

WE HAVE NOW ADDRESSED ALL OF THE FEEDBACK AND CORRECTED OUR EXCERSISES  
(27/06)

s216169, Magnus Bengtsson

s194268, Anton Jørgensen

Group 12

## Exercise 1

In this exercise you should implement everything including the tests (e.g. the chi-square and KS tests) yourself. Later, when your code is working you are free to use builtin functions.

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).

We write a LCG that can only take integer values. It is defined by:

$$X_{n+1} = (a X_n + c) \bmod M$$

Here,  $a$  is the multiplier,  $c$  is the shift, and  $M$  is the modulus.

We initialize the algorithm by setting  $X_0 = 2^{30} - 1$ , the multiplier  $a = 48271$ , the shift  $c = 0$ , and the modulus  $M = 2^{31} - 1$ .

Thus, we can generate 10000 pesuodo-random numbers.

```
import numpy as np
import math
import matplotlib.pyplot as plt
from scipy.stats import chisquare
from scipy.stats import chisquare, kstest, chi2, norm, pearsonr
from statsmodels.sandbox.stats.runs import runtest_1samp
from math import sqrt, exp
np.random.seed(1234)
```

```
def linear_congruential(x0, multiplier, shift, modulus, n=10000,
to_unit_interval=True):
    xs = np.empty(n + 1)
    xs[0] = x0
    for i in range(1, n + 1):
        xs[i] = (multiplier * xs[i - 1] + shift) % modulus

    if to_unit_interval:
        return xs[1:] / modulus
    return xs[1:]

values = linear_congruential(2**30 - 1, 48271, 0, 2 ** 31 - 1)
values
array([0.49998876, 0.95748378, 0.6993237 , ..., 0.90753563,
0.65215983,
0.40703805], shape=(10000,))
```

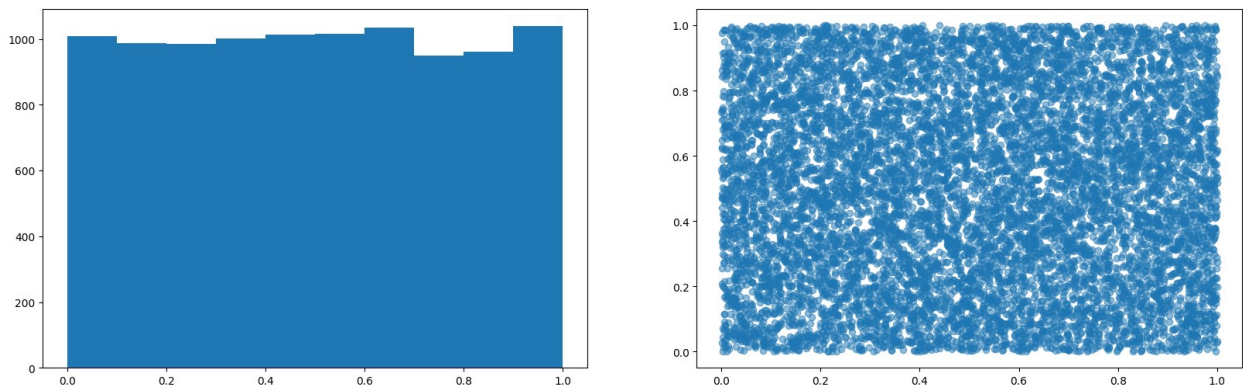
We construct a histogram and a scatterplot of the values. This helps reveal any potential patterns or correlations between consecutive numbers, which would indicate poor randomness.

This shows a reasonably uniform spread of points across the unit square, albeit definitely some reoccurrence, i.e., a short cycle.

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 6))

ax1.hist(values)
ax2.scatter(values[:-1], values[1:], alpha=0.5)

<matplotlib.collections.PathCollection at 0x1b90c85e420>
```



On the left, the histogram shows a fairly uniform distribution, suggesting the LCG is producing values that are evenly spread out, on the interval  $[0, 1]$ . The scatterplot on the right displays a seemingly uniform cloud of points across the plot, suggesting a weak serial correlation. If there was a strong visible pattern, it would suggest a deterministic structure and thus, poor randomness. So initially, this seems promising for LCG.

(b) Evaluate the quality of the generator by descriptive statistics (histogrammes, scatter plots) and statistical tests -  $\chi^2$ , Kolmogorov-Smirnov, run-tests, and correlation test.

To assess whether the values generated by the LCG are uniformly distributed, a chi-square goodness-of-fit test is done. Here, 10 classes is used, meaning the degrees of freedom is 9.

```
num_bins = 10
observed_counts, _ = np.histogram(values, bins=num_bins)
expected_count = len(values) / num_bins
expected_counts = np.full(num_bins, expected_count)

T = sum((obs-exp)**2 / exp for obs, exp in zip(observed_counts,
expected_counts))
df = 9
p_value = 1 - chi2.cdf(T, df)
print(f"Chi-square T = {T:.2f}, p-value = {p_value:.2f}")

Chi-square T = 7.58, p-value = 0.58
```

Here, we see the p-value of 0.58 is obtained, meaning the null hypothesis cannot be discarded of an uniform distribution. Thus, the numbers from the LFG generated do suggest to follow an Uniform distribution.

To further evaluate the uniformity of the numbers, the Kolmogorov-Smirnov test is also applied. We follow the methods described as in Numerisk Simulation, Chapter 3, by Iversen, and Simulation, Chapter 11, by Ross:

```
sorted_values = np.sort(values) #First sort
n = len(values)

D_plus = np.max([(i + 1)/n - x for i, x in enumerate(sorted_values)])
D_minus = np.max([x - i/n for i, x in enumerate(sorted_values)])
D_statistic = max(D_plus, D_minus)

lambda_val = (sqrt(n) + 0.12 + 0.11/sqrt(n)) * D_statistic
p_value = 2 * sum((-1)**(k-1) * exp(-2 * k**2 * lambda_val**2) for k
in range(1, 100))

print(f"KS statistic = {D_statistic:.2f}, p-value = {p_value:.2f}")

KS statistic = 0.01, p-value = 0.84
```

Since the p-value is 0.84, we again cannot reject the null hypothesis. Thus, there is no evidence against uniformity, supporting the idea of the LCG performing well in terms of uniformity.

Now, a Wald-Wolfowitz run test is done, which assesses randomness based on the median. It suggests if the sequence of values generated by the LCG has an abnormal amount of runs, suggesting if there is or is not randomness in the number sequence, as described by Simulation, Chapter 11, by Ross:

```

median = np.median(values)
binary_seq = np.where(values > median, 1, 0)

runs = 1
for i in range(1, len(binary_seq)):
    if binary_seq[i] != binary_seq[i - 1]:
        runs += 1

n1 = np.sum(binary_seq)
n0 = len(binary_seq) - n1

mu = 2 * n1 * n0 / (n1 + n0) + 1
var = (2 * n1 * n0 * (2 * n1 * n0 - n1 - n0)) / (((n1 + n0) ** 2) *
(n1 + n0 - 1))

z = (runs - mu) / sqrt(var)
p_value = 2 * (1 - norm.cdf(abs(z)))

print(f"Run test (Wald-Wolfowitz) score: z = {z:.2f}, p-value =
{p_value:.2f}")

Run test (Wald-Wolfowitz) score: z = -0.60, p-value = 0.55

```

Again, we fail to discard the null hypothesis, meaning there is no statistical evidence to suggest the process is non-random. Thus, the number of runs is consistent with what we would expect from a random binary sequence, supporting the idea the generated values exhibit randomness relative to the median.

To check whether the generated values are independent of each other, we calculate the lag-1 to lag-10 correlations- the correlation between  $U[i]$  and  $U[i+1]$  to  $U[i+10]$ :

We can print the first 10 lags:

```

max_lag = 10
correlations = []
p_values = []

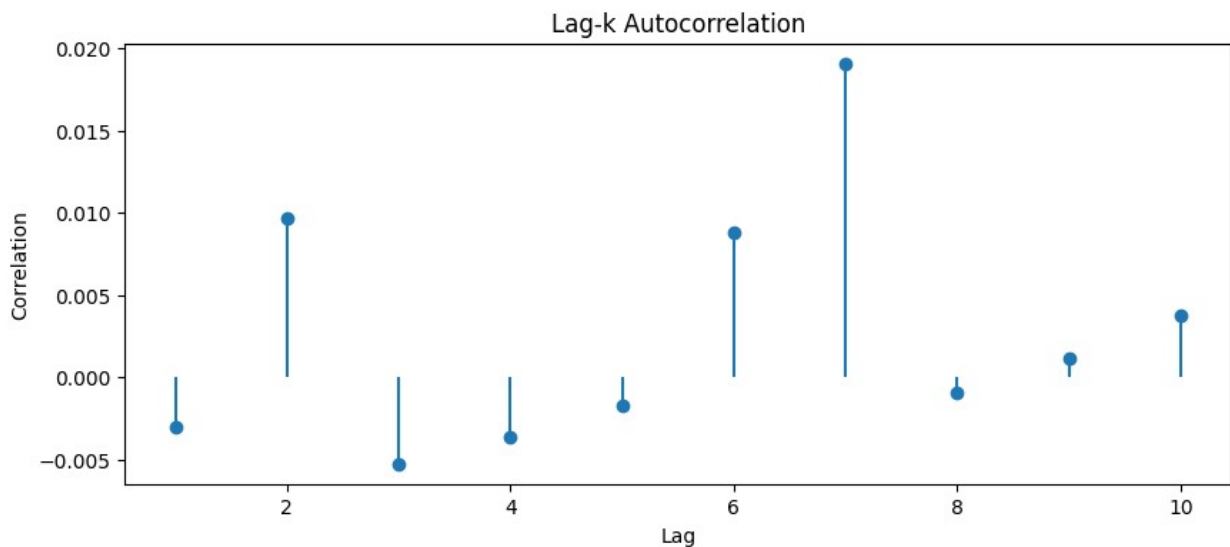
for lag in range(1, max_lag + 1):
    u1 = values[:-lag]
    u2 = values[lag:]
    corr, p_val = pearsonr(u1, u2)
    correlations.append(corr)
    p_values.append(p_val)
    print(f"Lag-{lag} correlation = {corr:.4f}, p-value =
{p_val:.4f}")

# Optionally plot
plt.figure(figsize=(10, 4))
plt.stem(range(1, max_lag+1), correlations, basefmt=" ")
plt.title("Lag-k Autocorrelation")

```

```
plt.xlabel("Lag")
plt.ylabel("Correlation")
plt.show()
```

```
Lag-1 correlation = -0.0030, p-value = 0.7608
Lag-2 correlation = 0.0097, p-value = 0.3343
Lag-3 correlation = -0.0053, p-value = 0.5979
Lag-4 correlation = -0.0036, p-value = 0.7157
Lag-5 correlation = -0.0017, p-value = 0.8645
Lag-6 correlation = 0.0088, p-value = 0.3796
Lag-7 correlation = 0.0191, p-value = 0.0563
Lag-8 correlation = -0.0009, p-value = 0.9271
Lag-9 correlation = 0.0011, p-value = 0.9112
Lag-10 correlation = 0.0037, p-value = 0.7102
```



As the first 10 lags show minimal correlation, they suggest almost none of a correlation between units. Furthermore, every lag has a high p-value, meaning we can reject the null hypothesis - thus, we do not have sufficient evidence of the values being dependent.

Therefore, we would with some decent certainty deem our LCG for generating random, uniform-distributed numbers.

(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.

To identify a good combination of parameters  $a$ ,  $b$ , and  $M$  for the LCG, multiple configurations are now evaluated, testing them for uniformity and randomness.

Here, a despicable choice is presented, where  $X_0=7$ ,  $a=6$ ,  $c=0$ ,  $M=11$ :

```

test1 = linear_congruential(7, 6, 0, 11)
test1

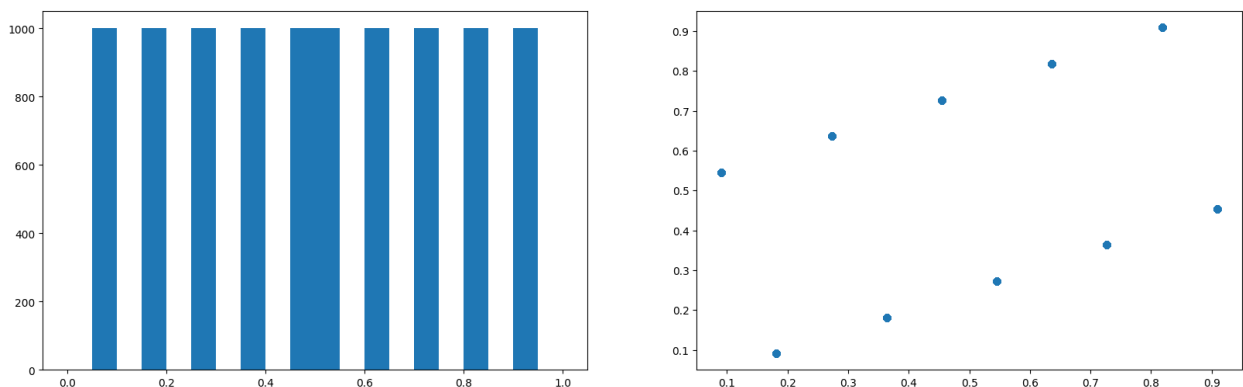
array([0.81818182, 0.90909091, 0.45454545, ..., 0.54545455,
       0.27272727,
       0.63636364], shape=(10000,))

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 6))

ax1.hist(test1, bins=20, range=(0, 1))
ax2.scatter(test1[:-1], test1[1:], alpha=0.5)

<matplotlib.collections.PathCollection at 0x1b908fa6450>

```



The histogram displays a few amount of bins, while the scatter plots display 10000 generated in a short loop - short cycle. Thus, it would suggest to violate randomness.

We print the different statistics:

```

num_bins = 10
observed_counts, _ = np.histogram(test1, bins=num_bins)
expected_count = len(test1) / num_bins
expected_counts = np.full(num_bins, expected_count)

T = sum((obs-exp)**2 / exp for obs, exp in zip(observed_counts,
expected_counts))
df = 9
p_value = 1 - chi2.cdf(T, df)
print(f"Chi-square T = {T:.2f}, p-value = {p_value:.2f}")

sorted_test1 = np.sort(test1) #First sort
n = len(test1)

D_plus = np.max([(i + 1)/n - x for i, x in enumerate(sorted_test1)])
D_minus = np.max([x - i/n for i, x in enumerate(sorted_test1)])
D_statistic = max(D_plus, D_minus)

lambda_val = (sqrt(n) + 0.12 + 0.11/sqrt(n)) * D_statistic

```

```

p_value = 2 * sum((-1)**(k-1) * exp(-2 * k**2 * lambda_val**2) for k
in range(1, 100))

print(f"KS statistic = {D_statistic:.2f}, p-value = {p_value:.2f}")

median = np.median(test1)
binary_seq = np.where(test1 > median, 1, 0)

runs = 1
for i in range(1, len(binary_seq)):
    if binary_seq[i] != binary_seq[i - 1]:
        runs += 1

n1 = np.sum(binary_seq)
n0 = len(binary_seq) - n1

mu = 2 * n1 * n0 / (n1 + n0) + 1
var = (2 * n1 * n0 * (2 * n1 * n0 - n1 - n0)) / (((n1 + n0) ** 2) *
(n1 + n0 - 1))

z = (runs - mu) / sqrt(var)
p_value = 2 * (1 - norm.cdf(abs(z)))

print(f"Run test (Wald-Wolfowitz) score: z = {z:.2f}, p-value =
{p_value:.2f}")
max_lag = 10
correlations = []
p_values = []

for lag in range(1, max_lag + 1):
    u1 = values[:-lag]
    u2 = values[lag:]
    corr, p_val = pearsonr(u1, u2)
    correlations.append(corr)
    p_values.append(p_val)
    print(f"Lag-{lag} correlation = {corr:.4f}, p-value =
{p_val:.4f}")

Chi-square T = 0.00, p-value = 1.00
KS statistic = 0.09, p-value = 0.00
Run test (Wald-Wolfowitz) score: z = 20.00, p-value = 0.00
Lag-1 correlation = -0.0030, p-value = 0.7608
Lag-2 correlation = 0.0097, p-value = 0.3343
Lag-3 correlation = -0.0053, p-value = 0.5979
Lag-4 correlation = -0.0036, p-value = 0.7157
Lag-5 correlation = -0.0017, p-value = 0.8645
Lag-6 correlation = 0.0088, p-value = 0.3796
Lag-7 correlation = 0.0191, p-value = 0.0563
Lag-8 correlation = -0.0009, p-value = 0.9271

```

```
Lag-9 correlation = 0.0011, p-value = 0.9112
Lag-10 correlation = 0.0037, p-value = 0.7102
```

Although the  $\chi^2$  test suggests perfect uniformity, the KS test and run test both strongly reject the hypothesis of randomness, revealing structural flaws in the generator. The lag correlation results show no significant linear dependence, but this alone is not enough to validate randomness given the other clear failures.

Moving on to the good case, where  $a=16807$ ,  $c=0$ ,  $M=2^{31}-1$ . Specifically, this configuration is known as the Park–Miller LCG:

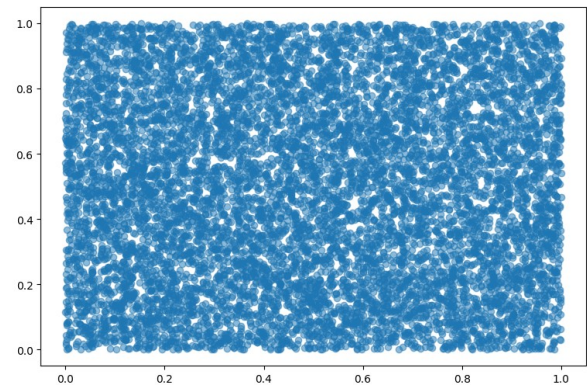
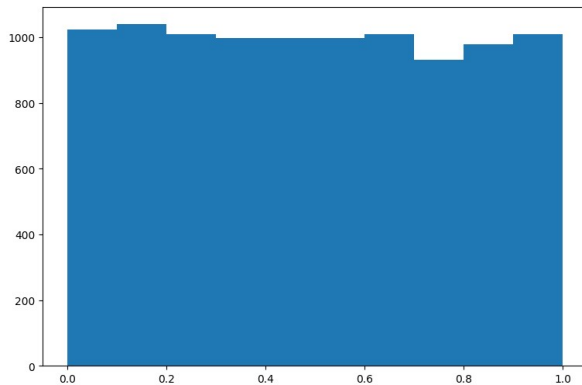
```
test2 = linear_congruential(48271, 16807, 0, 2**31 - 1)
test2

array([0.37778667, 0.46057146, 0.82450768, ..., 0.82405734,
       0.93177724,
       0.38008405], shape=(10000,))

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 6))

ax1.hist(test2)
ax2.scatter(test2[:-1], test2[1:], alpha=0.5)

<matplotlib.collections.PathCollection at 0x1b90d132360>
```



Here, the histogram shows that the values are evenly distributed. Additionally, the scatter plot indicate no visible patterns, which is also to be expected as this configuration has a really long cycle of  $M-1$ .

Now for the tests:

```
num_bins = 10
observed_counts, _ = np.histogram(test2, bins=num_bins)
expected_count = len(test2) / num_bins
expected_counts = np.full(num_bins, expected_count)

T = sum((obs-exp)**2 / exp for obs, exp in zip(observed_counts,
```



```

expected_counts))
df = 9
p_value = 1 - chi2.cdf(T, df)
print(f"Chi-square T = {T:.2f}, p-value = {p_value:.2f}")

sorted_test2 = np.sort(test2) #First sort
n = len(test2)

D_plus = np.max([(i + 1)/n - x for i, x in enumerate(sorted_test2)])
D_minus = np.max([x - i/n for i, x in enumerate(sorted_test2)])
D_statistic = max(D_plus, D_minus)

lambda_val = (sqrt(n) + 0.12 + 0.11/sqrt(n)) * D_statistic
p_value = 2 * sum((-1)**(k-1) * exp(-2 * k**2 * lambda_val**2) for k
in range(1, 100))

print(f"KS statistic = {D_statistic:.2f}, p-value = {p_value:.2f}")

median = np.median(test2)
binary_seq = np.where(test2 > median, 1, 0)

runs = 1
for i in range(1, len(binary_seq)):
    if binary_seq[i] != binary_seq[i - 1]:
        runs += 1

n1 = np.sum(binary_seq)
n0 = len(binary_seq) - n1

mu = 2 * n1 * n0 / (n1 + n0) + 1
var = (2 * n1 * n0 * (2 * n1 * n0 - n1 - n0)) / (((n1 + n0) ** 2) *
(n1 + n0 - 1))

z = (runs - mu) / sqrt(var)
p_value = 2 * (1 - norm.cdf(abs(z)))

print(f"Run test (Wald-Wolfowitz) score: z = {z:.2f}, p-value =
{p_value:.2f}")
max_lag = 10
correlations = []
p_values = []

for lag in range(1, max_lag + 1):
    u1 = values[:-lag]
    u2 = values[lag:]
    corr, p_val = pearsonr(u1, u2)
    correlations.append(corr)
    p_values.append(p_val)
    print(f"Lag-{lag} correlation = {corr:.4f}, p-value =
{p_val:.4f}")

```

```
Chi-square T = 7.40, p-value = 0.60
KS statistic = 0.01, p-value = 0.17
Run test (Wald-Wolfowitz) score: z = 0.08, p-value = 0.94
Lag-1 correlation = -0.0030, p-value = 0.7608
Lag-2 correlation = 0.0097, p-value = 0.3343
Lag-3 correlation = -0.0053, p-value = 0.5979
Lag-4 correlation = -0.0036, p-value = 0.7157
Lag-5 correlation = -0.0017, p-value = 0.8645
Lag-6 correlation = 0.0088, p-value = 0.3796
Lag-7 correlation = 0.0191, p-value = 0.0563
Lag-8 correlation = -0.0009, p-value = 0.9271
Lag-9 correlation = 0.0011, p-value = 0.9112
Lag-10 correlation = 0.0037, p-value = 0.7102
```

The  $\chi^2$ -test shows a high p-value, meaning the null hypothesis cannot be discarded of an uniform distribution. Thus, the generated numbers do suggest to follow an Uniform distribution.

Furthermore, the KS-statistic shows a somewhat high p-value. Once again, we cannot reject the null hypothesis, and there is no evidence against uniformity.

And due to the high p-value of Wald-Wolfowitz run test, we fail to discard the null hypothesis, meaning there is no statistical evidence to suggest the process is non-random. Thus, the number of runs is consistent with what we would expect from a random binary sequence, supporting the idea the generated values exhibit randomness relative to the median.

As for the lag correlations, they all show relatively correlations, meaning generated values are mostly independent (from the last 10 values). As for the p-values, they are all relatively high, but especially lag-7 has a p-value of 0.06, which is close to a value of  $\alpha=0.05$ . Nonetheless, every lag has a high enough p-value, meaning we can reject the null hypothesis - we do not have sufficient evidence of the values being dependent.

Overall, the generator suggests to produce random, independent values.

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

Here, 10000 values are generated using NumPy's built-in random number generator and evaluated for statistical quality. Specifically, 10.000 random values are generated, where  $X \sim \text{Uniform}(0,1)$ :

```
values = np.random.uniform(0, 1, 10000)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 6))

ax1.hist(values)
ax2.scatter(values[:-1], values[1:], alpha=0.5)

num_bins = 10
observed_counts, _ = np.histogram(values, bins=num_bins)
```

```

expected_counts = np.full(num_bins, len(values) / num_bins)
T = sum((obs - exp)**2 / exp for obs, exp in zip(observed_counts,
expected_counts))
chi2_stat, p_val_chi2 = chisquare(f_obs=observed_counts,
f_exp=expected_counts)
ks_stat, p_val_ks = kstest(values, 'uniform')
median = np.median(values)
binary_seq = np.where(values > median, 1, 0)
z_stat, p_val_run = runstest_lsamp(binary_seq)
u1 = values[:-1]
u2 = values[1:]
correlation = np.corrcoef(u1, u2)[0, 1]

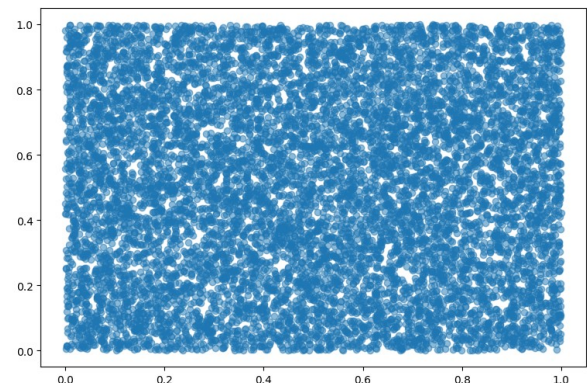
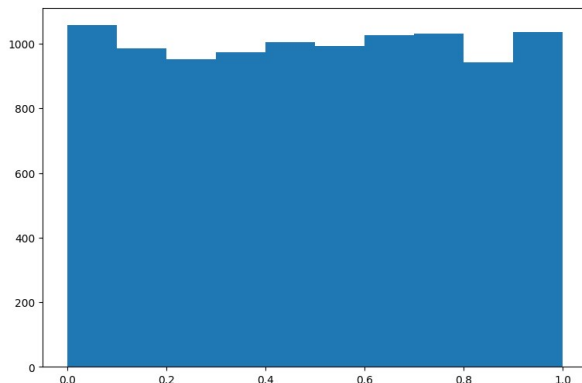
print(f"Chi-square T = {T:.4f}, p-value = {p_val_chi2:.10f}")
print(f"KS statistic = {ks_stat:.4f}, p-value = {p_val_ks:.10f}")
print(f"Run test z   = {z_stat:.4f}, p-value = {p_val_run:.4f}")
print(f"Lag-1 correlation = {correlation:.4f}")

```

```

Chi-square T = 12.9860, p-value = 0.1632407713
KS statistic = 0.0075, p-value = 0.6178824245
Run test z   = -0.7200, p-value = 0.4715
Lag-1 correlation = 0.0042

```



The histogram shows an even distribution, indicating good uniformity, while the scatter plot reveals no visible patterns, suggesting independence.

The Chi-square shows a p-value of approximately 0.16, indicating no significant deviation from the expected uniform distribution. Similarly, KS test yields a high p-value of 0.61, further supporting the hypothesis that the values are uniformly distributed over the interval [0,1].

The run test shows a p-value of 0.47, suggesting that the sequence does exhibit randomness.

The lag-1 correlation is very close to 0, confirming the lack of correlation between successive values and indicating strong independence.

Despite minor fluctuations expected in limited samples ( $n=10000$ ), the overall results show excellent statistical behavior from the system's built-in random number generator. It passes all basic tests of uniformity, independence, and randomness.

It is therefore considered suitable for use in simulation and statistical applications.

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.

I think testing just one sample can give a general idea of how good the generator is, but it's probably not enough on its own. Since random variation can affect the outcome, a single test might not reflect the generator's actual behavior. To be more confident, I tried generating and testing multiple samples. The results were consistent across runs which made me feel more sure that the generator is performing reliably and that the earlier results were not just "random" strokes of luck.

If  $H_0$  is true — meaning the generator produces truly uniform and independent values — then the p values obtained from repeated valid tests should themselves be uniformly distributed on the interval  $[0, 1]$ . This means we should roughly expect an even spread of p-values across multiple runs, with no systematic clustering or patterns near 0 or 1. Observing such a uniform distribution would support the correctness of both the generator and the statistical tests used.

# Exercise 2

## Discrete random variables

1. Choose a value for the probability parameter  $p$  in the geometric distribution and simulate 10,000 outcomes. You can experiment with a small, moderate and large value if you like.

In order to make a geometric distribution, we look in Section 3.2 Simulation, by Ross. Specifically, we first draw a number from a uniform distribution, then take the following function and use that as the data points:

$$X = \left\lceil \frac{\log(1 - U)}{\log(1 - p)} \right\rceil$$

Where  $U$  is the drawn number, and  $p$  is the  $p$  we set. This is equivalent to draw from a geometric distribution.

We simulate  $n=10000$  outcomes with the a geometric distribution, with the values  $p_1=0.2$ ,  $p_2=0.5$ , and  $p_3=0.8$  to test different values. For the  $\chi^2$  tests, we ensure the number of values for each bin is atleast 5.

```
import numpy as np
import math
import matplotlib.pyplot as plt
from scipy.stats import chisquare, geom
from statsmodels.sandbox.stats.runs import runtest_1samp
import time
np.random.seed(42)

p_values = [0.2, 0.5, 0.8]
n_samples = 10000
max_bins = 30

fig, axs = plt.subplots(1, 3, figsize=(18, 5))

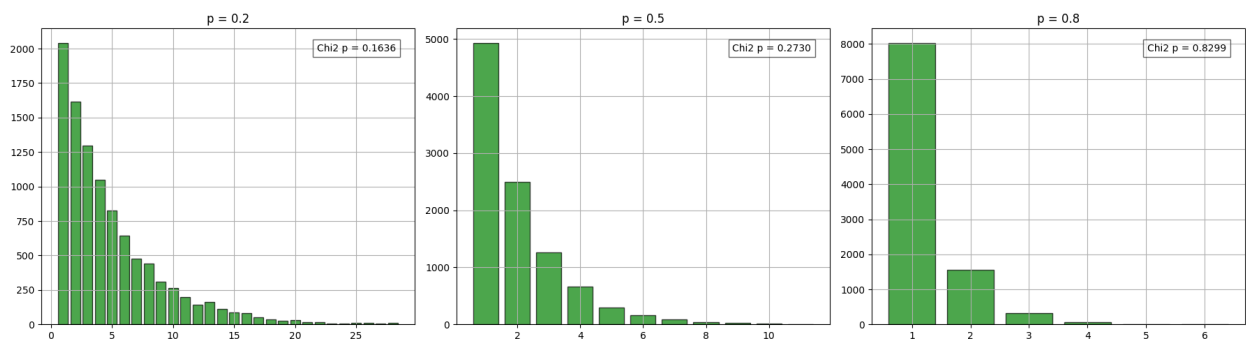
for i, p in enumerate(p_values):
    uniform_randoms = np.random.uniform(0, 1, size=n_samples)
    data = np.ceil(np.log(1 - uniform_randoms) / np.log(1 -
p)).astype(int)
    bins = np.arange(1, min(data.max(), max_bins) + 1)
    counts, _ = np.histogram(data, bins=np.append(bins, bins[-1] + 1))
    probs = geom.pmf(bins, p)
    expected_counts = probs * n_samples
    expected_counts *= counts.sum() / expected_counts.sum()
    cutoff_idx = np.argmax(expected_counts < 5)
```

```

if expected_counts[cutoff_idx] < 5:
    counts = np.append(counts[:cutoff_idx],
counts[cutoff_idx:].sum())
    expected_counts = np.append(expected_counts[:cutoff_idx],
expected_counts[cutoff_idx:].sum())
    bins = np.append(bins[:cutoff_idx], f'>={bins[cutoff_idx]}')
# optional label adjustment
chi2_stat, p_val = chisquare(f_obs=counts, f_exp=expected_counts)
x_labels = range(1, len(counts) + 1)
axs[i].bar(x_labels, counts, width=0.8, align='center', alpha=0.7,
edgecolor='black', color='green')
axs[i].set_title(f'p = {p}')
axs[i].grid(True)
axs[i].text(0.95, 0.95, f'Chi2 p = {p_val:.4f}', ha='right',
va='top',
transform=axs[i].transAxes, fontsize=10,
bbox=dict(facecolor='white', alpha=0.6))

plt.tight_layout()
plt.show()

```




The experiments show that as  $p$  increases, the distribution becomes more concentrated near 1.

Additionally, the  $\chi^2$  test compares the observed frequencies to the expected frequencies in a geometric model. For  $p=0.2$ , it has a low p value, meaning the null hypothesis is discarded - therefore, there is evidence to suggest that the observed data does not fit a geometric distribution well.

As for  $p=0.5$  and  $p=0.8$ , they both have high p-values, meaning we fail to discard the null hypothesis, suggesting those p-values result in a good fit between the simulated values and the fit of a geometric distribution.

## 2. Simulate the 6 point distribution with

1. Generate simulated values from the following distributions 

- (a) Exponential distribution
- (b) Normal distribution (at least with standard Box-Mueller)
- (c) Pareto distribution, with  $\beta = 1$  and experiment with different values of  $k$  values:  $k = 2.05$ ,  $k = 2.5$ ,  $k = 3$  and  $k = 4$ .

Verify the results by comparing histograms with analytical results and perform tests for distribution type.

(a) by applying a direct (crude) method

(b) by using the the rejection method

(c) by using the Alias method

For the 6-point distribution, we have the following:

$$X = \{1, 2, 3, 4, 5, 6\}$$

$$p_i = \{7/48, 5/48, 1/8, 1/16, 1/4, 5/16\}$$

We set  $n = 10000$

```
n = 10000
X = np.array([1, 2, 3, 4, 5, 6])
P = np.array([7/48, 5/48, 1/8, 1/16, 1/4, 5/16])
cum_P = np.cumsum(P)

U = np.random.uniform(0, 1, n)
samples = np.zeros(n, dtype=int)

def chi2_test(samples, p, check=True):
    _, n = np.unique(samples, return_counts=True)
    n_exp = p * np.sum(n)
    T = np.sum((n - n_exp)**2 / n_exp)
    dof = len(p) - 1
    if check:
        print("Degrees of freedom:", dof)
```

```
# test stat follows chi2(k - 1), week 1 slide 10
return 1 - chi2.cdf(T, len(p) - 1)
```

(a) The direct (crude) method works by dividing the unit interval  $[0, 1]$  into segments, where a uniform random number is generated and mapped to the corresponding interval

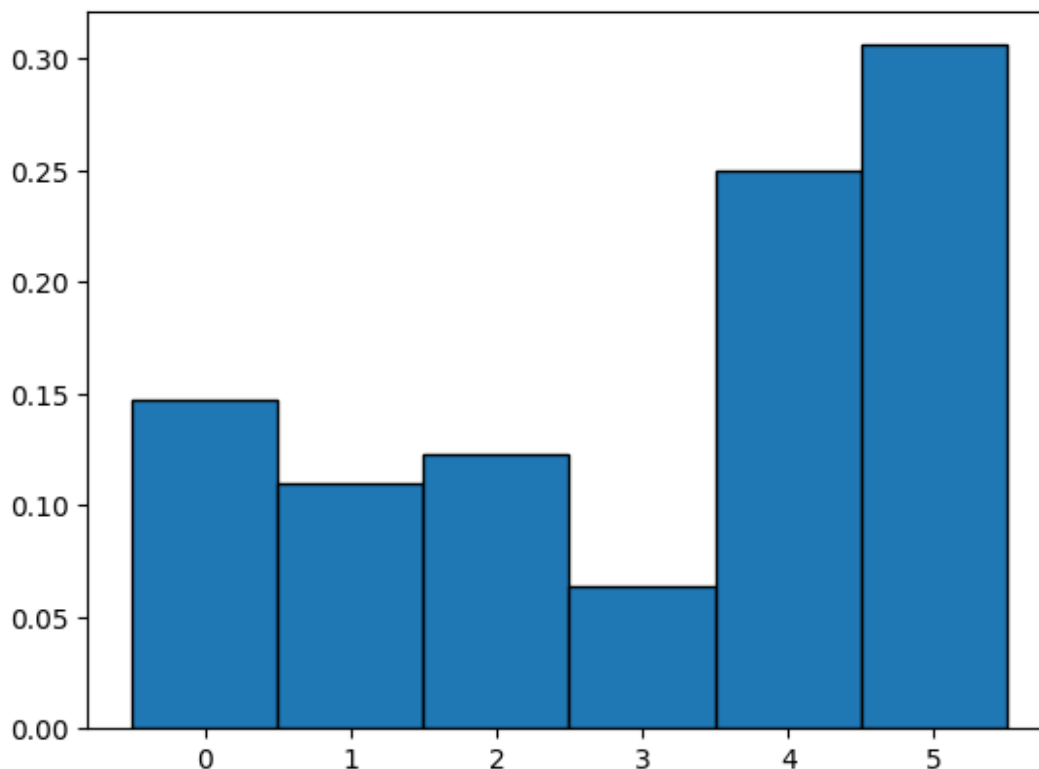
```
def discrete_crude(n, p):
    # not exactly the same crude method as in the slides, but
    # definitely crude
    us = uniform.rvs(size=n)

    vals = np.zeros_like(us)
    thresh = p[0]

    for p_i in p[1:]:
        vals += thresh < us
        thresh += p_i

    return vals

crude_vals = discrete_crude(n, P)
plt.hist(crude_vals, bins=np.arange(len(P)+1)-0.5, edgecolor='black',
density=True)
plt.show()
print("P-val (crude)      :", chi2_test(crude_vals, P))
```





```
Degrees of freedom: 5
P-val (crude)      : 0.4503273164766495
```

The histogram closely matches the target probabilities, with higher bars for outcomes with larger assigned probabilities, confirming that the method correctly maps uniform random values to the specified discrete distribution. The degrees of freedom is calculated as  $6 - 1 = 5$ . Furthermore, the p-value is high, meaning the generated distribution in the method would follow a  $\chi^2$  distribution.

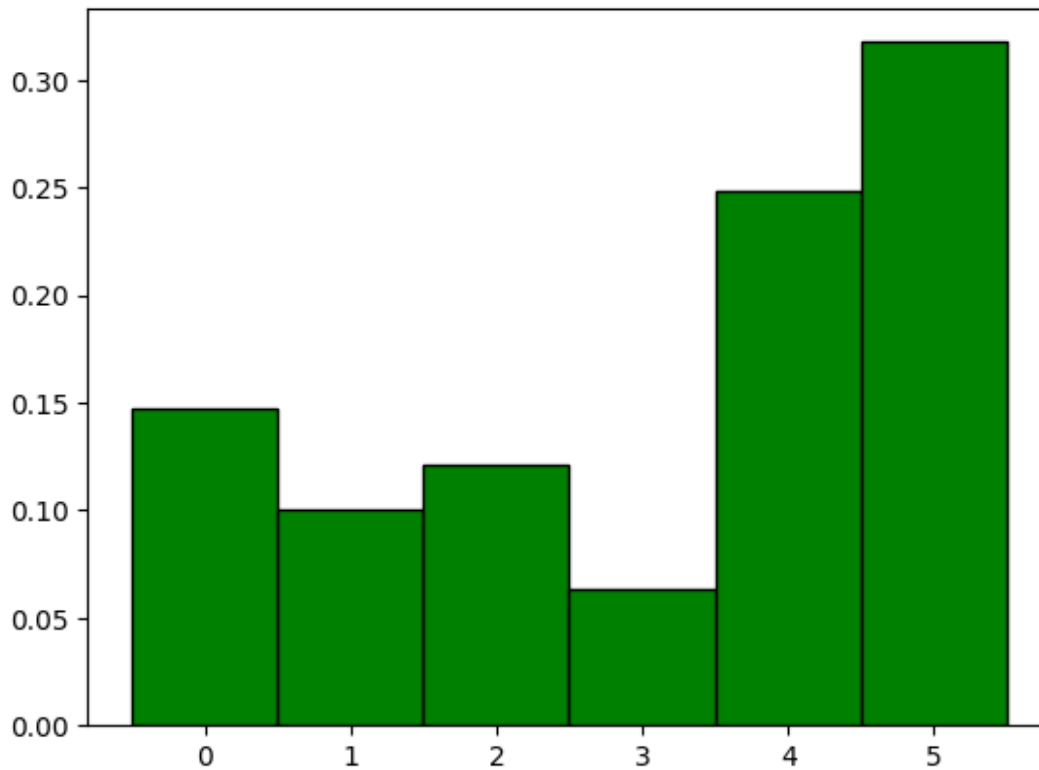
(b) The rejection method works by generating candidate outcomes from a uniform distribution and accepting them with a probability proportional to the target probability.  $c$  is calculated as the maximum value of  $p_i$  multiplied by 6.

```
def discrete_rejection(n, p):
    c = np.max(p)
    k = len(p)

    vals = np.empty(n)
    for j in range(n):
        while True:
            i = int(np.floor(k * uniform.rvs())) # int may just round
            down on its own, but better safe than sorry
            if uniform.rvs() <= p[i] / c:
                vals[j] = i
                break

    return vals

rejection_vals = discrete_rejection(n, P)
plt.hist(rejection_vals, bins=np.arange(len(P)+1)-0.5,
         edgecolor='black', density=True, color='green')
plt.show()
print("P-val (rejection):", chi2_test(rejection_vals, P))
```



Degrees of freedom: 5  
P-val (rejection): 0.6251799233936399

The bars are almost identical to the ones in the histogram from the previous assignment, suggesting this method also correctly maps uniform random values to the specified discrete distribution. Once again, the p-value is high, suggesting the data is consistent with the distribution of  $P$ .

(c) Now, the Alias method is used to sample from a discrete distribution with unequal probabilities. It involves a preprocessing step where two tables are constructed: a probability table ( $F$ ), and an alias table ( $L$ ):

```
def generate_f_and_l(p):
    k = len(p)
    L = np.arange(k)
    F = p * k
    G = np.where(F >= 1)[0]
    S = np.where(F <= 1)[0]

    while S.size:
        i = G[0]
        j = S[0]
        L[j] = i
        F[i] = F[i] - (1 - F[j])
        if F[i] < 1 - 1e-8:
```

```

        G = G[1:]
        S = np.append(S, np.array([i]))
        S = S[1:]

    return F, L

def discrete_alias(n, f, l):
    k = len(f)

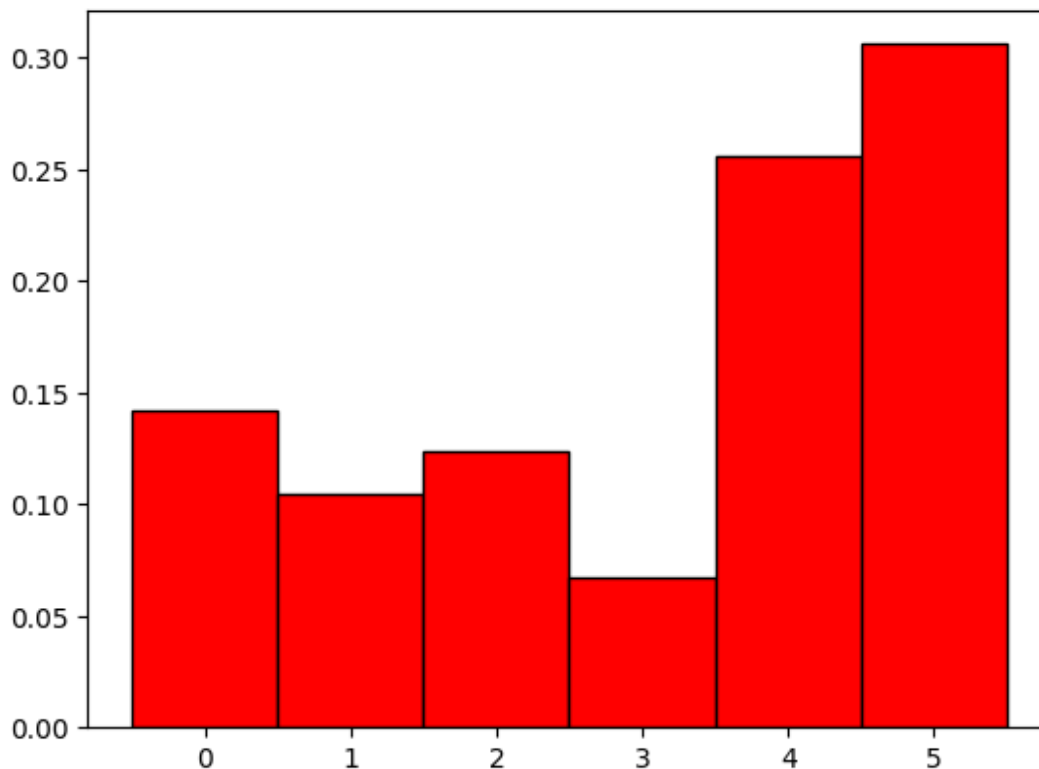
    vals = np.empty(n)
    for j in range(n):
        i = int(np.floor(k * uniform.rvs()))
        if uniform.rvs() <= f[i]:
            vals[j] = i
        else:
            vals[j] = l[i]

    return vals

fs, ls = generate_f_and_l(P)
alias_vals = discrete_alias(n, fs, ls)

plt.hist(alias_vals, bins=np.arange(len(P)+1)-0.5, edgecolor='black',
density=True, color='red')
plt.show()
print("P-val (alias)      :", chi2_test(alias_vals, P))

```



```
Degrees of freedom: 5
P-val (alias)      : 0.24777427066016733
```

In this instance, a high p-value for the Alias method is obtained, meaning the method correctly maps uniform random values to the specified discrete distribution (in this instance). However, as the p value alternates between runs, it would be valuable to see if the p-value changes across e.g., 5 runs with each method:

```
num_trials = 5
pvals_crude, pvals_rejection, pvals_alias = [], [], []

fs, ls = generate_f_and_l(P)

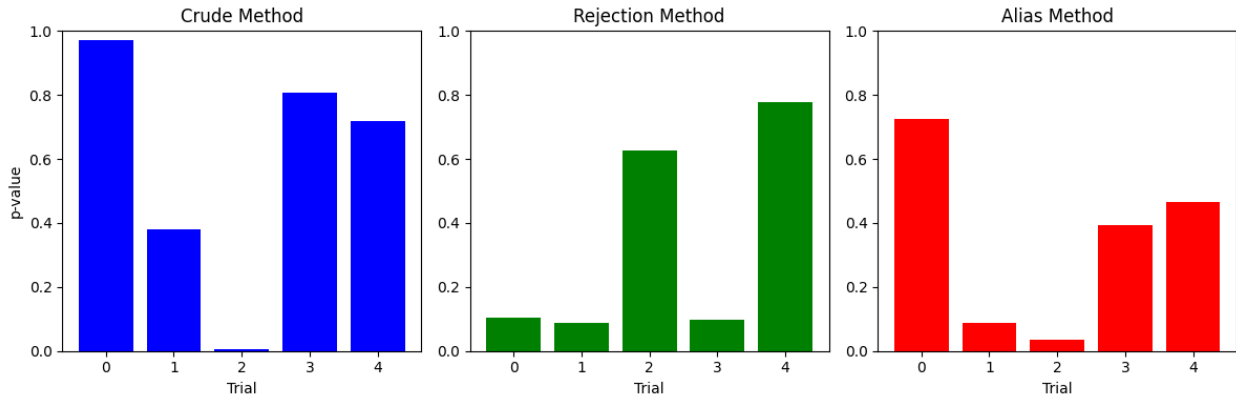
for _ in range(num_trials):
    pvals_crude.append(chi2_test(discrete_crude(n, P), P,
    check=False))
    pvals_rejection.append(chi2_test(discrete_rejection(n, P), P,
    check=False))
    pvals_alias.append(chi2_test(discrete_alias(n, fs, ls), P,
    check=False))

plt.figure(figsize=(12, 4))
plt.subplot(1, 3, 1)
plt.bar(range(num_trials), pvals_crude, color='blue')
plt.title("Crude Method")
plt.ylim(0, 1)
plt.xlabel("Trial")
plt.ylabel("p-value")

plt.subplot(1, 3, 2)
plt.bar(range(num_trials), pvals_rejection, color='green')
plt.title("Rejection Method")
plt.ylim(0, 1)
plt.xlabel("Trial")

plt.subplot(1, 3, 3)
plt.bar(range(num_trials), pvals_alias, color='red')
plt.title("Alias Method")
plt.ylim(0, 1)
plt.xlabel("Trial")

plt.tight_layout()
plt.show()
```



These results show that the p-values vary between runs for each of the three sampling methods. This inherent variability is expected due to the randomness in sampling and the finite sample size ( $n = 10,000$ ). However, as the number of samples  $n \rightarrow \infty$ , the empirical distribution produced by each method should converge to the true target distributions. And naturally, the p-values from the  $\chi^2$ -tests should tend toward a uniform distribution on  $[0, 1]$ .

3. Compare the three different methods using adequate criteria, then discuss the results.

```
results = {}
for name, method in [
    ('Direct Method', lambda: discrete_crude(n, P)),
    ('Rejection Method', lambda: discrete_rejection(n, P)),
    ('Alias Method', lambda: discrete_alias(n, *generate_f_and_l(P)))
]:
    start = time.time()
    samples = method()
    end = time.time()

    obs_counts = np.array([np.sum(samples == xi) for xi in bins])
    expected_counts = P * n
    obs_dist = obs_counts / n
    kl = entropy(obs_dist + 1e-10, P)
    mae = np.mean(np.abs(obs_counts - expected_counts))
    runtime_ms = (end - start) * 1000

    results[name] = {
        'samples': samples,
        'kl': kl,
        'mae': mae,
        'runtime_ms': runtime_ms
    }

# --- Plotting ---
fig, axs = plt.subplots(1, 3, figsize=(18, 5))
colors = ['blue', 'green', 'red']
```

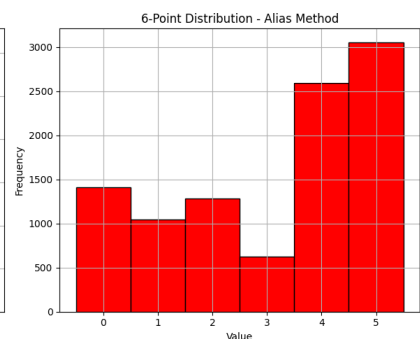
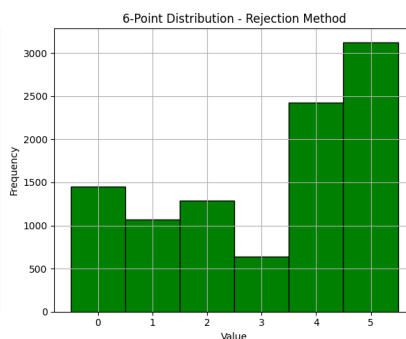
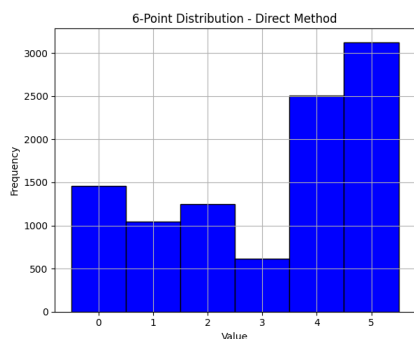
```

for i, (method_name, res) in enumerate(results.items()):
    axs[i].hist(res['samples'], bins=np.arange(len(P)+1)-0.5,
edgecolor='black', color=colors[i])
    axs[i].set_title(f"6-Point Distribution - {method_name}")
    axs[i].set_xlabel("Value")
    axs[i].set_ylabel("Frequency")
    axs[i].set_xticks(X)
    axs[i].grid(True)

plt.tight_layout()
plt.show()

# --- Print Metrics Table ---
print(f"{'Method':<20} {'KL Divergence':<15} {'MAE':<10} {'Runtime (ms)':<15}")
for name, res in results.items():
    print(f"{name:<20} {res['kl']:<15.6f} {res['mae']:<10.2f} {res['runtime_ms']:<15.2f}")

```



Method	KL Divergence	MAE	Runtime (ms)
Direct Method	0.000017	4.78	4.37
Rejection Method	0.000225	26.78	2701.49
Alias Method	0.000372	40.78	1206.61

All three methods approximate the target 6-point distribution, but with varying levels of accuracy. They all share almost the same histograms.

The Direct method achieves the lowest KL divergence (Kullback-Leibler divergence, meaning how much information is lost when we approximate the true distribution using the method), and also the lowest MAE (mean average error). Furthermore, it is also the fastest method.

The Rejection method has a higher KL divergence and MAE than the Direct method, while also taking massively longer to run.

The Alias method performs the worst of the bunch in terms of KL divergence and MAE, but does have a better runtime.

However, as this is a somewhat small-scale sampling with only  $n=10000$

4. Give recommendations of how to choose the best suite method in different settings, i.e., discuss the advantages and drawbacks of each method. If time permits substantiate by running experiments.

In this small case, the Direct method proved to be the best in terms of KL divergence, MAE, and runtime. However, if we were to upscale the number of samples  $n$ , say to millions, the Alias method would be more advantageous, since its sampling step runs in constant time  $O(1)$ , after an initial preprocessing phase. In contrast, the Direct and Rejection methods require  $O(nk)$ , which becomes increasingly costly as the number of classes  $k$  grows. But since this is a relatively small-scale setup, the Direct and Rejection methods outperform the Alias method.

## Exercise 3

1. Generate simulated values from the following distributions

(a) Exponential distribution

(b) Normal distribution (at least with standard Box-Mueller)

(c) Pareto distribution, with  $\beta = 1$  and experiment with different values of  $k$  values:  $k = 2.05$ ,  $k = 2.5$ ,  $k = 3$  and  $k = 4$ .

Verify the results by comparing histograms with analytical results and perform tests for distribution type.

1. Generate simulated values from the following distributions

(a) Exponential distribution

(b) Normal distribution (at least with standard Box-Mueller)

(c) Pareto distribution, with  $\beta = 1$  and experiment with different values of  $k$  values:  $k = 2.05$ ,  $k = 2.5$ ,  $k = 3$  and  $k = 4$ .

Verify the results by comparing histograms with analytical results and perform tests for distribution type.

(a) To generate values from an exponential distribution, we use the inverse transform method. We know that the exponential distribution has the following CDF:

$$F(x) = 1 - e^{-\lambda x}$$

If we solve for  $x = F^{-1}(u)$ , we get:

$$x = -\frac{1}{\lambda} \log(U), U \sim \text{Uniform}(0, 1)$$

For simplicity's sake, we set  $\lambda = 1$  and do  $n = 10000$  runs. We plot the distribution of generated values together with the theoretical PDF, and also do a KS test:

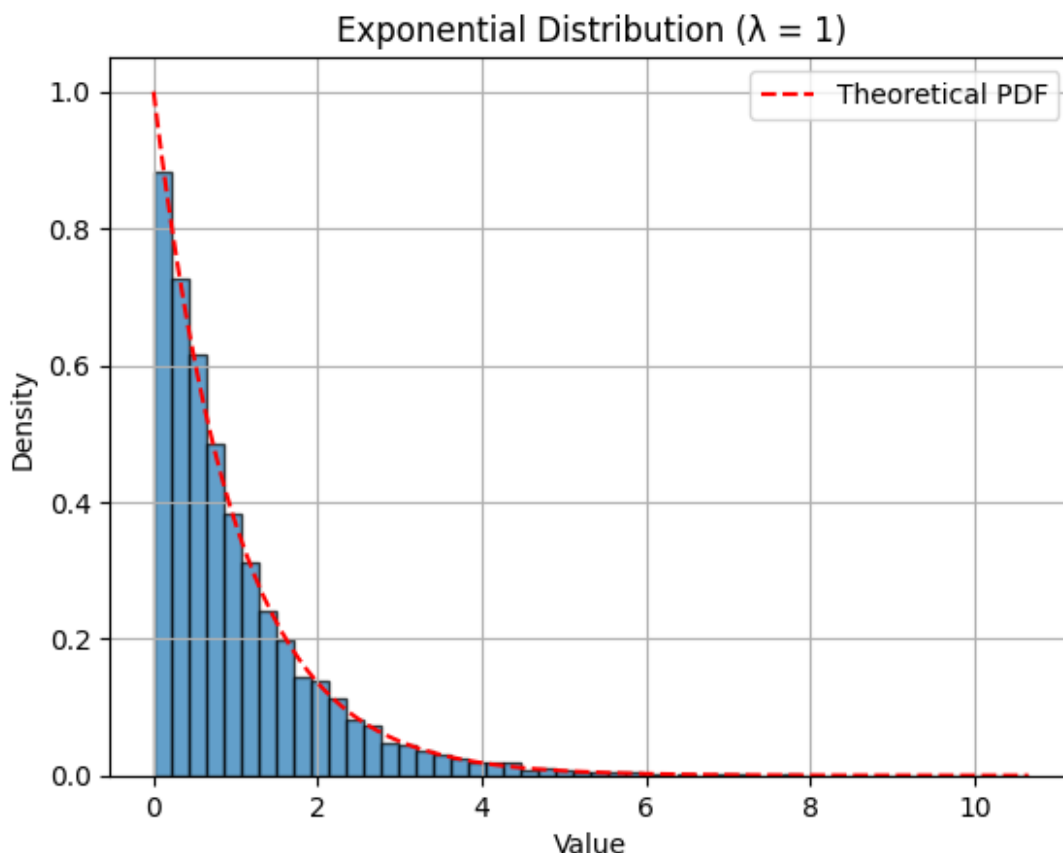
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import expon, kstest, norm
n = 10000
lam = 1

U = np.random.uniform(0, 1, n)
samples = -np.log(U) / lam

plt.hist(samples, bins=50, edgecolor='black', alpha=0.7, density=True)
x = np.linspace(0, np.max(samples), 200)
plt.plot(x, expon.pdf(x, scale=1/lam), 'r--', label='Theoretical PDF')
plt.title("Exponential Distribution ( $\lambda = 1$ )")
plt.xlabel("Value")
plt.ylabel("Density")
plt.legend()
plt.grid(True)
plt.show()

D, p = kstest(samples, 'expon', args=(0, 1/lam))
print(f"KS test p-value: {p:.4f}")
```





KS test p-value: 0.6099

KS test p-value: 0.6099

As can be observed, the simulated values from the exponential distribution with  $\lambda=1$  match the theoretical PDF, and the high KS test p-value confirms an excellent fit of the distribution.

(b) Here, we do the same but for a normal distribution with a standard Box-Mueller transformation. Specifically, the Box-Mueller transform is a method to generate samples from a standard normal distribution using two independent uniform random variables,  $U_1, U_2 \sim \text{Uniform}(0, 1)$ . After these have been generated, then  $Z_1$  and  $Z_2$  are computed:

$$Z_1 = \sqrt{-2 \log U_1} \cos(2\pi U_2)$$

$$Z_2 = \sqrt{-2 \log U_1} \sin(2\pi U_2)$$

In turn,  $Z_1, Z_2 \sim N(0, 1)$

We generate 10000 values using this method:

```
U1 = np.random.uniform(0, 1, n//2)
U2 = np.random.uniform(0, 1, n//2)
```

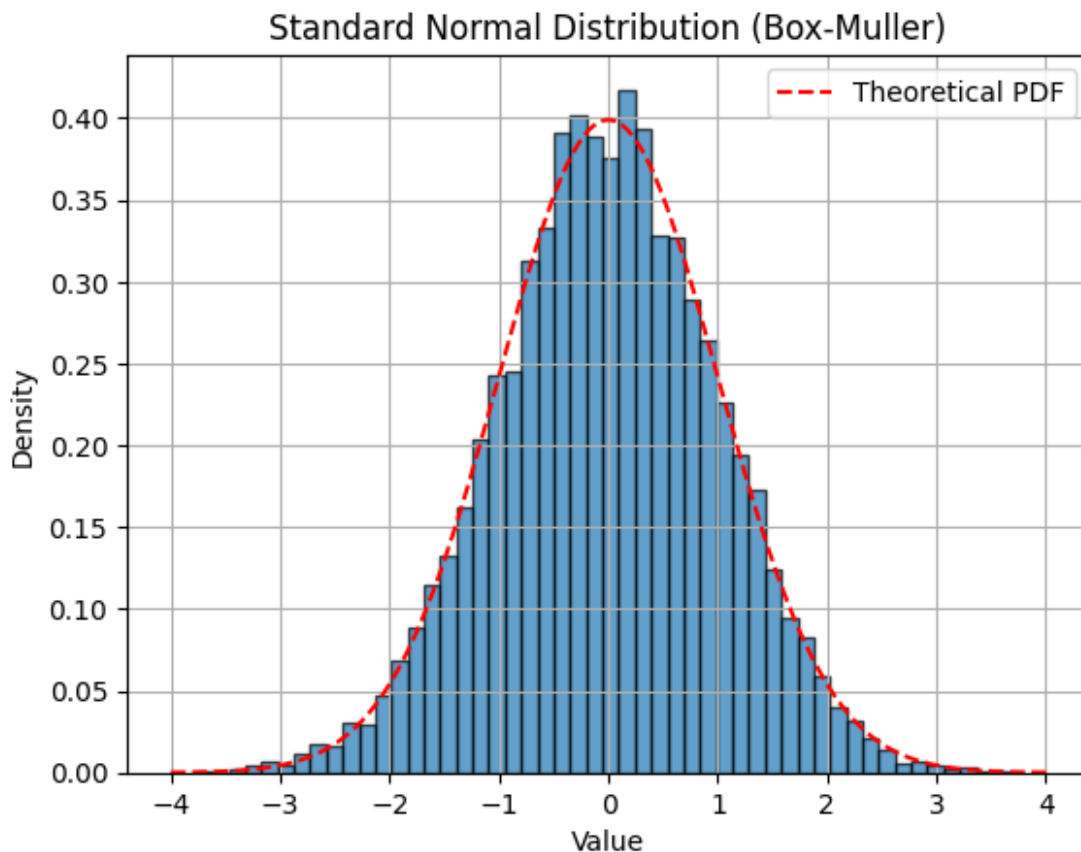
```

Z1 = np.sqrt(-2 * np.log(U1)) * np.cos(2 * np.pi * U2)
Z2 = np.sqrt(-2 * np.log(U1)) * np.sin(2 * np.pi * U2)
samples = np.concatenate([Z1, Z2])

plt.hist(samples, bins=50, density=True, edgecolor='black', alpha=0.7)
x = np.linspace(-4, 4, 200)
plt.plot(x, norm.pdf(x), 'r--', label='Theoretical PDF')
plt.title("Standard Normal Distribution (Box-Muller)")
plt.xlabel("Value")
plt.ylabel("Density")
plt.grid(True)
plt.legend()
plt.show()

D, p = kstest(samples, 'norm')
print(f"KS test p-value: {p:.4f}")

```



KS test p-value: 0.7394

KS test p-value: 0.7394

This shows that the Box-Mueller method is capable of accurately generating samples from the standard normal distribution, as the theoretical pdf matches closely to the 10000 generated values. Furthermore, the KS test value indicates no significant deviation from normality.

(c) Now, we generate 10000 values using the Pareto distribution

In the textbook, the CDF of the Pareto distribution is

$$F(x) = 1 - \left(\frac{\beta}{x}\right)^k, \text{ for } x \geq \beta$$

When executing the inverse transformation of this, the following is reached:

Solving this for  $x$  results in:

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

Here,  $U \sim \text{Uniform}(0, 1)$ .

Using this, 10000 values can be created for each different value of  $k$ :

```
k_values = [2.05, 2.5, 3, 4]
beta = 1

fig, axs = plt.subplots(2, 2, figsize=(14, 10))
axs = axs.flatten()

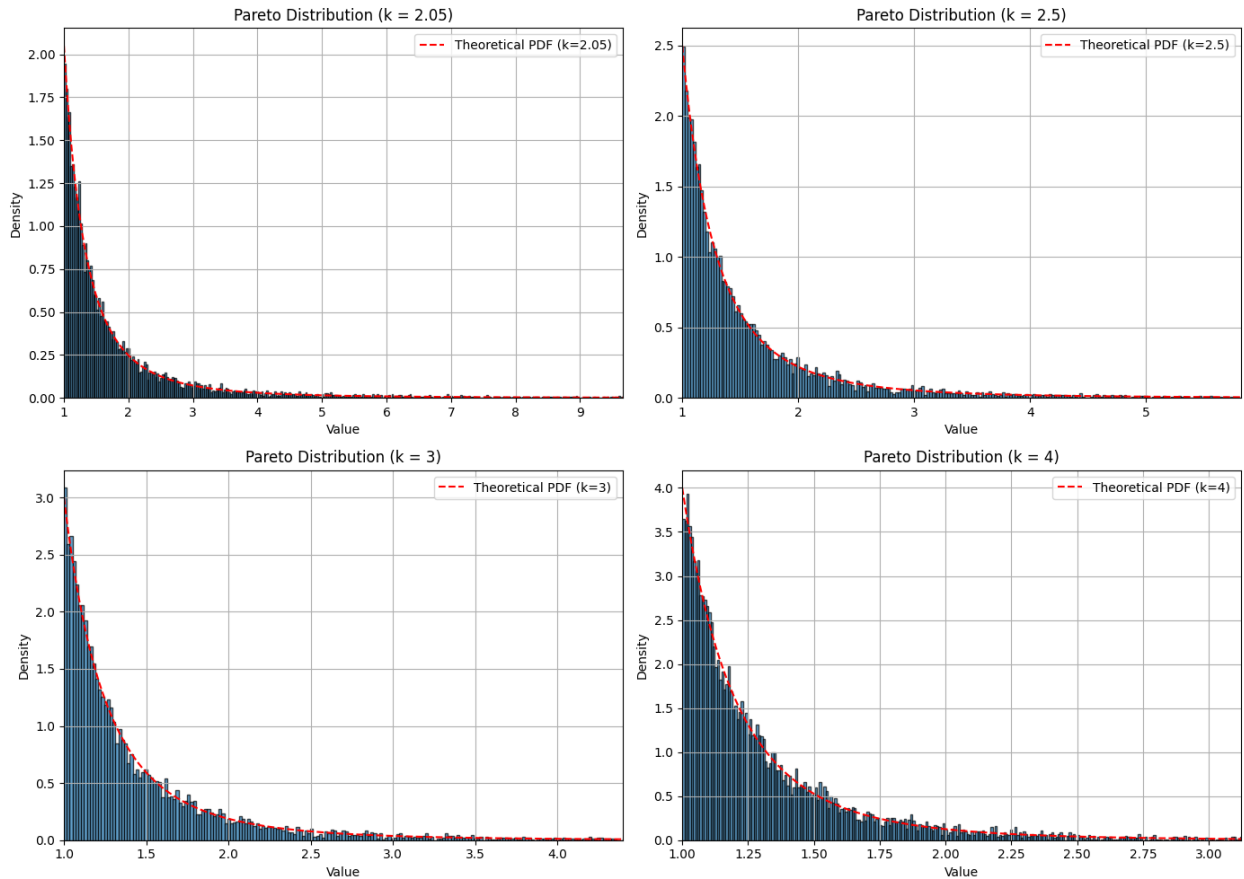
for i, k in enumerate(k_values):
    U = np.random.uniform(0, 1, n)
    samples = beta / ((1 - U) ** (1 / k))
    x_max = np.percentile(samples, 99)
    x = np.linspace(beta, x_max, 300)

    axs[i].hist(samples, bins=1500, density=True, alpha=0.7,
edgecolor='black')
    axs[i].plot(x, pareto.pdf(x, b=k, scale=beta), 'r--',
label=f'Theoretical PDF (k={k})')
    axs[i].set_title(f'Pareto Distribution (k = {k})')
    axs[i].set_xlabel('Value')
    axs[i].set_ylabel('Density')
    axs[i].set_xlim([beta, x_max])
    axs[i].grid(True)
    axs[i].legend()

    D, p = kstest(samples, 'pareto', args=(k, 0, beta))
    print(f"KS test p-value for k = {k}: {p:.4f}")

plt.tight_layout()
plt.show()
```

KS test p-value for  $k = 2.05$ : 0.3821  
KS test p-value for  $k = 2.5$ : 0.9570  
KS test p-value for  $k = 3$ : 0.1476  
KS test p-value for  $k = 4$ : 0.8333



KS test p-value for  $k = 2.05$ : 0.3821

KS test p-value for  $k = 2.5$ : 0.9570

KS test p-value for  $k = 3$ : 0.1476

KS test p-value for  $k = 4$ : 0.8333

The histograms show that the simulated Pareto distributions closely follow the theoretical PDFs for all tested  $k$ . As  $k$  increases, the tail becomes thinner and the distribution becomes more concentrated near 1. The KS test p-values are all above 0.05, confirming that the samples are consistent with the target Pareto distributions.

2. For the Pareto distribution with support on  $[\beta, \infty[$  compare mean value and variance, with analytical results, which can be calculated as  $E(X) = \beta * k / (k-1)$  (for  $k > 1$ ) and  $Var(X) = \beta^2 * (k / ((k-1)^2 * (k-2)))$ . Explain problems if any

Now, we compared the analytical results with the mean and variance of the Pareto distribution,  $E(X)$  and  $Var(X)$ , with the 4 different values of  $k$  and  $\beta=1$ :

```
print(f"{'k':<6} {'Empirical Mean':<18} {'Theoretical Mean':<20}
{'Empirical Var':<18} {'Theoretical Var'}")

for k in k_values:
    U = np.random.uniform(0, 1, n)
    samples = (1 / (1 - U))**(1 / k)

    empirical_mean = np.mean(samples)
    empirical_var = np.var(samples)

    if k > 1:
        theoretical_mean = beta * k / (k - 1)
    else:
        theoretical_mean = np.nan

    if k > 2:
        theoretical_var = (beta**2) * k / ((k - 1)**2 * (k - 2))
    else:
        theoretical_var = np.nan

    print(f"{'k':<6} {'empirical_mean':<18.4f} {'theoretical_mean':<20.4f}
{'empirical_var':<18.4f} {'theoretical_var'}")
```

k	Empirical Mean	Theoretical Mean	Empirical Var	Theoretical Var
2.05	1.9080	1.9524	4.3038	37.18820861678019
2.5	1.6602	1.6667	2.0525	2.222222222222223
3	1.5040	1.5000	0.7962	0.75
4	1.3254	1.3333	0.1946	0.222222222222222

```
\begin{array}{|c|c|c|c|c|} \hline k & \text{Empirical Mean} & \text{Theoretical Mean} & \text{Empirical Var} & \text{Theoretical Var} \\ \hline 2.05 & 1.9080 & 1.9524 & 4.3038 & 37.1882 \\ 2.5 & 1.6602 & 1.6667 & 2.0525 & 2.2222 \\ 3 & 1.5040 & 1.5000 & 0.7962 & 0.7500 \\ 4 & 1.3254 & 1.3333 & 0.1946 & 0.2222 \\ \hline \end{array}
```

For all values of  $k$ , the empirical means closely match the theoretical means, which aligns with the known properties of the Pareto distribution. However, for  $k=2.05$ , while the mean is defined, the variance is finite but extremely large, and becomes infinite as  $k \rightarrow 2$ . This explains

the large observed empirical variance and illustrates the sensitivity of the distribution's second moment in the heavy-tailed regime. For higher values of  $k \geq 2.5$ , both mean and variance converge more reliably to their theoretical values, and the distribution stabilizes. Thus, increasing  $k$  does reduce the variance and thus, improve the alignment of the theoretical and empirical moments.

### 3. For the normal distribution generate 100 95% confidence intervals for the mean and variance, each based on 10 observations. Discuss the results

This code simulates 100 independent samples of size 10 from a standard normal distribution, and constructs 95% confidence intervals for both the mean and variance. Afterwards, it estimates the empirical coverage probability by counting how often the true values fall within these intervals.

```
n = 10
N = 100
true_mean = 0
true_var = 1

mean_contains = 0
var_contains = 0

alpha = 0.05
t_crit = t.ppf(1 - alpha / 2, df=n - 1)

for _ in range(N):
    data = np.random.normal(0, 1, n)
    xbar = np.mean(data)
    s = np.std(data, ddof=1)
    s2 = np.var(data, ddof=1)

    ci_mean = (xbar - t_crit * s / np.sqrt(n), xbar + t_crit * s /
np.sqrt(n))
    if ci_mean[0] <= true_mean <= ci_mean[1]:
        mean_contains += 1

    chi2_lower = chi2.ppf(alpha / 2, df=n - 1)
    chi2_upper = chi2.ppf(1 - alpha / 2, df=n - 1)
    ci_var = ((n - 1) * s2 / chi2_upper, (n - 1) * s2 / chi2_lower)
    if ci_var[0] <= true_var <= ci_var[1]:
        var_contains += 1

print(f"Mean CI coverage: {mean_contains} / {N}")
print(f"Variance CI coverage: {var_contains} / {N}")

Mean CI coverage: 95 / 100
Variance CI coverage: 92 / 100
```

Out of 100 simulations, 95 out of 100 confidence intervals for the means contained the true population mean, which is 0.

And out of the 100 simulation, 92 out of 100 confidence intervals for the variance contained the true population mean, which is 1.

This shows that not every time the 95% confidence intervals capture the true parameter, but they do do it successfully 95% of the time. If we were to increase  $N$ , we would approximate 95 better.

## 4. Simulate from the Pareto distribution using composition.

In order to simulate from the Pareto distribution, we utilize the inverse transform method. This technique involves generating samples from a distribution,  $X \sim \text{Uniform}(0,1)$ , and then applying the inverse of the Pareto CDF:

$$F^{-1}(u) = \beta \cdot (1 - u)^{1/k}, \text{ for } u \in (0, 1)$$

If we solve this for  $x$  in terms of  $u = F(x)$ , we get:

$$u = 1 - \left(\frac{\beta}{x}\right)^k \rightarrow x = \frac{\beta}{U^{1/k}}$$

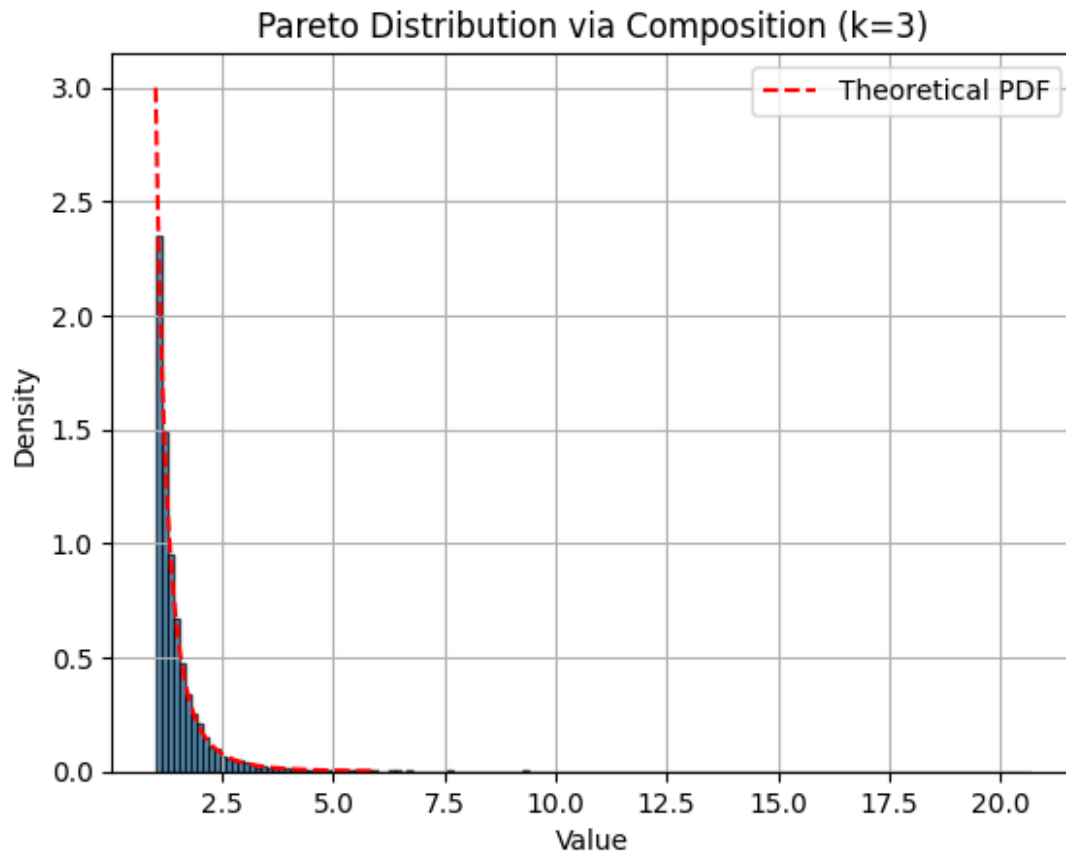
This, we can plot:

```
n = 10000
k = 3
beta = 1

U = np.random.uniform(0, 1, n)
X = beta / U**(1 / k)

x = np.linspace(1, np.percentile(X, 99.5), 200)
plt.hist(X, bins=150, density=True, edgecolor='black', alpha=0.7)
plt.plot(x, pareto.pdf(x, b=k, scale=beta), 'r--', label='Theoretical PDF')
plt.title(f'Pareto Distribution via Composition (k={k})')
plt.xlabel('Value')
plt.ylabel('Density')
plt.grid(True)
plt.legend()
plt.show()

D, p = kstest(X, 'pareto', args=(k, 0, beta))
print(f"KS test p-value: {p:.4f}")
```



KS test p-value: 0.8516

Here, we see the generated samples follow the theoretical PDF. Furthermore, the high KS indicates that the generated values are in fact consistent with a Pareto distribution.



## 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. This system is sufficiently simple such that the analytical solution is known. See the last slide for the solution. Verify your simulation program using this knowledge.

In this assignment, we are supposed to simulate a blocking system with no waiting room, meaning arriving customers either get served or blocked, if no servers are available. We are given:

- $m=10$  servers
- mean service time 8
- interarrival time 1
- we should run with 10 batches of 10.000 customers each
- service times are exponential

Using this information, we can begin modelling the problem as a discrete event simulation. I use Poisson-distributed arrivals and exponential arrival times, whereof I can calculate how many customers get blocked when entering the system:

```
import numpy as np
import heapq
import math
from scipy.stats import expon, erlang, bernoulli, norm, pareto,
rv_continuous
import matplotlib.pyplot as plt
from enum import Enum
from heapq import heappush, heappop, heapify
np.random.seed(1234)

class Event(Enum):
```

```

ARRIVED = "Arrived"
SERVICED = "Serviced"
BLOCKED = "Blocked"

def simulate_system(inter_arrival_dist, service_time_dist, m=10,
n_customers=10_000):

    service_units = [i for i in range(m)]

    arrival_times =
np.cumsum(inter_arrival_dist.rvs(size=n_customers))

    events = [(time.item(), Event.ARRIVED) for time in arrival_times]
    heapify(events)

    n_serviced = 0
    n_blocked = 0
    events_processed = []

    while events:
        event = heappop(events)

        match event:
            case (time, Event.ARRIVED):
                if service_units:
                    new_event = (
                        time + service_time_dist.rvs(),
                        Event.SERVICED,
                        service_units.pop(),
                    )
                else:
                    new_event = (
                        time,
                        Event.BLOCKED,
                    )
                heappush(events, new_event)
            case (_, Event.SERVICED, service_unit):
                service_units.append(service_unit)
                n_serviced += 1
            case (_, Event.BLOCKED):
                n_blocked += 1

        events_processed.append(event)

    return n_serviced, n_blocked, events_processed

def simulate_and_print_stats(inter_arrival_dist, service_time_dist,
iterations=10, m=10, n_customers=10_000):

```

```

blocked = np.empty(iterations)
for i in range(iterations):
    _, n_blocked, events = simulate_system(inter_arrival_dist,
service_time_dist, m, n_customers)
    blocked[i] = n_blocked

blocked /= n_customers

mean = np.mean(blocked)
std = np.std(blocked)
ci_low = mean - 1.96 * std
ci_high = mean + 1.96 * std

fig = plt.figure(figsize=(20, 9))
gs = fig.add_gridspec(2, 3)
ax1 = fig.add_subplot(gs[0, 0])
ax2 = fig.add_subplot(gs[0, 1])
ax3 = fig.add_subplot(gs[1, :])
ax4 = fig.add_subplot(gs[0, 2])

xs = np.linspace(0, 20, 200)

ax1.set_title("Inter-arrival distribution")
ax1.plot(xs, inter_arrival_dist.pdf(xs))
m_arrival = inter_arrival_dist.mean()
ax1.axvline(m_arrival, color="g", linestyle="--", label=f"Mean:
{m_arrival:.2f}")
ax1.legend()

ax2.set_title("Service time distribution")
ax2.plot(xs, service_time_dist.pdf(xs))
m_service = service_time_dist.mean()
ax2.axvline(m_service, color="g", linestyle="--", label=f"Mean:
{m_service:.2f}")
ax2.legend()

arrivals = [event[0] for event in events if event[1] ==
Event.ARRIVED]
blocks = [event[0] for event in events if event[1] ==
Event.BLOCKED]
ax3.set_title("Histogram of events (Last run)")
ax3.hist(arrivals, bins=100, label="Arrivals")
ax3.hist(blocks, bins=100, label="Blocked")
ax3.legend()

ax4.set_title(f"Block rate: {mean:.3f} ± {1.96 * std:.5f}")
ax4.bar(np.arange(iterations) + 1, blocked, alpha=0.8)
ax4.hlines(mean, xmin=0, xmax=iterations + 1, label="Mean")
ax4.hlines(ci_high, xmin=0, xmax=iterations + 1, linestyle="--",
label="95% interval")

```

```

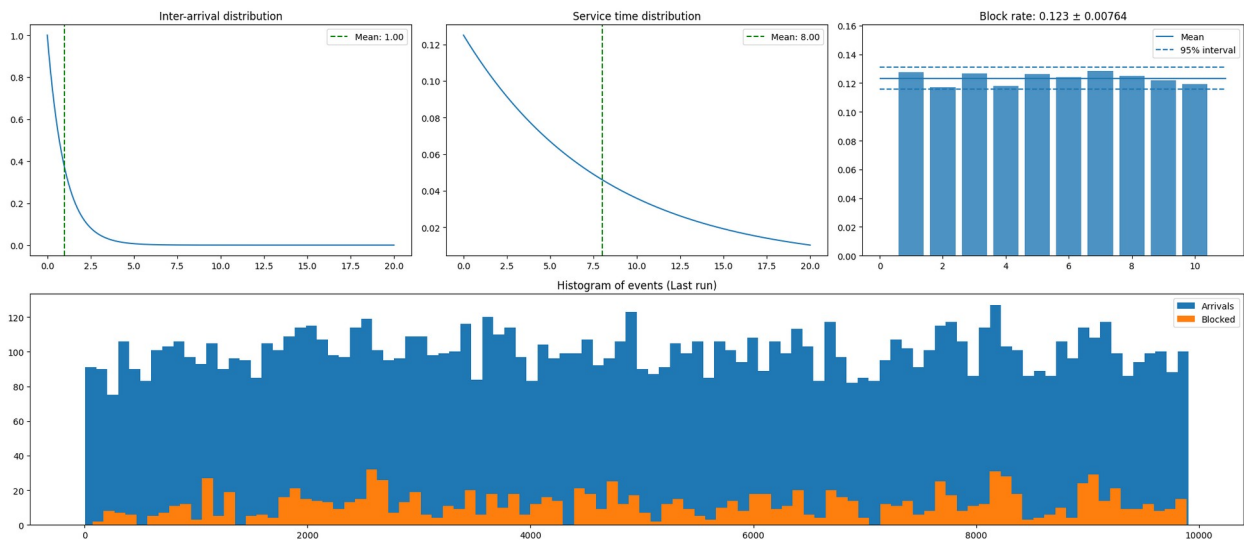
ax4.hlines(ci_low, xmin=0, xmax=iterations + 1, linestyle="--")
ax4.set_ylim(0, 1.25 * np.max(blocked))
ax4.legend()
fig.tight_layout(rect=[0, 0.03, 1, 1])
plt.show()

print("\nEstimated Blocking Probability: {:.4f}".format(mean))
print("95% Confidence Interval: ({:.4f}, {:.4f})".format(ci_low,
ci_high))

inter_arrival_dist = expon(scale=1)
service_time_dist = expon(scale=8)

mean = simulate_and_print_stats(inter_arrival_dist, service_time_dist)

```



```

Estimated Blocking Probability: 0.1234
95% Confidence Interval: (0.1158, 0.1310)

```

We observe the distribution of inter-arrivals, having a mean of 1, following an exponential distribution. Furthermore, the distribution of the service time also follows an exponential distribution, with a mean of 8. The blocking rate of the  $m=10$  servers can be observed to the right plot, wherein the average blocking rate is 12.04%. This means that circa 12% of customers were blocked when arriving, meaning 12% of customers came at a time where all servers  $m$  were occupied and thus, these customers were not served. The 95%-confidence also lays nicely around 12%, meaning we are quite certain that between 11% - 13% of customers would in general be blocked when arriving. When we see in the large plot that this blocking percentages fluctuates and different amounts of customers arrive at different time - some coming in bulk which puts more strain on the system.

Now, we can compare it to the analytical Erlang B result, which is calculates the blocking probability  $B(m, a)$ :

$$B(m, a) = \frac{\frac{A^m}{m!}}{\sum_{k=0}^m \frac{A^k}{k!}}$$

Here,  $m$  is the number of servers, and  $A$  is the offered traffic in Erlangs,  $A = s \cdot \lambda = 8$  Erlang:

```
A = 8
m = 10

def factorial(n):
    if n in (0, 1):
        return 1
    return n * factorial(n - 1)

a = A ** m / factorial(m)
b = sum(A ** i / factorial(i) for i in range(m + 1))

print(f"Analytical Erlang B blocking probability: {a / b:.4f}")
print(f"Absolute error vs simulation: {abs((a / b) - 0.1224) :.6f}")

Analytical Erlang B blocking probability: 0.1217
Absolute error vs simulation: 0.000739
```

Analytical Erlang B blocking probability: 0.1217

Absolute error vs simulation: 0.000739

Since there is only an absolute error of -0.0007, the simulation can be deemed a success, since it produced results consistent with the theoretical Erlang B model.

Furthermore, since the theoretical Erlang B blocking probability is within the 95%-confidence interval for the simulation of (0.1158, 0.1310), thus, the simulation is statistically consistent with the theoretical Erlang B model.

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

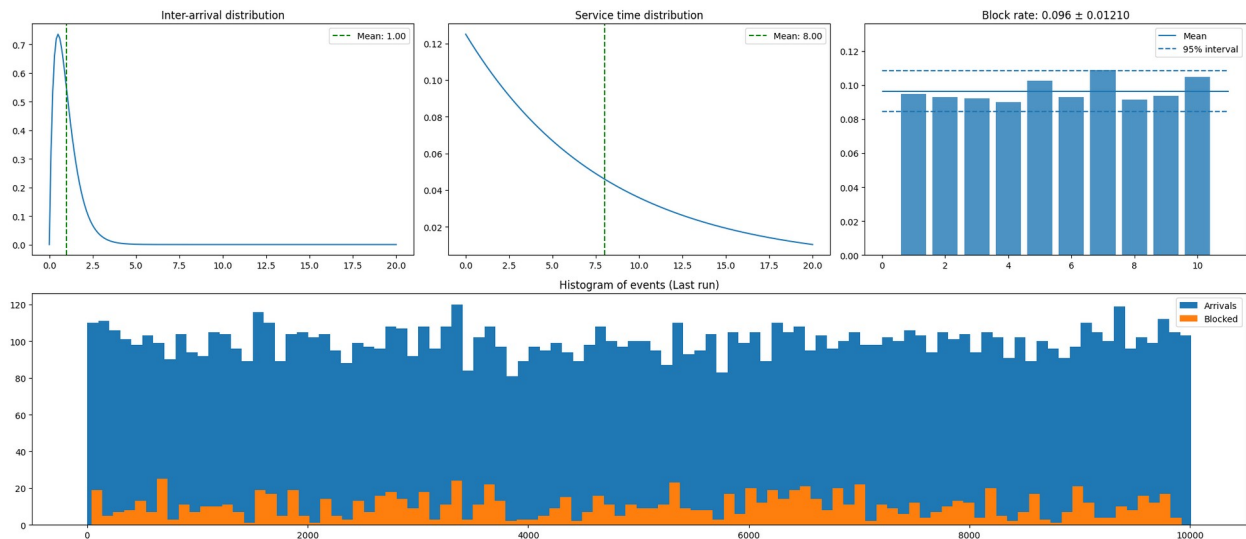
(a) Experiment with Erlang distributed inter arrival times The Erlang distribution should have a mean of 1

(a) To explore the effect of more regular arrival patterns, we simulate the same blocking system using Erlang-distributed interarrival times, specifically an Erlang(2, 2) distribution. This

maintains the same mean of 1 as the original exponential case but with lower variability. The rest of the system parameters remain unchanged.

```
inter_arrival_dist = erlang(a=2, scale=1/2)
service_time_dist = expon(scale=8)

simulate_and_print_stats(inter_arrival_dist, service_time_dist)
```



Estimated Blocking Probability: 0.0964  
95% Confidence Interval: (0.0843, 0.1085)

The simulation shows a blocking probability of approximately 9.6%, with a 95% confidence interval ranging from 8.43% to 10.85%.

This makes sense, since the Erlang(2,2) distribution has the same mean as the exponential distribution but lower variability. This results in more regular and evenly spaced arrivals, which reduces the chance that many customers arrive in a short span and overwhelm the system. Thus, the blocking probability is lower than with the more variable exponential arrivals, leading to a lower blocking probability.

(b) hyper exponential inter arrival times. The parameters for the hyper exponential distribution should be  $p_1 = 0.8$ ,  $\lambda_1 = 0.8333$ ,  $p_2 = 0.2$ ,  $\lambda_2 = 5.0$ .

(b) Now, we will utilize hyper exponential arrival times in the simulation. We are given the values:  $p_1=0.8$ ,  $p_2=0.2$ ,  $\lambda_1=0.8333$ , and  $\lambda_2=5$ :

```
class hyperexpon:
    def __init__(self, p1, lambda1, lambda2):
        self.p1 = p1
```

```

self.expon1 = expon(scale=1/lambda1)
self.expon2 = expon(scale=1/lambda2)

def mean(self):
    return self.p1 * self.expon1.mean() + (1 - self.p1) *
self.expon2.mean()

def rvs(self, size=1):
    mask = bernoulli.rvs(p=self.p1, size=size) == 1
    expon1 = self.expon1.rvs(size=size)
    expon2 = self.expon2.rvs(size=size)

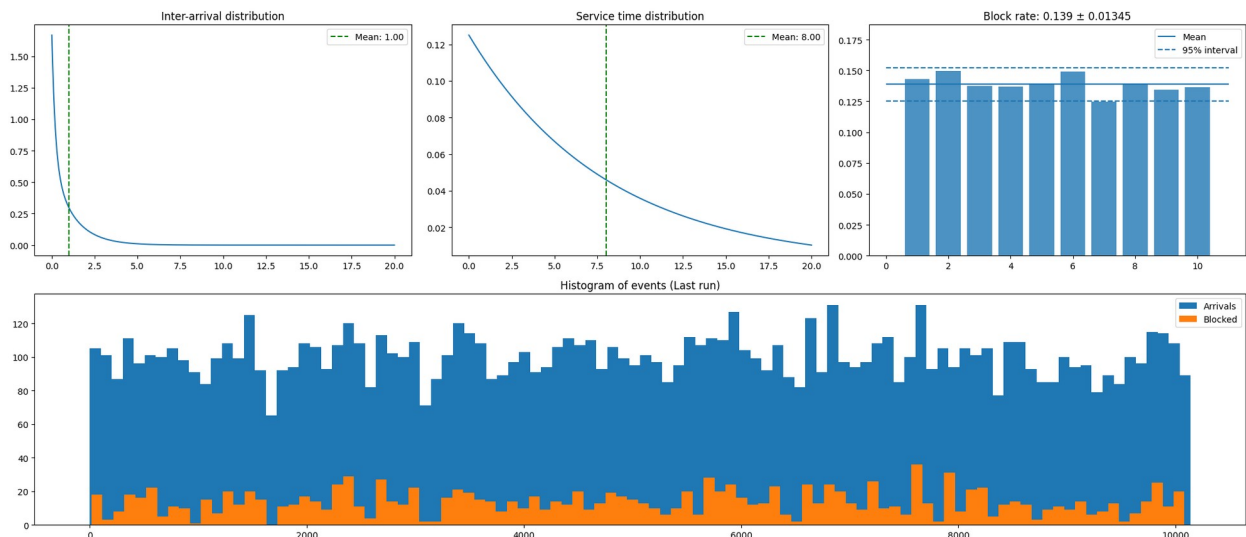
    return np.where(mask, expon1, expon2)

def pdf(self, xs):
    return self.p1 * self.expon1.pdf(xs) + (1 - self.p1) *
self.expon2.pdf(xs)

inter_arrival_dist = hyperexpon(p1=0.8, lambda1=0.8333, lambda2=5.0)
service_time_dist = expon(scale=8)

simulate_and_print_stats(inter_arrival_dist, service_time_dist)

```



Estimated Blocking Probability: 0.1389  
 95% Confidence Interval: (0.1254, 0.1523)

This yields a blocking probability of circa 13.89%, which is somewhat higher than both the exponential and Erlang cases, reflecting the increased variability in the arrival process. Specifically, with hyper exponential inter arrival times results in more customers arriving in short time frames, increasing the likelihood of customers not being served (i.e., the blocking probability).

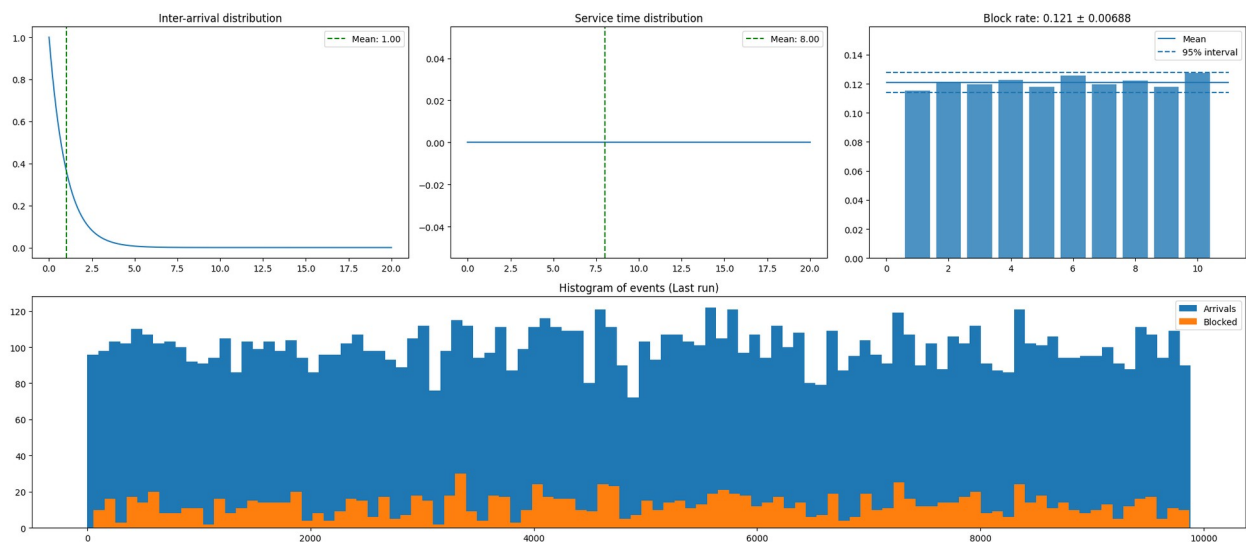
3. The arrival process is again a Poisson process like in Part 1. Experiment with different service time distributions with the same mean service time and  $m$  as in Part 1 and Part 2.

### (a) Constant service time

(a) The arrival process is once again a Poisson process. We set the average service time to a constant, 8:

```
inter_arrival_dist = expon(scale=1)
service_time_dist = type('', (), {
    'rvs': staticmethod(lambda size=None: 8 if size is None else
np.full(size, 8)),
    'pdf': staticmethod(lambda x: np.where(x == 8, 1.0, 0.0)),
    'mean': staticmethod(lambda: 8)
})() # Constant service time

simulate_and_print_stats(inter_arrival_dist, service_time_dist)
```



Estimated Blocking Probability: 0.1209  
95% Confidence Interval: (0.1140, 0.1277)

This is quite similar to the exponential case, it makes sense since the mean service time is the same. While this case reduces service variability, it does not affect the inherent randomness of the arrivals.



(b) Pareto distributed service times with at least  $k = 1.05$  and  $k = 2.05$ .

(b) Now, we set the service time to be both Pareto distributed with atleast  $k_1=1.05$  and maximumally  $k_2=2.05$ . In order to calculate the two  $\beta$  values, we utilize:

$$\beta = \frac{8(k-1)}{k}$$

```
# 4.3.b
```

```
k1 = 1.05
```

```
beta1 = 8 * (k1 - 1) / k1
```

```
service_time_dist = pareto(b=k1, scale=beta1)
```

```
inter_arrival_dist = expon(scale=1)
```

```
print("For k=1.05:")
```

```
simulate_and_print_stats(inter_arrival_dist, service_time_dist)
```

```
k2 = 2.05
```

```
beta2 = 8 * (k2 - 1) / k2
```

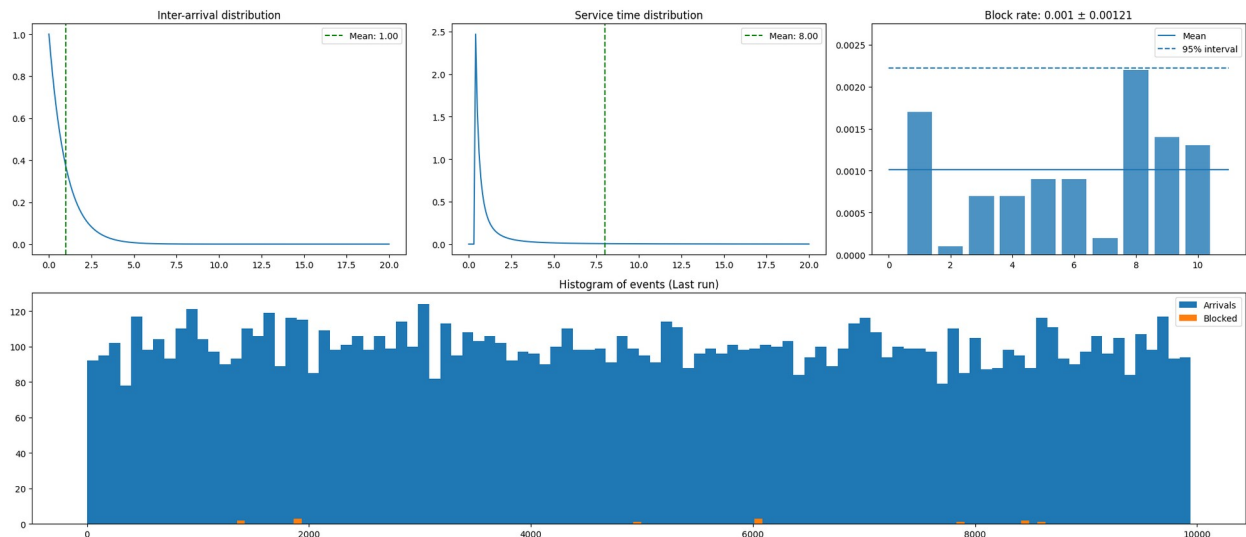
```
service_time_dist = pareto(b=k2, scale=beta2)
```

```
inter_arrival_dist = expon(scale=1)
```

```
print("\nFor k=2.05:")
```

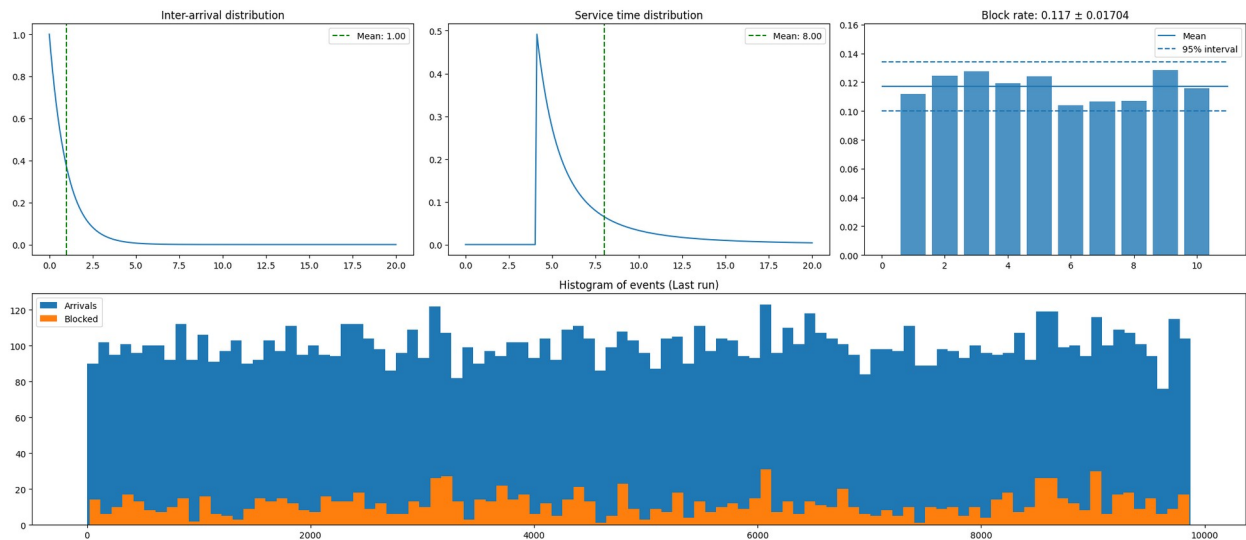
```
simulate_and_print_stats(inter_arrival_dist, service_time_dist)
```

For k=1.05:



Estimated Blocking Probability: 0.0010  
95% Confidence Interval: (-0.0002, 0.0022)

For k=2.05:



Estimated Blocking Probability: 0.1170  
 95% Confidence Interval: (0.0999, 0.1340)

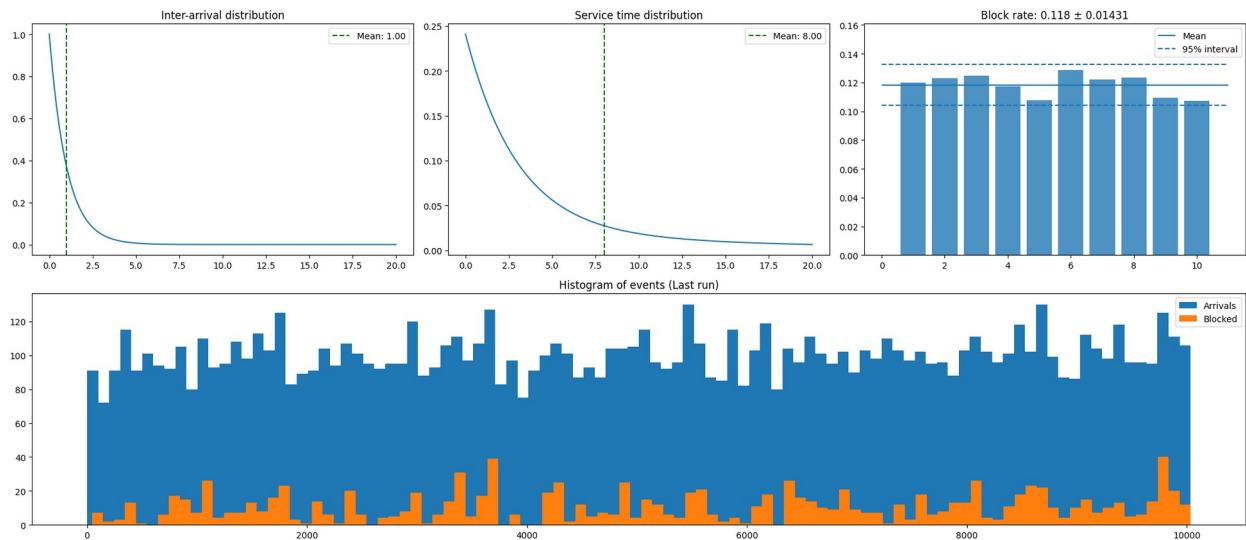
With  $k=1.05$ , the blocking probability is surprisingly low, despite the extremely heavy-tailed nature of the service time distribution. This is because most service times are actually very short, with only rare but extremely long jobs. These long jobs are infrequent enough that they do not cause significant blocking in the system. On the other hand, with  $k=2.05$ , service times are more concentrated but still variable, leading to more frequent server occupation and thus, blocking probability in the system.

### (c) Choose one or two other distributions

We set the service distribution to be a hyperexponential distribution, constructed with two exponential components: one with a high rate  $\lambda_1=1/3$ , and a low rate  $\lambda_2=1/18$ , with a probability of selecting the first component of  $p=1/3$ , and for selecting the second component  $1-p=2/3$ :

```
inter_arrival_dist = expon(scale=1)
service_time_dist = hyperexpon(1/3, 1/18, 1/3)

simulate_and_print_stats(inter_arrival_dist, service_time_dist)
```



Estimated Blocking Probability: 0.1183  
 95% Confidence Interval: (0.1040, 0.1327)

This yields a system with a slightly lower blocking probability than the exponentially-distributed service times. However, the hyperexponential distributed service times results in greater variability in the system, where some customers are served faster while others are served slower. This result reinforces the idea that service time variability has a significant effect on system performance, even when the mean remains unchanged.

## Exercise 5

1. Estimate the integral  $\int_0^1 e^x dx$  by simulation (the crude Monte estimator). Use eg. an estimator based on 100 samples and present the result as the point estimator and a confidence interval.

In order to estimate the following integral:

$$\int_0^1 e^x dx$$

we use the crude Monte-Carlo estimator by simulation.

First, we let  $U_1, \dots, U_n \sim \text{Uniform}(0, 1)$ . Then, we can write:

$$\int_0^1 e^x dx = E[e^U]$$

This we can use to simulate  $n=100$  samples of  $U$  and thus, compute the point estimator  $\hat{\theta}$ :

$$\hat{\theta} = \frac{1}{n} \sum_{i=1}^n e^{U_i}$$

We perform 10 runs, of 100 samples each:

```
import numpy as np
import scipy.stats as stats
from scipy.stats import uniform, norm, expon
import matplotlib.pyplot as plt

samples = uniform.rvs(size=(10, 100))

xs = np.exp(samples)

all_samples = xs.flatten()

plt.hist(all_samples, bins=15, density=True, alpha=0.7, color='blue',
edgecolor='lightblue')
plt.title("Histogram of Crude MC samples")
plt.xlabel("Value")
plt.ylabel("Density")
plt.show()

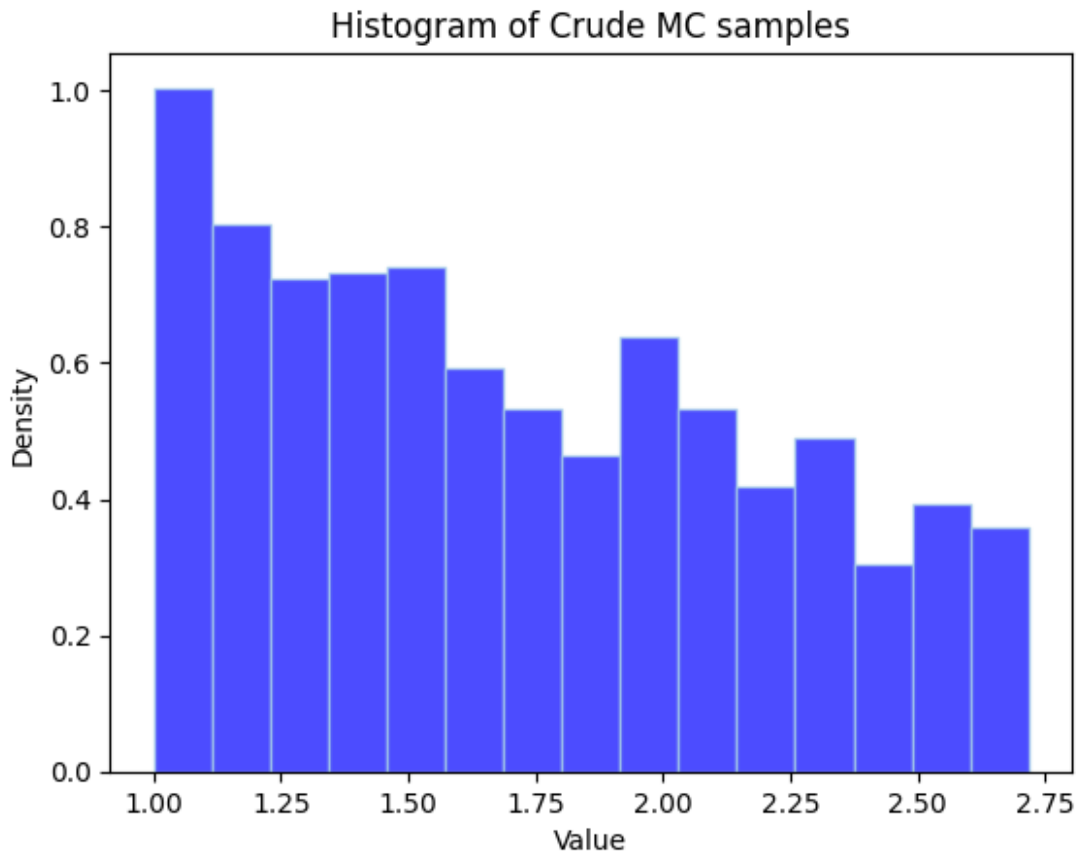
mean = np.mean(all_samples)
variance = np.var(all_samples, ddof=1)
```

```

standard_error = np.std(all_samples, ddof=1) /
np.sqrt(len(all_samples))
z = 1.96
ci_lower = mean - z * standard_error
ci_upper = mean + z * standard_error

print(f"Mean estimate of the integral: {mean:.4f}")
print(f"Sample variance: {variance:.6f}")
print(f"95% confidence interval: ({ci_lower:.4f}, {ci_upper:.4f})")

```



```

Mean estimate of the integral: 1.7120
Sample variance: 0.236983
95% confidence interval: (1.6818, 1.7422)

```

The histogram is right-skewed, meaning more values are located to the left in plot, also reflecting the nature of an exponential function.

Furthermore, the mean estimate is closely aligned to the integral's true value  $e - 1 \approx 1.718$ , while the true integral value is also within the confidence bounds. Thus, Monte Carlo estimator provides an appropriate approximation.

2. Estimate the integral  $\int_0^1 e^x dx$  using antithetic variables, with comparable computer resources.

To estimate

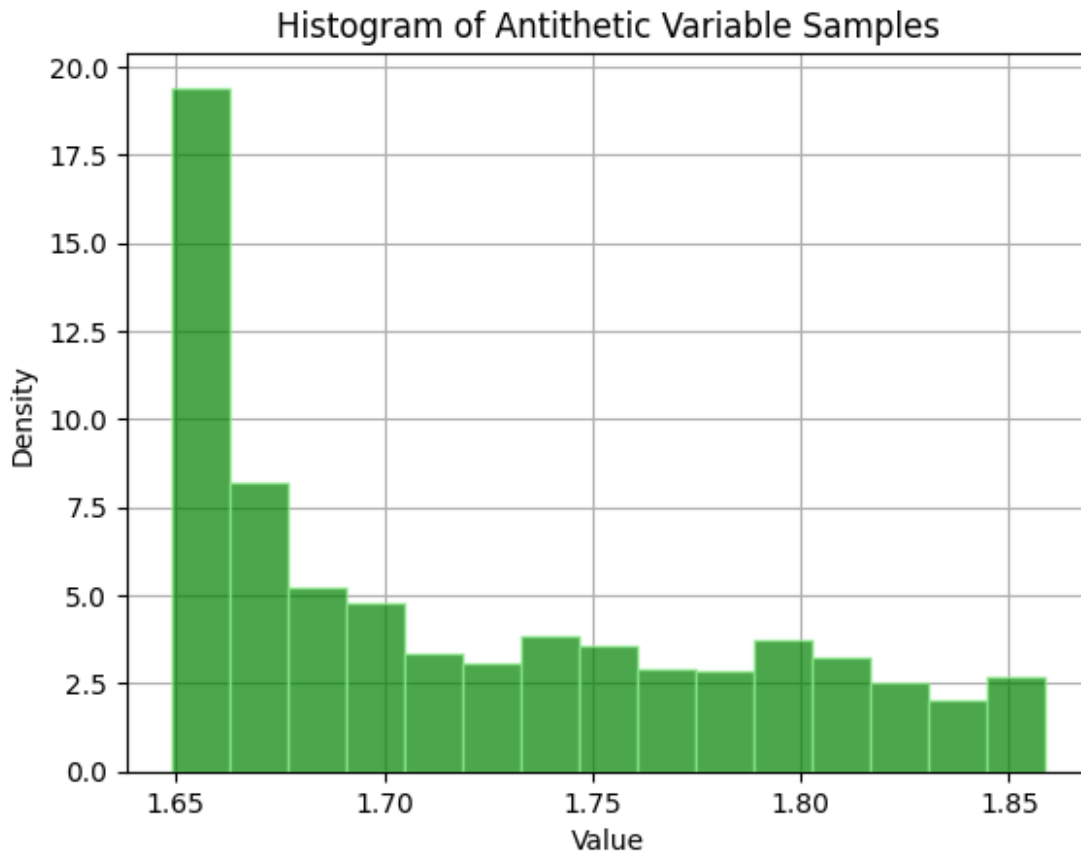
$$\int_1^0 e^x dx$$

using antithetic variables, we will utilize the variance reduction technique. Herein, we average each function  $e^U$  with its antithetic pair  $e^{1-U}$ , aiming to cancel out any variation. We perform 10 runs, of 100 samples each:

```
x_antithetic = (np.exp(samples) + np.exp(1 - samples)) / 2

all_samples = x_antithetic.flatten()
plt.figure()
plt.grid(True, zorder=0)
plt.hist(all_samples, bins=15, density=True, alpha=0.7,
         color='green', edgecolor='lightgreen', zorder=3)
plt.title("Histogram of Antithetic Variable Samples")
plt.xlabel("Value")
plt.ylabel("Density")
plt.show()

mean = np.mean(all_samples)
variance = np.var(all_samples, ddof=1)
standard_error = np.std(all_samples, ddof=1) /
np.sqrt(len(all_samples))
z = 1.96
ci_lower = mean - z * standard_error
ci_upper = mean + z * standard_error
print(f"Mean estimate of the integral: {mean:.4f}")
print(f"Sample variance: {variance:.6f}")
print(f"95% confidence interval: ({ci_lower:.4f}, {ci_upper:.4f})")
```



Mean estimate of the integral: 1.7188  
 Sample variance: 0.004073  
 95% confidence interval: (1.7148, 1.7227)

The distribution in the histogram are more concentrated, having a narrower spread and a higher peak, alluding to an exponential distribution more. It also has a lower variance than the previous method.

As for the mean of 1.7188, it approximates  $e - 1 \approx 1.718$  more than the previous method, while this method also has a narrower confidence interval.

3. Estimate the integral  $\int_0^1 e^x dx$  using a control variable, with comparable computer resources.

Estimating the integral using a control variate, we use  $U \sim \text{Uniform}(0, 1)$  as the control variate. We apply the corrected estimator:

$$\hat{\theta}_{cv} = \bar{X} + c(\bar{Y} - E[Y])$$

where  $c$  is chosen as to minimize the variance. As the slides say, we set  $c = -0.14086 \cdot 12$ :

```

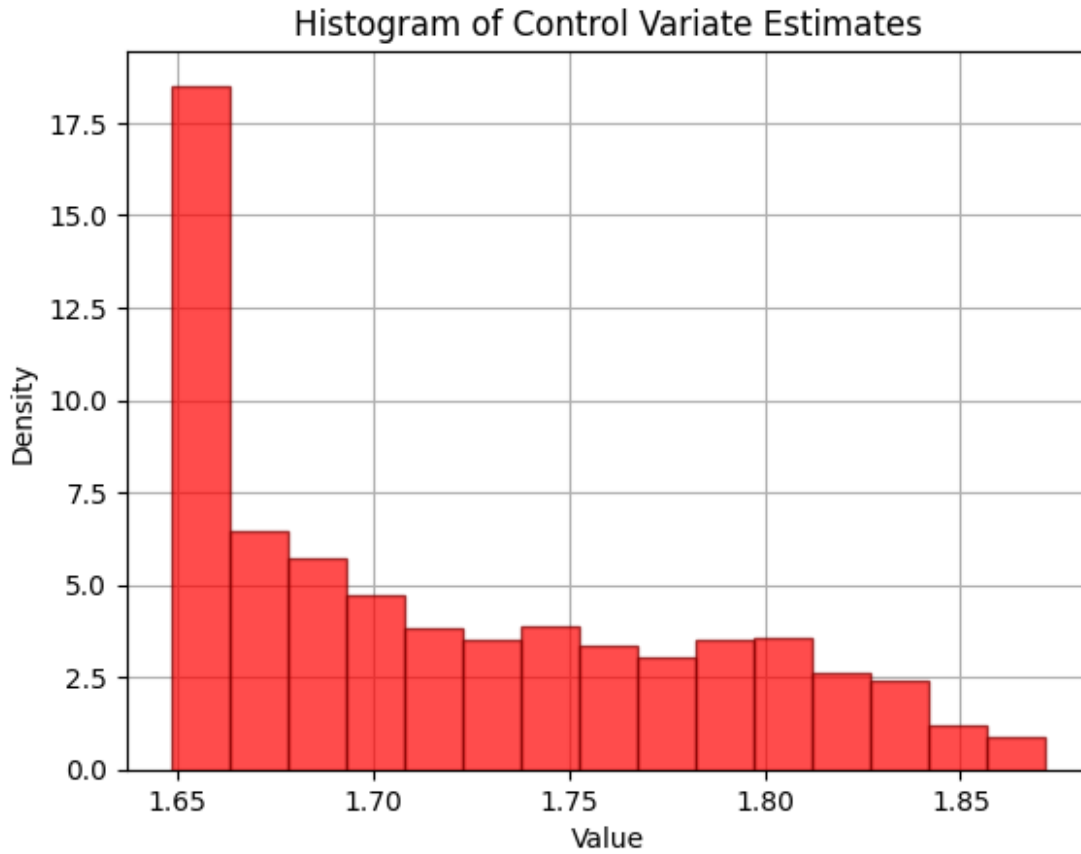
u = uniform.rvs(size=(10, 100))
x = np.exp(u)
c = -0.14086 * 12
corrected = x + c * (u - 0.5)
all_samples = corrected.flatten()

plt.figure()
plt.grid(True, zorder=0)
plt.hist(all_samples, bins=15, density=True, alpha=0.7,
         color='red', edgecolor='maroon', zorder=3)
plt.title("Histogram of Control Variate Estimates")
plt.xlabel("Value")
plt.ylabel("Density")
plt.show()

mean = np.mean(all_samples)
variance = np.var(all_samples, ddof=1)
standard_error = np.std(all_samples, ddof=1) /
np.sqrt(len(all_samples))
z = 1.96
ci_lower = mean - z * standard_error
ci_upper = mean + z * standard_error
print(f"Mean estimate of the integral: {mean:.4f}")
print(f"Sample variance: {variance:.6f}")
print(f"95% confidence interval: ({ci_lower:.4f}, {ci_upper:.4f})")

```





Mean estimate of the integral: 1.7189  
 Sample variance: 0.003939  
 95% confidence interval: (1.7150, 1.7228)

This is quite similar to the previous method and also has almost the same variance. Furthermore, the estimate and the confidence intervals are quite similar to the previous method, where the true integral value of  $e - 1 = 1.718$  is within the confidence interval, indicating that utilizing control variates is also a valid approximation of the integral

#### 4. Estimate the integral $\int_0^1 e^x dx$ using stratified sampling, with comparable computer resources

Using the stratified sampling method, we split the interval  $[0,1]$  into  $n=100$  equal-width subintervals (strata). Here, in each subinterval  $\left[\frac{i-1}{100}, \frac{i}{100}\right)$ , one sample is drawn, ensuring even coverage, and reducing variance.

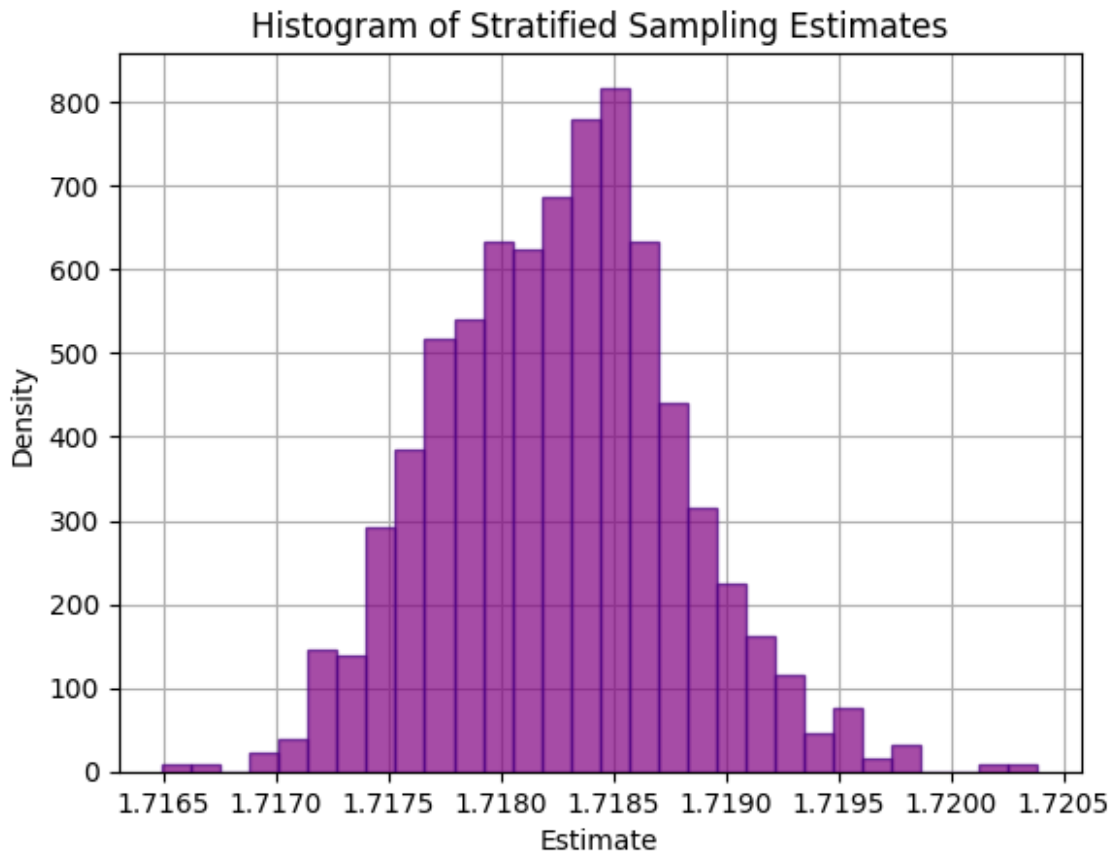
To compute the mean and the 95% confidence interval of the stratified sampling estimator, we repeat the full stratified sampling procedure a 1000 times. The resulting estimates are then used to calculate the sample mean and standard deviation, from which the confidence interval is constructed using the normal approximation.

```

n_strata = 100
reps = 1000
estimates = []
for _ in range(reps):
    strata_edges = np.linspace(0, 1, n_strata + 1)
    samples = np.array([
        np.random.uniform(low=strata_edges[i], high=strata_edges[i+1])
        for i in range(n_strata)
    ])
    values = np.exp(samples)
    estimates.append(np.mean(values))

estimates = np.array(estimates)
plt.figure()
plt.grid(True, zorder=0)
plt.hist(estimates, bins=30, density=True, alpha=0.7,
         color='purple', edgecolor='indigo', zorder=3)
plt.title("Histogram of Stratified Sampling Estimates")
plt.xlabel("Estimate")
plt.ylabel("Density")
plt.show()
mean = np.mean(estimates)
variance = np.var(estimates, ddof=1)
standard_error = np.std(estimates, ddof=1) / np.sqrt(len(estimates))
z = 1.96
ci_lower = mean - z * standard_error
ci_upper = mean + z * standard_error
print(f"Mean estimate of the integral: {mean:.4f}")
print(f"Sample variance: {variance:.6f}")
print(f"95% confidence interval: ({ci_lower:.4f}, {ci_upper:.4f})")

```



Mean estimate of the integral: 1.7183  
Sample variance: 0.000000  
95% confidence interval: (1.7182, 1.7183)

Surprisingly, using the stratified sampling method does not result in an exponential-shaped distribution, but instead produces a distribution that closely resembles a normal distribution. This occurs due to the Central Limit Theorem: even though each individual sample from  $\exp(U)$  follows a skewed distribution, the average of many independent estimates approximate a normal distribution.

This result in a mean estimate of the integral closest to  $e - 1 \approx 1.718$ , where the variance is miniscule. Furthermore, this method has also has the narrowest confidence interval.

## 5. Use control variates to reduce the variance of the estimator in exercise 4 (Poisson arrivals).

Taking inspiration from our code in Exercise 4, here, we applied a variance reduction technique by implementing control variates in the system. Specifically, we utilized the mean interarrival time as a control variate, allowing us to adjust the original estimator by subtracting the noise, reducing the estimators variance without introducing extra bias.

We do  $n=100$  iterations in the run:

```

import numpy as np
import heapq
from scipy.stats import expon, uniform
from enum import StrEnum
import matplotlib.pyplot as plt

np.random.seed(1234)

class Event(StrEnum, Enum):
    ARRIVED = "Arrived"
    SERVICED = "Serviced"
    BLOCKED = "Blocked"

def simulate_system_with_arrivals(arrival_times, service_time_dist,
m=10):
    service_units = list(range(m))
    events = [(time, Event.ARRIVED) for time in arrival_times]
    heapq.heapify(events)

    n_blocked = 0

    while events:
        event = heapq.heappop(events)
        time, event_type = event[0], event[1]

        if event_type == Event.ARRIVED:
            if service_units:
                finish_time = time + service_time_dist.rvs()
                heapq.heappush(events, (finish_time, Event.SERVICED,
service_units.pop()))
            else:
                heapq.heappush(events, (time, Event.BLOCKED))

        elif event_type == Event.SERVICED:
            service_units.append(event[2])
        elif event_type == Event.BLOCKED:
            n_blocked += 1

    return n_blocked

m = 10
n_customers = 10000
iterations = 100
service_time_dist = expon(scale=8)
lambda_poisson = 1
expected_interarrival = 1 / lambda_poisson
block_rates = np.empty(iterations)
mean_arrivals = np.empty(iterations)

for i in range(iterations):
    inter_arrivals =

```

```

expon(scale=expected_interarrival).rvs(size=n_customers)
    arrival_times = np.cumsum(inter_arrivals)
    mean_arrivals[i] = np.mean(inter_arrivals)

    n_blocked = simulate_system_with_arrivals(arrival_times,
service_time_dist, m)
    block_rates[i] = n_blocked / n_customers

c_opt = np.cov(block_rates, mean_arrivals)[0,1] /
np.var(mean_arrivals)
block_rates_cv = block_rates - c_opt * (mean_arrivals -
expected_interarrival)

mean_normal = np.mean(block_rates)
std_normal = np.std(block_rates)
ci_low_normal = mean_normal - 1.96 * std_normal
ci_high_normal = mean_normal + 1.96 * std_normal

mean_cv = np.mean(block_rates_cv)
std_cv = np.std(block_rates_cv)
ci_low_cv = mean_cv - 1.96 * std_cv
ci_high_cv = mean_cv + 1.96 * std_cv

print(f"Without control variates:")
print(f"Mean block rate: {mean_normal:.5f}, Std: {std_normal:.5f}")
print(f"95% CI: ({ci_low_normal:.5f}, {ci_high_normal:.5f})\n")

print(f"With control variates:")
print(f"Mean block rate: {mean_cv:.5f}, Std: {std_cv:.5f}")
print(f"95% CI: ({ci_low_cv:.5f}, {ci_high_cv:.5f})")
plt.figure(figsize=(10,6))
plt.bar(['Without CV', 'With CV'], [std_normal, std_cv],
color=['gray', 'green'])
plt.ylabel("Standard deviation of estimator")
plt.title("Variance Reduction via Control Variates")
plt.grid(True)
plt.show()

```

Without control variates:  
Mean block rate: 0.12102, Std: 0.00673  
95% CI: (0.10783, 0.13422)

With control variates:  
Mean block rate: 0.12132, Std: 0.00536  
95% CI: (0.11081, 0.13182)



Here, we see how implementing control variates to the simulation greatly benefits it, reducing variance and improving efficiency. In particular, introducing control variates also ensures a more narrow confidence interval, ensuring greater stability and certainty.

6. Demonstrate the effect of using common random numbers in exercise 4 for the difference between Poisson arrivals (Part 1) and a renewal process with hyperexponential interarrival times. Remark: You might need to do some thinking and some re-programming.

Now, we will return to exercise 4.1 and take some of the code as for inspiration.

We have set up the same system, but we compare the blocking abilities two systems: one with Poisson arrivals and one with hyperexponential interarrival times. Now, we will reimplement them but with Common Random Numbers (CRNs), hopefully reducing the variance of the difference between the systems.

As for the hyperexponential distribution we set  $\lambda_1=1/3$ ,  $\lambda_2=1/18$ , and  $p=1/3$ , like we did in Exercise 4.

```
class Event(StrEnum):
    ARRIVED = "Arrived"
    SERVICED = "Serviced"
    BLOCKED = "Blocked"

def generate_hyperexp_samples(u, p=0.5, lambda1=2.0, lambda2=0.5):
    selector = u < p
```

```

    samples = np.empty_like(u)
    samples[selector] = expon(scale=1/lambda1).ppf(u[selector] / p)
    samples[~selector] = expon(scale=1/lambda2).ppf((u[~selector] - p)
/ (1 - p))
    return samples

def simulate_system_from_arrivals(arrival_times, service_time_dist,
m=10):
    service_units = [i for i in range(m)]
    events = [(time, Event.ARRIVED) for time in arrival_times]
    heapq.heapify(events)

    n_served = 0
    n_blocked = 0

    while events:
        event = heapq.heappop(events)
        match event:
            case (time, Event.ARRIVED):
                if service_units:
                    finish_time = time + service_time_dist.rvs()
                    heapq.heappush(events, (finish_time,
Event.SERVED, service_units.pop()))
                else:
                    heapq.heappush(events, (time, Event.BLOCKED))
            case (_, Event.SERVED, unit):
                service_units.append(unit)
                n_served += 1
            case (_, Event.BLOCKED):
                n_blocked += 1

    return n_served, n_blocked

n_customers = 10000
m = 10
reps = 100
service_time_dist = expon(scale=8)
p = 1/3
lambda1 = 1/3
lambda2 = 1/18

differences = []

for _ in range(reps):
    uniforms = uniform.rvs(size=n_customers)

    poisson_interarrivals = expon(scale=1).ppf(uniforms)
    hyperexp_interarrivals = generate_hyperexp_samples(uniforms, p=p,
lambda1=lambda1, lambda2=lambda2)

```

```

poisson_arrivals = np.cumsum(poisson_interarrivals)
hyperexp_arrivals = np.cumsum(hyperexp_interarrivals)

n_served_pois, n_blocked_pois =
simulate_system_from_arrivals(poisson_arrivals, service_time_dist, m)
n_served_hyper, n_blocked_hyper =
simulate_system_from_arrivals(hyperexp_arrivals, service_time_dist, m)

block_rate_pois = n_blocked_pois / n_customers
block_rate_hyper = n_blocked_hyper / n_customers

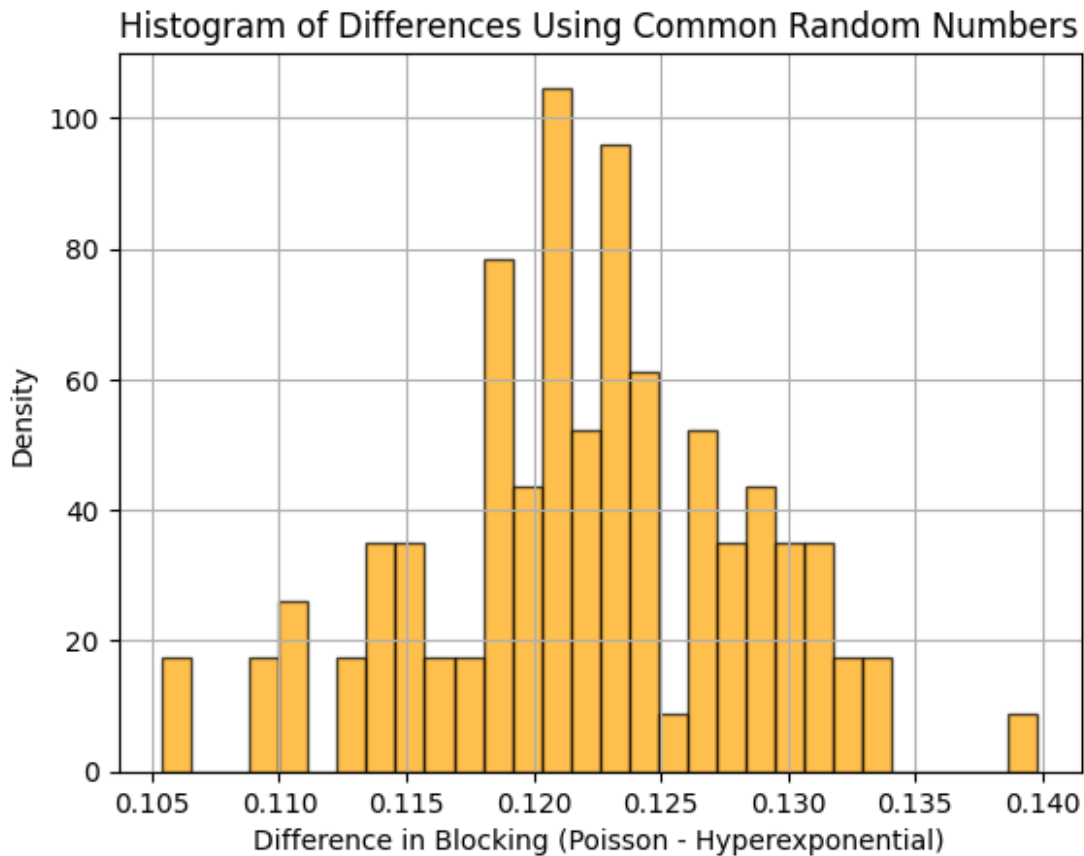
diff = block_rate_pois - block_rate_hyper
differences.append(diff)

differences = np.array(differences)
mean_diff = np.mean(differences)
var_diff = np.var(differences, ddof=1)
std_err = np.std(differences, ddof=1) / np.sqrt(len(differences))
z = 1.96
ci_low = mean_diff - z * std_err
ci_high = mean_diff + z * std_err

plt.hist(differences, bins=30, density=True, alpha=0.7,
color='orange', edgecolor='black')
plt.title("Histogram of Differences Using Common Random Numbers")
plt.xlabel("Difference in Blocking (Poisson - Hyperexponential)")
plt.ylabel("Density")
plt.grid(True)
plt.show()
print(f"Mean difference: {mean_diff:.6f}")
print(f"Sample variance of difference: {var_diff:.8f}")
print(f"95% confidence interval for difference: ({ci_low:.4f},
{ci_high:.4f})")

```





Mean difference: 0.122009  
 Sample variance of difference: 0.00004085  
 95% confidence interval for difference: (0.1208, 0.1233)

This shows on average, the Poisson system block 12.2% more customers than the hyperexponential system does. Furthermore, the variance is quite low, and the confidence interval is narrow, suggesting implementing CRNs greatly reduced simulation noise and reduced the overall variability of the system. The histogram also supports this, being bell-shaped like a normal distribution and concentrated.

7. For a standard normal random variable  $Z \sim N(0, 1)$  using the crude Monte Carlo estimator estimate the probability  $Z > a$ . Then try importance sampling with a normal density with mean  $a$  and variance  $\sigma^2$ . For the experiments start using  $\sigma^2 = 1$ , use different values of  $a$  (e.g. 2 and 4), and different sample sizes. If time permits experiment with other values for  $\sigma^2$ . Finally discuss the efficiency of the methods.

To estimate  $P(Z > a)$  for  $Z \sim N(0, 1)$  using the crude Monte-Carlo and importance sampling methods. For importance sampling, we use a  $N(0, 1)$  and apply a likelihood ratio weighting. The

experiments are done for the values  $a \in \{2, 4\}$ ,  $n \in \{1000, 10000\}$ , and  $\sigma^2=1$ , to compare the accuracies and variances:

```
for a in [2, 4]:
    print(f"a = {a}")
    print(f"Analytical solution: {1 - norm.cdf(a):.6f}")
    for n in [10, 100, 1000, 10_000]:
        print(f"N = {n}")
        zs = norm.rvs(size=n)

        crude = np.mean(zs > a)

        ys = norm.rvs(size=n, loc=a, scale=1)

        imp_sampl = np.mean((ys > a) * norm.pdf(ys) / norm.pdf(ys,
loc=a, scale=1))

        print(f"Crude estimate:      {crude:.6f}")
        print(f"Importance sampling: {imp_sampl:.6f}")

a = 2
Analytical solution: 0.022750
N = 10
Crude estimate:      0.100000
Importance sampling: 0.009619
N = 100
Crude estimate:      0.010000
Importance sampling: 0.022385
N = 1000
Crude estimate:      0.027000
Importance sampling: 0.021453
N = 10000
Crude estimate:      0.022800
Importance sampling: 0.022880
a = 4
Analytical solution: 0.000032
N = 10
Crude estimate:      0.000000
Importance sampling: 0.000028
N = 100
Crude estimate:      0.000000
Importance sampling: 0.000028
N = 1000
Crude estimate:      0.000000
Importance sampling: 0.000032
N = 10000
Crude estimate:      0.000000
Importance sampling: 0.000032
```

From this, we can conclude the following:

- For  $a=2$ , the analytical solution is approximately 0.02275. Both the Crude Monte Carlo and the Importance Sampling estimators are inaccurate with small sample sizes,  $N=10$ . However, by  $N=100$  the Importance Sampling is already close to the true value, while the Crude estimator remains off. As  $N$  increases further, both methods begin to converge toward the analytical solution, with Importance Sampling consistently performing more accurately and reliably.
- For  $a=4$ , the difference in performance is stark between the two. With the analytical solution being rather small, the Importance Sampling method begins to approximate decently-well it already by  $N=1000$ , whereas the Crude Monte Carlo never approximates it (for  $N=10000$ ).

In conclusion, Crude Monte Carlo requires a very large number of samples  $N$  to resolve rare events, meaning it's inefficient. Importance Sampling, by concentrating samples in the critical region (around  $a$ ), can achieve accurate estimates with far fewer samples  $N$  demonstrating superior variance reduction and computational efficiency.

8. Use importance sampling with  $g(x) = \lambda \exp(-\lambda * x)$  to calculate the integral  $\int_0^1 e^x dx$  of Question 1. Try to find the optimal value of  $\lambda$  by calculating the variance of  $h(X)f(X)/g(X)$  and verify by simulation. Note that importance sampling with the exponential distribution will not reduce the variance.

To estimate the integral of Question 1, we use importance sampling, by sampling from the following exponential distribution truncated to  $[0,1]$ , and varying the rate  $\lambda$ . We rewrite the integral as an expectation with respect to a distribution  $g(x)$ , and then approximate the expectation using Monte Carlo sampling:

$$\int_0^1 e^x dx = E_g \left[ \frac{e^x \cdot f(x)}{g(x)} \right]$$

Here,  $f(x)$  is the density of the uniform distribution, and  $g(x)$  is the exponential PDF.

Now, we can try a range of different values for  $\lambda$  and compute the sample and analytical variances. Thus, we should find an optimal  $\lambda$ , which minimizes the overall variance.

We simulate  $n=10000$  runs:

```
n = 100
vars = np.empty(n)
means = np.empty(n)
lambdas = np.linspace(0.5, 3.0, n)

def analytical_variance(xs):
    squared_variance = (np.exp(1) - 1) ** 2
    variance_of_square = 1 / xs / (2 + xs) * (np.exp(2 + xs) - 1)
    return variance_of_square - squared_variance
```

```

for i, gamma in enumerate(lambdas):
    g = expon(scale=1/gamma)
    gs = g.rvs(size=100_000)

    samples = np.exp(gs) * uniform.pdf(gs) / g.pdf(gs)
    samples = samples[gs <= 1]

    means[i] = np.mean(samples)
    vars[i] = np.var(samples)

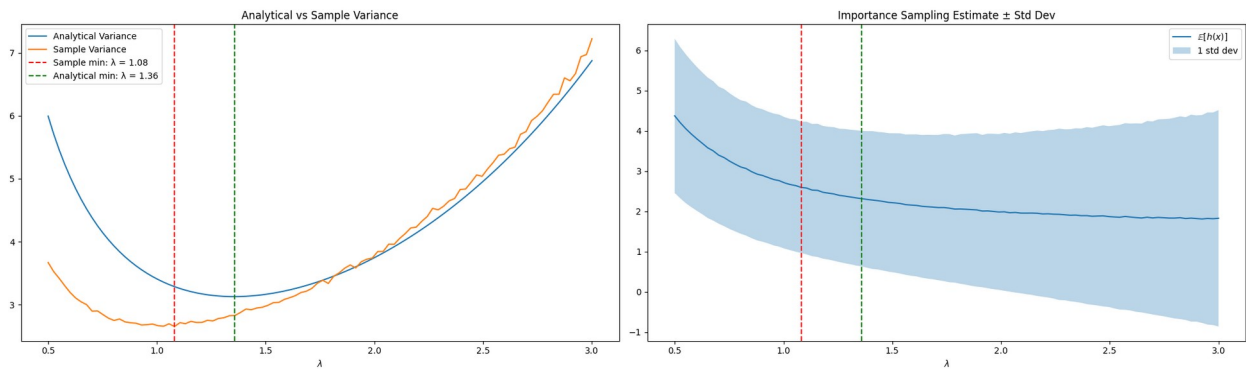
min_idx = np.argmin(vars)
analyt_vars = analytical_variance(lambdas)
analyt_min_idx = np.argmin(analyt_vars)
stds = np.sqrt(vars)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 6))

ax1.set_title("Analytical vs Sample Variance")
ax1.plot(lambdas, analyt_vars, label="Analytical Variance")
ax1.plot(lambdas, vars, label="Sample Variance")
ax1.set_xlabel(r"$\lambda$")
ax1.axvline(lambdas[min_idx], linestyle="--", color="r",
            label=f"Sample min:  $\lambda = \{lambdas[min\_idx]:.2f\}")$ ")
ax1.axvline(lambdas[analyt_min_idx], linestyle="--", color="g",
            label=f"Analytical min:  $\lambda = \{lambdas[analyt\_min\_idx]:.2f\}")$ ")
ax1.legend()

ax2.plot(lambdas, means, label=r"$\mathbb{E}[h(x)]$")
ax2.fill_between(lambdas, means - stds, means + stds, alpha=0.3,
                label="1 std dev")
ax2.axvline(lambdas[min_idx], linestyle="--", color="r")
ax2.axvline(lambdas[analyt_min_idx], linestyle="--", color="g")
ax2.set_title("Importance Sampling Estimate  $\pm$  Std Dev")
ax2.set_xlabel(r"$\lambda$")
ax2.legend()
plt.tight_layout()
plt.show()
print(f"Empirical minimum variance at  $\lambda = \{lambdas[min\_idx]:.4f\}")$ ")
print(f"Analytical minimum variance at  $\lambda = \{lambdas[analyt\_min\_idx]:.4f\}")$ ")

```



Empirical minimum variance at  $\lambda = 1.0808$   
 Analytical minimum variance at  $\lambda = 1.3586$

The result shows that importance sampling significantly reduces the variance of the estimator for the integral, and that the optimal choice of  $\lambda$  (which is around 1.36 analytically) minimizes this variance. The sample minimum  $\lambda$  is somewhat close to analytical  $\lambda$ , confirming the effectiveness of tuning  $\lambda$  and changing its values to optimize for efficiency.

## Exercise 6: Markov Chain Monte Carlo

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import chisquare
from math import factorial
from collections import Counter
```

1. The number of busy lines in a trunk group (Erlang system) is given by a truncated Poisson distribution

$$P(i) = c \cdot \frac{A^i}{i!}, i=0, \dots, m$$

Generate values from this distribution by applying the Metropolis-Hastings algorithm, verify with a  $\chi^2$ -test. You can use the parameter values from exercise 4.

Here, we generate samples from the truncated Poisson distribution  $P(i)$ . In it, we observe that  $A$  is the intensity of the traffic,  $m$  is the number of available lines in the system,  $c$  is a normalizing constant that ensures probabilities sum to 1, and  $m$  is the number of servers.

Looking back at exercise 4, we set  $m=10$  servers, and  $A=8$  Erlangs.

We implement the Metropolis-Hastings algorithm with a proposal distribution that chooses uniformly between +1 and -1 (relative to the current position). To reduce autocorrelation, we sample 20,000 points with stride 10 (only keeping every 10th point). Afterwards, a  $\chi^2$  test is done to compare the sampled distribution with the theoretical one:

```
np.random.seed(42)

A = 8
m = 10

# target distribution
def g(i):
    if 0 <= i <= m:
        return A**i / factorial(i)
    return 0

Z = sum(g(i) for i in range(m + 1))
true_probs = np.array([g(i)/Z for i in range(m + 1)])

def metropolis_hastings_1():
    num_samples = 20_000
```

```

samples = []
current = np.random.randint(0, m + 1)

for _ in range(num_samples):
    proposal = current + np.random.choice([-1, 1])

    if 0 <= proposal <= m:
        acceptance_ratio = g(proposal) / g(current)
        if np.random.rand() < min(1, acceptance_ratio):
            current = proposal
        samples.append(current)

samples = samples[:10]
num_samples = num_samples // 10

counts, bins = np.histogram(samples, bins=np.arange(-0.5, m+1.5,
1))
estimated_probs = counts / num_samples

chi2_stat, p_value = chisquare(counts, f_exp=true_probs *
num_samples)

return counts, estimated_probs, chi2_stat, p_value

counts, estimated_probs, chi2_stat, p_value = metropolis_hastings_1()

vals = [metropolis_hastings_1() for _ in range(100)]

fig, ax = plt.subplots(1, 3, figsize=(20, 6))

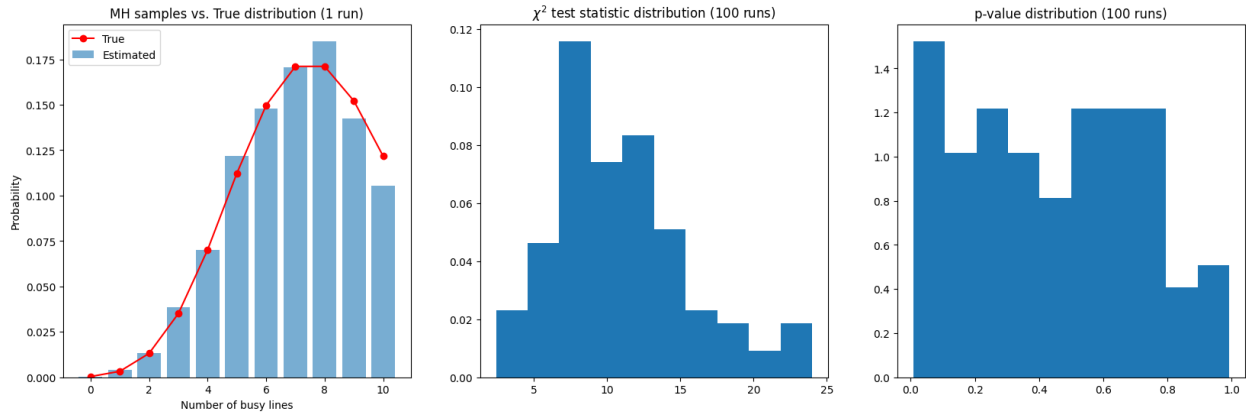
ax[0].bar(range(m+1), estimated_probs, alpha=0.6, label="Estimated")
ax[0].plot(range(m+1), true_probs, 'ro-', label="True")
ax[0].set_xlabel("Number of busy lines")
ax[0].set_ylabel("Probability")
ax[0].set_title("MH samples vs. True distribution (1 run)")
ax[0].legend()

ax[1].set_title(r"$\chi^2$ test statistic distribution (100 runs)")
ax[1].hist([val[2] for val in vals], density=True)

ax[2].set_title("p-value distribution (100 runs)")
ax[2].hist([val[3] for val in vals], density=True)

plt.show()

```



We see that the p-values are somewhat uniform, implying that the MH sampler approximates the target distribution. We have not however tested the uniformity explicitly, and even visually there seems to be some deviation from uniformity.

In our earlier experiments without the stride-tactic to reduce autocorrelation, nearly all p-values were less than 0.05.

2. For two different call types the joint number of occupied lines is given by  $P(i, j) = c \cdot \frac{A_1^i}{i!} \frac{A_2^j}{j!}, 0 \leq i + j \leq m$ .

You can use  $A_1, A_2 = 4$  and  $m = 10$ .

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

In all three cases test the distribution fit with a  $\chi^2$  test  
The system can be extended to an arbitrary dimension, and we can add restrictions on the different call types.

In this exercise we will need to plot many heatmaps of distributions. To reduce clutter, we define a function for this upfront:

```
def plot_solution(samples, title):
    counts = Counter(samples[0])
    heatmap = np.zeros((m+1, m+1))

    for (i, j), count in counts.items():
        heatmap[i, j] = count

    for i in range(m+1):
        for j in range(m+1):
```



```

        if i + j > m:
            heatmap[i, j] = np.nan

fig, ax = plt.subplots(1, 3, figsize=(20, 6))

ax[0].imshow(heatmap, origin='lower', cmap='viridis',
interpolation='none')
#ax[0].colorbar(label='Frequency')
ax[0].set_xlabel("Number of type 2 calls (j)")
ax[0].set_ylabel("Number of type 1 calls (i)")
ax[0].set_title(title)

p_vals = []
chi_vals = []

for sample in samples:
    counts = Counter(sample)
    heatmap = np.zeros((m+1, m+1))

    for (i, j), count in counts.items():
        heatmap[i, j] = count

    for i in range(m+1):
        for j in range(m+1):
            if i + j > m:
                heatmap[i, j] = np.nan

    unnormalized = np.zeros((m+1, m+1))
    for i in range(m+1):
        for j in range(m+1):
            if i + j <= m:
                unnormalized[i, j] = g(i, j)

    Z = np.nansum(unnormalized)
    expected_probs = unnormalized / Z
    expected_counts = expected_probs * len(sample)

    observed = []
    expected = []

    for i in range(m+1):
        for j in range(m+1):
            if i + j <= m:
                observed.append(heatmap[i, j])
                expected.append(expected_counts[i, j])

    chi2_stat, p_val = chisquare(f_obs=observed, f_exp=expected)

    p_vals.append(p_val)
    chi_vals.append(chi2_stat)

```

```

ax[1].set_title(r"$\chi^2$ test statistic distribution (100
runs)")
ax[1].hist(chi_vals, density=True)

ax[2].set_title("p-value distribution (100 runs)")
ax[2].hist(p_vals, density=True)

plt.show()

```

Now, we extend the blocking model to account two call types. The joint number of occupied lines is given by  $P(i, j)$ , where  $A_1, A_2 = 4$  and  $m = 10$ , for  $0 \leq i + j \leq m$ .

We use the Metropolis-Hastings algorithm to approximate the target distribution, using a proposal distribution that sets  $x = x + d_x$  and  $y = y + d_y$  where  $d_x$  and  $d_y$  are sampled uniformly and independently from  $\{-1, 0, 1\}$ :

```

np.random.seed(42)

A1, A2 = 4, 4
m = 10

def g(i, j):
    if 0 <= i <= m and 0 <= j <= m and i + j <= m:
        return (A1**i / factorial(i)) * (A2**j / factorial(j))
    return 0

def metropolis_hastings_2():
    num_samples = 50_000
    samples = []
    current = (0, 0)

    shifts = [[-1, -1], [-1, 0], [-1, 1], [0, -1], [0, 1], [1, -1],
[1, 0], [1, 1]]

    for _ in range(num_samples):
        i, j = current
        shift_idx = np.random.choice(len(shifts))
        di, dj = shifts[shift_idx]
        proposal = (i + di, j + dj)
        if 0 <= proposal[0] <= m and 0 <= proposal[1] <= m and
sum(proposal) <= m:
            g_current = g(i, j)
            g_proposed = g(*proposal)
            alpha = min(1, g_proposed / g_current) if g_current > 0
        else 1
            if np.random.rand() < alpha:
                current = proposal
            samples.append(current)

```

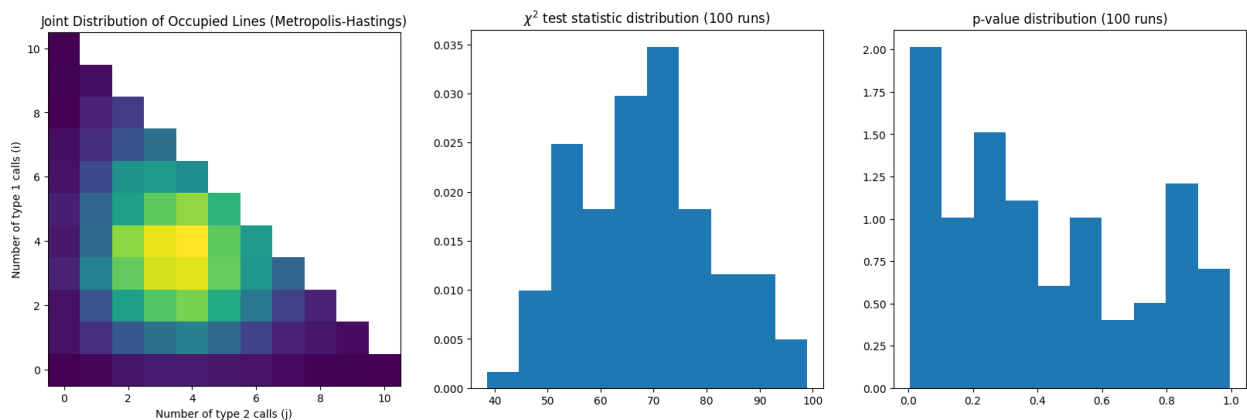
```

    samples = samples[::10]

    return samples

samples = [metropolis_hastings_2() for _ in range(100)]
plot_solution(samples, "Joint Distribution of Occupied Lines
(Metropolis-Hastings)")

```



This time the p-values mostly look uniform, but there is a noticeable peak near 0. This suggests that the algorithm fails to approximate the distribution.

(b) Use Metropolis-Hastings, coordinate wise to generate variates from this distribution. (b) Use Metropolis-Hastings, coordinate wise to generate variates from this distribution.

Here, we implement a coordinate-wise version Metropolis-Hastings algorithm, sampling again from  $P(i, j)$ . This in order to generate variates from the distribution:

```

np.random.seed(42)

def metropolis_hastings_3():
    num_samples = 25_000
    samples = []
    current = [0, 0]
    shifts = [-1, 1]
    for _ in range(num_samples):
        for d in [0, 1]:
            proposal = current[:]
            shift_idx = np.random.choice(2)
            proposal[d] += shifts[shift_idx]
            if 0 <= proposal[0] <= m and 0 <= proposal[1] <= m and
sum(proposal) <= m:
                g_current = g(*current)
                g_proposal = g(*proposal)

```

```

        alpha = min(1, g_proposal / g_current) if g_current >
0 else 1
        if np.random.rand() < alpha:
            current = proposal
            samples.append(tuple(current))

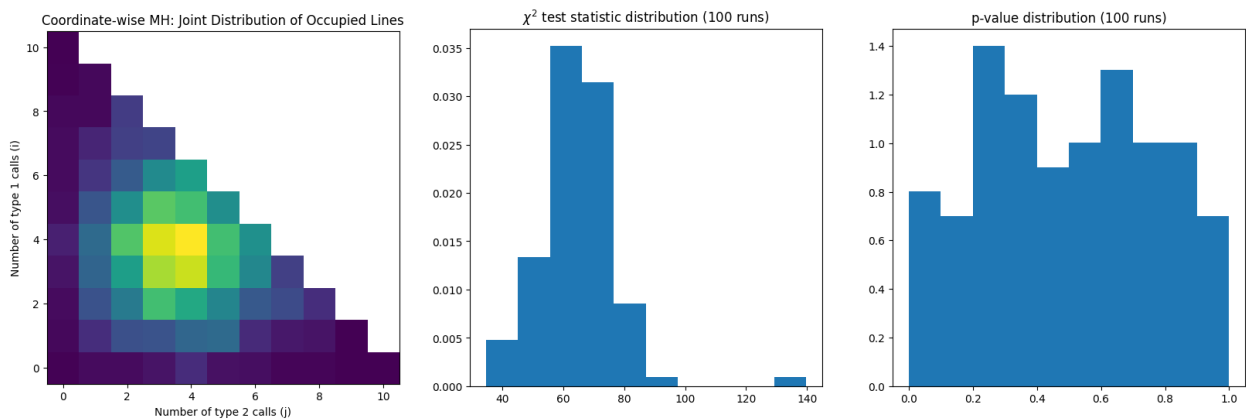
    samples = samples[::10]

    return samples

samples = [metropolis_hastings_3() for _ in range(100)]

plot_solution(samples, "Coordinate-wise MH: Joint Distribution of
Occupied Lines")

```



The p-values are somewhat uniform when judged visually, better than in the non-coordinate-wise version of MH.

(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.

Here, we use Gibbs sampling to generate samples from  $P(i, j)$ . We sample each variable using its exact conditional distribution.

$$P(i) = \sum_j P(i, j) = \sum_j \frac{A_1^i}{i!} \frac{A_2^j}{j!} c = \frac{A_1^i}{i!} \sum_j \frac{A_2^j}{j!} c = \frac{A_1^i}{i!} \tilde{c}$$

We do not know the constant  $\tilde{c}$  analytically, but can calculate it cheaply.

```

def gibbs_sampling():
    num_samples = 20_000
    samples = []
    i, j = 0, 0

```

```

    for _ in range(num_samples):
        i_max = m - j + 1
        p_vals = np.array([A1**i_candidate / factorial(i_candidate)
for i_candidate in range(i_max)])
        p_vals /= np.sum(p_vals)
        i = np.random.choice(i_max, p=p_vals)

        j_max = m - i + 1
        p_vals = np.array([A2**j_candidate / factorial(j_candidate)
for j_candidate in range(j_max)])
        p_vals /= np.sum(p_vals)
        j = np.random.choice(j_max, p=p_vals)

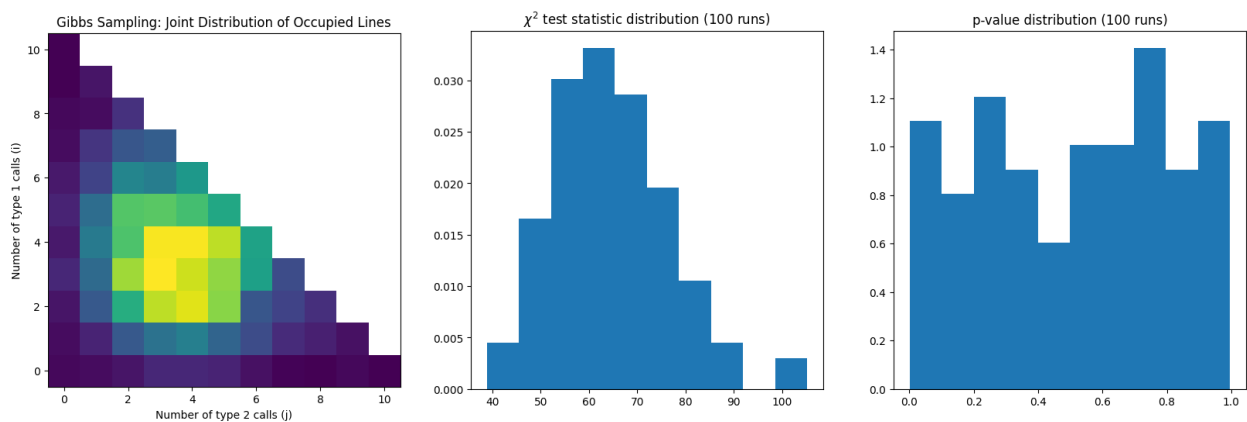
        samples.append((i, j))

    samples = samples[::10]
    return samples

samples = [gibbs_sampling() for _ in range(100)]

plot_solution(samples, "Gibbs Sampling: Joint Distribution of Occupied
Lines")

```



The p-values are visually uniformly distributed, indicating that the Gibbs sampling successfully approximates the target distribution.

Gibbs sampling seems to be the best solution for this specific distribution, but it of course requires that we can calculate the conditional distributions.

3. We consider a Bayesian statistical problem. The observations are  $X_i \sim N(\Theta, \Psi)$ , where the prior distribution of the pair  $(\Xi, \Gamma) = (\log(\Theta), \log(\Psi))$  is standard normal with correlation  $\rho = \frac{1}{2}$ . The joint density  $f(x, y)$  of  $(\Theta, \Psi)$  is

$$f(x, y) = \frac{1}{2\pi xy \sqrt{1 - \rho^2}} e^{-\frac{\log(x)^2 - 2\rho \log(x) \log(y) + \log(y)^2}{2(1 - \rho^2)}}$$

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  $(\Theta, \Psi)$  using Markov Chain Monte Carlo.

- (a) Generate a pair  $(\theta, \psi)$  from the prior distribution, i.e. the distribution for the pair  $(\Theta, \Psi)$ , by first generating a sample  $(\xi, \gamma)$  of  $(\Xi, \Gamma)$ .

Here, we generate a pair  $(\theta, \psi)$  from the prior distribution, where  $\log(\Theta) = \Xi$  and  $\log(\Psi) = \Gamma$ . The pair  $(\Xi, \Gamma)$  follows a bivariate normal distribution, where the correlation is set as  $\rho = 0.5$ .

In order to generate a pair, first, we generate a bivariate normal sample  $(\xi, \gamma)$ , which we can then transform back to  $(\theta, \psi)$  using

$$\theta = e^\xi, \xi = e^\theta$$

This is as described in Day 5 slides known as the change of variable.

```
np.random.seed(1)
rho = 0.5
cov = [[1, rho], [rho, 1]]

xi_gamma = np.random.multivariate_normal(mean=[0, 0], cov=cov)
xi, gamma = xi_gamma
theta_true = np.exp(xi)
psi_true = np.exp(gamma)

print(f"Sampled  $\xi, \gamma$  ( $\log(\theta), \log(\psi)$ ): ({xi:.4f}, {gamma:.4f})")
print(f"Transformed  $\theta, \psi$ : ({theta_true:.4f}, {psi_true:.4f})")

Sampled  $\xi, \gamma$  ( $\log(\theta), \log(\psi)$ ): (-1.1008, -1.7126)
Transformed  $\theta, \psi$ : (0.3326, 0.1804)
```

The generated values  $(\theta, \psi) = (0.3326, 0.1804)$  are realizations from the prior distribution. Both values are positive, meaning they are consistent with the assumed prior.

(b) Generate  $X_i = 1, \dots, n$  with the values of  $(\theta, \psi)$  you obtained in item 3a. Use  $n = 10$ .

Using the generated values, we simulate a sample of  $n = 10$  size from a normal distribution, where the mean is  $\theta$  and the variance is  $\psi$ :

$$X_i \sim N(\theta, \psi)$$

```
n = 10

np.random.seed(42)
X = np.random.normal(loc=theta_true, scale=np.sqrt(psi_true), size=n)

print("Generated X values:")
print(np.round(X, 4))

Generated X values:
[0.5436 0.2739 0.6077 0.9795 0.2331 0.2331 1.0033 0.6585 0.1332
 0.563 ]
```

These 10 generated values represent the synthetic observations drawn under the prior parameter.

(c) Derive the posterior distribution of  $(\Theta, \Psi)$  given the sample. Hint 1: Apply Bayes theorem in the density version. Hint 2: The sample mean and sample variance are independent. The sample mean follows a normal distribution, while a scaled version of the sample variance follows a  $\chi^2$  distribution. This can be used to simplify the expression.

In order to derive the posterior distribution, we utilize Bayes' Theorem:

$$\text{Posterior} \propto \text{Prior} \times \text{Likelihood}$$

Specifically for this case, it will be:

$$\text{Posterior}(\theta, \psi \mid X) \propto \text{Prior}(\theta, \psi) \times \text{Likelihood}(X \mid \theta, \psi)$$

From the slides, we observe:

- $\hat{X} \sim N(\Theta, \Psi/n)$
- $\frac{(n-1)S^2}{\Psi} \sim \chi^2_{n-1}$

Using this information, gives us the factored likelihood:

$$L(\theta, \psi) \propto \psi^{-\frac{n}{2}} \exp\left(-\frac{n(\bar{x} - \theta)^2}{2\psi} - \frac{(n-1)S^2}{2\psi}\right)$$

The posterior is then the product of the likelihood above, and the prior.

This allows us to generate MCMC samples.

(d) Generate MCMC samples from the posterior distribution of  $(\Theta, \Psi)$  using the Metropolis Hastings method.

In order to generate MCMC samples, we use the Metropolis-Hastings algorithm, sampling from the posterior distribution  $(\Theta, \Psi)$ .

We define the posterior using the log-likelihood of  $\bar{X}$  and  $S^2$ , and the log-prior of the bivariate normal density. We use the Metropolis-Hastings with a proposal distribution of  $N(0, 0.2^2)$ :

```
np.random.seed(42)

x_bar = np.mean(X)
s2 = np.var(X, ddof=1)

def log_posterior(xi, gamma):
    theta = np.exp(xi)
    psi = np.exp(gamma)
    log_like = -n / 2 * np.log(psi) - (n * (x_bar - theta)**2 + (n -
1) * s2) / (2 * psi)
    log_prior = -(xi**2 - 2 * rho * xi * gamma + gamma**2) / (2 * (1 -
rho**2))
    return log_like + log_prior

num_samples = 10000
samples = []
current = np.array([0.0, 0.0])
step_size = 0.2

for _ in range(num_samples):
    proposal = current + np.random.normal(scale=step_size, size=2)
    log_p_current = log_posterior(*current)
    log_p_proposal = log_posterior(*proposal)
    alpha = min(1, np.exp(log_p_proposal - log_p_current))
    if np.random.rand() < alpha:
        current = proposal
    samples.append(np.exp(current))

samples = np.array(samples)
theta_samples = samples[:, 0]
psi_samples = samples[:, 1]

fig, ax = plt.subplots(1, 1, figsize=(20, 6))
```

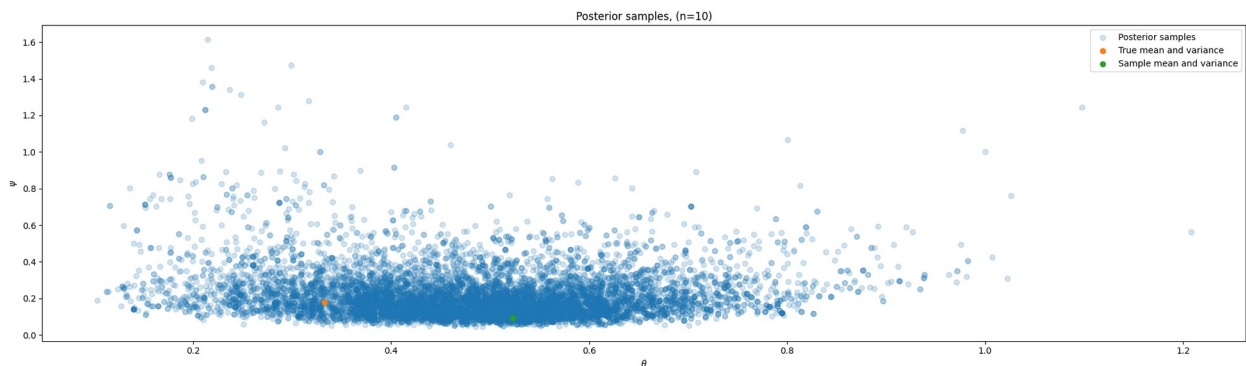


```

ax.scatter(theta_samples, psi_samples, alpha=0.2, label="Posterior
samples")
ax.scatter(theta_true, psi_true, label="True mean and variance")
ax.scatter(x_bar, s2, label="Sample mean and variance")
ax.set_title("Posterior samples, (n=10)")
ax.set_xlabel(r"$\theta$")
ax.set_ylabel(r"$\psi$")
ax.legend()

plt.tight_layout()
plt.show()

```



The posterior is centered around the sample mean and variance, but with high uncertainty. Given the low amount of data, the prior will dominate (even though it is rather weak), and the high uncertainty is therefore expected.

The posterior of  $\Psi$  is also right skewed (positive y-direction). This is consistent with an inverse- $\chi^2$  posterior, as suggested in the previous assignment.

(e) Repeat item 3d with  $n = 100$  and  $n = 1000$ , still using the values of  $(\theta, \psi)$  from item 3a. Discuss the results.

We repeat the posterior sampling, but now setting  $n=100$  and  $n=1000$ .

```

np.random.seed(69)

def simulate_and_sample(n, num_samples=10000, step_size=0.2):
    X = np.random.normal(loc=theta_true, scale=np.sqrt(psi_true),
size=n)

    x_bar = np.mean(X)
    s2 = np.var(X, ddof=1)

    def log_posterior(theta, psi):
        if theta < 0 or psi <= 0:
            return -np.inf
        log_like = -n / 2 * np.log(psi) - (n * (x_bar - theta)**2 + (n

```

```

- 1) * s2) / (2 * psi)
    log_prior = -(theta**2 - 2 * rho * theta * psi + psi**2) / (2
* (1 - rho**2))
    return log_like + log_prior

current = np.array([1.0, 1.0])
samples = []

for _ in range(num_samples):
    proposal = current + np.random.normal(scale=step_size, size=2)
    log_p_current = log_posterior(*current)
    log_p_proposal = log_posterior(*proposal)
    alpha = min(1, np.exp(log_p_proposal - log_p_current))
    if np.random.rand() < alpha:
        current = proposal
        samples.append(current)

samples = np.array(samples)
return samples[num_samples//10:, 0], samples[num_samples//10:, 1],
(x_bar, s2)

theta_10, psi_10, s_10 = simulate_and_sample(n=10)
theta_100, psi_100, s_100 = simulate_and_sample(n=100)
theta_1000, psi_1000, s_1000 = simulate_and_sample(n=1000)

fig, ax = plt.subplots(1, 3, figsize=(20, 6))

ax[0].scatter(theta_10, psi_10, alpha=0.2, label="Posterior samples")
ax[0].scatter(theta_true, psi_true, label="True mean and variance")
ax[0].scatter(*s_10, label="True mean and variance")
ax[0].set_title("Posterior samples, (n=10)")
ax[0].set_xlabel(r"$\theta$")
ax[0].set_ylabel(r"$\psi$")
ax[0].set_xlim(0, 1)
ax[0].set_ylim(0, 1)

ax[1].scatter(theta_100, psi_100, alpha=0.2, label="Posterior
samples")
ax[1].scatter(theta_true, psi_true, label="True mean and variance")
ax[1].scatter(*s_100, label="Sample mean and variance")
ax[1].set_title("Posterior samples, (n=100)")
ax[1].set_xlabel(r"$\theta$")
ax[1].set_ylabel(r"$\psi$")
ax[1].set_xlim(0, 1)
ax[1].set_ylim(0, 1)

ax[2].scatter(theta_1000, psi_1000, alpha=0.2, label="Posterior
samples")
ax[2].scatter(theta_true, psi_true, label="True mean and variance")

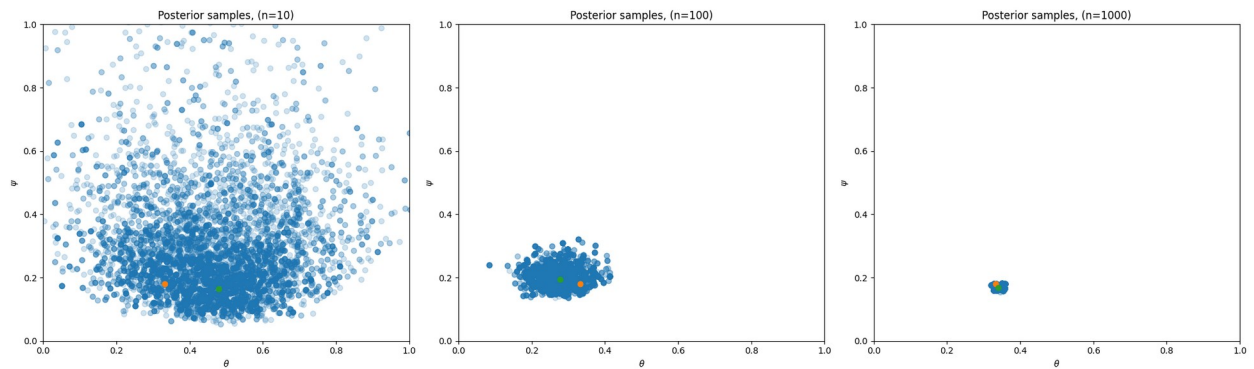
```

```

ax[2].scatter(*s_1000, label="Sample mean and variance")
ax[2].set_title("Posterior samples, (n=1000)")
ax[2].set_xlabel(r"$\theta$")
ax[2].set_ylabel(r"$\psi$")
ax[2].set_xlim(0, 1)
ax[2].set_ylim(0, 1)

plt.tight_layout()
plt.show()

```



As  $n$  increases, we see the distributions become narrower and more concentrated around the sample mean and variance, which corresponds to the likelihood dominating the prior as the amount of data increases.

## Exercise 7: Simulated annealing

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import uniform
```

1. Implement simulated annealing for the travelling salesman. As proposal, permute two random stations on the route. As cooling scheme, you can use e.g.  $T_k = 1/\sqrt{1+k}$  or  $T_k = -\log(k+1)$ , feel free to experiment with different choices. The route must end where it started. Initialise with a random permutation of stations.

Below we implement simulated annealing. It takes an initial state, as well as functions to calculate energy, permutations, and temperatures:

```
def simulated_annealing(initial_state, energy, permute, t_func, N):
    state = initial_state
    energies = np.empty(N)
    accepted = np.zeros(N)

    for k in range(N):
        T = t_func(k)
        proposed_state = permute(state)

        e_state = energy(state)
        e_proposed = energy(proposed_state)

        energies[k] = e_state

        accept_prob = min(1, np.exp(-(e_proposed - e_state) / T))

        if uniform.rvs() < accept_prob:
            state = proposed_state
            accepted[k] = 1

    return state, energies, accepted
```

We then define functions for generating permutations, generating temperature functions, and calculating energies of states. Note that we abuse the structure of notebooks slightly, `n_stations` and `cost_matrix` are not defined yet, but will be when the functions are called.

```

def permute(state):
    state = state.copy()
    indices = np.random.choice(n_stations, size=2, replace=False)
    state[indices] = state[indices[::-1]]
    return state

def get_t_func(t0, alpha):
    def t_func(k):
        return t0 * alpha ** k
    return t_func

def energy(state):
    cost = 0
    for i in range(n_stations):
        cost += cost_matrix[state[i], state[(i + 1) % n_stations]]
    return cost

```

(a) Have input be positions in the plane of the  $n$  stations. Let the cost of going  $i \rightarrow j$  be the Euclidian distance between station  $i$  and  $j$ . Plot the resulting route in the plane. Debug with stations on a circle.

We create 12 stations placed on a circle in the plane, and create the corresponding cost matrix. As seen in the code above, our cooling scheme is  $T_k = t_0 \alpha^k$ . We then run simulated annealing with the following parameters:

$N=500$  (number of iterations)

$t_0=100$  (temperature at  $T=0$ )

$\alpha=0.975$  (temperature scaling factor, per iteration)

```

np.random.seed(69)

n_stations = 12
stations = np.arange(n_stations)

theta = 2 * np.pi * stations / n_stations
x = np.cos(theta)
y = np.sin(theta)

cost_matrix = np.empty((n_stations, n_stations))
for i in range(n_stations):
    for j in range(n_stations):
        cost_matrix[i, j] = np.sqrt((x[i] - x[j]) ** 2 + (y[i] - y[j])
** 2)

N = 500
initial_state = np.random.permutation(np.arange(n_stations))

```

```

t_func = get_t_func(100, 0.975)

solution, energies, accepted = simulated_annealing(initial_state,
energy, permute, t_func, N)

/tmp/ipykernel_42823/1096169008.py:15: RuntimeWarning: overflow
encountered in exp
    accept_prob = min(1, np.exp(-(e_proposed - e_state) / T))

```

We then plot the temperature and energy curves, along with the initial state and the solution state:

```

def plot_route(ax, x, y, indices, title):
    ax.set_title(title)
    ax.plot(x, y, 'o') # draw points
    closed_indices = np.append(indices, indices[0]) # return to start
    ax.plot(x[closed_indices], y[closed_indices], '-') # draw route

accept_iters = np.where(accepted == 1)[0]

fig, axes = plt.subplots(2, 2, figsize=(20, 8))

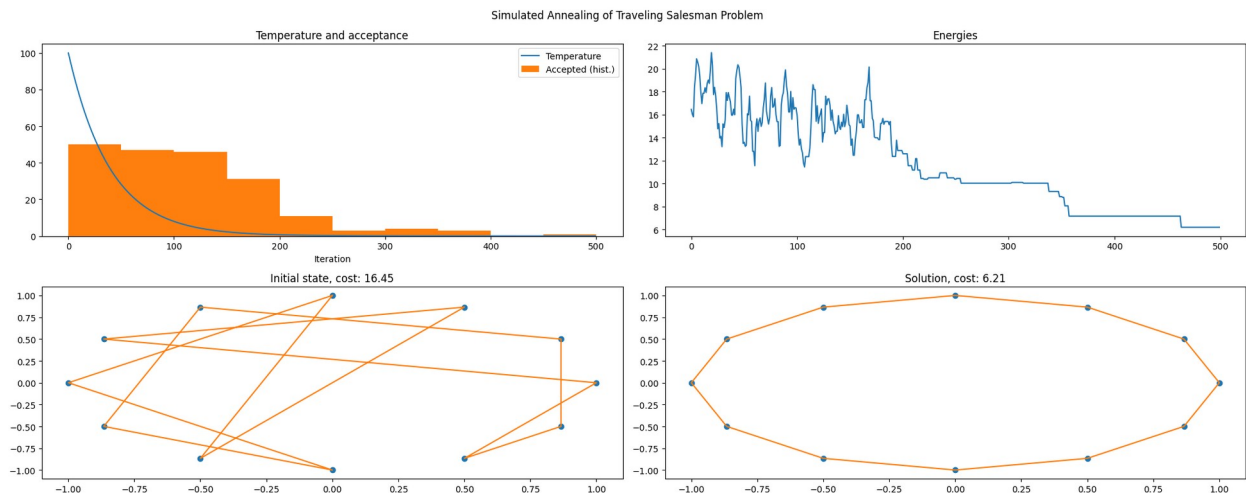
axes[0, 0].set_title("Temperature and acceptance")
axes[0, 0].plot(np.arange(N), t_func(np.arange(N)),
label="Temperature")
axes[0, 0].hist(accept_iters, range=(0, N), label="Accepted (hist.)")
axes[0, 0].set_xlabel("Iteration")
axes[0, 0].legend()

axes[0, 1].set_title("Energies")
axes[0, 1].plot(energies)
axes[0, 1].set_xlabel("Iteration")

fig.suptitle("Simulated Annealing of Traveling Salesman Problem")

plot_route(axes[1, 0], x, y, initial_state, f"Initial state, cost:
{energy(initial_state):.2f}")
plot_route(axes[1, 1], x, y, solution, f"Solution, cost:
{energy(solution):.2f}")
fig.tight_layout()

```



We get the expected solution; for points on a circle, the shortest TSP route is to visit them in order around the circle. Also, the energy curve and acceptance rate seems to taper off at a reasonable speed.

We tried the cooling schemes suggested in the exercise as well, but they did not work very well, so we switched to exponential dropoff. Our experiments taught us that a high initial temperature, coupled with a slow dropoff (that still ends at a low temperature) leads to good results.

(b) Then modify your programme to work with costs directly and apply it to the cost matrix from the course homepage.

Since our program is already built for cost matrices, we simply switch out the `cost_matrix`. This problem is harder than the debug case, so we adjust our parameters accordingly: Higher number of iterations gives more time to find a solution, higher  $t_0$  increases early exploration, and an  $\alpha$  closer to 1 lets the temperature decline slower to better match the higher iteration number.

$$N=10000$$

$$t_0=1000$$

$$\alpha=0.999$$

```
cost_matrix = np.array([
    [0, 225, 110, 8, 257, 22, 83, 231, 277, 243, 94, 30, 4, 265, 274,
     250, 87, 83, 271, 86],
    [255, 0, 265, 248, 103, 280, 236, 91, 3, 87, 274, 265, 236, 8, 24,
     95, 247, 259, 28, 259],
    [87, 236, 0, 95, 248, 110, 25, 274, 250, 271, 9, 244, 83, 250,
     248, 280, 29, 26, 239, 7],
    [8, 280, 83, 0, 236, 28, 91, 239, 280, 259, 103, 23, 6, 280, 244,
     259, 95, 87, 230, 84],
    [268, 87, 239, 271, 0, 244, 275, 9, 84, 25, 244, 239, 275, 83,
     110, 24, 274, 280, 84, 274],
    [21, 265, 99, 29, 259, 0, 99, 230, 265, 271, 87, 5, 22, 239, 236,
```

```

250, 87, 95, 271, 91],
[95, 236, 28, 91, 247, 93, 0, 247, 259, 244, 27, 91, 87, 268, 275,
280, 7, 8, 240, 27],
[280, 83, 250, 261, 4, 239, 230, 0, 103, 24, 239, 261, 271, 95,
87, 21, 274, 255, 110, 280],
[247, 9, 280, 274, 84, 255, 259, 99, 0, 87, 255, 274, 280, 3, 27,
83, 259, 244, 28, 274],
[230, 103, 268, 275, 23, 244, 264, 28, 83, 0, 268, 275, 261, 91,
95, 8, 277, 261, 84, 247],
[87, 239, 9, 103, 261, 110, 29, 255, 239, 261, 0, 259, 84, 239,
261, 242, 24, 25, 242, 5],
[30, 255, 95, 30, 247, 4, 87, 274, 242, 255, 99, 0, 24, 280, 274,
259, 91, 83, 247, 91],
[8, 261, 83, 6, 255, 29, 103, 261, 247, 242, 110, 29, 0, 261, 244,
230, 87, 84, 280, 100],
[242, 8, 259, 280, 99, 242, 244, 99, 3, 84, 280, 236, 259, 0, 27,
95, 274, 261, 24, 268],
[274, 22, 250, 236, 83, 261, 247, 103, 22, 91, 250, 236, 261, 25,
0, 103, 255, 261, 5, 247],
[244, 91, 261, 255, 28, 236, 261, 29, 103, 9, 242, 261, 244, 87,
110, 0, 242, 236, 95, 259],
[84, 236, 27, 99, 230, 83, 7, 259, 230, 230, 22, 87, 93, 250, 255,
247, 0, 9, 259, 24],
[91, 242, 28, 87, 250, 110, 6, 271, 271, 255, 27, 103, 84, 250,
271, 244, 5, 0, 271, 29],
[261, 24, 250, 271, 84, 255, 261, 87, 28, 110, 250, 248, 248, 22,
3, 103, 271, 248, 0, 236],
[103, 271, 8, 91, 255, 91, 21, 271, 236, 271, 7, 250, 83, 247,
250, 271, 22, 27, 248, 0]
])

```

```

n_stations = cost_matrix.shape[0]
stations = np.arange(n_stations)

```

```

# draw in circle pattern even though we dont know positions

```

```

theta = 2 * np.pi * stations / n_stations
x = np.cos(theta)
y = np.sin(theta)

```

```

t_func = get_t_func(1000, 0.999)

```

```

np.random.seed(3)

```

```

N = 10_000

```

```

initial_state = np.random.permutation(np.arange(n_stations))

```

```

solution, energies, accepted = simulated_annealing(initial_state,
energy, permute, t_func, N)

```



Again, we plot the solutions. This time the stations do not have a meaningful position in the plane, so we do not include the route plots. Instead, we plot the cost of each part of the route (cost of going from station to station), as a bar chart:

```
print(solution)
accept_iters = np.where(accepted == 1)[0]

fig, axes = plt.subplots(2, 2, figsize=(20, 8))

axes[0, 0].set_title("Temperature and acceptance")
axes[0, 0].plot(np.arange(N), t_func(np.arange(N)),
label="Temperature")
axes[0, 0].hist(accept_iters, range=(0, N), label="Accepted (hist.)")
axes[0, 0].set_xlabel("Iteration")
axes[0, 0].legend()

axes[0, 1].set_title("Energies")
axes[0, 1].plot(energies)
axes[0, 1].set_xlabel("Iteration")

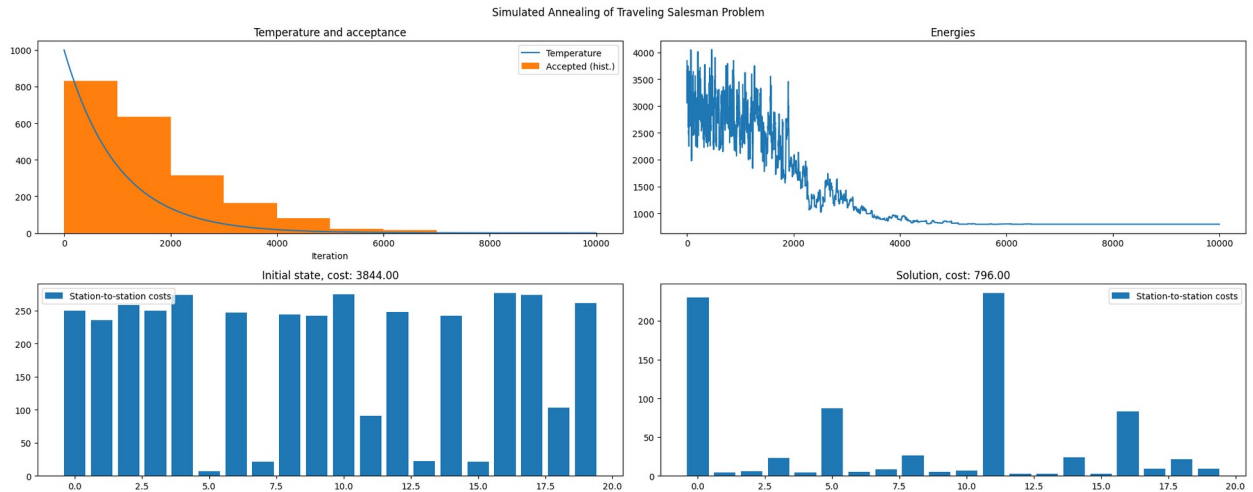
initial_costs = np.array([cost_matrix[initial_state[i],
initial_state[(i + 1) % n_stations]] for i in range(n_stations)])
solution_costs = np.array([cost_matrix[solution[i], solution[(i + 1) %
n_stations]] for i in range(n_stations)])

axes[1, 0].set_title(f"Initial state, cost:
{energy(initial_state):.2f}")
axes[1, 0].bar(np.arange(n_stations), initial_costs, label="Station-
to-station costs")
axes[1, 0].legend()

axes[1, 1].set_title(f"Solution, cost: {energy(solution):.2f}")
axes[1, 1].bar(np.arange(n_stations), solution_costs, label="Station-
to-station costs")
axes[1, 1].legend()

fig.suptitle("Simulated Annealing of Traveling Salesman Problem")
fig.tight_layout()
plt.show()
```

[ 9 0 12 3 11 5 10 19 2 17 16 6 1 8 13 18 14 4 7 15]



We obtain a solution with cost 796, where the visit order is:

$[9, 0, 12, 3, 11, 5, 10, 19, 2, 17, 16, 6, 1, 8, 13, 18, 14, 4, 7, 15]$

For this problem we do not know the cost of the optimal solution, but from the TA's we've gathered that costs in the range 750-900 are close to optimal.

We also see several signs of the algorithm having converged to a 'reasonable' solution:

- Acceptance rate drops off gradually
- Energy varies wildly in the beginning, and keeps varying at a smaller scale at later iterations
- Solution cost is much lower than initial state cost

## Exercise 8: Bootstrap

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import uniform
```

### 1. Exercise 13 in Chapter 8 of Ross (P.152).

(a)

Since  $\mu$  is unknown, we estimate it by taking the mean of the original sample.

Then, for each bootstrap sample we compute the difference between the sample mean and  $\mu$ .

The probability  $p$  can then be estimated as the proportion of these differences that fall outside the interval  $(a, b)$ .

(b)

```
xi = np.array([56, 101, 78, 67, 93, 87, 64, 72, 80, 69])
n = 10
a = -5
b = 5

n_bootstrap = 10_000

np.random.seed(42)
bootstrap_samples = np.random.choice(xi, size=(n_bootstrap, 10))

mu = np.mean(xi)

bootstrap = np.sum(bootstrap_samples / n, axis=1) - mu
p = np.mean(np.abs(bootstrap) < 5)
print(f"{p = :.3f}")

p = 0.767
```

From this, we obtain the bootstrap estimate  $p=0.767$ .

### 2. Exercise 15 in Chapter 8 of Ross (P.152).

We want to estimate the variance  $\sigma^2$  of the samples  $X_i$ , by the sample variance  $S^2 = \sum_{i=1}^n \hat{X}_i^2$ .

To understand how reliable this estimate is, we use the bootstrap technique to calculate  $Var(S^2)$ .

```

# seed for reproducability
np.random.seed(42)

# data given in the exercise
data = np.array([5, 4, 9, 6, 21, 17, 11, 20, 7, 10, 21, 15, 13, 16, 8])

# draw 10,000 bootstrap samples, each of size 15 (original sample size)
n = data.shape[0]
n_bootstrap = 10_000
bootstrap_samples = np.random.choice(data, size=(n_bootstrap, n))

# take the mean of each bootstrap sample ( $\bar{X}$ )
x_bar = np.mean(bootstrap_samples, axis=1).reshape(-1, 1)

# calculate sample variance for each bootstrap sample
S2 = np.sum((bootstrap_samples - x_bar)**2 / (n - 1), axis=1)

# calculate the variance of the sample variances  $S^2$ 
var_S2 = np.var(S2)

print(f"Var( $S^2$ ) = {var_S2:.2f}")

Var( $S^2$ ) = 59.14

```

From this we get the bootstrap estimate  $Var(S^2) = 59.14$

3. Write a subroutine that takes as input a “data” vector of observed values, and which outputs the median as well as the bootstrap estimate of the variance of the median, based on  $r = 100$  bootstrap replicates. Simulate  $N = 200$  Pareto distributed random variates with  $\beta = 1$  and  $k = 1.05$

First, we simulate 200 samples from the pareto distribution, and draw 100 bootstrap replicates:

```

def sample_pareto(n, k, beta):
    us = uniform.rvs(size=n)
    return beta * us ** (-1/k)

np.random.seed(69)

n = 200

sample = sample_pareto(n, k=1.05, beta=1)

bootstrap_samples = np.random.choice(sample, size=(100, n))

```

(a) Compute the mean and the median (of the sample)

```
sample_mean = np.mean(sample)
sample_median = np.median(sample)

print(f"Sample mean    = {sample_mean:.2f}")
print(f"Sample median = {sample_median:.2f}")

Sample mean    = 7.35
Sample median = 2.11
```

We get a sample mean of 7.35, and a sample median of 2.11.

(b) Make the bootstrap estimate of the variance of the sample mean.

```
bs_var_of_sample_mean = np.var(np.mean(bootstrap_samples, axis=1))

print(f"Variance of sample mean (bootstrap estimate):
{bs_var_of_sample_mean:.2f}")

Variance of sample mean (bootstrap estimate): 9.24
```

We find the bootstrap estimate of the sample mean to be 9.24.

(c) Make the bootstrap estimate of the variance of the sample median.

```
bs_var_of_sample_median = np.var(np.median(bootstrap_samples, axis=1))

print(f"Variance of sample median (bootstrap estimate):
{bs_var_of_sample_median:.3f}")

Variance of sample median (bootstrap estimate): 0.015
```

We find the bootstrap estimate of the sample median to be 0.015.

(d) Compare the precision of the estimated median with the precision of the estimated mean.

```
std_mean = np.sqrt(bs_var_of_sample_mean)
std_median = np.sqrt(bs_var_of_sample_median)
diff_of_stds = std_mean - std_median

print(f"Difference of var(sample mean) and var(sample median):
{bs_var_of_sample_mean - bs_var_of_sample_median:.3f}")
print(f"Difference of std(sample mean) and std(sample median):
{diff_of_stds:.3f}")

Difference of var(sample mean) and var(sample median): 9.221
Difference of std(sample mean) and std(sample median): 2.919
```

The difference between the variance of the sample mean and the variance of the sample median is 9.221.

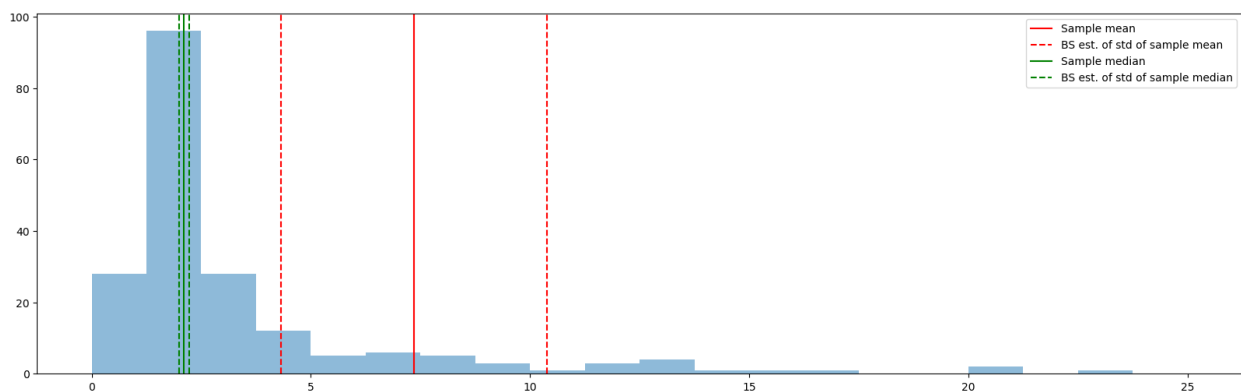
The difference between the standard deviation of the sample mean and the standard deviation of the sample median is 2.919.

```
fig, ax = plt.subplots(1, 1, figsize=(20, 6))
ax.hist(sample, range=(0, 25), bins=20, alpha=0.5)
ax.axvline(sample_mean, color="r", label="Sample mean")
ax.axvline(sample_mean + std_mean, color="r", linestyle="--",
label="BS est. of std of sample mean")
ax.axvline(sample_mean - std_mean, color="r", linestyle="--")

ax.axvline(sample_median, color="g", label="Sample median")
ax.axvline(sample_median + std_median, color="g", linestyle="--",
label="BS est. of std of sample median")
ax.axvline(sample_median - std_median, color="g", linestyle="--")

ax.legend()

plt.show()
```



We plot the sample mean and median, along with intervals of  $\pm 1$  their respective standard deviation.

We see that the median has a much narrower interval than the mean, which is exactly what we would expect from the pareto distribution, given how hard it is to estimate its mean (which we can assert using its first order moment distribution).