Zainab Al-Suwaykit
Daniel González Esparza

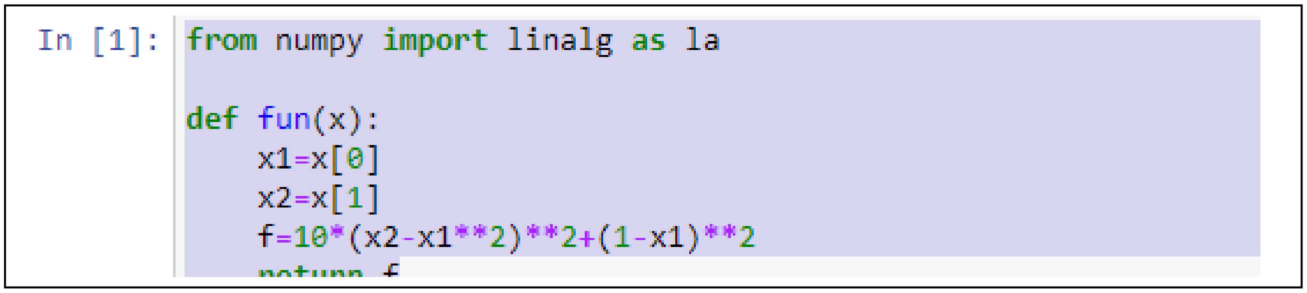
AMCS CS 212 Numerical Optimization Assignment 1

1. Function Plot. The Rosenbrock function is commonly used to test the behavior of optimization algorithms. The function is defined as:

$$f(x) = 10\left(x^2 - x_1^2 1\right)^2 + \left(1 - x_1\right)^2$$

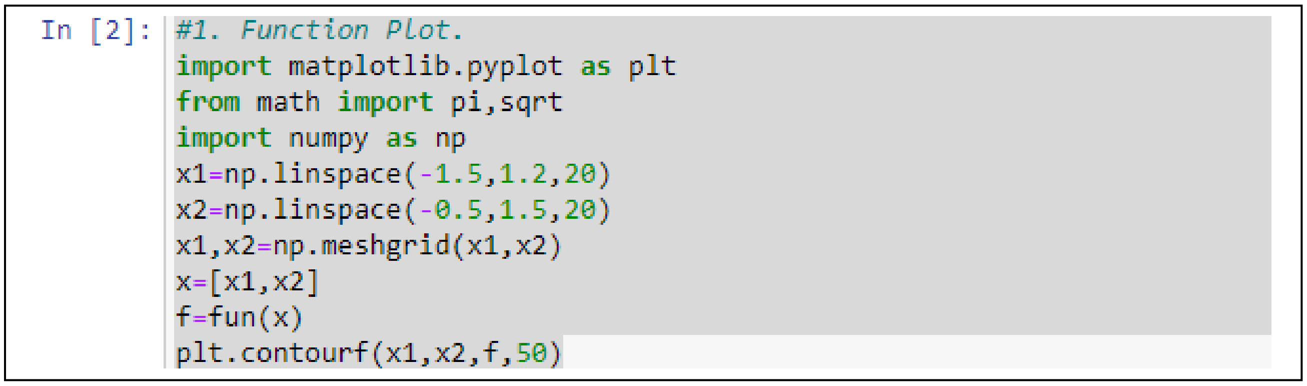
The minimizer of this function is the point x = [1, 1]^t. Draw contour lines of the function in the region near the minimum. You will need to use the function *meshgrid* first to generate a grid, compute the function values at the grid points, and then use the contour function and tune its parameters to obtain the desired contour plot.

Comments: The main part of the code to generate the graphic is given by the code in the **Fig 1.** In the **Fig. 2**, it can be observed that the Rosenbrock function has a minimum around the point x=[1,1] as it is said in the problem statement. It seems that this function is a good example for testing a numerical method of the any minimization method because, there is abrupt changes of gradient direction.



```
In [1]: from numpy import linalg as la

        def fun(x):
            x1=x[0]
            x2=x[1]
            f=10*(x2-x1**2)**2+(1-x1)**2
            return f
```



```
In [2]: #1. Function Plot.
        import matplotlib.pyplot as plt
        from math import pi,sqrt
        import numpy as np
        x1=np.linspace(-1.5,1.2,20)
        x2=np.linspace(-0.5,1.5,20)
        x1,x2=np.meshgrid(x1,x2)
        x=[x1,x2]
        f=fun(x)
        plt.contourf(x1,x2,f,50)
```

Fig. 1 Code compiled to plot the Rosenbrock function.

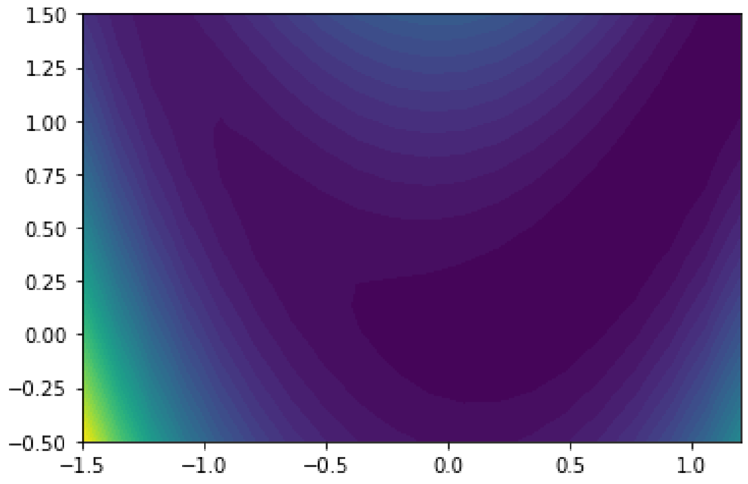
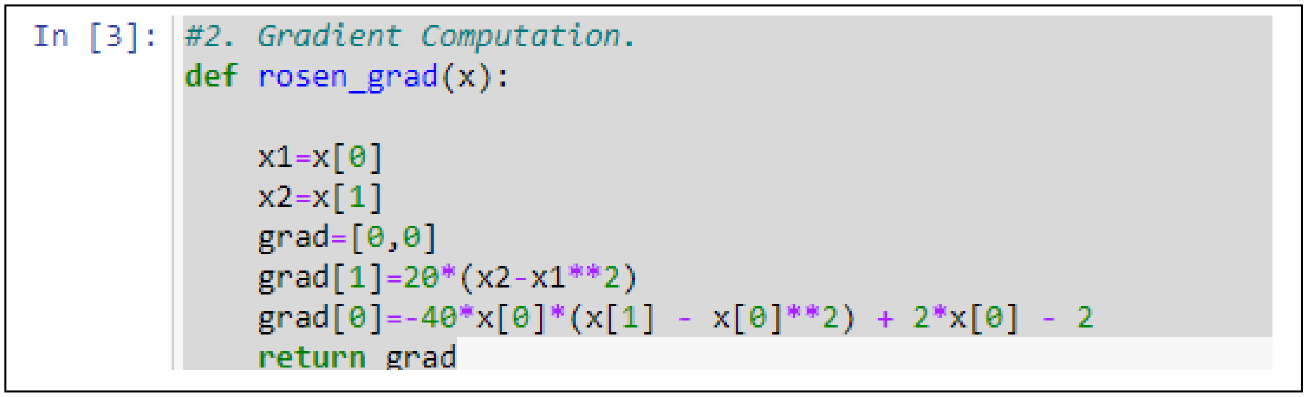


Fig. 2 Contour of the Rosenbrock function.

2. Gradient Computation. Compute the partial derivatives of the Rosenbrock function analytically and use them to write a function that returns the gradient of the Rosenbrock function.
def rosen_grad(x)



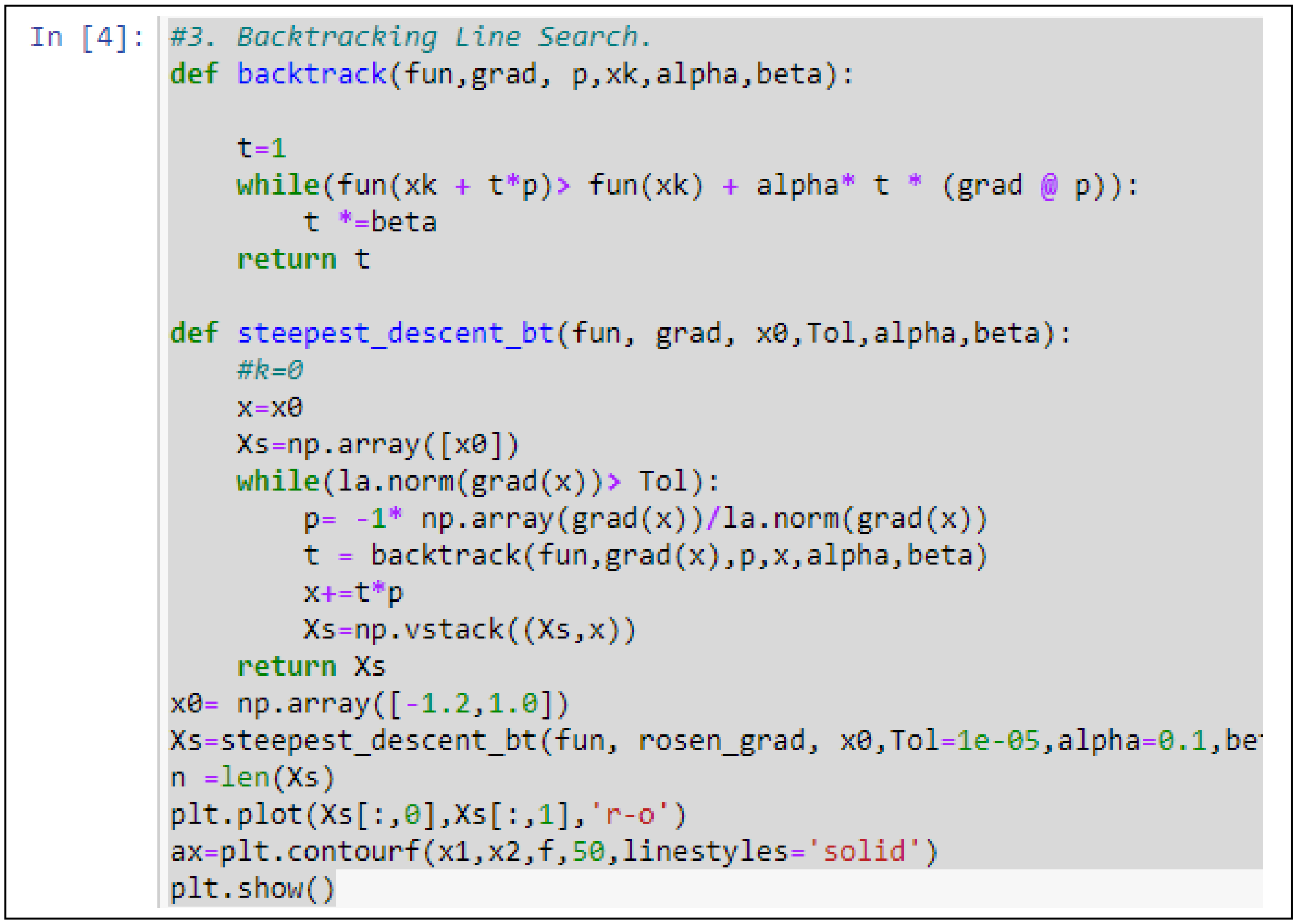
```
In [3]: #2. Gradient Computation.
        def rosen_grad(x):

            x1=x[0]
            x2=x[1]
            grad=[0,0]
            grad[1]=20*(x2-x1**2)
            grad[0]=-40*x[0]*(x[1] - x[0]**2) + 2*x[0] - 2
            return grad
```

Fig. 3 Code for generating the function gradient.

3. Backtracking Line Search. Write a routine for a backtracking line search method using the steepest descent direction. Test the performance of the algorithms on Rosenbrock's function starting at the point x0 = [−1.2, 1]t by finding the number of iterations till convergence to a gradient norm of 10−5. The header of this routine should look something like this:
def steepest_descent_bt(fun, grad, x0)

where the arguments fun and grad are Python functions that return the objective function value and its gradient respectively.



```
In [4]: #3. Backtracking Line Search.
        def backtrack(fun,grad, p,xk,alpha,beta):

            t=1
            while(fun(xk + t*p)> fun(xk) + alpha* t * (grad @ p)):
                t *=beta
            return t

        def steepest_descent_bt(fun, grad, x0,Tol,alpha,beta):
            #k=0
            x=x0
            Xs=np.array([x0])
            while(la.norm(grad(x))> Tol):
                p= -1* np.array(grad(x))/la.norm(grad(x))
                t = backtrack(fun,grad(x),p,x,alpha,beta)
                x+=t*p
                Xs=np.vstack((Xs,x))
            return Xs
        x0= np.array([-1.2,1.0])
        Xs=steepest_descent_bt(fun, rosen_grad, x0,Tol=1e-05,alpha=0.1,be
        n =len(Xs)
        plt.plot(Xs[:,0],Xs[:,1],'r-o')
        ax=plt.contourf(x1,x2,f,50,linestyles='solid')
        plt.show()
```

Fig. 4 Code for generating the Back tracking line method.

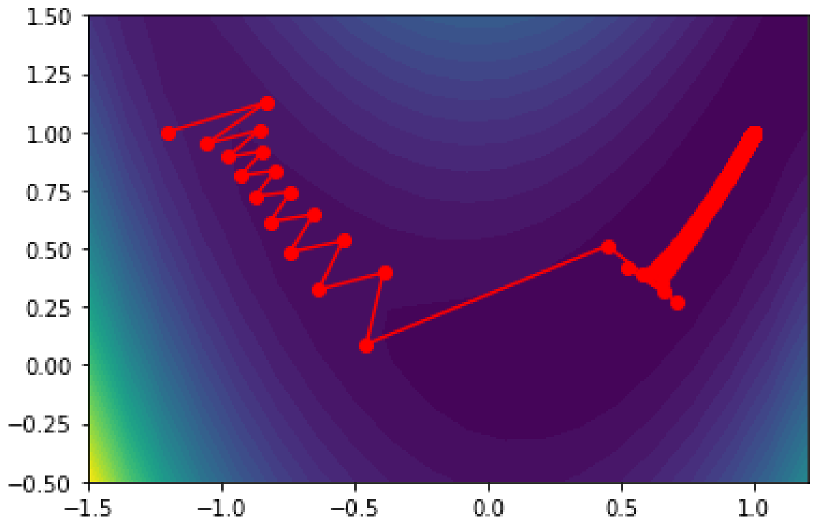


Fig. 5 Path followed to reach the minimum in the point (1,1).

4. Convergence Behavior. Assess the performance of the steepest descent method on the problem above by generating a semilog plot of the error $|f^k - f^*|$ vs. iteration count. Comment on the convergence rate.

For this exercise, we generated a triple plot with different values of alpha with same conditions of beta and initial point. Observe **Fig. 6**. The Figure shows a different convergence rate that reaches the same convergence conditions with different number of iterations. We observed that for larger alpha, the solution converges more quickly than for small values. This can be explained considering that, the larger value of alpha, the larger magnitude of the gradient of the function, giving us as consequence a huge decrement in f*. Even the convergence rate for larger alpha seems unstable due to the larger step taken between every point.

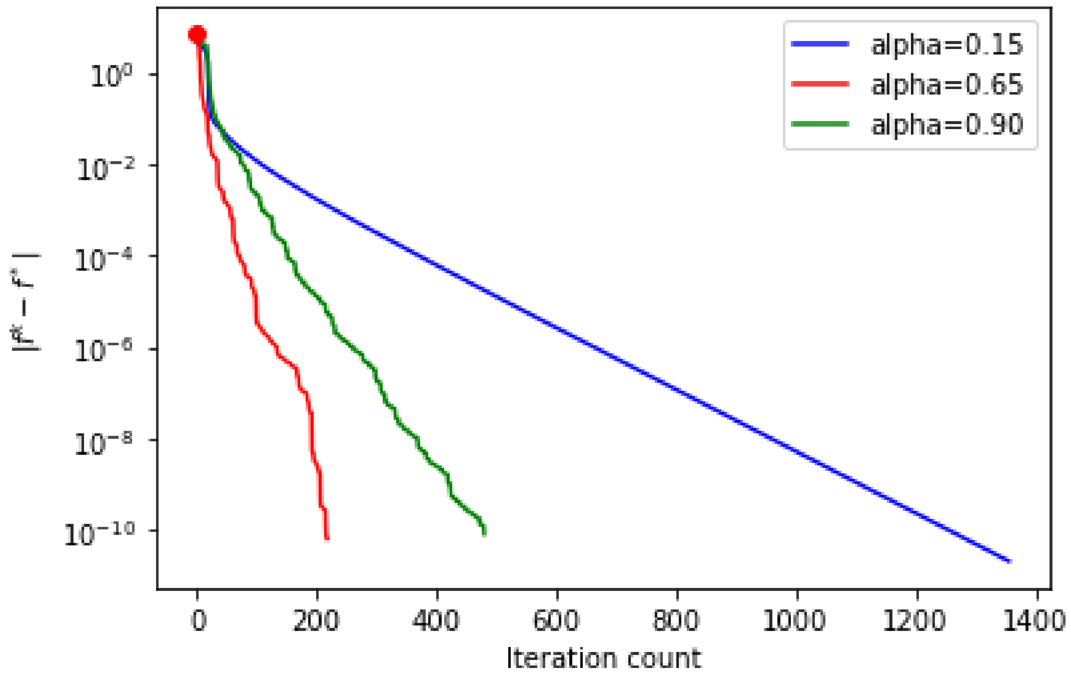


Fig. 6 Convergence rate for different values of alpha.