Markov Chain Monte Carlo: Metropolis-Hastings & Hamiltonian Monte Carlo

Siddharth Bhat (20161105) siddu.druid@gmail.com

October 23, 2020

1 Introduction

All the code for this project (including code to generate the report and its graphs) can be found at https://github.com/bollu/sampleraytracer.

1.1 MCMC sampling: the fundamental algorithm

markov-chain-monte-carlo methods are a class of algorithms which blend together two or three properties; these properties blend together in subtle ways, and permit us to solve very different-looking-problems using the same toolkit, by leveraging different portions of the algorithm's power. Broadly, they all solve the following problem:

> Given an **arbitrary** function $\mathfrak{P} : X \to \mathbb{R}$, **create a sampler** for the **normalized** probability distributed from $\mathfrak{P}$, $\hat{\mathfrak{P}}(x) \equiv \frac{\mathfrak{P}(x)}{\int \mathfrak{P}(x)dx}$. Furthermore, allow the sampling process to be **guided by** a proposal distribution: $\texttt{Prop} : X \to (X \to [0, 1])$. This proposal, for each location $x_0 \in X$, permits the specification of a '**likely direction** of higher likelihood', $\texttt{Prop}(x_0) : X \to \mathbb{R}$.

1.2 Sampling use case 1. Simulation And Sampling

Often, we do really want samples from a particular distribution. For example, we might often want to apply Bayes' rule and sample from the posterior distribution. Assume that we have a computational procedure to compute $P(X = x_0|Y = y_0)$. We now want to $P(Y|X = x_0)$.

$$P(Y|X = x_0) = \frac{P(X = x_0|Y)P(Y)}{\sum_y P(X = x_0|Y = y)}$$

There are two problems in sampling the above distribution:

1 The expression $P(Y = y|X = x)$ has no a-priori structure. Hence, we will need an algorithm that can sample from an arbitrary distribution.

2 The denominator term, which is a normalization factor can be extremely expensive to compute due to the summation involved

Hence, we need to use MCMC to sample from $P(Y|X = x_0)$, and we can expedite this by sampling from $\mathfrak{P}(Y|X = x_0) = P(X = x_0|Y)P(Y) \propto P(Y|X = x_0)$

1.3 Sampling use case 2. Gradient Free Optimisation

We want to maximize a function $f : X \to \mathbb{R}$. However, we lack gradients for f, hence we cannot use techniques such as gradient descent, or other techniques from convex optimisation. In such a case, we can consider f as some sort of unnormalized probability distribution, and use MCMC to sample from f.

That is, we set $\mathfrak{P} \equiv f$. Now, samples from $\mathfrak{P}$ will be 'more likely' to come from those regions where f is large, since a sampler will be more likely to sample from regions of high probability.

1.4 Sampling use case 3. Numerical Integration

We wish to calculate $\int_l^u f(x)dx$ where f is some complicated function with no closed form for the definite integral of f. In this case, we know that the average value of a function, denoted by $\mathbb{E}[f]$, can be computed as:

$$\mathbb{E}[f] \equiv \frac{1}{|u-l|} \int_l^u f(x)dx$$

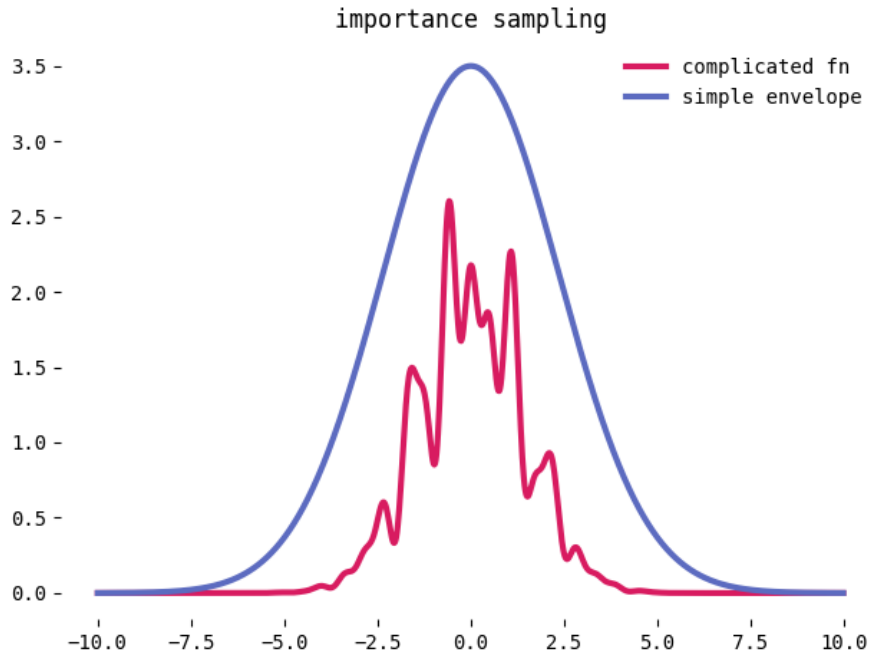
On rearrangement, we get:

$$\int_l^u f(x)dx = |u-l|\mathbb{E}[f]$$

Hence, if we have a good approximation of $\mathbb{E}[f]$, we have a good approximation for the integration. The naive approximation of $\mathbb{E}[f]$ can be found by using:

$$\mathbb{E}[f] \simeq \frac{1}{n} \sum_{i=1}^{n} f(x_i)$$

However, we often know *something* about f, just not its *precise* shape:



In this case, we can *guide* the sampling process, by setting the proposal `Prop` as the envelope of the function f. That way, we will be more likely to pick up *useful samples*, which rapidly sample from the region of space that has mass for the integration.

2 Where it all begins: The Metropolis Hastings sampler

2.1 The big idea

We wish to sample from a distribution $\mathfrak{P}$, but we do not know how to do so. The idea is that we build a markov chain $M[\mathfrak{P}, \texttt{Prop}]$ where $\mathfrak{P}, \texttt{Prop}$ are supplied by the user. We will show that the *stationary distribution* of $M[\mathfrak{P}, \texttt{Prop}]$ is going to be $\mathfrak{P}$. This will ensure that if we interpret *states of* M as samples, these samples will the distriubted according to $\mathfrak{P}$.

2.2 detailed balance: A tool for proofs

We will first require a condition that will enable to rapidly establish that some distribution $\mathfrak{P} : X \to \mathbb{R}$ is the stationary distribution of a markov chain M. If the transition kernel $K : X \times X \to \mathbb{R}$ of $M \equiv (X, K)$ is such that:

$$\forall x, x' \in X, \mathfrak{P}(x)K(x, x') = \mathfrak{P}(x')K(x', x)$$

Then $\mathfrak{P}$ is said to be **detailed balanced** with respect to M.

Theorem 1 *If $\mathfrak{P}$ is detailed balanced to $M \equiv (X, K : X \to (X \to [0, 1])$, then $\mathfrak{P}$ is the stationary distribution of M.*

Proof 1.1 *Let $\mathfrak{P}$ be detailed balanced to M. This means that:*

$$\forall x, x' \in X, \mathfrak{P}(x)K(x)(x') = \mathfrak{P}(x')K(x')(x)$$

Let us say say that we are in state $\mathfrak{P}$. We wish the find the probability distribution after one step of transition. The probability of being in some state x'_0 is going to be:

$$Pnext(x'_0) \equiv \sum_x \mathfrak{P}(x)K(x)(x'_0)$$

since we have $\mathfrak{P}(x)$ probability to be at a given x, and $K(x, x'_0)$ probability to go from x to x'_0. If we add over all possible $x \in X$, we get the probability of all states to enter in x'_0. Manipulating Pnext, we get:

$$
\begin{aligned}
&Pnext(x_0') \equiv \sum_x \mathfrak{P}(x) K(x)(x_0') \\
&= \sum_x \mathfrak{P}(x_0') K(x_0')(x) \quad \textit{(by detailed balance)} \\
&= \mathfrak{P}(x_0') \sum_x K(x_0')(x) \quad \textit{($P(x_0)$ is constant} \\
&= \mathfrak{P}(x_0') \cdot 1 \quad \textit{($K(x_0')$ is a distribution which is being summed over)} \\
&= \mathfrak{P}(x_0') \quad \textit{(eliminate multiplication with 1)}
\end{aligned}
$$

Hence, $Pnext(n_0') = \mathfrak{P}(x_0')$ if $\mathfrak{P}$ is the current state, and $\mathfrak{P}$ is in detailed balance with the kernel K. This means that $\mathfrak{P}$ is the stationary distribution of M. $\triangle$

2.3 Metropolis Hastings

There are three key players in the metropolis hasting sampler:

1. $\mathfrak{P} : X \to \mathbb{R}$: the probability distribution we wish to sample from.
2. $\texttt{Prop} : X \to (X \to \mathbb{R})$. For each $x_0 \in X$, provide a distribution $\texttt{Prop}(x) : X \to \mathbb{R}$ that is used to sample points around x_0. `Prop` for *proposal*.
3. $M[\mathfrak{P}, \texttt{Prop}] \equiv (X, K[\mathfrak{P}, \texttt{Prop}] : X \to \mathbb{R})$: The Metropolis Markov chain we will sample from, whose stationary distribution is $\mathfrak{P}$ — M for *Markov*.

We want the stationary distribution of $M[\mathfrak{P}, \texttt{Prop}]$ to be $\mathfrak{P}$. We also wish for $K[\mathfrak{P}, \texttt{Prop}](x_0) \sim \texttt{Prop}(x_0)$: That is, at a point x_0, we want to choose new points in a way that is 'controlled' by the proposal distribution $\texttt{Prop}(x_0)$, since this will allow us to 'guide' the markov chain towards regions where $\mathfrak{P}$ is high. If we had a gradient, then we could use $\mathfrak{P}'$ to 'move' from the current point x_0 to a new point. Since we lack a gradient, we will provide a custom $\texttt{Prop}(x_0)$ for each x_0 that will tell us how to pick a new x', in a way that will improve $\mathfrak{P}$. So, we tentatively define

$$K[\mathfrak{P}, \texttt{Prop}](x)(x') \stackrel{?}{\equiv} \texttt{Prop}(x)(x').$$

Recall that for $\mathfrak{P}$ to be a stationary distribution of K, it is sufficient for $\mathfrak{P}$ to be in detailed balance for K. So, we write:

$$\mathfrak{P}(x)K(x,x') \stackrel{?}{=} \mathfrak{P}(x')K(x',x) \quad \text{(going forward equally likely as coming back)}$$
$$\mathfrak{P}(x)Prop(x)(x') \stackrel{?}{=} \mathfrak{P}(x')Prop(x')(x) \quad \text{(this is a hard condition to satisfy)}$$

This is far too complicated a condition to impose on $Prop$ and $\mathfrak{P}$, and there is no reason for this condition to be satisfied in general. Hence, we add a custom

"fudge factor" $\alpha \in X \to (X \to \mathbb{R})$ that tells us how often to transition from x to x'. We redefine the kernel as:

$$K[\mathfrak{P}, \texttt{Prop}](x)(x') \equiv \texttt{Prop}(x)(x')\alpha(x)(x')$$

Redoing detailed balance with this new K, we get:

$\mathfrak{P}(x)K(x,x') \stackrel{?}{=} \mathfrak{P}(x')K(x',x)$ (going forward equally likely as coming back)
$\mathfrak{P}(x)Prop(x)(x')\alpha(x)(x') = \mathfrak{P}(x')Prop(x')(x)\alpha(x')(x)$ (Fudge a hard condition with α)
$\frac{\alpha(x)(x')}{\alpha(x')(x)} = \frac{\mathfrak{P}(x')Prop(x)(x')}{\mathfrak{P}(x)Prop(x')(x)}$ (Find conditions for α)

What we have above is a *constraint* for α. We now need to *pick* an α that satisfies this. A reasonable choice is:

$$\alpha(x)(x') \equiv \min\left(1, \frac{\mathfrak{P}(x')\texttt{Prop}(x')(x)}{\mathfrak{P}(x)\texttt{Prop}(x)(x')}\right)$$

.

since K is a transition kernel, we canot have its entries be greater tha n 1. Hence, we choose to clamp it with a $\min(1, \cdot)$. This finally gives us the kernel as:

$$\begin{aligned}
K[\mathfrak{P}, \texttt{Prop}](x)(x') &\equiv \texttt{Prop}(x)(x')\alpha(x)(x') \\
K[\mathfrak{P}, \texttt{Prop}](x)(x') &= \texttt{Prop}(x)(x') \min\left(1, \frac{\mathfrak{P}(x')\texttt{Prop}(x')(x)}{\mathfrak{P}(x)\texttt{Prop}(x)(x')}\right) \\
K[\mathfrak{P}, \texttt{Prop}](x)(x') &= \min\left(\texttt{Prop}(x)(x'), \frac{\mathfrak{P}(x')\texttt{Prop}(x')(x)}{\mathfrak{P}(x)}\right)
\end{aligned}$$

We can make sure that detailed balance is satisfied:

$$\begin{aligned}
&\mathfrak{P}(x)K[\mathfrak{P}, \texttt{Prop}](x)(x') = \mathfrak{P}(x) \min\left(\texttt{Prop}(x)(x'), \frac{\mathfrak{P}(x')\texttt{Prop}(x')(x)}{\mathfrak{P}(x)}\right) \\
&= \min\left(\mathfrak{P}(x)\texttt{Prop}(x)(x'), \mathfrak{P}(x)\frac{\mathfrak{P}(x')\texttt{Prop}(x')(x)}{\mathfrak{P}(x)}\right) \\
&= \min\left(\mathfrak{P}(x)\texttt{Prop}(x)(x'), \mathfrak{P}(x')\texttt{Prop}(x')(x)\right)
\end{aligned}$$

Note that the above right-hand-side is symmetric in x and x', and hence we can state that:

$$\mathfrak{P}(x)K[\mathfrak{P}, \texttt{Prop}](x)(x') = \min\left(\mathfrak{P}(x)\texttt{Prop}(x)(x'), \mathfrak{P}(x')\texttt{Prop}(x')(x)\right) = \mathfrak{P}(x')K[\mathfrak{P}, \texttt{Prop}](x')(x)$$

Hence, we can wrap up, stating that our design of K does indeed give us a markov chain whose stationary distribution is $\mathfrak{P}$, since $\mathfrak{P}$ is detailed balanced with $K[\mathfrak{P}, \texttt{Prop}]$. As an upshot, we also gained a level of control with `Prop`, where we are able to provide "good" samples for a given point.

2.4 Simplification when proposal is symmetric

If our function $\texttt{Prop}$ is symmetric: $\forall x, x' \texttt{Prop}(x)(x') = \texttt{Prop}(x')(x)$, then a lot of the above derivation becomes much simpler. We will perform those simplifications here for pedagogy.

When we have $\texttt{Prop}(x)(x') = \texttt{Prop}(x')(x)$, we can simply α:

$$\alpha(x)(x') \equiv \min\left(1, \frac{\mathfrak{P}(x')\texttt{Prop}(x')(x)}{\mathfrak{P}(x)\texttt{Prop}(x)(x')}\right)$$
$$\alpha(x)(x') = \min\left(1, \frac{\mathfrak{P}(x')}{\mathfrak{P}(x)}\right) \quad \text{[cancelling: } \texttt{Prop}(x)(x') = \texttt{Prop}(x')(x)\text{]}$$

This also makes the kernel look a lot more pleasing:

$$K[\mathfrak{P}, \texttt{Prop}](x)(x') \equiv \texttt{Prop}(x)(x')\alpha(x)(x')$$
$$K[\mathfrak{P}, \texttt{Prop}](x)(x') = \texttt{Prop}(x)(x') \min\left(1, \frac{\mathfrak{P}(x')}{\mathfrak{P}(x)}\right)$$

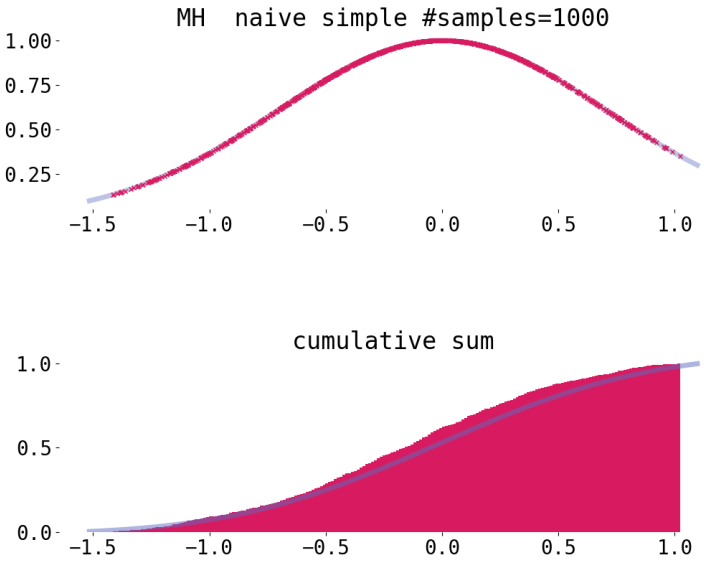
```
# prob is the distribution to sample from;
# symproposal is the *symmetric* proposal function
# symproposal: X -> X; produces a new 'X' from a
#              given 'X' with some distribution.
# prob: X -> |R: gives probability of point 'xi'.
# N: number of markov chain walks before returning a new sample.
def metropolis_hastings(prob, symproposal, x0, N):
  x = x0
  while True:
    for i in range(N):
      # xnext chosen with Prop(x)(x') prob.
      xnext = symproposal(x); px = prob(x); pxnext = prob(xnext);
      # x' chosen with Prop(x)(x') * alpha prob.
      r = uniform01(); alpha = min(1, pxnext / px); if r < alpha: x = xnext
    yield x
```

2.5 Implementing MH: Devil in the detailed balance

2.5.1 Naive implementation

Let us try to transcribe the equations we have derived for symmetric proposal into code, and see what happens. Let us choose $\mathfrak{P}(x) \equiv \texttt{normal}(0,1) = e^{-x*x}$ and the proposal function to be $P(x_0)(x) = \texttt{normal}(x_0, \texttt{1e-2})$ is, a Gaussian centered around x_0 with standard deviation `1e-2`. This gives us the code:

```
#  mcmc1d.py
def mhsimple(x0, prob, prop):
    yield x0; x = x0;
    while true:
        xnext = prop(x); p = prob(x); pnext = prob(xnext)
        r = np.random.uniform() + 1e-5;
        if r < pnext/p: x = xnext
        yield xnext
...
def exp(x): return np.exp(-x*x)
def expprop(x): return np.random.normal(loc=x, scale=1e-1)
...
nsamples = 1000
xs = list(itertools.islice(mhsimple(0, exp, expprop), nsamples))
```

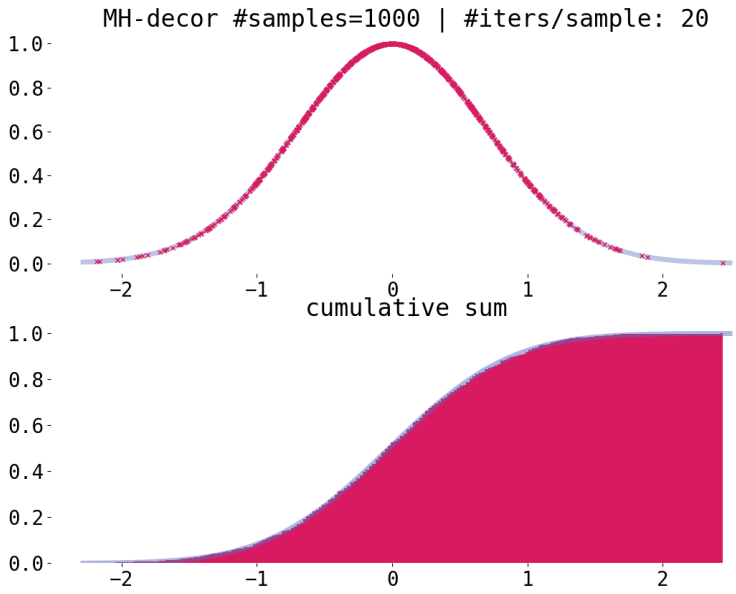


We can see from the plots of the raw samples and the cumulative distribution that we wind up overshooting. This is because the markov chain, by definition, has correlations between samples; However, we are supposed to be drawing *independent* samples from $\mathfrak{P}$ for plotting/downstream use.

2.5.2 Sampling per `iters_per_sample` steps

The solution is to take samples that are *spaced out* — that is, we do not consider each step from the markov chain as a sample. Rather, we consider the state we are in after some `iters_per_sample` steps to be a sample. In code, this is now:

```
#  mcmc1d.py
def mh_uncorr(x0, prob, prop, iters_per_sample):
    yield x0; x = x0;
    while True:
        for i in range(iters_per_sample):
            xnext = prop(x); p = prob(x); pnext = prob(xnext)
            r = np.random.uniform() + 1e-5;
            if r < min(1, pnext/p): x = xnext
        yield xnext
...
def exp(x): return np.exp(-x*x)
def expprop(x): return np.random.normal(loc=x, scale=1e-1)
...
NSAMPLES = 1000; ITERS_PER_SAMPLE = 20
xs = list(itertools.islice(mhsimple(0, exp, expprop, ITERS_PER_SAMPLE), NSAMPLES))
```



3 Hamiltonian Monte Carlo

If our target probability density $\mathfrak{P} : X \to [0, 1]$ is *differentiable*, then we can use the derivative of $\mathfrak{P}(x)$ to provide better proposals. The idea is as follows:

- Interpret the probability landscape as a potential, with points of high probability being "valleys" and points of low probability being "peaks" (ie, invert the probability density with a transform such as $U[\mathfrak{P}](x) = -\log \mathfrak{P}(x)$. This way, a ball rolling on this terrain will try to move towards the valleys — which are the locations of high probability $\mathfrak{P}$.
- For a proposal at a position $x_0 \in X$, keep a ball at x_0, *randomly choose its velocity*, simulate the ball according to classical mechanics (Newton's laws of motion) for a fixed duration $D \in \mathbb{R}$ and propose the final position as the final position of the ball. This is reversible and detailed balanced because *classical mechanics is reversible and detailed balanced.*

We wish to sample from a new distribution P_H, which is defined as:

$$
\begin{aligned}
&P_H[\mathfrak{P}](\mathbf{p},\mathbf{q}) \equiv \exp(-H[\mathfrak{P}](\mathbf{p},\mathbf{q})) \qquad \text{(Sample states with low energy: Boltzmann/Ising model)}\\
&= \exp(-(U[\mathfrak{P}](\mathbf{q}) + K(\mathbf{p}))) \qquad \text{(by defn. of hamiltonian, } H[\mathfrak{P}] = U[\mathfrak{P}] + K)\\
&= \exp(-(-\log \mathfrak{P}(\mathbf{q}) + \mathbf{p}^T\mathbf{p}/2)) \qquad \text{(by choice of } U \text{ as NLL, } K \text{ as classical KE)}\\
&= \exp(\log \mathfrak{P}(\mathbf{q}) - \mathbf{p}^T\mathbf{p}/2))\\
&= \exp(\log \mathfrak{P}(\mathbf{q})) \cdot \exp(-\mathbf{p}^T\mathbf{p}/2)\\
&= \mathfrak{P}(\mathbf{q}) \cdot \exp(-\mathbf{p}^T\mathbf{p}/2)
\end{aligned}
$$

Note that the probability distribution for $P_H[\mathfrak{P}]$ has split into two independent distributions: $\mathfrak{P}(\mathbf{q})$, and $\exp(-\mathbf{p}^T\mathbf{p}/2)$. Hence, sampling $(\mathbf{p},\mathbf{q}) \sim P_H[\mathfrak{P}]$ correctly is *equivalent* to sampling $\mathbf{q} \sim \mathfrak{P}$, and $\mathbf{p} \sim \exp(-\mathbf{p}^T\mathbf{p}/2)$. Therefore, if we can sample from P_H effectively, taking only $(\mathbf{q}, \_) \sim P_H[\mathfrak{P}]$ will correctly give us the samples we are looking for.

We will show that using MCMC to sample from the $P_H[\mathfrak{P}]$ above, with a particular proposal function $\texttt{Prop}_H$ that we shall define momentarily will provide us with samples $(\mathbf{p},\mathbf{q})$ such that the $\mathbf{q}$ is sampled from $\mathfrak{P}$. This begs the question: "why do we need $\mathbf{p}$ anyway?" We will show how introducing this $\mathbf{p}$ allows for rapid exploration of the space of possible $\mathbf{q}$s.

3.1 The proposal function for HMC

In classical mechanics, once we have a hamiltonian $H(\mathbf{q},\mathbf{p})$, the dynamics of the system are simulated according to the equations (called Hamilton's equations):

$$\frac{\partial \mathbf{q}}{\partial t} = \frac{\partial H[\mathfrak{P}]}{\partial \mathbf{p}} \qquad \frac{\partial \mathbf{p}}{\partial t} = \frac{-\partial H[\mathfrak{P}]}{\partial \mathbf{q}}$$

These equations, specialized to our situation become:

$$
\begin{aligned}
&\frac{\partial \mathbf{q}}{\partial t} = \frac{\partial H[\mathfrak{P}]}{\partial \mathbf{p}} = \frac{\partial (U[\mathfrak{P}](\mathbf{q}) + K(\mathbf{p}))}{\partial \mathbf{p}} = \frac{\partial K(\mathbf{p})}{\partial \mathbf{p}} = \frac{\partial (\mathbf{p}^T\mathbf{p})}{\partial \mathbf{p}} = 2\mathbf{p}\\
&\frac{\partial \mathbf{p}}{\partial t} = \frac{-\partial H[\mathfrak{P}]}{\partial \mathbf{q}} = \frac{-\partial (U[\mathfrak{P}](\mathbf{q}) + K(\mathbf{p}))}{\partial \mathbf{q}} = \frac{-\partial U[\mathfrak{P}](\mathbf{q})}{\partial \mathbf{q}} = \frac{-\partial - \log \mathfrak{P}(\mathbf{q})}{\partial \mathbf{q}} = \frac{\partial \log \mathfrak{P}(\mathbf{q})}{\partial \mathbf{q}}
\end{aligned}
$$

We will denote a solution to the above system of equations as $(\mathbf{q}_{final}, \mathbf{p}_{final}) \equiv X_H[\mathfrak{P}](\mathbf{q}_0, \mathbf{p}_0, T_0)$. That is, starting from the position $(\mathbf{p}_0, \mathbf{q}_0)$, simulate the system for $T_0 \in \mathbb{R}$ time, according to Hamilton's equations. The final (position, momentum) is $(\mathbf{q}_{final}, \mathbf{p}_{final})$.

Now, our proposal function will be:

$$\texttt{Prop}(\mathbf{q}_0, \_) \equiv \texttt{let } \mathbf{p}_0 \sim normal(0,1) \texttt{ in } X_H[\mathfrak{P}](\mathbf{q}_0, \mathbf{p}_0, T_0)$$

That is, we will propose the new point as one obtained from time evolution of the solutions to the hamiltonian; (a) we start from the point $\mathbf{q}_0$; (b) pick a *random velocity* $\mathbf{p}_0$; (c) simulate the system for T_0 time using hamilton's equations; (d) return the final (position, momentum) as our proposal.

3.2 Simulating the above equations to sample $\mathfrak{P} \equiv \exp(-x^2)$

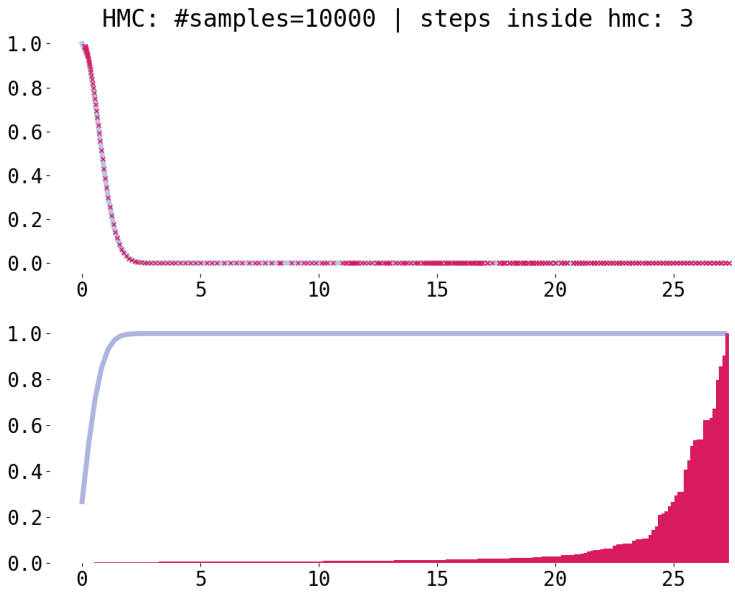
```
def euler(dhdp, dhdq, q, p, dt): % f(x + dx) = f(x) + f'(x) * dx
   pnew = p + -dhdq(q, p) * dt; qnew = q + dhdp(q, p) * dt
   return (qnew, pnew)


def hmc(q0, U, dU, nsteps, dt):
    # hamiltonian definition
    def h(q, p): return U(q) + 0.5*p*p
    def hdp(q, p): return  p
    def hdq(q, p): return dU(q)
    def nextsample(q, p): # run `euler` for n steps
        for _ in range(nsteps):
            (q, p) = euler(hdq, hdp, q, p, dt)
        return (q, p)

    yield q0; q = q0
    while True:
        p = np.random.normal(0, 1) # (a) pick random momentum
        (qnext, pnext) = nextsample(q, p) # (b) simulate next sample
        pnext = -p # (c) reverse momentum so our process is reversible
        r = np.random.uniform(); # (d) if accept according to MH, accept.
        if np.log(r) < h(q, p) - h(qnext, pnext): q = qnext
        yield q # return point
```

Let's quickly explain the code;

(a) We start by picking a random momentum for the proposal, which is sampled from a uniform normal distribution. This is `p = np.random.normal(0, 1)`.

(b) We now find the point `(qnext, pnext)` which is the value $(\mathbf{q}_{next}, \mathbf{p}_{next}) = X_H[\mathfrak{P}](\mathbf{q}, \mathbf{p}, dt)$. Unfortunately, we do not have a closed form for X_H. So, we solve Hamilton's equations using *numerical integration*. We use the Euler integrator, which uses the taylor expansion: $f(x + \Delta x) \simeq f(x) + f'(x)\Delta x$. In the code, this is performed by the function `euler`.

(c) We compute the acceptance probability α as $\mathfrak{P}_H(\mathbf{q}_{next}, \mathbf{p}_{next})/P_H(\mathbf{q}, \mathbf{p}) = \exp(H(\mathbf{q}_{next}, \mathbf{p}_{next}) - H(\mathbf{q}, \mathbf{p}))$. Rather that computing the acceptance as $r < \alpha$, we compute $\log r < \alpha$ for better numerical accuracy. This leads to the code being `if np.log(r) < h(q, p) - h(qnext, pnext)`.



This has sampled disastrously: what is going wrong?

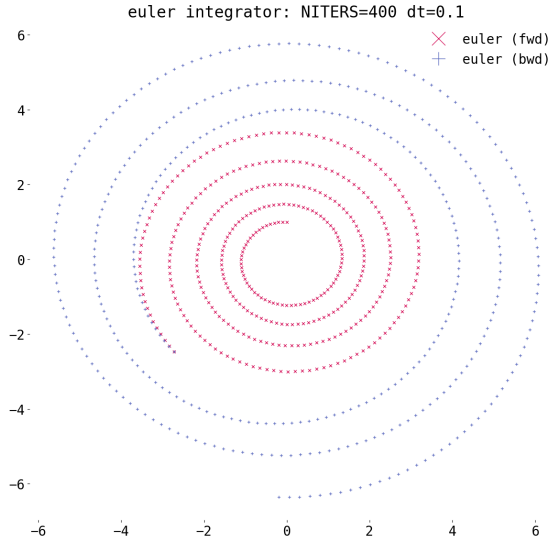
3.3 Simulation: Euler integration

Unfortunately, it turns out that running this simulation is in fact numerically unstable on using a naive integrator such as Euler. We shall explore this by trying to integrate the orbits of a planet under gravitational force. We are using a bog-standard Hamiltonian with $K(\mathbf{p}) = \mathbf{p}^T\mathbf{p}$, $U(\mathbf{q}) = \sqrt{(}\mathbf{q}^T\mathbf{q})$, $H(\mathbf{q},\mathbf{p}) = U(\mathbf{q})+K(\mathbf{p})$. We will integrate Hamilton's equations using the `euler` integrator and look at the orbits. Ideally, we want the orbits to be **time-reversible**, since our **proposal should be symmetric**!

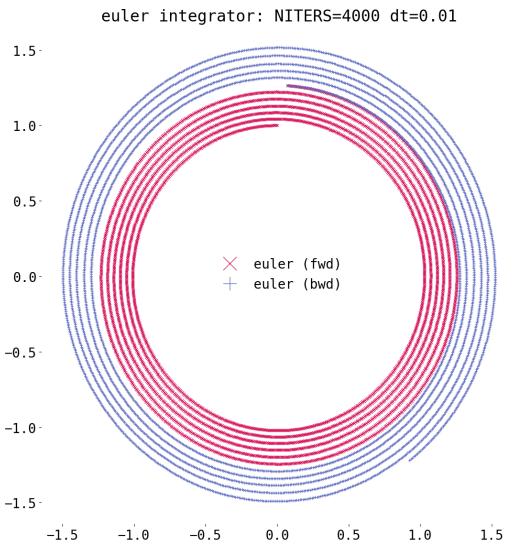
Let us say that we wish to simulate the orbit of a planet. Recall that we want the proposal to be symmetric: so, if we simulate the trajectory of the planet for N timesteps, each timestep of time δt, and then *reverse* the momentum of the planet, run the next phse of the simulation for N timesteps with timestep Δt, we should begin where we started. However, if we try to use the euler integration equations, here is what we see:

```
def euler(dhdp, dhdq, q, p, dt):
   pnew = p + -dhdq(q, p) * dt
   qnew = q + dhdp(q, p) * dt
   return (qnew, pnew)
...
def planet(q, p, integrator, n, dt):
```

```
STRENGTH = 0.5
# minimise potential V(q): q, K(p, q) p^2
# H = STRENGTH * |q| (potential) + p^2/2 (kinetic)
def H(qcur, pcur): return STRENGTH * np.linalg.norm(q) + np.dot(p, p) / 2
def dhdp(qcur, pcur): return p
def dhdq(qcur, pcur): return STRENGTH * 2 * q / np.linalg.norm(q)

for i in range(n): (q, p) = integrator(dhdp, dhdq, q, p, dt) ...
return np.asarray(qs), np.asarray(ps)
```



We can clearly see the forward trajectory (in pink) spiralling out, and the backward trajectory (in blue), spiralling out even more. We can attempt to fix this by making Δt smaller:

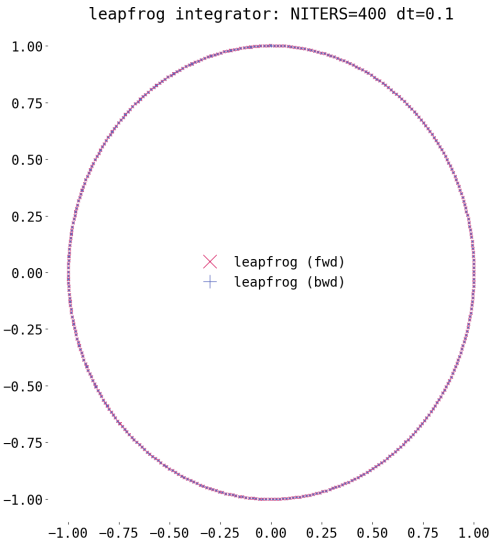


to no avail. Indeed, this is a **fundamental limitation of euler integration**. Hence, we will need to explore more refined integration schemes.

3.4 Simulation: Symplectic integrator

The type of integrators that will allow us to get 'reasonable orbits' that do not decay with time are knows as *symplectic integrators*. (An aside: the word *symplectic* comes from Weyl, who substituted the latin root in the word *complex* by the corresponding greek root. It is a branch of differential geometry which formalizes the consructs needed to carry out hamiltonian mechanics on spaces that are more complicated that Euclidian $\mathbb{R}^n$).

```
## dq/dt = dH/dp|_{p0, q0}, dp/dt = -dH/dq|_{p0, q0}
def leapfrog(dhdp, dhdq, q0, p0, dt):
    p0 += -dhdq(q0, p0) * 0.5 * dt # kick: half step momentum
    q0 += dhdp(q0, p0) * dt # drift: full step position
    p0 += -dhdq(q0, p0) * 0.5 * dt # kick: half step momentum
    return (q0, p0)
```



It is clear from the plots that the leapfrog integrator is stable: orbits stay as orbits, and running-in-reverse works as expected. We provide a formal proof of time reversibility below. We also prove that this type of integrator preserves energy. In total, these two properties ensure that the **proposals we make will be symmetric**.

3.5 Proof that leapfrog is time-reversible

We will assume that $H(\mathbf{q}, \mathbf{p}) \equiv U(\mathbf{q}) + K(\mathbf{p})$. That is, the kinetic energy depends only on momentum, and the potential energy depends only on position. This allows us to write the derivatives as $\frac{\partial H}{\partial \mathbf{q}} \equiv U'(\mathbf{q})$, $\frac{\partial H}{\partial \mathbf{p}} \equiv K'(\mathbf{p})$. This also modifies Hamilton's equations into:

$$\frac{\partial \mathbf{q}}{\partial t} = \frac{\partial H(\mathbf{q}, \mathbf{p})}{\partial \mathbf{p}} = K'(\mathbf{p}) \qquad \frac{\partial \mathbf{p}}{\partial t} = -\frac{\partial H(\mathbf{q}, \mathbf{p})}{\partial \mathbf{q}} = -U'(\mathbf{q})$$

We will also assume that $K'(-\mathbf{p}) = -K'(\mathbf{p})$. This is true of our usual choice of kinetic energy being $K(\mathbf{p}) \equiv \mathbf{p}^T\mathbf{p}$. We will write down the forward simulation according to the leapfrog equations, and then the backward steps. This will show us that the final state of the backward equations $(\mathbf{q}_1', \mathbf{p}_2')$ will be equal to the reversed initial stte $(\mathbf{q}_0, -\mathbf{p}_0)$.

fwd 1 $[\mathbf{q}_0, \mathbf{p}_0]$

fwd 2 $[\mathbf{q}_0, \mathbf{p}_1 \equiv \mathbf{p}_0 - 0.5U'(\mathbf{q}_0)dt]$

fwd 3 $[\mathbf{q}_1 \equiv \mathbf{q}_0 + K'(\mathbf{p}_1)dt, \mathbf{p}_1]$

fwd 4 $[\mathbf{q}_1, \mathbf{p}_2 \equiv \mathbf{p}_1 - 0.5U'(\mathbf{q}_1)dt]$

bwd 1 $\left[\mathbf{q}_0' \equiv \mathbf{q}_1 = \mathbf{q}_0 + K'(\mathbf{p}_1)dt, \mathbf{p}_0' \equiv -\mathbf{p}_2 = \mathbf{p}_1 - 0.5U'(\mathbf{q}_1)dt\right]$

bwd 2 $\left[\mathbf{q}_0', \mathbf{p}_1' \equiv \mathbf{p}_0' - 0.5U'(\mathbf{q}_0')dt = -(\mathbf{p}_1 - 0.5U'(\mathbf{q}_1)dt) - 0.5U'(\mathbf{q}_1)dt = -\mathbf{p}_1\right]$

bwd 3 $\left[\mathbf{q}_1' \equiv \mathbf{q}_0' + K'(\mathbf{p}_1')dt = (\mathbf{q}_0 + K'(\mathbf{p}_1)dt) + K'(-\mathbf{p}_1)dt = (\mathbf{q}_0 + K'(\mathbf{p}_1)dt) - K'(\mathbf{p}_1)dt = \mathbf{q}_0, \mathbf{p}_1'\right]$

bwd 4 $\left[\mathbf{q}_1' \equiv \mathbf{q}_0, \mathbf{p}_2' \equiv \mathbf{p}_1' - 0.5U'(\mathbf{q}_1') = -\mathbf{p}_1 - 0.5U'(\mathbf{q}_0)dt = -(\mathbf{p}_0 - 0.5U'(\mathbf{q}_0)) - 0.5U'(\mathbf{q}_0)dt = -\mathbf{p}_0\right]$

n	fwd q	fwd p
1	$\mathbf{q}_0$	$\mathbf{p}_0$
2	$\mathbf{q}_0$	$\mathbf{p}_1 = \mathbf{p}_0 - 0.5U'(\mathbf{q}_0)dt$
3	$\mathbf{q}_1 = \mathbf{q}_0 + K'(\mathbf{p}_1)dt$	$\mathbf{p}_1$
4	$\mathbf{q}_1$	$\mathbf{p}_2 = \mathbf{p}_1 - 0.5U'(\mathbf{q}_1)dt$

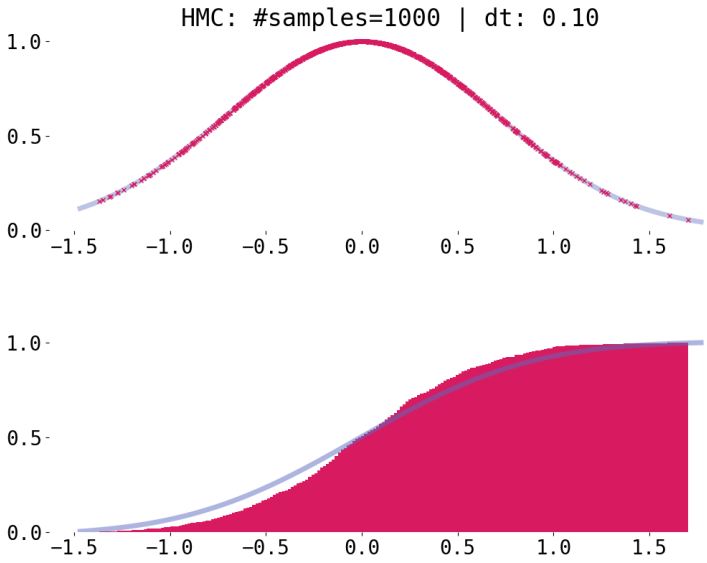
n	bwd q	bwd p
1	$\mathbf{q}_0' = \mathbf{q}_1$	$\mathbf{p}_0' = -\mathbf{p}_2$
2	$\mathbf{q}_0' = \mathbf{q}_1$	$\mathbf{p}_1' = -\mathbf{p}_1$
3	$\mathbf{q}_1' = \mathbf{q}_0$	$\mathbf{p}_1' = -\mathbf{p}_1$
4	$\mathbf{q}_1' = \mathbf{q}_0$	$\mathbf{p}_2' = -\mathbf{p}_0$

Indeed, we see that the backward equations where we set $(\mathbf{q} \mapsto \mathbf{q}, \mathbf{p} \mapsto -\mathbf{p})$ retraces our dynamics, just running backward. Hence, **the leapfrog equations are reversible**, and therefore act as a **symmetric proposal**.

3.6 Using HMC on the 1D gaussian

Now that we have the math to setup a symmetric proposal distribution using leapfrog, let's code up HMC on a 1D gaussian:

```
def leapfrog(dhdq, dhdp, q0, p0, dt):
    p0 += -dhdq(q0, p0) * 0.5 * dt  # kick: half step momentum
    q0 += dhdp(q0, p0) * dt # drift: full step position
    p0 += -dhdq(q0, p0) * 0.5 * dt # kick: half step momentum
    return (q0, p0)

def hmc(q0, U, dU, nsteps, dt):
    def h(q, p): return U(q) + 0.5 * p*p
    def nextsample(q, p):
        for _ in range(nsteps):
            def hdp(q, p): return p
            def hdq(q, p): return dU(q)
            (q, p) = leapfrog(hdq, hdp, q, p, dt)
        return (q, p)

    yield q0; q = q0
    while True:
        p = np.random.normal(0, 1)
        (qnext, pnext) = nextsample(q, p)
        pnext = -p # reverse momentum so our process is reversible
        r = np.random.uniform();
        if np.log(r) < h(q, p) - h(qnext, pnext): q = qnext
        yield q
...
def neglogexp(x): return -1 * logexp(x)
def neglogexpgrad(x): return -1 * logexpgrad(x)
xs = list(take_every_nth(DECORRELATE_STEPS,
    itertools.islice(hmc(1, neglogexp, neglogexpgrad, NSTEPS, DT), NSAMPLES*DECORRELATE_STEPS)))
```

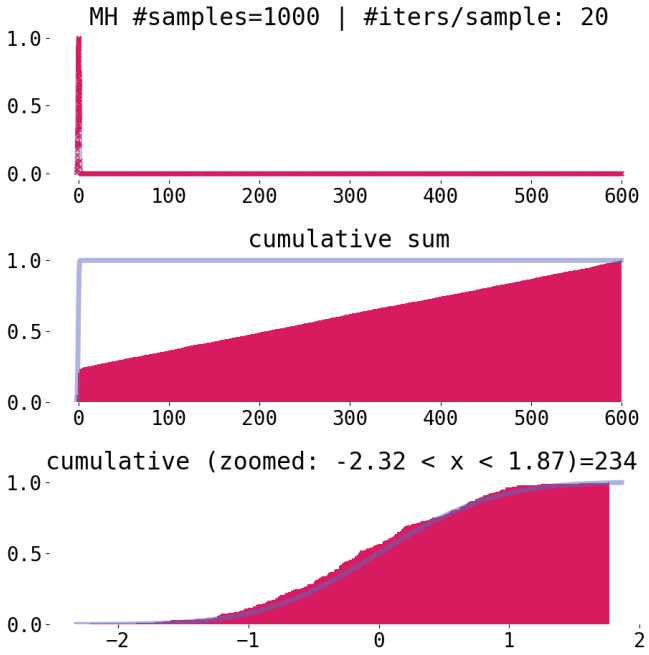


This looks far worse than the results we got from vanilla metropolis hastings: why would anybody use this technique?

3.7 HMC v/s MH when the proposal is atypical

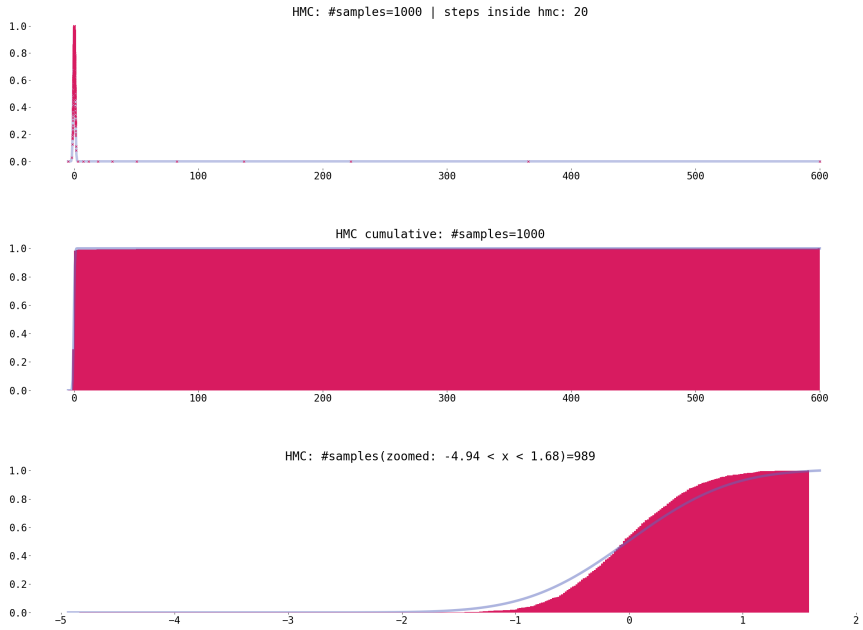
So far, I've hidden one thing under the rug: the *initialization* of the distribution. We've been starting at $x = 1$ — let's now change that, and start at $x = 600$. Note that this is very far from the "region of interest" in the case of a standard normal distribution: 99% of the probability mass is in $[-3, 3]$ (the 3σ rule).

3.7.1 MH starting at $x = 600$



Note that out of the 1000 samples we started from, only 234 are present in the region of $-2 \leq x \leq 2$. We have wasted around 80% of the samples we have *moving through the space* to get to the typical set.

3.7.2 HMC starting at $x = 600$



Note that out of the 1000 samples we started from, only 987 are present in the region of $-2 \leq x \leq 2$. Notice the distribution of samples (pink crosses) in the hamiltonian monte-carlo case: most samples clustered around the gaussian; our initial samples that are far away are powered by a strong potential energy to move towards the center.

This is in stark contrast to the MH case, where the entire x-axis is pink, due to the 'current point' having to move, proposal-by-proposal (which allows at most a movement of distance 1), from 600 to 0.

References

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

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

[3] Marin Kobilarov, Keenan Crane, and Mathieu Desbrun. Lie group integrators for animation and control of vehicles. *ACM transactions on Graphics (TOG)*, 28(2):1–14, 2009.