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?*  
     
   For the constructor, I would put a pointer towards the data of random numbers

For the destructor, it would look some like this:

if( NULL != data )

{

delete [] data ;

}

I would define the private values as the following:

private:

int sampleSize, minValue, maxValue;

int data[];

Compile and link the code (use make\_gaussian\_random) then generate pairs of uniformly and gaussian distributed numbers in random\_numbers.dat.

1. 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?*

Text, letter

Description automatically generated

Chart, scatter chart

Description automatically generated

1. 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?*

Chart, line chart

Description automatically generated

**They look random because the points in the data spans a large range from 0 to 5000.**

1. 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.)

Chart, line chart, histogram

Description automatically generated

**Yes**

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!)

**The the root-mean-square step length is approximately the square root of the product of the square average distance R and N points**

1. 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.
2. *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.

Chart

Description automatically generated with medium confidence

1. 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?)*

**The distance R is approximately the product of the square root of N and the root-mean-square step length.**

1. 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?*

**10**

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?*  
  
Chart, scatter chart

Description automatically generated

Chart, scatter chart

Description automatically generated

**It agrees with the expectations because since R is the square root of N, the log-log plot of R vs. N (second figure for the this question) has a slope of approximately 0.5 and that is consistent with the dependence of R on N.**

Monte Carlo Integration: Uniform and Gaussian Sampling

Your goal is to estimate the D-dimensional integral of

(x1 + x2 + ... + xD)2 1/(2pi sigma2)D/2 exp(-(x12+x22+...+xD2)/(2 sigma2))

where each of the variables ranges from -infinity to +infinity. The exact answer is D\*sigma2. [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?*

**By comparing how far the estimates are from the exact value (1 in this case).**

1. 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?*  
     
   **The slope and the b values change every time I make the plot**
2. 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?*

**The dimension of 3 has a slope of roughly -0.6 after a few runs and the dimension of 10 has a slope of roughly -0.1 after a few runs.**

**It is consistent with the notes where it says it improves the variance of the data.**

1. 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?*

**It returns the results with exponents to go along with it.**  
  
  
Repeat the analysis in different dimensions. *Why are the results better than for uniform sampling?* *Can you do D=100?*

**The graphs are more consistent than uniform sampling. I was able to do D=100.**

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?*

**The errors are slightly error than the uniform and gaussian sampling**

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.

**Advantages:**

* + - **It can just be called instead of hard coding it.**
    - **Can pack multiple functions.**

**Disadvantages:**

* + - **Sometimes it can be very complicated to link when compiling.**

1. An advantage of programming with classes is the ease of extending or generalizing a code. List two ways to extend the class definition.

* **using subclass\_name**
* **Declaring a private and public class**

1. 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!*