480: Activities 12

Online handouts: gaussian_random.cpp, random_walk.cpp, and other listings.

Today we'll play some games with the GSL random number generators.

Your goals for today:

- Generate some random walks and verify their properties.
- Try out primitive Monte Carlo integration.
- Take a look at an alternative version of the random walk code using classes.

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

Random Number Generation

The program gaussian_random.cpp calls GSL routines to generate both uniformly distributed and gaussian distributed numbers.

1. Look at the gaussian_random.cpp code (there is a printout) and identify where the random number generators are allocated, seeded, and called. *If you were creating a RandomNumber class, what statement(s) would you put in the constructor and destructor? What private variables would you define?* Well, if I could help it, I wouldn't do much with the destructor. In the constructor I would have number of numbers, type of distribution, and bounds. Also, an option sigma argument. I don't really see the point of private variables here, besides forcing it to generate new numbers upon changing seed, sigma, etc.

2. Compile and link the code (use make_gaussian_random) then generate pairs of uniformly and gaussian distributed numbers in random numbers.dat.
3. Devise and carry out a way to use gnuplot to roughly check that the random numbers are uniformly distributed. [Hint: Read the notes. Your eye is a good judge of nonuniformity in two dimensions.] *What did you do?* Just plotted the first 2 columns. Looks pretty uniform.

3. You can check the distributions more quantitatively by making histograms of the random numbers. Think about how you would do that. Then take a look at gaussian_random_new.cpp, which has added crude histogramming (as well as automatic seeding). Use the makefile to compile and run with about 100,000 points. Look at random_histogram.dat. *Use gnuplot to plot appropriate columns (with appropriate ranges of y) to check the uniform and gaussian distributions. Do they look random? In what way?* Yeah they worked. I put very little effort into showing though.

4. Run gaussian_random_new.plt to plot and fit the gaussian distributions with gnuplot. Try 1,000,000 points and 10,000 points. *Do you reproduce the parameters of the gaussian distribution?* (You may need to set b to a reasonable starting point such as the approximate peak height to get a useful fit.) Yeah.

Random Walking

We'll generate random walks in two dimensions using method 2 from the list in Section b of the Activities 12 notes. In particular we'll start at the origin: (x,y) = (0,0) and for each step select Delta_x at random in the range

[-sqrt(2), sqrt(2)] and Delta_y in the same range. So positive and negative steps in each direction are equally likely. The code random_walk.cpp implements this plan.

1. *What is the rms step length?* (Note: this is tricky!) Isn't it just 0? If not, x_rms = sqrt(2)/2, and the norm step size is then 1. (This turns out to be true.)

2. Look at the random_walk.cpp code and identify where the random number generator is allocated, seeded, and called. Compile and link the code (use make_random_walk) and generate a random walk of 6400 steps.
3. *Plot the random walk (stored in "random_walk.dat") using gnuplot (use "with lines" to connect the points).* Repeat a couple of times to get some intuition for what the walks look like.
4. Check (using an editor) for the endpoints of a few walks. *Roughly how does a typical distance R from the origin scale with N? (Can you reproduce the derivation from the notes of how the average of R scales with N?)* Yeah, you even gave us code that does it for us.

5. Now we'll study more systematically how the final distance from the origin R = sqrt(x_final^2 + y_final^2) scales with the number of steps N. Note that now we don't need to save anything from a run except the value of R. The value of R will fluctuate from run to run, so for each N we want to average over a number of trials. *How many trials should you use?* A lot. Apparently the answer is at least sqrt(N), but I do not remember the statistics behind that.

   Edit the code to make multiple runs for each value of N and takes the average of R. *Make (and sketch) an appropriate plot that reveals the dependence of R on N.* [The code random_walk_length.cpp and plot file random_walk_length.plt implement this task. Try it yourself before looking at those.] *Does it agree with expectations?* Yes it does, but I did not try it myself.

Monte Carlo Integration: Uniform and Gaussian Sampling

Your goal is to estimate the D-dimensional integral of

$$(x1 + x2 + \ldots + xD)^2 \quad 1/(2pi\ sigma^2)^{D/2}\ exp(-(x1^2+x2^2+\ldots+xD^2)/(2\ sigma^2))$$

where each of the variables ranges from -infinity to +infinity. The exact answer is D*sigma^2. [Note that the integral without the squared sum in front is normalized to be one.]

The basic Monte Carlo integration method is described in Section d of the Activities 12 notes. In particular, equations (12.15) and (12.16) show that the integral is given approximately by the range(s) times the average of the function evaluated at N random vectors. (So for a 5-dimensional integral, each vector is a set of 5 random numbers {x1,x2,x3,x4,x5}.)

1. Look at mc_integration.cpp to see how this is implemented for our test integral. Because the integral has infinite limits, we approximate it with finite lower and upper limits. How would you choose these? Well, you want the boundaries to be reasonably close to limiting behaviors, so, do that. Another option involves a change in variable, like how the gsl qagi integration works.

2. The dimension is initially set to D=1 (called `dim` in the code). Compile it with make_mc_integration and run it several times. After each run, use mc_integration.plt to make a fitted plot of the error. *What do you*

observe? error tends to (1/sqrt(N))

3. Next try changing the dimension to 3 and then to 10, repeating the last part. *What do you observe? Is it consistent with the notes?* error tends to (1/sqrt(N))! YES!

4. If you have time, modify the code to apply equations (12.34) and (12.35), where we identify w(x) as the normalized gaussian part of the integral and use the GSL routine for generating gaussian-distributed random numbers (from gaussian_random.cpp). [If you are short of time, use mc_integration_new.cpp.] *What should the integrand function return in this case?* I am short on time, but generally the idea of using weight functions like that is to transform the system into something easy to integrate, like a constant or line. So, I am gonna guess it becomes flatter.

   Repeat the analysis in different dimensions. *Why are the results better than for uniform sampling? Can you do D=100?* Well, this function is 0 at r = 0, while the gaussian has its max value there. It would probably improve if we used a gaussian with a much bigger sigma than the function though! Yes, yes I can.

Monte Carlo Integration: GSL Routines

Run a test program to do a simple D-dimensional integral as in the last section but with the Vegas and Miser proograms.

1. Take a look at the program gsl_monte_carlo_test.cpp while also looking at Monte Carlo integration in the online GSL library.
2. The initial integral is not a great test. After compiling and running the program, change the integrand to something more interesting (use your imagination!). Don't worry about knowing the exact answer; compare the results from the different routines. *What do you find?* Miser > Plain >> Vegas

---

C++ Class for a Random Walk

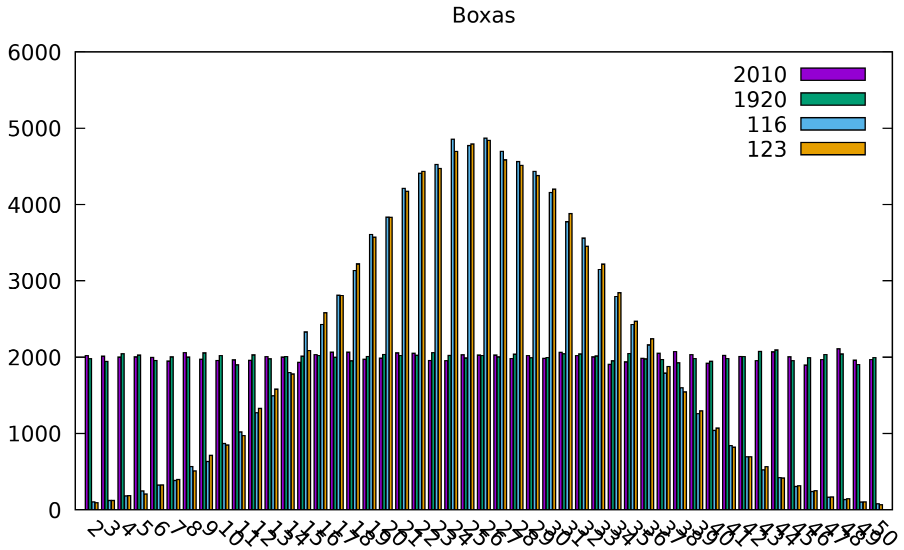
The random walk code random_walk.cpp is basically written as a C code with C++ input and output. Here we reimplement the code as a C++ class.

1. In the RandomWalk directory, compile and link RandomWalk_test (using make_RandomWalk_class_test). Run it to generate "RandomWalk_test.dat", which you should plot with gnuplot to verify that the output looks the same as from random_walk.cpp.
2. Compare the old and new code (you have printouts of each). Discuss with your partner the advantages (and any disadvantages) of the definition of RandomWalk as a class. List some. You can import that class into other codes, you can clean up inputs, and if done right, improve readability.

3. An advantage of programming with classes is the ease of extending or generalizing a code. List two ways to extend the class definition. I would allow multiple random distributions and different boundries to be set, all as optional arguements upon creation.

4. As time permits, modify the code to do the following:
   - Extend the code to make available (with "get" functions) the x- and y-components of the last step (what are called delta_x and delta_y internally).
   - Allow for the upper and lower limits of the step size to be initialized by the user. (And you still want to be able to use the current version that doesn't require these.) [Hint: Can you have more than one constructor?]

   - *How would you allow for the random number generator to be changed? Implement it!*


Boxas
2010
1920
116
123
0
1000
2000
3000
4000
5000
6000