Imports

In [1]:
```
import numpy as np
```

Matrix cities

In [2]:
```
cities = np.array([
    [0, 2, 9, 10, 7, 14, 11],
    [1, 0, 6, 4, 12, 8, 10],
    [15, 7, 0, 8, 6, 9, 13],
    [6, 3, 12, 0, 9, 11, 5],
    [7, 12, 6, 9, 0, 4, 8],
    [14, 8, 9, 11, 4, 0, 6],
    [11, 10, 13, 5, 8, 6, 0]
])
```

AOC Parameters

- **num_ants** : Number of agents exploring the solutions
- **num_iterations** : Number of generations
- **decay** : Pheromone evaporations rate
- **alpha** : How much Pheromone strenght effect paths
- **beta** : How much distance effect path
- **num_cities** : shape of matrix

In [4]:
```
num_ants = 10
num_iterations = 100
decay = 0.1
alpha = 1
beta = 2
num_cities = cities.shape[0]
```

Initializing Pheromones

- This is will generate 7x7 sqaure matrix with value filled as 0.142 (1/7 = 1.42)

In [13]:
```
pheromone = np.ones((num_cities,num_cities)) / num_cities
print(pheromone)
```

```
[[0.14285714 0.14285714 0.14285714 0.14285714 0.14285714 0.14285714
  0.14285714]
 [0.14285714 0.14285714 0.14285714 0.14285714 0.14285714 0.14285714
  0.14285714]
 [0.14285714 0.14285714 0.14285714 0.14285714 0.14285714 0.14285714
  0.14285714]
 [0.14285714 0.14285714 0.14285714 0.14285714 0.14285714 0.14285714
  0.14285714]
 [0.14285714 0.14285714 0.14285714 0.14285714 0.14285714 0.14285714
  0.14285714]
 [0.14285714 0.14285714 0.14285714 0.14285714 0.14285714 0.14285714
  0.14285714]
 [0.14285714 0.14285714 0.14285714 0.14285714 0.14285714 0.14285714
  0.14285714]]
```

Tracking best solution

float('inf') : Is a largest positive integer

In [9]:
```
best_cost = float('inf')
best_path = None
```

Distance calculation Function

- route-distance function
- cities[route[i - 1], route[i]] : Gives distance from current_city to previous_city

In [14]:
```python
def route_distance(route) :
    dist = 0
    for i in range(len(route)) :
        dist += cities[route[i - 1], route[i]]
    return dist
```

choose next city

- This function is used to randomly pick the next city an ant should visit, based on a probability distribution. (based of parameter p)

In [12]:
```python
def select_next_city(probabilities) :
    return np.random.choice(range(len(probabilities)), p=probabilities)
```

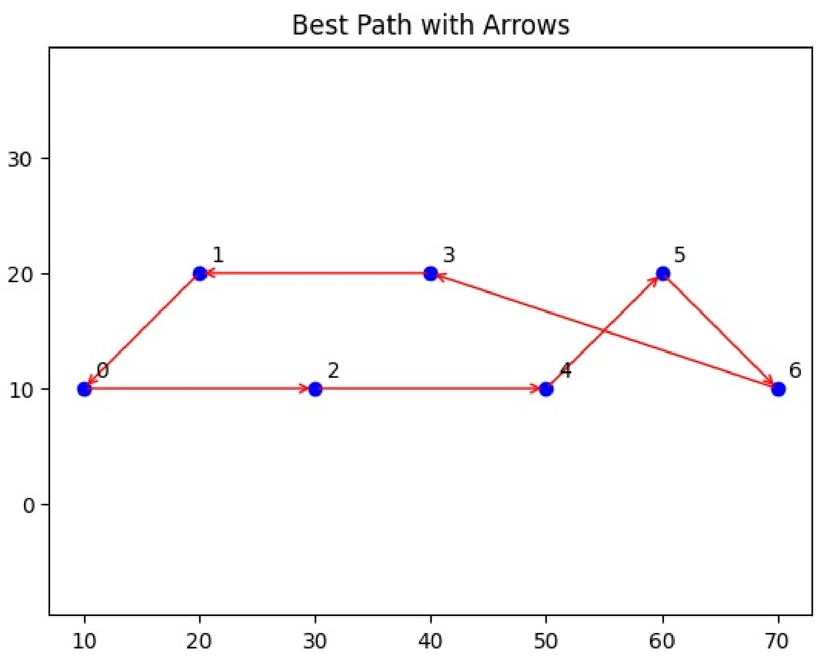
Main ACO Loop

In [23]:
```python
for iterations in range(num_iterations) :
    all_routes = []
    all_distance = []

    for ant in range(num_ants) :
        visited = []
        current_city = np.random.randint(num_cities) # randomly select cities
        visited.append(current_city) # Stored visited city


        while len(visited) < num_cities :
            unvisited = list(set(range(num_cities)) - set(visited)) # num_cities - visisted
            pheromone_value = np.array([pheromone[current_city][j] for j in unvisited])
            distances = np.array([cities[current_city][j] for j in unvisited])

            # Parameter updates
            heuristic = 1 / distances
            prob = (pheromone_value ** alpha) * (heuristic ** beta)
            prob /= prob.sum() # Normalize probabilities so they add up to 1

            # Next cities updates
            next_city = unvisited[select_next_city(prob)]
            visited.append(next_city)
            current_city = next_city


        # sortes the routes and distances
        route = visited
        distances = route_distance(route)
        all_routes.append(route)
        all_distance.append(distances)

        # select the best cost
        if distances < best_cost :
            best_cost = distances
            best_path = route

    # Pheromone evaporation
    pheromone *= (1-decay)


    # pheromone updates
    for route, dist in zip(all_routes,all_distance) :
        for i in range(num_cities) :
            a,b = route[i - 1], route[i]
            pheromone[a][b] += 1/dist




print("Best path:", best_path + [best_path[0]])  # return to start
print("Best cost:", best_cost)
```

```
Best path: [4, 5, 6, 3, 1, 0, 2, 4]
Best cost: 34
```

In [21]:
```python
import matplotlib.pyplot as plt

# Simple fixed coordinates for 7 cities
coords = [
    (10, 10), (20, 20), (30, 10),
    (40, 20), (50, 10), (60, 20), (70, 10)
]
```

```python
# Plot best path (including return to start)
path = best_path + [best_path[0]]

# Draw path with arrows
for i in range(len(path) - 1):
    x1, y1 = coords[path[i]]
    x2, y2 = coords[path[i + 1]]
    plt.annotate(
        '', xy=(x2, y2), xytext=(x1, y1),
        arrowprops=dict(arrowstyle='->', color='red')
    )

# Draw cities as blue dots and label them
for i, (x, y) in enumerate(coords):
    plt.plot(x, y, 'bo')  # blue dot
    plt.text(x + 1, y + 1, str(i))  # label with city index

plt.title("Best Path with Arrows")
plt.axis("equal")
plt.show()
```



```
In [ ]:
```

CSA Implementation

- clonal selection algorithm implementation using sphere function f(x) = x**2

Imports

- numpy : Provides numerical computations (usefull while working with arrays)

In [1]:
```
import numpy as np
```

Fitness function (Objective function)

- Purpose : To determine how good a solution is? (Lower the values better the solution)
- input : x vector
- output : sum of squares of vector's values

In [2]:
```
def fitness_function(x) :
    return np.sum(x**2)
```

Initialize population function

- Creates a sample population
- input : size, lower_bound, upper_bound
- output : randomly generated sample populations

In [3]:
```
def initialize_population(size,lower_bound,upper_bound) :
    return np.random.uniform(lower_bound,upper_bound,size)
```

Clone function

- input : - possible solutions - anitbodies, num_of_clones
- create a clones of possible solutions

In [4]:
```
def clone(antibodies, num_of_clones) :
    return np.repeat(antibodies,num_of_clones)
```

Mutation function - Improvement function

- input clones, mutation_rate, usually 0.1
- output : A mutated set of solutions

In [5]:
```
def mutation(clones, mutation_rate) :
    noise = np.random.normal(0,mutation_rate,clones.shape)
    return clones + noise
```

Select_best_function

- input : mutated set, num_of_best to be selected
- ouput : best solution among mutated set (improved solution set)
- We calcuated fitness of each solution in population and sort it in fitness list, then we sort the fitness solutions (cause lower the value of solution best the solution), then using indices we return best values of solutions from population set

In [6]:
```
def select_best(population, num_of_best) :
    fitness = np.array([fitness_function(x) for x in population])
    sorted_indices = np.argsort(fitness)
    return population[sorted_indices[:num_of_best]]
```

Main CSA core function

In [7]:
```
def clonal_selection_algorithm(pop_size=10, generation=20, upper_bound = -10, lower_bound = 10, clone_factor = !
    population = initialize_population(pop_size,lower_bound,upper_bound)

    for evolution in range(generation) :
        fitness = np.array([fitness_function(x) for x in population])

        # select half best
        best = select_best(population,pop_size//2)

        # clone it
        clones = clone(best, clone_factor)

        # Imporved the cloned solution set
        mutated_clones = mutation(clones, mutation_rate)

        # select best solutions among mutated sets
        new_best = select_best(mutated_clones, pop_size)
        population = new_best

        best_solution = population[np.argmin([fitness_function(x) for x in population])]
        print(f" Evolution Stage : {evolution + 1} best solution : {best_solution:.5f} fitness : {fitness_funct:

    return best_solution
```

In [8]:
```
best = clonal_selection_algorithm()
print("-" * 58)
print(f"| Final best solution.center(50) : {best} |")
print("-" * 58)
```

```
 Evolution Stage : 1 best solution : 0.44911 fitness : 0.20170
 Evolution Stage : 2 best solution : 0.33911 fitness : 0.11500
 Evolution Stage : 3 best solution : 0.08732 fitness : 0.00762
 Evolution Stage : 4 best solution : 0.08118 fitness : 0.00659
 Evolution Stage : 5 best solution : 0.00372 fitness : 0.00001
 Evolution Stage : 6 best solution : 0.00143 fitness : 0.00000
 Evolution Stage : 7 best solution : 0.00690 fitness : 0.00005
 Evolution Stage : 8 best solution : -0.00658 fitness : 0.00004
 Evolution Stage : 9 best solution : 0.01189 fitness : 0.00014
 Evolution Stage : 10 best solution : -0.00341 fitness : 0.00001
 Evolution Stage : 11 best solution : 0.00214 fitness : 0.00000
 Evolution Stage : 12 best solution : -0.01200 fitness : 0.00014
 Evolution Stage : 13 best solution : 0.00753 fitness : 0.00006
 Evolution Stage : 14 best solution : -0.00338 fitness : 0.00001
 Evolution Stage : 15 best solution : -0.00022 fitness : 0.00000
 Evolution Stage : 16 best solution : 0.00262 fitness : 0.00001
 Evolution Stage : 17 best solution : 0.00387 fitness : 0.00001
 Evolution Stage : 18 best solution : -0.00235 fitness : 0.00001
 Evolution Stage : 19 best solution : -0.00240 fitness : 0.00001
 Evolution Stage : 20 best solution : -0.00030 fitness : 0.00000
----------------------------------------------------------
| Final best solution.center(50) : -0.00030287414790200647 |
----------------------------------------------------------
```

In [ ]:

In [ ]:

In [ ]:

Implement AIS (Artificial Immune System)

- AIS on random synthetic generated

Imports

In [1]:
```python
import numpy as np
import matplotlib.pyplot as plt
```

Generate Synthetic Data

- Create artificial data which mimic real world data
- n = 200 : number of data points
- X : np.random.rand(n,2) this will generate 2D array, and each element will have range between 0 and 1.
- y : lables : if x1 and x2 both values > 0.5 then True(1) else False(0)
- astype(int) convert True to 1 and False to 0

In [2]:
```python
def generate_data(n=200):
    X = np.random.rand(n, 2)
    y = ((X[:, 0] > 0.5) & (X[:, 1] > 0.5)).astype(int)
    return X, y

# Visualization -
# X = np.array([[0.32, 0.79], [0.94, 0.68], [0.12, 0.24], [0.45, 0.91], [0.89, 0.10]])
# y = np.array([0, 1, 0, 0, 0])
```

Train NSA Classifier

- Non-Self-Association classifier a biologically inspired algorithm used primarily for anomaly detection
- inputs :
- X_self An array of safe data.
- num_detectors.
- radius: The minimum allowed distance between a detector and any point in X_self.

In [3]:
```python
def train_nsa(X_self, num_detectors=50, radius=0.1):
    detectors = [] # list of detectors
    while len(detectors) < num_detectors: # until all candidates are placed in list
        candidate = np.random.rand(2)  # candidates detecter point randomly generated
        if all(np.linalg.norm(candidate - x) > radius for x in X_self): # np.linalg.norm is Euclidean distance
            detectors.append(candidate)
    return detectors
```

Prediction Function

- This function makes predictions based on the detector locations.
- For each test data point x, the function checks if it is within a radius distance of any of the detector points.
- If x is within the radius of a detector, it is classified as "damaged" (1). Otherwise, it is classified as "safe" (0).

In [5]:
```python
def predict_nsa(X, detectors, radius=0.1):
    predictions = []
    for x in X:
        # If close to any detector, classify as "damaged" (1)
        if any(np.linalg.norm(x - d) <= radius for d in detectors):
            predictions.append(1)
        else:
            predictions.append(0)  # Else "safe" (0)
    return np.array(predictions)
```

Training and Testing

- X_train, X_test = X[:split], X[split:] splits the data into an 80% training set and a 20% test set.
- y_train, y_test = y[:split], y[split:] splits the labels accordingly.
- X_self = X_train[y_train == 0] extracts all data points from X_train where the label is 0 (safe points).

```
In [8]: X,y = generate_data()
        split = int(0.8 * len(X))
        X_train, X_test = X[:split], X[split:] # split train and test of X into 80 : 20
        y_train, y_test = y[:split], y[split:] # split lables into 80% train 20% test
        X_self = X_train[y_train == 0] # only takes X_train where y_train is 0

        # Example
        # Suppose X = [[0.32, 0.79], [0.94, 0.68], [0.12, 0.24], [0.45, 0.91], [0.89, 0.10]] and y = [0, 1, 0, 0, 0]. Th
        # X_train = [[0.32, 0.79], [0.94, 0.68], [0.12, 0.24]]
        # X_self = [[0.32, 0.79], [0.12, 0.24]]
```

Train NSA Model and Make Predictions

- detectors = train_nsa(X_self, num_detectors=50, radius=0.1) trains the NSA model by finding 50 detectors from the "safe" training points.
- y_pred = predict_nsa(X_test, detectors, radius=0.1) uses the trained detectors to make predictions on the test set X_test.

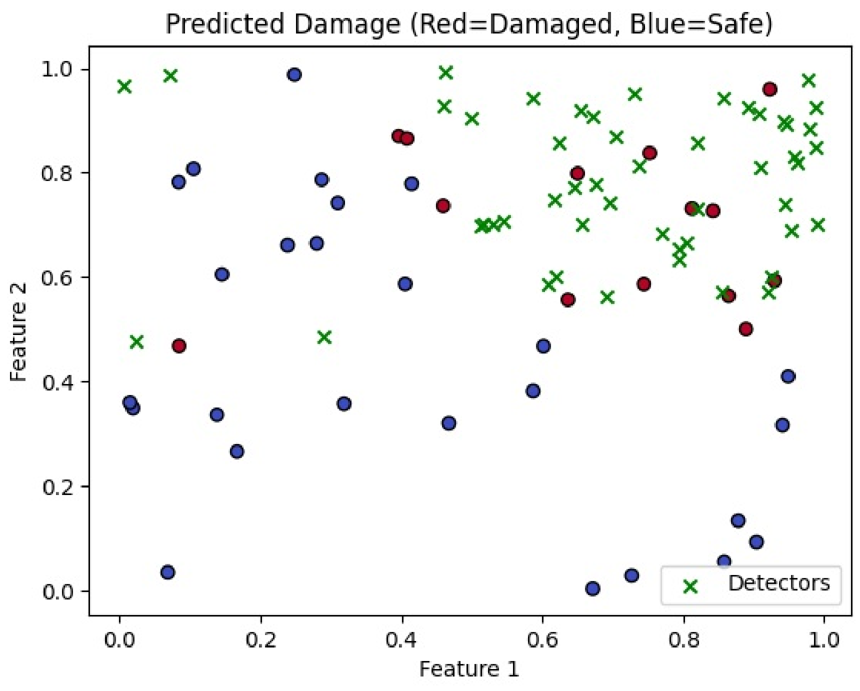
```
In [9]: detectors = train_nsa(X_self, num_detectors=50, radius=0.1)
        y_pred = predict_nsa(X_test, detectors, radius=0.1)
```

Calculate Accuracy

```
In [10]: acc = np.mean(y_pred == y_test)
         print("Accuracy:", acc)

Accuracy: 0.9
```

Visualization

```
In [11]: plt.scatter(X_test[:, 0], X_test[:, 1], c=y_pred, cmap='coolwarm', edgecolor='k')
         plt.scatter(np.array(detectors)[:, 0], np.array(detectors)[:, 1], c='green', marker='x', label='Detectors')
         plt.xlabel('Feature 1')
         plt.ylabel('Feature 2')
         plt.title('Predicted Damage (Red=Damaged, Blue=Safe)')
         plt.legend()
         plt.show()
```



```
In [ ]:
```

Implement DEAP

- Implement DEAP on fitness_function (sphere function) x**2

1 Imports

- **random**: To generate random values
- **deap**: Distributed evolutionary algorithm.
    1. **base**: Provides base classes and toolboxes used to define algorithms
    2. **creator**: Used to create custom dynamic classes
    3. **algorithms**: `varAND` provides predefined algorithms
    4. **tools**: Provides ready-to-use functions like mutation, evaluation, etc.

In [1]:
```
import random
from deap import creator, base, algorithms, tools
```

2 Fitness function

- input individual
- output sum of squares of all elements in individuals
- return tuple example : [10,] and not 10

In [2]:
```
def fitness_function(individual) :
    return sum(x**2 for x in individual),
```

3 Custom Classes

- FitnessMin : class used to minimize the function
- Individual : List with .fitness attribute

In [3]:
```
creator.create("FitnessMin", base.Fitness, weights = (-1.0,))
creator.create("Individual", list, fitness = creator.FitnessMin)
```

4 Toolbox setup

In [4]:
```
toolbox = base.Toolbox()
```

5 Registering some functions to toolbox

- attr_float
- individual
- population

In [5]:
```
toolbox.register("attr_float", random.uniform, -5.0, 5.0)
toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox.attr_float, n=3)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
```

6 Functions

1. Evaluation
2. Mutation
3. Selection
4. Crossover

In [6]:
```
toolbox.register("evaluate", fitness_function)
toolbox.register("mate", tools.cxBlend, alpha = 0.5)
toolbox.register("mutate", tools.mutGaussian, mu=0, sigma=1, indpb = 0.2)
toolbox.register("select", tools.selTournament, tournsize = 3)
```

7️⃣Generated population

```
In [7]: population = toolbox.population(n=50)
```

8️⃣Evolution for 20 Generations

- varAND algorithms manages crossover probability and mutation

```
In [12]: generation = 20
         for gen in range(generation) :
             offspring = algorithms.varAnd(population, toolbox, cxpb=0.5, mutpb=0.1)
             fits = toolbox.map(toolbox.evaluate, offspring)
             for fit, ind in zip(fits, offspring) :
                 ind.fitness.values = fit
             population = toolbox.select(offspring, k=len(population))
```

```
In [13]: best_ind = tools.selBest(population, k=1)[0]
         best_fitness = best_ind.fitness.values[0]
         print(best_ind)
         print(f"{best_fitness:.5f}")
```

```
[0.0024871492396969733, 0.0002426028547953176, -0.015107476000039795]
0.00023
```

```
In [ ]:
```

```
In [ ]:
```

AIM -

To Implement - Fuzzy Logic Operations

PROBLEM STATEMENT -

Implement

1. Union
2. Intersection
3. Complement
4. Difference operations on fuzzy sets.
5. Also create fuzzy relations by Cartesian product of any two fuzzy sets and perform max-min composition on any two fuzzy relations.

❄Importing required Libraries

1. numpy - Provide numerical computation

In [1]:
```
import numpy as np
```

❄Fuzzy sets

In [3]:
```
A = np.array([0.2, 0.4, 0.7, 0.8,0.4,0.5])
B = np.array([0.1, 0.8, 0.2, 0.3,0.6,0.3])
```

❄Fuzzy Operation -

1. OR fuzzy operation / Union
2. And fuzzy operation / Intersection
3. Not fuzzy operation / Complement
4. Differences of A and B fuzzy set
5. Addition of A and B fuzzy set
6. Cartesian Product
7. max-min composition

In [4]:
```
def union(A,B) :
    return np.minimum(A,B)
```

In [5]:
```
def intersection(A,B) :
    return np.maximum(A,B)
```

In [6]:
```
def complement(A) :
    return 1-A
```

In [7]:
```
def difference(A,B) :
    return np.maximum(A - B,0)
```

In [8]:
```
def addition(A,B) :
    return np.minimum(A + B,1)
```

In [11]:
```
def cartesian_product(A, B):
    return np.outer(A, B)
```

In [10]:
```
def max_min_composition(R1, R2):
    return np.maximum(np.minimum(R1, R2), 0)
```

❄Storing results

In [13]:
```
R1 = cartesian_product(A,B)
R2 = cartesian_product(B,A)
result = max_min_composition(R1,R2)
```

❄Displaying results

In [54]:
```
print("+" * 54)
print("Fuzzy operations".center(54))
print("+" * 54)
print(f"| Union of A and B        : {union(A,B)}|")
print(f"| Intersection of A and B : {intersection(A,B)}|")
print(f"| Complement of A and B   : {complement(A)}|")
print(f"| Union of A and B        : {difference(A,B)}|")
print(f"| Union of A and B        : {addition(A,B)}|")
print("+" * 54)
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++
                   Fuzzy operations
++++++++++++++++++++++++++++++++++++++++++++++++++++++
| Union of A and B        : [0.1 0.4 0.2 0.3 0.4 0.3]|
| Intersection of A and B : [0.2 0.8 0.7 0.8 0.6 0.5]|
| Complement of A and B   : [0.8 0.6 0.3 0.2 0.6 0.5]|
| Union of A and B        : [0.1 0.  0.5 0.5 0.  0.2]|
| Union of A and B        : [0.3 1.  0.9 1.  1.  0.8]|
++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

In [66]:
```
print("+" * 54)
print("Cartesian product : A x B".center(54))
print("+" * 54)
print(R1)
print("+" * 54)
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++
              Cartesian product : A x B
++++++++++++++++++++++++++++++++++++++++++++++++++++++
[[0.02 0.16 0.04 0.06 0.12 0.06]
 [0.04 0.32 0.08 0.12 0.24 0.12]
 [0.07 0.56 0.14 0.21 0.42 0.21]
 [0.08 0.64 0.16 0.24 0.48 0.24]
 [0.04 0.32 0.08 0.12 0.24 0.12]
 [0.05 0.4  0.1  0.15 0.3  0.15]]
++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

In [67]:
```
print("+" * 54)
print("Cartesian product : B x A".center(54))
print("+" * 54)
print(R2)
print("+" * 54)
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++
              Cartesian product : B x A
++++++++++++++++++++++++++++++++++++++++++++++++++++++
[[0.02 0.04 0.07 0.08 0.04 0.05]
 [0.16 0.32 0.56 0.64 0.32 0.4 ]
 [0.04 0.08 0.14 0.16 0.08 0.1 ]
 [0.06 0.12 0.21 0.24 0.12 0.15]
 [0.12 0.24 0.42 0.48 0.24 0.3 ]
 [0.06 0.12 0.21 0.24 0.12 0.15]]
++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

In [69]:
```
print("+" * 54)
print("max-min composition".center(54))
print("+" * 54)
print(result)
print("+" * 54)
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++
                 max-min composition
++++++++++++++++++++++++++++++++++++++++++++++++++++++
[[0.02 0.04 0.04 0.06 0.04 0.05]
 [0.04 0.32 0.08 0.12 0.24 0.12]
 [0.04 0.08 0.14 0.16 0.08 0.1 ]
 [0.06 0.12 0.16 0.24 0.12 0.15]
 [0.04 0.24 0.08 0.12 0.24 0.12]
 [0.05 0.12 0.1  0.15 0.12 0.15]]
++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

In [ ]: