Session 3 activities worksheet

Due: January 19 *Regrades until:* February 16

Online handouts: Gnuplot fitting example; C++ formatting; listings of codes

Your goals for today and next time (in order of priority):

- Finish some leftover tasks from Session 2
- Look at a quick demo about comparing floating-point numbers
- Use a makefile to compile a project with multiple .cpp files and a header (.h) file
- Duplicate the figure in the notes 3d with a log-log plot, then fit the slopes with Gnuplot
- Practice coding an algorithm: Add a new subroutine with the 3/8 rule and analyze [PS#2]
- Add an appropriate integration routine from the GSL and analyze [PS#2] the error

Please discuss and compare answers with others. The instructors will bounce around the room to ask and answer questions.

You should make or change to a 5810 sub-directory, download session03.zip from Carmen and unzip it.

A) Comparing Floating Point Numbers

A common task in a computational problem is to check whether two floating point numbers are the same. In C++, the comparison operator is == (not just one =), so it might seem that a simple if statement would do the job. Here's an example of why this fails in general:

1. Look at the file number_comparison.cpp in an editor. *What does it do?*

   → The file compares two numbers. The float value 1.0 and the division of cos(pi/4)/sin(pi/4) using a pre-defined pi and uses and if statement to print if the two floats are equal or not.

2. Compile, link, and run the program (there is a makefile). *Why doesn't it give the answer you want?* Add statements to print out x1 and x2 to check your response. *What did you add?*

   → It doesn't give us the answer we want because the actual calculation of x1 is not exactly equal to x2. If you print out x1 and x2 with more precision using "setprecision", we see that the x2 is 1.0, but x1 is not exactly 1.0, it is 0.9999 (etc. more digits), so when checking for equality, we get that the floats are *not* equal.

3. *Suggest a better way to compare two numbers based on the idea that the relative inaccuracy of any number can be as large as a specified precision* eps *(which may be greater than the machine precision).*

→ A better way would be normalizing the relative size of the numbers and checking if they are less than some epsilon precision. So, this would be something along the lines of abs(x-y) / (max(abs(x), abs(y))), and checking if that is <= epsilon, for some small epsilon. That way relative inaccuracies from machine precision can be accounted for.

B) Numerical Derivatives: Pass 1

The Session 3 notes have a short introduction to numerical differentiation (see the Hjorth-Jensen notes online for more detail). These are among the simplest algorithms for us to derive and to verify the theoretical approximation and round-off errors (the errors for most real-world algorithms are noisy).

1. Look at the file derivative_test_simple.cpp while reviewing the discussion in the notes. (Also look at the last section in the notes, which describes pointers to functions, which are used here.) *What part(s) of the code do you not understand? What is being printed?*

   →One thing that confused me with this code is that FLOAT is all of a sudden capitalized. I was confused if this is just a normal, allowable thing in C++, or if something else is going on. I understand the syntax and the forward_diff and central_diff function based off the notes, and how the test functions relate. Nothing gets printed to the screen, but the different relative errors like log10(h) are outputted to a .dat file.

2. Use the makefile to compile, link and run the code, generating the file derivative_test_simple.dat. Add a statement to print a header line to the output file with a label for each column (and start the line with # so that it doesn't screw up gnuplot). *Show an instructor your output file.*

   → Done.

3. Make a graph with gnuplot with two plots: the logarithm of the relative error for forward difference vs. the logarithm of h (which is Delta-x) and the analogous plot for central difference. *Print the graph into a file and note the file name here.*

   → The file is "relative_error.pdf" in the zip.

4. *Are the slopes in each region consistent with the analysis of errors in the notes? Which is the better algorithm? Explain. What value of h is optimal for each algorithm? If you switched to single precision, would the slopes of the lines change? What would the graph look like?*

   → The slopes in the region are consistent with the analysis in the notes, getting the optimal h values of about 10^-8 for forward difference and 10^-5 for central difference, which is very close to the values predicted by the notes. The better algorithm is the central difference algorithm, because it has higher accuracy (due to higher order), and it has a better error floor, reaching a lower relative error compared to the forward difference. If we switched to single precision, the slopes of the lines shouldn't change, but the graph would change by having a larger relative error. So, the graph would shift up, and it would shift to the right, as the machine precision is also increasing.

C) Makefiles for multiple project files (including header file)

Many of the example programs started as "all-in-one" C programs from the Landau/Paez text. One of these was integ.c, which we converted to C++ and then split up into:

- integ_test.cpp, which has the main program and the function to be integrated
- integ_routines.cpp, which has the integration functions themselves
- integ_routines.h, which has function prototypes (a prototype tells the compiler the function return type and the type of all its arguments)

There is also a function in gauss.cpp and there is make_integ_test to compile it all. In a subsequent step, we modified the codes so that the function is passed as an argument to the integration functions.

The idea is that the integration routines should be isolated in a file by themselves. The main program just invokes these routines. The header file conveys the prototype information about the integration routines to the main program and to any other functions that might call the routines. (Note: Later we'll consider going further and defining an integration *class*.)

1. Compare the trapezoid_rule and Simpsons_rule functions in integ_routines.cpp to equations (3.15)-(3.19) and the table in the Session 3 class notes. *Can you see how the algorithms are implemented?*
   What is the advantage of having the routines in a separate file from the main program if you want to test that they work on known integrals or if you later improve the algorithm?

   → I can see how the algorithms are implemented, using an interval and a sum, and looping over some number of points and adding to the sum. The two methods are different. The advantage of having a separate file allows for better testing and validation (like writing a dedicated test suite), it allows you to improve your algorithms better and helps usability of a larger code base. It especially helps for later improvements by having one separate file instead of one large file you must dig through.

2. Create the executable integ_test using the makefile make_integ_test and run it to generate the integ.dat output file. *Does the output file makes sense?*

   → The output file makes sense. It outputs the N, and then the different integral sums. Since Simpson requires odd intervals, the output file only outputs odd N.

3. Change integ_test.cpp to output *relative* (rather than absolute) errors. Note that only the files that have changed since the last compilation are recompiled each time!
4. Use Gnuplot to reproduce the figure in Section 3e of the notes. *Briefly explain what you can learn from the plot (remember slopes!).* Consider **all** regions of the graphs.

→ You learn a lot from this plot. The slopes roughly tell us the order of the function, showing that the trapezoid rule is the lowest order, then Simpson's rule, and then Gaussian. If we fit the slopes, we can find the true order. For the methods, we see as N increases, they start to reach their relative error, but as N becomes very large (especially clear for the region log10(N)>2), we start seeing fluctuations, and the error stops declining. This comes from precision limits, going past the optimal N, and this is a region where rounding dominates. The plot is "graph_2e.pdf".

5. *Bonus:* Change the loop in integ_test.cpp so that the points on the log-log plot are evenly spaced. *What did you change?*

   → For this, I defined an upper and lower range for log(N) and the number of points we want. Then we loop over k points and calculate log(N) using the min and max log(N) and convert that into an integer i we can use for the different rules (trapezoid, Simpson, etc.). We also add a check for the Simpson rule, as it needs odd i, and we break the loop if we go over the max number of intervals.

D) Finding the Approximation Error From a Log-Log Plot

From the plot in the previous section, we can estimate the approximation errors by eye. Now we want to actually fit lines to find the slope using Gnuplot. Use the handout as a guide.

1. Modify the code so that it outputs the logarithm base 10 (log10) of N and the relative errors.
2. *What are the slopes of the trapezoid and Simpson's rule plots in the regions where they are linear?*

   → Using fits, the slope of the trapezoid region is about -2.10 and the Simpson region is -4.73. The plot is "graph_2e_D.pdf".

3. *Are the slopes consistent with the analysis in the text? Now try to fit the round-off error region and interpret the "slope".*

   → The slopes are relatively consistent from the text. We would expect slopes of -2 and -4 for trapezoid and Simpson, respectively. We get close for the trapezoid region. The Simpson slope is a little further off but is likely due to the accuracy of our fit. If we fit the round-off error region for the Simpson fit, we get a "slope" of about 0.5, which implies that the errors fit a random walk model. For the trapezoid area, we still get a slope around -2.8, which is likely due to the convergence of the trapezoid region not being done for the given range, and the fact that we don't have a lot of round-off error region for that fit compared to the Simpson plot.

E) Extra: Coding an Algorithm (complete in PS#2)

This is primarily practice converting an algorithm to working code. Your goal is to add a new function to integ_routines.cpp, called three_eighth_rule, which implements the 3/8 rule from the table in Section 3d of the notes.

1. From the rule and the discussion in the text, write some pseudocode to explain how you would integrate a function with this rule.

   →
   Calculate h: by dividing the total range (xmax – xmin) by the number of intervals (num_pts-1)
   Initialize sum: and start with endpoint contributions of f(xmin) + f(xmax)
   Loop through internal points: utilize a weight of 2 if the dummy index "i" is a multiple of 3, otherwise the weight is 3 (because we are going to factor out "3h/8").
   Multiply: the total sum by 3h/8.

2. Implement your pseudocode in C++ by adding a new routine (called three_eighth_rule) to integ_routines.cpp (be sure to add a prototype to integ_routines.h).

   → Done

3. Modify integ_test.cpp to output to a new file the results from the new routine, and add them to the previous plot. Explain the approximation error. [Warning: If you don't change the way that integ_test.cpp loops through the number of intervals, you will likely run into a subtle error that prevents you from getting the approximation predicted theoretically!]

   → The new plot is "graph_2e_E.pdf". We can see (and calculate) the slope, seeing that it is roughly parallel to the Simpson line, which makes sense since both of their errors should go as 1/N^4.

F) Extra: GSL Scientific Library Yet Again (complete in PS#2)

1. Go to the web page with the GSL manual (it is linked from the 5810 Carmen page). Find an appropriate integration routine for the test integral we've been working on. *Which one did you choose, and why?*

   → I chose "gsl_integration_qag" since it works to adaptively integrate f(x) over a finite range (a,b) with relative and absolute error limits, which is appropriate for our task.

2. Add another calculation to integ_test.cpp (with output to a file) using this GSL routine.
3. Compare the accuracy of the GSL routine to the others on an error plot.

   → The graph is "graph_F.pdf". The accuracy is much better and lower than the other routines. This has to do with the fact that the QAG routine is adaptive step and uses N to divide into N pieces. The fact that it is flat shows that even with low partitions, the function is strong enough to converge to a low error solution (close to double machine precision), which is also likely due to the well-defined integrand of e^-x.

G) Reflection

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

This worksheet addressed many of the learning goals. It taught me how to write correct, clear code, as I had to write a lot of code in this worksheet compared to the previous two sessions. I learned many more tools of numerical integration and differentiation for computational physics. With slope and error analysis, I also learned more about the limitations and behavior of computational physics and computation in general.