A detailed reference of MCMC algorithms

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

April 21, 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 Fundamental Problem of MCMC sampling

Given a weak, simple sampler of the form $\texttt{rand} : S \to S \times \texttt{int}$, build a sampler $\texttt{sampler}(P, X) : T \to T \times X$ which returns value distributed according to some *un-normalized* distribution of choice $P : X \to \mathbb{R}$.

The un-normalized constraint is important: it is what allows us to exploit MCMC to a wide variety of scienarios.

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. Recall the formla:

$$P(Y|X) = P(X|Y).P(Y)/P(X)$$

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.

1.4 Sampling use case 3. Numerical 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 Detail 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 **detail balanced** with respect to M.

Theorem 1 *If $\mathfrak{P}$ is detail 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 detail 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:

$$\begin{aligned}
&Pnext(x'_0) \equiv \sum_x \mathfrak{P}(x)K(x)(x'_0) \\
&= \sum_x \mathfrak{P}(x'_0)K(x'_0)(x) \quad \textit{(by detail balance)} \\
&= \mathfrak{P}(x'_0) \sum_x K(x'_0)(x) \quad \textit{($P(x_0)$ is constant} \\
&= \mathfrak{P}(x'_0) \cdot 1 \quad \textit{($K(x'_0)$ is a distribution which is being summed over)} \\
&= \mathfrak{P}(x'_0) \quad \textit{(eliminate multiplication with 1)}
\end{aligned}$$

Hence, $Pnext(n'_0) = \mathfrak{P}(x'_0)$ if $\mathfrak{P}$ is the current state, and $\mathfrak{P}$ is in detail 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. $\texttt{Prop}$ 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 detail 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 detail balance with this new K, we get:

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

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 detail 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 detail 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 `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:

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

This also makes the kernel look a lot more pleasing:

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

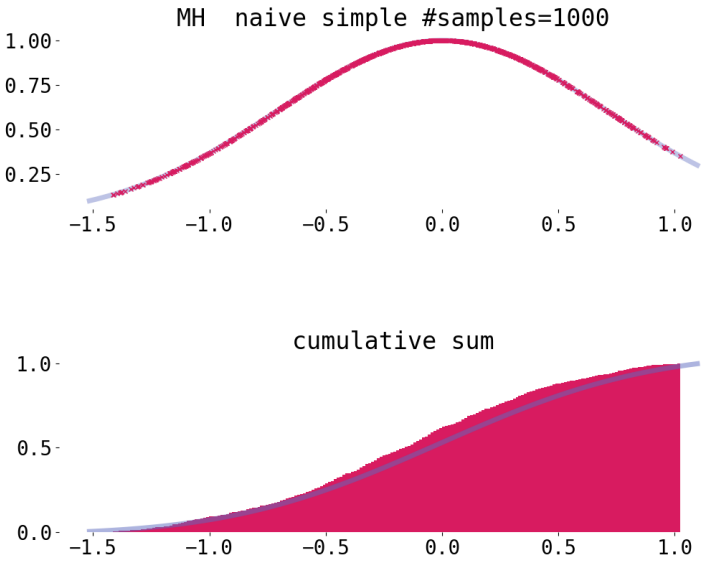
```
# prob is the distribution to sample from;
# symproposal is the *symmetric* proposal function
# symproposal: X -> X; produces a new 'X' from a
#              given 'X' with some distribution.
# prob: X -> |R: gives probability of point 'xi'.
# N: number of markov chain walks before returning a new 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 Details

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))
```

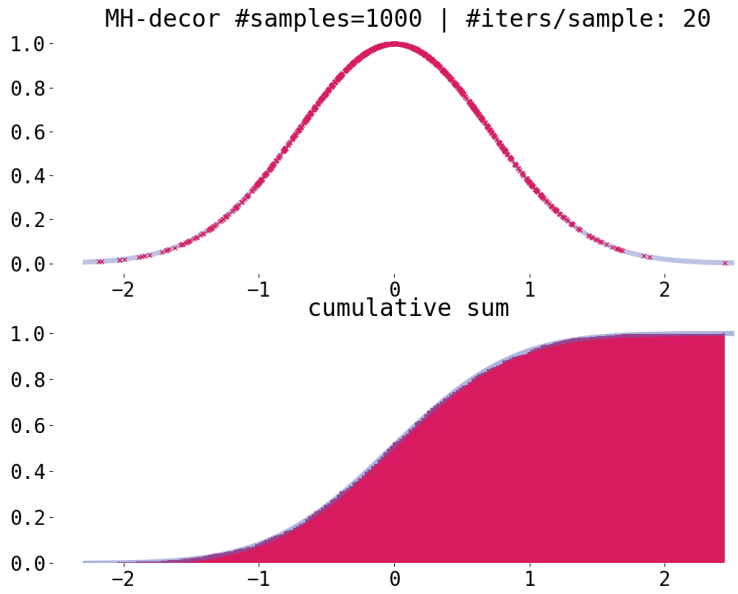


We can see from the plots of the raw samples and the cumulative distribution that we wind up overshooting. This is because the markov chain, by definition, has correlations between samples; However, we are supposed to be drawing *independent* samples from $\mathfrak{P}$ for plotting/downstream use.

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 $U[\mathfrak{P}](x) = e^{-\mathfrak{P}(x)}$. 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 detail balanced because *classical mechanics is reversible and detail balanced.*

We need the derivative of $\mathfrak{P}(x)$ to be able to simulate the ball according to Newton's laws. If we can do this, though, we are able to cover large amounts of terrain.

3.1 Newton's laws of motion: Hamilton's equations

$$\frac{\partial \mathbf{q}}{\partial t} = \frac{\partial H}{\partial \mathbf{p}} \qquad \frac{\partial \mathbf{p}}{\partial t} = -\frac{\partial H}{\partial \mathbf{q}}$$

In our case, the choice of the hamiltonian will consider the negative-log-probability-density to be the *energy*. So, higher probability has lower energy. We know that particles like to go towards lower energy states.

$$H(\mathbf{q}, \mathbf{p}) \equiv U(\mathbf{q}) + K(\mathbf{p}) \quad K(\mathbf{p}) \equiv \mathbf{p}^T M^{-1} \mathbf{p}/2$$

where M is a symmetric positive-definite matrix known as a "mass matrix".

3.2 Modifying Hamilton's equations for generating proposals

We wish to maximize $\mathfrak{P}(x)$. To do this, we will consider a new energy function $U[\mathfrak{P}](x) \equiv -\log(\mathfrak{P}(x))$. When $U[\mathfrak{P}] = -\log(\mathfrak{P}(x))$ is minimized, then $\mathfrak{P}(x)$ will be maximized.

We use the proposal function as:

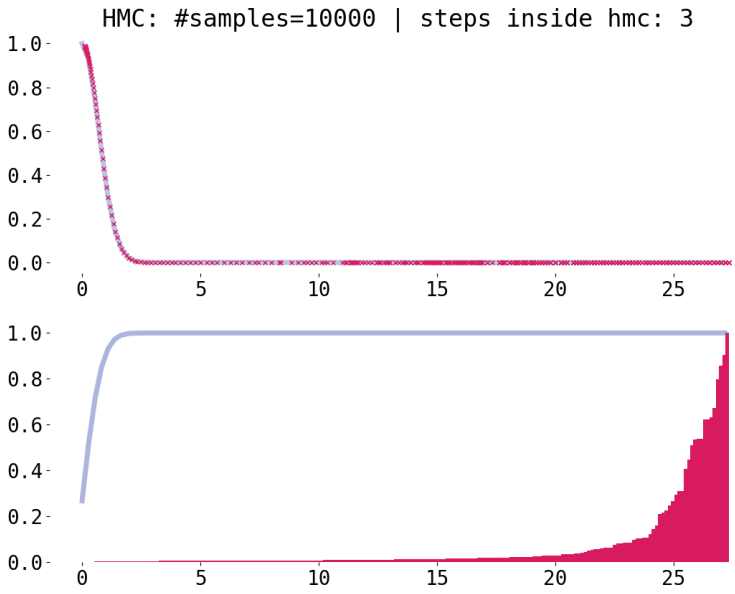
- We are currently at the location $\mathbf{q}_0$.
- Pick a uniform randomly momentum $\mathbf{p}$.
- Simulate the system according to the differential equations mentioned above, to produce a new $(\mathbf{q}', \mathbf{p}')$.
- Use this $(\mathbf{q}', \mathbf{p}')$ as the proposed point from the proposal.
- Use metropolis-hastings accept-reject.

3.3 Simulating the above regime

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



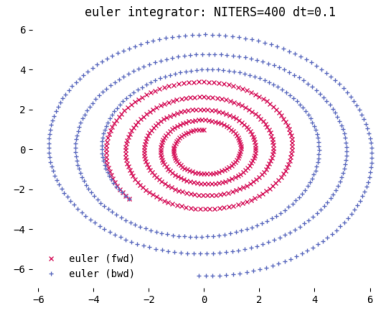
This has sampled disastrously: what is going wrong?

3.4 Simulation: Euler integration

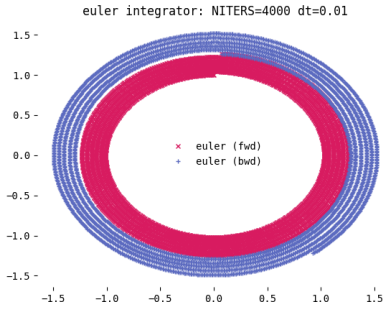
Unfortunately, it turns out that running this simulation is in fact numerically unstable on using a naive simulation schemes. For example, let us say that we wish to simulate the orbit of a planet. Recall that we want the proposal to be symmetric: so, if we simulate the trajectory of the planet for N 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)
```



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:

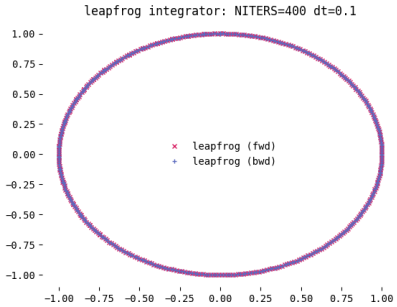


to no avail. Indeed, this is a **fundamental limitation of euler integration**. Hence, we will need to explore more refined integration schemes.

3.5 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. Indeed, we can prove the symplecticity/energy-preserving property of this integrator.

3.6 Proof that leapfrog is symplectic

We will show that in phase space, the leapfrog integrator preserves infinitesimal volumes of the form $dV \equiv d\mathbf{p} \times d\mathbf{q}$.

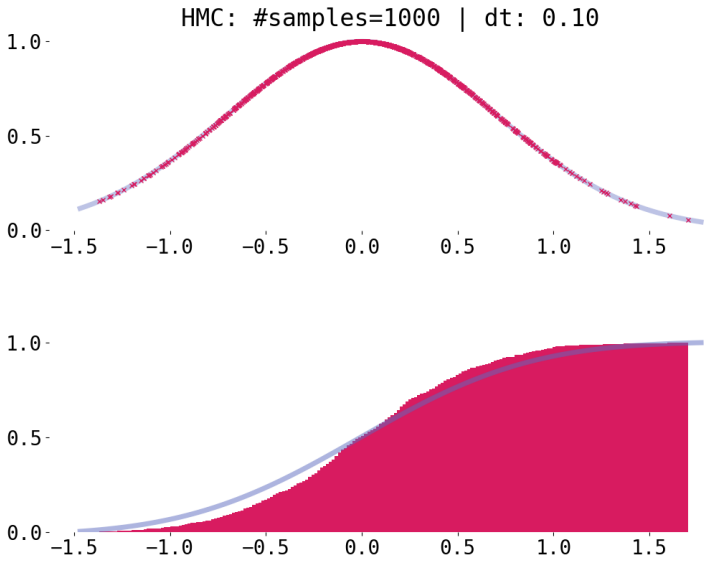
3.7 Using HMC on the 1D gaussian

Now that we have the math to setup a symmetric proposal distribution, 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)))
```

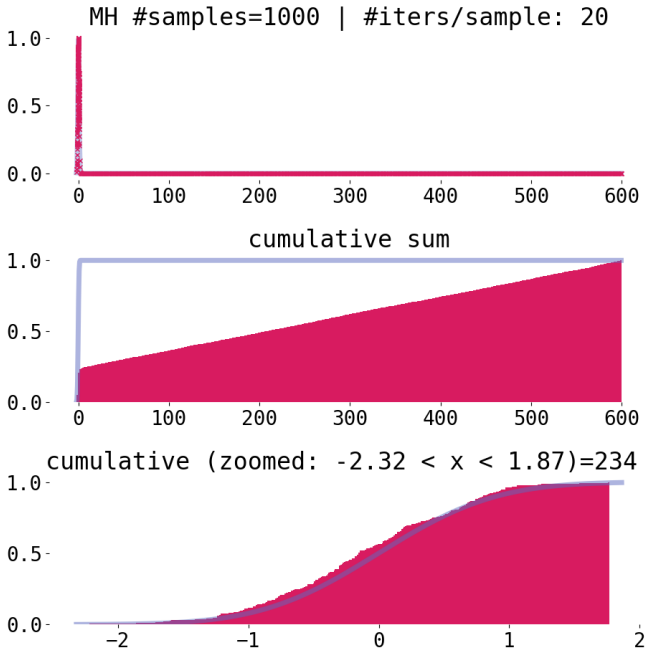


This looks far worse than the results we got from vanilla metropolis hastings: why would anybody use this technique?

3.8 HMC v/s MH when the proposal is atypical

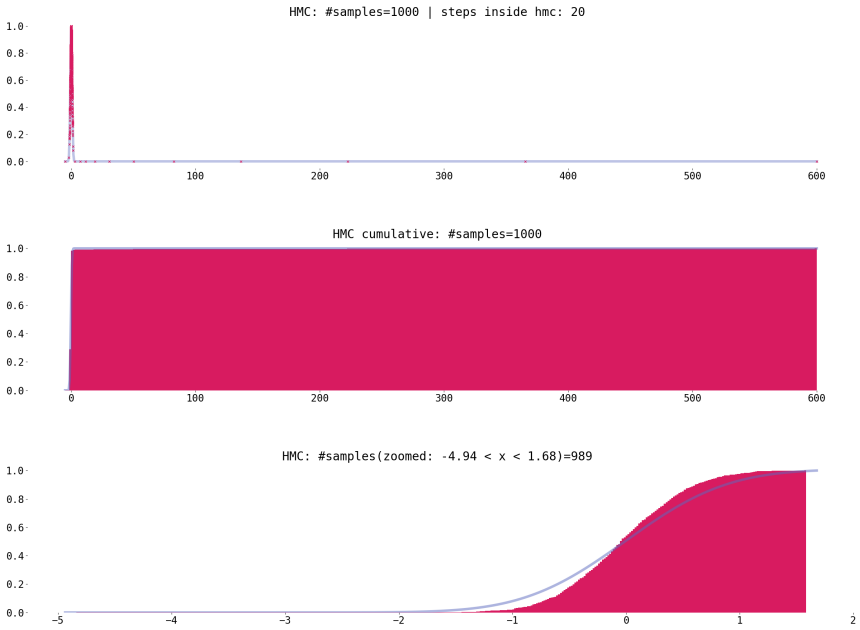
So far, I've hidden one thing under the rug: the *initialization* of the distribution. We've been starting at $x = 1$ — let's now change that, and start at $x = 600$. Note that this is very far from the "region of interest" in the case of a standard normal distribution: 99% of the probability mass is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

3.8.1 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.

3.8.2 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.