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

1.

Suppose it is required to sample from the density $\pi(x)$. Also suppose $\pi(x)$ is invertible, in the sense that if $\pi(x) > u$ then the set of x for which this holds is known, and call it $A_u = \{x : \pi(x) > u\}$. Consider the joint density
$f(x,u) = \mathbf{1}(0 < u < \pi(x))$
Hence, describe the Gibbs sampler for sampling from $\pi(x)$. Provide all the necessary details.

We can implement a Gibbs sampler by sampling from $f(u|x)$ and $f(x|u)$ repeatedly, then throwing away the sampled u values.
$f(u|x)$ is a uniform distribution (pdf=1) on $(0, \pi(x))$. $f(u|x) \sim U(0, \pi(x))$.
In our case, $f(u|x) \sim U(0, \frac{1}{1+x^4})$
$f(x|u) = \mathbf{1}(x : \pi(x) > u) = \mathbf{1}(A_u)$. Therefore, $f(x|u) \sim U(A_u)$.
In our case, $f(x|u) \sim U(-(\frac{1}{u} - 1)^{1/4}, (\frac{1}{u} - 1)^{1/4})$

the signs here should be proportional to instead of distributed as

Here we are sampling from $f(x)$ using the *data augmentation via slice sampling* algorithm, as described in chapter 2 of "Liang, F., Liu, C., & Carroll, R. (2011). *Advanced Markov chain Monte Carlo methods: learning from past samples* (Vol. 714). John Wiley & Sons."
Graphically, the algorithm is shown below where you first take u between 0 and $f(x)$ then given that u you sample x from the horizontal line corresponding to $\{x : f(x) \geq u\}$.

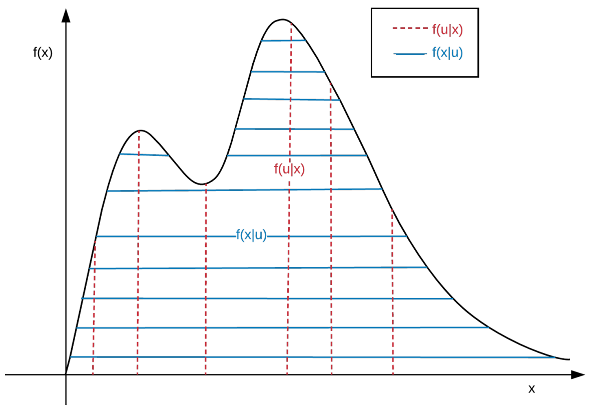

Figure 1: Graphical representation of slice sampling

Key advantage of slice sampling:

- the sampling scheme is adapted to the density under consideration, and it can be applied for any density as long as we can evaluate the density and find its inverse.

For more details on the graphical interpretation of univariate slice sampling, we can refer to "Neal, R. M. (2003). Slice sampling. *Annals of statistics*, 705-741".
Another highly cited slice sampling paper "Kalli, M., Griffin, J. E., & Walker, S. G. (2011). Slice sampling mixture models. *Statistics and computing*, 21(1), 93-105."

Why does this work?
To show that this works we need to show that the Markov chain described above with the corresponding transition density given by $P(x', u'|x, u) = f(x'|u)f(u'|x')$ leads to the stationary distribution $f(x', u')$ from $f(x, u)$ where x' and u' denote the samples obtained at the next step. In other words, we need to show that: $f(x', u') = \int_u \int_x P(x', u'|x, u)f(x, u)dxdu$
$\int_u \int_x P(x', u'|x, u)f(x, u)dxdu = \int_u \int_x f(x'|u)f(u'|x')f(x, u)dxdu$
$= f(u'|x') \int_u f(x'|u) \int_x f(x, u)dxdu$
$= f(u'|x') \int_u f(x'|u)f(u)du$
$= f(u'|x') \int_u f(x', u)du$
$= f(u'|x')f(x') = f(x', u')$
Therefore, we are sampling from the stationary distribution $f(x', u')$ using the Gibbs sampling approach.

Throwing away u and you get x?
To verify that this works we need to show that $f(x) = \int f(x, u)du$. In effect, we are saying that if you ignore the values of u and their probabilities for different values of (x, u) and aggregate all values of x that correspond to the same u, you will get x with probability $f(x)$. It is apparent that $\int f(x, u)du = \int \mathbf{1}(0 < u < \pi(x))du = \pi(x)$

Now let us implement and run the Gibbs sampler using the code below and for the functions $\pi(x) \propto \frac{1}{1+x^4}$ with $-\infty < x < \infty$. We note that the density $\pi(x)$ has the shape shown below, this makes slice sampling convenient since the horizontal slices are clearly defined.

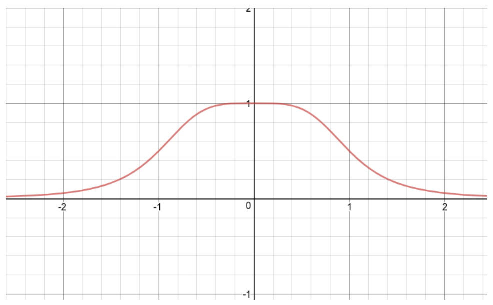

Figure 2: shape of the density to be sampled from (not normalized)

Algorithm:
(1) start with an initial point x_0
(2) sample u from $U(0, \frac{1}{1+x^4})$
(3) given u, sample x from $U(-(\frac{1}{u} - 1)^{1/4}, (\frac{1}{u} - 1)^{1/4})$
(4) return to step (2) with updated x and repeat.
(5) throw away u, your x's come from $\pi(x)$
(6) calculate $\hat{I} = \frac{\sum_1^N x_i^2}{N}$ as an approximation for $\tilde{I} = \int x^2\pi(x)$

code:

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

# define a sample from f(u|x)
def fux(x):
        u = nprand.uniform(low=0, high=1.0 / (1 + x**4))
        return u

# define a sample from f(x|u)
def fxu(u):
        fun = ((1.0 / u) - 1.0)**(1.0 / 4)
        x = nprand.uniform(low=-fun, high=fun)
        return x

# implement sampling
if __name__ == '__main__':
        numofiter = 1  # iterator
        total_iter = 50000  # total iterations
        x = 0  # starting value of x
        samples = []  # store samples
        numerator = []
        Ihat = []  # integral estimator
        while numofiter <= total_iter:
                u = fux(x)
                x = fxu(u)
                samples.append(x)
                numerator.append(x**2)
                Ihat.append(float(sum(numerator)) / len(numerator))
                numofiter += 1
        # plot a graph showing distribution
        plt.hist(samples, bins=50, facecolor='g', normed=True)
        plt.title("generated samples from f(x)")
        plt.xlabel("samples")
        plt.ylabel("normalized frequency")
        plt.tight_layout()
        plt.savefig('samples.png')
        plt.show()
        # print Ihat and plot a graph showing its convergence
        print("... integral estimator ...")
        print(Ihat[-1])
        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()
```

The following figure shows the obtained samples (normalized histogram plot) for $50,000$ iterations.

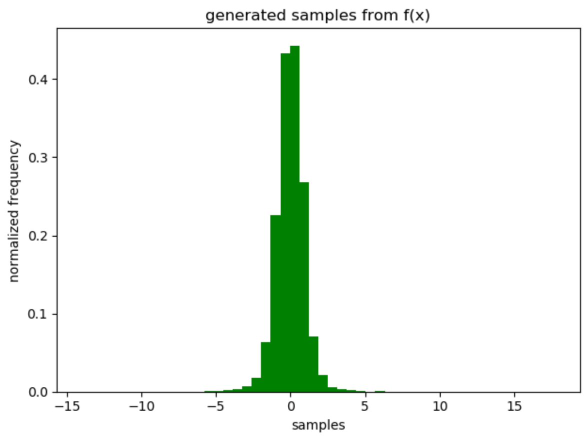

Figure 3: normalized density $\pi(x)$

For $50,000$ iterations, the variance $\tilde{I} = \int x^2\pi(x)$ was approximated using $\hat{I} = \frac{\sum_1^N x_i^2}{N}$ to be $\hat{I} = 0.942$. The following figure shows the convergence of the estimator. Note however that the variance of the estimator is relatively high even after $50,000$ iterations. In some cases, we get $\hat{I}$ closer to 1.

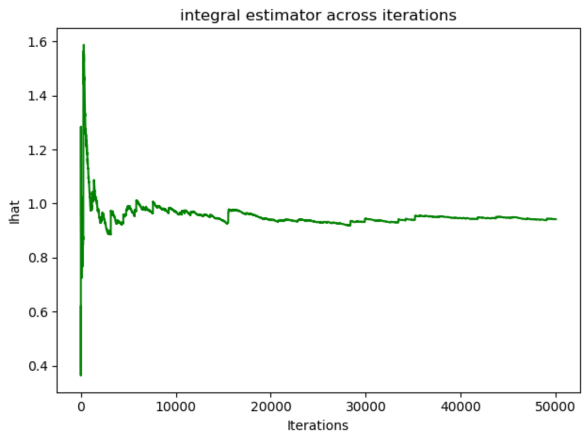

Figure 4: convergence of estimator $\hat{I}$

2.

consider the data model (likelihood) $g(x|\theta) = \text{Normal}(x|\mu, \sigma^2)$
where $\theta = (\mu, \lambda)$ and the precision $\lambda = \frac{1}{\sigma^2}$.
Priors:
$f(\mu) = \text{Normal}(\mu|0, \tau^2)$ and $f(\lambda) = \text{Gamma}(\lambda|a, b)$
find the condition densities (posteriors) for μ and λ given a data sample. Find, $f(\mu|\lambda, data)$ and $f(\lambda|\mu, data)$.

for a normal likelihood, the mean around which the data is assumed to be normally distributed has a normal conjugate prior.
From Wikipedia, and as shown in class, for the given normal prior, the posterior is given by $f(\mu|\lambda, data) = N(.|\mu', \sigma'^2)$, where μ' and σ'^2 are the posterior hyperparameters and they are defined as follows:
$\mu' = \frac{1}{\frac{1}{\tau^2}+\frac{n}{\sigma^2}}(\frac{\sum_1^n x_i}{\sigma^2})$
$\sigma'^2 = (\frac{1}{\tau^2} + \frac{n}{\sigma^2})^{-1}$
where n is the number of collected data points.

for a normal likelihood, the precision $\lambda = \frac{1}{\sigma^2}$ which represents the scatter of the data around the mean has a Gamma conjugate prior.
From Wikipedia, and as shown in class, for the given Gamma prior, the posterior is given by $f(\lambda|\mu, data) = Ga(.|a', b')$, where a' and b' are the posterior hyperparameters and they are defined as follows:
$a' = a + \frac{n}{2}$
$b' = b + \frac{\sum_1^n (x_i-\mu)^2}{2}$

A Gibbs sampler for sampling from $f(\mu, \lambda|data)$.
Following the same argument used in problem 1, we can show that the transition density defined by $P(\mu_{n+1}, \lambda_{n+1}|\mu_n, \lambda_n, data) = f(\mu_{n+1}|\lambda_n, data)f(\lambda_{n+1}|\mu_{n+1}, data)$ leads to the samples from the stationary distribution $f(\mu_{n+1}, \lambda_{n+1}|data)$ from $f(\mu_n, \lambda_n|data)$ where μ_{n+1} and λ_{n+1} are the samples at the next step.

To set up the priors, let the prior parameters be $\tau = a = b = 1.5$

Algorithm:
(1) start with λ_0
(2) sample μ_{n+1} from $f(\mu|\lambda, data) = N(.|\mu', \sigma'^2) = N(.|\frac{1}{\frac{1}{\tau^2}+\frac{n}{\sigma^2}}(\frac{\sum_1^n x_i}{\sigma^2}), (\frac{1}{\tau^2} + \frac{n}{\sigma^2})^{-1})$, using $\sigma^2 = \frac{1}{\lambda_n}$
(3) sample λ_{n+1} from $f(\lambda|\mu, data) = Ga(.|a', b') = Ga(.|a + \frac{n}{2}, b + \frac{\sum_1^n (x_i-\mu_{n+1})^2}{2})$
(4) repeat from step (2), you may throw away first few samples to reduce impact of priors

This algorithm is implemented for $n = 51$, $\sum_{i=1}^n x_i^2 = 39.6$ and $\sum_1^n x_i = 10.2$.

code:

```
"""
This code implements problem 2 on the
midterm for Monte Carlo methods in
statistics SDS386D
We implement Bayesian posterior sampling
for a normal likelihood using conjugate
priors and observed data

@cnyahia
"""

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


# define a sample from f(mu|lamda, data)
def sample_mu(prev_lambda, sum_nums, n, tau_squared):
        """
        sampling of mu
        :param prev_lambda: lambda at the previous iteration
        :param sum_nums: sum of the data
        :param n: total data points
        :param tau_squared: prior parameter on mew
        :return: mu
        """
        new_mean = 1.0 / ((1.0 / tau_squared) + float(n) * prev_lambda) *
        float(sum_nums) * prev_lambda
        new_var = 1.0 / ((1.0 / tau_squared) + float(n) * prev_lambda)
        new_std = np.sqrt(new_var)
        mu = nprand.normal(loc=new_mean, scale=new_std)
        return mu


# define a sample from f(lambda|mu, data)
def sample_lambda(mew, sum_nums, sum_squares, n, a, b):
        """
        sampling of lambda
        :param mew: value of mew conditioned upon
        :param sum_nums: sum(data)
        :param sum_squares: sum(data**2)
        :param n: total data points
        :param a: prior parameter
        :param b: prior parameter
        :return: new_lambda
        """
        new_a = a + float(n) / 2
        new_b = b + (1.0 / 2) * (sum_squares - 2 * mew * sum_nums + n * mew**2)
        scale = 1.0 / new_b
        new_lambda = nprand.gamma(new_a, scale=scale)
        return new_lambda

# implement sampling
if __name__ == '__main__':
        numofiter = 1  # iterator
        total_iter = 80000  # total iterations
        lam = 3  # initial value for lambda
        tau = 1.5
        tau_squared = tau**2  # mean prior
        a = 1.5  # precision prior
        b = 1.5  # precision prior
        n = 51  # sample total
        sum_squared = 39.6
        sum_nums = 10.2
        samples_mu = []  # store samples mu
        samples_lambda = []  # store samples lambda
        while numofiter <= total_iter:
                mu = sample_mu(lam, sum_nums, n, tau_squared)
                lam = sample_lambda(mu, sum_nums, sum_squared, n, a, b)
                samples_mu.append(mu)
                samples_lambda.append(lam)
                numofiter += 1

        print("... plotting samples of mu ...")
        # plot a graph showing distribution
        plt.hist(samples_mu, bins=50, facecolor='g', normed=True)
        plt.title("generated samples for mu")
        plt.xlabel("samples")
        plt.ylabel("normalized frequency")
        plt.tight_layout()
        plt.savefig('samples_mu.png')
        plt.show()

        print("... plotting samples of lambda ...")
        # plot a graph showing distribution
        plt.hist(samples_lambda, bins=50, facecolor='g', normed=True)
        plt.title("generated samples for lambda")
        plt.xlabel("samples")
        plt.ylabel("normalized frequency")
        plt.tight_layout()
        plt.savefig('samples_lambda.png')
        plt.show()
```

The sampled values of μ and λ are plotted using a normalized histogram as shown below.

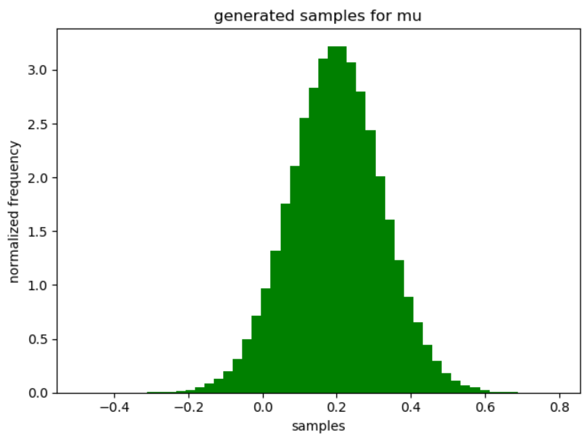

Figure 5: normalized samples of μ

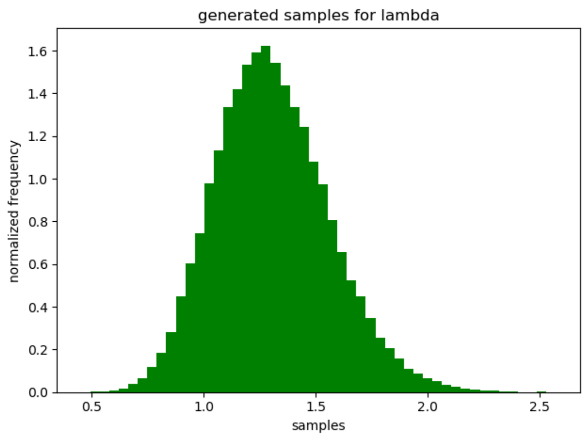

Figure 6: normalized samples of λ

The histograms clearly show that the estimates reflect the sample mean and variance. The sample mean $\bar{x} = \frac{\sum_1^n x_i}{n} = \frac{10.2}{51} = 0.2$. This matches the posterior for μ which is normally distributed around 0.2

The sample variance $s^2 = \frac{\sum_1^n x_i^2 - \frac{(\sum_1^n x_i)^2}{n}}{n-1} = \frac{39.6 - \frac{(10.2)^2}{51}}{50} = 0.7512$. This implies that the sample precision $\hat{\lambda} = \frac{1}{s^2} = \frac{1}{0.7512} = 1.33$. This matches the posterior for λ which is distributed around 1.33

The differences in densities between priors and posteriors (after adjusting based on collected data) are shown in the figures below:

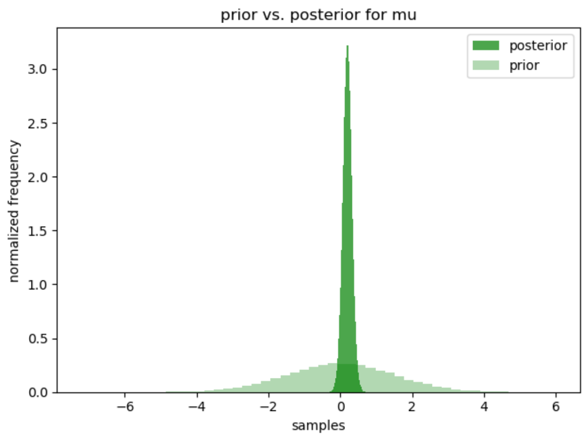

Figure 7: difference in prior and posterior densities for μ

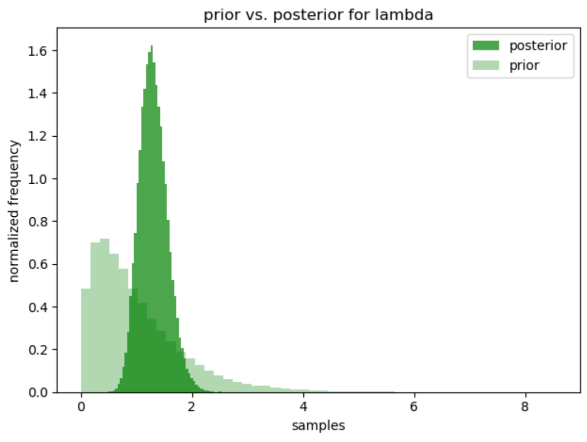

Figure 8: difference in prior and posterior densities for λ

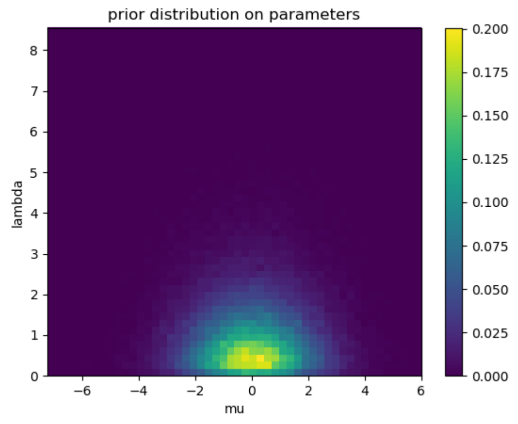

Figure 9: 2-dimensional prior density (μ, λ)

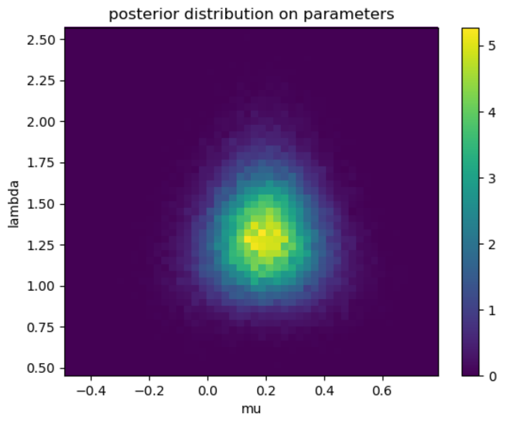

Figure 10: 2-dimensional posterior density (μ, λ)

From the figures above, it is clear how the density $f(\mu, \lambda|data)$ is shifted to be centered around $(0.2, 1.33)$ compared to the prior density $f(\mu, \lambda)$ which is centered around the origin. This indicates the shift to match the sample mean and sample variance.

3.

Suppose it is required to sample from the power law distribution. Sample X where $P(X = k) \propto k^{-\alpha}, \quad k \in \{1, 2, 3, ...\}$ for $\alpha > 1$. It is decided to do this using a Metropolis-Hastings algorithm with proposal

$$q(x'|x) = \begin{cases} x' = x + 1 & \text{probability } 1/2 \\ x = x - 1 & \text{probability } 1/2 \end{cases}$$

for $x > 1$ and $q(2|1) = 1$.
Effectively, we have a random walk proposal that does not go below 1.

The Metropolis-Hastings algorithm was first introduced by Metropolis et al. in 1953 and generalized by Hastings in 1970. This algorithm works by using a finite irreducible Markov chain to sample from a distribution $\pi(x)$. The desired density $\pi(x)$ should be the stationary distribution associated with the Markov chain transition density P such that $\pi = \pi P$.

Following the description in "Hastings, W. K. (1970). Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1), 97-109." the transition density is given by:
$p_{n,n+1} = p(x_{n+1}|x_n) = \frac{\Gamma(x_n)\alpha(x_{n+1},x_n)q(x_{n+1}|x_n)}{\Gamma(x_n)} + (1 - \Gamma(x_n))\mathbf{1}(x_{n+1} = x_n)$
where,
$\Gamma(x_n) = \int \alpha(x_{n+1}, x_n)q(x_{n+1}|x_n)d(x_{n+1})$
$\alpha(x_{n+1}, x_n)$ is a symmetric function of q that is bounded by 1

This transition density indicates that if $x_{n+1} = x_n$, then the probability of transitioning to the same point is $p_{ii} = p(x_n|x_n) = 1 - \int_{n \neq n+1} \alpha(x_{n+1}, x_n)q(x_{n+1}|x_n)$. The probability that you transfer to a different point $x_{n+1} \neq x_n$ is $p_{ij} = p(x_{n+1}|x_n) = \alpha(x_{n+1}, x_n)q(x_{n+1}|x_n)$

We showed on homework 2 that this Metropolis-Hastings transition matrix satisfies the reversibility condition $\pi_i p_{ij} = \pi_j p_{ji}$. For an irreducible and aperiodic chain, the stationary distribution exists and is unique. The reversibility property ensures that we converge to the desired density π as the stationary distribution for such a chain (the eigenvalues of an irreducible and aperiodic chain that satisfies the reversibility property are greater than -1 and less than or equal to 1, we can use this to define a bound on the deviation from the stationary distribution that converges to zero).

A good choice for $\alpha(x_{n+1}|x_n)$ (in terms of moving rate) is given by:
$\alpha(x_{n+1}, x_n) = \min\{1, \frac{\pi(x_{n+1})}{\pi(x_n)} \frac{q(x_n|x_{n+1})}{q(x_{n+1}|x_n)}\}$

The Metropolis-Hastings algorithm proceeds as follows:
(1) sample x^*_{n+1} from $q(x'|x)$
(2) sample u from $U(0, 1)$
(3) if $u < \alpha(x^*_{n+1}, x_n)$ then $x_{n+1} = x^*_{n+1}$. Otherwise, $x_{n+1} = x_n$
This works since $P(u < \alpha(x^*_{n+1}, x_n)) = \Gamma(x_n)$ for $u \sim U(0, 1)$ and x sampled from $q(x^*_{n+1}, x_n)$, so rejection sampling for $\tilde{q} \propto \alpha(x^*_{n+1}, x_n)q(x^*_{n+1}, x_n)$ determines how we should sample from the transition density $p(x_{n+1}|x_n)$.

To implement the algorithm we need to specify $\alpha(x_{n+1}|x_n)$
$\alpha(x_{n+1}|x_n) = \min\{1, \frac{x_{n+1}^{-\alpha}}{x_n^{-\alpha}} \frac{q(x_n|x_{n+1})}{q(x_{n+1}|x_n)}\}$
if x_{n+1} and x_n are both not equal to 1, then $q(x_{n+1}|x_n) = q(x_n|x_{n+1}) = 1/2$.
$\alpha(x_{n+1}|x_n) = \min\{1, \frac{x_{n+1}^{-\alpha}}{x_n^{-\alpha}}\}$
Special cases:
$\star$(1) $x_{n+1} = 2$, $x_n = 1$, then $q(x_{n+1}|x_n) = 1$ and $q(x_n|x_{n+1}) = 1/2$
$\alpha(x_{n+1}|x_n) = \min\{1, \frac{x_{n+1}^{-\alpha}}{x_n^{-\alpha}}(1/2)\}$
$\star$(2) $x_{n+1} = 1$, $x_n = 2$, then $q(x_{n+1}|x_n) = 1/2$ and $q(x_n|x_{n+1}) = 1$
$\alpha(x_{n+1}|x_n) = \min\{1, \frac{x_{n+1}^{-\alpha}}{x_n^{-\alpha}}(2)\}$

The following code implements the algorithm, the code also verifies the sample frequencies against evaluating the function explicitly using k^{-3} for $k = \{1, 2, 3..., 30\}$:

code:

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

# define alpha
def alpha(xp, xm, power):
        if (xp != 1) and (xm != 1):
                ratio = float(xp ** -power) / (xm ** -power)
                alfa = min(1, ratio)
        elif (xp == 2) and (xm == 1):
                ratio = (1.0/2) * (float(xp ** -power) / (xm ** -power))
                alfa = min(1, ratio)
        elif (xp == 1) and (xm == 2):
                ratio = 2.0 * (float(xp ** -power) / (xm ** -power))
                alfa = min(1, ratio)
        else:
                raise Exception('... error computing alpha ...')
        return alfa

# implement the sampling from q
def sample_q(xm):
        die = nprand.uniform(low=0, high=1)
        if xm == 1:
                sample = 2
        else:
                if die <= 0.5:
                        sample = xm + 1
                else:
                        sample = xm - 1
        return sample

# implement the Metropolis-Hastings algorithm
if __name__ == '__main__':
        numofiter = 1  # iterator
```

```
total_iter = 80000  # total iterations
power = 3  # power alpha for the power law
x_n = 3  # initial value
samples = []
while numofiter <= total_iter:
        samples.append(x_n)
        x_star = sample_q(x_n)  # sample x_star from x_{n}
        u = nprand.uniform(low=0, high=1)  # sample from uniform
        if u < alpha(x_star, x_n, power):
                x_n = x_star  # set the value at the next iteration
        # else, do nothing, x_n at next iteration is x_n
        numofiter += 1
print("... plotting samples of power law ...")
# plot a graph showing distribution
plt.hist(samples, bins=50, facecolor='g', normed=True)
plt.title("generated samples for power law")
plt.xlabel("samples")
plt.ylabel("normalized frequency")
plt.tight_layout()
plt.savefig('samples_plaw.png')
plt.show()
# verify
k = 1
k_total = 30
plaw_fun = []
k_vals = []
while k <= k_total:
        k_vals.append(k)
        plaw_fun.append(k**(-power))
        k += 1
print("... power law evaluated using function ...")
plt.bar(k_vals, plaw_fun)
plt.title("power law evaluated using function")
plt.xlabel("k")
plt.ylabel("k^{-3}")
plt.tight_layout()
plt.savefig('plaw_fun.png')
plt.show()
```

Histogram showing the samples obtained using the M-H algorithm detailed above, and their corresponding frequency:

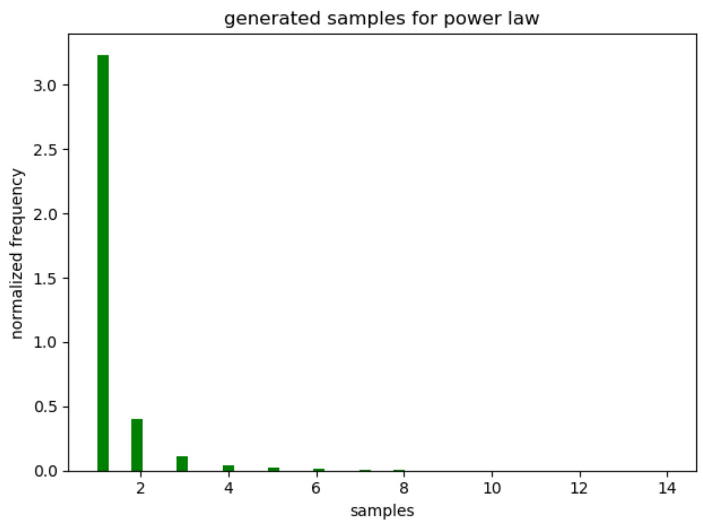

Figure 11: normalized frequency of samples obtained using MH algorithm

Bar plot showing the value of the function $f(k) = k^{-3}$ evaluated at $k = \{1, 2, 3, ..., 30\}$

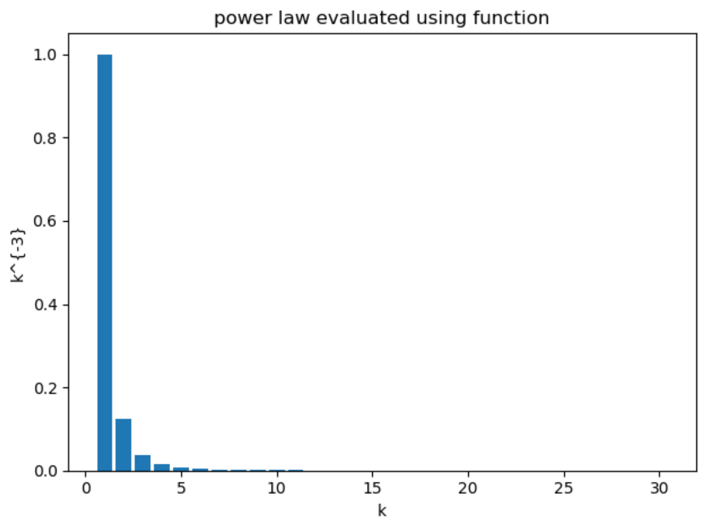

Figure 12: Bar plot for $f(k) = k^{-3}$

It is clear that the samples match the true density function up to a normalization constant.

4.

consider the joint density $\pi(x, y)$ and a Gibbs sampling framework to sample a Markov chain with stationary density π. The conditionals are $\pi(x|y)$ and $\pi(y|x)$ and while it is easy to sample $\pi(x|y)$, it is not possible to sample $\pi(y|x)$ directly. Describe a way how to proceed using a Metropolis Hastings step and describe all the details clearly and also why it works.

Procedure and why it works: (in the discussion below, f is used instead of π for density)
The original Gibbs sampling transition is given by $P(x_{n+1}, y_{n+1}|x_n, y_n) = f(x_{n+1}|y_{n+1})f(y_{n+1}|x_n)$ such that we should sample $f(y_{n+1}|x_n)$ then $f(x_{n+1}|y_{n+1})$ repeatedly to get samples from the stationary density $f(x_{n+1}, y_{n+1})$. To verify that this sampling procedure results in sampling from the stationary density, we can see that:
$\int_x \int_y P(x_{n+1}, y_{n+1}|x_n, y_n)f(x_n, y_n)dy_n dx_n = \int_x \int_y f(x_{n+1}|y_{n+1})f(y_{n+1}|x_n)f(x_n, y_n)dy_n dx_n$
$= f(x_{n+1}|y_{n+1}) \int_x \int_y f(y_{n+1}|x_n)f(x_n, y_n)dy_n dx_n = f(x_{n+1}|y_{n+1}) \int_x f(y_{n+1}|x_n) \int_y f(x_n, y_n)dy_n dx_n$
$= f(x_{n+1}|y_{n+1}) \int_x f(y_{n+1}|x_n)f(x_n)dx_n = f(x_{n+1}|y_{n+1}) \int_x f(y_{n+1}, x_n)dx_n = f(x_{n+1}|y_{n+1})f(y_{n+1})$
$= f(x_{n+1}, y_{n+1})$
In case $f(y_{n+1}|x_n)$ is not easy to sample directly, we can use a Metropolis-Hastings step to sample $f(y_{n+1}|x_n)$ such that a sample is taken from a MH chain with transition density $P_M(y_{n+1}|x_n, y_n)$. This MH chain will have a proposal density $q(y_{n+1}|x_n, y_n)$ and
$\alpha(y_{n+1}, y_n) = \min\{1, \frac{f(y_{n+1}|x_n)}{f(y_n|x_n)} \frac{q(y_n|y_{n+1}, x_n)}{q(y_{n+1}|y_n, x_n)}\}$
The new Gibbs sampling transition density is $\tilde{P}(x_{n+1}, y_{n+1}|x_n, y_n) = f(x_{n+1}|y_{n+1})P_M(y_{n+1}|x_n, y_n)$
We need to show that this transition density satisfies the stationary condition as well.
We know that a MH chain is reversible, and the following condition is satisfied:
$P_M(y_{n+1}|x_n, y_n)f(y_n|x_n) = f(y_{n+1}|x_n)P_M(y_n|x_n, y_{n+1})$

showing stationarity:
$\int_x \int_y \tilde{P}(x_{n+1}, y_{n+1}|x_n, y_n)f(x_n, y_n)dy_n dx_n = \int_x \int_y f(x_{n+1}|y_{n+1})P_M(y_{n+1}|x_n, y_n)f(x_n, y_n)dy_n dx_n$
$= f(x_{n+1}|y_{n+1}) \int_x \int_y P_M(y_{n+1}|x_n, y_n)f(y_n|x_n)f(x_n)dy_n dx_n$
$= f(x_{n+1}|y_{n+1}) \int_x \int_y f(y_{n+1}|x_n)P_M(y_n|x_n, y_{n+1})f(x_n)dy_n dx_n$ (by MH reversibility)
$= f(x_{n+1}|y_{n+1}) \int_x f(y_{n+1}|x_n)f(x_n) \int_y P_M(y_n|x_n, y_{n+1})dy_n dx_n$
$= f(x_{n+1}|y_{n+1}) \int_x f(y_{n+1}|x_n)f(x_n)dx_n$ ($P_M(y_n|x_n, y_{n+1})$ is a transition matrix, in the discrete case, rows in a right stochastic matrix sum up to 1, integrating $P_M(y_n|x_n, y_{n+1})$ over y is equivalent to summing up rows in discrete case)
$= f(x_{n+1}|y_{n+1}) \int_x f(y_{n+1}, x_n)dx_n = f(x_{n+1}|y_{n+1})f(y_{n+1})$
$= f(x_{n+1}, y_{n+1})$ (stationarity of the Gibbs transition is satisfied)
Therefore, by sampling $f(y_{n+1}|x_n)$ using a MH step given by the transition density $P_M(y_{n+1}|x_n, y_n)$ in the Gibbs sampling framework, we end up sampling from the desired stationary distribution. Remaining is to identify a proposal $q(y_{n+1}|x_n, y_n)$ that is easy to sample and appropriate for the conditional density $f(y|x)$.

Implementation for $\pi(x, y) \propto e^{-4x}y^3e^{-3ye^x}, \quad x, y > 0$
For using the Gibbs sampling framework, we need to sample $\pi(x_{n+1}|y_{n+1})$ and $\pi(y_{n+1}|x_n)$.
$\pi(x_{n+1}|y_{n+1}) \propto e^{-4x}e^{-3ye^x}$
We can sample this density using rejection sampling. Take $\pi(x_{n+1}|y_{n+1}) \propto e^{-4x}e^{-3ye^x} = h(x)g(x)$ where $h(x) = e^{-4x}$ is a density that is easy to sample from since
$h(x) \propto$ Exponential$(\lambda = 4)$, and $g(x) = e^{-3ye^x}$ is a bounded function on $x > 0$ (refer to figure).
$g(x)$ also attains its maximum at $x = 0$, as shown in the figure below.

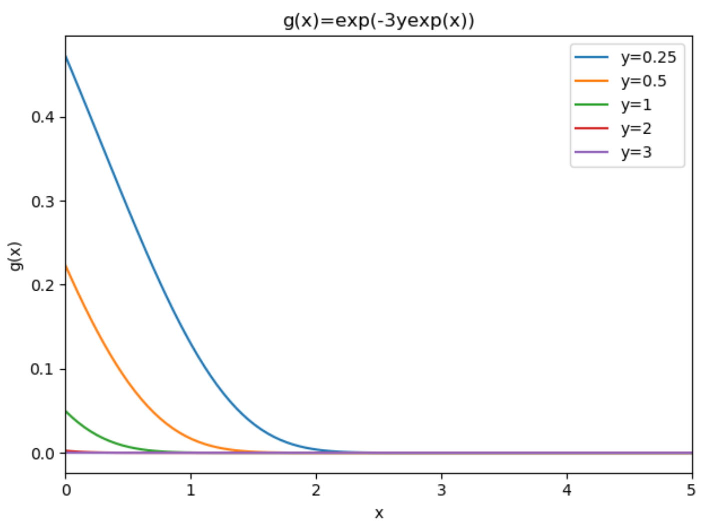

Figure 13: $g(x) = e^{-3ye^x}$ for different values of y

Figure 13 shows that the function $g(x)$ is bounded for $y > 0$, and it attains its maximum at $x = 0$.
Therefore, $M = e^{-3y}$ is the upper bound on $g(x)$.
To implement rejection sampling for $\pi(x|y)$:
(1) take x from $h(x) =$Exponential$(\lambda = 4)$
(2) take u from $U(0, 1)$
(3) evaluate $g(x)/M = \frac{e^{-3ye^x}}{e^{-3y}} = e^{-3ye^x+3y} = e^{3y(1-e^x)}$
(4) if $u < g(x)/M$, accept x as coming from $\pi(x|y) = h(x)g(x) = e^{-4x}e^{-3ye^x}$
We note that rejection sampling works since the above framework samples from $f(x, u) = M\mathbf{1}(u < \frac{g(x)}{M})\mathbf{1}(u < 1)h(x)$ and $\pi(x|y) = \int_u f(x, u)du$

In our iterative Gibbs sampling framework, we only need to get one accepted value of $\pi(x_n|y_n)$ before returning to sample $\pi(y_{n+1}|x_n)$ using the obtained sample.
$\pi(y_{n+1}|x_n) \propto y^3 e^{-3ye^x}$
In this case $\pi(y_{n+1}|x_n) \propto Gamma(\alpha = 4, \beta = 3e^x)$. But, since the question specifies that it is not possible to sample $\pi(y|x)$ directly, I will assume that it is not and proceed with a Metropolis-Hastings step.
Generally, the conditional density is not easy to sample using rejection sampling or a similar framework since it is difficult to find a good proposal and a corresponding bounded function. For RS, we need to show that the function is bounded for every value of x. Equivalently, we need to check that $g(y)$ is log-concave for every value of x. Then, if we are able to show that the function is bounded, to have a good proposal, we will have to adapt the proposal to the specific value of x and the corresponding shape of $g(y)$. It is not always easy to find a non-negative function $h(y)$ over the domain that has a finite integral over that domain, and which leads to decomposition of $\pi(y) = h(y)g(y)$ with a bounded positive function $g(y)$.

In this case, we can sample $\pi(y_{n+1}|x_n)$ using a Metropolis-Hastings algorithm, and as shown before, this would lead to sampling from the desired stationary in a Gibbs sampling framework.

The Metropolis-Hastings transition density is given by

$P_M(y_{n+1}|x_n, y_n) = \frac{\Gamma(y_n)\alpha(y_{n+1},y_n)q(y_{n+1}|y_n,x_n)}{\Gamma(y_n)} + (1-\Gamma(y_n))\mathbf{1}(y_{n+1} = y_n)$
where,
$\Gamma(y_n) = \int \alpha(y_{n+1}, y_n)q(y_{n+1}|y_n, x_n)d(y_{n+1})$
$\alpha(y_{n+1}, y_n) = \min\{1, \frac{f(y_{n+1}|x_n)}{f(y_n|x_n)}\frac{q(y_n|y_{n+1},x_n)}{q(y_{n+1}|y_n,x_n)}\}$

The Metropolis-Hastings algorithm proceeds as follows:
(1) sample y^*_{n+1} from $q(y_{n+1}|y_n, x_n)$
(2) sample u from $U(0, 1)$
(3) if $u < \alpha(y^*_{n+1}, y_n)$ then $y_{n+1} = y^*_{n+1}$. Otherwise, $y_{n+1} = y_n$

To implement the algorithm we need to specify $\alpha(y_{n+1}|y_n)$ and $q(y_{n+1}|y_n, x_n)$.
Let us try a log-normal proposal that is independent of x_n, and with mean y_n.
$q(y_{n+1}|y_n, x_n) = q(y_{n+1}|y_n) = \frac{1}{y_{n+1}}e^{-\frac{1}{2\sigma^2}(log(y_{n+1})-log(y_n))^2}$
This implies that:
$\alpha(y_{n+1}, y_n) = \min\{1, \frac{y^3_{n+1}e^{-3y_{n+1}e^{x_n}}}{y^3_n e^{-3y_n e^{x_n}}}\frac{y_{n+1}}{y_n}\} = \min\{1, \frac{y^4_{n+1}e^{-3y_{n+1}e^{x_n}}}{y^4_n e^{-3y_n e^{x_n}}}\}$
∼∼
$\rightarrow$ Now that we have sampled $\pi(y_{n+1}|x_n)$ using 1 step of MH, go back and do RS using y_{n+1} to sample $\pi(x_{n+1}|y_{n+1})$.
Proceed between RS and MH repeatedly to sample the stationary :)
∼∼

Finally, to evaluate the integral $I = \int\int xy\pi(x, y)dxdy$, we can use an estimator $\hat{I} = \frac{\sum_{(x,y)} x_i(y_i)}{N}$ where N is the total number of iterations which is equal to the total number of obtained samples (x, y) from $\pi(x, y)$.

The following code implements the Gibbs sampling framework discussed above and evaluates the integral
code:

```
"""
This is the last problem of the Midterm
for Monte Carlo methods in statistics
SDS386D

The code implements Gibbs by using RS and
MH repeatedly

@cnyahia
"""

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


# sample from h(x) for rejection sampling
def RS_sample_hx():
        """
        This function is used to sample h(x)
        :return: exponential sample
        """
        beta = 1.0 / 4  # scale parameter
        expon = nprand.exponential(beta)
        return expon


# rejection sampling, evaluate ratio g(x)/M
def RS_evaluate_ratio(x, y):
        """
        for a given value of x
        evaluate g(x)/M
        :param x: current val of x
        :return: ratio
        """
        ratio = np.exp(3 * y * (1 - np.exp(x)))
        return ratio


# MH proposal
def MH_proposal(y, sigma=1):
        """
        sample the log-normal MH proposal
        the parameters are mean and sigma of the
        normally distributed log(Y), where
        Y is log-normally distributed
        :param y: log-normal mean
        :param sigma: log-normal variance
        :return: sample
        """
        mean_of_logx = np.log(y)
        sample = nprand.lognormal(mean_of_logx, sigma)
        return sample


# define alpha for MH
def MH_alpha(yp, ym, x):
        """
        define alpha for the MH sampling of
        pi(yp|ym, x)
        :param yp: y_plus
        :param ym: y_minus
        :param x: current x
        :return: alpha
        """
        ratio_numerator = (yp**4) * np.exp(-3 * yp * np.exp(x))
        ratio_denom = (ym**4) * np.exp(-3 * ym * np.exp(x))
        ratio = ratio_numerator / ratio_denom
        alpha = min(1, ratio)
        return alpha


# implement the sampling algorithm
if __name__ == '__main__':
        samples_y = []
        samples_x = []
        Ihat = []  # integral estimator
        numerator = []  # numerator of Ihat
        x = 1  # initial starting value
        y = 1  # initial starting value
        numofiter = 1
        total_iter = 80000  # total iterations
        while numofiter <= total_iter:
                # sample y from x using MH
                samples_y.append(y)
                y_star = MH_proposal(y)  # q in MH
                u_MH = nprand.uniform(low=0, high=1)
                if u_MH < MH_alpha(y_star, y, x):
                        y = y_star  # set the value at the next iteration
                # you have sampled y|x, now sample x|y using RS
                sampled_RS = False
                while not sampled_RS:
                        x = RS_sample_hx()  # sample hx in RS
                        u = nprand.uniform(low=0, high=1)
                        g_by_M = RS_evaluate_ratio(x, y)
                        if u < g_by_M:
                                samples_x.append(x)  # append x
                                sampled_RS = True  # end once sample is
                                # obtained

                numerator.append(x * y)
                Ihat.append(float(sum(numerator)) / len(numerator))
                numofiter += 1

        print("... plotting samples of x ...")
        plt.hist(samples_x, bins=50, facecolor='g', normed=True)
        plt.title("normalized samples of x")
        plt.xlabel("samples")
        plt.ylabel("normalized frequency")
        plt.tight_layout()
        plt.savefig('p4_x_samples.png')
        plt.show()

        print("... plotting samples of y ...")
        plt.hist(samples_y, bins=50, facecolor='g', normed=True)
        plt.title("normalized samples of y")
        plt.xlabel("samples")
        plt.ylabel("normalized frequency")
        plt.tight_layout()
        plt.savefig('p4_y_samples.png')
        plt.show()

        print("... integral estimator ...")
        print(Ihat[-1])
        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('p4_ihat.png')
        plt.show()

        print("... plotting pi(x,y) in two dimensions ...")
        fig, ax = plt.subplots()
        h = ax.hist2d(samples_x, samples_y, bins=50, normed=True)
        plt.title("distribution of pi(x, y)")
        # plt.tight_layout()
        plt.xlabel("x")
        plt.ylabel("y")
        plt.colorbar(h[3], ax=ax)
        plt.savefig('p4_pixy.png')
        plt.show()
```

The following figures shows the obtained distributions:

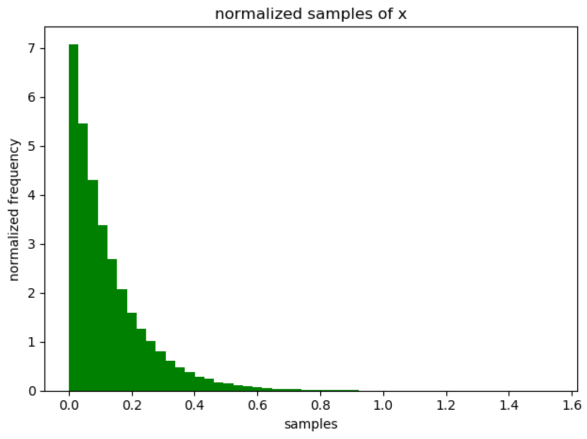

Figure 14: the density of the collected x samples

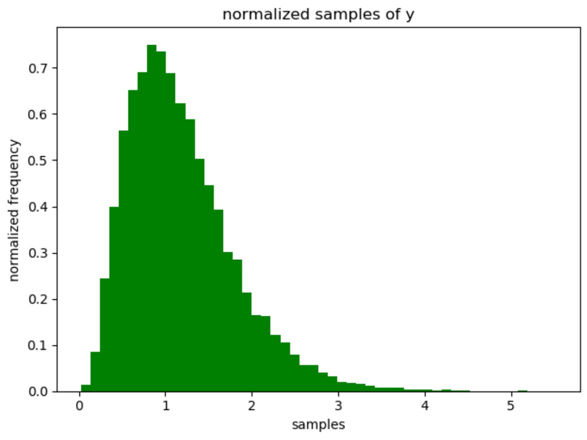

Figure 15: the density of the collected y samples

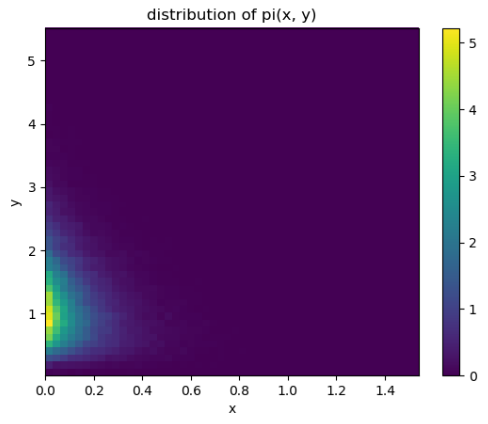

Figure 16: the joint density $\pi(x, y)$

The integral $I = \int \int xy\pi(x, y)dxdy$ estimator $\hat{I} = \frac{\sum_{(x,y)} x_i(y_i)}{N}$ converged to a value of 0.132 after $80,000$ iterations. The following figure shows the convergence of the integral estimator across iterations.

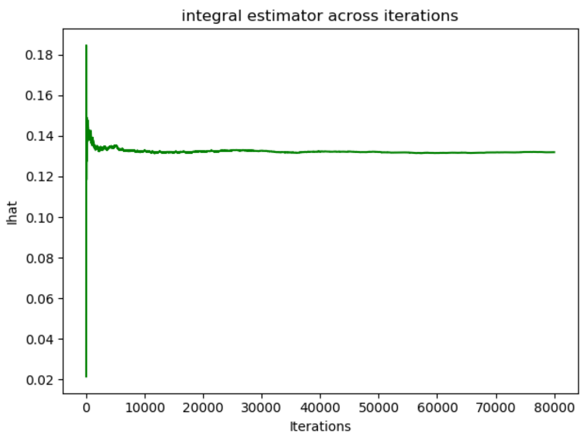

Figure 17: convergence of the integral estimator $\hat{I}$