480/905: Activities 9-10 (Revised 3/30/2021)

*Online handouts:* listings of Circle class files, private\_vs\_public.cpp, and square\_test.cpp; "Using GSL Interpolation Functions", listing of GslSpline and test files, ode\_test.cpp listing, CL mystery guide

In this session we'll take a further look at C++ classes, try out a GSL adaptive differential equation solver, briefly look at interpolation and cubic splines, take a first look at Python scripts for C++ programs, and do the "Command Line Mystery"

Optimization 101: Squaring a Number

One of the most common floating-point operations is to square a number. Two ways to square x are: pow(x,2) and x\*x. Let's test how efficient they are.

1. Look at the printout for the square\_test.cpp code. It implements these two ways of squaring a number. The "clock" function from time.h is used to find the elapsed time. Each operation is executed a large number of times (determined by "repeat") so that we get a reasonably accurate timing.
2. We've set the optimization to its lowest value, -O0 ("minus oh zero"), to start in make\_square\_test.
3. Compile square\_test.cpp (using make\_square\_test) and run it. Adjust "repeat" if the minimum time is too small. *Record the times here. Which way to square x is more efficient?*

A picture containing text, plant, bird

Description automatically generated

**x\*x is more efficient**

1. If you have an expression (rather than just x) to square, coding (expression)\*(expression) is awkward and hard to read. Wouldn't it be better to call a function (e.g., squareit(expression)? Add to square\_test.cpp a function:  
   double squareit (double x)  
   that returns x\*x. Add a section to the code that times how long this takes (just copy one of the other timing sections and edit it appropriately, making sure to keep the "final y" cout statement). *How does it compare to the others? What is the "overhead" in calling a function (that is, how much extra time does it take)? When is the overhead worthwhile?*  
     
   A picture containing text, plant, bird

   Description automatically generated

**The overhead is 0.088593. The overhead was sort of worth it since the difference in it was quite small.**

1. Another alternative, common from C programming: use #define to define a macro that squares a number. Add  
   #define sqr(z) ((z)\*(z))  
   somewhere before the start of main. (The extra ()'s are safeguards against unexpected behavior; **always** include them!) Add a section to the code to time how long this macro takes; what do you find?
2. One final alternative: add an "inline" function called square:  
   inline double square (double x) { return (x\*x); };  
   that is a function prototype **and** the function itself. Put it up top with the squareit prototype. Add a section to the code to time how long this function takes. *What is your conclusion about which of these methods to use?*

**Despite putting in the alternatives, using the default x\*x is still the most efficient way because it takes the shortest time.**

1. Finally, we'll try the simplest way to optimize a code: let the compiler do it for you! Change the compile flag -O0 (no optimization) to -O2 (that's the uppercase letter O, not a zero). Recompile and run the code. *How do the times for each operation compare to the times before you optimized? What do you conclude?*

Text

Description automatically generated  
  
**Before optimizations, everything except the pow function took approximately 0.19 seconds more to complete the task, while the pow function took 2 seconds more.**

**The O2 flag made completing the task more efficient.**

1. In your project programs, once they are debugged and running, you'll want to use the -O2 (or maybe -O3) optimization flag.

GSL Interpolation Routines

We'll use the example of a *theoretical* scattering cross section as a function of energy to try out the GSL interpolation routines. The (x,y) data, with x-->E and y-->sigmath, is given in the bottom row of the table in section 10c of the session notes (note we are NOT fitting sigmaexp). You might think we should be doing this for the *experimental* cross section. Usually we will fit rather than interpolate such data because it is noisy and we also want to validate our interpolations against known functions.

* 1. Start with the gsl\_spline\_test\_class.cpp code (and corresponding makefile). Take a look at the printout and try running the code. Note that we've used a Spline class as a "wrapper" for the GSL functions, just as we did earlier with the Hamiltonian class. Compare the implementation to the example on the "Using GSL Interpolation Functions" handout. *Questions?* **No**
  2. Instead of the sample function in the code, you will *change* the program to interpolate the data in the table from the notes. This will require deleting some of the code and adding new lines. Set npts and the (x,y) arrays equal to the appropriate values when you declare them. Declare them on separate lines. An array x[4] can be initialized with the values 1., 2., 3., and 4. with the declaration:  
     double x[4] = {1., 2., 3., 4. };
  3. Use the code to generate a cubic spline interpolation for the cross section from 0 to 200 MeV in steps of 5 MeV. *Output this data and the exact results from equation (10.7) in the notes to a file for plotting with gnuplot and try it out. Plot the exact results "with lines" and the spline using "with linespoints" (or "w linesp"), so you can see both the individual points and the trends.*

Chart, line chart

Description automatically generated

* 1. Now modify the Spline class to allow for a polynomial interpolation (see the GSL handout) and change the gsl\_spline\_test\_class.cpp main program to generate linear and polynomial interpolations as well and add code to print the results to your output file. *Did you succeed?*  
       
     **Yes**
  2. *Generate a graph with all three interpolations plotted along with the exact result. Comment here (a sketch might help) on the strengths and weaknesses of the different interpolation methods,*  *both near the peak and globally.*  
       
       
       
     Chart, line chart

     Description automatically generated

**The cubic spline interpolation has a peak and shape that is similar to the exact result. Meanwhile the polynomial interpolation creates 2 local peaks, and it is “off” from the exact result. However the polynomial interpolation has another peak that is similar to the cubic interpolation and the exact answer.**

Command Line Mystery

The "Command Line Mystery" is a whodunit designed to give you some practice with useful shell commands and how to string them together (with "pipes"). Follow the instructions on the clmystery handout. *Did you solve the mystery?*

**I skipped this part**

Python Scripts for C++ Programs

This exercise is just a first exposure to what is possible with Python scripts. The listings for the scripts and revised versions of the area.cpp C++ programs are in the Activities 10 notes.

* 1. Look at area\_cmdline.cpp first and try it out (there is a makefile), first omitting an argument when executing it. Then look at and try run\_area\_cmdline1.py. *Change the list of numbers to generate the area for radii from 5 to 25 spaced by 5. Did you succeed?* **No**
  2. Modify both area\_cmdline.cpp so that it takes *two* arguments, the radius and an integer called again. Change the code so the output line is repeated again times. Modify run\_area\_cmdline1.py so it works with this new version. *Did you succeed?* **No**
  3. Try out run\_area\_cmdline2.py, modifying value\_list1 and value\_list2 to help you understand how they work. *Questions? [Note: this might fail on Cygwin]***No**
  4. For now, just look through run\_area\_cmdline3.py and try running it. Note the use of findall and sorting, which may come in handy later.
  5. Look at area\_files.cpp and try it out (there is a makefile). There is also a Python script, run\_area\_files2.py, to try. (CHALLENGE) Modify the program and script so that the input file has an extra column for the integer again introduced in part 2.

Cubic Splining

Here we'll look at how to use cubic splines to define a function from arrays of x and y values. A question that always arises is: How many points do we need? Or, what may be more relevant, how accurate will our function (or its derivatives) be for a given spacing of x points?

* 1. We'll re-use the Spline class from the last section and the original gsl\_spline\_test\_class.cpp function, which splined an array.
  2. The goal is to modify the code so that it splines the ground-state hydrogen wave function: u(r) = 2\*r\*exp(-r)
  3. Your task is to determine how many (equally spaced) points to use to represent the wave function. Suppose you need the derivative of the wave function to be accurate to one part in 106 for 1 < r < 4 (absolute, not relative error) *Devise (and carry out!) a plan that will tell you the spacing and the number of points needed to reach this goals. What did you do?*

1. int npts = 200;
2. double x\_values[npts];
3. double y\_values[npts];
5. for (int i = 0; i < npts; i++)
6. {
7. double x\_temp = double(i) / 50.;
8. x\_values[i] = x\_temp;
9. y\_values[i] = 2\*pow(x\_temp,(-1\*x\_temp));
10. }
11. string type = "cubic";
12. Spline my\_cubic\_spline (x\_values, y\_values, npts, type);
13. ofstream hydro\_out ("hydro.dat");
14. hydro\_out << "    x    absolute  " <<endl;
15. // Evaluate the spline and derivatives
16. for(int i = 0; i<200; i++){
17. double hydro = hydrogen(i);
18. double y = my\_cubic\_spline.y (i);
19. double y\_deriv = my\_cubic\_spline.yp (i);
20. hydro\_out << fixed << setprecision(6)
21. << i << "  " << y << "  " << log10 (fabs (y\_deriv- hydro)) <<  “ “
22. << endl;
23. }
24. cout << "data stored in hydro.dat\n";
25. hydro\_out.close ();
26. return (0);      // successful completion
27. }
28. double hydrogen(double x) {
29. double y = 2\*pow(x,-x);
30. return y;

1. Now suppose you need integrals over the wave function to be accurate to 0.01%. *Devise (and carry out!) a plan that will tell you the spacing and the number of points needed to reach this goals.*

To try out integrals, use one of the GSL integration routines on an integral involving the splined u(r) that you know the answer to (hint: what is the total probability?). Note: The qags\_test.cpp program from the Activities 4 files can be quickly adapted for this exercise.