# initial population of random bitstring

pop = [randint(0, 2, n\_bits).tolist() for \_ in range(n\_pop)

# enumerate generations

for gen in range(n\_iter):

# evaluate all candidates in the population

scores = [objective(c) for c in pop]

# tournament selection

def selection(pop, scores, k=3):

# first random selection

selection\_ix = randint(len(pop))

for ix in randint(0, len(pop), k-1):

# check if better (e.g. perform a tournament)

if scores[ix] < scores[selection\_ix]:

selection\_ix = ix

return pop[selection\_ix]

# select parents

selected = [selection(pop, scores) for \_ in range(n\_pop)]

# crossover two parents to create two children

def crossover(p1, p2, r\_cross):

# children are copies of parents by default

c1, c2 = p1.copy(), p2.copy()

# check for recombination

if rand() < r\_cross:

# select crossover point that is not on the end of the string

pt = randint(1, len(p1)-2)

# perform crossover

c1 = p1[:pt] + p2[pt:]

c2 = p2[:pt] + p1[pt:]

return [c1, c2]

# mutation operator

def mutation(bitstring, r\_mut):

for i in range(len(bitstring)):

# check for a mutation

if rand() < r\_mut:

# flip the bit

bitstring[i] = 1 - bitstring[i]

# create the next generation

children = list()

for i in range(0, n\_pop, 2):

# get selected parents in pairs

p1, p2 = selected[i], selected[i+1]

# crossover and mutation

for c in crossover(p1, p2, r\_cross):

# mutation

mutation(c, r\_mut)

# store for next generation

children.append(c)

# genetic algorithm

def genetic\_algorithm(objective, n\_bits, n\_iter, n\_pop, r\_cross, r\_mut):

# initial population of random bitstring

pop = [randint(0, 2, n\_bits).tolist() for \_ in range(n\_pop)]

# keep track of best solution

best, best\_eval = 0, objective(pop[0])

# enumerate generations

for gen in range(n\_iter):

# evaluate all candidates in the population

scores = [objective(c) for c in pop]

# check for new best solution

for i in range(n\_pop):

if scores[i] < best\_eval:

best, best\_eval = pop[i], scores[i]

print(">%d, new best f(%s) = %.3f" % (gen, pop[i], scores[i]))

# select parents

selected = [selection(pop, scores) for \_ in range(n\_pop)]

# create the next generation

children = list()

for i in range(0, n\_pop, 2):

# get selected parents in pairs

p1, p2 = selected[i], selected[i+1]

# crossover and mutation

for c in crossover(p1, p2, r\_cross):

# mutation

mutation(c, r\_mut)

# store for next generation

children.append(c)

# replace population

pop = children

return [best, best\_eval]

# objective function

def objective(x):

return x[0]\*\*2.0 + x[1]\*\*2.0

# define range for input

bounds = [[-5.0, 5.0], [-5.0, 5.0]]

# bits per variable

n\_bits = 16

# mutation rate

r\_mut = 1.0 / (float(n\_bits) \* len(bounds))

# initial population of random bitstring

pop = [randint(0, 2, n\_bits\*len(bounds)).tolist() for \_ in range(n\_pop)]

# decode bitstring to numbers

def decode(bounds, n\_bits, bitstring):

decoded = list()

largest = 2\*\*n\_bits

for i in range(len(bounds)):

# extract the substring

start, end = i \* n\_bits, (i \* n\_bits)+n\_bits

substring = bitstring[start:end]

# convert bitstring to a string of chars

chars = ''.join([str(s) for s in substring])

# convert string to integer

integer = int(chars, 2)

# scale integer to desired range

value = bounds[i][0] + (integer/largest) \* (bounds[i][1] - bounds[i][0])

# store

decoded.append(value)

return decoded

# decode population

decoded = [decode(bounds, n\_bits, p) for p in pop]

# evaluate all candidates in the population

scores = [objective(d) for d in decoded]

# genetic algorithm

def genetic\_algorithm(objective, bounds, n\_bits, n\_iter, n\_pop, r\_cross, r\_mut):

# initial population of random bitstring

pop = [randint(0, 2, n\_bits\*len(bounds)).tolist() for \_ in range(n\_pop)]

# keep track of best solution

best, best\_eval = 0, objective(decode(bounds, n\_bits, pop[0]))

# enumerate generations

for gen in range(n\_iter):

# decode population

decoded = [decode(bounds, n\_bits, p) for p in pop]

# evaluate all candidates in the population

scores = [objective(d) for d in decoded]

# check for new best solution

for i in range(n\_pop):

if scores[i] < best\_eval:

best, best\_eval = pop[i], scores[i]

print(">%d, new best f(%s) = %f" % (gen, decoded[i], scores[i]))

# select parents

selected = [selection(pop, scores) for \_ in range(n\_pop)]

# create the next generation

children = list()

for i in range(0, n\_pop, 2):

# get selected parents in pairs

p1, p2 = selected[i], selected[i+1]

# crossover and mutation

for c in crossover(p1, p2, r\_cross):

# mutation

mutation(c, r\_mut)

# store for next generation

children.append(c)

# replace population

pop = children

return [best, best\_eval]

# define range for input

bounds = [[-5.0, 5.0], [-5.0, 5.0]]

# define the total iterations

n\_iter = 100

# bits per variable

n\_bits = 16

# define the population size

n\_pop = 100

# crossover rate

r\_cross = 0.9

# mutation rate

r\_mut = 1.0 / (float(n\_bits) \* len(bounds))

# perform the genetic algorithm search

best, score = genetic\_algorithm(objective, bounds, n\_bits, n\_iter, n\_pop, r\_cross, r\_mut)

print('Done!')

decoded = decode(bounds, n\_bits, best)

print('f(%s) = %f' % (decoded, score))

>0, new best f([-0.785064697265625, -0.807647705078125]) = 1.268621

>0, new best f([0.385894775390625, 0.342864990234375]) = 0.266471

>1, new best f([-0.342559814453125, -0.1068115234375]) = 0.128756

>2, new best f([-0.038909912109375, 0.30242919921875]) = 0.092977

>2, new best f([0.145721435546875, 0.1849365234375]) = 0.055436

>3, new best f([0.14404296875, -0.029754638671875]) = 0.021634

>5, new best f([0.066680908203125, 0.096435546875]) = 0.013746

>5, new best f([-0.036468505859375, -0.10711669921875]) = 0.012804

>6, new best f([-0.038909912109375, -0.099639892578125]) = 0.011442

>7, new best f([-0.033111572265625, 0.09674072265625]) = 0.010455

>7, new best f([-0.036468505859375, 0.05584716796875]) = 0.004449

>10, new best f([0.058746337890625, 0.008087158203125]) = 0.003517

>10, new best f([-0.031585693359375, 0.008087158203125]) = 0.001063

>12, new best f([0.022125244140625, 0.008087158203125]) = 0.000555

>13, new best f([0.022125244140625, 0.00701904296875]) = 0.000539

>13, new best f([-0.013885498046875, 0.008087158203125]) = 0.000258

>16, new best f([-0.011444091796875, 0.00518798828125]) = 0.000158

>17, new best f([-0.0115966796875, 0.00091552734375]) = 0.000135

>17, new best f([-0.004730224609375, 0.00335693359375]) = 0.000034

>20, new best f([-0.004425048828125, 0.00274658203125]) = 0.000027

>21, new best f([-0.002288818359375, 0.00091552734375]) = 0.000006

>22, new best f([-0.001983642578125, 0.00091552734375]) = 0.000005

>22, new best f([-0.001983642578125, 0.0006103515625]) = 0.000004

>24, new best f([-0.001373291015625, 0.001068115234375]) = 0.000003

>25, new best f([-0.001373291015625, 0.00091552734375]) = 0.000003

>26, new best f([-0.001373291015625, 0.0006103515625]) = 0.000002

>27, new best f([-0.001068115234375, 0.0006103515625]) = 0.000002

>29, new best f([-0.000152587890625, 0.00091552734375]) = 0.000001

>33, new best f([-0.0006103515625, 0.0]) = 0.000000

>34, new best f([-0.000152587890625, 0.00030517578125]) = 0.000000

>43, new best f([-0.00030517578125, 0.0]) = 0.000000

>60, new best f([-0.000152587890625, 0.000152587890625]) = 0.000000

>65, new best f([-0.000152587890625, 0.0]) = 0.000000

Done!

f([-0.000152587890625, 0.0]) = 0.000000