Session 4 activities worksheet

Due: January 24 *Regrades until:* February 21

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.
- Modify the integration code from Session 3 to use a different function, then evaluate an integral with a singularity. This is one of the Assignment 2 problems, but you can get a head start here.

Please work in pairs (more or less). The instructors will bounce around the room and answer questions.

A) 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 generate a figure. *You'll have to look at the code to identify the columns in derivative_test.dat. Add code to print column headings.*
   → Code added!
2. Edit the plot file to add the corresponding plots and fits for the central derivative and extrapolated central derivative approximations. *Note the file name of the resulting postscript file here.*
   → The file is “derivative_test_plt2.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).*

   → The slope of the extrapolated difference graph is +4.0. This means that the extrapolated difference integration method has an error proportional to h^4, allowing for a much smaller error.

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.*

→ By using Romberg integration method, you can use the results of the Richardson extrapolation to get higher-order results. So, since we eliminated the fourth-order error result, we can now use the Richard extrapolation/Romberg integration to target O(h^6) instead. You would want to combine two O(h^4) estimates to cancel the h^4 term, resulting in O(h^6) being left.

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

   → The source of the error on the left side of the graph has to do with computer error, mainly the round-off error as the mesh size becomes small and starts reaching machine precision increases. All the slopes are the same because they all divide by a factor of h, so in this low mesh size region this division causes the same slope of about -1. This is also a sign of subtractive cancellation.

5. Start a jupyter notebook (look back at session 1 if you forgot how to), load derivative_test_plt.ipnyb in the session_04 directory, run the notebook, and look through it. *What do you not understand in the notebook?*

   → I understand most of it, as I have used Python extensively, including polyfit, plotting, and saving figures. The only new thing for me is using “np.loadtxt” instead of some other loading method, but I believe I understand how it works.

6. The file derivative_test_plt.py included in your session files is the python code you would have gotten if you downloaded the jupyter notebook as a python script (with that "magic" "%matplotlib inline" line commented out). Run it via `python3 derivative_test_plt.py` (back in the interactive OSC desktop session). *What happens?*

   → The code works as expected but the plot pops up on the screen and must be closed to get the file to complete and the image to save.

B) 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*x^{beta}.

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*x^{beta} 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 to generate a postscript file, the name of which you should note here.*
   → The file is "derivate_test_plt3.pdf".
5. *Did you find the same slopes with your new function? Why are the slopes the same but the intercepts different?*

   → Yes (with some rounding due to fit), we find the same slopes with the new function. The slopes are the same because the order of accuracy will still be the same, since we are using the same methods. The y-intercepts differ though because they depend on the second derivatives (for example in the forward difference), which differ for both test functions.

C) 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. We'll use Mathematica to generate the eigenvalues of a 4 by 4 Hilbert matrix. Bring up Mathematica (on your local machine or in the online version) and generate the matrix Amat using the following

Mathematica code. (You could also use HilbertMatrix after loading LinearAlgebra`MatrixManipulation`; see the Help Browser for more information.)

```
MyHilbertMatrix[n_] := Table[1/(i + j - 1), {i, 1, n}, {j, 1, n}]
Amat = MyHilbertMatrix[4]
MatrixForm[Amat]
Eigenvalues[Amat] // N
```

The last two commands present the matrix in the conventional way and calculates the eigenvalues.

3. Run eigen_test with a dimension 4 matrix (i.e., 4x4) and compare the answers to the ones given by Mathematica. 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!).
4. 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.
5. *Figure out a way to determine how the execution time scales with the size of the matrix (e.g., is it a power law? If so, what is the exponent?). Describe your method. (For a plus: attach a graph showing the scaling, with a fit.)* [Note: you'll need the matrix dimension to be 200 or more.]

   → You can define a loop that increases the dimension repeatedly and stores the time to a .dat file. Once we have this, we can plot and fit the exponent using log-log, and doing so gets us the exponent of about 2.3. The plot file is “eigen_test.pdf”. It seems it fits a power law between x^2 and x^3.

D) Extra: More Complicated Integrals

1. Choose one of the integrals with singularities from the "Integrals with Singularities or Discontinuities" handout [Eq.(9) is a good choice!].
   → I chose equation 9.
2. Determine an "exact" answer analytically or using Mathematica (see instructors for assistance).
3. Modify your integration code from Session 3 so that you can analyze the integral with one of the integration rules several ways, using an error plot to compare (these may not all apply to a given integral):
   1. Brute force (unmodified)
   2. Subtracting the singularity
   3. Changing variables

   What do you find?

   → I was able to use all three methods. For subtraction, I integrated 1/(1+x)sqrt(x) – 1/sqrt(x) and then manually added 2sqrt(2) back. For changing variables, I integrated from 0 to sqrt(2) of 2/(1+u^2). What I found, using the numerical integration that the brute force did the worst at approximating the integral, due to the singularity. Subtracting the singularity worked a bit better, but wasn’t as close as simply changing variables such that the singularity was removed.

E) Reflection

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

This worksheet helped with the following goals. It helped me write clear, correct computer code, and learn some more of the algorithms of computational physics. I understand a bit more of the limitations of computational solutions, especially in relation to computational time.