

MIE1613 – Stochastic Simulation Homework 1

Due: February 2, 11:59pm

Problem 1:

1a) Calculating theta exactly using the definition of expected value:

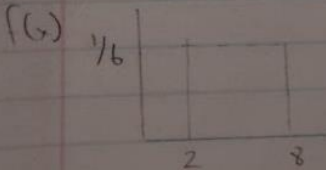
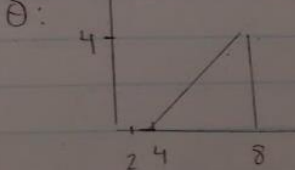
MIE1613 - Stochastic Simulation
Wednesday, February 2nd, 2022
Homework #1

1) X is uniformly distributed in $[2, 8]$ - Continuous Random Variable
 $\theta = E[(X-4)^+]$ $a^+ = \max(a, 0) \rightarrow$ Cannot be negative

a) Compute θ exactly using definition of expected value

$f(x) = \frac{1}{b-a} = \frac{1}{8-2} = \frac{1}{6}$ $0 \rightarrow 1$ 0.5
 $2 \rightarrow 8$

$E(x) = \int_{-\infty}^{\infty} x f(x) dx$ Expected Value: $E(\theta) = \frac{2}{6}(0) + \frac{4}{6}((4-2)/2)$
 $= \frac{4}{6} \times 2 = \frac{8}{6} = \frac{4}{3}$

$f(x)$  θ : 

$f(x) = \int x \frac{1}{6} = \frac{1}{12} x^2 \Big|_0^4 = \frac{16}{12} = \frac{4}{3}$ $x-4$

We can observe that the exact value of the theta is $4/3$ (1.3333).

1b) Using Monte Carlo Simulation with 5000 samples: (Using Google Colab)

Question 1

```
[3] import numpy as np
import matplotlib.pyplot as plt
from statistics import mean

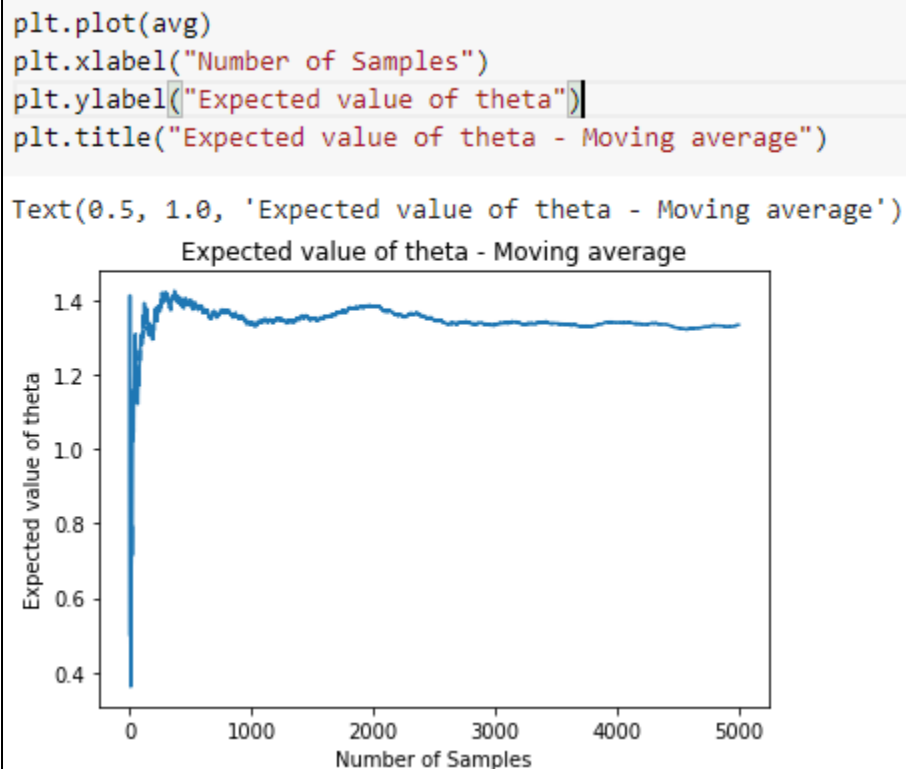
np.random.seed(1)

list = [] # List of experimental theta values
avg = [] # List of MOving average for part 1c
for i in range(5000):
    list.append(max(0, 6*np.random.random() + 2 - 4))
    avg.append(mean(list))

print ('95% CI:', np.mean(list), "+/-",
        1.96*np.std(list, ddof = 1)/np.sqrt(5000))

95% CI: 1.3328728457484547 +/- 0.036893839228342606
```

1c) Plotting convergence of Monte Carlo estimate to the exact value as number of samples increase



Question 2a:

Question 2a - Generate one sample path of $T(S)$ of the time to $T = 1000$ (Instead of time to failure)

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

# start with 2 functioning components at time 0
clock = 0
S = 2
T=1000 # For 1000 days

# fix random number seed
np.random.seed(1)

# initialize the time of events
NextRepair = float('inf')
NextFailure = np.ceil(6*np.random.random())
# lists to keep the event times and the states
EventTimes = [0]
States = [2]
# Define variables to keep the area under the sample path
# and the time and state of the last event
Area = 0.0
Tlast = 0
Slast = 2

while clock < T: # While the time is less than 1000, we count the number of functioning samples
    # advance the time
    clock = min(NextRepair, NextFailure)

    # For these eqns, the latter state is being calculated
    if NextRepair <= NextFailure and S==1: #Going from State 1 to State 2 (Repair)
        # next event is completion of a repair
        S = S + 1
        NextRepair = float('inf') # No more awaiting repair times
        print('Fix at 1') # Sanity check
```

```
if NextRepair <= NextFailure and S==1: #Going from State 1 to State 2 (Repair)
    # next event is completion of a repair
    S = S + 1
    NextRepair = float('inf') # No more awaiting repair times
    print('Fix at 1') # Sanity check
elif S==0: # Going from State 0 to State 1 (System failed to operating)
    NextRepair = clock + 2.5 # component repaired, immediately start another repair
    S = S+1
    NextFailure = clock + np.ceil(6*np.random.random())
    print('Fix at 0') # Sanity check
elif S==2: # Going from State 2 to State 1 (component fail)
    S = S - 1
    NextFailure = clock + np.ceil(6*np.random.random())
    NextRepair = clock + 2.5 # component failed, immediately start repair
    print('Fail at 2') # Sanity check

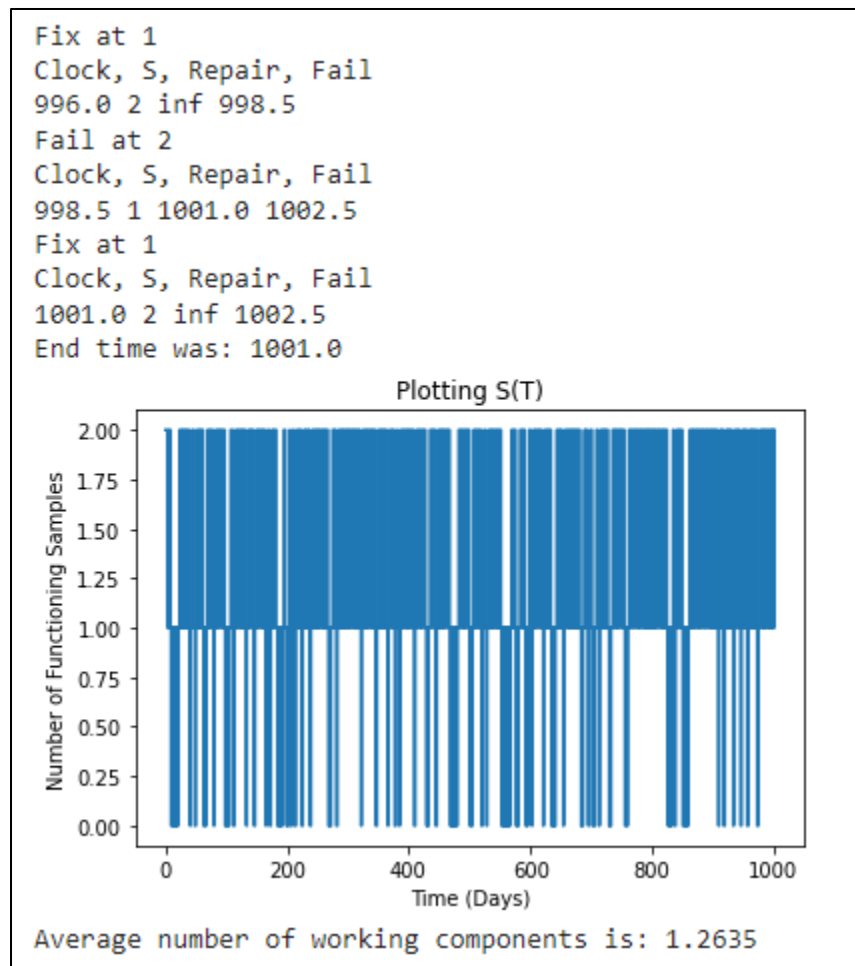
else: # S=1, going from state 1 to 0 (System failed)
    # next event is a failure
    S = S - 1
    NextFailure = float('inf') # Cannot fail anymore, waiting repairs
    print('Fail at 1') # Sanity check

# Calculating the area under the line
Area = Area + (clock - Tlast)* Slast
Tlast = clock
Slast = S
print("Clock, S, Repair, Fail")
print(clock, S, NextRepair, NextFailure)

# save the time and state
EventTimes.append(clock)
States.append(S)

# plot the sample path
print ('End time was:', clock)
plt.plot(EventTimes, States, drawstyle = 'steps-post')
plt.xlabel('Time (Days)')
plt.ylabel('Number of Functioning Samples')
plt.title('Plotting S(T)')
plt.show()
# Need to take account when the clock passes T=1000 by subtracting the area that is beyond T=1000
print('Average number of working components is:', (Area - (clock - T)*Slast)/T)
```

Output:



The average number of functional components until time $T = 1000$ based on one replication of the simulation is 1.2635.

Question 2b:

Question 2b - Determining availability

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

# start with 2 functioning components at time 0
clock = 0
S = 2
T=1000 # Until Day 1000

# fix random number seed
np.random.seed(1)

# initialize the time of events
NextRepair = float('inf')
NextFailure = np.ceil(6*np.random.random())
# lists to keep the event times and the states
EventTimes = [0]
States = [2]
# Define variables to keep the area under the sample path
# and the time and state of the last event
up = 0
Area = 0.0
Tlast = 0
Slast = 2

while clock < T:
    # advance the time
    clock = min(NextRepair, NextFailure)

    if NextRepair <= NextFailure and S==1: #State 2
        # next event is completion of a repair
        S = S + 1
        NextRepair = float('inf') #Check
    elif S==0: #State 1
        NextRepair = clock + 2.5
        S = S+1
        NextFailure = clock + np.ceil(6*np.random.random())
```

```
elif S==2: #State 1
    S = S - 1
    NextFailure = clock + np.ceil(6*np.random.random())
    NextRepair = clock + 2.5

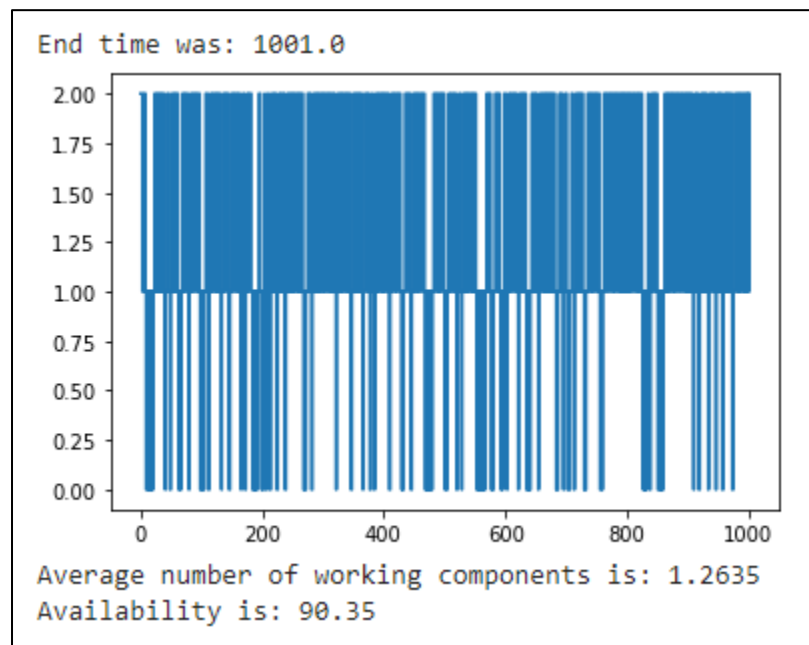
else: #S= 1
    # next event is a failure
    S = S - 1
    NextFailure = float('inf') #Check

# Availability, the rest of the code is the same as 2a
up = up + (clock - Tlast)*min(Slast, 1) # If it's state 1 or 2, it will be equal to 1

Area = Area + (clock - Tlast)* Slast
Tlast = clock
Slast = S

# save the time and state
EventTimes.append(clock)
States.append(S)

# plot the sample path
print ('End time was:', clock)
plt.plot(EventTimes, States, drawstyle = 'steps-post')
plt.show()
print('Average number of working components is:', (Area - (clock - T)*Slast)/T)
print('Availability is:', (up - (clock - T)*min(Slast,1))/T * 100)
```



The availability until time $T = 1000$ based on one replication of the simulation is 90.35%.

2c) Compare T=1000 with T=3000:

Average Functioning Components at Time 1000: 1.2635, Time 3000: 1.2695

Availability at Time 1000: 90.35%, Time 3000: 91.36666%

We can see that the average functioning components converges to about 1.27 and that the availability of the system converges to 91.3% as number of days increase.

Question 3:

Question 3 - Run multiple replications and estimate the time to failure based off multiple spares

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

# Set number of replications
R = 1000
# Define lists to keep samples of the outputs across replications
TTF_list = []
Ave_list = []

# fix random number seed
np.random.seed(1)

for rep in range(0,R): # Number of replication loops
    # start with N functioning components at time 0
    clock = 0
    N = 4 # Variable number of total components (Replace with 2, 3, 4)
    S = N
    # initialize the time of events
    NextRepair = float('inf')
    NextFailure = np.ceil(6*np.random.random())
    EventTimes = [0]
    States = [S]
    # Define variables to keep the area under the sample path
    # and the time and state of the last event
    Area = 0.0
    Tlast = 0
    Slast = S

    while S > 0: # Time to failure, system is still functioning
        # advance the time
        clock = min(NextRepair, NextFailure)

        if NextRepair <= NextFailure and S == (N - 1): # Going from N-1 to N, a repair
            # next event is completion of a repair
            S = S + 1
            NextRepair = float('inf')
        elif NextRepair <= NextFailure: # A repair (If repair = failure time, will repair first.)
```



```
NextRepair = clock + 2.5
elif NextRepair <= NextFailure: # A repair (If repair = failure time, will repair first.)
# This assumptions increases our expected time to failure
# next event is completion of a repair
S = S + 1
NextRepair = NextRepair + 2.5
elif S==N: # Going from N to N-1, next event is a failure
S = S - 1
NextFailure = clock + np.ceil(6*np.random.random())
NextRepair = clock + 2.5
else: # next event is a failure
S = S - 1
NextFailure = clock + np.ceil(6*np.random.random())
# Update the area under the sample path and the
# time and state of the last event
Area = Area + (clock - Tlast)* Slast
Tlast = clock
Slast = S

# save the TTF and average # of func. components
TTF_list.append(clock)
Ave_list.append(Area/clock)

print('Estimated expected TTF:', np.mean(TTF_list))
print('Estimated expected ave. # of func. comp. till failure:', np.mean(Ave_list))

print ('95% CI for TTF:', np.mean(TTF_list), "+/-",
1.96*np.std(TTF_list, ddof = 1)/np.sqrt(R))
print ('95% CI for ave. # of func. comp.:', np.mean(Ave_list), "+/-",
1.96*np.std(Ave_list, ddof = 1)/np.sqrt(R))

Estimated expected TTF: 990.0
Estimated expected ave. # of func. comp. till failure: 3.09325530496805
95% CI for TTF: 990.0 +/- 61.72667191920702
95% CI for ave. # of func. comp.: 3.09325530496805 +/- 0.007057533818077841
```

N=2:

Estimated expected TTF: 14.193

Estimated expected ave. # of func. comp. till failure: 1.5585877994579558

95% CI for TTF: 14.193 +/- 0.7212458114694061

95% CI for ave. # of func. comp.: 1.5585877994579558 +/- 0.008130579538131004

N = 3:

Estimated expected TTF: 110.959

Estimated expected ave. # of func. comp. till failure: 2.239361844747321

95% CI for TTF: 110.959 +/- 7.058870779760439

95% CI for ave. # of func. comp.: 2.239361844747321 +/- 0.0084373435720208

N=4:

Estimated expected TTF: 990.0

Estimated expected ave. # of func. comp. till failure: 3.09325530496805

95% CI for TTF: 990.0 +/- 61.72667191920702

95% CI for ave. # of func. comp.: 3.09325530496805 +/- 0.007057533818077841

Question 4: The standard error of an estimator is defined as the standard deviation of that estimator. Sample mean is an estimator of $E(X)$ where X_i 's are i.i.d. samples of the random variable X . What is the standard error of \bar{X}_n . Assume that the standard deviation of X is σ .

The average of n i.i.d. variables is $1/\sqrt{n}$ as variable as any one of the random variables. This relates to the law of large numbers and central limit theorem.

Thus, the standard error of \bar{X}_n is σ/\sqrt{n} .

Question 5 - X-bar averages and histograms

```
np.random.seed(1)
avg = []
k = 1
i = 1
while i <= 1000:
    avg.append(np.random.random())
    i = i+1 # No for loop, only 1 sample
print('Average of X1 for 1000 samples is: ', np.mean(avg))
plt.figure(0)
plt.xlabel('Xbar value')
plt.ylabel('Number of occurrences')
plt.title('Xbar 1')
plt.hist(avg)

list = []
avg = []
i=1
k = 2
while i <= 1000:
    sum = 0
    for x in range(k): # For loop, average of 2 samples per replication
        sum = sum + np.random.random() # Sum of all X's

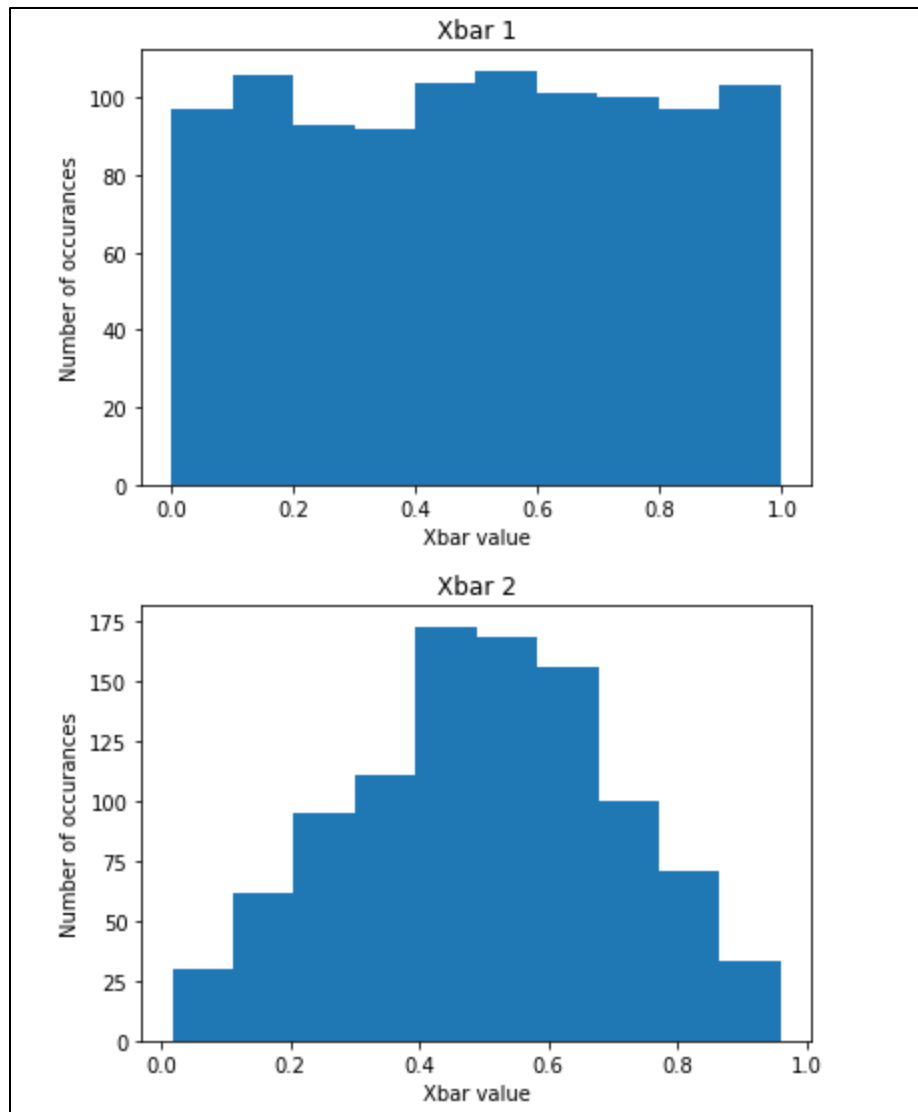
    avg.append(sum /k) # Average of X's to get Xbar-i
    i = i+1
print('Average of X2 for 1000 samples is: ', np.mean(avg))
plt.figure(1)
plt.xlabel('Xbar value')
plt.ylabel('Number of occurrences')
plt.title('Xbar 2')
plt.hist(avg)
```

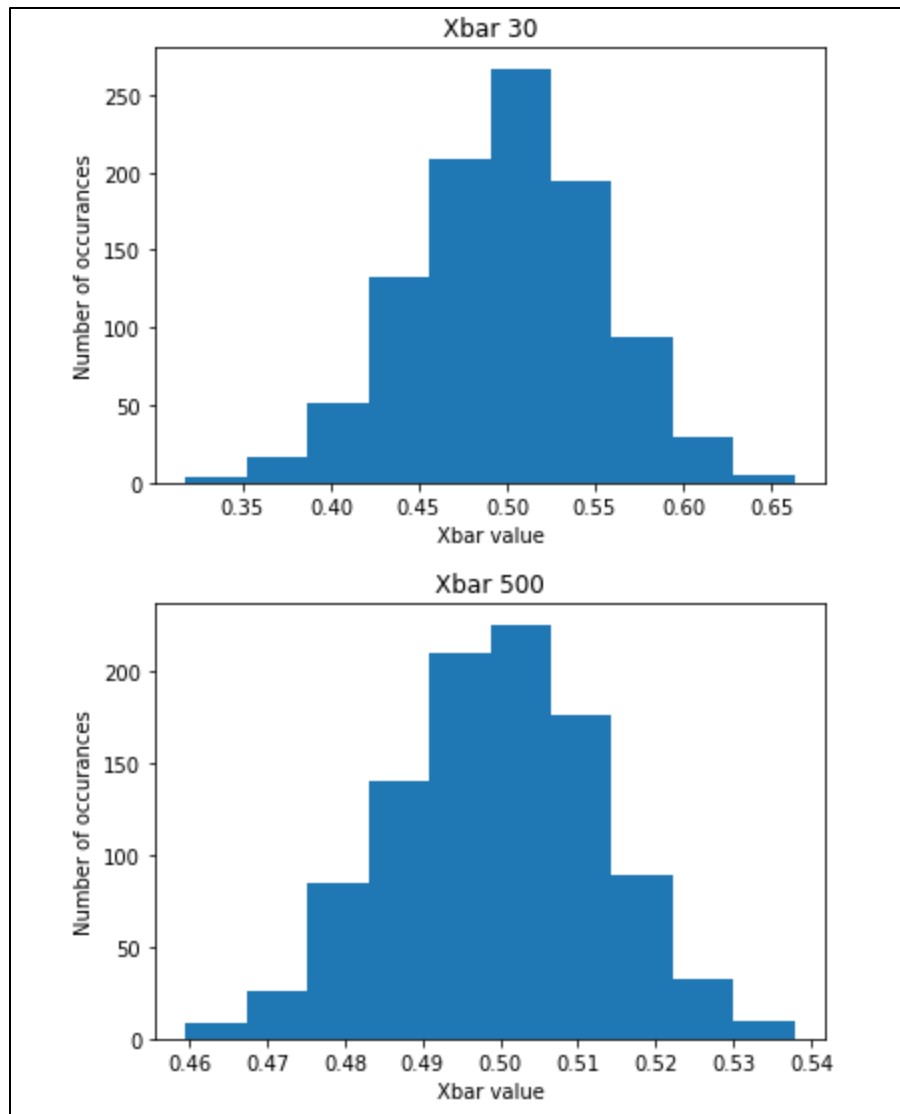
```
list = []
avg = []
i=1
k = 30
while i <= 1000:
    sum = 0
    for x in range(k): # For loop, average of 30 samples per replication
        sum = sum + np.random.random() # Sum of all X's

    avg.append(sum /k) # Average of X's to get Xbar-i
    i = i+1
print('Average of X30 for 1000 samples is: ', np.mean(avg))
plt.figure(3)
plt.xlabel('Xbar value')
plt.ylabel('Number of occurances')
plt.title('Xbar 30')
plt.hist(avg)

list = []
avg = []
i=1
k = 500
while i <= 1000:
    sum = 0
    for x in range(k): # For loop, average of 500 samples per replication
        sum = sum + np.random.random() # Sum of all X's

    avg.append(sum /k) # Average of X's to get Xbar-i
    i = i+1
print('Average of X500 for 1000 samples is: ', np.mean(avg))
plt.figure(4)
plt.xlabel('Xbar value')
plt.ylabel('Number of occurances')
plt.title('Xbar 500')
plt.hist(avg)
```





As n increases, the mean converges to 0.50 and the variability shrinks (we see that the horizontal scale is shrinking). The central limit theorem states that if you have a population with mean μ and standard deviation σ and take sufficiently large random samples from the population, then the distribution of the sample means will be approximately normally distributed. We can see that the data is fitting more into a bell shape curve with narrower bands.