Import Libraries

```
!pip install deap

import array
import random
import numpy as np
import matplotlib.pyplot as plot
import matplotlib.cm as cm

from deap import base, creator, tools
from deap.benchmarks.tools import hypervolume
from sympy.combinatorics.graycode import random_bitstring, gray_to_bin, bin_to_gray
```

```
Collecting deap
  Downloading deap-1.3.1-cp37-cp37m-manylinux_2_5_x86_64.manylinux1_x86_64.manylinux_2_12_x86_64.manylinux2010_x86_64.whl (160
     |████████████████████████████████| 160 kB 14.1 MB/s
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (from deap) (1.21.6)
Installing collected packages: deap
Successfully installed deap-1.3.1
```

Constants

```
# Binary length
BITS = 10
# -4.0 <= x1,x2,x3 <= 4.0
LOWER_BOUND, UPPER_BOUND = - 4, 4
# Number of genes (decision variables)
DECISION_VARIABLES = 3

POPULATION_SIZE = 25
NUMBER_OF_GENERATIONS = 30
OFFSPRING_SIZE = 25
```

```
CROSS_PROBABILITY = 0.9
MUTATION_PROBABILITY = 1/(BITS * DECISION_VARIABLES)

random.seed(10)
```

Functions

```
def gray_to_real(gray, lower, upper):
  """ Converts between gray coding to a
  real value in range [lower, upper] """
  l = len(gray)
  # Converts to binary, then integer
  binary = gray_to_bin(gray)
  integer = int(binary, 2)
  # Converts to real-valued
  x = lower + (upper - lower) * (1 / (2**l  - 1)) * integer
  return x
```

```
def evaluation(individual):
  """ Evaluation function """
  # An individual's genes
  g1, g2, g3 = individual[0], individual[1], individual[2]
  # Converts from gray code to real-valued
  x1, x2, x3 = gray_to_real(g1, LOWER_BOUND, UPPER_BOUND), gray_to_real(g2, LOWER_BOUND, UPPER_BOUND), gray_to_real(g3, LOWER_BOUN

  f1 = ((x1/2)**2 + (x2/4)**2 + x3**2) / 3
  f2 = (((x1/2) - 1)**2 + ((x2/4) - 1)**2 + (x3 - 1)**2) / 3
  return f1, f2
```

```
def check_dominated(individual, current_front):
  # Compare with individuals in current front starting from last
  for compare_ind in current_front[::-1]:
    if compare_ind.fitness.dominates(individual.fitness):
```

```
    return True
```

```
def sequential_search(ind, fronts):
  # Current front to check
  front_index = 0
  while True:
    current_front = fronts[front_index]
    dominated = check_dominated(ind, current_front)
    if not dominated:
      # Adds individual to current front
      fronts[front_index].append(ind)
      return fronts
    front_index += 1
    if front_index + 1 > len(fronts):
      # Add individual to new front
      new_front = [ind]
      fronts.append(new_front)
      return fronts
```

```
def efficient_ND_sort(population):
  """ Efficient non-dominated sorting """
  # Copies population to prevent population changing
  copy_population = toolbox.clone(population)
  # Sort population by f1, then by f2 is f1 is equal
  copy_population.sort(key=lambda ind: (ind.fitness.values[0], ind.fitness.values[1]))
  # Assigns best f1 to front 1
  fronts = [[copy_population[0]]]
  copy_population.remove(copy_population[0])
  for ind in copy_population:
    fronts = sequential_search(ind, fronts)
  return fronts
```

```
def assignCrowdingDist(front):
  """ Deap function to assign crowding distance to individuals in a front """
  if len(front) == 0:
    return
```

```
  distances = [0.0] * len(front)
  crowd = [(ind.fitness.values, i) for i, ind in enumerate(front)]

  nobj = len(front[0].fitness.values)

  for i in range(nobj):
      crowd.sort(key=lambda element: element[0][i])
      distances[crowd[0][1]] = float("inf")
      distances[crowd[-1][1]] = float("inf")
      if crowd[-1][0][i] == crowd[0][0][i]:
          continue
      norm = nobj * float(crowd[-1][0][i] - crowd[0][0][i])
      for prev, cur, next in zip(crowd[:-2], crowd[1:-1], crowd[2:]):
          distances[cur[1]] += (next[0][i] - prev[0][i]) / norm

  for i, dist in enumerate(distances):
      front[i].fitness.crowding_dist = dist
```

```
def compete(first, second):
  # Comparision based on dominance
  if first.fitness.dominates(second.fitness):
    return first
  elif second.fitness.dominates(first.fitness):
    return second
  # Comparision based on crowding distance
  if first.fitness.crowding_dist > second.fitness.crowding_dist:
    return first
  elif first.fitness.crowding_dist < second.fitness.crowding_dist:
    return second
  # Random selection
  if random.random() <= 0.5:
    return first
  return second
```

```
def binaryTournament(population, pair_number):
```

```
""" Binary tournament selection
Prevents the same individuals being repeatedly selected """
# Pairs of selected individuals
selected = []
# Copies population
copy_pop = [ind for ind in population]

# Number of parent pairs
for i in range(pair_number):
  # Randomly select two individuals
  first = random.choice(copy_pop)
  # Remove from population to prevent repeat selection
  copy_pop.remove(first)
  # Checks if all individuals selected
  if (len(copy_pop) == 0):
    # Adds more individuals for selection
    copy_pop = [ind for ind in population]
  second = random.choice(copy_pop)
  copy_pop.remove(second)
  if (len(copy_pop) == 0):
    copy_pop = [ind for ind in population]
  first_parent = compete(first, second)

  # Randomly select two more individuals
  first = random.choice(copy_pop)
  copy_pop.remove(first)
  if (len(copy_pop) == 0):
    copy_pop = [ind for ind in population]
  second = random.choice(copy_pop)
  if (len(copy_pop) == 0):
    copy_pop = [ind for ind in population]
  copy_pop.remove(second)
  second_parent = compete(first, second)
  selected.append([first_parent, second_parent])

return selected
```

```
def flipMutation(individual, probability=0.1):
  # Iterates through decision variables
  for i in range(len(individual)):
    # Creates updated decision variable
    new_var = ""
    # Iterates through bits in decision variable
    for bit in individual[i]:
      new_bit = bit
      # Flip bit
      if random.random() <= probability:
        if bit == '0':
          new_bit = '1'
        if bit == '1':
          new_bit = '0'
      new_var += new_bit
    # Updates mutated decision variable
    individual[i] = new_var
  return individual
```

Genetic Algorithm

```
# Creates a fitness for minimization of a problem with 2 objectives
creator.create("FitnessMin", base.Fitness, weights=(-1.0, -1.0))
# Creates class Individual with fitness set for minimization
creator.create("Individual", list, fitness=creator.FitnessMin)

toolbox = base.Toolbox()

# Generation function for decision variables using 10 bit gray coding
toolbox.register("gray_code", random_bitstring, BITS)
# Initializers for individual and population
toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox.gray_code, DECISION_VARIABLES)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

# Genetic operators
toolbox.register("evaluate", evaluation) # Uses evaluation function
```

```
toolbox.register("sort", efficient_ND_sort) # Non-dominated sorting into fronts
toolbox.register("crowd", assignCrowdingDist) # Assign crowding distance to each individual of the list
toolbox.register("tournament", binaryTournament) # Tournament selection
toolbox.register("mate", tools.cxUniform, indpb=0.5) # Uniform crossover with 50% chance of exchange
toolbox.register("mutate", flipMutation, probability=MUTATION_PROBABILITY) # Flip mutation with chance = 1 / chromosome length
```

```
def main():
  stats = tools.Statistics()
  logbook = tools.Logbook()
  logbook.header = "generation", "x1", "x2", "x3", "f1", "f2", "front_number", "crowding_distance"

  # Initiate the population
  pop = toolbox.population(POPULATION_SIZE)
  print("Intial Population is \n",pop)
  print("The Length of the initial Population", len(pop))
  # Hypervolume over generations
  hypervolumes = []

  # Evaluate fitness of population
  invalid_individuals = [ind for ind in pop if not ind.fitness.valid]
  fitnesses = list(map(toolbox.evaluate, pop))
  for ind, fit in zip(invalid_individuals, fitnesses):
    ind.fitness.values = fit

  # Find the worst f1 and f2 values
  copy_pop = toolbox.clone(pop)
  # Sort population by f1 and f2
  copy_pop.sort(key=lambda x: x.fitness.values[0], reverse=True)
  worst_f1 = copy_pop[0].fitness.values[0]
  copy_pop.sort(key=lambda x: x.fitness.values[1], reverse=True)
  worst_f2 = copy_pop[0].fitness.values[1]
  # Sets reference point for hypervolume
  reference = [worst_f1, worst_f2]
  print("Worst f1: " + str(worst_f1))
  print("Worst f2: " + str(worst_f2))
  print("\n")

```

```
# Sort population into fronts
fronts = toolbox.sort(pop)
updated_pop = []
for i in range(len(fronts)):
  front = fronts[i]
  # Assign crowding distance to individuals in each front
  toolbox.crowd(front)
  for ind in front:
    updated_pop.append(ind)
    # Print out individuals and fitness
    logbook.record(generation=0, x1=ind[0], x2=ind[1], x3=ind[2], f1=ind.fitness.values[0], f2=ind.fitness.values[1], front_numb
    print(logbook.stream)
print("\n")
# Updates population so that individuals have crowding distance
pop = updated_pop

# Calculates hypervolume of generation 0
hv = hypervolume(fronts[0], reference)
hypervolumes.append(hv)

for generation in range(1, NUMBER_OF_GENERATIONS + 1):
  # Selects parents through tournament selection
  parent_pairs = toolbox.tournament(pop, len(pop))

  parents = []
  offspring = []
  for pair in parent_pairs:
    # Makes copies of parents to modify
    parent1 = toolbox.clone(pair[0])
    parent2 = toolbox.clone(pair[1])
    offspring1 = toolbox.clone(parent1)
    offspring2 = toolbox.clone(parent2)
    # Cross over
    if random.random() <= CROSS_PROBABILITY:
      toolbox.mate(offspring1, offspring2)
    # Mutate both offspring
    toolbox.mutate(offspring1)
```

```
    toolbox.mutate(offspring2)
    parents.append(parent1)
    offspring.append(offspring1)
    parents.append(parent2)
    offspring.append(offspring2)
  # Caps to 25 offspring
  offspring = offspring[:OFFSPRING_SIZE]
  parents = parents[:OFFSPRING_SIZE]
  print("\n")
  print("The parents in the generation {} are \n{}".format(generation, parents))
  print("The total count of parents in the generation {} are {}".format(generation, len(parents)))
  print("The offspring in the generation {} are \n{}".format(generation, offspring))
  print("The total count of offsprings in the generation {} are {}".format(generation, len(offspring)))
  # Offspring still have parent's fitness
  parent_f1 = [off.fitness.values[0] for off in offspring]
  parent_f2 = [off.fitness.values[1] for off in offspring]
  # Offspring fitness is updated
  fitnesses = list(map(toolbox.evaluate, offspring))
  for off, fit in zip(offspring, fitnesses):
    off.fitness.values = fit
  #if generation == 1 or generation == 10 or generation == 20 or generation == NUMBER_OF_GENERATIONS:

  # Print graph
  if generation == 1 or generation == 10 or generation == 20 or generation == NUMBER_OF_GENERATIONS:
    offspring_f1 = [off.fitness.values[0] for off in offspring]
    offspring_f2 = [off.fitness.values[1] for off in offspring]
    # Plot parents and offspring fitness
    plot.figure()
    plot.title('Parents and Offspring Fitness for Generation ' + str(generation))
    plot.plot(parent_f1, parent_f2, 'ro', alpha=0.5, label='Parents')
    plot.plot(offspring_f1, offspring_f2, 'bo', alpha=0.5, label='Offspring')
    plot.xlabel('f1')
    plot.ylabel('f2')
    plot.legend()

  # Combine parents and offspring then sort
  combined_pop = parents + offspring
```

```
    fronts = toolbox.sort(combined_pop)
    sorted_pop = []
    for i in range(len(fronts)):
      front = fronts[i]
      # Assign crowding distance to individuals in each front
      toolbox.crowd(front)
      # Sort individual's in front by crowding distance in descending order
      front.sort(key=lambda x: x.fitness.crowding_dist, reverse=True)
      for ind in front:
        # Ensures individual's are unique to improve diversity
        duplicated = False
        for ind2 in sorted_pop:
          if ind == ind2:
            duplicated = True
        if not duplicated:
          sorted_pop.append(ind)
    # Select top individuals
    pop = sorted_pop[:POPULATION_SIZE]

    # Hypervolume using the worst objective values as the reference point
    hv = hypervolume(fronts[0], reference)
    hypervolumes.append(hv)

    # Print graph
    if generation == 1 or generation == 10 or generation == 20 or generation == NUMBER_OF_GENERATIONS:
      # Worst individuals
      rejected_pop = sorted_pop[POPULATION_SIZE:]
      # Fitness of selected and rejected individuals
      selected_f1 = [ind.fitness.values[0] for ind in pop]
      selected_f2 = [ind.fitness.values[1] for ind in pop]
      rejected_f1 = [ind.fitness.values[0] for ind in rejected_pop]
      rejected_f2 = [ind.fitness.values[1] for ind in rejected_pop]

      # Plot fronts
      plot.figure()
      plot.title('Fronts for Generation ' + str(generation))
      # Iterable rainbow colour map
```

```
    colours = iter(cm.hsv(np.linspace(0, 1, len(fronts))))
    for f in range(len(fronts)-1, 0, -1):
      front_f1s = [ind.fitness.values[0] for ind in fronts[f]]
      front_f2s = [ind.fitness.values[1] for ind in fronts[f]]
      plot.plot(front_f1s, front_f2s, 'o', color=next(colours), label='Front ' + str(f))
    plot.xlabel('f1')
    plot.ylabel('f2')
    plot.legend(loc='upper left', bbox_to_anchor=(1.05, 1))

    # Plot parents and offspring fitness
    plot.figure()
    plot.title('Selected Solutions Fitness for Generation ' + str(generation))
    plot.plot(selected_f1, selected_f2, 'o', alpha=0.5, label='Selected')
    plot.plot(rejected_f1, rejected_f2, 'o', alpha=0.5, label='Not selected')
    plot.xlabel('f1')
    plot.ylabel('f2')
    plot.legend()

  # Plot hypervolume over generations
  hvs = np.array(hypervolumes)
  # Makes list of generation numbers same shape as hypervolumes list
  gens = np.zeros_like(hvs)
  for i in range(0, len(gens)):
    gens[i] = i
  plot.figure()
  plot.title('Hypervolume Over Generations')
  plot.plot(gens, hvs, '-*', color='indigo')
  plot.xlabel('Generation')
  plot.ylabel('Hypervolume')
  plot.show()

  return pop
```

```
# Run the genetic algorithm
pop = main()
print("Population: \n" + str(pop))
print("\n")
```

```
Intial Population is
 [['0110011100', '1100101011', '1111011010'], ['1100000110', '1111010010', '0011000110'], ['0010011110', '1000111000', '11110
The Length of the initial Population 25
Worst f1: 6.087783202058105
Worst f2: 11.195310083778537
```

generation	x1	x2	x3	f1	f2	front_number	crowding_distance
0	0010011010	1101011111	0100111111	0.438074	2.23899	1	inf
0	1010001110	1101001001	0101100001	0.468535	0.964787	1	1
0	1110010000	1111111100	1111110001	0.824706	0.177916	1	inf
0	0010010111	0011110111	1100100001	0.639304	2.50147	2	inf
0	0110011100	1100101011	1111011010	0.722229	1.49349	2	0.455462
0	0010011110	1000111000	1111000110	1.02239	1.44696	2	0.688963
0	1001100000	0100111010	1110100000	1.77846	0.666369	2	inf
0	0110011011	0111011001	0101000001	0.640634	3.10886	3	inf
0	0011111011	1111111111	1101101010	0.74027	2.00159	3	0.484633
0	0011111011	1011101111	1101000011	1.04417	1.8415	3	0.386424
0	1101110001	0000101010	1010001110	1.77435	1.73362	3	0.515367
0	1001100111	0011110110	1110011001	2.31126	1.3673	3	inf
0	0110110000	0000110010	1111000101	0.912611	2.41505	4	inf
0	0111011011	1001110111	1010100100	2.32643	1.52976	4	inf
0	1010001111	0011110011	0010010100	2.12146	4.34465	5	inf
0	1010110100	0100111101	1000111001	4.86439	2.72135	5	1
0	1000110111	1111100100	1001101000	5.0039	2.27174	5	inf
0	1100010000	1010110101	0011100010	2.24888	4.46947	6	inf
0	1011110100	0101000101	1000110010	5.25205	3.0282	6	inf
0	1100000110	1111010010	0011000110	2.96714	5.72993	7	inf
0	0111000100	0001100001	1000001110	5.45397	4.77981	7	inf
0	1101000000	0111111001	0001011011	3.42411	6.4176	8	inf
0	1001000111	0100010110	0000110101	5.34444	7.83352	9	inf
0	1010001111	0010001011	0000011000	5.45937	8.69919	10	inf
0	0011001110	0001000110	0000000100	6.08778	11.1953	11	inf

```
The parents in the generation 1 are
[['1010001111', '0011110011', '0010010100'], ['1010001110', '1101001001', '0101100001'], ['0110110000', '0000110010', '111100
The total count of parents in the generation 1 are 25
The offspring in the generation 1 are
```

```
[['0010000111', '0011110011', '0101100001'], ['1010001110', '1101001000', '0010010100'], ['1101110001', '0010110010', '101000
The total count of offsprings in the generation 1 are 25


The parents in the generation 2 are
[['1110010000', '1100101011', '1111010001'], ['1001100000', '0100111010', '0100111111'], ['0010011011', '0110111010', '010011
The total count of parents in the generation 2 are 25
The offspring in the generation 2 are
[['1001100000', '0100111010', '1111010001'], ['1110000000', '1100101011', '0100111111'], ['0010000111', '0110111010', '010011
The total count of offsprings in the generation 2 are 25


The parents in the generation 3 are
[['0010000111', '0110111010', '0100111111'], ['1110010000', '1100001011', '1100100001'], ['1101110001', '0010110010', '010110
The total count of parents in the generation 3 are 25
The offspring in the generation 3 are
[['0010000111', '0110111010', '0100111111'], ['1110010000', '1101001011', '1100000001'], ['1010011010', '1000111000', '111100
The total count of offsprings in the generation 3 are 25


The parents in the generation 4 are
[['1110010000', '1111011101', '1100111001'], ['1110010000', '1101001011', '1100000001'], ['1101000000', '1101011111', '010011
The total count of parents in the generation 4 are 25
The offspring in the generation 4 are
[['1110010000', '1111011101', '1100000001'], ['1110010000', '1101001111', '1100111001'], ['1101000000', '1100001011', '010011
The total count of offsprings in the generation 4 are 25


The parents in the generation 5 are
[['1110010000', '1100001011', '1100100001'], ['1110010000', '1111011101', '1100000001'], ['1110010000', '1001110111', '110011
The total count of parents in the generation 5 are 25
The offspring in the generation 5 are
[['1110010000', '1111011101', '1100100001'], ['1110010000', '1100001011', '0100000001'], ['1110010000', '1000111000', '110011
The total count of offsprings in the generation 5 are 25


The parents in the generation 6 are
[['1100010000', '0101001011', '1100000001'], ['1100100000', '1101001001', '1111000110'], ['1110011000', '1001110111', '110110
The total count of parents in the generation 6 are 25
The offspring in the generation 6 are
[['1100010100', '0101001011', '1111000110'], ['1100100000', '1101001001', '1100000001'], ['1110011000', '1000111000', '111100
```

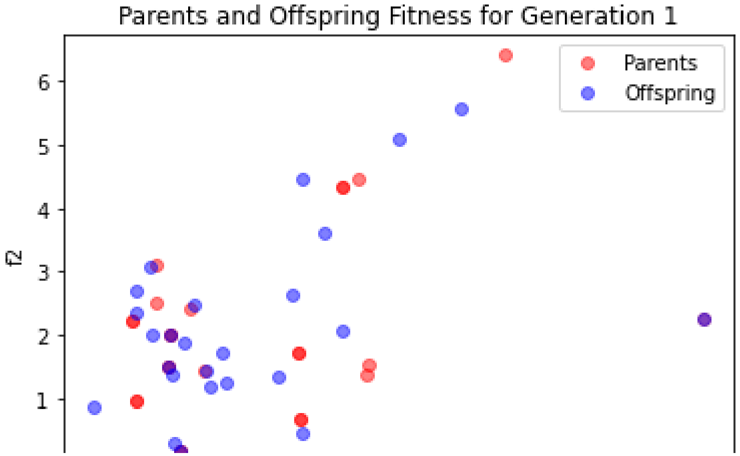
```
The total count of offsprings in the generation 6 are 25


The parents in the generation 7 are
[['1100010001', '1100001011', '1100011001'], ['1110010000', '1100001011', '1100100001'], ['1110011000', '1000111000', '11110
The total count of parents in the generation 7 are 25
The offspring in the generation 7 are
[['1110010000', '1100001011', '1100011001'], ['1100010001', '1100001011', '1100100001'], ['1110010001', '1111011101', '11100
The total count of offsprings in the generation 7 are 25


The parents in the generation 8 are
[['1010011010', '1000111000', '1111000110'], ['1110010000', '1111011101', '1100100001'], ['1110011000', '1000111000', '11110
The total count of parents in the generation 8 are 25
The offspring in the generation 8 are
[['1110010000', '1111011101', '1100100001'], ['1010011010', '1010111000', '1111110110'], ['1010011000', '1000111000', '11110
The total count of offsprings in the generation 8 are 25


The parents in the generation 9 are
[['1110011000', '1000111000', '1111000110'], ['1010011010', '1001110111', '1100100110'], ['1100010001', '1100001011', '11001
The total count of parents in the generation 9 are 25
The offspring in the generation 9 are
[['1110001000', '1000111000', '1100100110'], ['1010011010', '1001110111', '1111000110'], ['1100010000', '0001110111', '11001
The total count of offsprings in the generation 9 are 25


The parents in the generation 10 are
[['1110010001', '1000111000', '1100010101'], ['1100010001', '1100001011', '1100111001'], ['1110011000', '1000111000', '11110
The total count of parents in the generation 10 are 25
The offspring in the generation 10 are
[['1110010001', '1000111000', '1100111001'], ['1100010001', '1100001011', '1100010100'], ['1110011000', '1000111000', '11111
The total count of offsprings in the generation 10 are 25


The parents in the generation 11 are
[['1100010001', '1100001011', '1100111001'], ['1110010001', '1000111000', '1100111001'], ['1110011000', '1000111000', '11110
The total count of parents in the generation 11 are 25
The offspring in the generation 11 are
[['1100010001', '1100001011', '1100111001'], ['1110000001', '1001111000', '1100111101'], ['1010011000', '1000111000', '11110
The total count of offsprings in the generation 11 are 25
```

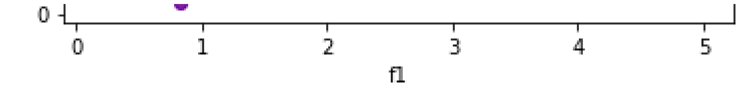
```
The parents in the generation 12 are
[['1100010000', '1100000011', '1100011011'], ['1110010000', '1010001011', '1111000110'], ['1100000000', '1100000011', '110001
The total count of parents in the generation 12 are 25
The offspring in the generation 12 are
[['1100110000', '1100000011', '1100011011'], ['1100010000', '1010001011', '1111000110'], ['1101000000', '1100000011', '010001
The total count of offsprings in the generation 12 are 25


The parents in the generation 13 are
[['1100000100', '1010001111', '1100110001'], ['1110010000', '1010001011', '1111000110'], ['1010011000', '1010001011', '111100
The total count of parents in the generation 13 are 25
The offspring in the generation 13 are
[['1100000100', '1010001011', '1111000110'], ['1110010000', '1010001110', '1100111001'], ['1110000010', '1010001011', '111100
The total count of offsprings in the generation 13 are 25


The parents in the generation 14 are
[['1100000000', '1100000011', '1100011011'], ['0100100000', '1100000011', '1100110001'], ['1110010000', '1010000011', '110011
The total count of parents in the generation 14 are 25
The offspring in the generation 14 are
[['0000100000', '1101000011', '1100011011'], ['1100000000', '1100000011', '1000110001'], ['1110000101', '1010000011', '110011
The total count of offsprings in the generation 14 are 25


The parents in the generation 15 are
[['1110100000', '1100000011', '1100011010'], ['1110010000', '1110010101', '1111000010'], ['1110000010', '1010110000', '111100
The total count of parents in the generation 15 are 25
The offspring in the generation 15 are
[['1111010101', '1110010101', '1100011010'], ['1110100010', '1100000011', '1111001010'], ['1110000010', '1010110000', '111100
The total count of offsprings in the generation 15 are 25


The parents in the generation 16 are
[['0100100000', '1100000011', '1100110001'], ['1100000000', '1010001111', '1100110001'], ['1100100000', '1100000011', '110011
The total count of parents in the generation 16 are 25
The offspring in the generation 16 are
[['0100100000', '1100000011', '1100110001'], ['1100000000', '1010001111', '1100110001'], ['1100100000', '1000010100', '111100
The total count of offsprings in the generation 16 are 25
```

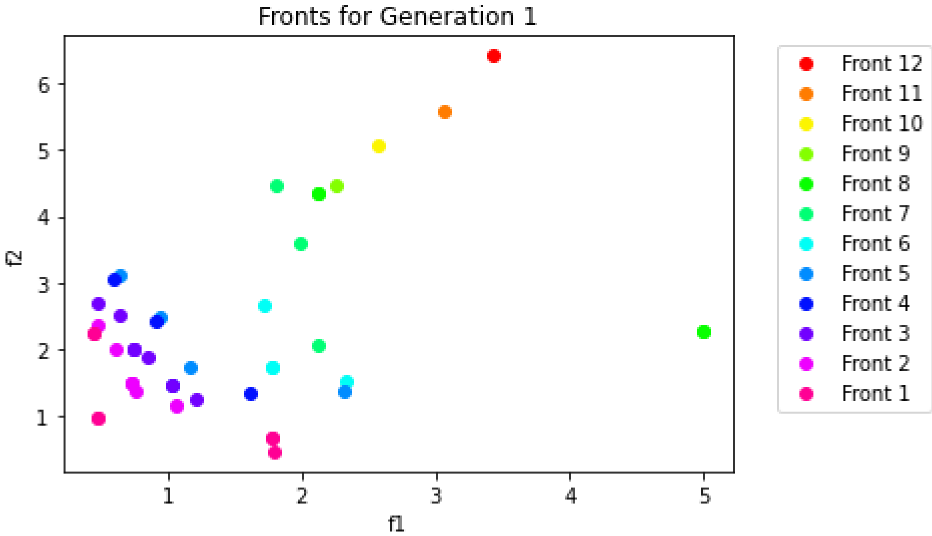
```
The parents in the generation 17 are
[['0100100000', '1100000011', '1100110001'], ['1100000000', '1010111000', '1111000110'], ['1100000000', '1100000011', '110100
The total count of parents in the generation 17 are 25
The offspring in the generation 17 are
[['0101100000', '1100000011', '1100110001'], ['1100000000', '1010101000', '1111000110'], ['0110010000', '1111011101', '110011
The total count of offsprings in the generation 17 are 25


The parents in the generation 18 are
[['1110000000', '1000111000', '1111000010'], ['1110010001', '1010010101', '1111000010'], ['1100100011', '1000111010', '110011
The total count of parents in the generation 18 are 25
The offspring in the generation 18 are
[['1110000100', '1000111001', '1011000010'], ['1110000001', '1010010101', '1111000010'], ['1100100011', '1000111010', '110011
The total count of offsprings in the generation 18 are 25


The parents in the generation 19 are
[['1110011000', '1000010000', '1111000010'], ['1110000000', '1000111000', '1111000010'], ['0100100000', '1100000011', '110011
The total count of parents in the generation 19 are 25
The offspring in the generation 19 are
[['1110011000', '1000010000', '1111000010'], ['1110000000', '1000111000', '1111000010'], ['0111010000', '1100000011', '110011
The total count of offsprings in the generation 19 are 25


The parents in the generation 20 are
[['1110001000', '1000010000', '1111000111'], ['1101001000', '1111011010', '1100111001'], ['1110011000', '1000010000', '111100
The total count of parents in the generation 20 are 25
The offspring in the generation 20 are
[['1101001000', '1111011010', '1111000111'], ['1110001000', '1000010010', '1100111001'], ['1110011000', '1100000011', '110011
The total count of offsprings in the generation 20 are 25


The parents in the generation 21 are
[['1101001000', '1110010110', '1101000010'], ['1110011001', '1000010000', '1111000010'], ['1100111000', '1000111000', '110001
The total count of parents in the generation 21 are 25
The offspring in the generation 21 are
[['1110011001', '1000010000', '1111010011'], ['1101001000', '1110010110', '1101000110'], ['1100000000', '1100100011', '110001
The total count of offsprings in the generation 21 are 25


The parents in the generation 22 are
```

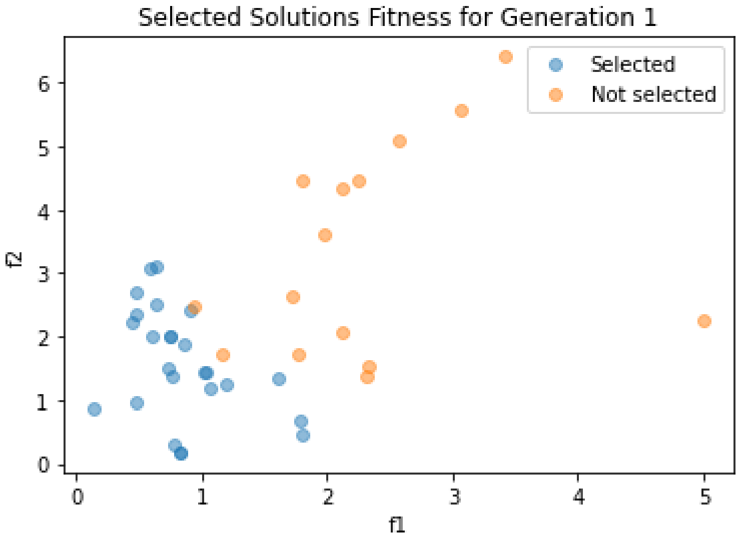
```
[['0100000000', '1000010000', '1101011011'], ['1110011101', '1110010110', '1111000110'], ['1110011001', '1000010000', '111100
The total count of parents in the generation 22 are 25
The offspring in the generation 22 are
[['1100000000', '1000010000', '1101011011'], ['1110011101', '1110010110', '1111000110'], ['1110011000', '1000010000', '101100
The total count of offsprings in the generation 22 are 25


The parents in the generation 23 are
[['0100000000', '1010111000', '1100010010'], ['1101001000', '1110010110', '1101000110'], ['1110011000', '1110010110', '111100
The total count of parents in the generation 23 are 25
The offspring in the generation 23 are
[['0100000000', '1010111000', '1100111010'], ['1101001000', '0110010110', '1101000110'], ['1110011001', '1000010000', '011100
The total count of offsprings in the generation 23 are 25


The parents in the generation 24 are
[['1110011001', '1000010100', '1100011011'], ['1101000000', '1000010000', '1101010010'], ['1101001000', '1111011010', '111100
The total count of parents in the generation 24 are 25
The offspring in the generation 24 are
[['1101000000', '1000010000', '1100011111'], ['1110101001', '1001010100', '1101010010'], ['1101001000', '1111011010', '110011
The total count of offsprings in the generation 24 are 25


The parents in the generation 25 are
[['1100000000', '1100000011', '1100011011'], ['1110011000', '1000010100', '1111000110'], ['1101001000', '1110010110', '110100
The total count of parents in the generation 25 are 25
The offspring in the generation 25 are
[['0100000000', '1000010100', '1111000110'], ['1110011000', '1100000011', '1100011011'], ['1101001000', '1110010110', '110100
The total count of offsprings in the generation 25 are 25


The parents in the generation 26 are
[['1111001010', '1011011010', '1111000110'], ['1101001010', '1110010110', '1100010010'], ['1110101001', '1001010100', '110101
The total count of parents in the generation 26 are 25
The offspring in the generation 26 are
[['1111001010', '1011011010', '1111000110'], ['1100001010', '1110010110', '1100010010'], ['1110101001', '1000010100', '111100
The total count of offsprings in the generation 26 are 25


The parents in the generation 27 are
[['1110011000', '1000010101', '1111000110'], ['1110011000', '1000010100', '1111000110'], ['1111000001', '1110010110', '110100
```

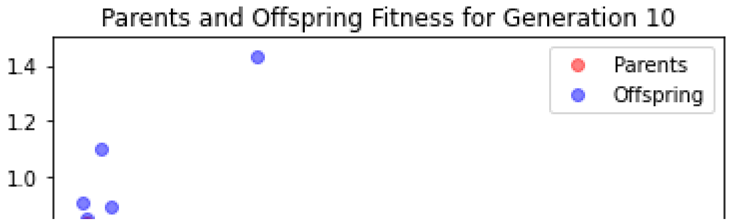
```
The total count of parents in the generation 27 are 25
The offspring in the generation 27 are
[['1010011000', '1100010100', '1011000110'], ['1010011000', '1100010101', '1111000110'], ['1110101001', '1000010100', '11110(
The total count of offsprings in the generation 27 are 25


The parents in the generation 28 are
[['1100000000', '1100000011', '1100011011'], ['1100000000', '0110010110', '1100011110'], ['1110011000', '1000011000', '11110(
The total count of parents in the generation 28 are 25
The offspring in the generation 28 are
[['1100000000', '0110000110', '1100011110'], ['1100000000', '1100000011', '1100011011'], ['1110011000', '1111011010', '11110(
The total count of offsprings in the generation 28 are 25


The parents in the generation 29 are
[['1111000001', '1100001011', '1101001111'], ['1110011000', '1000010001', '1100111010'], ['1110011000', '1110010110', '11110(
The total count of parents in the generation 29 are 25
The offspring in the generation 29 are
[['1000011000', '1000110001', '1101001111'], ['1110000001', '1100001010', '1101111010'], ['1110011000', '1000011000', '11111(
The total count of offsprings in the generation 29 are 25


The parents in the generation 30 are
[['1111001010', '1110010110', '1101010010'], ['1101001010', '1111011010', '1100010011'], ['1110011000', '1000011000', '11111(
The total count of parents in the generation 30 are 25
The offspring in the generation 30 are
[['1111101010', '1110010110', '1101010010'], ['1101001000', '0111011010', '1000010011'], ['1110011000', '1000011000', '11010(
The total count of offsprings in the generation 30 are 25
```

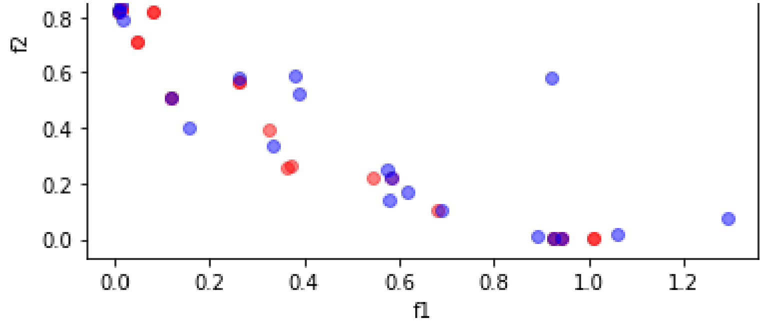


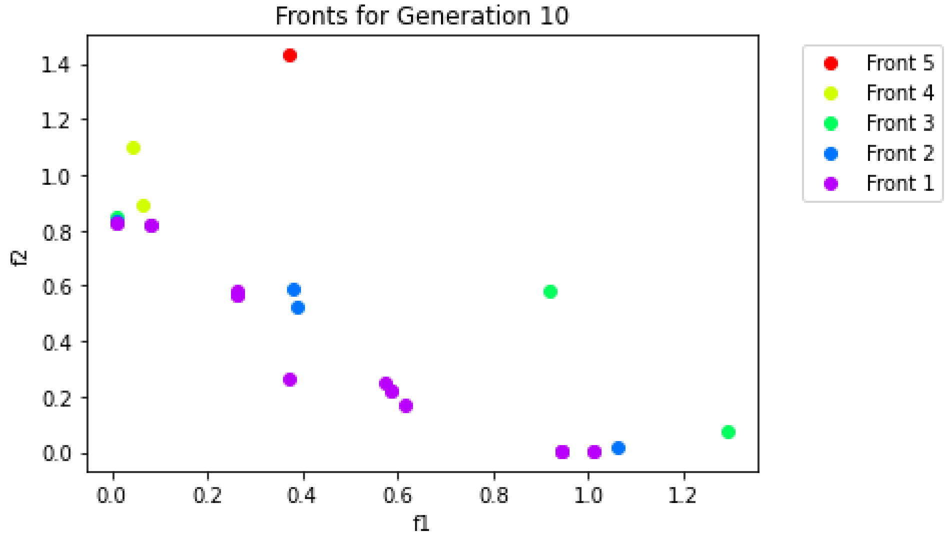

f1

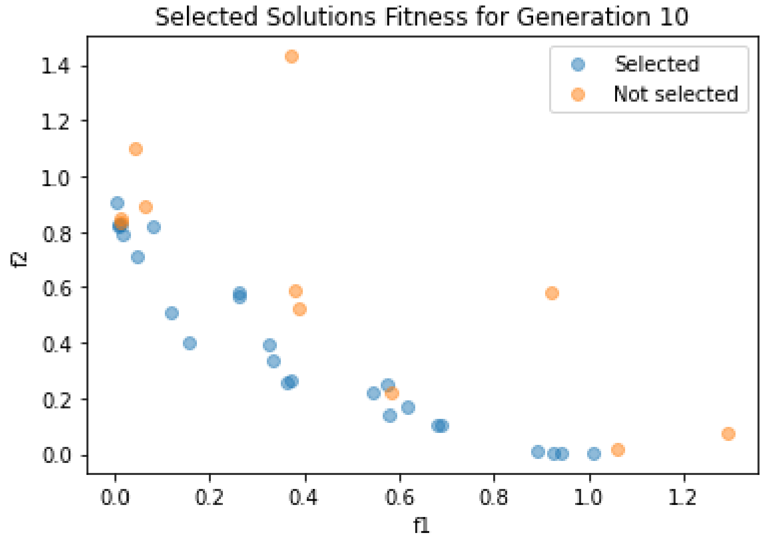

Fronts for Generation 1
f1
f2
Front 12
Front 11
Front 10
Front 9
Front 8
Front 7
Front 6
Front 5
Front 4
Front 3
Front 2
Front 1

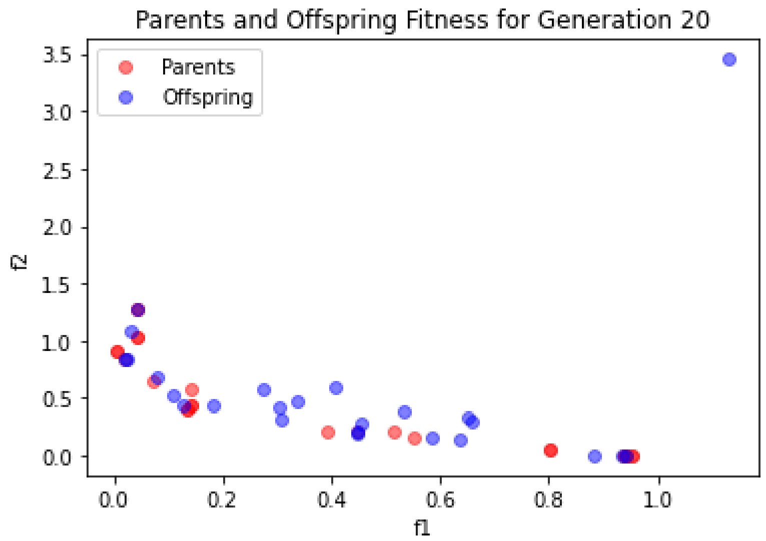

Selected Solutions Fitness for Generation 1
f1
f2
Selected
Not selected

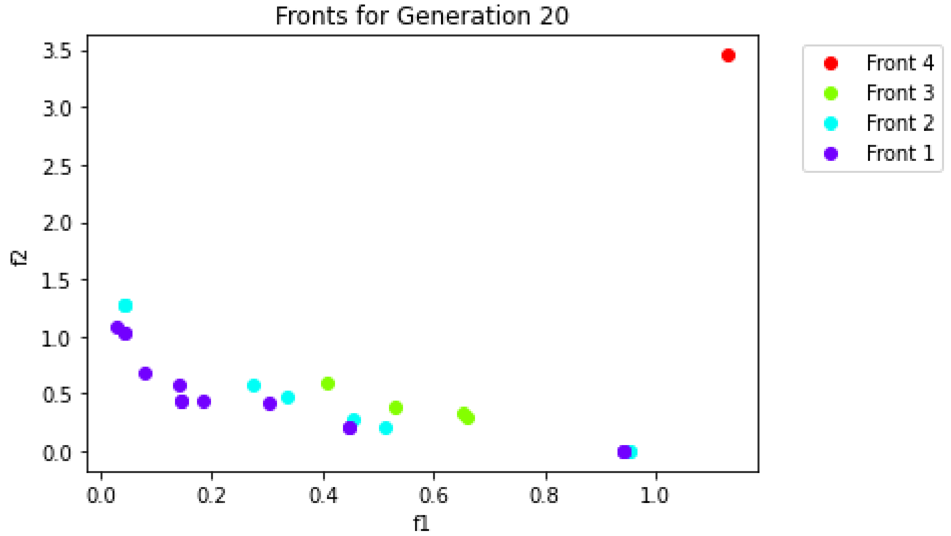

Parents and Offspring Fitness for Generation 10
Parents
Offspring

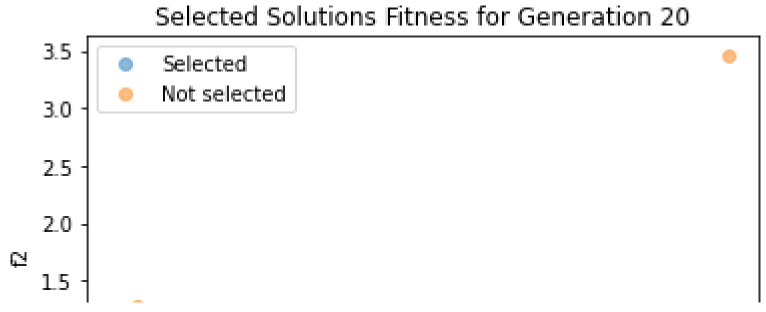

f2
f1

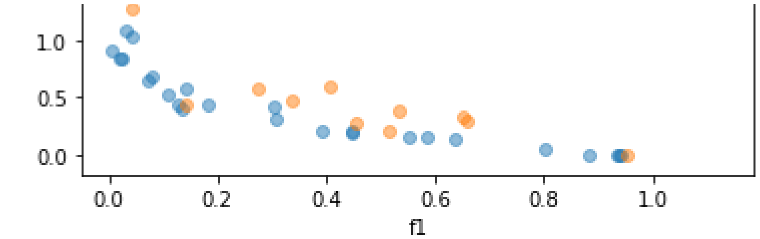

Fronts for Generation 10
Front 5
Front 4
Front 3
Front 2
Front 1
f2
f1


Selected Solutions Fitness for Generation 10
Selected
Not selected
f2
f1

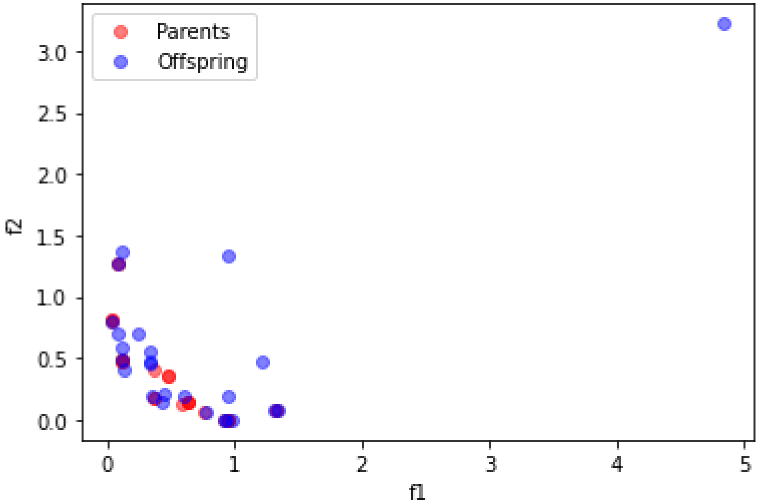

Parents and Offspring Fitness for Generation 20
Parents
Offspring
f1
f2


Fronts for Generation 20
Front 4
Front 3
Front 2
Front 1
f1
f2

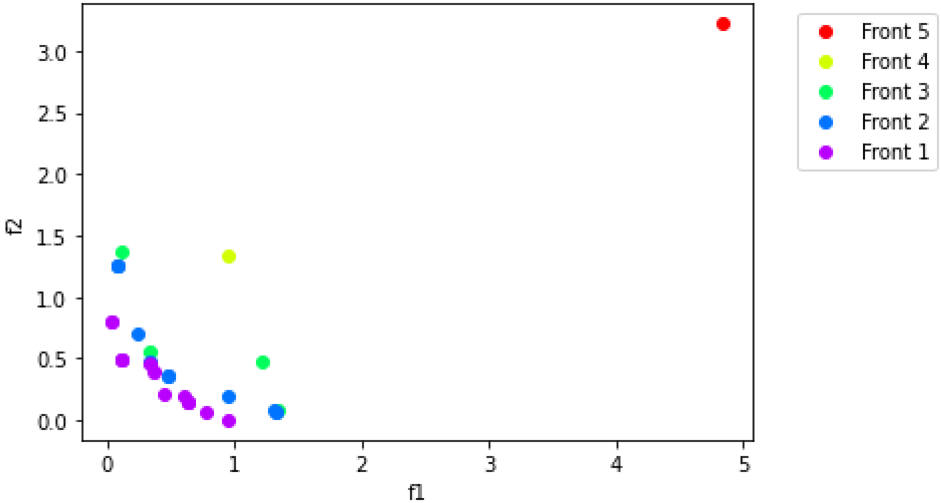

Selected Solutions Fitness for Generation 20
Selected
Not selected
f2



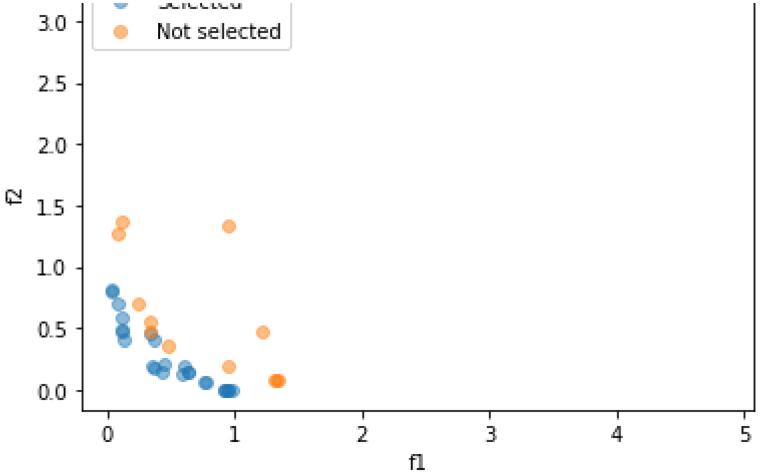
Parents and Offspring Fitness for Generation 30

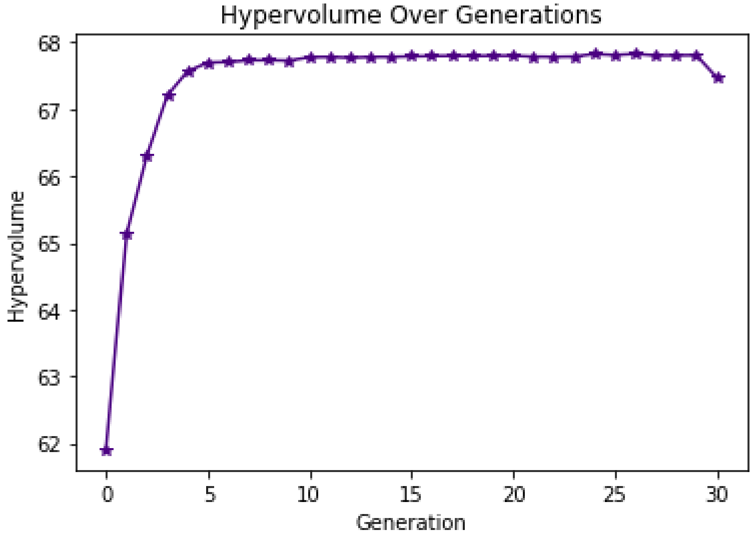


Fronts for Generation 30



Selected Solutions Fitness for Generation 30






```
Population:
[['1100000000', '1100000011', '1100111010'], ['1110000001', '1000011000', '1111000000'], ['1101001010', '1111011010', '110011
```

```
1
```

✓ 0s completed at 11:13 PM