480/905: Session 3 (2/2/21-2/4/21)

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

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

* Look at a quick demo about comparing floating-point numbers
* make plots illustrating a) truncation errors and b) roundoff errors in numerical derivatives
* Use a makefile to compile a project with multiple .cpp files and a header (.h) file
* make plots illustrating a) truncation errors and b) roundoff errors in numerical integration
* Duplicate the figure in the notes 3d with a log-log plot, then fit the slopes with Gnuplot

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

You should change to your PHY480 sub-directory, download session03.zip from the course homepage and unzip it.

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?*

**Compare if two float values are equal or not**

1. 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?*  
     
   **Initially the code did not give me a correct answer because there was a period after every value.**

**I added “cout” to print x1 and x2**

1. *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).*  
   * **provide user input**
   * **Add “setprecision( )” to the code**

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?*

**Relatively, I understand all of the code**

**It print an output file called derivative\_test\_simple.dat**

1. Use the makefile to compile, link and run the code, generating the file derivative\_test\_simple.dat. There is a missing #include statement you need to add (compare with previous codes). 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).
2. 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. *(Make sure you include a pdf, png, or ps of your plots when you turn in the activity sheet.)*
3. *Are the slopes in each region consistent with the analysis of errors in the ch3 notes? Which is the better algorithm and why?*  
     
   **The slope are consistent with the analysis of errors in the ch3 notes and the forward difference is the better algorithm because it has a higher h value than the central difference**
4. *In the ch 3 notes, we discuss how one can estimate a-priori the optimal choice for h by setting the truncation error equal to the roundoff error and solving for h (see eqs. 3.9 and 3.13.) Plug in #'s to find the optimal h values for the two algorithms. Does these theoretical estimates agree with what your graphs are telling you?*

**From graph:**

**Forward difference = h = 10^-7.94088**

**Central difference = h = 10^-4.91338994**

**It theoretically does not agree with the graph because unless the second derivative happens to vanish, the error should be proportional to h**

1. *If you switched to single precision, would the slopes of the lines in your plots change? What would the graph look like?*

**The graph would have the optimal h value shifted to the right of the x-axis because the machine precision was altered**

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?*

**Yes I can see how the algorithms are implemented**

**The routines sort of act like a built in function that is imported to the cpp file**

1. 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?*

**Yes**

1. 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!
2. Use Gnuplot to reproduce the figure in Section 3d of the notes. *Briefly explain what you can learn from the plot. I.e., are there different "regions" where the slopes are qualitatively different? If so, what does each region signify?*  
     
     
   **From the plot, I learnt that there are different optimal values for the relative errors.**

**There are different regions in the graph where the slopes are qualitatively different**

**Each region signify elementary weights for subintervals in the table**

**Slopes:**

* **Trapezoid: -0.00488086**
* **Simpson’s: -0.010614**
* **Gauss: 8.13523e-08**

**The values of the slopes does make sense**

1. Change the loop in integ\_test.cpp so that the points on the log-log plot are evenly spaced. *What did you change?*

**I changed += 2 in the loop to \*=1.5**

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 on fitting in Gnuplot 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?*

**m for trapezoid = -2.08034  
m for Simpson’s = -1.25481**

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

**I established the round-off to be from x-values 0 to 2.**

**m for trapezoid = -2.20605  
m for Simpson’s = -0.263007**