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
```

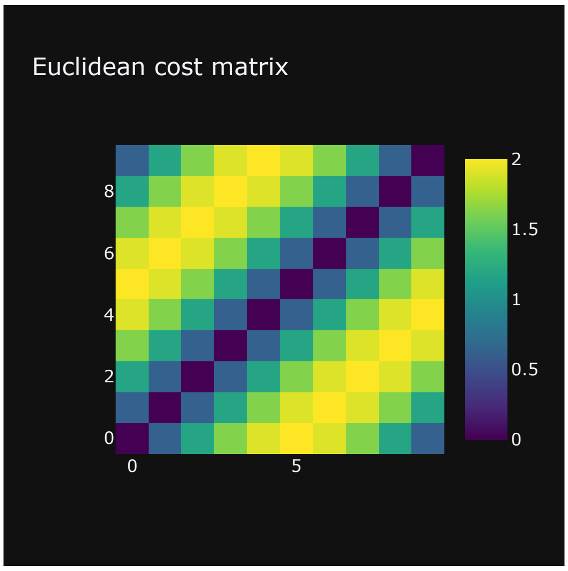
(a) Euclidean Distance

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()
```

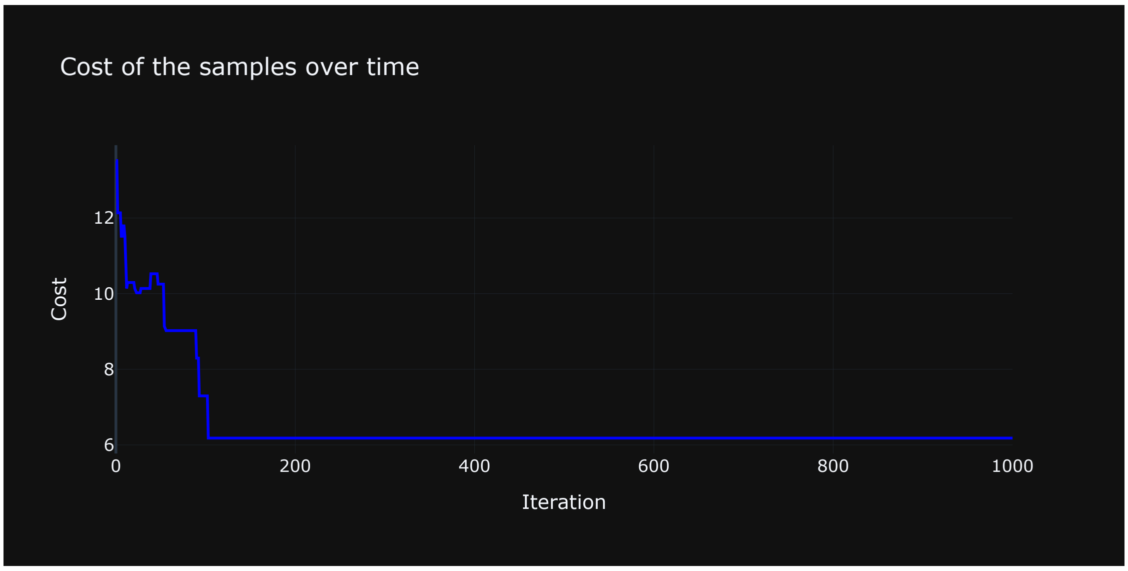


Now that we have our cost matrix, we will implement the Metropolis-Hastings algorithm, our sampler and the cost function.

In [ ]:
```python
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
```

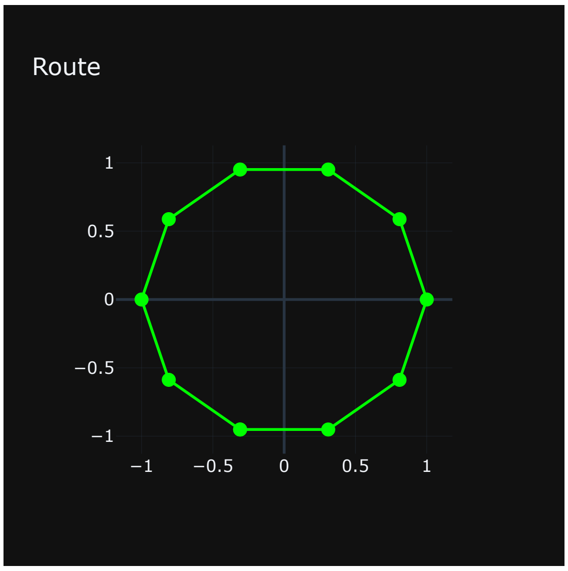
In [ ]:
```python
def f(x, C):
    return np.sum(C[x, np.roll(x, -1)])

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 [ ]:
```python
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])
```



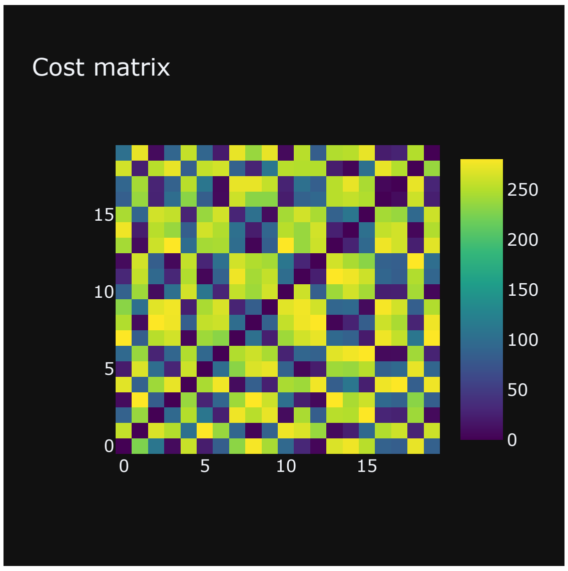
By plotting the route of the cities, we can see that the algorithm has found the shortest route between the cities, which is where the salesman visits the closest neighbouring city.

(b) Cost Matrix

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

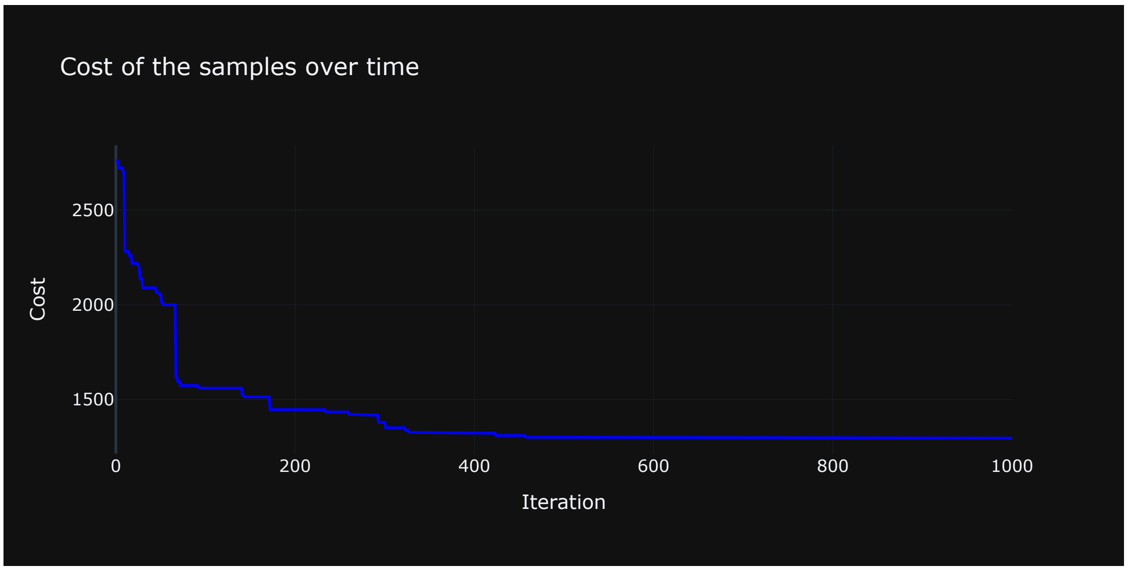
```
In [ ]: # Load cost.csv
        cost_matrix = np.loadtxt("data/cost.csv", delimiter=",")

        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.]
```