

11. 780.20 Session 11

a. Follow-ups to Sessions 9 and 10

For most of you there will be several tasks left unfinished from Sessions 9 and 10. You'll do the "Cubic Splining" task from Session 9 and the last part of the "Nonlinear Least-Squares Fitting with GSL Routines" task from Session 10 as Assignment 4 (last assignment besides project). We'll return to the sodium-chlorine molecule problem in Session 14.

b. Random Numbers

Monte Carlo methods provide powerful techniques for simulating experiments, solving complicated many-body systems, and getting nonperturbative results from quantum field theories. They are the basis for evaluating higher dimensional integrals, including the approximation of path integrals. In this session we'll look at a principal ingredient of Monte Carlo calculations, random numbers, with some basic applications. In the next sessions we'll simulate a spin system and look at variational Monte Carlo.

A sequence of numbers is random if *uncorrelated*: that is, if you know x_1, x_2, \dots, x_i , then x_{i+1} is not predictable. The distribution of random numbers need not be uniform; for example, if the distribution is gaussian, some ranges of numbers will be more or less likely than other ranges. This may not be obvious when looking at a short sequence. However, if we generate a large number of random numbers and plot them in histogram form, we expect that the histogram will approach the shape of the probability distribution function or PDF (e.g., flat if uniform or like a gaussian if a gaussian distribution or ...).

Computers generate *pseudo*-random numbers, which are not truly random but simulate them effectively. Some basic features of a good random number generator (abbreviated rng) [2]:

1. Produces a uniform distribution in $[0, 1]$.
2. Correlations between random numbers are negligible.
3. The period before the pseudo-random sequence repeats is as large as possible (e.g., 10^9 or greater).
4. The algorithm should be fast.

The trade-off between the last three features are what distinguishes different rng's. There are various tests of random numbers. A quick (but not infallible) eyeball check if numbers are random is to plot them (e.g., with gnuplot) by treating them as (x, y) pairs (that is, the first two numbers form the first pair, the next two form the second pair, and so on). Another test is to calculate the average of the k 'th power of a set of N numbers:

$$\langle x^k \rangle \equiv \frac{1}{N} \sum_{i=1}^N x_i^k p(x_i) , \quad (11.1)$$

where the PDF $p(x) = 1$ for $0 \leq x \leq 1$ for a uniform distribution. For large N , the results should (if the numbers are random) approach the limiting integral

$$\langle x^k \rangle \xrightarrow{N \rightarrow \infty} \int_0^1 dx p(x) x^k = \int_0^1 dx x^k = \frac{1}{k+1}, \quad (11.2)$$

so we should find the mean μ and standard deviation σ to be

$$\mu = \langle x \rangle = \frac{1}{2}, \quad \sigma = \sqrt{\langle x^2 \rangle - \mu^2} = \frac{1}{\sqrt{12}} \approx 0.2886. \quad (11.3)$$

A comparison of the mean and standard deviation of the N numbers to these limiting results provides a test of randomness. More generally we would consider the *autocorrelation function*

$$C_k \equiv \frac{\langle x_{i+k} x_i \rangle - \langle x_i \rangle^2}{\langle x_i^2 \rangle - \langle x_i \rangle^2}, \quad (11.4)$$

with $C_0 = 1$. If $C_k \neq 0$ for $k \neq 0$, then the random numbers are not, in fact, independent.

We'll not worry here about the theory of generating random numbers (see Refs. [3] and [2]) but let GSL generate them. (Every operating system or programming language will generally have a random number generator but it is more reliable and portable to use the GSL routines, which include some very good generators.) An rng is initialized with a seed: here that means a (long) integer. Starting with a different seed will lead to a different sequence of pseudo-random numbers, but if you start with the same seed, you'll get the same sequence (which hardly sounds random!). Most of the time we'll use a function called `random_seed` to generate the seed for us.

We will often want our random numbers to be generated for different PDF's. A uniform distribution in the interval $[a, b]$ (rather than just $[0, 1]$) is defined by [2]

$$p(x) = \frac{1}{b-a} \theta(x-a) \theta(b-x), \quad (11.5)$$

where $\theta(z) = 1$ if $z > 0$ and is otherwise zero. A normal or gaussian distribution on $[-\infty, \infty]$ is specified by the mean μ and standard deviation σ through

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x-\mu)^2/2\sigma^2}. \quad (11.6)$$

The GSL routines can generate essentially all of the common PDF's. In the next session, we'll look at generating PDF's according to Boltzmann factors via the Metropolis algorithm.

c. Random Walks

How far do we get *on average* in a two-dimensional random walk? Let's suppose we take N steps, equally likely in each direction, according to a prescription for determining Δx_i and Δy_i in the i 'th step. The mean displacement in the x direction should be zero by symmetry, and similarly in

the y direction. But the average square of the distance R^2 will be nonzero. If we take our N step random walk many times and average, then

$$R^2 \approx \langle (\Delta x_1 + \Delta x_2 + \cdots + \Delta x_N)^2 + (\Delta y_1 + \Delta y_2 + \cdots + \Delta y_N)^2 \rangle, \quad (11.7)$$

where $\langle \cdots \rangle$ denotes the average. Since the steps are supposed to be uncorrelated, we should find

$$\langle \Delta x_i \Delta x_j \rangle = \langle \Delta y_i \Delta y_j \rangle = 0 \quad \text{if } i \neq j, \quad (11.8)$$

which means that all the cross terms average to zero in Eq. (11.7). Now for any i , the average of $\Delta x_i^2 + \Delta y_i^2$ will be the same, so

$$\langle \Delta x_i^2 + \Delta y_i^2 \rangle \equiv \langle r^2 \rangle \quad \text{for any } i. \quad (11.9)$$

Then

$$R^2 \approx N \langle r^2 \rangle \quad (11.10)$$

or

$$R \approx \sqrt{N} r_{\text{rms}}, \quad (11.11)$$

where r_{rms} is the *root-mean-squared* step size. Since the mean is zero, R^2 is the *variance* and R is the *standard deviation* for the walk. Note that we would get the same result as Eq. (11.11) in three or higher dimensions. To summarize, the total (absolute) distance covered after N steps is $N r_{\text{rms}}$, but the *net* radial distance from the origin of the walk is, on average, only $\sqrt{N} r_{\text{rms}}$ [1].

There are many possible ways to conduct a random walk. Here is the list of random walk methods from Landau and Paez [1] (We will use the second method in Session 11):

1. Choose

$$\Delta x = \cos \theta \quad \Delta y = \sin \theta \quad (11.12)$$

where θ is taken at random from $[0, 2\pi]$. (Note: this will not give a uniform walk if θ is distributed uniformly.)

2. Choose Δx and Δy uniformly in the interval $[-\sqrt{2}, \sqrt{2}]$. *Is this the correct interval?*
3. Choose Δx uniformly in the range $[-1, 1]$ and then generate the sign of Δy randomly but get its magnitude from $\Delta y = \pm \sqrt{1 - \Delta x^2}$.
4. Choose the major compass direction (north, south, east, or west) randomly and take a unit step in that direction.
5. Choose among the major and minor compass directions uniformly (N, NW, W, SW, S, SE, E, NE).

d. C++ Class for a Random Walk

Included in the Session 11 tarball is the code `random_walk.cpp`, which is basically a C code with C++ input and output. Also included in the tarball is a division of the code into a class called `RandomWalk` and a test program `RandomWalk_test.cpp`. Here are some comments on the implementation.

- The code `random_walk.cpp` has the implementation of the random walk mixed up with its application. The user shouldn't have to know the internal details of how steps are taken or how the walk is represented internally (e.g., in cartesian or polar coordinates). If we wanted to put a random walk in another program, we'd have to carefully cut-and-pasted pieces of this code. Then, when we modified the walk (e.g., by adding a new capability or maybe fixing a bug), we'd have to go back and update it everywhere in the previous codes.
- We could improve the code simply by defining functions that do the major tasks of the random walk: initializing the walk, taking a step, finalizing the walk, and putting the functions in a separate file. But then we'd still have to keep track of the *data*, the variables of the walk, in the main program. So a better solution is to introduce a *class*, which incorporates both the data and the functions.
- In `RandomWalk.h`, we see that the variables describing the walk, such as the upper and lower limits of the random step, the current position, and the details of the random number generator, are all *private*. This means that any of the `RandomWalk` functions can access and potentially change them, while they are not visible to the outside program (in this case `RandomWalk_test.cpp`).
 - We always have a *constructor* function, which is invoked when a new `RandomWalk` object is declared, and a *destructor* function, which is invoked when the object is destroyed. In general we will have a *copy* constructor as well, with instructions on how to make a copy of our `RandomWalk` object. In this case, the automatically generated default version is adequate.
 - The functions `get_x` and `get_y` provide the interface that lets the outside function get (but not modify!) the current *x* and *y* coordinates. Note that they are defined *inline*, entirely within the header file. This is recommended for simple functions like these.
- In `RandomWalk.cpp`, the constructor, destructor, and remaining function (`step`) are implemented. We have basically just cut-and-pasted the original code from `random_walk.cpp` into the appropriate places. Note how `step` has access to `x` and `delta_x` (and so on); they are neither passed to it or declared within it.
- In the main function, we create the `RandomWalk` object with


```
RandomWalk my_random_walk (x, y);
```

 where `x` and `y` are used in the constructor to define the initial position of the walk. Note that `my_random_walk` refers to a particular object. An independent one could be created with


```
RandomWalk my_other_random_walk (x, y);
```

 We use the “dot” notation to call functions:


```
my_random_walk.step (); // take a step
```

 and so on.
- There are many possible extensions, some of which are suggested in the 1094 Session 11 guide.

e. Monte Carlo Integration: Uniform Sampling

We've looked at a variety of integration rules designed for one-dimensional integration, which improve as inverse powers of the number of points N used (e.g., Simpson's rule went like $1/N^4$). One might guess that the best way to do a multi-dimensional integrals is just to do iterated one-dimensional integrals, with an integration rule applied to each. This works for two- and three-dimensional integrals, but starting somewhere around $D = 4$ or $D = 5$, it is usually more effective to turn to a Monte Carlo method. While it's hard to believe that picking places to evaluate an integrand at random is better than picking in an organized fashion, it is nevertheless true.

The basic idea starts with the expression from calculus that the average of a function $f(x)$ in the interval $[a, b]$ (denoted $\langle f \rangle$) is related to the integral over that interval by

$$\langle f \rangle = \frac{1}{b-a} \int_a^b dx f(x) , \quad (11.13)$$

which implies the integral I can be calculated from

$$I = \int_a^b dx f(x) = (b-a)\langle f \rangle . \quad (11.14)$$

If we use Monte Carlo methods to estimate the mean value of f , we have an estimate for the integral. Therefore, we generate a sequence $\{x_i\}$ of N random numbers uniformly distributed in $[a, b]$ and take the average, so that

$$\int_a^b dx f(x) \approx (b-a) \frac{1}{N} \sum_{i=0}^{N-1} f(x_i) . \quad (11.15)$$

This looks like an integration rule with randomly distributed nodes x_i and equal weights $w_i = (b-a)/N$. The generalization to many dimensions is immediate. For example,

$$\int_a^b dx \int_c^d dy f(x, y) = (b-a)(d-c)\langle f \rangle \approx (b-a)(d-c) \frac{1}{N} \sum_{i=0}^{N-1} f(x_i, y_i) . \quad (11.16)$$

It's the same formula!

How well does this work? Along with the average $\langle f \rangle$ we can calculate the variance σ^2 or the standard deviation σ . The variance of the integral with f is defined as [2]

$$\sigma_f^2 \equiv \langle f^2 \rangle - \langle f \rangle^2 \quad (11.17)$$

$$= \frac{1}{N} \sum_{i=0}^{N-1} f(x_i)^2 - \left(\frac{1}{N} \sum_{i=0}^{N-1} f(x_i) \right)^2 , \quad (11.18)$$

and is a measure of how much f deviates from its average in the integration region. The idea is then to consider a measurement of the integral to be the result for a fixed value of N . If we do this M times, then the average for the integral I ,

$$\langle I \rangle_M = \frac{1}{M} \sum_{j=0}^{M-1} \langle f \rangle_j , \quad (11.19)$$

and the variance for $M = N$ is

$$\sigma_N^2 = \frac{1}{N} \left[\left\langle \left(\frac{1}{N} \sum_{i=0}^{N-1} f(x_i) \right)^2 \right\rangle - \left(\left\langle \frac{1}{N} \sum_{i=0}^{N-1} f(x_i) \right\rangle \right)^2 \right]. \quad (11.20)$$

As with the random walk, cross terms in the first sum vanish in the large N limit, leaving the result we're looking for:

$$\sigma_N^2 \approx \frac{1}{N} (\langle f^2 \rangle - \langle f \rangle^2). \quad (11.21)$$

The implication is that the error σ_N is proportional to $1/\sqrt{N}$, so to get an extra decimal place we will typically need 100 times as many points. This sounds terrible compared to Simpson's rule or even the trapezoid rule, *but we have the same scaling in any number of dimensions*. This eventually wins in higher dimensions, since applying Simpson's rule, for example, requires the N points to be divided among D dimensions, so the actual scaling is $N^{-4/D}$, which will be worse than $N^{-1/2}$ for large enough D . (In practice, as noted above, the cross-over tends to be around $D = 4$, but it depends on the integral in question.)

The discussion so far has been based on distributing the points uniformly. This is wasteful, since we may be evaluating the integrand many times where it is very small. We can do better with *importance sampling*. The most accurate function that can be integrated with Monte Carlo integration is a constant, in which case we get exactly the correct answer with a uniform distribution. We can approach this result for a non-constant f if we use a weight function $w(x)$ for which $f(x)/w(x)$ is reasonably constant. Thus, we consider

$$I = \int_a^b dx f(x) = \int_a^b dx w(x) \frac{f(x)}{w(x)} \quad (11.22)$$

estimated with random numbers distributed according to $w(x)$. Then the estimate is

$$I = \left\langle \frac{f}{w} \right\rangle \approx \frac{1}{N} \sum_{i=0}^{N-1} \frac{f(x_i)}{w(x_i)}. \quad (11.23)$$

This approach can dramatically reduce the variance. The GSL routines for adaptive Monte Carlo integration you try in Session 11 carry out this strategy. More on this next time!

f. References

- [1] R.H. Landau and M.J. Paez, *Computational Physics: Problem Solving with Computers* (Wiley-Interscience, 1997).
- [2] M. Hjorth-Jensen, *Computational Physics*. These are notes from a course offered at the University of Oslo. See the 780.20 webpage for links.
- [3] W. Press *et al.*, *Numerical Recipes in C* (Cambridge, 1992). Individual chapters are available online from <http://www.nrbook.com/a/>. There are also versions for Fortran and C++.