Module III

Module 3: Random numbers and Monte Carlo methods

When we talked about algorithmic complexity, we thought about time, space, and parallelization as fundamental resources. There is another, more subtle resource: *randomness*. If we allow our algorithms access to a source of random numbers, we can unlock solutions to problems that would otherwise be computationally intractable. We can think of the ability to make a random choice as a computational primitive that, like the introduction of a `for` loop or an `if` statement, fundamentally expands what we can easily achieve. We can trade the guarantee of a deterministic answer for a high-quality statistical one, designing algorithms that get around the "curse of dimensionality" by sampling the problem space instead of trying to exhaustively explore it.

Figure III.1: Caravaggio's *The Cardsharps*, depicting the use of a not-so-random number generator. (Painting: c. 1594. Oil on canvas. Kimbell Art Museum, Fort Worth. Image via Google Arts & Culture)

You might worry: if our computers are fundamentally deterministic machines, where will we find a source of random numbers? We'll start this module by exploring pseudorandom number generators (PRNGs), which are deterministic algorithms that aim to produce long sequences of numbers that appear statistically indistinguishable from a truly random sequence. This apparent limitation is a benefit for scientific studies: by specifying a "seed" we can get our PRNGs to reproduce exactly the same sequence of "random" numbers every time. This means we can both use randomness as a resource *and* guarantee reproducible results.

With this tool in hand, we'll cover classic applications in physics, ranging from direct use in estimating numerical integrals to Markov Chain Monte Carlo (MCMC) methods for simulating particles in space or spins on a lattice. We'll close by combining the Hamiltonian dynamics from Module II with these MCMC methods to build a Hamiltonian Monte Carlo algorithm and apply it to one of the central tasks of modern science: Bayesian parameter inference.

For a deeper dive into the methods and physics discussed in this module, consider the following references [2, 3, 49].

Chapter 11

Pseudorandomness and Monte Carlo integration

What does randomness mean in a deterministic computer, and how can we harness this peculiar resource to solve concrete problems? We'll first explore the first question by looking at simple pseudorandom number generator – not because we should ever use it for research, but because seeing how it works (and how it fails) will help us understand the nature of the tools we're relying on. We'll also see, in a recurring theme, just how close we are to scraping the boundaries of current research. The second half of the chapter will focus on a classic application that addresses the second question. There we'll use randomness to estimate numerical integrals that might otherwise take the age of the universe to deterministically evaluate.

11.1 Pseudorandom number generators

A true random number generator would be a physical device that draws on a truly unpredictable process – perhaps one that watches for the radioactive decay of a sample. These "hardware random number generators" or "true random number generators" have a long history[95]. Such devices have value, but (1) they typically produce random samples at a rate that is far too low for our purposes and (2) by their very nature they are inherently non-reproducible. Instead, we'll rely on PRNGs – a combination of data and algorithm that produces a completely reproducible sequence of apparently random numbers given an initial state. The quality of PRNGs is judged by how well the output sequence passes a battery of statistical tests: to what extent and in which ways does that sequence have the properties you would expect of a true RNG?

11.1.1 A simple PRNG: linear congruential generators

At their core, all PRNGs are state machines: they hold an internal state, and each time you ask for a number they perform some mathematical operation to first update their state and then

[95] One early example used a disk spinning at some high rate (which could "if necessary, be made constant to a high degree of approximation by means of a tuning fork"!) in the dark. The disk was periodically lit up so that a human could try to transcribe the digit they happened to see every three or four seconds. This was a hilarious method, but it was explicitly and favorably compared with selecting "random" digits from a telephone book [50].

transform that state into some output. One of the oldest and simplest of these is the Linear Congruential Generator (LCG) [51]. It generates a sequence of integers with the following simple recurrence relation:

$$X_{n+1} = (aX_n + c) \mod m. \tag{11.1}$$

Here X_n is the current internal state of the generator, and the constants a, c, and m are parameters the describe different LCGs. The choice of these "magic numbers" is critical: poor choices lead to a very poor quality PRNG. An example of such a generator is shown in code block 11.1 – it uses a modulus $m = 2^{32}$ so that the properties of computer integer arithmetic automatically handles the modulus operation, but this is just a convenience.

```
mutable struct LCG
    state::UInt32
    a::UInt32
    c::UInt32
    # We'll use a modulus of 2^32, and directly use `/` below
end
function LCG()
    seed = time_ns() & 0xFFFFFFFF
    # numbers from cc65's `rand()` function
    return lcg(UInt32(seed), UInt32(16843009), UInt32(3014898611))
end

function rand_uint!(rng::LCG)
    rng.state = rng.a * rng.state + rng.c
    return Int(rng.state)
end
function rand_real!(rng::LCG)
    rng.state = rng.a * rng.state + rng.c
    return Float64(rng.state) / Float64(typemax(UInt32))
end
```

Code block 11.1: A simple LCG. A mutable struct holds the state and parameters of our generator, and a simple constructor uses (a) 32 bits from the current system time and (b) magic numbers for a and c (constants taken from cc65's `rand` function, released under the zlib license). The `rand_uint!` and `rand_real` functions mutate the state of the LCG, and return pseudorandom positive integers in $[0, 2^{32})$ and floats in $[0, 1)$, respectively.

The memory footprint of this RNG is small – just a single 32 bits for the state and 64 bits to hold the parameters – and the operations needed to generate random numbers are extremely fast. For convenience I've used the computer's internal time as the initial state of the generator – in an actual application you would choose a specific seed (or at least have a mechanism for recording the seed you used). Figure 11.1 shows the result of generating a few thousand real numbers with this generator. The results in the left panel look reasonable: at a minimum, the generator uniformly samples the space.

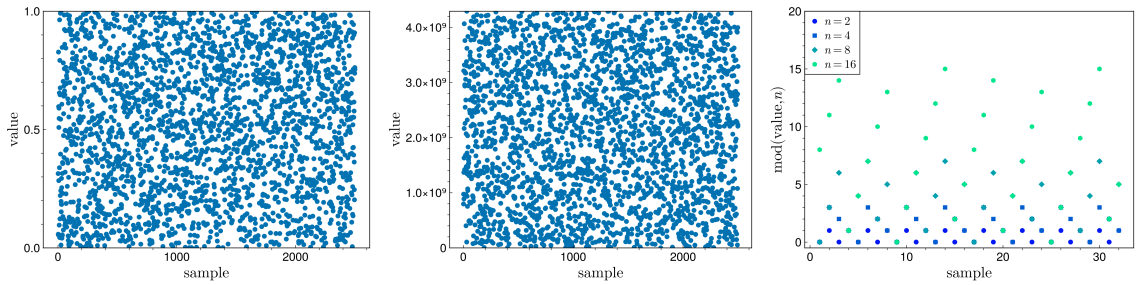

Figure 11.1: Samples from the LCG in code block 11.1. (Left) 2500 floating point samples, roughly uniformly distributed in the unit interval. (Center) 2500 integer samples, again roughly uniformly distributed in the space of `UInt32`s. (Right) Integer samples modulo small even numbers, where deterministic correlations in the integer sequence is seen.

Sadly, "uniformly samples the output domain" is not the only statistical property we usually want. The flaws of this particular LCG are on display when looking at the actual integer sequence of the internal state. Imagine using it to flip an imaginary coin; one method might be to assign "heads" to all of the even numbers and "tails" to all of the odd numbers. The result, shown in the right panel of Fig. 11.1, is highly suspicious (as, indeed, are all of the "modulo small even numbers" properties of the internal state). Problems like this were particularly problematic in early uses of LCGs for Monte Carlo problems [52].

11.1.2 Modern approaches

The precise failure mode of different LCGs depends sensitively on the magic numbers that define them – some have fixed points or short limit cycles, eventually repeating the same output sequence forever, some have the property of Fig. 11.1, in which not all of the bits of the state have the same statistical properties, and so on. Better approaches might start with an LCG but add nonlinearity in the function that maps the internal state to the output, or build nonlinearity into the state transformation itself. It will come as little surprise, then, that different programming languages have different default implementations for their `rand` function.

For isntance, python and many other languages use a "Mersenne Twister" generator [53], whose internal state and parameters take up a relatively whopping 20032 bits, but which has a cycle length (the number of output states that can be generated before the RNG repeats itself) of $2^{19937} - 1$. Numpy uses a "permuted congruential generator" [54], which uses a LCG with a power-of-two modulus but then adds a transformation between the state and the output to maintain the speed and light footprint of an LCG while avoiding their worst statistical properties. Julia uses the quite recent `Xoshiro256++` generator [55], which takes the 256 bits of internal state and performs particular sequences of xor, shift, and rotate operations on the bits. I bring all of this up to emphasize that generation of random numbers continues to be an active and evolving area of research: there are many competing demands on a PRNG – high statistical quality, fast, minimal internal state, possibly cryptographically secure – and different implementations balance those demands differently.

Do not roll your own RNG

Unless you are actively doing research on pseudorandom number generation, in your own code just use a well-tested library implementation for your RNG. Many a scientific "result" in the bad old days was caused by poor statistical properties or buggy implementations of a RNG.

11.1.3 Reproducibility and Seeding

As mentioned in the module introduction, the fact that PRNGs are deterministic should be viewed not as a flaw but an essential feature for scientific computing. By controlling the initial state, or "seed" of a PRNG, we can guarantee that we will get the exact same sequence of random numbers every time we run our code.

In practice, if we don't provide a seed most programming languages will initialize the PRNG with a source of entropy – a common choice is the one demonstrated in code block 11.1, in which the current nanosecond count on a system clock is used. This is why a program that uses this default will produce different outcomes each time it is run. You should think of this as non-negotiable in your scientific work. You can set the seed for Julia's default RNG like so:

```
julia> using Random

julia> Random.seed!(123321)
```

This will guarantee that the stream of numbers resulting from subsequent calls to `rand()` will be identical[96]. Explicitly setting the seed – or using a RNG to choose the seed but then recording the value used – is a basic practice for reproducible scientific work.

11.1.4 Generating non-uniform random numbers

If we were stuck with generating pseudorandom floats and ints uniformly in some range our toolbox would feel a bit limited. Many physical processes correspond to distributions that are not uniform, so is there a way to directly sample from some of these more interesting distributions?

Perhaps the most fundamental method is known as *inverse transform sampling*. The technique relies on the probability density function (PDF), $p(x)$, and its corresponding cumulative distribution function (CDF). The CDF, $P(x)$, gives the probability that a random variable X drawn from $p(x)$ will have a value less than or or equal to x:

$$P(x) = \int_{-\infty}^{x} p(x')\,\mathrm{d}x'. \tag{11.2}$$

[96] An important caveat is that the implementation details of the RNG may change between due to bug fixes, speed improvements, or algorithmic changes to Julia. Thus, the stream of numbers will be the same with a fixed seed on a fixed *version of Julia*.

By definition, the CDF is a monotonically increasing function whose range is between zero and one.

The key insight is that if we draw a random variable X from the distribution $p(x)$, then the transformed variable $y = P(X)$ will be uniformly distributed in the interval $[0, 1]$. Inverse transform sampling runs this logic in reverse: we generate a uniform random number $y \in [0, 1]$ – which we already know how to do – and then transform that into a random number x drawn from $p(x)$ by calculating $x = P^{-1}(y)$. As long as we can analytically calculate the inverse of the target CDF, we're done!

A canonical example of this is to generate samples from the exponential distribution. The probability distribution itself is parameterized by λ, and given by

$$p(x; \lambda) = \lambda e^{-\lambda x}, \quad \text{for } x \geq 0. \tag{11.3}$$

The CDF is found by directly integrating this up to x:

$$P(x) = \int_0^x \lambda e^{-\lambda x'} \, \mathrm{d}x' = 1 - e^{-\lambda x}. \tag{11.4}$$

Finally, we find the inverse function by setting $y = P(x)$ and solving for x:

$$y = 1 - e^{-\lambda x} \quad \Rightarrow x \quad = -\frac{1}{\lambda} \ln(1 - y).$$

We now have a simple algorithm: draw a uniform random number and apply this formula. In code (and using Julia's built in random number generator), this is as simple as:

```
using Random
function rand_exp(lambda)
    y = rand()
    return -log(1.0 - y) / lambda
end
```

This technique is powerful, but it's limited to distributions where the CDF can be easily inverted in closed form. An example of a common distribution without this property is the Gaussian distribution: its CDF involves the error function, which has no simple inverse. For this and other distributions, other techniques are required. Common ones involve variable transformation method (such as the Box-Muller transform [56], which generates pairs of uniformly distributed random numbers and maps them to pairs of Gaussian distributed random numbers) and rejection-sampling methods (which work by drawing samples from simple distributions and probabilistically accepting or rejecting them to match a target distribution). Eventually we'll see that the distributions we're interested in in the coming chapters are too complex for any of these methods to work well; for now our key takeaway is that our simple uniform PRNG can serve as the fundamental building block from which all other distributions can be constructed.

11.2 Application: Monte Carlo Integration

We now turn to the second question posed at the start of this chapter: how to use randomness to solve a deterministic problem? One of the most classic, straightforward, and powerful applications is Monte Carlo integration.

11.2.1 The Basic Idea: Throwing Darts

We, in fact, already did this in its basic form in Section 4.3, where we calculated π by "throwing darts" at a unit square and counting the fraction that landed inside an inscribed circle. The first generalization of this idea is to find the value of a one-dimensional definite integral, $\int_a^b f(x)\,\mathrm{d}x$, by computing the average value of the function over the interval and multiplying it by the length of the interval. The Monte Carlo approach estimates the average by sampling the function at N points chosen at random uniformly throughout the interval:

$$\int_a^b f(x)\,\mathrm{d}x \approx (b-a)\cdot\frac{1}{N}\sum_{i=1}^{N} f(x_i). \tag{11.5}$$

11.2.2 Convergence and the curse of dimensionality

Why is this a good idea? Mathematically, the central limit theorem tells us that as long as we have a reasonable, well-behaved functions, the error in our estimate of the average value will behave like the standard error of the mean of a random sample. As a result, the error of our estimate of the integral with N samples will scale as

$$\mathrm{Error}_{\mathrm{MC}} \propto \frac{1}{\sqrt{N}}. \tag{11.6}$$

This is an error which converges slowly: to reduce the error by a factor of 10, we need to generate 100 times as many samples. This should be contrasted with a deterministic method like even the simple trapezoid rule. There, splitting the interval into N subintervals (i.e., in a setting in which we perform the same number of *function evaluations* as in the MC estimate above), the error scales as

$$\mathrm{Error}_{\mathrm{trap.}} \propto \frac{1}{N^2}. \tag{11.7}$$

Perhaps the question above should have said "*Is* this a good idea?" The power becomes apparent when we move to higher dimensions. Consider integrating a polite function over a d-dimensional unit hypercube, $[0,1]^d$. The error in the Monte Carlo approach is still controlled by the central limit theorem, and has an error which remains $\mathcal{O}(N^{-1/2})$. Similarly, the error in the trapezoid rule continues to be set by *the size of the discrete subregions*: we have N function evaluations, but them must be split into a grid composed of M points along each dimension: $N = M^d$. Since the error scales like $1/M^2$, we find that the error for our integral scales like $\mathcal{O}(N^{-2/d})$. Said another way: achieving a certain precision in $d = 1$ with the trapezoid rule might require $N = 10$ points with the trapezoid rule and $N = 10^4$ points with the MC approach. To get that same precision in 10 dimensions with the trapezoid rule would require $N = 10^{10}$

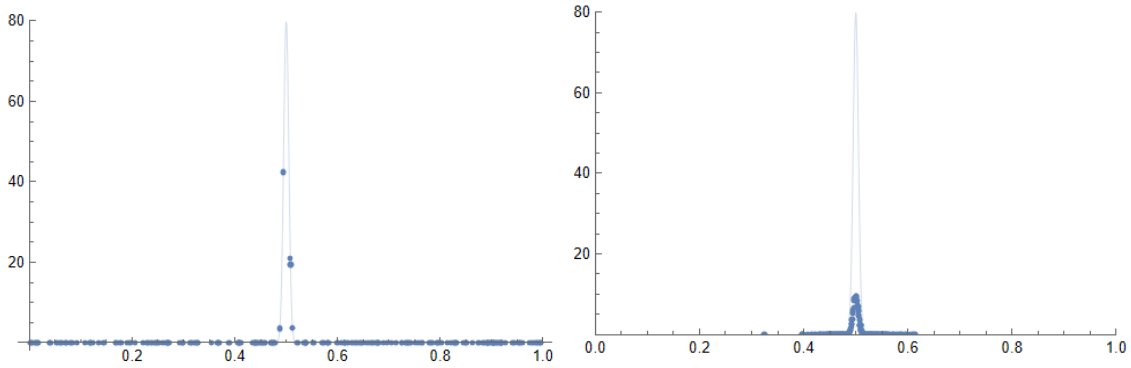

Figure 11.2: (Left) A narrow Gaussian function (thin line) together with 200 uniformly randomly sampled locations in the unit interval. (Right) The same function, but now the points are themselves sampled from a Gaussian distribution and reweighted accordingly.

function evaluations, whereas the MC method still needs only of order 10^4 (albeit with a larger prefactor, often one which scales *linearly* with d).

Thus, as d increases, the deterministic method becomes *exponentially* slower if we try to maintain the same accuracy. This catastrophically bad scaling in high dimensions is famously called the "curse of dimensionality." For typical, well-behaved functions, a dimensionality of $d = 4$ is the tipping point where the effectiveness of the deterministic vs random methods flips. For the extraordinarily high-dimensional integrals that appear in fields like statistical mechanics, MC methods are not just a better option, they are the *only* option.

11.3 Importance sampling

The Monte Carlo approach is incredibly powerful, but there is something a bit brute-force and, perhaps, inelegant about how it uniformly samples random numbers in the interval of interest. We often know at least some things about the functions we are trying to integrate, and for functions with sharp peaks or other localized features, the majority of the N samples we use might be "wasted" in regions where the integrand is essentially zero. Multiple evaluations of the function in those regions don't contribute very much to our understanding of the integral, but they still cost computational time and slow down the convergence.

To be concrete, consider the Monte Carlo integration of a narrow Gaussian function, like the one shown in Fig. 11.2. With the direct MC method, we randomly sample the value of the function throughout the unit interval, but only a small fraction of these samples actually contribute to the integrand. The asymptotic scaling of the error here hasn't changed, but the prefactor has gotten massively worse, as our estimate of the integral is dominated by the statistical noise from the handful of lucky "hits" when we sample in the correct region.

11.3.1 Biasing and re-weighting

We can do better by no longer sampling uniformly but instead focusing our computational effort where it matters – the is the core idea behind *importance sampling*. Instead of drawing samples

from a uniform distribution, we instead draw them from a different probability distribution, $p(x)$, which puts more of its weight in regions where the integrand $f(x)$ deviates from zero.

In order not to let this non-uniform distribution bias our estimate, we rewrite our original integral like so:

$$I = \int \frac{f(x)}{p(x)} p(x)\,\mathrm{d}x. \tag{11.8}$$

We can interpret this as the expectation value of the function $g(x) = f(x)/p(x)$ with respect to the probability distribution $p(x)$. This lets us write our MC estimator as

$$I \approx \frac{1}{N}\sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)}, \quad \text{where } x_i \text{ is drawn from } p(x). \tag{11.9}$$

Algorithmically, this is straightforward. We draw a sample x_i from the "importance" distribution $p(x)$, and then add the value $f(x_i)/p(x_i)$ to a running sum. The $1/p(x_i)$ is the "re-weighting" factor that corrects for the biased sampling by giving less weight to samples drawn from high-probability regions and more weight to to the rare sample drawn from low-probability regions. This is visualized in the right panel of Fig. 11.2 – we again draw 200 points but now from a narrow Gaussian distribution close to our target distribution, and re-weight those points accordingly. The resulting estimate is, in fact, on average much closer to the correct answer for a fixed number of samples.

Why is this better? The goal of importance sampling is to reduce the variance in our estimate. A low-variance estimator is one that converges quickly to the correct answer with few samples, and it can be shown that the variance of our estimator in Eq. (11.9) is minimized with a particular choice for our sampling distribution:

$$p_{\text{ideal}}(x) = \frac{|f(x)|}{\int |f(x)|\,\mathrm{d}x}. \tag{11.10}$$

This equation says that the perfect sample distribution is the one proportional to the absolute value of the function we are trying to integrate. In the case where $f(x)$ itself is non-negative, it even tells us that we could get the exact answer with a single sample!

Remarkable, but also perhaps a bit circular? Of course, that's only because constructing this perfect sampling distribution requires knowing how it is normalized, $\int |f(x)|\,\mathrm{d}x$, and this is basically the integral we are trying to estimate in the first place. Thus, the are of importance sampling is not finding the perfect distribution, but to choose a simpler distribution $p(x)$ that (a) we can easily sample from and (b) mimics the shape of our original function better than a uniform distribution does.

11.3.2 The bridge to statistical physics

This insight leads directly to the heart of computational statistical mechanics. Calculating a thermodynamic observable for an equilibrium system corresponds to a evaluating a high-dimensional integral over all possible states of the system. For a system at temperature T, the probability of being in a particular state μ with energy $E(\mu)$ is given by the famous Boltzmann distribution,

$$p(\mu) \propto e^{-\beta E(\mu)}, \tag{11.11}$$

where $\beta = (k_B T)^{-1}$. Let's heuristically think of this as the *ultimate importance distribution*, provided to us by nature itself. It tells us that the important states – the ones that contribute the most to any thermodynamic average – are the low-energy states. Our goal, then, is to perform a Monte Carlo calculation by drawing samples from this distribution.

And yet, for any non-trivial system, the space of possible states is unimaginably vast – considering a collection of N particles in a box of volume V, every different arrangement of particles in that volume is its own state. The volume to the power of avogadro's number is a terrifyingly large number, and that's before we say anything about the momentum degrees of freedom in the system. How could we possibly draw independent samples directly from such a high-dimensional monstrosity? In a related challenge, the normalization constant for the Boltzmann distribution is the partition function,

$$Z = \sum_{\mu} e^{-\beta E(\mu)}. \tag{11.12}$$

Sum over *all* of the immense number of these states? How?

This all seems... hard. How will we perform importance sampling with a distribution we don't know how to sample from, and whose normalization constant we can't hope to compute? This is the problem that the Metropolis algorithm [38], and the field of Markov Chain Monte Carlo (MCMC) was invented to solve – that's what we'll turn to in the next chapter.

Chapter 12

The Metropolis algorithm

In the previous chapter we ended with a powerful but puzzling idea: the key to Monte Carlo integration is importance sampling – a way of focusing our computational effort on the "important" regions of a problem – and the optimal distribution to use for the importance distribution is one which already contains information about the very integral you are trying to perform.

For a system in thermal equilibrium nature hands us the Boltzmann distribution, $p(\mu) \propto e^{-\beta E(\mu)}$. To calculate any thermodynamic averages for our system we need to draw samples from this distribution, but how? It is defined over a state space of terrifyingly, stare-into-the-abyss large dimensionality. We typically cannot even calculate its normalization constant (the partition function Z), let alone draw independent samples from it

This chapter introduces a solution which is a workhorse of computational statistical physics: the *Metropolis* algorithm. This is a brilliant method that lets us generate a sequence of states that, in the long term, correctly samples from the Boltzmann distribution even without knowing Z. We'll first develop this algorithm from first principles, and then apply it to canonical model systems.

12.1 Markov chain Monte Carlo

The conceptual leap is actually to abandon the idea of drawing *independent* samples altogether. Instead we will try to construct a random walk – which we'll call a *Markov chain* – that explores the state space of our system. A Markov chain is a sequence of states in which the next state depends only on the current state (and not any of the states that came before); it is a "memoryless" process. The challenge will be to design the rules of the random walk so that, in the long run, the fraction of the time the system spends in any particular state μ is directly proportional to the true probability of being in that state, $p(\mu)$.

The idea of constructing these memoryless sequence of states goes back to the work of Markov in the early 20th century [57]. In an early application, Markov analyzed 20000 characters from the writing of Pushkin's poem Eugene Onegin [58]. His method was to first define the "states" of his system – is the current character a vowel or a consonant – and then empirically measure the probability of moving between them. This creates a *transition matrix*, T, where T_{ij} is the probability of transitioning from state i to j.

For example, we can analyze the complete works of Shakespeare to construct the transition

From \ To	a	e	i	o	u	C
a	$\frac{957}{312\,083}$	$\frac{1176}{312\,083}$	$\frac{12\,809}{312\,083}$	$\frac{404}{312\,083}$	$\frac{4429}{312\,083}$	$\frac{292\,308}{312\,083}$
e	$\frac{15\,519}{162\,506}$	$\frac{7881}{162\,506}$	$\frac{19\,777}{487\,518}$	$\frac{5527}{243\,759}$	$\frac{621}{81\,253}$	$\frac{127\,587}{162\,506}$
i	$\frac{4179}{136\,025}$	$\frac{5619}{136\,025}$	$\frac{297}{54\,410}$	$\frac{5166}{136\,025}$	$\frac{277}{27\,205}$	$\frac{237\,867}{272\,050}$
o	$\frac{1178}{83\,535}$	$\frac{571}{66\,828}$	$\frac{1228}{83\,535}$	$\frac{7211}{167\,070}$	$\frac{60\,179}{334\,140}$	$\frac{12\,353}{16\,707}$
u	$\frac{6}{217}$	$\frac{5581}{137\,795}$	$\frac{594}{27\,559}$	$\frac{852}{137\,795}$	$\frac{159}{137\,795}$	$\frac{124\,423}{137\,795}$
C	$\frac{82\,563}{842\,369}$	$\frac{13\,425}{76\,579}$	$\frac{76\,699}{842\,369}$	$\frac{297\,076}{2\,527\,107}$	$\frac{66\,532}{2\,527\,107}$	$\frac{1\,242\,688}{2\,527\,107}$

Figure 12.1: The transition matrix between some vowels and any consonant, as determined by analyzing the complete works of Shakespeare [61]. The entries of the dominant eigenvector predict the empirical distribution of vowels and consonants in the source within a precision of 10^{-7}. However, a random walk using this matrix produces samples like: "gsfsaedptoawfjtqxiiz." Very poetic, but also demonstrative of the fact that the model captures "equilibrium statistics" but not the long-range correlations of real language.

matrix shown in Fig. 12.1, which shows the probability of transitioning in the text from the vowels $\{a, e, i, o, u\}$ to each other or to any consonant. If we represent the current state as the probability vector (e.g., the vector $\{1, 0, 0, 0, 0, 0\}$, corresponding to "the current state is definitely the letter a"), we can find the probability of the next state simply by multiplying by the transition matrix. The power of the Markov chain is that is we apply this matrix repeatedly, the state vector will inevitably converge to a unique *stationary distribution* – a vector that no longer changes upon further multiplication by T. This stationary distribution is, by definition, the dominant eigenvector of the transition matrix [59, 60] and, as Markov showed, it almost perfectly predicts the overall frequency of vowels and consonants in the text.

This result is both powerful and surprising. As the gibberish in the caption shows, our simple Markov model has failed to learn much about the actual structure of the English language. But it has perfectly succeeded at a more narrow task: finding the correct equilibrium distribution of the characters themselves. This is an important lesson: the Markov chain may not be a perfect replica of the system; it is a carefully crafted tool designed to reproduce some of the correct statistical properties in the long time limit.

The process illustrated above – using a Markov chain to generate samples from a Monte Carlo calculation of some property – is known as Markov Chain Monte Carlo (MCMC). To emphasis again: we will construct a simulation in which the state at step $n+1$ depends only on its state at step n, and not on the full history of the trajectory. The central task will be to define transition probabilities, $T(\mu \to \mu')$ for moving from state μ to state μ'. This set of probabilities must be carefully crafted so that the stationary distribution of the Markov chain (i.e., the distribution of states it settles into after running for a long time) is precisely the Boltzmann distribution.

12.1.1 Engineering the stationary distribution

We're making progress, but there are still major questions here. In the case of letter transition probabilities we just empirically measured the transition matrix – this was a descriptive model. But how do we turn a similar idea into a *prescriptive* method, in which we do not just measure but design transition rules that will guarantee the stationary distribution matches the target? A full treatment, along with the actual conditions under which this whole machinery will work, would require a deep dive into the theory of ergodic Markov chains. Instead, we'll think through some clear necessary conditions, and the physical shortcut that will make our algorithmic approach more tractable.

One condition that must clearly be satisfied is the condition of *global balance*. Suppose you already have a stationary distribution, in which the probability of observing your system in state i perfectly matches the true equilibrium probability, which we'll denote π_i. In order for the continued application of the transition matrix not to take you *away* from the stationary distribution, the total flow of probability into and out of each state must be balanced: mathematically, global balance corresponds to

$$\sum_i T_{ij}\pi_i = \pi_j = \sum_k T_{jk}\pi_k. \tag{12.1}$$

This is a very general kind of "no net flux of probability" condition, but it is in general hard to use to to actually build the transition matrix.

We can make our life easier by imposing a stricter requirement: rather than having this global balance of inflowing and outflowing probability, we demand that the total flow of probability between every pair of states is balance. This *detailed balance* condition – which clearly also satisfies the demands of global balance – is

$$T_{ij}\pi_i = T_{ji}\pi_j. \tag{12.2}$$

Markov chains satisfying detailed balance are called reversible Markov chains. And indeed, when modeling physical systems with microscopically reversible dynamics, there is a natural justification for adopting the additional constraints that detailed balance imposes.

12.1.2 The Metropolis-Hastings algorithm

With that as background, let's return to our target of studying thermodynamical systems that can reside in microstates μ. Although there is a relationship, the approach we're about to outline marks a fundamental shift from the importance-sampling strategy. As we'll see, the Metropolis [38] algorithm gives up on the idea of drawing independent samples from a simple distribution and then reweighting to correct for bias; instead it provides a way to generate a sequence of correlated samples drawn directly from the correct, complex distribution.

So: the Metropolis algorithm splits the task of constructing the transition rule $T(\mu \to \mu')$ into two steps: a *proposal* and an *acceptance/rejection* step. The proposal step selects a potential new state, μ', starting from the current state μ. This is done according to some probability distribution $g(\mu \to \mu')$. For many physical problems we have the luxury of choosing simple symmetric proposal distributions – uniformly randomly picking a spin to flip or a particle to displace, for instance – and for such distributions we have $g(\mu \to \mu') = g(\mu' \to \mu)$. The

next step decides whether to accept the proposed state update or not, which happens with probability $A(\mu \to \mu')$.

In combination we have a total transition probability of $T(\mu \to \mu') = g(\mu \to \mu') \cdot A(\mu \to \mu')$. Substituting this into the detailed balance condition for a symmetric proposal distribution gives

$$p(\mu) \cdot g(\mu \to \mu') \cdot A(\mu \to \mu') = p(\mu') \cdot g(\mu' \to \mu) \cdot A(\mu' \to \mu) \quad \Rightarrow \quad \frac{A(\mu \to \mu')}{A(\mu' \to \mu)} = \frac{p(\mu')}{p(\mu)}. \tag{12.3}$$

That is: for this subdivision of the problem, imposing detailed balance corresponds to a condition on the ratio of the acceptance probabilities.

A brilliantly simple choice was made by Metropolis et al.:

$$A(\mu \to \mu') = \min\left(1, \frac{p(\mu')}{p(\mu)}\right), \tag{12.4}$$

i.e., proposals to more probable states are always accepted, and proposals to less probable states are accepted some of the time[97] Intuitively, this is a strong starting point: such a Markov chain can easily get near local minima, but also with some probability escape them and explore the entire state space.

For our physical system and for nature's importance function, $p(\mu) = \exp(-\beta E(\mu))/Z$, the ratio of probabilities is

$$\frac{p(\mu')}{p(\mu)} = \frac{Z}{Z} \frac{e^{-\beta E(\mu')}}{e^{-\beta E(\mu)}} = e^{-\beta \Delta E}.$$

The partition function which we didn't know in the first place cancels out, leaving us with a powerful result: the acceptance probability depends only on the change in energy, which is often a local and easily computable quantity.

This simple acceptance rule represents an interesting shift in our strategy relative to the importance-sampling problem we ended Chapter 11 with. There, direct importance sampling required us to draw samples from a simple distribution, $q(\mu)$, and reweight them to evaluate an integral. In the context of computing ensemble averages with respect to a Boltzmann weight, $\langle A(\mu) \rangle = \int A(\mu) p(\mu) \, \mathrm{d}\mu$, a resulting estimator would be

$$\langle A(\mu) \rangle \approx \frac{1}{N} \sum_{i=1}^{N} \frac{A(\mu_i) p(\mu_i)}{q(\mu_i)}.$$

The problem is that, well, we have no good way to draw independent samples from $p(\mu)$ itself, and no idea of a good, simple $q(\mu)$ to sample from instead.

The Metropolis algorithm solves this problem with a very different strategy. Instead of asking, "How do we choose a reasonable q and then correct for sampling from the wrong distribution?", it provides a mechanism for generating *correlated* samples $\{\mu_i\}$ that are drawn

[97] The assumption of a symmetric proposal distribution is the distinction between the original Metropolis algorithm and the more general "Metropolis-Hastings" algorithm [62]. The Hastings generalization allows for asymmetric proposals by modifying the acceptance ratio we're about to derive to include the proposal probabilities, with an acceptance ratio given in Eq. (13.1).

directly from the true distribution. It achieves the "ideal" sampling scenario, and our estimator for any thermodynamic average becomes a simple, unweighted sum:

$$\langle A \rangle \approx \frac{1}{N} \sum_{i=1}^{N} A(\mu_i). \tag{12.5}$$

The price we pay is that our samples are no longer independent; this introduces subtleties we will address in the next section. But with that said, we can now state the complete algorithm for our Markov chain:

1. Start in state μ_i with energy $E(\mu_i)$.
2. Propose a trial move to a new state μ'.
3. Evaluate ΔE.

4a. If $\Delta E \leq 0$, accept the move: $\mu_{i+i} = \mu'$.

4b. If $\Delta E > 0$, generate a uniform random number $r \in [0, 1)$. If $r < e^{-\beta \Delta E}$ accept the move: $\mu_{i+i} = \mu'$. If not, reject the move and set $\mu_{i+1} = \mu_i$.

5 . Repeat.

12.2 Case Study I: The Ising model

For our first case study using the technology above, let's study the Ising model, which is typically introduced as a toy model for magnetic systems. Schematically depicted in Fig. 12.2, the Ising model consists of a set of N spins, $\mu = \{s_i\}$. There are 2^N possible states μ, and so clearly for a lattice of even quite modest size there are more states than we could even enumerate, let alone actually actual work with. This is a canonical use case for MCMC-based sampling: we'll use the Metropolis algorithm to generate a sequence containing a tiny fraction of the possible states, but we will generate them with the *correct* distribution of probabilities.

These spins interact with their neighbors, and may also be coupled to an external magnetic field. The Hamiltonian governing the spins is

$$\mathcal{H} = -J \sum_{\langle ij \rangle} s_i s_j - h \sum_i s_i, \tag{12.6}$$

where J determines the strength of the spin-spin interaction, h is the external field, and $\sum_{\langle ij \rangle}$ indicates a sum over all spins i and j that are neighbors of each other. This model has many variations[98], but for now we will focus on the simplest case of nearest-neighbor spins with constant J on a hyper-cubic lattice.

[98] Is J the same for all pairs of spins? What lattice do the spins live on? Should neighbors only be nearest-neighbor, or should we include other interactions?

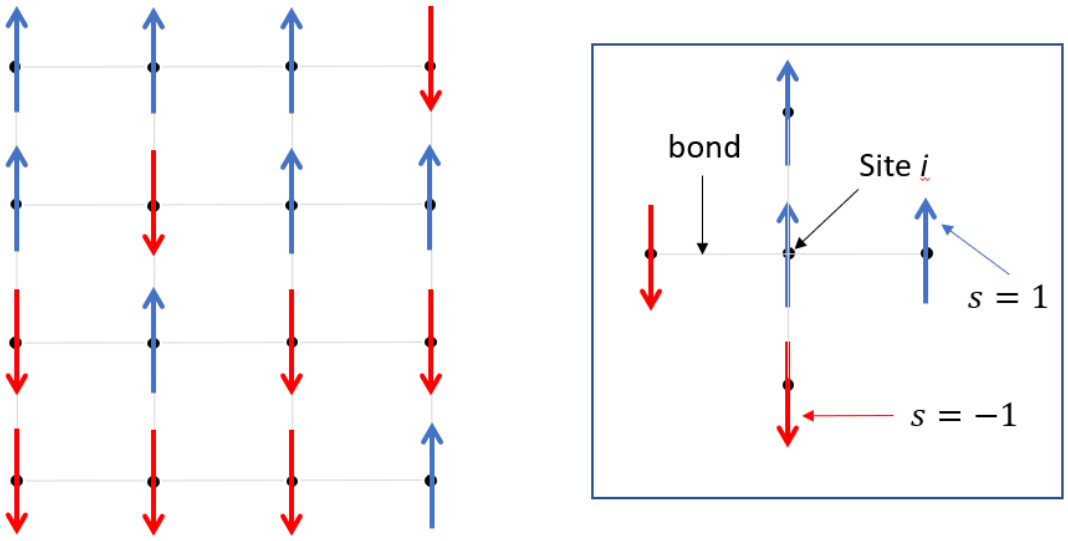

Figure 12.2: (Left) A schematic representation of the square lattice Ising model. (Right) On each site "spins" are either up or down, corresponding to $s_i = \pm 1$. Sites with a bond between them are included in the term which sums over neighboring sites in the Hamiltonian.

The Ising model

Although seemingly simple, this model is connected to some of the deepest insights in 20th century physics. It was introduced by Wilhelm Lenz [63], and Lenz's student's dissertation constructed the exact solution for a one-dimensional lattice [64]. Ising stopped doing research, partly because he thought he had proved that the model he spent so much intellectual effort on had no physical relevance[a]; only much later in life did he find out that this model had become incredibly influential and, indeed, a cornerstone that helped build modern statistical physics.

[a] His career was also tragically cut short by religious persecution. He was forbidden from teaching and conducting research, fled Germany, and performed forced labor dismantling the Maginot line railroad before eventually emigrating to the US [65].

12.2.1 A top-down design

Let's implement a Metropolis Monte Carlo simulation of the simple Ising model on a hypercubic lattice, but lets do so in a way that anticipates some of the natural ways we might want to extend it. Just as in Chapter 8, we'll think about a flexible top-down design: we'll first think about how we want to structure a single *metropolis_step!*, and what that design contract in turn guides the rest of our lower-level implementation. Throughout we'll lean on multiple dispatch to write general code that nevertheless efficiently specializes on the particular structures we want to work with.

We begin by thinking through what the general statement of the Metropolis algorithm tells us we will need. Clearly we will need something for the state of whatever system we have under

study, and we will need a way of implementing the proposal function, g, that can suggest a state μ' given the state μ_i. We also need a way of calculating differences in energy – and hence energies in the first place, and then an implementation of the acceptance function A. Without even knowing that we have a lattice model in mind, we can already directly write this high-level description:

```
function metropolis_step!(system, hamiltonian, move::AbstractMCMove;
β=1.0)
    trial_move_info = propose_move(system, hamiltonian, move)

    ΔE = calculate_ΔE(system, trial_move_info, hamiltonian, move)

    if ΔE <= 0.0 || rand() < exp(-β * ΔE)
        accept_move!(system, trial_move_info, move)
    end
    return nothing
end
```

Code block 12.1: A possible implementation of a general step in a Metropolis Monte Carlo simulation.

12.2.2 System data structures

As before, the very first thing our high-level API tells us we need is a "system," and we could go in many different directions here. This is a key point in our work where we could try to write hyper-specialized code for the square lattice Ising model above, or try to write something extremely generic but not necessarily particularly performant, and so on. In this case, let's allow ourselves to be guided by the knowledge that we'll certainly be studying lattice-based spin systems, but we *also* have spent time thinking about particle based systems. It might be interesting to be able to compare both an ODE- and Monte-Carlo-based approaches to a particle system, so even though this case study is for a lattice model, let's make sure we can handle different types of systems.

Not a problem. Let's begin by setting up some basic lattice functionality, and then build a system on top of that foundation. We'll start with an abstract lattice type, and some of the basic functions we will be sure to implement for any concrete lattice we want to study:

```
abstract type AbstractLattice end

function num_sites(lattice::AbstractLattice) end

function neighbors!(lattice::AbstractLattice, site::Int) end
```

Here we see the first concession to generality: in a fixed lattice there might not be a need for `neighbors!` function: the neighbors of each site in the lattice are fixed forever! Why, then, are

we defining a mutating function? It's because if we want to use our code to also work for a dynamical particle system, our eventual `calculate_ΔE` function needs an interface that can work for systems whose neighbors can change at every time step. This is a real-world design trade-off that both illustrates the "meet-in-the-middle" design process and the potential cost of abstraction: do we prioritze maximum performance for a specific case, or do we build a more general interface that can be reused for other problems?

```
struct HypercubicLattice{D} <: AbstractLattice
    dims::NTuple{D, Int}
    neighbors::Vector{Int}
    neighbor_list::Vector{Vector{Int}}
end

function HypercubicLattice(dims::NTuple{D, Int}) where {D}
  N = prod(dims)
  neighs = Vector{Int}(undef,2*D)
  neighbor_list = [Vector{Int}() for _ in 1:N]
  cartesian_indices = CartesianIndices(dims)
  linear_indices = LinearIndices(dims)
  for i in 1:N
    current_idx = cartesian_indices[i]
    for d in 1:D
      for offset in (-1, 1)
        mod_offset = mod1(current_idx[d] + offset,dims[d])
        neigh_idx = Base.setindex(current_idx, mod_offset, d)
        push!(neighbor_list[i], linear_indices[neigh_idx])
      end
    end
  end
    HypercubicLattice{D}(dims, neighs,neighbor_list)
end
num_sites(lattice::HypercubicLattice) = prod(lattice.dims)

function neighbors!(lattice::HypercubicLattice, site::Int)
    lattice.neighbors .= lattice.neighbor_list[site]
end
```

Code block 12.2: A concrete `Hypercubic` lattice with precomputed nearest neighbors. Please forgive the abuse of consistent indentation – I wanted all of these lines to fit onto a single line of text in these notes.

Code block 12.2 gives an implementation of a lattice of fixed (nearest-neighbor) topology that, nevertheless, is optimized to compute nearest-neighbor lists without allocations[99].

[99] Whether this is, in fact, the optimal approach might depend on the details. Perhaps precomputing neighbor

This is most of the work we need to do for now; with a lattice defined we can set up a simple system of spins that includes both the spins (of different possible types!) themselves and the lattice they live on.

```
abstract type AbstractSystem end

struct SpinSystem{T, L <: AbstractLattice} <: AbstractSystem
    spins::Vector{T}
    lattice::L
end
```

12.2.3 Monte Carlo moves

Next we could tackle the Hamiltonian and the corresponding energy function (which is the next argument in the `metropolis_step!` function), but since it's needed on the first line let's look ahead to the `move::AbstractMCMove` and `propose_move` functions. We discussed earlier that there is a tremendous flexibility in what the proposal function g in the Metropolis algorithm can actually be: it could propose completely new states uniformly regardless of the current state of the system, or it could propose the most minor changes to the current state, or anything in between. The choice will strongly influence *how quickly* the sequence of states converges to the stationary distribution, and how correlated the sequence is, but choosing a good proposal function is part of the art of Monte Carlo methods. We'll make sure we set up the technology to implement different proposed types of state changes, and then concretely implement the simplest possible MC move for an Ising model: picking any single spine and flipping its sign. The types might look like this:

lists is better, or perhaps using the logic of the lattice to recompute neighbor indices on the fly is better than memory lookups. As always, if you are interested in this level of optimization you should *measure and benchmark*.

```
abstract type AbstractMCMove end

struct SingleSpinFlip <: AbstractMCMove end

struct SpinFlipInfo
    index::Int
end

function propose_move(system::SpinSystem{T}, hamiltonian,
                      move::SingleSpinFlip)
    random_index = rand(1:length(system.spins))
    return SpinFlipInfo(random_index)
end

function accept_move!(system::SpinSystem{T}, move_info::SpinFlipInfo,
                      move::SingleSpinFlip) where {T <: Integer}
        system.spins[move_info.index] *= -one(T)
end
```

You can see that, while we were at it, we defined not only the idea of a "single spin flip" MC move, but also the information that gets returned from the `propose_move` function, and what it would mean to accept a proposed move.

12.2.4 Energy calculations

All that remains for a complete implementation are the structures that encode a Hamiltonian and the functions that compute energy differences! For the first, we can set up a standard type hierarchy of abstract and concrete ways of calculating the energy – concrete Hamiltonians are where we'll store the parameters themselves – along with the function we'll need to implement for each concrete type:

```
abstract type AbstractEnergy end
# function we'll need to define for each concrete Hamiltonian:
function calculate_ΔE(system::AbstractSystem, move_info,
                      h::AbstractEnergy, move::AbstractMCMove) end
```

For our nearest-neighbor Ising model, the corresponding concrete struct and implementation of the ΔE calculation when a spin is flipped is shown in code block 12.3.

And with that, we're done! We have a complete, still efficient implementation of a Metropolis simulation of the Ising model on a hypercubic lattice. It may have taken us slightly longer to build this than the monolithic script would have been, but still: in less than 100 lines of code we have a structure that, as we'll see in the next case study, is actually *very* easy to generalize to other systems!

```
struct IsingHamiltonian{T} <: AbstractEnergy
    J::T
    h::T
end

function calculate_ΔE(system::SpinSystem, move_info::SpinFlipInfo,
                      ham::IsingHamiltonian, move::SingleSpinFlip)
    neighbors!(system.lattice, move_info.index)
    spin_i = system.spins[move_info.index]

    neighbor_sum = sum(@view system.spins[system.lattice.neighbors])
    ΔE = 2.0 * spin_i * (ham.J * neighbor_sum + ham.h)
    return ΔE
end
```

Code block 12.3: A concrete `IsingHamiltonian` structure, and the corresponding (fairly simple!) calculation of how much the energy changes when a spin is flipped.

12.2.5 Analyzing MCMC data

We have a nice, modular framework, and running this simulation will produce a long sequence of states, $\{\mu_0, \mu_1, \mu_2, ...\}$. The central promise of the Metropolis method is that after a long enough number of iterations this sequence is a representative sample from the Boltzmann distribution. But how do we actually turn this raw sequence of microscopic configurations into meaningful measurements for macroscopic observables, such as the average magnetization $\langle M \rangle$? The naive approach would be to calculate the magnetization $M = \sum_i^N s_i$ for every state in our sequence and take the average. This is dangerously wrong.

The first problem is the same one we encountered in our particle-based simulations. Our simulation began from an artificial initial state – artificial because we probably did not pick a representative configuration that has a high likelihood of appearing in the steady-state distribution. As the simulation runs, the system must "relax" or "thermalize" as it moves from this unusual initial state to a statistical steady state. If we were to include this early set of states in our average, the memory of the initial conditions would bias the final result, but the solution is simple: as before, we discard these initial samples. As before, the standard practice is to monitor a bulk observable as a function of the number of Monte Carlo steps, measure a timescale associated with the decay of this observable to its equilibrium value (Fig. 10.3, and discard all states in the first several multiples of this time scale.

The second problem is more subtle. After discarding the initial equilibration data, it would be tempting to take the remaining samples and then calculate the average and standard error of the mean. But this is also wrong: by their very construction, Markov chains generate a sequence of *correlated* states[100]. For instance, in the single-spin-flip version of the code above, state μ_{i+1} is either *exactly the same* as state μ_i, or it is different in the value of exactly one spin.

[100] Indeed, this is a live issue when studying molecular dynamics: configurations separated by a short time are *also* highly correlated with each other.

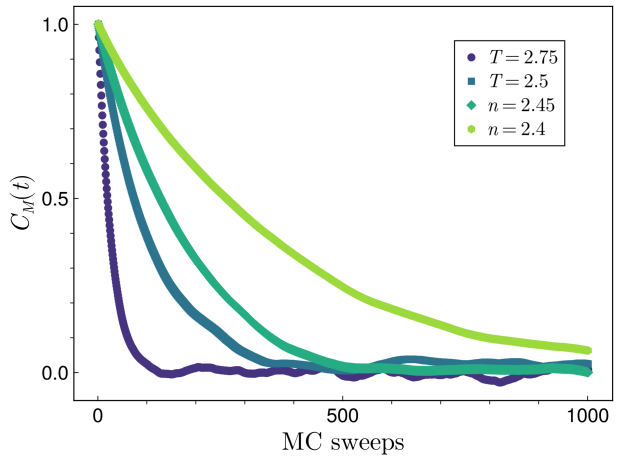

Figure 12.3: The autocorrelation function in the two-dimensional Ising model with $J = 1$, $h = 0$, at various temperatures, as measured via MC simulations on a 50×50 lattice.

These are hardly independent samples, and if treated them as independent we would radically underestimate the true statistical error in our measurements. That is: our error bars would be meaninglessly small, leaving us with a completely false sense of precision.

Instead, we have to actually (again) *measure* how long the memory of a typical state in the sequence persists. We can quantify this with an autocorrelation function, $C(t)$, associated with some observable. For an observable like the magnetization, $M(t)$, we can define the autocorrelation $C_M(t)$ by first defining the average $\langle M \rangle$, where $\langle \ldots \rangle$ means to average over all configurations at time t_0 within equilibrated set. We then define the instantaneous fluctuation of the observable, $\delta M(t) = M(t) - \langle M \rangle$. Finally, we measure the correlations of these fluctuations, which are then typically normalized so that the function begins at one:

$$C_M(t) == \frac{\langle \delta M(t_0) \delta M(t_0 + t) \rangle}{\langle (\delta M)^2 \rangle}, \tag{12.7}$$

Figure 12.3 shows a few such autocorrelation functions at different temperatures in a two-dimensional Ising model

This function measures how correlated the fluctuations in magnetization are with themselves, separated by a "time" of t MC steps. We have normalized the function so that $C(0) = 1$, but that is just a convention. Such functions typically decays exponentially[101], giving us a characteristic autocorrelation time τ. This is, roughly speaking, the number of MCMC steps we have to take before the system has "forgotten" an earlier state, and thus after which we can imagine that we have gathered another independent sample for our measurement.

The practitioner's analysis pipeline

Putting this all together, we have the following very standard pattern:

1. **Generate:** Run long MCMC simulations, saving the state (or the observables you care about) frequently.

[101] Perhaps you are detecting a theme, here

2. **Equilibrate:** Plot the observable versus the number of MC steps, identify the equilibration time t_{eq}, and ignore all data generated before this point in the following.
3. **Decorrelate:** Calculate the autocorrelation time, τ, for the remaining equilibrated data.
4. **Decimate:** Create a smaller data set of effectively independent samples by taking data points from the equilibrated trajectory only every τ (or every 2τ, or every 3τ, etc).
5. **Evaluate:** Calculate the average and the standard error of the mean only on this final data set.

12.3 Case Study II: Particle-based systems

While spin systems are a canonical use case for Metropolis Monte Carlo, the algorithms is much more general. We can apply the exact same MCMC machinery to the kinds of particle-based systems we studied in Module II, providing a powerful alternative to the molecular dynamics methods we developed there.

The goal, here, is fundamentally different. In MD we integrated the equations of motion to follow the true, physical evolution of the system over time, generating a *trajectory*. In MC, our goal is to correctly sample the equilibrium distribution of *configurations*. The sequence of states in an MCMC simulation is a carefully constructed random walk through phase space, and not a physical path. While we often talk about "Monte Carlo time" – typically the number of Metropolis steps taken – this is merely a measure of computational effort and not a faithful representation of the dynamics of the system over actual time.

Part of the beauty of the modular framework we have built is that we can implement this entirely new simulation paradigm by reusing almost all of our existing code. The logic for particle data structures, periodic boundary conditions, and potential energy calculations from Chapter 10 can and should be directly reused. With that as a foundation, we only need to write a few new methods to define the Monte Carlo "move" in this case, and we'll have a complete, functional simulation.

12.3.1 Implementation details

First, we define our `ParticleSystem` and `Particle` structs exactly as we already have. We'll even make the decision to keep the `velocity` field in the particles; while it will go unused in our pure MC simulation, keeping the data structure the same ensures compatibility[102].

[102] And allows for future extensions, such as "hybrid Monte Carlo" methods [66].

```
struct Particle{D,T}
    position::SVector{D,T}
    velocity::SVector{D,T}
    mass::T
end

struct ParticleSystem{D,T} <: AbstractSystem
    particles::Vector{Particle{D,T}}
    boundary_conditions::AbstractBoundaryConditions
end
```

Let's define the concrete MC move we will use for our particle-based simulations. One choice, analogous to the single spin flips in the Ising model, is to randomly displace a particle by some amount. This structure holds a key parameter – the maximum displacement in any direction that we will propose – which will be important to tune so that the system can explore a reasonable amount of phase space and that move get accepted a reasonable amount of the time.

```
struct SingleParticleMove{T} <: AbstractMCMove
    max_displacement::T
end
struct ParticleMoveInfo{D,T}
    index::Int
    new_position::SVector{D,T}
end
```

The proposal and acceptance functions are straightforward implementations that dispatch on this new move type, with the proposal function returning new structure that, as before, contains the information needed to accept a move if the Metropolis criteria is satisfied.

```
function propose_move(system::ParticleSystem{D,T}, hamiltonian,
                      move::SingleParticleMove) where {D,T}

    idx = rand(1:length(system.particles))
    p_old = system.particles[idx]
    random_vec = rand(SVector{D,T}) .* 2.0 .- 1.0
    displacement = random_vec * move.max_displacement

    return ParticleMoveInfo(idx, p_old.position + displacement)
end

function accept_move!(system::ParticleSystem,
    move_info::ParticleMoveInfo, move::SingleParticleMove)
    p_old = system.particles[move_info.index]

    p_new = Particle(move_info.new_position, p_old.velocity,
p_old.mass)

    system.particles[move_info.index] = p_new
end
```

To handle the physics, we introduce a `ParticleHamiltonian` that holds an instance of a new potential type. This allows us to plug in a `LennardJones` struct, a `HarmonicRepulsion` struct, or any other potential that conforms to our interface. It's one extra layer of indirection, but it allows us to decouple the physical law from the simulation algorithm, exactly as we might want

```
abstract type AbstractHamiltonian end

struct LennardJones{T} <: AbstractHamiltonian
    epsilon::T
    sigma::T
    cutoff_sq::T
end

struct ParticleHamiltonian{P<:AbstractHamiltonian} <: AbstractEnergy
    potential::P
    # Additional structures for, e.g. for cell list calculations
end
```

Finally, we implement the function to calculate the change in energy associated with our proposed move. We adopt the strategy of computing the contribution to the system energy due to the selected particle – not the total energy of the system! – and then recompute what would happen if we temporarily place the particle at the proposed new position. Even for this step we should be sure to reuse all of our (e.g.) cell list technology to make the calculation as efficient as possible, and then we should be *absolutely sure* to restore the system to its original state

before returning the energy difference. This allows us to reuse a set of helper functions while also correctly calculating the δE needed for the Metropolis acceptance criteria.

```
function calculate_ΔE(system::ParticleSystem, info::ParticleMoveInfo,
                      ham::ParticleHamiltonian, move::SingleParticleMove)

    particle_idx = info.index
    # A helper function we need to write
    E_old = _particle_energy(info.index, system, ham)

    p_old = system.particles[info.index]
    p_ghost = Particle(info.new_position, p_old.velocity, p_old.mass)

    system.particles[info.index] = p_ghost
    E_new = _particle_energy(info.index, system, ham)

    system.particles[particle_idx] = p_old
    return E_new - E_old
end
```

12.3.2 Comparing Methods: MC vs. MD

We have now spent time developing two complete simulation engines – MD and MC – that can be applied to study the same particle fluids. When we run both simulations at the same state point (NVT) and calculate structural properties like the radial distribution function, we expect to get results that are statistically identical. This is both a profound validation of both methods, and also a beautiful demonstration of statistical physics: two completely different microscopic processes (deterministic time evolution and a stochastic random walk) have converged to the *same* equilibrium structure.

Both methods work, so – you might ask – which is better? The answer depends on the scientific question you are trying to get at, as they provide fundamentally different kinds of information. MD generates trajectories, and so it is the correct way to study dynamical properties directly: diffusion, viscosity, relaxation times, and so on. It's main disadvantage is that, for system with large energy barriers between different states, it can take an extremely long time for the natural dynamics of the system to explore the entire relevant parts of phase space. In contrast, as we've emphasized, MCMC generates a sequence of configurations, and knows nothing about real time. However, because its "moves" are not constrained to lie along physical paths, you can invent Monte Carlo moves that *much more quickly* sample complex phase spaces. This can make MC methods much more efficient than MD for constructing phase diagrams, evaluating free energies, or finding the equation of state of some system.

In short: they are not really competitors – they are complementary tools. MD tells you how a system gets from state A to B; MC is a powerful shortcut for figuring out the properties of A and B themselves.

Chapter 13

Hamiltonian Monte Carlo and Bayesian inference

In the previous chapters we developed and compared two different simulation paradigms: molecular dynamics, which follows physical trajectories of a system, and Monte Carlo, which performs statistical random walks to explore the steady state distribution of configurations. Notably, in Metropolis Monte Carlo we are free to be wildly creative in how we craft the proposal function $g(\mu \to \mu')$. This freedom will be the key to our next advance.

The standard Metropolis algorithms we introduced in Chapter 12 used simple, local proposals – flipping a single spin, or translating a single particle. These are powerful, but also suffer from a potentially crippling inefficiency: they tend to perform very slow random walks around local neighborhoods of configuration space. Especially in high dimensions, it can take a large amount of time and computational effort for samples in our sequence of states to decorrelate. Perhaps we could start designing smarter proposals, that make take us to distant states that are, at the same time, very likely to be accepted? In this concluding chapter of Module III, we'll explore an interesting marriage of the MD and MC techniques we've been studying. We'll use the resulting methods to solve a canonical scientific task: estimating the parameters of a model that best fit a set of data.

13.1 Hybrid Monte Carlo

In what may seem like a surprising detour, let's first extend the code that performs our Metropolis simulations of particle-based systems by introducing a new move. Rather than have the state μ' arise by picking a particle at random from configuration μ_i and translating it a little bit, Let's imagine starting at μ_i and generating a proposed state μ' by *performing a molecular dynamics simulation of some duration!* Rather than just moving a single random particle by a small amount as we did in Section 12.3, this new state involves all of the particles following some physical dynamics to move to a new position. This will likely decorrelate the sequence of MC states much more quickly than individual particle-based moves. We can also do this in a way that yields a high acceptance probability for our proposals: if we use *symplectic integrators* from Section 9.2 we know that the energy of the system will be approximately conserved, and so the energy differences between states will remain quite small.

This idea of using Hamiltonian dynamics as part of MCMC was originally called "hybrid Monte Carlo" [66, 67], and with the work we have do it is actually remarkably easy to implement. As shown in code block 13.1, the primary task is to extend the `AbstractMCMove` part of our type hierarchy by introducing a `HamiltonianMove` struct. Our strategy will be to store in this structure all of the components – a way of calculating forces, a method of integration, etc – that we need to plug into our existing API for running MD simulations. As has sometimes been the case, we will make a concession to performance in these definitions: since the "move information" would in principle need to contain the entire configuration of the system, and it would be expensive to copy that much data all of the time, we keep a scratch space for data in the `HamiltonianMove` struct itself.

```
struct HamiltonianMove{F <: AbstractForceCalculator,
                       I <: AbstractIntegrator, D, T} <: AbstractMCMove
    force_calculator::F
    integrator::I
    temperature::T
    time_step_size::T
    n_steps::Int
    scratch_system::ParticleSystem{D,T} #to avoid making copies
end

struct HamiltonianMoveInfo{T}
    delta_e::T
end
```

Code block 13.1: A kind of MC move that involves simulating a particle system via molecular dynamics.

The one potentially unexpected wrinkle we have introduced here is at the beginning of the `propose_move` function: before launching our simulation we randomize particle velocities by drawing them from the appropriate Maxwell-Boltzmann distribution. Why? Well, molecular dynamics is in the "Hastings" regime of Metropolis-Hastings, in that outside of extremely unusual pairs of states there is no reason to think that MD will result in a symmetric proposal distribution: $g(\mu \to \mu') \neq g(\mu' \to \mu)$. For such asymmetric proposals the acceptance probabilitiy must be [62]

$$A(\mu \to \mu') = \min\left(1, \frac{p(\mu')g(\mu' \to \mu)}{p(\mu)g(\mu \to \mu')}\right). \tag{13.1}$$

That seems to be a problem: if we maintain the particle velocities, MD is *deterministic* – from configuration μ there is a state μ' that MD *always* proposes, and in general if we have gone from state μ to μ' the odds that continuing the MD trajectory for the same amount of time will take you from μ' back to μ would require a laughably unlikely conspiracy. But if $g(\mu' \to \mu) = 0$, we are left with an acceptance probability of zero; that doesn't do anybody any good.

One elegant solution, outlined in code block 13.2 might be to have the proposed move integrate the equations of motion forward and then flip all of the particle momenta. This

```
function propose_move(system::ParticleSystem{D,T}, hamiltonian,
                      move::HamiltonianMove{F, I, D, T}) where {F, I, D, T}
    t = sqrt(move.temperature)
    for (i, p) in enumerate(system.particles)
        move.scratch_system.particles[i] = Particle(
            p.position, randn(SVector{D,T}) * t/sqrt(p.mass), p.mass
        )
    end

    old_energy = total_energy(move.scratch_system, hamiltonian)
    run_simulation!(move.scratch_system, move.force_calculator,
                    move.integrator, move.time_step_size, move.n_steps)

    new_energy = total_energy(move.scratch_system, hamiltonian)
    return HamiltonianMoveInfo(new_energy-old_energy)
end
```

Code block 13.2: A `propose_move` function for hybrid MC.

would bring us back to a symmetric proposal distribution, but without further tinkering we would end up constructing a Markov chain that only oscillated between two states. A better solution is to recall that in our MCMC approach we are not trying to trace system dynamics, we are trying to sample the phase space. The randomization of momenta at each step is what allows the Markov chain to explore the space effectively, preventing it from getting stuck in deterministic loops. That is, randomizing particle momenta before each MD run is a nice way of simultaneously giving us a symmetric g *and* introducing enough noise to actually explore phase space.

The acceptance step is then a standard Metropolis criteria, but because symplectic integrators *almost* conserve the total Hamiltonian, the change in $\Delta\mathcal{H}$ will be small and the acceptance probability will be high. This is the central trick of HMC.

13.2 Bayesian parameter inference

Forget something as specific as simulating particles for a moment. A much broader task that arises in all of the sciences is *inferring the unknown parameters of some model* from a set of (often noisy) experimental data. The traditional approach to this problem, which I suspect you have already learned, is to cast parameter inference as a point estimation problem: given a model and some data, what is the single best-fit set of parameters that minimizes the difference between the model and the data. Perhaps the most common version of this is to perform a least-squares fit, but more generally one can define a loss function on the difference between the model and the data and try to minimize the loss function with respect to the model's parameters.

Bayesian inference offers an alternate perspective. Rather than trying to find the single

best-fit set of parameters, the goal is to construct the entire *posterior probability distribution*, $p(\vec{\theta}|D)$, which represents the probability of the parameter values $\vec{\theta}$ given the observed data D. This distribution doesn't just give us the most likely parameter values – it gives us the full range of their uncertainties and any correlations that might exist between them.

The foundation for this approach is Bayes' theorem [68, 69]:

$$p(\vec{\theta}|D) = \frac{p(D|\vec{\theta})p(\vec{\theta})}{p(D)}. \tag{13.2}$$

You may not have seen this before, so let's be clear on the notation and on the physical meaning of each of these terms. We've already mentioned the posterior, $p(\vec{\theta}|D)$, and we read those symbols as "the probability of $\vec{\theta}$ given (or conditioned on) D" – this is the quantity we want to compute. The *likelihood* $p(D|\vec{\theta})$ asks, "Given a specific set of parameters $\vec{\theta}$ that go into a model $f(x, \vec{\theta})$, what is the probability of observing our actual data D?"

Answering that question is where our physical model and our assumptions about experimental noise enter. A common starting point is to assume that each of the N data points $y_i \in D$ all are subject to independent Gaussian noise of some standard deviation σ. Under that assumption, the likelihood is given by a product of probabilities:

$$P(D|\vec{\theta}) \propto \prod_{i=1}^{N} \exp\left(-\frac{(y_i - f(x_i, \vec{\theta}))^2}{2\sigma^2}\right) \tag{13.3}$$

The next components of Eq. (13.2) is the *prior*, $p(\vec{\theta})$. The prior represents our knowledge about the parameters *before* we see the data: we might have a "flat" prior if we really have no idea of what the parameters cold be, or an "informed" prior if we have some physical constraints on (e.g., by symmetry) or other knowledge of the values the parameters can take. Finally we have the *evidence*, $p(D) = \int p(D|\vec{\theta})p(\vec{\theta})\,\mathrm{d}\vec{\theta}$, which appears as a normalization constant.

The structure to really burn into your brain is **posterior** $\propto$ **likelihood** $\times$ **prior**. The evidence is a normalization constant that, much like the partition function Z in the previous chapter, is in generally impossible to actually compute. But just like Z, we don't actually need it for sampling! If we use MCMC to explore the space of model parameters, we can make sure that we only ever care about ratios of the posterior, for which the normalization factor cancels out.

While a standard Metropolis random walk could (in principle) explore this parameter space, for even modestly complex models it becomes incredibly inefficient. The simple local moves that work well for the Ising model turn out to be poorly suited to navigating the complex and often strongly correlated "landscapes" of posterior distributions. This inefficiency is not just an inconvenience: it is often the primary barrier to performing a Bayesian analysis at all. This sets the stage for our final application: using a surprising but physically-motivated MCMC algorithm for our Bayesian inference tasks.

13.3 Hamiltonian Monte Carlo

The core idea is going to be to map the Bayesian inference problem onto a Hamiltonian framework [66, 49, 67]. We imagine that the parameters of the model, correspond to the "positions"

of some fictitious particles (typically of unit "mass"), and we furthermore invent some fictitious "momenta," $\{p_i\}$ as addition degrees of freedom. We'll assume these fictitious momenta have a standard kinetic energy term, $K = \sum_i p_i^2/2$. Finally, we want the system to be drawn to areas in which the posterior is large (i.e., likely values of parameters given our data), so we will invent a "potential energy" for our system:

$$U(\vec{\theta}) = -\log p(\vec{\theta}|D). \tag{13.4}$$

This negative log-posterior function, indeed, has "energy minima" at probability maxima.

From here, we can directly plug into our hybrid/Hamiltonian Monte Carlo framework. The one caveat is that Eq. (13.4) is a potential energy rather different from the pairwise potentials we have studied in the context of more normal particle systems. Rather than being a pair-potential, the data almost certainly couple all of "positional" degrees of freedom together (c.f. Eqs. (13.2) and (13.3)). Thus, the calculation of the "force," $F = -\nabla U(\vec{\theta}) = \nabla \log p(\vec{\theta}|D)$, is often a bit thorny.

HMC for inference

In the context of our Bayesian inference problem, the HMC algorithm becomes:

1. Start at parameter state $\vec{\theta}$.
2. Assign random momenta to the parameters, drawing each from a zero mean, unit variance normal distribution.
3. Calculate the current $E_{\text{old}} = \mathcal{H}(\vec{\theta}, \vec{p}) = -\log p(\vec{\theta}|D) + K(\vec{p})$.
4. Simulate Hamiltonian dynamics for M time steps with $\Delta t = \varepsilon$ using a symplectic integrator (e.g., velocity Verlet), to get $\{\vec{\theta}', \vec{p}'\}$.
5. Calculate $E_{\text{new}} = \mathcal{H}(\vec{\theta}', \vec{p}')$.
6. Accept the state θ' with probability $\min\left(1, e^{-\Delta E}\right)$.

Key parameters that require tuning for specific models, priors, and data are the step size and the integration time, Δt and M.

13.3.1 Case study: inferring exponential decay

Let's pause and apply this framework to a common, simple problem we've discussed several times in the last few chapters: determining the parameters of an exponential decay from noisy data. Perhaps we've measured some quantity y at various times t, and we believe that the underlying process is actually exponentially decaying:

$$y(t) = ae^{-bt}. \tag{13.5}$$

Our goal is to infer the initial amplitude a and the inverse timescale b from a set of N data points, (t_i, y_i), assuming that we know the measurement noise (which we will here take to be independent Gaussian noise whose standard deviation is σ).

First, for this model we can transcribe Eq. (13.3) and (up to a constant) write the log-likelihood:

$$\log p(D|a,b) = -\frac{1}{2\sigma^2}\sum_{i=1}^{N}\left(y_i - ae^{-bt}\right)^2$$

We also need to *specify our priors*: what do we know or believe about the parameters a and b? Simple, relatively uniformative choices might be that they are positive numbers that we otherwise know nothing about:

$$p(a) \propto 1 \text{ for } a > 0; \quad p(b) \propto 1 \text{ for } b > 0.$$

Combining these and ignoring the constants, we want to sample

$$\log p(a,b|D) \approx -\frac{1}{2\sigma^2}\sum_{i=1}^{N}(y_i - ae^{-bt_i})^2 \quad (\text{for } a > 0, b > 0) \tag{13.6}$$

This combination of model, likelihood, and prior is one of the few where we can analytically write down the gradient that we need in order to run our "molecular dynamics." We have:

$$-\frac{\partial U}{\partial a} = \frac{1}{\sigma^2}\sum_{i=1}^{N} e^{-bt_i}\left(y_i - ae^{-bt_i}\right) \tag{13.7}$$

$$-\frac{\partial U}{\partial b} = \frac{-1}{\sigma^2}\sum_{i=1}^{N} at_i e^{-bt_i}\left(y_i - ae^{-bt_i}\right) \tag{13.8}$$

$$\tag{13.9}$$

With this in hand, we can directly follow the HMC algorithm above. Starting with some relatively arbitrary initial parameter values, we choose our HMC parameters, implement a `LogPosteriorHamiltonian` for our specific model and then run the `metropolis_step!()` many times.

The output, as with any MC simulation, will be a long list of samples. We need to check for convergence, and measure autocorrelation times to get independent samples. But having done so, we are in a position concretely understand how the space of models intersects with our data. We could create a 2D histogram of the accepted $\{a, b\}$ samples – the densest region should be centered around the true values, and the shape of the distribution reveals both the uncertainties we should have and any correlations between a and b. One would typically find the *marginal posterior distributions* – the histograms of just the a values and of just the b values – and calculate / report the mean value along with other standard statistical measures.

Let's see what this starts to look like *in our code*. The first thing we want to do is add new structs that fit into our type hierarchy for *both* the metropolis acceptance step (for which we need a concrete `AbstractEnergy` which we will dispatch on to compute changes in energy) and for the MD step (for which we need a concrete `AbstractForceCalculator` on which we can dispatch to compute the forces).

```
struct LogPosteriorHamiltonian{M,P} <: AbstractEnergy
    data::Vector{Vector{Float64}}
    data_sigma::Float64
    model::M
    prior::P
end
struct LogPosteriorHamiltonianExpModel{E <: AbstractEnergy}
                                    <: AbstractForceCalculator
       U::E
end
```

These simple versions represent relatively hard-coded versions of these structs – our concrete energy struct has data, a kind of error term for the data, and space for model and prior functions, but the concrete force calculator is quite specific. Code block 13.3 shows an implementation of the `compute_acceleration!` function that dispatches on our exponential model. That code block in particular probably feels a bit different than most of the code we've been writing: rather than a generic function it is hyper-specialized (in this case, for (1) a simple exponential decay model, (2) errors in the data assumed to be independent Gaussians of the same variance, and (3) a flat prior on the model parameters).

```
"""
Specialized: exp model, indep gaussian errors, and a flat prior
"""
function compute_acceleration!(accelerations::Vector{SVector{D,T}},
           system::ParticleSystem{D,T,B},
           calculator::LogPosteriorHamiltonianExpModel) where {D,T,B}
    a = system.particles[1].position[1]
    b = system.particles[2].position[1]
    sigma_squared = calculator.U.data_sigma^2
    acceleration_a = zero(T)
    acceleration_b = zero(T)
    for d in calculator.U.data
        common_factor = exp(-b*d[1]) * (d[2] - a*exp(-b*d[1]))
        acceleration_a += common_factor
        acceleration_b -= a * d[1] * common_factor
    end
    accelerations[1] = SVector{1,T}(acceleration_a / sigma_squared)
    accelerations[2] = SVector{1,T}(acceleration_b / sigma_squared)
    return nothing
end
```

Code block 13.3: A `calculate_acceleration` function for HMC, concretely implementing the mapping from a very specific Bayesian log posterior to the gradient of a energy functional.

It's a little bit inelegant, and yet we can actually tie all of these components together! We can use the same, e.g., `VerletIntegrator` we introduced in Chapter 9 with our new "gradient of

the log posterior for the exponential model" `ForceCalculator` to do the molecular dynamics that represent each metropolis step, and tie it together with exactly the same MCMC code we wrote in Chapter 12. Figure 13.1 shows some of the results of such an analysis, in which the above machinery is applied to "fake data" generated by adding adding uncorrelated Gaussian noise on top of an exponential model.

Noise parameters

This has been a simplified, extremely idealized setting. For instance, it is rare that we have the luxury of actually knowing the true distribution of errors to use for our likelihood function in Eq. (13.3). Much more commonly the noise parameters (like σ) are treated as additional unknown parameters in our model, each with its own prior distribution. This additional complexity fits seemlessly into the HMC framework: we account for these extra parameters when defining the negative log-posterior and calculating its gradient, we we then aim to infer the joint posterior distribution for both the model parameters *and* the noise parameters.

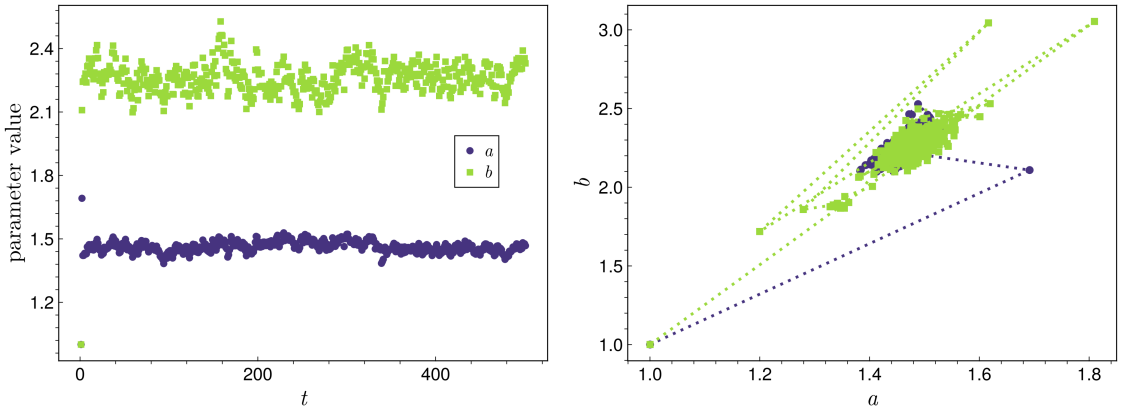

Figure 13.1: **Bayesian inference of model parameters** (Left) traces of the sequence of parameter values after each metropolis step in an HMC algorithm applied to data generated by adding noise on top of an exponentially decaying function. (Right) the trajectory of model parameter points after each metropolis step, with two different values of the HMC tuning parameters (Δt and M).

13.4 Flies in the ointment

Using Hamiltonian Monte Carlo for model parameter inference is a powerful, principled tool in the scientific toolbox. It is, however, not without some sticking points.

13.4.1 The trouble with tuning

First, the trouble with those tuning parameters: how exactly should we choose the integration time and the total run length? These are completely fictitious parameters – we are using artificial

Hamiltonian dynamics to sample space, but its not like the model for our data actually *has* Hamiltonian dynamics – so other than "the Markov chain generates independent samples cheaply" there are precious few objective criteria to help us determine them.

Manually tuning them is challenging. For Δt, too small a value means a very slow, computationally costly exploration of parameters space; too large a value means the integrator error grows and the acceptance rate goes down. For M, too few total steps just performs an inefficient random walk; too many total steps tends to result in trajectories that "U-turn" back on themselves, wasting a lot of our computational effort. What a headache.

A state-of-the-art solution to this problem are techniques like the "No-U-Turn Sampler" (NUTS) [70]. At a high level, the basic idea of it is to adaptively, automatically choose how long to run the Hamiltonian trajectory for *each* proposal. It does this by building a trajectory and stopping when it detects that the path is starting to "turn back" on itself (a tricky thing to quantify in high-dimensional model spaces!). The payoff, though, is that this completely removes the need for manual tuning of HMC parameters, making Bayesian inference accessible and robust for a wide range of problems[103]

13.4.2 The problem with partials

The other problem is related to computing the gradient of the negative log-posterior function we use as the potential energy in our Hamiltonian dynamics. Even in the simplest possible case – an exponential model, completely flat priors, and independent Gaussian errors in our experimental data – computing the relevant partial derivatives with respect to model parameters is at least a little bit tedious to do and error prone to encode. It also feels unsatisfying. One can imagine a vast zoo of potential models you might want to use to fit to data, a similarly large space of possible priors you might have, and a range of ways that the error in the likelihood function might work – do we really have to write down a different concrete implementation of the `compute_acceleration!` function for every single one of these variations? Fortunately, there is an unexpected computational tool that will swoop to the rescue, enabling us to compute the gradients of complex log-posteriors just as readily as we can write down new models. Prepare for Module V!

[103] i.e., you can and should turn to off-the-shelf solutions for your actual research.

Bibliography

[1] Harold Abelson and Gerald Jay Sussman. *Structure and interpretation of computer programs*. The MIT Press, 1996.

[2] Cristopher Moore and Stephan Mertens. *The nature of computation*. Oxford University Press, 2011.

[3] Werner Krauth. *Statistical mechanics: algorithms and computations*, volume 13. OUP Oxford, 2006.

[4] Daan Frenkel. Simulations: The dark side. *The European Physical Journal Plus*, 128:1–21, 2013.

[5] Daan Frenkel and Berend Smit. *Understanding molecular simulation: from algorithms to applications*. Elsevier, 2023.

[6] Alex Gezerlis. *Numerical methods in physics with Python*, volume 1. Cambridge University Press Cambridge, UK, 2023.

[7] Kyle Novak. *Numerical Methods for Scientific Computing: The Definitive Manual for Math Geeks*. Equal Share Press, 2022.

[8] William Jones. *Synopsis Palmariorum Matheseos: Or, a New Introduction to the Mathematics*. J. Matthews for Jeff. Wale at the Angel in St. Paul's Church-Yard, 1706.

[9] William Oughtred. *Clavis Mathematicae denuo limita, sive potius fabricata*. Lichfield, 1631.

[10] Florian Cajori. *A history of mathematical notations*, volume 1. Courier Corporation, 1993.

[11] Tom Kwong. *Hands-On Design Patterns and Best Practices with Julia: Proven solutions to common problems in software design for Julia 1. x*. Packt Publishing Ltd, 2020.

[12] Lee Phillips. *Practical Julia: A Hands-on Introduction for Scientific Minds*. No Starch Press, 2023.

[13] Berni J Alder and Thomas Everett Wainwright. Studies in molecular dynamics. i. general method. *The Journal of Chemical Physics*, 31(2):459–466, 1959.

[14] Gregorii Aleksandrovich Galperin. Playing pool with π (the number π from a billiard point of view). *Regular and chaotic dynamics*, 8(4):375–394, 2003.

[15] F Chiappetta, C Meringolo, P Riccardi, R Tucci, A Bruzzese, and G Prete. Boyle, huygens and the 'anomalous suspension'of water. *Physics Education*, 59(4):045026, 2024.

[16] Scott Chacon and Ben Straub. *Pro git*. Springer Nature, 2014.

[17] Greg Wilson, Dhavide A Aruliah, C Titus Brown, Neil P Chue Hong, Matt Davis, Richard T Guy, Steven HD Haddock, Kathryn D Huff, Ian M Mitchell, Mark D Plumbley, et al. Best practices for scientific computing. *PLoS biology*, 12(1):e1001745, 2014.

[18] Greg Wilson, Jennifer Bryan, Karen Cranston, Justin Kitzes, Lex Nederbragt, and Tracy K Teal. Good enough practices in scientific computing. *PLoS computational biology*, 13(6):e1005510, 2017.

[19] Robert Nystrom. *Game programming patterns*. Genever Benning, 2014.

[20] Volker Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969.

[21] Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. New bounds for matrix multiplication: from alpha to omega. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3792–3835. SIAM, 2024.

[22] Tony Freeth, Yanis Bitsakis, Xenophon Moussas, John H Seiradakis, Agamemnon Tselikas, Helen Mangou, Mary Zafeiropoulou, Roger Hadland, David Bate, Andrew Ramsey, et al. Decoding the ancient greek astronomical calculator known as the antikythera mechanism. *Nature*, 444(7119):587–591, 2006.

[23] Arieh Iserles. *A first course in the numerical analysis of differential equations*. Number 44. Cambridge university press, 2009.

[24] Ernst Hairer, Christian Lubich, and Gerhard Wanner. Structure-preserving algorithms for ordinary differential equations. *Geometric numerical integration*, 31, 2006.

[25] Leonhard Euler. *Institutiones calculi integralis*, volume 1. Impensis Academiae Imperialis Scientiarum, 1768.

[26] Carl Runge. Über die numerische auflösung von differentialgleichungen. *Mathematische Annalen*, 46(2):167–178, 1895.

[27] Wilhelm Kutta. *Beitrag zur näherungsweisen Integration totaler Differentialgleichungen*. Teubner, 1901.

[28] John C Butcher. Implicit runge-kutta processes. *Mathematics of computation*, 18(85):50–64, 1964.

[29] Ch Tsitouras. Runge–kutta pairs of order 5 (4) satisfying only the first column simplifying assumption. *Computers & mathematics with applications*, 62(2):770–775, 2011.

[30] Hermann Weyl. *The classical groups: their invariants and representations*, volume 1. Princeton university press, 1939.

[31] Loup Verlet. Computer" experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Physical review*, 159(1):98, 1967.

[32] Carl Størmer. Sur les trajectoires des corpuscules électriques dans l'espace sous l'action du magnétisme terrestre. *Archives des Sciences Physiques et Naturelles*, 24:5–18, 113–158, 221–247, 317–364, 1907. Published in four parts.

[33] Robert I McLachlan and G Reinout W Quispel. Splitting methods. *Acta Numerica*, 11:341–434, 2002.

[34] Isaac Newton. *Philosophiæ Naturalis Principia Mathematica*. Jussu Societatis Regiæ ac Typis Josephi Streater, Londini, 1687.

[35] Eugene Borisovich Dynkin. Calculation of the coefficients in the campbell-hausdorff formula. In *Dokl. Akad. Nauk. SSSR (NS)*, volume 57, pages 323–326, 1947.

[36] Mark E Tuckerman. *Statistical mechanics: theory and molecular simulation*. Oxford university press, 2023.

[37] William Camden. *Remaines of a greater worke, concerning Britaine, the inhabitants thereof, their languages, names, surnames, empreses, wise speeches, poësies, and epitaphes*. Printed by G. Eld for Simon Waterson, London, 1605.

[38] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

[39] John Edward Jones. On the determination of molecular fields.—ii. from the equation of state of a gas. *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, 106(738):463–477, 1924.

[40] Robert Bridson. Fast poisson disk sampling in arbitrary dimensions. *SIGGRAPH sketches*, 10(1):1, 2007.

[41] Kurt Kremer and Gary S Grest. Dynamics of entangled linear polymer melts: A molecular-dynamics simulation. *The Journal of Chemical Physics*, 92(8):5057–5086, 1990.

[42] Roger W Hockney and James W Eastwood. *Computer simulation using particles*. IOP Publishing Ltd., 1988.

[43] Hans C Andersen. Molecular dynamics simulations at constant pressure and/or temperature. *The Journal of chemical physics*, 72(4):2384–2393, 1980.

[44] Shuichi Nosé. A unified formulation of the constant temperature molecular dynamics methods. *The Journal of chemical physics*, 81(1):511–519, 1984.

[45] William G Hoover. Canonical dynamics: Equilibrium phase-space distributions. *Physical review A*, 31(3):1695, 1985.

[46] Glenn J Martyna, Mark E Tuckerman, Douglas J Tobias, and Michael L Klein. Explicit reversible integrators for extended systems dynamics. *Molecular Physics*, 87(5):1117–1157, 1996.

[47] Owen G Jepps, Gary Ayton, and Denis J Evans. Microscopic expressions for the thermodynamic temperature. *Physical Review E*, 62(4):4757, 2000.

[48] Albert Einstein. Über die von der molekularkinetischen theorie der wärme geforderte bewegung von in ruhenden flüssigkeiten suspendierten teilchen. *Ann. d. Phys.(Leipzig)*, 17:549, 1905.

[49] Steve Brooks, Andrew Gelman, Galin Jones, and Xiao-Li Meng. *Handbook of markov chain monte carlo*. CRC press, 2011.

[50] Maurice G Kendall and B Babington Smith. Randomness and random sampling numbers. *Journal of the royal Statistical Society*, 101(1):147–166, 1938.

[51] WE Thomson. A modified congruence method of generating pseudo-random numbers. *The Computer Journal*, 1(2):83–83, 1958.

[52] George Marsaglia. Random numbers fall mainly in the planes. *Proceedings of the National Academy of sciences*, 61(1):25–28, 1968.

[53] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[54] Melissa E O'neill. Pcg: A family of simple fast space-efficient statistically good algorithms for random number generation. *ACM Transactions on Mathematical Software*, 204:1–46, 2014.

[55] David Blackman and Sebastiano Vigna. Scrambled linear pseudorandom number generators. *ACM Transactions on Mathematical Software (TOMS)*, 47(4):1–32, 2021.

[56] George EP Box and Mervin E Muller. A note on the generation of random normal deviates. *The annals of mathematical statistics*, 29(2):610–611, 1958.

[57] A. A. Markov. Issledovanie zamechatel'nogo sluchaya zavisimyh ispytanij. *Izvestiya Akademii Nauk*, 1(3):61–80, 1907. Translated into French as "Recherches sur un cas remarquable d'epreuves dependantes", Acta Mathematica, 33, (1910), 87-104.

[58] A. A. Markov. An example of statistical investigation of the text eugene onegin concerning the connection of samples in chains. *Science in Context*, 19(4):591–600, 2006.

[59] Oskar Perron. Zur theorie der matrices. *Mathematische Annalen*, 64(2):248–263, 1907.

[60] Georg Frobenius, Ferdinand Georg Frobenius, Ferdinand Georg Frobenius, Ferdinand Georg Frobenius, and Germany Mathematician. Über matrizen aus nicht negativen elementen. 1912.

[61] William Shakespeare. *The Complete Works of William Shakespeare*. Project Gutenberg, 1994. Project Gutenberg EBook 100 Accessed October 8, 2025.

[62] W Keith Hastings. Monte carlo sampling methods using markov chains and their applications. 1970.

[63] Wilhelm Lenz. Beitrag zum verständnis der magnetischen erscheinungen in festen körpern. *Z. Phys.*, 21:613–615, 1920.

[64] Ernst Ising. Beitrag zur theorie des ferromagnetismus. *Zeitschrift für Physik*, 31(1):253–258, 1925.

[65] Sigismund Kobe. Ernst ising—physicist and teacher. *Journal of statistical physics*, 88(3):991–995, 1997.

[66] Simon Duane, Anthony D Kennedy, Brian J Pendleton, and Duncan Roweth. Hybrid monte carlo. *Physics letters B*, 195(2):216–222, 1987.

[67] Michael Betancourt. A conceptual introduction to hamiltonian monte carlo. *arXiv preprint arXiv:1701.02434*, 2017.

[68] Thomas Bayes. An essay towards solving a problem in the doctrine of chances. *Philosophical Transactions of the Royal Society of London*, 53:370–418, 1763. Communicated by Richard Price.

[69] Pierre Simon Laplace. Mémoire sur la probabilité de causes par les évenements. *Mémoire de l'académie royale des sciences*, 1774.

[70] Matthew D Hoffman, Andrew Gelman, et al. The no-u-turn sampler: adaptively setting path lengths in hamiltonian monte carlo. *J. Mach. Learn. Res.*, 15(1):1593–1623, 2014.

[71] Stanley J Farlow. *Partial differential equations for scientists and engineers*. Courier Corporation, 1993.

[72] Sandro Salsa. *Partial differential equations in action*. Springer, 2016.

[73] Zhuoqiang Guo, Denghui Lu, Yujin Yan, Siyu Hu, Rongrong Liu, Guangming Tan, Ninghui Sun, Wanrun Jiang, Lijun Liu, Yixiao Chen, et al. Extending the limit of molecular dynamics with ab initio accuracy to 10 billion atoms. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 205–218, 2022.

[74] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[75] John Wallis. *Opera mathematica*, volume 2. E Theatro Sheldoniano, Oxonii, 1693.

[76] Leonhard Euler. *Introductio in analysin infinitorum*, volume 2. MM Bousquet, 1748.

[77] Eduard Study. *Geometrie der Dynamen. Die Zusammensetzung von Kräften und verwandte Gegenstände der Geometrie*. B. G. Teubner, Leipzig, 1903.

[78] RWHT Hudson. Geometrie der dynamen. die zusammensetzung von kräften, und verwandte gegenstände der geometrie. von e. study.(leipzig, teubner, 1903.) pp. 603. m. 21. *The Mathematical Gazette*, 3(44):15–16, 1904.

[79] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of machine learning research*, 18(153):1–43, 2018.

[80] William Kingdon Clifford. Preliminary sketch of biquaternions. *Proceedings of the London Mathematical Society*, 1(1):381–395, 1873.

[81] Robert Edwin Wengert. A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–464, 1964.

[82] Isaac Newton. *The Method of Fluxions and Infinite Series; With Its Application to the Geometry of Curve-Lines*. Henry Woodfall, London, 1736. Originally written in Latin as 'De Methodis Serierum et Fluxionum' c. 1671.

[83] Jarrett Revels, Miles Lubin, and Theodore Papamarkou. Forward-mode automatic differentiation in julia. *arXiv preprint arXiv:1607.07892*, 2016.

[84] Seppo Linnainmaa. Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics*, 16(2):146–160, 1976.

[85] Bert Speelpenning. *Compiling fast partial derivatives of functions given by algorithms*. University of Illinois at Urbana-Champaign, Urbana-Champaign, 1980.

[86] Alan Mathison Turing et al. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[87] Adrian Hill, Guillaume Dalle, and Alexis Montoison. An illustrated guide to automatic sparse differentiation. In *ICLR Blogposts 2025*, 2025.