SDS386D: Monte Carlo Methods in Statistics **Cesar N. Yahia**
Homework 3 Prof. S.G. Walker

1.

Suppose $\pi(a,b) \propto a^4 b^6 e^{-a-b-3ab}$
for $a, b > 0$ is required to be sampled.
Find $\pi(a|b)$ and $\pi(b|a)$
and hence find a transition density $p(a_{n+1}, b_{n+1}|a_n, b_n)$ which has π as the stationary distribution.

$\pi(a|b) \propto a^4 e^{-a-3ab} \propto a^4 e^{-a(1+3b)}$ which is $Gamma(.|5, 1+3b)$ where rate $= \beta = 1+3b$ and shape $= \alpha = 5$

$\pi(b|a) \propto b^6 e^{-b-3ab} \propto b^6 e^{-b(1+3a)}$ which is $Gamma(.|7, 1+3a)$ where rate $= \beta = 1+3a$ and shape $= \alpha = 7$
Therefore, the transition density is given by $P(a_{n+1}, b_{n+1}|a_n, b_n) = \pi(a_{n+1}|b_{n+1})\pi(b_{n+1}|a_n)$. This has $\pi(a,b)$ as the stationary.

Implement the chain and use the output to evaluate the integral $I = \int\int ab\pi(a,b)dadb$
The algorithm is as follows:
(1) start with $a_{n=0}$ set at any value
At an iteration n
(2) sample b_{n+1} from $\pi(b_{n+1}|a_n)$
(3) sample a_{n+1} from $\pi(a_{n+1}|b_{n+1})$
(4) $\hat{I}_{n+1} = \frac{n\hat{I}_n + a_{n+1}b_{n+1}}{n+1}$, $\hat{I}_0 = 0$

This algorithm was implemented in Python as follows:
Note that when you are sampling from a gamma in Python, the input is the scale parameter instead of the rate parameter, where the scale θ is given by $\theta = \frac{1}{\beta}$

```
import numpy.random as nprand
import matplotlib.pyplot as plt

# define pi(a_{n+1}|b_{n+1})
def piab(bnp1):
        """
        This is the marginal of pi(a|b)
        :param bnp1: b_{n+1} from previous pi(b|a)
        :return: a_{n+1}
        """
        anp1 = nprand.gamma(5, 1.0 / (1 + 3 * bnp1))
        return anp1

# define pi(b_{n+1} | a_{n})
def piba(an):
        bnp1 = nprand.gamma(7, 1.0 / (1 + 3 * an))
        return bnp1
```

```
# implement the sampling and integration
if __name__ == '__main__':
        numofiter = 1  # iterator
        total_iter = 5000  # total iterations
        numerator = []
        Ihat = []  # store integral value
        an = 2  # initial value of a
        while numofiter <= total_iter:
                bnplus1 = piba(an)
                anplus1 = piab(bnplus1)
                product = anplus1 * bnplus1
                numerator.append(product)
                an = anplus1  # update the current value of an
                Ihat.append(sum(numerator) / len(numerator))
                numofiter += 1

        print("... Integral estimator ...")
        print(Ihat[-1])
        # plot a graph showing convergence of integral
        Iterations = range(1, len(Ihat) + 1)
        plt.plot(Iterations, Ihat, 'g')
        plt.title("integral estimator across iterations")
        plt.xlabel("Iterations")
        plt.ylabel("Ihat")
        plt.tight_layout()
        plt.savefig('Ihat.png')
        plt.show()
```

For 5000 iterations, the results indicate that the integral estimator $\hat{I} = 1.44$

The figure below shows the convergence of the estimator towards its value

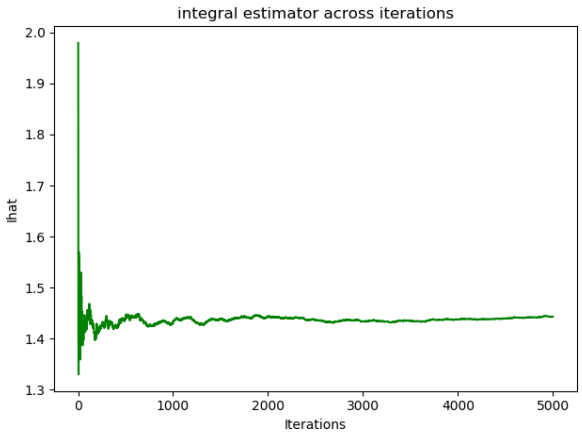

Figure 1: Convergence of estimator $\hat{I}$ towards 1.44

2.

Suppose we wish to sample $\pi(x) \propto \frac{\exp(-\frac{1}{2}x^2)}{1+x^2}$ using a MH algorithm with proposal density $q(x'|x) = N(x'|x, \sigma^2)$ (Normal centered at previous value with variance σ^2)

If σ is two small, the algorithm would require a long time to converge to the stationary since you will be searching in the vicinity of the current sample instead of exploring the rest of the distribution $\pi(x)$. This is especially problematic if the starting value is far from the concentration of $\pi(x)$ since we will be spending a lot of time sampling values that do not represent the range over which the density is concentrated. Even if we start at a location with high density, a small variance would imply that it would take a long time to explore the tails of the distribution.

If the variance σ^2 is too high, a lot of the proposal will be to places where π is small, and those proposals will be rejected. Therefore, we will have a low acceptance probability and it will take a long time to explore the desired stationary distribution.

The Metropolis-Hastings algorithm:
(1) start with an arbitrary initial value x_0
(2) sample $x^\star_{n+1}$ from $q(x_{n+1}|x_n) = N(x_n, \sigma^2)$
(3) sample u from $U(0,1)$ (a uniform on $(0,1)$)
(4) evaluate $\alpha(x^\star_{n+1}, x_n) = \min\{1, \frac{\pi(x^\star_{n+1})}{\pi(x_n)} \frac{q(x_n|x^\star_{n+1})}{q(x^\star_{n+1}|x_n)}\}$
(5) if $u < \alpha(x^\star_{n+1}, x_n)$ then $x_{n+1} = x^\star_{n+1}$. Else, $x_{n+1} = x_n$

We still have to determine the expression for $\alpha(x^\star_{n+1}, x_n)$
We can see that $\frac{q(x_n|x^\star_{n+1})}{q(x^\star_{n+1}|x_n)} = 1$, also dropping the star for notation purposes, we can see that $\frac{\pi(x^\star_{n+1})}{\pi(x_n)} = \frac{e^{(-1/2)X^2_{n+1}}(1+X^2_n)}{e^{(-1/2)X^2_n}(1+X^2_{n+1})}$ which can be evaluated easily.

Following that, we implement the above algorithm as shown below:

```
import numpy.random as nprand
import matplotlib.pyplot as plt
import numpy as np


# define alpha
def alpha(xn1, xn):
        numerator = np.exp(-0.5 * xn1**2) * (1 + xn**2)
        denominator = np.exp(-0.5 * xn**2) * (1 + xn1**2)
        fraction = numerator / denominator
        alfa = min(1.0, fraction)
        return alfa


# define function for sampling
def normal_sample(xn, var):
        standard_dev = np.sqrt(var)
```

```
        sample = nprand.normal(loc=xn, scale=standard_dev)
        return sample


# implement metropolis hastings algorithm
if __name__ == '__main__':
        numofiter = 1  # iterator
        xn = 3  # start with an arbitrary value of x0
        variance = 1  # choose a variance for proposal
        number_of_iterations = 5000
        number_accept = 0
        number_reject = 0
        samples = []
        while numofiter <= number_of_iterations:
                samples.append(xn)
                xn1 = normal_sample(xn, variance)
                u = nprand.uniform(low=0.0, high=1.0)
                alpha_val = alpha(xn1, xn)
                if u < alpha_val:
                        number_accept += 1
                        xn = xn1  # for next round take xn1
                else:
                        number_reject += 1
                        # do nothing, for next round xn = xn
                numofiter += 1

        # plot a histogram of samples
        plt.hist(samples, bins=50, facecolor='g', normed=True)
        plt.title("pi(x)")
        plt.xlabel("x")
        plt.ylabel("frequency")
        # fig = plt.gcf()
        plt.savefig('piofx.png')
        plt.show()
```

For a variance of 1 and a starting position of $x_0 = 3$, we get the following distribution for $\pi(x)$ after 5000 iterations. The acceptance rate was 60%.

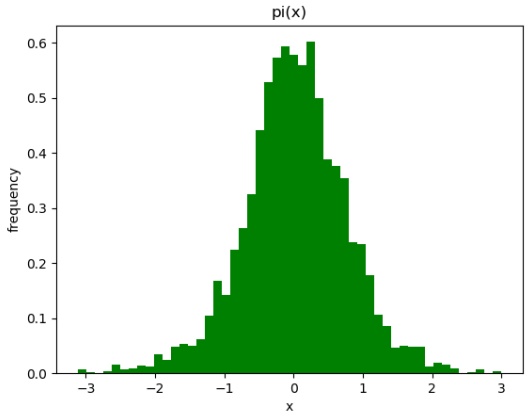

Figure 2: Distribution of $\pi(x)$ for $\sigma^2 = 1$

For demonstration, taking a variance of 0.002 with same starting position and number of iterations, we get the following distribution for $\pi(x)$. As apparent, the algorithm still did not fully explore the space, and we are moving slowly. More iterations are needed to adequately represent $\pi(x)$. Note that we observe a very high acceptance rate in the range of 97% which indicates that almost every place we move to lies in $\pi(x)$. From the figure you can also see as the samples *slowly* move from 3 to the location at which the density is concentrated.

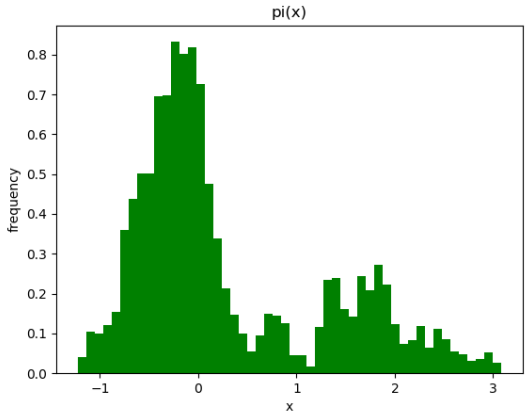

Figure 3: Distribution of $\pi(x)$ for $\sigma^2 = 0.002$

For demonstration, taking a variance of 80 with same starting position and number of iterations, we get the following distribution for $\pi(x)$. As expected, the *acceptance rate was only 9.7%* compared to the 60% for a variance of one. This implies that we reject most moving proposals and we still need more iterations to have enough samples that represent the distribution.

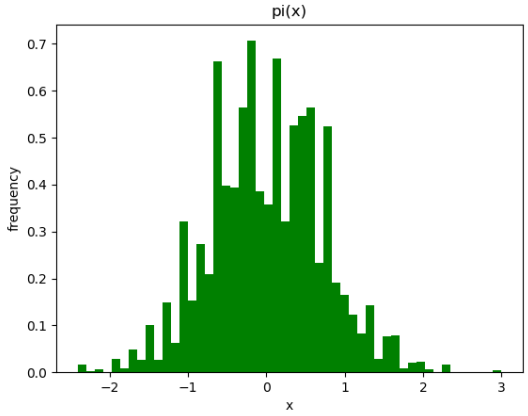

Figure 4: Distribution of $\pi(x)$ for $\sigma^2 = 80$

Based on the results above and the shape of $\pi(x)$, a suitable variance would be $\sigma^2 = 0.3$ which gives a standard deviation of $\sigma^2 = 0.55$. If the distribution was approximately normal a proposal standard deviation of 0.55 would visually be about one standard deviation of the distribution.

The figure below shows the value of the estimator $\hat{I}$ for the integral. As expected from the shape of the distribution, the integral which represents the expected value converges to zero.

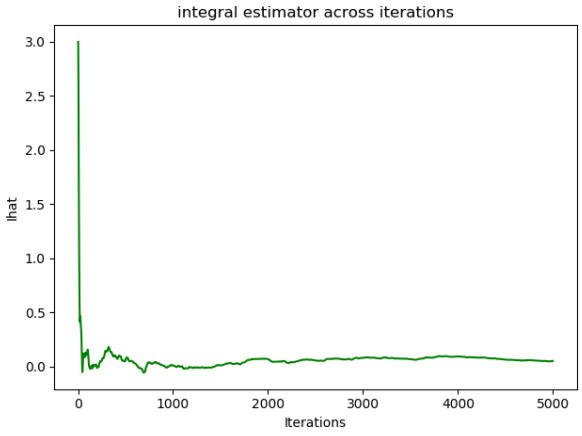

Figure 5: integral estimator

If you chose a proposal $q(x'|x) = N(x'|-x, \sigma^2)$ then you sample from a normal with a mean being negative of the current sample value. This just implies that we look in a different place for the proposals.

To test this we modify the following function in the code by adding a negative sign for the location parameter

```
# define function for sampling
def normal_sample(xn, var):
        standard_dev = np.sqrt(var)
        sample = nprand.normal(loc=-xn, scale=standard_dev)
        return sample
```

We still get similar results as shown below, but you can notice that at the first couple of iterations the integral estimator goes to the negative side indicating that our second sample was less than -3. This happens because we sampled the second value from a proposal centered at -3 due to a starting position of 3

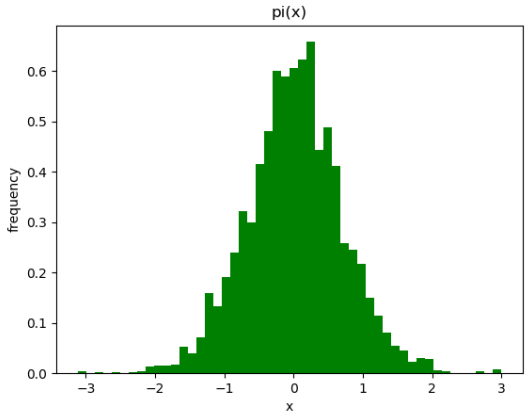

Figure 6: Distribution of $\pi(x)$ for $q(x'|x) = N(x'| - x, \sigma^2)$

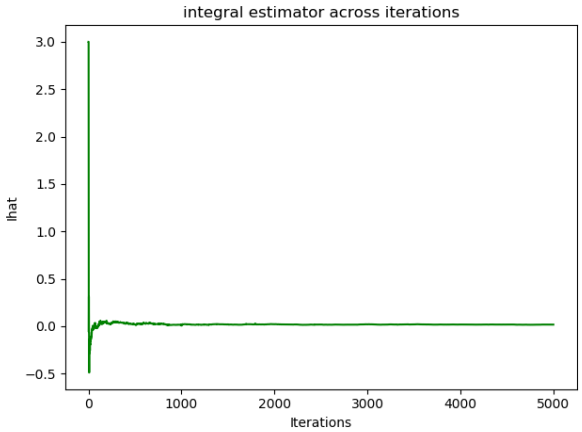

Figure 7: integral estimator for $q(x'|x) = N(x'| - x, \sigma^2)$

3.

Suppose a posterior density for the parameter θ is given by $f(\theta|data) \propto e^{\theta a} e^{-me^{\theta}} e^{-0.5\theta^2}$
This posterior corresponds to a Poisson likelihood with $\lambda = e^{\theta}$ and a standard normal prior on θ.

Let $m = a = 1$, then $f(\theta|data) \propto e^{\theta} e^{-e^{\theta}} e^{-0.5\theta^2}$
To sample from this density we can use a Metropolis-Hastings algorithm. The algorithm proceeds as follows:
(1) start with an arbitrary initial value θ_0
(2) sample $\theta^{\star}_{n+1}$ from $q(\theta_{n+1}|\theta_n)$
(3) sample u from $U(0,1)$ (a uniform on $(0,1)$)
(4) evaluate $\alpha(\theta^{\star}_{n+1}, \theta_n) = \min\{1, \frac{f(\theta^{\star}_{n+1}|data)}{f(\theta_n|data)} \frac{q(\theta_n|\theta^{\star}_{n+1})}{q(\theta^{\star}_{n+1}|\theta_n)}\}$
(5) if $u < \alpha(\theta^{\star}_{n+1}, \theta_n)$ then $\theta_{n+1} = \theta^{\star}_{n+1}$. Else, $\theta_{n+1} = \theta_n$

if I centered this then the terms would have canceled andn alpha would have just been the ratio of the pi's.. actually makes the problem easier

To implement this algorithm we need to specify a proposal $q(\theta_{n+1}|\theta_n)$. One simple proposal that is independent of the current theta value is to take $q(\theta_{n+1}|\theta_n) = N(0,1) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}\theta^2_{n+1}}$ i.e a standard normal distribution. Similarly, $q(\theta_n|\theta_{n+1}) = N(0,1) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}\theta^2_{n}}$. Therefore, $\frac{q(\theta_n|\theta_{n+1})}{q(\theta_{n+1}|\theta_n)} = e^{\frac{1}{2}(\theta^2_{n+1} - \theta^2_n)}$.
We also have that $\frac{f(\theta^{\star}_{n+1}|data)}{f(\theta_n|data)} = \frac{e^{\theta_{n+1}} e^{-e^{\theta_{n+1}}} e^{-0.5\theta^2_{n+1}}}{e^{\theta_n} e^{-e^{\theta_n}} e^{-0.5\theta^2_n}}$.
So, $\alpha(\theta^{\star}_{n+1}, \theta_n) = \min\{1, \frac{f(\theta^{\star}_{n+1}|data)}{f(\theta_n|data)} \frac{q(\theta_n|\theta^{\star}_{n+1})}{q(\theta^{\star}_{n+1}|\theta_n)}\} = \min\{1, \frac{e^{\theta_{n+1}} e^{-e^{\theta_{n+1}}} e^{-0.5\theta^2_{n+1}}}{e^{\theta_n} e^{-e^{\theta_n}} e^{-0.5\theta^2_n}} e^{\frac{1}{2}(\theta^2_{n+1} - \theta^2_n)}\}$ (for notation dropping the star in last expression)

Following that, we implement the algorithm above as shown below:

```
import numpy.random as nprand
import matplotlib.pyplot as plt
import numpy as np


# define alpha
def alpha(thetan1, thetan):
        numerator = np.exp(thetan1) * np.exp(-np.exp(thetan1)) *
        np.exp(-0.5 * thetan1**2) * \
        np.exp(0.5 * (thetan1**2 - thetan**2))
        denominator = np.exp(thetan) * np.exp(-np.exp(thetan)) *
        np.exp(-0.5 * thetan**2)
        fraction = numerator / denominator
        alfa = min(1.0, fraction)
        return alfa


# implement metropolis hastings algorithm
if __name__ == '__main__':
        numofiter = 1  # iterator
        thetan = 3  # start with an arbitrary value of theta0
        number_of_iterations = 5000
        number_accept = 0
```

```
number_reject = 0
samples = []
while numofiter <= number_of_iterations:
        samples.append(thetan)
        thetan1 = nprand.normal(loc=0, scale=1.0)  # sample from
        # standard normal
        u = nprand.uniform(low=0.0, high=1.0)
        alpha_val = alpha(thetan1, thetan)
        if u < alpha_val:
                number_accept += 1
                thetan = thetan1  # for next round take thetan1
        else:
                number_reject += 1
                # do nothing, for next round thetan = thetan
        numofiter += 1

# print fraction accepted
print(float(number_accept / number_of_iterations))

# plot a histogram of samples
plt.hist(samples, bins=50, facecolor='g', normed=True)
plt.title("f(theta)")
plt.xlabel("theta")
plt.ylabel("pdf")
# fig = plt.gcf()
plt.savefig('part3.png')
plt.show()
```

For a starting position of $\theta_0 = 3$, we get the following distribution for $f(\theta|data)$ after 5000 iterations. The acceptance rate was 77%.

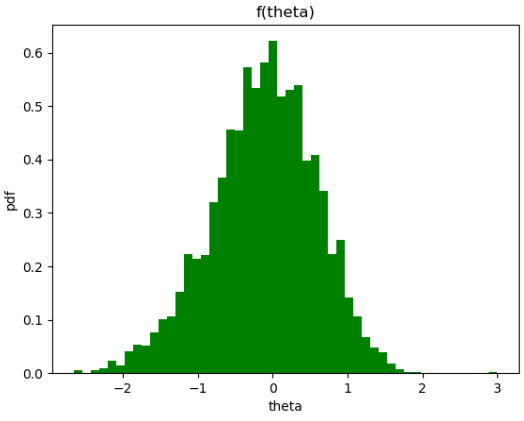

Figure 8: Distribution of $f(\theta|data)$ for m=1, a=1

We change the parameters m and a to see how the posterior would look like for a different data set. Let us take m=3 and a=1. All what this changes in the algorithm is $\alpha(\theta^{\star}_{n+1}, \theta_n)$

Now we have that
$\alpha(\theta_{n+1}, \theta_n) \min\{1, \frac{e^{\theta_{n+1}} e^{-3e^{\theta_{n+1}}} e^{-0.5\theta_{n+1}^2}}{e^{\theta_n} e^{-3e^{\theta_n}} e^{-0.5\theta_n^2}} e^{\frac{1}{2}(\theta_{n+1}^2 - \theta_n^2)}\}$

For a starting position of $\theta_0 = 3$, we get the following distribution for $f(\theta|data)$ after 5000 iterations. The acceptance rate was 53%.

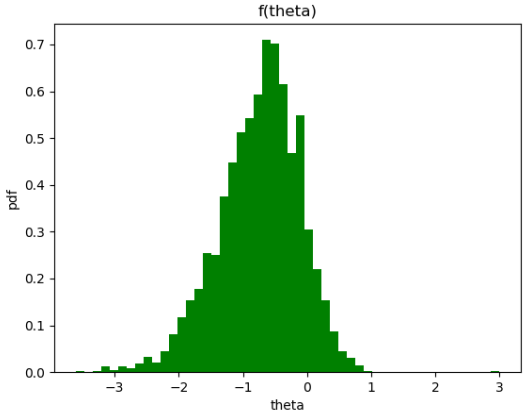

Figure 9: Distribution of $f(\theta|data)$ for m=3, a=1

Notice that the posterior distribution is skewed compared to m=1. This reflects the change in data used to update the posterior.

Note that the Markov chain corresponding to the Metropolis-Hastings algorithm has the following transition density.
$P(\theta_{n+1}|\theta_n) = \frac{\Gamma(x_n)\alpha(\theta_{n+1},\theta_n)q(\theta_{n+1}|\theta_n)}{\Gamma(\theta_n)} + (1 - \Gamma(\theta_n))1(\theta_{n+1} = \theta_n)$ where
$\Gamma(\theta_n) = \int \alpha(\theta_{n+1}, \theta_n)q(\theta_{n+1}|\theta_n)$
We showed in class and the previous homework that stationarity is satisfied for this Markov chain transition density.