```
from numpy.random import randint
from numpy.random import rand
import math
```

```
# The objective function is a two-dimensional inverted Gaussian function, centred at (7, 9)
def objective(x):
  y = math.exp(((x[0]-7)**2) + (x[1]-9)**2)
  return y
```

```
# The decode function decodes binary bitstrings to numbers for each input and scales the value to fall within defined bounds
def decode(bounds, n_bits, bitstring):
  decoded = []  # Create an empty list to hold the decoded values
  largest = 2**n_bits  # The largest value that can be represented with the given number of bits
  for i in range(len(bounds)):

    # Extract the substring for the current value
    start, end = i * n_bits, (i * n_bits) + n_bits  # Define the start and end indices of the substring
    substring = bitstring[start:end]  # Extract the substring

    # Convert the substring to a string of characters
    chars = ''.join([str(s) for s in substring])  # Join the values in the substring together into a string of characters

    # Convert the string of characters to an integer
    integer = int(chars, 2)  # Convert the binary number string into an integer

    # Scale the integer to the desired range
    value = bounds[i][0] + (integer/largest) * (bounds[i][1] - bounds[i][0])  # Scale the integer to a value within the defined bounds

    # Store the decoded value
    decoded.append(value)
  return decoded
```

```
#Let us understand the decoding part first that takes the bounds as input along with the
#number of bits and the actual bitstring itself and decodes the bitstring back
#to original values that fall within the test bounds.
test_bounds=[[-10.0, 10.0], [-10.0, 10.0]]
test_n_bits = 16
#test_n_pop = 100
#Generate a random bit string (of values 0 and 1) of length n_bits*len(bounds). In our case 16*2 = 32
test_bit_string = randint(0, 2, test_n_bits*len(test_bounds)).tolist()
decoded_values = decode(test_bounds, test_n_bits, test_bit_string)
print(test_bit_string)
print(decoded_values)
```

```
[0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1]
[-7.381591796875, -2.87017822265625]
```

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

    # Randomly select one index from the population as the initial selection
    selection_ix = randint(len(pop))

    # Perform a tournament among k randomly selected individuals
    for ix in randint(0, len(pop), k-1):

        # Check if the current individual has a better score than the selected one
        if scores[ix] < scores[selection_ix]:
            # Update the selected individual if a better one is found
            selection_ix = ix
    # Return the best individual from the tournament
    return pop[selection_ix]
```

```
#Let us understand the selection process.
#Create a test population
test_bounds=[[-10.0, 10.0], [-10.0, 10.0]]
test_n_bits = 16
test_n_pop = 100
#Create a random population from which we will select based on the scores (from the objective function)
pop = [randint(0, 2, test_n_bits*len(test_bounds)).tolist() for _ in range(test_n_pop)]
# decode population
test_decoded = [decode(test_bounds, test_n_bits, p) for p in pop]
# evaluate all candidates in the population using our objective function
test_scores = [objective(d) for d in test_decoded]
#Run the selection to pick the selected ones from our population
test_selection = selection(pop, test_scores, k=3)

print("From a population of :", len(pop), " the selected pop is: ")
print(test_selection)
```

```
From a population of : 100  the selected pop is:
[1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1]
```

```
def crossover(p1, p2, r_cross):

    # Children are copies of the parents by default
    c1, c2 = p1.copy(), p2.copy()
    # Check if recombination should occur
    if rand() < r_cross:
        # Select a crossover point (not at the end of the string)
        pt = randint(1, len(p1)-2)
        # Perform crossover in the children
        c1 = p1[:pt] + p2[pt:]
        c2 = p2[:pt] + p1[pt:]
    # Return the two children
    return [c1, c2]
```

```
#Let us understand the Crossover .
test_r_cross = 0.9 #Crossover rate
test_bounds=[[-10.0, 10.0], [-10.0, 10.0]]
test_n_bits = 16
test_p1 = randint(0, 2, test_n_bits*len(test_bounds)).tolist()
test_p2 = randint(0, 2, test_n_bits*len(test_bounds)).tolist()
test_c1, test_c2 = crossover(test_p1, test_p2, test_r_cross)
print("Parent 1 is: ", test_p1)
print("Child 1 is : ", test_c1)
print("Parent 2 is: ", test_p2)
print("Child 2 is : ", test_c2)
```

```
Parent 1 is:  [1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1]
Child 1 is :  [1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1]
Parent 2 is:  [1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0]
Child 2 is :  [1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0]
```

```
#The crossover process can generate offsprings that are very similar to the parents. This might cause a new generation with low diversity.
# The mutation process solves this problem by changing the value of some features in the offspring at random.


import random

def mutation(bitstring, r_mut):

    rand = random.random
    for i in range(len(bitstring)):
        # Check for a mutation
        if rand() < r_mut:
            # Flip the bit
            bitstring[i] = 1 - bitstring[i]
    return bitstring
```

```
#Run the cell a few times to see random mutations
#Let us understand the mutation.
# define range for input
test_bounds = [[-10.0, 10.0], [-10.0, 10.0]]
test_n_bits = 16
# mutation rate
r_mut = 1.0 / (float(test_n_bits) * len(test_bounds))
test_bitstring = randint(0, 2, test_n_bits*len(test_bounds)).tolist()
print("String before mutation is ", test_bitstring)
mutation(test_bitstring, r_mut)
print("String after mutation is  ", test_bitstring)
```

```
String before mutation is  [1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0]
String after mutation is   [1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0]
```

```
### Putting all together into our Genetic algorithm that runs until it finds the best
#The whole fitness assignment, selection, recombination, and mutation process is
#repeated until a stopping criterion is satisfied.
#Each generation is likely to be more adapted to the environment than the old one.

# genetic algorithm implementation
def genetic_algorithm(objective, bounds, n_bits, n_iter, n_pop, r_cross, r_mut):


    # initialize the population with random bitstrings
    pop = [randint(0, 2, n_bits * len(bounds)).tolist() for _ in range(n_pop)]

    # track the best solution found so far
    best, best_eval = 0, objective(decode(bounds, n_bits, pop[0]))

    # iterate over generations


# define range for input
bounds = [[-10.0, 10.0], [-10.0, 10.0]]
# 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('############################################################')
decoded = decode(bounds, n_bits, best)
print('The result is (%s) with a score of %f' % (decoded, score))
```

```
>0, new best f([6.59576416015625, -4.91668701171875]) = 15226255570032929632811703095627561406785600829184526631384854884011445909140610
>0, new best f([6.66107177734375, -2.20306396484375]) = 36107644856932539651911364317910757750034257044423311136.000000
>0, new best f([5.64239501953125, 3.71673583984375]) = 8372481809677.679688
>0, new best f([7.9803466796875, 6.5802001953125]) = 912.794440
>0, new best f([7.2625732421875, 9.986572265625]) = 2.835645
>0, new best f([7.00775146484375, 9.33441162109375]) = 1.118391
>2, new best f([7.00775146484375, 9.2803955078125]) = 1.081860
>4, new best f([7.16339111328125, 9.195556640625]) = 1.067094
>5, new best f([7.00714111328125, 9.1986083984375]) = 1.040287
>5, new best f([6.9500732421875, 9.1497802734375]) = 1.025240
>6, new best f([6.8939208984375, 9.07196044921875]) = 1.016567
>7, new best f([6.96868896484375, 9.0716552734375]) = 1.006134
>9, new best f([7.07550048828125, 8.992919921875]) = 1.005767
>9, new best f([7.0281982421875, 9.0032958984375]) = 1.000806
>12, new best f([7.01080322265625, 9.0008544921875]) = 1.000117
>14, new best f([6.990966796875, 9.0032958984375]) = 1.000092
>16, new best f([6.99127197265625, 9.0008544921875]) = 1.000077
>17, new best f([6.99798583984375, 8.99749755859375]) = 1.000010
>28, new best f([7.003173828125, 8.99993896484375]) = 1.000010
>28, new best f([6.99951171875, 8.997802734375]) = 1.000005
>32, new best f([7.0001220703125, 8.9984130859375]) = 1.000003
>33, new best f([7.0001220703125, 9.00146484375]) = 1.000002
>40, new best f([6.99951171875, 8.9996337890625]) = 1.000000
>41, new best f([7.0001220703125, 8.9996337890625]) = 1.000000
>43, new best f([7.0001220703125, 8.99993896484375]) = 1.000000
############################################################
The result is ([7.0001220703125, 8.99993896484375]) with a score of 1.000000
```