In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
#from matplotlib import rc
```

In [2]:
```
%matplotlib inline
#from __future__ import division
#rc('font',**{'family':'sans-serif','sans-serif':['Helvetica']})
#rc('text', usetex=True)
```

# Mid-term exam

## 1.

Consider the joint density $f(x, u) = 1(0 < u < \pi(x))$, the conditional density should be:

i. given x, u should satisfy a uniform distribution $U(0, \pi(x))$

ii. given u, x should satisfy a uniform distribution on $A_u = \{x : \pi(x) > u\}$

Thus, the algorithm of the Gibbs sampler:

1. sample $x_0$ from a proposed initial density $q_0(x)$
2. sample u from $U(0, \pi(x))$
3. sample x from $U(A_u)$
4. iterate 2 and 3 until convergence is reached and enough samples are generated

Specifically, to sample $\pi(x) \propto 1/(1 + x^4)$

In [3]:
```
def pi(x):
    return 1/(1+x**4)
```

From the above algorithm and joint density, $u|_x \sim U(0, 1/(1 + x^4))$:

In [4]:
```
def draw_u(x):
    u = np.random.uniform()
    while u >= pi(x):
        u = np.random.uniform()
    return u
```

And $x|_u \; U(-\pi^{-1}(u), \pi^{-1}(u))$, where $\pi^{-1}(u)$ is defined by:

$$\pi^{-1}(u) = (\frac{1}{u} - 1)^{1/4}$$

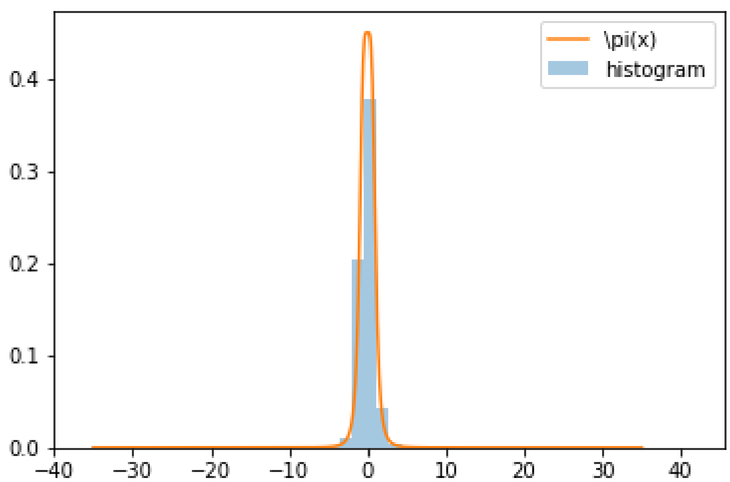
In [5]:
```
def draw_x(u):
    x = np.random.uniform(-pow(1/u-1, 1/4), pow(1/u-1, 1/4))
    return x
```

Use this Gibbs sampler to generate a million samples:

In [6]:
```
u = np.random.uniform()
N = 1000000
samples = np.zeros(N)
for i in range(N):
    x = draw_x(u)
    samples[i] = x
    u = draw_u(x)
```

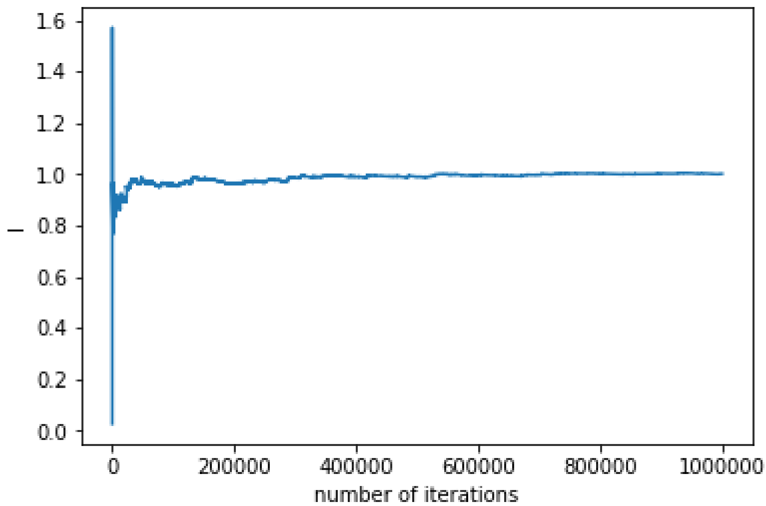
Plot the histogram of histogram of the samples, tt can be shown that this is consistent with the actual distribution:

In [7]:
```
sns.distplot(samples, kde=False, norm_hist=True, label="histogram")
x = np.linspace(-35, 35,1000)
plt.plot(x, 1/(1+x**4) * np.sqrt(2) / np.pi, label='\pi(x)')
plt.legend()
plt.show()
```




Compute the Monte Carlo integral for $I = \int x^2 \pi(x) dx$, plot the convergence curve

In [8]:
```
I = samples**2
for i in range(1, N):
    I[i] = (I[i-1]*i + samples[i]**2)/(i+1) #fast, probably unstable way
 to compute average vs iteration

plt.plot(range(N), I)
plt.xlabel("number of iterations")
plt.ylabel("I")
plt.show()
```




Use the first 200000 iterations as a burn-in simulation and compute the MC approximation $\hat{I}_N = \sum_i^n x_i y_i / n$ to estimate variance:

In [9]:
```
(samples**2)[200000:].mean()
```

Out[9]: 1.010457223164956

Note that I assume $\pi(x)$ is a probability function. Thus this MC integration is close to the actual integration which is:

$$I = \frac{\int x^2 \pi(x) dx}{\int \pi(x) dx} = 1$$

# 2.

From Bayes' theory:

$$f(\mu, \lambda \mid data) \propto \prod_{i=1}^{n} g(x_i \mid \mu, \lambda) \cdot f(\mu) \cdot f(\lambda)$$
$$= \lambda^{n/2} e^{\lambda \sum_{i=1}^{n} -(x_i - \mu)^2/2} \cdot \frac{1}{\tau} e^{-\mu^2/2\tau^2} \cdot \lambda^{a-1} e^{-\lambda b}$$

Since the size of the samples is rather small, I picked larg $\tau$ and small a and b so that the Markov chain is less sensitive to the coefficients:

In [10]:
```
tau = 10; a = 2; b = 5
n = 51
```

Rearange $f(\mu, \lambda \mid data)$ and get conditional densities for $\mu$ and $\lambda$:

$$p(\mu|\lambda, data) \propto e^{\lambda \sum_{i=1}^{n} -(x_i-\mu)^2/2} \cdot e^{-\mu^2/2\tau^2}$$
$$\propto e^{-\frac{1}{2}(\mu - \frac{\lambda n\bar{x}}{1/\tau^2+n\lambda})^2 \cdot (1/\tau^2+n\lambda)}$$

$\mu|_\lambda \sim N(\frac{\lambda n\bar{x}}{1/\tau^2+n\lambda}, \frac{1}{1/\tau^2+n\lambda})$

In [11]:
```
def draw_mu(lamda):
    return np.random.normal(lamda*10.2/(1/tau**2+n*lamda), np.sqrt(1/(n*
lamda+1/tau**2)))
```

$$p(\lambda|\mu, data) \propto \lambda^{n/2+a-1} e^{-\frac{1}{2}\lambda[b+\sum_i (x_i-\mu)^2]}$$

$\lambda|_\mu \sim Gamma(n/2 + a, \frac{b+\sum_i (x_i-\mu)^2}{2})$

In [12]:
```
def draw_lamda(mu):
    return np.random.gamma(shape = a+n/2, scale=2/(b+(39.6 -2*mu*10.2 +
n*mu**2) - 10.2**2/n))
```

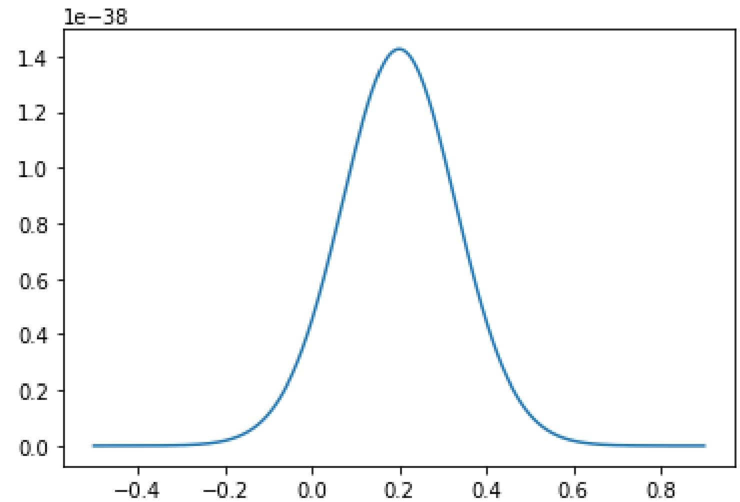
Similar to the previous question, the Gibbs sampler starts with an initial $\mu_0$, then sample $\lambda$ and $\mu$ iteratively based on the above two conditional densities in a round Robin fashion (100000 samples were generated):

In [13]:
```
mu = np.random.normal(0.2, 0.751)
N2 = 100000
samples2 = np.zeros((N2, 2))
for i in range(N2):
    lamda = draw_lamda(mu)
    mu = draw_mu(lamda)
    samples2[i][0] = mu
    samples2[i][1] = lamda
```

Consider the marginal densities for $\mu$:

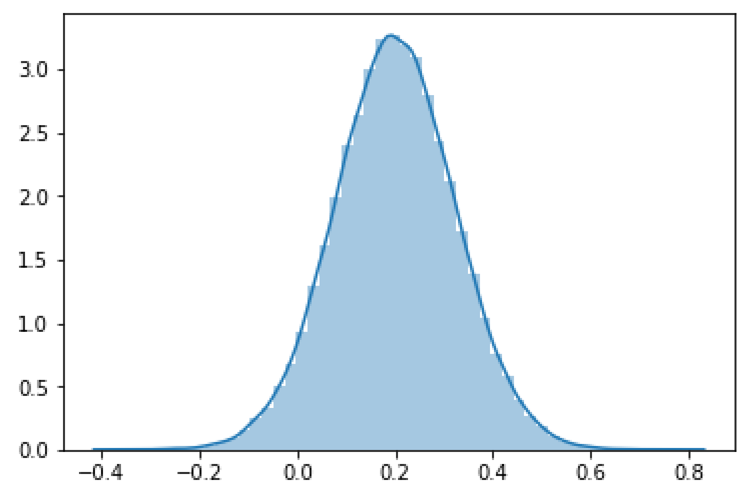
$$p(\mu|data) \propto \frac{f(\mu, \lambda \mid data)}{p(\lambda|\mu, data)}$$
$$\propto \exp(-\mu^2/2\tau^2) \cdot [b + \frac{\sum_i (x_i - \mu)^2}{2}]^{-(a+n/2)}$$

In [14]:
```
x = np.linspace(-0.5, 0.9, 1001)
y = np.exp(-x**2/2/tau**2)*(b + (39.6 - 2*10.2*x + n*x**2)/2)**(-a-n/2)
plt.plot(x,y, label = r"p(\mu)")
plt.show()
```




The sample distribution for $\mu$:

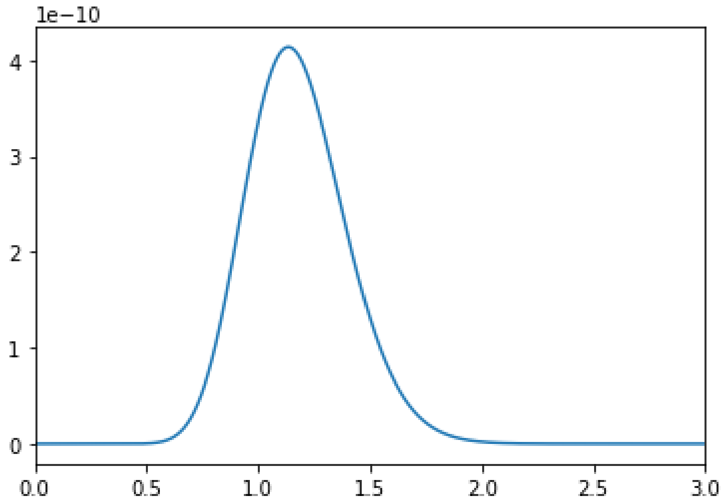
In [15]:
```
sns.distplot(samples2[:,0], norm_hist=True, label="histogram")
plt.show()
```




The marginal density of $\lambda$:

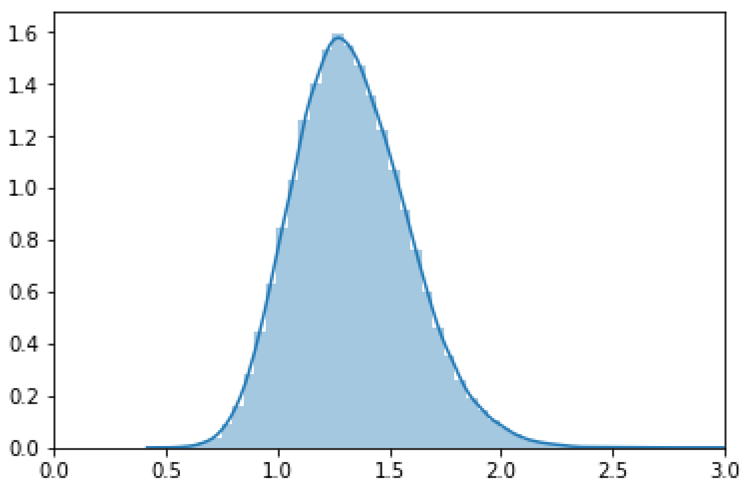
$$p(\lambda|data) \propto\propto \frac{f(\mu, \lambda \mid data)}{p(\mu \mid \lambda, data)}$$
$$\propto \lambda^{n/2+a-1} e^{-\lambda b} \sqrt{1/\tau^2 + n\lambda} e^{-\frac{1}{2}[\lambda \sum_{i=1}^{n} x_i^2 - \frac{(\sum_i x_i \lambda)^2}{n\lambda + 1/\tau^2}]}$$

In [16]:
```
x = np.linspace(0,3,1001)
y = x**(n/2+a-1)*np.exp(-x*b) * np.sqrt(1/tau**2+n*x) * np.exp((10.2*x)*
*2/(n*x+1/tau**2)/2 - x*39.6/2)
plt.plot(x, y)
plt.xlim(0,3)
plt.show()
```




The sample distribution of $\lambda$:

In [17]:
```
sns.distplot(samples2[:,1])
plt.xlim(0,3)
plt.show()
```



The sample distribution is consistent with the real marginal distribution.

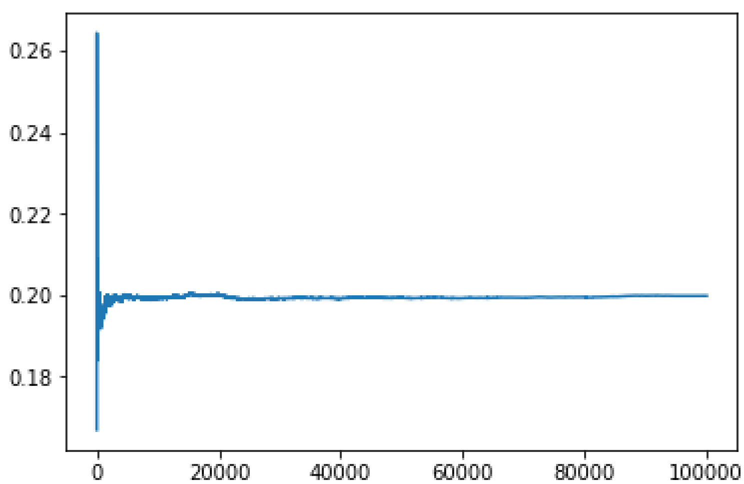
With this sampler, we can estimate the value of $\mu$ and $\lambda$ by taking the sample mean:

$$\bar{\mu} = \frac{1}{N}\sum_{i}^{N} \mu_i$$

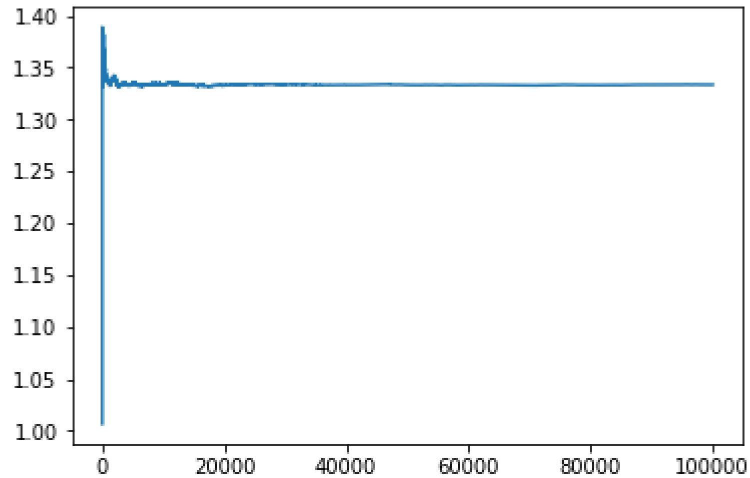
$$\bar{\lambda} = \frac{1}{N}\sum_{i}^{N} \lambda_i$$

Now plot the convergence curve of $\mu$ and $\lambda$ respectively:

In [18]:
```
mu = samples2[:,0].copy()
lamda = samples2[:,1].copy()

for i in range(1, N2):
    mu[i] = (mu[i-1]*i + samples2[i, 0])/(i+1)
    lamda[i] = (lamda[i-1]*i + samples2[i,1])/(i+1)

plt.plot(range(N2), mu)
plt.show()
```



In [19]:
```
plt.plot(range(N2), lamda)
plt.show()
```




In [20]:
```
samples2[-20000:].mean(axis=0)
```

Out[20]:
```
array([ 0.20111011,  1.33477983])
```

$\bar{\mu}$ and $\bar{\lambda}$ from sampling the posterior with the Gibbs sampler are 0.2012 and 1.329 respectively. The mean of the data is $10.2/51 = 0.2$ and inverse variance of the data is $1/\frac{1}{n-1}\sum_i (x_i - \bar{x})^2 = 1/[(39.6 - 2 \cdot 0.2 \cdot 10.2 + n \cdot 0.2^2)/(n-1)] = 1.3312$. Thus the estimates agree with the sample mean and variance.

# 3.

In [21]:
```
alpha3 = 3
```

Given the power law distribution: $P(X = k) \propto k^{-\alpha},\ k \in \{1, 2, 3, \ldots\}, \alpha > 1$

In [22]:
```
def pi(k):
    return 1/np.power(k, alpha3)
```

Proposal for Metropolis-Hasting algorithm (1D random walk):

In [23]:
```
def q(x2, x1):
    assert abs(x2-x1) == 1 #check is the two states are next to each oth
er
    if x2 == 2 and x1 == 1:
        return 1
    return 1/2
```

The $\alpha(x_2, x_1)$ for Metropolis-Hasting:

In [24]:
```
def alpha(x2, x1):

    return min(1, pi(x2) / pi(x1) * q(x1, x2) / q(x2, x1))
```

Algorithm for Metropolis-Hasting:

1. Start from a $x_1$
2. sample $x_2^*$ from proposal probability
3. sample u from U(0,1)
4. accept $x_2^*$ if $u < \alpha(x_2^*, x_1)$, else stay at $x_1$
5. iterate through 2-4 until enough samples are acquired.

For this problem, we start the Markov chain at 1 and iterate 10000 times.

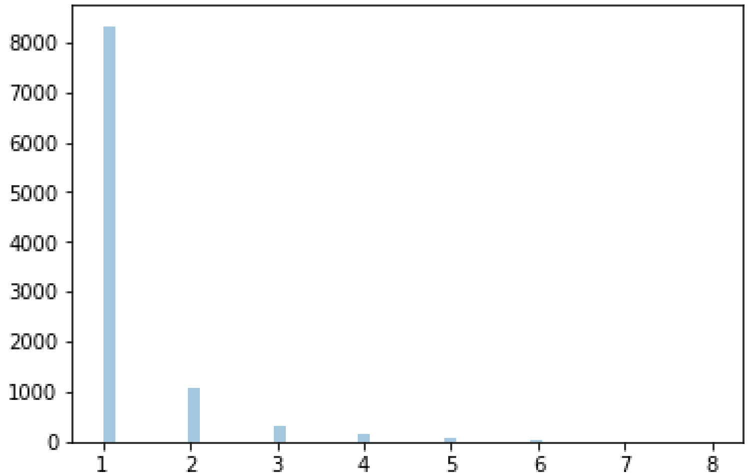
In [25]:
```
x1 = 1
N3 = 10000
samples3 = np.zeros(N3)
for i in range(N3):
    x2_tmp = 2 if x1 == 1 else np.random.choice([x1+1, x1-1])
    u = np.random.uniform()
    x2 = x2_tmp if u < alpha(x2_tmp, x1) else x1
    x1 = x2
    samples3[i] = x2
```

Note that the initial position matters in this problem. If the initial position $x_0$ is very large, $q(x_0)$ will be very close to $q(x_0 - 1)$ and $q_(x_0 + 1)$, thus $\alpha(x_0 + 1, x_0)$ is also close to 1 ($\alpha(x_0 - 1, x_0)$ = 1). This means that the Markov chain could walk to the right easily and takes very long iterations to converge to the true density.

Sample distribution:

In [26]:
```
sns.distplot(samples3, kde = False)
```

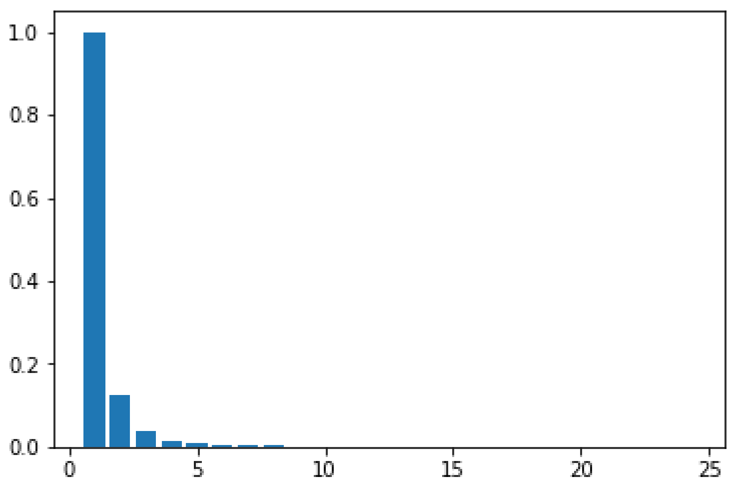
Out[26]: <matplotlib.axes._subplots.AxesSubplot at 0x1a1d7d59b0>




True density function:

In [27]:
```
X3 = range(1, 25)
plt.bar(X3, 1/np.power(X3, alpha3))
```

Out[27]: <Container object of 24 artists>




# 4.

Consider the joint density $\pi(x, y)$ and a Gibbs sampling framework is usedto sample a Markov chain with stationary density $\pi$. The conditionals are $\pi(x|y)$ and $\pi(y|x)$ and while it is easy to sample $\pi(x|y)$ it is not possible tosample $\pi(y|x)$ directly.

How to proceed using a Metropolis-Hasting step:

1. start from an initial $x_0, y_0$
2. sample $x_1$ from $\pi(x|y)$ directly
3. sample $y_1$ from $\pi(y|x)$ using one step of Metropolis-Hasting:

   i. sample $y^*$ from proposal density $q(y^*|y)$.

   ii. sample u from U(0,1)

   iii. accept $y^*$ if $u < \alpha(y^*, y_0)$, else stay at $y_0$. In this case:
   $$\alpha(y_2, y_1) = \frac{\pi(y_2|x_1)}{\pi(y_1|x_1)} \frac{q(y_1|y_2)}{q(y_2|y_1)}$$
4. repeat 2 and 3 until enough samples are acquired.

Why does this work:

The trainsition density: $P(x', y'|x, y) = \pi(x'|y') \cdot P_M(y'|x, y)$

Then, consider $\pi(x', y')$:

$$\begin{aligned} \int\int P(x', y'|x, y)\pi(x, y)\, dx\, dy &= \int\int \pi(x'|y') \cdot P_M(y'|x, y)\pi(x, y)\, dx\, dy \\ &= \pi(x'|y') \int\int P_M(y'|x, y)\pi(x, y)\, dx\, dy \end{aligned}$$

Note that $P_M(y'|x, y)\pi(y|x) = P_M(y|x, y')\pi(y'|x)$

$$\begin{aligned} &= \pi(x'|y') \int\int P_M(y|x, y')\pi(y'|x)/\pi(y|x) \cdot \pi(y, x)\, dx\, dy \\ &= \pi(x'|y') \int\int P_M(y|x, y')\pi(y'|x)\pi(x)\, dx\, dy \\ &= \pi(x'|y')\pi(y') \\ &= \pi(x', y') \end{aligned}$$

Given $\pi(x, y) \propto e^{-4x} y^3 e^{-3ye^x}$. $\pi(y|x)$ is easy to sample. $y|_x \sim Gamma(4, 3e^x)$. However, $\pi(x|y) \propto e^{-(4x+3ye^x)}$ would require metropolis-hasting step. Use a lognormal proposal density to do this Metropolis step:

$$q(x_2|x_1) = \frac{1}{x_2} e^{-(\log x_2 - \log x_1)^2/2}$$

In [28]:
```
def pi4(x, y):
    return np.exp(-4*x-3*y*np.exp(x))

def q4(x2, x1):
    return 1/x2 * np.exp(-1/2 * (np.log(x2) - np.log(x1))**2)

def alpha4(x2, x1, y):
    return min(1, pi4(x2, y) / pi4(x1, y) * x2/x1)
```

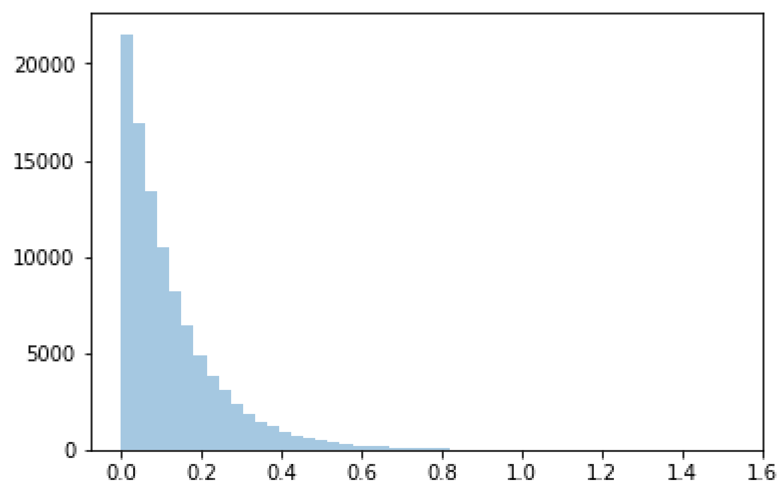
In [29]:
```
def MH_x(x, y):
    x_tmp = np.random.lognormal(np.log(x), 1)
    u = np.random.uniform()
    x = x_tmp if u < alpha4(x_tmp, x, y) else x
    return x

def draw_y(x):
    return np.random.gamma(shape = 4, scale = 1/3/np.exp(x))
```

In [30]:
```
N4 = 100000
samples4 = np.zeros((N4, 2))
x = 1
y = np.random.gamma(4, 5)
for i in range(N4):
    y = draw_y(x)
    x = MH_x(x, y)
    samples4[i, 0] = x
    samples4[i, 1] = y
```

After 100000 iterations, the sample distribution of x:

In [31]:
```
sns.distplot(samples4[:,0], kde=False)
```

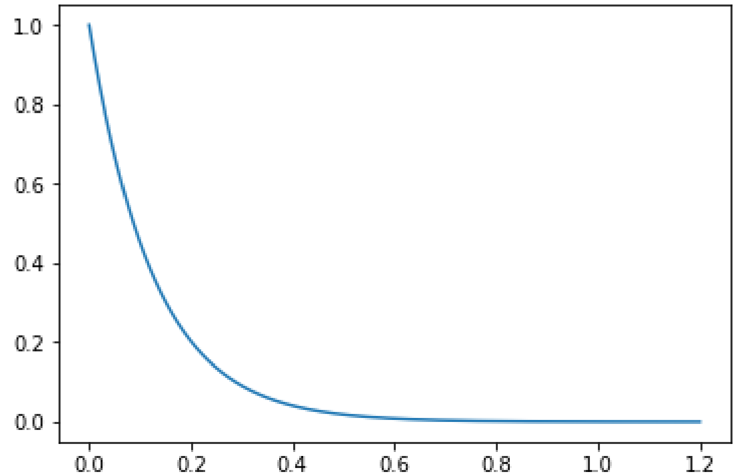
Out[31]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a1d953898>
```




The true marginal density of $x \propto e^{-8x}$

In [32]:
```
x = np.linspace(0,1.2, 1001)
plt.plot(x, np.exp(-8*x))
```

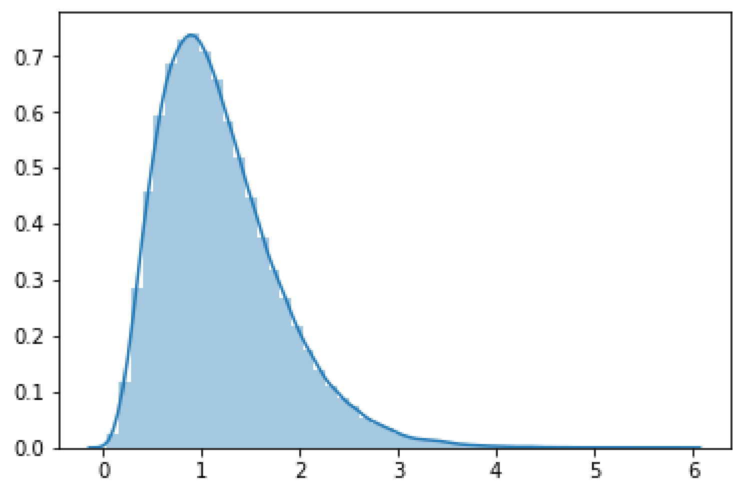
Out[32]: [<matplotlib.lines.Line2D at 0x1a1da4f7f0>]




The sample distribution of y:

In [33]:
```
sns.distplot(samples4[:, 1])
```

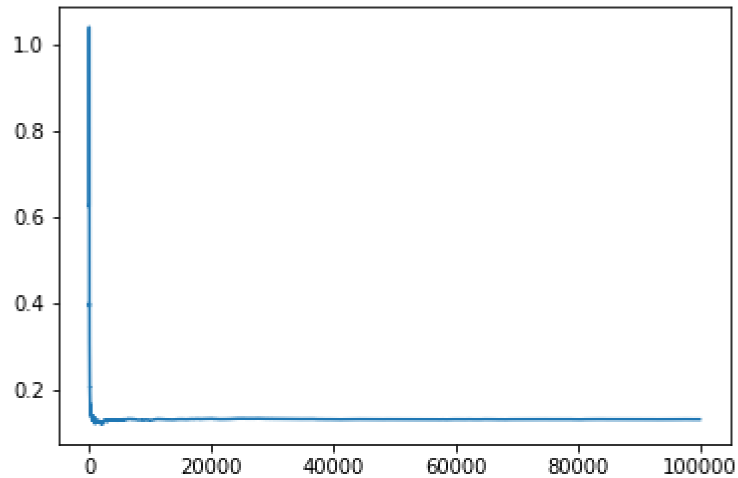
Out[33]: <matplotlib.axes._subplots.AxesSubplot at 0x1a1d980d30>




Plot the convergence of the Monte Carlo integration:

In [34]:
```
I4 = (samples4[:,0] * samples4[:,1])
for i in range(1, N4):
    I4[i] = (samples4[:i,0]*samples4[:i,1]).mean()

plt.plot(range(N4), I4)
```

Out[34]: [<matplotlib.lines.Line2D at 0x1a1e958b70>]




Use the first 20000 steps as burn-in and compute the estimated integration:

In [35]:
```
print("Estimated integration: ",(samples4[20000:,0] * samples4[20000:,1
]).mean())
```

```
Estimated integration:  0.131891629783
```

This estimation also agrees with the true integrationw which is approximately 0.1317