Stochastic Simulation (MIE1613H) - Homework 1 (Solutions)

Due: February 3, 2021

Problem 1. **(20 Pts.)** Assume that X is uniformly distributed in $[2, 8]$ (X is a continuous random variable). We are interested in computing $\theta = E[(X-4)^+]$. (Note: $a^+ = \text{Max}(a, 0)$, i.e, if $a < 0$ then $a^+ = 0$ and if $a \geq 0$ then $a^+ = a$.)

(a) Compute θ exactly using the definition of expected value.

(5 points) Recalling that the pdf of a uniform random variable is $\frac{1}{b-a}$ for $x \in [a, b]$ and using the definition of the expected value of a function of a continuous random variable we have,

$$
\begin{aligned}
\theta = E[(X-4)^+] &= \int_a^b (x-4)^+ \frac{1}{b-a} dx \\
&= \int_4^8 \frac{1}{6}(x-4)dx \\
&= \frac{x^2}{12} - \frac{4x}{6}\bigg|_4^8 = (\frac{64}{12} - \frac{32}{6}) - (\frac{16}{12} - \frac{16}{6}) = \frac{4}{3} \approx 1.333.
\end{aligned}
$$

(b) Estimate θ using Monte Carlo simulation and provide a 95% confidence interval for your estimate. **Note**: Use the `np.random.random()` method in Python and transform it to a sample of X. You may NOT use other built-in methods of Python to generate the samples.

(10 points) To estimate θ we generate $n = 10,000$ iid samples of the random variable $(X-4)^+$ and compute the sample average. To transform a sample from a uniformly distributed random variable in $[0, 1]$ to one in $[a, b]$ we first multiply it by $(b-a)$ to get a sample between $[0, b-a]$ and then add a to it to obtain a sample in $[a, b]$. The sample average estimate is 1.32 and the 95% CI is given by $[1.30, 1.35]$ which includes the exact value.

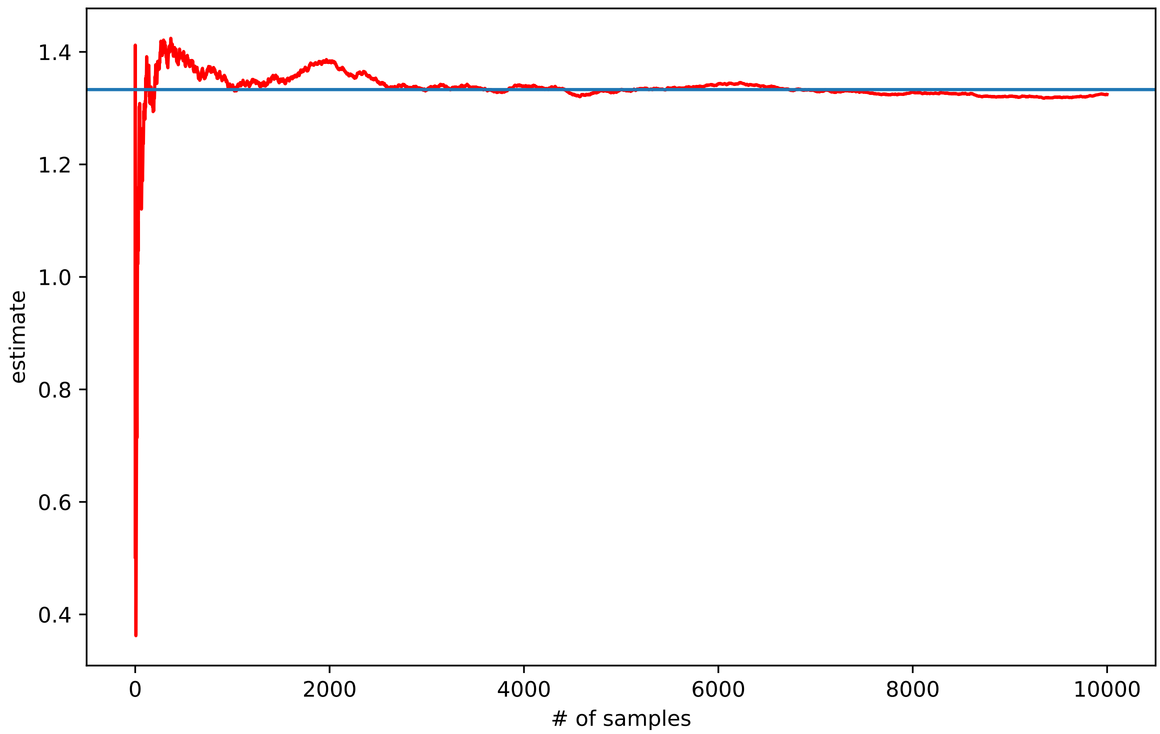
(c) Create a plot that demonstrates the convergence of the Monte Carlo estimate to the exact value as the number of samples increases.

(5 points) See the source code and the output below.

```python
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats
np.random.seed(1)

def mean_confidence_interval_95(data):
    a = 1.0*np.array(data)
    n = len(a)
    m, se = np.mean(a), stats.sem(a)
    h = 1.96*se
```

```
    return m, m-h, m+h

samples = []
estimates = []
for n in range(0,10000):
    # generate a sample of the random variable and append to the list
    X = (8-2)*np.random.random() + 2
    samples.append(max(X-4, 0))
    estimates.append(np.mean(samples))
print("The estimate and 95% CI:", mean_confidence_interval_95(samples))

plt.plot(estimates, 'r')
plt.xlabel('# of samples')
plt.ylabel('estimate')
plt.axhline(y = 1.333)
plt.rcParams['figure.figsize'] = (10, 6)
plt.rcParams.update({'font.size': 14})
plt.show()
```

```
The estimate and 95% CI: (1.3244243249303704, 1.2984542811761661, 1.3503943686845747)
```



Problem 2. **(30 Pts.)** In the TTF example from the first class we simulated the system until the time of first failure. Modify the simulation model to simulate the system for a given number of days denoted by T. Assume that all other inputs and assumptions are the same as in the original example.

(a) What is the average number of functional components until time $T = 1000$ based on one replication of the simulation?

(10 points) The logic is modified as follows. When the system fails, i.e., $S = 0$, one repair is still pending. Therefore, we do not reschdule another repair but set the NextFailure to ∞. In addition, if $S = 1$ after the completion of a repair, we need to schedule the repair of the other component, and the failure of the component that just started working. When the simulation ends, i.e., EndSimulation event occurs, we call the EndSimulation function to update the area variable. Simulating one sample path of the process $S(t)$ for $T = 1000$ time units we have

$$\frac{1}{1000}\int_0^{1000} S(t)dt = 1.2645.$$

```
import numpy as np

def EndSimulation ():
    global Slast
    global Tlast
    global Area

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

def Failure ():
    global S
    global Slast
    global Tlast
    global Area
    global NextFailure
    global NextRepair

    S = S - 1
    if S == 0:
        NextFailure = float('inf')
        # repair already in progress
    if S == 1:
        NextRepair = clock + 2.5
        NextFailure = clock + np.ceil(6*np.random.random())
    # Update the area under the sample path and the time and state at the last event
    Area = Area + (clock - Tlast)* Slast
    Tlast = clock
    Slast = S

def Repair():
    global S
    global Slast
    global Tlast
    global Area
    global NextFailure
    global NextRepair

    S = S + 1
    if S == 1:
```

```python
        NextRepair = clock + 2.5
        NextFailure = clock + np.ceil(6*np.random.random())
    else: #S==2
        NextRepair = float('inf')
    Area = Area + Slast * (clock - Tlast)
    Slast = S
    Tlast = clock

def Timer():
    global clock
    global NextRepair
    global NextFailure

    if NextEndSimulation < NextFailure and NextEndSimulation < NextRepair:
        result = "EndSimulation"
        clock = NextEndSimulation

    elif NextFailure < NextRepair:
        result = "Failure"
        clock = NextFailure

    else:
        result = "Repair"
        clock = NextRepair
    return result


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

clock = 0
S = 2
# initialize the time of events
NextRepair = float('inf')
NextFailure = np.ceil(6*np.random.random())
NextEndSimulation = 1000
# 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
NextEvent = Timer()

while NextEvent!="EndSimulation":
    NextEvent = Timer()
    if NextEvent == "Repair":
        Repair()
    elif NextEvent == "Failure":
        Failure()
    else:
        EndSimulation()

print('Average # of func. comp. till time T = 1000:', Area/clock)
```

```
Average # of func.  comp.  till time T = 1000:  1.2645
```

(b) We say that the system is available provided that there is at least one functional component. Denote by $A(t)$ a process that takes value 1 if the system is available and 0 otherwise. Then,

$$\bar{A}(T) = \frac{1}{T}\int_0^T A(t)dt,$$

is the average system availability from time 0 to T. Modify your simulation model to estimate the average system availability until $T = 1000$ based on one replication of the simulation.

(10 points) The logic is modified as follows. We start with $A = 1$ as the system is initially available. Then, when the system fails, i.e., $S = 0$, we need to set $A = 0$ and the NextFailure to ∞. In addition, if $S = 1$ after the completion of a repair, we need to set $A = 1$, schedule the repair of the other component, and schedule the failure of the component that just started working. Note that we now use the Area variable in the code to calculate $\int_0^T A(t)dt$. Simulating one sample path of the process $A(t)$ for $T = 1000$ time units we have

$$\frac{1}{1000}\int_0^{1000} A(t)dt = 0.9035.$$

```python
import numpy as np

def EndSimulation ():
    global Alast
    global Tlast
    global Area

    Area = Area + (clock - Tlast) * Alast
    Tlast = clock
    Alast = A

def Failure ():
    global S
    global A
    global Alast
    global Tlast
    global Area
    global NextFailure
    global NextRepair

    S = S - 1
    if S == 0:
        # system becomes unavailable
        A = 0
        # Update the area under the sample path and the time and status at
            the last event
        Area = Area + (clock - Tlast) * Alast
        Tlast = clock
        Alast = A
        # repair already in progress
        NextFailure = float('inf')
```

```
    else: # S==1
        NextRepair = clock + 2.5
        NextFailure = clock + np.ceil(6*np.random.random())

def Repair():
    global S
    global A
    global Alast
    global Tlast
    global Area
    global NextFailure
    global NextRepair

    S = S + 1
    if S == 1:
        # system becomes available
        A = 1
        # Area does not change since the system was unavailable
        # Update the time and status at the last event
        Tlast = clock
        Alast = A

        NextRepair = clock + 2.5
        NextFailure = clock + np.ceil(6*np.random.random())

    else: # S==2
        NextRepair = float('inf')

def Timer():
    global clock
    global NextRepair
    global NextFailure

    if NextEndSimulation < NextFailure and NextEndSimulation < NextRepair:
        result = "EndSimulation"
        clock = NextEndSimulation

    elif NextFailure < NextRepair:
        result = "Failure"
        clock = NextFailure

    else:
        result = "Repair"
        clock = NextRepair
    return result


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

clock = 0
S = 2
# system is available at time 0
A = 1
```

```python
# initialize the time of events
NextRepair = float('inf')
NextFailure = np.ceil(6*np.random.random())
NextEndSimulation = 1000
# Define variables to keep the area under the sample path
# and the time and state of the last event
Area = 0.0
Tlast = 0
Alast = 1
NextEvent = Timer()

while NextEvent!="EndSimulation":
    NextEvent = Timer()
    if NextEvent == "Repair":
        Repair()
    elif NextEvent == "Failure":
        Failure()
    else:
        EndSimulation()

print('Average system availability till time T = 1000:', Area/clock)
```

```
Average system availability till time T = 1000:  0.9035
```

(c) Obtain the above estimates until $T = 3000$ and compare them with the estimates from part (a) and (b). Summarize your observation in one sentence.

(10 points) For $T = 3000$ we get

$$\bar{S}(T) = \frac{1}{3000}\int_0^{3000} S(t)dt = 1.2697,$$

and

$$\bar{A}(T) = \frac{1}{3000}\int_0^{3000} A(t)dt = 0.9137.$$

We observe that the results of part (a) and (b) are approximately the same with those in part (c) suggesting that the time averages are converging to some constants θ_1 and θ_2, which are the long-run average number of functioning components and the long-run average system availability, respectively:

$$\theta_1 = \lim_{T\to\infty} \frac{1}{T}\int_0^{T} S(t)dt.$$

$$\theta_2 = \lim_{T\to\infty} \frac{1}{T}\int_0^{T} A(t)dt.$$

(See also Page 5 of the textbook).

HINT: To get the simulation to stop at time T use the functional version of the code and create a new event called EndSimulation, with a variable NextEndSimulation as part of the event calendar. Note that you also need to modify the logic of the model in the failure and repair event functions.

Problem 3. **(30 Pts.)** Modify the TTF simulation so that instead of 2 components there can be any number of components. The number of components N should be an input of the simulation model. As before, assume that one component is active and the rest are kept as spares. Also, only one component can be repaired at a time. Run your simulation for 1000 replications and report a 95% confidence interval for the expected time to failure of the system when $N = 2, 3$, and 4.

(30 points) With N components the logic is modified as follows. When a failure happens, we have $S = N - 1$ or $S < N - 1$. If $S = N - 1$, then a new component starts working and a new repair begins. Therefore, we need to schedule both the next failure and repair events. If $S < N - 1$ and $S \neq 0$, then a repair is already in progress and therefore we only schedule the next failure for the component that just started working. If $S = 0$, then the system fails. When a repair is completed we have $S = N$ or $S < N$. If $S = N$, then a failure is still pending and all components are functioning. Therefore we only need to set the next repair time to ∞. If $S < N$, again a failure is still pending but a new repair starts which we need to schedule.

Based on 1000 replications the 95% CI for the expected time to failure of the system when $N = 2, 3$, and 4 is $[13.75, 15.15], [104.75, 117.68]$ and $[994.30, 1117.36]$, respectively.

```python
# The TTF example with multiple components; still 1 repair at a time
import numpy as np
from scipy import stats

comp = 4 # number of components available
Ylist = [] #  keeps samples of time to failure
Avelist = [] # keeps samples of average # of func. components
np.random.seed(1)

def mean_confidence_interval_95(data):
    a = 1.0*np.array(data)
    n = len(a)
    m, se = np.mean(a), stats.sem(a)
    h = 1.96*se
    return m, m-h, m+h

for reps in range(1000):
        # initialize clock, next events, state
        clock = 0
        S = comp
        NextRepair = float('inf')
        NextFailure = np.random.choice([1,2,3,4,5,6], 1)
        # define variables to keep the last state and time, and the area under the sample path
        Slast = S
        Tlast = clock
        Area = 0

        while S > 0: # While system is functional
                # Determine the next event and advance time
                clock = np.min([NextRepair, NextFailure])
                event = np.argmin([NextRepair, NextFailure])
                if event == 0: # Repair
                        S += 1
                        if S == 1: # this would never be the case for the current while loop
                                NextRepair = clock + 2.5
```

```
                NextFailure = clock + np.random.choice([1,2,3,4,5,6], 1)
            if S < comp: # a new repair starts
                NextRepair = clock + 2.5
            if S == comp: # all components are functional
                NextRepair = float('inf')

        else: # Failure
            S -= 1
            if S == comp - 1: # A new component starts working; a new repair starts
                NextRepair = clock + 2.5
                NextFailure = clock + np.random.choice([1,2,3,4,5,6], 1)
            else: # a new component starts working
                NextFailure = clock + np.random.choice([1,2,3,4,5,6], 1)

        # update the area udner the sample path
        Area = Area + Slast * (clock - Tlast)
        # record the current time and state
        Slast = S
        Tlast = clock
    # add samples to the lists
    Ylist.append(clock)
    Avelist.append(Area/clock)

# print the estimates with a 95% confidence interval
print('The estimate and 95% CI for the expected time to failure:',
    mean_confidence_interval_95(Ylist))
```

```
The estimate and 95% CI for the expected time to failure:
(1055.831, 994.3019475153425, 1117.3600524846572)
```

Problem 4. (5 Pts.) The standard error of an estimator is defined as the standard deviation of that estimator. In the lecture, we introduced the sample mean $\bar{X}_n = (1/n)\sum_{i=1}^{n} X_i$ as an estimator of $E[X]$ where X_i's are iid samples of the random variable X. What is the standard error of the estimator $\bar{X}_n$? Assume that the standard deviation of X is σ.

(5 points) The variance of the estimator is given by

$$Var(\bar{X}_n) = Var(\frac{1}{n}\sum_{i=1}^{n} X_i) = \frac{1}{n^2} nVar(X_1) = \frac{1}{n}Var(X_1).$$

Therefore, the standard deviation is

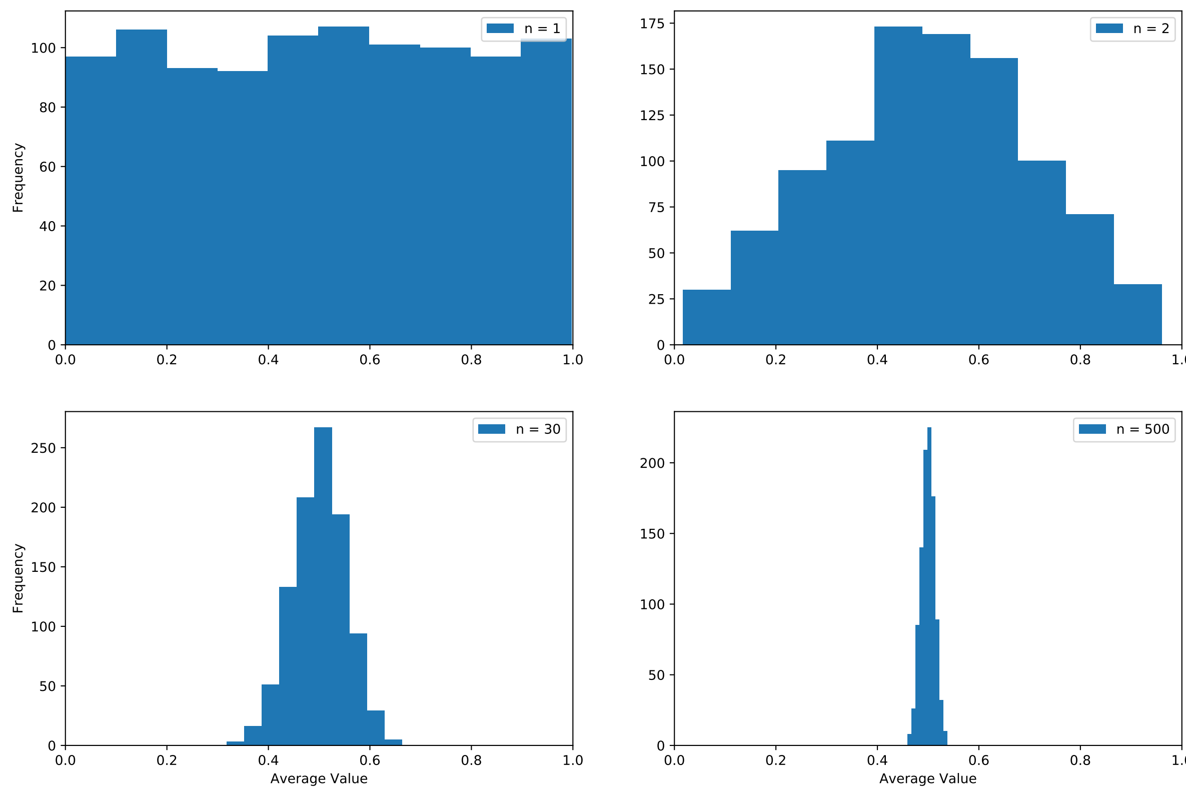
$$\sqrt{Var(\bar{X}_n)} = \sqrt{\frac{1}{n}Var(X_1)} = \frac{\sigma}{\sqrt{n}}.$$

Problem 5. (15 Pts.) Assume that $\{X_i; i \geq 1\}$ is a sequence of independent uniform random variables between $(0,1)$, i.e., $X_i \sim \text{Unif}(0,1)$ for all $i \geq 1$. Consider the sample average of n such random variables,

$$\bar{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i.$$

(a) Simulate 1000 samples of $\bar{X}_1, \bar{X}_2, \bar{X}_{30}, \bar{X}_{500}$ and plot a histogram of the samples for each case (i.e. 4 histograms in total). What happens to the mean and variability of the samples as n increases? Relate your observations to the Central Limit Theorem discussed in the class. **Note**: You can plot a histogram in Python using the `plt.hist` method of the `matplotlib.pyplot` library.

(15 points) The mean of the samples remains at 0.5 while the variability in samples decreases as n increases. The shape of the histogram becomes more symmetric and bell-shaped. This is consistent with the Central Limit Theorem which predicts that the distribution of sample average is approximatley normal with mean $E[X_1]$ and variance $\sigma/\sqrt{n}$.




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

np.random.seed(1)

# n = [1, 2, 30, 500]
def Xbar (n):
    # Keep samples of average
    Avelist = []
    for i in range (1000):
        Avelist.append(np.mean(np.random.random(n)))
```

```
    return Avelist

plt.figure(figsize = (15, 12))

# For n = 1
plt.subplot(221)
plt.hist(Xbar(1), label = 'n = ' + str(1))
plt.xlim(xmin=0, xmax=1)
plt.ylabel('Frequency')
plt.legend()

# For n = 2
plt.subplot(222)
plt.hist(Xbar(2), label = 'n = ' + str(2))
plt.xlim(xmin=0, xmax=1)
plt.legend()

# For n = 30
plt.subplot(223)
plt.hist(Xbar(30), label = 'n = ' + str(30))
plt.xlim(xmin=0, xmax=1)
plt.xlabel('Average Value')
plt.ylabel('Frequency')
plt.legend()

# For n = 500
plt.subplot(224)
plt.hist(Xbar(500), label = 'n = ' + str(500))
plt.xlim(xmin=0, xmax=1)
plt.xlabel('Average Value')
plt.legend()

plt.show()
```