HMC for Univariate Gaussian case

August 2, 2018

This is a mini report to explore the implementation of HMC for Univariate Gaussian. First, I utilize HMC to draw 1000 samples from N(0,1) through the analytic solutions of Hamilton's Equation. I have included them in appendix for references. Next, I plot the dynamics of HMC in this situation to observe how the estimator moves in detail. I plot 3 trajectories, coupled with the dynamics. Finally, the pseudocode is presented for this simple case.

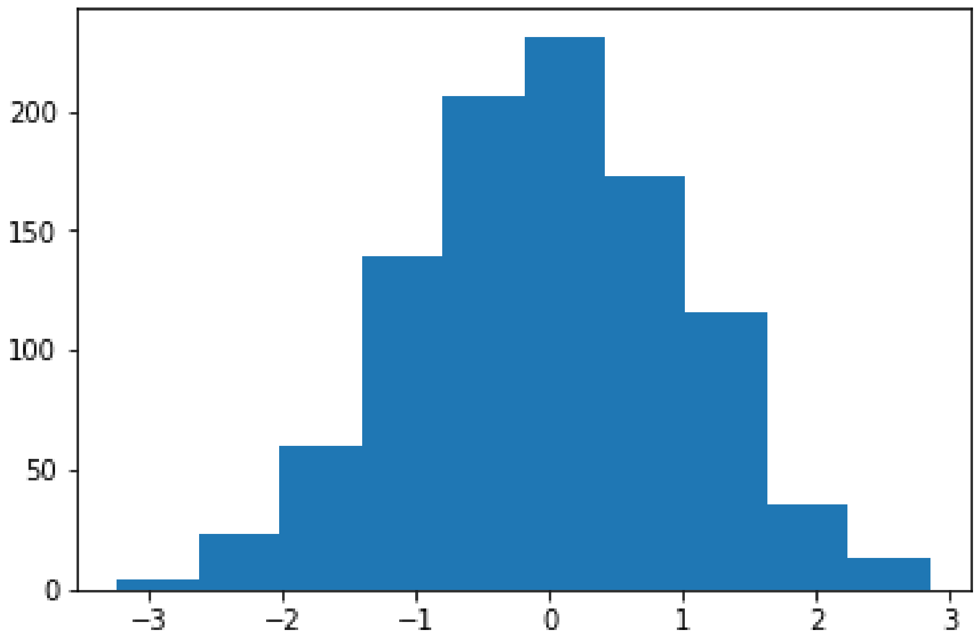
The following are Python codes to generate 1000 samples from N(0,1):

```
In [66]: import numpy as np
         import math
         import matplotlib.pyplot as plt

         #generate M number of samples
         M=1000

         q=np.zeros(M)
         p=np.zeros(M)

         #start from a random point in pi(q), or start from any arbitrary point
         q0=np.random.normal(0,1,1)
         #q0=10
         # set a fixed integrating time
         t = math.pi/2
         for m in range(M):
             if m==0:
                 q_0=q0
             p_0=np.random.normal(0,1,1)
             p_t=p_0*math.cos(t) - q_0*math.sin(t)
             q_t=p_0*math.sin(t) - q_0*math.cos(t)
             q[m]=q_t
             p[m]=p_t
             q_0 = q_t
         print('sample mean is', np.mean(q))
```

```
        print("sample var is", np.var(q))
        plt.hist(q)
        plt.show()
```

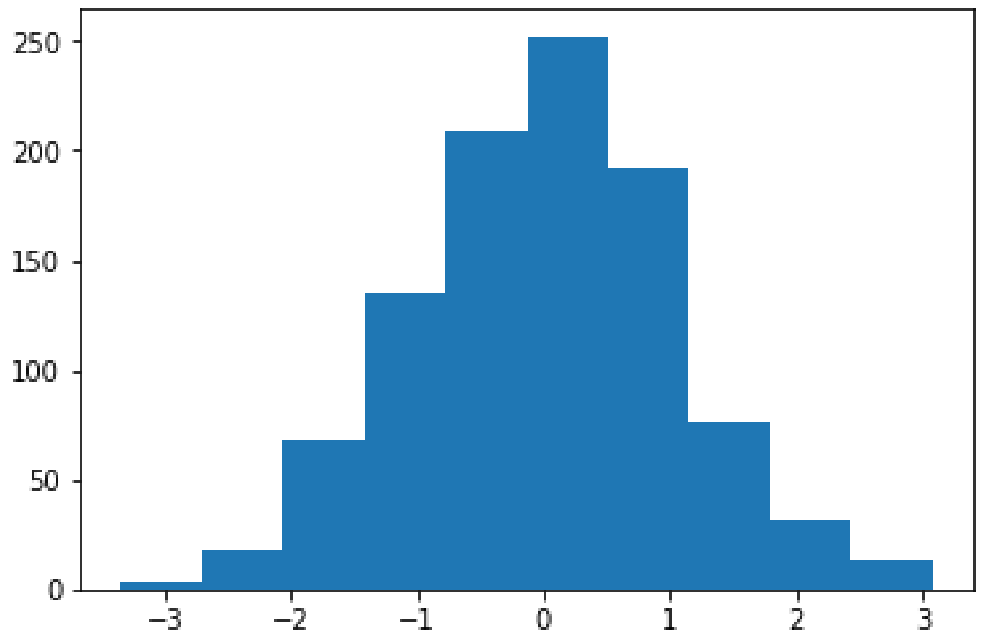
And the statistical properties and histogram of the output:

```
sample mean is -0.011566658609l3969
sample var is 1.0293501950765684
```



If we directly generate 1000 samples from N(0,1), using:

```
In [64]: plt.hist(np.random.normal(0,1,1000))
         plt.show()
```



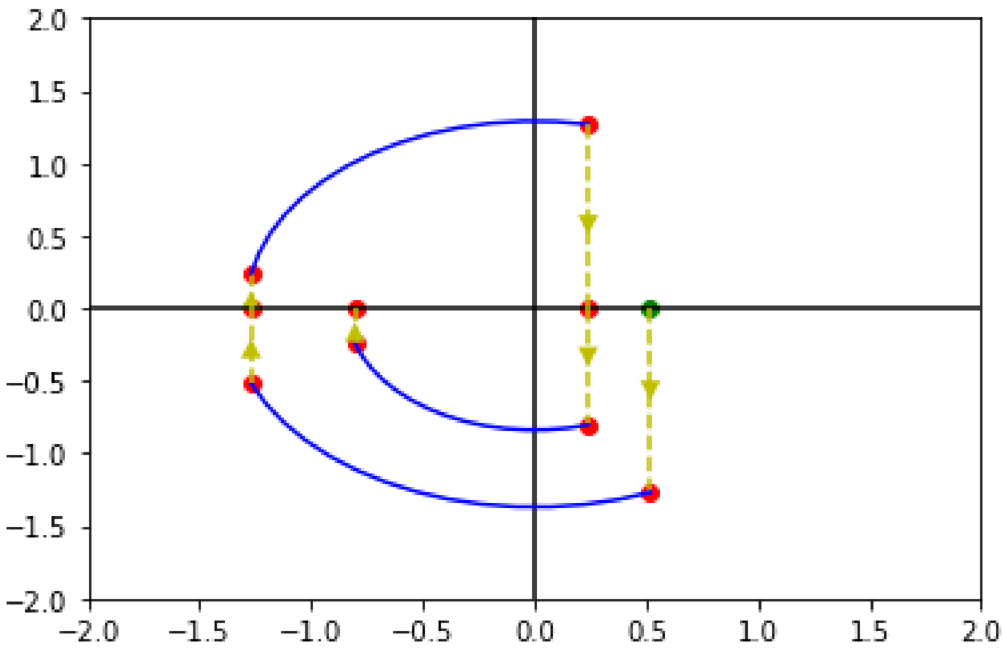
Comments: HMC sampler gives us a very nice output, according to histogram comparison as well as sample mean and variance calculation. In my experiment, if we start from any arbitrary point, say 50, the sampler comes back to the effective region very quickly, resulting in extremely small amount of outliers(i.e 1 or 2). It works well even for a ridiculous starting point, 5000 for instance. However, the only hyperparameter we need to tweak carefully is the integration time T. Either too short or long will cause ineffective exploration and significantly affect the result. I will reason this observation more comprehensively in my thesis. In this case, I used $\frac{\pi}{2}$ as it gives relatively good outputs, but yet the optimal choice I guess.

Next I plot 3 iterations of the above sampler in python. Here are the codes:

```
In [232]: import os
          os.chdir('/Users/rui/Documents/MasterThesis(Yuhui Xia)')
          import random
          import numpy as np
          import math
          import matplotlib.pyplot as plt

          np.random.seed(52)
          fig=plt.figure()
          fig,ax=plt.subplots()
          ax.set_ylim([-2,2])
          ax.set_xlim([-2,2])
          ax.axhline(y=0,color='k')
```

```
        ax.axvline(x=0,color='k')

        x0=np.random.normal(0,1,1)
        T=np.linspace(0,math.pi/2, num=15)
        ax.scatter(x0,0,color='g')
        for j in range(3):
            y0=np.random.normal(0,1,1)
            ax.scatter(x0,y0,color='r')
            ax.plot([x0,x0],[0,y0],linestyle="--",color='y')
            plt.arrow(x0[0],0,0,y0[0]/2, shape='full', color='y',
                      lw=0, length_includes_head=True, head_width=.09)
            P=[y0]
            Q=[x0]

            def ptraj(q_0,p_0,t):
                p_t=p_0*math.cos(t) - q_0*math.sin(t)
                return p_t
            def qtraj(q_0,p_0,t):
                q_t=p_0*math.sin(t) + q_0*math.cos(t)
                return q_t

            for t in T:
                q_t=qtraj(x0,y0,t)
                p_t=ptraj(x0,y0,t)
                Q.append(q_t)
                P.append(p_t)
            ax.plot(Q,P,'y-',color='b')
            ax.scatter(Q[-1],P[-1],color='r')
            ax.scatter(Q[-1],0,color='r')
            ax.plot([Q[-1],Q[-1]],[P[-1],0],linestyle='--',color='y')
            plt.arrow(Q[-1][0],P[-1][0]/2,0,-P[-1][0]/10, shape='full', color='y',
                      lw=0, length_includes_head=True, head_width=.09)
            x0=Q[-1]
        fig.savefig('HMC_dynamic.png')
```

```
<matplotlib.figure.Figure at 0x11f93c128>
```



Comments: In this graph, the green dot is our starting point and the reds are samples generated. The blue curves are moving trajectories. Therefore, it is intuitive to understand HMC as a process involving 3 steps:

step 1: starting from the green dot(x_0), we generating a y_0 from N(0,1) so that we lift the green to a 2 dimensional plane, located at (x_0,y). (just like random walk, the green dot can be lifted up or pulled down anywhere vertically)

step 2: integrate the red dot over some time T, according to the analytical solutions of Hamilton's equations, which forms the blue trajectories. This step is deterministic!

step 3: Project away the new y at the end of the trajectory, and we are left with a generated sample point.

Appendices

The analytical solutions of Hamilton's Equations.

Assume we need to draw samples from N(0,1). So

$$x \sim N(0,1)$$

Introducing

$$y \sim N(0,1)$$

Algorithm 1 HMC for standard normal

```
Given $x_0$ as the starting point, integration time t and M, the number of samples required
for $m = 1$ to $M$ do:
    Sample $y_0$ from $N(0,1)$
    $x_t = y_0 sin(t) + x_0 cos(t)$
    $y_t = y_0 cos(t) - x_0 sin(t)$
    Store $x_t$ and $y_t$ in matrices $X$ and $Y$
    $x_0 \leftarrow x_t$
end for
Return X, Y
```

We have seen the general solutions for Hamilton's Equation are:

$$x_t = Bsin(t) - Acos(t)$$

$$y_t = Asin(t) + Bcos(t)$$

Where A,B depends on the initial conditions. Now that if we have obtained x_0 and y_0, the solutions become:

$$x_0 = Bsin(0) - Acos(0) = -A$$

$$y_0 = Asin(0) + Bcos(0) = B$$

Therefore, we obtain the the formulas in Algorithm 1, as follow:

$$x_t = y_0 sin(t) + x_0 cos(t)$$

$$y_t = y_0 cos(t) - x_0 sin(t)$$