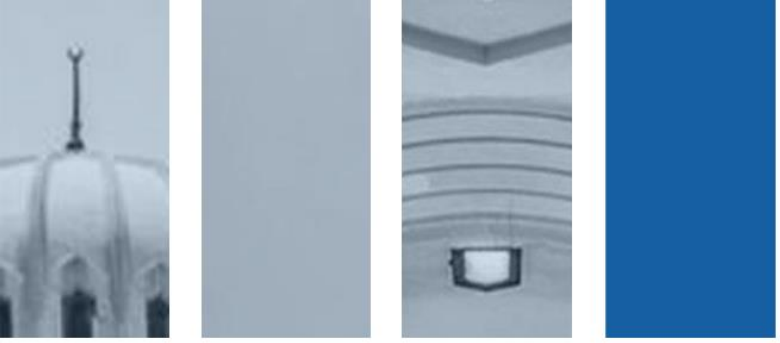


Optimization

Program:

Course Code: CSE473

Course Name: Computational intelligence

Ain Shams University
Faculty of Engineering
Spring Semester – 2022

Submitted for: **Dr. Hossam Abdelmunim**

Student Names:

Eslam Sayed Rady	1902236
Mohamed Hussein Adel	1802683
Abdelrahman Adel Saeed	1805626

Submission Contents

Table osf Contents

1.Problem definition and importance

An optimization problem is nonlinear if the objective function $f(x)$ or any of the inequality constraints $c_i(x) \leq 0$, $i = 1, 2, \ldots, m$, or equality constraints $d_j(x) = 0$, $j = 1, 2, \ldots, n$, are nonlinear functions of the vector of variables x. For example, if x contains the components x_1 and x_2, then the function $3 + 2x_1 - 7x_2$ is linear, whereas the functions $(x_1)^3 + 2x_2$ and $3x_1 + 2x_1x_2 + x_2$ are nonlinear.

Nonlinear problems arise when the objective or constraints cannot be expressed as linear functions without sacrificing some essential nonlinear feature of the real world system. For example, the folded conformation of a protein molecule is believed to be the one that minimizes a certain nonlinear function of the distances between the nuclei of its component atoms—and these distances themselves are nonlinear functions of the positions of the nuclei. In finance, the risk associated with a portfolio of investments, as measured by the variance of the return on the portfolio, is a nonlinear function of the amounts invested in each security in the portfolio. In chemistry, the concentration of each chemical in a solution is often a nonlinear function of time, as reactions between chemicals usually take place according to exponential formulas.

Nonlinear problems can be categorized according to several properties. There are problems in which the objective and constraints are smooth functions, and there are nonsmooth problems in which the slope or value of a function may change abruptly. There are unconstrained problems, in which the aim is to minimize (or maximize) the objective function $f(x)$ with no restrictions on the value of x, and there are constrained problems, in which the components of x must satisfy certain bounds or other more complex interrelationships. In convex problems the graph of the objective function and the feasible set are both convex (where a set is convex if a line joining any two points in the set is contained in the set). Another special case is quadratic programming, in which the constraints are linear but the objective function is quadratic; that is, it contains terms that are multiples of the product of two components of x. (For instance, the function $3(x_1)^2 + 1.4x_1x_2 + 2(x_2)^2$ is a quadratic function of x_1 and x_2.) Another useful way to classify nonlinear problems is according to the number of variables (that is, components of x). Loosely speaking, a problem is said to be “large” if it has more than a thousand or so variables, although the threshold of “largeness” continually increases as computers become more powerful. Another useful distinction is between problems that are computationally “expensive” to evaluate and those that are relatively cheap, as is the case in linear programming.

Nonlinear programming algorithms typically proceed by making a sequence of guesses of the variable vector x (known as iterates and distinguished by superscripts x^1, x^2, x^3, …) with the goal of eventually identifying an optimal value of x. Often, it is not practical to identify the globally optimal value of x. In these cases, one must settle for a local optimum—the best value in some region of the feasible solutions. Each iterate is chosen on the basis of knowledge about the constraint and objective functions gathered at earlier iterates. Most nonlinear programming algorithms are targeted to a particular subclass of problems. For example, some algorithms are specifically targeted to large, smooth unconstrained problems in which the matrix of second derivatives of f(x) contains few nonzero entries and is expensive to evaluate, while other algorithms are aimed specifically at convex quadratic programming problems

2.Methods and algorithms

Optimization problems involve continuous functions, and differentiable optimization problems are thus a special case. In continuous optimization, we shall mostly be concerned with this special case since many classical optimization methods rely on the computation of derivatives.

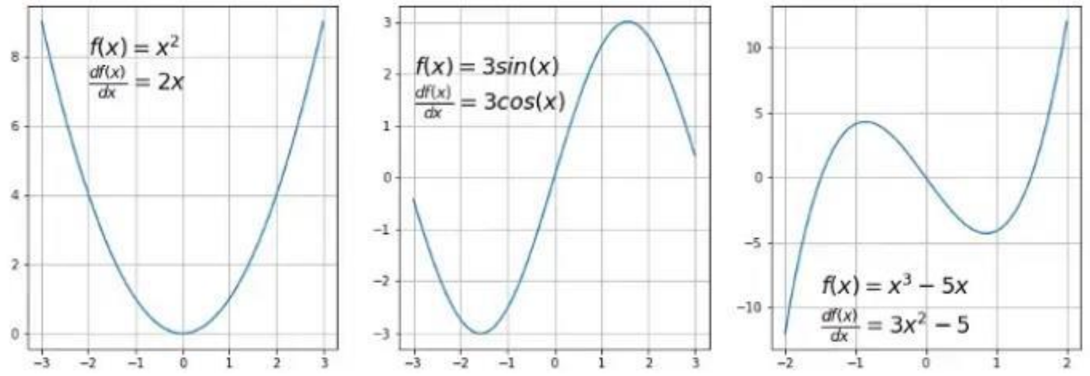
We will now turn to the issue of algorithms for solving such problems. The taxonomy of such algorithms can be constructed in different ways. For example, one may distinguish between analytical and numerical (typically iterative) methods. However, here, we shall make the division with respect to problem type, distinguishing between unconstrained and constrained optimization, starting with the former.

2.1 Conventional gradient descent

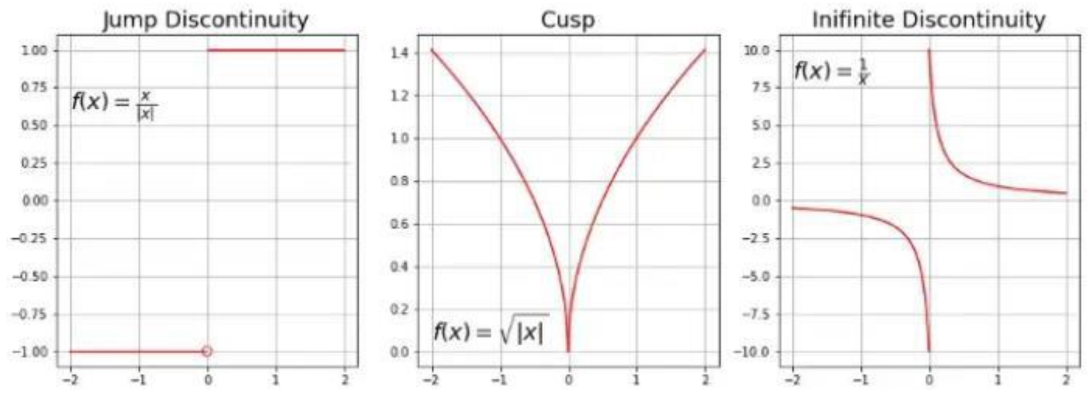
Gradient descent (GD) is an iterative first-order optimization algorithm used to find a local minimum/maximum of a given function. This method is commonly used in *machine learning* (ML) and *deep learning*(DL) to minimize a cost/loss function (e.g. in a linear regression).

Gradient descent algorithm does not work for all functions. There are two specific requirements. A function must be differentiable and convex.

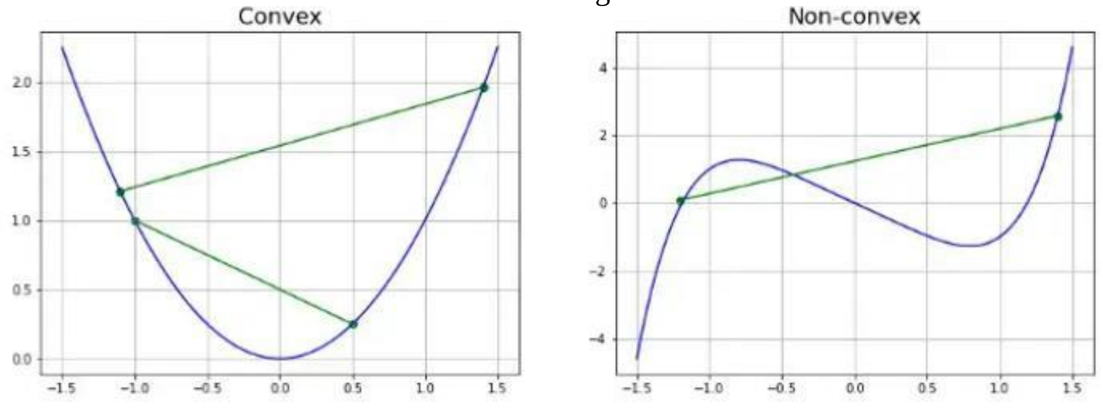
First, what does it mean it has to be differentiable? If a function is differentiable, it has a derivative for each point in its domain, not all functions meet these criteria. First, let’s see some examples of functions meeting this criterion.



Typical non-differentiable functions have a step a cusp or a discontinuity.



Next requirement — function must be convex. For a univariate function, this means that the line segment connecting two function's points lays on or above its curve (it does not cross it). If it does it means that it has a local minimum which is not a global one.



Gradient Descent Algorithm iteratively calculates the next point using gradient at the current position, scales it (by a learning rate) and subtracts obtained value from the current position (makes a step). It subtracts the value because we want to minimize the function (to maximize it would be adding). This process can be written as:

$$p_{n+1} = p_n - \eta \nabla f(p_n)$$

There's an important parameter **η** which scales the gradient and thus controls the step size. In machine learning, it is called **learning rate** and have a strong influence on performance.

- The smaller learning rate the longer GD converges, or may reach maximum iteration before reaching the optimum point

- If learning rate is too big the algorithm may not converge to the optimal point (jump around) or even to diverge completely.

2.2 Newton Raphson method

The **Newton-Raphson method** (also known as Newton's method) is a way to quickly find a good approximation for the root of a real-valued function f(x) = 0*f*(*x*)=0. It uses the idea that a continuous and differentiable function can be approximated by a straight-line tangent to it.

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2} f''(x_0)(x - x_0)^2 \equiv f_{[2]}(x),$$

-where H(x0) is the Hessian matrix

$$H = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1\, \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1\, \partial x_n} \\ \frac{\partial^2 f}{\partial x_2\, \partial x_1} & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n\, \partial x_1} & \cdots & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix}$$

By solving equation

$$x^* = x_0 - \frac{f'(x_0)}{f''(x_0)}.$$

Or by general form

$$\mathbf{x}_{j+1} = \mathbf{x}_j - \eta_j H^{-1}(\mathbf{x}_j) \nabla f(\mathbf{x}_j).$$

2.3line search gradient descent

Line search method is an iterative approach to find a local minimum of a multidimensional nonlinear function using the function's gradients. It computes a search direction and then finds an acceptable step length that satisfies certain standard conditions.

Line search methods can be categorized into exact and inexact methods. The exact method, as in the name, aims to find the exact minimizer at each iteration, while the inexact method computes step lengths to satisfy conditions including Wolfe and Goldstein conditions. Line search and trust-region methods are two fundamental strategies for locating the new iterate given the current point. With the ability to solve the unconstrained optimization problem, line search is widely used in many cases including machine learning, game theory and other fields.

In summary, line search gradient method's steps are:

1-Select a starting point $x0$.

2-Repeat the following steps until $fn=f(xn)$ coverages to a local minimum

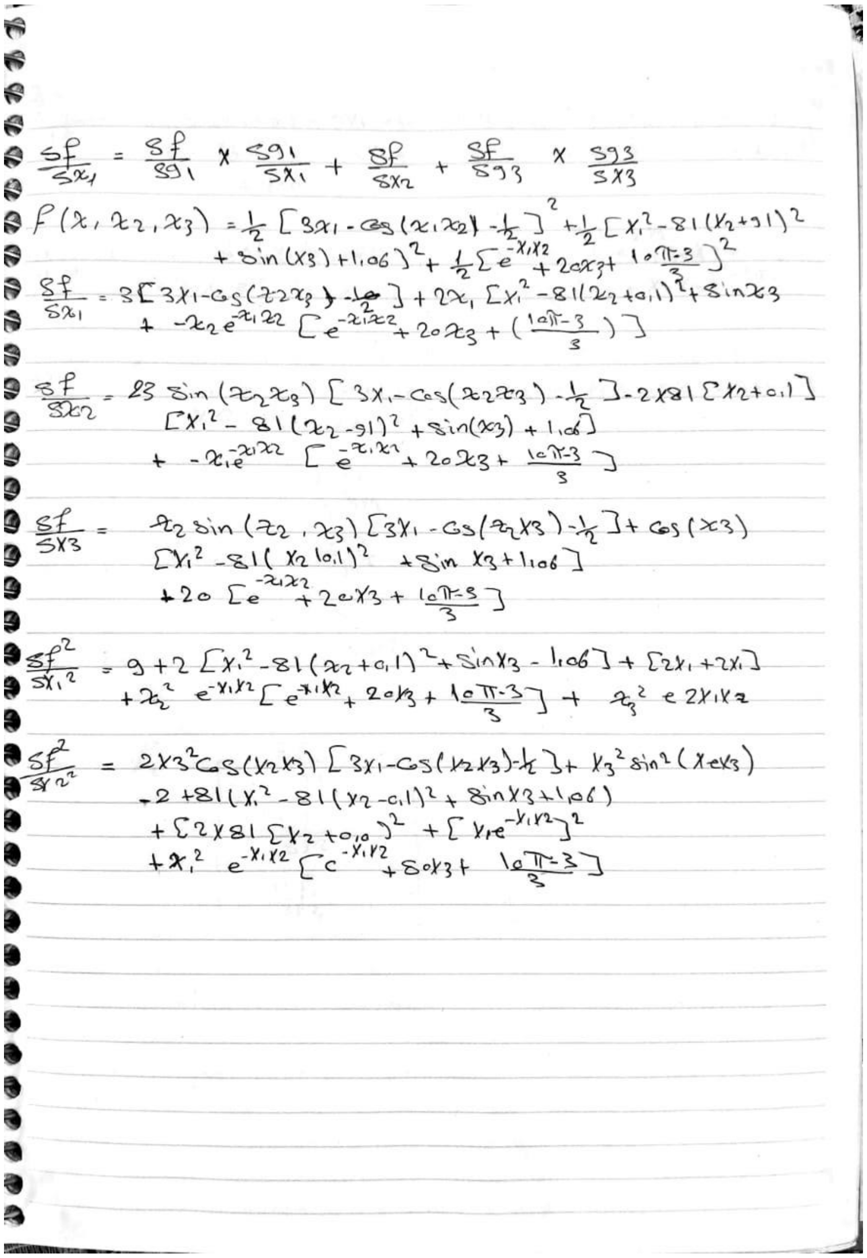
maximum number of iterations reached.

3-Choose a descent direction pn starting at xn, defined as: $\nabla fnpn<0$ for $\nabla fnpn\neq0$.

4-Find a step length $\alpha k>0$ so that $f(xn+\alpha npn)<fn$.

5-Set $xn+1=xn+\alpha npn$.

3. Find an expression for the gradient vector of the function F and expression for the Hessian matrix of the function F



$$\frac{\delta f}{\delta x_1} = \frac{\delta f}{\delta g_1} \times \frac{\delta g_1}{\delta x_1} + \frac{\delta f}{\delta x_2} + \frac{\delta f}{\delta g_3} \times \frac{\delta g_3}{\delta x_3}$$

$$f(x_1, x_2, x_3) = \frac{1}{2}\left[3x_1 - \cos(x_1 x_2) - \frac{1}{2}\right]^2 + \frac{1}{2}\left[x_1^2 - 81(x_2 + 0.1)^2 + \sin(x_3) + 1.06\right]^2 + \frac{1}{2}\left[e^{-x_1 x_2} + 20x_3 + \frac{10\pi - 3}{3}\right]^2$$

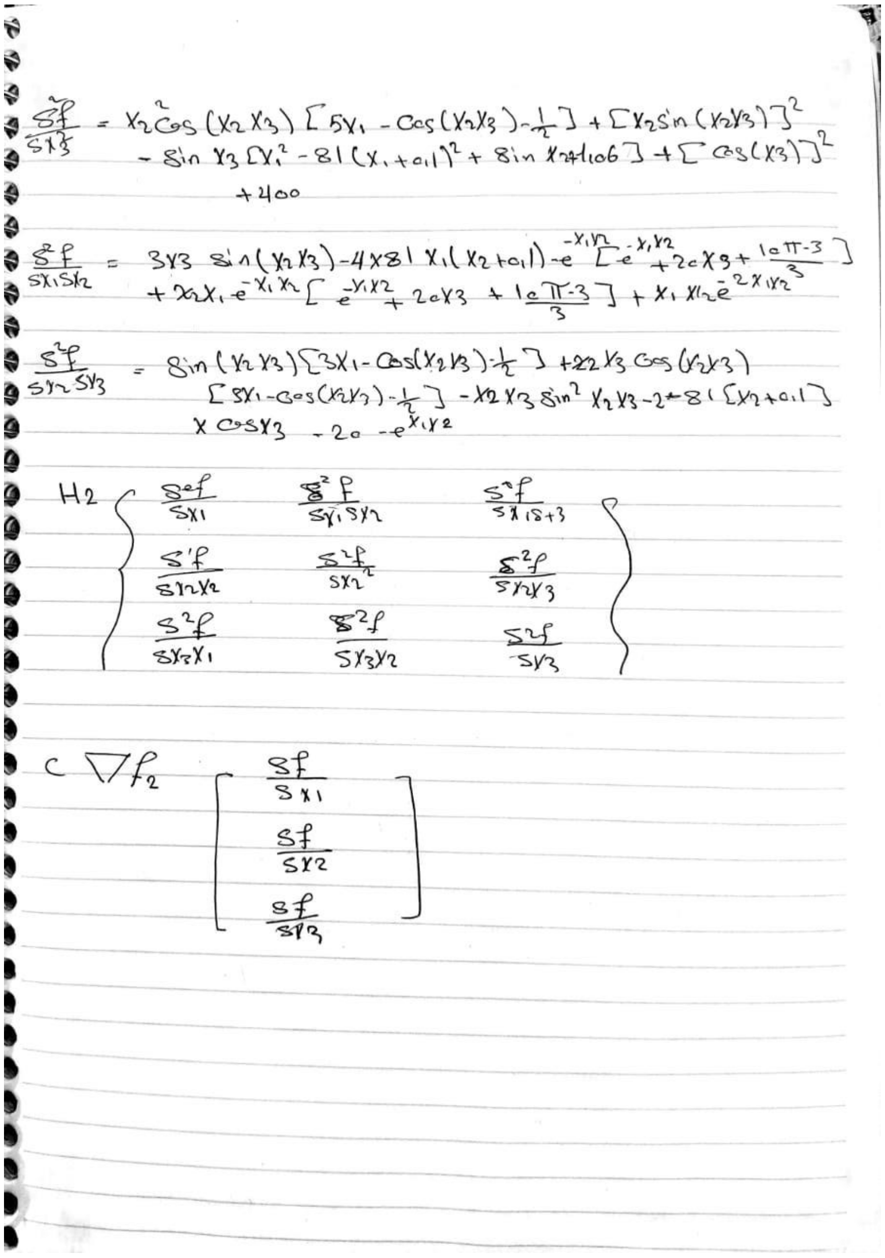
$$\frac{\delta f}{\delta x_1} = 3\left[3x_1 - \cos(x_2 x_3) - \frac{1}{2}\right] + 2x_1\left[x_1^2 - 81(x_2 + 0.1)^2 + \sin x_3 + -x_2 e^{-x_1 x_2}\left[e^{-x_1 x_2} + 20x_3 + \left(\frac{10\pi - 3}{3}\right)\right]\right]$$

$$\frac{\delta f}{\delta x_2} = 23 \sin(x_2 x_3)\left[3x_1 - \cos(x_2 x_3) - \frac{1}{2}\right] - 2 \times 81\left[x_2 + 0.1\right]\left[x_1^2 - 81(x_2 - 91)^2 + \sin(x_3) + 1.06\right] + -x_1 e^{-x_1 x_2}\left[e^{-x_1 x_2} + 20x_3 + \frac{10\pi - 3}{3}\right]$$

$$\frac{\delta f}{\delta x_3} = x_2 \sin(x_2, x_3)\left[3x_1 - \cos(x_2 x_3) - \frac{1}{2}\right] + \cos(x_3)\left[x_1^2 - 81(x_2 10.1)^2 + \sin x_3 + 1.06\right] + 20\left[e^{-x_1 x_2} + 20x_3 + \frac{10\pi - 3}{3}\right]$$

$$\frac{\delta f^2}{\delta x_1^2} = 9 + 2\left[x_1^2 - 81(x_2 + 0.1)^2 + \sin x_3 - 1.06\right] + \left[2x_1 + 2x_1\right] + x_2^2 e^{-x_1 x_2}\left[e^{-x_1 x_2} + 20x_3 + \frac{10\pi - 3}{3}\right] + x_3^2 e 2x_1 x_2$$

$$\frac{\delta f^2}{\delta x_2^2} = 2x_3^2 \cos(x_2 x_3)\left[3x_1 - \cos(x_2 x_3) - \frac{1}{2}\right] + x_3^2 \sin^2(x_2 x_3) - 2 + 81(x_1^2 - 81(x_2 - 0.1)^2 + \sin x_3 + 1.06) + \left[2 \times 81\left[x_2 + 0.0\right)^2 + \left[x_1 e^{-x_1 x_2}\right]^2 + x_1^2 e^{-x_1 x_2}\left[c^{-x_1 x_2} + 80x_3 + \frac{10\pi - 3}{3}\right]$$



$$\frac{\delta^2 f}{\delta x_3^2} = x_2^2 \cos(x_2 x_3)\left[5x_1 - \cos(x_2x_3) - \frac{1}{2}\right] + \left[x_2 \sin(x_2x_3)\right]^2 - \sin x_3\left[x_1^2 - 81(x_1+0.1)^2 + \sin x_3 + 1.06\right] + \left[\cos(x_3)\right]^2 + 400$$

$$\frac{\delta^2 f}{\delta x_1 \delta x_2} = 3x_3 \sin(x_2x_3) - 4 \times 81\, x_1(x_2+0.1) - e^{-x_1x_2}\left[e^{-x_1x_2} + 20x_3 + \frac{10\pi-3}{3}\right] + x_2 x_1 e^{-x_1x_2}\left[e^{-x_1x_2} + 20x_3 + \frac{10\pi-3}{3}\right] + x_1 x_2 e^{-2x_1x_2}$$

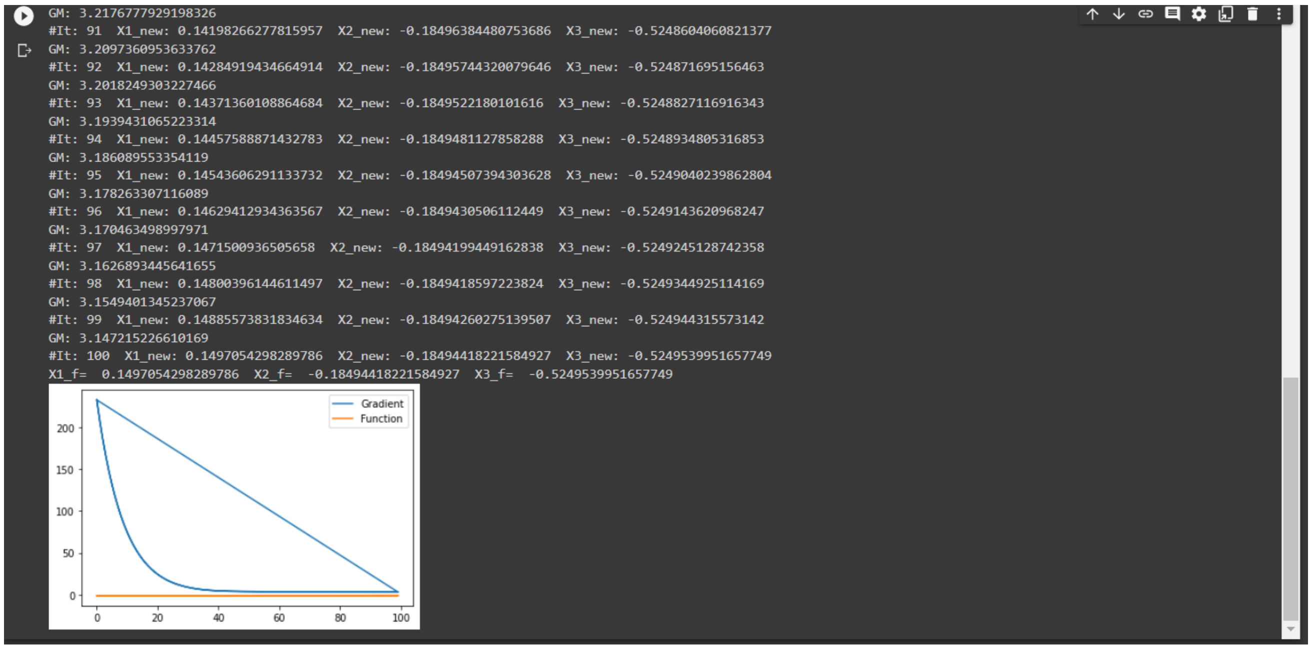
$$\frac{\delta^2 f}{\delta x_2 \delta x_3} = \sin(x_2x_3)\left[3x_1 - \cos(x_2x_3) - \frac{1}{2}\right] + x_2 x_3 \cos(x_2x_3)\left[3x_1 - \cos(x_2x_3) - \frac{1}{2}\right] - x_2 x_3 \sin^2 x_2 x_3 - 2 \times 81\left[x_2 + 0.1\right] \times \cos x_3 - 20 - e^{x_1x_2}$$

$$H_2 \begin{Bmatrix} \frac{\delta^2 f}{\delta x_1} & \frac{\delta^2 f}{\delta x_1 \delta x_2} & \frac{\delta^2 f}{\delta x_1 \delta x_3} \\ \frac{\delta^2 f}{\delta x_2 x_2} & \frac{\delta^2 f}{\delta x_2^2} & \frac{\delta^2 f}{\delta x_2 x_3} \\ \frac{\delta^2 f}{\delta x_3 x_1} & \frac{\delta^2 f}{\delta x_3 x_2} & \frac{\delta^2 f}{\delta x_3} \end{Bmatrix}$$

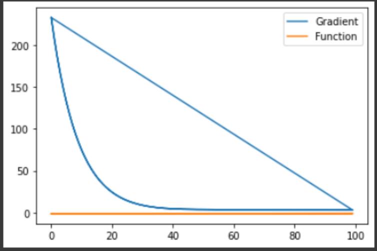
$$C \quad \nabla f_2 \quad \begin{bmatrix} \frac{\delta f}{\delta x_1} \\ \frac{\delta f}{\delta x_2} \\ \frac{\delta f}{\delta x_3} \end{bmatrix}$$

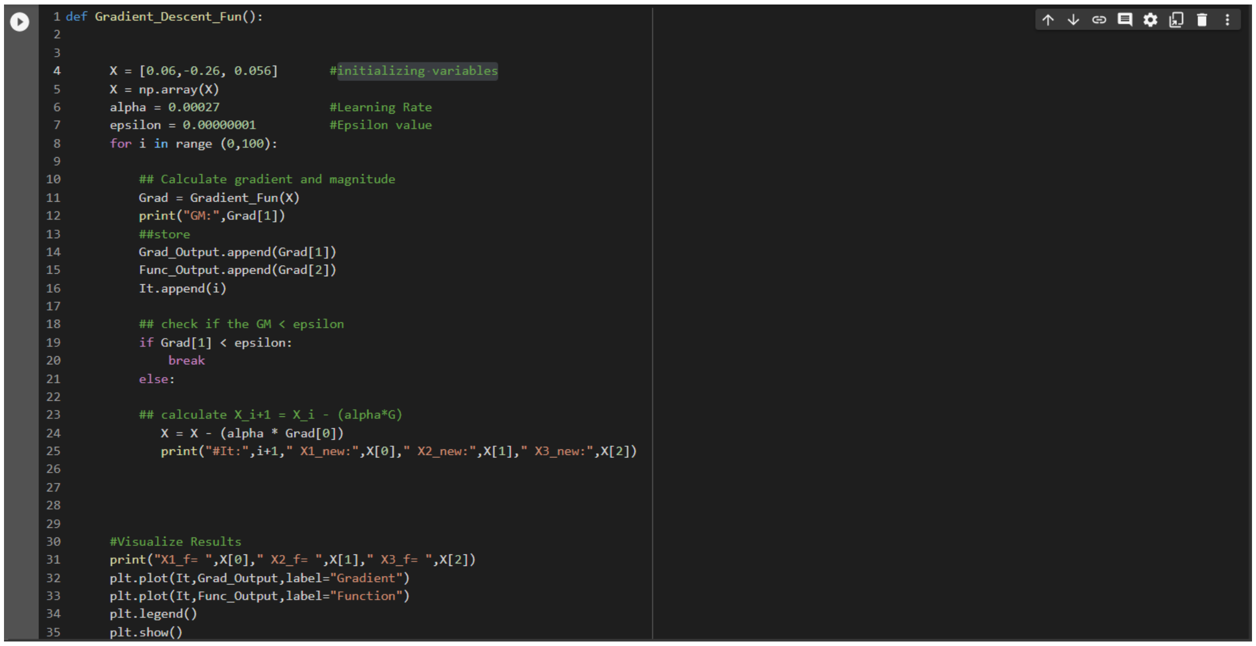
4. Experimental Results (samples of your trails) and discussions.

<u>**Gradiant samples and function:**</u>



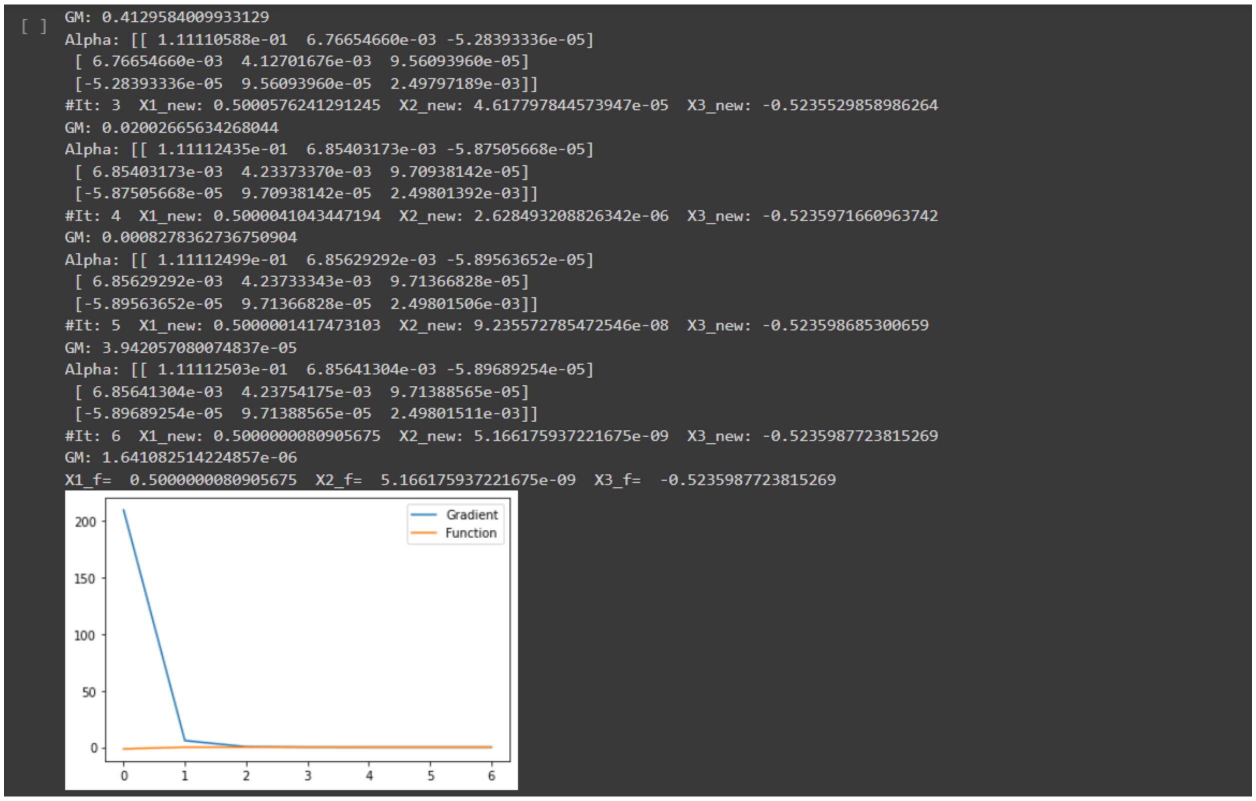
```
GM: 3.2176777929198326
#It: 91  X1_new: 0.14198266277815957  X2_new: -0.18496384480753686  X3_new: -0.524860406082137 7
GM: 3.2097360953633762
#It: 92  X1_new: 0.14284919434664914  X2_new: -0.18495744320079646  X3_new: -0.524871695156463
GM: 3.2018249303227466
#It: 93  X1_new: 0.14371360108864684  X2_new: -0.1849522180101616  X3_new: -0.5248827116916343
GM: 3.1939431065223314
#It: 94  X1_new: 0.14457588871432783  X2_new: -0.1849481127858288  X3_new: -0.5248934805316853
GM: 3.186089553354119
#It: 95  X1_new: 0.14543606291133732  X2_new: -0.18494507394303628  X3_new: -0.5249040239862804
GM: 3.178263307116089
#It: 96  X1_new: 0.14629412934363567  X2_new: -0.1849430506112449  X3_new: -0.5249143620968247
GM: 3.170463498997971
#It: 97  X1_new: 0.1471500936505658  X2_new: -0.18494199449162838  X3_new: -0.5249245128742358
GM: 3.1626893445641655
#It: 98  X1_new: 0.14800396144611497  X2_new: -0.1849418597223824  X3_new: -0.5249344925114169
GM: 3.1549401345237067
#It: 99  X1_new: 0.14885573831834634  X2_new: -0.18494260275139507  X3_new: -0.524944315573142
GM: 3.147215226610169
#It: 100  X1_new: 0.1497054298289786  X2_new: -0.18494418221584927  X3_new: -0.5249539951657749
X1_f=  0.1497054298289786  X2_f=  -0.18494418221584927  X3_f=  -0.5249539951657749
```



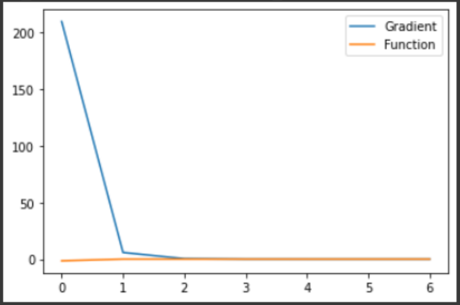


```python
def Gradient_Descent_Fun():


    X = [0.06,-0.26, 0.056]       #initializing variables
    X = np.array(X)
    alpha = 0.00027               #Learning Rate
    epsilon = 0.00000001          #Epsilon value
    for i in range (0,100):

        ## Calculate gradient and magnitude
        Grad = Gradient_Fun(X)
        print("GM:",Grad[1])
        ##store
        Grad_Output.append(Grad[1])
        Func_Output.append(Grad[2])
        It.append(i)

        ## check if the GM < epsilon
        if Grad[1] < epsilon:
            break
        else:

        ## calculate X_i+1 = X_i - (alpha*G)
           X = X - (alpha * Grad[0])
           print("#It:",i+1," X1_new:",X[0]," X2_new:",X[1]," X3_new:",X[2])




    #Visualize Results
    print("X1_f= ",X[0]," X2_f= ",X[1]," X3_f= ",X[2])
    plt.plot(It,Grad_Output,label="Gradient")
    plt.plot(It,Func_Output,label="Function")
    plt.legend()
    plt.show()
```

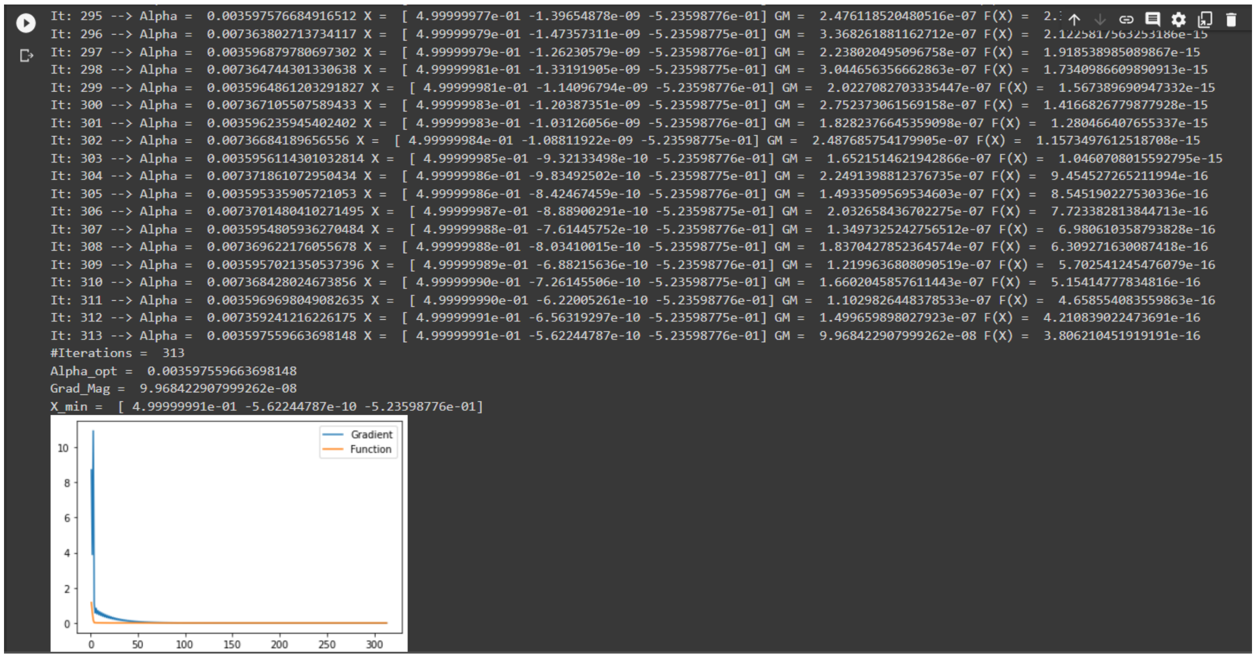
Newton Raphson:



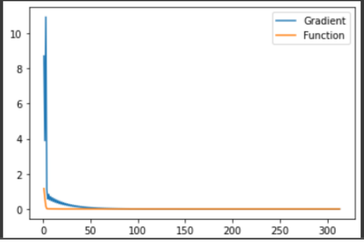
```
[ ]  GM: 0.4129584009933129
     Alpha: [[ 1.11110588e-01  6.76654660e-03 -5.28393336e-05]
      [ 6.76654660e-03  4.12701676e-03  9.56093960e-05]
      [-5.28393336e-05  9.56093960e-05  2.49797189e-03]]
     #It: 3  X1_new: 0.5000576241291245  X2_new: 4.617797844573947e-05  X3_new: -0.5235529858986264
     GM: 0.02002665634268044
     Alpha: [[ 1.11112435e-01  6.85403173e-03 -5.87505668e-05]
      [ 6.85403173e-03  4.23373370e-03  9.70938142e-05]
      [-5.87505668e-05  9.70938142e-05  2.49801392e-03]]
     #It: 4  X1_new: 0.5000041043447194  X2_new: 2.628493208826342e-06  X3_new: -0.5235971660963742
     GM: 0.0008278362736750904
     Alpha: [[ 1.11112499e-01  6.85629292e-03 -5.89563652e-05]
      [ 6.85629292e-03  4.23733343e-03  9.71366828e-05]
      [-5.89563652e-05  9.71366828e-05  2.49801506e-03]]
     #It: 5  X1_new: 0.5000001417473103  X2_new: 9.235572785472546e-08  X3_new: -0.523598685300659
     GM: 3.942057080074837e-05
     Alpha: [[ 1.11112503e-01  6.85641304e-03 -5.89689254e-05]
      [ 6.85641304e-03  4.23754175e-03  9.71388565e-05]
      [-5.89689254e-05  9.71388565e-05  2.49801511e-03]]
     #It: 6  X1_new: 0.5000000080905675  X2_new: 5.166175937221675e-09  X3_new: -0.5235987723815269
     GM: 1.641082514224857e-06
     X1_f=  0.5000000080905675  X2_f=  5.166175937221675e-09  X3_f=  -0.5235987723815269
```



Steepest:



```
It: 295 --> Alpha =  0.003597576684916512 X =  [ 4.99999977e-01 -1.39654878e-09 -5.23598776e-01] GM =  2.476118520480516e-07 F(X) =  2.
It: 296 --> Alpha =  0.007363802713734117 X =  [ 4.99999979e-01 -1.47357311e-09 -5.23598775e-01] GM =  3.368261881162712e-07 F(X) =  2.1225817563253186e-15
It: 297 --> Alpha =  0.003596879780697302 X =  [ 4.99999979e-01 -1.26230579e-09 -5.23598776e-01] GM =  2.238020495096758e-07 F(X) =  1.918538985089867e-15
It: 298 --> Alpha =  0.007364744301330638 X =  [ 4.99999981e-01 -1.33191905e-09 -5.23598775e-01] GM =  3.044656356662863e-07 F(X) =  1.7340986609890913e-15
It: 299 --> Alpha =  0.0035964861203291827 X =  [ 4.99999981e-01 -1.14096794e-09 -5.23598776e-01] GM =  2.0227082703335447e-07 F(X) =  1.567389690947332e-15
It: 300 --> Alpha =  0.007367105507589433 X =  [ 4.99999983e-01 -1.20387351e-09 -5.23598775e-01] GM =  2.752373061569158e-07 F(X) =  1.4166826779877928e-15
It: 301 --> Alpha =  0.003596235945402402 X =  [ 4.99999983e-01 -1.03126056e-09 -5.23598776e-01] GM =  1.8282376645359098e-07 F(X) =  1.280466407655337e-15
It: 302 --> Alpha =  0.00736684189656556 X =  [ 4.99999984e-01 -1.08811922e-09 -5.23598775e-01] GM =  2.487685754179905e-07 F(X) =  1.1573497612518708e-15
It: 303 --> Alpha =  0.0035956114301032814 X =  [ 4.99999985e-01 -9.32133498e-10 -5.23598776e-01] GM =  1.6521514621942866e-07 F(X) =  1.0460708015592795e-15
It: 304 --> Alpha =  0.007371861072950434 X =  [ 4.99999986e-01 -9.83492502e-10 -5.23598775e-01] GM =  2.2491398812376735e-07 F(X) =  9.454527265211994e-16
It: 305 --> Alpha =  0.003595335905721053 X =  [ 4.99999986e-01 -8.42467459e-10 -5.23598776e-01] GM =  1.4933509569534603e-07 F(X) =  8.545190227530336e-16
It: 306 --> Alpha =  0.0073701480410271495 X =  [ 4.99999987e-01 -8.88900291e-10 -5.23598775e-01] GM =  2.032658436702275e-07 F(X) =  7.723382813844713e-16
It: 307 --> Alpha =  0.0035954805936270484 X =  [ 4.99999988e-01 -7.61445752e-10 -5.23598776e-01] GM =  1.3497325242756512e-07 F(X) =  6.980610358793828e-16
It: 308 --> Alpha =  0.007369622176055678 X =  [ 4.99999988e-01 -8.03410015e-10 -5.23598775e-01] GM =  1.8370427852364574e-07 F(X) =  6.309271630087418e-16
It: 309 --> Alpha =  0.0035957021350537396 X =  [ 4.99999989e-01 -6.88215636e-10 -5.23598776e-01] GM =  1.2199636808090519e-07 F(X) =  5.702541245476079e-16
It: 310 --> Alpha =  0.007368428024673856 X =  [ 4.99999990e-01 -7.26145506e-10 -5.23598775e-01] GM =  1.6602045857611443e-07 F(X) =  5.1541477783 4816e-16
It: 311 --> Alpha =  0.0035969698049082635 X =  [ 4.99999990e-01 -6.22005261e-10 -5.23598776e-01] GM =  1.1029826448378533e-07 F(X) =  4.658554083559863e-16
It: 312 --> Alpha =  0.007359241216226175 X =  [ 4.99999991e-01 -6.56319297e-10 -5.23598775e-01] GM =  1.499659898027923e-07 F(X) =  4.210839022473691e-16
It: 313 --> Alpha =  0.003597559663698148 X =  [ 4.99999991e-01 -5.62244787e-10 -5.23598776e-01] GM =  9.968422907999262e-08 F(X) =  3.806210451919191e-16
#Iterations =  313
Alpha_opt =  0.003597559663698148
Grad_Mag =  9.968422907999262e-08
X_min =  [ 4.99999991e-01 -5.62244787e-10 -5.23598776e-01]
```



Comment:

newton-Raphson’s method was the fastest to reach minimum error, on the other hand gradient descent was faster than the normal gradient decent
this shows that newton-Raphson’s methods is perfect with simple equations but with complex one hessian matrix become more complex so at then we can go to gradient descent.

Appendix

Code have more details and comments

https://colab.research.google.com/drive/1ri9ojeH1THsfSd9ASO46TILwHyGUEfWY#scrollTo=4dS4qgLoYAOh