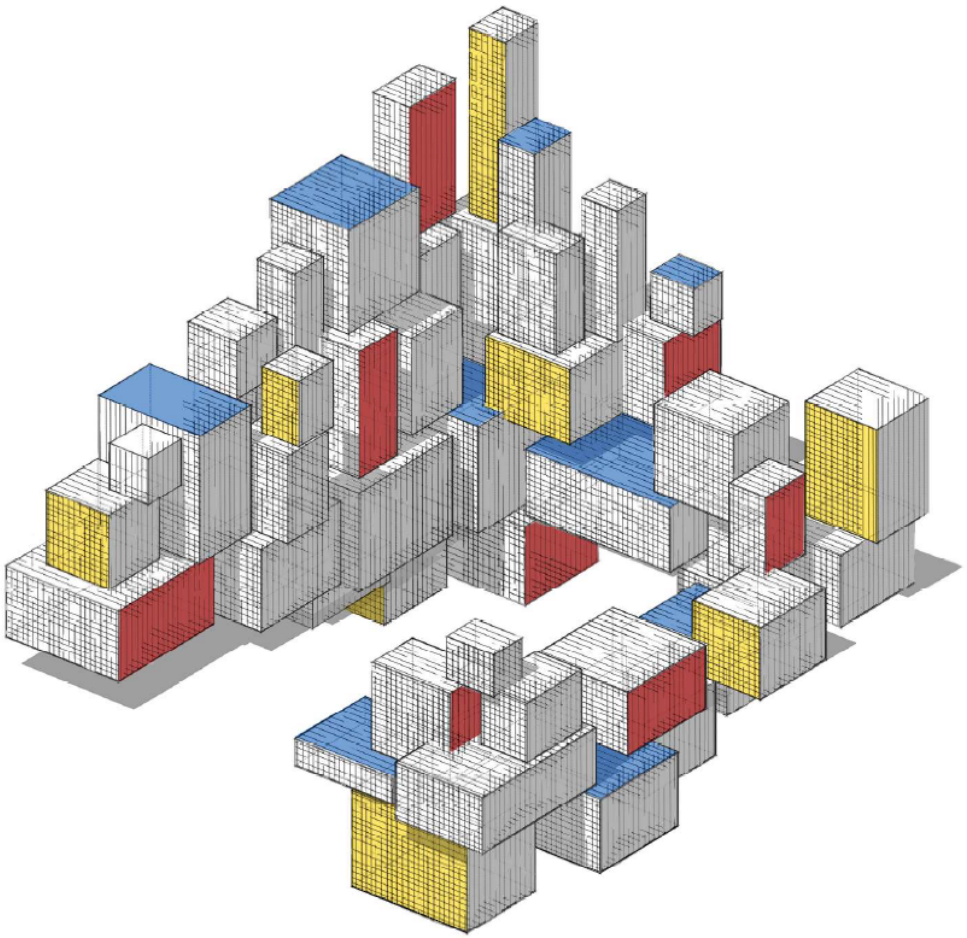
Monte Carlo for Estimation



Purpose

In this section we describe how Monte Carlo simulation can be used to estimate complicated integrals, probabilities, and expectations. The topics include:

- Crude Monte Carlo
- Monte Carlo Integration
- The Bootstrap Method

Crude Monte Carlo

Suppose we wish to compute the expectation

$$\mathbb{E}Y = \mu = \begin{cases} \int y f(y)\,\mathrm{d}y & \text{(continuous case)} \\ \sum y f(y) & \text{(discrete case).} \end{cases}$$

If Y is a complicated function of other random variables, it would be difficult to obtain an exact expression for $f(y)$.

The idea of crude Monte Carlo (CMC) is to approximate μ by simulating many independent copies $Y_1, \ldots, Y_N$ of Y and then take their sample mean $\overline{Y}$ as an estimator of μ.

All that is needed is an algorithm to simulate such copies.

Law of Large Numbers and Central Limit Theorem

By the Law of Large Numbers, $\overline{Y}$ converges to μ as $N \to \infty$, provided the expectation of Y exists.

Moreover, by the Central Limit Theorem, $\overline{Y}$ approximately has a $\mathcal{N}(\mu, \sigma^2/N)$ distribution for large N, provided that the variance $\sigma^2 = \mathbb{V}\mathrm{ar}\, Y < \infty$.

This enables the construction of an approximate $(1-\alpha)$ confidence interval for μ:

$$\left(\overline{Y} - z_{1-\alpha/2}\frac{S}{\sqrt{N}}, \quad \overline{Y} + z_{1-\alpha/2}\frac{S}{\sqrt{N}}\right), \tag{1}$$

where S is the sample standard deviation of the $\{Y_i\}$ and z_γ denotes the γ-quantile of the $\mathcal{N}(0,1)$ distribution.

The quantity $S/\sqrt{N}$ is the estimated standard error and $S/(\overline{Y}\sqrt{N})$ is the estimated relative error.

Crude Monte Carlo

Algorithm 1: Crude Monte Carlo for Independent Data

input: Simulation algorithm for $Y \sim f$, sample size N, confidence level $1-\alpha$.

output: Point estimate and approximate $(1-\alpha)$ confidence interval for $\mu = \mathbb{E}Y$.

1 Simulate $Y_1, \ldots, Y_N \overset{\text{iid}}{\sim} f$.

2 $\overline{Y} \leftarrow \frac{1}{N}\sum_{i=1}^{N} Y_i$

3 $S^2 \leftarrow \frac{1}{N-1}\sum_{i=1}^{N}(Y_i - \overline{Y})^2$

4 **return** $\overline{Y}$ and conf. interval $\left(\overline{Y} - z_{1-\alpha/2}\frac{S}{\sqrt{N}}, \quad \overline{Y} + z_{1-\alpha/2}\frac{S}{\sqrt{N}}\right)$.

It is often the case that the output Y is a function of some underlying random vector or stochastic process; that is, $Y = H(\boldsymbol{X})$, where H is a real-valued function and $\boldsymbol{X}$ is a random vector or process.

Monte Carlo Integration

In Monte Carlo integration, simulation is used to evaluate complicated integrals.

Consider, for example, the integral

$$\mu = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \sqrt{|x_1 + x_2 + x_3|}\, \mathrm{e}^{-(x_1^2+x_2^2+x_3^2)/2}\, \mathrm{d}x_1\, \mathrm{d}x_2\, \mathrm{d}x_3.$$

Defining $Y = |X_1 + X_2 + X_3|^{1/2}(2\pi)^{3/2}$, with $X_1, X_2, X_3 \overset{\text{iid}}{\sim} \mathcal{N}(0,1)$, we can write $\mu = \mathbb{E}Y$.

Using the following Python program, with a sample size of $N = 10^6$, we obtained an estimate $\overline{Y} = 17.031$ with an approximate 95% confidence interval $(17.017, 17.046)$.

mcint.py

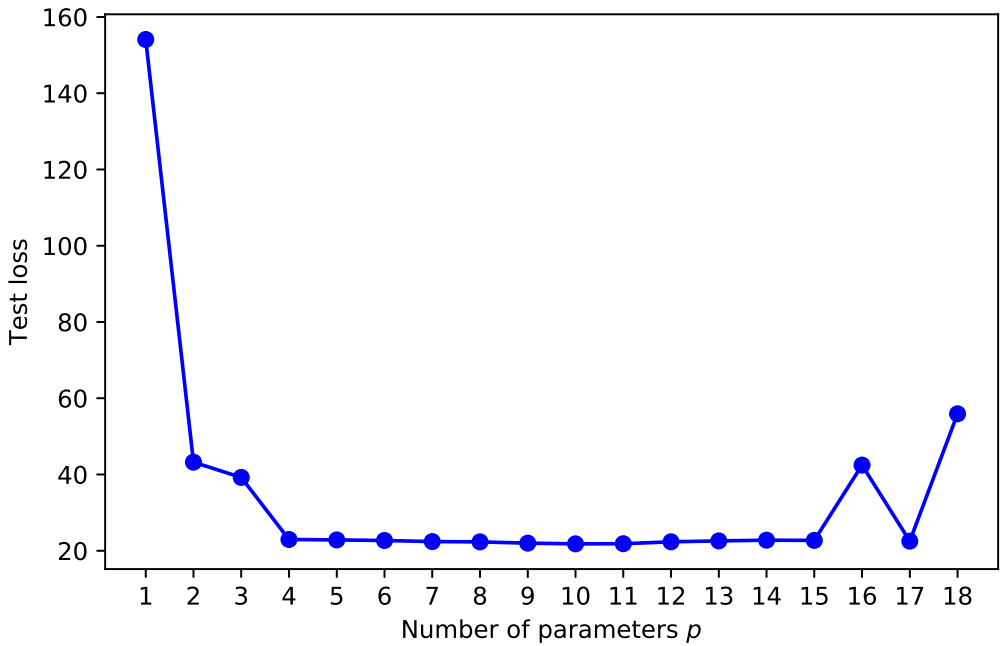
```
import numpy as np
from numpy import pi

c = (2*pi)**(3/2)
H = lambda x: c*np.sqrt(np.abs(np.sum(x,axis=1)))
N = 10**6
z = 1.96
x = np.random.randn(N,3)
y = H(x)
mY = np.mean(y)
sY = np.std(y)
RE = sY/mY/np.sqrt(N)
print('Estimate = {:3.3f}, CI = ({:3.3f},{:3.3f})'.format(
        mY, mY*(1-z*RE), mY*(1+z*RE)))
```

```
Estimate = 17.031, CI = (17.017,17.046)
```

Monte Carlo for Statistical Learning

We return to our running polynomial regression example.
Data: $(U_i, Y_i), i = 1, \ldots, 100$, where the $\{U_i\} \sim_{\text{iid}} \mathcal{U}(0, 1)$ and, given $U_i = u_i$, $Y_i \sim \mathcal{N}(10 - 140u_i + 400u_i^2 - 250u_i^3,\ 25)$.

We found the following test loss (estimate of the squared-error generalization risk) as a function of the number of parameters in the model. But how accurate is this graph?



Estimating the Generalization Risk

Because we know in this case the exact model for the data, we can use Monte Carlo simulation to estimate the generalization risk (for a fixed training set) and the expected generalization risk (averaged over all training sets) precisely.

All we need to do is repeat the data generation, fitting, and validation steps many times and then take averages of the results. The following Python code repeats 100 times:

1. Simulate the training set of size $n = 100$.
2. Fit models up to size $k = 8$.
3. Evaluate the test loss using a test set with the same sample size $n = 100$.

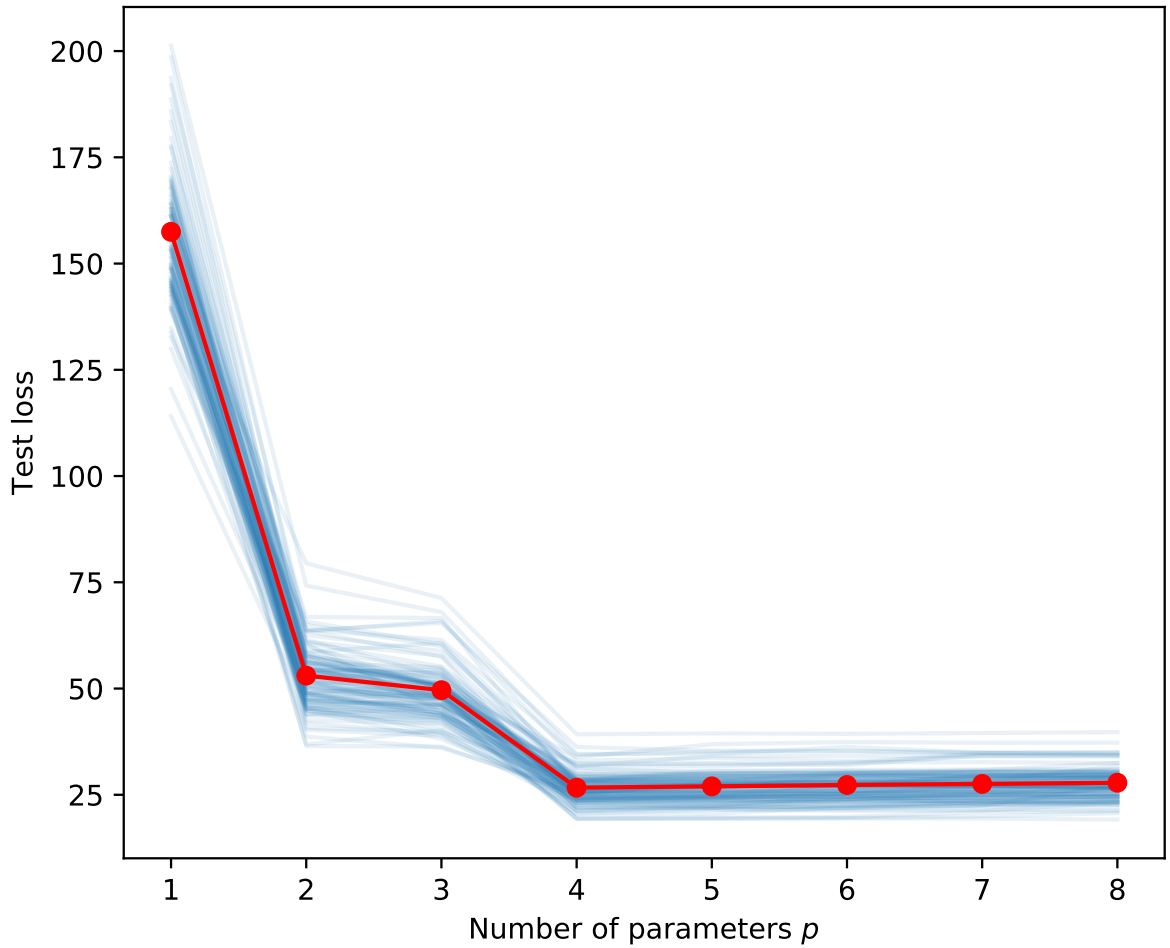
CMCtestloss.py

```
import numpy as np, matplotlib.pyplot as plt
from numpy.random import rand, randn
from numpy.linalg import solve

def generate_data(beta, sig, n):
    u = rand(n, 1)
    y = (u ** np.arange(0, 4)) @ beta + sig * randn(n, 1)
    return u, y

beta = np.array([[10, -140, 400, -250]]).T
n = 100, sig = 5, betahat = {}
totMSE = np.zeros(8), max_p = 8, p_range = np.arange(1, max_p + 1, 1)

for N in range(0,100):
    u, y = generate_data(beta, sig, n)  #training data
    X = np.ones((n, 1))
    for p in p_range:
       if p > 1:
         X = np.hstack((X, u**(p-1)))
         betahat[p] = solve(X.T @ X, X.T @ y)
```

```
    u_test, y_test = generate_data(beta, sig, n)  #test data
    MSE = []
    X_test = np.ones((n, 1))
    for p in p_range:
        if p > 1:
            X_test = np.hstack((X_test, u_test**(p-1)))
        y_hat = X_test @ betahat[p] # predictions
        MSE.append(np.sum((y_test - y_hat)**2/n))

    totMSE = totMSE + np.array(MSE)
    plt.plot(p_range, MSE,'C0',alpha=0.1)

plt.plot(p_range,totMSE/N,'r-o')
plt.xticks(ticks=p_range)
plt.xlabel('Number of parameters $p$')
plt.ylabel('Test loss')
plt.tight_layout()
plt.savefig('MSErepeatpy.pdf',format='pdf')
plt.show()
```

Expected Generalization Risk

We see that there is some variation in the test losses, due to the randomness in both the training and test sets. To obtain an accurate estimate of the expected generalization risk, take the average of the test losses.



Bootstrap Method

The bootstrap method combines CMC estimation with resampling.

The idea is as follows: Suppose we wish to estimate a number μ via some estimator $Y = H(\mathcal{T})$, where $\mathcal{T} := \{X_1, \ldots, X_n\}$ is an iid sample from some unknown cdf F.

It is assumed that Y does not depend on the order of the $\{X_i\}$.

To assess the quality (for example, accuracy) of the estimator Y, one could draw independent replications $\mathcal{T}_1, \ldots, \mathcal{T}_N$ of $\mathcal{T}$ and find sample estimates for quantities such as:

- variance $\mathbb{V}\mathrm{ar}\, Y$
- bias $\mathbb{E}Y - \mu$
- mean squared error $\mathbb{E}(Y - \mu)^2$

Bootstrap Method

However, it may be too time-consuming or simply not feasible to obtain such replications.

An alternative is to resample the original data; that is, given an outcome $\tau = \{x_1, \ldots, x_n\}$ of $\mathcal{T}$, simulate an iid sample $\mathcal{T}^* := \{X_1^*, \ldots, X_n^*\}$ from the empirical cdf F_n of τ.

Rationale: empirical cdf F_n is close to the actual cdf F and gets closer as n gets larger. Hence, any quantities depending on F, such as $\mathbb{E}_F g(Y)$, where g is a function, can be approximated by $\mathbb{E}_{F_n} g(Y)$. The latter is usually still difficult to evaluate, but it can be simply estimated via CMC as

$$\frac{1}{K} \sum_{i=1}^{K} g(Y_i^*),$$

where $Y_1^*, \ldots, Y_K^*$ are independent random variables, each distributed as $Y^* = H(\mathcal{T}^*)$.

Bootstrap Estimators

The bootstrap estimate of the expectation of Y is

$$\widehat{\mathbb{E}Y} = \overline{Y}^* = \frac{1}{K}\sum_{i=1}^{K} Y_i^*,$$

which is simply the sample mean of $\{Y_i^*\}$.

The bootstrap estimate for $\mathbb{V}\mathrm{ar}\,Y$ is the sample variance

$$\widehat{\mathbb{V}\mathrm{ar}\,Y} = \frac{1}{K-1}\sum_{i=1}^{K}(Y_i^* - \overline{Y}^*)^2. \tag{2}$$

More Bootstrap Estimators

Bootstrap estimators for the bias and MSE are $\overline{Y}^* - Y$ and $\frac{1}{K}\sum_{i=1}^{K}(Y_i^* - Y)^2$, respectively. Note that for these estimators the unknown quantity μ is replaced with its original estimator Y.

Confidence intervals can be constructed in the same fashion. We mention two variants:

- Normal method: $1-\alpha$ confidence interval for μ is given by

$$(Y \pm z_{1-\alpha/2} S^*),$$

  where S^* is the bootstrap estimate of the standard deviation of Y; that is, the square root of (2).

- Percentile method: the upper and lower bounds of the $1-\alpha$ confidence interval for μ are given by the $1-\alpha/2$ and $\alpha/2$ quantiles of Y, which in turn are estimated via the corresponding sample quantiles of the bootstrap sample $\{Y_i^*\}$.

Bootstrapping the Ratio Estimator

A common scenario in stochastic simulation is that the output of the simulation consists of iid pairs of data $(C_1, R_1), (C_2, R_2), \ldots$, where each C is interpreted as the length of a period of time — a so-called cycle — and R is the reward obtained during that cycle.

Such a collection of random variables $\{(C_i, R_i)\}$ is called a renewal reward process. Typically, the reward R_i depends on C_i.

Let A_t be the *average reward* earned by time t; that is, $A_t = \sum_{i=1}^{N_t} R_i/t$, where $N_t = \max\{n : C_1 + \cdots + C_n \leqslant t\}$ counts the number of complete cycles at time t. It can be shown (exercise!) that

$$A_t \to \frac{\mathbb{E}R}{\mathbb{E}C} \quad \text{(the long-run average reward)}.$$

Estimation of the Long-run Average Reward

Estimation of the ratio $\mathbb{E}R/\mathbb{E}C$ from data $(C_1, R_1), \ldots, (C_n, R_n)$ is easy: take the ratio estimator

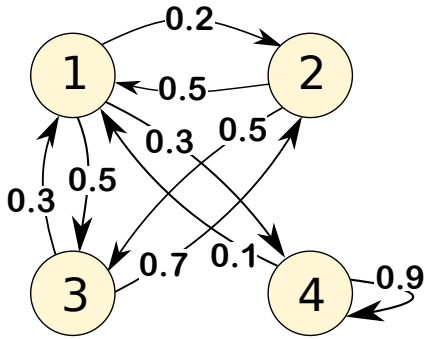
$$A = \frac{\overline{R}}{\overline{C}}.$$

However, this estimator A is not unbiased and it is not obvious how to derive confidence intervals.

Fortunately, the bootstrap method can come to the rescue: simply resample the pairs $\{(C_i, R_i)\}$, obtain ratio estimators $A_1^*, \ldots, A_K^*$, and from these compute quantities of interest such as confidence intervals.

Example: Renewal Reward Markov Chain

Consider a 4-state Markov chain starting at state 1 at time 0.



Let $T_1, T_2, \ldots$ be the times that the process returns to state 1. This breaks up the time interval into independent cycles of lengths $C_i = T_i - T_{i-1}$, $i = 1, 2, \ldots$. During the i-th cycle a reward

$$R_i = \sum_{t=T_{i-1}}^{T_i - 1} \varrho^{t - T_{i-1}} \, r(X_t)$$

is received, where $r(i)$ is some fixed reward for visiting state $i \in \{1, 2, 3, 4\}$ and $\varrho \in (0, 1)$ is a discounting factor. Clearly, $\{(C_i, R_i)\}$ is a renewal reward process.

Renewal Reward Markov Chain

The figure shows the outcomes of 1000 pairs (C, R), using $r(1) = 4, r(2) = 3, r(3) = 10, r(4) = 1$, and $\varrho = 0.9$.

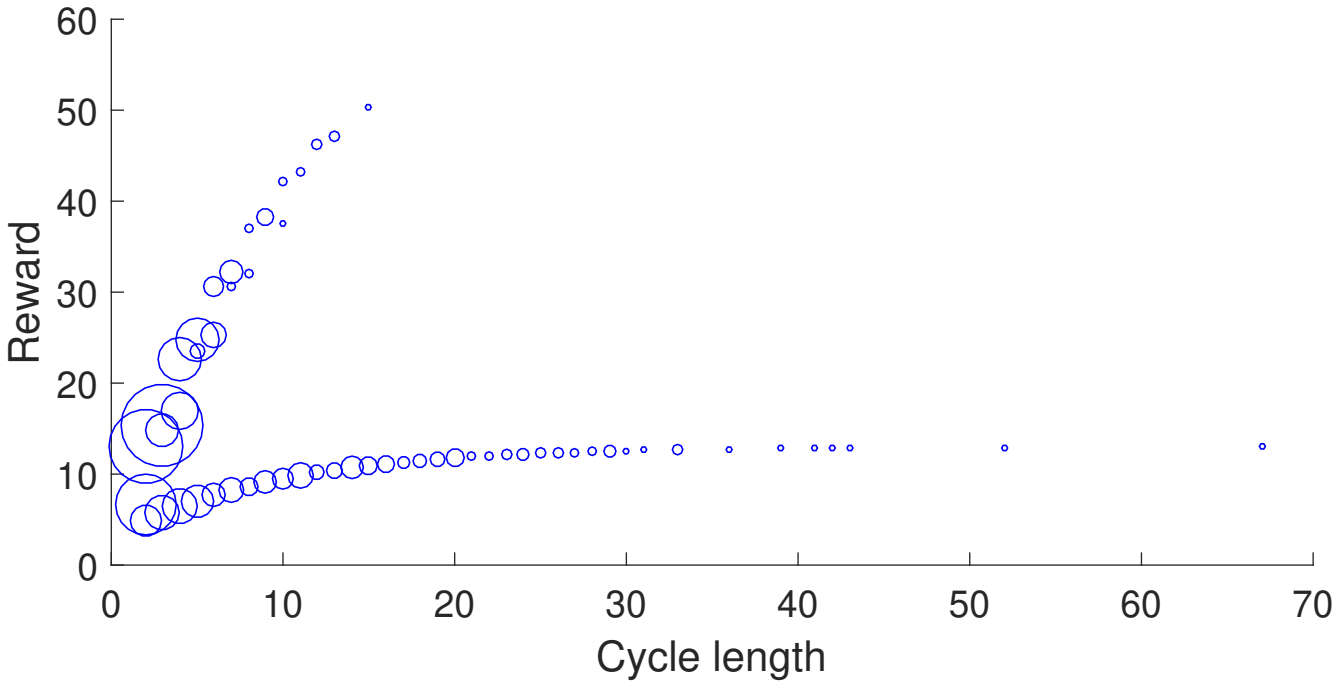
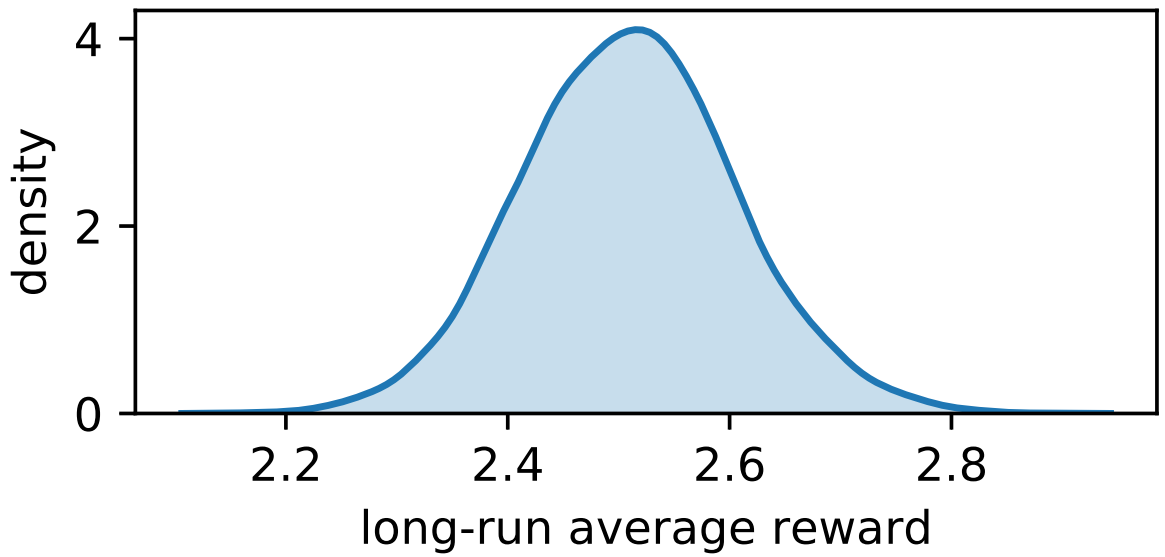


Figure: Each circle represents a (cycle length, reward) pair. The varying circle sizes indicate the number of occurrences for a given pair. For example, (2,15.43) is the most likely pair here, occurring 186 out of a 1000 times. It corresponds to the cycle path $1 \to 3 \to 2 \to 1$.

Long-run Average Reward

The long-run average reward is estimated as 2.50 for our data. But how accurate is this estimate? The figure below shows a density plot of the bootstrapped ratio estimates, where we independently resampled the data pairs 1000 times.



This indicates that the true long-run average reward lies between 2.2 and 2.8 with high confidence.

ratioest.py

```
import numpy as np, matplotlib.pyplot as plt, seaborn as sns
from numba import jit
np.random.seed(123), n = 1000
P = np.array([[0, 0.2, 0.5, 0.3],
              [0.5 ,0, 0.5, 0],
              [0.3, 0.7, 0, 0],
              [0.1, 0, 0, 0.9]])
r = np.array([4,3,10,1])
Corg = np.array(np.zeros((n,1)))
Rorg = np.array(np.zeros((n,1)))
rho=0.9

@jit()   #for speed-up; see Appendix
def generate_cyclereward(n):
   for i in range(n):
       t=1
       xreg = 1   #regenerative state  (out of 1,2,3,4)
       reward = r[0]
       x= np.amin(np.argwhere(np.cumsum(P[xreg-1,:]) > np.random.rand()
            )) + 1
```

```
        while x != xreg:
            t += 1
            reward += rho**(t-1)*r[x-1]
            x = np.amin(np.where(np.cumsum(P[x-1,:]) > np.random.rand())
                ) + 1
        Corg[i] = t
        Rorg[i] = reward
    return Corg, Rorg

Corg, Rorg = generate_cyclereward(n)
Aorg = np.mean(Rorg)/np.mean(Corg)
K = 5000
A = np.array(np.zeros((K,1)))
C = np.array(np.zeros((n,1)))
R = np.array(np.zeros((n,1)))
for i in range(K):
    ind = np.ceil(n*np.random.rand(1,n)).astype(int)[0]-1
    C = Corg[ind]
    R = Rorg[ind]
    A[i] = np.mean(R)/np.mean(C)

plt.xlabel('long-run average reward')
plt.ylabel('density')
sns.kdeplot(A.flatten(),shade=True)
```