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.

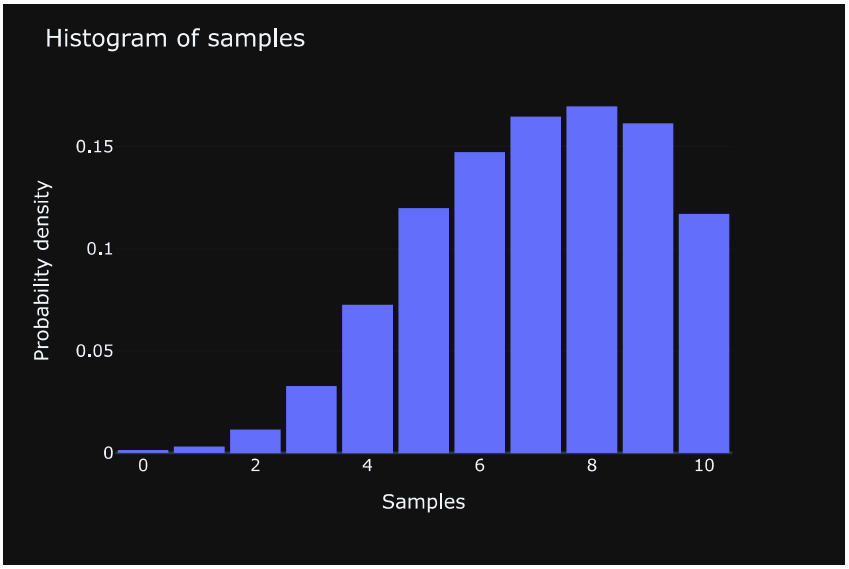
The Metropolis-Hastings algorithm works by taking an initial start guess, which in this example for the Erlang system in an integer between 0 and m. We then propose a new state by sampling a new integer between 0 and m. For the old state and the new state, we calculate their probabilities using the formula 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)
```

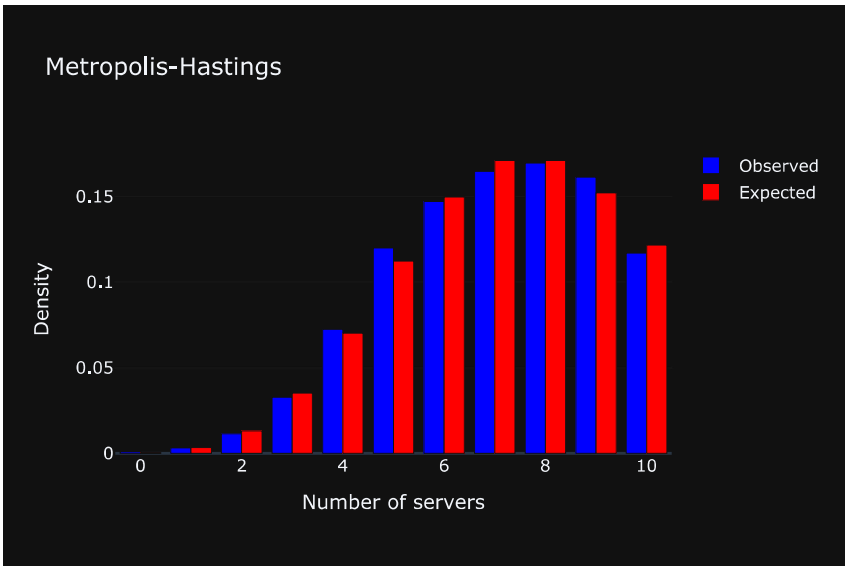
In [ ]:
```
# Make a histogram
fig = px.histogram(x=samples.flatten(), nbins=m+1, histnorm='probability density')
fig.update_layout(title='Histogram of samples', xaxis_title='Samples', yaxis_title='Probability density', bargap=0.1)
fig.update_layout(width=600, height=400)
fig.show()
```



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: 0.0026
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

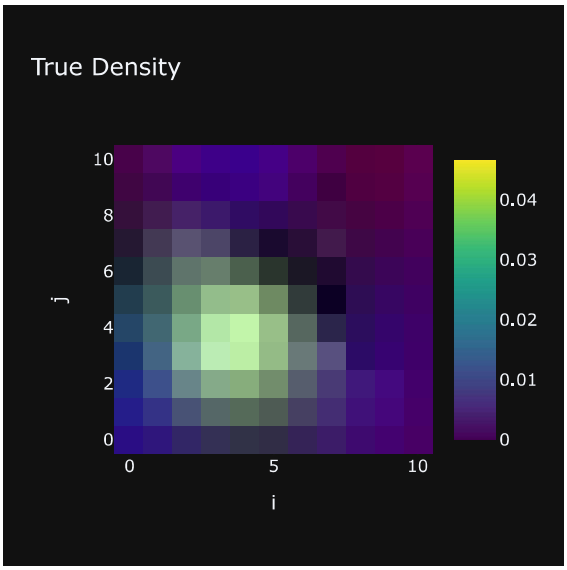
Now we expand the Erlang system to include two different call types. The probability of having i and j customers in the system is given by the Erlang distribution:

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

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

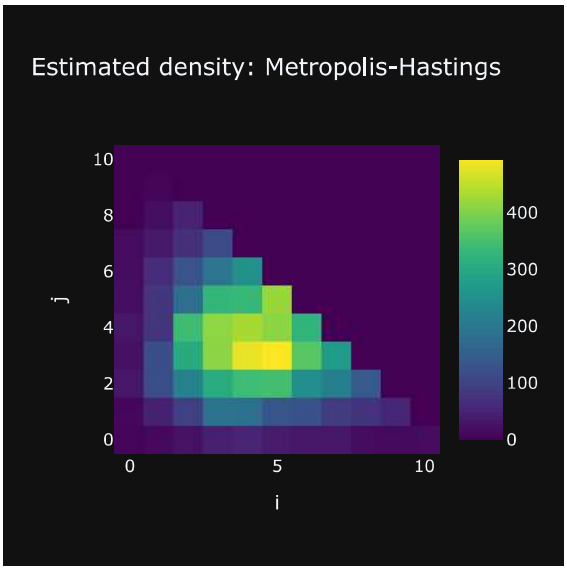
We ran the Erlang system with two different call types through the Metropolis-Hastings algorithm for 10000 iterations with the given parameters.

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=True

# 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: 0.1332
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.

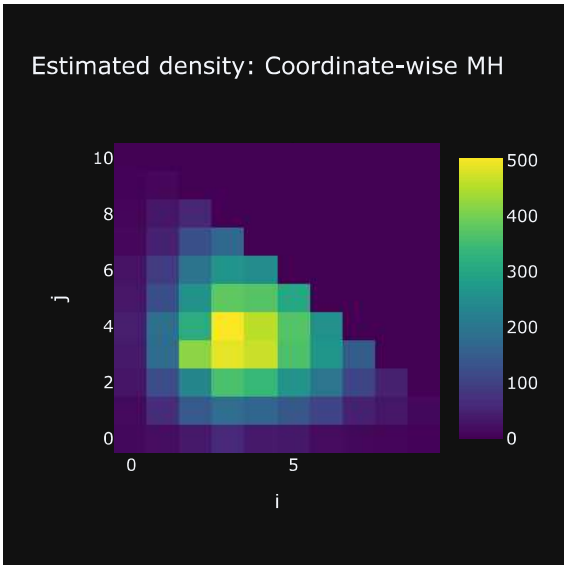
In [ ]:
```
def CW_metropolis_hastings(g, proposal_sampler, n, x0):

    # 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 [ ]:
```
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 [ ]:
```
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=True

# 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: 0.0078
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:

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

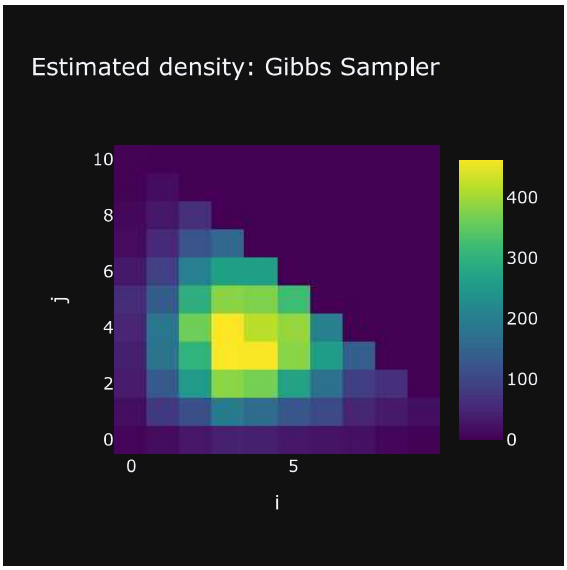
Down below, we implement these two functions and use them in the Gibbs sampler.

In [ ]:
```python
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 [ ]:
```python
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: 0.0078
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 = 2.0396
ψ = 3.4096
```

(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 = 5.7028
x2 = 9.0472
x3 = -1.2781
x4 = -2.0476
x5 = -0.8077
x6 = 1.9098
x7 = -7.1817
x8 = 1.7344
x9 = -0.9412
x10 = 1.9327
```

(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 = np.sum(norm.logpdf(X, theta, psi))

    # Log-posterior
    log_posterior = log_likelihood + log_prior

    return np.exp(log_posterior)
```

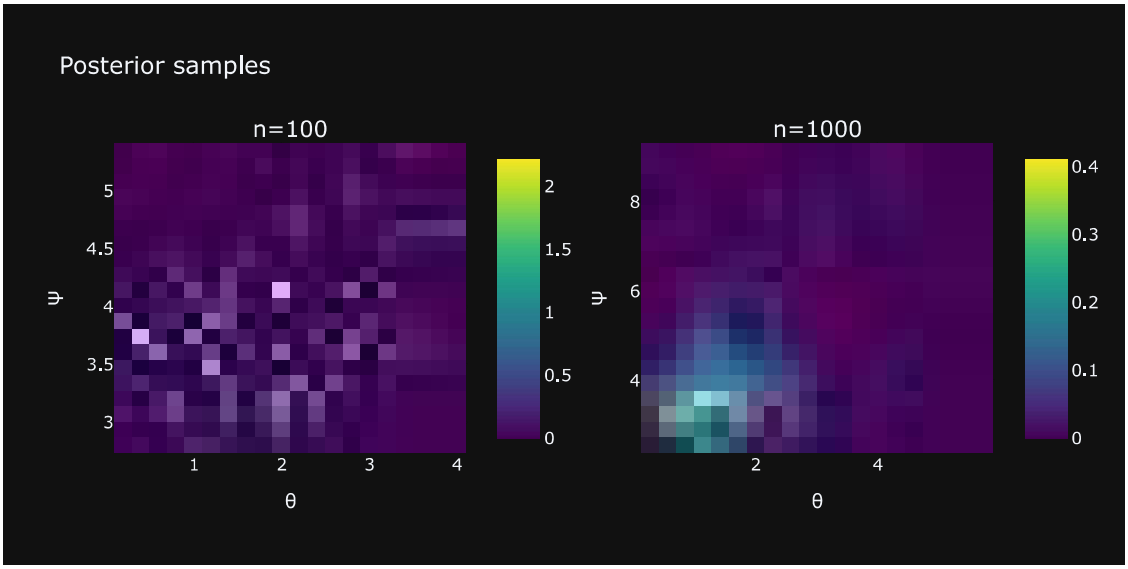
(d) and (e) Metropolis-Hastings

Now we run the Metropolis-Hastings algorithm for 100 and 1000 iterations with the given parameters and compare the results.

In [ ]:
```
# Sampler for the proposal distribution
proposal_sampler = lambda x: np.random.normal(x, 0.5, 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])
```

In [ ]:
```
# Compute histograms
values100, x100, y100 = np.histogram2d(samples100[:,0], samples100[:,1], bins=20, density=True)
values1000, x1000, y1000 = np.histogram2d(samples1000[:,0], samples1000[:,1], bins=20, density=True)

# Create subplots
fig = sp.make_subplots(rows=1, cols=2, subplot_titles=('n=100', 'n=1000'), horizontal_spacing=0.2)
fig.add_trace(go.Heatmap(z=values100, x=x100, y=y100, colorscale='Viridis', name='n=100', colorbar_x=0.42), row=1, col=1)
fig.add_trace(go.Heatmap(z=values1000, x=x1000, y=y1000, colorscale='Viridis', name='n=1000'), row=1, col=2)
fig.update_xaxes(title_text='θ', row=1, col=1)
```

```
fig.update_xaxes(title_text='θ', row=1, col=2)
fig.update_yaxes(title_text='ψ', row=1, col=1)
fig.update_yaxes(title_text='ψ', row=1, col=2)
fig.update_layout(title='Posterior samples', width=800, height=400)
fig.show()
```



We see that for the smaller samplesize of $n = 100$, the posterior distribution is not as well estimated as for the larger samplesize of $n = 1000$, where the distribution of the samples are much more concentrated around the true value of the parameters. This shows that more samples are required to explore and get a good estimate of the posterior distribution.