Computational Physics (Physics 760) Exercise #4

Dr. Evan Berkowitz `e.berkowitz@fz-juelich.de`
Dr. Stefan Krieg `s.krieg@fz-juelich.de`

Posted 8 Nov 2024
Due 15 Nov 2024 at 18:00

Don't forget to make your code available to and useable by your tutor / grader!

4 [20 points] Markov-Chain Monte Carlo

The Metropolis (or Metropolis-Hastings) algorithm for Markov-Chain Monte Carlo sampling from a distribution P is

1. Start with some legal sample x in the domain of P.
2. Use a randomized update algorithm to *propose* a new configuration y with probability $g(y \leftarrow x)$.
3. Compute the *acceptance probability*
$$A(y \leftarrow x) = \min\left(1, \frac{P(y)g(x \leftarrow y)}{P(x)g(y \leftarrow x)}\right) \tag{1}$$
4. Sample a single number from the uniform distribution on $[0, 1]$, and if A is bigger *accept* the proposal y as the next configuration. Otherwise *reject* the proposal y as the next configuration, repeating x as the next configuration.
5. Begin again with the next configuration as the new x.

This produces a stream on configurations that are distributed according to P but are not independent.

4.1 [20 points] The Ising Model in 1D

Consider the Ising model in 1 dimension on N sites with periodic boundary conditions and constant external magnetic field h,

$$Z[\beta, J, h] = \sum_{\sigma} e^{-\beta(H[J]-hM)} \qquad H[J] = -J \sum_{\langle x,y\rangle} \sigma_x \sigma_y \qquad M = \sum_x \sigma_x \qquad \beta = 1/T \tag{2}$$

with dimensionful parameters (in units where $k_B = 1$). The expectation value of an operator O is given by

$$\langle O \rangle = \frac{1}{Z[\beta, J, h]} \sum_{\sigma} e^{-\beta(H[J]-hM)} O[\sigma] \tag{3}$$

As mentioned in lecture, everything is known about the 1D case. In particular,

$$Z[\beta, J, h] = \lambda_+^N + \lambda_-^N \qquad \lambda_\pm = e^{J/T}\left(\cosh(h/T) \pm \sqrt{\sinh^2(h/T) + e^{-4J/T}}\right). \tag{4}$$

In this exercise we will compute magnetizations stochastically and compare them to the exactly-known results, as a way to make sure we know what we're doing.

- (1pt) As written there are three dimensionful parameters, J, h, and $\beta = 1/T$. Find a set of TWO dimensionless parameters and rewrite the exact result (4) in terms of those variables.
- (1pt) The expected net magnetization M (which is already dimensionless) is given by

$$\langle M \rangle = T \frac{\partial}{\partial h} \log Z[\beta, J, h] \tag{5}$$

  Derive an expression for $\langle M \rangle$ in terms of your 2 chosen dimensionless variables by actually differentiating Z (4). You don't have to get it to as simple a form as possible, but make sure it's simple enough to easily program as a function of your 2 dimensionless variables.

The *magnetization per spin* $m = M/N$ is intensive and (hopefully) goes to an h-dependent constant as $N \to \infty$ (ie. in the *thermodynamic limit*). A good check on your expression is that it should have the limit

$$\lim_{N\to\infty} \langle m \rangle = \lim_{N\to\infty} \left\langle \frac{M}{N} \right\rangle = \frac{\sinh h/T}{\sqrt{\sinh^2 h/T + e^{-4J/T}}}. \tag{6}$$

You don't have to demonstrate this limit formally. However, check it numerically:

- (2pt) On a single figure plot $\langle m = M/N \rangle$ (using the expression you just found) for $J = 0.75$, $\beta = 1$ for $h \in [-1, +1]$ for $N \in \{1, 2, 4, 8, 16, \infty\}$ using what you found for $\langle M \rangle$ in the previous part and the exact m (6).

Call

$$S = \beta(H - hM) \tag{7}$$

the *action*.[1]

- (2pts) Implement a function which, given the dimensionless parameters you picked and a configuration σ computes the action S. You can put the parameters in as arguments, or make the function a class member and pass the parameters into the class constructor, or whatever; the organization of the code is up to you.

[1] It isn't the action, but it plays the role the action plays in quantum field theories.

- (2pts) Implement a function which, given the dimensionless parameters, a configuration σ, and a location x, computes

$$\Delta S(\sigma, x) = S(\sigma \text{ except flip the spin on site } x) - S(\sigma). \tag{8}$$

But *don't* implement this by just calling S twice and subtracting; there is a much much faster way. The computational cost of evaluating S scales with the number of lattice sites; you should find an implementation of ΔS that doesn't scale that way. (Hint: there are a LOT of cancellations!)

- (2pts) Run your implementations for S and ΔS on larger and larger lattices, measuring how much time an evaluation takes. (You might need to evaluate multiple times to get a reliable timing.) Create a persuasive figure that shows the linear scaling of the cost of S with the lattice size but the essentially-constant cost of evaluating ΔS. Note that you might have to go to pretty big lattices before the asymptotic scaling is apparent.

Let a *single-site update* be a proposal $\sigma' \leftarrow \sigma$ where σ' matches σ everywhere except on a single randomly-chosen site x, where the spin is flipped. Then $g(\sigma' \leftarrow \sigma) = g(\sigma \leftarrow \sigma')$ (since both are $1/N$, the odds the differing site is selected) and the Metropolis-Hastings acceptance probability is

$$A = \min\left(1, \frac{P(\sigma')}{P(\sigma)} \times \frac{g(\sigma \leftarrow \sigma')}{g(\sigma' \leftarrow \sigma)}\right) = \min\left(1, \frac{e^{-S(\sigma')}}{e^{-S(\sigma)}} \times 1\right) = \min\left(1, e^{-(S(\sigma')-S(\sigma))}\right) = \min\left(1, e^{-\Delta S(\sigma,x)}\right) \tag{9}$$

A single *sweep* of this update across the lattice visits all N sites in a random order and proposes flipping the spin there, accepting or rejecting (and updating the configuration) as it goes.

- (10 pts) For $N = 20$, $J = 0.75$, $\beta = 1$, compute $\langle m \rangle$ by Markov-Chain Monte Carlo for about 20 evenly-spaced values of h in $[-1, +1]$, starting from the configuration that's all +1s. Generate 10000 samples (the full outcome space has size $2^{N=20} \simeq 10^6$), each separated by a sweep of the single-site update. Repeat each sampling a few times so you can confidently quote an error bar on $\langle m \rangle$.

(While you are getting this last part to work you will probably find it beneficial to work on smaller lattices and to generate fewer samples. But the 'final' figure is what we're after!)

4.2 [0 points] The Bootstrap

In the future you will do larger, more costly MCMC computations and just repeating the sampling a few times will be prohibitive.

As a reminder, in that case it is common to *bootstrap*. The idea is that you generate a large number N of configurations. If you reach into a bag full of the whole probability space and draw out some configurations, that experience will look identical to reaching into a bag full of the N configurations instead.

So the method (described on 2024-10-30, with a demo implementation) is

1. Work hard to get a set of iid samples $X = \{x_1, x_2, x_3, \cdots x_n\}$
2. Draw the *bootstrap sample* of size a. We draw a times from X with replacement; each member of X has the same likelihood.
3. Measure whatever expectation values you like on the members of bootstrap sample.
4. Repeat steps 2 and 3 B times, making different random draws each time. This gives B different sets of expectation values.
5. Compute estimates by averaging across the bootstrap samples' expectation values, uncertainties from standard deviations of those expectation values.

Suppose you have many observables you're interested in. It's good to measure them on the same bootstraps because (aside from the lower computational cost), the resulting expectation values fluctuate together. For example, suppose in the Ising model you wanted to measure the susceptibility

$$\chi_M = \langle M^2 \rangle - \langle M \rangle^2 \tag{10}$$

Then we can estimate χ_M on each bootstrap ensemble: on each ensemble we can measure $\langle M \rangle$ and $\langle M^2 \rangle$. Just to re-emphasize, the averages on a bootstrap (re)sample are *already expectation values*! But the fluctuations are correlated with one another: if $\langle M \rangle$ is particularly large on a bootstrap draw, $\langle M^2 \rangle$ probably is too, so the estimate of χ_M on that ensemble isn't as large as you might think (or as you might have estimated if you had computed $\langle M \rangle$ and $\langle M^2 \rangle$ independently).

For things we measure bootstrap-sample-by-bootstrap-sample we say that they are measured *under the bootstrap*. If there is any justice (the central limit theorem holds) then quantities measured under the bootstrap should be normally distributed.

You may want to remind yourself how to make this work, you will be asked to estimate error bars in this way on assignment 5.

4.3 [0 points] Fourier Acceleration of the Autocorrelation Function

Critical to the bootstrap resampling procedure is that the observables be *independent*. When we did iid sampling the configurations were independent so the observables inherited that independence.

But when we do MCMC, each configuration is some modification of the configuration before it. They're not independent! That's the price we pay for importance sampling. But we can use bootstrap estimation anyway, as long as we take configurations that are far enough apart in the Markov chain. How far apart is far enough?

The *autocorrelation function* of a timeseries of length n for a real-valued operator O is

$$C(\Delta t) = \langle (O(t)-\mu)(O(t+\Delta t)-\mu)\rangle = \frac{1}{n-|\Delta t|}\sum_{t_0=0}^{n-|\Delta t|-1}(O(t_0)-\mu)(O(t_0+\Delta t)-\mu) \tag{11}$$

tells us how related each figuration was what came before (and, because it is symmetric in Δt, what comes after). We can *either* use the sample average to estimate μ or use theoretical knowledge (if we have any from symmetry or whatever other considerations). The *normalized autocorrelation function*

$$\Gamma(\Delta t) = \frac{C(\Delta t)}{C(0)} \tag{12}$$

starts at 1 and, if there is any justice, decays exponentially. We can compute the *integrated autocorrelation time* up to some time T:

$$\tau(T) = \frac{1}{2} + \sum_{\Delta t=1}^{T}\Gamma(\Delta t) \tag{13}$$

A common choice is to pick T to be the time where Γ first goes negative, but there are others.

The *discrete Fourier transform* F of a timeseries f of length N is[2]

$$F = \mathrm{DFT}[f] \qquad\qquad F_k = \frac{1}{\sqrt{N}}\sum_{t=0}^{N-1} f_t e^{-2\pi i k t/N} \tag{14}$$

$$f = \mathrm{DFT}^{-1}[F] \qquad\qquad f_t = \frac{1}{\sqrt{N}}\sum_{k=0}^{N-1} F_k e^{+2\pi i k t/N} \tag{15}$$

and k takes integer values $k \in [0, 1, \cdots, N-1]$. DFTs are often computed with a very clever, fast algorithm called the *fast Fourier transform* or FFT.

If we imagine that the timeseries repeats in time so that the first sample occurs after the last sample (which, to be clear, it very much does not!), then we can dramatically accelerate the computation of C, from the residuals r

$$r_t = O_t - \mu \qquad\qquad C(\Delta t) = DFT^{-1}\left[DFT[r]\times DFT^{-1}[r]\right](\Delta t) \tag{16}$$

Since the cost of fast fourier transforms are $\sim N\log N$, this approach is much cheapter than the $\sim N^2$ cost of the obvious direct implementation.

- Prove this expression for C. You will probably need the fact that $\sum_{t=0}^{N-1} e^{+2\pi i k t/N} = N\delta_{k,0}$.
- Implement it! You can use ffts provided by numerical libraries, but do NOT use any library's built-in convolution. You may need to be careful about the normalization. In the future it may be helpful for you to be able to pass the mean μ as an argument, in case you know it from symmetry (or by some other argument). My numpy implementation of Γ is below.

[2] Different conventions for the normalizations are used; be careful! I like the unitary convention but tastes differ.

- Sample 20 iid normally distributed random numbers, compute $\Gamma(\Delta t)$ and show that it is symmetric around $\Delta t = 0$ (you may have to count negative Δt from the back of the array, depending on your library's fft conventions).
- Implement the integrated autocorrelation time τ, cutting off the sum at the first zero-crossing.

A python implementation of the Fourier-accelerated autocorrelation function is

```
import numpy as np
def autocorrelation(timeseries, mean=None):
    if mean is None:
        mean = timeseries.mean()
    fluctuations = timeseries-mean

    C = np.fft.ifft(np.fft.fft(fluctuations) * np.fft.ifft(fluctuations)).real
    return C/C[0]
```

If you are working in python you can use this implementation in the remainder of the course.