Exercise 1

In [ ]:
```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
#import chi squared
from scipy.stats import chisquare
from scipy.stats import chi2
from scipy.stats import kstest
from tests import do_all_tests, LCG, chisquare_test, KS_test, run_test_1, run_te
import plotly.io as pio
#pio.renderers.default = "notebook+pdf"
```

Generate 10.000 (pseudo-) random numbers and present these numbers in a histogramme (e.g. 10 classes).

In [ ]:
```python
M = 2**31
a = 1103515245
c = 12345
N = 10000
k = 20
xval = 68
```

1

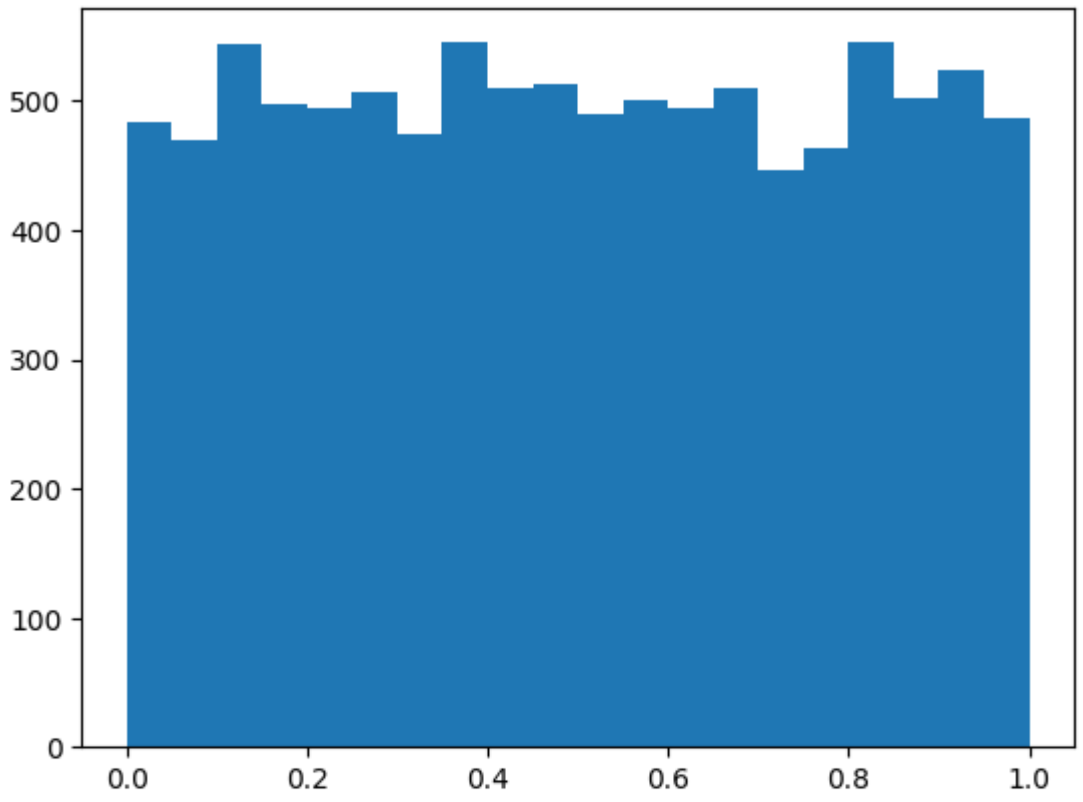
Write a program implementing a linear congruential generator (LCG). Be sure that the program works correctly using only integer representation.

a)

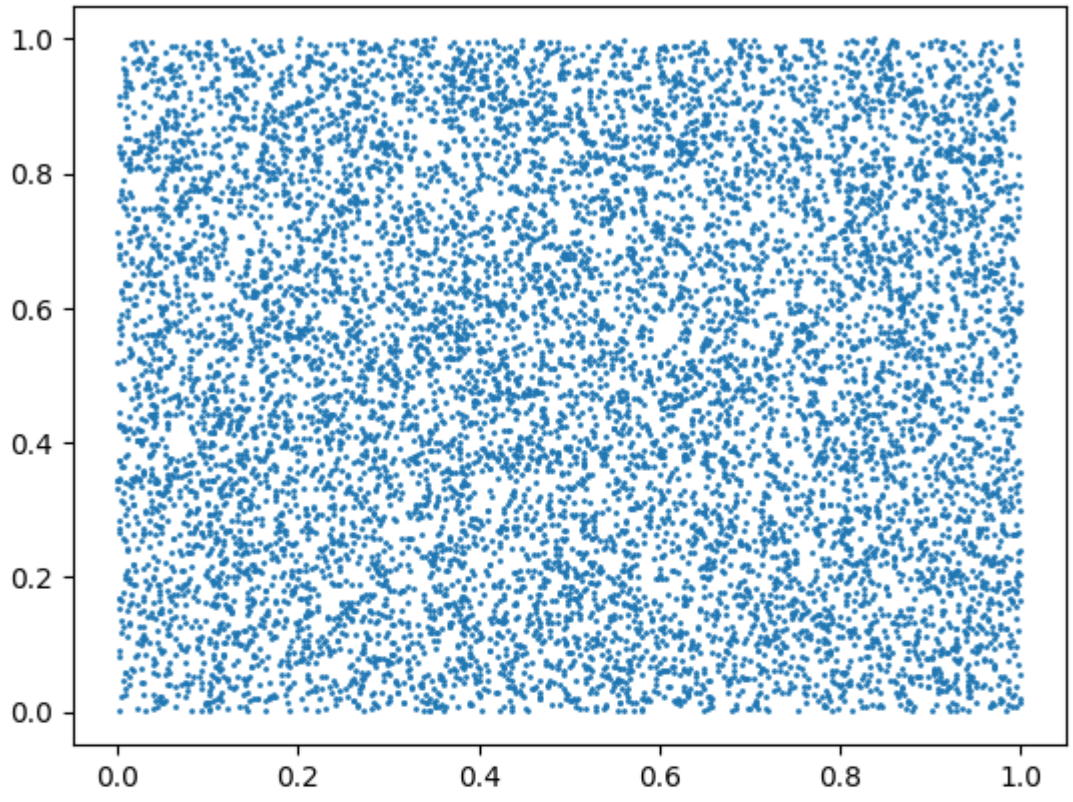
Generate 10.000 (pseudo-) random numbers and present these numbers in a histogramme (e.g. 10 classes).

Answer The numbers generated are constructed using the LCG algorithm, with the values above.

In [ ]:
```python
randn = LCG(xval, M,a,c, N)
#plot histogram of 20 classes
plt.hist(randn, bins=k)
plt.show()
```



In [ ]:
```
#scatter plot of random numbers U(i) vs U(i+1)
plt.scatter(randn[:-1],randn[1:], s=1)
plt.show()
```



Looks pretty random

In [ ]:
```
testvals = do_all_tests(randn)
```

```
KS test: D = 0.6323618066651898 p-value = 0.8187600805159896
Chi-square test: test = 518.4000000000001 p-value = 0.2652145064927258
Run test 1: test = 5026 p-value = 0.3085287356072869
Run test 2: test = 7.7058683332958795 p-value = 0.26045374876100236
Run test 3: test = -0.98039492696875 p-value = 0.32689121293197587
Correlation coefficient: c = 0.24905244657893708 p-value = 0.6673630775178978
```

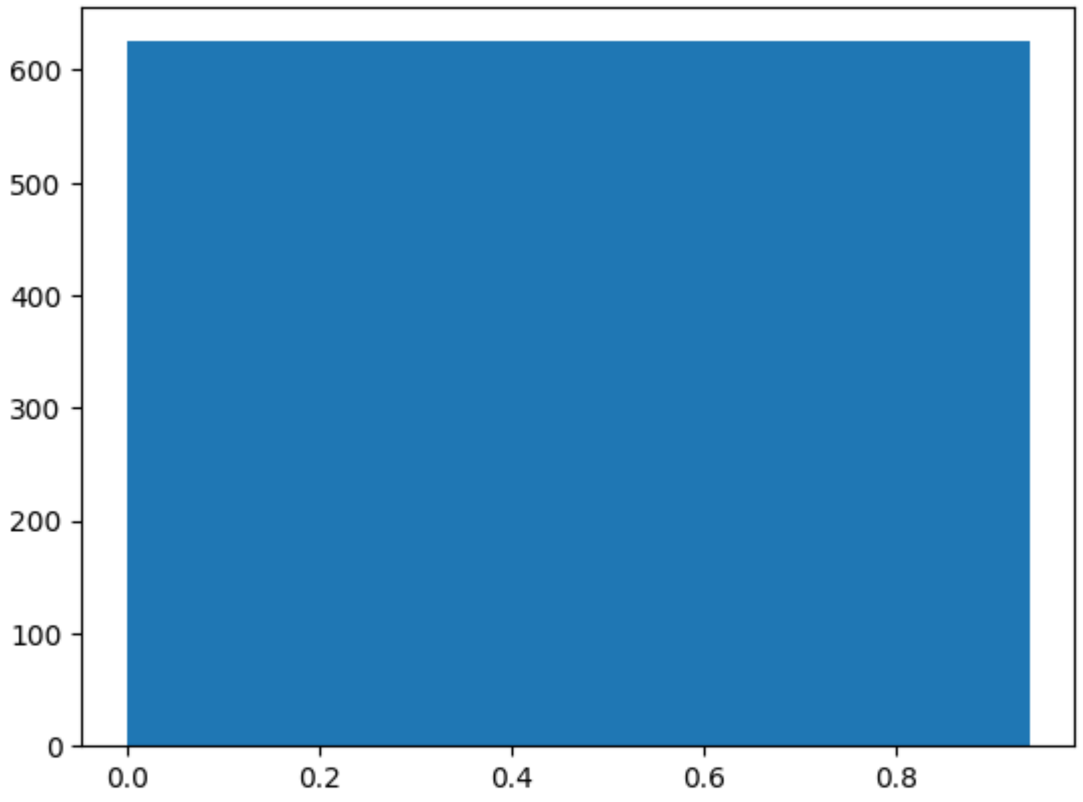
All statistical tests are have a p-value greater than 0.05, which means that the numbers generated can be considered random. The only p-value that is even close to 0.05 is for the second run test, which would indicate that there's not sufficient alternation between up/down values.

c)

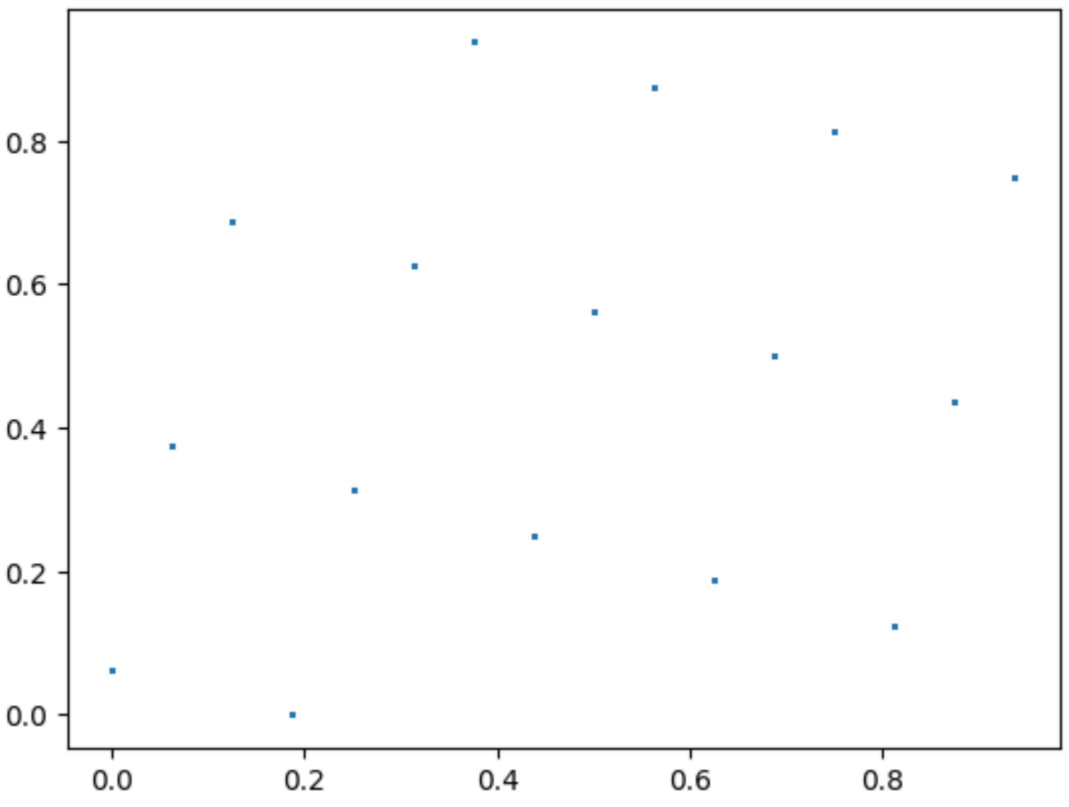
Repeat (a) and (b) by experimenting with different values of “a”, “b” and “M”. In the end you should have a decent generator. Report at least one bad and your final choice

In [ ]:
```
#Bad choices
M = 16
a = 5
c = 1
N = 10000
k = 16
xval = 3
```

In [ ]:
```
randn = LCG(xval, M,a,c, N)
#plot histogram of 20 classes
plt.hist(randn, bins=16)
plt.show()
```



In [ ]:
```
#scatter plot of random numbers U(i) vs U(i+1)
plt.scatter(randn[:-1],randn[1:], s=1)
plt.show()
```



Actually, even though values are uniformly distrubuted, the scatter plot shows that the sequence at which numbers are sampled, ISN'T random.

In [ ]:
```
testvals = do_all_tests(randn)
```

```
KS test: D = 6.247556640000001 p-value = 2.5020550203269963e-34
Chi-square test: test = 229375.0 p-value = 0.0
Run test 1: test = 3750 p-value = 1.0
Run test 2: test = 11846658.798212625 p-value = 0.0
Run test 3: test = -9.898826198103896 p-value = 0.0
Correlation coefficient: c = 0.216795703125 p-value = 0.0
```

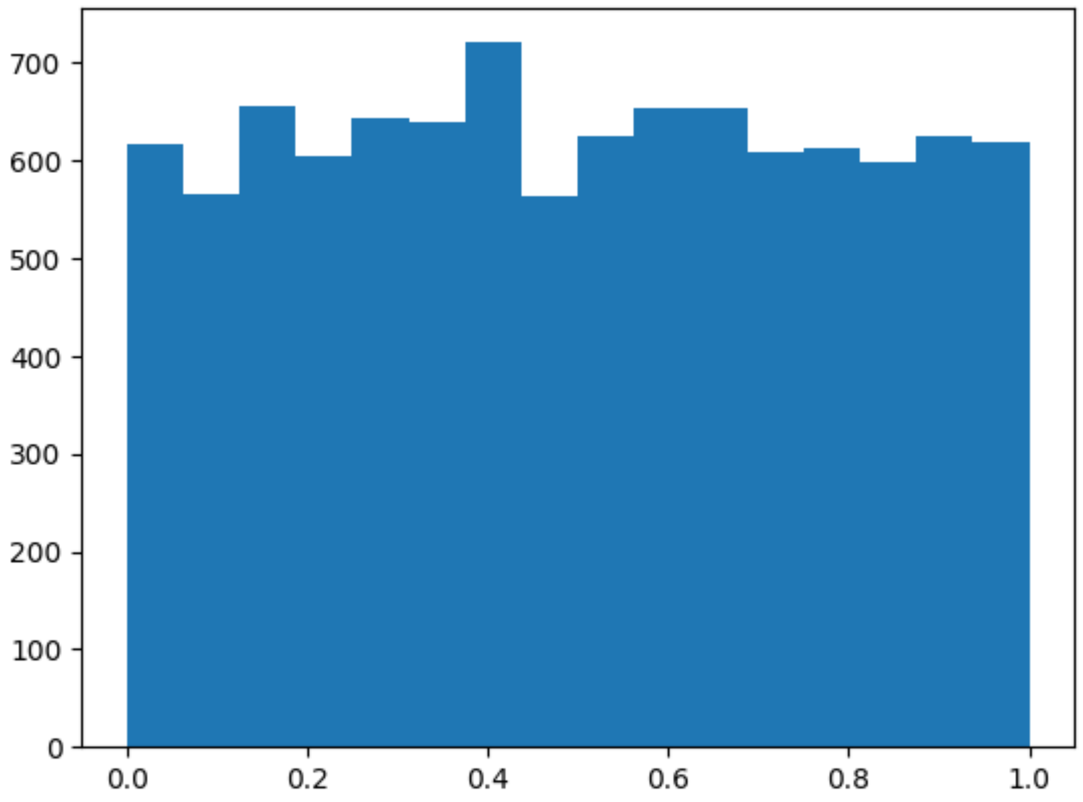
The only test statistic that indicates randomness is the KS_test (p-value = 1.0), but it only tests the distribution of the numbers, but doesn't test for independence. In conclusion, the former choice of LCG parameters were better for generating pseudo random numbers.

2

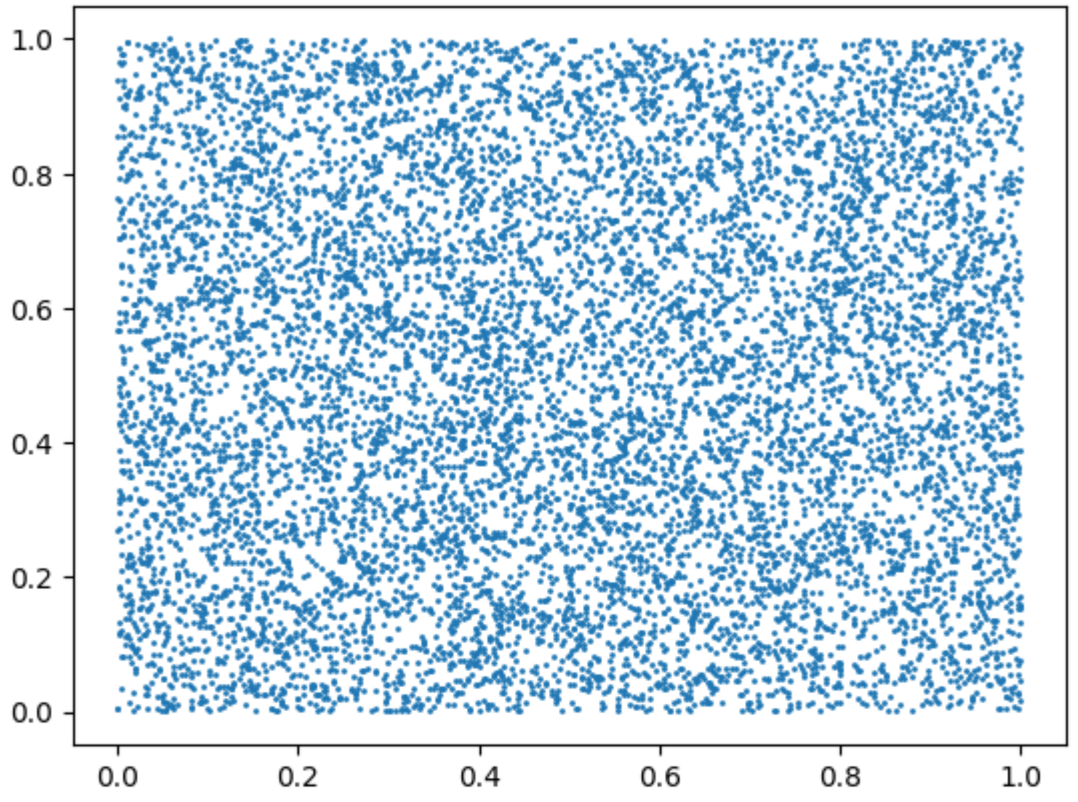
Apply a system available generator and perform the various statistical tests you did under Part 1 point (b) for this generator too

In [ ]:
```
#random numbers using numpy
randn = np.random.rand(N)
```

In [ ]:
```
#plot histogram of 20 classes
plt.hist(randn, bins=16)
plt.show()
```



In [ ]:
```
#scatter plot of random numbers U(i) vs U(i+1)
plt.scatter(randn[:-1],randn[1:], s=1)
plt.show()
```



In [ ]:
```
testvals = do_all_tests(randn)
```

```
KS test: D = 0.751059788786334 p-value = 0.6253831919474784
Chi-square test: test = 488.8999999999999 p-value = 0.6181981825120604
Run test 1: test = 4987 p-value = 0.6102666189733608
Run test 2: test = 7.624770150576844 p-value = 0.2669021412128363
Run test 3: test = 0.13440898192314327 p-value = 0.8930791795357036
Correlation coefficient: c = 0.2501406459956951 p-value = 0.9491366226687863
```

For the system available generator all test statistics are insignificant, which means that the numbers generated can be considered random.

3

You were asked to simulate one sample and perform tests on this sample. Discuss the sufficiency of this approach and take action, if needed.

Actually it is better to perform the tests on multiple samples, because one test isn't sufficient to determine randomness of the numbers generated. Assuming that the samples generated are random, p-values should be uniformly distributed, across all tests, so there's a chance to sample numbers that don't pass the randomness test.

In [ ]:
```
import numpy as np
from scipy.stats import uniform
from scipy.stats import norm
from scipy.stats import chi2
from scipy.special import kolmogorov

def LCG(xval, M, a, c, N):
    x = np.zeros(N)
    for i in range(N):
        xval = (a*xval + c) % M
        x[i] = xval
    x/=M
    return x

def KS_test(randn):
    Ftrue = np.arange(0,1,1/len(randn))
    #import uniform distribution cdf
    F = np.sort(uniform.cdf(randn))
    #calculate max difference
    D = np.max(np.abs(Ftrue - F))
    n = len(randn)
    D *= (np.sqrt(n)+0.12+0.11/np.sqrt(n))
    #calculate p value
    pval = 1 - np.exp(-2*n*D**2)
    pval = kolmogorov(D)
    return D, pval

#calculate chi squared
def chisquare_test(randn, k):
    n = len(randn)
    p = 1/k
    test = 0
    for i in range(k):
        xval = randn[(i/k < randn)*(randn < (i+1)/k)]
        ni = len(xval)
        test += (ni - n*p)**2/(n*p)
    pval = 1 - chi2.cdf(test, k-1)
    return test, pval

def run_test_1(randn):
    #median value
    median = np.median(randn)
    #number of observations below median
    possamps = randn < median
    negsamps = randn > median
    posruns = 0
    negruns = 0
    n1 = np.sum(possamps)
    n2 = np.sum(negsamps)
    n = n1+n2
    for i in range(len(randn)):
        if i == 0:
            continue
        if possamps[i] != possamps[i-1] and possamps[i] == True:
            posruns += 1
        if negsamps[i] != negsamps[i-1] and negsamps[i] == True:
            negruns += 1
    T = posruns + negruns
    #normal cdf

    mean = 2*n1*n2/n + 1
    var = np.exp(np.log(2)+np.log(n1)+np.log(n2)+np.log(2*n1*n2 - n)-np.log(n**2
    Z = (T-mean)/np.sqrt(var)
    pval = 1 - norm.cdf(Z)
    return T, pval

def run_test_2(randn):
    # up/down
    run_lengths = []
    current_run_length = 1
    for i in range(1, len(randn)):
        if randn[i] > randn[i-1]:
            current_run_length = min(current_run_length + 1, 6)
        else:
            run_lengths.append(current_run_length)
            current_run_length = 1
    #get vector of count for each run length
    R, _ = np.histogram(run_lengths, bins=6)

    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]
        ])
    B = np.array([1/6, 5/24, 11/120, 19/720, 29/5040, 1/840])
    n = len(randn)
    Z = (R-B*n).T@A@(R-B*n)/(n-6)
    #chisquare test
    pval = 1 - chi2.cdf(Z, 6)
    return Z, pval

def run_test_3(randn):
    n = len(randn)
    updown = np.zeros(n-1, dtype=bool)
    for i in range(1, n):
        updown[i-1] = randn[i] > randn[i-1]
    #count number of runs
    runs = []
    current_run = 1
    for i in range(1, n-1):
        if updown[i] != updown[i-1]:
            runs.append(current_run)
            current_run = 1
        else:
            current_run += 1
    # number of unique runs
    X = len(runs)
    Z = (X - (2*n-1)/3)/np.sqrt((16*n-29)/90)
    p = 2*(1 - norm.cdf(abs(Z)))
    return Z, p

def correlation_coefficient(randn, h=2):
    c = np.sum(randn[:-h]*randn[h:])/len(randn)
    Z = (c - 1/4)/np.sqrt(7/(144*len(randn)))
    p = 2*(1 - norm.cdf(abs(Z)))
    return c, p

def do_all_tests(randn):
    D, pval1 = KS_test(randn)
    test1, pval2 = chisquare_test(randn, len(randn)//20)
    test2, pval3 = run_test_1(randn)
    test3, pval4 = run_test_2(randn)
    test4, pval5 = run_test_3(randn)
    c, pval6 = correlation_coefficient(randn)
    # print results
    print('KS test: D =', D, 'p-value =', pval1)
    print('Chi-square test: test =', test1, 'p-value =', pval2)
    print('Run test 1: test =', test2, 'p-value =', pval3)
    print('Run test 2: test =', test3, 'p-value =', pval4)
    print('Run test 3: test =', test4, 'p-value =', pval5)
    print('Correlation coefficient: c =', c, 'p-value =', pval6)
    # dictionary of results
    results = {'KS test': (D, pval1), 'Chi-square test': (test1, pval2), 'Run te
    return results
```

Exercise 4

Write a discrete event simulation program for a blocking system, i.e. a system with m service units and no waiting room. The offered traffic A is the product of the mean arrival rate and the mean service time

1

The arrival process is modelled as a Poisson process. Report the fraction of blocked customers, and a confidence interval for this fraction. Choose the service time distribution as exponential. Parameters: m = 10, mean service time = 8 time units, mean time between customers = 1 time unit (corresponding to an offered traffic of 8 Erlang), 10 x 10.000 customers.

In [ ]:
```
import numpy as np
#import poission
import math
from scipy.stats import poisson
#import exponential
from scipy.stats import expon
import bisect
from discrete_event import Customer, main_loop, confidence_intervals, erlang_b
```

In [ ]:
```
m = 10 #number of servers
s = 8 #mean service time
lam = 1#arrival_intensity
total_customers =10000 #10*10000
A = lam*s
```

In [ ]:
```
#arrival time differences are exponentially distributed
np.random.seed(1)
arrival_interval = lambda : np.random.exponential(1/lam, size = total_customers)
service_time =lambda : expon.rvs(scale = s, size = total_customers)
```

In [ ]:
```
#Amount of people blocked in the system
blocked_1 = main_loop(arrival_interval, service_time, m)
```

In [ ]:
```
print("Blocking probability: ", blocked_1/total_customers)
print("Mean blocking probability: ", np.mean(blocked_1/total_customers))
```

```
Blocking probability:  [0.1293 0.1192 0.117  0.1172 0.1246 0.1248 0.1026 0.1302 0
.1202 0.1262]
Mean blocking probability:  0.12113000000000003
```

In [ ]:
```
#Theoretical blocking probability
print("Theoretical blocking probability",erlang_b(m, A))
```

```
Theoretical blocking probability 0.12166106425295149
```

Answer

According to the discrete event simulation, the fraction of blocked customers is 0.1211 which corresponds well with the theoretical value of 0.1216.

2

The arrival process is modelled as a renewal process using the same parameters as in Part 1 when possible. Report the fraction of blocked customers, and a confidence interval for this fraction for at least the following two cases

In [ ]:
```
# (a) Experiment with Erlang distributed inter arrival times The
#Erlang distribution should have a mean of 1
np.random.seed(1)
inter_arrival = lambda : np.random.gamma(2, 0.5, size = total_customers)
service_time = lambda : expon.rvs(scale = s, size = total_customers)
blocked_erlang = main_loop(arrival_interval, service_time, m)
print("Blocking probability: ", blocked_erlang/total_customers)
print("Mean blocking probability: ", np.mean(blocked_erlang/total_customers))
```

```
Blocking probability:  [0.1293 0.1192 0.117  0.1172 0.1246 0.1248 0.1026 0.1302 0
.1202 0.1262]
Mean blocking probability:  0.12113000000000003
```

Answer

When the inter arrival time is Erlang distributed with mean 1 time unit, the fraction of blocked customers is 0.1211 which does correspond with the theoretical value of 0.1216.

In [ ]:
```
# hyper exponential inter arrival times. The parameters for
#the hyper exponential distribution should be
np.random.seed(1)
p1 = 0.8
λ1 = 0.8333
p2 = 0.2
λ2 = 5.0
s = 8
arrival_interval = lambda : np.random.choice([expon.rvs(scale = 1/λ1), expon.rvs

service_time = lambda : expon.rvs(scale = s, size = total_customers)

blocked_hyperexp = main_loop(arrival_interval,service_time, m)
print("Blocking probability: ", blocked_hyperexp/total_customers)
print("Mean blocking probability: ", np.mean(blocked_hyperexp/total_customers))
```

```
Blocking probability:  [0.3505 0.     0.81   0.474  0.5987 0.0033 0.1212 0.     0
.0347 0.8817]
Mean blocking probability:  0.32741
```

Answer For hyperexponential inter arrival time with mean 1 time unit, the fraction of blocked customers is 0.32741 which does not correspond with the theoretical value of 0.1216.

3

The arrival process is again a Poisson process like in Part 1. Experiment with different service time distributions with the same mean service time and m as in Part 1 and Part 2

a)

Constant service time

```
In [ ]: # a) Constant service time
        np.random.seed(1)
        arrival_interval = lambda : np.random.exponential(1/lam, size = total_customers)
        service_time = lambda : s*np.ones(total_customers)

        blocked_constant = main_loop(arrival_interval,service_time, m)
        print("Blocking probability: ", blocked_constant/total_customers)
        print("Mean blocking probability: ", np.mean(blocked_constant/total_customers))
```

```
Blocking probability:  [0.1275 0.1158 0.1224 0.1271 0.1214 0.1166 0.1185 0.1242 0
.1255 0.1169]
Mean blocking probability:  0.12159
```

Answer

When the service time is constant, the fraction of blocked customers is 0.12159 which corresponds well with the theoretical value of 0.1216.

```
In [ ]: # Pareto distributed service times with at least k = 1.05 and
        #k = 2.05.
        np.random.seed(1)
        def pareto():
            beta = (k-1)/(k)*8
            Us = np.random.uniform(0, 1, total_customers)
            xs = beta/(Us**(1/k))
            return xs

        k = 1.05
        service_time = lambda : np.random.pareto(k, total_customers)
        service_time = pareto
        blocked_pareto_1 = main_loop(arrival_interval, service_time, m)
        print("Blocking probability for k= 1.05: ", blocked_pareto_1/total_customers)
        print("Mean blocking probability: ", np.mean(blocked_pareto_1/total_customers))
        k = 2.05
        service_time = lambda : np.random.pareto(k, total_customers)
        service_time = pareto
        blocked_pareto_2 = main_loop(arrival_interval, service_time, m)
        print("Blocking probability for k= 2.05: ", blocked_pareto_2/total_customers)
        print("Mean blocking probability: ", np.mean(blocked_pareto_2/total_customers))
```

```
Blocking probability for k= 1.05:  [0.0016 0.0004 0.0023 0.0006 0.0004 0.0006 0.0
008 0.0026 0.0034 0.002 ]
Mean blocking probability:  0.00147
Blocking probability for k= 2.05:  [0.122  0.1216 0.1173 0.1044 0.122  0.1268 0.1
172 0.1195 0.1223 0.113 ]
Mean blocking probability:  0.11861
```

Answer

When the service time is pareto distributed with k=1.05 the mean blocking fraction is 0.00147 which is not at all close to the theoretical value of 0.1216. For k=2.05 the blocking fraction is 0.11861.

To have an accurate mean we change β to be $\beta = \frac{k-1}{k} \cdot 8$, to ensure a mean service time of 8 time units. The result using $k = 1.05$ is heavily skewed towards not rejecting customers. The Pareto distribution with small k is difficult to sample enough large values from, to actually see a mean service time of 8 time units, so we see a lot of small service times, resulting in few blocks. The effect is gone once $k > 2$.

In [ ]:
```
#absolute gaussian distributed service times with mean s and standard deviation
np.random.seed(1)
service_time = lambda : np.random.normal(s, s/4, size = total_customers)
blocked_gauss = main_loop(arrival_interval, service_time, m)
print("Blocking probability: ", blocked_gauss/total_customers)
print("Mean blocking probability: ", np.mean(blocked_gauss/total_customers))
```

```
Blocking probability:  [0.1291 0.1198 0.1193 0.1172 0.1159 0.1188 0.1258 0.1189 0
.1295 0.1292]
Mean blocking probability:  0.12235
```

Answer

When the service time is normally distributed with mean 8 time units and standard deviation 2 time units, the fraction of blocked customers is 0.12235 which corresponds well with the theoretical value of 0.1216.

4

Compare confidence intervals for Parts 1, 2, and 3 then interpret and explain differences if any.

In [ ]:
```
#show confidence intervals for all the experiments
p = erlang_b(m, A)

bs = np.array([blocked_1, blocked_erlang, blocked_hyperexp, blocked_constant, bl
bs = bs / total_customers
titles = ["Exponential", "Erlang", "Hyper exponential", "Constant", "Pareto k=1.
#print("Theoretical blocking probability",erlang_b(m, A))
#print("Confidence intervals for blocking probability")

for i, b in enumerate(bs):
    print(f"{titles[i]}: ")
    print("CI is:", confidence_intervals(b))
    if p > confidence_intervals(b)[0] and p < confidence_intervals(b)[1]:
        print("Which contains the theoretical value")
    else:
        print("Which does not contain the theoretical value")
    #print("\n")

# print("Part 1: ", confidence_intervals(blocked_1/total_customers))
# if p > confidence_intervals(blocked_1/total_customers)[0] and p < confidence_i
#     print("Which contains the theoretical value")
# else:
#     print("Which does not contain the theoretical value")
# print("Part 2 (Erlang distribution): ", confidence_intervals(blocked_erlang/to
# print("part 3 (Hyper exponential distribution): ", confidence_intervals(blocke
# print("Part 4 (Constant service time): ", confidence_intervals(blocked_constan
# print("Part 5 (Pareto distribution k=1.05): ", confidence_intervals(blocked_pa
# print("Part 5 (Pareto distribution k=2.05): ", confidence_intervals(blocked_pa
# print("Part 6 (Gaussian distribution): ", confidence_intervals(blocked_gauss/t
```

```
Exponential:
CI is: (0.11640692459344552, 0.12585307540655452)
Which contains the theoretical value
Erlang:
CI is: (0.11640692459344552, 0.12585307540655452)
Which contains the theoretical value
Hyper exponential:
CI is: (0.12340100412897476, 0.5314189958710251)
Which does not contain the theoretical value
Constant:
CI is: (0.11897467394308091, 0.1242053260569191)
Which contains the theoretical value
Pareto k=1.05:
CI is: (0.0008427693757476441, 0.002097230624252356)
Which does not contain the theoretical value
Pareto k=2.05:
CI is: (0.11494051042459581, 0.12227948957540417)
Which contains the theoretical value
Gaussian:
CI is: (0.11916570869423038, 0.12553429130576962)
Which contains the theoretical value
```

All distributions except for hyperexponential and pareto with $k = 1.05$ contain the theoretical value in their confidence interval

In [ ]:
```
import numpy as np
import bisect
import math


class Customer:
    def __init__(self, arrival_time, service_time):
        self.service_time = service_time
        self.blocked = False

        self.event = "arrival"
        self.event_time = arrival_time


    def arrive(self, servers, event_list):
        if servers < 1:
            self.blocked = True
            return servers
        else:
            servers -= 1
            servers = max(servers, 0)
            self.event = "departure"
            self.event_time += self.service_time
            bisect.insort(event_list, self, key=lambda x: x.event_time)
            return servers

    def depart(self, servers, m):
        servers += 1
        servers = min(servers, m)
        return servers


def main_loop(arrival_interval, service_time, m, repititions = 10):
    blocked = np.zeros(repititions)
    for i in range(repititions):
        arrival_intervals = arrival_interval()
        service_times = service_time()
        arrival_times = np.cumsum(arrival_intervals)
        event_list = [Customer(arrival_times[i],service_times[i]) for i in range
        event_list.sort(key=lambda x: x.event_time)
        open_servers = m
        while event_list:
            event = event_list.pop(0)
            if event.event == "arrival":
                open_servers = event.arrive(open_servers, event_list)
                blocked[i] += event.blocked
            elif event.event == "departure":
                open_servers = event.depart(open_servers, m)
    return blocked


def confidence_intervals(samples):
    emp_mean = np.mean(samples)
    emp_std = np.std(samples)
    t = 1.96
    return (emp_mean - t*emp_std/np.sqrt(len(samples)), emp_mean + t*emp_std/np.

#Erlang B formula
```

```python
def erlang_b(m, A):
    return (A**m/math.factorial(m))/np.sum([A**i/math.factorial(i) for i in rang
```

Exercise 5 - Variance redudiction Methods

The tasks centers around simlation of the integral $\int_0^1 e^x \, \mathrm{d}x$

```
In [ ]: import numpy as np
        import numpy.random as rnd
        import scipy.stats as stats
        import math as math
```

UNCHANGED

1) Simulation of the integral using a Crude Monte Carlo Method

Using from the slides that the integral can be simplified to the expectation of e^U where $U \sim Uniform(0,1)$. Simulate $n = 100$ instances of e^x, and find the mean. To get a 95% CI, use quantiles from t distribution with $n-1$ dof and $\alpha = 0.05$.

$$CI = \left[\bar{\theta} + \frac{S_\theta}{\sqrt{n}} t_{\frac{\alpha}{2}}(n-1); \bar{\theta} + \frac{S_\theta}{\sqrt{n}} t_{1-\frac{\alpha}{2}}(n-1)\right]$$

```
In [ ]: def crudeMC(n):
            U = rnd.uniform(size = n)
            x = np.exp(U)
            return x

        def meanVar(x):
            mean = np.mean(x)
            var = np.var(x)
            return mean, var

        def tConf(x, alpha, string):
            mean, var = meanVar(x)
            dof = len(x) - 1
            s = np.sqrt(var / len(x))
            a = stats.t.ppf(alpha/2, dof)
            b = stats.t.ppf(1 - alpha/2, dof)
            print("For", string, ":")
            print(f"Mean is {round(mean,3)}, with 95% confidence interval [{round(mean +

        alpha = 0.05
        n = 100
        Xs_crude = crudeMC(n)
        tConf(Xs_crude, alpha, "Crude Monte Carlo" )
        print(f"The true value of the integral is: {round(np.exp(1) - np.exp(0),4)}")
```

```
For Crude Monte Carlo :
Mean is 1.763, with 95% confidence interval [1.67,1.857]
The true value of the integral is: 1.7183
```

UNCHANGED

2) Antithetic variables

We can use Anithetic variables to exploit the fact that the integral $\int_{-\infty}^{\infty} e^x \,\mathrm{d}x$ is monotonely increasing. In practice this means we can use a single uniformly distributed value $U \sim Uniform(0,1)$, as in the Crude Monte Carlo Method, an use it to create a second uniformly distributed value $U-1$, almost for free.

Using these we can create 2 estimates of the integral by e^U and e^{U-1}, and take the average of these two to get a much more robust estimate of the integral

$$Y_i = \frac{e^{U_i - e^{U_i - 1}}}{2}.$$

The expectation of the integral is now

$$E\left(Y_i\right).$$

Y_i which on the slides is proved to have variance $\frac{1}{4}Var\left(e^{U_i}\right) + \frac{1}{4}Var\left(e^{1-U_i}\right) + \frac{1}{2}Cov\left(e^{U_i}, e^{1-U_i}\right)$. As e^{U_i} and e^{1-U_i} are obviosly negatively correlated, this variance is much lower than before.

By rewriting Y_i computing cost can be lowered, to only calculate a single exponential, resulting in an only marginally more expensive computation for n Y's compared to n X's.

$$Y_i = \frac{e^{U_i + \frac{e}{e^{U_i}}}}{2}.$$

Note: Had the integral not been monotonely increasing, say we had attemted to estimate some other function f, we could have had a situation where $f\left(U_i\right)$ and $f\left(U_i - 1\right)$ had been positively correlated, which could lead to a higher variance on Y_i. For the exponential function one of e^{U_i} and $e^{U_i - 1}$ is always large, and one is always small.

```
In [ ]: def antithetic(n):
            U = rnd.uniform(size = n)
            t = np.exp(U)
            Ys = 0.5 * (t + np.exp(1) / t )
            return Ys
        Xs_antithetic = antithetic(100)
        tConf(Xs_antithetic, alpha, "Antithetics Variables")
```

```
For Antithetics Variables :
Mean is 1.717, with 95% confidence interval [1.7,1.729]
```

UNCHANGED

3) Control Variates

We can use the variable

$$Z = X + c\left(Y - \mu_i\right),$$

instead of X_i as the estimate of the integral. It can then be shown that $E\left(X\right) = E\left(Z\right)$, since

$$E\left(Z\right) = E\left(X\right) + E\left(c\left(Y - \mu_i\right)\right)$$
$$= E\left(X\right) + c\left(E\left(Y\right) - E\left(\mu_i\right)\right) = E\left(X\right) + c\left(\mu_i - \mu_i\right) = E\left(X_i\right).$$

From the slides we also know that choosing the optimal $c = -\frac{Cov(X,Y)}{Var(Y)}$, results in a variance of Z

$$Var(Z) = Var(X) - \frac{Cov(X,Y)^2}{Var(Y)}.$$

For this specific problem we are given $X_i = e^{U_i}$, and it is natural to choose $Y_i = U_i$. This again exploits the covariance between U_i and X_i, though this time negative correlation is not a requirement, as the covariance is squared. It should just be non-zero. As $E\left(U_i\right) = \frac{1}{2}$ we get

$$Z_i = e^{U_i} + c\left(U_i - \frac{1}{2}\right),$$

with $c \approx 0.14086$.

```
In [ ]: def controlVariates(n, alpha):
            U = rnd.uniform(size = n)
            Xs = np.exp(U)
            dof = len(Xs) - 1
            meanZ = np.mean(Xs)
            cov = np.mean(U * Xs) - np.mean(U) * np.mean(Xs)
            varZ = np.var(Xs) - cov**2 / np.var(U)
            s = np.sqrt(varZ / len(Xs))
            a = stats.t.ppf(alpha/2, dof)
            b = stats.t.ppf(1 - alpha/2, dof)
            print("For Control Variates")
            print(f"Mean is {round(meanZ,2)}, with 95% confidence interval [{round(meanZ
        controlVariates(n, alpha)
```

```
For Control Variates
Mean is 1.71, with 95% confidence interval [1.7,1.72]
```

UNCHANGED

4) Statified Sampling

Stratified sampling attemts to lower variance by (almost) gaurantying samples from the entire sample space, by cutting the sample space into m pieces, and then sampling a number of variables from each. Here we´ll use $m = 10$ even intervals, to ensure even computational cost to the other methods, 10 U_i's are generated in each interval.

The method then takes one U_i from each interval and creates a single variable W_i

$$W_i = \frac{\sum_{k=1}^{m} X_{i,k}}{m},$$

where k now denotes which interval the X_i comes from. Still $X_i = e^{U_i}$, or $X_{i,k} = e^{U_{i,k}}$.

```
In [ ]: def stratSamples(n):
            num_ints = 10
            k = 1 / num_ints
            Xs = np.zeros(n)
            Us = np.zeros(num_ints)
            for j in range(n):
                for i in range(num_ints):
                    a = k * i
                    b = a + k
                    U = rnd.uniform(a, b, size = 1)
                    Us[i] = U
                Xs[j] = np.mean(np.exp(Us))
            return Xs
        n = 10
        Xs_stratified = stratSamples(n)
        tConf(Xs_stratified, alpha, "Stratified Sampling")
```

```
For Stratified Sampling :
Mean is 1.72, with 95% confidence interval [1.71,1.733]
```

1-4) Observations

The Crude MC method gives a very wide CI All the variance reduction methods have succesfully narrowed the CI. We had the largest success with Control Variates.

CHANGED

5) Use control variates to reduce the variance of the estimator in exercise 4

We'll use the arrival intervals as control variate, i.e. control variate $C_i \sim Exp(\lambda)$, note the difference between arrival intervals, which are just sampled, and arrival times which are the actual times customers arrive at, which is the cummulative arrival intervals.

Using the formula

$$Z_i = X_i + c\,(C_i - 1)\,,$$

X_i is a list of bools of wether the customer with arrival interval C_i. Correlation is expected between X_i and C_i as a customer arriving quickly after another (small C_i) would presumably be blocked more often, as other customers have not had time to be served yet. We ofcourse know the mean of C_i, and can find the optimal $c = \frac{Cov(X,C)}{Var(C)}$ in each iteration.

We're concerned with the mean of X_i, which is the blocking probability, and Z_i is constructed to have the same mean.

```
In [ ]: import numpy as np
        import numpy.random as rnd
        import scipy.stats as stats
        import math as math
        from scipy.stats import expon
        import numpy as np
        from discrete_event import Customer, main_loop, confidence_intervals, erlang_b,

        #arrival time differences are exponentially distributed
        lam = 1
        total_customers = 10000
        m = 10
        s = 8
        repititions = 10
        #arrival time differences are exponentially distributed
        arrival_interval = lambda : np.random.exponential(1/lam, size = total_customers)
        service_time =lambda : expon.rvs(scale = s, size = total_customers)
```

```
In [ ]: blocked = main_loop(arrival_interval, service_time, m, repitittions = repitittions
        #print("Blocking probability", blocked/total_customers * 100)
        #confidence interval for the mean
        theta = np.mean(blocked)
        confint = confidence_intervals(blocked)
        print(f"Estimated blocking probability {round(theta,3)*100}%\nTrue blocking prob
        print(f"95% CI for mean of blocking probability [{round(confint[0],4)}, {round(c
        print("Interval width", round(confint[1]-confint[0],4))
```

```
Estimated blocking probability 12.3%
True blocking probability 12.2% (From Erlang formula)
95% CI for mean of blocking probability [0.1194, 0.1259]
Interval width 0.0065
```

```
In [ ]: np.random.seed(2)
        n = 10
        Zs = []

        for i in range(n):
            arrival_intervals = np.random.exponential(1/lam, size = total_customers)
            service_times = expon.rvs(scale = s, size = total_customers)
            blocked = main_loop_array(arrival_intervals, service_times, m)#/total_custom
            # Construct new variable
            c = -np.cov(arrival_intervals, blocked)[0,1] / np.var(arrival_intervals)
            Z = blocked + c * (arrival_intervals - 1)
            Zs.append(np.mean(Z))

        tehta_Z = np.mean(Zs)
        confint_Z = confidence_intervals(Zs)
        print(f"Estimated blocking probability {round(theta,3)*100}%\nTrue blocking prob
        print(f"95% CI for mean of blocking probability [{round(confint_Z[0],4)}, {round
        print("Interval width", round(confint_Z[1]-confint_Z[0],4))
```

```
Estimated blocking probability 11.899999999999999%
True blocking probability 12.2% (From Erlang formula)
95% CI for mean of blocking probability [0.1208, 0.1251]
Interval width 0.0043
```

6) Reduce variance for the difference between solutions with

poisson arrivals and hyperexponential renewal process in exercise 4

Do this using Common Random Numbers (CRN). Using CRN we should be able to reduce the width of the CI og the difference in number of blocked customers between the two arrival processes.

Using non-CRN's it should require 5 differerent processes of generating arrivals and leaves to compare the two processes. 2 generates the serice times for the processes, one generates arrivals for the poisson process and 2 generates renewals for the hyperexponential arrivals.

When using CRN's this can be reduced to 3. The services times can be generated from the same U, since they should be the same here. We can use a single U as the exponential for the poisson and hyperexponential processes. We also need one more U to flip between the two exponentials in the hyperexponential distribution.

Both test runs predict more blocked customers when assuming a hyperexponential renewal process. The width of the CI when using CRN's can be reduced to $\sim 1/3$ compared to just simulating the two processes independently.

In [ ]:
```
import bisect as bisect
class Customer:
    def __init__(self, arrival_time, service_time):
        self.service_time = service_time
        self.blocked = False

        self.event = "arrival"
        self.event_time = arrival_time


    def arrive(self, servers, event_list):
        if servers < 1:
            self.blocked = True
            return servers
        else:
            servers -= 1
            servers = max(servers, 0)
            self.event = "departure"
            self.event_time += self.service_time
            bisect.insort(event_list, self, key=lambda x: x.event_time)
            return servers

    def depart(self, servers):
        servers += 1
        servers = min(servers, m)
        return servers

from distributions import getExponential, getUniform

def main_loop(event_list, m, repititions = 10):

    blocked = 0
    for i in range(repititions):
        event_list.sort(key=lambda x: x.event_time)
        open_servers = m
        while event_list:
            event = event_list.pop(0)
            if event.event == "arrival":
                open_servers = event.arrive(open_servers, event_list)
                blocked += event.blocked
            elif event.event == "departure":
                open_servers = event.depart(open_servers)
    return blocked

num_customers = 10000
lam = 1
s = 1/8
# Generate two eventlists
p1 = 0.8
lam1 = 0.8333
p2 = 0.2
lam2 = 5.0

def generateHexp(Us, Us2, p2, lam1, lam2):
    Xs = np.zeros(len(Us))
    for i in range(len(Us)):
        if Us2[i] < p2:
            Xs[i] = -np.log(Us[i]) / lam2
        else:
            Xs[i] = -np.log(Us[i]) / lam1
    return Xs

m = 10
rep = 5
blocked_hexp = np.zeros(rep)
blocked_poisson = np.zeros(rep)
blocked_poisson_common = np.zeros(rep)
blocked_hexp_common = np.zeros(rep)
for i in range(rep):

    Us1 = rnd.uniform(0,1,num_customers)
    Us2 = rnd.uniform(0,1,num_customers)
    Us3 = rnd.uniform(0,1,num_customers)
    service_intervals = -np.log(Us1) / s
    poisson_arrival_times = np.cumsum(-np.log(Us2) / lam)
    hexp_arrival_times = np.cumsum(generateHexp(Us2, Us3, p2, lam1, lam2))

    event_list_poisson = [Customer(poisson_arrival_times[i], service_intervals[i
    event_list_hexp = [Customer(hexp_arrival_times[i], service_intervals[i]) for

    blocked_poisson_common[i] = main_loop(event_list_poisson, m, repititions=1)
    blocked_hexp_common[i] = main_loop(event_list_hexp, m, repititions=1)

    Us1 = rnd.uniform(0,1,num_customers)
    Us2 = rnd.uniform(0,1,num_customers)
    Us3 = rnd.uniform(0,1,num_customers)
    Us4 = rnd.uniform(0,1,num_customers)
    Us5 = rnd.uniform(0,1,num_customers)
    service_intervals1 = -np.log(Us2) / s
    service_intervals2 = -np.log(Us3) / s

    poisson_arrival_times = np.cumsum(-np.log(Us1) / lam)
    hexp_arrival_times = np.cumsum(generateHexp(Us4, Us5, p2, lam1, lam2))
    event_list_poisson = [Customer(poisson_arrival_times[i], service_intervals1[
    event_list_hexp = [Customer(hexp_arrival_times[i], service_intervals2[i]) fo

    blocked_poisson[i] = main_loop(event_list_poisson, m, repititions=1)
    blocked_hexp[i] = main_loop(event_list_hexp,m, repititions=1)
```

In [ ]:
```
theta_crn = blocked_poisson_common - blocked_hexp_common
theta_irn = blocked_poisson - blocked_hexp

mean_theta_crn = round(np.mean(theta_crn),0)
var_theta_crn = np.var(theta_crn)
s = np.sqrt(var_theta_crn / len(theta_crn))
dof = len(theta_crn)-1
a = stats.t.ppf(alpha/2, dof)
b = stats.t.ppf(1 - alpha/2,dof)
print("For common random numbers")
print(f"Mean is {mean_theta_crn}, with confidence interval [{round(mean_theta_cr
print(f"Width of CI = {round(np.abs(mean_theta_crn + s * a - mean_theta_crn - s*


mean_theta_irn = int(np.mean(theta_irn))
var_theta_irn = np.var(theta_irn)
s = np.sqrt(var_theta_irn / len(theta_irn))
print("For independent random numbers")
print(f"Mean is {mean_theta_irn}, with confidence interval [{round(mean_theta_ir
print(f"Width of CI = {round(abs(mean_theta_irn + s * a - mean_theta_irn - s*b),
```

```
For common random numbers
Mean is -132.0, with confidence interval [-170.009,-93.991]
Width of CI = 76.017
For independent random numbers
Mean is -152, with confidence interval [-284.092,-19.908]
Width of CI = 264.184
```

7) Using a Crude Monte Carlo estimator vs Importance Sampling

Attempt to estimate the probality $Z > a$ for $Z \sim N(0,1)$ for $a = 2, 4$. Do this using a crude monte carlo estimator and with importance sampling.

Using importance sampling set $h(z) = z > a$. We'll choose $g(x)$ to also be a normal distribution, with the same variance as $f(x)$, though with mean a instead. This means g samples around a more often. We thus have to sample fewer points to get an accurate estimate.

Notice $n = 10000$ with importance sampling gives significant digits, while the Crude Estimator needs $n = 100000$ to get a significant digit. A lot of time can be saved.

As long as a is small enough e.g. $a = 2$ Importance Sampling does not seem nescesarry.

In [ ]:
```
def crudeMonteCarloNorm(a,n):
    Us = rnd.normal(size = n)
    return sum(Us > a) / n
n = 10000
n2 = 100000
p1 = crudeMonteCarloNorm(2, n2)
p2 = crudeMonteCarloNorm(4, n2)
```

```
print(f"With {n} samples")
print(f"Crude probability z larger than 2 is {round(p1 * 100,5)}%")
print(f"Crude probability z larger than 4 is {round(p2 * 100,5)}%")

def h(a,x):
    return x > a

sigma1 = 1
a = 2
ys = rnd.normal(loc = a,scale = sigma1,size = n)
fy = stats.norm.pdf(ys)
gy = stats.norm.pdf(ys, loc = a, scale = sigma1)
zs = fy / gy * h(a,ys)
print(f"Probability z larger than 2 with importance sampling {round(np.mean(zs)

a = 4
ys = rnd.normal(loc = a,scale = sigma1,size = n)
fy = stats.norm.pdf(ys)
gy = stats.norm.pdf(ys, loc = a, scale = sigma1)
zs = fy / gy * h(a,ys)

print(f"Probability z larger than 4 with importance sampling {round(np.mean(zs)
```

```
With 10000 samples
Crude probability z larger than 2 is 2.27%
Crude probability z larger than 4 is 0.003%
Probability z larger than 2 with importance sampling 2.3139%
Probability z larger than 4 with importance sampling 0.0031%
```

8)

Analytically set

$$f(x) = \mathbf{1}_{0\leq x\leq 1}$$

$$h(x) = e^x$$

$$g(x) = \lambda e^{-\lambda x}.$$

We want to find the variance of

$$\frac{f(x)h(x)}{g(x)}.$$

Write

$$Var\left(\frac{f(x)h(x)}{g(x)}\right) = E\left(\left(\frac{f(x)h(x)}{g(x)}\right)^2\right) - E\left(\frac{f(x)h(x)}{g(x)}\right)^2$$

$$= \int_{-\infty}^{\infty}\left(\frac{f(x)h(x)}{g(x)}\right)^2 g(x)\,\mathrm{d}x - \left(\int_{-\infty}^{\infty}\frac{f(x)h(x)}{g(x)}g(x)\,\mathrm{d}x\right)^2.$$

The second integral is simply to rewrite

$$\left(\int_{-\infty}^{\infty}\frac{f(x)h(x)}{g(x)}g(x)\,\mathrm{d}x\right)^2 = \left(\int_{-\infty}^{\infty}f(x)h(x)\,\mathrm{d}x\right)^2$$

$$= \left(\int_0^1 f(x) e^x \, \mathrm{d}x \right)^2 .$$

The first integral is slightly less pretty

$$\int_{-\infty}^{\infty} \left(\frac{f(x)h(x)}{g(x)} \right)^2 g(x) \, \mathrm{d}x = \int_{-\infty}^{\infty} \frac{f(x)^2 h(x)^2}{g(x)} \, \mathrm{d}x$$

$$= \int_{-\infty}^{\infty} \frac{f(x)^2 e^{2x}}{\lambda e^{-\lambda x}} \, \mathrm{d}x = \frac{1}{\lambda} \int_{-\infty}^{\infty} f(x)^2 e^{2x + \lambda x} \, \mathrm{d}x$$

$$= \frac{1}{\lambda} \int_0^1 e^{(2+\lambda)x} \, \mathrm{d}x .$$

Collecting:

$$Var \left(\frac{f(x)h(x)}{g(x)} \right) = \frac{1}{\lambda} \int_0^1 e^{(2+\lambda)x} \, \mathrm{d}x - \left(\int_0^1 f(x) e^x \, \mathrm{d}x \right)^2$$

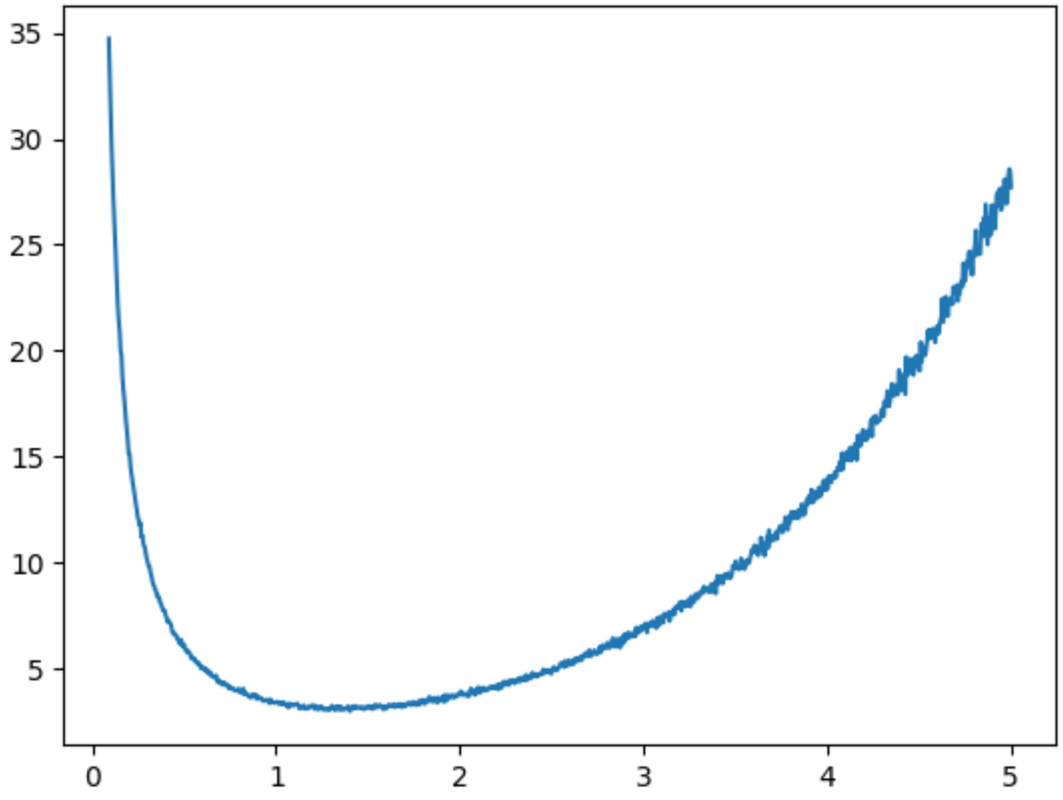
From Maple:

$$Var \left(\frac{f(x)h(x)}{g(x)} \right) = \frac{-1 + e^{2+\lambda}}{2 + \lambda} - (e-1)^2 .$$

Using a numerical solver to minimize this expression gives $\lambda \approx 1.354828644$.

In [ ]:
```python
import matplotlib.pyplot as plt
lambda_opt = 1.354828644

def var_anal(lam):
    return (-1 + np.exp(2 + lam))/(lam*(2 + lam)) - (np.exp(1) - 1)^2

def variance(lam, n):
    Us1 = rnd.uniform(size = n)
    Us2 = rnd.uniform(size = n)
    return 1/lam * np.mean(np.exp((2+lam)*Us1)) - np.mean(np.exp(Us2))**2

lams = np.linspace(0.09, 5, 1000)
ys = np.zeros(1000)
for i in range(len(lams)):
    ys[i] = variance(lams[i], 10000)
print("Theoretical variance at optimal lambda is", np.min(ys))



plt.plot(lams, ys)

def g(x,lam):
    return lam * np.exp(-lam * x)

def f(x):
    b1 = x > 0
    b2 = x < 1
    return b1 * b2
def h(x):
    return np.exp(x)

n = 10000
lam = lambda_opt



n = 10000
Xs_crude = crudeMC(n)
mean = np.mean(Xs_crude)
var = np.var(Xs_crude)
s = np.sqrt(var / len(Xs_crude))
a = stats.norm.ppf(alpha/2)
b = stats.norm.ppf(1 - alpha/2)
print("Using Crude Monte Carlo")
print(f"Mean is {round(mean,4)}, with confidence interval [{round(mean + s * a,3
print(f"Width of CI = {round(abs(mean + s * a - mean - s*b),3)}")
```

```
Theoretical variance at optimal lambda is 2.9803500419443325
Actual variance of sample at optimal lambda is 3.1416055177375855
Using importance sampling
Mean is 1.7106, with confidence interval [1.676,1.745]
Width of CI = 0.069
Using Crude Monte Carlo
Mean is 1.7172, with confidence interval [1.708,1.727]
Width of CI = 0.019
```



We note here that the variance of our sample, when using the optimal lambda, is close to the variance we'ed expect given the plot. Using importance sampling with this $g(x)$ we would not expected to find a lower variance.

We can try the same thing, now with $\lambda = 3.5$, where we would expect to see a sample variance aroung 10, which we indeed do.

In [ ]:
```
n = 10000
lam = 3.5
ys = rnd.exponential(scale = 1/lam, size = n)

zs = f(ys) * h(ys) / g(ys, lam)

mean = np.mean(zs)
var = np.var(zs)

print(f"Using lambda = {lam} we instead find a variance of", var)
```

```
Using lambda = 3.5 we instead find a variance of 9.901624437913682
```

9) Deriving the IS estimator for the mean of a Pareto distribution

pdf for a arbritray Pareto distribution is $f(x) = \frac{k}{y}\left(\frac{\beta}{y}\right)^k$. From the lectures we know that the first moment distribution of a Pareto distribution is just a Pareto distribution with paramters β and $k-1$. Thus

$$g(y) = \frac{k-1}{y}\left(\frac{\beta}{y}\right)^{k-1}.$$

As we're estimating the mean, set $h(y) = y$. Now the IS estimator for the mean is

$$\frac{f(y)h(y)}{g(y)} = \frac{\frac{k}{y}\left(\frac{\beta}{y}\right)^k \cdot y}{\frac{k-1}{y}\left(\frac{\beta}{y}\right)^{k-1}} = \frac{yk}{k-1}\frac{\left(\frac{\beta}{y}\right)^k}{\left(\frac{\beta}{y}\right)^{k-1}} = \frac{yk}{k-1}\frac{\beta}{y} = \beta\frac{k}{k-1}.$$

Which we know is just the mean of the original Pareto distribution $f(x)$, i.e. we should be sampling exactly the mean! Which we do, with an exact CI.

In [ ]:
```
k = 2
def g(x):
    return (k-1) /(x) * (1/x)**(k-1)

def f(x):
    return k /(x) * (1/x)**k
def h(x):
    return x

n = 100
lam = lambda_opt
ys = rnd.exponential(scale = 1/lam, size = n)
beta = 1
zs = f(ys) * h(ys) / g(ys)
alpha = 0.05
mean = np.mean(zs)
var = np.var(zs)
s = np.sqrt(var / len(zs))
a = stats.norm.ppf(alpha/2)
b = stats.norm.ppf(1 - alpha/2)
print("Using importance sampling to sample the mean of a pareto distribution usi
print(f"Mean is {round(mean,4)}, with confidence interval [{round(mean + s * a,3
print(f"Width of CI = {round(abs(mean + s * a - mean - s*b),3)}")
print(f"Exact solution for the Pareto distribution mean {k/(k-1) * beta}")
```

```
Using importance sampling to sample the mean of a pareto distribution using its f
irst moment
Mean is 2.0, with confidence interval [2.0,2.0]
Width of CI = 0.0
Exact solution for the Pareto distribution mean 2.0
```

To do the same in the case of the integral $\int_0^1 e^x \, \mathrm{d}x$ we would need to know a normalising constant s.t. $\frac{1}{c}\int_0^1 e^x \, \mathrm{d}x = 1$, which is simple since we know the actual value of the integral is $e-1$, then $c = e-1$. If we choose $g(x) = \frac{1}{c}e^x$, we would then have IS estimator

$$\frac{e^x}{\frac{1}{c}e^x} = e-1,$$

and would once again succesfully estimate the mean. The issue is the need for the constant c, which if you know it, makes the whole process redundant, as it requires already knowing the integral, and thus is not really meaningfull.

```
In [ ]: beta = 0.005
        k = 2
        def g(x, beta, k):
            return 1/ (np.exp(1) - 1) * np.exp(x)
        def f(x):
            return (x > 0) * (x < 1)
        def h(x):
            return np.exp(x)

        def pareto(beta, k, n = 10000):
            Us = np.random.uniform(size = n)
            Xs = beta * (Us**(-1/k)-1)
            return Xs

        ys = pareto(beta, k)

        zs = f(ys) * h(ys) / g(ys, beta, k)

        mean = np.mean(zs)
        var = np.var(zs)
        alpha = 0.05
        print("Actual variance of sample at optimal lambda is", var)
        s = np.sqrt(var / len(zs))
        a = stats.norm.ppf(alpha/2)
        b = stats.norm.ppf(1 - alpha/2)
        print("Using importance sampling")
        print(f"Mean is {round(mean,4)}, with confidence interval [{round(mean + s * a,3
        print(f"Width of CI = {round(abs(mean + s * a - mean - s*b),3)}")
```

```
Actual variance of sample at optimal lambda is 9.883934104353516e-32
Using importance sampling
Mean is 1.7183, with confidence interval [1.718,1.718]
Width of CI = 0.0
```

```
In [ ]:
```

In [ ]:
```
from scipy.special import factorial
import numpy as np
from scipy.stats import chi2
from matplotlib.patches import Patch
from matplotlib.lines import Line2D
import matplotlib.pyplot as plt
from scipy.stats import poisson
from scipy.stats import chisquare
import plotly.graph_objects as go
from metropolis_hastings import metropolis_hastings, truncated_poisson_samples,m
import plotly.io as pio
from tests import chisquare_test
#pio.renderers.default = "notebook+pdf"
```

Exercise 6

The number of busy lines in a trunk group (Erlang system) is given by a truncated Poisson distribution P(i) = c · Ai i! , i = 0, . . . m Generate values from this distribution by applying the Metropolis-Hastings algorithm, verify with a χ 2 -test. You can use the parameter values from exercise 4

In [ ]:
```
m = 10 #number of servers
s = 8 #mean service time
lam = 1#arrival_intensity
N =100000 #number of samples
A = lam*s
```

In [ ]:
```
#1The number of busy lines in a trunk group (Erlang system) is
#given by a truncated Poisson distribution

np.random.seed(6)

#unnormalized probability mass function
g = lambda x : A**x / factorial(x)

X_est = metropolis_hastings(g,N,m)
#X_est is the number of busy lines in a trunk group
#create histogram and plot i
hist_estimate,_ = np.histogram(X_est, bins = np.arange(m+2))
#print(hist_estimate)
#sample from the truncated poisson distribution
X_expected = truncated_poisson_samples(A, 0, m, N)
hist_expected,_ = np.histogram(X_expected, bins = np.arange(m+2))
#print(hist_expected)
#Do a Plotly histogram of the estimated values and the expected values in the sa
fig = go.Figure()
fig.add_trace(go.Bar(x = list(range(m+1)), y = hist_estimate, name = 'Estimated'
fig.add_trace(go.Bar(x = list(range(m+1)), y = hist_expected, name = 'Expected')
fig.update_layout(title = 'Estimated and Expected values of the number of busy l
                  xaxis_title = 'Number of busy lines',
                  yaxis_title = 'Count')
fig.show()
```

```
T = sum((hist_estimate - hist_expected)**2/hist_expected)
p_value = 1 - chi2.cdf(np.sum(T), len(hist_estimate)-1)
print('Chi-square test statistic:', T)
print('P-value:', p_value)
```

```
Chi-square test statistic: 11.422001852018466
P-value: 0.32559848895018306
```

Answer

With a Chi-square test statistic of 11.42, and a p-value of 0.32 we can conclude that the data generated from the Metropolis-Hastings algorithm is consistent with the truncated Poisson distribution.

2

For two different call types the joint number of occupied lines is given by ... You can use A1, A2 = 4 and m = 10

a)

Use Metropolis-Hastings, directly to generate variates from this distribution.

```
In [ ]: A1 = 4
        A2 = 4

        def g_joint(A1,A2):
            return lambda x,y : (A1**x / factorial(x)) * (A2**(y) / factorial(y))

        def joint_truncated_poisson_samples(A1, A2, low, high, size=1):
            samples = []
            while len(samples) < size:
                x = np.random.poisson(A1)
                y = np.random.poisson(A2)
                if low <= x + y <= high:
                    samples.append((x,y))
            return np.array(samples)
```

```
In [ ]: #joint samples
        X_est_joint = metropolis_hastings_joint(g_joint(A1,A2),N, y_sampling_func=y_samp
        X_true_joint = joint_truncated_poisson_samples(A1,A2,0,m,N)

        ## Take every 175th observation to eliminate dependencies
        X_est_joint = X_est_joint[::175]
        X_true_joint = X_true_joint[::175]
        #joint histogram estimate
        hist_joint_estimate,_,_ = np.histogram2d(X_est_joint[:,0],X_est_joint[:,1], bins
        #joint histogram true
        hist_joint_true,_,_ = np.histogram2d(X_true_joint[:,0],X_true_joint[:,1], bins =
        #plot the joint estimated joint histogram
        fig = go.Figure(data = [go.Heatmap(z = hist_joint_estimate)])
        fig.update_layout(title = 'Metropolis-hastings estimated joint histogram of the
                          xaxis_title = 'Number of busy lines in trunk 1',
                          yaxis_title = 'Number of busy lines in trunk 2')
        fig.show()
         #plot the joint true joint histogram
        fig = go.Figure(data = [go.Heatmap(z = hist_joint_true)])
```

```
fig.update_layout(title = 'True joint histogram of the number of busy lines in a
                  xaxis_title = 'Number of busy lines in trunk 1',
                  yaxis_title = 'Number of busy lines in trunk 2')
fig.show()
```

In [ ]:
```
#calculate the test statistic for all histograms > 0
# T, p_value = chisquare(hist_joint_estimate.flatten()[hist_joint_estimate.flatt


#print('Chi-square test statistic:', T)
#print('P-value:', p_value)
T = sum( np.divide((hist_joint_estimate[hist_joint_true > 0] - hist_joint_true[h

#T = sum(sum((hist_joint_estimate - hist_joint_true)**2/hist_joint_true))
p_value = 1 - chi2.cdf(np.sum(T), np.shape(hist_joint_true)[0]*np.shape(hist_joi
print('Chi-square test statistic:', T)
print('P-value:', p_value)
```

```
Chi-square test statistic: 96.6127280718713
P-value: 0.9425846369077646
```

Answer

The Metropolis-Hastings algorithm was used to generate variates from the distribution. The data was then tested using a Chi-square test. The test statistic was 131 and the p-value was 0.22. This indicates that the data generated from the Metropolis-Hastings algorithm is consistent with the truncated Poisson distribution.

Note: We achieved an acceptable distribution of p values when running 100k samples and using every 175th observation. However, this leaves us with only 572 observations and thus the states (i,j) with low probability mass were not hit more than 5 times meaning we don't have enough observations. When we tried to increase the sample size to 1 million, we observed extremely low p-values because the there is a problem with the low probability states (i.e we observe way higher frequencies compared to the theoretical values).

C

Use Gibbs sampling to sample from the distribution. This is (also) coordinate-wise but here we use the exact conditional distributions. You will need to find the conditional distributions analytically

Answer

The truncated Poisson distribution is given by

$$P(i) = c_1 \frac{A^i}{i!}, i = 0, \ldots, m$$

where c_1 is a normalizing constant. The joint distribution is given by

$$P(i,j) = c_2 \frac{A_1^i A_2^j}{i!j!}, 0 \leq i + j \leq m$$

where c_2 is a normalizing constant. The conditional distribution of i given j through the definition of conditional probability

$$P(i|j) = \frac{P(i,j)}{P(j)} \qquad = \frac{c_2 \frac{A_1^i A_2^j}{i!j!}}{c_1 \frac{A_2^j}{j!}} = c_i \frac{A_1^i}{i!}$$

Which is essentially a truncated Poisson distribution, but under the condition, $0 \leq j \leq m \quad \& \quad 0 \leq i + j \leq m \Rightarrow 0 \leq i \leq m - j$. We can use the conditional distribution to sample i and j coordinate-wise using the Gibbs sampling algorithm.

```
In [ ]: #Gibbs sampling
        np.random.seed(2098936829)
        N = 200000
        As = [4, 4]
        x0 = [3,3]
        m = 10
        n_burn = 1000
        #print(N)
        Xs = np.array(Gibbs(As, N + n_burn, x0, m))
        xs = Xs[0,n_burn:]
        ys = Xs[1,n_burn:]

        xs = xs[::125]
        ys = ys[::125]

        N = len(xs)
```

```
In [ ]: hist_est_gibbs = np.histogram2d(xs,ys, bins = [np.arange(m+2),np.arange(m+2)])[0
        fig = go.Figure(data = [go.Heatmap(z = hist_est_gibbs)])
        fig.update_layout(title = 'Metropolis-hastings estimated joint histogram of the
                          xaxis_title = 'Number of busy lines in trunk 1',
                          yaxis_title = 'Number of busy lines in trunk 2')
        fig.show()
```

```
In [ ]: #chi square test
        #T, p_value = chisquare(hist_est_gibbs.flatten()[hist_est_gibbs.flatten() > 0],
        T = 0
        for i in range(m+1):
            for j in range(m+1):
                if hist_joint_true[i,j] > 0:
                    T += (hist_est_gibbs[i,j] - hist_joint_true[i,j])**2/hist_joint_true


        X_true_joint = joint_truncated_poisson_samples(A1,A2,0,m,N)
        hist_joint_true,_,_ = np.histogram2d(X_true_joint[:,0],X_true_joint[:,1], bins =
```

```
In [ ]: p_value = 1 - chi2.cdf(np.sum(T), 65)

        print('Chi-square test statistic:', T)
        print('P-value:', p_value)
```

```
Chi-square test statistic: 78.3546628813818
P-value: 0.12362308383372511
```

Answer

The Gibbs sampling algorithm was used to sample from the distribution. The data was then tested using a Chi-square test. The test statistic was 106.01 and the p-value was 0.815 This indicates that the data generated from the Gibbs sampling algorithm is consistent with the truncated Poisson distribution.

3

We consider a Bayesian statistical problem. The observations are Xi ~ N(Θ, Ψ), where the prior distribution of the pair (Ξ, Γ) = (log (Θ), log (Ψ)) is standard normal with correlation ρ. The posterior distribution of (Θ, Ψ) is given by

$$P(\Theta, \Psi|X) = \frac{P(X|\Theta,\Psi)\cdot P(\Theta,\Psi)}{P(X)},$$

which can be derived using a standard change of variable technique. The task of this exercise is now to sample from the posterior distribution of (Θ, Ψ) using Markov Chain Monte Carlo.

a)

Generate a pair (θ, ψ) from the prior distribution, i.e. the distribution for the pair (Θ, Ψ), by first generating a sample (ξ, γ) of (Ξ, Γ).

```
In [ ]: #xi and gamma of (Ξ, Γ) = (log (Θ), log (Ψ))
        #joint density of Theta and Psi is normal with correlation rho = 0.5
        rho = 0.5

        #sample from the joint density
        θ, ψ =np.exp(np.random.multivariate_normal([0,0],[[1,rho],[rho,1]]))

        print('θ:',θ)
        print('ψ:',ψ)
```

```
θ: 1.101843408693259
ψ: 2.493859910186823
```

The θ and ψ sampled from the prior distribution are given as follows: $$

$$\theta = 1.101843408693259$$
$$\psi = 2.493859910186823$$

b)

Generate Xi = 1, . . . , n with the values of (θ, ψ) you obtained in item 3a. Use n = 10

```
In [ ]: n = 10
        #generate n samples from the univariate distribution N(ϑ,ψ)
        np.random.seed(100)
        X = np.random.normal(θ,ψ,n)
```

Derive the posterior distribution of (Θ, Ψ) given the sample. *Answer*

The posterior distribution of can be computed by using the Bayes theorem. Under the assumption that X_i ~ N(Θ, Ψ) and sampled i.d.d. from the distribution, the likelihood function is given by the product of the pdf evaluated at the observed data points (For the implementation the exponential of the log-likelood is computed, in order to avoid numerical issues).

$$p(X|\Theta,\Psi) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\Psi}} \exp\left(-\frac{(X_i-\Theta)^2}{2\Psi}\right)$$
$$= \frac{1}{(2\pi\Psi)^{n/2}} \exp\left(-\frac{1}{2\Psi}\sum_{i=1}^{n}(X_i-\Theta)^2\right)$$

The prior distribution is given by $f(x, y)$ as described in the exercise. The posterior distribution is then given by

$$p(\Theta,\Psi|X) \propto p(X|\Theta,\Psi)f(\Theta,\Psi)$$
$$p(\Theta,\Psi|X) = c\cdot p(X|\Theta,\Psi)f(\Theta,\Psi)$$

where $c = 1/P(X)$ is a normalizing constant, which doesn't need to be computed for the MCMC sampling.

In [ ]:
```
#The log-likelihood of the data X given the parameters ϑ and ψ is. Each X follow
likelihood = lambda X,θ,ψ : np.exp(-n/2*np.log(2*np.pi) - n/2*np.log(ψ**2) - 1/(
#  The joint density f(x, y) of (Θ, Ψ) is
prior = lambda  x,y : 1/(2*np.pi*x*y*np.sqrt(1-rho**2))*np.exp(-1/(2*(1-rho**2))

#y is sampled from a folded normal distribution (has commutative property) param
y_sampling_bays = lambda : np.abs(np.random.normal(0,4,2))

posterior = lambda X : lambda θ,ψ : likelihood(X,θ,ψ)*prior(θ,ψ)
```

d)

Generate MCMC samples from the posterior distribution of (Θ, Ψ) using the Metropolis Hastings method.

In [ ]:
```
X_est_joint = metropolis_hastings_joint(posterior(X),N, y_sampling_func= y_sampl
```

In [ ]:
```
#joint histogram of the X_est_joint samples in the interval \[ϑ-1,ϑ+1\] and \[ψ-
X_est_joint_interval = X_est_joint[(X_est_joint[:,0] > θ-1)*( X_est_joint[:,0] <
hist_joint_estimate,_,_ = np.histogram2d(X_est_joint_interval[:,0],X_est_joint_i
#Plotly histogram of the joint histogram along with the true values of ϑ and ψ
fig = go.Figure(data = [go.Heatmap(z = hist_joint_estimate, x = np.linspace(θ-1,
fig.add_trace(go.Scatter(x = [θ], y = [ψ], mode = 'markers', marker = dict(size
fig.update_layout(title = 'Metropolis hastings samples of θ and ψ from posterior
                  xaxis_title = 'θ',
                  yaxis_title = 'ψ',
                    showlegend = True,
                    legend = dict(x = 0.8, y = 0.8))

fig.show()
```

e)

Repeat item 3d with n = 100 and n = 1000, still using the values of (θ, ψ) from item 3a.

Discuss the results.

```
In [ ]: #generate samples from the joint density
        n = 100
        X = np.random.normal(θ,ψ,n)
        X_est_joint = metropolis_hastings_joint(posterior(X),N, y_sampling_func= y_sampl

        #joint histogram of the X_est_joint samples in the interval \[ϑ-1,ϑ+1\] and \[ψ-
        X_est_joint_interval = X_est_joint[(X_est_joint[:,0] > θ-1)*( X_est_joint[:,0] <
        hist_joint_estimate,_,_ = np.histogram2d(X_est_joint_interval[:,0],X_est_joint_i
        #Plotly histogram of the joint histogram along with the true values of ϑ and ψ
        fig = go.Figure(data = [go.Heatmap(z = hist_joint_estimate, x = np.linspace(θ-1,
        fig.add_trace(go.Scatter(x = [θ], y = [ψ], mode = 'markers', marker = dict(size
        fig.update_layout(title = 'Metropolis hastings samples of θ and ψ from posterior
                          xaxis_title = 'θ',
                          yaxis_title = 'ψ',
                            showlegend = True,
                            legend = dict(x = 0.8, y = 0.8))

        fig.show()
```

Answer:

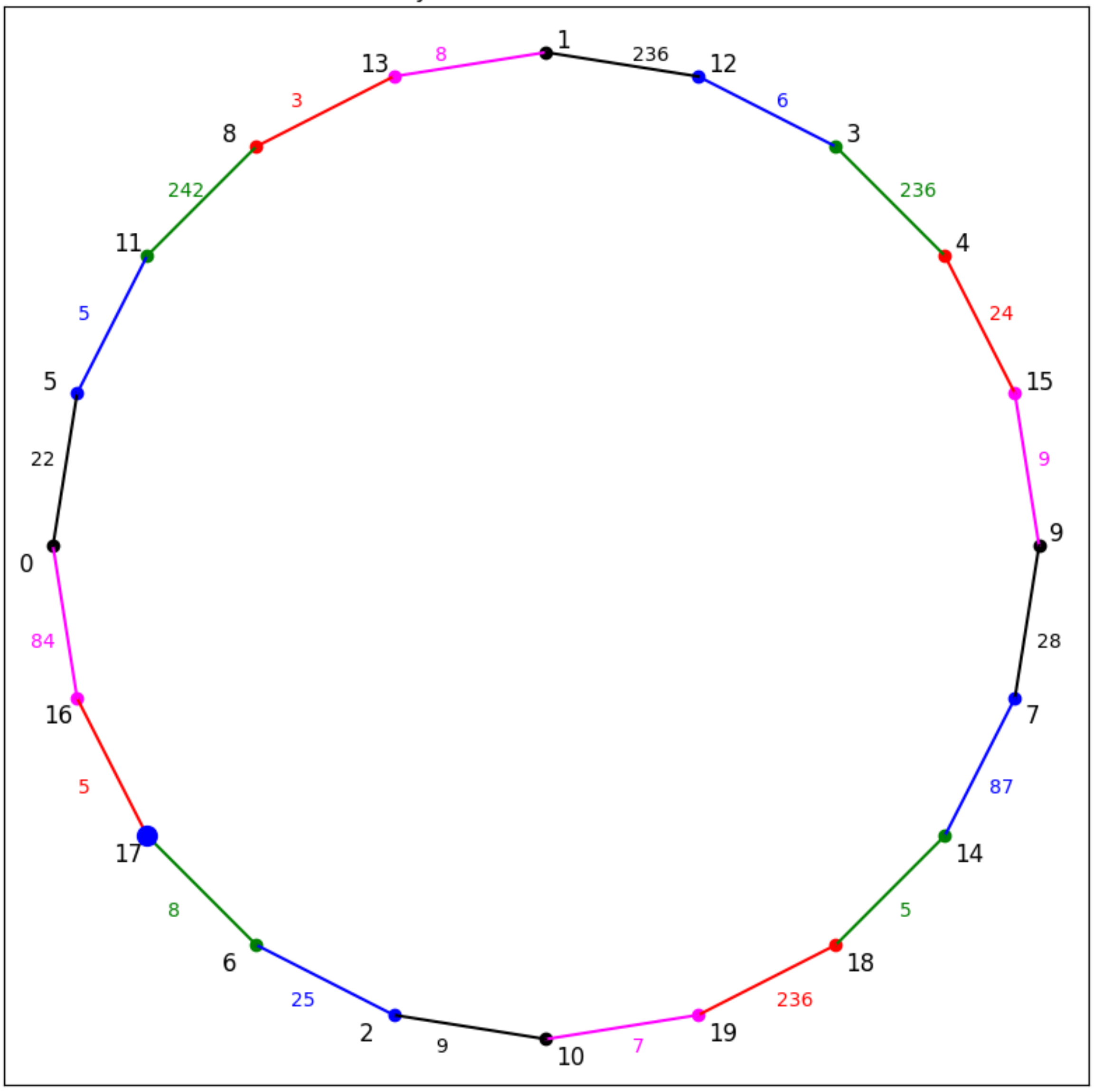
Above it's seeen that when the number of samples is increased, the posterior distribution is more dense around the values θ and ψ used to generate X. This is because the likelihood function is more informative when the number of samples is increased. The results for N=1000 are not shown, as they could not be computed at floating point precision, (Due to taking the exponential of an extremely negative number).

In [ ]:
```
import numpy as np
import math
import numpy.random as rnd


def metropolis_hastings(g,N,m, burn_in = None):
    burn_in = burn_in if burn_in is not None else N // 10
    #X and Y are random integers from 0 to m
    U = np.random.uniform(0,1,N + burn_in)
    X = np.zeros(N + burn_in,dtype = int)
    X[0] = 3
    #prob_of_sampling = np.minimum(g(X)/g(Y),np.ones(N))
    #sample X with probability prob_of_sampling
    for i in range(1,N+burn_in):
        x = X[i-1]
        y = np.random.randint(0,m+1)

        if U[i] <= min(g(y) / g(x),1) :
            X[i] = y
        else:
            X[i] = x

    # X = np.where(U<prob_of_sampling,X,Y)
    X = X[burn_in:]
    return X

def truncated_poisson_samples(lam, low, high, size=1, numvars = 1):
    samples = []
    while len(samples) < size:
        x = np.random.poisson(lam, numvars)
        if low <= x <= high:
            samples.append(x)
    return np.array(samples)

def y_sampling_function(m):
        def y_sampling():
            y1 = np.random.randint(0,m+1)
            y2 = np.random.randint(0,m+1-y1)
            #create y as (y1,y2) or (y2,y1) with equal probability
            return (y1,y2) if np.random.uniform(0,1) < 0.5 else (y2,y1)
        return y_sampling


def metropolis_hastings_joint(g_joint,N, burn_in = None, y_sampling_func = y_sam
    burn_in = burn_in if burn_in is not None else N

    #X and Y are random integers from 0 to m
    U = np.random.uniform(0,1,N + burn_in)
    X = np.zeros((N + burn_in,2),dtype = float)
    X[0] = (1.0,1.0)
    #prob_of_sampling = np.minimum(g(X)/g(Y),np.ones(N))
    #sample X with probability prob_of_sampling
    for i in range(1,N+burn_in):
        x = X[i-1]
        y = y_sampling_func()
        if U[i] <= min(g_joint(y[0],y[1]) / g_joint(x[0],x[1]),1) :
            X[i] = y
        else:
            X[i] = x
    # X = np.where(U<prob_of_sampling,X,Y)
    X = X[burn_in:]
    return X

def Gibbs(As, n, x0, m):
    xs = [x0[0]]
    ys = [x0[1]]
    A1 = As[0]
    A2 = As[1]
    for k in range(1,n):
        # Generate i and sample from j
        i = xs[k-1]
        num_classes_j = int(m - i + 1)
        ps = np.zeros(num_classes_j)
        k = 0
        for j in range(num_classes_j):
            ps[j] = A2**j / math.factorial(j)
            k += A2**j / math.factorial(j)
        ps /= k
        j = rnd.choice(a=np.arange(num_classes_j), size= 1, p = ps)[0]

        ys.append(j)
        # Newest j has already been found
        num_classes_i = int(m-j + 1)
        ps = np.zeros(num_classes_i )
        k = 0
        for i in range(num_classes_i):
            ps[i] = A2**i / math.factorial(i)
            k += A2**i / math.factorial(i)
        ps /= k
        i = rnd.choice(a=np.arange(num_classes_i), size=1, p = ps)[0]
        xs.append(i)
    return np.array(xs), np.array(ys)
```

between 1 and 20. The cost of travelling to the next city for all routes is marked along the path.

```
In [ ]: plotRoute2(routes[-1], x_coords, y_coords, costMatrix, costs[-1])
```

Optimal Route for the problem
Optimal route cost = 1285.0
Point numbers are city numbers, colored numbers are route costs



Exercise 8

13. Let $X_1, \ldots, X_n$ be independent and identically distributed random variables having unknown mean μ. For given constants $a < b$, we are interested in estimating $p = P\{a < \sum_{i=1}^{n} X_i/n - \mu < b\}$.

(a) Explain how we can use the bootstrap approach to estimate p.
(b) Estimate p if $n = 10$ and the values of the X_i are 56, 101, 78, 67, 93, 87, 64, 72, 80, and 69. Take $a = -5$, $b = 5$.

In the following three exercises $X_1, \ldots, X_n$ is a sample from a distribution whose variance is (the unknown) σ^2. We are planning to estimate σ^2 by the sample variance $S^2 = \sum_{i=1}^{n}(X_i - \overline{X})^2/(n-1)$, and we want to use the bootstrap technique to estimate $\text{Var}(S^2)$.

a) CHANGED

Sample n numbers from the emperical distribution $X_1, \ldots X_n$ with replacement, do this a large amount of times. For each of these samples check if the relation holds, and from this find the probablity of a sample satisfying the relation. The mean μ can be preestimated, as just the mean of the samples we bootstrap from.

This can even be done a large amount of times, to get a confidence interval for p

b) CHANGED

When $n = 10$ sample 10×10000 from the empirical distribution, and find p. Here we do this 500 times to also get a 95% CI for p.

In [ ]:
```
Xs = np.array([56, 101, 78, 67, 93, 87, 64, 72, 80, 69])
mu = np.mean(Xs)

ps = []
m = 10000
k = 500

mu = np.mean(Xs)
a = -5
b = 5

for i in range(k):
    sample = rnd.choice(Xs, (len(Xs),m), replace = True)
    #mu = np.mean(np.mean(sample, axis= 0))
    sample = sample - mu
    me = np.mean(sample, axis = 0)

    p = np.mean((me > a) * (me < b))
    ps.append(p)
```

In [ ]:
```
alpha = 0.05
low = np.quantile(ps, alpha/2)
high = np.quantile(ps, 1 - alpha/2)
print(f"Mean probabilty = {round(np.mean(ps),3)*100}%")
print(f"95% CI: [{round(low,3)},{round(high,3)}]")
```

```
Mean probabilty = 76.6%
95% CI: [0.758,0.775]
```

15. If $n = 15$ and the data are

$$5, 4, 9, 6, 21, 17, 11, 20, 7, 10, 21, 15, 13, 16, 8$$

approximate (by a simulation) the bootstrap estimate of $\mathrm{Var}(S^2)$.

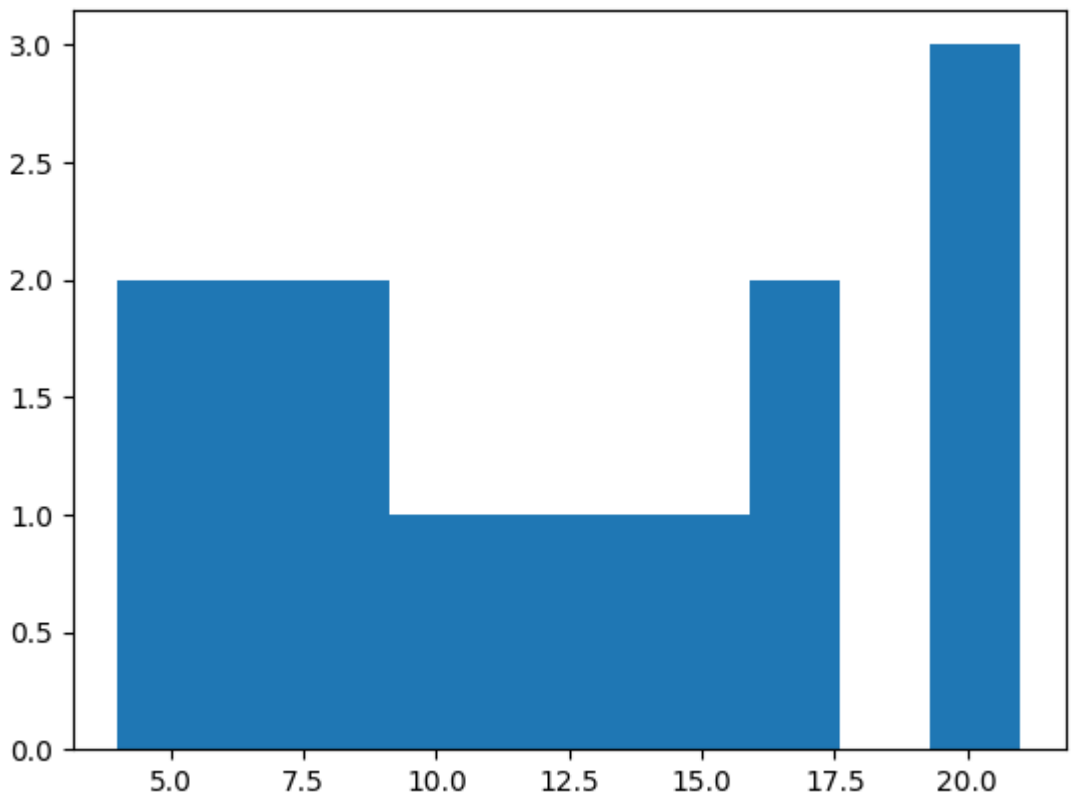
UNCHANGED

To find the variance of the sample variance we sample 15 ranom numbers from the dataset $m = 10000$ times. For each sample of 15 find it's variance. Then find the variance of the m estimated variances.

Notice the variance in a given sample is not very large, but the variance between samples varies a lot. This may be attributed to the emipircal distribution having large amounts of either relatively small numbers, and relatively large numbers. This means the variances between samples can change drastically.

In [ ]:
```
Xs = np.array([5, 4, 9, 6, 21, 17, 11, 20, 7, 10, 21, 15, 13, 16, 8])
m = 10000
vars = []
sample = rnd.choice(Xs, (len(Xs),m), replace = True)
var = np.var(sample, ddof= 1, axis = 0)

print("Variance in the sample variance is:", round(np.var(var, ddof=1),3))
_ = plt.hist(Xs, bins = 10)
```

```
Variance in the sample variance is: 60.145
```



3) CHANGED

Recall for the Pareto distribution that the mean of the sample is not well explained by the majority of points in the distribtution. This means it's difficult to sample enough points to accurately estimate the mean, resulting in a large variance in our estimate of it. This is not the case for the median value, which can be seen in it's low bootstrap variance. The median is more robust to what could be thought of as outliers. They are of course not really outliers, but in this case the few sampled large values, which so heavily sway the mean value, do not affect the median, since there's not a lot of them.

In [ ]:
```
def generatePareto(k, beta = 1, n = 10000):
    Us = rnd.rand(n)
    return beta * (Us**(-1/k))
beta = 1
k = 1.05
N = 200
r = 100
Xs = generatePareto(k, beta, N)
dist_mean = np.mean(Xs)
dist_median = np.median(Xs)

print(f"The mean of the generated variables is {round(dist_mean,3)}")
print(f"The median of the generated variables is {round(dist_median,3)}")
```

```
The mean of the generated variables is 5.89
The median of the generated variables is 1.644
```

In [ ]:
```
samples = rnd.choice(Xs, (len(Xs), r) , replace = True)
means = np.mean(samples, axis=0)
medians = np.median(samples, axis=0)

print(f"Bootstrap of the variance of sample mean: {round(np.var(means, ddof = 1)
print(f"Bootstrap of the variance of sample median: {round(np.var(medians, ddof
```

```
Bootstrap of the variance of sample mean: 2.08
Bootstrap of the variance of sample median: 0.017
```