Computational Intelligence (COM3013 Coursework)

Ashwin Baiju
University Student Number: 6707489

Department of Computer Science: MSc Data Science
University email: ab03123@surrey.ac.uk

Other email: ashwinsbaiju@gmail.com

***Abstract*—This document contains detailed results of the questions and its supporting plots and code outputs. The code that was used to get these outputs has been appended at the end of this notebook**

I. GENETIC ALGORITHM

For the following function,

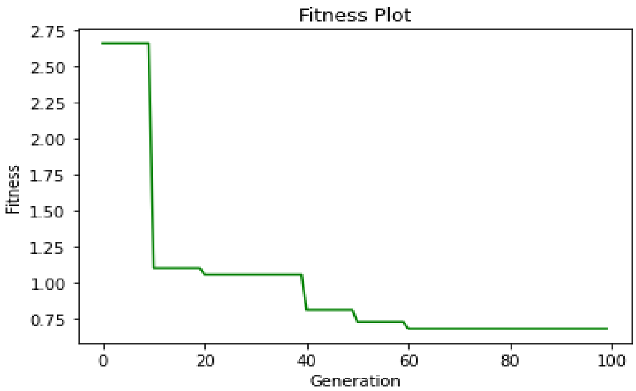
$$f(x_1, x_2) = 2 + 4.1\, x_1^2 - 2.1\, x_1^4 + {}^{1}\, x_1^6 + x_1 x_2 - 4\,(x_2 - 0.05)^2 + 4\, x_2^4$$

1.1 Use a genetic algorithm to find minimum of the function.

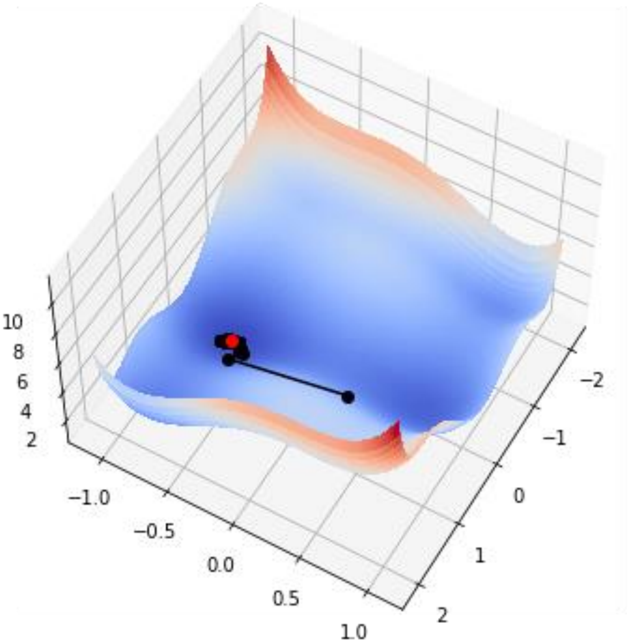
a) The fitness of the fittest individual after 100 generations and its decoded values for x1 and x2.

```
-- End of (successful) evolution --
Best individual is [1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1,
0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1], 0.6795858715108456
Decoded x1, x2 is 0.0937101524323225, -0.7403494324535131
```

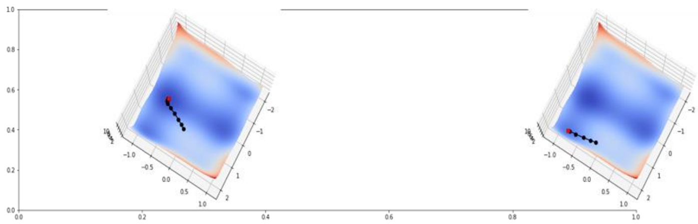
b) A plot of the fitness of the fittest individual across the generations



c) A 3D surface plot of the function f across the range -2.1 < x1 < 2.1 and -1.1 < x2 < 1.1, and a series of the fittest individual in each generation with the final fittest individual in red



1.2 Implement gradient descent for the same function f. Show a 3D surface plot of the function, with the series of improving solutions found during the descent from a suitable starting point. Show a second 3D plot to illustrate that a different starting point can lead to a different final solution.



II. PARTICLE SWARM OPTIMISATION

2. Find the minimum of the following function for 20 dimensions (n=20), using particle swarm optimisation

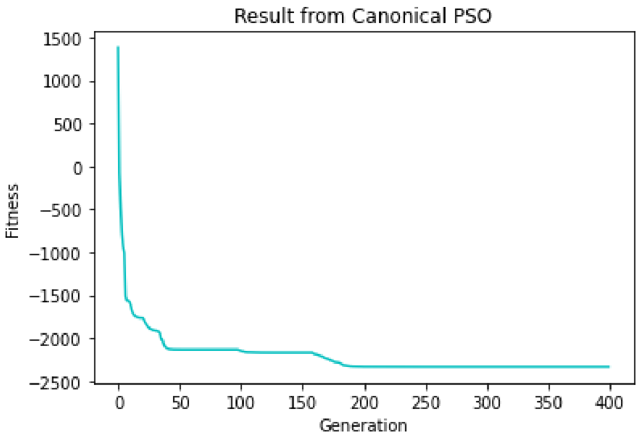
$$f(X) = -\sum_{i=1}^{n}\left[x_i \sin\left(\sqrt{|x_i|}\right)\right]$$

$-500 \le x_i \le 500\,, i = 1, 2, \cdots, n$

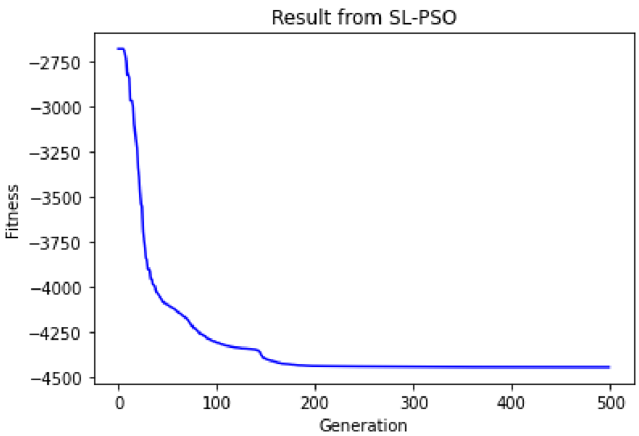
a) The fitness of the fittest individual after 400 generations, and its decoded value for x_i, using canoncial PSO

```
Best fitness is  (-2332.0787350361406,)
best particle position is  [-118.19531593928878, -99.85880039416286, -50.72753697526448, -298.683282393932, 43.2062
12541736434, 69.22003421175647, 47.224225394877806, -341.6863531502434, -294.3439103065735, -8.54295621696627, 446.
87920136938294, 241.45228315533612, -104.73703411720557, -173.37063635036708, -9.934096833954307, -300.097031047748
15, -306.43034638924473, 408.95554813325305, 71.25806516794238, -243.38783538612947]
```

b) A plot of the fitness of the global best individual across the generations, using the canonical PSO



c) A plot of the fitness of the global best individual across the generations, using the social learning PSO (SL-PSO)



III. MULTI-OBJECTIVE OPTIMISATION

3. This task is to solve the following multi-objective optimization problem using the elitist non-dominated sorting genetic algorithm (NSGA-II)

$$\min\{f_1, f_2\}$$

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

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

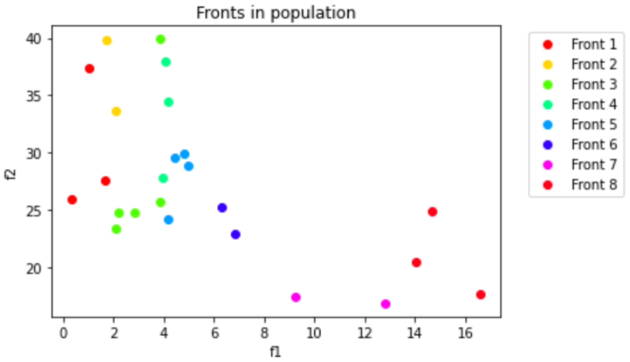
$$-4.0 \le x_1, x_2, x_3 \le 4.0$$

3.1 Encode the decision variables using Gray coding, each using 10 bits. Set the population size to 24 and randomly generate an initial population. List the 24 initial solutions in a table.

	x1	x2	x3	(f1, f2)
0	-1.818182	-0.394917	-1.153470	(4.158613644135455, 24.210651490653845)
1	0.035191	1.223851	-2.967742	(9.233901453021803, 17.42044222622375)
2	-2.162268	-3.632454	-1.036168	(4.78980512764095, 29.908294002632477)
3	-1.145650	-3.296188	2.443793	(1.71923867253507, 39.83242785387602)
4	3.710655	1.442815	3.382209	(4.147714925775057, 34.50981473978322)
5	2.310850	0.324536	-3.640274	(12.779389139585184, 16.877655191109163)
6	-2.647116	-1.333333	-0.598240	(3.937880931239667, 27.884504488048133)
7	0.739003	-3.601173	-2.091887	(6.3171401661034565, 25.208231989612035)
8	-2.209189	2.709677	0.488759	(2.1879486970236357, 24.782372782976626)
9	0.246334	-2.303030	2.529814	(1.0100606861699217, 37.387946828593094)
10	2.091887	-0.887586	-0.512219	(2.1108555715369155, 23.362184287870548)
11	-2.913001	3.038123	-3.945259	(16.56598689491135, 17.667797575099826)
12	2.084066	-2.733138	-2.193548	(6.855705544464888, 22.91379078159592)
13	-1.747801	-3.726295	-1.051808	(4.442670845158467, 29.54596074644299)
14	-1.857283	-1.137830	-0.989247	(3.8556671205275626, 25.709401089673776)
15	2.084066	-2.107527	-3.874878	(14.011960569624165, 20.504950819544945)
16	3.460411	-2.717498	1.865103	(2.0771187842992247, 33.673091535253405)
17	-2.154448	0.832845	-0.332356	(2.844126507384325, 24.80798154701229)
18	-1.059629	-3.491691	-3.913978	(14.658079764518426, 24.956620270257876)
19	1.755621	1.317693	1.262952	(0.33661828910648445, 25.91305004470134)
20	3.045943	0.215054	3.702835	(4.057291998996719, 38.00661977337926)
21	3.499511	0.347996	1.505376	(1.6683521657901055, 27.611993474178792)
22	2.694037	-0.950147	3.726295	(3.8389454068392683, 39.92696650570571)
23	-2.647116	-2.584555	-0.981427	(4.950708090135049, 28.93253749962984)

3.2. Sort the solutions in the initial generation using the efficient non-dominated sorting. List the solutions together with their front number in a table in the following format and sort them according to their front number in an ascending order. Print out the sorted individuals in objective values and front number. Also show a 2D plot of the solutions in objective space, indicating which belong on the same fronts.

	x1	x2	x3	(f1, f2)	Front number
0	3.499511	0.347996	1.505376	(1.6684, 27.612)	1
1	1.755621	1.317693	1.262952	(0.3366, 25.9131)	1
2	0.246334	-2.303030	2.529814	(1.0101, 37.3879)	1
3	-1.145650	-3.296188	2.443793	(1.7192, 39.8324)	2
4	3.460411	-2.717498	1.865103	(2.0771, 33.6731)	2
5	-2.154448	0.832845	-0.332356	(2.8441, 24.808)	3
6	-2.209189	2.709677	0.488759	(2.1879, 24.7824)	3
7	2.091887	-0.887586	-0.512219	(2.1109, 23.3622)	3
8	2.694037	-0.950147	3.726295	(3.8389, 39.927)	3
9	-1.857283	-1.137830	-0.989247	(3.8557, 25.7094)	3
10	3.045943	0.215054	3.702835	(4.0573, 38.0066)	4
11	3.710655	1.442815	3.382209	(4.1477, 34.5098)	4
12	-2.647116	-1.333333	-0.598240	(3.9379, 27.8845)	4
13	-1.818182	-0.394917	-1.153470	(4.1586, 24.2107)	5
14	-2.647116	-2.584555	-0.981427	(4.9507, 28.9325)	5
15	-2.162268	-3.632454	-1.036168	(4.7898, 29.9083)	5
16	-1.747801	-3.726295	-1.051808	(4.4427, 29.546)	5
17	2.084066	-2.733138	-2.193548	(6.8557, 22.9138)	6
18	0.739003	-3.601173	-2.091887	(6.3171, 25.2082)	6
19	2.310850	0.324536	-3.640274	(12.7794, 16.8777)	7
20	0.035191	1.223851	-2.967742	(9.2339, 17.4204)	7
21	2.084066	-2.107527	-3.874878	(14.012, 20.505)	8
22	-1.059629	-3.491691	-3.913978	(14.6581, 24.9566)	8
23	-2.913001	3.038123	-3.945259	(16.566, 17.6678)	8



APPENDIX

Q. 1.1 a) and b)

```
import random
from sympy.combinatorics.graycode import GrayCode
from sympy.combinatorics.graycode import gray_to_bin
from deap import creator, base, tools, algorithms
import numpy as np
import matplotlib.pyplot as plt

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

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,....


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 eval_sphere(individual):
    sep=separatevariables(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),
#----------
# Operator registration
#----------
# register the goal / fitness function
toolbox.register("evaluate", eval_sphere)

# register the crossover operator
toolbox.register("mate", tools.cxTwoPoint)
```

```
# 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):
    indasstring=''.join(map(str, c))
    degray=gray_to_bin(indasstring)
    numasint=int(degray, 2) # convert to int from base 2 list
    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:])

plfits = []  #list of fitness values inversed to minimisation values from maximisation
betters = [] # list of individuals with the best fitness in each generation

def main():

    # create an initial population of individuals (where
    # each individual is a list of integers)
    pop = toolbox.population(n=popSize)


    # Evaluate the entire population
    fitnesses = list(map(toolbox.evaluate, pop))
    #print(fitnesses)
    for ind, fit in zip(pop, fitnesses):
        #print(ind, fit)
        ind.fitness.values = fit

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

    # Extracting all the fitnesses of
    fits = [ind.fitness.values[0] for ind in pop]

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

    # Begin the evolution
    while g < iterations:
        # A new generation
        g = g + 1
        print("-- Generation %i --" % g)


        # Select the next generation individuals
        offspring = tools.selBest(pop, nElitists) + toolbox.select(pop,len(pop)-nElitists)
        good_ind = tools.selBest(pop, 1)[0]
```

```
    betters.append(separatevariables(good_ind))
    # 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]):

        # cross two individuals with probability CXPB
        if random.random() < crossProb:
            #print('before crossover ',child1, child2)
            toolbox.mate(child1, child2)
            #print('after crossover ',child1, child2)

            # fitness values of the children
            # must be recalculated later
            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


    # The population is entirely replaced by the offspring
    pop[:] = offspring
    plfits.append(1/max(fits))

    if g%dspInterval ==0:
        # Gather all the fitnesses in one list and print the stats
        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[0]**(-1)))
print("Decoded x1, x2 is %s, %s" % (separatevariables(best_ind)))
plt.plot(np.arange(0,100),plfits,'g-')
plt.xlabel("Generation")
plt.ylabel("Fitness")
```

```
    plt.title("Fitness Plot")
    plt.show()


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

Q. 1.1 c)

```
from pylab import *

import matplotlib
import matplotlib.pyplot as plt
import numpy as np

from mpl_toolkits.mplot3d.axes3d import Axes3D

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

xlist, ylist = map(list, zip(*betters))
zlist = []
for i in range(100):
    zlist.append(f(xlist[i],ylist[i]))

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

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[99], ylist[99], zlist[99], color="r", marker='o', zorder=10)
ax.view_init(55, 30)
```

Q. 1.2

```
from pylab import *

import matplotlib
import matplotlib.pyplot as plt
import numpy as np

from mpl_toolkits.mplot3d.axes3d import Axes3D

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

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)
```

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

def dx2(x1,x2):
    return x1-8*(x2-0.05)+16*x2**3

x1 = 1    #first initialising point (x1)
x2 = 0    #first initialising point (x2)

xlist=[]  #values of x1 for consecutive steps of gradient descent
ylist=[]  #values of x2 for consecutive steps of gradient descent
zlist=[]  #list of objective values of minimisation function
alpha=0.05

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

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(xlist[29], ylist[29], zlist[29], color="r", marker='o', zorder=10)
ax.view_init(80, 30)

x1 = 1.5  #second initialising point (x1)
x2 = 0    #second initialising point (x2)
xlist1=[]
ylist1=[]
zlist1=[]
alpha=0.05

for step in range(0,30):
    newx1=x1-alpha*(dx1(x1,x2))
    x2=x2-alpha*(dx2(x1,x2))
    x1=newx1
    z=f(x1,x2)
    xlist1.append(x1)
    ylist1.append(x2)
    zlist1.append(z)

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(xlist1, ylist1,zlist1, color="k", marker='o', zorder=10)
ax.plot3D(xlist1[29], ylist1[29], zlist1[29], color="r", marker='o', zorder=10)
ax.view_init(80, 30)
```

Q. 2. a) and b)

```
import operator
import random
```

```
import numpy
import math
from deap import base
from deap import benchmarks
from deap import creator
from deap import tools
import numpy as np
import matplotlib.pyplot as plt

posMinInit      = -500     #limits of particle position
posMaxInit      = +500     #limits of particle position
VMaxInit        = 1.5
VMinInit        = 0.5
populationSize  = 50
dimension       = 20
interval        = 10
iterations      = 400

#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)
# particle rerpresented by list of 5 things
# 1. fitness of the particle,
# 2. speed of the particle which is also going to be a list,
# 3.4. limit of the speed value,
# 5. best state the particle has been in so far.

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))

    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))]


    #clamp limits
    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)
```

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

def eval_func(part):
    f=0
    for i in range(dimension):
        f+=(part[i]*math.sin(math.sqrt(abs(part[i]))))
    return (-f,)

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_func)

fitpar = [] #list of fitness values of best particle



def main():
    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
                fitpar.append(best.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 fitness is ',best.fitness)
    for i in range(400-len(fitpar)):
        fitpar.append(fitpar[len(fitpar)-1])
```

```
    print('best particle position is ',best)
    plt.plot(np.arange(0,iterations),fitpar[:400],'c-')
    plt.xlabel("Generation")
    plt.ylabel("Fitness")
    plt.title("Result from Canonical PSO")
    return pop, logbook, best

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

Q. 2. c)

```
import operator
import random
import numpy
import math
from deap import base
from deap import benchmarks
from deap import creator
from deap import tools

posMinInit      = -500
posMaxInit      = +500
VMaxInit        = 1.5
VMinInit        = 0.5
dimension       = 20
interval        = 10
iterations      = 500
populationSize  = 100+int(dimension/10)

#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)
# particle rerpresented by list of 5 things
# 1. fitness of the particle,
# 2. speed of the particle which is also going to be a list,
# 3.4. limit of the speed value,
# 5. best state the particle has been in so far.

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*epsilon*(center[j]-part[j])
        part[j]=part[j]+part.speed[j]

    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)

def eval_func(part):
    f=0
    for i in range(dimension):
        f+=(part[i]*math.sin(math.sqrt(abs(part[i]))))
    return (-f,)

toolbox = base.Toolbox()
toolbox.register("particle", generate, size=dimension, smin=-15, smax=15)
toolbox.register("population", tools.initRepeat, list, toolbox.particle)
toolbox.register("update", updateParticle)
toolbox.register("evaluate", eval_func)

fitpar = [] #list of fitness values of best particle

def main():
    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)
        fitpar.append(pop[0].fitness.values)
        #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)

    plt.plot(np.arange(0,iterations),fitpar,'b-')
    plt.xlabel("Generation")
    plt.ylabel("Fitness")
    plt.title("Result from SL-PSO")
    return pop, logbook

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

Q. 3.1 and 3.2

```
import random
import array
import random
import json
import numpy
import pandas as pd
from sympy.combinatorics.graycode import GrayCode
from sympy.combinatorics.graycode import random_bitstring, gray_to_bin, bin_to_gray
from deap import creator, base, tools, algorithms
from deap.benchmarks.tools import diversity, convergence, hypervolume

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

numofbits = 10
dimension = 3   #number of decision variables
popSize   = 24  #population size

random.seed(10)

def eval_func(individual):
    sep = separatevariables(individual)
    f1 = (((sep[0]-0.6)/1.6)**2 + (sep[1]/3.4)**2 + (sep[2]-1.3)**2)/2
    f2 = (((sep[0]/1.9)-2.3)**2 + ((sep[1]/3.3)-7.1)**2 + (sep[2]+4.3)**2)/3
    F1 = round(f1, 4)
    F2 = round(f2, 4)
    return (F1,F2)

toolbox = base.Toolbox()
toolbox.register("gray_code", random_bitstring, numofbits) #gray code generator
toolbox.register("individual", tools.initRepeat, creator.Individual,
    toolbox.gray_code, dimension)
```

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


def chrom2real(c):
    numasint=int(c, 2) # convert to int from base 2 list
    numinrange=-4+8*numasint/1023  #maximum possible value with 10 bit strings
    return numinrange

def separatevariables(v):
    indasstring=''.join(map(str, v))
    v=gray_to_bin(indasstring)
    return chrom2real(v[0:numofbits]),chrom2real(v[numofbits:2*numofbits]),chrom2real(v[2*numofbits:])


pop = toolbox.population(n=popSize)

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


decod_values = []    #The decoded values from gray coded individuals
for i in range(24):
    decod_values.append((separatevariables(pop[i])))

table = pd.DataFrame(decod_values, columns = ["x1","x2","x3"])
table["f1","f2"] = fitnesses
print(table)  #table for question 3.1



x,y = map(list, zip(*fitnesses))
import matplotlib.pyplot as plt
plt.plot(x,y,'k*')
plt.show

# Efficient non-dominated sorting

def domicheck(ind, current_front):
    for ind in current_front[::-1]:
        for j in reversed(range(len(current_front))):
            if (ind[1]<current_front[j][1]):
                return True
        return False

def front_classifier(ind, fronts):
    index = 0
    while True:
        current_front = fronts[index]
        dominated = domicheck(ind, current_front)
        if not dominated:
            fronts[index].append(ind)
            return fronts
        index += 1
        if index+1 > len(fronts):
            new_front = [ind]
            fronts.append(new_front)
            return fronts

fronts = [[]]
```

```
def efficient_NDsorting(fitnesses):
    fitnesses_copy = toolbox.clone(fitnesses)
    fitnesses_copy.sort()
    fronts = [[fitnesses_copy[0]]]
    fitnesses_copy.pop(0)
    for ind in fitnesses_copy:
        fronts = front_classifier(ind, fronts)
    return fronts

fronts = efficient_NDsorting(fitnesses)
print("\n")

def front_finder(fit,fronts):
    for i in range(len(fronts)):
        for j in range(len(fronts[i])):
            if (fit==fronts[i][j]):
                return i+1
            else:
                continue

table1 = pd.DataFrame(decod_values, columns=["x1","x2","x3"])
table1["f1","f2"] = fitnesses  #a second table created for adding front number

front_list = []  #list of front numbers
for i in range(len(fitnesses)):
    front_list.append(front_finder(fitnesses[i],fronts))

table1["Front number"] = front_list
front_table = table1.sort_values(by = "Front number").reset_index(drop=True)
front_table.index = np.arange(0,len(front_table))
front_table.style.highlight_max(subset=["f1","f2"],color="red",axis=0) #highlight worst f1,f2 values
print(front_table) #table arranged in ascending order of front number

#2D plot of fronts with different colours
frontx = []  #list of f1 values
fronty = []  #list of f2 values
plt.figure()
plt.title("Fronts in population")
colours = iter(cm.hsv(np.linspace(0,1,len(fronts))))
for i in range(len(fronts)):
    for j in range(len(fronts[i])):
        frontx.append(fronts[i][j][0])
        fronty.append(fronts[i][j][1])
    plt.plot(frontx,fronty,'o',color=next(colours),label="Front %s"%(i+1))
    frontx = []
    fronty = []
plt.xlabel("f1")
plt.ylabel("f2")
plt.legend(loc="upper left", bbox_to_anchor=(1.05, 1))
plt.show()
```