Laboratory 1

Variant 4

Class Group 105

Group 24

By Dimitrios Gkolias and Vasileios Ntalas

Introduction

The task which we are given is to develop a program that minimizes a function using Newton's method algorithm. The given function is $f(x, y) = 2 \cdot \sin x + 3 \cdot \cos y$ which its global minimum is when $\sin x = -1$ and $\cos y = -1$, so for the given visualization range of [-5, 5], the global minimums are:

$(x, y) = \left(-\frac{\pi}{2}, \pi\right), \left(-\frac{\pi}{2}, -\pi\right), \left(\frac{3\pi}{2}, \pi\right), \left(\frac{3\pi}{2}, -\pi\right)$, or approximately:

$$(x, y) = (-1.5708, 3.1416), (-1.5708, -3.1416), (4.7124, 3.1416), (4.7124, -3.1416)$$

The objective of this lab is to:

- Implement Newton's method to find the function's minimums.
- Test different initial points and step sizes to observe how they impact convergence.
- Visualize the optimization process within the domain x, y ∈ [-5, 5].
- Analyze the results and discuss how step size and initial conditions affect convergence.

Newton's Method is a local optimization algorithm used to find the minimum of a function. It uses second-order derivative information to quickly converge to the minimum. The method works by iteratively updating the current guess for the optimal solution based on the gradient and Hessian matrix of the function. It is commonly used for training machine and deep learning models, solving physics-based simulations and engineering design problems, and is also applied in financial modeling and risk analysis.

The big advantage of Newton's method is its fast convergence, namely because it has quadratic convergence near the optimum, meaning it converges very quickly once close to the solution. It also uses second-order derivative information (Hessian) to refine step direction, improving accuracy. However, on the other hand, Newton's method is computationally expensive, because the computing and inverting the Hessian matrix is computationally expensive,

especially for high-dimensional problems. Moreover, it is very sensitive to the initial guess and it may fail to converge if the initial guess is far from the true solution. Moreover, if the Hessian is not positive definite, the method may not converge or may converge to a saddle point, and the step size needs to be carefully chosen to ensure convergence- poor choices can lead to divergence.

Through this study, we aim to gain insights into the strengths and limitations of Newton's method and how parameter selection influences its performance.

Implemenation

For the implementation of the code, we are going to show you step-by-step how the solution works:

1. Symbolization of the function

```
import sympy as sp

x_sym, y_sym = sp.symbols('x y')

def function(x, y):
    return 2*np.sin(x)+3*np.cos(y)

def symbolic_function(x, y):
    return 2*sp.sin(x) + 3*sp.cos(y)
```

In order to apply generally the code in every function we might later want to test, we have to make the function and its variables symbolic. That way it keeps the function in mathematical form, which allows us to later compute derivatives (gradient), the Hessian matrix (second derivatives), and perform algebraic simplifications. We accomplish this by using the sympy library allowing us to symbolize our function and to transform our variables x, y as symbols.

2. Computing the gradient of the function

```
def gradient(f, variables):
    grad_expr = [sp.diff(f, var) for var in variables]
    grad_func = sp.lambdify(variables, grad_expr, 'numpy')
    return grad_func
```

We made a function for computing the gradient of a given function and variables, in which using the function *diff* of the sympy library we calculate the partial derivative of the function with respect to each variable (in our case, x and y). Next using the *lambdify* function we convert the result to a numerical form that can be evaluated efficiently using NumPy, and we return the gradient of the function.

3. Computing the Hessian of the function

```
def hessian(f, variables):
    grad_expr = [sp.diff(f, var) for var in variables]
    hessian_expr = [[sp.diff(g, var) for var in variables] for g in grad_expr]
    hessian_func = sp.lambdify(variables, hessian_expr, 'numpy')
    return hessian_func
```

Similarly we make a function for the calculation of the Hessian matrix of a given function and variables. We calculate the hessian by applying again the *diff* function in the gradients of the function for each variable. Again we *lambdify* the function in a numeric form and we return it.

4. Newton's method function

```
def newton_method(initial_guess, alpha, tol=1e-6, max_iter=1000):
    grad_func = gradient(symbolic_function(x_sym, y_sym), [x_sym, y_sym])
    hess_func = hessian(symbolic_function(x_sym, y_sym), [x_sym, y_sym])
    x_k = np.array(initial_guess, dtype=float)
    iter_count = 0
    path = [x_k.copy()]

    for _ in range(max_iter):
        grad = np.array(grad_func(x_k[0], x_k[1]))
        hess = np.array(hess_func(x_k[0], x_k[1]))

        if np.linalg.det(hess) == 0:
            print("Hessian is singular, stopping optimization.")
            break

        step = np.linalg.solve(hess, -grad)
        x_k += alpha * step
        path.append(x_k.copy())
        iter_count += 1

        if np.linalg.norm(step) < tol:
            break
```

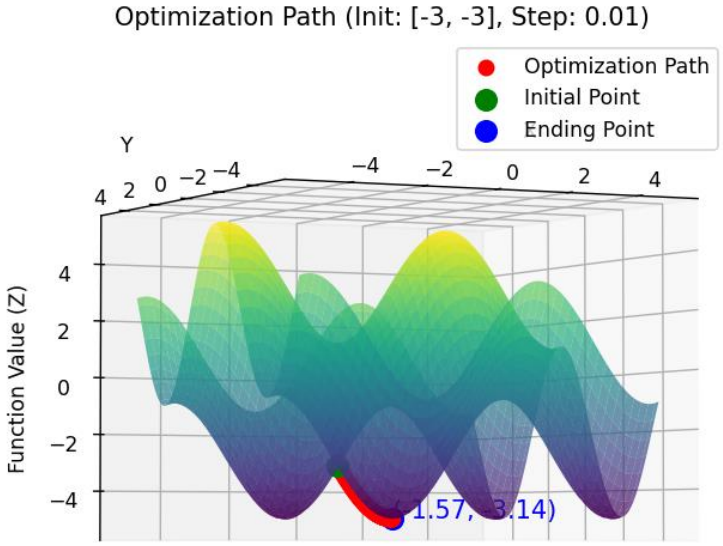
```
    return x_k, iter_count, np.array(path)
```

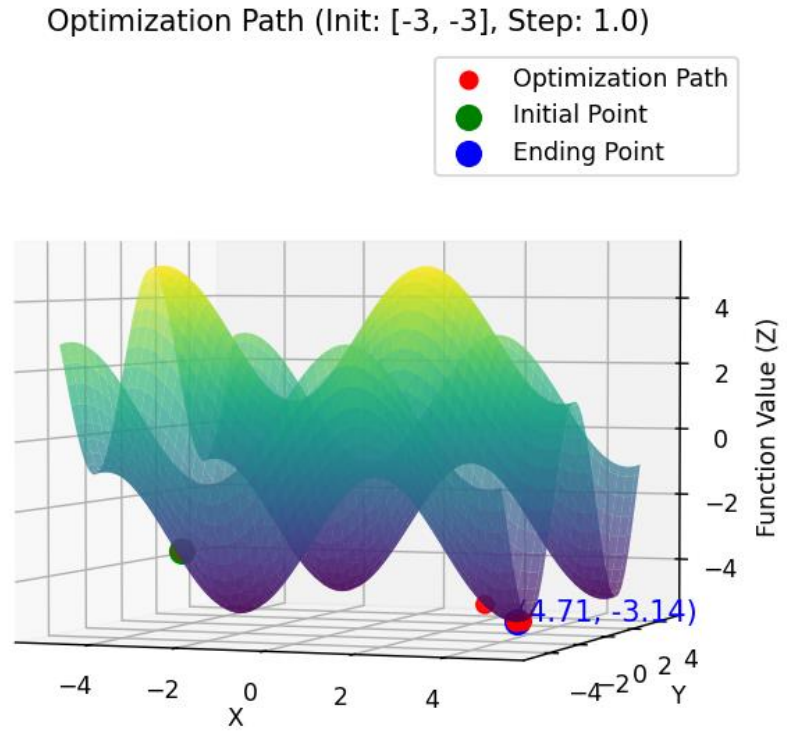
Now we proceed with the actual Newton's method implementation where we define a function with parameters the initial guess from which the algorithm will start searching, the constant alpha (a_k) which is the step size we choose to use in the calculation of the next point of search ($x_{k+1} = x_k - a_k \cdot \left(H_f(x_k)\right)^{-1} \cdot \nabla f(x_k)$). Also, we have the tolerance and max iterations of the algorithm as parameters.

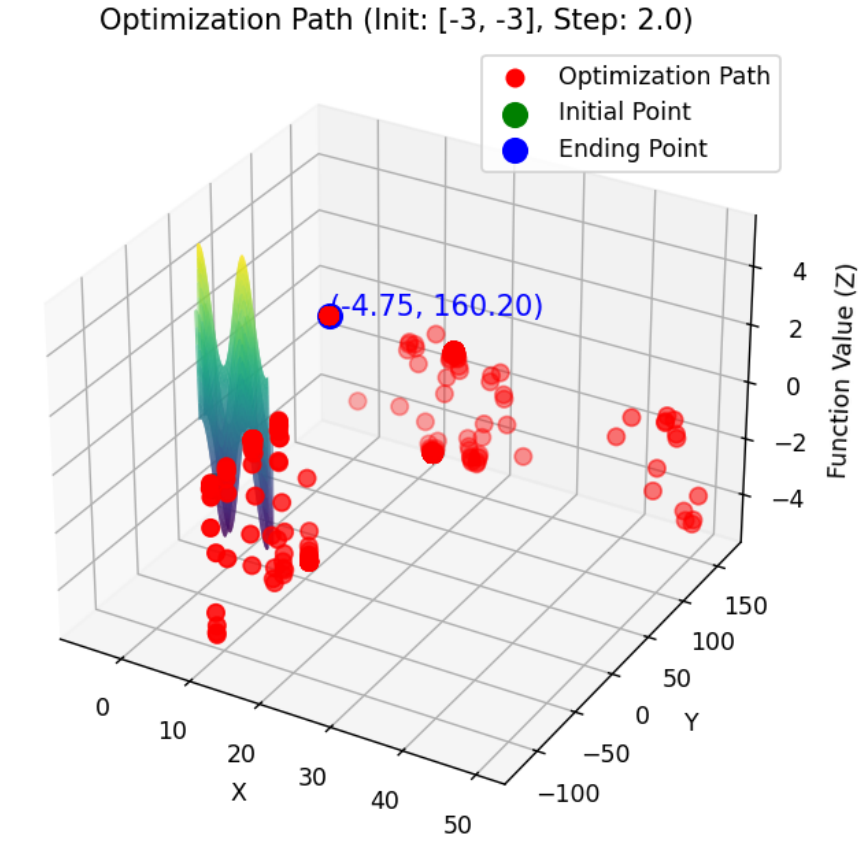
We start by creating the gradient and the hessian function of the task's function with the corresponding variables, we ensure that the initial point is in numerical format that supports the upcoming mathematical calculations, we initialize the iteration count to zero and make a copy of the initial point so we can later track the path of the searching procedure.

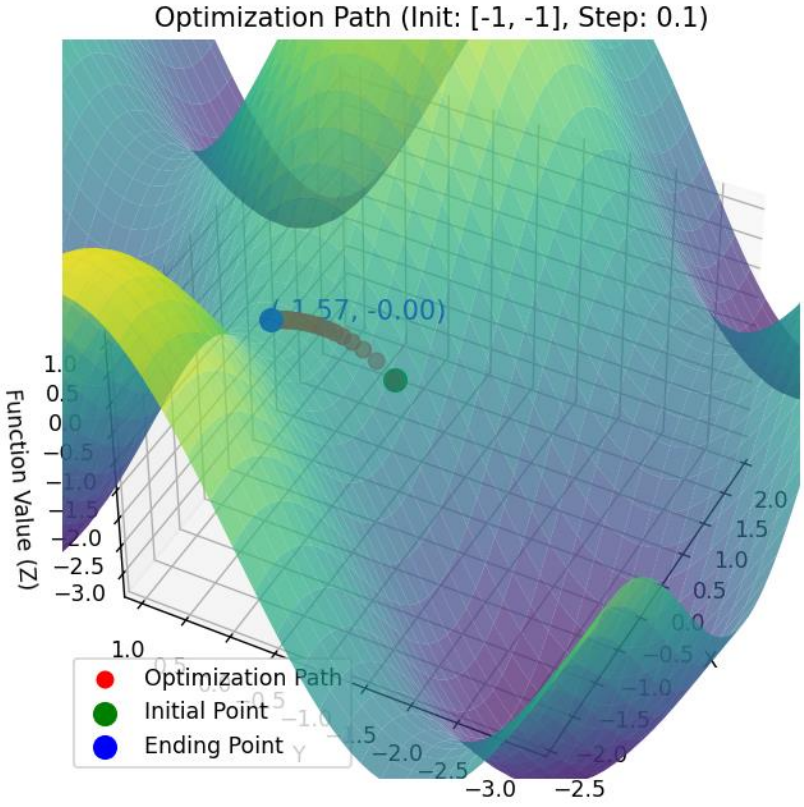
Next, we enter a loop for the max iterations inside of which we compute iteratively the gradient and the hessian of the corresponding "initial" point of the iteration we are in. We check if the Hessian is singular, and if yes we stop the procedure, but if not we continue with the calculation of the step we are going to make for the next point we check, namely $step = H_f(x_k)^{-1} \cdot \left(-\nabla f(x_k)\right)$. We compute the step by using the *np.linalg.solve* function which finds this step by solving the linear system: $H_f(x_k) \cdot step = -\nabla f(x_k)$. Then we update the point/vector by multiplying the step with the step size a_k (*aplha*) and adding it to our current point. Finally, we append a copy of the current point to our path and we increase the iteration count by one. The algorithm ends when the Euclidean norm of the step vector is lower than the tolerance we have defined. In the end, we return the last point/vector, the iteration count and the path that was followed.

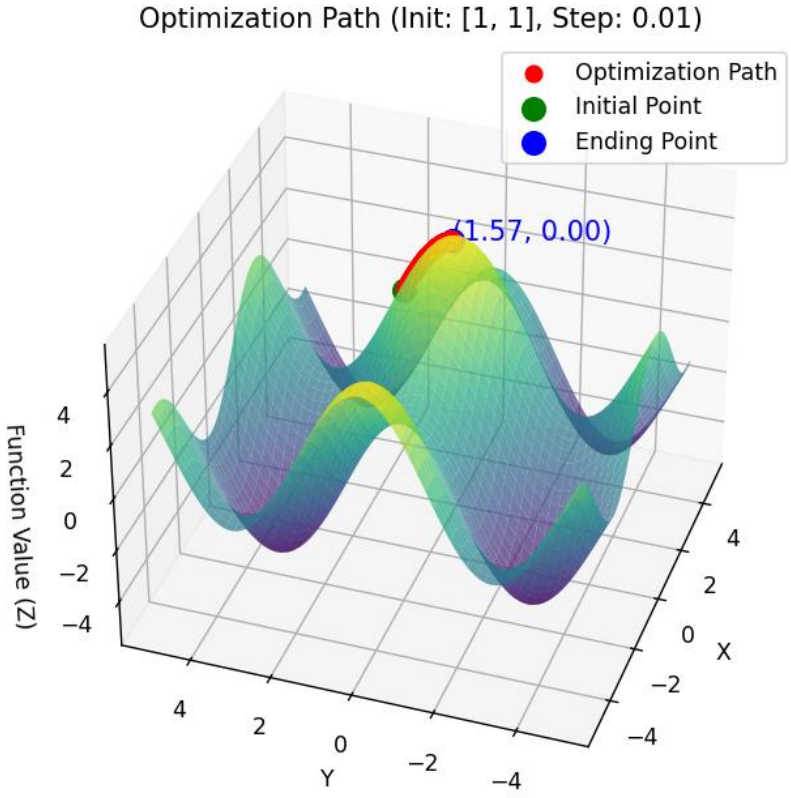
We also made a simple visualization function and we tested the Newton's method for many different initial points and step sizes. Here are some visualizations from the different initial point and step size pairs:

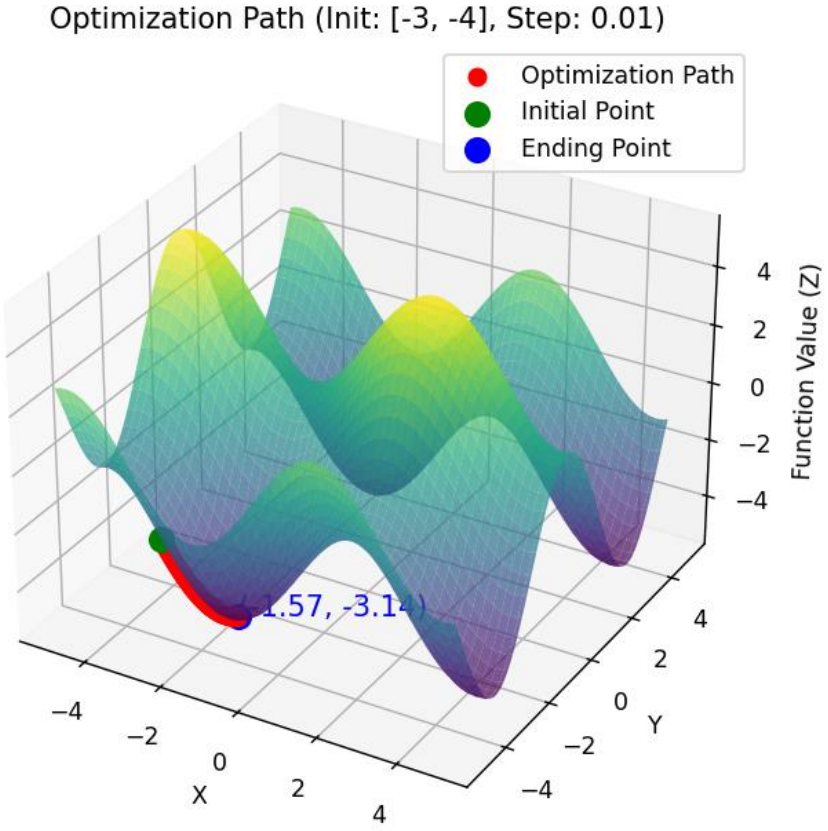

Optimization Path (Init: [-3, -3], Step: 0.01)
Optimization Path
Initial Point
Ending Point
Y
Function Value (Z)
(1.57, -3.14)

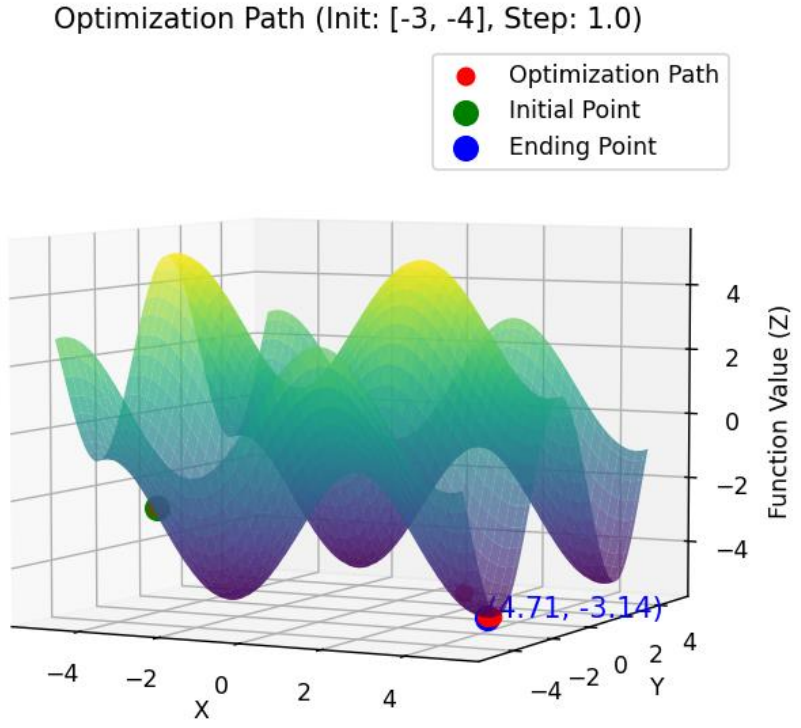

Optimization Path (Init: [-3, -3], Step: 1.0)
Optimization Path
Initial Point
Ending Point
Function Value (Z)
(4.71, -3.14)
X
Y

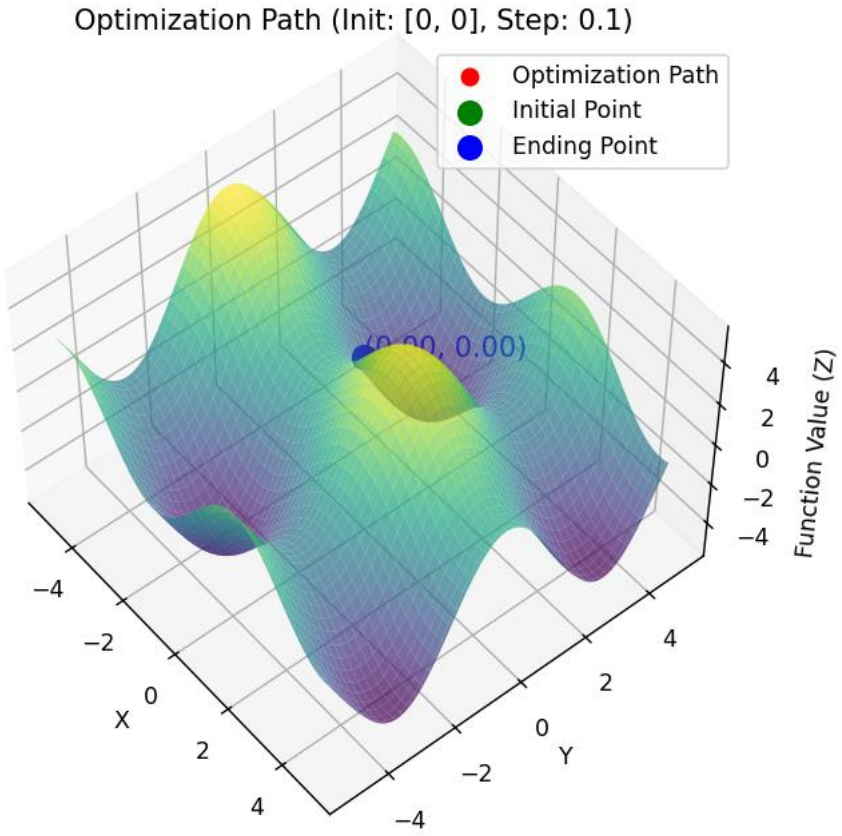

Optimization Path (Init: [-3, -3], Step: 2.0)
Optimization Path
Initial Point
Ending Point
(-4.75, 160.20)
Function Value (Z)
X
Y


Optimization Path (Init: [-1, -1], Step: 0.1)
(1.57, -0.00)
Function Value (Z)
Optimization Path
Initial Point
Ending Point
X
Y


Optimization Path (Init: [1, 1], Step: 0.01)
Optimization Path
Initial Point
Ending Point
(1.57, 0.00)
Function Value (Z)
4
2
0
−2
−4
X
Y


Optimization Path (Init: [-3, -4], Step: 0.01)
Optimization Path
Initial Point
Ending Point
Function Value (Z)
X
Y


Optimization Path (Init: [-3, -4], Step: 1.0)
Optimization Path
Initial Point
Ending Point
(4.71, -3.14)
Function Value (Z)
X
Y


Optimization Path (Init: [0, 0], Step: 0.1)
Optimization Path
Initial Point
Ending Point
(0.00, 0.00)
Function Value (Z)
X
Y

Discussion

The results of our implementation are the following demonstrated in a table:

Initial Guess	Learning Rate	Minimum Approximation	Approximation in Terms of π	Iterations
[-3, -3]	0.01	[-1.5708, -3.1416]	$(-\pi/2, -\pi)$	1000
[-3, -3]	0.1	[-1.5708, -3.1416]	$(-\pi/2, -\pi)$	130
[-3, -3]	1.0	[4.7124, -3.1416]	$(3\pi/2, -\pi)$	5
[-3, -3]	2.0	[-4.7500, 160.1963]	Divergence	1000
[-1, -1]	0.01	[-1.5708, -0.00004]	$(-\pi/2, 0)$	1000
[-1, -1]	0.1	[-1.5708, -0.0000007]	$(-\pi/2, 0)$	132
[-1, -1]	1.0	[-1.5708, 0.0000]	$(-\pi/2, 0)$	5
[-1, -1]	2.0	[-230.9097, -28.3360]	Divergence	1000
[3, 3]	0.01	[1.5708, 3.1416]	$(\pi/2, \pi)$	1000
[3, 3]	0.1	[1.5708, 3.1416]	$(\pi/2, \pi)$	130
[3, 3]	1.0	[-4.7124, 3.1416]	$(-3\pi/2, \pi)$	5
[3, 3]	2.0	[4.7500, -160.1963]	Divergence	1000
[-3, -4]	0.01	[-1.5708, -3.1416]	$(-\pi/2, -\pi)$	1000
[-3, -4]	0.1	[-1.5708, -3.1416]	$(-\pi/2, -\pi)$	133
[-3, -4]	1.0	[4.7124, -3.1416]	$(3\pi/2, -\pi)$	5
[-3, -4]	2.0	[-4.7500, -9.4235]	Divergence	1000
[0, 0]	0.01	Hessian is singular	Hessian is singular	0
[0, 0]	0.1	Hessian is singular	Hessian is singular	0
[0, 0]	1.0	Hessian is singular	Hessian is singular	0
[0, 0]	2.0	Hessian is singular	Hessian is singular	0

Impact of different initial vectors:

The choice of the initial point significantly affected the convergence of Newton's method. Firstly, starting near a global minimum, for example from (-3, 3) close to (-1.5707, 3.1415), resulted in fast convergence, typically within few iterations. So, we notice that the closer we start to the minimum the better are the chances we find the global minimum and even faster with fewer iterations.

On the other hand, starting farther from a global minimum required more iterations, as the function had to navigate through larger gradients before reaching an optimum, or even in some

cases getting stuck in saddle points and local maximums, as its the case for the vectors close to ± 1 that we tested. Most cases when we start from a vector not close to the global minimum the method fails. Moreover, if we start from (0, 0) where the Hessian cannot be calculated then we can't find the global minimum with the Newton's method and the optimization is terminated.

Impact of different step size parameters (a_k):

The choice of step size (a_k) played a crucial role as well in the performance of the algorithm. In particular, we observed that with a small step size (as *0.01*) the convergence is a lot slower (reached the maximum 1000 iterations), meaning that the method needed more iterations to locate the minimum. On the other hand, a large step size (as 2.*0*) led to divergence, where the algorithm overshot the optimal point and failed to converge. Overall, we had better results with a moderate step size (as *0.1* and *1.0*) which balanced speed and stability, often converging efficiently, with the step size *1.0* to be the most efficient, requiring less iterations for finding the global minimum.

Conclusion

In conclusion, Newton's method performs well when the initia; guess is reasonably close to the actual minimum, and in most cases poorly when otherwise. Also, choosing the proper step size is really important, as we observed that a higher step size leads to divergence and a lower step size to really slow convergence.

In general, Newton's method is highly effective for function minimization when the initial guess and learning rate are properly chosen. It exhibits quadratic convergence when close to a minimum, allowing rapid optimization. However, it is highly sensitive to both initial conditions and step size. Poor choices can lead to slow convergence, divergence, or even failure due to a singular Hessian matrix. Therefore, careful parameter tuning and understanding of the function's properties are essential for successful application.