480/905: Activities Week 4 Feb 2021

Grade: check + (can you include the figures in the word document next time?)

*Online handouts:* GSL eigensystems documentation; singular integrals eigen\_test.cpp and derivative\_test.cpp, pointer\_test.cpp, and qags\_test listings.

*Your (always optimistic!) goals:*

* Run and deconstruct a code comparing numerical derivative methods, including Richardson extrapolation.
* Modify the derivative code to apply to a different function with more than one passed parameters (structures and pointers!).
* Run a sample program to find eigenvalues and eigenvectors of a matrix and determine how the running time scales with the size of the matrix.

Please work in small groups The instructors will bounce around and answer questions.

Numerical Derivatives and Richardson Extrapolation

Take a look at the derivative\_test.cpp handout. The derivative\_test.cpp code evaluates the numerical derivative of a defined function (funct) four ways: forward derivative, central derivative, extrapolated central derivative, and with a GSL routine. The function is defined according to the GSL conventions, which is a generalization of the derivative\_test\_simple.cpp code from Session 3. It uses "void pointers" to pass extra parameters (like alpha) to the functions.

1. Compile and link derivative\_test.cpp (using make\_derivative\_test). Run it to generate derivative\_test.dat and then use derivative\_test.plt in gnuplot (see the handout on using a plot file) to verify the figure on the back of the handout. You'll have to look at the code to identify the columns in derivative\_test.dat. Add code to print column headings.
2. Edit the plot file to add the corresponding plots and fits for the central derivative and extrapolated central derivative approximations. *Sketch the plot here and also generate a ps,png or pdf.*
3. In Session 3, you saw the functions forward\_diff and central\_diff (notice that the function name is passed as a pointer). *Explain the slope of the extrap\_diff graph (see the discussion in the Session 4 notes).*

**It has a linear slope with 1 as the slope and 0.3 as the intercept**  
  
  
  
  
This method is an example of "Richardson extrapolation," in which you use calculations at two different mesh sizes to derive a much better estimate than either one individually. *Describe how you would get an even higher-order result.*  
  
**Modify the functions based on higher order term in the Taylor expansion of the Richardson extrapolation**

1. *What is the source of error on the left side of the graph (smaller mesh sizes)? Why are the slopes the same?*

**Might be precision or round-off error**

Pointer Games

This exercise is practice in writing or modifying code based on examples. Take a look at the pointer\_test.cpp handout. It gives examples of how to pass several types of variables to a function using the void pointer named params\_ptr. In derivative\_test.cpp, a double named alpha is passed to the function test\_function. Your job is to modify the code so that **two** variables, alpha and beta, are passed to the function alpha\*xbeta.

1. Start by defining a structure with the two parameters alpha and beta (see pointer\_test.cpp for an example).
2. Modify test\_function and test\_function\_derivative for alpha\*xbeta and its derivative, getting alpha and beta from the passed params\_ptr (again, see pointer\_test.cpp for an example in f\_osu\_parameters).
3. Modify the main program in derivative\_test.cpp to load alpha and beta into your structure (see the main code in pointer\_test.cpp for an example). Simply choose values for alpha and beta (2. and 3./2., for example).
4. *Test the numerical derivatives at x=2. Use the gnuplot plot file with small modifications generate a plot. Sketch what you get.*
5. *Did you find the same slopes with your new function? Why are the slopes the same but the intercepts different?*

**In my new function, the value of the slope is still 1.0 although this time it’s going to a negative trajectory. The intercept is different due to the passed parameters that we had to implement in the functions.**

Linear Algebra with GSL Routines

The GSL library has many functions defined to set up and manipulate vectors and matrices. To do so, it defines various structures such as "gsl\_matrix" and "gsl\_vector", and functions such as "gsl\_matrix\_set" to set the value of an element in the matrix. It is all a bit intimidating at first, so we'll take a look at a basic example to see the general set-up. In particular, the program in eigen\_test.cpp creates a Hilbert matrix (as described in the Session 4 background notes) of user-specified dimension and then calls a routine to find its eigenvalues and eigenvectors.

An important issue with numerical computations, and linear algebra in particular, is how the computation time scales with the size of the problem. The program in eigen\_test.cpp includes two calls to the "clock" function, before and after the eigenvalue routine, to time how long the routine takes to run. Your task is to figure out how the time scales with the size of the matrix (e.g., does it go like a power of the dimension of the matrix?).

1. The session 4 zip file from the webpage should contain eigen\_test.cpp and make\_eigen\_test. Compile and link eigen\_test using make\_eigen\_test.
2. You should always verify with test cases that a program is working. Figure out how to use Python to find the eigenvalues of a 4 by 4 Hilbert matrix (Hint: Google is your friend.).

**Using the SciPy library and I turn in eigen\_test.py**

1. Run eigen\_test with a dimension 4 Hilbert matrix (i.e., 4x4) and compare the answers to the ones given by Python. Try to trace through the code on the printout to identify what the different GSL function calls do (you are **NOT** expected to understand the calls in detail at this point!).

**The answers were similar but not exactly the same because I used .rand in python to generate the matrix**

1. Edit eigen\_test.cpp and comment out the section that prints to the screen, so that the only output is the time the routine takes to run.
2. Can you make a theoretic guess of how the execution time scales with the size n of the matrix? E.g., is it a power law? If so, what is the exponent? Explain your reasoning.  
     
   **It could be due to matrix multiplication rules**
3. *Now figure out a way to determine how the execution time scales with the size of the matrix using the code and an appropriate graph that shows the scaling, with a fit. Does it agree with your expectations? Make sure you include a png, ps, or pdf of your plot.* [Note: you'll need to go to matrix dimensions of 200 or more to "see" this.]

**It agreed with my expectations because I expected a linear pattern more the greater the dimension, the greater the time it will generate the eigenvalues and eigenvectors**