12. 6810 Session 12

a. Follow-ups to Sessions 10 and 11

- **Interpolation.** Here are some take-away points from our brief look at interpolation:
  - Only use polynomial interpolation over small intervals. Our mistake in Session 10 was to use an N^{th} order polynomial to fit N points over a large interval. While this guaranteed the interpolating function would include all of the data points, the function we were approximating *did not* look like a polynomial globally. The consequence was that in between the fit points at the edges of an interval the polynomial sometimes deviated wildly from the real function. This is called "Runge's phenomenonon".
  - Taylor's expansion does guarantee it will look like a *low-order* polynomial over a short-enough distance, as long as there are no singular points in the interval. (The Weierstrass approximation theorem says we can approximate any continuous function over an interval as closely as we desire, but this requires we allow arbitrarily high-order polynomials.) How short? If you plot a function, there is a characteristic distance over which it changes rapidly in a given region. Short compared to that distance in that region.
  - The polynomial does have the advantage that it is *smooth*. That means that all of the derivatives at any point are continuous. If we put together a series of order $n < N$ polynomials, we have to stitch them together and it is not possible to make all of the derivatives continuous.
  - A good compromise (usually) is the cubic spline, which joins together cubic (third-order) polynomials requiring that the function be continuous and the first and second derivatives be continuous. This looks smooth to the human eye in most cases. By default, this is what Mathematica does when it creates an `InterpolatingFunction` object, but you can change the degree of the polynomial or switch from what is called spline interpolation to what is called Hermite interpolation (I'm not sure what this means internally).
  - In practice, cubic splines may have a problem at the ends of the interval, where a boundary condition is assumed that may differ from the function being approximated, leading to large errors. One solution, if possible, is simply to extend the fitting interval beyond where we need to know the function accurately. This may be possible even when we are at a physical boundary, by continuing the function appropriately (e.g., a radial wave function with definite parity for $r > 0$ can be continued smoothly to $r < 0$ by reflecting it in the origin, with a minus sign if needed to make it smooth, to make the interpolation accurate for small r).
- **Extrapolation.** We've seen that interpolation can be tricky. What is we need to extrapolate, that is, to predict the value of a function outside of the region where we have sample points. One option to consider is rational function interpolation/extrapolation. There is a good description in section 3.2 of Numerical Recipes [3]; we'll just make some background comments here.

A rational function is a *quotient* of polynomials,

$$\frac{P_\mu(x)}{Q_\nu(x)} = \frac{p_0 + p_1 x + \cdots + p_\mu x^\mu}{q_0 + q_1 x + \cdots + q_\nu x^\nu} \ , \tag{12.1}$$

which has $m+1 \equiv \mu+\nu+1$ unknown p_is and q_is (we can take $q_0 = 1$ without loss of generality). So we need this many tabulated functional values to construct the rational function. If we know the function analytically, we can construct a rational function approximation by demanding that its power series expansion agrees with the first $m + 1$ terms of the Taylor series expansion of the function (about $x = 0$). This is called a *Padé approximant* (see section 5.12 of Numerical Recipes). For tabulated data, there is an algorithm by Bulirsch and Stoer to find the coefficients.

Rational functions are generally better than simple polynomials because they can model a much richer analytic structure, namely they can have poles. These are likely to occur for non-real values of x, where they can cancel poles in the function of interest that screw up polynomial approximations [3].

- **Nonlinear curve fitting.** In either linear or nonlinear curve fitting, we minimize a "figure-of-merit"or "objective" function, which is often a least-squares function (that is, the sum of squares of deviations of the data from the candidate fit function), to determine the set of parameters that define the candidate fit function. (See Ref. [3] for a discussion of why least-squares is often the relevant choice for the figure-of-merit function.) Finding the parameters when they appear linearly in the candidate function is a direct linear algebra problem (see the next item). The strategy in the nonlinear case is to *linearize* the candidate function around a guess for the vector of parameters and then iterate (that is, correct the guess after solving the linear problem and repeat the linearization until some stability criterion is reached).

- **Linear least-squares problem.** [This discussion, including notation, is based on "Estimating Errors in Least-Squares Fitting" by P.H. Richter. See there for more details.] The general least-squares problem to be solved is to minimize the function

$$\chi^2(\mathbf{a}) = \sum_{i=1}^{N} \frac{[y_i - y(x_i, \mathbf{a})]^2}{\sigma_i} \ , \tag{12.2}$$

with respect to the components of $\mathbf{a}$, where

$$\mathbf{a} = \{a_1, a_2, \ldots, a_M\} \tag{12.3}$$

is the vector of M coefficients, and the y_i are the N data points, each with uncertainty σ_i. The example of fitting $y(t) = Ae^{-\lambda t} + b$ is of this form, with $a_0 = A$, $a_1 = b$, and $a_2 = \lambda$. The parameter a_2 is the only nonlinear one in this case (because it is up in the exponential). When we expanded an exact radial wave function in harmonic oscillator wave functions,

$$u_{\rm exact}(r) = \sum_{i=1}^{D} c_i \, u_i^{\rm HO}(r) \ , \tag{12.4}$$

to determine the coefficients c_i, we were solving a linear least-squares problem (with $N = D$ and $a_i = c_i$). The linear least-squares problem can conveniently be cast in matrix form. In this case, the function y takes the form

$$y(x; \mathbf{a}) \equiv \sum_{j=1}^{M} a_j X_j(x) \ , \tag{12.5}$$

where the $X_j(x)$ are a set of M basis functions. Because we apply this expression in Eq. (12.2) at N points, we can introduce the $N \times M$ matrix $\mathbf{X}$, with

$$X_j(x_i) \longrightarrow \mathbf{X}_{ij} \ . \tag{12.6}$$

We can then rewrite the expression for y in matrix form:

$$y(x_i; \mathbf{a}) = \sum_{j=1}^{M} \mathbf{X}_{ij} \mathbf{a}_j \quad (i = 1, \ldots, N) \qquad \text{or} \qquad \mathbf{y} = \mathbf{X}\mathbf{a} \ . \tag{12.7}$$

Similarly, if we define $\mathbf{b}_i \equiv \mathbf{y}_i/\sigma_i$ and $\mathbf{A}_{ij} \equiv \mathbf{X}_{ij}/\sigma_i$, then the linear χ^2 function is

$$\chi^2(\mathbf{a}) = (\mathbf{b} - \mathbf{A}\mathbf{a})^\intercal (\mathbf{b} - \mathbf{A}\mathbf{a}) \ , \tag{12.8}$$

where $\mathbf{M}^\intercal$ means the transpose of matrix $\mathbf{M}$. Then the least-squares problem is directly solved as:

$$\frac{\partial}{\partial a_j}\chi^2 = 0 \ , \ \forall j \quad \Longrightarrow \quad (\mathbf{A}^\intercal\mathbf{A})\mathbf{a} = \mathbf{A}^\intercal\mathbf{b} \quad \Longrightarrow \quad \mathbf{a} = (\mathbf{A}^\intercal\mathbf{A})^{-1}\mathbf{A}^\intercal\mathbf{b} \equiv \mathbf{C}\mathbf{A}^\intercal\mathbf{b} \ , \tag{12.9}$$

which defines the *covariance matrix* $\mathbf{C}$. So given $y_1, \ldots, y_N$, we get a determined result for $a_1, \ldots, a_M$. What is the uncertainty in each a_j due to the uncertainties in the y_i's? In matrix form,

$$\delta\mathbf{a} = \mathbf{C}\mathbf{A}^\intercal\delta\mathbf{b} \tag{12.10}$$

so that the covariance of $\mathbf{a}$ is

$$\sigma^2_{\mathbf{a}} \equiv \langle \delta\mathbf{a}\delta\mathbf{a}^\intercal \rangle = \langle \mathbf{C}(\mathbf{A}^\intercal\delta\mathbf{b})(\mathbf{A}^\intercal\delta\mathbf{b})^\intercal\mathbf{C}^\intercal \rangle = \mathbf{C}\mathbf{A}^\intercal\langle\delta\mathbf{b}\delta\mathbf{b}^\intercal\rangle\mathbf{A}\mathbf{C}^\intercal = \mathbf{C}(\mathbf{A}^\intercal\mathbf{A})\mathbf{C}^\intercal = \mathbf{C} \ , \tag{12.11}$$

where we have used (by assumption) that the errors in the y_i are *independent*, so

$$\langle \delta y_i \delta y_j \rangle = \delta_{ij}\sigma_i^2 \quad \Longrightarrow \quad \langle \delta\mathbf{b}\delta\mathbf{b}^\intercal \rangle = \mathbf{I} \text{ (the identity matrix)} \ . \tag{12.12}$$

Then the individual variances are

$$\sigma^2_{a_j} = \mathbf{C}_{jj} \ \text{ (with no summation on } j) \tag{12.13}$$

or the one-sigma uncertainty for each parameter is given by the square root of the corresponding diagonal matrix element of the covariance matrix. We need to remember the assumption about independent errors, which may be true for (at least some) experimental data, but won't always be valid for other applications of this procedure.

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 $x_1, x_2, \cdots, x_i$, 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 negliglible.
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:

$$\langle x^k \rangle \equiv \frac{1}{N} \sum_{i=1}^{N} x_i^k \, p(x_i) \ , \tag{12.14}$$

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

$$\langle x^k \rangle \overset{N\to\infty}{\longrightarrow} \int_0^1 dx \, p(x) \, x^k = \int_0^1 dx \, x^k = \frac{1}{k+1} \ , \tag{12.15}$$

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

$$\mu = \langle x \rangle = \frac{1}{2} \ , \qquad \sigma = \sqrt{\langle x^2 \rangle - \mu^2} = \frac{1}{\sqrt{12}} \approx 0.2886 \ . \tag{12.16}$$

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*

$$C_k \equiv \frac{\langle x_{i+k} x_i \rangle - \langle x_i \rangle^2}{\langle x_i^2 \rangle - \langle x_i \rangle^2} \ , \tag{12.17}$$

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. While this hardly sounds random, it can be very useful because you can compare results between two codes using the same pseudo-random sequence. Most of the time we'll use a function called `random_seed` to generate the seed for us automatically.

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]

$$p(x) = \frac{1}{b-a} \theta(x-a)\, \theta(b-x) \ , \tag{12.18}$$

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

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x-\mu)^2/2\sigma^2} \ . \tag{12.19}$$

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

$$R^2 \approx \langle (\Delta x_1 + \Delta x_2 + \cdots + \Delta x_N)^2 + (\Delta y_1 + \Delta y_2 + \cdots + \Delta y_N)^2 \rangle \ , \tag{12.20}$$

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

$$\langle \Delta x_i \Delta x_j \rangle = \langle \Delta y_i \Delta y_j \rangle = 0 \quad \text{if } i \neq j \ , \tag{12.21}$$

which means that all the cross terms average to zero in Eq. (12.20). Now for any i, the average of $\Delta x_i^2 + \Delta y_i^2$ will be the same, so

$$\langle \Delta x_i^2 + \Delta y_i^2 \rangle \equiv \langle r^2 \rangle \quad \text{for any } i \ . \tag{12.22}$$

Then

$$R^2 \approx N \langle r^2 \rangle \tag{12.23}$$

or

$$R \approx \sqrt{N} r_{\rm rms} \ , \tag{12.24}$$

where $r_{\rm rms}$ 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. (12.24) in three or higher dimensions. To summarize, the total (absolute) distance covered after N steps is $N r_{\rm rms}$, but the *net* radial distance from the origin of the walk is, on average, only $\sqrt{N} r_{\rm rms}$ [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 12):

1. Choose
$$\Delta x = \cos\theta \qquad \Delta y = \sin\theta \tag{12.25}$$
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 $[-\sqrt{2}, \sqrt{2}]$. *What interval would you choose if you wanted* $r_{\rm rms} = 1$ *?*
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 12 zip file is the code `random_walk.cpp`, which is basically a C code with C++ input and output. Also included in the zip file (in a subdirectory) 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-paste 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 12 guide.

e. Monte Carlo Integration: Uniform and Gaussian 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 the places to evaluate an integrand at random is better than picking in an organized fashion, it is nevertheless true!

Our discussion here is based on Ref. [1]. (See Refs. [2] and [3] for alternative discussions and additional details.) 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

$$\langle f \rangle = \frac{1}{b-a} \int_a^b dx\, f(x) \;, \tag{12.26}$$

which implies the integral I can be calculated from

$$I = \int_a^b dx\, f(x) = (b-a)\langle f \rangle \;. \tag{12.27}$$

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

$$\int_a^b dx\, f(x) \approx (b-a)\frac{1}{N}\sum_{i=0}^{N-1} f(x_i) \equiv (b-a)\langle f \rangle \;. \tag{12.28}$$

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,

$$\int_a^b dx \int_c^d dy\, f(x,y) = (b-a)(d-c)\langle f \rangle \approx (b-a)(d-c)\frac{1}{N}\sum_{i=0}^{N-1} f(x_i, y_i) \;, \tag{12.29}$$

where the x_i are distributed uniformly in $[a, b]$ and the y_i uniformly in $[c, d]$. 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]

$$\sigma_f^2 \equiv \langle f^2 \rangle - \langle f \rangle^2 \tag{12.30}$$

$$= \frac{1}{N}\sum_{i=0}^{N-1} f(x_i)^2 - \left(\frac{1}{N}\sum_{i=0}^{N-1} f(x_i)\right)^2 \;, \tag{12.31}$$

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,

$$\langle I \rangle_M = \frac{1}{M}\sum_{j=0}^{M-1} \langle f \rangle_j \;, \tag{12.32}$$

and the variance for $M = N$ is

$$\sigma_N^2 = \frac{1}{N}\left[\left\langle \left(\frac{1}{N}\sum_{i=0}^{N-1} f(x_i)\right)^2 \right\rangle - \left(\left\langle \frac{1}{N}\sum_{i=0}^{N-1} f(x_i) \right\rangle\right)^2\right] \;. \tag{12.33}$$

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

$$\sigma_N^2 \approx \frac{1}{N}\left(\langle f^2\rangle - \langle f\rangle^2\right) \ . \tag{12.34}$$

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

$$I = \int_a^b dx\, f(x) = \int_a^b dx\, w(x)\, \frac{f(x)}{w(x)} \tag{12.35}$$

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

$$I = \left\langle \frac{f}{w} \right\rangle \approx \frac{1}{N}\sum_{i=0}^{N-1} \frac{f(x_i)}{w(x_i)} \ . \tag{12.36}$$

This approach can dramatically reduce the variance. If the function $f(x)$ contains a Gaussian, then we can choose $w(x)$ to be the same Gaussian, greatly improving our estimate over uniform sampling (which means $w(x)$ is a constant). We will test this using a GSL routine in Session 12. If we don't have an obvious choice for $w(x)$, we can develop it *adaptively.* The GSL routines for adaptive Monte Carlo integration you will try in Session 12 carry out this strategy. More on this next time!

Another method described in Ref. [1] is a *variance reduction* or *subtraction technique.* The idea is to construct a flatter function to integrate via Monte Carlo. If we can find a function $g(x)$ that is close to $f(x)$ in $[a, b]$ and whose integral we know:

$$|f(x) - g(x)| \le \epsilon \quad \text{for } a \le x \le b \ , \tag{12.37}$$

and

$$\int_a^b dx\, g(x) = J \ , \tag{12.38}$$

then we can use

$$\int_a^b dx\, f(x) = \int_a^b dx\, [f(x) - g(x)] + J \ , \tag{12.39}$$

and the second integral has a (much) smaller variance than the original, so Monte Carlo integration should be (much) more effective.

f. References

[1] R.H. Landau and M.J. Paez, *Computational Physics: Problem Solving with Computers* (Wiley-Interscience, 1997). [See the 6810 info webpage for details on a new version.]

[2] M. Hjorth-Jensen, *Computational Physics* (2013). These are notes from a course offered at the University of Oslo. See the 6810 webpage for links to excerpts.

[3] W. Press *et al.*, *Numerical Recipes in C++*, 3rd ed. (Cambridge, 2007). Chapters from the 2nd edition are available online from http://www.nrbook.com/a/. There are also Fortran versions.