OPTIMIZATION FINAL PROJECT

Project Report

AN IMPLEMENTATION OF THE STEEPEST DESCENT ALGORITHM

Hoodaty Soumodeep, Kumar Akshita

December 27, 2023

Submitted under the supervision of

Dr. Sorin Mihai Grad
Professor of Optimization,
Institut Polytechnique de Paris

Acknowledgement

We would like to extend our sincere gratitude to our professor, Dr. Sorin Mihai Grad, for his continuous guidance, patience and availability throughout the duration of this project. His unwavering support has helped us complete this project successfully.

We would also like to thank the professors and staff of Ecole Polytechnique for providing us with the required resources. We have gained an invaluable experience and strengthened our knowledge of the discipline during the course of this project.

Lastly, we would also like to thank our friends and classmates with whom we could engage in fruitful discussions and gain valuable insights.

Contents

1 Introduction

The **Steepest Descent Algorithm** or the **Gradient Descent Algorithm** is a method for solving general unconstrained optimization problems, i.e., minimizing a general nonlinear function.

Consider the optimization problem

$$f : \mathbb{R}^n \to \mathbb{R} \text{ a } C^1 \text{ function}, \ \phi \neq U \subseteq \mathbb{R}^n, \ U \text{ open}$$

$$(PU) \ \inf_{x \in U} f(x)$$

Necessary First Order Optimality Condition:
When $\bar{x} \in U$ is a local optimal solution to (PU), then $\nabla f(\bar{x}) = 0$.

Sufficient First Order Optimality Condition:
When f and U are convex, then $\bar{x} \in U$ is an optimal solution to (PU) iff $\nabla f(\bar{x}) = 0$.

$d \in \mathbb{R}^n$ is called a **descent direction** of f at $x_0 \in U$ when there exists a $t_0 > 0$ such that $x_0 + td \in U$, and $f(x_0 + td) < f(x_0) \ \forall t \in (0,t_0]$. $d \in \mathbb{R}^n$ is a descent direction when $\nabla f(x_0)^T d < 0$.

A step size $t \in \mathbb{R}$ is called **efficient** for f at $x_0 \in U$ and $d \in \mathbb{R}^n$ such that $\nabla f(x_0)^T d < 0$ when $\exists c > 0$ (independent of x_0 and d such that $f(x_0 + td) \leq f(x_0) - c\left(\frac{\nabla f(x_0)^T}{\|d\|}\right)^2$.

A function is **Lipschitz continuous** if there exists a constant L such that the absolute difference between the function values at any two points is bounded by the product of the Lipschitz constant and the absolute difference between the points. Mathematically, a function f: $\mathbb{R}^n \to \mathbb{R}^m$ is Lipschitz continuous if there exists a constant L$\geq$ 0 such that for all x,y $\in \mathbb{R}^n$, the following inequality holds:

$$\|f(x) - f(y)\| \leq L\|x - y\|$$

Remark: A sufficient condition for ensuring the existence of efficient step sizes is the Lipschitz continuity of $\nabla f(x)$.

Let $x_0 \in U$. $d \in \mathbb{R}^n$ is said to be a **gradient-related descent direction** for f in X when $\exists c > 0$ such that $\nabla f(x_0)^T d \geq c\,\|\nabla f(x_0)\|\|d\|$.

When $x \in U$, $\nabla f(x) \neq 0$, then $-\nabla f(x)$ is the direction of the *steepest descent* of f at x, i.e., $d = \frac{-1}{\|\nabla f(x)\|}\nabla f(x)$ is the optimal solution to the optimization problem

$$(P) \ \inf_{\|d\|=1,\, d \in \mathbb{R}^n} \nabla f(x)^T d.$$

STEEPEST DESCENT ALGORITHM
Let U = $\mathbb{R}^n$

1. Choose a starting point $x^0 \in \mathbb{R}^n$, let K:=0
2. If $\nabla f(x^k) = 0$, STOP!

3. Determine an efficient stepsize $t^k > 0$ and let $x^{k+1} = x^k - t_k \nabla f(x^k)$
4. Let $K + K + 1$. Go to Step 1.

STEP SIZES
Let $x \in U,\ d \in \mathbb{R}^n$ such that $\nabla f(x_0)^T d < 0$.

1. The **Exact step size** $t_0 \in \mathbb{R}$ defines the optimal solution of $\inf_{t\geq 0,\, x+td\in U} f(x+td)$.
2. The **Armijo step size** $t_A \in \mathbb{R}$ is a solution of the inequality system

$$\begin{array}{rcl} f(x+td) & \leq & f(x) + \delta t \nabla f(x)^T d \\ t & \geq & -\sigma \dfrac{\nabla f(x)^T}{\|d\|^2} \\ & t \in \mathbb{R} & \end{array}$$

   for some constant $\delta \in (0,1)$ and $\sigma > 0$ independent of x and d.
   The Armijo step size can be computed using the *Armijo-Goldstein Algorithm*:

   (a) Let $\delta \in (0,1)$, $\sigma > 0$, $0 \leq \beta_1 \leq \beta_2 < 1$
   (b) Take $t_j \geq -\sigma \frac{\nabla f(x)^T d}{\|d\|^2}$ and $j = 0$.
   (c) If $f(x + t_j d) \leq f(x) + \delta t_j \nabla f(x)^T d$ then $t_A = t_j$. STOP!
   (d) Else, take $t_{j+1} \in [\beta_1 t_j, \beta_2 t_j]$. Let $j = j+1$ and go to Step c).

   Particularly, for the *steepest descent algorithm*, substituting $d = -\nabla f(x)$, the Armijo step size is a solution of the following system of inequalities:

$$\begin{array}{rcl} f(x - t\nabla f(x)) & \leq & f(x) - \delta t \|\nabla f(x)\|^2 \\ t & \geq & \sigma \\ & t \in \mathbb{R} & \end{array}$$

   Step b) of the *Armijo Goldstein Algorithm* implies choice of t such that $t \geq \sigma$ and the stopping condition step c) is $f(x - t_j \nabla f(x)) \leq f(x) - \delta t_j \|\nabla f(x)\|^2$

3. The **Wolfe-Powell step size** $t_p \in \mathbb{R}$ is a solution of the inequality system (in t):-

$$\begin{array}{rcl} f(x+td) & \leq & f(x) + \delta t \nabla f(x)^T d \\ \nabla f(x+td)^T d & \geq & \beta \nabla f(x)^T d \\ & t \in \mathbb{R} & \end{array}$$

   In particular, for the steepest descent algorithm, substituting $d = -\nabla f(x)$ in the above system of inequalities, the Wolfe-Powell step size is a solution of the system

$$\begin{array}{rcl} f(x - t\nabla f(x)) & \leq & f(x) - \delta t \|\nabla f(x)\|^2 \\ -\nabla f(x - t\nabla f(x))^T \nabla f(x) & \geq & -\beta \|\nabla f(x)\|^2 \\ & t \in \mathbb{R} & \end{array}$$

Remark: The exact step size is efficient when $S_f(f(x))$ is compact and $\nabla f(x)$ is Lipschitz continuous around x. Both the Armijo and Wolfe-Powell step sizes are efficient.

The aim of this project is to implement the Steepest Descent Algorithm to find the points of minima of:

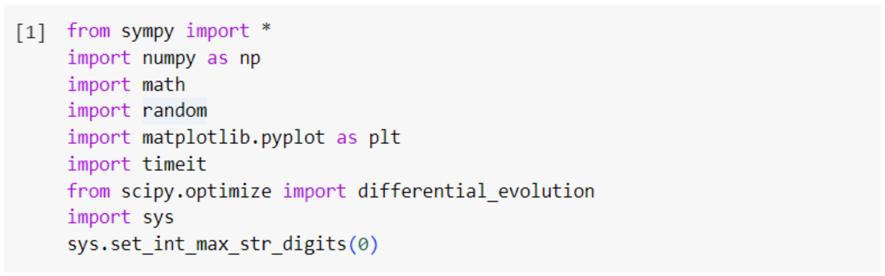
1. *Rosenbrock's Function*: $(1 - x_1)^2 + 100(x_2 - x_1^2)^2$
2. *Himmelblau's Function*: $(x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2$

with the starting points:

1. $x_0 = (0, 0)^T$
2. $x_0 = (\pi + 1, \pi - 1)^T$

2 Implementation

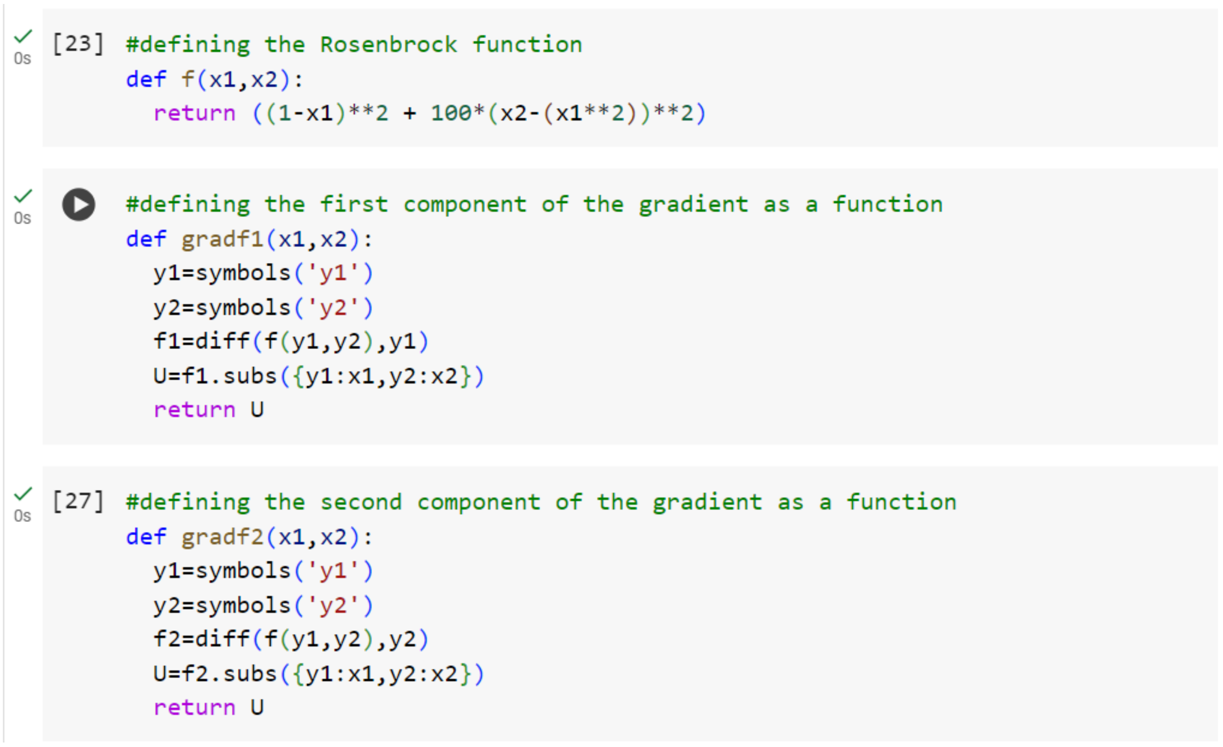
The first step is to import all the required packages. We will be using the sympy, random and the numpy packages for the computations, matplotlib package for plotting the graphs of the two functions, along with the graphs of their gradients. The timeit package helps us to print the runtime of each function. We need to import the scipy.optimize package to be able use the differential_evolution method to compute and check the critical points in the Himmelblau's function. We also import the sys package to increase the limit of the integer values, as otherwise it is difficult to get outputs in certain cases.



```
[1]  from sympy import *
     import numpy as np
     import math
     import random
     import matplotlib.pyplot as plt
     import timeit
     from scipy.optimize import differential_evolution
     import sys
     sys.set_int_max_str_digits(0)
```

2.1 Rosenbrock's Function

We begin by defining the Rosenbrock's function f with two parameters - x1 and x2. This function returns the value of the Rosenbrock's function at the point (x1,x2).
Next, we need to define the gradient function. Since we know that the Rosenbrock's function is a function from $\mathbb{R}^2$ to $\mathbb{R}$, the gradient vector $\nabla f(x1, x2)$ will be a function from $\mathbb{R}^2$ to $\mathbb{R}^2$ and will have two components. So, we use two different python functions: *gradf1*, that returns the first component of the gradient vector of f evaluated at the point (x1,x2) provided by the user, and *gradf2*, that returns the second component of the gradient vector of f evaluated at the point (x1,x2).

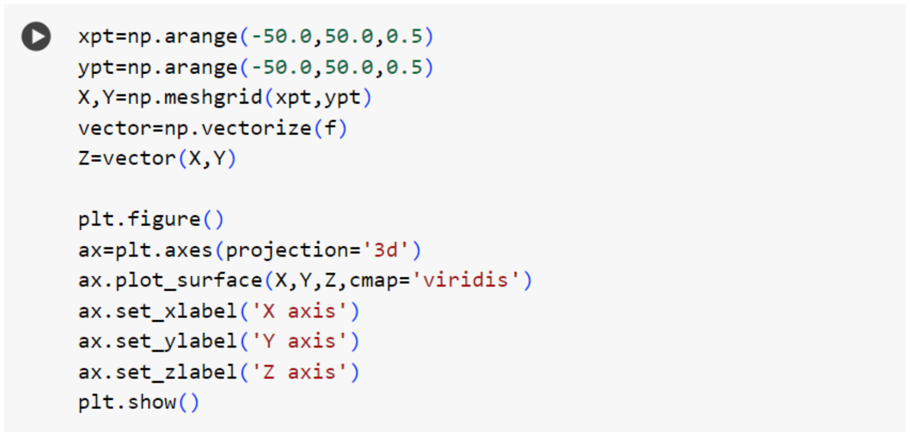


```
[23] #defining the Rosenbrock function
     def f(x1,x2):
       return ((1-x1)**2 + 100*(x2-(x1**2))**2)
```

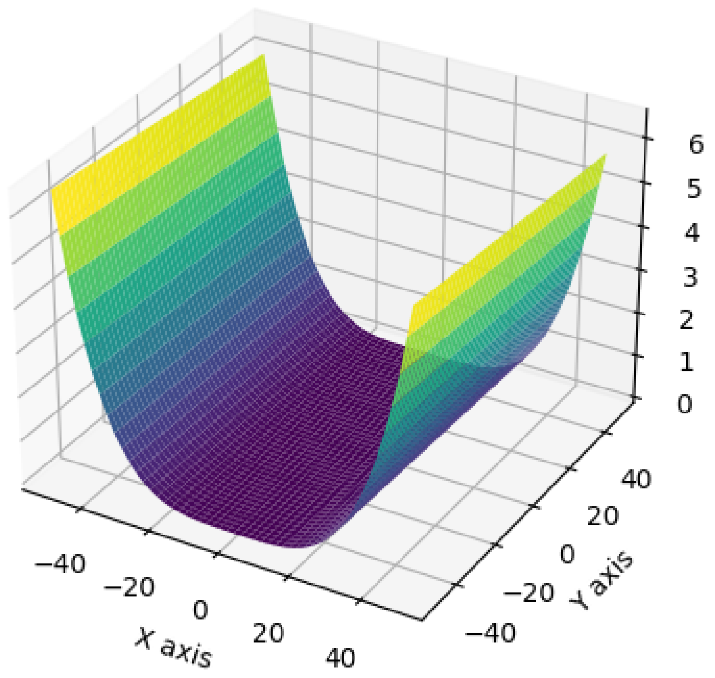
```
#defining the first component of the gradient as a function
def gradf1(x1,x2):
  y1=symbols('y1')
  y2=symbols('y2')
  f1=diff(f(y1,y2),y1)
  U=f1.subs({y1:x1,y2:x2})
  return U
```

```
[27] #defining the second component of the gradient as a function
     def gradf2(x1,x2):
       y1=symbols('y1')
       y2=symbols('y2')
       f2=diff(f(y1,y2),y2)
       U=f2.subs({y1:x1,y2:x2})
       return U
```

First of all, to visualize the behaviour of the function we plot its graph and the graph of its gradient.
For plotting the function, we use meshgrid to create a 2D grid of the set of points which would be used to plot the graph, then after the function f is vectorised, we evaluate the function on the grid and obtain the values which are stored in Z. Using matplotlib, we use the *plot_surface* function to plot the function as a 3D surface.

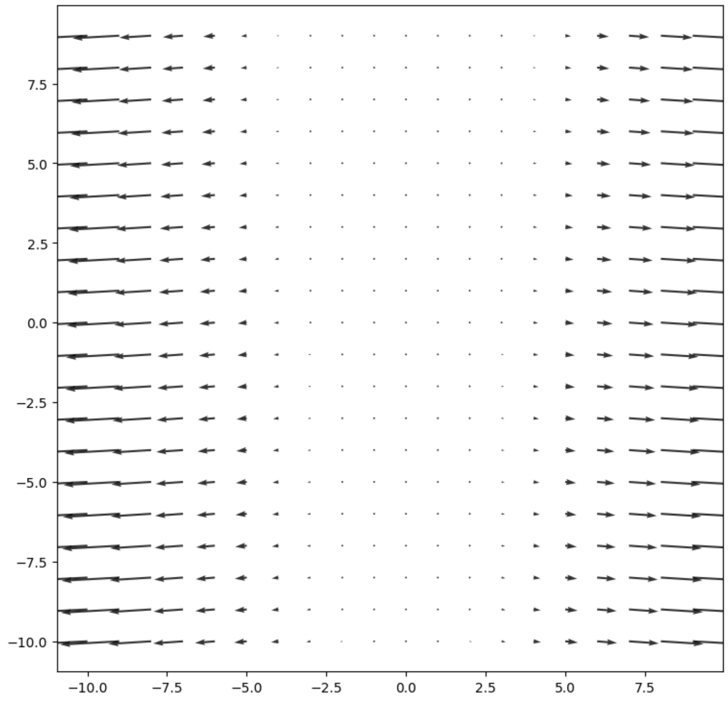


```
xpt=np.arange(-50.0,50.0,0.5)
ypt=np.arange(-50.0,50.0,0.5)
X,Y=np.meshgrid(xpt,ypt)
vector=np.vectorize(f)
Z=vector(X,Y)

plt.figure()
ax=plt.axes(projection='3d')
ax.plot_surface(X,Y,Z,cmap='viridis')
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')
plt.show()
```

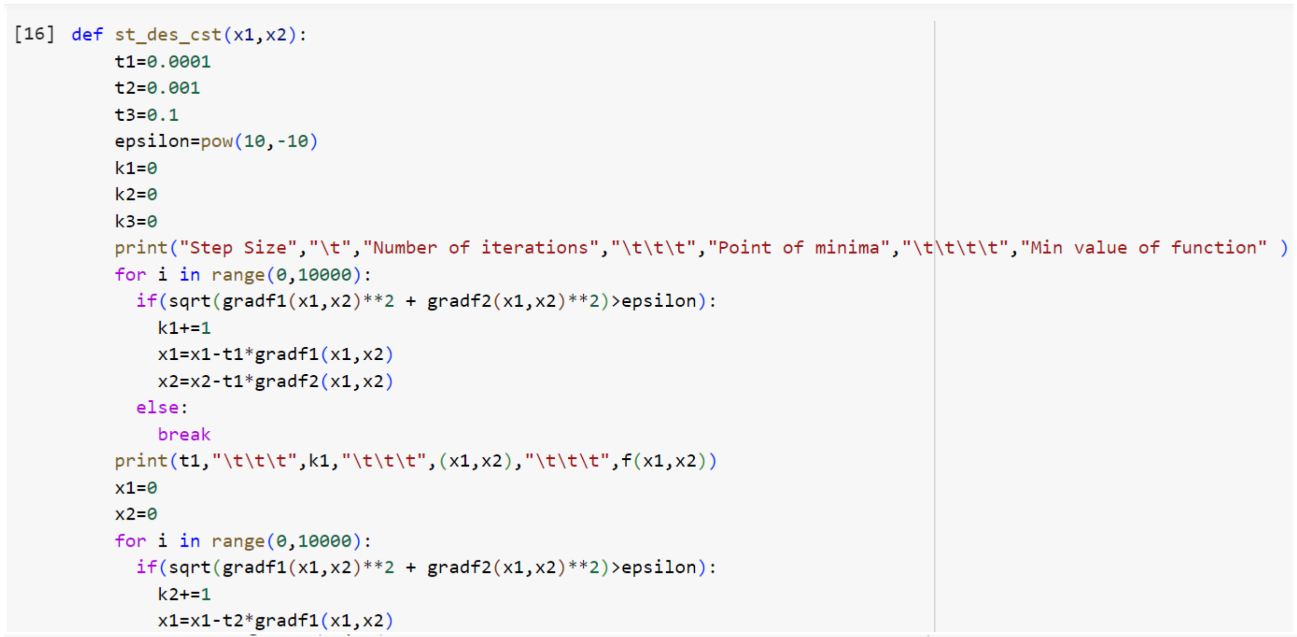


In the end, we create a quiver plot using matplotlib's quiver function. Quiver plot helps us visualize the gradient. It plots vectors depicting the direction of the gradient at each point. We again use the meshgrid function to define the set of points(2D) on which the function will be evaluated. Next, we define symbolic variables y1 and y2 and use lambdify to convert the symbolic gradient functions gradg1 and gradg2 into NumPy functions. After this, the gradients at each point are evaluated, in the 2D grid, using the NumPy functions created in the previous step. The X and Y arrays provide the grid points, and grad1 and grad2 provide the directions of the vectors at those points. alpha is set to control the transparency of the arrows. set_aspect('equal') ensures that the aspect ratio of the plot is equal. As we can see from the plot, the vectors move outwards in both the negative and positive direction on the X-axis as well as in the upward direction, which is similar to what we observe in the 3D plot of the function.

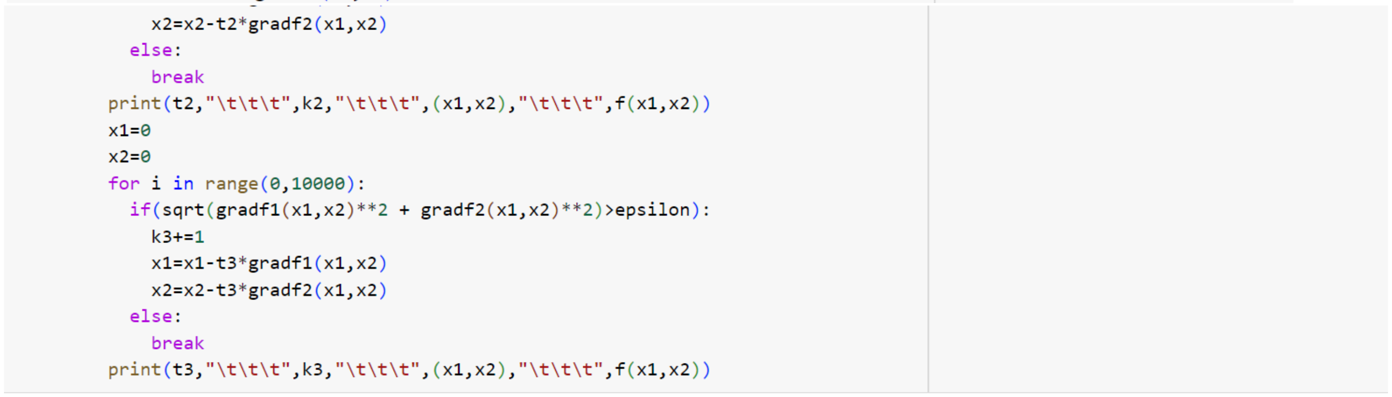
```
xpt=np.arange(-10.0,10.0,0.8)
ypt=np.arange(-10.0,10.0,0.8)
X,Y=np.meshgrid(xpt,ypt)
y1=symbols('y1')
y2=symbols('y2')
grad1_func = lambdify((y1, y2), gradf1(y1, y2), 'numpy')
grad2_func = lambdify((y1, y2), gradf2(y1, y2), 'numpy')

# Compute the gradients using NumPy functions
grad1 = grad1_func(X, Y)
grad2 = grad2_func(X, Y)

fig, ax = plt.subplots(figsize=(9, 9))
ax.quiver(X, Y, grad1, grad2,alpha=0.8)
ax.set_aspect('equal')
plt.show()
```



We now implement the steepest descent algorithm using a constant step size. For this, we define the function *st_des_cst* with the paramters x1 and x2, that are the coordinates of the starting point x_0. We wish to compare different constant step sizes and see the behaviour of the algorithm and the obtained points of minima. We have chosen the constant step sizes of t1=0.0001, t2=0.01 and t3=0.1. For the steepest descent algorithm, we look for the point X^k where ∇f(X^k)=0. However, in practice, it is difficult to achieve the condition when ∇f=0 exactly. So, we consider an admissible error of $\epsilon = 10^{-10}$, i.e we look for an X^k such that $\|\nabla f(X^k)\| < \epsilon$. Instead of a while loop (in which case the computation limit gets exceeded), we use a for loop in order to limit the number of iterations to 10,000. The *if* condition then tests for the desired stopping condition, i.e if the stopping condition has not been achieved (if $\nabla f(x) > \epsilon$), it updates the values of x1 and x2 according to the updation criteria in the algorithm. As soon as the stopping condition is achieved, we break the loop and print the obtained point of minima and function value at this point. This is done for all the three constant step sizes, one at a time.

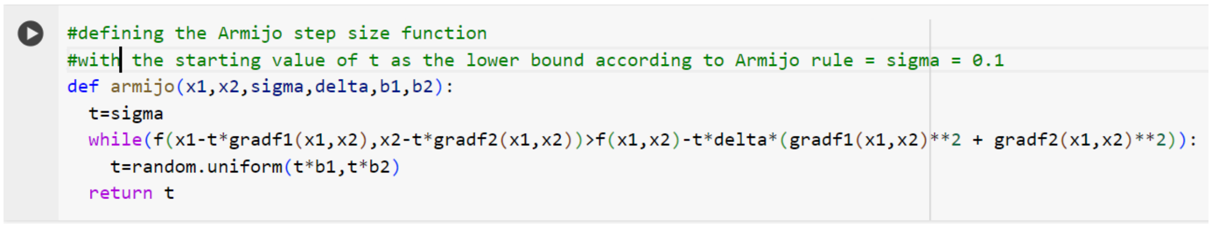


```
[16] def st_des_cst(x1,x2):
       t1=0.0001
       t2=0.001
       t3=0.1
       epsilon=pow(10,-10)
       k1=0
       k2=0
       k3=0
       print("Step Size","\t","Number of iterations","\t\t\t","Point of minima","\t\t\t\t","Min value of function" )
       for i in range(0,10000):
         if(sqrt(gradf1(x1,x2)**2 + gradf2(x1,x2)**2)>epsilon):
           k1+=1
           x1=x1-t1*gradf1(x1,x2)
           x2=x2-t1*gradf2(x1,x2)
         else:
           break
       print(t1,"\t\t\t",k1,"\t\t\t",(x1,x2),"\t\t\t",f(x1,x2))
       x1=0
       x2=0
       for i in range(0,10000):
         if(sqrt(gradf1(x1,x2)**2 + gradf2(x1,x2)**2)>epsilon):
           k2+=1
           x1=x1-t2*gradf1(x1,x2)
```



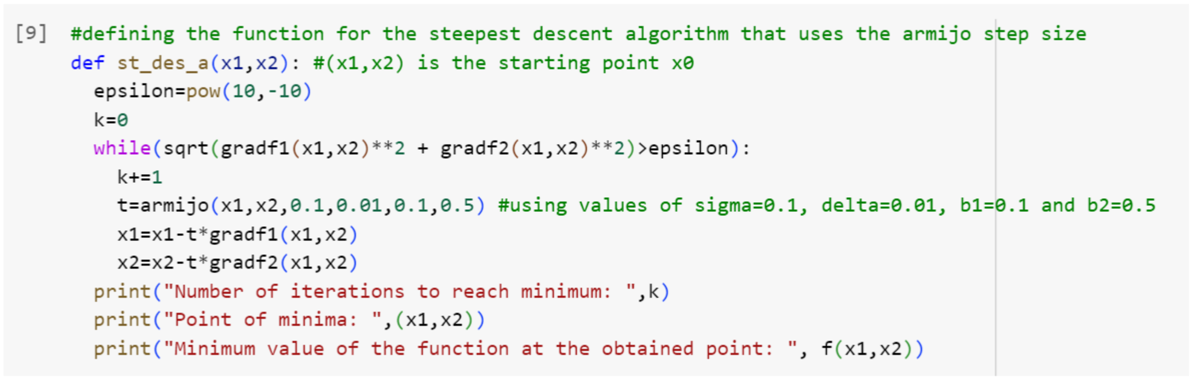
```
           x2=x2-t2*gradf2(x1,x2)
         else:
           break
       print(t2,"\t\t\t",k2,"\t\t\t",(x1,x2),"\t\t\t",f(x1,x2))
       x1=0
       x2=0
       for i in range(0,10000):
         if(sqrt(gradf1(x1,x2)**2 + gradf2(x1,x2)**2)>epsilon):
           k3+=1
           x1=x1-t3*gradf1(x1,x2)
           x2=x2-t3*gradf2(x1,x2)
         else:
           break
       print(t3,"\t\t\t",k3,"\t\t\t",(x1,x2),"\t\t\t",f(x1,x2))
```

We now define the *armijo* function with the parameters x1, x2, sigma, delta, b1 and b2. This function evaluates and returns the armijo step size for the point (x1,x2), using the values of sigma, delta, b1 and b2 provided as arguments. From the Goldstein Armijo algorithm, we know that σ gives a lower bound for the value of the step size t. We have initialized t with a value equal to the lower bound σ. The while loop of the function then tests the second condition of the Goldstein Armijo algorithm, and when the condition is not satisfied, updates t to a new value chosen randomly from the interval $[t\beta 1, t\beta 2]$.

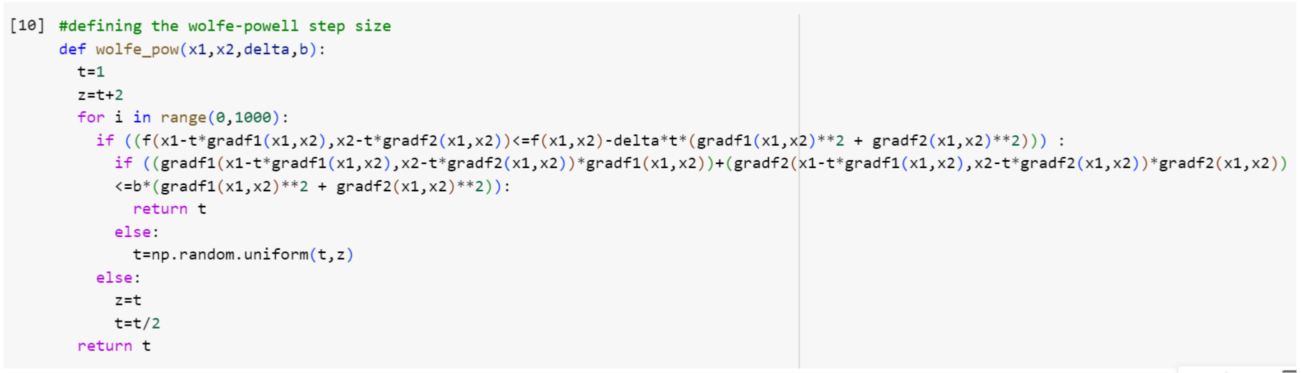


```
#defining the Armijo step size function
#with the starting value of t as the lower bound according to Armijo rule = sigma = 0.1
def armijo(x1,x2,sigma,delta,b1,b2):
  t=sigma
  while(f(x1-t*gradf1(x1,x2),x2-t*gradf2(x1,x2))>f(x1,x2)-t*delta*(gradf1(x1,x2)**2 + gradf2(x1,x2)**2)):
    t=random.uniform(t*b1,t*b2)
  return t
```

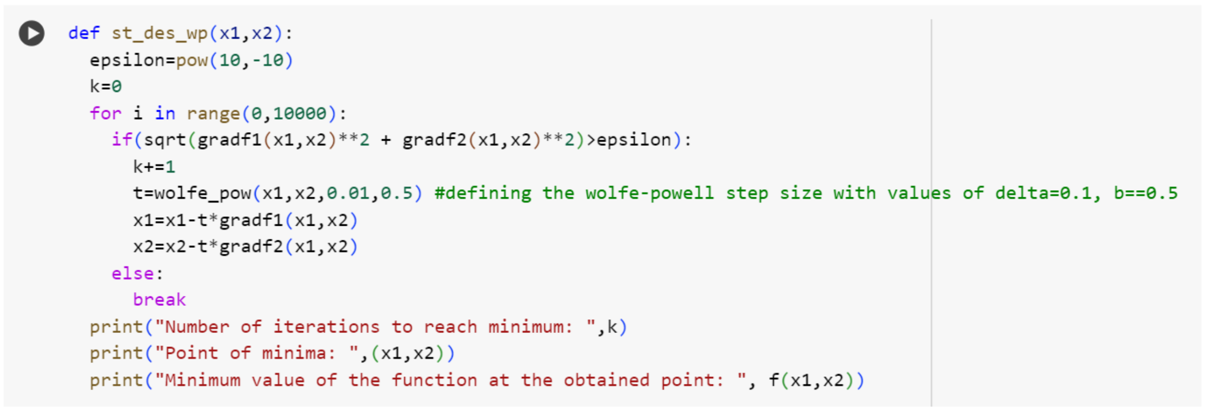
Next, we define a function for the steepest descent algorithm *st_des_a*(x1,x2), that uses the armijo step size to find the point of minima of the Rosenbrock function f. This function takes the parameters x1 and x2, that are the coordinates of the starting point x_0. We look for the point X^k where ∇f(X^k)=0. However, in practice, it is difficult to achieve the condition when ∇f=0 exactly. So, we consider an admissible error of $\epsilon = 10^{-10}$, i.e we look for an X^k such that $\|\nabla f(X^k)\| < \epsilon$. The while loop then tests for the condition $\|\nabla f(X^k)\| > epsilon$, i.e the case when the stopping condition has not been achieved. It runs and updates the coordinates x1 and x2 until the gradient of the obtained point is within the admissible error. For the Armijo step size, we have used the values of $\sigma = 0.1$, δ=delta=0.01, and 0 < β1=b1=0.1 < β2=b2=0.5 < 1. Finally, the function prints the minima and the minimum value of the function.


```
[9]  #defining the function for the steepest descent algorithm that uses the armijo step size
     def st_des_a(x1,x2): #(x1,x2) is the starting point x0
       epsilon=pow(10,-10)
       k=0
       while(sqrt(gradf1(x1,x2)**2 + gradf2(x1,x2)**2)>epsilon):
         k+=1
         t=armijo(x1,x2,0.1,0.01,0.1,0.5) #using values of sigma=0.1, delta=0.01, b1=0.1 and b2=0.5
         x1=x1-t*gradf1(x1,x2)
         x2=x2-t*gradf2(x1,x2)
       print("Number of iterations to reach minimum: ",k)
       print("Point of minima: ",(x1,x2))
       print("Minimum value of the function at the obtained point: ", f(x1,x2))
```

Now, we wish to implement the algorithm using the Wolfe-Powell step size. We define a function *wolfe_pow* that evaluates the Wolfe-Powell step size. This function has the parameters x1, x2 that are the coordinates of the starting point x_0. Keeping in mind the computation time, we need to limit the number of iterations to 1000, and in order to do so we have used a *for* loop. The first *if* statement in the loop tests the first Wolfe-Powell condition while the second one tests the second Wolfe-Powell condition. If both the conditions are satisfied, the current value of t is returned. If the second condition is not satisfied while the first one is satisfied, we realise that we need to increase t and we assign it a random value in the interval (t,z), where z was assigned an initial value of t+2. The second else statement is for the case when the first condition is not satisfied, in which case we need to reduce t, and we update it to the new value t/2. Finally, the function returns the value of the Wolfe-Powell step size.


```
[10] #defining the wolfe-powell step size
     def wolfe_pow(x1,x2,delta,b):
       t=1
       z=t+2
       for i in range(0,1000):
         if ((f(x1-t*gradf1(x1,x2),x2-t*gradf2(x1,x2))<=f(x1,x2)-delta*t*(gradf1(x1,x2)**2 + gradf2(x1,x2)**2))) :
           if ((gradf1(x1-t*gradf1(x1,x2),x2-t*gradf2(x1,x2))*gradf1(x1,x2))+(gradf2(x1-t*gradf1(x1,x2),x2-t*gradf2(x1,x2))*gradf2(x1,x2))
           <=b*(gradf1(x1,x2)**2 + gradf2(x1,x2)**2)):
             return t
           else:
             t=np.random.uniform(t,z)
         else:
           z=t
           t=t/2
       return t
```

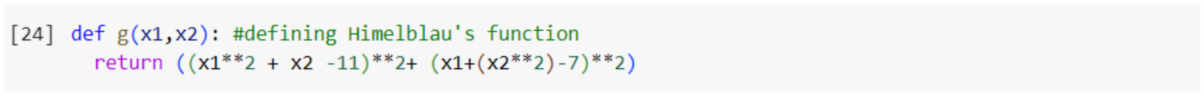
We have then defined a function *st_des _wp* for the Steepest Descent Algorithm that uses the Wolfe-Powell step size. This function has the parameters x1 and x2, that are the coordinates of the starting point x_0. The code is the same as that for *st_des _a*, except that we have changed the value of t to the Wolfe-Powell step size and instead of a while loop we have used a for loop with a range of 10,000. This is because in this case, a simple while loop keeps running indefinitely and the computation limit gets exceeded. So, we have limited the number of iterations to 10,000.



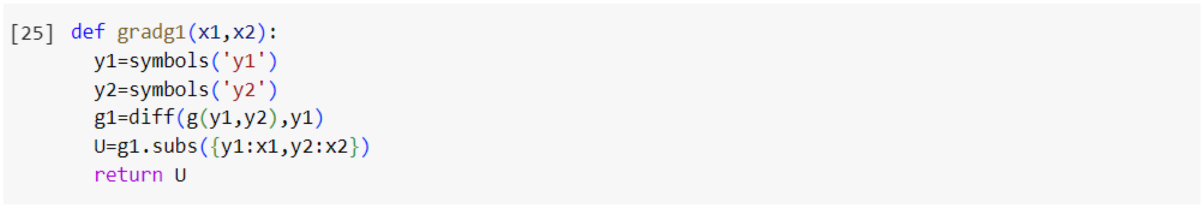
```
def st_des_wp(x1,x2):
  epsilon=pow(10,-10)
  k=0
  for i in range(0,10000):
    if(sqrt(gradf1(x1,x2)**2 + gradf2(x1,x2)**2)>epsilon):
      k+=1
      t=wolfe_pow(x1,x2,0.01,0.5) #defining the wolfe-powell step size with values of delta=0.1, b==0.5
      x1=x1-t*gradf1(x1,x2)
      x2=x2-t*gradf2(x1,x2)
    else:
      break
  print("Number of iterations to reach minimum: ",k)
  print("Point of minima: ",(x1,x2))
  print("Minimum value of the function at the obtained point: ", f(x1,x2))
```

2.2 Himmelblau's Function

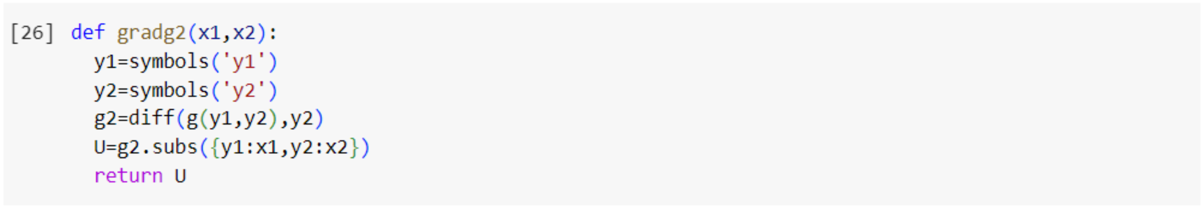
We begin by defining the Himmelblau's function f with two parameters - x1 and x2. This function returns the value of the Himmelblau's function at the point (x1,x2).
Next, we need to define the gradient function. Since we know that the Himmelblau's function is a function from $\mathbb{R}^2$ to $\mathbb{R}$, the gradient vector $\nabla g(x1, x2)$ will be a function from $\mathbb{R}^2$ to $\mathbb{R}^2$ and will have two components. So, we use two different python functions: *gradg1*, that returns the first component of the gradient vector of g evaluated at the point (x1,x2) provided by the user, and *gradg2*, that returns the second component of the gradient vector of g evaluated at the point (x1,x2).



```
[24] def g(x1,x2): #defining Himelblau's function
       return ((x1**2 + x2 -11)**2+ (x1+(x2**2)-7)**2)
```

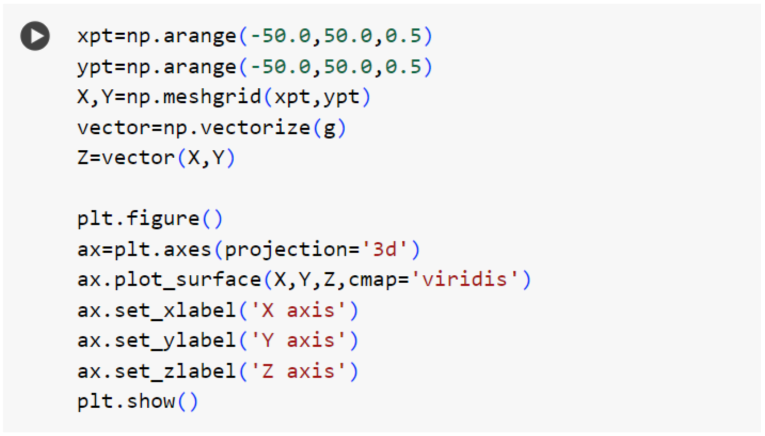


```
[25] def gradg1(x1,x2):
       y1=symbols('y1')
       y2=symbols('y2')
       g1=diff(g(y1,y2),y1)
       U=g1.subs({y1:x1,y2:x2})
       return U
```

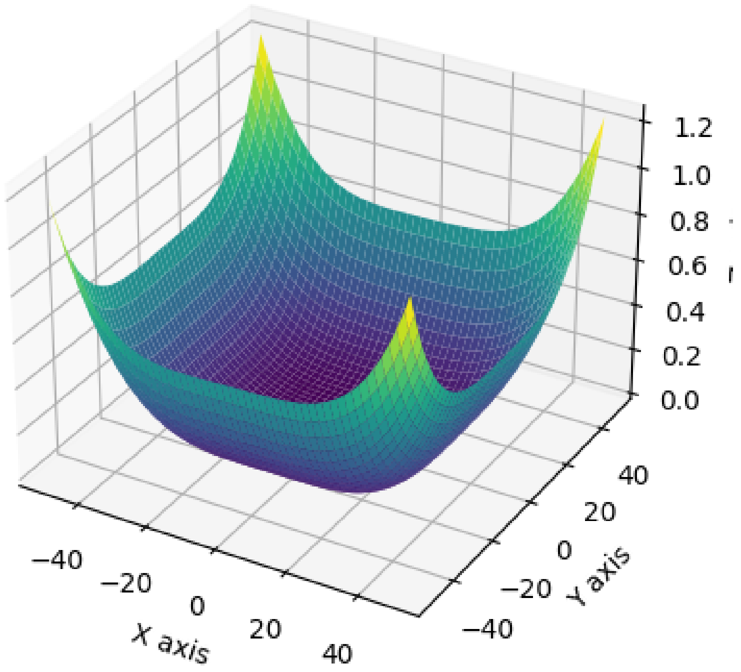


```
[26] def gradg2(x1,x2):
       y1=symbols('y1')
       y2=symbols('y2')
       g2=diff(g(y1,y2),y2)
       U=g2.subs({y1:x1,y2:x2})
       return U
```

First of all, to visualize the behaviour of the function we plot its graph and the graph of its gradient.
For the graph of the function, we use meshgrid to create a 2D grid of the set of points which would be used to plot the graph, then after the function f is vectorised, after which we evaluate the function on the grid and obtain the values which are stored in Z. Using mathplotlib, we use the *plot_surface* function to 3D plot the function.

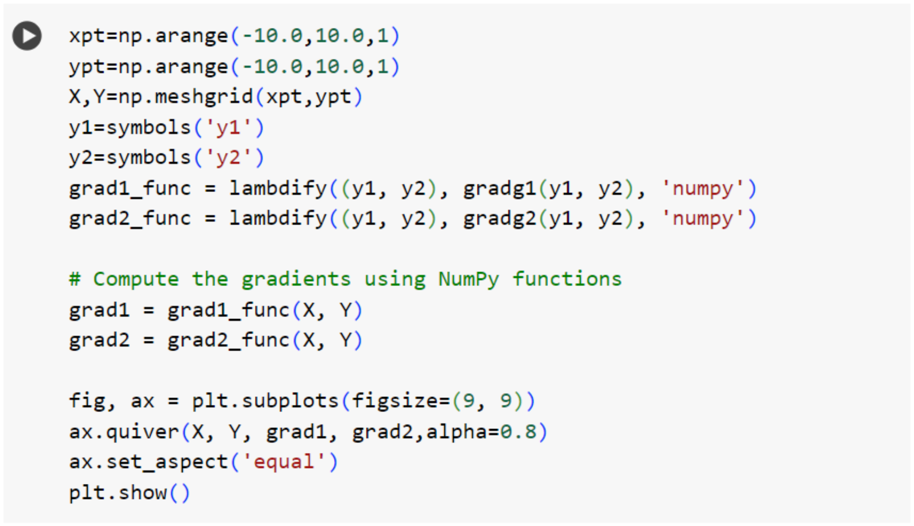


```
xpt=np.arange(-50.0,50.0,0.5)
ypt=np.arange(-50.0,50.0,0.5)
X,Y=np.meshgrid(xpt,ypt)
vector=np.vectorize(g)
Z=vector(X,Y)

plt.figure()
ax=plt.axes(projection='3d')
ax.plot_surface(X,Y,Z,cmap='viridis')
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')
plt.show()
```

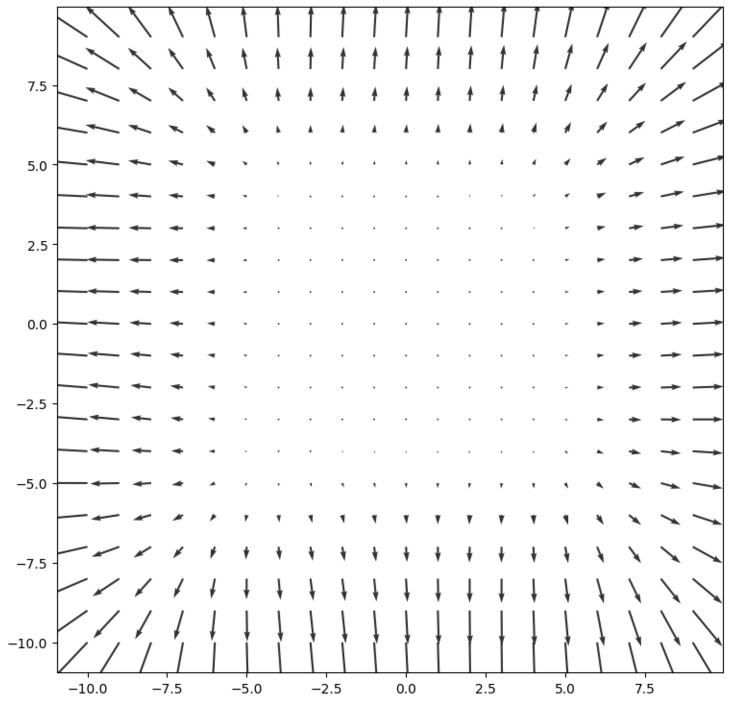


Then for the gradient of g, we create a quiver plot using Matplotlib's quiver function. Quiver plot helps us visualize the gradient. It plots vectors depicting the direction of the gradient at each point. We again define the meshgrid function to define the set of points(2D) on which the function will be evaluated. Next, we define symbolic variables y1 and y2 and use lambdify to convert the symbolic gradient functions gradg1 and gradg2 into NumPy functions. After which, the gradients at each point are evaluated, in the 2D grid, using the

NumPy functions created in the previous step. The X and Y arrays provide the grid points, and grad1 and grad2 provide the components of the vectors at those points. alpha is set to control the transparency of the arrows. *set_aspect*('equal') ensures that the aspect ratio of the plot is equal. As we can see from the plot, the vectors move outwards in both the negative and positive direction on the X-axis and Y-axis as well as outwards from the plane, which is similar to what we observe in the 3D plot of the function.

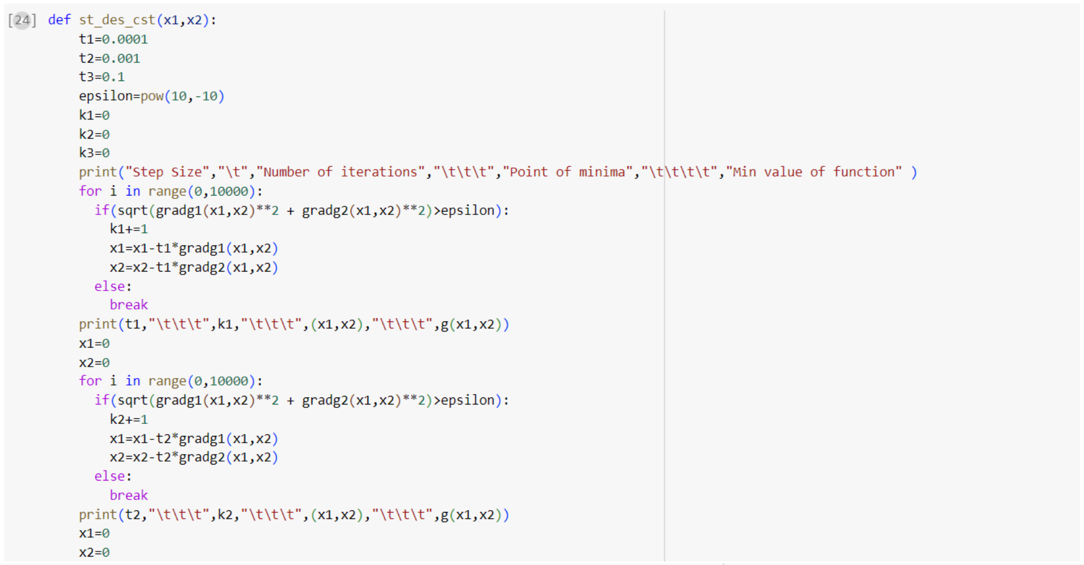


```
xpt=np.arange(-10.0,10.0,1)
ypt=np.arange(-10.0,10.0,1)
X,Y=np.meshgrid(xpt,ypt)
y1=symbols('y1')
y2=symbols('y2')
grad1_func = lambdify((y1, y2), gradg1(y1, y2), 'numpy')
grad2_func = lambdify((y1, y2), gradg2(y1, y2), 'numpy')

# Compute the gradients using NumPy functions
grad1 = grad1_func(X, Y)
grad2 = grad2_func(X, Y)

fig, ax = plt.subplots(figsize=(9, 9))
ax.quiver(X, Y, grad1, grad2,alpha=0.8)
ax.set_aspect('equal')
plt.show()
```

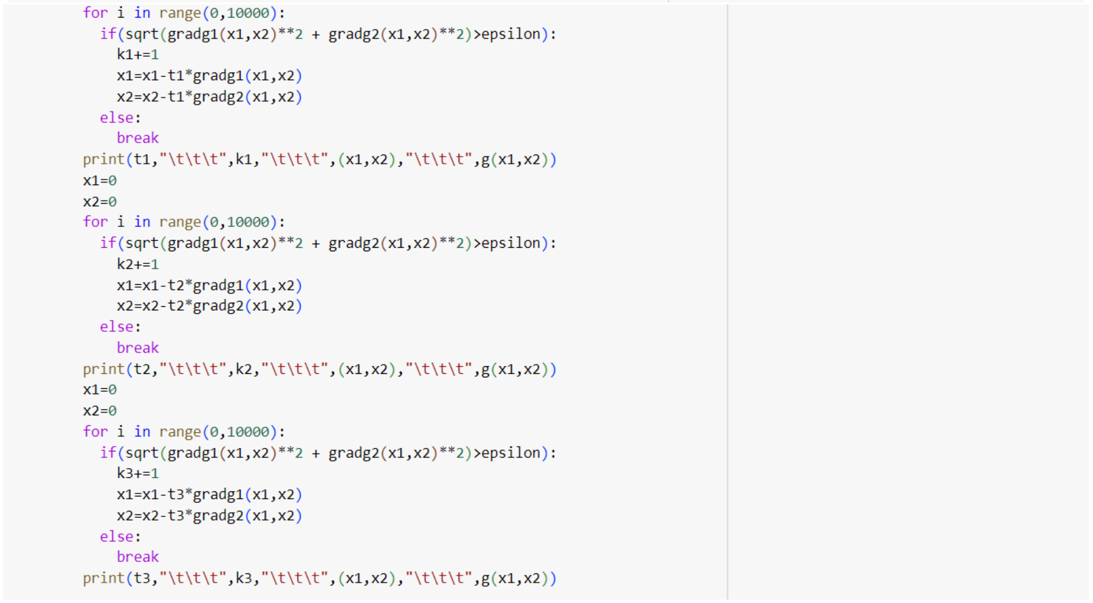


We now implement the steepest descent algorithm using a constant step size. For this, we define the function *st_des_cst* with the paramters x1 and x2, that are the coordinates of the starting point x_0. We wish to compare different constant step sizes and see the behaviour of the algorithm and the obtained points of minima. We have chosen the constant step sizes of t1=0.0001, t2=0.01 and t3=0.1. For the steepest descent algorithm, we look for the point X^k

where $\nabla g(\mathrm{X}^k)$=0. However, in practice, it is difficult to achieve the condition when ∇f=0 exactly. So, we consider an admissible error of $\epsilon = 10^{-10}$, i.e we look for an X^k such that $\|\nabla g(X^k)\| < \epsilon$. Instead of a while loop (in which case the computation limit gets exceeded), we use a for loop in order to limit the number of iterations to 10,000. The *if* condition then tests for the desired stopping condition, i.e if the stopping condition has not been achieved (if $\nabla g(x) > \epsilon$), it updates the values of x1 and x2 according to the updation criteria in the algorithm. As soon as the stopping condition is achieved, we break the loop and print the obtained point of minima and function value at this point. This is done for all the three constant step sizes, one at a time.

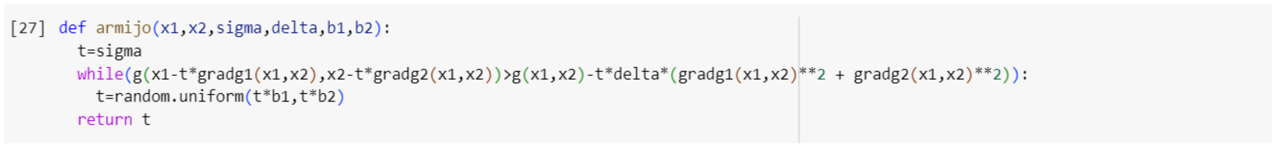


```
[24] def st_des_cst(x1,x2):
       t1=0.0001
       t2=0.001
       t3=0.1
       epsilon=pow(10,-10)
       k1=0
       k2=0
       k3=0
       print("Step Size","\t","Number of iterations","\t\t\t","Point of minima","\t\t\t","Min value of function" )
       for i in range(0,10000):
         if(sqrt(gradg1(x1,x2)**2 + gradg2(x1,x2)**2)>epsilon):
           k1+=1
           x1=x1-t1*gradg1(x1,x2)
           x2=x2-t1*gradg2(x1,x2)
         else:
           break
       print(t1,"\t\t\t",k1,"\t\t\t",(x1,x2),"\t\t\t",g(x1,x2))
       x1=0
       x2=0
       for i in range(0,10000):
         if(sqrt(gradg1(x1,x2)**2 + gradg2(x1,x2)**2)>epsilon):
           k2+=1
           x1=x1-t2*gradg1(x1,x2)
           x2=x2-t2*gradg2(x1,x2)
         else:
           break
       print(t2,"\t\t\t",k2,"\t\t\t",(x1,x2),"\t\t\t",g(x1,x2))
       x1=0
       x2=0
```



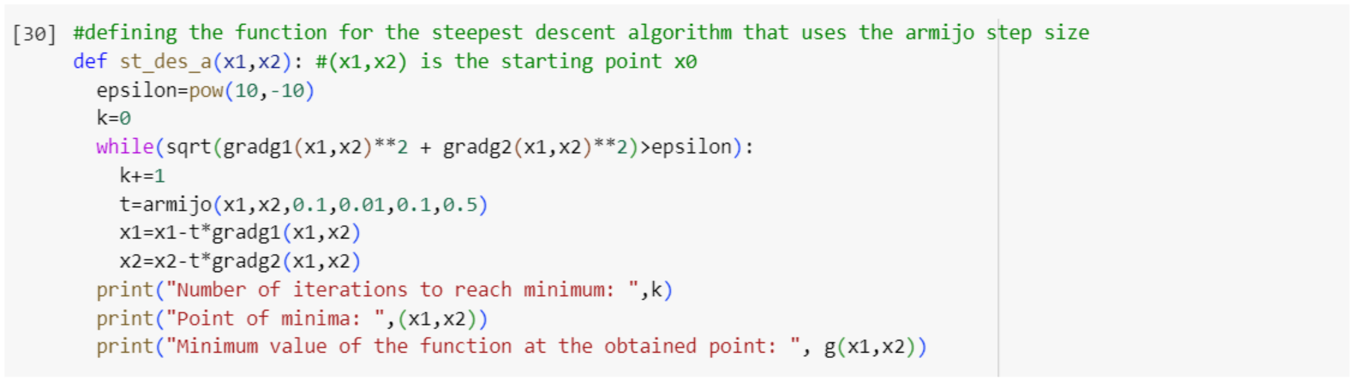
```
       for i in range(0,10000):
         if(sqrt(gradg1(x1,x2)**2 + gradg2(x1,x2)**2)>epsilon):
           k1+=1
           x1=x1-t1*gradg1(x1,x2)
           x2=x2-t1*gradg2(x1,x2)
         else:
           break
       print(t1,"\t\t\t",k1,"\t\t\t",(x1,x2),"\t\t\t",g(x1,x2))
       x1=0
       x2=0
       for i in range(0,10000):
         if(sqrt(gradg1(x1,x2)**2 + gradg2(x1,x2)**2)>epsilon):
           k2+=1
           x1=x1-t2*gradg1(x1,x2)
           x2=x2-t2*gradg2(x1,x2)
         else:
           break
       print(t2,"\t\t\t",k2,"\t\t\t",(x1,x2),"\t\t\t",g(x1,x2))
       x1=0
       x2=0
       for i in range(0,10000):
         if(sqrt(gradg1(x1,x2)**2 + gradg2(x1,x2)**2)>epsilon):
           k3+=1
           x1=x1-t3*gradg1(x1,x2)
           x2=x2-t3*gradg2(x1,x2)
         else:
           break
       print(t3,"\t\t\t",k3,"\t\t\t",(x1,x2),"\t\t\t",g(x1,x2))
```

We now define the *Armijo* function with the parameters x1, x2, sigma, delta, b1 and b2. This function evaluates and returns the armijo step size for the point (x1,x2), using the values of sigma, delta, b1 and b2 provided as arguments. From the Goldstein Armijo algorithm, we know that σ gives a lower bound for the value of the step size t. We have initialized t with a value equal to the lower bound σ. The while loop of the function then tests the second condition of the Goldstein Armijo algorithm, and when the condition is not satisfied, updates t to a new value chosen randomly from the interval $[t\beta1, t\beta2]$.



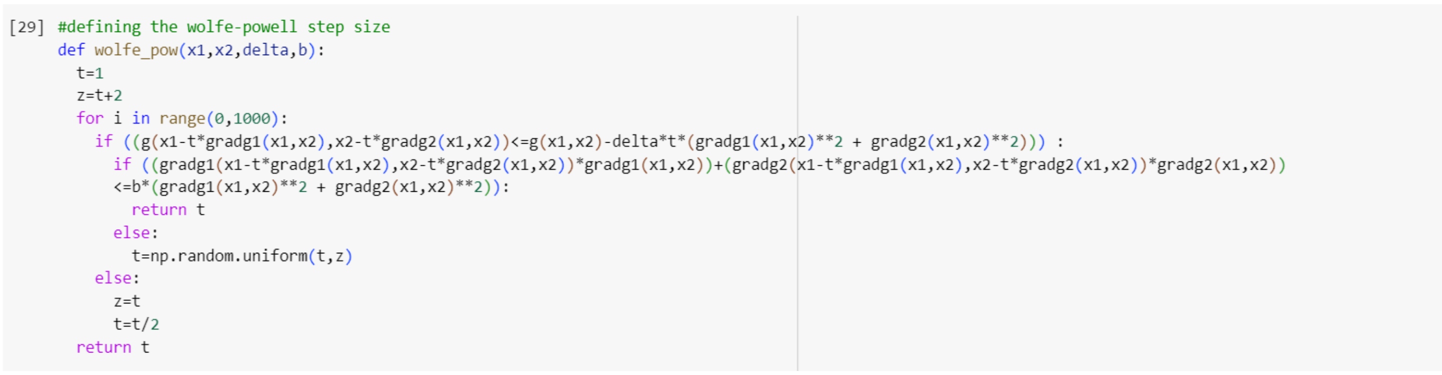
```
[27] def armijo(x1,x2,sigma,delta,b1,b2):
       t=sigma
       while(g(x1-t*gradg1(x1,x2),x2-t*gradg2(x1,x2))>g(x1,x2)-t*delta*(gradg1(x1,x2)**2 + gradg2(x1,x2)**2)):
         t=random.uniform(t*b1,t*b2)
       return t
```

Next, we define a function for the steepest descent algorithm st_des_a(x1,x2), that uses the armijo step size to find the point of minima of the Himmelblau function g. This function takes the parameters x1 and x2, that are the coordinates of the starting point x_0. We look for the point X^k where ∇g(X^k)=0. However, in practice, it is difficult to achieve the condition when ∇g=0 exactly. So, we consider an admissible error of $\epsilon = 10^{-10}$, i.e we look for an X^k such that $\|\nabla g(X^k)\| < \epsilon$. The while loop then tests for the condition $\|\nabla g(X^k)\| > epsilon$, i.e the case when the stopping condition has not been achieved. It runs and updates the coordinates x1 and x2 until the gradient of the obtained point is within the admissible error. For the Armijo step size, we have used the values of $\sigma = 0.1$, δ=delta=0.01, and 0 < $\beta1$=b1=0.1 < $\beta2$=b2=0.5 < 1. Finally, the function prints the minima and the minimum value of the function.

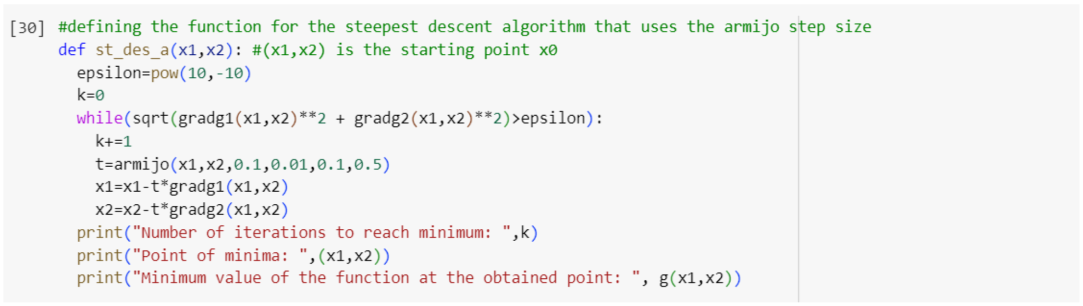


```
[30] #defining the function for the steepest descent algorithm that uses the armijo step size
     def st_des_a(x1,x2): #(x1,x2) is the starting point x0
       epsilon=pow(10,-10)
       k=0
       while(sqrt(gradg1(x1,x2)**2 + gradg2(x1,x2)**2)>epsilon):
         k+=1
         t=armijo(x1,x2,0.1,0.01,0.1,0.5)
         x1=x1-t*gradg1(x1,x2)
         x2=x2-t*gradg2(x1,x2)
       print("Number of iterations to reach minimum: ",k)
       print("Point of minima: ",(x1,x2))
       print("Minimum value of the function at the obtained point: ", g(x1,x2))
```

Finally, we wish to use the Wolfe-Powell step size for the steepest descent algorithm. To do so, we define a function *wolfe_pow* that evaluates the Wolfe-Powell step size. This function has the parameters x1, x2 that are the coordinates of the starting point x_0. Keeping in mind the computation time, we need to limit the number of iterations to 1000, and in order to do so we have used a *for* loop. The first *if* statement in the loop tests the first Wolfe-Powell condition while the second one tests the second Wolfe-Powell condition. If both the conditions are satisfied, the current value of t is returned. If the second condition is not satisfied while the first one is satisfied, we realise that we need to increase t and we assign it a random value in the interval (t,z), where z was assigned an initial value of t+2. The second else statement is for the case when the first condition is not satisfied, in which case we need to reduce t, and we update it to the new value t/2. Finally, the function returns the value of the Wolfe-Powell step size.


```
[29] #defining the wolfe-powell step size
     def wolfe_pow(x1,x2,delta,b):
       t=1
       z=t+2
       for i in range(0,1000):
         if ((g(x1-t*gradg1(x1,x2),x2-t*gradg2(x1,x2))<=g(x1,x2)-delta*t*(gradg1(x1,x2)**2 + gradg2(x1,x2)**2))) :
           if ((gradg1(x1-t*gradg1(x1,x2),x2-t*gradg2(x1,x2))*gradg1(x1,x2))+(gradg2(x1-t*gradg1(x1,x2),x2-t*gradg2(x1,x2))*gradg2(x1,x2))
           <=b*(gradg1(x1,x2)**2 + gradg2(x1,x2)**2)):
             return t
           else:
             t=np.random.uniform(t,z)
         else:
           z=t
           t=t/2
       return t
```

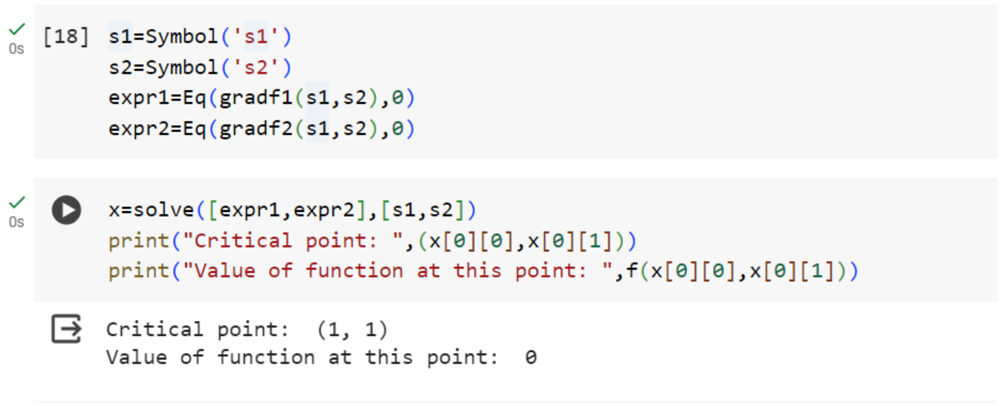
We have then defined a function *st_des _wp* for the Steepest Descent Algorithm that uses the Wolfe-Powell step size. This function has the parameters x1 and x2, that are the coordinates of the starting point x_0. The code is the same as that for *st_des _a*, except that we have changed the value of t to the Wolfe-Powell step size and instead of a while loop we have used a for loop with a range of 10,000. This is because in this case, a simple while loop keeps running indefinitely and the computation limit gets exceeded. So, we have limited the number of iterations to 10,000.


```
[30] #defining the function for the steepest descent algorithm that uses the armijo step size
     def st_des_a(x1,x2): #(x1,x2) is the starting point x0
       epsilon=pow(10,-10)
       k=0
       while(sqrt(gradg1(x1,x2)**2 + gradg2(x1,x2)**2)>epsilon):
         k+=1
         t=armijo(x1,x2,0.1,0.01,0.1,0.5)
         x1=x1-t*gradg1(x1,x2)
         x2=x2-t*gradg2(x1,x2)
       print("Number of iterations to reach minimum: ",k)
       print("Point of minima: ",(x1,x2))
       print("Minimum value of the function at the obtained point: ", g(x1,x2))
```

3 Results and Conclusion

3.1 Rosenbrock's Function

In order to check the accuracy and efficiency of our implementation of the steepest descent algorithm, we find the critical points of the Rosenbrock function, i.e the point x=(x1,x2) for which $\nabla f(x) = 0$. We find that the only critical point of the function is (1,1), which is hence the minima of the function (since the Rosenbrock function is convex, $\nabla f(x) = 0$ is a sufficient condition for a minima). The minimum value of the function is 0.

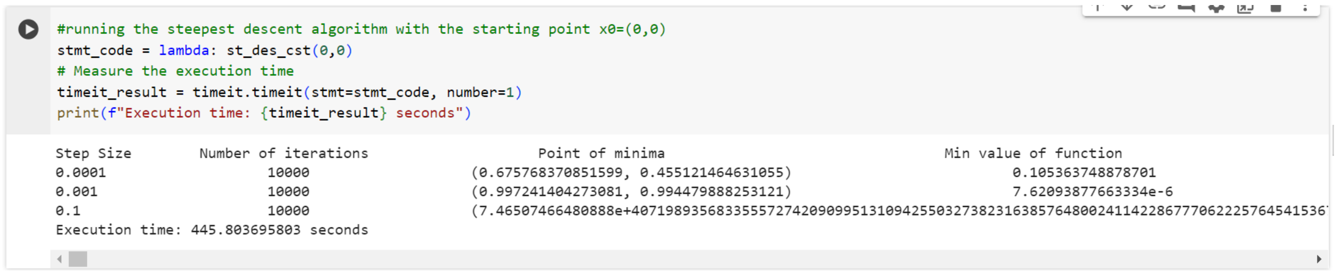


```
[18] s1=Symbol('s1')
     s2=Symbol('s2')
     expr1=Eq(gradf1(s1,s2),0)
     expr2=Eq(gradf2(s1,s2),0)
```

```
x=solve([expr1,expr2],[s1,s2])
print("Critical point: ",(x[0][0],x[0][1]))
print("Value of function at this point: ",f(x[0][0],x[0][1]))
```

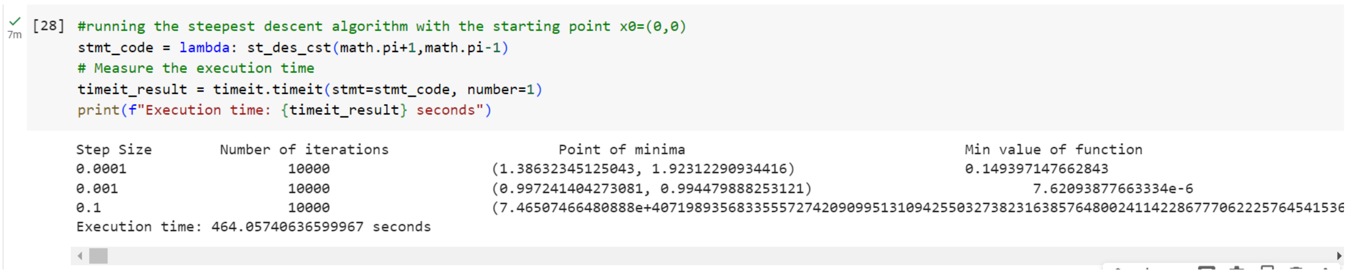
```
Critical point:  (1, 1)
Value of function at this point:  0
```

We first run the function *st _des _cst*, i.e the steepest descent algorithm that uses and compares 3 different constant stepsizes. We run the algorithm for the initial points (0,0) and (π+1,π-1). For both these points, we can see from the output that when the stepsize is taken as 0.0001, the point of minima is quite far from the actual point of minima which is (1,1), and hence the value of the function does not come out to be 0. Next we can see that the step size of 0.001 gives us a point of minima that is very close to the actual one, and the value of the function at this point is very close to the actual minimum value of 0. As we go a decimal place higher or lower, we can see that the results change drastically, since with step size 0.1, with the same number of iterations, the obtained point of minima is extremely large and hence incorrect. In this case, because of the step size being large, the algorithm likely overshoots the actual minima. The observed behaviour for the three constant step sizes is almost the same for both the starting points.



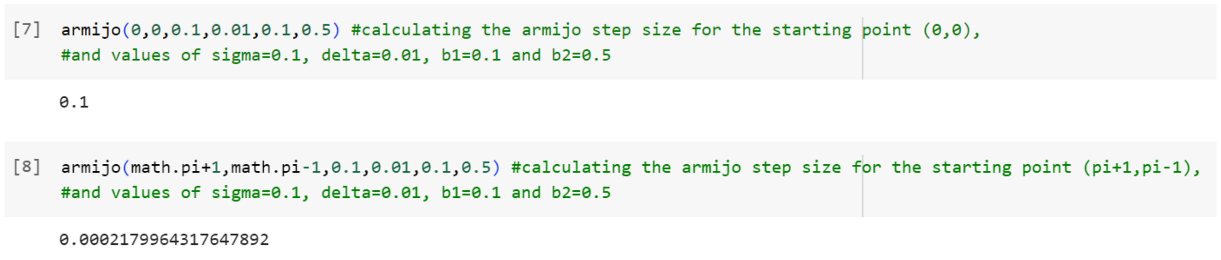
```
#running the steepest descent algorithm with the starting point x0=(0,0)
stmt_code = lambda: st_des_cst(0,0)
# Measure the execution time
timeit_result = timeit.timeit(stmt=stmt_code, number=1)
print(f"Execution time: {timeit_result} seconds")
```

```
Step Size        Number of iterations                 Point of minima                                   Min value of function
0.0001                10000                 (0.675768370851599, 0.455121464631055)                           0.105363748878701
0.001                 10000                 (0.997241404273081, 0.994479888253121)                           7.62093877663334e-6
0.1                   10000                 (7.46507466480888e+4071989356833555727420909951310942550327382316385764800241142286777062225764541536
Execution time: 445.803695803 seconds
```



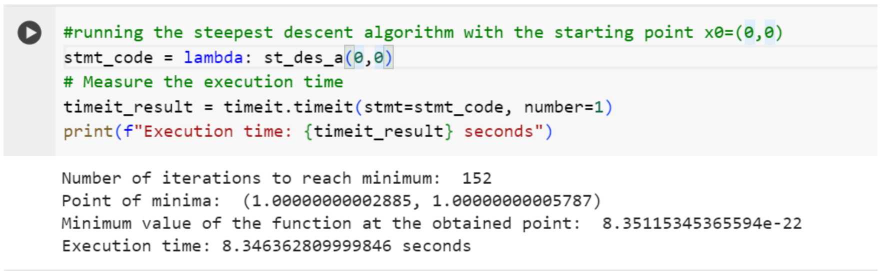
```
[28] #running the steepest descent algorithm with the starting point x0=(0,0)
     stmt_code = lambda: st_des_cst(math.pi+1,math.pi-1)
     # Measure the execution time
     timeit_result = timeit.timeit(stmt=stmt_code, number=1)
     print(f"Execution time: {timeit_result} seconds")

     Step Size         Number of iterations                 Point of minima                                   Min value of function
     0.0001                 10000                 (1.38632345125043, 1.92312290934416)                        0.149397147662843
     0.001                  10000                 (0.997241404273081, 0.994479888253121)                               7.62093877663334e-6
     0.1                    10000                 (7.46507466480888e+4071989356833555727420909951310942550327382316385764800241142286777062225764541536
     Execution time: 464.05740636599967 seconds
```

Next, we test the *armijo* function, with the starting point (0,0), taking the parameters as $\sigma = 0.1, \delta = 0.01, \beta_1 = 0.1$ and $\beta_2 = 0.5$. We get a step size of 0.1, which was actually the initial value of t, which was assigned an initial value equal to σ. Similarly, we run the *armijo* function, with the other starting point (π+1,π-1), taking the values of the parameters same as before. We get a step size which is much smaller than the one we got with the point (0,0).



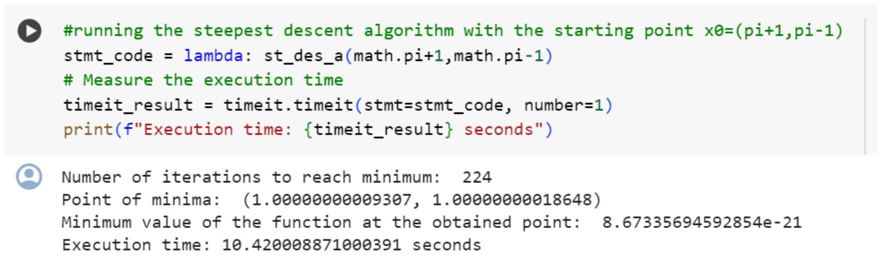
```
[7] armijo(0,0,0.1,0.01,0.1,0.5) #calculating the armijo step size for the starting point (0,0),
    #and values of sigma=0.1, delta=0.01, b1=0.1 and b2=0.5

    0.1

[8] armijo(math.pi+1,math.pi-1,0.1,0.01,0.1,0.5) #calculating the armijo step size for the starting point (pi+1,pi-1),
    #and values of sigma=0.1, delta=0.01, b1=0.1 and b2=0.5

    0.0002179964317647892
```

When we run the function *st_des_a*, i.e the steepest descent algorithm with the armijo step size, taking the initial point as (0,0), where the step size is calculated at every iteration using the *armijo* function, we find that the minima is obtained after just 152 iterations, and the point of minima is very close to the actual minima (1,1) and the minimum function value is very close to the actual minimum value 0. We also run the *timeit* function from the timeit package and find that the execution time is approximately only 8.346 seconds.
(Here, the number parameter is used to specify the number of executions of the statement or function that we want to time). The same is now done but by taking the initial points as (π+1,π-1), and again, we obtain a point of minima which is very close to the actual minima, after just 224 iterations. The execution time was approximately just 10 seconds. Multiple executions of these blocks give us slightly different outputs. However, the number of iterations are almost always a few hundreds, and the execution time close to a few seconds, which is much less than in the case of the constant step size. Hence, the obtained minima is much more accurate and faster when we use the armijo step size than when we use a constant step size.



```
#running the steepest descent algorithm with the starting point x0=(0,0)
stmt_code = lambda: st_des_a(0,0)
# Measure the execution time
timeit_result = timeit.timeit(stmt=stmt_code, number=1)
print(f"Execution time: {timeit_result} seconds")
```

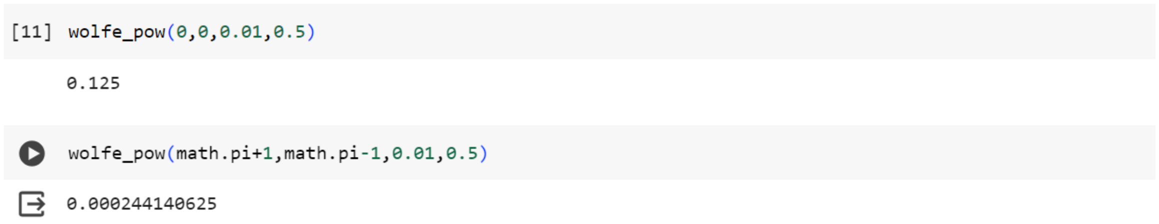
```
Number of iterations to reach minimum:  152
Point of minima:  (1.00000000002885, 1.00000000005787)
Minimum value of the function at the obtained point:  8.35115345365594e-22
Execution time: 8.346362809999846 seconds
```



```
#running the steepest descent algorithm with the starting point x0=(pi+1,pi-1)
stmt_code = lambda: st_des_a(math.pi+1,math.pi-1)
# Measure the execution time
timeit_result = timeit.timeit(stmt=stmt_code, number=1)
print(f"Execution time: {timeit_result} seconds")
```

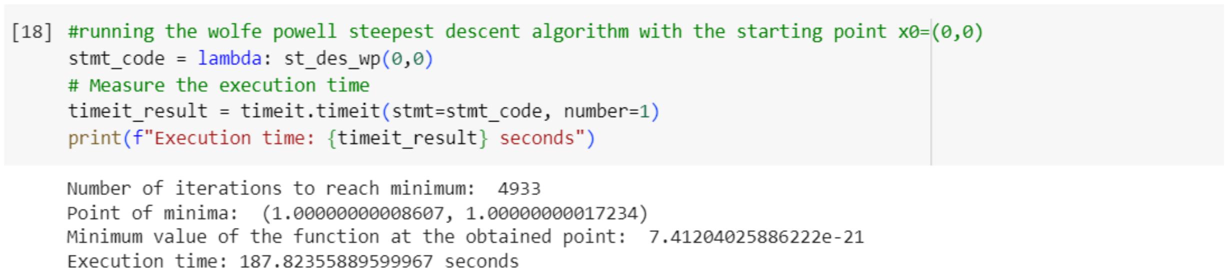
```
Number of iterations to reach minimum:  224
Point of minima:  (1.00000000009307, 1.00000000018648)
Minimum value of the function at the obtained point:  8.67335694592854e-21
Execution time: 10.420008871000391 seconds
```

Finally, we run the *wolfe_pow* function to calculate the Wolfe-Powell stepsize. The function here is run taking the initial points as (0,0) and then (π+1,π-1). The parameters are taken as $\delta = 0.01$ and $\beta = 0.5$. We get a Wolfe-Powell step size of 0.125 for the point (0,0) and a step size of approximately 0.000244 for the point $(\pi + 1, \pi - 1)$.



```
[11] wolfe_pow(0,0,0.01,0.5)

     0.125

     wolfe_pow(math.pi+1,math.pi-1,0.01,0.5)

     0.000244140625
```

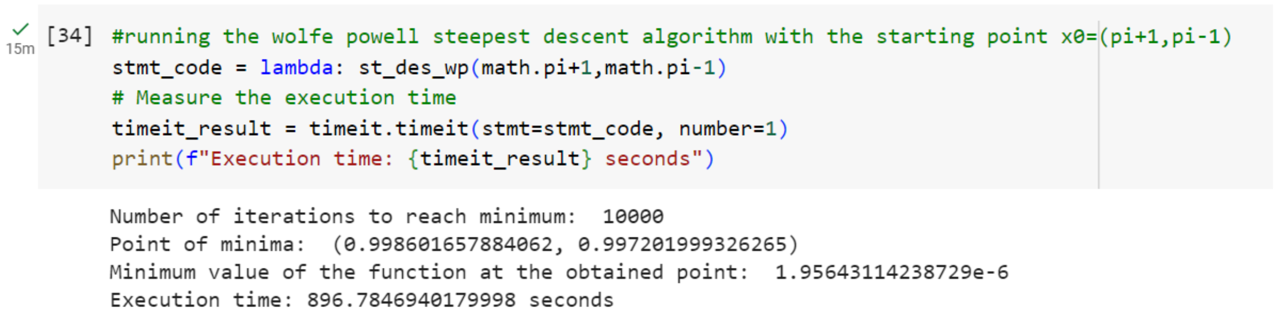
When we run the function *st_des_wp*, i.e the steepest descent algorithm with the Wolfe-Powell step size, taking the initial point as (0,0), where the step size is calculated at every iteration using the *wolfe_pow* function We find that the point of minima obtained is very close to the actual minima, but is obtained after 4933 iterations. We also run the *timeit* function from the timeit package and observe that the execution takes approximately 187.824 seconds. We notice that the obtained point of minima has almost the same accuracy as the one obtained using the armijo step size, but the time taken for convergence to the minima and the number of iterations is much larger when the Wolfe-Powell step size is used.



```
[18] #running the wolfe powell steepest descent algorithm with the starting point x0=(0,0)
     stmt_code = lambda: st_des_wp(0,0)
     # Measure the execution time
     timeit_result = timeit.timeit(stmt=stmt_code, number=1)
     print(f"Execution time: {timeit_result} seconds")
```

```
Number of iterations to reach minimum:  4933
Point of minima:  (1.00000000008607, 1.00000000017234)
Minimum value of the function at the obtained point:  7.41204025886222e-21
Execution time: 187.82355889599967 seconds
```

The same is now done but by taking the initial points as (π+1,π-1). We see that the obtained minima is much less accurate when we start from this point but it is still close to the actual minima of (1,1). The minimum function value, too, is quite close to the actual minimum function value of 0. However, the number of iterations here is 10,000, the maximum limit that we had set for the loop, which is extremely large. The execution time of approximately 896.7 seconds, too, is very high. Hence, the level of accuracy is much lower and the time of execution much higher than that in case of the armijo step size, which is much more efficient than the Wolfe-Powell step size when we start from the point $(\pi + 1, \pi - 1)$.

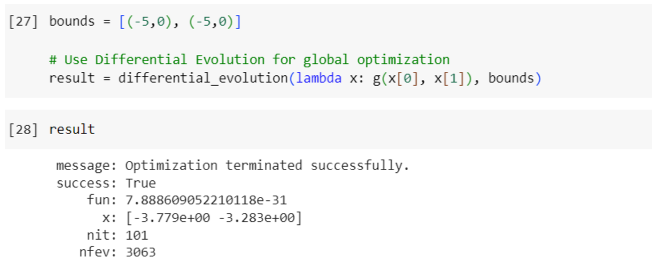


```
15m [34] #running the wolfe powell steepest descent algorithm with the starting point x0=(pi+1,pi-1)
         stmt_code = lambda: st_des_wp(math.pi+1,math.pi-1)
         # Measure the execution time
         timeit_result = timeit.timeit(stmt=stmt_code, number=1)
         print(f"Execution time: {timeit_result} seconds")

         Number of iterations to reach minimum:  10000
         Point of minima:  (0.998601657884062, 0.997201999326265)
         Minimum value of the function at the obtained point:  1.95643114238729e-6
         Execution time: 896.7846940179998 seconds
```

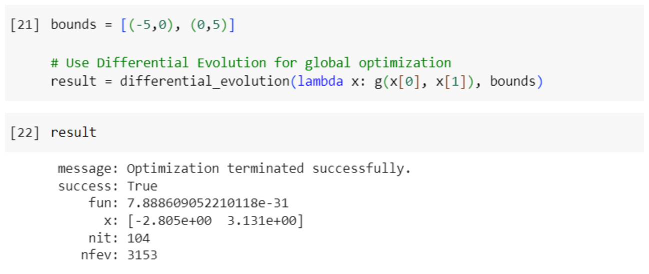
Hence, we conclude that the steepest descent algorithm is much more efficient and faster when the armijo step size is used instead of the wolfe-powell (which gives a similar level of accuracy as the armijo step size) or a constant step size, in case of the Rosenbrock function.

3.2 Himmelblau's Function

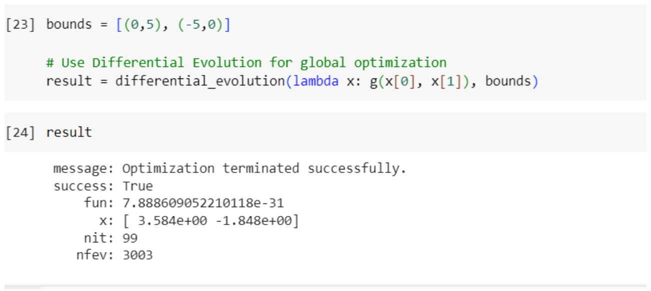
In order to check the accuracy and efficiency of our implementation of the steepest descent algorithm, we find the critical points of the Himmelblau's function, i.e., the point x=(x1,x2) for which $\nabla f(x) = 0$. However, the direct *solve* command does not work well in the case of numerical optimization problems, and hence we have used the *differential_evolution* function from the scipy.optimize package. We need to define the bounds on which we want to find the global minima in, and we need to change the bounds to check for different critical points. We find that there are 4 critical points of the function, by taking bounds with different signs and their combinations. As the degree of the equation of the function is 4, we can conclude that these are the only critical points. Now, when we input these points in the function, all of them yield a value very close to 0 or 0 itself, which is the actual minimum value of the function.



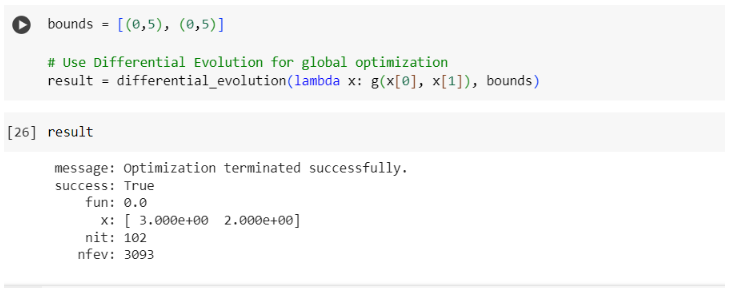
```
[27] bounds = [(-5,0), (-5,0)]

     # Use Differential Evolution for global optimization
     result = differential_evolution(lambda x: g(x[0], x[1]), bounds)

[28] result

      message: Optimization terminated successfully.
      success: True
          fun: 7.888609052210118e-31
            x: [-3.779e+00 -3.283e+00]
          nit: 101
         nfev: 3063
```



```
[21] bounds = [(-5,0), (0,5)]

     # Use Differential Evolution for global optimization
     result = differential_evolution(lambda x: g(x[0], x[1]), bounds)

[22] result

      message: Optimization terminated successfully.
      success: True
          fun: 7.888609052210118e-31
            x: [-2.805e+00  3.131e+00]
          nit: 104
         nfev: 3153
```

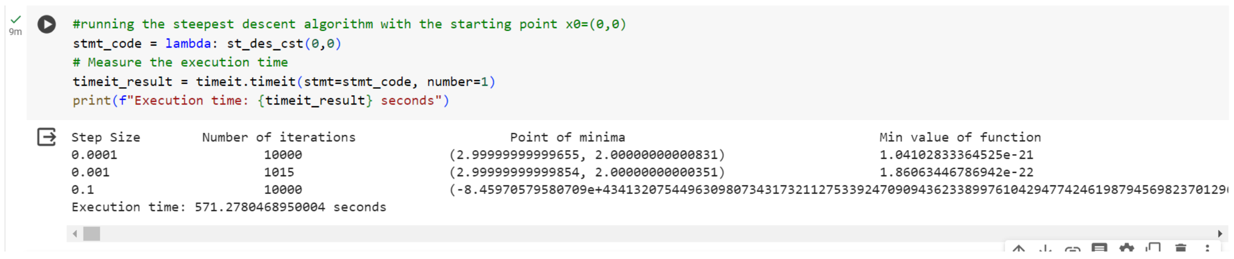


```
[23] bounds = [(0,5), (-5,0)]

     # Use Differential Evolution for global optimization
     result = differential_evolution(lambda x: g(x[0], x[1]), bounds)

[24] result

      message: Optimization terminated successfully.
      success: True
          fun: 7.888609052210118e-31
            x: [ 3.584e+00 -1.848e+00]
          nit: 99
         nfev: 3003
```

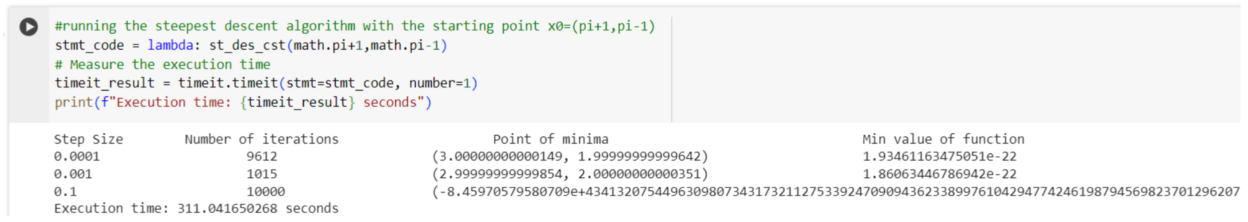


```
bounds = [(0,5), (0,5)]

# Use Differential Evolution for global optimization
result = differential_evolution(lambda x: g(x[0], x[1]), bounds)

[26] result

      message: Optimization terminated successfully.
      success: True
          fun: 0.0
            x: [ 3.000e+00  2.000e+00]
          nit: 102
         nfev: 3093
```

We now run the function *st _des _cst*, i.e the steepest descent algorithm that uses and compares 3 different constant step sizes. We run the algorithm for the initial points (0,0) and (π+1,π-1). For both these points, we can see from the output that when the step size is taken as 0.0001, and 0.001 the point of minima is quite close to the actual point of minima which is (3,2), and hence the value of the function does comes out to be 0. However, it is to be noted that the step size 0.001 took a lot less number of iterations(1015) to reach closer to the point of minima than the step size 0.0001(10000 and 9612 for $(0, 0)$ and (π+1,π-1) respectively), which is equal or very close to the maximum limit of iterations that we had used. Next we can see that the step size of 0.1, because of the step size being large, the algorithm likely overshoots the actual minima and hence this is not a good step size. We can see from the output that the observed behaviour for the three constant step sizes is almost the same for both the starting points.



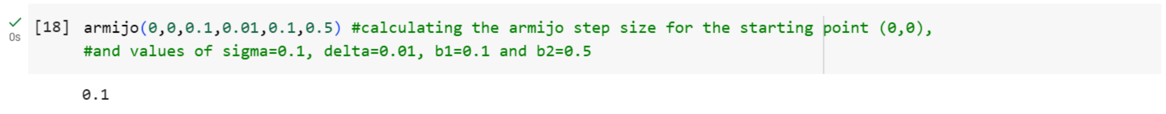
```
#running the steepest descent algorithm with the starting point x0=(0,0)
stmt_code = lambda: st_des_cst(0,0)
# Measure the execution time
timeit_result = timeit.timeit(stmt=stmt_code, number=1)
print(f"Execution time: {timeit_result} seconds")

Step Size        Number of iterations                 Point of minima                                Min value of function
0.0001                  10000                  (2.99999999999655, 2.00000000000831)                  1.04102833364525e-21
0.001                   1015                   (2.99999999999854, 2.00000000000351)                  1.86063446786942e-22
0.1                     10000                  (-8.45970579580709e+43413207544963098073431732112753392470909436233899761042947742461987945698237012 9
Execution time: 571.2780468950004 seconds
```



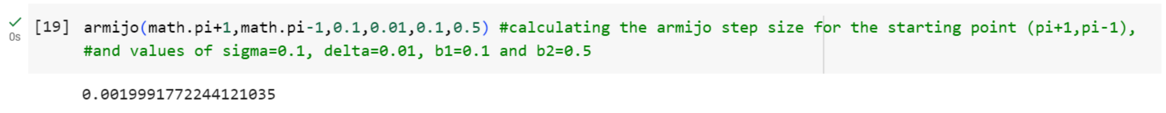
```
#running the steepest descent algorithm with the starting point x0=(pi+1,pi-1)
stmt_code = lambda: st_des_cst(math.pi+1,math.pi-1)
# Measure the execution time
timeit_result = timeit.timeit(stmt=stmt_code, number=1)
print(f"Execution time: {timeit_result} seconds")
```

```
Step Size       Number of iterations            Point of minima                                   Min value of function
0.0001                  9612            (3.00000000000149, 1.99999999999642)                      1.93461163475051e-22
0.001                   1015            (2.99999999999854, 2.00000000000351)                      1.86063446786942e-22
0.1                     10000           (-8.45970579580709e+4341320754496309807343173211275339247090943623389976104294774246198794569823701296207
Execution time: 311.041650268 seconds
```

Next, we test the *armijo* function, with the starting point (0,0), taking the parameters as $\sigma = 0.1, \delta = 0.01, \beta_1 = 0.1$ and $\beta_2 = 0.5$. We get a step size of 0.1, which was actually the initial value of t, which was assigned an initial value equal to σ. Similarly, we run the *armijo* function, with the other starting point (π+1,π-1), taking the values of the parameters same as before. We get a step size which is much smaller than the one we got with the point (0,0).

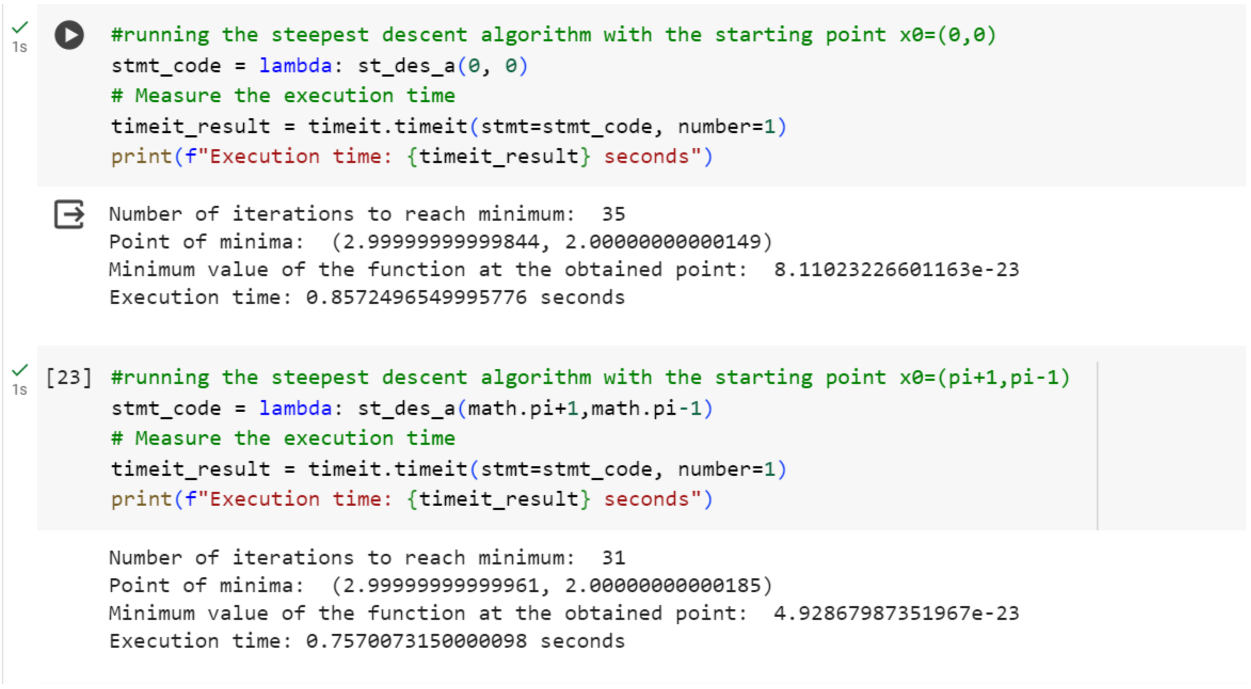


```
[18] armijo(0,0,0.1,0.01,0.1,0.5) #calculating the armijo step size for the starting point (0,0),
     #and values of sigma=0.1, delta=0.01, b1=0.1 and b2=0.5

     0.1
```



```
[19] armijo(math.pi+1,math.pi-1,0.1,0.01,0.1,0.5) #calculating the armijo step size for the starting point (pi+1,pi-1),
     #and values of sigma=0.1, delta=0.01, b1=0.1 and b2=0.5

     0.0019991772244121035
```

When we run the function *st_des_a*, i.e the steepest descent algorithm with the armijo step size, taking the initial point as (0,0), where the step size is calculated at every iteration using the *armijo* function, we find that the minima is obtained after just 35 iterations, and the point of minima is very close to the actual minima (3,2) and the minimum function value is very close to the actual minimum value 0. We also run the *timeit* function from the timeit package and find that the execution time is approximately only 0.857 seconds. (Here, the number parameter is used to specify the number of executions of the statement or function that we want to time).
The same is now done but by taking the initial points as (π+1,π-1), and again, we obtain a point of minima which is very close to the actual minima, after just 31 iterations. The execution time was approximately just 0.757 seconds. Multiple executions of these blocks give us slightly different outputs. However, the number of iterations are almost always less than 50, and the execution time close to a few milli-seconds, which is much less than in the case of the constant step size. Hence, the obtained minima is much more accurate and faster when we use the armijo step size than when we use a constant step size.



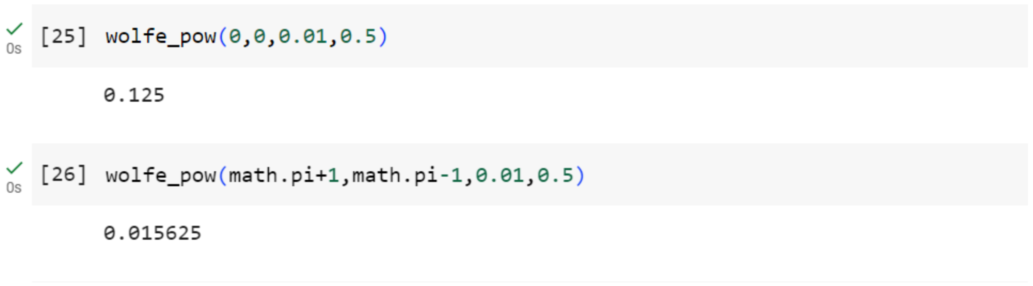
```
#running the steepest descent algorithm with the starting point x0=(0,0)
stmt_code = lambda: st_des_a(0, 0)
# Measure the execution time
timeit_result = timeit.timeit(stmt=stmt_code, number=1)
print(f"Execution time: {timeit_result} seconds")
```

```
Number of iterations to reach minimum:  35
Point of minima:  (2.99999999999844, 2.00000000000149)
Minimum value of the function at the obtained point:  8.11023226601163e-23
Execution time: 0.8572496549995776 seconds
```

```
[23] #running the steepest descent algorithm with the starting point x0=(pi+1,pi-1)
stmt_code = lambda: st_des_a(math.pi+1,math.pi-1)
# Measure the execution time
timeit_result = timeit.timeit(stmt=stmt_code, number=1)
print(f"Execution time: {timeit_result} seconds")
```

```
Number of iterations to reach minimum:  31
Point of minima:  (2.99999999999961, 2.00000000000185)
Minimum value of the function at the obtained point:  4.92867987351967e-23
Execution time: 0.7570073150000098 seconds
```

Finally, we run the *wolfe_pow* function to calculate the Wolfe-Powell stepsize. The function here is run taking the initial points as (0,0) and then (π+1,π-1). The parameters are taken as $\delta = 0.01$ and $\beta = 0.5$. We get a Wolfe-Powell step size of 0.125 for the point (0,0) and a step size of approximately 0.015635 for the point $(\pi + 1, \pi - 1)$.



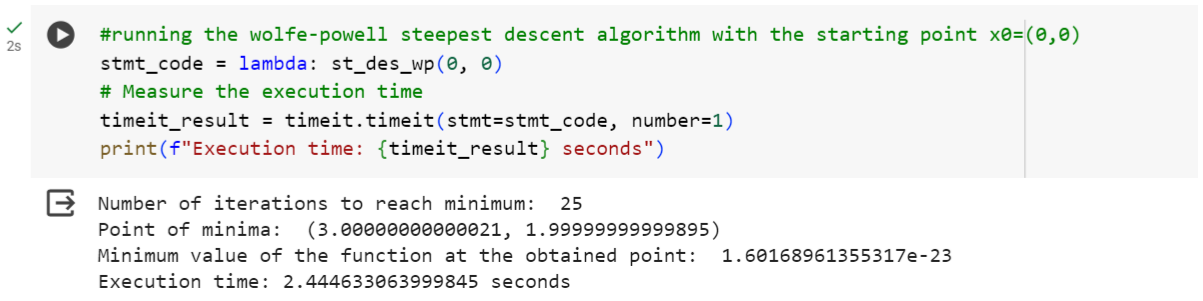
```
[25] wolfe_pow(0,0,0.01,0.5)
```

```
0.125
```

```
[26] wolfe_pow(math.pi+1,math.pi-1,0.01,0.5)
```

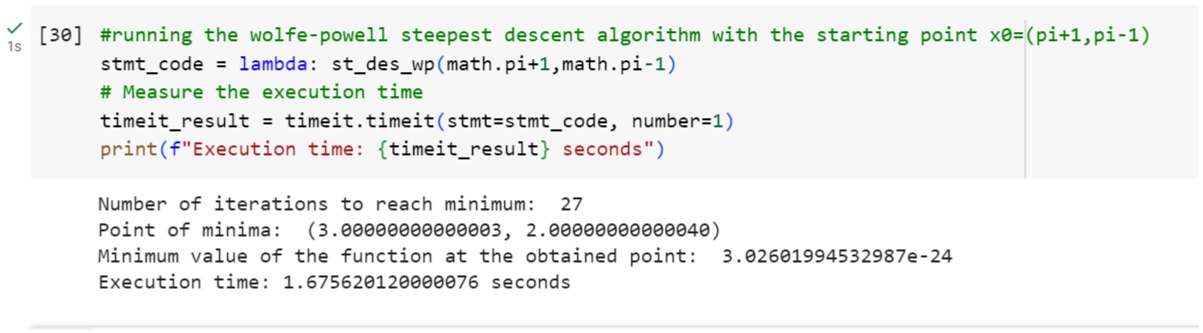
```
0.015625
```

When we run the function *st_des_wp*, i.e the steepest descent algorithm with the Wolfe-Powell step size, taking the initial point as (0,0), where the step size is calculated at every iteration using the *wolfe_pow* function We find that the point of minima obtained is very close to the actual minima, but is obtained after 25 iterations. We also run the *timeit* function from the timeit package and observe that the execution takes approximately 2.444 seconds. We notice that the obtained point of minima has almost the same accuracy as the one obtained using the armijo step size, and the time taken for convergence to the minima slightly more but the number of iterations is comparable or slightly lesser when the Wolfe-Powell step size is used.



```
#running the wolfe-powell steepest descent algorithm with the starting point x0=(0,0)
stmt_code = lambda: st_des_wp(0, 0)
# Measure the execution time
timeit_result = timeit.timeit(stmt=stmt_code, number=1)
print(f"Execution time: {timeit_result} seconds")
```

```
Number of iterations to reach minimum:  25
Point of minima:  (3.00000000000021, 1.99999999999895)
Minimum value of the function at the obtained point:  1.60168961355317e-23
Execution time: 2.444633063999845 seconds
```



```
[30] #running the wolfe-powell steepest descent algorithm with the starting point x0=(pi+1,pi-1)
stmt_code = lambda: st_des_wp(math.pi+1,math.pi-1)
# Measure the execution time
timeit_result = timeit.timeit(stmt=stmt_code, number=1)
print(f"Execution time: {timeit_result} seconds")
```

```
Number of iterations to reach minimum:  27
Point of minima:  (3.00000000000003, 2.00000000000040)
Minimum value of the function at the obtained point:  3.02601994532987e-24
Execution time: 1.675620120000076 seconds
```

The same is done next, but by taking the initial points as (π+1,π-1). We see that the obtained minima is a lot more accurate when we start from this point and is more close to the actual minima of (3,2). The minimum function value, too, is quite close to the actual minimum function value of 0. The number of iterations here is 27, which is quite efficient. The execution time of approximately 1.6756 seconds, too, is very reasonable. Hence, the level of accuracy is a little bit higher and the time of execution is also a few seconds, which is more than that in case of the armijo step size.

Hence, we conclude that the steepest descent algorithm is much more efficient and accurate when the armijo or the wolfe-powell step size is used instead of a constant step size, in case of the Himmelblau function.

4 Limitations

Although the steepest descent algorithm is a very useful method for finding solutions of optimization problems, it has several limitations that are particularly relevant in our implementation of the algorithm to finding the minima of the Rosenbrock and Himmelblau functions.

1. While defining the steepest descent function for all the step sizes, we actually require the use of while loops. However, except in the case of the Armijo step size where the stopping condition is satisfied after a few hundred iterations, we cannot use the while loop in case of the Wolfe-Powell and the constant step sizes. This is because the computation time limit exceeds the permissible limit, and the while loop keeps running indefinitely. Hence, we need to limit the number of iterations to 10,000 or even to 1000 in some cases, which adversely affects the accuracy and correctness of the obtained solution.

2. The actual stopping condition i.e $\nabla f(x) = 0$ in the steepest descent algorithm is hard to achieved in practice. So, we instead use an admissible error ϵ (the value of this error in our program is 10^{-10}) such that $\|\nabla f(x)\| < \epsilon$. The lower the value of this ϵ, the more accurate the result. However, if we keep decreasing the value of ϵ to, say, 10^{-30} or 10^{-40}, the execution time increases drastically, thus affecting the efficiency of the program.

3. The steepest descent algorithm is actually best suited for functions whose gradient is Lipschitz continuous. However, gradients of both the Rosenbrock function and the Himmelblau function are not Lipschitz continuous on $\mathbb{R}^2$. This may be the reason why the solutions obtained using the steepest descent algorithm are not much accurate in some cases (for example, in case of implementation of the steepest descent method with the Wolfe-Powell step size, when the starting point is $(\pi + 1, \pi - 1)$).

5 References

Meza, J.C., 2010. Steepest descent. Wiley Interdisciplinary Reviews: Computational Statistics, 2(6), pp.719-722.

Grad, S.M. Lecture Notes. Optimization[MAP554D].

scipy.optimize.minimize — SciPy v1.11.4 Manual. https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html.

differential_evolution https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.differential_evolution.html

Campbell, S. (2023) Python Timeit() with Examples. https://www.guru99.com/timeit-python-examples.html.

GeeksforGeeks (2022) Quiver Plot in Matplotlib. https://www.geeksforgeeks.org/quiver-plot-in-matplotlib/.

Shi, Z.J. and Shen, J., 2005. Step-size estimation for unconstrained optimization methods. Computational & Applied Mathematics, 24, pp.399-416.

Kia, S.S. Lecture 4. Optimization Methods