PHYS358: Midterm 1 (due Oct 02, 9:30am)

Metropolis-Hastings and Path Integrals: The Harmonic Oscillator

Note: Please read the complete set of instructions before you start.

Goal: The goal of the following task is twofold: (1) To introduce an extremely powerful sampling method (the Metropolis-Hastings algorithm), and (2) to use that algorithm to calculate path integrals. Therefore, this Midterm consists of two parts. We will first revisit the sampling problem, and then we will discuss path integrals. Please also consult the reading materials included with this assignment – they discuss pretty much all the technical details.

Part I: Metropolis-Hastings algorithm

It might be useful to remind yourself of the sampling problem discussed in Homework 03. There, you developed a sampling technique based on geometry arguments, namely *rejection sampling*. The main short-coming of rejection sampling is that it can get extremely inefficient if your *target distribution* $P(x)$ has much smaller support than your *proposal distribution* $Q(x)$. Remember the example of sampling from a very narrow Gaussian, given a uniform proposal distribution. Even if the proposal distribution is very close to the target (as is the case for the Lorentzian and the normal distribution), the situation deteriorates quickly when sampling over a N-dimensional space, since the difference between the two distributions ΔV increases with the power of the dimensions, ΔV^N.

This is where the Metropolis-Hastings (MH) algorithm comes in. The main strength of the MH algorithm lies in the fact that it is nearly completely agnostic about the target distribution, i.e. we do not have to find a proposal distribution that is as close as possible to the target, to make the algorithm efficient. Both the Chalmers report and the MacKay paper give succinct summaries, down to algorithmic form, of the method. The main idea is the following:

Imagine you'd like to draw samples $\{x_r\}_{r=1}^{R}$ from a distribution $P(x)$. Instead of approximating $P(x_r)$ by $Q(x_r)$ over the whole range of desired x_r (i.e. instead of finding a $Q(x)$ that globally approximates $P(x)$), the MH algorithm uses a **local** $Q(x)$. Since you're generating a series of $\{x_r\}$, clearly, this is an iterative process. So, let's say that at iteration $t \in 1 \dots T$, you found an acceptable $x^{(t)}$. We will call this a **state**. The question is: how do we pick the next x so that it is distributed following $P(x)$? Answer: Pick a **trial state** x' distributed according to $Q(x'; x^{(t)})$. For example, if $Q(x)$ were the normal distribution $N(0, 1)$ – a popular choice –, then $Q(x'; x^{(t)}) = N(x^{(t)}, \delta)$, i.e. it would be a Gaussian centered on the current state $x^{(t)}$ with a width of δ. We will discuss that width δ in a moment.

What's left is to decide whether x' should be accepted, i.e. whether it will be a good representation of $P(x)$. Thus, we calculate

$$a = \frac{P(x')}{P(x^{(t)})}. \tag{1}$$

Clearly, if $a \geq 1$, x' is a location where $P(x')$ is larger than $P(x^{(t)})$ – thus, we should accept the trial state x' and make it $x^{(t+1)} = x'$.

If that were all, then we'd just get a sequence of x moving towards a maximum of $P(x)$. And that's the problem. Imagine, $P(x)$ was a double Gaussian, with peaks at $x = \pm 3$ and a width of

$\sigma = 0.2$ (i.e. much narrower that the separation of the peaks). If we just followed the prescription in eq. 1, and we started with $x^{(0)} = -3.5$, it would be very likely that we just sampled the left peak, i.e. our algorithm would fail to sample the whole distribution.

Therefore, we need a second condition for acceptance (and this is the whole point about the MH-algorithm). Imagine, $a < 1$, i.e. the trial state x' seems to be a worse guess than the current state $x^{(t)}$. To use the example of the double Gaussian, let's say $x' = -2.5$, and $x^{(t)} = -2.8$. If you accepted x', you'd be walking away from the left peak of $P(x)$ (not good), **but you'd be walking toward the right peak** of $P(x)$, i.e. **you give the algorithm the chance to explore a range of x that may lead to other peaks in** $P(x)$. Therein lies the main power of the MH algorithm. In terms of a prescription, if $a < 1$, we accept x' if $U(0,1) \leq a$ (meaning $x^{(t+1)} = x'$), and we'll discard it otherwise (meaning we assign the current state to the new state $x^{(t+1)} = x^{(t)}$). This last statement is crucial: it means that the states $x^{(t)}$ are correlated. It is only by the law of large numbers (i.e. large T) that the resulting sequence of states converges to $\{x_r\}$ drawn from $P(x)$.

MT1(a) Implementation of a MH sampler [20 pts]: Given the discussion above, implement the MH algorithm in `mci_methast.py`. As usual, you are given the infrastructure, so the only tasks that remain are to **(i) implement the MH algorithm** in the function `methast`, and to **(ii) implement the Box-Müller method (see reading materials)** in the function `rnorm`.

Solution: See `mci_methast.py`. Key point is to implement the acceptance prescription correctly. The Box-Müller method can be implemented using pairs of random numbers.

MT1(b) The pitfalls of MH sampling [15 pts]: `mci_methast.py` takes three arguments: the probability density function $P(x)$ to be sampled, the number of tries R, and the step size δ. This last parameter needs some more scrutiny. In your implementation, it will be the width of $Q(x', x)$, i.e. the width of the normal distribution from which you are chosing the trial step x'. Clearly, there must be some constraints on δ. The following steps will guide you through how to identify these constraints:

1. Run `mci_methast.py exp 10000 0.1`, i.e $R = 10^4$, $\delta = 0.1$ for the exponential distribution. Compare the resulting histogram to the expected distribution (red line).

2. Rerun and increase $\delta = 0.5, 1.0, 2.0, 3.0, 4.0$. Write down what you notice happening to the histogram while you increase δ. Also, write down how the right-hand plot changes. It shows a typical "quality measure" for the MH algorithm, namely how well the Markov chain $x^{(t)}$ samples the domain from which the random numbers **should** be drawn (i.e., the plot shows the step number t against $x^{(t)}$).

3. Determine the "best" δ for the exponential distribution.

4. Repeat steps (1) through (3) for the other three functions (`normal, crazy, lognormal`), and give a reason for your choice of the most suitable sampling step δ for each function. Specifically, describe what determines a suitable δ (*Hint: Look at* $P(x)$).

5. Describe how the choices of δ and R are connected. Explore $R = 10^3, 10^4, 10^5$.

6. Finally, which of the four functions has the least constraints on δ, once $\delta > 1$? How does this affect attempts to determine e.g. the mean value of that distribution?

Solution: For small step sizes, the domain is not well sampled - essentially the MH algorithm proceeds via a random walk, i.e. it takes long ($\sqrt{N}$) to converge, because the sampling of the domain proceeds in a series of coherent steps. For large step sizes, the whole domain is sampled, but small-scale variations in $P(x)$ are not traced out - the steps in x are too large to "see" local structure. Suitable numbers are $\delta_{\texttt{exp}} = 0.5$, $\delta_{\texttt{normal}} = 1.0$, $\delta_{\texttt{crazy}} = 1.0$, $\delta_{\texttt{lognormal}} = 1.0$. δ should be around the "typical" scale of the function, i.e. the width of a peak. For larger R, the effects of small δ can be alleviated to some extent, but at the cost of large run times. The lognormal function has the least constraints – larger δ just sample the slowly decaying tail more and more. Thus, large outliers can occur occasionally, affecting the convergence of the mean.

Part II: Path Integrals

We will now use a special variant of the Metropolis-Hastings algorithm to calculate path integrals, i.e. to determine the most likely path in space-time a single particle in a potential $V(x)$ is expected to take. The details of the path integral mechanism is beyond what we can discuss here – see the Chalmers report for an introduction. You could think about it in the following way:

Assume you want to calculate the path of a (quantum-mechanical) particle traveling from $(t_a, \mathbf{x}_a)$ to $(t_b, \mathbf{x}_b)$, where $(t, \mathbf{x})$ describes the position in space-time. Now, the particle could take an infinite number of possible paths (see Fig. 2 in Chalmers report), and thus, the **probability amplitude** to travel from a to b is

$$K(b,a) = \sum_{allpaths} \phi[x(t)]. \tag{2}$$

Clearly, without any additional information about the probability amplitude for one single path, $\phi[x(t)]$, it's not even clear that $K(b,a)$ converges. Enters the principle of least action, stating in classical mechanics that the particle will take a path ϕ along which the action

$$S[x(t)] = \int_{t_a}^{t_b} L(\dot{x}, x, t)dt \tag{3}$$

is a minimum, with the Lagrangian $L = T - V$. The quantum-mechanical analogue is that while the actions along each path $x(t)$ are the same, they have different phases, i.e.

$$\phi[x(t)] = C\, e^{i\frac{S[x(t)]}{\hbar}}, \tag{4}$$

and thus, the total probability amplitude

$$K(b,a) = C \sum_{allpaths} e^{i\frac{S[x(t)]}{\hbar}}. \tag{5}$$

The **probability** then is $P(b,a) = |K(b,a)|^2$. Eq. 5 suggests why this works: if different paths collect different phases, it stands to reason that longer paths, i.e. larger "detours", collect larger phases, and thus, when being averaged over, are more likely to cancel out. Only the paths that lead to a coherent addition of the phases (i.e. paths that under a small displacement collect only a small phase difference) will contribute to the sum, and thus determine the resulting path. The constant C ensures that the probabilities are appropriately normalized.

To construct the actual path integral, imagine we discretize one single path $x(t)$ in the time t: we split the time axis into N bins of size $\epsilon \equiv (t_b - t_a)/N$ (we assume periodic boundaries, meaning that $x_a = x_b$). For each time $t_i = t_a + i\epsilon$ there is an associated $x(t_i)$. We'd still like to calculate the sum over all paths. So, analogously to our Monte Carlo formulation of the integral

$$\Phi = \int_a^b f(x)dx \approx \frac{1}{R}\sum_{r=1}^{R} f(x_r) \;\text{ with }\; x_r \in U(a,b), \tag{6}$$

we now integrate over the whole x-space **for each** x_i. You could picture this as constructing an infinite number of paths by shifting each x randomly around, for each new path. This is actually pretty much what we'll do in a moment.

There's obviously much more to it – details can be found in the Chalmers report. One of the main issues computationally is that we'd like to get rid of the imaginary argument in the

exponential. This is done by writing the time coordinate in Euclidean time, or $it \to \tau$. Again, the details are beyond our scope here (see section 2.7 of Chalmers report), but the result is that the action formulated in terms of the Lagrangian turns into an action using the Hamiltonian,

$$\exp\left(i\frac{L\delta t}{\hbar}\right) \to \exp\left(-\frac{H\delta\tau}{\hbar}\right), \tag{7}$$

i.e. the cancellation of phases turns into an exponential decay. Paths with large actions will be suppressed exponentially.

The remaining issue to address is how to explore "all possible paths". Here, the MH algorithm enters. We are interested in a path modification that reduces the action along a given path. Therefore, we randomly pick **one single** $x(t_i)$ along the path, and perturb it to a "trial state" $x'(t_i)$,

$$x_i' = x_i + \delta(2u-1) \text{ with } u \in U(0,1). \tag{8}$$

How do we decide whether this is a "good" modification? Obviously, the action along the path should be reduced. This is equivalent to the statement that the "total energy" summed along the path $x(t)$ should be reduced. The total energy is given by the Hamiltonian. We write the energy as

$$E(x_i, x_{i+1}) = \frac{1}{2}m\left(\frac{x_{i+1}-x_i}{\epsilon}\right)^2 + V(x_i), \tag{9}$$

where the first term on the RHS is the kinetic energy. Note that the required "velocity"[1] is calculated by taking the 1st order derivative $\dot{x_i} \equiv (x_{i+1}-x_i)/\epsilon$. In principle, to determine whether the new path has a lower action, we can then determine the total energy along the path by summing eq. 9 over all i. This is unnecessary, though, since all other x_i are kept constant. Therefore, the **energy difference** can be calculated as

$$\Delta E[x] = E(x_{i-1}, x_i') + E(x_i', x_{i+t}) - E(x_{i-1}, x_i) - E(x_i, x_{i+1}). \tag{10}$$

With this in hand, we proceed in the usual MH manner: if $\Delta E < 0$, we improved the path, so we accept x_i' as the new x_i. If $\Delta E > 0$, we accept x_i' with probability $\exp(-\epsilon\Delta E/\hbar)$, i.e. choose $u \in U(0,1)$ and accept x_i' if $u \leq \exp(-\epsilon\Delta E/\hbar)$. *Remember rejection sampling here: The effect of this acceptance probability is that the resulting paths are distributed as* $\exp(-\epsilon\Delta E/\hbar)$, i.e. the probability for paths decays with the exponential of their action.

MT1(c): Implementation of Path Integral Monte Carlo [20 pts]: Implement the integrator in the function `integrate_path` which you can find in `pathintegral.py`. Everything else is already being provided (initializations, tests, setting of potentials etc). Check out the rest of the code, to get an overview of what's happening there. I recommend running `pathintegral.py -h` first, to see what arguments it takes. Section 4.3 of the Chalmers report gives a succinct summary of the algorithm – I recommend to follow that outline. Note that the calculation of the probability as well as the expectation value of the operator (total energy) is already implemented in the code.

Solution: See `pathintegral.py`. Key point is to get the acceptance probabilities right, and to think about how to deal with the periodic boundaries.

[1] All paths take the same time to go from x_a to x_b, since t_a and t_b are being kept fixed. Therefore, if we shift the x_i around in an attempt to find a "better" path, the "velocities" between i and $i+1$ must be changing (ϵ is constant). This somewhat strange interpretation is a result of the translation from real time to Euclidean time. In real time (see eq. 7), the change of x_i would introduce a phase shift.

MT1(d): Testing the Path Integral Monte Carlo implementation [15 pts]: The following steps will guide you through testing your implementation:

1. Run `pathintegral.py harmonic 100 10000 0.1`, i.e. $N = 100$ support points along a path, $T = 10,000$ path realizations, and $\delta = 0.1$. Describe the three plots that should appear. What is each plot showing?

2. Use the potential plot, the provided accuracy measure, and your experience gathered in MT1(a,b) to identify a suitable δ.

3. For the δ you chose in (1), increase the length of the Markov chain to $T = 100,000$ and discuss the result.

4. Let's switch to something more fun: the instanton potential (feel free to look this up): `pathintegral.py instanton 100 100000 1.0`. Explain why the algorithm can reproduce "quantum tunneling".

5. Why are the two peaks of different heights? What needs to be changed to make them the same height?

6. Finally, run `pathintegral.py box 100 100000 0.1`. What topic that we have discussed in class informs you about the correct solution? (*Hint: Remember that the probability* $|\psi|^2$ *is the square of the wave function.*).

Solution: The three plots are: Histogram of the path positions $X(i,t)$, the corresponding probability $|\psi(x)|^2$, the analytical solution (black line), and in the third plot, the potential $V(x)$. For the harmonic oscillator $\delta = 1$ is ok. Increasing the length of the Markov chain leads to more path modifications, and thus to a better (denser) sampling. For the instanton potential, the fact that the algorithm occasionally explores paths with higher energies allows the distribution to cover two regions separated by a maximum in the potential – hence the two peaks. The box potential should result in a probability $|\psi^2| \propto \sin^2(x)$, as expected for a standing wave. Standing waves we've discussed in the context of ODE boundary/eigen value problems.