# Experiment 3: Optimization
## Experiment 03: (A)

* Aim :- Write a program for implementing gradient descent.

* Libraries :- Numpy, matplotlib, math,

* Theory :- In mathematics, gradient descent is a first order iterative optimization algorithm for finding a local minimum of a differentiable function.

• It is usually calculated by,
current position →→→ initial parameters
baby step →→→ learning rate
direction →→→ partial derivate (gradient)

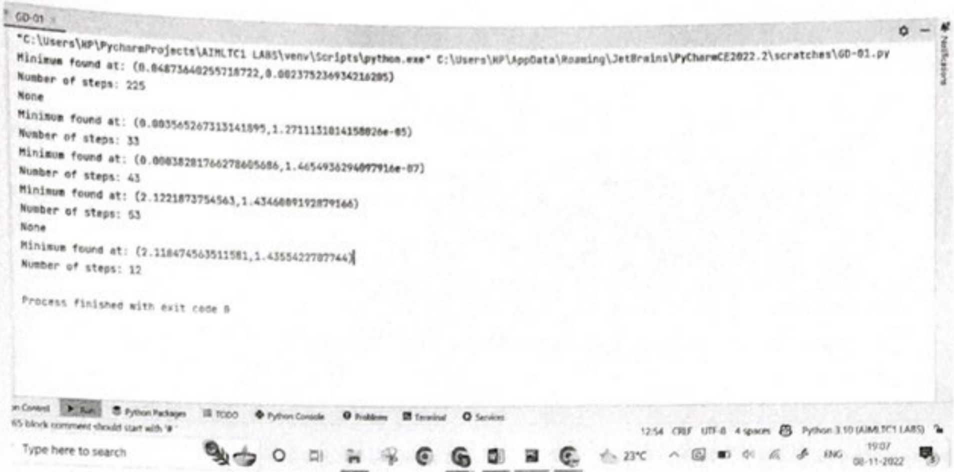
* Algorithm:
1. Start with random initial values for the parameters.
2. Predict values of target variables using the current parameters.
3. Calculate the cost associated with prediction
4. Have we minimized the cost?
   If yes, then go to step 6.
   If no, then go to step 5.
5. Update the parameter, values using the gradient descent algorithm & return to step-2.
6. We have our final updated parameters.

*

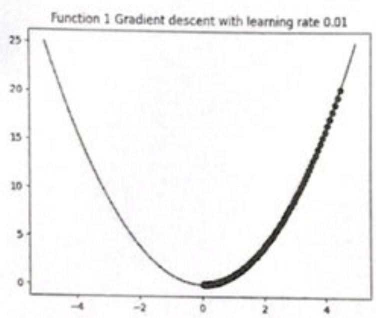
```
from numpy import *
from matplotlib.pyplot import *
import math
import sympy
import matplotlib.pyplot as plt

# Function 1 and its derivative
f1 = lambda x: x * x
deriv_f1 = lambda x: 2 * x
# Function 2 and its derivative
f2 = lambda x: (((sin(10 * math.pi * x)) / (2 * x)) + pow((x - 1), 4))
deriv_f2 = lambda x: -((sin(31.416 * x)) / 2 * pow(x, 1)) + ((15.708 * cos(31.416 *
x)) / (x)) + (4 * pow((x - 1), 3))

errorMargin = 0.001


# l_r is the learning rate
def gradientDesc(functionName, function, deriv, low, up, x_new, x_prev, precision,
l_r):
    x = linspace(low, up, 150)
    x_list, y_list = [x_new], [function(x_new)]
    while abs(x_new - x_prev) > precision:
        x_prev = x_new
        d_x = - deriv(x_prev)
        x_new = x_prev + (l_r * d_x)
        x_list.append(x_new)
        y_list.append(function(x_new))
    plt.scatter(x_list, y_list, c="g")
    plt.plot(x_list, y_list, c="b")
    plt.plot(x, function(x), c="r")
    plt.title(str(functionName) + " Gradient descent with learning rate " +
str(l_r))
    plt.show()
    print("Minimum found at: (" + str(x_new) + "," + str(function(x_new)) + ")")
    print("Number of steps: " + str(len(x_list)))


# Func 1 with three learning rates
functionName = "Function 1"
print(gradientDesc(functionName, f1, deriv_f1, -5, 5, 4.5, 0, errorMargin, 0.01))
gradientDesc(functionName, f1, deriv_f1, -5, 5, 4.5, 0, errorMargin, 0.1)
gradientDesc(functionName, f1, deriv_f1, -5, 5, 4.5, 0, errorMargin, 0.9)

# Func 2 with three learning rates
functionName = "Function 2"
print(gradientDesc(functionName, f2, deriv_f2, 0.5, 2.5, 2.4, 0, errorMargin,
0.001))
gradientDesc(functionName, f2, deriv_f2, 0.5, 2.5, 2.4, 0, errorMargin, 0.005)
gradientDesc(functionName, f2, deriv_f2, 0.5, 2.5, 2.4, 0, errorMargin, 0.009)
```

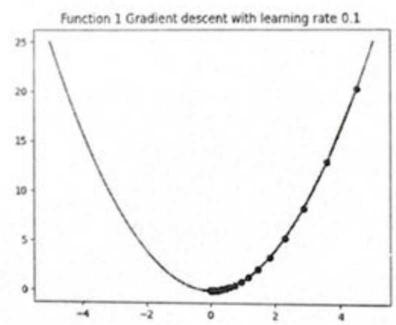
Gradient Descent LAB3(A) 20190802107

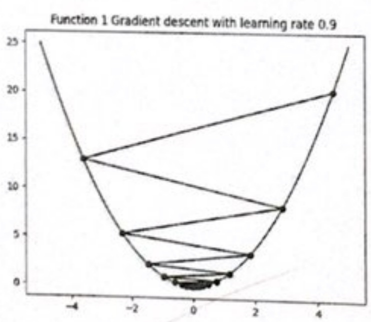
OUTPUT:

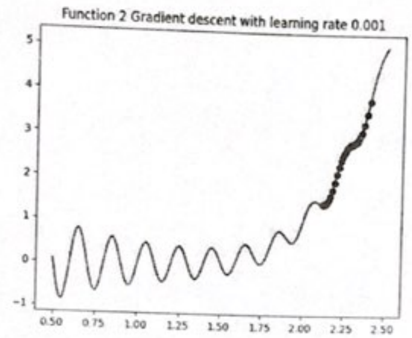


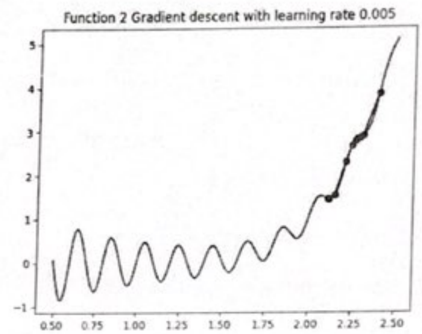
```
"C:\Users\HP\PycharmProjects\AIMLTC1 LABS\venv\Scripts\python.exe" C:\Users\HP\AppData\Roaming\JetBrains\PyCharmCE2022.2\scratches\GD-01.py
Minimum found at: (0.04873640255718722,0.002375236934216205)
Number of steps: 225
None
Minimum found at: (0.003565267313141895,1.271113181415802 6e-05)
Number of steps: 33
Minimum found at: (0.0003828176627860 5686,1.4654936294097916e-07)
Number of steps: 43
Minimum found at: (2.1221873754563,1.434600919287916 6)
Number of steps: 53
None
Minimum found at: (2.118474563511581,1.4355422787744)
Number of steps: 12

Process finished with exit code 0
```

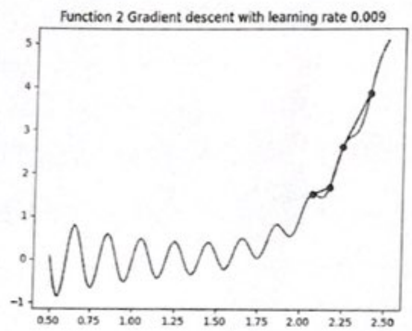



Function 2 Gradient descent with learning rate 0.001


Function 2 Gradient descent with learning rate 0.005


Function 2 Gradient descent with learning rate 0.009

* formula of gradient Descent algorithm :-

$$X = X - lr * \frac{d}{dX} f(X)$$

Where,

$X$, = parameters to be optimized

$f(X)$ = cost function

$lr$ = learning rate

* Conclusion :- Hence, we have implemented gradient descent in python successfully.

# Experiment 03 (b)

* Aim :- Write a program to implement PSO (Particle Swarm Optimisation). Algorithm.

* Software :- Pycharm, Google Collab

* Libraries :- random, math, copy, sys.

* Theory :-

The only two equations that make bare bones PSO algorithm. "k" references the current iteration, "k+1" implies next iteration.

→ Particle posn: $x^i_{k+1} = x^i_k + v^i_{k+1}$

Particle velocity :- $v^i_{k+1} = w_k v^i_k + c_1 r_1 (p^i_k - x^i_k) + c_2 r_2 (p^g_k - x^i_k)$

Here,

| Variable | Definition |
|---|---|
| $x^i_k$ | particle position |
| $v^i_k$ | particle position best individual |
| $p^i_k$ | best individual particle position. |
| $p^g_k$ | best swarm position |
| $w_k$ | constant inertia weight |
| $c_1, c_2$ | cognitive & social parameters |
| $r_1, r_2$ | random numbers between 0 & 1. |

→ from particle velocity equation, two groups emerge:

1. Social term : $c_2 r_2 (p^g_k - x^i_k)$
2. cognitive term : $c_1 r_1 (p^i_k - x^i_k)$

```
import random
import math
import copy
import sys


def fitness_rastrigin(position):
  fitnessVal = 0.0
  for i in range(len(position)):
    xi = position[i]
    fitnessVal += (xi * xi) - (10 * math.cos(2 * math.pi * xi)) + 10
  return fitnessVal


def fitness_sphere(position):
    fitnessVal = 0.0
    for i in range(len(position)):
        xi = position[i]
        fitnessVal += (xi*xi);
    return fitnessVal;


class Particle:
  def __init__(self, fitness, dim, minx, maxx, seed):
    self.rnd = random.Random(seed)

    # initialize position of the particle with 0.0 value
    self.position = [0.0 for i in range(dim)]

     # initialize velocity of the particle with 0.0 value
    self.velocity = [0.0 for i in range(dim)]

    # initialize best particle position of the particle with 0.0 value
    self.best_part_pos = [0.0 for i in range(dim)]

    # loop dim times to calculate random position and velocity
    # range of position and velocity is [minx, max]
    for i in range(dim):
      self.position[i] = ((maxx - minx) *
        self.rnd.random() + minx)
      self.velocity[i] = ((maxx - minx) *
        self.rnd.random() + minx)

    # compute fitness of particle
    self.fitness = fitness(self.position) # curr fitness

    # initialize best position and fitness of this particle
    self.best_part_pos = copy.copy(self.position)
    self.best_part_fitnessVal = self.fitness # best fitness

# particle swarm optimization function
def pso(fitness, max_iter, n, dim, minx, maxx):
```

```
# hyper parameters
w = 0.729    # inertia
c1 = 1.49445 # cognitive (particle)
c2 = 1.49445 # social (swarm)

rnd = random.Random(0)

# create n random particles
swarm = [Particle(fitness, dim, minx, maxx, i) for i in range(n)]

# compute the value of best_position and best_fitness in swarm
best_swarm_pos = [0.0 for i in range(dim)]
best_swarm_fitnessVal = sys.float_info.max # swarm best

# computer best particle of swarm and it's fitness
for i in range(n): # check each particle
  if swarm[i].fitness < best_swarm_fitnessVal:
    best_swarm_fitnessVal = swarm[i].fitness
    best_swarm_pos = copy.copy(swarm[i].position)

# main loop of pso
Iter = 0
while Iter < max_iter:

  # after every 10 iterations
  # print iteration number and best fitness value so far
  if Iter % 10 == 0 and Iter > 1:
    print("Iter = " + str(Iter) + " best fitness = %.3f" % best_swarm_fitnessVal)

  for i in range(n): # process each particle

    # compute new velocity of curr particle
    for k in range(dim):
      r1 = rnd.random()    # randomizations
      r2 = rnd.random()

      swarm[i].velocity[k] = (
                                (w * swarm[i].velocity[k]) +
                                (c1 * r1 * (swarm[i].best_part_pos[k] - swarm[i].position
                                (c2 * r2 * (best_swarm_pos[k] -swarm[i].position[k]))
                              )


      # if velocity[k] is not in [minx, max]
      # then clip it
      if swarm[i].velocity[k] < minx:
        swarm[i].velocity[k] = minx
      elif swarm[i].velocity[k] > maxx:
        swarm[i].velocity[k] = maxx


    # compute new position using new velocity
    for k in range(dim):
      swarm[i].position[k] += swarm[i].velocity[k]
```

```
      # compute fitness of new position
      swarm[i].fitness = fitness(swarm[i].position)

      # is new position a new best for the particle?
      if swarm[i].fitness < swarm[i].best_part_fitnessVal:
        swarm[i].best_part_fitnessVal = swarm[i].fitness
        swarm[i].best_part_pos = copy.copy(swarm[i].position)

      # is new position a new best overall?
      if swarm[i].fitness < best_swarm_fitnessVal:
        best_swarm_fitnessVal = swarm[i].fitness
        best_swarm_pos = copy.copy(swarm[i].position)

    # for-each particle
    Iter += 1
  #end_while
  return best_swarm_pos


print("\nBegin particle swarm optimization on rastrigin function\n")
dim = 2
fitness = fitness_rastrigin


print("Goal is to minimize Rastrigin's function in " + str(dim) + " variables")
print("Function has known min = 0.0 at (", end="")
for i in range(dim-1):
  print("0, ", end="")
print("0)")

num_particles = 40
max_iter = 100

print("Setting num_particles = " + str(num_particles))
print("Setting max_iter    = " + str(max_iter))
print("\nStarting PSO algorithm\n")



best_position = pso(fitness, max_iter, num_particles, dim, -10.0, 10.0)

print("\nPSO completed\n")
print("\nBest solution found:")
print(["%.6f"%best_position[k] for k in range(dim)])
fitnessVal = fitness(best_position)
print("fitness of best solution = %.6f" % fitnessVal)

print("\nEnd particle swarm for rastrigin function\n")


print()
print()


# Driver code for Sphere function
```

```
print("\nBegin particle swarm optimization on sphere function\n")
dim = 3
fitness = fitness_sphere


print("Goal is to minimize sphere function in " + str(dim) + " variables")
print("Function has known min = 0.0 at (", end="")
for i in range(dim-1):
  print("0, ", end="")
print("0)")

num_particles = 50
max_iter = 100

print("Setting num_particles = " + str(num_particles))
print("Setting max_iter    = " + str(max_iter))
print("\nStarting PSO algorithm\n")



best_position = pso(fitness, max_iter, num_particles, dim, -10.0, 10.0)

print("\nPSO completed\n")
print("\nBest solution found:")
print(["%.6f"%best_position[k] for k in range(dim)])
fitnessVal = fitness(best_position)
print("fitness of best solution = %.6f" % fitnessVal)

print("\nEnd particle swarm for sphere function\n")
```

```
Goal is to minimize Rastrigin's function in 2 variables
Function has known min = 0.0 at (0, 0)
Setting num_particles = 40
Setting max_iter    = 100

Starting PSO algorithm

Iter = 10 best fitness = 0.502
Iter = 20 best fitness = 0.133
Iter = 30 best fitness = 0.045
Iter = 40 best fitness = 0.008
Iter = 50 best fitness = 0.000
Iter = 60 best fitness = 0.000
Iter = 70 best fitness = 0.000
Iter = 80 best fitness = 0.000
Iter = 90 best fitness = 0.000

PSO completed


Best solution found:
['0.000001', '-0.000007']
fitness of best solution = 0.000000


End particle swarm for rastrigin function
```

```
Begin particle swarm optimization on sphere function

Goal is to minimize sphere function in 3 variables
Function has known min = 0.0 at (0, 0, 0)
Setting num_particles = 50
Setting max_iter     = 100

Starting PSO algorithm

Iter = 10 best fitness = 0.189
Iter = 20 best fitness = 0.012
Iter = 30 best fitness = 0.001
Iter = 40 best fitness = 0.000
Iter = 50 best fitness = 0.000
Iter = 60 best fitness = 0.000
Iter = 70 best fitness = 0.000
Iter = 80 best fitness = 0.000
Iter = 90 best fitness = 0.000

PSO completed


Best solution found:
['0.000004', '-0.000001', '0.000007']
fitness of best solution = 0.000000

End particle swarm for sphere function
```

Using the two equations the basic flow structure of PSO routine is as follows:-

A) Initialize

1. Set constants: $k_{max}$, $w_k$, $c_1$, $c_2$
2. Randomly initialize particle positions
3. Randomly initialize particle velocities
4. Set $k:1$ (iteration Counter).

B) Optimize

1. Evaluate cost function $f_k^i$ at each particle position $x_k^i$
2. If $f_k^i \leq f_{best}^i$ then $f_{best}^i = f_k^i$ and $p_k^i = x_k^i$
3. If $f_k^i \leq f_{best}^g$ then $f_{best}^g = f_k^i$ and $p_k^g = x_k^i$
4. If stopping condition is satisfied, go to C.
5. Update all particle velocities
6. Update all particle positions
7. Increment $k$
8. Go to B(1).

C) Terminate

* Conclusion :- Hence, the main concept behind PSO, which is evident from particle velocity equation above, is that there is constant balance between three distinct forces pulling on each particle:
1. Particle velocity inertia ; (2) Distance from individual particles best $pos^n$
3. Distance from swarm's best $posn^n$.