COM3013 Coursework

Arthur Butcher
Computer Science Student
University of Surrey
ab02038@surrey.ac.uk

Introduction

In this document I will be answering the questions detailed in the 'CW1_COM3013_2021' pdf file. For each question, I will be giving a brief explanation of what code I used as a base, what I modified to fit the question, and how I created the accompanying graphs.

Question 1

1.1 a)

Fittest Individual = 1.4716970971395036

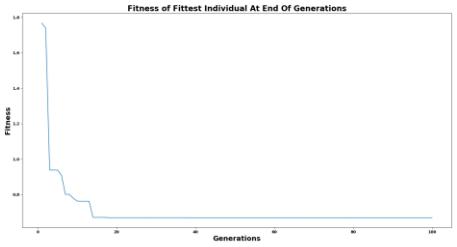
X1 = 0.091552734375

X2 = -0.73272705078125

To achieve these results, I used the code file 'GA1', as given in week 2, as a base. First I changed the eval_fit function to limit the fitness values to be between -5 and 5, and the number of bits per chromosome to 15, as defined in the question. I then changed the fitness function to match the one given in the question, ensuring that the comma at the end of the return statement was retained, as the function must return a tuple.

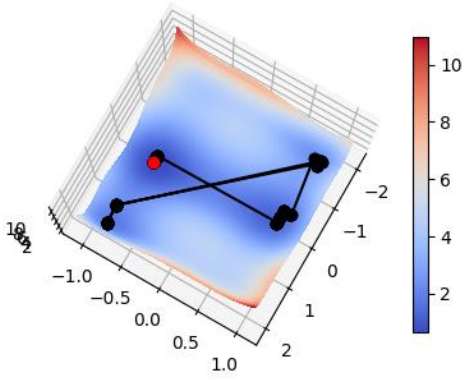
Once these were changed, the code output the aforementioned values.

1.1 b)



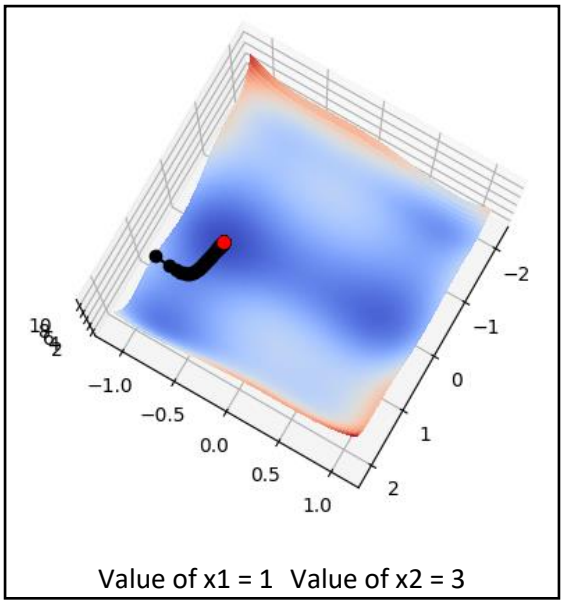
To plot the fitness of the fittest individual for every generation, I created a list called 'individual_best'. I appended the fittest individual for every generation to the list, ensuring the fitness values were converted to correctly represent the actual fitness values, overcoming DEAPs inability to find the minimal value natively. I then plotted this list along the Y axis, with the generations along the X axis.

1.1 c)

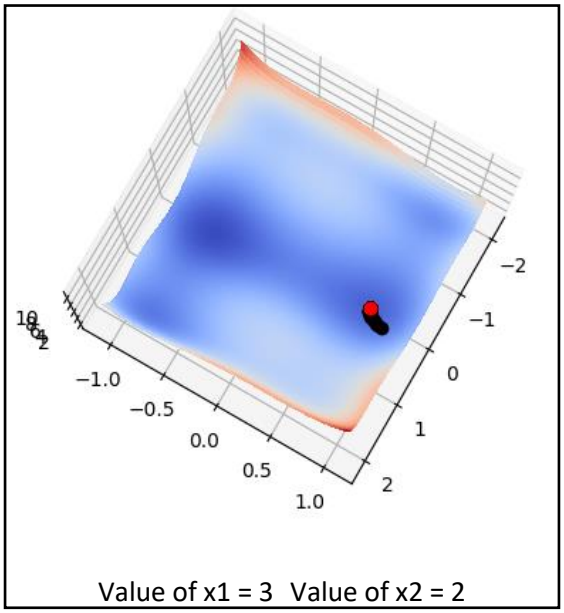


To achieve this graph, I used the code from the 3D graph example provided in week 2. I adjusted the 'xrange' and the 'yrange' values to correlate with the question, and create another copy of the function provided, though this time designed it to take raw x1 and x2 values, as opposed to the individuals. I then used a for loop to cycle through every individual, appending it's x1 and x2 values to the corresponding lists, and appending the resulting function value to another list. Once the for loop was completed, I plotted these points on the 3D graph. I then pulled the final value from each list and plotted that separately, only this time in a red colour, to show that this was the lowest/fittest value.

1.2)



Value of x1 = 1 Value of x2 = 3



Value of x1 = 3 Value of x2 = 2

To plot these graphs, I had to create two separate functions: dx with respect to x1 (dx1), and dx with respect to x2 (dx2). I then created a for loop for 200 steps, with each step creating new x1 and x2 values using a small alpha of 0.01 (to avoid memory overflow errors) and appending these two corresponding lists, whilst again generating function values from these x1 and x2 values and appending these to their own list. I then plotted these lists on the 3D graph. The initial values for x1 and x2 are displayed above each graph.

Question 2

a)

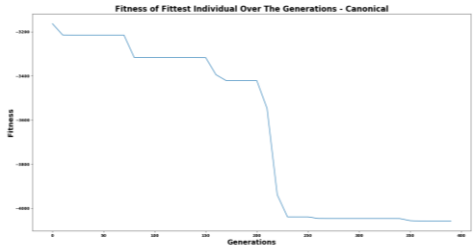
Fitness of individual: -4057.54

Value of x1: 72.74892583776702

To generate these values, I used the sample PSO code given in week 3 as a foundation. The first change I made was to change the posMinInit and posMaxInit values to the corresponding minimum and maximum values defined in the question (-500 and 500).

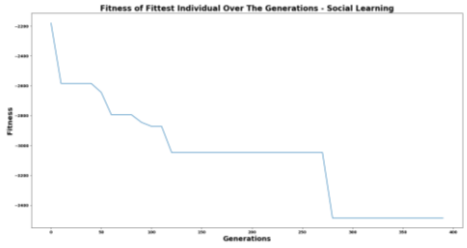
I also made a point of uncommenting the clamping code that, in the example PSO file, was commented out. This resulted in none of the x values exceeding the minimum and maximum values.

b)



To print this graph, I simply used the logbook, which was already being used to print the generation, minimal fitness, maximum fitness etc. I plotted the ‘gen’ values along the x axis, and the ‘min’ values along the y axis.

c)



To generate this graph, I used the sample social PSO code as a base, before changing the evaluation function to match the given fitness equation.

To stop the x values of individuals from exceeding the bounds, I also implemented a very simple if statement in the ‘updateParticle’ function that checked if the particles position exceeds the bounds, and

generating new randomised position values between the upper and lower bounds if they did.

Question 3

3.1)

Individual	x1	x2	x3	f1	f2
1	3.078125	-3.6171875	0.46875	6.051844221	30.12654
2	2.5859375	1.1875	2.125	4.029274576	29.19732
3	2	3.2578125	-0.2890625	3.8917533	18.33656
4	3.4296875	2.3671875	-2.359375	13.31958568	14.91648
5	3.6953125	-1.796875	-0.78125	9.713946115	23.64873
6	-0.7265625	1.2265625	0.1796875	0.979021975	24.1777
7	1.2734375	2.953125	-2.5078125	8.50671929	14.79051
8	-2.1484375	1.6171875	-2.0859375	8.349627613	20.12117
9	-2.3046875	0.359375	-1.484375	6.763677166	23.04811
10	1.515625	-3	-0.4140625	3.104545378	27.16765
11	-0.390625	-1.3359375	-1.8984375	5.274979611	22.78932
12	0.015625	-0.3671875	2.140625	0.359289253	32.9121
13	-2.234375	3.3671875	-3.25	13.55021335	16.71567
14	-3.0859375	1.1640625	-2.0234375	10.74779119	22.03578
15	1.90625	2.4296875	2.5	2.946793303	29.47284
16	-3.0859375	0.7421875	-1.4296875	8.915986005	23.63493
17	2.6484375	-0.671875	0.40625	4.224377991	25.43744
18	2.9765625	-1.859375	0.625	5.184165211	27.84063
19	2.7265625	0.8203125	1.7890625	4.181977024	28.2556
20	-0.4453125	-2.3359375	1.984375	0.577783238	35.62637
21	-3.84375	-0.0390625	2.265625	8.481914448	37.45807
22	0.3046875	-2.296875	1.8984375	0.45761454	34.59222
23	-2.640625	-4	1.59375	4.51822663	39.14743
24	0.1796875	-0.9140625	1.40625	0.059299661	30.61505

To generate these values, I used the sample solution to Q3 code from week 4 as my base.

I changed the calcFitness function to divide up the individual correctly into three x values, each with 10 bits. I then used the chrom2real function to translate these values from greycode to decimal. Once the x values were in decimal form, they could be fed into the fitness functions. The resulting individuals decoded x values and their fitness values are shown above.

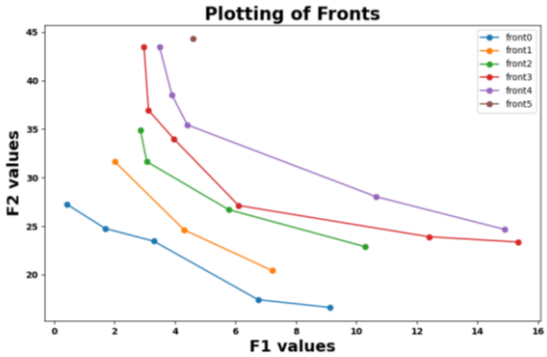
3.2)

Individual	f1	f2	Front
1	1.258522837	38.1652883	1
2	1.665494712	28.9146201	1
3	2.460227857	22.5877714	1
4	3.658259271	22.327974	1
5	4.718045732	19.0791189	1
6	7.844815971	16.9864968	1
7	14.61812178	16.0694136	1
8	2.329809431	29.7391858	2
9	3.981719408	24.0154483	2
10	5.051796887	23.4775971	2
11	10.40743922	23.0489535	2
12	13.44286294	17.7898338	2
13	2.626781161	36.1628384	3
14	2.674674474	33.6017584	3
15	4.043488901	32.3326802	3
16	5.424690852	27.2787757	3
17	7.614948241	24.2277198	3
18	15.37739903	20.5861641	3
19	15.95865425	17.9981926	3
20	3.333035352	41.8174417	4
21	5.007910037	39.1939167	4
22	7.862945801	36.6571916	4
23	5.039127961	46.2378713	5
24	6.029661608	41.4762805	5

I implemented the Efficient Non-Dominated sorting algorithm from the week 4 lectures, before feeding it with the population pre-sorted by their f1 values (minus the 1st indv), and a 'fronts' array containing the 1st individual ready to be populated with the rest.

I also made a point of using the 'isDominated' function that is part of DEAPs tools library.

Once the fronts were returned, I cycled through the population, finding the front sub-list they belonged to, and adding that fronts value as an attribute to that individual.



I pulled each individuals f1 and f2 values and appended them to lists for each front. I then plotted each front, allowing pyplot to assign colours automatically.

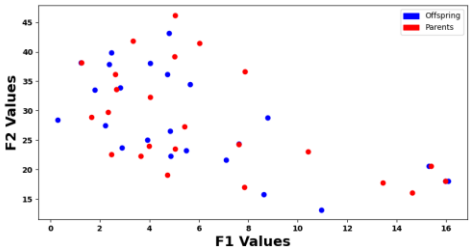
3.3)

Individual	f1	f2	Front	Crowding Distance
1	1.258522837	38.1652883	1	inf
2	1.665494712	28.9146201	1	0.3974736481898589
3	2.460227857	22.5877714	1	0.22362871468020348
4	3.658259271	22.327974	1	0.16389780837678072
5	4.718045732	19.0791189	1	0.2775576781491221
6	7.844815971	16.9864968	1	0.43862854343336033
7	14.61812178	16.0694136	1	inf
8	2.329809431	29.7391858	2	inf
9	3.981719408	24.0154483	2	0.38447339049203544
10	5.051796887	23.4775971	2	0.329548156985502
11	10.40743922	23.0489535	2	0.6155266095079646
12	13.44286294	17.7898338	2	inf
13	2.626781161	36.1628384	3	inf
14	2.674674474	33.6017584	3	0.15856128519669632
15	4.043488901	32.3326802	3	0.27718331691860953
16	5.424690852	27.2787757	3	0.3570415172087895
17	7.614948241	24.2277198	3	0.5574882476258343
18	15.37739903	20.5861641	3	0.48439719759451416
19	15.95865425	17.9981926	3	inf
20	3.333035352	41.8174417	4	inf
21	5.007910037	39.1939167	4	1.0
22	7.862945801	36.6571916	4	inf
23	5.039127961	46.2378713	5	inf
24	6.029661608	41.4762805	5	inf

To calculate the crowing distances for each individual, I used DEAPs 'assignCrowdingDist'

function, which assigned crowding distances to all the individuals in the fronts assigned to it.

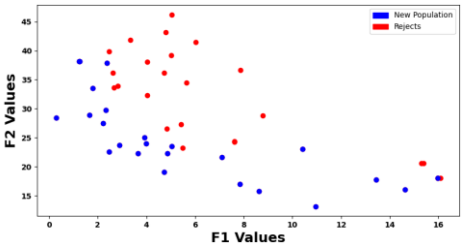
3.4)



To generate the offspring, I used DEAPs 'selTournamentDCD' tool. Once these offspring had been generated, I then cycled through the offspring, applying Uniform crossover with a flip probability of 0.9 as requested.

I then plotted the parent population fitness values onto the graph, marking them in red, before plotting the mutated offspring fitness values in blue.

3.5)



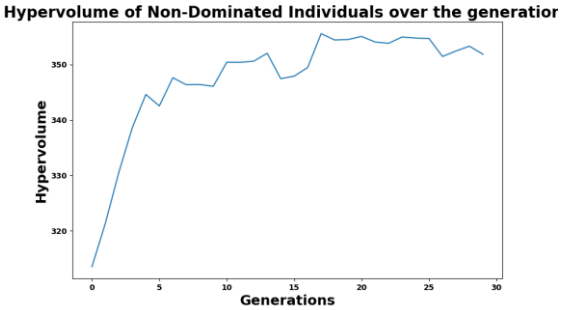
To create this graph, I combined the offspring and the parents into one combined list, before plotting the fitness values of each population all in red.

I then sorted the combined population by the f1 values and fed it into my EffNonDom algorithm I created in 3.2. I then used these resulting fronts to assign the crowding distance to each individual, before sorting each front by that crowding distance, largest to smallest, ensuring that only the most unique indvs were picked for the new population.

I then iterated through these fronts, starting from 0, and added each indv to a new list until that new list reached the required new population (24).

Finally, I plotted these values over the combined population, marking these selected indvs in blue.

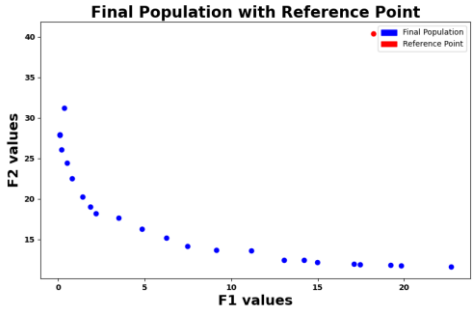
3.6)



I first created an array to store the hypervolume values and appended the hypervolume of the first population it. To get a reference point, I sorted the initial population first by f2, and then by f1, pulling the worst values for each and using them in the hypervolume function.

I then looped the process in 3.4 and 3.5 30 times, each time selecting the new best 24 individuals and appending the resulting populations hypervolume to the array.

I then simply plotted the resulting hypervolume array on the graph, with its value son the y axis and the generations on the x axis.



Finally, I plotted the final populations fitness values on the graph in blue, and plotted the worst f1 and f2 values that I made for the hypervolume function in red.

Appendix

Code for Question 1

```
# Question 1

import random
import matplotlib.pyplot as plt
import numpy as np
import matplotlib
from mpl_toolkits.mplot3d.axes3d import Axes3D
from sympy.combinatorics.graycode import gray_to_bin
from deap import creator, base, tools

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

popSize = 50
dimension = 2
numOfBits = 15
iterations = 100
dspInterval = 10
nElitists = 1
omega = 5
crossPoints = 2
crossProb = 0.6
flipProb = 1. / (dimension * numOfBits)
mutateprob = .1
maxnum = 2 ** numOfBits

individual_best = []
overall_best = []

toolbox = base.Toolbox()

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

toolbox.register("individual", tools.initRepeat, creator.Individual,
                 toolbox.attr_bool, numOfBits * dimension)

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


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


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


# 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_fit(individual):
    sep = separate_variables(individual)
    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)))
    return 1.0 / (0.01 + f),
```

```
def convert_fitness(fitness):
   return (1 / fitness) - 0.01


def f(x1, x2):
   return (2 + (4.1 * (x1 ** 2)) - (2.1 * (x1 ** 4)) + ((1 / 3) * (x1 ** 6)) + (x1 * x2)
         - (4 * ((x2 - 0.05) ** 2)) + (4 * (x2 ** 4)))


def dx1(x1, x2):
   return 2 * x1 ** 5 - 8.4 * x1 ** 3 + 8.2 * x1 + x2


def dx2(x1, x2):
   return x1 + 16 * x2 ** 3 - 8 * x2 + 0.4


toolbox.register("evaluate", eval_fit)

toolbox.register("mate", tools.cxTwoPoint)

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

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

hall_of_fame = tools.HallOfFame(1)

stats = tools.Statistics()

stats.register('Min', np.min)
stats.register('Max', np.max)
stats.register('Avg', np.mean)
stats.register('Std', np.std)

logbook = tools.Logbook()


def main():
   pop = toolbox.population(n=popSize)

   fitnesses = list(map(toolbox.evaluate, pop))

   for ind, fit in zip(pop, fitnesses):
      ind.fitness.values = fit

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

   g = 0

   hall_of_fame.clear()

   while g < iterations:

      g = g + 1
      print("-- Generation %i --" % g)

      offspring = tools.selBest(pop, nElitists) + toolbox.select(pop, len(pop) - nElitists)

      offspring = list(map(toolbox.clone, offspring))

      for child1, child2 in zip(offspring[::2], offspring[1::2]):
```

```
        if random.random() < crossProb:
            toolbox.mate(child1, child2)

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

    for mutant in offspring:

        if random.random() < mutateprob:
            toolbox.mutate(mutant)
            del mutant.fitness.values

    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

    this_gen_fitness = []
    for ind in offspring:
        this_gen_fitness.append(ind.fitness.values[0])

    hall_of_fame.update(offspring)

    stats_of_this_gen = stats.compile(this_gen_fitness)

    stats_of_this_gen['Generation'] = g

    logbook.append(stats_of_this_gen)

    pop[:] = offspring

    individual_best.append(convert_fitness(tools.selBest(pop, 1)[0].fitness.values[0]))
    overall_best.append(tools.selBest(pop, 1)[0])

    if g % dspInterval == 0:
        fits = [ind.fitness.values[0] for ind in pop]

        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" % (separate_variables(best_ind)))

# plt.figure(figsize=(20, 10))
#
# plt.plot(logbook.select('Generation'), logbook.select('Min'))
#
# plt.title("Fitness of Fittest Individual Over The Generations", fontsize=20, fontweight='bold')
# plt.xlabel("Generations", fontsize=18, fontweight='bold')
# plt.ylabel("Fitness", fontsize=18, fontweight='bold')
# plt.xticks(fontweight='bold')
```

```
# plt.yticks(fontweight='bold')

plt.figure(figsize=(20, 10))
plt.plot(logbook.select('Generation'), individual_best)
plt.title("Fitness of Fittest Individual At End Of Generations", fontsize=20, fontweight='bold')
plt.xlabel("Generations", fontsize=18, fontweight='bold')
plt.ylabel("Fitness", fontsize=18, fontweight='bold')
plt.xticks(fontweight='bold')
plt.yticks(fontweight='bold')

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)

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

ax = fig.add_subplot(1, 2, 2, projection='3d')
p = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=matplotlib.cm.coolwarm, linewidth=0, antialiased=False,
                    zorder=0)

fig.colorbar(p, shrink=0.5)

x1list = []
x2list = []
zlist = []

for x in range(len(overall_best)):
    decision_variables = separate_variables(overall_best[x])
    x1 = decision_variables[0]
    x2 = decision_variables[1]
    x1list.append(x1)
    x2list.append(x2)
    z = f(x1, x2)
    zlist.append(z)
ax.plot3D(x1list, x2list, zlist, color="k", marker="o", zorder=10)

decision_variables = separate_variables(overall_best[len(overall_best) - 1])
x1 = decision_variables[0]
x2 = decision_variables[1]
ax.plot3D([x1], [x2], [f(x1, x2)], color="#FF0000", marker="o", zorder=10)
ax.view_init(80, 30)

plt.show()

# 1.2 code

x1 = 1
x2 = 3

xlist = []
ylist = []
zlist = []
alpha = 0.01

for step in range(0, 200):
    newx1 = x1 - alpha * (dx1(x1, x2))
    x2 = x2 - alpha * (dx2(x1, x2))
    x1 = newx1
    z = f(x1, x2)
    print(x1, x2, z)
    xlist.append(x1)
    ylist.append(x2)
```

```
        zlist.append(z)

    x = xlist[-1]
    y = ylist[-1]
    z = zlist[-1]

    fig = plt.figure(figsize=(10, 10))

    # surface_plot with color grading and color bar
    ax = fig.add_subplot(1, 2, 1, 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(x, y, z, color='red', marker='o', zorder=10)
    ax.view_init(80, 30)

    plt.show()

    x1 = 3
    x2 = 2

    xlist = []
    ylist = []
    zlist = []
    alpha = 0.01

    for step in range(0, 200):
        newx1 = x1 - alpha * (dx1(x1, x2))
        x2 = x2 - alpha * (dx2(x1, x2))
        x1 = newx1
        z = f(x1, x2)
        print(x1, x2, z)
        xlist.append(x1)
        ylist.append(x2)
        zlist.append(z)

    x = xlist[-1]
    y = ylist[-1]
    z = zlist[-1]

    fig = plt.figure(figsize=(10, 10))

    # surface_plot with color grading and color bar
    ax = fig.add_subplot(1, 2, 1, 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(x, y, z, color='red', marker='o', zorder=10)
    ax.view_init(80, 30)

    plt.show()


if __name__ == '__main__':
    main()
```

Code for Question 2

Canonical

```
# Question 2 a and b
import operator
import random
import numpy
import math
from math import sin, sqrt
import matplotlib.pyplot as plt
from deap import base
from deap import creator
from deap import tools

posMinInit = - 500
posMaxInit = + 500
VMaxInit = 1.5
VMinInit = 0.5
populationSize = 50
dimension = 20
interval = 10
iterations = 400
maxnum = 2 ** 2

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

creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
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
    part.smax = smax
    return part

def update_particle(part, best, weight):
    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))]
    v_r2 = [c2 * x for x in map(operator.mul, r2, map(operator.sub, best, part))]

    part.speed = [0.7 * x for x in map(operator.add, v_r0, map(operator.add, v_r1, v_r2))]

    for i, speed in enumerate(part.speed):
        if abs(speed) < part.smin:
            part.speed[i] = math.copysign(part.smin, speed)
        elif abs(speed) > part.smax:
            part.speed[i] = math.copysign(part.smax, speed)

    part[:] = list(map(operator.add, part, part.speed))

def eval_indv(individual):
    return -sum(x * sin(sqrt(abs(x))) for x in individual),
```

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


def main():
    pop = toolbox.population(n=populationSize)
    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

    for g in range(iterations):
        w = wmax - (wmax - wmin) * g / iterations

        for part in pop:
            part.fitness.values = toolbox.evaluate(part)

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

            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)

        if g % interval == 0:
            logbook.record(gen=g, evals=len(pop), **stats.compile(pop))
            print(logbook.stream)

    print('best particle position is ', best)

    plt.figure(figsize=(20, 10))

    plt.plot(logbook.select('gen'), logbook.select('min'))

    plt.title("Fitness of Fittest Individual Over The Generations - Canonical", fontsize=20, fontweight='bold')
    plt.xlabel("Generations", fontsize=18, fontweight='bold')
    plt.ylabel("Fitness", fontsize=18, fontweight='bold')
    plt.xticks(fontweight='bold')
    plt.yticks(fontweight='bold')

    plt.show()

    return pop, logbook, best


if __name__ == "__main__":
    main()
```

Social

```
# Question 2 c

import random
import numpy
import matplotlib.pyplot as plt
import math
from math import sin, sqrt
from deap import base
from deap import creator
from deap import tools

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

epsilon = dimension / 100.0 * 0.01


def getcenter(pop):
    center = list()
    for j in range(dimension):
        centerj = 0
        for i in pop:
            centerj += i[j]
        centerj /= populationSize
        center.append(centerj)
    return center


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

    demonstrator = random.choice(list(pop[0:i]))

    for j in range(dimension):
        part.speed[j] = r1 * part.speed[j] + r2 * (demonstrator[j] - part[j]) + r3 * epsilon * (center[j] - part[j])
        part[j] = part[j] + part.speed[j]

def eval_indv(individual):
    return -sum(x * sin(sqrt(abs(x))) for x in individual),

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_indv)


def main():
    pop = toolbox.population(n=populationSize)
    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)

    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

    for g in range(iterations):

        for part in pop:
            part.fitness.values = toolbox.evaluate(part)

        pop.sort(key=lambda x: x.fitness, reverse=True)

        center = getcenter(pop)

        for i in reversed(range(len(pop) - 1)):

            if random.uniform(0, 1) < prob[i + 1]:
                toolbox.update(pop[i + 1], pop, center, i + 1)

        if g % interval == 0:
            logbook.record(gen=g, evals=len(pop), **stats.compile(pop))
            print(logbook.stream)

    plt.figure(figsize=(20, 10))

    plt.plot(logbook.select('gen'), logbook.select('min'))

    plt.title("Fitness of Fittest Individual Over The Generations - Social Learning", fontsize=20, fontweight='bold')
    plt.xlabel("Generations", fontsize=18, fontweight='bold')
    plt.ylabel("Fitness", fontsize=18, fontweight='bold')
    plt.xticks(fontweight='bold')
    plt.yticks(fontweight='bold')

    plt.show()

    return pop, logbook


if __name__ == "__main__":
    main()
```

Code for Question 3

```
# question 3 1 - 6

import random

import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import numpy
from deap import base
from deap import creator
from deap import tools
from deap.benchmarks.tools import hypervolume
from deap.tools.emo import assignCrowdingDist
from deap.tools.emo import isDominated
from sympy.combinatorics.graycode import bin_to_gray, gray_to_bin

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)
    numinrange = -4 + 8 * numasint / (2 ** 10)
    return numinrange


def calcFitness(individual):
    x1_bits = individual[0:10]
    x2_bits = individual[10:20]
    x3_bits = individual[20:30]
    x1 = ("".join(str(i) for i in x1_bits))
    x2 = ("".join(str(i) for i in x2_bits))
    x3 = ("".join(str(i) for i in x3_bits))
    x1_gray = bin_to_gray(x1)
    x2_gray = bin_to_gray(x2)
    x3_gray = bin_to_gray(x3)
    x1 = chrom2real(x1_gray)
    x2 = chrom2real(x2_gray)
    x3 = chrom2real(x3_gray)
    f1 = (((x1 / 0.6) / 1.6) ** 2 + (x2 / 3.4) ** 2 + (x3 - 1.3) ** 2.0) / 2.0
    f2 = ((x1 / 1.9 - 2.3) ** 2 + (x2 / 3.3 - 7.1) ** 2 + (x3 + 4.3) ** 2.0) / 3.0
    return f1, f2


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


def eval_fit(individual):
    seperated_indv = separatevariables(individual)
    f1 = (((seperated_indv[0] / 0.6) / 1.6) ** 2 + (seperated_indv[1] / 3.4) ** 2 + (
        seperated_indv[2] - 1.3) ** 2.0) / 2.0
    f2 = ((seperated_indv[0] / 1.9 - 2.3) ** 2 + (seperated_indv[1] / 3.3 - 7.1) ** 2 + (
        seperated_indv[2] + 4.3) ** 2.0) / 3.0
    return f1, f2


def algorithm(indivdual, fronts):
```

```
    x = len(fronts)
    k = 1

    while True:
        dominated = False
        for indv in reversed(fronts[k - 1]):
            if isDominated(indv.fitness.values, indivdual.fitness.values):
                dominated = True
                break

        if not dominated:
            fronts[k - 1].append(indivdual)
            return k
            break

        else:
            k = k + 1
            if k > x:
                fronts.append([indivdual])
                return x + 1
                break


def efficient_non_dominated_sort(pop_sorted_by_f1):
    front = [[]]

    q = pop_sorted_by_f1[0]
    front[0].append(q)

    for ind in pop_sorted_by_f1[1:]:
        algorithm(ind, front)

    return front


def take_first(elem):
    return elem[0]


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, indpb=0.9)
flipProb = 1.0 / 30
toolbox.register("mutate", tools.mutFlipBit, indpb=flipProb)
toolbox.register("select", tools.selNSGA2)


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

    total_population = 24

    pop = toolbox.population(n=total_population)

    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
```

```
pop = toolbox.select(pop, len(pop))

# code for 3.1

for individual in pop:
    x_values = separatevariables(individual)
    fitness_values = eval_fit(individual)
    x1 = x_values[0]
    x2 = x_values[1]
    x3 = x_values[2]
    f1 = fitness_values[0]
    f2 = fitness_values[1]
    print(x1, x2, x3, f1, f2)

# code for 3.2

pop.sort(key=lambda x: x.fitness.values[1])

worstF2 = pop[-1].fitness.values[1]

pop.sort(key=lambda x: x.fitness.values[0])

worstF1 = pop[-1].fitness.values[0]

fronts = efficient_non_dominated_sort(pop)

for ind in pop:
    i = 0
    needs_home = True
    while needs_home:
        if ind in fronts[i]:
            needs_home = False
            ind.front = i + 1
        else:
            i += 1

pop.sort(key=lambda x: x.front)

for individual in pop:
    print(individual.fitness.values[0], individual.fitness.values[1], individual.front)

worst_indv = pop[-1]

plt.figure(figsize=(10, 6))

i = 0
front_f1 = []
front_f2 = []
while i != len(fronts):
    for ind in fronts[i]:
        front_f1.append(ind.fitness.values[0])
        front_f2.append(ind.fitness.values[1])
    plt.plot(front_f1, front_f2, marker='o', label='front' + str(i))
    i += 1
    front_f1.clear()
    front_f2.clear()

plt.title("Plotting of Fronts", fontsize=20, fontweight='bold')
plt.xlabel("F1 values", fontsize=18, fontweight='bold')
plt.ylabel("F2 values", fontsize=18, fontweight='bold')
plt.legend(loc='best')
plt.xticks(fontweight='bold')
```

```
plt.yticks(fontweight='bold')
plt.show()

# code for 3.3

for front in fronts:
    assignCrowdingDist(front)

for individual in pop:
    print(individual.fitness.values[0], individual.fitness.values[1], individual.front,
        individual.fitness.crowding_dist)

# code for 3.4

offspring = tools.selTournamentDCD(pop, len(pop))
offspring = [toolbox.clone(ind) for ind in offspring]

for ind1, ind2 in zip(offspring[::2], offspring[1::2]):
    toolbox.mate(ind1, ind2)

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

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

f1_off_values = []
f2_off_values = []
for ind in offspring:
    f1_off_values.append(ind.fitness.values[0])
    f2_off_values.append(ind.fitness.values[1])

f1_pop_values = []
f2_pop_values = []
for ind in pop:
    f1_pop_values.append(ind.fitness.values[0])
    f2_pop_values.append(ind.fitness.values[1])

plt.figure(figsize=(10, 6))
plt.xlabel("F1 Values", fontsize=18, fontweight='bold')
plt.ylabel("F2 Values", fontsize=18, fontweight='bold')
plt.xticks(fontweight='bold')
plt.yticks(fontweight='bold')

f1_blue = mpatches.Patch(color='blue', label='Offspring')
f2_red = mpatches.Patch(color='red', label='Parents')

plt.legend(handles=[f1_blue, f2_red])

plt.scatter(f1_off_values, f2_off_values, color='blue')
plt.scatter(f1_pop_values, f2_pop_values, color='red')

plt.show()

# code for 3.5

combined_individuals = pop + offspring
combined_individuals.sort(key=lambda x: x.fitness.values[0])
fronts_combined = efficient_non_dominated_sort(combined_individuals)
```

```
for front in fronts_combined:
    assignCrowdingDist(front)
    front.sort(key=lambda x: x.fitness.crowding_dist, reverse=True)

i = 24
front = 0
indv = 0
new_pop = []
while i != 0:
    if indv == len(fronts_combined[front]):
        front += 1
        indv = 0
    new_pop.append(fronts_combined[front][indv])
    indv += 1
    i -= 1

f1_new_values = []
f2_new_values = []
for ind in new_pop:
    f1_new_values.append(ind.fitness.values[0])
    f2_new_values.append(ind.fitness.values[1])

f1_combined_values = []
f2_combined_values = []
for ind in combined_individuals:
    f1_combined_values.append(ind.fitness.values[0])
    f2_combined_values.append(ind.fitness.values[1])

plt.figure(figsize=(10, 6))
plt.xlabel("F1 Values", fontsize=18, fontweight='bold')
plt.ylabel("F2 Values", fontsize=18, fontweight='bold')
plt.xticks(fontweight='bold')
plt.yticks(fontweight='bold')

f1_blue = mpatches.Patch(color='blue', label='New Population')
f2_red = mpatches.Patch(color='red', label='Rejects')

plt.legend(handles=[f1_blue, f2_red])

plt.scatter(f1_combined_values, f2_combined_values, color='red')
plt.scatter(f1_new_values, f2_new_values, color='blue')

plt.show()

# code for 3.6

pop = new_pop

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"

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

indvs_in_front_1 = []
for x in pop:
```

```
    if x.front == 1:
        indvs_in_front_1.append(x)

hypervol = [hypervolume(indvs_in_front_1, [worstF1, worstF2])]

for gen in range(1, 30):
    offspring = tools.selTournamentDCD(pop, len(pop))
    offspring = [toolbox.clone(ind) for ind in offspring]

    for ind1, ind2 in zip(offspring[::2], offspring[1::2]):
        toolbox.mate(ind1, ind2)

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

    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

    combined_individuals = pop + offspring
    combined_individuals.sort(key=lambda x: x.fitness.values[0])
    fronts_combined = efficient_non_dominated_sort(combined_individuals)

    for front in fronts_combined:
        assignCrowdingDist(front)
        front.sort(key=lambda x: x.fitness.crowding_dist, reverse=True)

    i = 24
    front = 0
    indv = 0
    new_pop = []
    while i != 0:
        if indv == len(fronts_combined[front]):
            front += 1
            indv = 0
        new_pop.append(fronts_combined[front][indv])
        indv += 1
        i -= 1

    pop = new_pop

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

    indvs_in_front_1 = []
    for x in pop:
        if x.front == 1:
            indvs_in_front_1.append(x)

    hypervol.append(hypervolume(indvs_in_front_1, [worstF1, worstF2]))
    print(logbook.stream)

plt.figure(figsize=(10, 6))
plt.plot(logbook.select('gen'), hypervol)
plt.title("Hypervolume of Non-Dominated Individuals over the generations", fontsize=20, fontweight='bold')
plt.xlabel("Generations", fontsize=18, fontweight='bold')
plt.ylabel("Hypervolume", fontsize=18, fontweight='bold')
plt.xticks(fontweight='bold')
plt.yticks(fontweight='bold')
plt.show()
```

```
    f1_values = []
    f2_values = []
    for ind in pop:
        f1_values.append(ind.fitness.values[0])
        f2_values.append(ind.fitness.values[1])

    plt.figure(figsize=(10, 6))
    plt.scatter(f1_values, f2_values, color='blue')
    plt.scatter(worstF1, worstF2, color="red")
    f1_blue = mpatches.Patch(color='blue', label='Final Population')
    f2_red = mpatches.Patch(color='red', label='Reference Point')
    plt.legend(handles=[f1_blue, f2_red])
    plt.title("Final Population with Reference Point", fontsize=20, fontweight='bold')
    plt.xlabel("F1 values", fontsize=18, fontweight='bold')
    plt.ylabel("F2 values", fontsize=18, fontweight='bold')
    plt.xticks(fontweight='bold')
    plt.yticks(fontweight='bold')
    plt.show()


if __name__ == "__main__":
    main()
```