Genetic algorithms

Evans Wahome Gichuki
6677196, eg00850@surrey.ac.uk

Abstract

Genetic algorithms are a type of algorithms that aim to optimize/ find the maximum or minimum of a function. In this paper we introduce, illustrate, and discuss genetic algorithms. We discuss what components make up genetic algorithms and how they are developed to solve certain minimization problems. Using Python's DEAP library, we solve several problems, using gradient descent, genetic algorithms, and particle swarm optimization.

Introduction

Genetic algorithms represent one branch of the field of study called evolutionary computation [3], This means that they imitate the biological processes of reproduction and natural selection to come up with better optimized solutions. Like in evolution, many of a genetic algorithm's processes are random, however genetic algorithms allow the developer to set the level of control and randomization. These algorithms are far more powerful and efficient than random search and exhaustive search algorithms [3] which tend to take on a brute force approach hence can take a long time to compute depending on the complexity of the problem. Genetic algorithms have shown to be better as they require no extra information about the given problem. This inturn allows them to find solutions to problems that other optimization methods cannot handle**.**

1 Evolutionary algorithms

These types of problems consist of an objective function(e.g minimize/maximize), decision variables(possible solutions e.g population of chromosomes), selection process for the parents, crossover(to produce next generation of chromosomes), random mutation and constraints(x can only be between -2 and +2)

Optimization is trying to find minima or maxima of equations. In Genetic algorithms we create a class of population based on guided stochastic search heuristics inspired from biological evolution.

Each individual is a candidate solution to the problem. Many individuals put together form the population. The larger the population the faster you can get to a solution. Each individual is associated with a fitness value.

With single objective optimization, we have one objective function and one decision variable. If we wanted to minimize, we would take the lowest point in the function. This can be done using gradient descent, genetic algorithms, particle swarm optimization. For such problems, one single optimum solution can be found

$$f(x_1, x_2) = 2 + 4.1\, x_1^2 - 2.1\, x_1^4 + \frac{1}{3} x_1^6 + x_1 x_2 - 4\,(x_2 - 0.05)^2 + 4\, x_2^4$$

Using the above function as our fitness function, a population size of 50 and 30 bits per chromosome

a) The fittest individual after 100 generations was BestIndividual: [1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0] which had a fitness of (1.19753681047364,). Its decoded values for x1 and x2 were 0.29039354994893074 and -0.7204980496317148 respectively.

b) Fittest individual across the generations. The fitness of individuals is shown to decrease. This is good as the problem being a minimization problem we would want the value with the smallest fitness

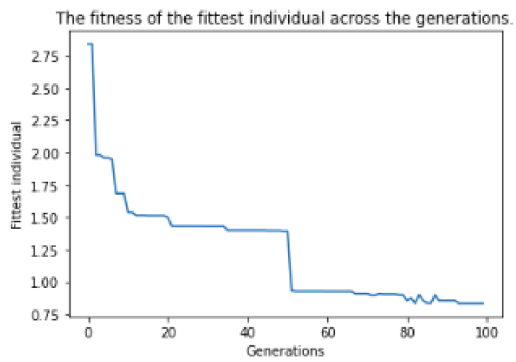
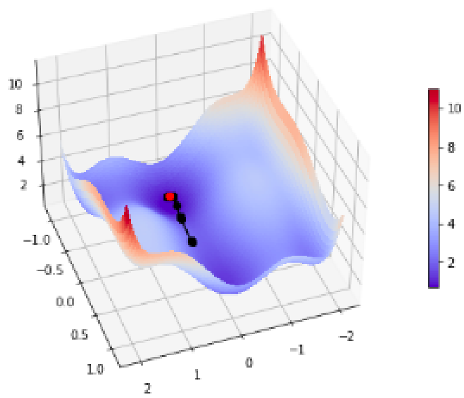


Fig 1.1.0 lowering of fitnesses across generations

c) The fittest individuals across generations are shown with the black line, which finishes with the red dot that shows the fittest individual after the 100th generation. The points would normally converge to the local/global minima of the plane which is shown by the darker blue regions.



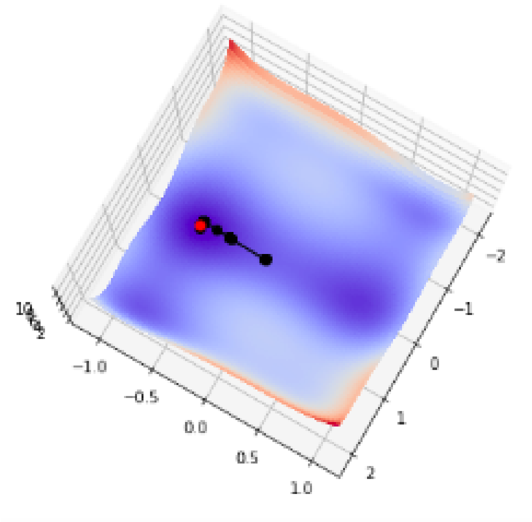


Fig 1.1.1 fittest individuals across the generations

1.2 Gradient descent

In Gradient descent, we have an initial value x0 and we want to find the next best value i.e x1. In the case of a minimization problem, we want it to be the lowest no possible. The new value of x is gotten using the below formula

$$x^{(k+1)} = x^{(k)} + \alpha^{(k)} d^{(k)}$$

given

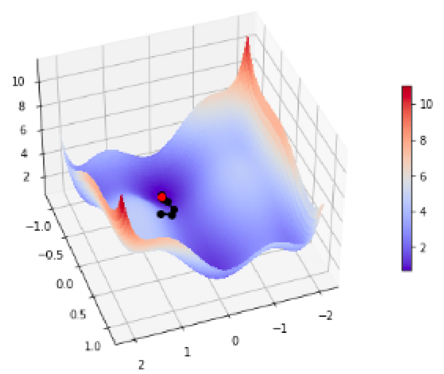
(1) d^k : the direction to go
(2) α^k : a scalar step size

given

D is The gradient of the objective function.

We use subtraction if it's a minimization problem and addition if it's a maximization problem.

The step size should not be too large or too small. If it's too large the search may diverge, and if it is too slow the search would be too slow. The disadvantage of the gradient descent is that there is no way to determine the right learning rate.



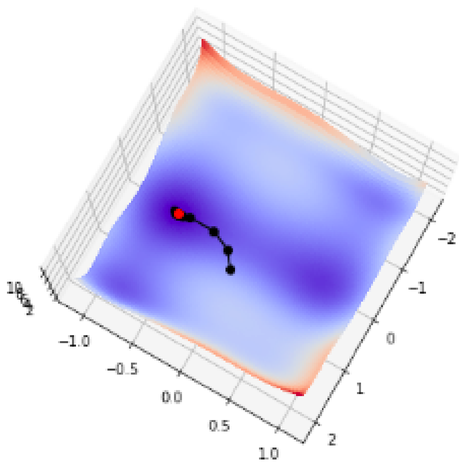


Fig 1.2.1 gradient descent in action

Q2 Particle swarm optimization

Particle swarm optimization (PSO) algorithm are based on swarms replicated from animal behaviour. These swarms are able to change the search pattern based on the experience of it own and other members.[1] We have particles that are possible solutions to our problem. These solutions move around to find a better solution to the problem. Each of these solutions has to keep a record of its best solution, we would also keep track of the global best for the whole swarm. Each particle will then try to move towards its personal best and the global best and will end up somewhere in between. It is also changing its velocity and position using the formula 1 and 2 shown below.

$$v_i(t+1) = w\, v_i(t) + c_1 r_{i1}(t)\,(p_i^{best} - x_i(t)) + c_2 r_{i2}(t)\,(g^{best} - x_i(t)) \quad (1)$$
$$x_i(t+1) = x_i(t) + v_i(t+1) \quad (2)$$

Fig 2.1.0 change in velocity and change in position

These two functions help the swarm arrive at the best possible solution.

$$f(X) = -\sum_{i=1}^{n}\left[x_i \sin\left(\sqrt{|x_i|}\right)\right]$$
$$-500 \le x_i \le 500\ , i = 1, 2, \cdots, n$$

Fig 2.1.0 PSO objective function and constraints

For this problem, the fitness function eval_sphere2() contains the formula in fig 2.1.1 which takes in a particle and returns its fitness. The updateParticle function takes in the current particle, the best particle and the weight value, this then updates the position of the current particle to one closer to the best particle's position.

a) The fitness of the fittest individual after 400 generations was (-1.466146129363015e+16,) with a best particle position of [276.89139446295155, 412.913611280705, 49.07923261279501, 223.22461569724211, 397.37212202545425, -399.12811329083945, -189.45823832743216, 394.4365757985657, -299.3970769185938, 205.84771636268508, 52.50406260594047, -325.642560812928, -338.6742839688656, -264.57984526328966, 42.50114038265769, 363.7351292122668, -311.9756832122891, -30.717099275866904, 50.16611838258484, 426.36008378859106]

The fitness of the global best individuals across generations using:

b) Canonical PSO

In canonical PSO we update the position and velocity of the particles in regards to any better particle(learning)[2]. This is done using the formula in fig 2.1.0. The output came out as expected as the minima was decreasing per generation and the last generation showed to have the lowest value.

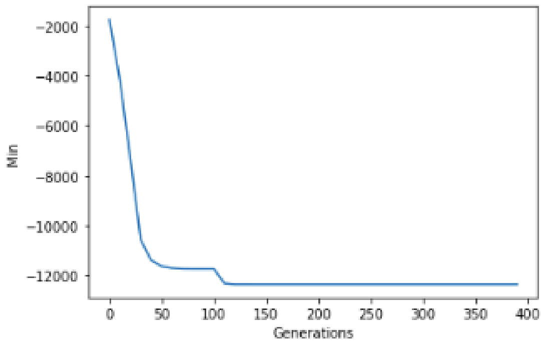


Fig 2.1 canonical PSO

c) Social learning PSO (SL-PSO)

In social learning we first sort all the individuals from best to worst based on fitness. We then update the position and velocity of the particles in regards to any better particle(learning) this is done using the formula in fig 2.1.0. Continue the above steps until termination [1]. The output showed us getting closer to the optimal minima as the min showed to decrease across the generations.

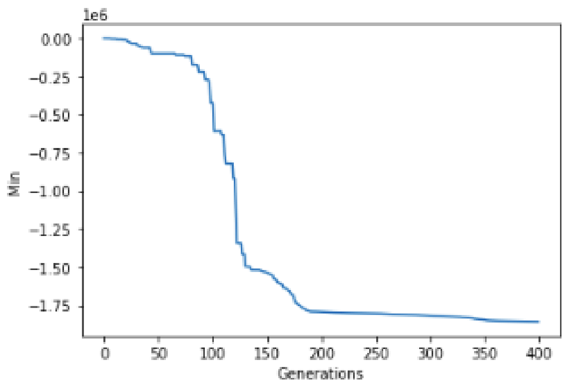


Fig 2.2 social learning PSO

Question 3

3.1

In a genotype to phenotype map, we want a small change in the genotype to result in small changes in the phenotype and large changes in genotype resulting in large changto vary in 1 bites in phenotype. To do this, we use a technique known as gray coding as it only allows two successive values .

In the Decision space i.e x1,x2,x3 these are decision variables that show us what is being measured. While in the objective space i.e f1,f2 we have factors such as price and quality. Basically, every point in the decision space can undergo the functions f1 and f2 and be placed in the objective space. While doing Multi objective optimization we normally look at the decisions in the objective space. Decision space == chromosome values and Objective space == fitness

We use a multi objective minimizing fitness named FitnessMin, which will take two negative weights which help the DEAP library know its a multi objective minimization problem. Due to the different types of evolution algorithms, a large variety of individuals are possible. For this problem we will initialize our individuals as a list of boolean values i.e 1 and 0.

We start off by declaring the MU as 24 as instructed. We then create a population with 24 random individuals and assign them to the pop variable. We will 1st check if the individual has proper fitness, which they don't. We then proceed to evaluate each individual to get their fitness. This is done using the fitness function (fitnessFunction). As we are using binary coding representation, the function takes an individual of length 30, separates it into 3 sub arrays also known as the decision variables, and passes each sub array to the chrom2real function which will return the real-valued decoded decision variable of each sub array The function responsible for the real-valued conversion is as shown below.

$$x_i = a_i + (b_i - a_i)\frac{1}{2^l - 1}\left(\sum_{j=0}^{l-1} s_j 2^j\right)$$

Fig 3.1.1 real value conversion formula

The separatevariables() function returns the decoded values of the decision variables. i.e x1,x2, and x3. These are also known as the phenotypes. These will be passed as variables into two functions i.e fctn1() and fctn2(). These functions will be responsible for returning the F1 and F2 values respectively using the formulas below.

$$f_1 = [((x_1\text{-}0.6)/1.6)^2 + (x_2/3.4)^2 + (x_3\text{-}1.3)^2] / 2.0$$

$$f_2 = [(x_1/1.9 - 2.3)^2 + (x_2/3.3 - 7.1)^2 + (x_3 + 4.3)^2] / 3.0$$

$$-4.0 \leq x_1, x_2, x_3 \leq 4.0$$

Fig 3.1.2 f1 and f2

The values returned will be appended to an array named table which will be returned as output in the form of a dataframe as shown below.

	x1	x2	x3	f1	f2
0	-0.898438	0.906250	-2.132812	6.366162	19.657087
1	-1.109375	-2.554688	-2.406250	7.721125	24.635088
2	3.007812	-1.367188	-1.742188	5.840636	21.173701
3	-1.539062	2.132812	-1.656250	5.460128	19.437291
4	-1.625000	-1.640625	3.531250	3.572578	43.000329
5	-3.710938	-1.273438	-0.898438	6.116427	28.566084
6	-1.578125	-3.203125	0.789062	1.500908	33.611486
7	-2.054688	-2.265625	1.875000	1.763769	36.731666
8	-2.281250	-0.546875	2.664062	2.564675	37.847818
9	-0.398438	3.023438	2.125000	0.930395	28.606235
10	-2.609375	0.453125	0.726562	2.185032	29.079638
11	1.046875	-0.570312	2.437500	0.700025	33.782298
12	-0.968750	3.351562	2.367188	1.535959	29.788790
13	-2.343750	-2.132812	0.750000	2.040514	32.664588
14	-3.039062	-0.804688	-0.335938	3.952632	28.283999
15	-2.453125	3.054688	2.945312	3.577741	34.504375
16	-2.765625	3.773438	2.765625	3.902284	33.169276
17	-3.468750	3.445312	-3.531250	15.417249	18.095584
18	-2.664062	3.929688	3.914062	6.165467	38.698375
19	-1.554688	3.382812	2.101562	1.722982	29.202667
20	2.031250	-2.695312	-0.039062	1.610855	27.448622
21	-1.085938	0.492188	-3.218750	10.775182	19.243075
22	-2.156250	2.195312	-1.671875	6.108244	20.037139
23	-1.187500	1.648438	2.937500	2.082289	34.834427

Fig 3.1.3 decode x1,x2,x3 and f1,f2 values

3.2 Efficient non dominated sorting

There are several sorting algorithms used in genetic algorithms.

1. Non Dominated sorting(O M N3)
2. Fast non dominated sorting (O M N2)
3. Efficient non dominated sorting. Best case (O M sqrt(N)) which could be reduced to (O MN log(N)) and worst case (O M N2)

given M being the number of objectives and N being the population size. From the above we can see the Efficient non dominated sorting provides the best run time.

	Front	F1	F2
0	0	6.328515	30.413456
3	0	6.640172	17.468222
13	0	8.414510	15.380566
1	1	4.885158	34.749188
2	1	4.882650	30.924723
4	1	5.139072	25.453437
6	1	4.923927	24.332499
14	1	11.462664	16.930566
9	1	14.995199	21.500056
10	1	10.218716	17.770034
16	2	14.049633	19.893223
23	2	10.177581	18.265319
8	2	0.568623	27.178727
5	2	0.503241	34.934040
12	3	0.729085	30.895435
7	3	1.903150	34.948416
17	3	4.597864	27.289920
18	3	16.337955	26.316725
15	4	3.134038	35.305735
21	4	3.212434	31.459975
22	4	3.037531	29.189122
11	4	3.541547	41.562229
19	5	4.156666	39.799240
20	6	4.337947	46.946347

Fig 3.2.1 sorted fronts f1 and f2

The table above shows the worst objective i.e f 1 : f 1 * = max{f 1}=16.3379 and f 2 : f 2 * = max{ f 2 } = 46.94 value shaded in yellow above.

The values in the table above were then plotted in 2d showing the respective fronts as shown below.

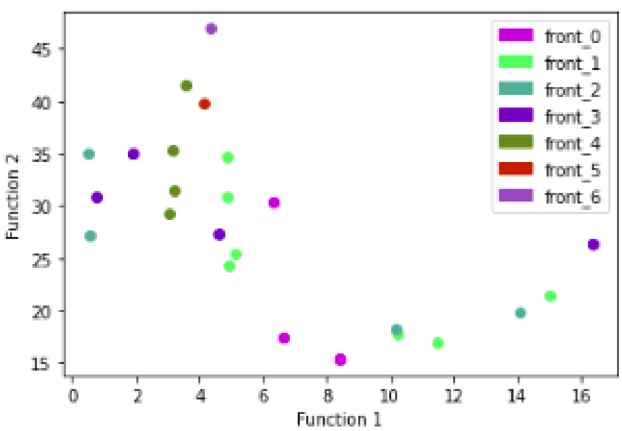


Fig 3.2.2 plot showing individuals and fronts

3.3 Crowding distance

The main aim of crowding distance is to calculate the spread and distribution of the solutions. E.g what are the best 3 solutions from the same front?

Based on crowding distance, this would be the solutions most far apart as they would give more diversity in the solution space. Items closer together have a lower crowding distance as they don't have that spread/ distribution required and will therefore be quite similar. We would normally assign a very large value to the 2 individuals furthest from each other(extreme solutions) in the same front. For my solution I used the already present value 4444444444444444.

For crowding distance we would calculate the average side length of two of the current solutions , neighbouring solutions.

	F1	F2	Front	Crowding Distance
0	6.328515	6.328515	0	4.444444e+15
1	6.640172	6.640172	0	6.079756e-01
2	8.414510	8.414510	0	4.444444e+15
3	4.885158	4.885158	1	4.444444e+15
4	4.882650	4.882650	1	1.473676e-01
5	5.139072	5.139072	1	2.239333e-01
6	4.923927	4.923927	1	4.596212e-01
7	14.995199	14.995199	1	6.081908e-01
8	10.218716	10.218716	1	5.961345e-01
9	11.462664	11.462664	1	4.444444e+15
10	0.503241	0.503241	2	4.444444e+15
11	0.568623	0.568623	2	8.933334e-01
12	14.049633	14.049633	2	1.327849e+00
13	10.177581	10.177581	2	4.444444e+15
14	1.903150	1.903150	3	4.444444e+15
15	0.729085	0.729085	3	3.893756e-01
16	4.597864	4.597864	3	1.154213e+00
17	16.337955	16.337955	3	4.444444e+15
18	3.541547	3.541547	4	4.444444e+15
19	3.134038	3.134038	4	2.048191e-01
20	3.212434	3.212434	4	3.457733e-01
21	3.037531	3.037531	4	4.444444e+15
22	4.156666	4.156666	5	4.444444e+15
23	4.337947	4.337947	6	4.444444e+15

Fig 3.3.1 crowding distance per individual

3.4

We spawn 24 parent individuals. These individuals are then passed to the fitness function where they are split into x1,x2, and x3 . These values will then be passed as variables to the formulas in fig 3.1.2 which returns a decoded f1 and f2 value.

NSGA-II Mate Selection: Tournament selection with MOO

Tournament selection using crowding distance and dominance was then used to select the parents with the best fitnesses

- Choose two solutions randomly
- The solution with the better(lower) rank wins.
- If the solutions have the same rank, the one with the larger crowding distance wins.
- If the 2 solutions have the same rank and the same crowding distance, choose a winner randomly.

We then created 24 offspring individuals who made the population grow to 48. We then applied uniform crossover using the tools.cxUniform within the mate function with a probability of 0.9. We then applied a mutation at a probability of 0.03333. This then created the population as shown below.

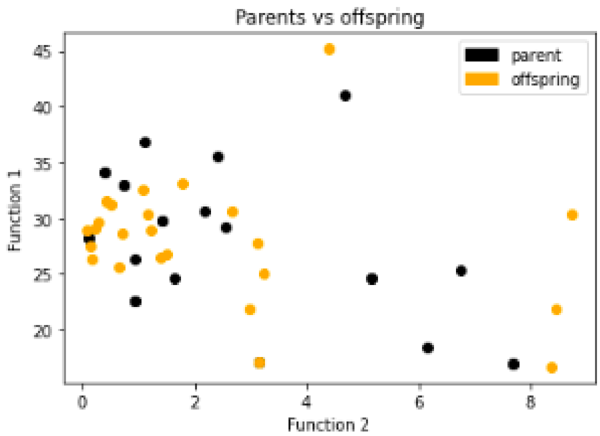


Fig 3.4.1 parent population vs offspring population

3.5

The set of all the pareto optimal solutions is called the pareto set. These solutions dominate all other solutions. (a set of all the optimal solutions). Being on the pareto front means you cannot make one thing better without making the other worse. The best individuals will mostly line up along the pareto front as below.

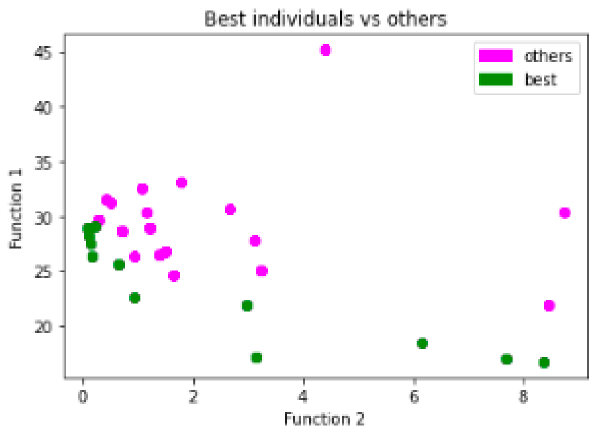


Fig 3.5.1 best individuals vs the remaining population

3.6

With the D- metrics, you need to have the reference set present to compare with.. You might be dealing with a hard optimization problem which sometimes doesn't have a reference set. THis is however different from Hypervolume.

Hyper-volume needs some Nadir point(The worst possible point). This could be the individual with the worst score. In this solution we used the values with the largest F1 and F2 score i.e MAX(F1) and MAX(F2) as it is a minimization problem. The choice of Nadir points affects the results we will get. This is one of the drawbacks of this technique as it requires a decent nadir point

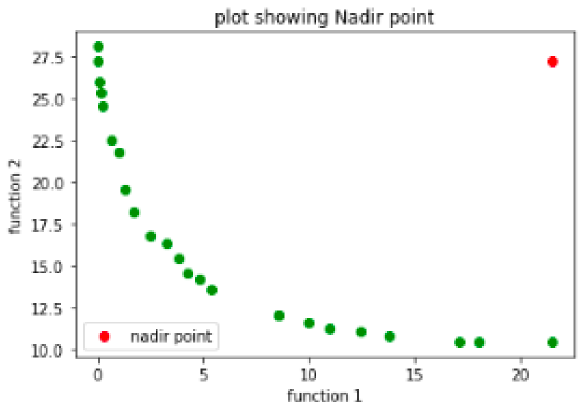


Fig 3.6.1 hypervolume graph showing the Nadir point

The green points are the solutions that came out of the algorithm. Because we are dealing with 2 objective functions/ loss functions/ cost functions i.e (f1,f2). We want to find the area(2 dimensions) of the figure that forms after joining the points. If we had 3 objective functions, we would get the volume. If we had 4 objective functions we would get the volume in objective space which is called the hypervolume. (NB: we use hyper when we are going to higher dimensions).

The area of the shape is the hypervolume.

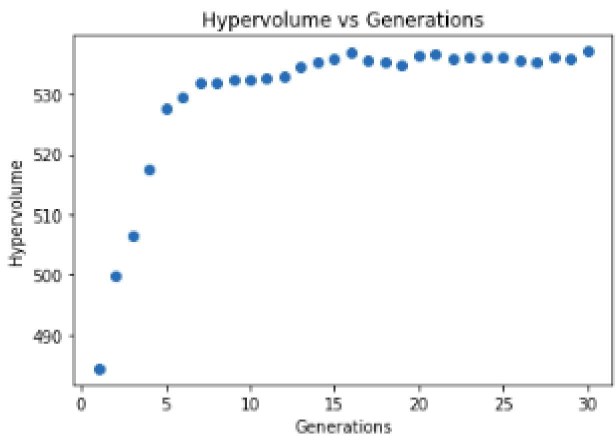


Fig 3.6.2 hypervolume per generation

From the graph above we can see the hypervolume increase as the generations increase. This is good as the larger the value of H the better.This means you have gotten closer to the pareto front wherever it is. It also means that you have a better spread. If the spread and diversity was poor the volume would be smaller.

References

[1] 191-205, 2015 [2] R. Cheng and Y. Jin. A social learning particle swarm optimization algorithm for scalable optimization. Information Sciences,

[2)]R. Cheng and Y. Jin. A competitive swarm optimizer for large-scale optimization. IEEE Transactions on Cybernetics, 45

[3] Kinnear, K. E. (1994). A Perspective on the Work in this Book. In K. E. Kinnear (Ed.), Advances in Genetic Programming (pp. 3-17). Cambridge: MIT Press.

Appendix

1

a

```
popSize     = 50 #Population size

dimension   = 2 #Number of decision variable x

numOfBits   = 30 #Number of bits in the chromosomes

iterations  = 100 #Number of generations to be run

dspInterval = 10

nElitists   = 1 #number of elite individuals selected

omega       = 5

crossPoints = 2 #variable not used. instead tools.cxTwoPoint

crossProb   = 0.6

flipProb    = 1. / (dimension * numOfBits) #bit mutate prob

mutateprob  = .1 #mutation prob

maxnum      = 2**numOfBits #absolute max size of number coded by binary list 1,0,0,1,1,....


creator.create("FitnessMax", base.Fitness, weights=(1.0,)) #only supports maximization(-1 for minimization)

creator.create("Individual", list, fitness=creator.FitnessMax)


# the goal ('fitness') function to be maximized

# fitness function: sphere model

#

#   f(x1,x2) = 2 + (4.1*x1**2)-(2.1*x1**4)+(1/3*x1**6)+(x1*x2)-4(x2-0.05)**2+(4*x2**4)

def eval_sphere(individual):#this is our fitness function

    # the individual has x1 and x2 inside of it(20 bits).

#    print("individual: ",individual)

    sep=separatevariables(individual)#separate x1 and x2
```

```
#     print("x1: ",sep[0])

#     print("x2: ",sep[1])

    f=
2+4.1*(sep[0]**2)-2.1*(sep[0]**4)+1/3*(sep[0]**6)+sep[0]*sep[1
]-4*(sep[1]-0.05)**2+4*sep[1]**4

#     print("fitness: ",f)

    return 1.0/(0.01+f), # DEAP doesn't allow minimisation for
roulette selection

                    # so we convert to maximisation


toolbox = base.Toolbox()


# Attribute generator

#                   define 'attr_bool' to be an attribute ('gene')

#                   which corresponds to integers sampled uniformly

#                   from the range [0,1] (i.e. 0 or 1 with equal

#                   probability)

toolbox.register("attr_bool", random.randint, 0, 1)


# Structure initializers

#                     define 'individual' to be an individual

#                          consisting of numOfBits*dimension 'attr_bool'
elements ('genes')

toolbox.register("individual", tools.initRepeat, creator.Individual,

    toolbox.attr_bool, numOfBits*dimension)#initializing the
individual size


# define the population to be a list of individuals

toolbox.register("population", tools.initRepeat, list,
toolbox.individual)


# register the goal / fitness function

toolbox.register("evaluate", eval_sphere)


# register the crossover operator

toolbox.register("mate", tools.cxTwoPoint) #two point crossover


# register a mutation operator with a probability to

# flip each attribute/gene of 0.05

toolbox.register("mutate", tools.mutFlipBit, indpb=flipProb)


# operator for selecting individuals for breeding the next

# generation: This uses fitness proportionate selection,

# also known as roulette wheel selection

toolbox.register("select", tools.selRoulette, fit_attr='fitness')


# Convert chromosome to real number

# input: list binary 1,0 of length numOfBits representing number
using gray coding

# output: real value

def chrom2real(c):#what weve been doing in class on paper

    indasstring=''.join(map(str, c))

#     print("individual as string: ",indasstring)

    degray=gray_to_bin(indasstring)

    numasint=int(degray, 2) # convert to int from base 2 list

#     print("number as int: ",numasint)

    numinrange=-5+10*numasint/maxnum

    return numinrange


# input: concatenated list of binary variables

# output: tuple of real numbers representing those variables

def separatevariables(v):

    return chrom2real(v[0:numOfBits]),chrom2real(v[numOfBits:])


def main():

    import random

    #random.seed(64)

    maxlist=[]

    x1vals = []

    x2vals = []

    zvals = []

    combined = []


    pop = toolbox.population(n=popSize)
```

```
    for individ in pop:

        sep=separatevariables(individ)

    # Evaluate the fitness of the entire population
    fitnesses = list(map(toolbox.evaluate, pop))
#    print("fitnesses: ",fitnesses)
      for ind, fit in zip(pop, fitnesses):#pair up the fitnesses to the individual
#        print("---: ",ind, fit)
        ind.fitness.values = fit # assign the fitness to the individual

    print("  Evaluated %i individuals" % len(pop))

    # Extracting all the fitnesses of
    fits = [ind.fitness.values[0] for ind in pop]
#    print("fits",fits)

    # Variable keeping track of the number of generations
    g = 0

    # Begin the evolution
    while g < iterations:
#        print("BEST GUY: ",tools.selBest(pop, 1)[0])
        # A new generation
        g = g + 1
        print("-- Generation %i --" % g)

        # Select the next generation individuals
          #selBest selects the best individual while select uses roullete wheel
                        offspring = tools.selBest(pop, nElitists) + toolbox.select(pop,len(pop)-nElitists)
        # Clone the selected individuals
        offspring = list(map(toolbox.clone, offspring))

        # Apply crossover and mutation on the offspring
        # make pairs of offspring for crossing over
        for child1, child2 in zip(offspring[::2], offspring[1::2]):
#           print("******************************")
            # cross two individuals with probability CXPB
            if random.random() < crossProb:
#               print('before crossover ',child1, child2)
                toolbox.mate(child1, child2)

                del child1.fitness.values
                del child2.fitness.values

        for mutant in offspring:

            # mutate an individual with probability mutateprob
            if random.random() < mutateprob:
                toolbox.mutate(mutant)
                del mutant.fitness.values

        # Evaluate the individuals with an invalid fitness
        invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
        fitnesses = map(toolbox.evaluate, invalid_ind)
        for ind, fit in zip(invalid_ind, fitnesses):
            ind.fitness.values = fit

        pop[:] = offspring
```

```
    fits = []
    i=0
    maximum = 0
    vals = []
    for ind in pop:
        i+=1
        x1,x2 =separatevariables(ind)
        x1vals.append(x1)
        x2vals.append(x2)
        zvals.append(ind.fitness.values[0])

        if ind.fitness.values[0] > maximum:
            maximum = ind.fitness.values[0]
            vals = [x1,x2,ind.fitness.values[0]]


        fits.append(ind.fitness.values[0])

    combined.append(vals)

    maxlist.append(1/max(fits))
    length = len(pop)
    mean = sum(fits) / length
    sum2 = sum(x*x for x in fits)
    std = abs(sum2 / length - mean**2)**0.5

    print("  Min %s" % min(fits))
    print("  Max %s" % max(fits))
    print("  Avg %s" % mean)
    print("  Std %s" % std)

print("-- End of (successful) evolution --")
    best_ind = tools.selBest(pop, 1)[0]
    print("Best individual is %s, %s" % (best_ind, best_ind.fitness.values))
    print("Decoded x1, x2 is %s, %s" % (separatevariables(best_ind)))
#   plt.plot(maxlist)
    res = {
        "BestIndividual":best_ind,
        "fitness":best_ind.fitness.values,
        "x1":separatevariables(best_ind)[0],
        "x2":separatevariables(best_ind)[1],
        "maxlist":maxlist,
        "x1vals":x1vals,
        "x2vals":x2vals,
        "zvals":zvals,
        "combined":np.array(combined)

    }
    return res
answer = main()
print("BestIndividual: ",answer["BestIndividual"])
print("fitness: ",answer["fitness"])
print("x1: ",answer["x1"])
print("x2: ",answer["x2"])
```

b

```
plt.plot(answer["maxlist"])
plt.title("The fitness of the fittest individual across the generations.")
plt.xlabel("Generations")
plt.ylabel("Fittest individual")
plt.show()
```

C

```
# answer = main()
def f(x1,x2):
```

```
f= 2+(4.1*(x1**2))-(2.1*(x1**4))+(1/3*(x1**6))+(x1*x2)-(4*(x2-0.05)**2)+(4*x2**4)

#    print("F: ",f)

    return f


xrange = np.linspace(-2.1, 2.1, 100)

yrange = np.linspace(-1.1, 1.1, 100)

X,Y = np.meshgrid(xrange, yrange)

Z = f(X, Y)


zvals = answer["combined"][:,2].tolist()

x2vals = answer["combined"][:,1].tolist()

x1vals = answer["combined"][:,0].tolist()


fig = plt.figure(figsize=(26,6))


# surface_plot with color grading and color bar

ax = fig.add_subplot(1, 2, 1, projection='3d')

p = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=matplotlib.cm.coolwarm, linewidth=0, antialiased=False, zorder=0)

ax.plot3D(x1vals, x2vals,zvals, color="k", marker='o', zorder=10)

ax.plot3D(x1vals[-1], x2vals[-1],zvals[-1], color="red", marker='o', zorder=10)

ax.view_init(40, 70)


cb = fig.colorbar(p, shrink=0.5)


ax = fig.add_subplot(1, 2, 2, projection='3d')

ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=matplotlib.cm.coolwarm, linewidth=0, antialiased=False, zorder=0)

ax.plot3D(x1vals, x2vals,zvals, color="k", marker='o', zorder=10)

ax.plot3D(x1vals[-1], x2vals[-1],zvals[-1], color="red", marker='o', zorder=10)

ax.view_init(80, 30)
```

1.2

```
def dx1(x1,x2):#calculates change

    return 8.2*x1-8.4*x1**3 + 2*x1**5 + x2


def dx2(x1,x2):

    return x1-8*x2+0.4+16*x2**3

# x1=1

# x2=1

# import random

x1 = random.randint(-2,2)

x2 = random.randint(-1,1)


xlist=[]

ylist=[]

zlist=[]

alpha=0.1

for step in range (0,30):

    newx1=x1-alpha*(dx1(x1,x2))

    x2=x2-alpha*(dx2(x1,x2))

    x1=newx1

    z=f(x1,x2)

    xlist.append(x1)

    ylist.append(x2)

    zlist.append(z)


fig = plt.figure(figsize=(26,6))


# surface_plot with color grading and color bar

ax = fig.add_subplot(1, 2, 1, projection='3d')

p = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=matplotlib.cm.coolwarm, linewidth=0, antialiased=False, zorder=0)

ax.plot3D(xlist, ylist,zlist, color="k", marker='o', zorder=10)

ax.plot3D(xlist[-1], ylist[-1],zlist[-1], color="red", marker='o', zorder=10)

ax.view_init(40,70)


cb = fig.colorbar(p, shrink=0.5)
```

```
ax = fig.add_subplot(1, 2, 2, projection='3d')

ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=matplotlib.cm.coolwarm, linewidth=0, antialiased=False, zorder=0)

ax.plot3D(xlist, ylist,zlist, color="k", marker='o', zorder=10)

ax.plot3D(xlist[-1], ylist[-1],zlist[-1], color="red", marker='o', zorder=10)

ax.view_init(80, 30)
```

2

a

```
posMinInit      = -500
posMaxInit      = + 500
VMaxInit        = 1.5
VMinInit        = 0.5
populationSize  = 50
dimension       = 20
interval        = 10
iterations      = 400
# num_neighbours  = 5

#Parameter setup

wmax = 0.9 #weighting
wmin = 0.4
c1   = 2.0
c2   = 2.0

creator.create("FitnessMin", base.Fitness, weights=(-1.0,)) # -1 is for minimise
creator.create("Particle", list, fitness=creator.FitnessMin, speed=list, smin=None, smax=None, best=None)

def generate(size, smin, smax):
    part = creator.Particle(random.uniform(posMinInit, posMaxInit) for _ in range(size))
    part.speed = [random.uniform(VMinInit, VMaxInit) for _ in range(size)]
    part.smin = smin #speed clamping values
    part.smax = smax
    return part

def updateParticle(part, best, weight):
    #implementing speed = 0.7*(weight*speed + c1*r1*(localBestPos-currentPos) + c2*r2*(globalBestPos-currentPos))
    #Note that part and part.speed are both lists of size dimension
    #hence all multiplies need to apply across lists, so using e.g. map(operator.mul, ...

    r1 = (random.uniform(0, 1) for _ in range(len(part)))
    r2 = (random.uniform(0, 1) for _ in range(len(part)))

    v_r0 = [weight*x for x in part.speed]
    v_r1 = [c1*x for x in map(operator.mul, r1, map(operator.sub, part.best, part))] # local best
    v_r2 = [c2*x for x in map(operator.mul, r2, map(operator.sub, best, part))] # global best

    part.speed = [0.7*x for x in map(operator.add, v_r0, map(operator.add, v_r1, v_r2))]
for j, speed in enumerate(part.speed):
      if abs(speed) < part.smin:
         part.speed[j] = math.copysign(part.smin, speed)
      elif abs(speed) > part.smax:
         part.speed[j] = math.copysign(part.smax, speed)

    # update position with speed
    part[:] = list(map(operator.add, part, part.speed))

def f2(individual):

    f=(-np.sum(x*np.sin(np.sqrt(np.abs(x)))for x in individual))
    return f

def eval_sphere2(particle):#this is our fitness function
#    print("heeerree: ",particle)
    z = f2(particle)
    return (z,)

toolbox = base.Toolbox()
toolbox.register("particle", generate, size=dimension, smin=-3, smax=3)
toolbox.register("population", tools.initRepeat, list, toolbox.particle)
toolbox.register("update", updateParticle)
toolbox.register("evaluate", eval_sphere2) #sphere function is built-in in DEAP

def main2():
    pop = toolbox.population(n=populationSize) # Population Size
    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("avg", numpy.mean)
    stats.register("std", numpy.std)
    stats.register("min", numpy.min)
    stats.register("max", numpy.max)

    logbook = tools.Logbook()
    logbook.header = ["gen", "evals"] + stats.fields

    best = None

    #begin main loop
    for g in range(iterations):
      w = wmax - (wmax-wmin)*g/iterations #decaying inertia weight
```

```
    for part in pop:
      part.fitness.values = toolbox.evaluate(part) #actually only
one fitness value

      #update local best
      if (not part.best) or (part.best.fitness < part.fitness):
#lower fitness is better (minimising)
      #   best is None   or  current value is better              #< is
overloaded
        part.best = creator.Particle(part)
        part.best.fitness.values = part.fitness.values

      #update global best
      if (not best) or best.fitness < part.fitness:
        best = creator.Particle(part)
        best.fitness.values = part.fitness.values

    for part in pop:
      toolbox.update(part, best,w)

    # Gather all the fitnesses in one list and print the stats
    # print every interval
    if g%interval==0: # interval
      logbook.record(gen=g, evals=len(pop),
**stats.compile(pop))
      print(logbook.stream)
      #print('best ',best, best.fitness)

  print('best particle position is ',best)
  return pop, logbook, best

pop, logbook, best = main2()
print(best.fitness.values)
```

b

```
yvalues = []
xvalues=[]
for i in logbook:
  yvalues.append(i["min"])
  xvalues.append(i["gen"])

plt.plot(xvalues,yvalues)
plt.xlabel("Generations")
plt.ylabel("Min")
plt.show()
```

C

```
posMinInit      = -500
posMaxInit      = + 500
VMaxInit        = 1.5
VMinInit        = 0.5
dimension       = 20
interval        = 10
iterations      = 400#50*dimension
populationSize  = 100+int(dimension/10)
# print("tis",populationSize)

#variables used in SL-PSO
epsilon = dimension/100.0*0.01 # social influence of swarm centre

# function to get the mean positions of the inviduals (swarm
centre)
def getcenter(pop):
  center=list()
  for j in range(dimension): # count through dimensions
    centerj = 0
    for i in pop: # for each particle
      centerj += i[j] # sum up position in dimention j
    centerj /= populationSize # Average
    center.append(centerj)
  return center

creator.create("FitnessMin", base.Fitness, weights=(-1.0,)) # -1 is
for minimise
creator.create("Particle", list, fitness=creator.FitnessMin,
speed=list, smin=None, smax=None, best=None)

def generate(size, smin, smax):
  part = creator.Particle(random.uniform(posMinInit, posMaxInit)
for _ in range(size))
  part.speed = [random.uniform(VMinInit, VMaxInit) for _ in
range(size)]
  part.smin = smin #speed clamping values
  part.smax = smax
  return part

def updateParticle(part,pop,center,i):
  r1 = random.uniform(0, 1)
  r2 = random.uniform(0, 1)
  r3 = random.uniform(0, 1)

  #Randomly choose a demonstrator for particle i from any of
particles 0 to i-1, the Particle i
  #updates its velocity by learning from the demonstrator and the
mean position of the swarm
  demonstrator=random.choice(list(pop[0:i]))

  for j in range(dimension): # count through dimensions

part.speed[j]=r1*part.speed[j]+r2*(demonstrator[j]-part[j])+r3*eps
ilon*(center[j]-part[j])
    part[j]=part[j]+part.speed[j]

op=[]
def f3(individual):
  f=(-np.sum(x*np.sin(np.sqrt(np.abs(x)))for x in individual))
  return f

def eval_sphere3(particle):#this is our fitness function
  z = f3(particle)
  return (np.sum(z),)

toolbox = base.Toolbox()
toolbox.register("particle", generate, size=dimension, smin=-3,
smax=3)
toolbox.register("population", tools.initRepeat, list,
toolbox.particle)
```

```
toolbox.register("update", updateParticle)
toolbox.register("evaluate", eval_sphere3) #sphere function is
built-in in DEAP

def main3():
   pop = toolbox.population(n=populationSize) # Population Size
   stats = tools.Statistics(lambda ind: ind.fitness.values)
   stats.register("avg", numpy.mean)
   stats.register("std", numpy.std)
   stats.register("min", numpy.min)
   stats.register("max", numpy.max)

   #intialize the learning probabilities
   prob=[0]*populationSize
   for i  in range(len(pop)):
      prob[populationSize - i - 1] = 1 - i/(populationSize - 1)
      prob[populationSize - i - 1] = pow(prob[populationSize - i -
1], math.log(math.sqrt(math.ceil(dimension/100.0))))

   logbook = tools.Logbook()
   logbook.header = ["gen", "evals"] + stats.fields

   #begin main loop
   for g in range(iterations):

      for part in pop:
         part.fitness.values = toolbox.evaluate(part) #actually only
one fitness value

      #Sort the individuals in the swarm in ascending order. i.e.,
particle 0 is the best
      pop.sort(key=lambda x: x.fitness, reverse=True)
      #calculate the center (mean value) of the swarm
      center = getcenter(pop)

      for i  in reversed(range(len(pop)-1)):  # start with worst
particle, and go in reverse towards best
                                    # don't do element 0 (best). Hence
the i+1 below.
         if random.uniform(0, 1)<prob[i+1]: #learning probability
for that particle
            toolbox.update(pop[i+1],pop,center,i+1)

      # Gather all the fitnesses in one list and print the stats
      # print every interval
#        if g%interval==0: # interval
      logbook.record(gen=g, evals=len(pop), **stats.compile(pop))
      print(logbook.stream)

   return pop, logbook
pop, logbook2 = main3()
yvalues = []
xvalues=[]
for i in logbook2:
   yvalues.append(i["min"])
   xvalues.append(i["gen"])

plt.plot(xvalues,yvalues)
plt.xlabel("Generations")
plt.ylabel("Min")
plt.show()
```

3

3.1

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0, -1.0))
creator.create("Individual", list, fitness=creator.FitnessMin)

toolbox = base.Toolbox()
table = []

def chrom2real(c):
   indasstring=''.join(map(str, c))
   degray=gray_to_bin(indasstring)
   numasint=int(degray, 2) # convert to int from base 2 list
   maxnum      = 2**10
   numinrange=-4+8*numasint/maxnum
#     print("numinrange: ", numinrange)
   return numinrange

# input: concatenated list of binary variables
# output: tuple of real numbers representing those variables
def separatevariables(v):
   return
chrom2real(v[0:10]),chrom2real(v[10:20]),chrom2real(v[20:30])

def fctn1(x1,x2,x3):
   return (((x1-0.6)/1.6)**2+(x2/3.4)**2+(x3-1.3)**2)/2.0

def fctn2(x1,x2,x3):
   return ((x1/1.9-2.3)**2+(x2/3.3-7.1)**2+(x3+4.3)**2)/3.0



def fitnessFunction(individual):
   x1,x2,x3=separatevariables(individual)
   f1=fctn1(x1,x2,x3)
   f2=fctn2(x1,x2,x3)
   table.append([x1,x2,x3,f1,f2])
   return f1,f2

toolbox.register("attr_bool", random.randint, 0, 1)
toolbox.register("individual", tools.initRepeat,
creator.Individual,toolbox.attr_bool, 30)
toolbox.register("population", tools.initRepeat, list,
toolbox.individual)

toolbox.register("evaluate", fitnessFunction)
toolbox.register("mate", tools.cxTwoPoint)
flipProb=1.0/9
toolbox.register("mutate", tools.mutFlipBit, indpb=flipProb)
toolbox.register("select", tools.selNSGA2)

def main(seed=None):
#     random.seed(seed)
```

```
    MU = 24

    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("avg", numpy.mean, axis=0)
    stats.register("std", numpy.std, axis=0)
    stats.register("min", numpy.min, axis=0)
    stats.register("max", numpy.max, axis=0)

    logbook = tools.Logbook()
    logbook.header = "gen", "evals", "std", "min", "avg", "max"

    pop = toolbox.population(n=MU)
    print("len of pop",len(pop))

    # Evaluate the individuals with an invalid fitness
    invalid_ind = [ind for ind in pop if not ind.fitness.valid]
    print("invalid fitnesses individuals",len(invalid_ind))
    fitnesses = toolbox.map(toolbox.evaluate, invalid_ind)
    for ind, fit in zip(invalid_ind, fitnesses):
        ind.fitness.values = fit

    record = stats.compile(pop)
    logbook.record(gen=0, evals=len(invalid_ind), **record)

    return pop, logbook

pop, stats = main()
nwTable = pd.DataFrame(table,
columns=["x1","x2","x3","f1","f2"])
nwTable
```

3.2

```
#First function to optimize
def function1(x1,x2,x3):
    value = (((x1-0.6)/1.6)**2+(x2/3.4)**2+(x3-1.3)**2)/2.0
    return value

#Second function to optimize
def function2(x1,x2,x3):
    value = ((x1/1.9-2.3)**2+(x2/3.3-7.1)**2+(x3+4.3)**2)/3.0
    return value

#Function to find index of list
def index_of(a,list):
    for i in range(0,len(list)):
        if list[i] == a:
            return i
    return -1

#Function to sort by values
def sort_by_values(list1, values):
    sorted_list = []
    while(len(sorted_list)!=len(list1)):
        if index_of(min(values),values) in list1:
            sorted_list.append(index_of(min(values),values))
        values[index_of(min(values),values)] = math.inf
    return sorted_list


def efficient_non_dominated_sort(first, second):# efficient non
dominated sort
    unsorted=first
    first.sort()
#    print(first)
    frontvalues=[[unsorted.index(first[0])]]
    for i in range(len(first)-1):
        my_val=first[i+1]
        idx=unsorted.index(my_val)
        check=0
        counter=0
        for z in frontvalues:
            for j in z:
#                print(j)
                if second[idx]<second[j]:
                    check=1
                    m=idx

                else:
                    check=0
                    n=idx
                    counter+=1
                    break
#            print("z is: ",z)
        if check==1:
            frontvalues[counter].append(m)
        else:
            frontvalues.append([n])
#    print("these are: ",frontvalues)
    return frontvalues


#Function to calculate crowding distance
def crowding_distance(values1, values2, front):
    distance = [0 for i in range(0,len(front))]
    sorted1 = sort_by_values(front, values1[:])
    sorted2 = sort_by_values(front, values2[:])
    distance[0] = 4444444444444444
    distance[len(front) - 1] = 4444444444444444
    for k in range(1,len(front)-1):
        distance[k] = distance[k]+ (values1[sorted1[k+1]] -
values1[sorted1[k-1]])/(max(values1)-min(values1))
    for k in range(1,len(front)-1):
        distance[k] = distance[k]+ (values2[sorted2[k+1]] -
values2[sorted2[k-1]])/(max(values2)-min(values2))
    return distance

#Function to carry out the crossover
def crossover(a,b):
    r=random.random()
    if r>0.5:
        return mutation((a+b)/2)
    else:
        return mutation((a-b)/2)

#Function to carry out the mutation operator
```

```
def mutation(solution):
    mutation_prob = random.random()
    if mutation_prob <1:
        solution = min_x+(max_x-min_x)*random.random()
    return solution

#Main program starts here
pop_size = 24
max_gen = 1

#Initialization
min_x=-4
max_x=4

x1=[min_x+(max_x-min_x)*random.random() for i in
range(0,pop_size)]
x2=[min_x+(max_x-min_x)*random.random() for i in
range(0,pop_size)]
x3=[min_x+(max_x-min_x)*random.random() for i in
range(0,pop_size)]
print("x1 values",x1)
print("x2 values",x2)
print("x3 values",x3)
gen_no=0
while(gen_no<max_gen):
    function1_values = [function1(x1[i],x2[i],x3[i])for i in
range(0,pop_size)]
    function2_values = [function2(x1[i],x2[i],x3[i])for i in
range(0,pop_size)]
    non_dominated_sorted_solution =
efficient_non_dominated_sort(function1_values[:],function2_value
s[:])


    crowding_distance_values=[]
    for i in range(0,len(non_dominated_sorted_solution)):

crowding_distance_values.append(crowding_distance(function1_v
alues[:],function2_values[:],non_dominated_sorted_solution[i][:]))


    gen_no = gen_no + 1

Function1,function2

color_array = len(non_dominated_sorted_solution)
color = ["#"+''.join([random.choice('ABCDEF0123456789') for j
in range(6)]) for i in range(color_array)]
fronts_array = ["front_"+str(i) for i in range(color_array)]
print(color)
print(fronts_array)
fin = []
for i in range(color_array):
    fin.append(mpatches.Patch(color=color[i],
label="front_"+str(i)))for index,j in enumerate(function1_values):
    for i in range(0,len(non_dominated_sorted_solution)):
        if index in non_dominated_sorted_solution[i]:

plt.scatter(function1_values[index],function2_values[index],color=
color[i])

plt.xlabel("Function 1")
plt.ylabel("Function 2")
plt.legend(handles=fin,loc='best')
plt.show()
myarr = []
for index,j in enumerate(function1_values):
    for i in range(0,len(non_dominated_sorted_solution)):
        if index in non_dominated_sorted_solution[i]:

myarr.append([i,function1_values[index],function2_values[index]]
)

print("the worst values have been highlighted in yellow")
df = pd.DataFrame(myarr,
columns=["Front","F1","F2"]).sort_values(by=['Front'])
df.style.applymap(lambda x : "background-color: yellow" if (x ==
max(df["F1"]) or x == max(df["F2"]) ) else False)
```

3.3

```
Non_dominated_sorted_solution
f = 0
another_table = []
for m,n in
zip(non_dominated_sorted_solution,crowding_distance_values):
    for y,z in zip(m,n):

another_table.append([function1_values[y],function1_values[y],f,z
])
    f+=1

pd.DataFrame(another_table, columns=["F1","F2",
"Front","Crowding Distance"])
```

3.4

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0, -1.0))
creator.create("Individual", list, fitness=creator.FitnessMin)

toolbox = base.Toolbox()


def chrom2real(c):
    indasstring=''.join(map(str, c))
    degray=gray_to_bin(indasstring)
    numasint=int(degray, 2) # convert to int from base 2 list
    maxnum      = 2**10
    numinrange=-4+8*numasint/maxnum
    return numinrange

# input: concatenated list of binary variables
# output: tuple of real numbers representing those variables
def separatevariables(v):
```

```
    return
chrom2real(v[0:10]),chrom2real(v[10:20]),chrom2real(v[20:30])

def fctn1(x1,x2,x3):
    return (((x1-0.6)/1.6)**2+(x2/3.4)**2+(x3-1.3)**2)/2.0

def fctn2(x1,x2,x3):
    return ((x1/1.9-2.3)**2+(x2/3.3-7.1)**2+(x3+4.3)**2)/3.0


def calcFitness(individual):
    x1,x2,x3=separatevariables(individual)
    f1=fctn1(x1,x2,x3)
    f2=fctn2(x1,x2,x3)

    return f1,f2

#Function to carry out the crossover
def crossover(a,b):
    r=random.random()
    if r>0.5:
        return mutation((a+b)/2)
    else:
        return mutation((a-b)/2)

#Function to carry out the mutation operator
def mutation(solution):
    mutation_prob = random.random()
    if mutation_prob <1:
        solution = min_x+(max_x-min_x)*random.random()
    return solution


toolbox.register("attr_bool", random.randint, 0, 1)
toolbox.register("individual", tools.initRepeat, creator.Individual,
    toolbox.attr_bool, 30)
toolbox.register("population", tools.initRepeat, list,
toolbox.individual)

toolbox.register("evaluate", calcFitness)
toolbox.register("mate", tools.cxUniform)
flipProb=1.0/30
toolbox.register("mutate", tools.mutFlipBit, indpb=flipProb)
toolbox.register("select", tools.selNSGA2)

def main(seed=None):
    random.seed(seed)

    NGEN = 1
    MU = 24
    CXPB = 0.9

    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("avg", numpy.mean, axis=0)
    stats.register("std", numpy.std, axis=0)
    stats.register("min", numpy.min, axis=0)
    stats.register("max", numpy.max, axis=0)

    logbook = tools.Logbook()
    logbook.header = "gen", "evals", "std", "min", "avg", "max"

    pop = toolbox.population(n=MU)
    pop = toolbox.select(pop, len(pop))
    # Evaluate the individuals with an invalid fitness
    invalid_ind = [ind for ind in pop if not ind.fitness.valid]
    fitnesses = toolbox.map(toolbox.evaluate, invalid_ind)
    for ind, fit in zip(invalid_ind, fitnesses):
        ind.fitness.values = fit

    # This is just to assign the crowding distance to the individuals
    # no actual selection is done

    pop = tools.selTournamentDCD(pop, len(pop))

    record = stats.compile(pop)

    logbook.record(gen=0, evals=len(invalid_ind), **record)

    # Begin the generational process
    offspring = tools.selTournamentDCD(pop, len(pop))
    # selTournamentDCD means Tournament selection based on
dominance (D)
    # followed by crowding distance (CD). This selection requires
the
    # individuals to have a crowding_dist attribute
    offspring = [toolbox.clone(ind) for ind in pop]

    for ind1, ind2 in zip(offspring[::2], offspring[1::2]):
    #make pairs of all (even,odd) in offspring
        if random.random() <= CXPB:
            toolbox.mate(ind1, ind2,0.5)

        toolbox.mutate(ind1)
        toolbox.mutate(ind2)
        del ind1.fitness.values, ind2.fitness.values

    # Evaluate the individuals with an invalid fitness
    invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
    fitnesses = toolbox.map(toolbox.evaluate, invalid_ind)
    for ind, fit in zip(invalid_ind, fitnesses):
        ind.fitness.values = fit

    fin = []
    color = ["black","orange"]
    d = ["parent","offspring"]
    for i in range(2):
        fin.append(mpatches.Patch(color=color[i], label=d[i]))

    print("3.4")
    for a in pop:

plt.scatter(a.fitness.values[0],a.fitness.values[1],color="black")

    for a in offspring:

plt.scatter(a.fitness.values[0],a.fitness.values[1],color="orange")

    plt.xlabel("Function 2")
    plt.ylabel("Function 1")
    plt.title("Parents vs offspring")
```

```
    plt.legend(handles=fin)
    plt.show()




    fig1, ax1 = plt.subplots()
    # Select the next generation population
    pop = toolbox.select(pop + offspring, MU)
    best_individuals = toolbox.select(pop + offspring, MU)
    all_individuals=toolbox.select(pop + offspring, MU*2)

    # generate the legend
    fin = []
    color = ["magenta","green"]
    d = ["others","best"]
    for i in range(2):
       fin.append(mpatches.Patch(color=color[i], label=d[i]))
    #        plotting
```

3.5

```
    for p in all_individuals:

ax1.scatter(p.fitness.values[0],p.fitness.values[1],color="magenta",
label="others")
    for p in best_individuals:

ax1.scatter(p.fitness.values[0],p.fitness.values[1],color="green",lab
el="best")
    plt.xlabel("Function 2")
    plt.ylabel("Function 1")
    plt.title("Best individuals vs others")
    plt.legend(handles=fin)
    plt.show()



    record = stats.compile(pop)
    logbook.record(gen=1, evals=len(invalid_ind), **record)

    #print("Final population hypervolume is %f" %
hypervolume(pop, [11.0, 11.0]))

    return pop, logbook

if __name__ == "__main__":
    pop, stats = main()
```

3.6

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0, -1.0))
creator.create("Individual", list, fitness=creator.FitnessMin)

toolbox = base.Toolbox()

def chrom2real(c):
    indasstring=''.join(map(str, c))
    degray=gray_to_bin(indasstring)
    numasint=int(degray, 2) # convert to int from base 2 list
    maxnum      = 2**10
    numinrange=-4+8*numasint/maxnum
    return numinrange

# input: concatenated list of binary variables
# output: tuple of real numbers representing those variables
def separatevariables(v):
    return
chrom2real(v[0:10]),chrom2real(v[10:20]),chrom2real(v[20:30])


def calcFitness(individual):
    x1,x2,x3=separatevariables(individual)
    f1=(((x1-0.6)/1.6)**2+(x2/3.4)**2+(x3-1.3)**2)/2.0
    f2=((x1/1.9-2.3)**2+(x2/3.3-7.1)**2+(x3+4.3)**2)/3.0

    return f1,f2

#Function to carry out the crossover
def crossover(a,b):
    r=random.random()
    if r>0.5:
       return mutation((a+b)/2)
    else:
       return mutation((a-b)/2)

#Function to carry out the mutation operator
def mutation(solution):
    mutation_prob = random.random()
    if mutation_prob <1:
       solution = min_x+(max_x-min_x)*random.random()
    return solution


toolbox.register("attr_bool", random.randint, 0, 1)
toolbox.register("individual", tools.initRepeat, creator.Individual,
    toolbox.attr_bool, 30)
toolbox.register("population", tools.initRepeat, list,
toolbox.individual)

toolbox.register("evaluate", calcFitness)
toolbox.register("mate", tools.cxUniform)
flipProb=1.0/30
toolbox.register("mutate", tools.mutFlipBit, indpb=flipProb)
toolbox.register("select", tools.selNSGA2)

def main(seed=None):

    NGEN = 30
    MU = 24
    CXPB = 0.9

    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("avg", numpy.mean, axis=0)
    stats.register("std", numpy.std, axis=0)
```

```
    stats.register("min", numpy.min, axis=0)
    stats.register("max", numpy.max, axis=0)

    logbook = tools.Logbook()
    logbook.header = "gen", "evals", "std", "min", "avg", "max"

    pop = toolbox.population(n=MU)
    pop = toolbox.select(pop, len(pop))
    # Evaluate the individuals with an invalid fitness
    invalid_ind = [ind for ind in pop if not ind.fitness.valid]
    fitnesses = toolbox.map(toolbox.evaluate, invalid_ind)
    for ind, fit in zip(invalid_ind, fitnesses):
        ind.fitness.values = fit

    # This is just to assign the crowding distance to the individuals
    # no actual selection is done

    pop = tools.selTournamentDCD(pop, len(pop))


    record = stats.compile(pop)
    logbook.record(gen=0, evals=len(invalid_ind), **record)

    # Begin the generational process
    hypervals=[]
    for gen in range(0, NGEN):
        # Vary the population
        offspring = tools.selTournamentDCD(pop, len(pop))
        # selTournamentDCD means Tournament selection based on
dominance (D)
        # followed by crowding distance (CD). This selection requires
the
        # individuals to have a crowding_dist attribute
        offspring = [toolbox.clone(ind) for ind in offspring]

        for ind1, ind2 in zip(offspring[::2], offspring[1::2]):
        #make pairs of all (even,odd) in offspring
            if random.random() <= CXPB:
                toolbox.mate(ind1, ind2,0.5)

            toolbox.mutate(ind1)
            toolbox.mutate(ind2)
            del ind1.fitness.values, ind2.fitness.values

        # Evaluate the individuals with an invalid fitness
        invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
        fitnesses = toolbox.map(toolbox.evaluate, invalid_ind)
        for ind, fit in zip(invalid_ind, fitnesses):
            ind.fitness.values = fit

        record = stats.compile(pop)
        logbook.record(gen=gen, evals=len(invalid_ind), **record)

        hypervals.append(hypervolume(pop,
[max(df['F1']),max(df['F2'])]))

    print("hypervolume: ",hypervals)

    return pop, logbook,hypervals
```

```
if __name__ == "__main__":
    pop, stats,hypervolumes = main()

max(df['F1']),max(df['F2'])


creator.create("FitnessMin", base.Fitness, weights=(-1.0, -1.0))
creator.create("Individual", list, fitness=creator.FitnessMin)

toolbox = base.Toolbox()


def chrom2real(c):
    indasstring=''.join(map(str, c))
    degray=gray_to_bin(indasstring)
    numasint=int(degray, 2) # convert to int from base 2 list
    maxnum      = 2**10
    numinrange=-5+10*numasint/maxnum
    return numinrange

# input: concatenated list of binary variables
# output: tuple of real numbers representing those variables
def separatevariables(v):
    return
chrom2real(v[0:10]),chrom2real(v[10:20]),chrom2real(v[20:30])
def calcFitness(individual):
    x1,x2,x3=separatevariables(individual)
    f1=(((x1-0.6)/1.6)**2+(x2/3.4)**2+(x3-1.3)**2)/2.0
    f2=((x1/1.9-2.3)**2+(x2/3.3-7.1)**2+(x3+4.3)**2)/3.0
    return f1,f2

#Function to carry out the crossover
def crossover(a,b):
    r=random.random()
    if r>0.5:
        return mutation((a+b)/2)
    else:
        return mutation((a-b)/2)

#Function to carry out the mutation operator
def mutation(solution):
    mutation_prob = random.random()
    if mutation_prob <1:
        solution = min_x+(max_x-min_x)*random.random()
    return solution


toolbox.register("attr_bool", random.randint, 0, 1)
toolbox.register("individual", tools.initRepeat, creator.Individual,
    toolbox.attr_bool, 30)
toolbox.register("population", tools.initRepeat, list,
toolbox.individual)

toolbox.register("evaluate", calcFitness)
toolbox.register("mate", tools.cxUniform)
flipProb=1.0/30
toolbox.register("mutate", tools.mutFlipBit, indpb=flipProb)
toolbox.register("select", tools.selNSGA2)

def main(seed=None):
```

```
    h_vals=[]
    random.seed(seed)

    NGEN = 30
    MU = 24
    CXPB = 0.9

    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("avg", numpy.mean, axis=0)
    stats.register("std", numpy.std, axis=0)
    stats.register("min", numpy.min, axis=0)
    stats.register("max", numpy.max, axis=0)

    logbook = tools.Logbook()
    logbook.header = "gen", "evals", "std", "min", "avg", "max"

    pop = toolbox.population(n=MU)
    pop = toolbox.select(pop, len(pop))
    # Evaluate the individuals with an invalid fitness
    invalid_ind = [ind for ind in pop if not ind.fitness.valid]
    fitnesses = toolbox.map(toolbox.evaluate, invalid_ind)
    for ind, fit in zip(invalid_ind, fitnesses):
        ind.fitness.values = fit

    # This is just to assign the crowding distance to the individuals
    # no actual selection is done

    pop = tools.selTournamentDCD(pop, len(pop))


    record = stats.compile(pop)
    logbook.record(gen=0, evals=len(invalid_ind), **record)

    # Begin the generational process

    for gen in range(0, NGEN):
        # Vary the population
        offspring = tools.selTournamentDCD(pop, len(pop))
        # selTournamentDCD means Tournament selection based on
dominance (D)
        # followed by crowding distance (CD). This selection requires
the
        # individuals to have a crowding_dist attribute
        offspring = [toolbox.clone(ind) for ind in offspring]

        for ind1, ind2 in zip(offspring[::2], offspring[1::2]):
        #make pairs of all (even,odd) in offspring
            if random.random() <= CXPB:
                toolbox.mate(ind1, ind2,0.5)

            toolbox.mutate(ind1)
            toolbox.mutate(ind2)
            del ind1.fitness.values, ind2.fitness.values

        # Evaluate the individuals with an invalid fitness
        invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
        fitnesses = toolbox.map(toolbox.evaluate, invalid_ind)
        for ind, fit in zip(invalid_ind, fitnesses):
            ind.fitness.values = fit

        # Select the next generation population
        pop = toolbox.select(pop + offspring, MU)
        best = toolbox.select(pop + offspring, MU)
        all_individuals=toolbox.select(pop + offspring, MU*2)




        record = stats.compile(pop)
        logbook.record(gen=gen, evals=len(invalid_ind), **record)
        h_vals.append(hypervolume(pop,
[max(df['F1']),max(df['F2'])]))


    return h_vals,pop

if __name__ == "__main__":
    h_vals,pop = main()

plt.scatter([i+1 for i in range(30)],h_vals)
plt.xlabel("Generations")
plt.ylabel("Hypervolume")
plt.title("Hypervolume vs Generations")
plt.show()

# f1f2
# plt.scatter()

# for i in
# for i in pop:
f1f2 = numpy.array([ind.fitness.values for ind in pop])
f1f2
maxf1 = 0
maxf2 = 0
for i in f1f2:
    plt.scatter(i[0],i[1],color= "green")
    if i[0] > maxf1:
        maxf1 = i[0]
    elif i[1] > maxf2:
        maxf2 = i[1]

plt.scatter(maxf1,maxf2, color="red",label="nadir point")
plt.title("plot showing Nadir point")
plt.legend()
plt.xlabel("function 1")
plt.ylabel("function 2")
plt.show()
```