
   |       ^~~
```

- error: unused variable ‘x2p’ [-Werror=unused-variable]

```
50 | float x2p = (-2. * c) / (b - disc);   // second root, new formula
   |       ^~~
```

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 pages 5-7!)
   - X1p and X2p are more accurate. As you can see from equation 2.15, 2.16, 2.19 and 2.20, the x1 and x2 formula is a worse version of the formula, with more relative error. This stems from how it calculates the square root.

Incorrect-- one of these is right, the other is wrong.

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!):
   - Line 64: float disc = pow (b * b - 4. * a * c, 0.5); // define discriminant
   - Line 80: float x1_best = 0., x1_worst= 0., x2_best= 0., x2_worst= 0.;

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 the plot on the back. Try some of the extra commands, including output to a postscript file and opening it using an appropriate postscript viewer.
   - Note: when I try to do this, I get the following error about X11
   - gnuplot: unable to open display ''
   - gnuplot: X11 aborted.
   - gnuplot>

   Please ask in class. These types of errors are *usually* pretty easy to correct

   I'm unsure what it means, and wasn't able to find much online beyond the need to "configure x11". The rest of this section of the assignment is still doable however, So I will continue.

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, the plot is qualitatively and quantitatively consistent. We're using a log scale, and so the straight line we see is actually a quadratic.

Not sure i understand. How does the log-log plot verify (or not verify) the analysis in the notes? I.e., how should the errors scale with 1/c?

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 answers 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 = $5x10^{-8}$ 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? What do you get when you carry it out?* (Hint: recall the discussion of "machine precision.")
    - We would expect to get 1.000
    - When carried out, we get this answer

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, the first way of summing or the 2nd way?*
    - We would expect to get 1.53238
    - Backwards will be closer to the exact answer

(how did you expect such a precise value? Do you mean you expect something near 1.5?)

3. *If you repeat the exercise in double precision, what do you expect will happen? Now actually perform the double precision calculation. Does it agree with your expectations?*
    - We would expect to get 1.5. We instead get 1.4999999999999999 repeating, which is basically 1.5
    - This does agree with expectations

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?*
    - I'm not totally sure what to do here. Looking through summation1a I can make sense of it, but I don't currently feel comfortable enough to have done that myself. The pseudocode in summation1a makes sense as a way of performing double precision.

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!).
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):*

   Missing...

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

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?*
   - The bessel function being outputted is for upwards and downwards recursion. Thus, it is a double precision function. The three columns are radius, theta, and phi in spherical coordinates.

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). *Print out your plot.*
   - Unfortunately, X11 is still causing problems, and my plot is not properly printing.

Let's work this out in class or office hours.