LNMIIT
The LNM Institute of Information Technology

Soft Computing Project

Project report submitted By

Nayan Goyal – 21UCS137

Rajdeep Tiwari – 21UCS166

Sahil Sidana – 21UCS177

Shreyansh Jain – 21UCS198

Course Instructor

Dr. Aloke Datta

Project Folder Link: SC_Project

Problem 1: Design a fuzzy logic controller using Mamdani's approach to determine the wash time of a domestic washing machine, assuming that inputs are dirt and grease on clothes. Use 5 descriptor for dirt (i.e., VSD, SD, MD, HD, VHD) and 3 descriptor for grease (i.e., SG, MG, HG) and 5 descriptors for output variables of wash time (i.e., VST, ST, MT, HT, VHT). Use triangular membership functions for fuzzy classes and CoG method for defuzzification operations. The set of rules for fuzzy logic tables are given below.

	SG	MG	HG
VSD	VST	VST	ST
SD	VST	ST	MT
MD	ST	MT	HT
HD	MT	HT	VHT
VHD	HT	VHT	VHT

The input grease is given in the scale of 0 to 50, whereas dirt is in the scale 0 to 100. Output should be
in the range 0 to 60 min. Write a program to find the wash time in minutes for any value of grease and
dirt within the range.

Solution 1:- Implemented in Matlab Fuzzy Logic Designer.

Using 5 descriptor for dirt (i.e., VSD, SD, MD, HD, VHD) and 3 descriptor for grease (i.e., SG, MG, HG) and 5 descriptors for output variables of wash time (i.e., VST, ST, MT, HT, VHT).

Input variables:

Step 1: Dirt = Very Small Dirt, Small Dirt, Medium Dirt, High Dirt, Very High Dirt.

= < VSD, SD, MD, HD, VHD >

Grease = Small Grease, Medium Grease, High Grease.

= < SG, MG, HG >

Output variable:

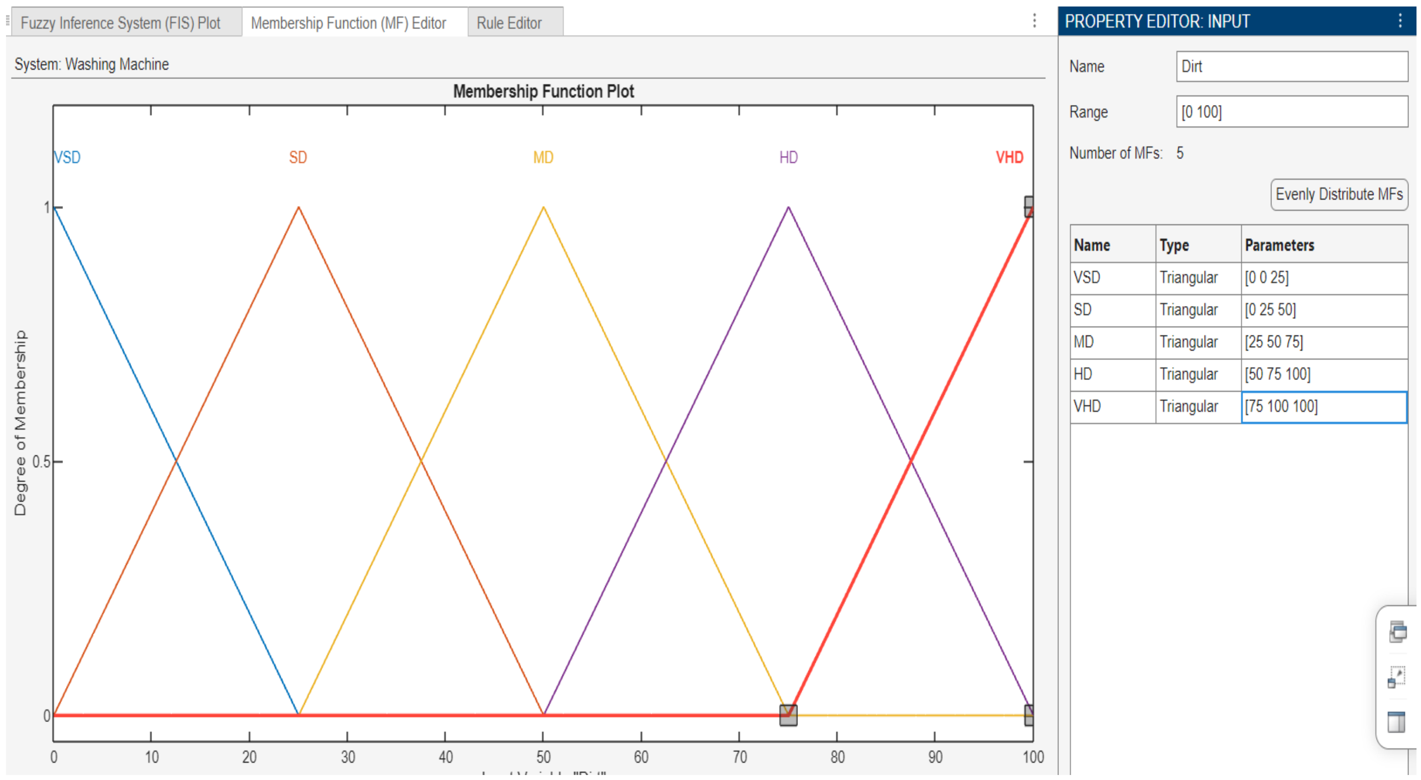
Wash time = Very Small Time, Small Time, Medium Time, High Time, Very High Time.

= < VST, ST, MT, HT, VHT >

Step 2: Membership function for each i/p and o/p variable.

Fuzzification:

i/p1- Dirt:



System: Washing Machine

PROPERTY EDITOR: INPUT

Name	Dirt
Range	[0 100]
Number of MFs:	5

Name	Type	Parameters
VSD	Triangular	[0 0 25]
SD	Triangular	[0 25 50]
MD	Triangular	[25 50 75]
HD	Triangular	[50 75 100]
VHD	Triangular	[75 100 100]

Using Triangular Membership Function with Evenly Distribute MF's

Values of a,b,c for each is:

VSD - a=0,b=0,c=25

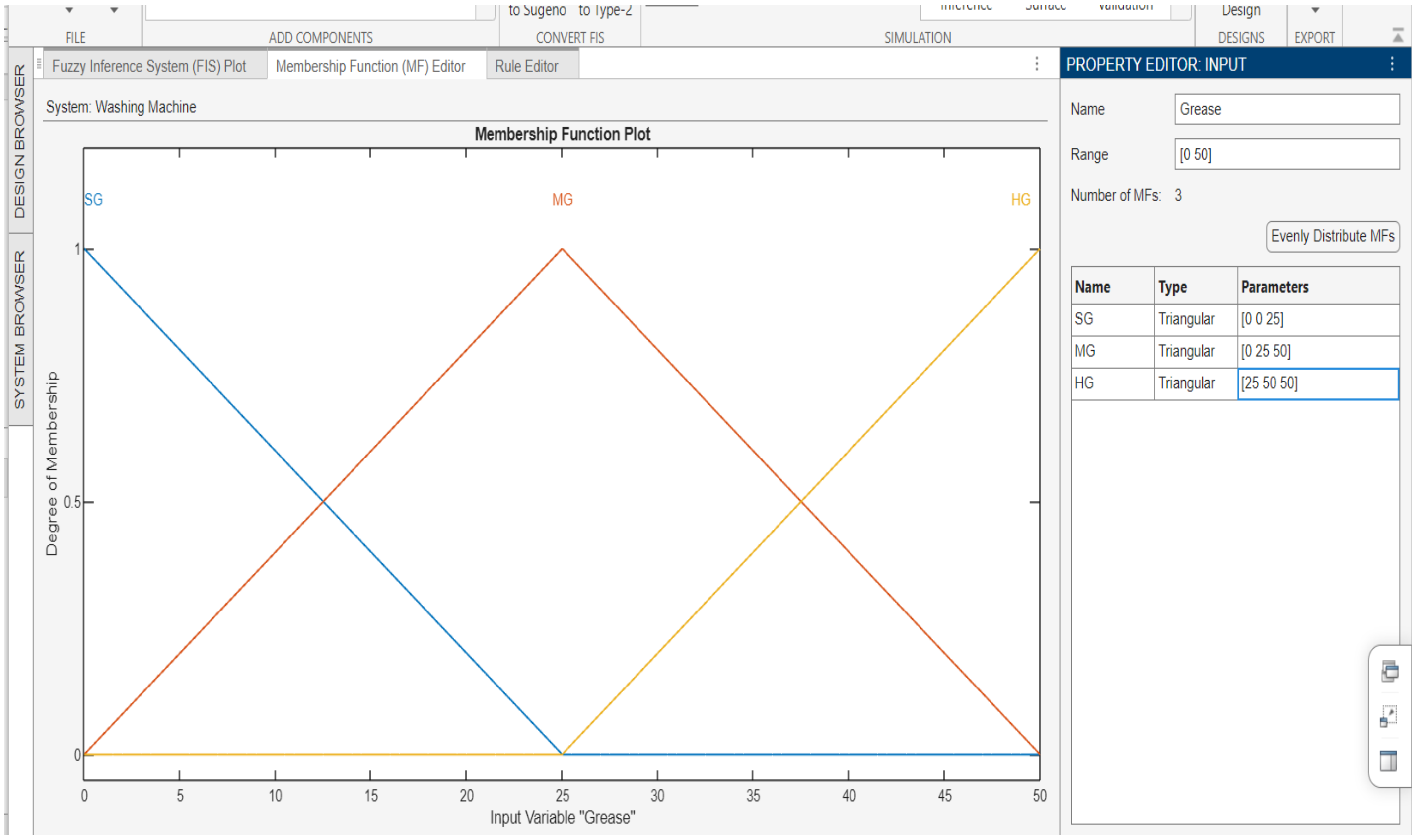
SD - a=0,b=25,c=50

MD - a=25,b=50,c=75

HD - a=50,b=75,c=100

VHD - a=75,b=100,c=100

i/p2- Grease:



System: Washing Machine

Name: Grease

Range: [0 50]

Number of MFs: 3

Name	Type	Parameters
SG	Triangular	[0 0 25]
MG	Triangular	[0 25 50]
HG	Triangular	[25 50 50]

Using Triangular Membership Function with Evenly Distribute MF's

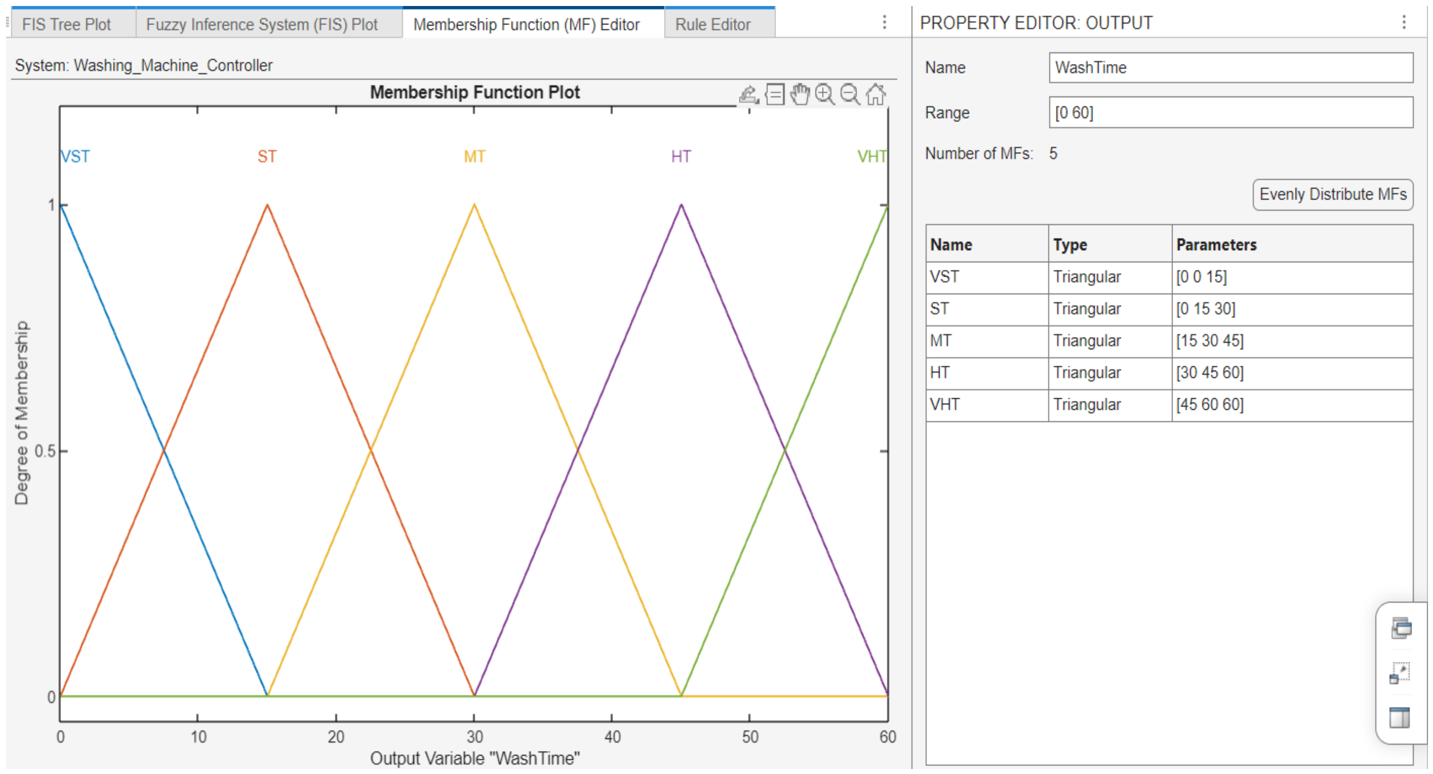
Values of a,b,c for each is:

SG - a=0,b=0,c=25

MG - a=0,b=25,c=50

HG - a=25,b=50,c=50

o/p1- Time:



System: Washing_Machine_Controller

PROPERTY EDITOR: OUTPUT

Name: WashTime

Range: [0 60]

Number of MFs: 5

Name	Type	Parameters
VST	Triangular	[0 0 15]
ST	Triangular	[0 15 30]
MT	Triangular	[15 30 45]
HT	Triangular	[30 45 60]
VHT	Triangular	[45 60 60]

Using Triangular Membership Function with Evenly Distribute MF’s

Values of a,b,c for each is:

VST - a=0,b=0,c=15

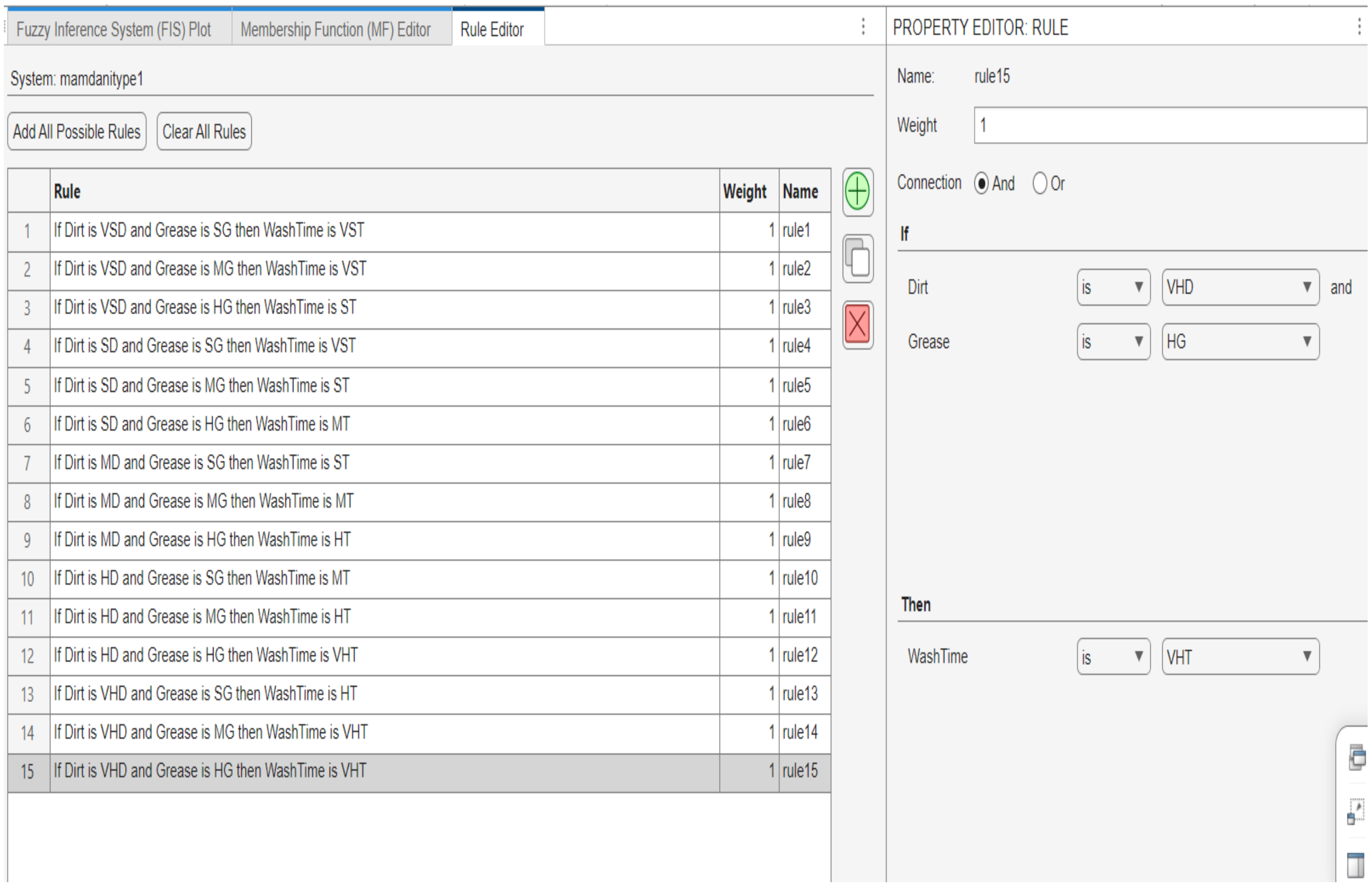
ST - a=0,b=15,c=30

MT - a=15,b=30,c=45

HT - a=30,b=45,c=60

VHT - a=45,b=60,c=60

Step-3: Creating all possible Rules from Table given in Problem.


Fuzzy Inference System (FIS) Plot
Membership Function (MF) Editor
Rule Editor
Add All Possible Rules
Clear All Rules

System: mamdanitype1

	Rule	Weight	Name
1	If Dirt is VSD and Grease is SG then WashTime is VST	1	rule1
2	If Dirt is VSD and Grease is MG then WashTime is VST	1	rule2
3	If Dirt is VSD and Grease is HG then WashTime is ST	1	rule3
4	If Dirt is SD and Grease is SG then WashTime is VST	1	rule4
5	If Dirt is SD and Grease is MG then WashTime is ST	1	rule5
6	If Dirt is SD and Grease is HG then WashTime is MT	1	rule6
7	If Dirt is MD and Grease is SG then WashTime is ST	1	rule7
8	If Dirt is MD and Grease is MG then WashTime is MT	1	rule8
9	If Dirt is MD and Grease is HG then WashTime is HT	1	rule9
10	If Dirt is HD and Grease is SG then WashTime is MT	1	rule10
11	If Dirt is HD and Grease is MG then WashTime is HT	1	rule11
12	If Dirt is HD and Grease is HG then WashTime is VHT	1	rule12
13	If Dirt is VHD and Grease is SG then WashTime is HT	1	rule13
14	If Dirt is VHD and Grease is MG then WashTime is VHT	1	rule14
15	If Dirt is VHD and Grease is HG then WashTime is VHT	1	rule15

PROPERTY EDITOR: RULE

Name: rule15

Weight: 1

Connection: [● And] ○ Or

If

Dirt is VHD and

Grease is HG

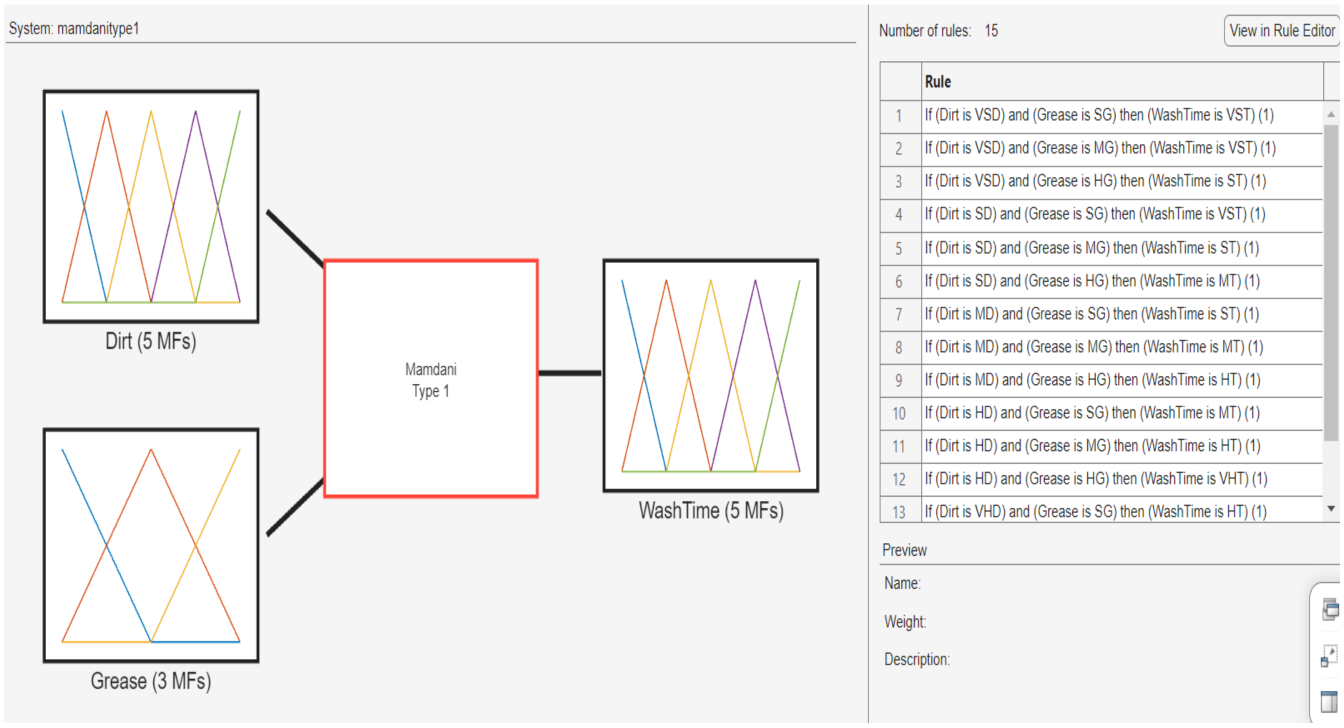
Then

WashTime is VHT

Describe Rules in form like:

If dirt is ___ and grease is ___ then time is ___.

Fuzzy Inference System View in Matlab.



Number of rules: 15

	Rule
1	If (Dirt is VSD) and (Grease is SG) then (WashTime is VST) (1)
2	If (Dirt is VSD) and (Grease is MG) then (WashTime is VST) (1)
3	If (Dirt is VSD) and (Grease is HG) then (WashTime is ST) (1)
4	If (Dirt is SD) and (Grease is SG) then (WashTime is VST) (1)
5	If (Dirt is SD) and (Grease is MG) then (WashTime is ST) (1)
6	If (Dirt is SD) and (Grease is HG) then (WashTime is MT) (1)
7	If (Dirt is MD) and (Grease is SG) then (WashTime is ST) (1)
8	If (Dirt is MD) and (Grease is MG) then (WashTime is MT) (1)
9	If (Dirt is MD) and (Grease is HG) then (WashTime is HT) (1)
10	If (Dirt is HD) and (Grease is SG) then (WashTime is MT) (1)
11	If (Dirt is HD) and (Grease is MG) then (WashTime is HT) (1)
12	If (Dirt is HD) and (Grease is HG) then (WashTime is VHT) (1)
13	If (Dirt is VHD) and (Grease is SG) then (WashTime is HT) (1)

Name:

Weight:

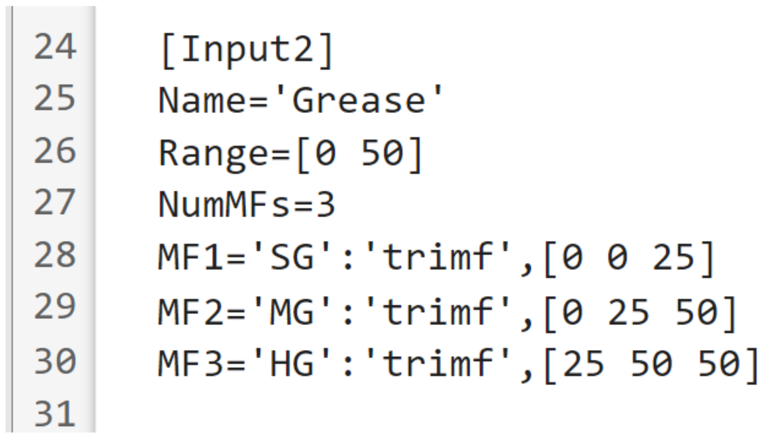
Description:

Matlab Code Generated:

1. i/p1:

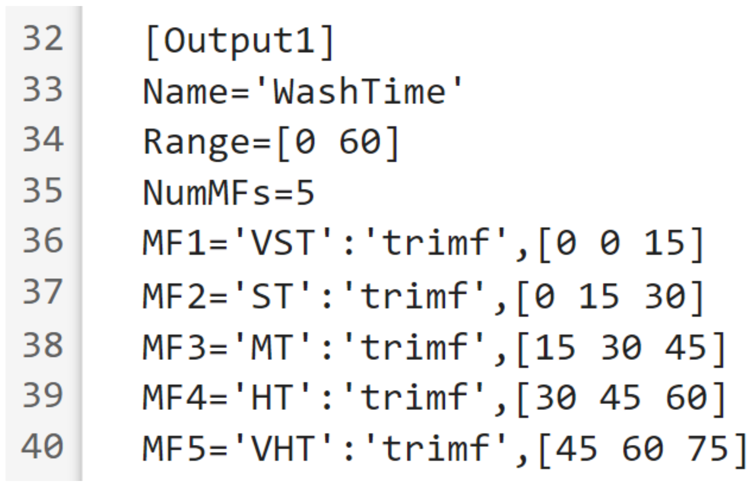
```
[Input1]
Name='Dirt'
Range=[0 100]
NumMFs=5
MF1='VSD':'trimf',[0 0 25]
MF2='SD':'trimf',[0 25 50]
MF3='MD':'trimf',[25 50 75]
MF4='HD':'trimf',[50 75 100]
MF5='VHD':'trimf',[75 100 100]
```

2.i/p2:



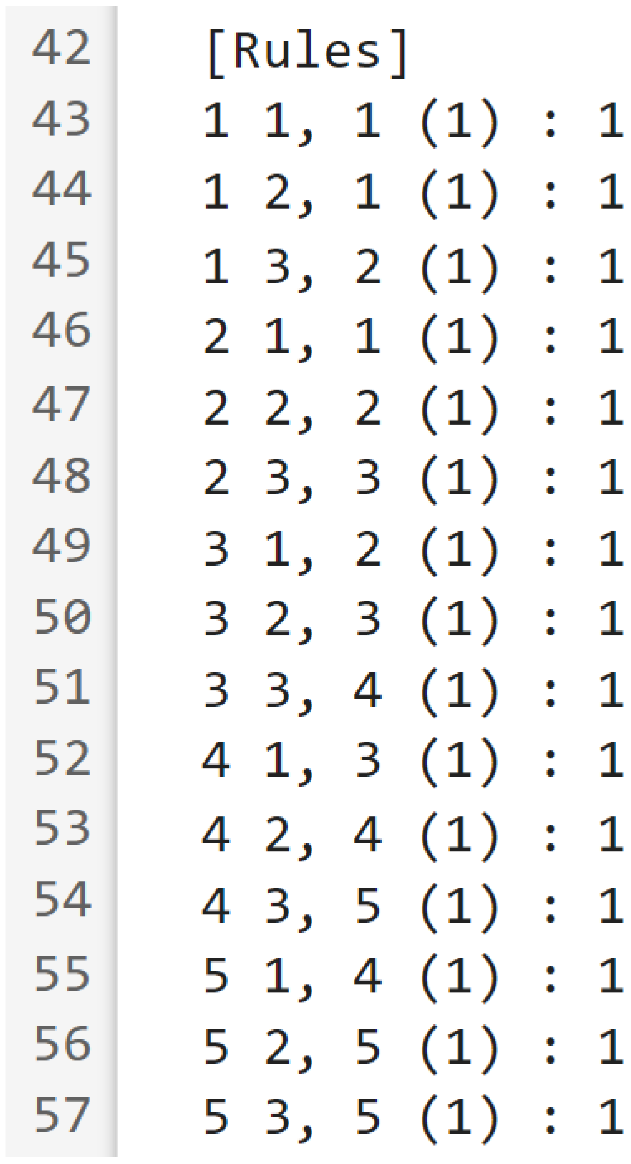
```
[Input2]
Name='Grease'
Range=[0 50]
NumMFs=3
MF1='SG':'trimf',[0 0 25]
MF2='MG':'trimf',[0 25 50]
MF3='HG':'trimf',[25 50 50]

```

3.o/p:



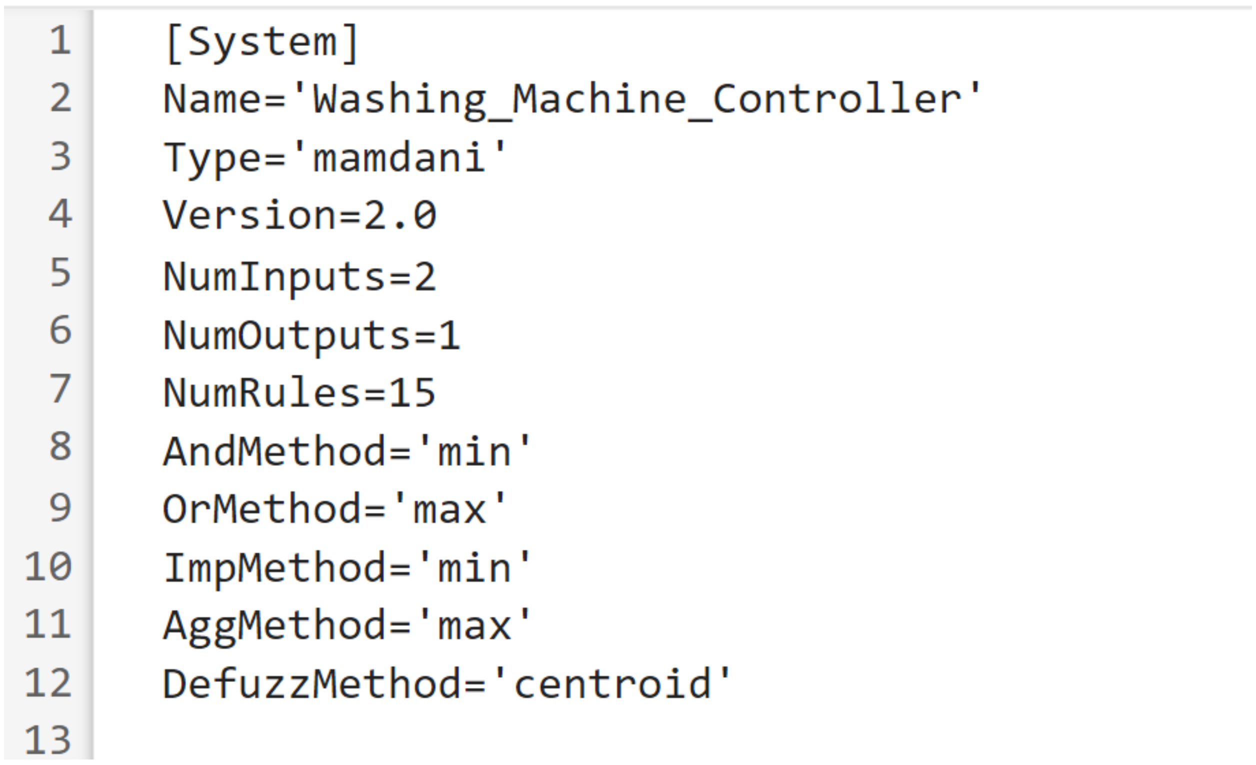
```
[Output1]
Name='WashTime'
Range=[0 60]
NumMFs=5
MF1='VST':'trimf',[0 0 15]
MF2='ST':'trimf',[0 15 30]
MF3='MT':'trimf',[15 30 45]
MF4='HT':'trimf',[30 45 60]
MF5='VHT':'trimf',[45 60 75]
```

4.Rules:



```
[Rules]
1 1, 1 (1) : 1
1 2, 1 (1) : 1
1 3, 2 (1) : 1
2 1, 1 (1) : 1
2 2, 2 (1) : 1
2 3, 3 (1) : 1
3 1, 2 (1) : 1
3 2, 3 (1) : 1
3 3, 4 (1) : 1
4 1, 3 (1) : 1
4 2, 4 (1) : 1
4 3, 5 (1) : 1
5 1, 4 (1) : 1
5 2, 5 (1) : 1
5 3, 5 (1) : 1
```

5. System:



```
[System]
Name='Washing_Machine_Controller'
Type='mamdani'
Version=2.0
NumInputs=2
NumOutputs=1
NumRules=15
AndMethod='min'
OrMethod='max'
ImpMethod='min'
AggMethod='max'
DefuzzMethod='centroid'

```

Working through the Environment:

How to run a .fis file in matlab ?

Open the Fuzzy Logic Designer App

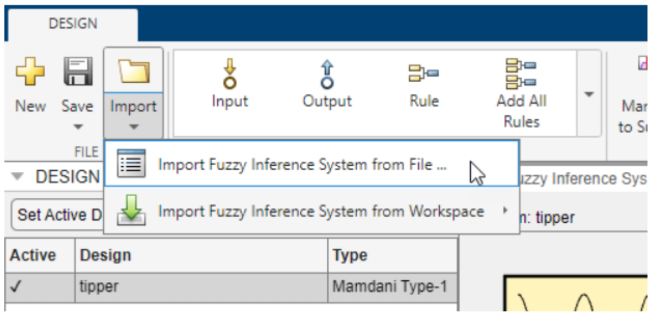
- MATLAB Toolstrip: On the **Apps** tab, under **Control System Design and Analysis**, click the app icon.
- MATLAB command prompt: Enter `fuzzyLogicDesigner`.

Examples

> Import FIS from Workspace

⌄ Import FIS from File

In Fuzzy Logic Designer, select **Import** > **Import Fuzzy Inference System from File**.



Active	Design	Type
✓	tipper	Mamdani Type-1

Then, in the Import Fuzzy Inference System dialog box, select a FIS or MAT file and click **Open**.

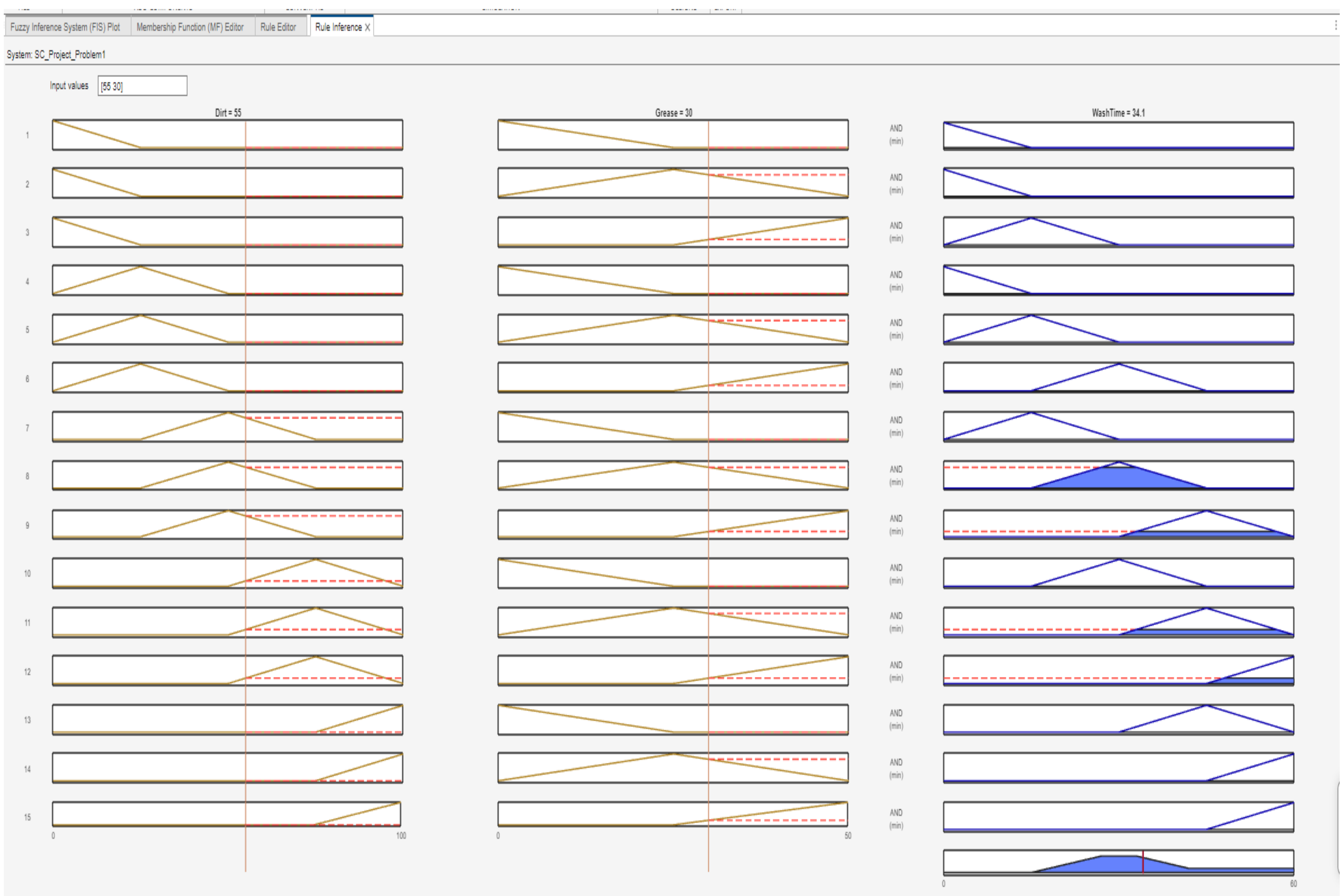
1.Type fuzzyLogicDesigner in command prompt and press enter.

2.Import the .fis file(Fuzzy Inference System) that you want to run.

Working through some examples:

Ex1: i/p1- Dirt=55, i/p2-Grease=30

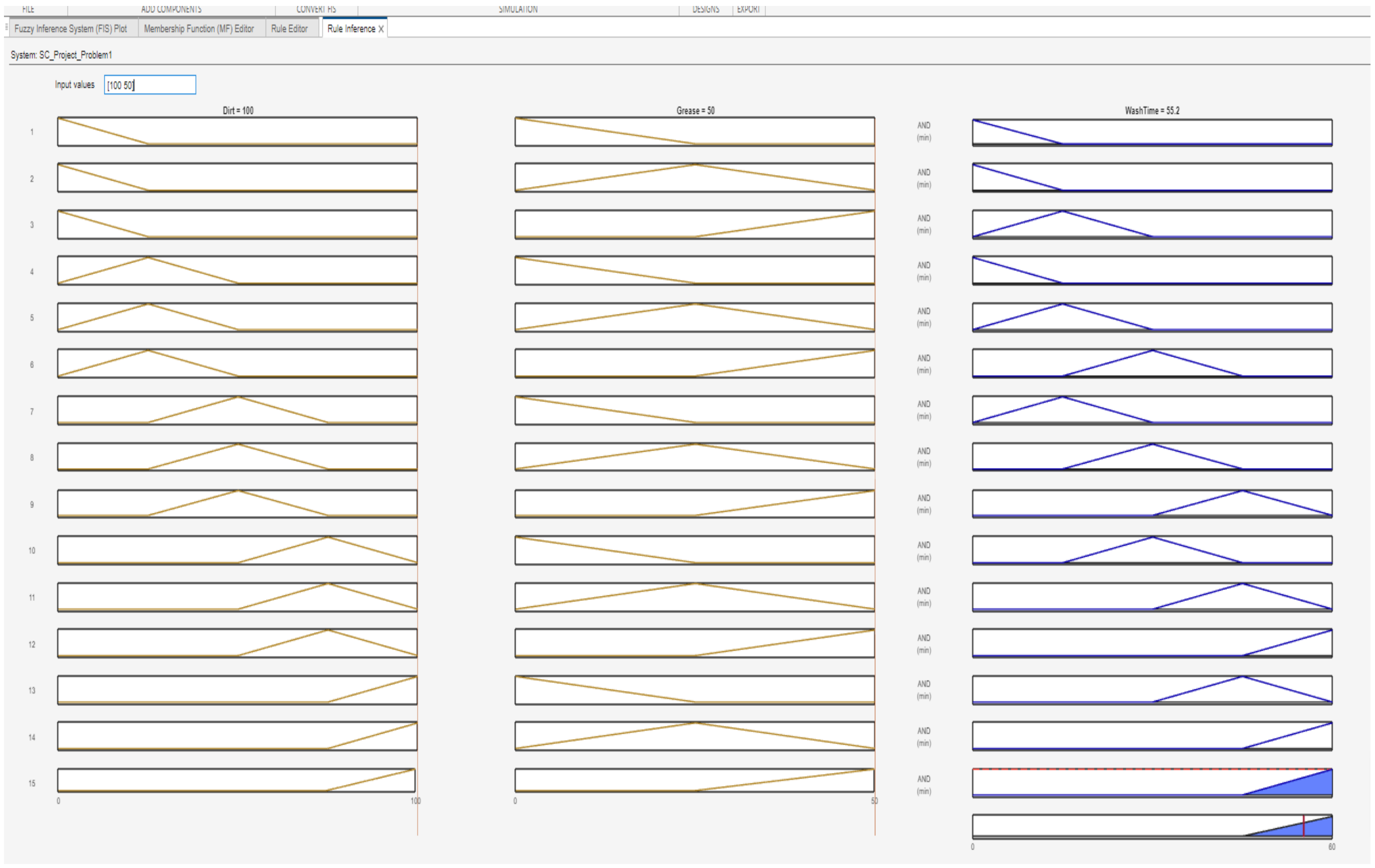
Simulating Rule Inference and giving input as [55 30]. We get:


Fuzzy Inference System (FIS) Plot
Membership Function (MF) Editor
Rule Editor
Rule Inference
System: SC_Project_Problem1
Input values [55 30]
Dirt = 55
Grease = 30
WashTime = 34.1
AND (min)

We conclude wash time as 34.1 minutes.

Ex2: i/p1- Dirt=100, i/p2-Grease=50 (Maximum Values)

Simulating Rule Inference and giving input as [100 50]. We get:



System: SC_Project_Problem1

Input values [100 50]

Dirt = 100

Grease = 50

WashTime = 55.2

We conclude wash time as 55.2 minutes.

Problem 2: Solve the traveling salesman problem using GA. The number of cities is 15. The weight matrix will be given as input, and it represents the distance of each city. After 20 iterations you will stop the algorithm. The number of chromosomes is 15. Output is the sequence of cities to travel by the salesman (best solution given by GA). You may consider elitism principle and roulette-wheel based selection procedure. You can go for one point crossover and crossover points will be randomly selected. Take extra care so that each city will represent once in the solution.

Solution 2 :- This problem is solved by applying genetic algorithms.

The Algorithm utilizes a weight matrix as an input,representing the distances between the cities.

Key components of this algorithm include elitism,roulette wheel selection procedure,single point crossover and preventing the occurrence of the same city more than one time in solution.

This algorithm is applied to get the optimal solution.

Input Variables:

No Of Cities=15

No Of Chromosomes (Population Size)=15

No Of Iterations=20

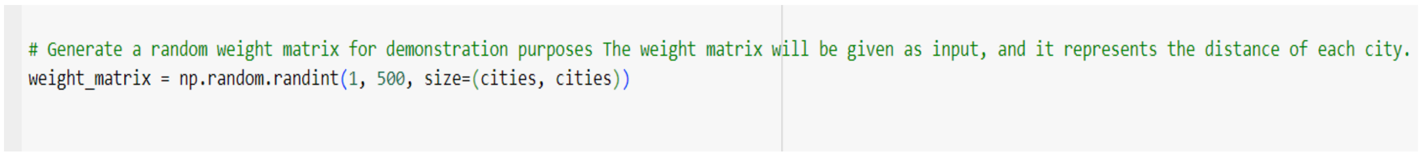
Weight Matrix (Randomly Generated and Then given as an input to the Genetic Algorithm)

Output Variable:

Complete Optimal path with cost of each path travelled and the total optimal cost.

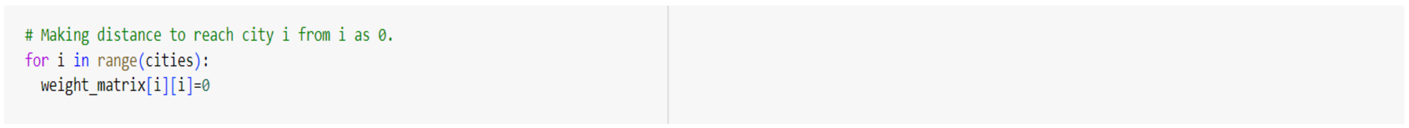
Step 1:- Generates The Weight Matrix:

The Weight Matrix is generated by Random Integers in the range of 1 and 500 where each integer represents the distance between a pair of cities using the following code:



```
# Generate a random weight matrix for demonstration purposes The weight matrix will be given as input, and it represents the distance of each city.
weight_matrix = np.random.randint(1, 500, size=(cities, cities))
```

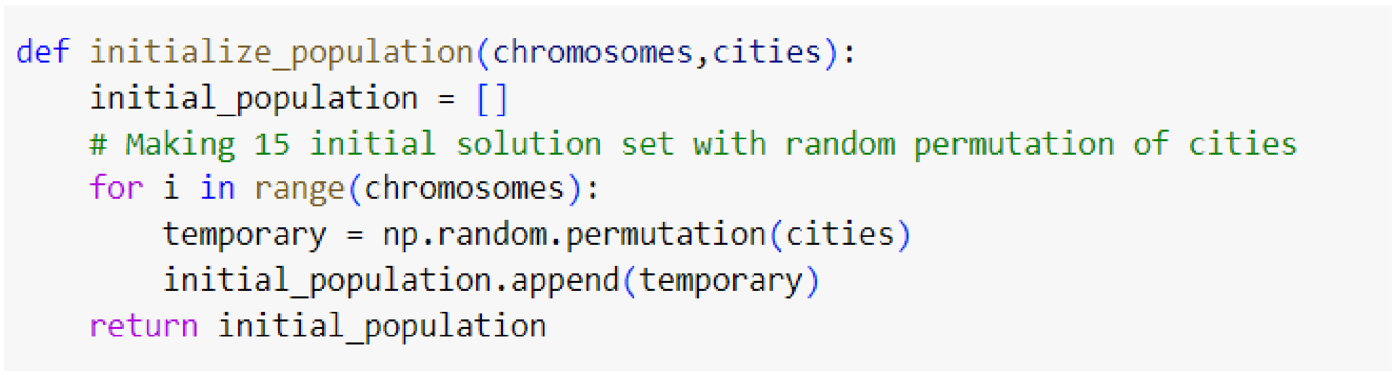
The following piece of code initializes the weight_matrix with value 0:



```
# Making distance to reach city i from i as 0.
for i in range(cities):
  weight_matrix[i][i]=0
```

Step 2:- Initialization The Population:

In order to start the algorithm,we need to first generate an initial population.This is done using the following piece of code:



```python
def initialize_population(chromosomes,cities):
    initial_population = []
    # Making 15 initial solution set with random permutation of cities
    for i in range(chromosomes):
        temporary = np.random.permutation(cities)
        initial_population.append(temporary)
    return initial_population
```

Here a function named initialize_population is created which takes the number of chromosomes and the number of cities as inputs.

We have taken an array initial_population to store each initial possible solution. Basically it stores a randomly generated permutation of cities (a possible path travelled) like: 6->5->14->4->7->8->10->3->12->13->11->15->2->9->1

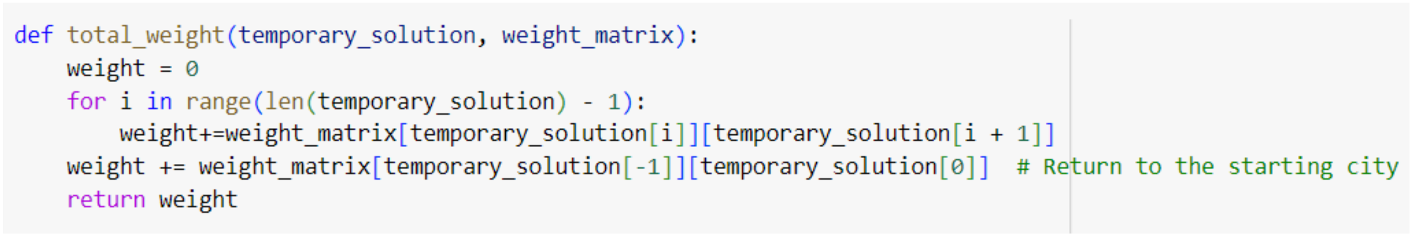
And in this way it stores a total of 15 initial chromosomes(a possible path).

And then it returns initial_population.

Now run the algorithm 20 times->

Step 3:- Calculating the cost of each path:

For each of the possible solution in the population array,we will calculate the total path cost using the following piece of code:



```
def total_weight(temporary_solution, weight_matrix):
    weight = 0
    for i in range(len(temporary_solution) - 1):
        weight+=weight_matrix[temporary_solution[i]][temporary_solution[i + 1]]
    weight += weight_matrix[temporary_solution[-1]][temporary_solution[0]]  # Return to the starting city
    return weight
```

Here a function named total_weight is defined which takes temporary_solution (one possible path or solution from population array) and weight_matrix as inputs.

A variable named weight is initialized to 0 which represents the total path cost.

Now for each consecutive pair of cities in the given path, take the distance between these two cities from the weight_matrix and then add it to the weight variable.

And then it returns this weight variable which will be stored in an array (shown further below).

This weight will be used to select the most suitable chromosome.

This weight value will be used in the calculation of fitness.

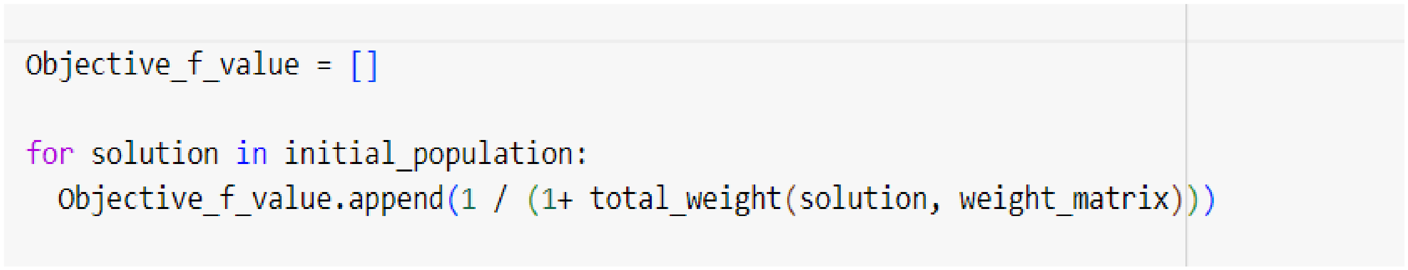
Step 4:- Calculating Fitness Value:

To select the best chromosome out of all the population,we will need some criteria.This criteria is the fitness value for each chromosome (possible path).

For each chromosome in the initial population,we will calculate the fitness value to determine the best chromosome

Fitness value= (1/(1+total cost of a chromosome(path)))

This is done using the following piece of code:



```
Objective_f_value = []

for solution in initial_population:
  Objective_f_value.append(1 / (1+ total_weight(solution, weight_matrix)))
```

Here we have taken an array named Objective_f_value to store the f values of each chromosome.

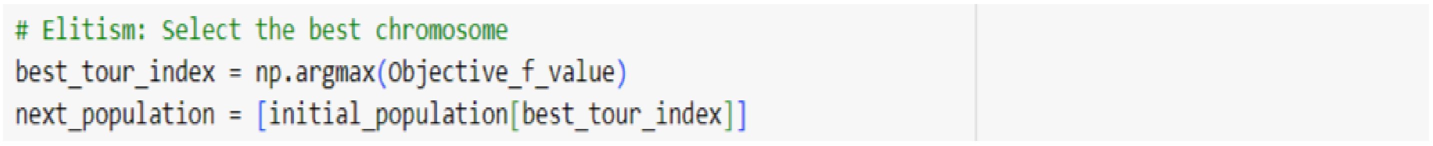
Step 5:- Elitism:

Elitism involves preserving the best - performing individuals (chromosomes) from one generation to the next

This ensures that the fittest solutions are not lost during the evaluation process.

The idea is to maintain a certain level of diversity in the population while giving preference to the best solutions obtained so far.

This is done using the following piece of code:



```
# Elitism: Select the best chromosome
best_tour_index = np.argmax(Objective_f_value)
next_population = [initial_population[best_tour_index]]
```

Here the best_tour_index variable stores the index of the chromosome with the highest fitness (lowest total weight in this context).

The best performing chromosome is selected from the current population based on its index (best_tour_index).

This chromosome is added to the list of the next population.

Step 6:- Roulette Wheel Selection Procedure:

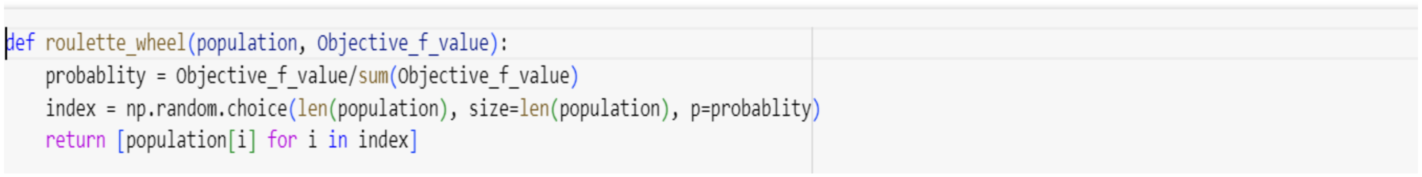
A stochastic selection procedure that used a fitness based probability system to determine an individual’s likelihood of being chosen.

With it ,an individual’s selection is more likely the more fit it is.

The first component of this method is that an individual fitness is proportional to its chances of being chosen.

This is not sufficient though.This is because,if the population has n individuals,then the summation of probabilities of their selection equals one. As a consequence,we have to normalize all values for the individual probabilities.

Roulette wheel selection is implemented by using the following piece of code:



```
def roulette_wheel(population, Objective_f_value):
    probablity = Objective_f_value/sum(Objective_f_value)
    index = np.random.choice(len(population), size=len(population), p=probablity)
    return [population[i] for i in index]
```

Here a function named roulette_wheel is defined which takes the entire population and the Objective_f_value as input.

In this function,a variable named probability is used to calculate the probability assigned for each individual by dividing each individual’s fitness (Objective_f_value) by total fitness (sum(Objective_f_value)), creating a probability distribution.

Also, np.random.choice function is used. This function generates a random sample from a given 1-D array.

It takes (len(population) as an input which means that it will arrange evenly spaced values within the range of (1-len(population)).

It takes size=len(population) as another input which indicates how many individuals to choose p=probability and assigns the probabilities calculated earlier to each individual.

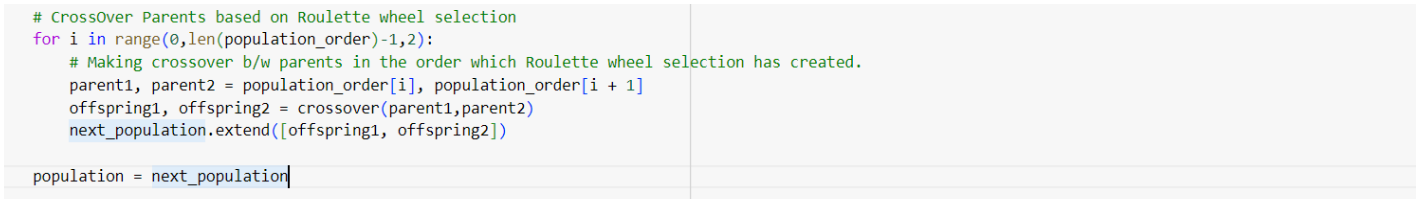
The roulette_wheel function returns a list containing the selected individuals from the population and stores it in a list named population_order.

A random selection with probabilities corresponding to the fitness values is essentially carried out, with the selection being based on the indices produced by np.random.choice.

Step 7:- Crossover Operator:

Crossover is a reproduction process.Two chromosomes are randomly selected from the mating pool to cross over in order to produce superior offspring. Crossover is a genetic operator used to vary the programming of a chromosome from one generation to the next.It generates only two offspring.

This is done through the following piece of code:



```
# CrossOver Parents based on Roulette wheel selection
for i in range(0,len(population_order)-1,2):
    # Making crossover b/w parents in the order which Roulette wheel selection has created.
    parent1, parent2 = population_order[i], population_order[i + 1]
    offspring1, offspring2 = crossover(parent1,parent2)
    next_population.extend([offspring1, offspring2])

population = next_population
```

The above piece of code is performing crossover between pairs of parents selected through roulette wheel selection procedure(Step 6) to create new offspring.

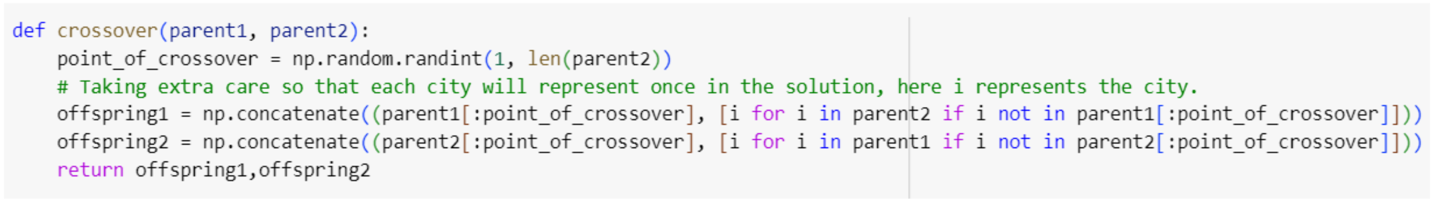
The loop iterates over the indices of the list population_order. population_order[i], population_order[i+1] represents the two consecutive parents chosen for the crossover. The crossover function is called with parent 1 and parent 2.

As parameters and the new chromosomes produced are stored in offspring1 and offspring2. The offspring chromosomes are added to the list next_population.

After all crossover operations are performed,the population variable is updated with the currently generated new_population .

Step 8:- Single Point Crossover Function:

The Crossover Operation explained above is implemented using the following function:



```
def crossover(parent1, parent2):
    point_of_crossover = np.random.randint(1, len(parent2))
    # Taking extra care so that each city will represent once in the solution, here i represents the city.
    offspring1 = np.concatenate((parent1[:point_of_crossover], [i for i in parent2 if i not in parent1[:point_of_crossover]]))
    offspring2 = np.concatenate((parent2[:point_of_crossover], [i for i in parent1 if i not in parent2[:point_of_crossover]]))
    return offspring1,offspring2
```

This function has a variable named point_of_crossover which takes any random integer between the range of (1,len(parent2)).

The first offspring is created by concatenating two parts->

The first part consists of the elements from the beginning of Parent 1 up to randomly chosen crossover points (not included).

The second part is created from parent 2 by adding elements that are not already present in the first part.This ensures that each city is represented only once in the solution.

The second offspring is created in a similar manner by concatenating two parts.It takes the beginning of parent 2 till crossover point and then concatenates the part of the first parent which is not already taken in the first part.

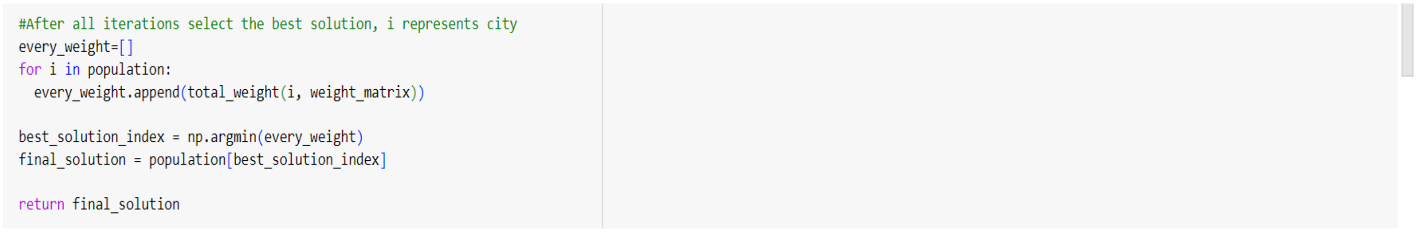
This function finally returns the offspring1 and offspring 2.

ALL THE STEPS FROM STEP 3 TO STEP 8 ARE REPEATED 20 TIMES.

After all the iterations have been completed,we take a list named every_weight which contains weights (cost) of all the chromosomes present in the final population list.

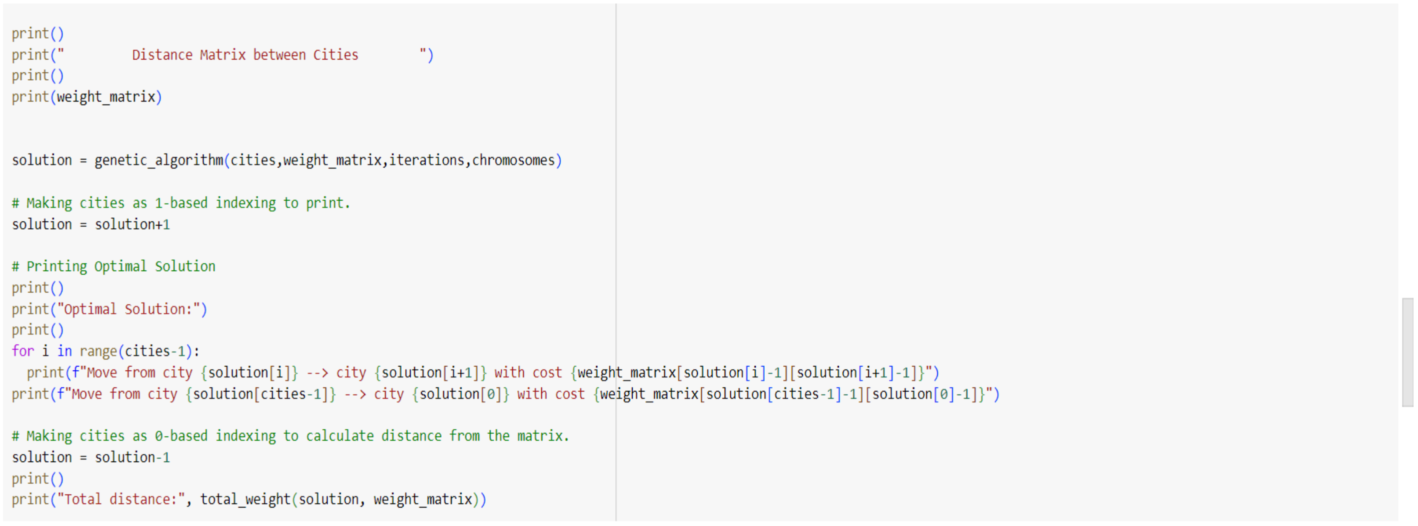
And then we find the minimum of all these weights to get the index of the best possible solution and store it in the variable named best_solution_index and then we store the best chromosome (path) in the variable named final_solution and then return it.

Above is done using the following piece of code:



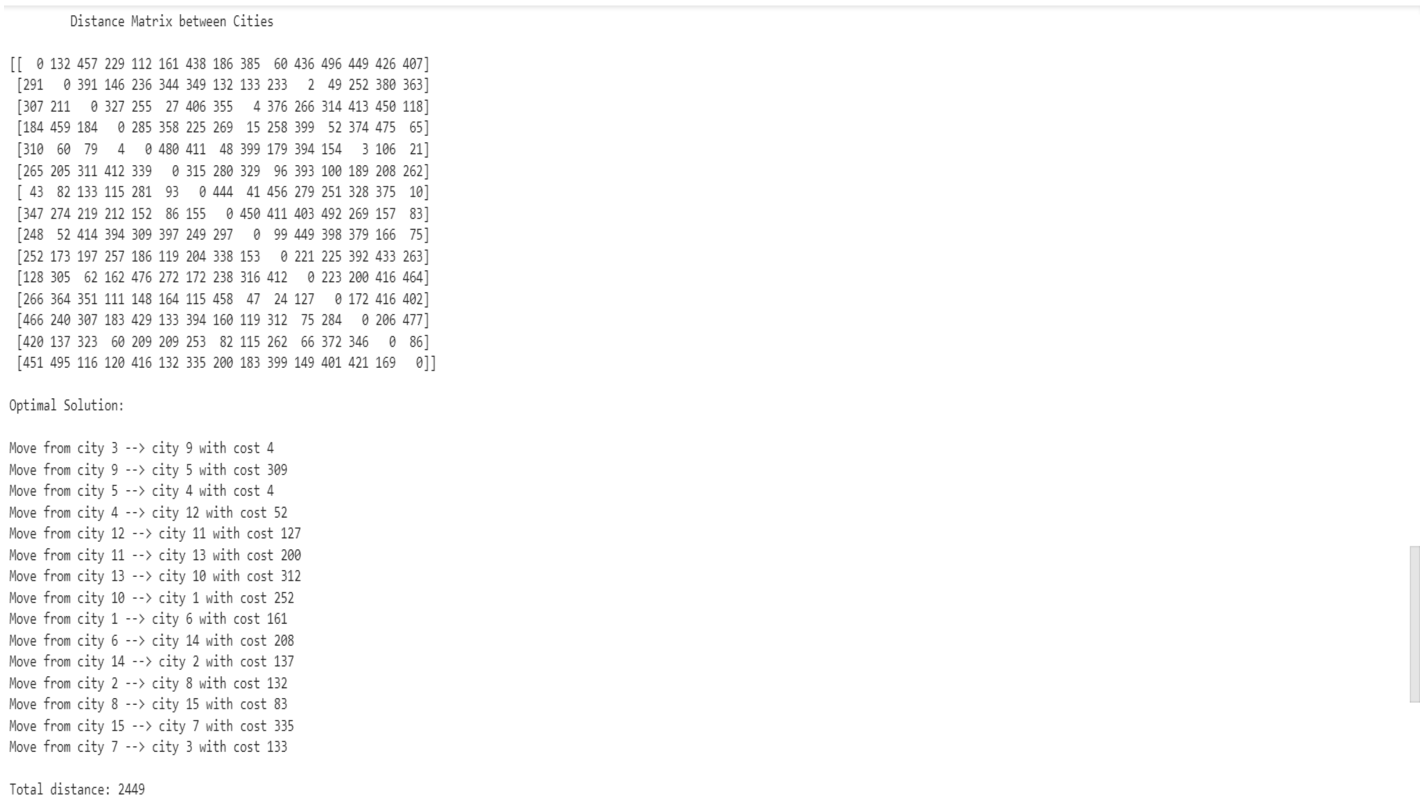
```python
#After all iterations select the best solution, i represents city
every_weight=[]
for i in population:
  every_weight.append(total_weight(i, weight_matrix))

best_solution_index = np.argmin(every_weight)
final_solution = population[best_solution_index]

return final_solution
```

The following piece of code is used to get the output:



```python
print()
print("        Distance Matrix between Cities       ")
print()
print(weight_matrix)


solution = genetic_algorithm(cities,weight_matrix,iterations,chromosomes)

# Making cities as 1-based indexing to print.
solution = solution+1

# Printing Optimal Solution
print()
print("Optimal Solution:")
print()
for i in range(cities-1):
  print(f"Move from city {solution[i]} --> city {solution[i+1]} with cost {weight_matrix[solution[i]-1][solution[i+1]-1]}")
print(f"Move from city {solution[cities-1]} --> city {solution[0]} with cost {weight_matrix[solution[cities-1]-1][solution[0]-1]}")

# Making cities as 0-based indexing to calculate distance from the matrix.
solution = solution-1
print()
print("Total distance:", total_weight(solution, weight_matrix))
```

OUTPUT:->



```
        Distance Matrix between Cities

[[  0 132 457 229 112 161 438 186 385  60 436 496 449 426 407]
 [291   0 391 146 236 344 349 132 133 233   2  49 252 380 363]
 [307 211   0 327 255  27 406 355   4 376 266 314 413 450 118]
 [184 459 184   0 285 358 225 269  15 258 399  52 374 475  65]
 [310  60  79   4   0 480 411  48 399 179 394 154   3 106  21]
 [265 205 311 412 339   0 315 280 329  96 393 100 189 208 262]
 [ 43  82 133 115 281  93   0 444  41 456 279 251 328 375  10]
 [347 274 219 212 152  86 155   0 450 411 403 492 269 157  83]
 [248  52 414 394 309 397 249 297   0  99 449 398 379 166  75]
 [252 173 197 257 186 119 204 338 153   0 221 225 392 433 263]
 [128 305  62 162 476 272 172 238 316 412   0 223 200 416 464]
 [266 364 351 111 148 164 115 458  47  24 127   0 172 416 402]
 [466 240 307 183 429 133 394 160 119 312  75 284   0 206 477]
 [420 137 323  60 209 209 253  82 115 262  66 372 346   0  86]
 [451 495 116 120 416 132 335 200 183 399 149 401 421 169   0]]

Optimal Solution:

Move from city 3 --> city 9 with cost 4
Move from city 9 --> city 5 with cost 309
Move from city 5 --> city 4 with cost 4
Move from city 4 --> city 12 with cost 52
Move from city 12 --> city 11 with cost 127
Move from city 11 --> city 13 with cost 200
Move from city 13 --> city 10 with cost 312
Move from city 10 --> city 1 with cost 252
Move from city 1 --> city 6 with cost 161
Move from city 6 --> city 14 with cost 208
Move from city 14 --> city 2 with cost 137
Move from city 2 --> city 8 with cost 132
Move from city 8 --> city 15 with cost 83
Move from city 15 --> city 7 with cost 335
Move from city 7 --> city 3 with cost 133

Total distance: 2449
```