02443 - Handin of Exercises

s183529 - Jonas Søeborg Nielsen

s194323 - Aleksander Svendstorp

Computer Exercise 1: Generation and Testing of Random Numbers

In [ ]:
```python
# Plotting
import plotly.graph_objects as go
import plotly.express as px
import plotly.subplots as sp
import plotly.io as pio
pio.renderers.default = "notebook+pdf"
pio.templates.default = "plotly_dark"

# Utilities
import numpy as np
import pandas as pd
from scipy.stats import chi2, norm
from scipy.special import kolmogorov
```

Part 1 - Linear Congruential Generator (LCG)

First we implement a Linear Congruential Generator (LCG) to generate random numbers. The implementation can be seen in the python code below.

In [ ]:
```python
def LCG(a:int, c:int, M:int, x0:int, n:int, as_int=False):
    """
    LCG generates a list of random numbers using the Linear Congruent Generation method.
    """
    # Preallocate and initialize array for pseudorandom numbers
    X = np.zeros(n+1, dtype=int)
    X[0] = x0

    # Generate pseudorandom numbers
    for i in range(1, n+1):
        X[i] = (a*X[i-1] + c) % M

    if as_int:
        return X[1:]
    else:
        return X[1:]/M
```

To make sure our implementation works as intended, we compare with an example from the slides and check if the same stream of random numbers is generated.

In [ ]:
```python
M = 16
a = 5
c = 1
x0 = 3

U_true = np.array([0, 1, 6, 15, 12, 13, 2, 11, 8, 9, 14, 7, 4, 5, 10, 3])
U_LCG = LCG(a, c, M, x0, 16, as_int=True)

print(f"True sequence:\t{U_true}")
print(f"LCG sequence:\t{U_LCG}")
```

```
True sequence:  [ 0  1  6 15 12 13  2 11  8  9 14  7  4  5 10  3]
LCG sequence:   [ 0  1  6 15 12 13  2 11  8  9 14  7  4  5 10  3]
```

Below we prepare some functions for plotting histograms and scatter plots of the generated random numbers.

In [ ]:
```python
# Histogram
def plot_histogram(U, title="", n_bins=20):
    fig = go.Figure(go.Histogram(x=U, xbins=dict(start=0, end=1, size=1/n_bins), histnorm='probability density', marker=dict(color=
    fig.add_trace(go.Scatter(x=[0, 1], y=[1, 1], mode='lines', line=dict(color='Red', width=2)))
    fig.update_layout(title=title, xaxis_title="Value", yaxis_title="Density", width=600, height=400, bargap=0.1, showlegend=False)
    fig.show()

# Correlation plot
def plot_correlation(U, title="Correlation plot of consequtive numbers."):
    fig = go.Figure(go.Scatter(x=U[:-1], y=U[1:], mode='markers', marker=dict(size=2, color='Blue')))
```

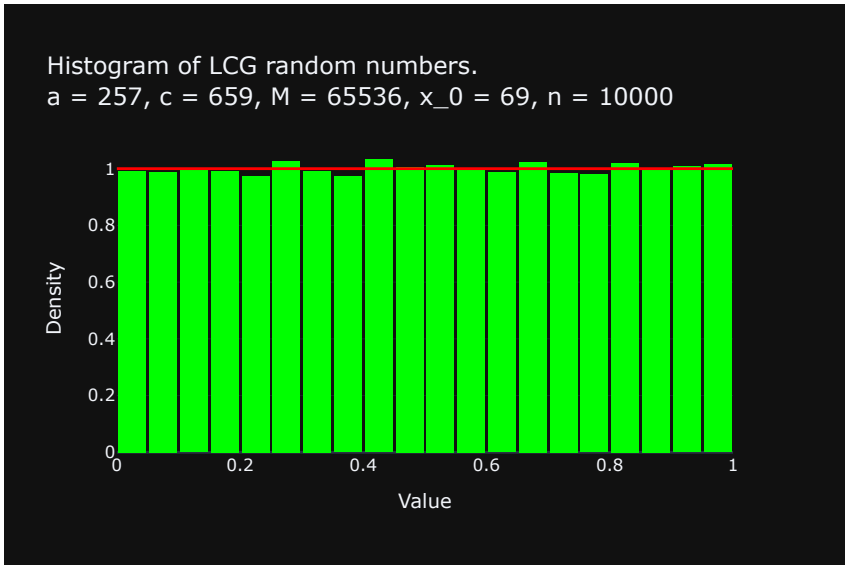
```
    fig.update_layout(title=title, xaxis_title=r"$U_{i-1}$", yaxis_title=r"$U_{i}$", width=600, height=600)
    fig.show()
```

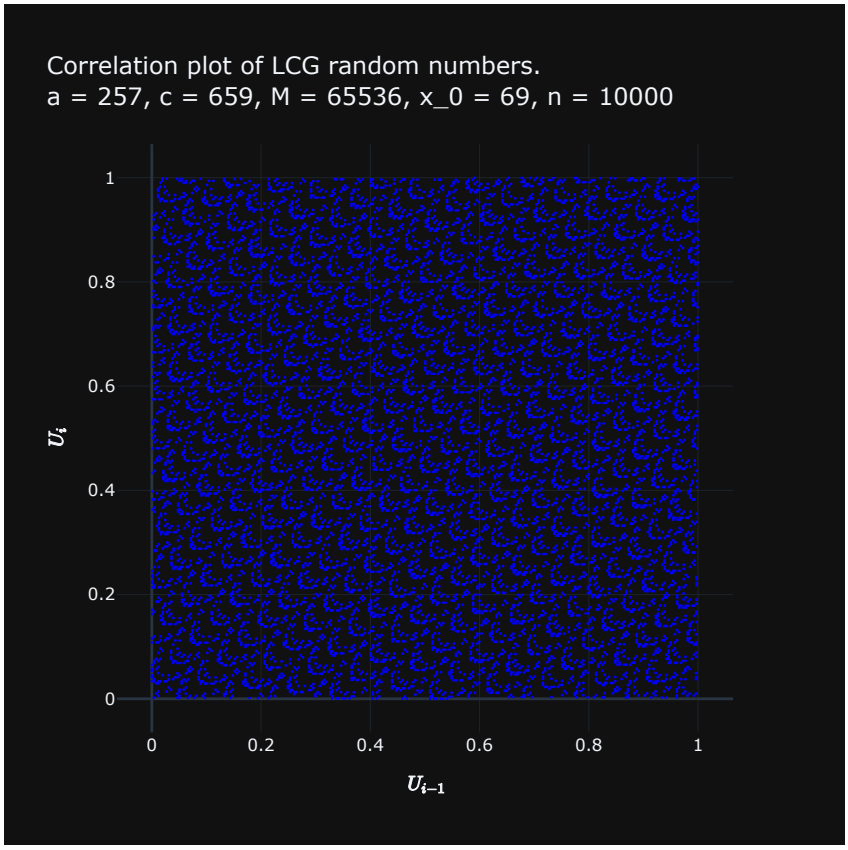
(a) Generating 10000 pseudorandom numbers

Using our implementation of the LCG algorithm, we generate 10000 pseudo random numbers. To ensure full cycle length, the parameters of the RNG is chosen according to the criteria in slide 14 of lecture 2.

We plot the histogram of the numbers and a scatter plot of the numbers in pairs of two.

In [ ]:
```
a = 257     # Prime
c = 659     # Prime
M = 65536   # 2^16
x0 = 69     # 3*23
n = 10000   # Number of samples

# Generate pseudorandom numbers
U = LCG(a, c, M, x0, n)

# Plot histogram
plot_histogram(U, title="Histogram of LCG random numbers.<br>a = {}, c = {}, M = {}, x_0 = {}, n = {}".format(a, c, M, x0, n))

# Plot correlation
plot_correlation(U, title="Correlation plot of LCG random numbers.<br>a = {}, c = {}, M = {}, x_0 = {}, n = {}".format(a, c, M, x0,
```





Considering the histogram, the numbers appear to be fairly uniformly distributed. However, the scatter plot clearly shows that there some underlying structure of these pseudo random numbers.

(b) RNG evaluation

Now we want to run different tests to evaluate the quality of the RNG. Below we implement the tests before running and comparing them.

Functions for testing distribution of random numbers

In [ ]:
```python
def chi_sq_test(U, n_classes=10):

    # Compute expected number of observations in each class
    n_expected = len(U) / n_classes

    # Count number of observations in each class
    n_obs, _ = np.histogram(U, bins=n_classes)

    # Compute test statistic
    T_obs = np.abs(np.sum((n_obs - n_expected)**2 / n_expected))

    # Compute p-value
    df = n_classes-1 # when number of estimated parameters is m=1
    p = 1 - chi2.cdf(T_obs, df)

    return T_obs, p
```

In [ ]:
```python
def kolmogorov_smirnov_test(U):

    # Get number of observations
    n = len(U)

    # Setup expected values of F
    F_exp = np.linspace(0, 1, n+1)[1:]

    # Compute test statistic
    Dn = max(abs(F_exp-np.sort(U)))

    # Compute p-value
    p = kolmogorov(Dn)

    return Dn, p
```

Runtests for testing independence of random numbers

In [ ]:
```
def above_below_runtest1(U):

    median = np.median(U)
    n1 = np.sum(U < median)
    n2 = np.sum(median < U)

    # Compute total number of observed runs
    temp = U > median
    T_obs = sum(temp[1:] ^ temp[:-1])

    # Compute p-value
    mean = 2*n1*n2/(n1 + n2) + 1
    log_expr = np.log(2) + np.log(n1) + np.log(n2) + np.log(2*n1*n2 - n1 - n2) - 2*np.log(n1 + n2) - np.log(n1 + n2 - 1)
    var = np.exp(log_expr)
    Z_obs = (T_obs - mean) / np.sqrt(var)
    p = 2 * (1 - norm.cdf(np.abs(Z_obs)))

    return T_obs, p
```

In [ ]:
```
def up_down_runtest2(U):

    n = len(U)

    # Get indeces where runs change (Append -1 and n-1 at ends to handle first and last run)
    idx = np.concatenate(([-1], np.where(U[1:]-U[:-1] < 0)[0], [len(U)-1]))

    # Compute run lengths and count them (clamp to 6)
    run_lengths = np.clip(idx[1:] - idx[:-1], 1, 6)
    R = np.array([np.count_nonzero(run_lengths == i) for i in range(1, 7)])

    # Compute test statistic
    A = np.array([
        [4529.4, 9044.9, 13568, 18091, 22615, 27892],
        [9044.9, 18097, 27139, 36187, 45234, 55789],
        [13568, 27139, 40721, 54281, 67852, 83685],
        [18091, 36187, 54281, 72414, 90470, 111580],
        [22615, 45234, 67852, 90470, 113262, 139476],
        [27892, 55789, 83685, 111580, 139476, 172860]
    ])
    B = np.array([1/6, 5/24, 11/120, 19/720, 29/5040, 1/840])
    Z_obs = (1/(n - 6)) * (R - n*B).T @ A @ (R - n*B)

    # Compute p-value
    p = 1 - chi2.cdf(Z_obs, 6)

    return Z_obs, p
```

In [ ]:
```
def up_and_down_runtest3(U):

    n = len(U)

    # Find runs (Append 0 at ends to handle first and last run)
    seq = np.concatenate(([0], np.sign(U[1:] - U[:-1]), [0]))

    # Get indeces where runs change
    idx = np.flatnonzero(seq[:-1] != seq[1:])

    # Compute run lengths
    run_lengths = idx[1:] - idx[:-1]
    X_obs = len(run_lengths)

    # Compute test statistic
    Z_obs = (X_obs - (2*n-1)/3) / np.sqrt((16*n - 29) / 90)

    # Compute p-value
    p = 2*(1 - norm.cdf(np.abs(Z_obs)))

    return Z_obs, p
```

In [ ]:
```
def corr_coef(U, h=2):
    n = len(U)
    ch = np.sum(U[:n-h]*U[h:])/(n-h)
    Z = (ch - 0.25)/(7/(144*n))
    p = 2*(1 - norm.cdf(np.abs(ch)))
    return ch, p
```

In [ ]:
```
def test_random_numbers(U):
```

```
    tests = [chi_sq_test, kolmogorov_smirnov_test, above_below_runtest1, up_down_runtest2, up_and_down_runtest3, corr_coef]
    table = np.array([test(U) for test in tests])

    df = pd.DataFrame(np.round(table, 2),
                      index=["Chi squared", "Kol-Smi", "Above/Below (I)", "Up/Down (II)", "Up and Down (III)", "Correlation"],
                      columns=["Test statistic", "p-value"]
    )
    print(df)
```

In [ ]:
```
test_random_numbers(U)
```

```
                  Test statistic  p-value
Chi squared                 1.38     1.00
Kol-Smi                     0.00     1.00
Above/Below (I)          4974.00     0.59
Up/Down (II)              368.35     0.00
Up and Down (III)          18.52     0.00
Correlation                 0.25     0.80
```

In the table above we have plotted the test statistics and the corresponding p-values for each of the tests. We see that the chi squared test and the Kolmogorov-Smirnov test have p-values above 0.05, which means we cannot reject the null hypothesis. For these two tests, the null hypothesis is that the random numbers are uniformly distributed. The next four tests are to test if the numbers we have generated are independent. We get mixed results from these tests. Two of the tests have p value above 0.05 meaning we cannot reject the null hypothesis, while the other two have p value below 0.05 meaning we can reject the null hypothesis. However by looking at the scatter plot of our generated points from part (a) we can see that the points are not independent. This is because the points are not scattered randomly, but rather in a pattern.

So in conclusion, the random numbers we have generated are not independent, but they are uniformly distributed. This is most likely since our choice of parameters can be further tuned to get better results.

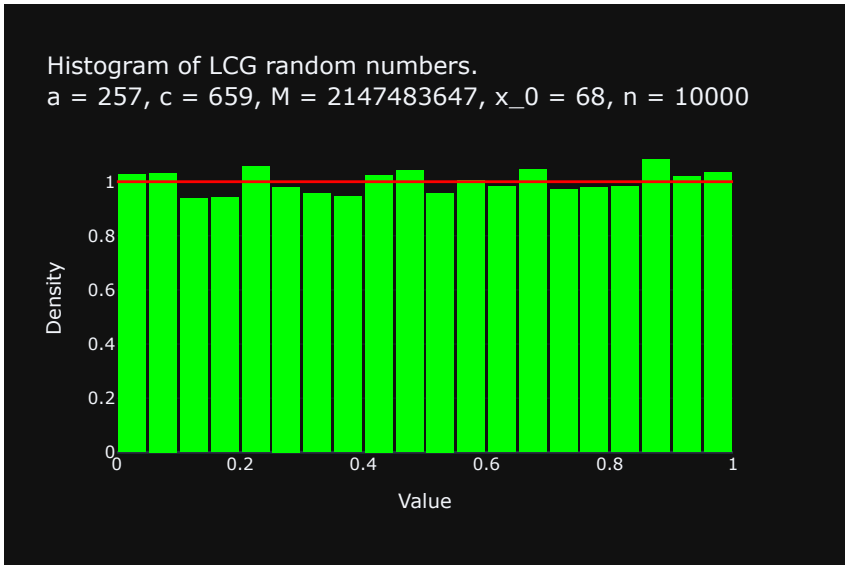
(c) Experimenting with the RNG

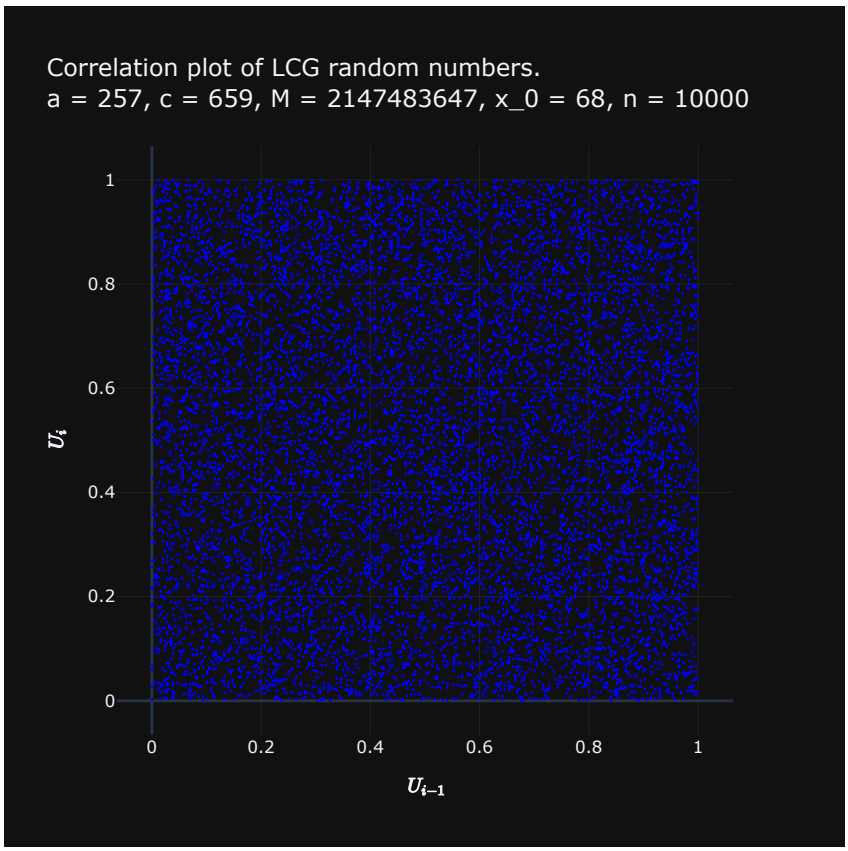
It is clear that our initial choice of parameters for the RNG are not optimal. While the histogram looks fairly uniform, the scatterplot clearly shows a repeating pattern in these number. Additionally, the RNG fail runtests I and II. We suspect that this is due to M being small compared to the number of generated numbers.

To rectify this and hopefully generate better pseudo random numbers, we will increase M to $2^{31} - 1$, a large Mersenne prime.

In [ ]:
```
a = 257     # Prime
c = 659     # Prime
M = 2**31-1 # Large Mersenne Prime
x0 = 69-1   # 3*23 -1
n = 10000   # Number of samples

# Generate pseudorandom numbers
U = LCG(a, c, M, x0, n)

# Plot histogram
plot_histogram(U, title="Histogram of LCG random numbers.<br>a = {}, c = {}, M = {}, x_0 = {}, n = {}".format(a, c, M, x0, n))

# Plot correlation
plot_correlation(U, title="Correlation plot of LCG random numbers.<br>a = {}, c = {}, M = {}, x_0 = {}, n = {}".format(a, c, M, x0,

test_random_numbers(U)
```

```
C:\Users\jonas\AppData\Local\Temp\ipykernel_7040\2080446664.py:11: RuntimeWarning:

overflow encountered in scalar multiply
```





	Test statistic	p-value
Chi squared	11.23	0.26
Kol-Smi	0.01	1.00
Above/Below (I)	5090.00	0.08
Up/Down (II)	8.94	0.18
Up and Down (III)	0.28	0.78
Correlation	0.25	0.80

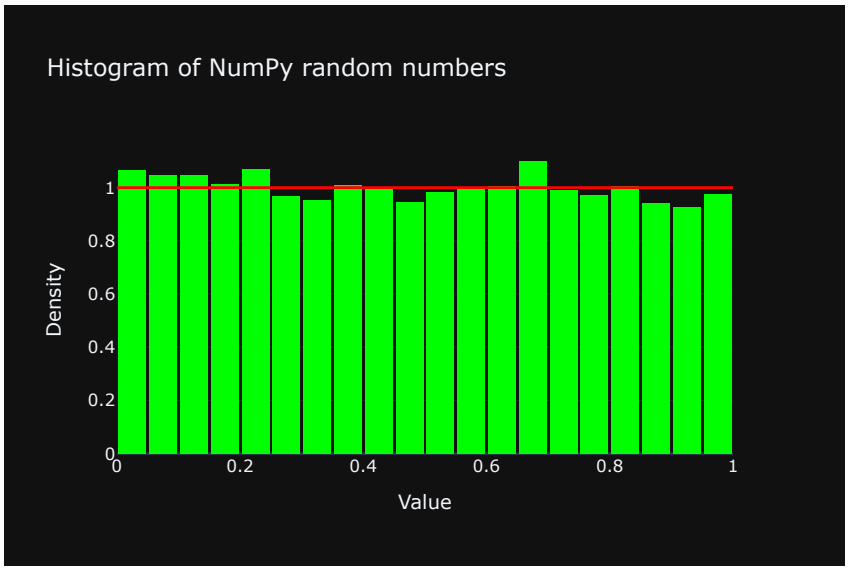
These new pseudorandom appears fairly uniformly distributed in the histogram, and the scatterplot shows no clear pattern within the numbers. All of the tests yield high p-values, thus the hypothesis of these numbers being independent and uniformly distributed cannot be rejected.

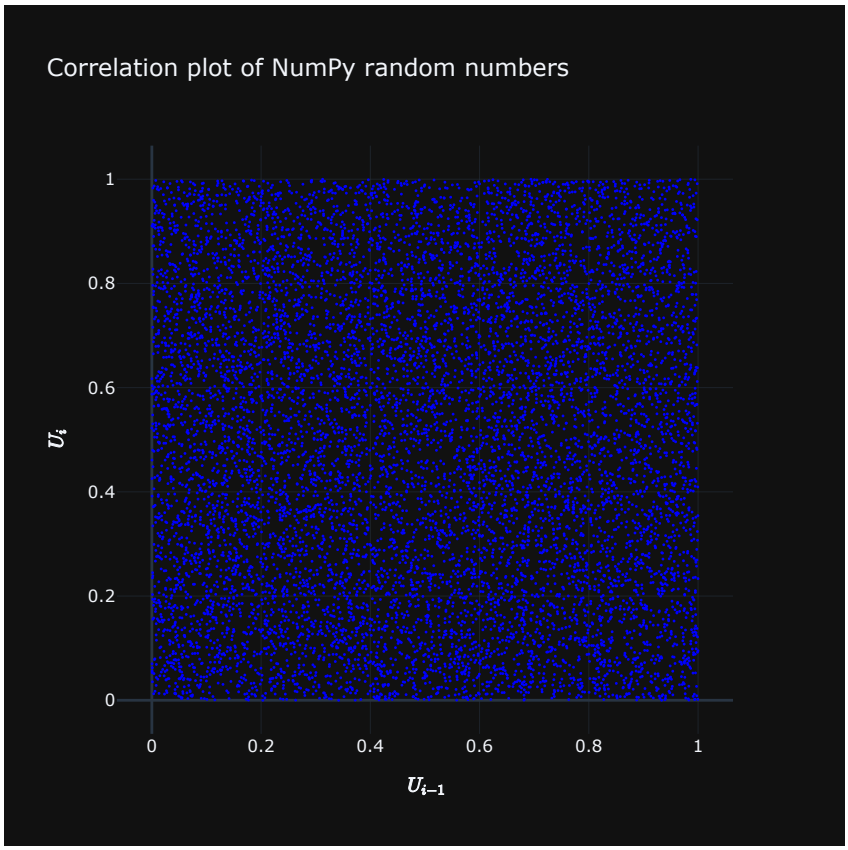
Part 2 - System Available Generator (NumPy)

NumPy uses a Mersenne Twister generator, called MT19937, from the standard C++ library (https://cplusplus.com/reference/random/mt19937/).

In [ ]:
```
# Using numpy to generate random numbers
U = np.random.rand(10000)
```

In [ ]:
```
# Histogram
plot_histogram(U, title="Histogram of NumPy random numbers")

# Plot correlation
plot_correlation(U, title="Correlation plot of NumPy random numbers")

test_random_numbers(U)
```





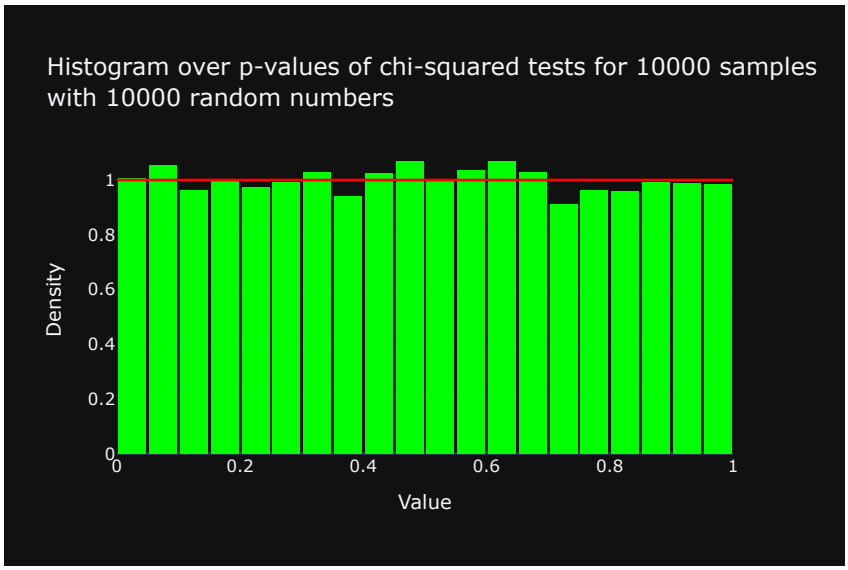
	Test statistic	p-value
Chi squared	11.99	0.21
Kol-Smi	0.01	1.00
Above/Below (I)	5005.00	0.94
Up/Down (II)	6.30	0.39
Up and Down (III)	1.01	0.31
Correlation	0.24	0.81

The numbers generated by NumPy appear to random, independent and uniformly distributed according to our current tests.

Part 3 - Discussion of One Sample Approach

To effectively evaluate a Random Number Generator (RNG), generating only one sample of random numbers is insufficient. Testing a single sample can occasionally produce misleading results, falsely rejecting the hypothesis that the numbers are uniformly distributed or independent. Given the hypothesis is true, the p-values obtained from these tests are expected to be uniformly distributed. Consequently, some samples may exhibit significantly low p-values by chance. Relying on just one sample increases the risk of incorrectly rejecting the hypothesis based on these results. By generating multiple samples, we can determine if the p-values consistently indicate significance or follow a uniform distribution, providing a more robust assessment of the RNG's performance.

In [ ]:
```python
# Run chi-squared test 10000 times
n_samples = 10000
n_random_numbers = 10000
p_values = np.zeros(n_samples)
for i in range(n_samples):
    U = np.random.rand(n_random_numbers)
    _, p_values[i] = chi_sq_test(U, n_classes=10)

# Plot histogram of p-values
plot_histogram(p_values, title=f"Histogram over p-values of chi-squared tests for {n_samples} samples<br>with {n_random_numbers} ra
```



The histogram above shows that the p-values of the chi squared test are uniformly distributed. This indicates that the RNG is working as intended, generating uniformly distributed random numbers.

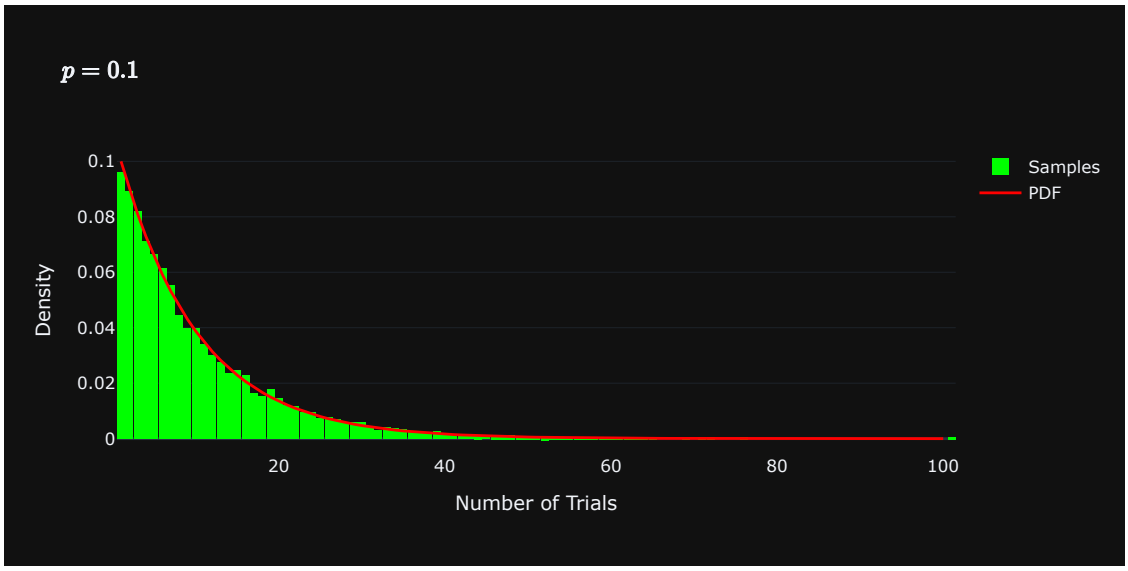
Computer Exercise 2: Sampling from Discrete Distributions

In [ ]:
```
# Plotting
import plotly.graph_objects as go
import plotly.express as px
import plotly.subplots as sp
import plotly.io as pio
pio.renderers.default = "notebook+pdf"
pio.templates.default = "plotly_dark"

# Utilities
import numpy as np
import pandas as pd
from scipy.stats import geom
from utils import chi_sq_test, kolmogorov_smirnov_test
```

Part 1 - Simulation using p

First we will simulate numbers from the geometric distribution with parameter p. We sample numbers from the uniform distribution and use the formula from the slides to calculate our samples. We choose $p = 0.1$.

In [ ]:
```
# Set p to a fixed value on the open interval between 0 and 1
p = 0.1
N = 10000

# Generate uniform random numbers
U = np.random.uniform(0, 1, N)

# Use formula from lecture 3 slide 9 to get geometric distributed discrete numbers
X = np.ceil(np.log(U)/np.log(1-p))

# Histogram
fig = go.Figure(go.Histogram(x=X, histnorm='probability density', marker_color='Lime', name='Samples'))
fig.add_trace(go.Scatter(x=np.arange(min(X), max(X)), y=geom.pmf(np.arange(min(X), max(X)), p), mode='lines', name='PDF', line=dict
fig.update_layout(title=f"Geometric Distribution with $p={p}$", xaxis_title="Number of Trials", yaxis_title="Density", width=800, h
fig.show()
```



We see that the samples generated clearly follow a geometric distribution with parameter $p = 0.1$.

Part 2 - Simulating 6-point distribution

For the next part we are given probabilities for a 6-point distribution. We will then sample numbers from this distribution using different methods.

First we implement functions for the methods before running the sampling and comparing.

In [ ]:
```
ps = np.array([7/48, 5/48, 1/8, 1/16, 1/4, 5/16])
```

(a) Direct method

For the direct method, we approximate the CDF from the given probabilities. We then take samples from the uniform distribution and do a linear search from the approximated CDF and return the corresponding value.

In [ ]:
```
def direct(n, ps):
    # Generate uniform random numbers
    U = np.random.rand(N)

    # Convert to discrete random numbers using the given probabilities
    X = np.searchsorted(np.cumsum(ps), U)

    return X
```

(b) Rejection method

With the rejection method we sample two numbers from the uniform distribution. We use the first number to calculate the sample value and the second number to decide if we accept the sample or not.

In [ ]:
```
def rejection(n, ps):
    c = max(ps)
    k = len(ps)

    X = np.zeros(n, dtype=int)
    for i in range(len(X)):
        while True: # Could theoretically run forever...
            U1, U2 = np.random.rand(2)
            I = np.floor(k * U1).astype(int)
            if U2 <= ps[I]/c:
                X[i] = I + 1
                break

    return X
```

(c) Alias method

To use the Alias method we first need to generate two tables. These tables will be used to keep track of when we are going to use the alias method and what the corresponding value is. We then sample from the uniform distribution and use the tables to return the corresponding value.

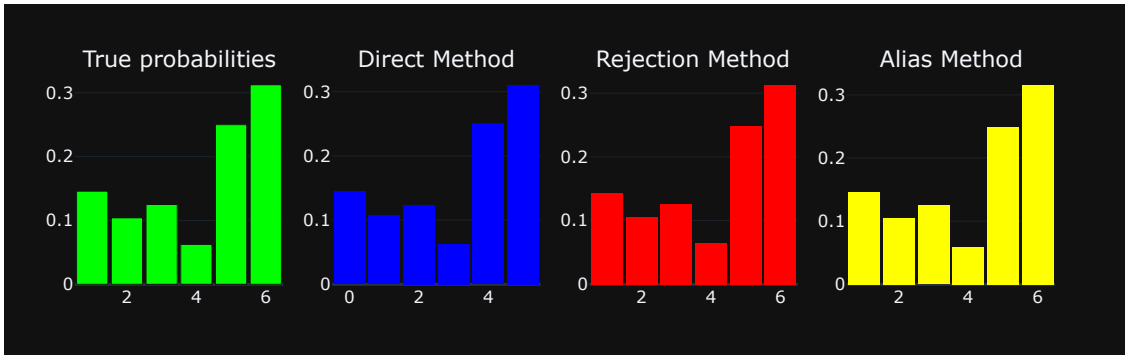
In [ ]:
```
def alias(N, ps):
    k = len(ps)

    # Generating Alias tables
    L = np.arange(k)
    F = k*ps
    G = np.where(F >= 1)[0]
    S = np.where(F <= 1)[0]

    while len(S) != 0:
        i = G[0]
        j = S[0]
        L[j] = i
        F[i] -= (1 - F[j])
        if F[i] < 1 - np.finfo(float).eps:
            G = np.delete(G, 0)
            S = np.append(S, i)
        S = np.delete(S, 0)

    # Computing values
    X = np.zeros(N, dtype=int)

    # Generate random numbers
    U1 = np.random.rand(N)
    U2 = np.random.rand(N)

    # Perform Alias method
    I = np.array(np.floor(k * U1)).astype(int)
    mask = U2 <= F[I]
    X[mask] = I[mask] + 1
    X[~mask] = L[I[~mask]] + 1

    return X
```

Part 3 - Comparison

We know use the three methods to generate samples from the given 6-point distribution and compare it with the given probabilities.

Histograms

In [ ]:
```
X_d = direct(N, ps)
X_r = rejection(N, ps)
X_a = alias(N, ps)

# Plot histograms
fig = sp.make_subplots(rows=1, cols=4, subplot_titles=("True probabilities", "Direct Method", "Rejection Method", "Alias Method"))
fig.add_trace(go.Bar(x=np.arange(1, 7), y=ps, marker_color='Lime'), row=1, col=1) # True probabilities
fig.add_trace(go.Histogram(x=X_d, histnorm='probability density', marker_color='Blue'), row=1, col=2) # Direct method
fig.add_trace(go.Histogram(x=X_r, histnorm='probability density', marker_color='Red'), row=1, col=3) # Rejection method
fig.add_trace(go.Histogram(x=X_a, histnorm='probability density', marker_color='Yellow'), row=1, col=4) # Alias method
fig.update_layout(height=250, width=800, showlegend=False, bargap=0.1, margin=dict(l=50, r=50, t=50, b=50))
fig.show()
```



All methods yield discrete random numbers that appear to be correctly distributed when inspecting the histogram.

Tests

In [ ]:
```
methods = ["Direct", "Rejection", "Alias"]
print("-"*36)
for i, X in enumerate([X_d, X_r, X_a]):
    tab = np.array([chi_sq_test(X, ps), kolmogorov_smirnov_test(X, ps)])
    df = pd.DataFrame(np.round(tab, 2), index=["Chi squared", "Kol-Smi"], columns=["Test statistic", "p-value"])
    print(">>> " + methods[i] + " method <<<")
    print(df)
    print("-"*36)
```

```
------------------------------------
>>> Direct method <<<
             Test statistic  p-value
Chi squared            1.17     0.95
Kol-Smi                0.00     1.00
------------------------------------
>>> Rejection method <<<
             Test statistic  p-value
Chi squared            1.68     0.89
Kol-Smi                0.00     1.00
------------------------------------
>>> Alias method <<<
             Test statistic  p-value
Chi squared            2.55     0.77
Kol-Smi                0.00     1.00
------------------------------------
```

For each method, we have calculated the p-value using both the chi squared test and the Kolmogorov Smirnov test. All p-values do not give us ground for rejecting the null hypothesis, i.e. that the data is drawn from the same distribution as the given probabilities.

Part 4 - Pros and Cons of the different methods

The rejection method has the con that to achieve the desired number of samples then it potentially needs to run for many more iterations due to the rejections. Many rejections can happen when you have some probabilities that are very low and some probabilities that are quite high. With $c = \max(p_i)$ then p_i/c will also be quite low when p_i is low. This means that the probability of accepting a sample is quite low.

The Alias method does not have any rejection, it does however have some initialization. This initialization can be quite costly if you have a large number of samples to generate. The sampling itself is however quite fast, so in cases where you need to repeatedly sample from the same distribution, the Alias method is a good choice.

The direct method is the simplest of the three methods. It does not require any initialization and is quite fast. The downside is that it requires a linear search for each sample. This can be quite costly if you have a large number of samples to generate.

Computer Exercise 3: Sampling from Continuous Distributions

In [ ]:
```
# Plotting
import plotly.graph_objects as go
import plotly.express as px
import plotly.subplots as sp
import plotly.io as pio
pio.renderers.default = "notebook+pdf"
pio.templates.default = "plotly_dark"

# Utilities
import numpy as np
from scipy.stats import t, chi2, kstest
```

Part 1 - Generating simulated values

In this exercise, we will generate random samples from a continuous distribution using the inverse transform method.

First we define the relevant functions needed to transform uniform samples to the needed samples.

In [ ]:
```
def exp_pdf(x, y=1):
    return y*np.exp(-y*x)

def pareto_pdf(x, k=20, beta=1):
    return k*beta**k / x**(k+1)

def gaussian_pdf(x, mu=0, sigma=1):
    return 1/np.sqrt(2*np.pi)*np.exp(-((x-mu)**2)/(2*sigma**2))

def uniform_2_exponential(U, y=1):
    X = -np.log(U)/y
    return X

def uniform_2_pareto(U, k=20, beta=1):
    X = beta*(U**(-1/k))
    return X

def uniform_2_normal(U):
    n = len(U)
    U1, U2 = U[:int(n/2)], U[int(n/2):]
    theta = 2*np.pi*U2
    r = np.sqrt(-2*np.log(U1))
    Z1, Z2 = r*np.array([np.cos(theta), np.sin(theta)])
    return np.concatenate((Z1, Z2))
```

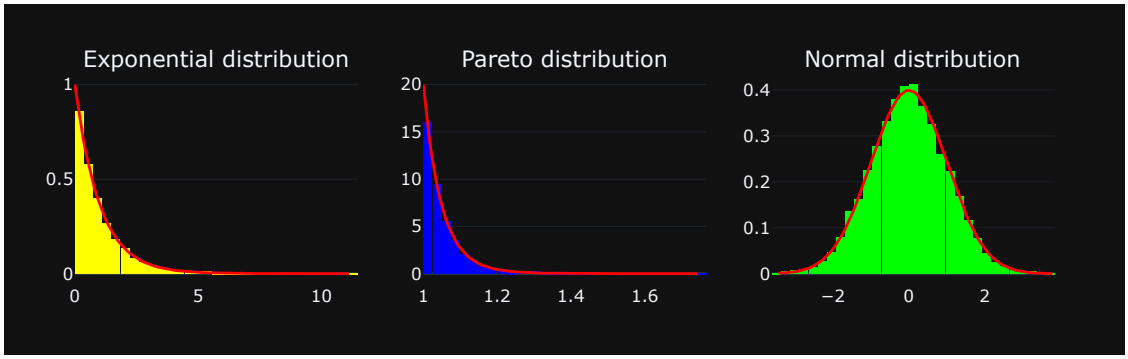
Generate random numbers from exponential, pareto and normal distributions

Using the defined functions, we can generate random samples from the exponential, pareto and normal distributions.

In [ ]:
```
U = np.random.rand(10000)

X_exp = uniform_2_exponential(U)
X_pareto = uniform_2_pareto(U)
X_norm = uniform_2_normal(U)
```

Plot histograms of the generated random numbers with superimposed pdf

In [ ]:
```
# Plot
num_bins = 30
fig = sp.make_subplots(rows=1, cols=3, subplot_titles=("Exponential distribution", "Pareto distribution", "Normal distribution"))

x_exp = np.linspace(min(X_exp), max(X_exp), 100, endpoint=True)
fig.add_trace(go.Scatter(x=x_exp, y=exp_pdf(x_exp), mode='lines', line=dict(color='red')), row=1, col=1)
fig.add_trace(go.Histogram(x=X_exp, xbins=dict(start=0, size=(x_exp[-1]-x_exp[0])/num_bins), histnorm='probability density', name='

x_pareto = np.linspace(min(X_pareto), max(X_pareto), 100, endpoint=True)
fig.add_trace(go.Scatter(x=x_pareto, y=pareto_pdf(x_pareto), mode='lines', line=dict(color='red'), showlegend=False), row=1, col=2)
fig.add_trace(go.Histogram(x=X_pareto, xbins=dict(start=1, size=(x_pareto[-1]-x_pareto[0])/num_bins), histnorm='probability density

x_norm = np.linspace(min(X_norm), max(X_norm), 100, endpoint=True)
fig.add_trace(go.Scatter(x=x_norm, y=gaussian_pdf(x_norm), mode='lines', line=dict(color='red'), showlegend=False), row=1, col=3)
fig.add_trace(go.Histogram(x=X_norm, xbins=dict(size=(x_norm[-1]-x_norm[0])/num_bins), histnorm='probability density', name='Sample

fig.update_layout(height=250, width=800, showlegend=False, bargap=0.1, margin=dict(l=50, r=50, t=50, b=50))
```



Above are histrograms of the generated samples from the three distributions along with the probability density functions. It is seen that the samples generated using the inverse transform method are consistent with the theoretical pdfs.

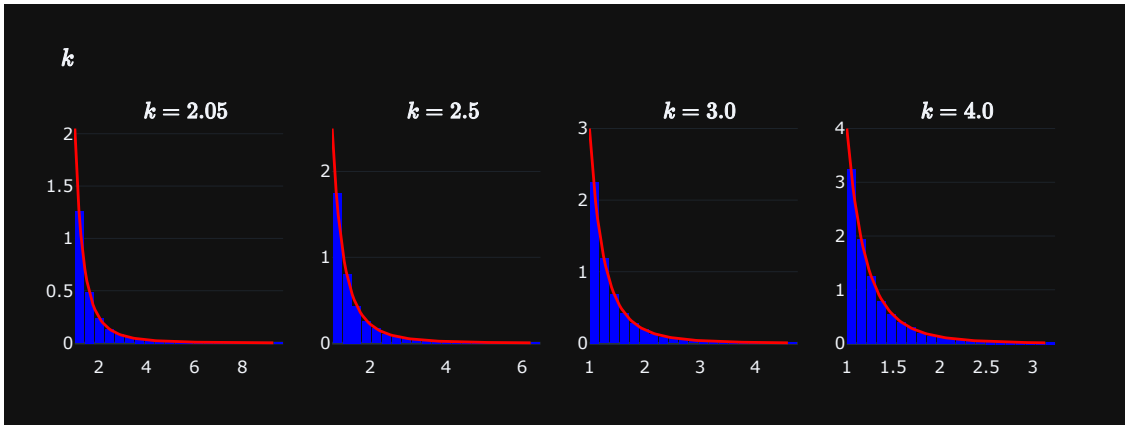
In [ ]:
```
# Perform Kolmogorov-Smirnov test for distribution type
ks_exp, exp_p = kstest(X_exp, 'expon')
ks_pareto, pareto_p = kstest(X_pareto, 'pareto', args=(20,))
ks_norm, norm_p = kstest(X_norm, 'norm')


print(f"The p-value for the exponential distribution: {exp_p:.4f}")
print(f"The p-value for the pareto distribution: {pareto_p:.4f}")
print(f"The p-value for the normal distribution: {norm_p:.4f}")
```

```
The p-value for the exponential distribution: 0.1354
The p-value for the pareto distribution: 0.1354
The p-value for the normal distribution: 0.1135
```

Using the kolmogorov-smirnov test, we can test the null hypothesis that the samples are drawn from the theoretical distribution. The p-values are all greater than 0.05, so we cannot reject the null hypothesis.

We further investigate sampling from the pareto distribution with different values of k.

In [ ]:
```
ks = np.array([2.05, 2.5, 3, 4])
ps = np.zeros(len(ks))

# plot using plotly for each of the ks
fig = sp.make_subplots(rows=1, cols=len(ks), subplot_titles=[r"$k = {}$".format(k) for k in ks])
for i, k in enumerate(ks):
    X = uniform_2_pareto(U, k)
    X_99 = X[X < np.quantile(X, 0.99)] # Remove the highest 1% of the values for plotting
    x = np.linspace(min(X_99), max(X_99), 100, endpoint=True)
    T, p = kstest(X, 'pareto', args=(k,))
    ps[i] = p
    fig.add_trace(go.Scatter(x=x, y=pareto_pdf(x, k), mode='lines', name='Pareto PDF', line=dict(color='red')), row=1, col=i+1)
    fig.add_trace(go.Histogram(x=X_99, xbins=dict(start=1, size=(max(X_99)-min(X_99))/20), histnorm='probability density', name='Pa

fig.update_layout(height=300, width=800, showlegend=False, bargap=0.1, margin=dict(l=50, r=50, t=80, b=50), title="Pareto distribut
fig.show()
```



Above we have plotted samples from the pareto distribution with different values of k along with the analytical pdf. The analytical expression for the expectation of the pareto distribution is:

$$E[X] = \frac{k}{k-1}\beta$$

So when we increase k we expect that the mean should increase towards β. This is also what see happening in the histograms above. The analytical expression for the variance of the pareto distribution is:

$$Var[X] = \frac{\beta^2 k}{(k-1)^2(k-2)}$$

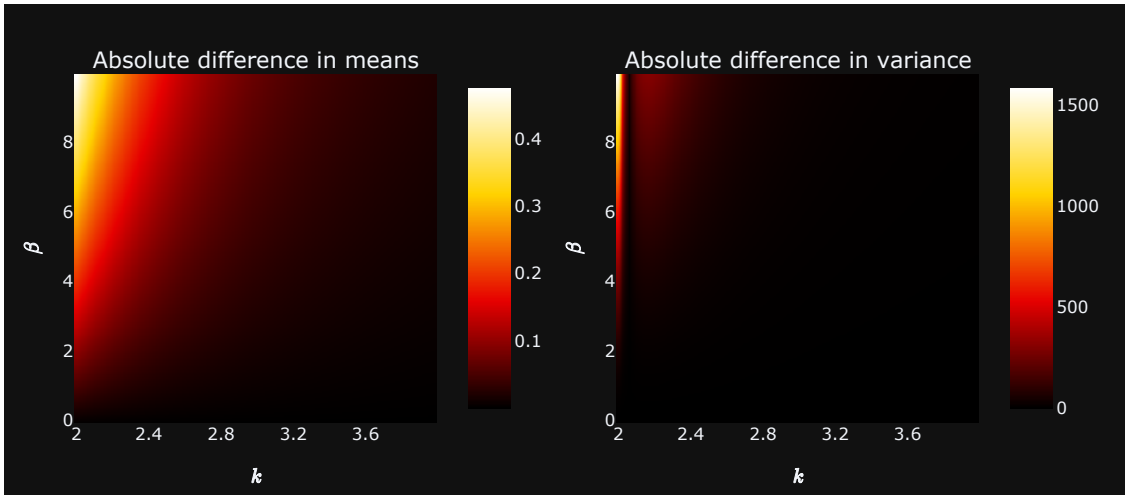
So when we increase k we expect that the variance should decrease. This is also what we see happening in the histograms above as the x-axis' range is getting smaller and smaller as k increases.

In [ ]:
```
for i, k in enumerate(ks):
    print(f"The p-value for the pareto distribution with k = {k}: {ps[i]:.4f}")
```

```
The p-value for the pareto distribution with k = 2.05: 0.1354
The p-value for the pareto distribution with k = 2.5: 0.1354
The p-value for the pareto distribution with k = 3.0: 0.1354
The p-value for the pareto distribution with k = 4.0: 0.1354
```

Part 2 - Comparison of simulation and analytic results for Pareto distribution

In [ ]:
```
nk = 100
nbeta = 100

ks = np.linspace(2.05, 4, nk)
betas = np.linspace(0.05, 10, nbeta)

mean_diff = np.zeros((nk, nbeta))
var_diff = np.zeros((nk, nbeta))
E = np.zeros((nk, nbeta))
V = np.zeros((nk, nbeta))

U = np.random.rand(10000)

for i, k in enumerate(ks):
    for j, beta in enumerate(betas):
        X = uniform_2_pareto(U, k, beta)
        EX = beta*k/(k-1)
        VarX = beta**2 * k / ((k-1)**2 * (k-2))
        mean_diff[j, i] = abs(np.mean(X) - EX)
        var_diff[j, i] = abs(np.var(X) - VarX)

fig = sp.make_subplots(rows=1, cols=2, subplot_titles=["Absolute difference in means", "Absolute difference in variance"], horizonta
fig.add_trace(go.Heatmap(z=mean_diff, colorscale='hot', zsmooth='best', colorbar_x=0.42), row=1, col=1)
fig.add_trace(go.Heatmap(z=var_diff, colorscale='hot', zsmooth='best'), row=1, col=2)
fig.update_xaxes(title_text=r"$k$", tickvals=np.arange(0, nk, 20), ticktext=np.round(ks[::20], 1), row=1, col=1)
fig.update_yaxes(title_text=r"$\beta$", tickvals=np.arange(0, nbeta, 20), ticktext=np.round(betas[::20]), row=1, col=1)
fig.update_xaxes(title_text=r"$k$", tickvals=np.arange(0, nk, 20), ticktext=np.round(ks[::20], 1), row=1, col=2)
fig.update_yaxes(title_text=r"$\beta$", tickvals=np.arange(0, nbeta, 20), ticktext=np.round(betas[::20]), row=1, col=2)
fig.update_layout(height=350, width=800, margin=dict(l=50, r=50, t=50, b=50))
fig.show()
```



As k gets close to 2 it is seen that the difference between the simulated and the analytic variance increases. The analytic variance of the Pareto distribution is undefined for $k \leq 2$, and tends towards ∞ as k approaches 2. The simulated variance is calculated as the sample variance of the generated random numbers, and is therefore always finite. This could explain the increased difference between the simulated and the analytic variance as k approaches 2.

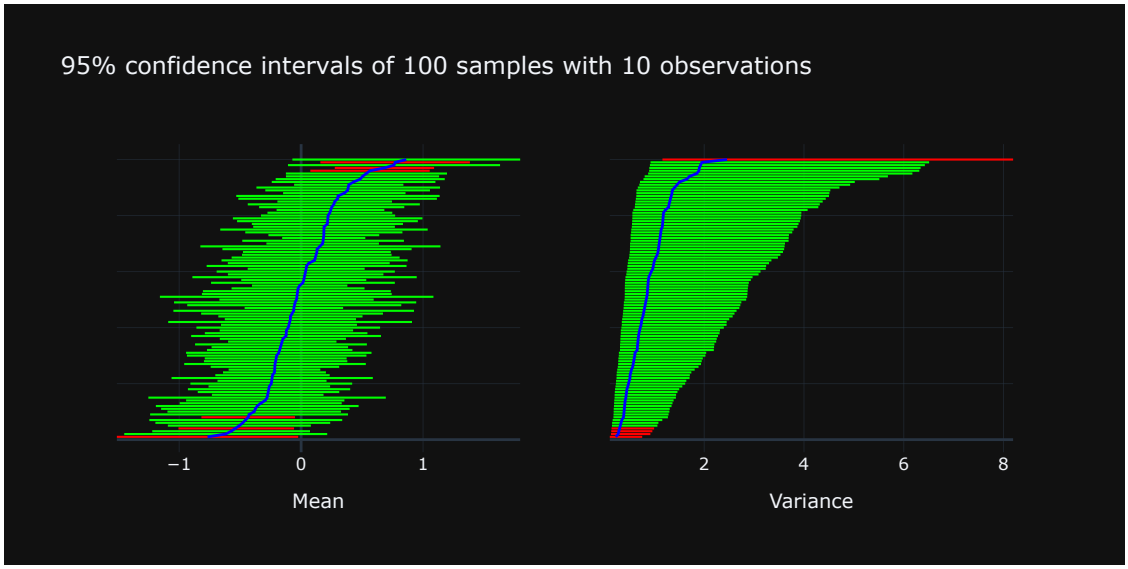
Part 3 - Confidence intervals of normal distribution

For this we generate 10 samples from the normal distribution and calculate a 95% confidence interval for the mean and the variance. We repeat that 100 times and plot the confidence intervals.

In [ ]:
```
alpha = 0.05
n_samples = 100
N = 10

# Generate samples
U = np.random.rand(n_samples, N)
X = uniform_2_normal(U)

# Compute confidence intervals
means = X.mean(axis=1)
stds = X.std(axis=1)
CI_means = means[:, None] + t.ppf([alpha/2, 1-alpha/2], N-1)[None, :] * stds[:, None]/np.sqrt(N)
CI_vars = (N-1)*stds[:, None]**2 / chi2.ppf([1-alpha/2, alpha/2], N-1)[None, :]
```

In [ ]:
```
# Sort samples
sort_idx = np.argsort(means)
means = means[sort_idx]
CI_means = CI_means[sort_idx]

sort_idx = np.argsort(stds)
stds = stds[sort_idx]
CI_vars = CI_vars[sort_idx]

# Count fraction of confidence intervals that contain the true value
mean_fraction = np.mean((CI_means[:, 0] <= 0) & (CI_means[:, 1] >= 0))
var_fraction = np.mean((CI_vars[:, 0] <= 1) & (CI_vars[:, 1] >= 1))
print(f"The fraction of CIs containing the true mean: {mean_fraction:.2f}")
print(f"The fraction of CIs containing the true variance: {var_fraction:.2f}")

# Plot
fig = sp.make_subplots(rows=1, cols=2)

# Mean
for i, CI in enumerate(CI_means):
    color = 'lime' if CI[0] <= 0 and CI[1] >= 0 else 'red'
    fig.add_trace(go.Scatter(x=CI, y=2*[(i+1)/n_samples], mode='lines', line=dict(color=color, width=1.5)), row=1, col=1)
fig.add_trace(go.Scatter(x=means, y=np.linspace(1, n_samples, n_samples, endpoint=True)/n_samples, mode='lines', line=dict(color='b

# Variance
for i, CI in enumerate(CI_vars):
    color = 'lime' if CI[0] <= 1 and CI[1] >= 1 else 'red'
    fig.add_trace(go.Scatter(x=CI, y=2*[(i+1)/n_samples], mode='lines', line=dict(color=color, width=1.5)), row=1, col=2)
fig.add_trace(go.Scatter(x=stds**2, y=np.linspace(1, n_samples, n_samples, endpoint=True)/n_samples, mode='lines', line=dict(color=

# Update the layout
fig.update_layout(height=400, width=800, title_text=f"95% confidence intervals of 100 samples with 10 observations", showlegend=Fals
fig.update_xaxes(title_text="Mean", row=1, col=1); fig.update_xaxes(title_text="Variance", row=1, col=2)
fig.update_yaxes(showticklabels=False, row=1, col=1); fig.update_yaxes(showticklabels=False, row=1, col=2)

fig.show()
```

```
The fraction of CIs containing the true mean: 0.94
The fraction of CIs containing the true variance: 0.95
```

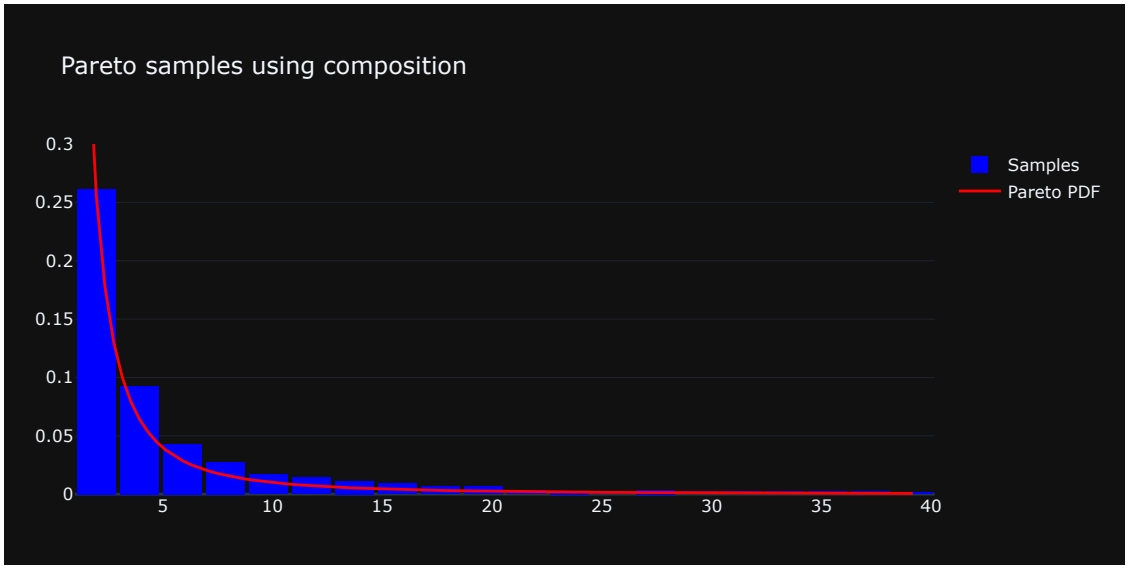


The confidence intervals for the mean and variance of the samples are plotted above. The green lines represent confidence intervals that contains the true mean and variance, while the red lines represent the opposite. It is seen that the fraction of confidence intervals that contain the true mean and variance is close to 0.95, indicating that the confidence intervals are robust.

Part 4 - Pareto distribution using composition

Now we wish to use composition to generate samples from the Pareto distribution. We do this by first generating samples, λ, from an exponential distribution. We then use these samples as the λ coefficient in the exponential distribution to generate samples from the Pareto distribution.

In [ ]:
```
num_samples = 10000

# Generate samples from uniform distribution
U1 = np.random.rand(num_samples)
U2 = np.random.rand(num_samples)

# Transform to Pareto distribution using composition method
beta = 1
y_comp = uniform_2_exponential(U1, beta)
X_comp = uniform_2_exponential(U2, y_comp)

# Plot
X_comp_q975 = X_comp[X_comp < np.quantile(X_comp, 0.975)] # Exclude extreme values for plotting purposes
x_pareto = np.linspace(min(X_comp_q975), max(X_comp_q975), 100, endpoint=True) # PDF points
fig = go.Figure(go.Histogram(x=X_comp_q975, xbins=dict(start=1, size=(max(X_comp_q975)-min(X_comp_q975))/20), histnorm='probability
fig.add_trace(go.Scatter(x=x_pareto, y=pareto_pdf(x_pareto, 1, beta), mode='lines', name='Pareto PDF', line=dict(color='red')))
fig.update_layout(height=400, width=800, title_text="Pareto samples using composition", bargap=0.1, margin=dict(l=50, r=50, t=100,
fig.update_xaxes(range=[1, None]); fig.update_yaxes(range=[0, 0.3])
fig.show()
```



In the plot above we see the samples generated using composition compared with the pdf of the Pareto distribution. We see that the samples generated using composition are consistent with the theoretical pdf.

Note that the shown histogram is truncated at the 97.5 percentile of the Pareto distribution. This is because the Pareto distribution has a long tail, and the histogram would be difficult to interpret if we included the entire range of the distribution.

Computer Exercise 4: Discrete Event Simulation

In [ ]:
```
# Plotting
import plotly.graph_objects as go
import plotly.express as px
import plotly.subplots as sp
import plotly.io as pio
pio.renderers.default = "notebook+pdf"
pio.templates.default = "plotly_dark"

# Utilities
import numpy as np
import pandas as pd
import math
from scipy.stats import t
```

Part 1 - Poisson Arrivals

Function for Discrete Event Simulation

In [ ]:
```
def blocking_system_simulation(
        num_service_units,
        num_customers,
        arrival_sample_fun,
        service_time_sample_fun,
        num_samples=None
    ):


    def single_sample():
        # Initialize the state of the system
        service_units_occupied = np.zeros(num_service_units)
        blocked_customers = 0

        # Main loop
        for _ in range(num_customers):

            # Sample arrival of a new customer
            arrival = arrival_sample_fun()

            # Update the state of the system
            service_units_occupied = np.maximum(0, service_units_occupied - arrival)

            # Check if a service unit is available
            if any(service_units_occupied == 0):
                # Sample the service time and assign the customer to the first available service unit
                service_time = service_time_sample_fun()
                service_unit = np.argmin(service_units_occupied)
                service_units_occupied[service_unit] = service_time
            else:
                # Block the customer
                blocked_customers += 1

        return blocked_customers/num_customers

    if num_samples is None:
        return single_sample()
    else:
        return np.array([single_sample() for _ in range(num_samples)])


def simulation_stats(theta_hats, alpha, verbose=False):
    n = len(theta_hats)
    theta_bar = np.mean(theta_hats)
    S = np.sqrt((np.sum(theta_hats**2) - n*theta_bar**2)/(n-1))
    CI = theta_bar + S/np.sqrt(n) * t.ppf([alpha/2, 1-alpha/2], n-1)

    if verbose:
        print(f"Simulated blocking probability: {np.round(theta_bar, 4)}")
        print(f"Standard deviation: {np.round(S, 4)}")
        print(f"{(1-alpha)*100}% confidence interval: {np.round(CI, 4)}")

    return np.array([theta_bar, theta_bar-CI[0], CI[1]-theta_bar])


def analytic_block_prob(lam, s, m):
```

```
    A = lam * s
    return A**m/math.factorial(m)/np.sum([A**i/math.factorial(i) for i in range(m+1)])
```

Run Simulation

In [ ]:
```
num_samples = 10
num_customers = 10000
m = 10
mean_service_time = 8
arrival_intensity = 1
alpha = 0.05

# Samplers
arrival_time_sampler = lambda: np.random.exponential(arrival_intensity)
service_time_sampler = lambda: np.random.exponential(mean_service_time)

# Compute the analytical blocking probability
B = analytic_block_prob(arrival_intensity, mean_service_time, m)
print(f"Analytical blocking probability: {np.round(B, 4)}")

# Simulate the blocking probability
theta_hats = blocking_system_simulation(m, num_customers, arrival_time_sampler, service_time_sampler, num_samples)
stats_exponential = simulation_stats(theta_hats, alpha, verbose=True)
```

```
Analytical blocking probability: 0.1217
Simulated blocking probability: 0.1202
Standard deviation: 0.004
95.0% confidence interval: [0.1173 0.1231]
```

Part 2 - Renewal Arrivals

We will now consider renewal arrivals.

(a) Erlang Inter Arrival Times

In [ ]:
```
# Erlang sampler
k = 1
arrival_time_sampler = lambda: np.random.gamma(k, 1/arrival_intensity)

# Simulate the blocking probability
theta_hats = blocking_system_simulation(m, num_customers, arrival_time_sampler, service_time_sampler, num_samples)
stats_erlang = simulation_stats(theta_hats, alpha, verbose=True)
```

```
Simulated blocking probability: 0.1194
Standard deviation: 0.0059
95.0% confidence interval: [0.1152 0.1236]
```

(b) Hyper Exponential Arrival Times

In [ ]:
```
# Hyperexponential sampler
p1 = 0.8; p2 = 0.2
lam1 = 0.8333; lam2 = 5.0

arrival_time_sampler = lambda: np.random.choice([np.random.exponential(1/lam1), np.random.exponential(1/lam2)], p=[p1, p2])

# Simulate the blocking probability
theta_hats = blocking_system_simulation(m, num_customers, arrival_time_sampler, service_time_sampler, num_samples)
stats_hyperexponential = simulation_stats(theta_hats, alpha, verbose=True)
```

```
Simulated blocking probability: 0.1406
Standard deviation: 0.0061
95.0% confidence interval: [0.1362 0.1449]
```

In [ ]:
```
# Collect results from varying arrival times
results_vary_arrival_times = np.array([stats_exponential, stats_erlang, stats_hyperexponential])
df_vary_arrival_times = pd.DataFrame(results_vary_arrival_times,
    columns=["Blocking Probability", "Lower Bound", "Upper Bound"])
df_vary_arrival_times["Distribution"] = ["Exponential", "Erlang", "Hyperexponential"]
```

Part 3 - Service Time Distribution

In [ ]:
```
arrival_time_sampler = lambda: np.random.exponential(1/arrival_intensity)
```

(a) Constant Service Time

In [ ]:
```
# Constant "sampler"
service_time_sampler = lambda: mean_service_time

# Simulate the blocking probability
theta_hats = blocking_system_simulation(m, num_customers, arrival_time_sampler, service_time_sampler, num_samples)
stats_const = simulation_stats(theta_hats, alpha, verbose=True)
```

```
Simulated blocking probability: 0.1206
Standard deviation: 0.0049
95.0% confidence interval: [0.1171 0.1241]
```

(b) Pareto Service Time

In [ ]:
```
# Pareto sampler

# k = 1.05
service_time_sampler = lambda: np.random.pareto(1.05) # Pareto with beta=1
theta_hats = blocking_system_simulation(m, num_customers, arrival_time_sampler, service_time_sampler, num_samples)
stats_pareto105 = simulation_stats(theta_hats, alpha)

# k = 2.05
service_time_sampler = lambda: np.random.pareto(2.05) # Pareto with beta=1
theta_hats = blocking_system_simulation(m, num_customers, arrival_time_sampler, service_time_sampler, num_samples)
stats_pareto205 = simulation_stats(theta_hats, alpha)
```

As k increases, the Pareto distribution skews more to the right, resulting in decreasing service times which in turn allows for more customers to be served.

(c) Half-Cauchy and Chi-Square Service Time

In [ ]:
```
# Half-Cauchy sampler
service_time_sampler = lambda: np.abs(np.random.standard_cauchy())

# Simulate the blocking probability
theta_hats = blocking_system_simulation(m, num_customers, arrival_time_sampler, service_time_sampler, num_samples)
stats_cauchy = simulation_stats(theta_hats, alpha, verbose=True)
```

```
Simulated blocking probability: 0.0339
Standard deviation: 0.023
95.0% confidence interval: [0.0175 0.0504]
```

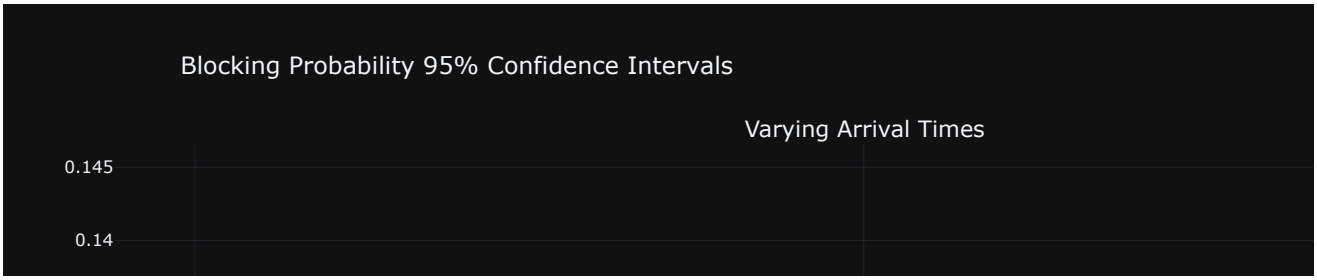
In [ ]:
```
# Chi-Square sampler
df = 8
service_time_sampler = lambda: np.random.chisquare(df)

# Simulate the blocking probability
theta_hats = blocking_system_simulation(m, num_customers, arrival_time_sampler, service_time_sampler, num_samples)
stats_chi2 = simulation_stats(theta_hats, alpha, verbose=True)
```

```
Simulated blocking probability: 0.1184
Standard deviation: 0.0046
95.0% confidence interval: [0.115  0.1217]
```

In [ ]:
```
# Collect results for varying service times
results_vary_service_times = np.array([stats_const, stats_pareto105, stats_pareto205, stats_cauchy, stats_chi2])
df_vary_service_times = pd.DataFrame(results_vary_service_times,
    columns=["Blocking Probability", "Lower Bound", "Upper Bound"])
df_vary_service_times["Distribution"] = ["Constant", "Pareto (k=1.05)", "Pareto (k=2.05)", "Half-Cauchy", "Chi-Square (df=8)"]
```

Part 4 - Confidence Intervals

In [ ]:
```
fig = sp.make_subplots(rows=1, cols=2, subplot_titles=["Varying Arrival Times", "Varying Service Times"])
scatter1 = px.scatter(df_vary_arrival_times, x="Distribution", y="Blocking Probability", error_y="Upper Bound", error_y_minus="Lowe
scatter2 = px.scatter(df_vary_service_times, x="Distribution", y="Blocking Probability", error_y="Upper Bound", error_y_minus="Lowe
fig.add_trace(scatter1.data[0], row=1, col=1)
fig.add_trace(scatter2.data[0], row=1, col=2)
fig.update_layout(title="Blocking Probability 95% Confidence Intervals")
fig.show()
```



When varying the arrival time distribution, we observe that the confidence intervals for the blocking probability have similar width. This consistency suggests that the uncertainty associated with the blocking probability remains unaffected by whether the arrivals are more dispersed or concentrated. However, it is important to note that the blocking probability itself is influenced by the arrival time distribution.

When varying the service time distribution, we observe that the confidence intervals for the blocking probability have different widths. This variation indicates that the uncertainty associated with the blocking probability is influenced by the characteristics of the service time distribution. Additionally, the blocking probability itself also seems to be affected by changes in the service time distribution.

Computer Exercise 5: Variance Reduction Methods

In [ ]:
```
# Plotting
import plotly.graph_objects as go
import plotly.express as px
import plotly.subplots as sp
import plotly.io as pio
pio.renderers.default = "notebook+pdf"
pio.templates.default = "plotly_dark"

# Utilities
import numpy as np
from scipy.optimize import root
from utils import describe_sample
```

In this exercise we will approximate the value of the integral $\int_0^1 \mathrm{e}^x$ with different sampling methods and compare their efficiency. To compare the true value of the integral is:

$$\int_0^1 \mathrm{e}^x \, dx = \mathrm{e} - 1 \approx 1.718281828459045$$

Part 1 - Integral Estimation by Crude Monte Carlo

Using the Crude Monte Carlo we method we exploit the fact that the integral of a function $f(x)$ over a domain $[a, b]$ can be estimated by the average of the function values at N random points x_i in the domain $[a, b]$. So we sample points from the uniform distribution between 0 and 1, tranforms those points using the exponential function and then calculate the average of the function values at those points.

In [ ]:
```
num_samples = 100

# Crude Monte Carlo
U = np.random.uniform(0, 1, num_samples)
X = np.exp(U)

describe_sample(X, title="Method: Crude Monte Carlo")
```

```
────────────────────────────────────────
        >>> SAMPLE STATISTICS <<<
        Method: Crude Monte Carlo
────────────────────────────────────────
Sample size: 100
Mean: 1.7075
Standard deviation: 0.5281
95% confidence interval: [1.6027 1.8123]
────────────────────────────────────────
```

Above we have reported the mean value of our samples, which represents the approximated value of the integral. Along with it we have reported the standard deviation and confidence interval of the mean value. For now we can see that the approximated value of the integral is a bit lower than the actual value. As we compute the integral with more methods, the tigthness of the confidence interval will be a good indicator of the accuracy of the method.

Part 2 - Integral Estimation by Antithetic Variables

To estimate the integral by using antithetic variables we transform the uniform samples with the following formula:

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

In [ ]:
```
Y = (np.exp(U) + np.exp(1 - U)) / 2

describe_sample(Y, title="Method: Antithetic Variates")
```

```
────────────────────────────────────────
        >>> SAMPLE STATISTICS <<<
       Method: Antithetic Variates
────────────────────────────────────────
Sample size: 100
Mean: 1.7310
Standard deviation: 0.0663
95% confidence interval: [1.7178 1.7441]
────────────────────────────────────────
```

With this method we see that the approximation of the integral is closer to the true value of the integral. We also see that the standard deviation has almost dropped with a factor 10. So using antithetic variables has improved the accuracy of the estimation compared to the crude Monte Carlo Method.

Analytically, as seen on the slides, there should be a variance reduction of 98% between these first two methods, however we only saw a variance reduction of 88%.

Part 3 - Integral Estimation by Control Variable

For the next method, we consider to sets of samples. U which is our uniformly distributed samples, and $X = e^U$.

To estimate the integral by using control variables we use the following formula:

$$Z_i = X + c\,(U - \mu_U)$$

where $c = \frac{-\mathrm{Cov}(X,U)}{\mathrm{Var}(U)}$ and $\mu_U = \frac{1}{2}$ is the mean of the uniform samples. We then use Z to estimate the integral.

In [ ]:
```
c = -np.cov(X,U)[0][1]/np.var(U)
Z = X + c*(U - 0.5)

describe_sample(Z, title="Method: Control Variates")
```

```
--------------------------------------------
        >>> SAMPLE STATISTICS <<<
        Method: Control Variates
--------------------------------------------
Sample size: 100
Mean: 1.7300
Standard deviation: 0.0656
95% confidence interval: [1.717  1.7431]
--------------------------------------------
```

With this method we see that the approximation of the integral is slightly closer to the true value of the integral. The standard deviation has the same size, so we do not get a more confident estimate of the integral compared to the antithetic variables method.

Part 4 - Integral Estimation by Stratified Sampling

With stratified sampling, we need to generate new numbers from the uniform distribution, where until now we have been able to use the same samples to approximate the integral. We generate a matrix of 10×100 uniform numbers. To do stratified sampling we then use the following formula:

$$W_i = \frac{1}{10}\sum_{j=1}^{10} e^{\frac{U_{ij}}{10}} e^{\frac{(j-1)}{10}}$$

So from our matrix we generate 100 stratified samples by using 10 uniform samples for each.

In [ ]:
```
strata_number = 10

U2 = np.random.uniform(0, 1, size=(num_samples, strata_number))
W = np.mean(np.exp((U2 + np.arange(strata_number))/strata_number), axis=1)

describe_sample(W, title="Method: Stratified Sampling")
```

```
--------------------------------------------
        >>> SAMPLE STATISTICS <<<
       Method: Stratified Sampling
--------------------------------------------
Sample size: 100
Mean: 1.7201
Standard deviation: 0.0166
95% confidence interval: [1.7168 1.7234]
--------------------------------------------
```

With stratified sampling we see that the approximation of the integral is closer to the true value of the integral. The standard deviation has also dropped with a factor 5, so we get a more confident estimate of the integral compared to the control variable method.

Part 5 - Variance Reduction in Blocking System with Control Variates

Using control variates, we wish to see if we can reduce the variance of the estimator from the Poisson arrival process considered in computer exercise 4.

In [ ]:
```python
def BS_control(
        arrival_times,
        service_times,
        control_variate,
        control_mu,
        num_service_units=10,
    ):

    num_samples, num_customers = arrival_times.shape

    # Sample loop
    out = np.zeros(num_samples)
    for i in range(num_samples):
        # Initialize the state of the system
        service_units_occupied = np.zeros(num_service_units)
        blocked_customers = np.zeros(num_customers)

        # Main loop
        for j in range(num_customers):

            # Update the state of the system
            service_units_occupied = np.maximum(0, service_units_occupied - arrival_times[i, j])

            # Check if a service unit is available
            if any(service_units_occupied == 0):
                # Sample the service time and assign the customer to the first available service unit
                service_unit = np.argmin(service_units_occupied)
                service_units_occupied[service_unit] = service_times[i, j]
            else:
                # Block the customer
                blocked_customers[j] = 1

        # Control variates
        X = blocked_customers
        Y = control_variate[i]
        mu_y = control_mu

        c = -np.cov(X,Y)[0][1]/np.var(Y)
        Z = X + c*(Y - mu_y)

        out[i] = np.mean(Z)

    return out
```

In [ ]:
```python
# Parameters
num_samples = 10
num_customers = 10000
m = 10
mean_service_time = 8
arrival_intensity = 1
```

In [ ]:
```python
# Generate the arrival and service times
A = np.random.exponential(arrival_intensity, (num_samples, num_customers))
S = np.random.exponential(mean_service_time, (num_samples, num_customers))

# Simulate the blocking probability
theta_hats = BS_control(A, S, A, arrival_intensity)
describe_sample(theta_hats, title="Blocking Probability: Control Variates")
```

```
--------------------------------------------------
       >>> SAMPLE STATISTICS <<<
 Blocking Probability: Control Variates
--------------------------------------------------
Sample size: 10
Mean: 0.1169
Standard deviation: 0.0062
95% confidence interval: [0.1124 0.1213]
--------------------------------------------------
```

Part 6 - Effect of Using Common Random Variates

In [ ]:
```python
# Simulation Parameters
num_samples = 100
num_customers = 10000
m = 10
mean_service_time = 8

# Possion parameters
arrival_intensity = 1
```

```
# Hyperexponential parameters (from exercise 4) mean=1
p1 = 0.8; p2 = 0.2
lam1 = 0.8333; lam2 = 5.0
```

Using Distinct Random Variates

In [ ]:
```
# Generate distinct random variates for arrival and service times
A_poisson = np.random.exponential(arrival_intensity, (num_samples, num_customers))
A_hyper = np.random.choice([np.random.exponential(1/lam1), np.random.exponential(1/lam2)], p=[p1, p2], size=(num_samples, num_custo
S_poisson = np.random.exponential(mean_service_time, (num_samples, num_customers))
S_hyper = np.random.exponential(mean_service_time, (num_samples, num_customers))

# Simulate the difference in blocking probability
theta_hats_poisson = BS_control(A_poisson, S_poisson, A_poisson, arrival_intensity)
theta_hats_hyperexponential = BS_control(A_hyper, S_hyper, A_hyper, arrival_intensity)
delta_theta_hats = theta_hats_poisson - theta_hats_hyperexponential
describe_sample(delta_theta_hats, title="Distinct Variates")
```

```
--------------------------------------------
        >>> SAMPLE STATISTICS <<<
            Distinct Variates
--------------------------------------------
Sample size: 100
Mean: 0.1180
Standard deviation: 0.0053
95% confidence interval: [0.1169 0.1191]
--------------------------------------------
```

Using Common Random Variates

In [ ]:
```
# Generate common random variates for arrival and service times
U1 = np.random.uniform(0, 1, (num_samples, num_customers))
U2 = np.random.uniform(0, 1, (num_samples, num_customers))
A_poisson = -np.log(U1)/arrival_intensity
A_hyper = np.where(U2 < p1, -np.log(U1)/lam1, -np.log(U1)/lam2)
S = np.random.exponential(mean_service_time, (num_samples, num_customers))

# Simulate the difference in blocking probability
theta_hats_poisson = BS_control(A_poisson, S, U1, 0.5)
theta_hats_hyperexponential = BS_control(A_hyper, S, U1, 0.5)
delta_theta_hats = theta_hats_poisson - theta_hats_hyperexponential
describe_sample(delta_theta_hats, title="Common Variates")
```

```
--------------------------------------------
        >>> SAMPLE STATISTICS <<<
             Common Variates
--------------------------------------------
Sample size: 100
Mean: -0.0171
Standard deviation: 0.0037
95% confidence interval: [-0.0179 -0.0164]
--------------------------------------------
```

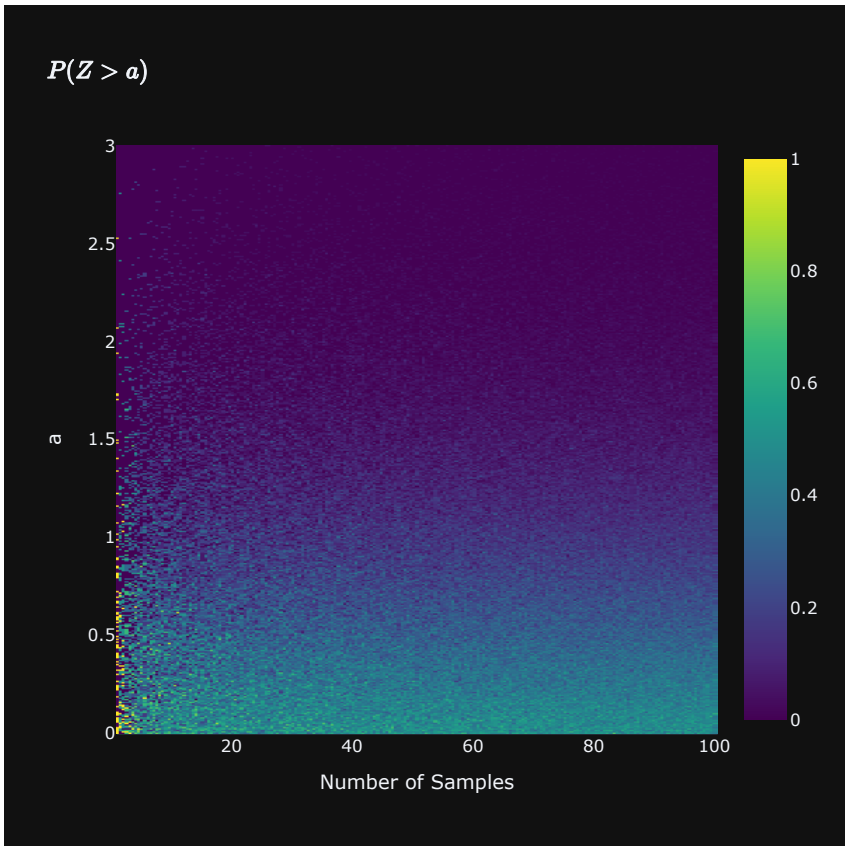
Part 7 - Probability Estimation

In this part we will revisit the crude Monte Carlo method to estimate the probability that $Z > a$ where $Z \sim \mathcal{N}(0, 1)$. We will do this by generating a set of standard normal random variables and then estimate the probability that $Z > a$ by directly considering only the samples from Z where $Z_i > a$.

In [ ]:
```
sigma=1
num_samples=1000
a = 2
Z = np.random.normal(0, sigma, num_samples)
I = Z > a

describe_sample(I, title="Crude Monte Carlo")

as_ = np.linspace(0, 3, 1000)
N = np.linspace(1, 100, 1000, dtype=int)

p = np.zeros((len(as_), len(N)))
for i, a in enumerate(as_):
    for j, n in enumerate(N):
        Z = np.random.normal(0, sigma, n)
        p[i,j] = np.mean(Z > a)

# Plot heat map
fig = go.Figure(data=go.Heatmap(z=p, x=N, y=as_, colorscale='Viridis'))
```

```
fig.update_layout(title=r"Crude Monte Carlo: Estimated $P(Z>a)$", xaxis_title="Number of Samples", yaxis_title="a", width=600, heigl
fig.show()
```

```
──────────────────────────────────────────
        >>> SAMPLE STATISTICS <<<
            Crude Monte Carlo
──────────────────────────────────────────
Sample size: 1000
Mean: 0.0180
Standard deviation: 0.1330
95% confidence interval: [0.0097 0.0263]
──────────────────────────────────────────
```



First we estimated the probability of $Z > a$ with $a = 2$ and 1000 samples. Here we find that $p(Z > a) \approx 0.0250$. The standard deviation of this result is 0.1526 which is quite high. We can see that the confidence interval is quite wide, so we are not very confident in our estimate.

Part 8 - Integral Estimation by Importance Sampling

In this part we will go back to considering a new method for approximating the integral $\int_0^1 \mathrm{e}^x$. This time we will use importance sampling. In importance sampling we start by using samples from the uniform distribution U, we then use the inversion method to generate samples from the exponential distribution $Y = \frac{-\log(U)}{\lambda}$. The importance sampling estimator is then given by:

$$\theta = \mathbb{E}\left(\frac{h(Y)f(Y)}{g(y)}\right)$$

As given by the exercise, we set $g(y) = \lambda \mathrm{e}^{-\lambda y}$. We set $h(y) = \mathrm{e}^y$ as this is the integral we wish to approximate, and $f(y) = \mathbf{1}_{[0,1]}(y)$ as we are only interested in the integral over the interval $[0, 1]$.

Now the final step before approximating θ is to find out what value of λ to use. We will do this by deriving λ from the condition that the variance of the estimator is minimized.

Determining variance minimizing λ by analytical derivation

Computing the analytic variance $\mathbb{V}\left[\frac{h(X)f(X)}{g(X)}\right]$. The functions $f(X)$, $g(X)$, and $h(X)$ are given as,

$$g(X) = \lambda e^{-\lambda X}, \quad f(X) = \mathbf{1}_{0 \le X \le 1}, \quad h(X) = e^X.$$

Using the definition of variance for a random variable,

$$\mathbb{V}\left[\frac{h(X)f(X)}{g(X)}\right] = \mathbb{E}\left[\left(\frac{h(X)f(X)}{g(X)}\right)^2\right] - \left(\mathbb{E}\left[\frac{h(X)f(X)}{g(X)}\right]\right)^2.$$

These expectations can be computed as follows,

$$\mathbb{E}\left[\frac{h(X)f(X)}{g(X)}\right] = \int_0^1 \frac{h(X)f(X)}{g(X)} g(X)dX = \int_0^1 h(X)dX = \int_0^1 e^X dX = e - 1$$

$$\mathbb{E}\left[\left(\frac{h(X)f(X)}{g(X)}\right)^2\right] = \int_0^1 \left(\frac{h(X)f(X)}{g(X)}\right)^2 g(X)dX = \int_0^1 \frac{h(X)^2}{g(X)}dX = \int_0^1 \frac{e^X}{\lambda e^{-\lambda X}}dX = \frac{1}{\lambda}\int_0^1 e^{(1+\lambda)X}dX = \frac{e^{1+\lambda}-1}{\lambda(1+\lambda)}.$$

Inserting into the variance formula,

$$\mathbb{V}\left[\frac{h(X)f(X)}{g(X)}\right] = \frac{e^{1+\lambda}-1}{\lambda(1+\lambda)} - (e-1)^2.$$

Differentiating the variance with respect to λ, setting it equal to zero and solving for λ yields,

$$\frac{d}{d\lambda}\left(\mathbb{V}\left[\frac{h(X)f(X)}{g(X)}\right]\right) = \frac{(\lambda^2-2)e^{2+\lambda}+2(\lambda+1)}{\lambda^2(2+\lambda)^2} = 0 \quad \Rightarrow \quad \lambda^* = 1.3548.$$

Note: The second derivative of the variance with respect to λ evaluated at λ^* is positive, thus λ^* is a minimizer.

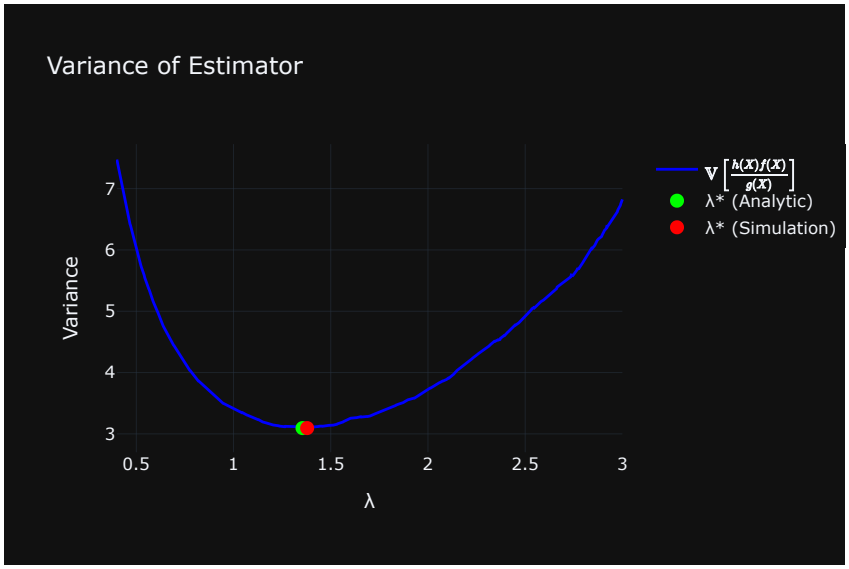
In [ ]:
```
lam_analytic = root(lambda λ: ((λ**2 - 2)*np.exp(2 + λ) + 2 * (λ + 1)) / (λ**2*(2 + λ)**2), 1).x[0]
print(f"λ* = {lam_analytic:.4f}")
```

```
λ* = 1.3548
```

Determining variance minimizing λ by simulation

Next we also determine the variance minimizing λ by simulation. We do this by computing the variance of the estimator for different values of λ and then find the value of λ that minimizes the variance.

In [ ]:
```
# Define functions
g = lambda x, λ: λ*np.exp(-λ*x)
f = lambda x: (x > 0)*(x < 1)
h = lambda x: np.exp(x)

# Sample
num_samples = 100000
U = np.random.uniform(0, 1, num_samples)

# Find the optimal lambda by simulation
lambdas = np.linspace(0.4, 3, 1000)
variances = np.zeros_like(lambdas)
for i, λ in enumerate(lambdas):
    Y = -np.log(U)/λ
    theta = h(Y)*f(Y)/g(Y, λ)
    variances[i] = np.var(theta)
lam_sim = lambdas[np.argmin(variances)]

# Plot the variance as a function of lambda
fig = go.Figure(data=go.Scatter(x=lambdas, y=variances, marker=dict(color='Blue'), name=r"$\mathbb{V}\left[\frac{h(X)f(X)}{g(X)}\rig
fig.add_trace(go.Scatter(x=[lam_analytic], y=[np.min(variances)], mode='markers', marker=dict(size=10, color='Lime'), name="λ* (Ana
fig.add_trace(go.Scatter(x=[lam_sim], y=[np.min(variances)], mode='markers', marker=dict(size=10, color='Red'), name="λ* (Simulatio
fig.update_layout(title="Variance of Estimator", xaxis_title="λ", yaxis_title="Variance", width=600, height=400)
fig.show()
```



Get sample statistics

Now we compare the estimates of θ using the two different λ values.

In [ ]:
```
describe_sample(h(Y)*f(Y)/g(Y, lam_sim), title=f"IS: λ={lam_sim:.2f} (Simulated)")
describe_sample(h(Y)*f(Y)/g(Y, lam_analytic), title=f"IS: λ={lam_analytic:.2f} (Analytic)")
```

```
──────────────────────────────────────────────────
        >>> SAMPLE STATISTICS <<<
         IS: λ=1.38 (Simulated)
──────────────────────────────────────────────────
Sample size: 100000
Mean: 1.6142
Standard deviation: 1.2903
95% confidence interval: [1.6062 1.6222]
──────────────────────────────────────────────────
──────────────────────────────────────────────────
        >>> SAMPLE STATISTICS <<<
          IS: λ=1.35 (Analytic)
──────────────────────────────────────────────────
Sample size: 100000
Mean: 1.6253
Standard deviation: 1.2840
95% confidence interval: [1.6173 1.6332]
──────────────────────────────────────────────────
```

From the two approximations of the integral using importance sampling, we see that the estimated value is a bit lower than the true value. We also see that the standard deviation is quite high compared to previous results. So the importance sampling method does not look promising with the given distribution.

Part 9 - Importance Sampling Estimator for Pareto Distribution

The first moment distribution of the Pareto distribution is given in lecture as,

$$g(x) = G_1(x) = 1 - \left(\frac{x}{\beta}\right)^{1-k}.$$

Using the same $f(x)$ and $h(x)$ as in the previous part,

$$f(x) = \mathbf{1}_{0 \le x \le 1}, \quad h(x) = e^x,$$

The importance sampling estimator is given by,

$$\theta = \mathbb{E}\left[\frac{h(Y)f(Y)}{g(Y)}\right] = \mathbb{E}\left[\frac{e^Y}{1-\left(\frac{Y}{\beta}\right)^{1-k}}\right],$$

where $Y \sim Pa(\beta, k)$. For samples close to β this becomes numerically unstable.

Computer Exercise 6: Markov Chain Monte Carlo

In [ ]:
```python
# Plotting
import plotly.graph_objects as go
import plotly.express as px
import plotly.subplots as sp
import plotly.io as pio
pio.renderers.default = "notebook+pdf"
pio.templates.default = "plotly_dark"

# Utilities
import numpy as np
import pandas as pd
from scipy.special import factorial
from scipy.stats import chi2, poisson, norm, multivariate_normal
```

In [ ]:
```python
def chisquare_test(n_obs, n_exp, df, title=None):

    # Compute the test statistic and p-value
    Z = np.sum(np.divide((n_obs - n_exp)**2, n_exp, where=n_exp!=0))
    p = 1 - chi2.cdf(Z, df)

    print("-"*30)
    print("   >>> Chi-Squared Test <<<   ")
    if title:
        pad = " "*int(max(0, ((30-len(title))/2)))
        print(pad + title + pad)
    print("-"*30)
    print(f"Degrees of Freedom: {int(df)}")
    print(f"Test Statistic: {Z:.4f}")
    print(f"p-value: {p:.4f}")
    print("-"*30)
```

Part 1 - Erlang System

In this exercise we wish to simulate the Erlang system using the Metropolis-Hastings algorithm. The probability of having i customers in the system is given by the Erlang distribution:

$$P(i) = c \cdot \frac{A^i}{i!} \quad \text{for} \quad i = 0, 1, 2, \ldots, m$$

where c is a normalization constant and A is the arrival rate. Below we implement the Metropolis-Hastings algorithm to simulate the Erlang system.

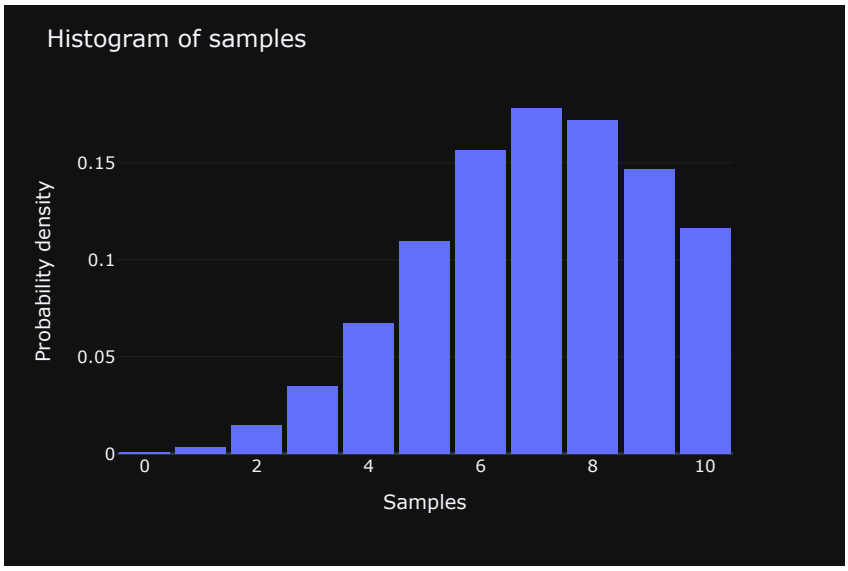
The Metropolis-Hastings algorithm works by taking an initial start guess, which in this example for the Erlang system in an integer between 0 and m. We then propose a new state by sampling a new integer between 0 and m. For the old state and the new state, we calculate their probabilities using the formula for the Erlang system. We then calculate the acceptance probability as the ratio of the new state probability to the old state probability. If the acceptance probability is greater than a random number between 0 and 1, we accept the new state. Otherwise, we keep the old state. We repeat this process for a number of iterations and then return the states we have visited.

In [ ]:
```python
def metropolis_hastings(g, proposal_sampler, n, x0):

    # Handle if x0 is a scalar
    x0 = np.array([x0]) if np.shape(x0) == () else x0

    X = np.zeros((n+1, len(x0)))
    X[0] = x0

    for i in range(n):
        # Propose a new state
        Y = proposal_sampler(X[i])
        Y = np.array([Y]) if np.shape(Y) == () else Y

        # Evaluate the target distribution
        g_Y = g(Y)
        g_X = g(X[i])

        # Accept or reject the new state
        if g_Y >= g_X:
            X[i+1] = Y
        else:
            accept = np.random.uniform(0,1) < g_Y/g_X
            X[i+1] = Y if accept else X[i]
```

```
    return X
```

In [ ]:
```
m = 10 # number of servers
s = 8 # mean service time
lam = 1
A = lam*s
x0 = np.random.randint(0,m+1)
num_samples = 10000
burn_in = 0

g = lambda x: A**x/(factorial(x))
proposal_sampler = lambda x: np.random.randint(0, m+1)

samples = metropolis_hastings(g, proposal_sampler, num_samples, x0)
```

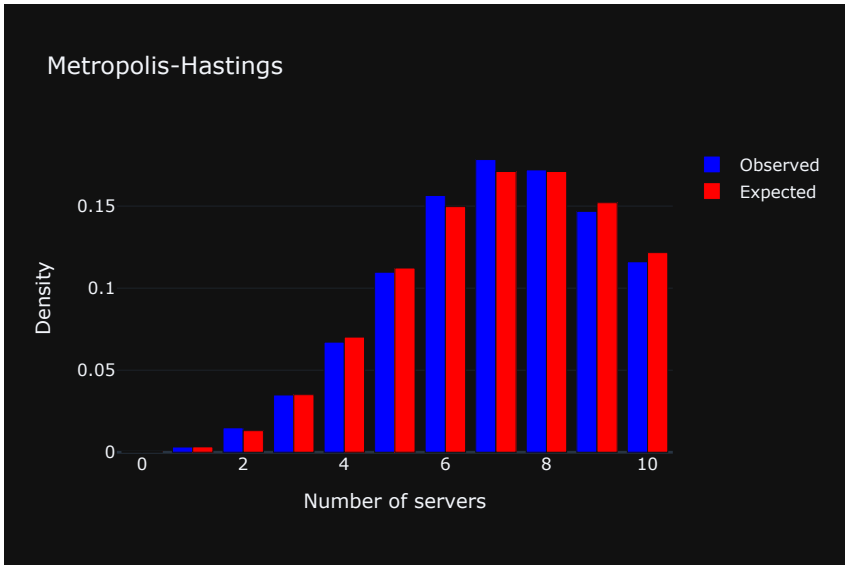
In [ ]:
```
# Make a histogram
fig = px.histogram(x=samples.flatten(), nbins=m+1, histnorm='probability density')
fig.update_layout(title='Histogram of samples', xaxis_title='Samples', yaxis_title='Probability density', bargap=0.1)
fig.update_layout(width=600, height=400)
fig.show()
```



We ran the Erlang system through the Metropolis-Hastings algorithm for 10000 iterations with the same parameters in computer exercise 4. In the histogram above we have shown the distribution of the number of customers in the system. The target distribution resembles a Poisson distribution with parameter $\lambda = A$.

In [ ]:
```
f_obs = np.bincount(samples[burn_in:].flatten().astype(int), minlength=m+1)
f_obs = f_obs/np.sum(f_obs)
f_exp = poisson.pmf(np.arange(m+1), A)
f_exp = f_exp/np.sum(f_exp)

# Perform chi-squared test
chisquare_test(f_obs, f_exp, m, title='Metropolis-Hastings')

# Make joint histogram of observed and expected values
fig = go.Figure()
fig.add_trace(go.Bar(x=np.arange(m+1), y=f_obs, name='Observed', marker_color='blue'))
fig.add_trace(go.Bar(x=np.arange(m+1), y=f_exp, name='Expected', marker_color='red'))
fig.update_layout(title='Metropolis-Hastings', xaxis_title='Number of servers', yaxis_title='Density')
fig.update_layout(width=600, height=400)
fig.show()
```

```
--------------------------------
   >>> Chi-Squared Test <<<
     Metropolis-Hastings
--------------------------------
Degrees of Freedom: 10
Test Statistic: 0.0015
p-value: 1.0000
--------------------------------
```



To further test if the Metropolis-Hastings algorithm works, we have performed a chi-squared test and plotted the observations along theoretical values for a poisson distribution.

The chi-squared test resulted in a p-value of 1, which means that we cannot reject the null hypothesis that the two distributions are the same. The two histograms also look very similar, which further supports the conclusion that the Metropolis-Hastings algorithm works well for this problem.

Part 2 - Two Different Call Types

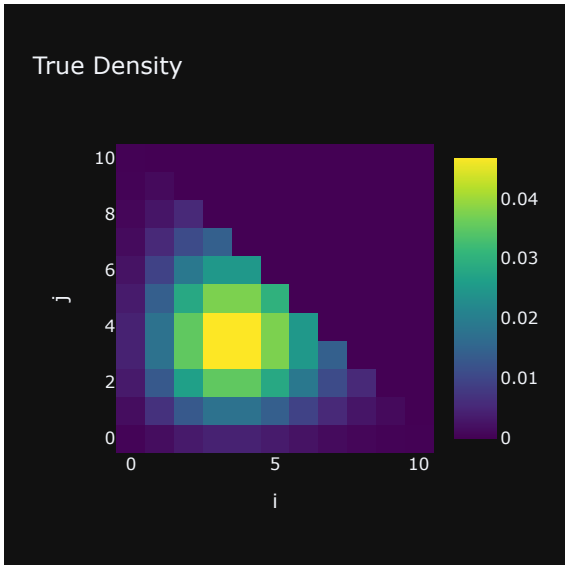
Now we expand the Erlang system to include two different call types. The probability of having i and j customers in the system is given by the Erlang distribution:

$$P(i,j) = c \cdot \frac{A_1^i}{i!} \cdot \frac{A_2^j}{j!} \quad \text{for} \quad 0 \leq i+j \leq m$$

First we setup the parameters and compute the expected values for the two call types.

In [ ]:
```
A1 = A2 = 4
m = 10

def bivariate_poisson_pmf(i, j, A, B):
    c = np.exp(-(A + B))
    return c * ((A**i / factorial(i)) * (B**j / factorial(j)))
```

In [ ]:
```
# Compute expected probability mass
f_exp = np.zeros((m+1, m+1))
for i in range(m+1):
    for j in range(m+1):
        if i+j <= m:
            f_exp[i,j] = bivariate_poisson_pmf(i, j, A1, A2)
f_exp = f_exp/np.sum(f_exp)

# Visualize expected probability mass
fig = go.Figure(data=go.Heatmap(z=f_exp, colorscale='Viridis'))
fig.update_layout(title='True Density', xaxis_title='i', yaxis_title='j', width=400, height=400)
fig.show()
```



This is the theoretical density for the two call types. This plot will be used later for comparing the theoretical values with the simulated values.

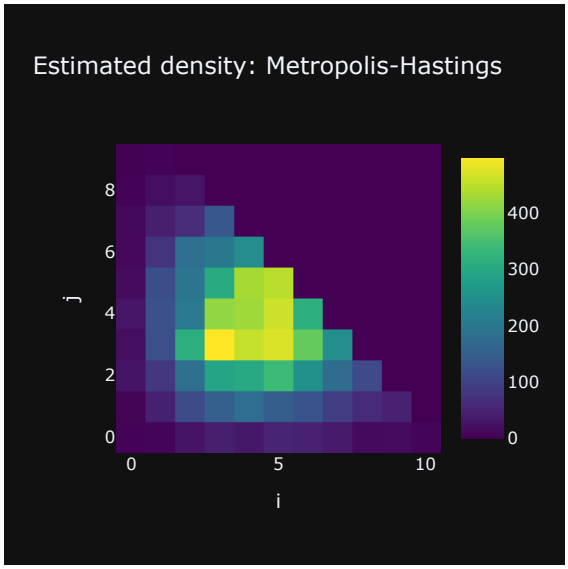
(a) Metropolis-Hastings

We ran the Erlang system with two different call types through the Metropolis-Hastings algorithm for 10000 iterations with the given parameters.

In [ ]:
```
num_samples = 10000
burn_in = 0
x0 = np.array([0, 0])

g = lambda x: (A1**x[0]/factorial(x[0])) * (A2**x[1]/factorial(x[1]))
def proposal_sampler(x):
    # Ensure that i+j <= m
    i = np.random.randint(0, m+1)
    j = np.random.randint(0, m+1-i)
    return np.array([i, j])

samples = metropolis_hastings(g, proposal_sampler, num_samples, x0)
```

In [ ]:
```
bin_edges = np.arange(m+2) - 0.5
f_obs, _, _ = np.histogram2d(samples[burn_in:,0].astype(int), samples[burn_in:,1].astype(int), bins=[bin_edges, bin_edges], density=

# Chi-square test
df = (m+1)*(m+2)/2
chisquare_test(f_obs, f_exp, df, title='Metropolis-Hastings')

# Make heatmap of observed values
fig = go.Figure(data=[go.Histogram2d(x=samples[:,0], y=samples[:,1], colorscale='Viridis')])
fig.update_layout(title='Estimated density: Metropolis-Hastings', xaxis_title='i', yaxis_title='j', width=400, height=400)
fig.show()
```

```
--------------------------------
   >>> Chi-Squared Test <<<
     Metropolis-Hastings
--------------------------------
Degrees of Freedom: 66
Test Statistic: 0.1067
p-value: 1.0000
--------------------------------
```



We have compared the theoretical distribution with the simulated distribution for the two call types. The two histograms look very similar, which indicates that the Metropolis-Hastings algorithm works well for this problem. We also calculated a p-value using a chi-squared test which resulted in a p-value of 1. So we cannot reject the null hypothesis that the two distributions are the same.

(b) Coordinate wise Metropolis-Hastings

Next we run the Erlang system with two different call types through the coordinate wise Metropolis-Hastings algorithm for 10000 iterations with the given parameters.

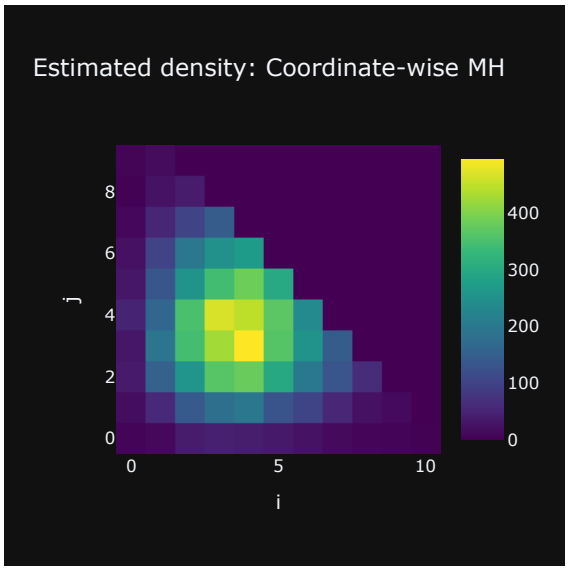
In [ ]:
```python
def CW_metropolis_hastings(g, proposal_sampler, n, x0):

    # Handle if x0 is a scalar
    x0 = np.array([x0]) if np.shape(x0) == () else x0

    X = np.zeros((n+1, len(x0)))
    X[0] = x0

    for i in range(n):
        X[i+1] = X[i]
        for j in range(len(x0)):
            # Propose a new state for the j-th coordinate
            Y = proposal_sampler(X[i+1], j)
            Y = np.array([Y]) if np.shape(Y) == () else Y

            # Evaluate the target distribution
            g_Y = g(Y)
            g_X = g(X[i+1])

            # Accept or reject the new state for the j-th coordinate
            if g_Y >= g_X:
                X[i+1] = Y
            else:
                accept = np.random.uniform(0,1) < g_Y/g_X
                X[i+1] = Y if accept else X[i+1]

    return X
```

In [ ]:
```python
def proposal_sampler(x, coord):
    new_x = x.copy()
    new_x[coord] = np.random.randint(0, m+1-x[abs(coord-1)])
    return new_x

samples = CW_metropolis_hastings(g, proposal_sampler, num_samples, x0)
```

In [ ]:
```python
bin_edges = np.arange(m+2) - 0.5
f_obs, _, _ = np.histogram2d(samples[burn_in:,0].astype(int), samples[burn_in:,1].astype(int), bins=[bin_edges, bin_edges], density=

# Chi-square test
df = (m+1)*(m+2)/2
chisquare_test(f_obs, f_exp, df, title='Coordinate-wise MH')

# Make heatmap of observed values
fig = go.Figure(data=[go.Histogram2d(x=samples[:,0], y=samples[:,1], colorscale='Viridis')])
```

```
fig.update_layout(title='Estimated density: Coordinate-wise MH', xaxis_title='i', yaxis_title='j', width=400, height=400)
fig.show()
```

```
──────────────────────────────
   >>> Chi-Squared Test <<<
      Coordinate-wise MH
──────────────────────────────
Degrees of Freedom: 66
Test Statistic: 0.0067
p-value: 1.0000
──────────────────────────────
```



We see that the results are very similar for the coordinate wise method. The calculated p-value does not allow us to reject the null hypothesis that the two distributions are the same. And the plot showing the estimate density is by inspection very similar to the theoretical density.

(c) Gibbs Sampler

The Gibbs sampler is different from the Metropolis-Hastings algorithm in the way that the Gibbs sampler does not reject any samples. It does however require more manual work in setting up as we have to calculate the conditional distributions for each variable given the others. We then update each coordinate in turn and repeat this process for a number of iterations.

To use the Gibbs sampler we first need to derive the conditional distribution of each variable given the others. So given $P(i,j)$ from above we need to derive $P(i|j)$ and $P(j|i)$. First we derive $P(i|j)$:

$$\begin{aligned} P(i|j) &= \frac{P(i,j)}{P(j)} = \frac{P(i,j)}{\sum_i P(i,j)} \\ &= \frac{P(i,j)}{\frac{A_2^j}{j!}\sum_{k=0}^{m-j} c\frac{A_1^k}{k!}} = \frac{c\frac{A_1^i}{i!}\frac{A_2^j}{j!}}{\frac{A_2^j}{j!}\sum_{k=0}^{m-j} c\frac{A_1^k}{k!}} \\ &= \frac{\frac{A_1^i}{i!}}{\sum_{k=0}^{m-j}\frac{A_1^k}{k!}} \end{aligned}$$

We have now derived the conditional distribution $P(i|j)$. A similar derivation for $P(j|i)$ can be made and show that:

$$P(j|i) = \frac{\frac{A_2^j}{j!}}{\sum_{k=0}^{m-i}\frac{A_2^k}{k!}}$$

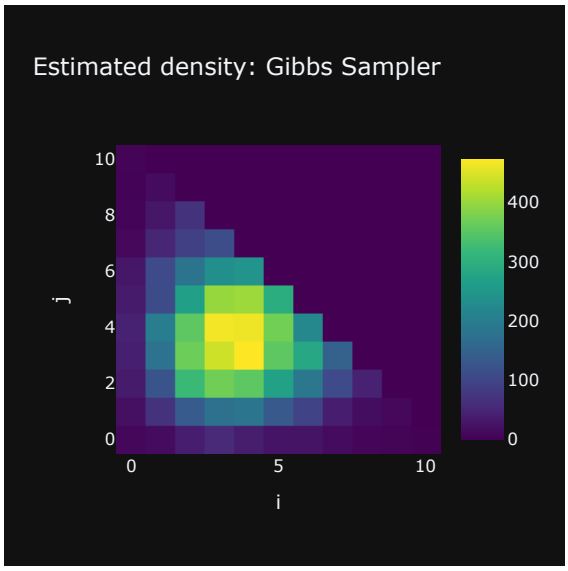
Down below, we implement these two functions and use them in the Gibbs sampler.

In [ ]:
```
def p_i_give_j(i, j, A1, m):
    numerator = (A1**i)/factorial(i)
    k = np.arange(m-j+1)
    denominator = np.sum((A1**k)/factorial(k))
    return numerator/denominator

def p_j_give_i(i,j,A2,m):
    numerator = (A2**j)/factorial(j)
    k = np.arange(m-i+1)
    denominator = np.sum((A2**k)/factorial(k))
```

```
    return numerator/denominator

def gibbs_sampler(n, x0, A1, A2, m):
    # Handle if x0 is a scalar
    x0 = np.array([x0]) if np.shape(x0) == () else x0

    X = np.zeros((n+1, len(x0)))
    X[0] = x0
    for k in range(n):
        i, j = X[k].astype(int)

        # There may be issue with the indexing here (overwriting i before accessing it in the next line)
        i = np.random.choice(np.arange(0,m-j+1), p=[p_i_give_j(h,j,A1,m) for h in range(m-j+1)])
        j = np.random.choice(np.arange(0,m-i+1), p=[p_j_give_i(i,h,A2,m) for h in range(m-i+1)])

        Y = np.array([i,j])
        X[k+1] = Y

    return X
```

In [ ]:
```
A1 = A2 = 4
m = 10
n = 10000
x0 = np.array([0, 0])

# Run Gibbs sampler
samples = gibbs_sampler(n, x0, A1, A2, m)

# Perform chi-squared test
chisquare_test(f_obs, f_exp, m, title='Gibbs Sampler')

# Make heatmap of observed values
fig = go.Figure(data=[go.Histogram2d(x=samples[:,0], y=samples[:,1], colorscale='Viridis')])
fig.update_layout(title='Estimated density: Gibbs Sampler', xaxis_title='i', yaxis_title='j', width=400, height=400)
fig.show()
```

```
──────────────────────────
   >>> Chi-Squared Test <<<
        Gibbs Sampler
──────────────────────────
Degrees of Freedom: 10
Test Statistic: 0.0067
p-value: 1.0000
──────────────────────────
```



With the Gibbs sampler we get similar results as with the Metropolis-Hastings algorithm. The p-value is 1, which means that we cannot reject the null hypothesis that the two distributions are the same. The plot showing the estimate density is also very similar to the theoretical density.

Part 3 - Bayesian Statistical Problem

In this part we wish to use the Metropolis-Hastings algorithm to sample from a multivariate normal distribution where the two variables are correlated. First we setup the prior function and the sampler needed for the Metropolis-Hastings algorithm.

In [ ]:
```
rho = 0.5
prior = multivariate_normal(np.array([0, 0]), np.array([[1, rho], [rho, 1]]))
```

```
sample_x = lambda theta, psi, n=1: norm.rvs(theta, psi, n)
```

(a) Generate a Sample from Prior Distribution

We use the defined prior function to generate a sample from the prior distribution.

In [ ]:
```
# Generate sample from prior
xi, gamma = prior.rvs()
theta, psi = np.exp(xi), np.exp(gamma)

print(f"θ = {theta:.4f}\nψ = {psi:.4f}")
```

```
θ = 5.4562
ψ = 6.8018
```

(b) Generate Observations

Now we generate 10 observations from the multivariate normal distribution with the given parameters.

In [ ]:
```
n = 10
X = sample_x(theta, psi, n)
print("\n".join([f"x{i+1} = {X[i]:.4f}" for i in range(n)]))
```

```
x1 = 7.0482
x2 = 8.7323
x3 = 11.9512
x4 = 21.5883
x5 = 4.1428
x6 = 3.9027
x7 = -1.2196
x8 = 1.7174
x9 = 11.3741
x10 = 17.6899
```

(c) Posterior Distribution

Before we can use the Metropolis-Hastings algorithm we need to derive the posterior density up to a constant.

Using Bayes theorem for densities, the posterior distribution of (θ, ψ) is given by,

$$f(\theta, \psi|x) = \frac{f(x|\theta, \psi)f(\theta, \psi)}{f(x)} \propto f(x|\theta, \psi)f(\theta, \psi).$$

The prior and likelihood densities are given as,

$$f(\theta, \psi) = \frac{1}{2\pi\theta\psi\sqrt{1-\rho^2}} e^{-\frac{\log(\theta)^2 - 2\rho\log(\theta)\log(\psi) + \log(\psi)^2}{2(1-\rho^2)}}, \quad f(x|\theta, \psi) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\psi^2}} e^{-\frac{(x_i-\theta)^2}{2\psi^2}}.$$

The posterior distribution is then given by,

$$f(\theta, \psi|x) \propto \frac{1}{2\pi\theta\psi\sqrt{1-\rho^2}} e^{-\frac{\log(\theta)^2 - 2\rho\log(\theta)\log(\psi) + \log(\psi)^2}{2(1-\rho^2)}} \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\psi^2}} e^{-\frac{(x_i-\theta)^2}{2\psi^2}}.$$

As the posterior distribution is only required up to a constant, we might as well get rid of the constant terms, yielding,

$$f(\theta, \psi|x) \propto \frac{1}{\theta\psi} e^{-\frac{\log(\theta)^2 - 2\rho\log(\theta)\log(\psi) + \log(\psi)^2}{2(1-\rho^2)}} \prod_{i=1}^{n} \frac{1}{\psi} e^{-\frac{(x_i-\theta)^2}{2\psi^2}}.$$

Simplifying the above expression, we get,

$$f(\theta, \psi|x) \propto \frac{1}{\theta\psi^{n+1}} e^{-\frac{\log(\theta)^2 - 2\rho\log(\theta)\log(\psi) + \log(\psi)^2}{2(1-\rho^2)}} e^{-\frac{1}{2\psi^2}\sum_{i=1}^{n}(x_i-\theta)^2}.$$

Implementation of Posterior Density

In [ ]:
```
def posterior(theta, psi, X, rho=0.5):

    # Ensure that theta and psi are positive
    if theta <= 0 or psi <= 0:
        return 0

    # log-prior (multiplicative constants are ignored as we only care about the posterior shape)
    log_prior = - np.log(theta) - np.log(psi) - (np.log(theta)**2 - 2*rho*np.log(theta)*np.log(psi) + np.log(psi)**2)/(2*(1-rho**2)
```

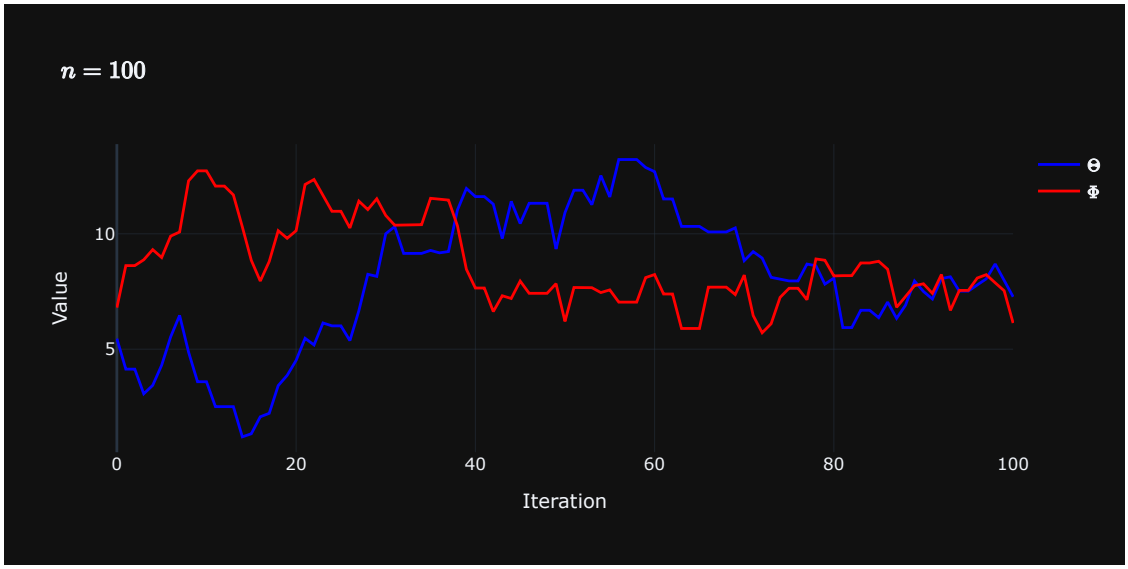
```
    # Log-likelihood
    log_likelihood = norm.logpdf(X, theta, psi).sum()

    # Log-posterior
    log_posterior = log_likelihood + log_prior

    return np.exp(log_posterior)
```

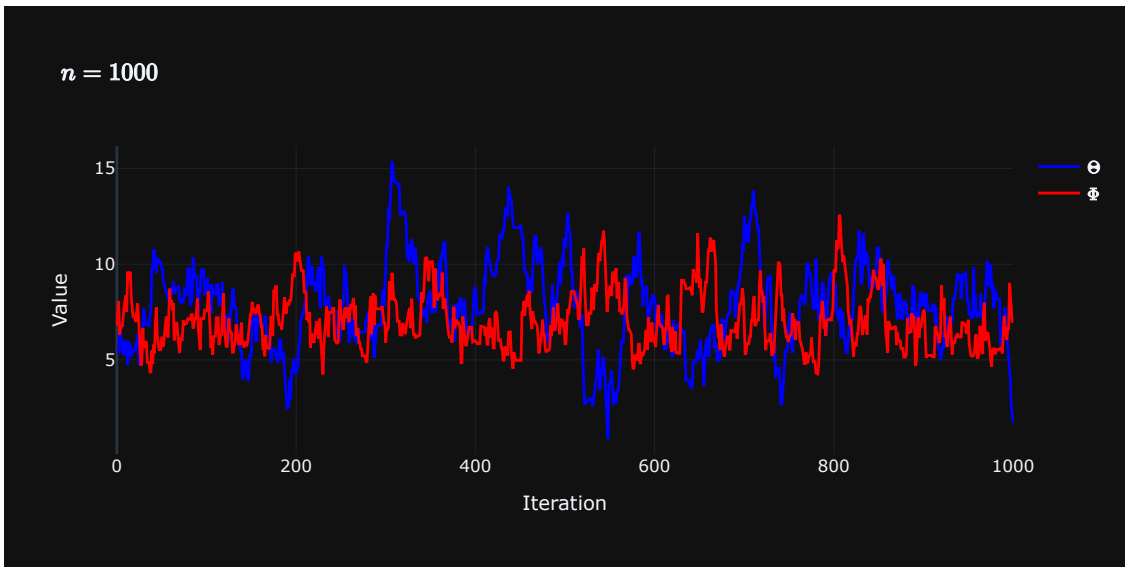
In [ ]:
```
# Vague attempt to use hint 2

# def posterior(theta, psi, X, rho=0.5):

#     # Ensure that theta and psi are positive
#     if theta <= 0 or psi <= 0:
#         return 0

#     # Sample size, sample mean and sample variance
#     n = len(X)
#     X_mu = np.mean(X)
#     X_var = np.var(X, ddof=1)

#     # Posterior marginal of sample mean and variance
#     post_mean = norm.pdf(theta, X_mu, psi/np.sqrt(n))
#     post_var = chi2.pdf((n-1)*X_var/psi**2, n-1)

#     # log-prior (multiplicative constants are ignored as we only care about the posterior shape)
#     log_prior = -np.log(theta) - (n+1)*np.log(psi) - (np.log(theta)**2 + np.log(psi)**2)/(2*(1-rho**2))

#     return np.exp(log_prior)*post_mean*post_var
```

(d) and (e) Metropolis-Hastings

Now we run the Metropolis-Hastings algorithm for 100 and 100 iterations with the given parameters and compare the results.

In [ ]:
```
# Sampler for the proposal distribution
proposal_sampler = lambda x: np.random.normal(x, 1, 2)

# Density of the target distribution (up to a constant)
target = lambda x: posterior(x[0], x[1], X)

# Run Metropolis-Hastings
samples100 = metropolis_hastings(target, proposal_sampler, 100, [theta, psi])
samples1000 = metropolis_hastings(target, proposal_sampler, 1000, [theta, psi])

# make trace plots for theta and phi
fig = go.Figure()
fig.add_trace(go.Scatter(x=np.arange(100+1), y=samples100[:,0], mode='lines', name=r'$\Theta$', line=dict(color='blue')))
fig.add_trace(go.Scatter(x=np.arange(100+1), y=samples100[:,1], mode='lines', name=r'$\Phi$', line=dict(color='red')))
fig.update_layout(title='Trace plots for Theta and Phi: $n = 100$', xaxis_title='Iteration', yaxis_title='Value', width=800, height=
fig.show()

# make trace plots for theta and phi
fig = go.Figure()
fig.add_trace(go.Scatter(x=np.arange(1000+1), y=samples1000[:,0], mode='lines', name=r'$\Theta$', line=dict(color='blue')))
fig.add_trace(go.Scatter(x=np.arange(1000+1), y=samples1000[:,1], mode='lines', name=r'$\Phi$', line=dict(color='red')))
fig.update_layout(title="Trace plots for Theta and Phi: $n = 1000$", xaxis_title='Iteration', yaxis_title='Value', width=800, height
fig.show()
```


$n = 100$
Θ
Φ
Value
10
5
0
20
40
60
80
100
Iteration


$n = 1000$
Θ
Φ
Value
15
10
5
0
200
400
600
800
1000
Iteration

Computer Exercise 7: Simulated Annealing

In [ ]:
```
# Plotting
import plotly.graph_objects as go
import plotly.express as px
import plotly.subplots as sp
import plotly.io as pio
pio.renderers.default = "notebook+pdf"
pio.templates.default = "plotly_dark"

# Utilities
import numpy as np
```

Part 1 - Simulated Annealing for TSP

The travelling salesman problem (TSP) is an optimization problem where the goal is to find the shortest possible route that visits a given set of cities and returns to the origin city. We will try and solve this problem using simulated annealing. We will implement aspects from simulated annealing into our Metroplis-Hastings algorithm.

We will use the cost function for the TSP for calculating the probabilities if the new state is accepted or not. The cost function is defined as the sum of the distances between the cities in the order they are visited. The cost function is given by:

$$\sum_{i=1}^{n-1} A(S_i, S_{i+1}) + A(S_n, S_1)$$

where $A(S_i, S_{i+1})$ is the distance between city S_i and city S_{i+1}.

The initial guess for the Metroplis-Hastings algorithm is a random permutation of the cities. The algorithm will then try to find a better permutation by swapping two cities and calculating the cost function for the new permutation. If the cost function is lower than the previous one, the new permutation is accepted. If the cost function is higher, the new permutation is accepted with a probability given by the Metropolis-Hastings algorithm. The algorithm will then continue to swap cities until the cost function converges.

In [ ]:
```
# Temperature functions
T1 = lambda k: 1/np.sqrt(1+k)
T2 = lambda k: -np.log(k+1)

num_stations1 = 10
num_samples = 1000
```

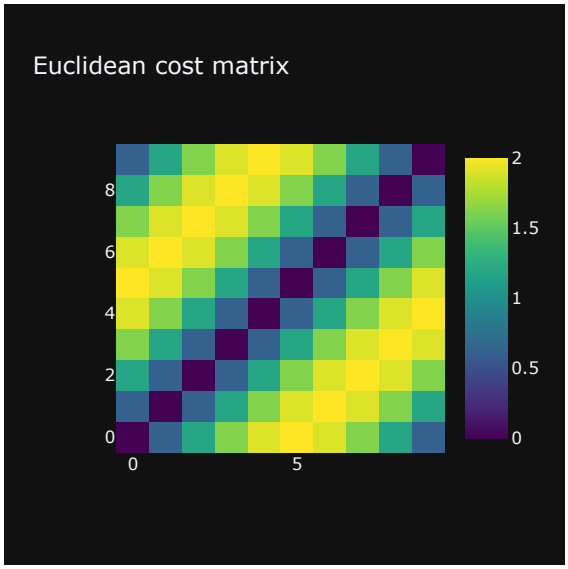
(a) Euclidean Distance

For this first part of the exercise, we will use the Euclidean distance as the distance function between the cities. We will assume that the cities are placed on the unit circle. The Euclidean distance between two cities S_i and S_j is given by:

$$A(S_i, S_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

First we define the cost matrix and plot it.

In [ ]:
```
def unit_circle_points(n):
    # Generate n equally spaced angles between 0 and 2*pi
    angles = np.linspace(0, 2*np.pi, n, endpoint=False)
    points = np.vstack([np.cos(angles), np.sin(angles)]).T
    return points

def euclidean_cost_matrix(points):
    # Calculate the pairwise distances between the points
    diff = points[:, None, :] - points[None, :, :]
    distances = np.linalg.norm(diff, axis=2)
    return distances

points1 = unit_circle_points(num_stations1)
C1 = euclidean_cost_matrix(points1)

# Plot the cost matrix
fig = go.Figure(data=go.Heatmap(z=C1, colorscale="Viridis"))
fig.update_layout(title="Euclidean cost matrix", width=400, height=400)
fig.show()
```

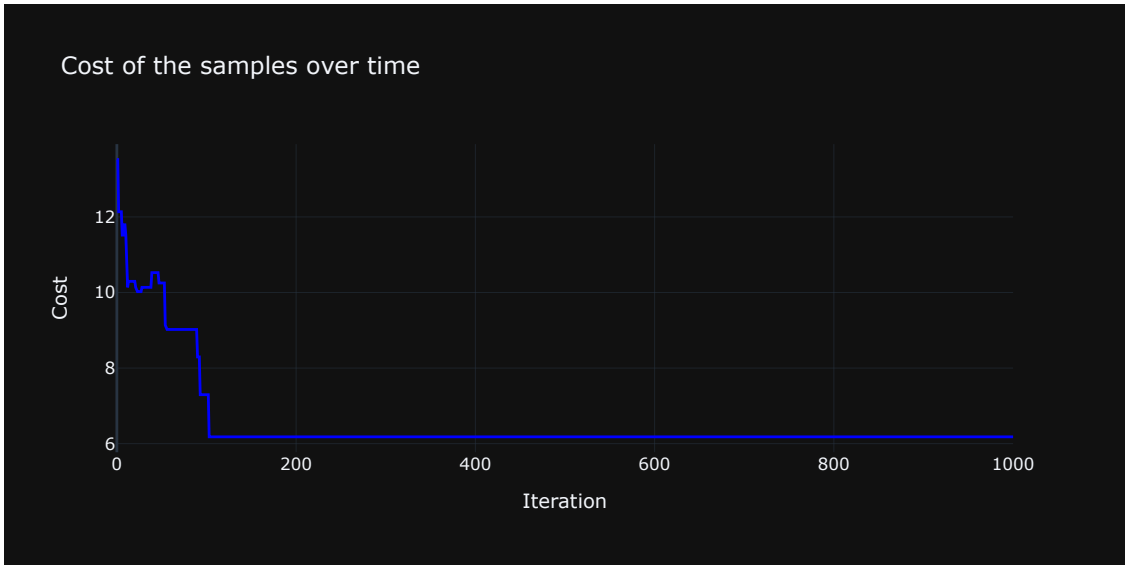


Now that we have our cost matrix, we will implement the Metropolis-Hastings algorithm, our sampler and the cost function.

In [ ]:
```
def metropolis_hastings(g, proposal_sampler, T, num_samples, x0, cost_matrix):

    # Handle if x0 is a scalar
    x0 = np.array([x0]) if np.shape(x0) == () else x0

    X = np.zeros((num_samples+1, len(x0)), dtype=int)
    X[0] = x0

    for i in range(num_samples):

        # Propose a new state
        Y = proposal_sampler(X[i])
        Y = np.array([Y]) if np.shape(Y) == () else Y

        # Evaluate the target distribution
        g_Y = g(Y, cost_matrix)
        g_X = g(X[i], cost_matrix)

        # Accept or reject the new state
        if g_Y <= g_X:
            X[i+1] = Y
        else:
            accept = np.random.uniform(0,1) < np.exp(-(g_Y - g_X) / T(i))
            X[i+1] = Y if accept else X[i]

    return X
```

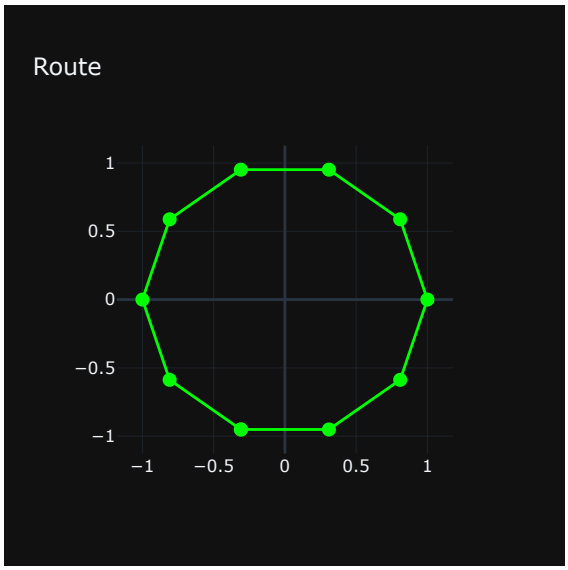
In [ ]:
```
def f(x, C):
    return np.sum(C[x, np.roll(x, -1)])

def proposal_sampler(x):
    # Randomly select two indices
    i, j = np.random.choice(len(x), 2, replace=False)

    # Swap them
    x_new = np.copy(x)
    x_new[i], x_new[j] = x[j], x[i]

    return x_new

x0_1 = np.random.choice(num_stations1, num_stations1, replace=False)
samples1 = metropolis_hastings(f, proposal_sampler, T1, num_samples, x0_1, C1)

# calculate the cost of the samples
costs1 = np.array([f(x, C1) for x in samples1])
```

In [ ]:
```
fig = go.Figure()
fig.add_trace(go.Scatter(x=np.arange(num_samples+1), y=costs1, mode='lines', marker=dict(color='blue')))
fig.update_layout(title="Cost of the samples over time", xaxis_title="Iteration", yaxis_title="Cost", width=800, height=400)
fig.show()
```



Above we have plotted the cost of the samples. We can see that the cost converges to a minimum value 6.18 which is approximately the circumference of the unit circle. We can also plot the path of the cities to see the route that the algorithm has found.

In [ ]:
```
def plot_route(points, order):
    # Reorder the coordinates according to the permutation
    order = np.concatenate([order, [order[0]]])
    x, y = points[order].T

    # Plot the points
    fig = go.Figure()
    fig.add_trace(go.Scatter(x=x, y=y, mode='markers+lines', marker=dict(size=10, color='Lime')))
    fig.update_layout(title="Route", width=400, height=400)
    fig.show()

plot_route(points1, samples1[-1])
```



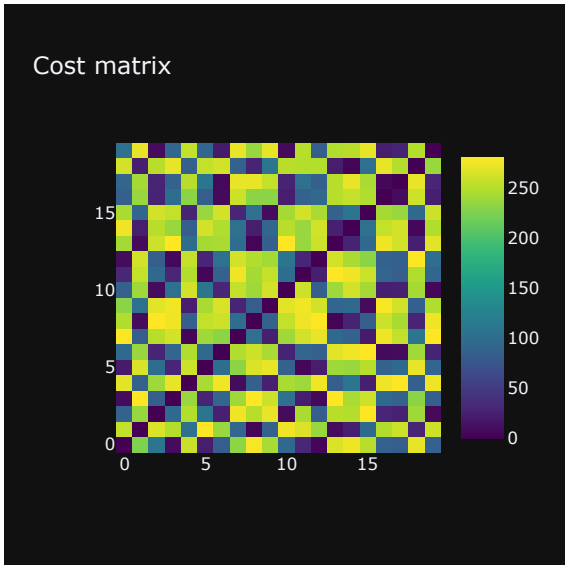
By plotting the route of the cities, we can see that the algorithm has found the shortest route between the cities, which is where the salesman visits the closest neighbouring city.

(b) Cost Matrix

Next we will use the same algorithm with the given cost matrix.

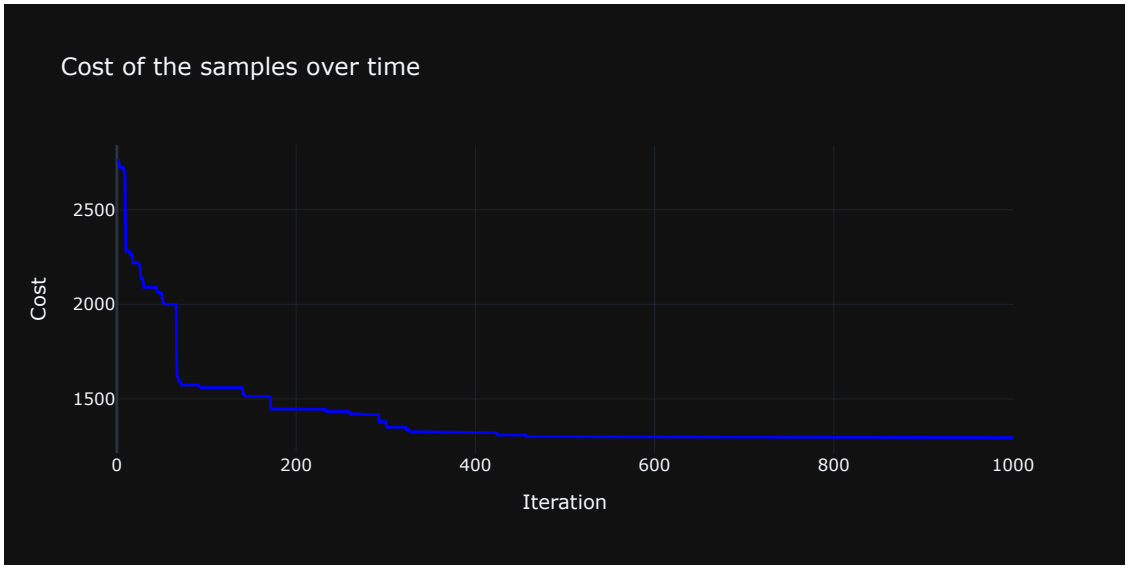
In [ ]:
```
# Load cost.csv
cost_matrix = np.loadtxt("data/cost.csv", delimiter=",")

num_stations2 = cost_matrix.shape[0]
x0_2 = np.random.choice(num_stations2, num_stations2, replace=False)

# Plot the cost matrix
```

```
fig = go.Figure(data=go.Heatmap(z=cost_matrix, colorscale="Viridis"))
fig.update_layout(title="Cost matrix", width=400, height=400)
fig.show()
```



Here we see that the given cost matrix is not symmetric, so the optimal route might not be as simple as in the previous case.

In [ ]:
```
samples2 = metropolis_hastings(f, proposal_sampler, T1, num_samples, x0_2, cost_matrix)

costs2 = np.array([f(x, cost_matrix) for x in samples2])

fig = go.Figure()
fig.add_trace(go.Scatter(x=np.arange(num_samples+1), y=costs2, mode='lines', marker=dict(color='blue')))
fig.update_layout(title="Cost of the samples over time", xaxis_title="Iteration", yaxis_title="Cost", width=800, height=400)
fig.show()
```



Here we again see that the cost of the route converges to a minimum value of 880. With this problem, we however saw that the algorithm does not always converge to a global minimum.

In [ ]:
```
cost_vec = np.zeros(10)

for i in range(10):
    samples3 = metropolis_hastings(f, proposal_sampler, T1, num_samples, x0_2, cost_matrix)
    cost_vec[i] = f(samples3[-1], cost_matrix)

print("Final costs:", cost_vec)
```

```
Final costs: [1220. 1145. 1105. 1331. 1278. 1140. 1150.  820. 1282. 1292.]
```

Computer Exercise 8: Bootstrap

In [ ]:
```
# Plotting
import plotly.graph_objects as go
import plotly.express as px
import plotly.subplots as sp
import plotly.io as pio
pio.renderers.default = "notebook+pdf"
pio.templates.default = "plotly_dark"

# Utilities
import numpy as np
```

Part 1 - Exercise 13 in Chapter 8 of Ross

(a) How to Estimate p using Bootstrap

Given the data, we sample n samples with replacement. Using these samples, we calculate deviation from the mean as seen in the exercise. We then repeat the sampling and calculation r times. Using the r values of the deviation we can estimate the probability that the deviation falls between a and b.

(b) Estimating p for an Example

Now we are given an array of data and use the above mentioned method to estimate the probability that $\sum_{i=1}^{n} X_i/n - \mu$ falls between a and b.

In [ ]:
```
n1 = 10
data1 = np.array([56, 101, 78, 67, 93, 87, 64, 72, 80, 69])
a = -5
b = 5

# Number of bootstrap samples
r = 100000

# Bootstrap
samples1 = np.random.choice(data1, (r, n1))
means1 = np.mean(samples1, axis=1)
mu1 = np.mean(means1)
deviations1 = means1 - mu1

# Compute the probability
count = np.histogram(deviations1, [a, b])[0][0]
prob = count/r
print("p =", prob)
```

```
p = 0.76774
```

We found that the probability is approximately 0.76.

Part 2 - Exercise 15 in Chapter 8 of Ross

In this exercise, we are given a data set and we are asked to estimate the variance of the data set. We use the bootstrap method to estimate the variance of the data set.

In [ ]:
```
n2 = 15
data2 = np.array([5, 4, 9, 6, 21, 17, 11, 20, 7, 10, 21, 15, 13, 16, 8])

# Bootstrap
samples2 = np.random.choice(data2, (r, n2))
means2 = np.mean(samples2, axis=1)
deviations2 = means2 - np.mean(means2)

mu2 = np.mean(means2)
res = (np.sum((samples2 - mu2)**2, axis=1))/(n2-1)

var = np.var(res)

print(f"s^2 = {var:.2f}")
```

```
s^2 = 55.90
```

We found the variance of the data set to be approximately 55.9.

Part 3 - Median Bootstrap Variance

In this exercise we wish to define a bootstrad method to estimate the median of a dataset and the variance of the median. We will use samples from the pareto distribution as our data set.

(a, b, c) Bootstrap Statistics

In [ ]:
```
def bootstrap(data, r):
    n = len(data)

    sample = np.random.choice(data, (r, n))
    medians = np.median(sample, axis=1)
    means = np.mean(sample, axis=1)

    mu = np.mean(means)
    var_mu = np.var(means, ddof=1)
    median = np.median(medians)
    var_median = np.var(medians, ddof=1)

    return mu, var_mu, median, var_median

N = 200
r = 100

beta = 1
k = 1.05

pareto = beta*(np.random.uniform(0,1,N)**(-1/k)-1)

# bootstrap
mu, var_mu, median, var_median = bootstrap(pareto, r)

print(f"μ = {mu:.2f}\nvar(μ) = {var_mu:.2f}\nmedian = {median:.2f}\nvar(median) = {var_median:.2f}")
```

```
μ = 8.81
var(μ) = 20.16
median = 0.79
var(median) = 0.02
```

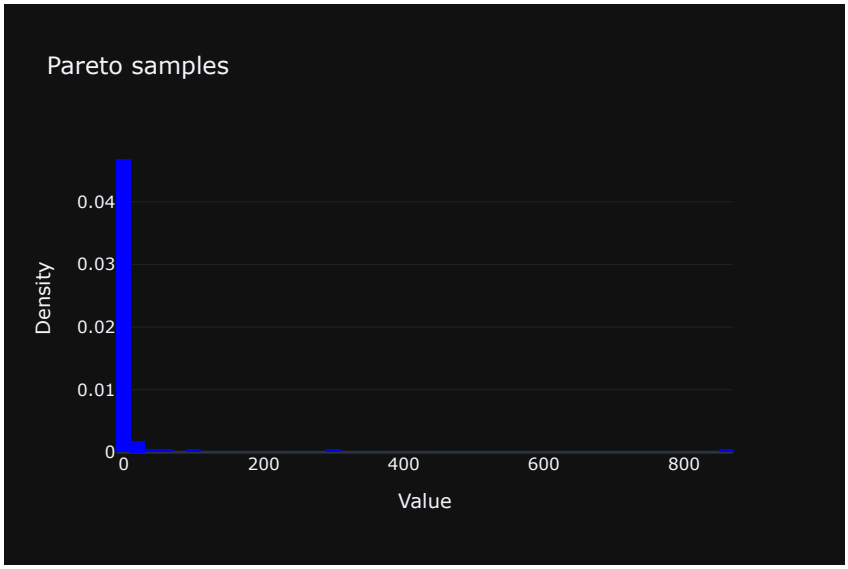
After having implemented a function for the bootstrap method, we generate 200 samples from the pareto distribution and calculate the median and variance of the median using the bootstrap method.

We found the median of the data set to be approximately 0.88 and the variance of the median to be approximately 0.01. So we are quite confident in the value of the median.

(d) Precision

The pareto distribution does not have a mean when $k = 1$. Consequently, the bootstrap estimate becomes increasingly unstable when k approaches 1. This is not the case for the median, which is well defined for all values of k. The explains why the estimate for the median is much more stable than that of the mean when $k = 1.05$.

In [ ]:
```
# Plot pareto
fig = go.Figure(go.Histogram(x=pareto, histnorm='probability density', marker=dict(color='Blue')))
fig.update_layout(title="Pareto samples", xaxis_title="Value", yaxis_title="Density", width=600, height=400)
fig.show()
```


Pareto samples
Density
0.04
0.03
0.02
0.01
0
0
200
400
600
800
Value