Technical University of Denmark

02443 Stochastic Simulation

s163927, Virgile Blanchet-Møhl
s184012, Simon Majgaard

Assignments

June 21, 2022

Exercise 1

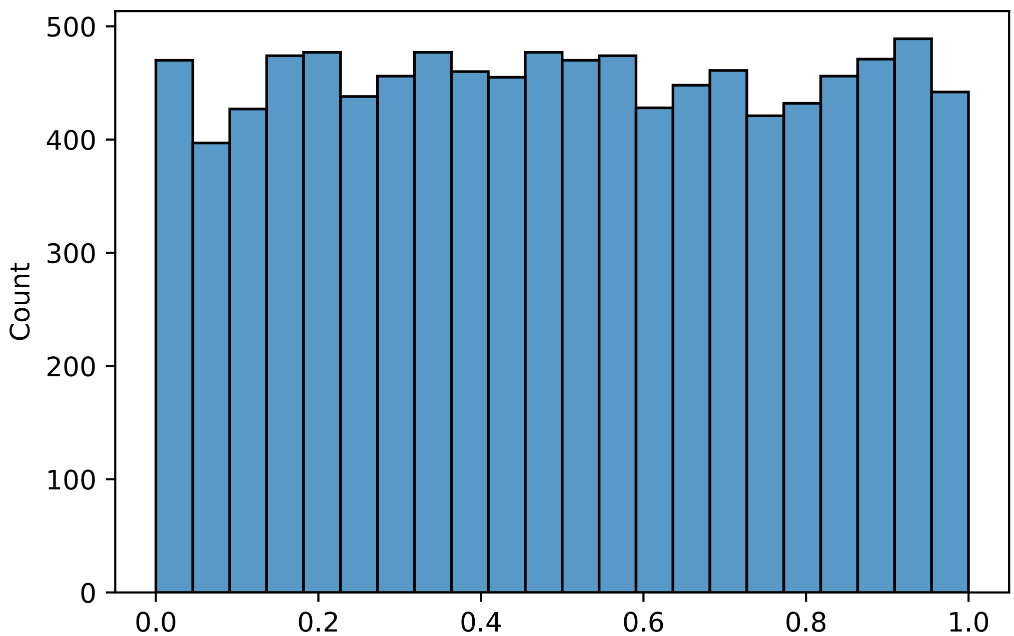
In [ ]:
```
%load_ext autoreload
%autoreload 2
```

In [ ]:
```
from src.my_random.tests import *
from src.my_random.gen import *
import scipy.stats as stats
```

Good example compared to Scipy's uniform generation

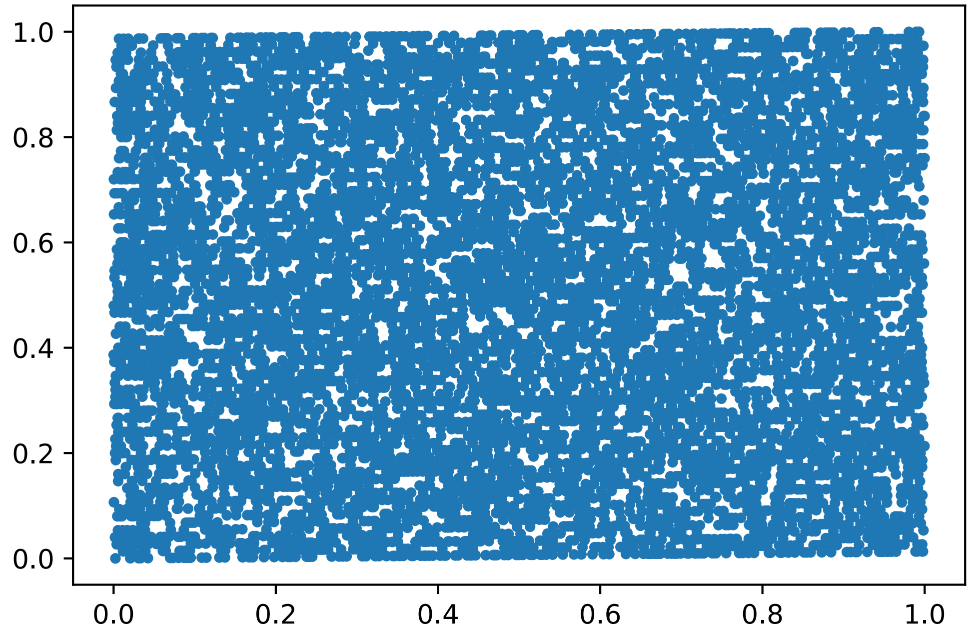
In [ ]:
```
u_lcg = [k for k in lcg(M=2**16+1, a=75, c=74, n=10_000, x=10)]
sns.histplot(u_lcg)
```

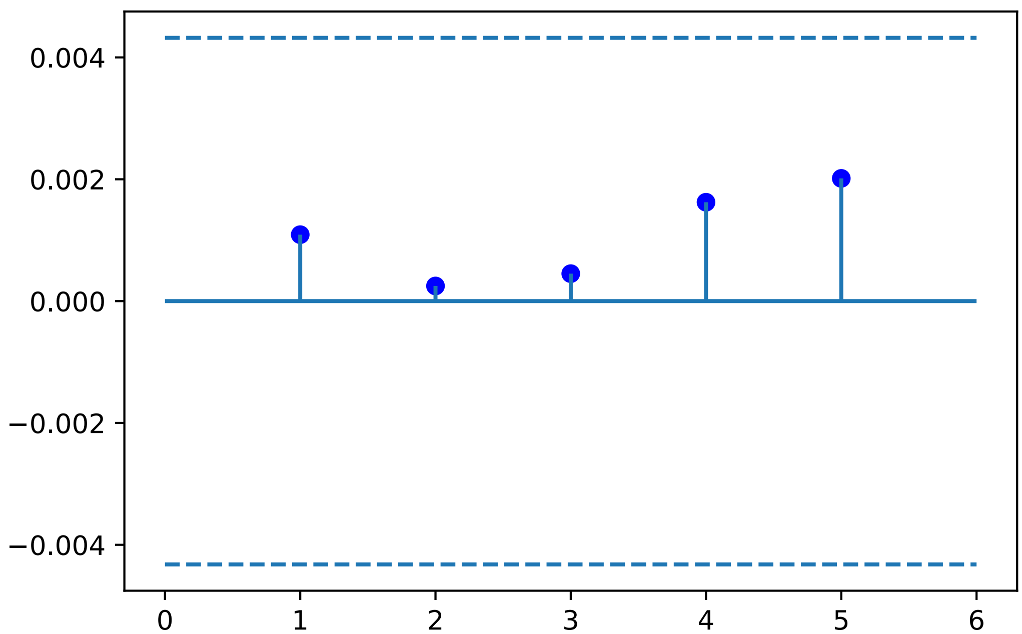
Out[ ]:
```
<AxesSubplot:ylabel='Count'>
```



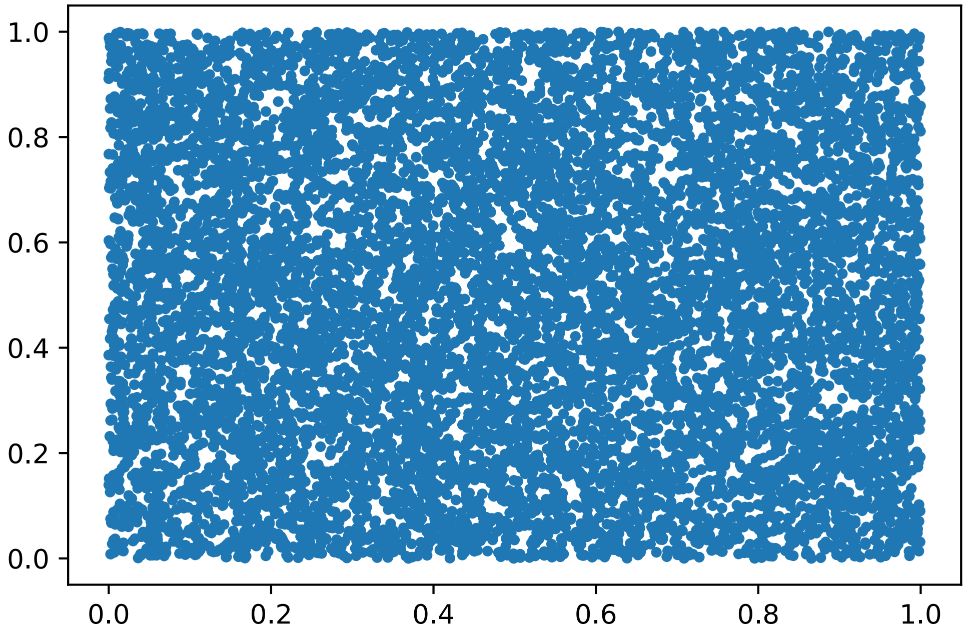
In [ ]:
```
u_scipy = stats.uniform.rvs(size=10_000)
all_test(np.array(u_lcg))
all_test(u_scipy)

# fig, ax = plt.subplots(1, 2)
# sns.histplot(u_lcg, ax=ax[0])
# sns.scatterplot(x = u_lcg[1:], y = u_lcg[:-1], ax=ax[1])
```

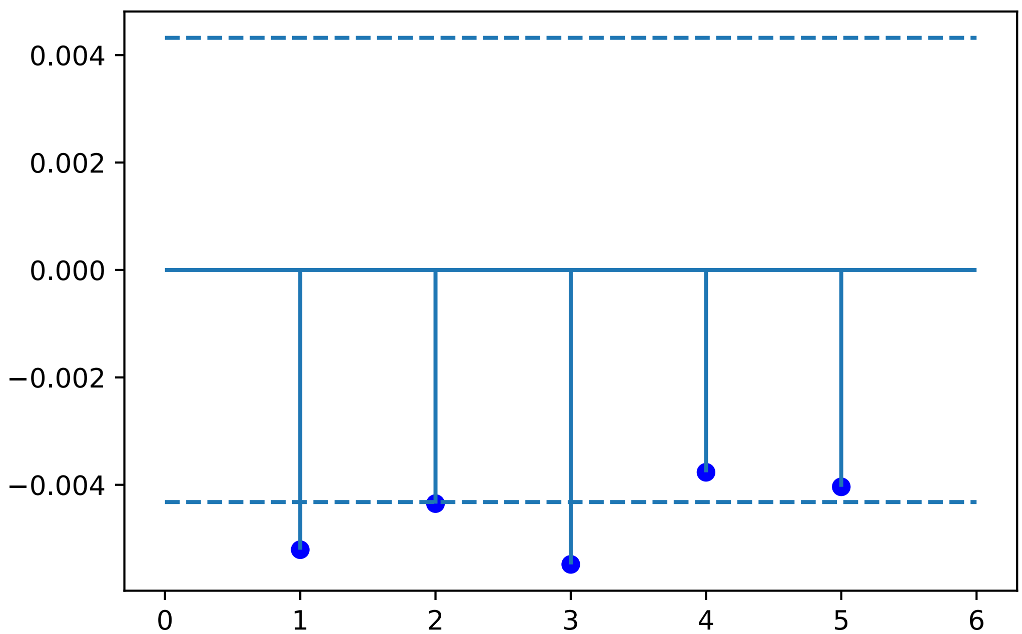
```
5001.0 2499.7499749975 5011
____________Uniform Distribution Tests____________
Chi^2 test with 100 groups:                  p=1.00
Kolmogorov Smirnof:                          T=7.33
_______________Independence Tests________________
Run Test 1: Above/below Median:              p=0.84
Run Test 2: Up/Down length count Test:       p=0.48
Run Test 3: Up/Down run count Test:          p=0.96
```






```
5001.0 2499.7499749975 5014
____________Uniform Distribution Tests___________
Chi^2 test with 100 groups:                 p=0.07
Kolmogorov Smirnof:                         T=7.32
_______________Independence Tests________________
Run Test 1: Above/below Median:             p=0.79
Run Test 2: Up/Down length count Test:      p=0.13
Run Test 3: Up/Down run count Test:         p=0.63
```





Out[ ]:

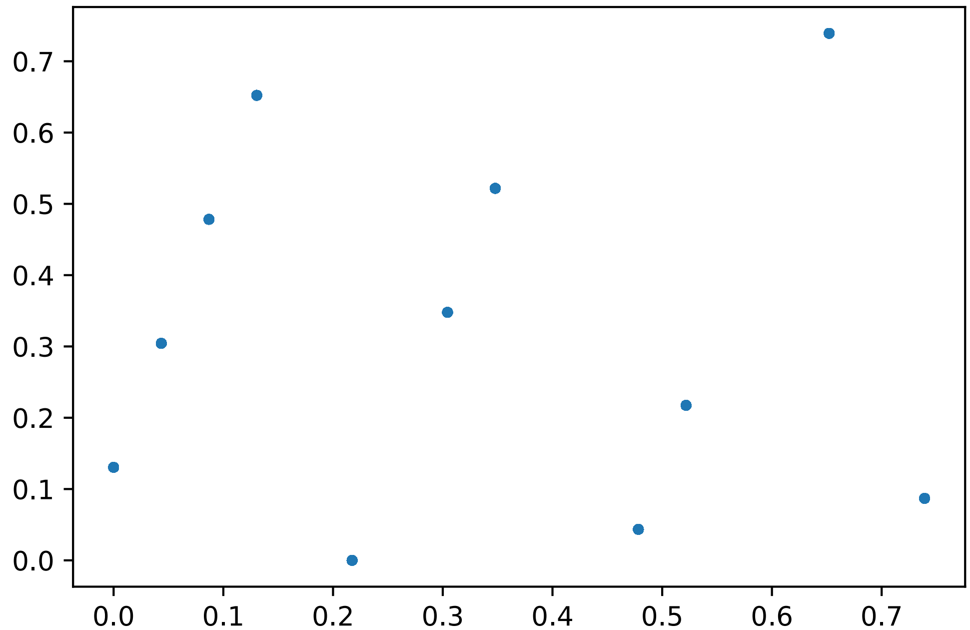
```
(0.06879168564234694,
 7.320575964274654,
 0.7948537440906605,
 0.126684966176001,
 0.6295992023085439)
```

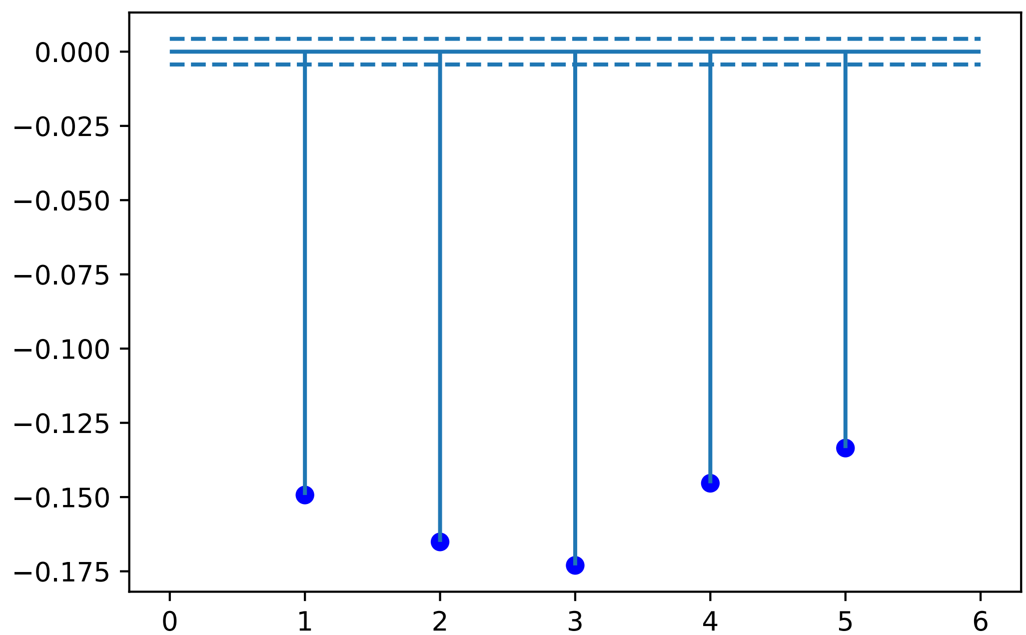
Bad Example

In [ ]:

```
u_lcg = [k for k in lcg(M=23, a=75, c=74, n=10_000, x=1)]
all_test(np.array(u_lcg))
```

```
4546.49994500055 2272.4999174978 5455
____________Uniform Distribution Tests___________
Chi^2 test with 100 groups:                 p=0.00
Kolmogorov Smirnof:                         T=4.17
_______________Independence Tests________________
Run Test 1: Above/below Median:             p=0.00
Run Test 2: Up/Down length count Test:      p=0.00
Run Test 3: Up/Down run count Test:         p=0.00
```





Out[ ]:
```
(0.0, 4.174312922730702, 0.0, 0.0, 0.0)
```

In general you would probably need to perform the tests multiple times, since the random number will lie outside the confidence interval about 5% of the time if it was truly random.

Exercise 2

In [ ]:
```
%load_ext autoreload
%autoreload 2
```

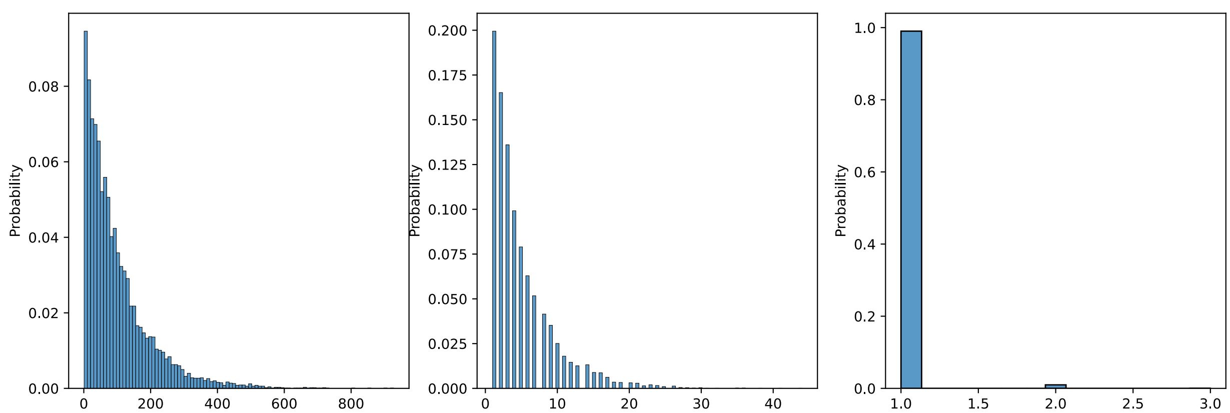
In [ ]:
```
from src.my_random.gen import *
from src.my_random.tests import chi2
import matplotlib.pyplot as plt
import seaborn as sns
import scipy.stats as stats
```

1)

In [ ]:
```
small_geom = geometric(0.01, 10_000)
medium_geom = geometric(.2, 10_000)

big_geom = geometric(.99, 10_000)
```

In [ ]:
```
fig, ax = plt.subplots(1,3, figsize=(15, 5))
sns.histplot(small_geom, ax=ax[0], stat='probability')
sns.histplot(medium_geom, ax=ax[1],stat='probability')
sns.histplot(big_geom, ax=ax[2],stat='probability')
```

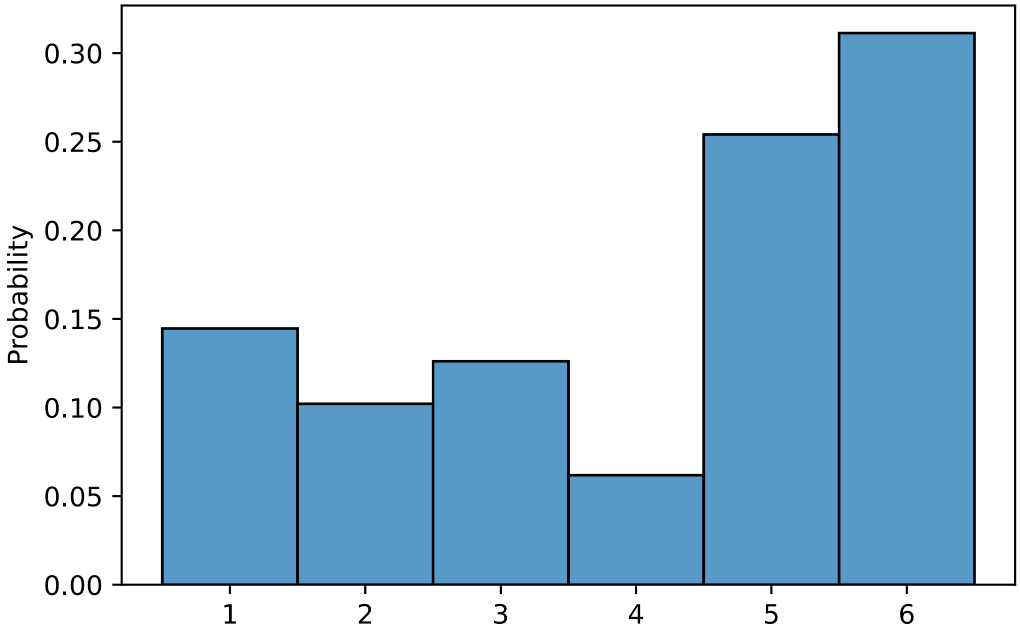
Out[ ]:
```
<AxesSubplot:ylabel='Probability'>
```



Ex2)

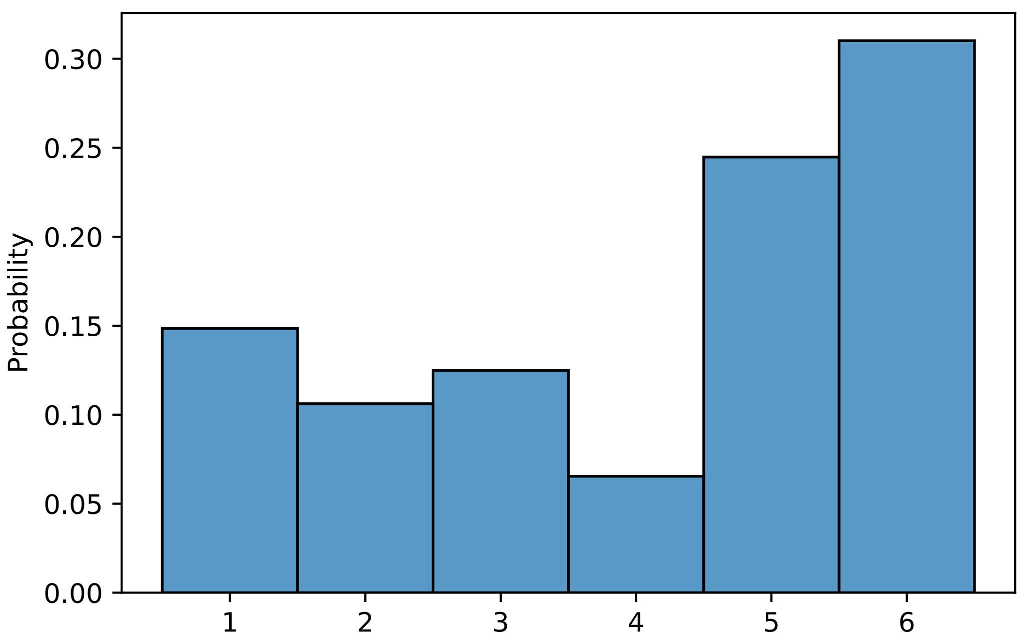
In [ ]:
```
p = [7/48, 5/48, 1/8, 1/16, 1/4, 5/16]
crude = discrete_crude(p, 10_000)
sns.histplot(crude, stat='probability', discrete=True)
```

Out[ ]:
```
<AxesSubplot:ylabel='Probability'>
```



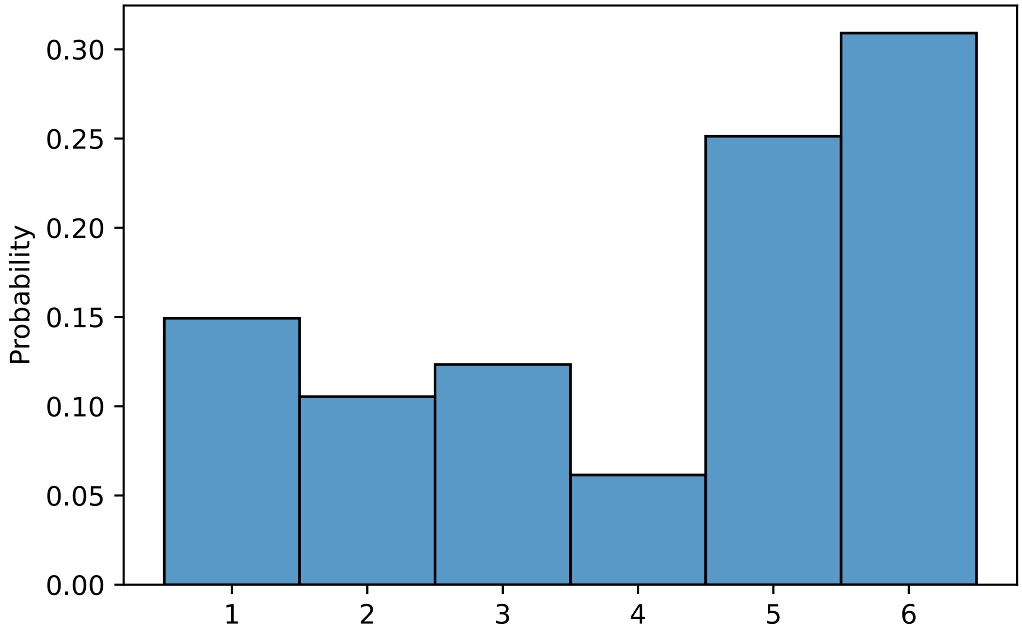

```
In [ ]: rej = discrete_rejection(p, 10_000)
        sns.histplot(rej, stat='probability', discrete=True)
```

```
Out[ ]: <AxesSubplot:ylabel='Probability'>
```




```
In [ ]: alias = discrete_alias(p, 10_000)
        sns.histplot(alias, stat='probability', discrete=True)
        p
```

```
[3 4]
[3 4 5]
Out[ ]: [0.14583333333333334, 0.10416666666666667, 0.125, 0.0625, 0.25, 0.3125]
```



Ex3)

In [ ]:
```
print(chi2(np.unique(alias, return_counts=True)[1], np.array(p)*10000),
chi2(np.unique(rej, return_counts=True)[1], np.array(p)*10_000),
chi2(np.unique(crude, return_counts=True)[1], np.array(p)*10_000))
```

```
0.8796523475911306 0.626142426234333 0.9234349066060533
```

All of the methods produce a chi squred p-value well within the confidence of 95%. If at all possible and not too hard to find analytically, the crude method is the way to go. It is computationally inexpensive compared to the other methods and easy to set up. \ The rejection method is very easy to setup, almost no matter how complex the system. However, if some of the categories are very unlikely, a lot of the samples will be rejected which would mean a lot of wasted computational power. \ To fix all these rejctions, the alias method is the way to go. However, the setup of this method requires computations as well. That means, that if you are only gonna need a small sample a couple of times, it may not be worth it computationally.

In [ ]:
```
stats.chisquare(np.unique(rej, return_counts=True)[1], np.array(p)*10_000)[1]
```

Out[ ]:
```
0.626142426234333
```

In [ ]:
```

```

Exercise 3

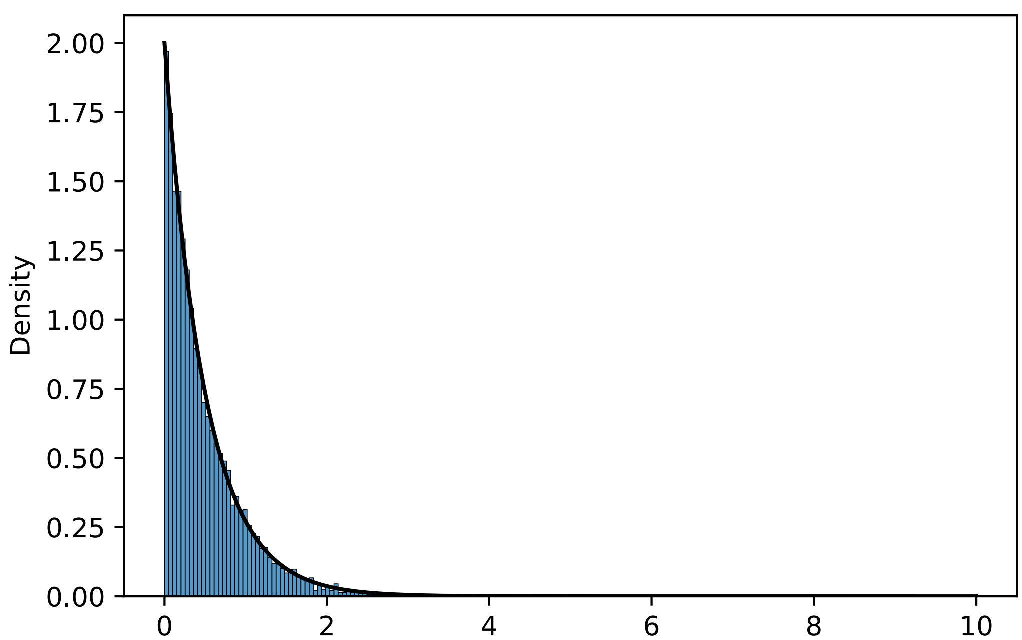
In [ ]:
```
%load_ext autoreload
%autoreload 2
```

In [ ]:
```
from src.my_random.gen import *
from src.my_random.tests import chi2, kolmogorov, emperical_dist
import matplotlib.pyplot as plt
import seaborn as sns

import scipy.stats as stats
import pandas as pd
```

Exponential Distribution

In [ ]:
```
lmbda = 2
exps = exponential(lmbda, 10_000)
h = sns.histplot(exps, stat='density')
x = np.linspace(0, 10, 1000)
sns.lineplot(x=x, y=stats.expon.pdf(x, scale=1/lmbda), color='k')
kolmogorov(exps, stats.expon())
```

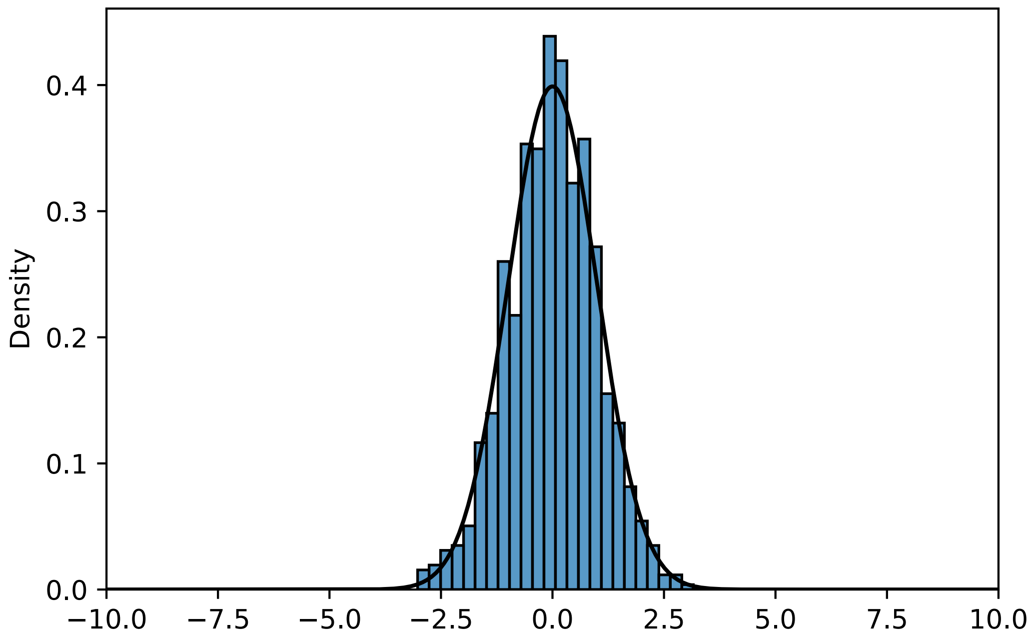
Out[ ]:
```
79.10421538962612
```



Normal Distribution

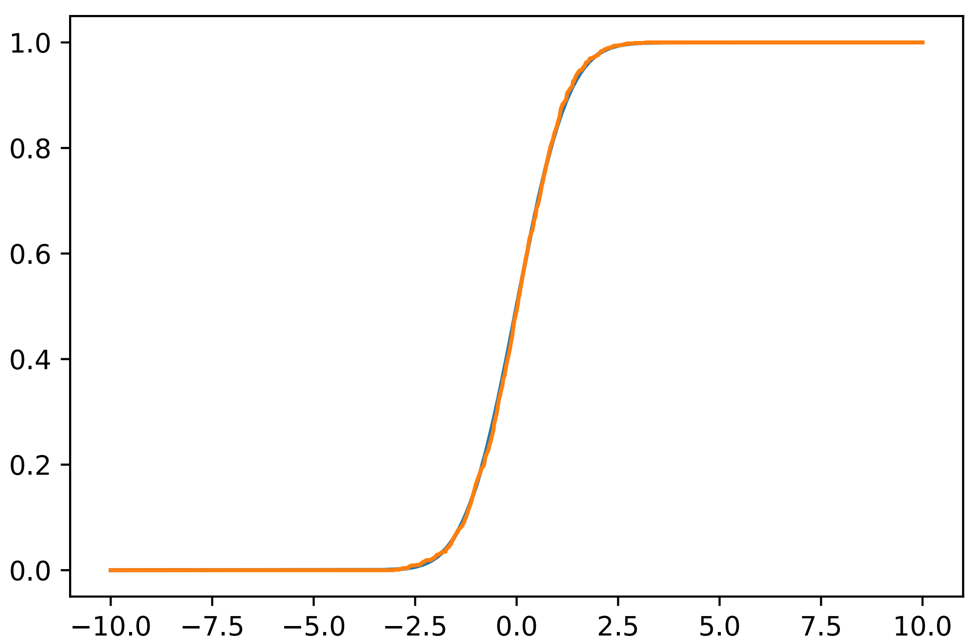
In [ ]:
```
norms = norm_box_mueller(1000)
h = sns.histplot(norms, stat='density')
x=np.linspace(-10,10,1000)
sns.lineplot(x=x, y=stats.norm.pdf(x), color='k')
h.set(xlim=(-10, 10))
```

Out[ ]: [(-10.0, 10.0)]



In [ ]:
```
sns.lineplot(x=x, y=stats.norm.cdf(x))
sns.lineplot(x=x, y=[emperical_dist(i, norms) for i in x])
kolmogorov(norms, stats.norm(), (-10, 10))
```

Out[ ]: (-62.65744413894945, -1.9736956037033961)

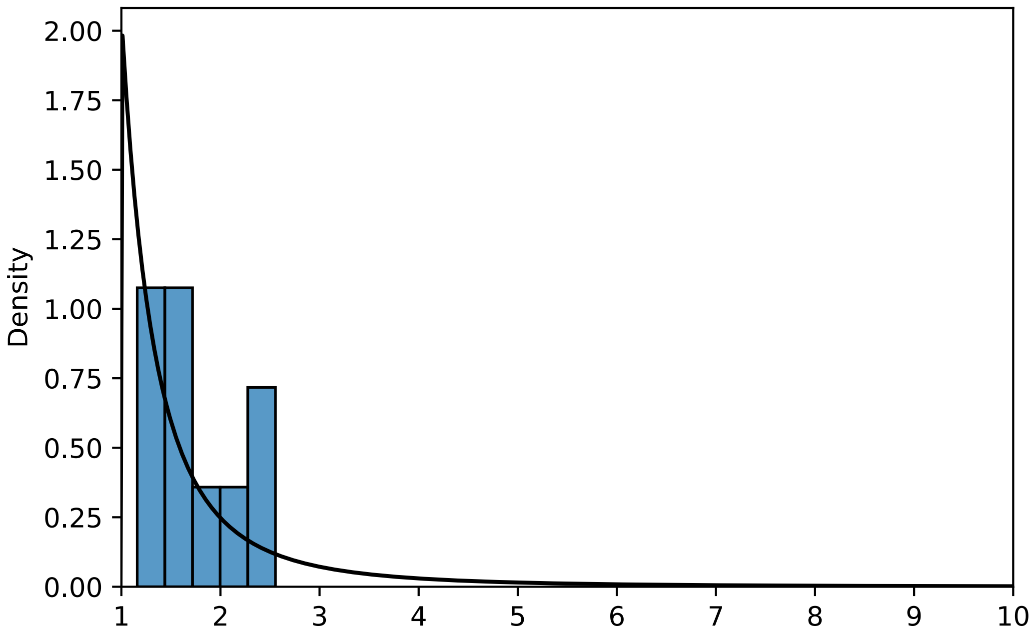


Pareto

In [ ]:
```
paretos = pareto(2.05, 1, 10)
h = sns.histplot(paretos, stat='density')
x=np.linspace(-10,10,1000)
sns.lineplot(x=x, y=stats.pareto.pdf(x, b=2.05, scale=1), color='k')
```

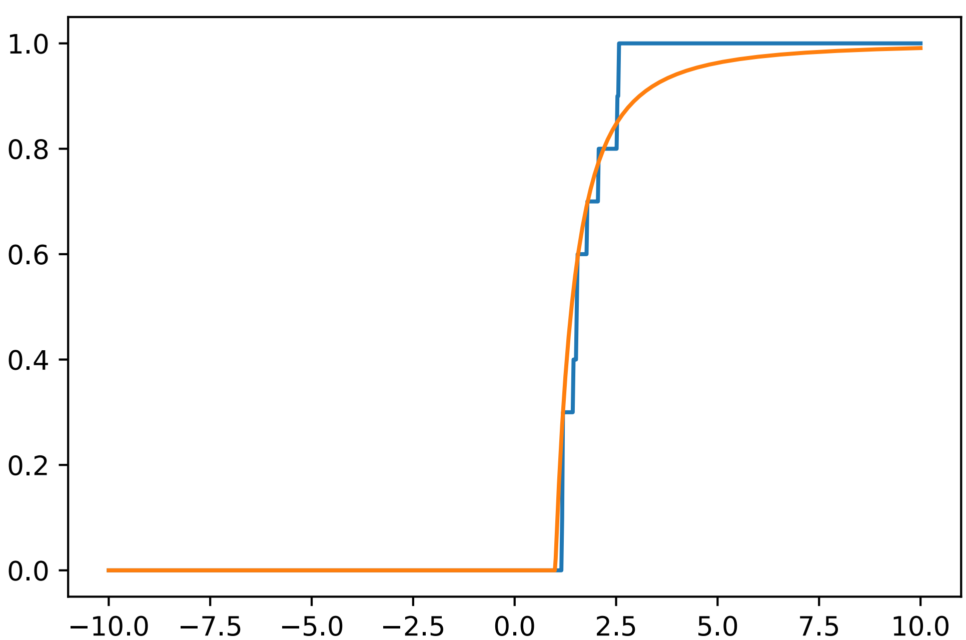
```
h.set(xlim=(1, 10))
kolmogorov(paretos, dist=stats.pareto(b=1, scale=2.05), range=(1,1e4))
```

Out[ ]: 6.842882392759879



In [ ]:
```
sns.lineplot(x = x, y = [emperical_dist(i, paretos) for i in x] )
sns.lineplot(x=x, y=stats.pareto(b=2.05, scale=1).cdf(x))
```

Out[ ]: <AxesSubplot:>



In [ ]:
```
ks = [2.05, 2.5, 3, 4]
df = pd.DataFrame({k: pareto(k, 1, 10) for k in ks})
```

In [ ]:
```
obs_stats =  df.aggregate(['mean', 'var'])
```

In [ ]:
```
def mean_pareto(beta, k):
    return beta*k/(k-1)
```

```
def var_pareto(beta, k):
    return beta**2*k/((k-1)**2 * (k-2))


true_means = [mean_pareto(1, i) for i in ks]
true_vars = [var_pareto(1, i) for i in ks]
true_stats = pd.DataFrame({'mean': true_means, 'var': true_vars})
true_stats.T
```

Out[ ]:

	0	1	2	3
mean	1.952381	1.666667	1.50	1.333333
var	37.188209	2.222222	0.75	0.222222

In [ ]:

```
obs_stats
```

Out[ ]:

	2.05	2.50	3.00	4.00
mean	1.797133	1.764837	1.329337	1.285407
var	0.815427	1.253118	0.426290	0.073788

Comparing the means and vars, we see that the estimates are are more off the smaller k are. Furthermore, we notice that the estimates of the means are way better than the variance estimates.

In [ ]:

```
norms = np.stack([norm_box_mueller(10) for _ in range(100)])
t = np.array(stats.t.interval(.95, 9))
t_confs = np.stack([t*(row.std()/np.sqrt(10)) + row.mean() for row in norms])

chi = np.array(stats.chi2.interval(.95, 9))
chi_confs = np.stack([9*row.var() / chi[::-1] for row in norms])

conf = norms.mean(axis=1) + np.array([-1.96*norms.std(axis=1), 1.96*norms.std(

mean_df = pd.DataFrame({'lwr': t_confs[:,0], 'mean':norms.mean(1), 'upr':t_con
var_df = pd.DataFrame({'lwr': chi_confs[:,0], 'var':norms.var(1), 'upr':chi_co
```

In [ ]:

```
mean_df.describe()
```

Out[ ]:

	lwr	mean	upr
count	100.000000	100.000000	100.000000
mean	-0.688873	-0.011229	0.666416
std	0.311284	0.261927	0.286284
min	-1.368979	-0.804567	-0.250260
25%	-0.925813	-0.183167	0.469010
50%	-0.687262	-0.048167	0.658813
75%	-0.493565	0.198056	0.850182
max	0.119087	0.665376	1.401509

In [ ]:
```
var_df.describe()
```

Out[ ]:

	lwr	var	upr
count	100.000000	100.000000	100.000000
mean	0.443607	0.937625	3.124967
std	0.187871	0.397091	1.323446
min	0.120146	0.253945	0.846361
25%	0.328122	0.693532	2.311441
50%	0.407786	0.861913	2.872629
75%	0.551490	1.165652	3.884945
max	1.201059	2.538607	8.460802

We see that the confidence intervals vary quite a lot both for the mean and the variance. In the extreme case of the mean, 0 is not even in the confidence interval, which in a lot of experiments would mean we would have concluded a statistical signficant result, even though this is gaussian noise.\ The variance has non-symmetrical confidence intervals. We see that especially the upper bound of the variance has a high standard deviation. This is to be expected, since we only have 10 observations

In [ ]:
```

```

Exercise 4

In [ ]:
```
%load_ext autoreload
%autoreload 2
```

In [ ]:
```
from src.my_random.event import BlockingEventSimulation, calculate_theoretical_
from scipy import stats
from dataclasses import dataclass
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

In [ ]:
```
def find_blocked_w_conf(sim: BlockingEventSimulation):
    blocked = []
    for i in range(10):
        blocked.append(sim.simulate(10_000, 10))

    mean = np.mean(blocked)
    sd = np.std(blocked)
    lwr, upr = stats.t.interval(0.95, 9)
    conf = [mean + sd/np.sqrt(10)*lwr, mean + sd/np.sqrt(10)*upr]

    return mean, conf
```

1. Poisson Process

In [ ]:
```
arr_dist = stats.expon()
serv_dist = stats.expon(scale=8)
pois_sim = BlockingEventSimulation(arr_dist, serv_dist)
blocked = []
for i in range(10):
    blocked.append(pois_sim.simulate(10_000, 10))
```

In [ ]:
```
find_blocked_w_conf(pois_sim)
```

Out[ ]:
```
(0.11945000000000001, [0.11576063734857538, 0.12313936265142465])
```

In [ ]:
```
calculate_theoretical_block_pct(10, 8)
```

Out[ ]:
```
0.12166106425295149
```

2. Renewal Processes

In [ ]:
```
@dataclass
class hyper_exp:
    p1: float
    p2: float
    lmbda1: float
    lmbda2: float
```

```
    def rvs(self, size):
        return self.p1 * stats.expon.rvs(size=size, scale=1/self.lmbda1) \
           + self.p2*stats.expon.rvs(size=size, scale = 1/self.lmbda2)
```

In [ ]:
```
arr_erl = stats.erlang(a=1)
arr_hyp = hyper_exp(0.8, .2, .8333, 5.0)
serv_dist = stats.expon(scale=8)
```

In [ ]:
```
sim_erl = BlockingEventSimulation(arr_erl, serv_dist)
sim_hyp = BlockingEventSimulation(arr_hyp, serv_dist)
```

In [ ]:
```
```

Out[ ]:
```
1.0
```

Erlang arrival times

In [ ]:
```
blocked = []
for i in range(10):
    blocked.append(sim_erl.simulate(10_000, 10))

mean = np.mean(blocked)
sd = np.std(blocked)
lwr, upr = stats.t.interval(0.95, 9)
conf = [mean + sd/np.sqrt(10)*lwr, mean + sd/np.sqrt(10)*upr]

mean, conf
```

Out[ ]:
```
(0.11786, [0.11363500739572827, 0.12208499260427175])
```

Hyper Exponential Arrival Times

In [ ]:
```
blocked = []
for i in range(10):
    blocked.append(sim_hyp.simulate(10_000, 10))

mean = np.mean(blocked)
sd = np.std(blocked)
lwr, upr = stats.t.interval(0.95, 9)
conf = [mean + sd/np.sqrt(10)*lwr, mean + sd/np.sqrt(10)*upr]

mean, conf
```

Out[ ]:
```
(0.11591, [0.11201390464069531, 0.11980609535930468])
```

3.) Service Distributions

In [ ]:
```
@dataclass
class constant_service_time:
    mean_time: float
    def rvs(self, size):
        return np.array([self.mean_time]*size)
```

```python
def pareto_mean_service(k, mean_time):
    scale = (k-1)*mean_time / k
    return stats.pareto(b = k, scale=scale)

arr_dist = stats.expon()
serv_const = constant_service_time(8)
serv_par_105 = pareto_mean_service(1.05, 8)
serv_par_205 = pareto_mean_service(2.05, 8)

const_sim = BlockingEventSimulation(arr_dist, serv_const)
par_105_sim = BlockingEventSimulation(arr_dist, serv_par_105)
par_205_sim = BlockingEventSimulation(arr_dist, serv_par_205)
```

In [ ]:
```python
def find_blocked_w_conf(sim: BlockingEventSimulation):
    blocked = []
    for i in range(10):
        blocked.append(sim.simulate(10_000, 10))

    mean = np.mean(blocked)
    sd = np.std(blocked)
    lwr, upr = stats.t.interval(0.95, 9)
    conf = [mean + sd/np.sqrt(10)*lwr, mean + sd/np.sqrt(10)*upr]

    return mean, conf
```

In [ ]:
```python
find_blocked_w_conf(const_sim)
```

Out[ ]:
```
(0.12015, [0.1175988222340343, 0.12270117776596572])
```

In [ ]:
```python
find_blocked_w_conf(par_105_sim)
```

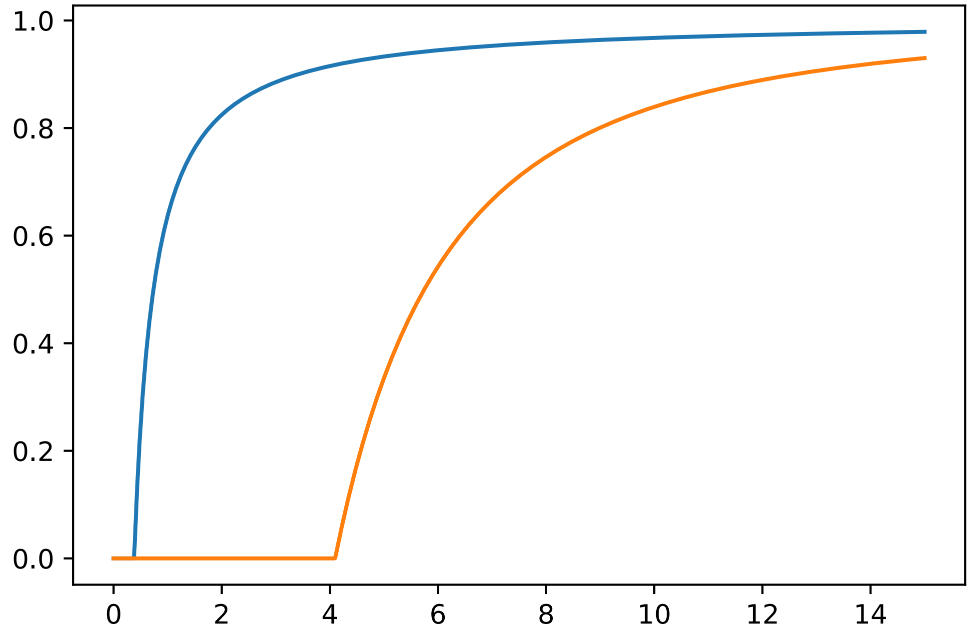
Out[ ]:
```
(0.0009299999999999999, [0.00029172683445569873, 0.0015682731655443012])
```

In [ ]:
```python
find_blocked_w_conf(par_205_sim)
```

Out[ ]:
```
(0.12036, [0.1141903877860545, 0.12652961221394549])
```

In [ ]:
```python
x = np.linspace(0,15,1000)

sns.lineplot(x=x, y=serv_par_105.cdf(x))
sns.lineplot(x=x, y=serv_par_205.cdf(x))
```

Out[ ]:
```
<AxesSubplot:>
```



In [ ]:
```
serv_par_105.median(), serv_par_105.mean()
```

Out[ ]:
```
(0.7371670693515371, 8.0)
```

Even though the mean time of the 2 pareto distributions are the same, the probability mass of the k=1.05 distribution is heavily weighted towards the beginning. i.e. the median is way to the left of the mean. Therefore, most of the costumers would be serviced very quickly, and the blocked costumers very low. Only with a huge simulation, the true amount of blocked costumers will appear.

Exercise 5: Variance reduction methods

In [ ]:
```
import scipy.stats as stats
from src.my_random import gen
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd
from scipy import random


def func(x):
    return np.exp(x)
```

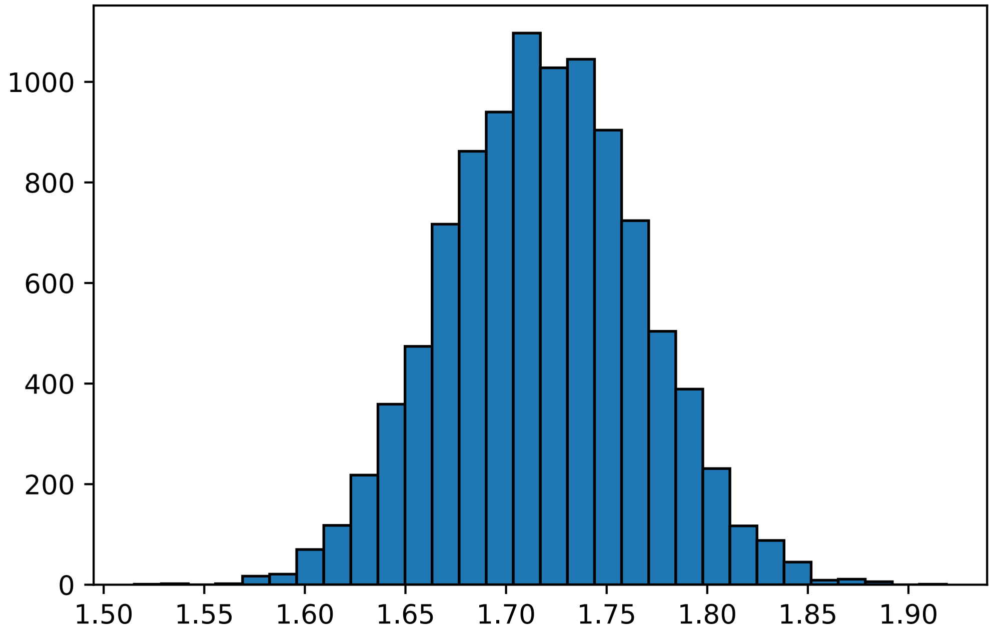
1.

In [ ]:
```
N = 100
runs = 10000
areas = []

for i in range(runs):
    xrand = stats.uniform.rvs(size=N)
    areas.append(np.mean(func(xrand)))

m = np.mean(areas)
s = np.std(areas)
dof = N-1
conf = 0.95

t = np.abs(stats.t.ppf((1-conf)/2,dof))
confInt = (m-s*t/np.sqrt(N),m+s*t/np.sqrt(N))
print('The point estimate of the crude Monte Carlo estimator is: ',m)
print('While the confidence interval at 95%','confidence is:',confInt)
```

```
The point estimate of the crude Monte Carlo estimator is:  1.7184951278504745
While the confidence interval at 95% confidence is: (1.7088397692667987, 1.728
1504864341504)
```

In [ ]:
```
plt.hist(areas, bins=30, ec= 'black')
```

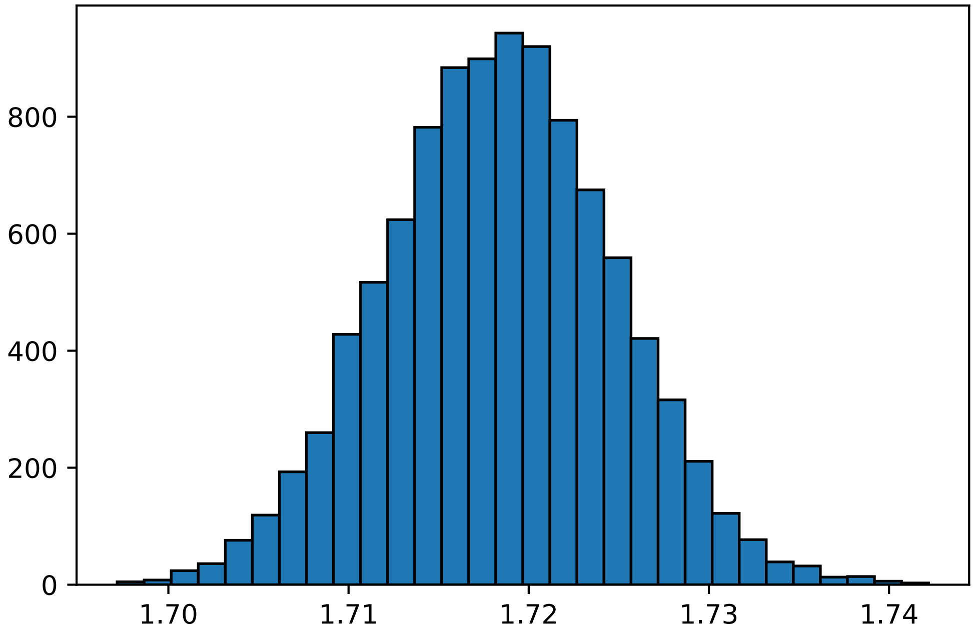
Out[ ]:
```
(array([1.000e+00, 2.000e+00, 0.000e+00, 2.000e+00, 1.700e+01, 2.100e+01,
        7.000e+01, 1.180e+02, 2.180e+02, 3.590e+02, 4.740e+02, 7.170e+02,
        8.620e+02, 9.400e+02, 1.097e+03, 1.028e+03, 1.045e+03, 9.040e+02,
        7.240e+02, 5.040e+02, 3.890e+02, 2.310e+02, 1.170e+02, 8.800e+01,
        4.500e+01, 9.000e+00, 1.100e+01, 6.000e+00, 0.000e+00, 1.000e+00]),
 array([1.51519593, 1.52865357, 1.54211121, 1.55556886, 1.5690265 ,
        1.58248414, 1.59594179, 1.60939943, 1.62285707, 1.63631471,
        1.64977236, 1.66323   , 1.67668764, 1.69014528, 1.70360293,
        1.71706057, 1.73051821, 1.74397586, 1.7574335 , 1.77089114,
        1.78434878, 1.79780643, 1.81126407, 1.82472171, 1.83817935,
        1.851637  , 1.86509464, 1.87855228, 1.89200992, 1.90546757,
        1.91892521]),
 <BarContainer object of 30 artists>)
```



2.

In [ ]:
```
N = 100
runs = 10000
areas = []


for i in range(runs):
    urand = stats.uniform.rvs(size=N)
    areas.append(np.mean((func(urand)+func(1)/func(urand))/2))

m = np.mean(areas)
s = np.std(areas)
dof = N-1
conf = 0.95

t = np.abs(stats.t.ppf((1-conf)/2,dof))
confInt = (m-s*t/np.sqrt(N),m+s*t/np.sqrt(N))
print('The point estimate of the antithetic Monte Carlo estimator is: ',m)
print('While the confidence interval at 95%',' confidence is:',confInt)

plt.hist(areas, bins=30, ec= 'black');
```

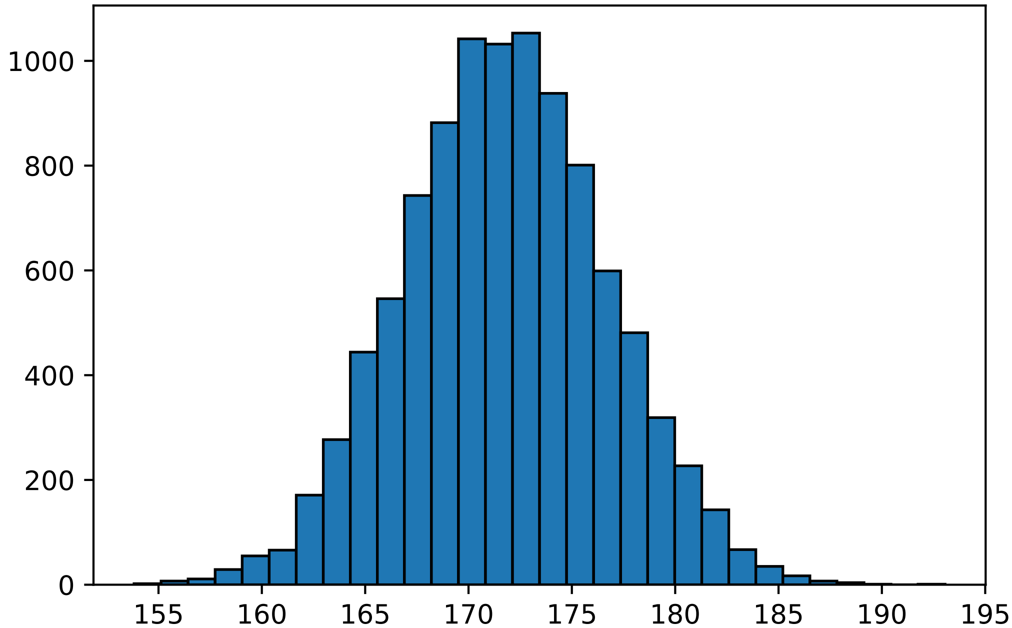
```
The point estimate of the antithetic Monte Carlo estimator is:  1.7183870169
627176
While the confidence interval at 95%  confidence is: (1.7171250095735278, 1.
7196490243519074)
```



3.

In [ ]:
```
N = 100
runs = 10000
areas = []

for i in range(runs):
    urand = stats.uniform.rvs(size=N)
    X = np.zeros(N)
    mu = 0.5
    #c = -(np.mean(urand*func(urand))-np.mean(urand)*np.mean(func(urand)))/np.
    c = 0.0039
    areas.append(np.mean(np.sum(func(urand) + c*(urand-mu))))


m = np.mean(areas)
s = np.std(areas)
dof = N-1
conf = 0.95

t = np.abs(stats.t.ppf((1-conf)/2,dof))
confInt = (m-s*t/np.sqrt(N),m+s*t/np.sqrt(N))
print('The point estimate of the Monte Carlo estimator using a control variabl
print('While the confidence interval at 95%','confidence is:',confInt)

plt.hist(areas, bins=30, ec= 'black');
```

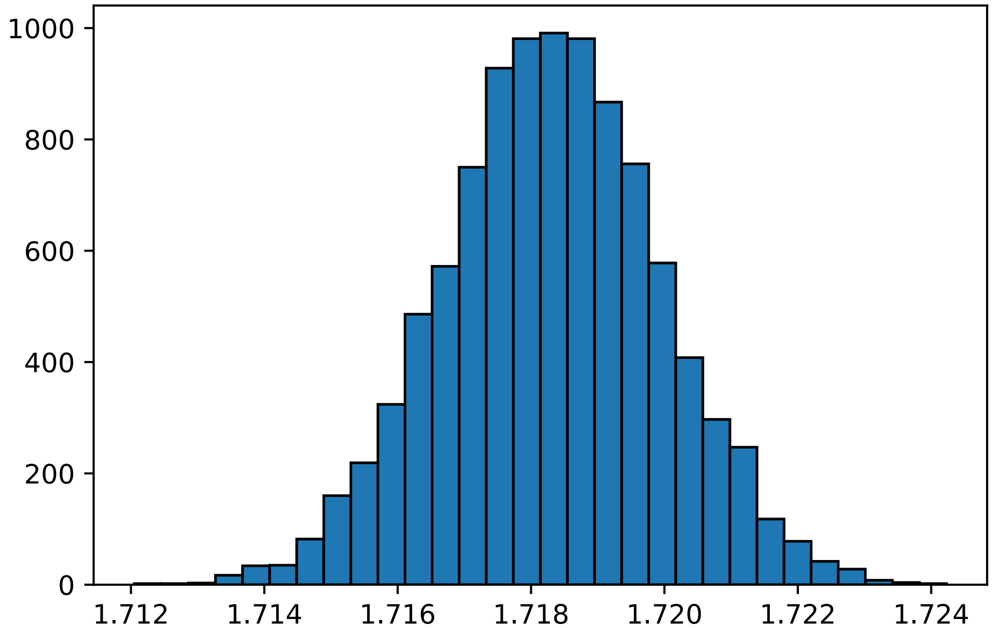
```
The point estimate of the Monte Carlo estimator using a control variable is:
171.77332159896093
While the confidence interval at 95% confidence is: (170.80038872057546, 17
2.7462544773464)
```



4.

In [ ]:
```
a=0
b=1
N = 100
strata = 10
runs = 10000
areas = []

for i in range(runs):
    urand = np.zeros((N,strata))
    for i in range(N):
        for j in range(strata):
            urand[i][j] = random.uniform(a,b)
    W = 0.0
    for i in range(N):
        for j in range(strata):
            W += func((urand[i][j]+j)/strata)/strata

    areas.append(W/float(N))

m = np.mean(areas)
s = np.std(areas)
dof = N-1
conf = 0.95
t = np.abs(stats.t.ppf((1-conf)/2,dof))
confInt = (m-s*t/np.sqrt(N),m+s*t/np.sqrt(N))
print('The point estimate of the Monte Carlo estimator using stratified sampli
print('While the confidence interval at 95%','confidence is:',confInt)
plt.hist(areas, bins=30, ec= 'black');
```

```
The point estimate of the Monte Carlo estimator using stratified sampling i
s:  1.718306774712098
While the confidence interval at 95% confidence is: (1.717984834463152, 1.71
86287149610437)
```



5.

In [ ]:
```
from src.my_random.eventBis import BlockingEventSimulation, calculate_theoreti
from dataclasses import dataclass
```

In [ ]:
```
arr_dist = stats.expon()

serv_dist = stats.expon(scale=8)
pois_sim = BlockingEventSimulation(arr_dist, serv_dist)
blocked = []
for i in range(10):
    blocked.append(pois_sim.simulate(10_000, 10))
```

In [ ]:
```
mean = np.mean(blocked)
sd = np.std(blocked)
lwr, upr = stats.t.interval(0.95, 9)
conf = [mean + sd/np.sqrt(10)*lwr, mean + sd/np.sqrt(10)*upr]

mean, conf
```

Out[ ]:
```
(0.12365000000000001, [0.11969863238092829, 0.12760136761907173])
```

In [ ]:
```
np.random.seed(seed=233423)
urand = stats.uniform.rvs(size=1000)
xrand = func(urand)
np.random.seed(seed=233423)
arr_times = stats.expon.rvs(size=1000)
exponE = arr_times.mean()
exponVar = arr_times.var()
uniE = xrand.mean()
uniVar = xrand.var()

print('Exponential dist E:',exponE,'and Var:',exponVar)
print('Unif dist into exponential E:',exponE,'and Var:',exponVar)
```

```
Exponential dist E: 0.9364349670734267 and Var: 0.8834338902973959
Unif dist into exponential E: 0.9364349670734267 and Var: 0.8834338902973959
```

In [ ]:

```
calculate_theoretical_block_pct(10, 8)
```

Out[ ]:

```
0.12166106425295149
```

6.

Reusing the same random seed we compare the prior results with hyperexponential interarrival times:

In [ ]:

```
@dataclass
class hyper_exp:
    p1: float
    p2: float
    lmbda1: float
    lmbda2: float

    def rvs(self, size):
        np.random.seed(seed=233423)
        return self.p1 * stats.expon.rvs(size=size, scale=1/self.lmbda1) \
            + self.p2*stats.expon.rvs(size=size, scale = 1/self.lmbda2)
```

In [ ]:

```
arr_erl = stats.erlang(a=1)
arr_hyp = hyper_exp(0.8, .2, .8333, 5.0)
serv_dist = stats.expon(scale=8)

sim_erl = BlockingEventSimulation(arr_erl, serv_dist)
sim_hyp = BlockingEventSimulation(arr_hyp, serv_dist)
```

In [ ]:

```
blocked = []
for i in range(10):
    blocked.append(sim_erl.simulate(10_000, 10))

mean = np.mean(blocked)
sd = np.std(blocked)
lwr, upr = stats.t.interval(0.95, 9)
conf = [mean + sd/np.sqrt(10)*lwr, mean + sd/np.sqrt(10)*upr]

mean, conf
```

Out[ ]:

```
(0.11943999999999999, [0.11327403842485124, 0.12560596157514872])
```

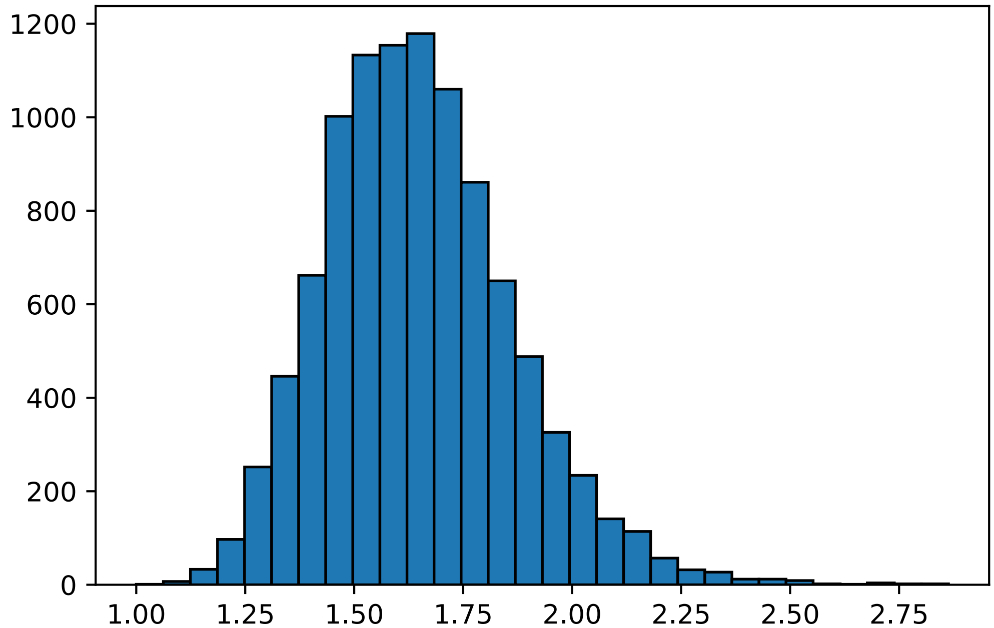
7.

In [ ]:

```
min = 0
max = 1
sig2 = 1
N = 100
runs = 10000
areas = []
np.random.seed()

for _ in range(runs):
    xrand = stats.norm.rvs(size=N)
    areas.append(np.mean(func(xrand)))

m = np.mean(areas)
```

```
s = np.std(areas)
dof = N-1
conf = 0.95

t = np.abs(stats.t.ppf((1-conf)/2,dof))
confInt = (m-s*t/np.sqrt(N),m+s*t/np.sqrt(N))
print('The point estimate of the crude Monte Carlo estimator is: ',m)
print('While the confidence interval at 95%','confidence is:',confInt,'also th

plt.hist(areas, bins=30, ec= 'black');
```

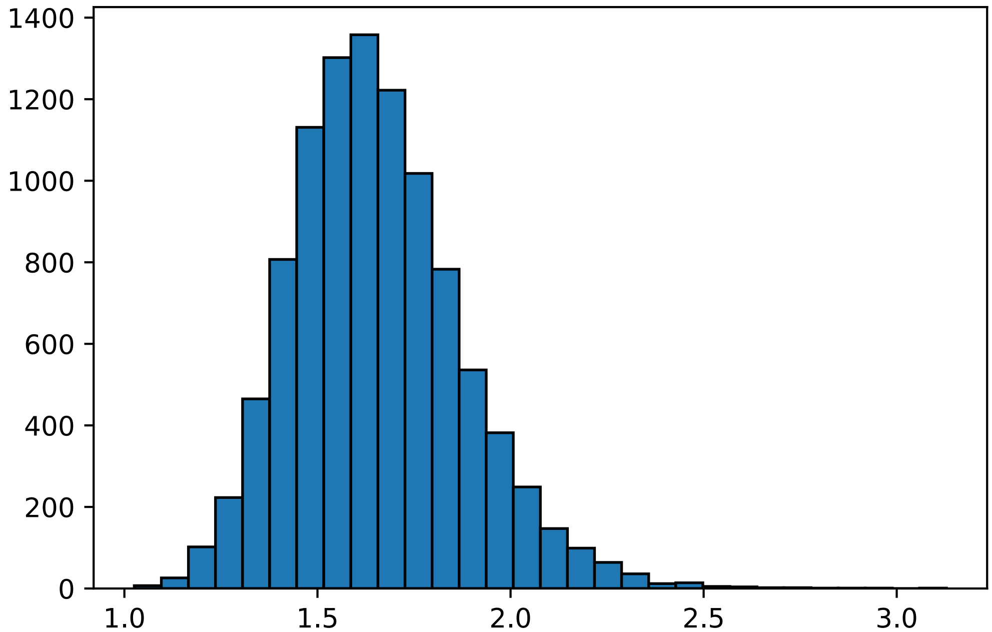
```
The point estimate of the crude Monte Carlo estimator is:  1.6491895840812825
While the confidence interval at 95% confidence is: (1.6062798655044943, 1.692
0993026580706) also this here 0.04290971857678806
```



In [ ]:
```
min = 0
max = 1
a = (0,2,4)
sig2 = 1
N = 100
runs = 10000
areas = np.zeros((len(a),runs))
np.random.seed()

for k in range(len(a)):
    for i in range(runs):
        xrand = stats.norm.rvs(size=N)
        frand = stats.norm.pdf(xrand)
        grand = stats.norm.pdf(xrand,loc=a[k],scale=1)
        integral = 0.0
        for j in range(N):
            integral += np.exp(xrand[j])*frand[j]/grand[j]
        areas[k][i]=(integral/float(N))*(max-min)
    m = np.mean(areas[k])
    s = np.std(areas[k])
    dof = N-1
    conf = 0.95
    t = np.abs(stats.t.ppf((1-conf)/2,dof))
    confInt = (m-s*t/np.sqrt(N),m+s*t/np.sqrt(N))
```

```
    print('The point estimate of the crude Monte Carlo estimator is: ',m)
    print('While the confidence interval at 95%','confidence is:',confInt,'wit


plt.hist(areas[0], bins=30, ec= 'black');
```

```
The point estimate of the crude Monte Carlo estimator is:  1.6540128907368432
While the confidence interval at 95% confidence is: (1.6105138397424097, 1.697
5119417312767) with a = 0
The point estimate of the crude Monte Carlo estimator is:  12.195267984674688
While the confidence interval at 95% confidence is: (11.881315323123935, 12.50
922064622544) with a = 2
The point estimate of the crude Monte Carlo estimator is:  292474.0311515448
While the confidence interval at 95% confidence is: (-36431.78996874974, 62137
9.8522718394) with a = 4
```

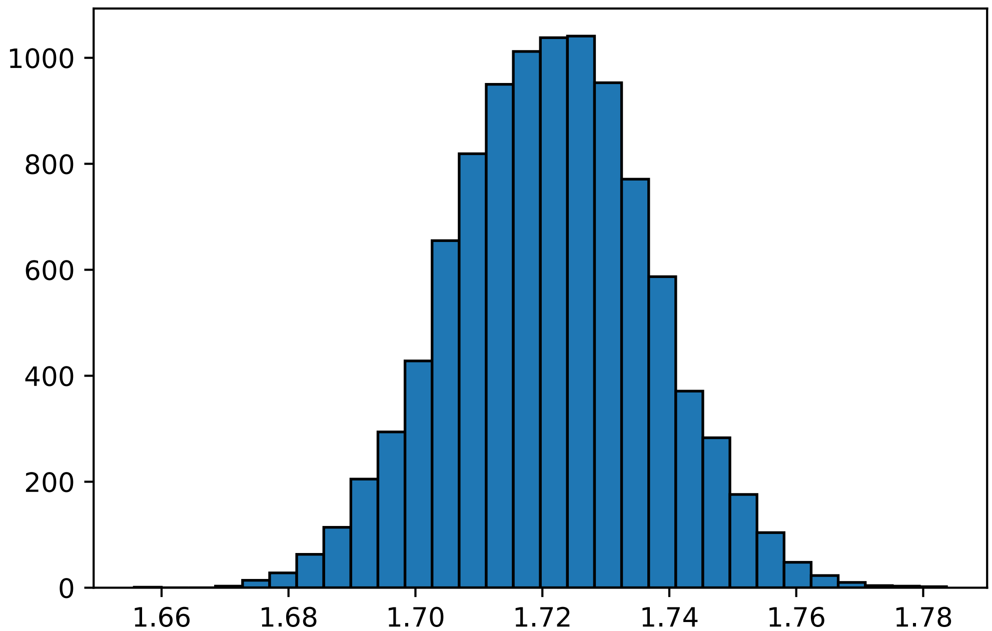


8.

In [ ]:
```
min = 0
max = 1
sig2 = 1
N = 100
runs = 10000
areas = []
lamb = -0.6835
np.random.seed()

for _ in range(runs):
    xrand = stats.uniform.rvs(size=N)
    frand = stats.uniform.pdf(xrand)
    grand = lamb*np.exp(-lamb*xrand)
    areas.append(np.abs(np.mean(np.exp(xrand)*frand/(grand))))

m = np.mean(areas)
s = np.std(areas)
dof = N-1
conf = 0.95
```

```
t = np.abs(stats.t.ppf((1-conf)/2,dof))
confInt = (m-s*t/np.sqrt(N),m+s*t/np.sqrt(N))
print('The point estimate of the crude Monte Carlo estimator is: ',m)
print('While the confidence interval at 95%','confidence is:',confInt)

plt.hist(areas, bins=30, ec= 'black');
```

```
The point estimate of the crude Monte Carlo estimator is:  1.7213303785319012
While the confidence interval at 95% confidence is: (1.7182141724276476, 1.724
4465846361547)
```



9. For the pareto case, using the First moment distribution of the pareto as sampling distribution, we derive the expected mean of the IS estimator to be equal to the theoretical mean. Should one know the first moment distribution of a distribution one is attempting to approximate, implementing the first moment as a sampling distribution would in theory make sense should the expected value be unknown and more difficult to compute.

$$\frac{x\frac{k\beta^k}{x^{k+1}}}{\frac{(k-1)\beta^{k-1}}{x^k}} = \frac{k}{k-1}\beta$$

Exercise 6

In [ ]:
```
%load_ext autoreload
%autoreload 2
```

In [ ]:
```
from scipy import stats
import numpy as np
import random
import matplotlib.pyplot as plt
import seaborn as sns

from src.my_random.mcmc import *
p = [1/3, 1/3, 1/3]

dx = [np.flatnonzero(stats.multinomial.rvs(1, p))[0] -1 for _ in range(3)]
dx
```

Out[ ]:
```
[0, 0, -1]
```

1) 1D Case

In [ ]:
```
x1 = mcmc_1(5, g_1, h_1, step_1)
obs_count, exp_dist = [], []
c = sum(g_1(p) for p in range(11))
for p in range(11):
    obs_count.append(len([x for i, x in enumerate(x1) if x==p and i%5 == 0]))
    exp_dist.append(g_1(p) / c)

exp_count = np.array(exp_dist) * sum(obs_count)
exp_count, np.array(obs_count)

stats.chisquare(obs_count, exp_count)
```

Out[ ]:
```
Power_divergenceResult(statistic=13.35705945884378, pvalue=0.2043876138036145
3)
```

2a) Proposed point is any of the 8 nearest points with equal probability

In [ ]:
```
x2a = mcmc(np.array([1,1]), g2, h2a, step=step2a)
x2b = mcmc(np.array([1,1]), g2, h2b, step=step2b)
```

In [ ]:
```
obs_count, exp_dist = [], []
c = sum(g2(p) for p in set_of_valid_points())
for p in set_of_valid_points(10):
    obs_count.append(len([x for i, x in enumerate(x2a) if x==p and i%5 == 0]))
    exp_dist.append(g2(p) / c)

exp_count = np.array(exp_dist) * sum(obs_count)
exp_count, np.array(obs_count)
```

```
stats.chisquare(obs_count, exp_count)
```

Out[ ]:

```
Power_divergenceResult(statistic=70.01446260628832, pvalue=0.3130872157135906
6)
```

2b) Proposed point is one of the 4 nearest point in the cardinal direction with equal probability

In [ ]:

```
obs_count, exp_dist = [], []
c = sum(g2(p) for p in set_of_valid_points())
for p in set_of_valid_points(10):
    obs_count.append(len([x for i, x in enumerate(x2b) if x==p and i%5 == 0]))
    exp_dist.append(g2(p) / c)

exp_count = np.array(exp_dist) * sum(obs_count)
exp_count, np.array(obs_count)

stats.chisquare(obs_count, exp_count)
```

Out[ ]:

```
Power_divergenceResult(statistic=70.3651636866106, pvalue=0.30281502820921335)
```

In [ ]:

```
sum(exp_dist)
```

Out[ ]:

```
1.0
```

2c) Gibbs sampling. Marginal distributions are found as $P(i|j) = \frac{P(i,j)}{\sum_i P(i,j)}$

In [ ]:

```
x2c = gibbs2c([1,1])
```

```
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
```

In [ ]:

```
obs_count, exp_dist = [], []
c = sum(g2(p) for p in set_of_valid_points())
for p in set_of_valid_points(10):
    obs_count.append(len([x for i, x in enumerate(x2c) if tuple(x)==p]))
    exp_dist.append(g2(p) / c)

exp_count = np.array(exp_dist) * sum(obs_count)
exp_count, np.array(obs_count)

stats.chisquare(obs_count, exp_count)
```

Out[ ]: Power_divergenceResult(statistic=52.427073479149186, pvalue=0.8694071598391406)

Continuous Case

The postererior distribution is given as

$$f_{\Theta,\Psi|X}(\theta,\psi) = c f_{X|\Theta,\Psi}(x)\, f_{\Theta,\Psi}(\theta,\psi)$$

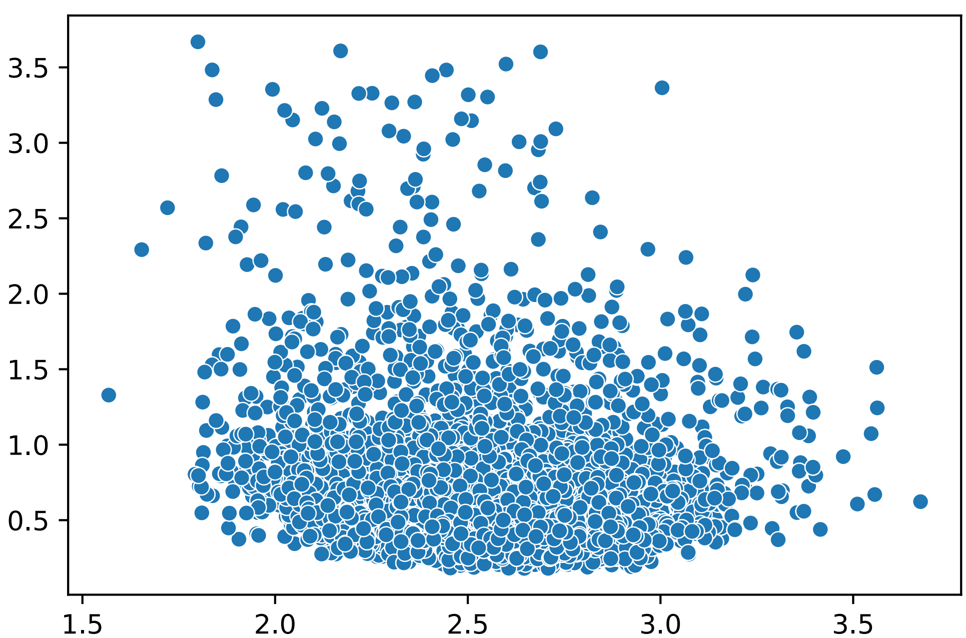
In [ ]:
```
np.random.seed(seed = 184012)
obs, true_par = gen_observations(10)
x3c = mcmc_continuous(np.log([np.mean(obs), np.var(obs)]), obs, g3, norm_step,
```

```
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
```

In [ ]:
```
x3c = np.stack(x3c)
```

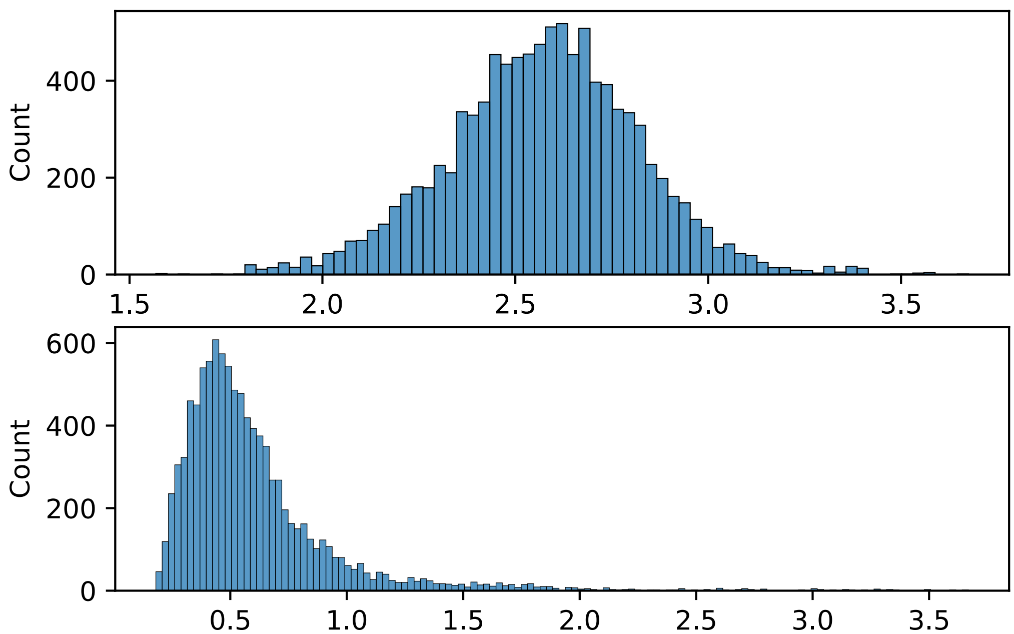
In [ ]:
```
sns.scatterplot(x = x3c[:,0], y=x3c[:,1])
```

Out[ ]: <AxesSubplot:>



In [ ]:
```
fig, ax = plt.subplots(2,1)
g = sns.histplot(x3c[:,0], ax=ax[0])
sns.histplot(x3c[:,1], ax=ax[1])
# g.set(xlim=(0,4))
```

Out[ ]: <AxesSubplot:ylabel='Count'>



In [ ]:
```
true_par, np.mean(obs)
```

Out[ ]: ((2.35759407089737, 0.5712394274331519), 2.641308398052893)

The method seems to be overshooting the mean while undershooting the variance the true value quite a bit with only 10 observations. We se the mean of the 10 observations is also way above the true parameter

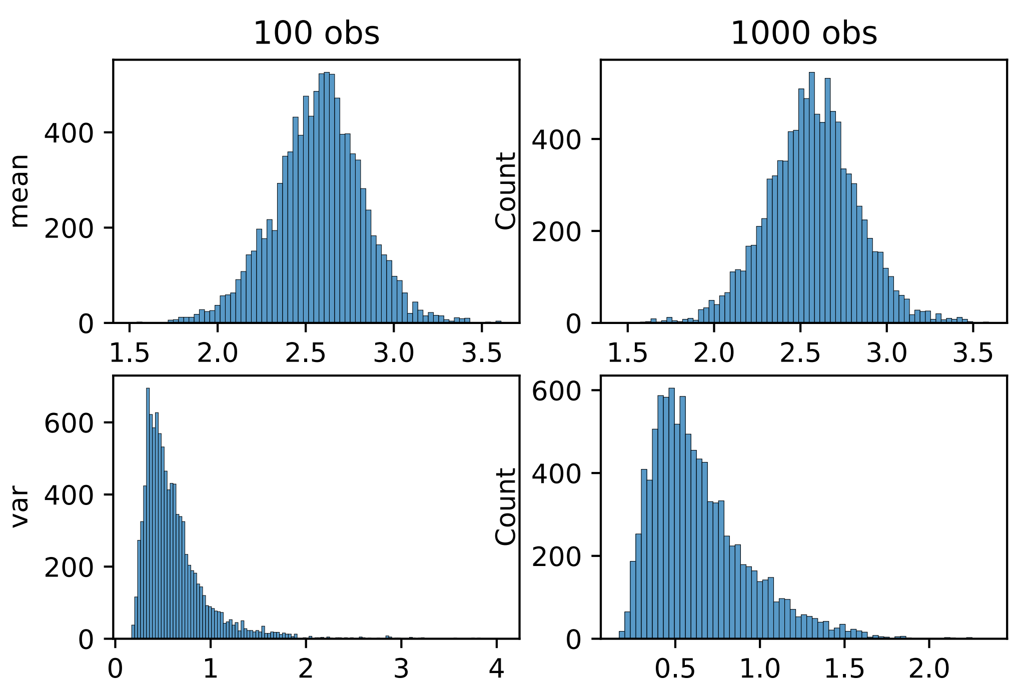
In [ ]:
```
np.random.seed(seed = 184012)
obs_100, true_par = gen_observations(100)
np.random.seed(seed = 184012)
obs_1000, true_par = gen_observations(1000)

x3c_100 = mcmc_continuous(np.log([np.mean(obs), np.var(obs)]), obs, g3, norm_s
x3c_1000 = mcmc_continuous(np.log([np.mean(obs), np.var(obs)]), obs, g3, norm_
```

```
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
```

In [ ]:
```python
x3c_100 = np.stack(x3c_100)
x3c_1000 = np.stack(x3c_1000)
```

In [ ]:
```python
fig, ax = plt.subplots(2,2)
g = sns.histplot(x3c_100[:,0], ax=ax[0,0])
sns.histplot(x3c_100[:,1], ax=ax[1,0])
g = sns.histplot(x3c_1000[:,0], ax=ax[0,1])
sns.histplot(x3c_1000[:,1], ax=ax[1,1])

ax[0,0].set_title('100 obs')
ax[0,0].set_ylabel('mean')
ax[1,0].set_ylabel('var')
ax[0,1].set_title('1000 obs')
```

Out[ ]:
```
Text(0.5, 1.0, '1000 obs')
```



In [ ]:
```
true_par
```

Out[ ]:
```
(2.35759407089737, 0.5712394274331519)
```

Exercise 7:

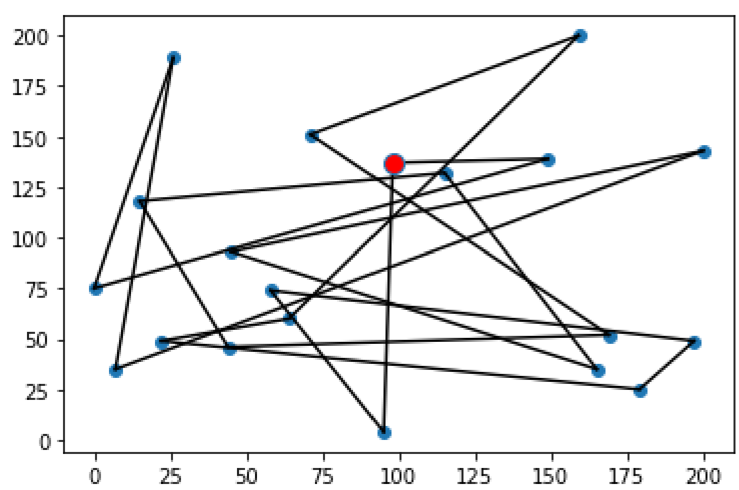
In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from scipy import stats
import networkx as nx
import random
from src.my_random.sales import*
```

a)

In [ ]:
```
n = 20
stations = gen_stations(n)
route = np.arange(0,n)
random.shuffle(route)
cost = euclDist(stations[:,route[n-1]],stations[:,route[0]])
```

In [ ]:
```
plt.figure()
plt.plot([stations[0,route[0]],stations[0,route[n-1]]],[stations[1,route[0]],s
for i in range(n-1):
    cost += euclDist(stations[:,route[i]],stations[:,route[i+1]])
    plt.plot([stations[0,route[i]],stations[0,route[i+1]]],[stations[1,route[i
plt.scatter(stations[0,:],stations[1,:])
plt.plot(stations[0,0],stations[1,0],marker='o',markerfacecolor='red',markersi

print('Total cost of this route:',cost)
```

```
Total cost of this route: 2410.3705531657756
```



b)

In [ ]:
```
df = pd.read_csv(r'C:/Users/lenovo/Documents/DTU/02443/cost.csv',header=None)
df
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Out[ ]:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	0	225	110	8	257	22	83	231	277	243	94	30	4	265	274	250	87	83
1	255	0	265	248	103	280	236	91	3	87	274	265	236	8	24	95	247	259
2	87	236	0	95	248	110	25	274	250	271	9	244	83	250	248	280	29	26
3	8	280	83	0	236	28	91	239	280	259	103	23	6	280	244	259	95	87
4	268	87	239	271	0	244	275	9	84	25	244	239	275	83	110	24	274	280
5	21	265	99	29	259	0	99	230	265	271	87	5	22	239	236	250	87	95
6	95	236	28	91	247	93	0	247	259	244	27	91	87	268	275	280	7	8
7	280	83	250	261	4	239	230	0	103	24	239	261	271	95	87	21	274	255
8	247	9	280	274	84	255	259	99	0	87	255	274	280	3	27	83	259	244
9	230	103	268	275	23	244	264	28	83	0	268	275	261	91	95	8	277	261
10	87	239	9	103	261	110	29	255	239	261	0	259	84	239	261	242	24	25
11	30	255	95	30	247	4	87	274	242	255	99	0	24	280	274	259	91	83
12	8	261	83	6	255	29	103	261	247	242	110	29	0	261	244	230	87	84
13	242	8	259	280	99	242	244	99	3	84	280	236	259	0	27	95	274	261
14	274	22	250	236	83	261	247	103	22	91	250	236	261	25	0	103	255	261
15	244	91	261	255	28	236	261	29	103	9	242	261	244	87	110	0	242	236
16	84	236	27	99	230	83	7	259	230	230	22	87	93	250	255	247	0	9
17	91	242	28	87	250	110	6	271	271	255	27	103	84	250	271	244	5	0
18	261	24	250	271	84	255	261	87	28	110	250	248	248	22	3	103	271	248
19	103	271	8	91	255	91	21	271	236	271	7	250	83	247	250	271	22	27

In [ ]:

```
w = np.zeros(n)
for l in range(n):
    w[l] = np.sum(df.values[l,:])

e = list(range(0,n,1))
pointA = random.choices(e,weights=1/w)
print('Route:',route)
indexA = np.where(route==pointA)[0]
print('A:',pointA,', index:',indexA)
```

```
Route: [ 9 11  6  8  4 15  1 19 12 17 14 16  2 10  7  5 18 13  3  0]
A: [0] , index: [19]
```

In [ ]:

```
n = 20
iterations = 1000
permutations = 2
stations = gen_stations(n)
route = np.arange(0,n)
optRoute = route
random.shuffle(route)
k = 0
costMin = np.sum(np.sum(df.values))
    T=1/np.sqrt(1+k)
```

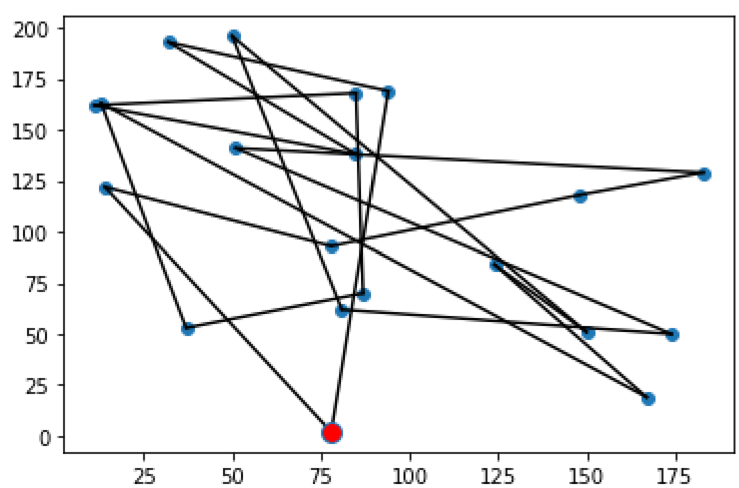
Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
    #T=-np.log(1+k)
    k+=1
    for j in range(permutations):
        w = np.zeros(n)
        e = list(range(0,n,1))
        for l in range(n):
            w[l] = np.sum(df.values[l,:])
        pointA = random.choices(e,weights=T/w)
        pointB = random.choices(e,weights=T/df.values[:,pointA])
        indexA = np.where(route==pointA)[0]
        indexB = np.where(route==pointB)[0]
        route[indexA], route[indexB] = route[indexB], route[indexA]

    cost = df.values[route[len(route)-1]][route[0]]
    for i in range(len(route)-1):
        cost += df.values[route[i]][route[i+1]]
    if cost<costMin:
        costMin = cost
        optRoute = route
    else:
        route = optRoute


print('The estimated optimal route follows this sequence:', optRoute,'and enta
plt.figure()
plt.plot([stations[0,route[0]],stations[0,route[n-1]]],[stations[1,route[0]],s
for i in range(n-1):
    plt.plot([stations[0,route[i]],stations[0,route[i+1]]],[stations[1,route[i
plt.scatter(stations[0,:],stations[1,:])
plt.plot(stations[0,0],stations[1,0],marker='o',markerfacecolor='red',markersi
```

```
<ipython-input-66-b04bce06d701>:20: RuntimeWarning: divide by zero encountered
in true_divide
  pointB = random.choices(e,weights=T/df.values[:,pointA])
The estimated optimal route follows this sequence: [ 0  4  3 12  7 10  8  2 18
11 15 13  6 17  9 14  5 19 16  1] and entails the following cost: 1912
```

Out[ ]:

```
[<matplotlib.lines.Line2D at 0x1bef6a389d0>]
```



Exercise 8)

Exercise 13 from book

a)

We take r subsets with replacement of the data length n, and calculate the emperical mean r times. Then, for each subset, we subtract the mean of all the means from the mean of each subset, and count how many of theese numbers are within the interval [a,b]

b)

In [ ]:
```
import numpy as np
x = np.array([56, 101, 78, 67, 93, 87, 64, 72, 80, 69])

r = 100000

X = [np.random.choice(x, len(x)) for _ in range(r)]
X = np.stack(X)
emp_mean = X.mean(axis=1)
mean = emp_mean.mean()
p = emp_mean - mean
p = np.count_nonzero(abs(p) < 5) / r
```

In [ ]:
```
p
```

Out[ ]:
```
0.76581
```

Exercise 15 from book

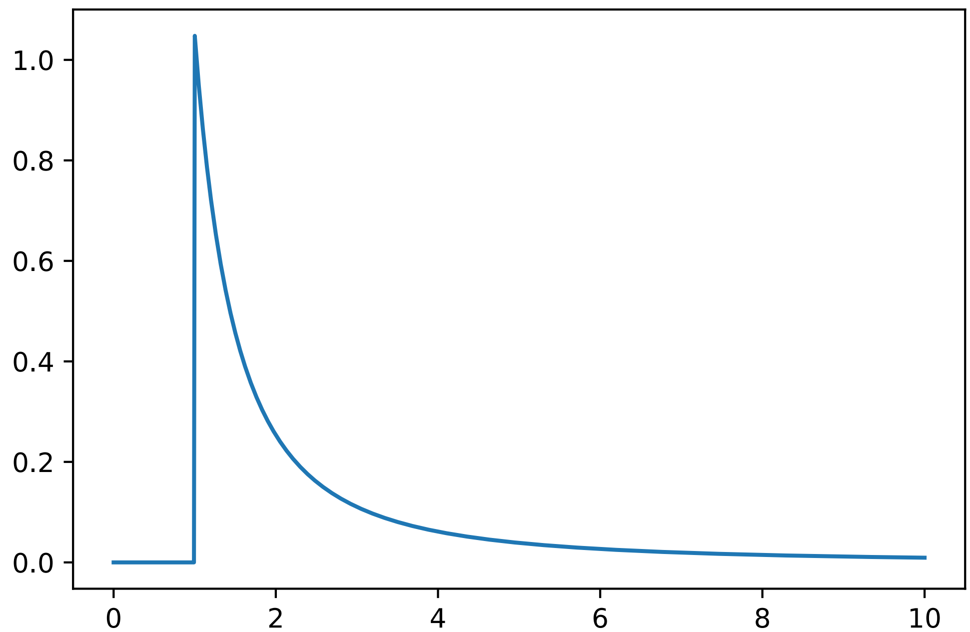
In [ ]:
```
n, r = 15, 10000
x = [5, 4, 9, 6, 21, 17, 11, 20, 7, 10, 21, 15, 13, 16, 8]
X = [np.random.choice(x, n) for _ in range(r)]
X = np.stack(X)
s2 = X.var(axis=1)

len(s2)

s2.var()
```

Out[ ]:
```
51.433067181649385
```

Exercise 8.3

In [ ]:
```python
from scipy.stats import pareto as sci_pareto
import seaborn as sns
import pandas as pd

def pareto(beta, k):
    return sci_pareto(b=k, scale=beta)

def bootstrap(data, stat_func=lambda x: np.median, size = 1000):
    X = [np.random.choice(data, len(data)) for _ in range(size)]
    stat = stat_func(X, axis=1)
    return stat.var()

x = np.linspace(0,10,1000)
sns.lineplot(x=x, y=pareto(1, 1.05).pdf(x))
```

Out[ ]: <AxesSubplot:>



In [ ]:
```python
sample = pareto(1, 1.05).rvs(size=200)
```

In [ ]:
```python
mean, median = sample.mean(), np.median(sample)
var_mean = bootstrap(sample, np.mean)
var_median = bootstrap(sample, np.median)
```

In [ ]:
```python
df = pd.DataFrame({'stat': [mean, median], 'var':[var_mean, var_median]}, inde
```

In [ ]:
```python
df
```

Out[ ]:

	stat	var
mean	4.141244	0.229267
median	1.749461	0.013682

The Precision of the median is much better

APPENDIX

Event.py

In [ ]:

```python
from dataclasses import dataclass
from enum import Enum, auto
from math import factorial
from matplotlib import pyplot as plt
import numpy as np
from scipy import stats
import seaborn as sns


class State(Enum):
    INCOMING = auto()
    IN_SERVICE = auto()
    SERVICED = auto()
    BLOCKED = auto()

@dataclass
class Event:
    state: State
    arrival_time: int
    departure_time: int


class EventList:

    events: 'list[Event]'
    in_service: 'list[Event]'

    def __init__(
        self, arrival_time_distribution: stats.rv_continuous,
        service_time_distribution: stats.rv_continuous,
        number_of_events
    ):
        self._arr_dist = arrival_time_distribution
        self._serv_dist = service_time_distribution
        self.events = self.generate_events(number_of_events)
        self.time_line = {}
        self.in_service = []

    @property
    def states(self):
        return [event.state for event in self.events]

    def update_in_service(self, time):
        for event in self.in_service:
            if event.departure_time <= time:
                event.state = State.SERVICED

        self.in_service = [event for event in self.events if event.state == St

    def generate_events(self, number_of_events: int) -> 'list[Event]':
```

```
        arr_times = self._arr_dist.rvs(size=number_of_events)
        arr_times = np.cumsum(arr_times)

        serv_times = self._serv_dist.rvs(size=number_of_events)
        dep_times = arr_times + serv_times
        return [Event(State.INCOMING, arr, dep) for arr, dep in zip(arr_times,

    def update_timeline(self, time):
        self.time_line[time] = self.states

    def __str__(self):
        str = ''
        for iter, (time, states) in enumerate(self.time_line.items()):
            if iter < len(self.events) - 10:
                continue
            str += f'TIME: {time:.2f}\n'
            for i, state in enumerate(states):
                if state != State.INCOMING:
                    str += f'Obs. {i}: {state.name}\n'
            str += '\n'
        return str

class BlockingEventSimulation:

    def __init__(
            self, arrival_time_distribution: stats.rv_continuous,
            service_time_distribution: stats.rv_continuous
    ):
        self.arrival_dist = arrival_time_distribution
        self.service_dist = service_time_distribution

    def simulate(self, max_events: int, service_units: int):
        blocked_count = 0
        event_list = EventList(self.arrival_dist, self.service_dist, max_event
        for event in event_list.events:
            time = event.arrival_time
            event_list.update_in_service(time)
            if len(event_list.in_service) < service_units:
                event.state = State.IN_SERVICE
                event_list.update_in_service(time)
            else:
                event.state = State.BLOCKED
                blocked_count += 1


            event_list.update_timeline(time)

        return blocked_count / max_events


def calculate_theoretical_block_pct(m, a):
    return (a**10/factorial(m))/ sum([a**i / factorial(int(i)) for i in range(

if __name__ == '__main__':
    pass
```

gen.py

In [ ]:
```python
import itertools
from re import I
from typing import Callable, Protocol
import numpy as np
from scipy.stats import uniform

def lcg(a, c, M, n, x=0):
    for _ in range(int(n)):
        x = (a*x + c) % M
        yield x / M

def geometric(p, size):
    u = uniform.rvs(size=size)
    return np.log(u) // np.log(1-p) + 1

def exponential(lmbda, size):
    u = uniform.rvs(size=size)
    return - np.log(u) / lmbda

def pareto(k , beta, size, loc=0):
    u = uniform.rvs(size=size)
    return beta*(u**(-1/k) - loc)

def norm_box_mueller(size):
    u1 = uniform.rvs(size=size)
    r = np.sqrt(-2*np.log(u1))
    return r*sin_cos(size)

def sin_cos(size):
    sin, cos = [], []
    while len(cos) < size / 2:
        v1, v2 = uniform.rvs(loc=-1, scale=2, size=2)
        r2 = v1**2 + v2**2
        if r2 <= 1:
            cos.append(v1/np.sqrt(r2))
            sin.append(v2/np.sqrt(r2))

    return np.array(sin + cos)


def discrete_crude(p, size):
    u =  uniform.rvs(size=size)
    probs = np.concatenate([[0], np.cumsum(p), [np.inf]], axis=0)
    x = np.zeros_like(u)
    for i, p in enumerate(probs):
        if 0 < i:
            x += ((probs[i-1] < u) & (u <= p)) * i

    return x

def discrete_rejection(p, size):
    c = max(p)
    k = len(p)
    I = []
    while len(I) < size:
        u1, u2 = uniform.rvs(size=2)
        i = int(np.floor(k*u1)) +1
        if u2 <= p[i-1]/c:
            I.append(i)
```

```
    return I

def discrete_alias(p, size):
    k = len(p)
    p = np.array(p)
    L = list(range(1,k+1))
    F = k*p
    G, S = np.where(F>=1)[0] + 1, np.where(F<=1)[0] + 1
    while len(S) > 0:
        i, j = int(G[0]), int(S[0])
        L[j-1] = i
        F[i-1] = F[i-1] - (1-F[j-1])
        if F[i-1] < 1:
            G = np.delete(G, 0)
            print(S)
            S = np.append(S, i)
            print(S)
        S = np.delete(S, 0)


    result = []
    while len(result) < size:
        u1, u2 = uniform.rvs(size=2)
        i = int(np.floor(k*u1)) + 1
        if u2 <= F[i-1]:
            result.append(i)
        else:
            result.append(L[i-1])

    return result
```

mcmc.py

In [ ]:

```
from itertools import product
from math import factorial, pi
import numpy as np
from scipy.stats import binom, uniform, multivariate_normal, norm
import random

def h_1(x, y, m=10):
    return .5 if abs(x-y) % (m-1) == 1 else 0

def step_1(x, m=10):
    dx = 1 if binom.rvs(1, .5) == 1 else -1
    return (x + dx) % (m+1)

def g_1(x, a=8, m=10):
    return a**x / factorial(x)

def mcmc_1(x0, g, h, step, a=8, m=10, size = 10_000, burn_in = 100):
    x = x0
    for i in range(burn_in):
        y = step_1(x, m)
        cond = (g(y) * h(y,x)) / (g(x)*h(x,y))
        if uniform.rvs() <= cond:
            x = y

    states = []
```

```
    for i in range(size):
        y = step_1(x, m)
        cond = (g(y) * h(y,x)) / (g(x)*h(x,y))
        if uniform.rvs() <= cond:
            x = y
        states.append(x)

    return states

def set_of_valid_points(m=10):
    point_in_set = lambda i,j: 0 <= i + j <= m \
        and i >= 0 \
        and j >=0
    return {(i,j) for i,j in product(range(m+1), repeat=2) if point_in_set(i,j

def nearby_points(x, m=10):
    for i,j in product([-1, 0, 1], repeat=2):
        if i == j:
            continue
        new_point = (x[0] + i, x[1] + j)
        if new_point in set_of_valid_points(m):
            yield new_point

def cardinal_points(x, m=10):
    for i,j in product([-1, 0, 1], repeat=2):
        if i == j:
            continue
        if i!=0 and j!=0:
            continue

        new_point = (x[0] + i, x[1] + j)
        if new_point in set_of_valid_points(m):
            yield new_point

def g2(x, m=10, a1=4, a2=4):
    return a1**x[0]* a2**x[1] / (factorial(x[0])*factorial(x[1]))


def h2a(x, y, m=10):
    if y[0] + y[1] > m:
        return 0
    valid_count = 0
    for p in nearby_points(x, m):
        valid_count+=1


    return 1/valid_count


def step2a(x, m):
    return random.choice([p for p in nearby_points(x, m)])


def h2b(x, y, m=10):
    if y[0] + y[1] > m:
        return 0

    valid_count = 0
    for p in cardinal_points(x, m):
        valid_count += 1
```

```
    return 1/valid_count


def step2b(x, m):
    return random.choice([p for p in cardinal_points(x, m)])



def mcmc(x0, g, h, step, a=8, m=10, size = 10_000, burn_in = 100):
    x = x0
    for _ in range(burn_in):
        y = step(x, m)
        cond = (g(y) * h(y,x)) / (g(x)*h(x,y))
        if uniform.rvs() <= cond:
            x = y

    states = []
    for _ in range(size):
        y = step(x, m)
        cond = (g(y) * h(y,x)) / (g(x)*h(x,y))
        if uniform.rvs() <= cond:
            x = y
        states.append(x)

    return states


def p2(i, j, m=10):
    return g2((i,j)) / sum([g2((i,j)) for i,j in set_of_valid_points(m=10)])

def get_marginal_g2(i, x, m=10):
    j = x[(i+1) % 2]
    return [p2(i,j) / sum(p2(k,j) for k in range(m-j+1)) for i in range(m-j+1)


def gibbs2c(x0, m = 10, size=10_000, burn=100):
    x = x0
    res = []
    for iter in range(size+burn):
        for i, x_i in enumerate(x):
            dist = get_marginal_g2(i, x, m)
            x[i] = np.random.choice(len(dist), p = dist)
        if iter >= burn:
            res.append(x.copy())

        if iter % 1000 == 0:
            print(iter)

    return res


def gen_xi_gamma(size=1):
    return multivariate_normal([0, 0], np.array([[1, .5],[.5, 1]])).rvs(size=s

def gen_theta_psi(size=1):
    return np.exp(gen_xi_gamma(size=size))

def gen_observations(size=1):
    mean, var = gen_theta_psi()
```

```
    return norm(mean, np.sqrt(var)).rvs(size=size), (mean, var)

def norm_step(x):
    dx = norm(loc = 0, scale=1e-1).rvs(2)
    return x + dx

def g3(x, obs):
    ln_pdf = 1/(2*pi*x[0]*x[1]*np.sqrt(1 - .5**2))\
        *np.exp(- (np.log(x[0])**2 - np.log(x[0])*np.log(x[1]) + np.log(x[1])*
            / 2*(1-.5**2))
    return np.exp(sum(norm(loc=x[0], scale=np.sqrt(x[1])).logpdf(obs))) * ln_p


def mcmc_continuous(x0, obs, g, step, burn_in=100, size=10_000):
    x = x0
    for _ in range(burn_in):

        y = step(x)
        cond = (g(np.exp(y), obs) / (g(np.exp(x), obs)))
        if uniform.rvs() <= cond:
            x = y

    states = []
    for i in range(size):
        y = step(x)
        cond = g(np.exp(y), obs) / g(np.exp(x), obs)
        if uniform.rvs() <= cond:
            x = y
        states.append(np.exp(x))
        if i % 1000 == 0:
            print(i)

    return states


if __name__ == '__main__':
    obs = gen_observations(10)
    print(mcmc_continuous([0,0], obs, g3, norm_step, size=10_000))
```

tests.py

In [ ]:

```
from typing import Iterable, Tuple
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import scipy.stats as stats
import scipy.optimize as opt


def group_count(p: Iterable, c: int) -> np.ndarray:
    split = 1 / c
    splits = np.zeros(c)
    for n in p:
        for i in range(c):
            if n <= split*(i+1):
                splits[i] += 1
                break
    return splits
```

```python
def chi2(obs: np.ndarray, exp: np.ndarray, df=None):
    if df is None:
        df = len(obs) - 1
    T = sum((obs - exp)**2 / exp)
    p = 1 - stats.chi2.cdf(x=T, df=df)
    return p

def group_chi_test(obs, n_groups):
    splits = group_count(obs, n_groups)
    p = chi2(splits, np.ones_like(splits)*len(obs)/n_groups)
    return p

def emperical_dist(x: float, obs: np.ndarray) -> np.ndarray:
    return 1/len(obs) * sum(obs <= x)

def kolmogorov(obs, dist=stats.uniform, range=(0,1)):
    n = len(obs)
    obj = lambda x: - emperical_dist(x, obs) + dist.cdf(x)
    d_n = opt.minimize_scalar(obj, bounds = range, method='bounded').x
    return (np.sqrt(n) + 0.12 + 0.11/np.sqrt(n))*d_n, d_n

def runtest_above_below_median(obs: np.ndarray) -> int:
    T, n1, n2 = counts_above_and_below_median(obs)
    mean = (2*n1*n2)/(n1+n2) + 1
    var = (2*n1*n2*(2*n1*n2 - n1 - n2)) / ((n1 + n2)**2 * (n1 + n2 -1))
    print(mean, var, T)
    T = (T - mean)/np.sqrt(var)

    return 2* (1- stats.norm.cdf(abs(T)))

def counts_above_and_below_median(obs: np.ndarray) -> Tuple[int, int, int]:
    r_a = obs > np.median(obs)
    r_b = obs < np.median(obs)
    x, count = 0, 0
    for a, b in zip(r_a, r_b) :
        if a==1 and x != 1:
            count += 1
            x = 1
        elif b==1 and x != -1:
            count += 1
            x = -1

    return count, sum(r_a), sum(r_b)


def runtest_up_down_lengths(obs: np.ndarray) -> int:
    n = len(obs)
    r = run_lengths_increasing_count(obs)
    a = np.array(
        [
        [4529.4, 9044.9, 13568, 18091, 22615, 27892],
        [9044.9, 18097, 27139, 36187, 45234, 55789],
        [13568, 27139, 40721, 54281, 67852, 83685],
        [18091, 36187, 54281, 72414, 90470, 111580],
        [22615, 45234, 67852, 90470, 113262, 139476],
        [27892, 55789, 83685, 111580, 139476, 172860]
        ]
    )
```

```python
    b = np.array([1/6, 5/24, 11/120, 19/720, 29/5040, 1/840])
    z = 1/(n-6) * ((r - n*b).T @ a @ (r - n*b))

    return 1 - stats.chi2.cdf(z, df=6)

def run_lengths_increasing_count(obs: np.ndarray) -> np.ndarray:
    prev_x = -np.inf
    r = np.zeros(6)
    count = 0
    for x in obs:
        if x <= prev_x:
            count = min(6, count)
            r[count-1] += 1
            prev_x = x
            count = 1
        else:
            prev_x = x
            count += 1

    count = min(6, count)
    r[count-1] += 1

    return r


def runtest_increase_decrease(obs):
    t = run_count_increase_decrease(obs)
    n = len(obs)
    z = (t - (2*n - 1)/3) / np.sqrt((16*n - 29)/90)
    return 2 * (1- stats.norm.cdf(abs(z)))



def run_count_increase_decrease(obs: np.ndarray):
    x, count, prev= 0, 0, obs[0]
    for y in obs[1:]:
        if x != 1 and y > prev:
            count += 1
            x = 1
        elif x != -1 and y <= prev:
            count += 1
            x = -1
        prev = y
    return count


def corr_est(obs, max_lag) -> np.ndarray:
    n = len(obs)
    c = np.zeros(max_lag)
    for lag in range(max_lag):
        low = obs[:n-lag-1]
        upp = obs[lag+1:]
        c[lag] = 1/(n-lag) * low @ upp

    return c

def plot_corr(obs: np.ndarray, max_lag=5, conf=0.05) -> None:
    n = len(obs)
    corr_coef = (corr_est(obs, max_lag) - 0.25)
    x = np.arange(1, len(corr_coef)+1)
```

```
    conf = stats.norm.ppf(1 - conf/2) * np.sqrt((7/(144*n)))
    plt.plot(x, corr_coef, 'ob')
    plt.vlines(x, np.zeros_like(x), corr_coef)
    plt.hlines([conf, 0, -conf], 0, max_lag+1, linestyles=['dashed', 'solid',
    plt.show()


def all_test(obs, groups=100, lag=5, plot=True):
    p_chi = group_chi_test(obs, 100)
    T_kol = kolmogorov(obs)
    p_ab_median = runtest_above_below_median(obs)
    p_ud = runtest_up_down_lengths(obs)
    p_inc_dec = runtest_increase_decrease(obs)

    print(f'____________Uniform Distribution Tests___________')
    print(f'Chi^2 test with {groups} groups:                p={p_chi:.2f}')
    print(f'Kolmogorov Smirnof:                        T={T_kol:.2f}')
    print(f'_______________Independence Tests________________')
    print(f'Run Test 1: Above/below Median:            p={p_ab_median:.2f}')
    print(f'Run Test 2: Up/Down length count Test:     p={p_ud:.2f}')
    print(f'Run Test 3: Up/Down run count Test:        p={p_inc_dec:.2f}')
    if plot:
        plt.plot(obs[1:], obs[0:-1], '.')
        plt.show()
        plot_corr(obs)

    return p_chi, T_kol, p_ab_median, p_ud, p_inc_dec


if __name__ == '__main__':
    obs = stats.uniform.rvs(size=10_000)
    all_test(obs)
```

sales.py

In [ ]:

```
import numpy as np
from scipy import stats
import random

def gen_stations(n, min = 0, max = 200):
    stations = np.zeros((2,n))
    for i in range(n):
        stations[0][i]=random.randint(min,max)
        stations[1][i]=random.randint(min,max)
    return stations

def euclDist(a,b):
    dist=np.sqrt(np.power(b[0]-a[0],2)+np.power(b[1]-a[1],2))
    return dist
```

eventBis.py

In [ ]:

```
from dataclasses import dataclass
from enum import Enum, auto
from math import factorial
from matplotlib import pyplot as plt
```

```python
import numpy as np
from scipy import stats
import seaborn as sns


class State(Enum):
    INCOMING = auto()
    IN_SERVICE = auto()
    SERVICED = auto()
    BLOCKED = auto()

@dataclass
class Event:
    state: State
    arrival_time: int
    departure_time: int


class EventList:

    events: 'list[Event]'
    in_service: 'list[Event]'

    def __init__(self, arrival_time_distribution: stats.rv_continuous, service_
        self._arr_dist = arrival_time_distribution
        self._serv_dist = service_time_distribution
        self.events = self.generate_events(number_of_events)
        self.time_line = {}
        self.in_service = []

    @property
    def states(self):
        return [event.state for event in self.events]

    def update_in_service(self, time):
        for event in self.in_service:
            if event.departure_time <= time:
                event.state = State.SERVICED

        self.in_service = [event for event in self.events if event.state == St

    def generate_events(self, number_of_events: int) -> 'list[Event]':
        arr_times = self._arr_dist.rvs(size=number_of_events)
        arr_times = np.cumsum(arr_times)

        serv_times = self._serv_dist.rvs(size=number_of_events)
        dep_times = arr_times + serv_times
        return [Event(State.INCOMING, arr, dep) for arr, dep in zip(arr_times,

    def update_timeline(self, time):
        self.time_line[time] = self.states

    def __str__(self):
        str = ''
        for iter, (time, states) in enumerate(self.time_line.items()):
            if iter < len(self.events) - 10:
                continue
            str += f'TIME: {time:.2f}\n'
            for i, state in enumerate(states):
                if state != State.INCOMING:
```

```
                    str += f'Obs. {i}: {state.name}\n'
            str += '\n'
        return str

class BlockingEventSimulation:

    def __init__(self, arrival_time_distribution: stats.rv_continuous, service_

        self.arrival_dist = arrival_time_distribution
        self.service_dist = service_time_distribution

    def simulate(self, max_events: int, service_units: int):
        blocked_count = 0
        event_list = EventList(self.arrival_dist, self.service_dist, max_event
        for event in event_list.events:
            time = event.arrival_time
            event_list.update_in_service(time)
            if len(event_list.in_service) < service_units:
                event.state = State.IN_SERVICE
                event_list.update_in_service(time)
            else:
                event.state = State.BLOCKED
                blocked_count += 1


            event_list.update_timeline(time)

        return blocked_count / max_events


def calculate_theoretical_block_pct(m, a):
    return (a**10/factorial(m))/ sum([a**i / factorial(int(i)) for i in range(

if __name__ == '__main__':
    arr = stats.expon()
    serv = stats.expon(scale=8)
    sim = BlockingEventSimulation(arr, serv)
    event, block_pct = sim.simulate(10_000, 10)
    a=8
    print((a**10/factorial(10))/ sum([a**i / factorial(int(i)) for i in range(
```