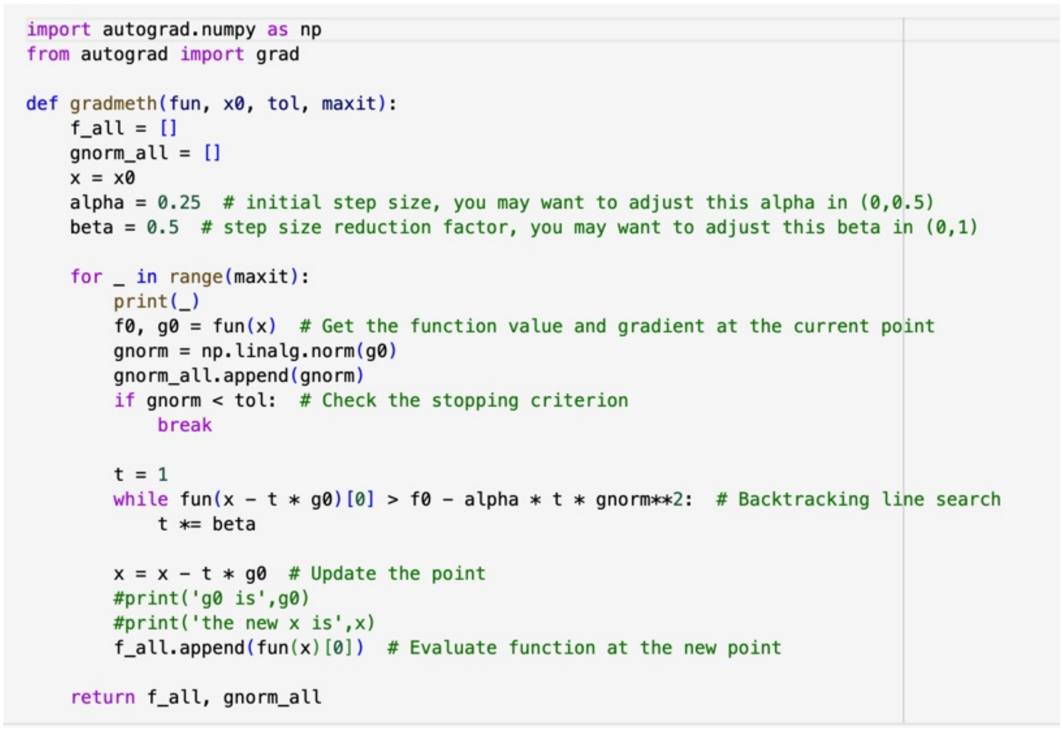
GRADIENT METHOD AND NEWTON'S METHOD

Theory

Author
Mason An
Where?
When?

Contents



```
import autograd.numpy as np
from autograd import grad

def gradmeth(fun, x0, tol, maxit):
    f_all = []
    gnorm_all = []
    x = x0
    alpha = 0.25  # initial step size, you may want to adjust this alpha in (0,0.5)
    beta = 0.5  # step size reduction factor, you may want to adjust this beta in (0,1)

    for _ in range(maxit):
        print(_)
        f0, g0 = fun(x)  # Get the function value and gradient at the current point
        gnorm = np.linalg.norm(g0)
        gnorm_all.append(gnorm)
        if gnorm < tol:  # Check the stopping criterion
            break

        t = 1
        while fun(x - t * g0)[0] > f0 - alpha * t * gnorm**2:  # Backtracking line search
            t *= beta

        x = x - t * g0  # Update the point
        #print('g0 is',g0)
        #print('the new x is',x)
        f_all.append(fun(x)[0])  # Evaluate function at the new point

    return f_all, gnorm_all
```

Figure 1: Gradient method code

1 Problem 1

1.1 a

We get the optimal value $p^* = -3.2281746031776803$. So in conclusion, the theoratical factor is 0.9999475472711422, which is significantly larger than the factor we got, which is 0.66016999. Hence the result we got from class is satisfied.

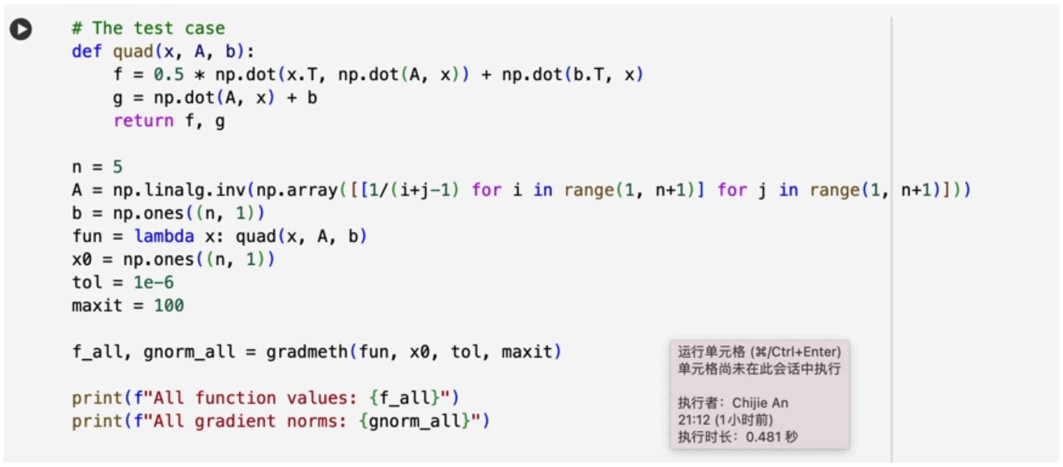
1.2 b

After calculation, the algorithm converges after 70 iterations and the optimal value $p^* = -67.46372056$. The estimated condition number $\frac{M}{m} = 0.64520435$. We estimate the condition number by first estimate c, by solving the equation $c^{100} = f(x_0) - p^*$. In order to simplify the calculation, we apply log to both sides. And then we ge the condition factor from c, by following the relationship $\frac{M}{m} = \frac{2\beta\alpha}{1-c}$. It's a bit wired for the estimation of condition number I got is less than 1.

2 Problem 2

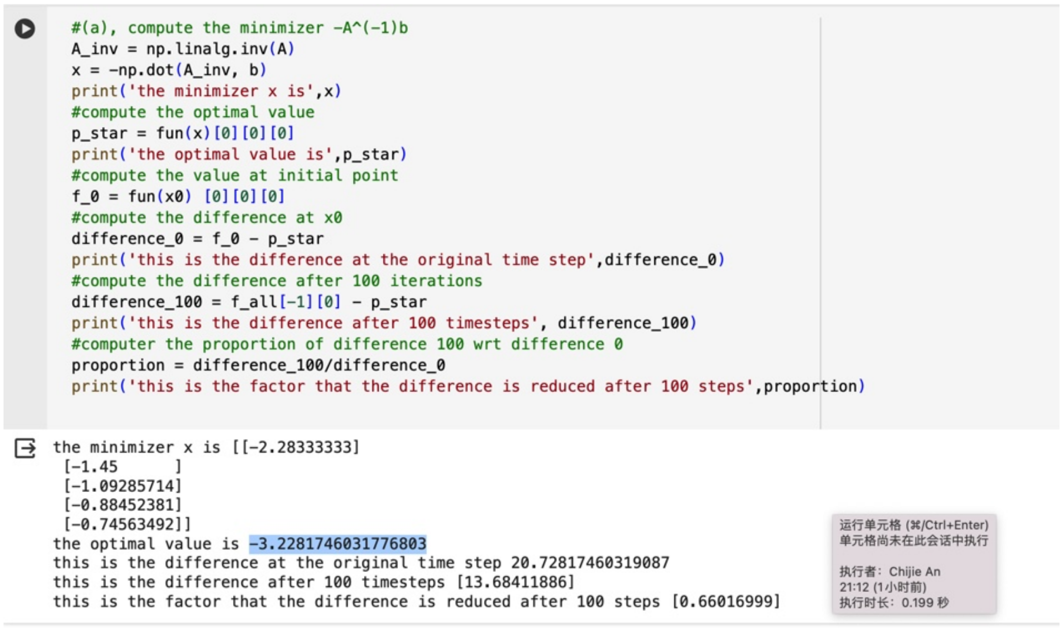
2.1 a

Implementing Newton's method to test case 1 is trivial, because the major difficulty of Newton's method is to calculate the hessian matrix, but test case 1 is simply a quadratic function, and we can write an explicit formula for the hessian matrix as A. Also, Newton's



```
# The test case
def quad(x, A, b):
    f = 0.5 * np.dot(x.T, np.dot(A, x)) + np.dot(b.T, x)
    g = np.dot(A, x) + b
    return f, g

n = 5
A = np.linalg.inv(np.array([[1/(i+j-1) for i in range(1, n+1)] for j in range(1, n+1)]))
b = np.ones((n, 1))
fun = lambda x: quad(x, A, b)
x0 = np.ones((n, 1))
tol = 1e-6
maxit = 100

f_all, gnorm_all = gradmeth(fun, x0, tol, maxit)

print(f"All function values: {f_all}")
print(f"All gradient norms: {gnorm_all}")
```

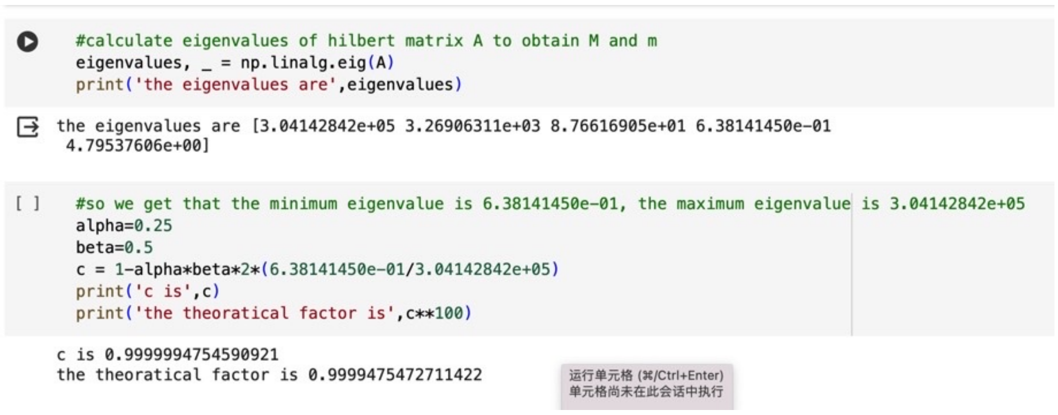
Figure 2: Implement on example case 1



```
#(a), compute the minimizer -A^(-1)b
A_inv = np.linalg.inv(A)
x = -np.dot(A_inv, b)
print('the minimizer x is',x)
#compute the optimal value
p_star = fun(x)[0][0][0]
print('the optimal value is',p_star)
#compute the value at initial point
f_0 = fun(x0) [0][0][0]
#compute the difference at x0
difference_0 = f_0 - p_star
print('this is the difference at the original time step',difference_0)
#compute the difference after 100 iterations
difference_100 = f_all[-1][0] - p_star
print('this is the difference after 100 timesteps', difference_100)
#computer the proportion of difference 100 wrt difference 0
proportion = difference_100/difference_0
print('this is the factor that the difference is reduced after 100 steps',proportion)
```

```
the minimizer x is [[-2.28333333]
 [-1.45      ]
 [-1.09285714]
 [-0.88452381]
 [-0.74563492]]
the optimal value is -3.2281746031776803
this is the difference at the original time step 20.72817460319087
this is the difference after 100 timesteps [13.68411886]
this is the factor that the difference is reduced after 100 steps [0.66016999]
```

Figure 3: Compute the optimal value and the differences to the optimal value, and the factor of reducing the distance to the optimal value



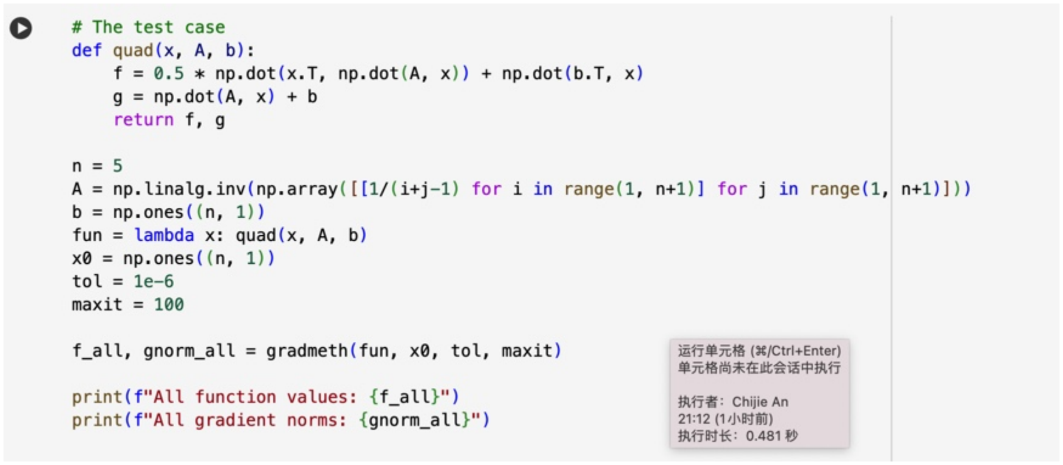
```
#calculate eigenvalues of hilbert matrix A to obtain M and m
eigenvalues, _ = np.linalg.eig(A)
print('the eigenvalues are',eigenvalues)
```

```
the eigenvalues are [3.04142842e+05 3.26906311e+03 8.76616905e+01 6.38141450e-01
 4.79537606e+00]
```

```
#so we get that the minimum eigenvalue is 6.38141450e-01, the maximum eigenvalue is 3.04142842e+05
alpha=0.25
beta=0.5
c = 1-alpha*beta*2*(6.38141450e-01/3.04142842e+05)
print('c is',c)
print('the theoratical factor is',c**100)
```

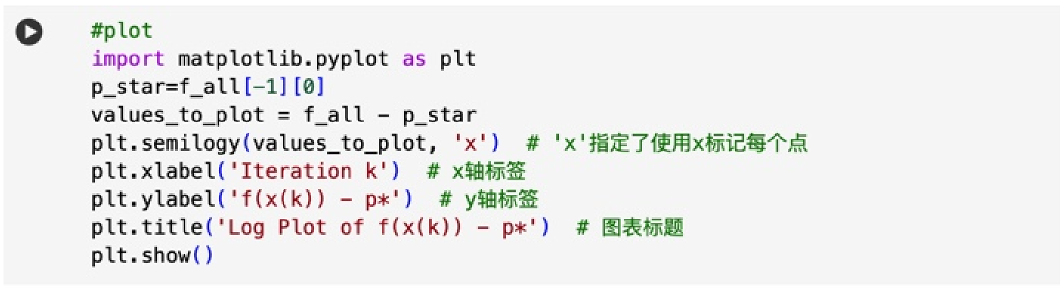
```
c is 0.9999994754590921
the theoratical factor is 0.9999475472711422
```

Figure 4: Compute the eigenvalues of A, and the theoritical factor that the algorithm can reduce the distance to the optimal value



```
# The test case
def quad(x, A, b):
    f = 0.5 * np.dot(x.T, np.dot(A, x)) + np.dot(b.T, x)
    g = np.dot(A, x) + b
    return f, g

n = 5
A = np.linalg.inv(np.array([[1/(i+j-1) for i in range(1, n+1)] for j in range(1, n+1)]))
b = np.ones((n, 1))
fun = lambda x: quad(x, A, b)
x0 = np.ones((n, 1))
tol = 1e-6
maxit = 100

f_all, gnorm_all = gradmeth(fun, x0, tol, maxit)

print(f"All function values: {f_all}")
print(f"All gradient norms: {gnorm_all}")
```

Figure 5: Code for the more interesting objective function of example case 2



```
#plot
import matplotlib.pyplot as plt
p_star=f_all[-1][0]
values_to_plot = f_all - p_star
plt.semilogy(values_to_plot, 'x')  # 'x'指定了使用x标记每个点
plt.xlabel('Iteration k')  # x轴标签
plt.ylabel('f(x(k)) - p*')  # y轴标签
plt.title('Log Plot of f(x(k)) - p*')  # 图表标题
plt.show()
```

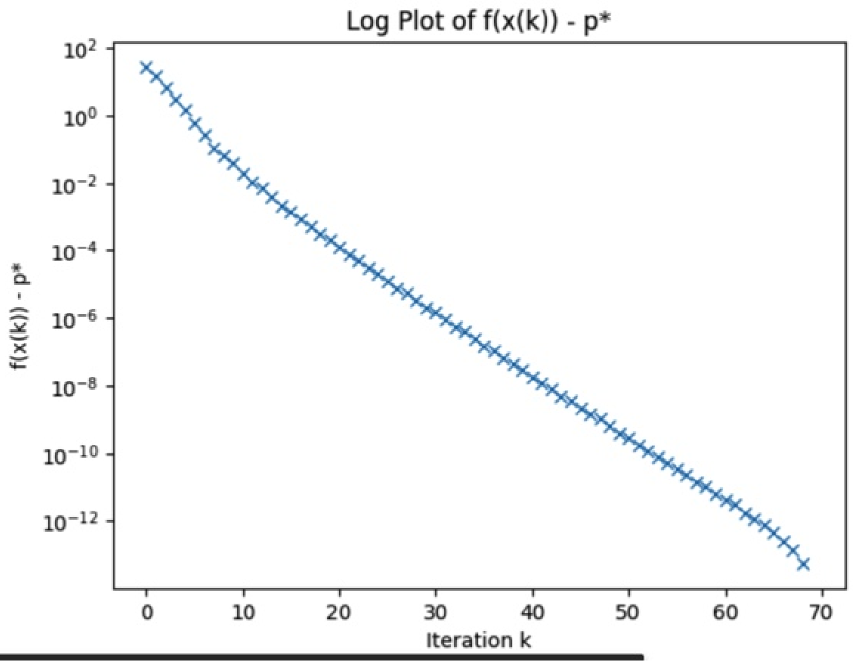
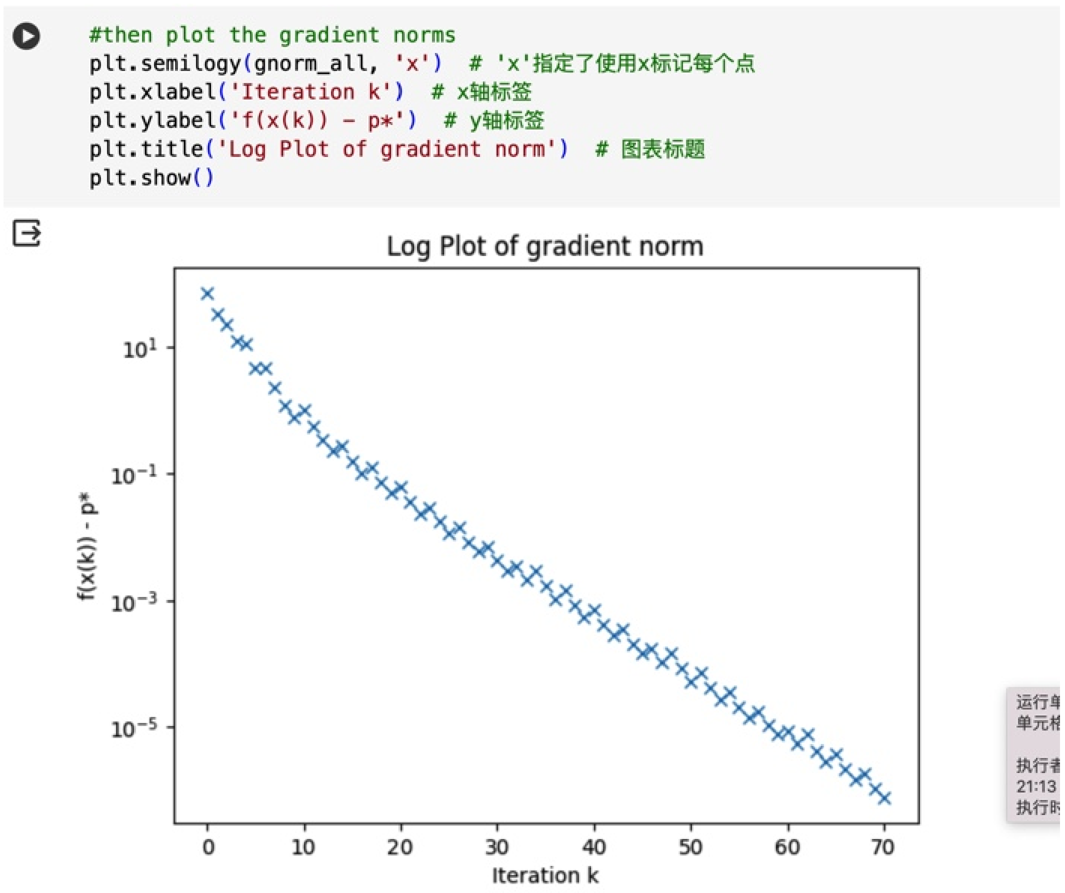
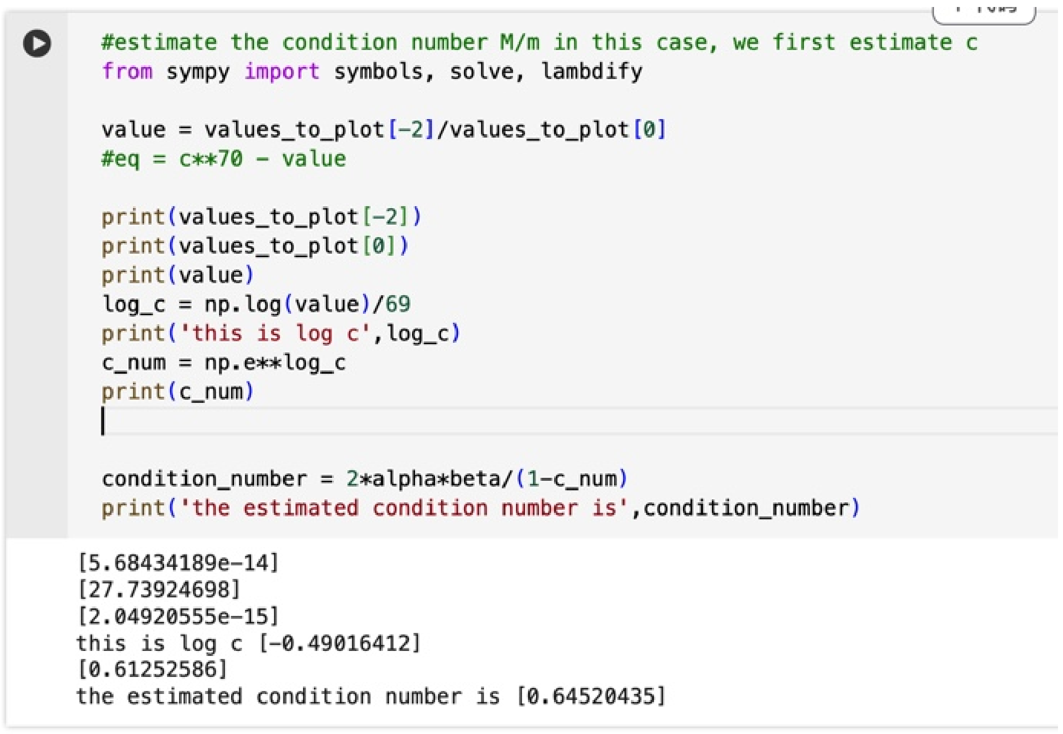


Figure 6: Log plot for the distance to the optimal function value



```
#then plot the gradient norms
plt.semilogy(gnorm_all, 'x')  # 'x'指定了使用x标记每个点
plt.xlabel('Iteration k')  # x轴标签
plt.ylabel('f(x(k)) - p*')  # y轴标签
plt.title('Log Plot of gradient norm')  # 图表标题
plt.show()
```

Figure 7: Code for gradient norm



```
#estimate the condition number M/m in this case, we first estimate c
from sympy import symbols, solve, lambdify

value = values_to_plot[-2]/values_to_plot[0]
#eq = c**70 - value

print(values_to_plot[-2])
print(values_to_plot[0])
print(value)
log_c = np.log(value)/69
print('this is log c',log_c)
c_num = np.e**log_c
print(c_num)


condition_number = 2*alpha*beta/(1-c_num)
print('the estimated condition number is',condition_number)
```

```
[5.68434189e-14]
[27.73924698]
[2.04920555e-15]
this is log c [-0.49016412]
[0.61252586]
the estimated condition number is [0.64520435]
```

Figure 8: Estimation of condition number

method converges much quicker than gradient method.

2.2 b

Newton's method converges after seven iterations. We will further use the theories mentioned in the notes to calculate the upper bound and lower bound of number of iterations.

First we calculate the following values: $m = 2, M = 281.5851919713861$, $L = 210.33578528828642$, $\eta = min\{1, 3(1-2\alpha)\}\frac{m^2}{L} = 0.01901721095398689$, $\gamma = \alpha\beta\eta^2\frac{m}{M^2} = 1.1402866297027554 \times 10^{-9}$.

To calculate the upper bound of number of steps, we suppose the norm of gradient is greater or equal to η all the time, and the number of steps can be calculated as,

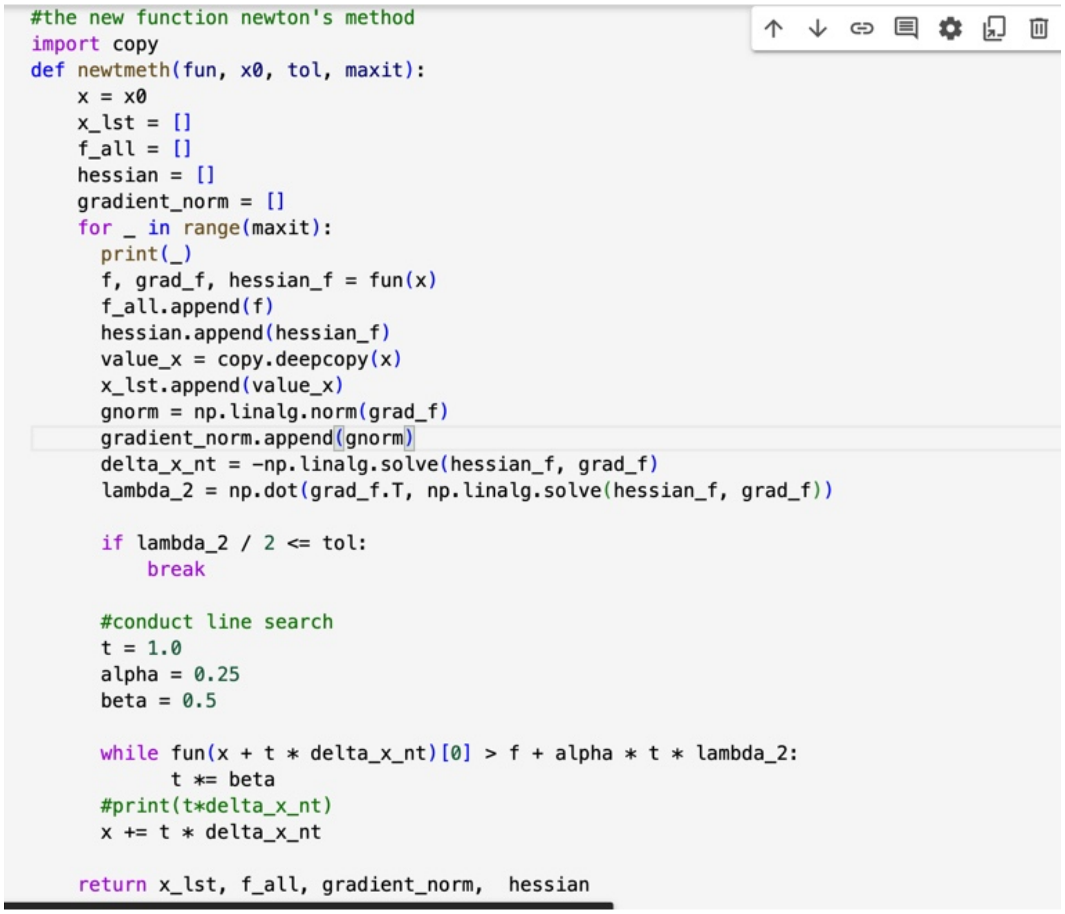
$$\frac{f(x_0) - p^*}{\gamma} = 2.43265564 \times 10^{10}$$

To calculate the lower bound of iteration steps, we suppose norm of gradient is less than η all the time, and the number of steps can be calculated as, if we take tolerance $\varepsilon = 1 \times 10^{-8}$,

$$log_2\, log_2\, \frac{2m^3}{L^2}\frac{1}{\varepsilon} = 3.9205147125541933$$

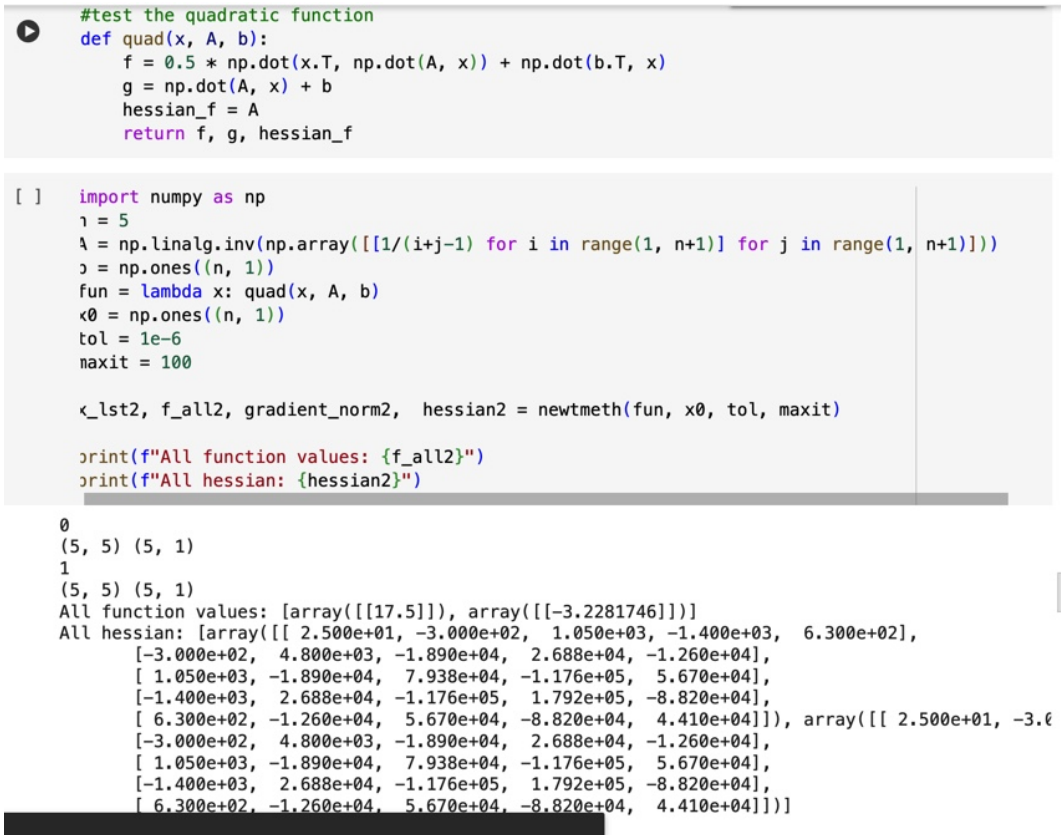
And as we can see, our result lies within these bounds.

Remark: some numerical results on the screen shots might not be correct because it's from an old version of my code. Please refer to the code I submitted for the correct version.



```python
#the new function newton's method
import copy
def newtmeth(fun, x0, tol, maxit):
    x = x0
    x_lst = []
    f_all = []
    hessian = []
    gradient_norm = []
    for _ in range(maxit):
      print(_)
      f, grad_f, hessian_f = fun(x)
      f_all.append(f)
      hessian.append(hessian_f)
      value_x = copy.deepcopy(x)
      x_lst.append(value_x)
      gnorm = np.linalg.norm(grad_f)
      gradient_norm.append(gnorm)
      delta_x_nt = -np.linalg.solve(hessian_f, grad_f)
      lambda_2 = np.dot(grad_f.T, np.linalg.solve(hessian_f, grad_f))

      if lambda_2 / 2 <= tol:
          break

      #conduct line search
      t = 1.0
      alpha = 0.25
      beta = 0.5

      while fun(x + t * delta_x_nt)[0] > f + alpha * t * lambda_2:
            t *= beta
      #print(t*delta_x_nt)
      x += t * delta_x_nt

    return x_lst, f_all, gradient_norm,  hessian
```

Figure 9: Code for Newton's Method

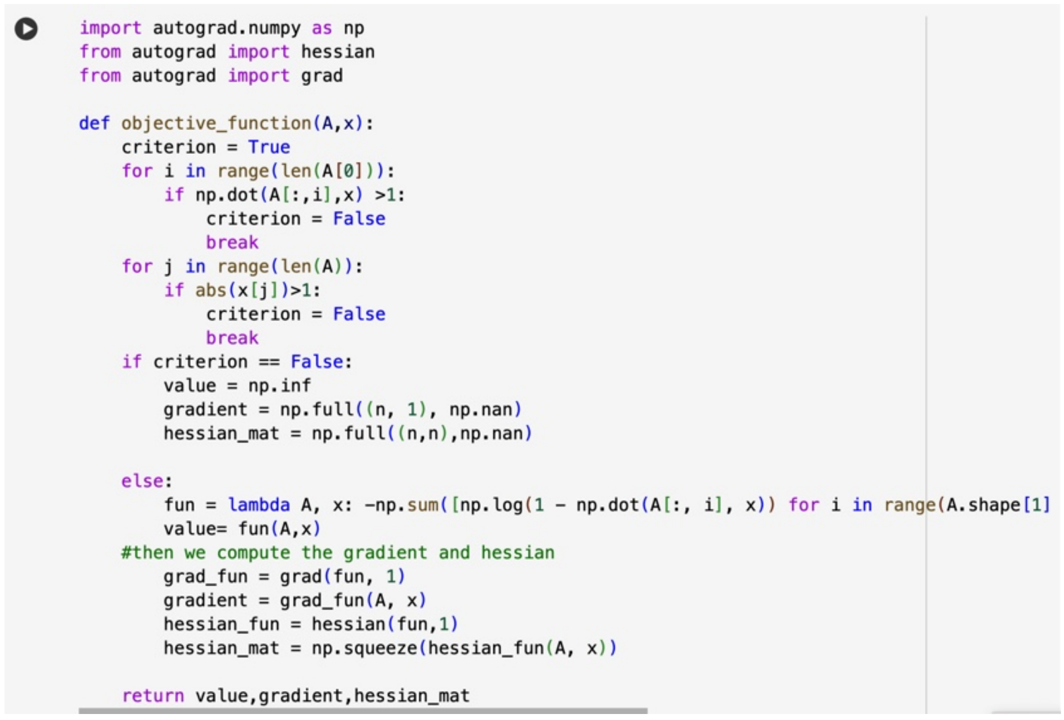


```
#test the quadratic function
def quad(x, A, b):
    f = 0.5 * np.dot(x.T, np.dot(A, x)) + np.dot(b.T, x)
    g = np.dot(A, x) + b
    hessian_f = A
    return f, g, hessian_f
```

```
import numpy as np
ı = 5
\ = np.linalg.inv(np.array([[1/(i+j-1) for i in range(1, n+1)] for j in range(1, n+1)]))
ɔ = np.ones((n, 1))
fun = lambda x: quad(x, A, b)
ĸ0 = np.ones((n, 1))
tol = 1e-6
ıaxit = 100

ĸ_lst2, f_all2, gradient_norm2,  hessian2 = newtmeth(fun, x0, tol, maxit)

ɔrint(f"All function values: {f_all2}")
ɔrint(f"All hessian: {hessian2}")
```

```
0
(5, 5) (5, 1)
1
(5, 5) (5, 1)
All function values: [array([[17.5]]), array([[-3.2281746]])]
All hessian: [array([[ 2.500e+01, -3.000e+02,  1.050e+03, -1.400e+03,  6.300e+02],
       [-3.000e+02,  4.800e+03, -1.890e+04,  2.688e+04, -1.260e+04],
       [ 1.050e+03, -1.890e+04,  7.938e+04, -1.176e+05,  5.670e+04],
       [-1.400e+03,  2.688e+04, -1.176e+05,  1.792e+05, -8.820e+04],
       [ 6.300e+02, -1.260e+04,  5.670e+04, -8.820e+04,  4.410e+04]]), array([[ 2.500e+01, -3.0
       [-3.000e+02,  4.800e+03, -1.890e+04,  2.688e+04, -1.260e+04],
       [ 1.050e+03, -1.890e+04,  7.938e+04, -1.176e+05,  5.670e+04],
       [-1.400e+03,  2.688e+04, -1.176e+05,  1.792e+05, -8.820e+04],
       [ 6.300e+02, -1.260e+04,  5.670e+04, -8.820e+04,  4.410e+04]])]
```

Figure 10: The simple quadratic function test case for Newton's method



```
import autograd.numpy as np
from autograd import hessian
from autograd import grad

def objective_function(A,x):
    criterion = True
    for i in range(len(A[0])):
        if np.dot(A[:,i],x) >1:
            criterion = False
            break
    for j in range(len(A)):
        if abs(x[j])>1:
            criterion = False
            break
    if criterion == False:
        value = np.inf
        gradient = np.full((n, 1), np.nan)
        hessian_mat = np.full((n,n),np.nan)

    else:
        fun = lambda A, x: -np.sum([np.log(1 - np.dot(A[:, i], x)) for i in range(A.shape[1]
        value= fun(A,x)
    #then we compute the gradient and hessian
        grad_fun = grad(fun, 1)
        gradient = grad_fun(A, x)
        hessian_fun = hessian(fun,1)
        hessian_mat = np.squeeze(hessian_fun(A, x))

    return value,gradient,hessian_mat
```

Figure 11: Objective function for the more interesting test case 2 for Newton's method

```python
import matplotlib.pyplot as plt

plt.figure()
p_star3 = f_all3[-1][0]
values_to_plot3 = f_all3 - p_star3
plt.semilogy(values_to_plot, 'x', color = 'blue',  label='Grad Method')
plt.semilogy(values_to_plot3,'x' ,color='red', label='Newt Method')
plt.title('Log Plot of f(x(k)) - p*')  # 图表标题
plt.show()
```

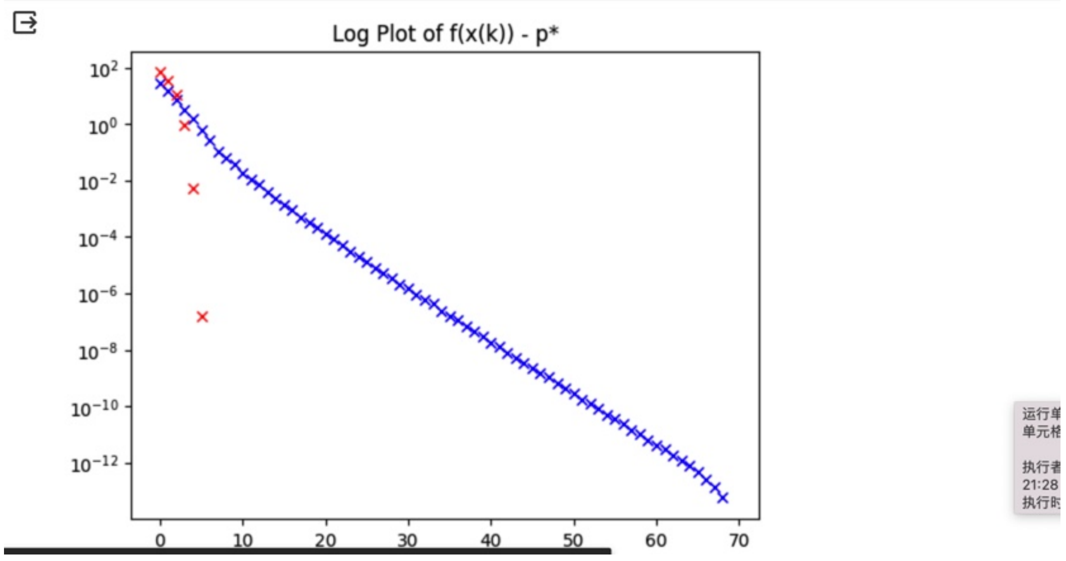


Figure 12: Plot of distance to optimal value compared to the original gradient method

```python
plt.semilogy(gnorm_all, 'x',color = 'blue',  label='Grad Method')
plt.semilogy(gnorm_all3, 'x',color='red', label='Newt Method')
plt.title('Log Plot of grad norm')
plt.show()
```

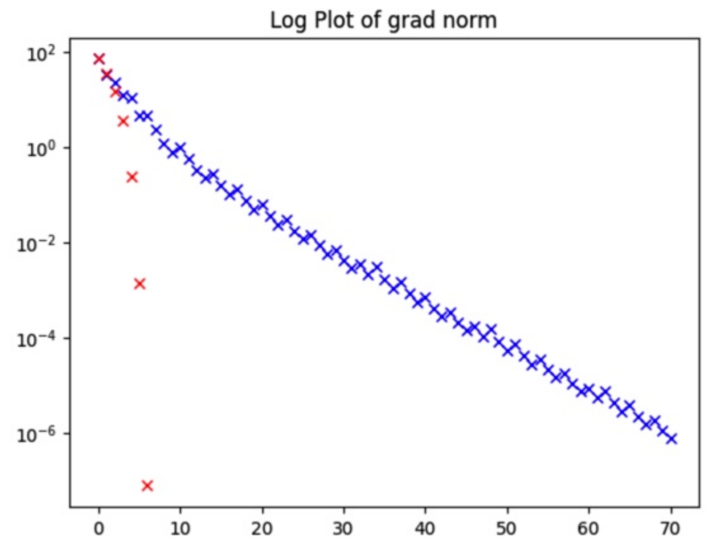
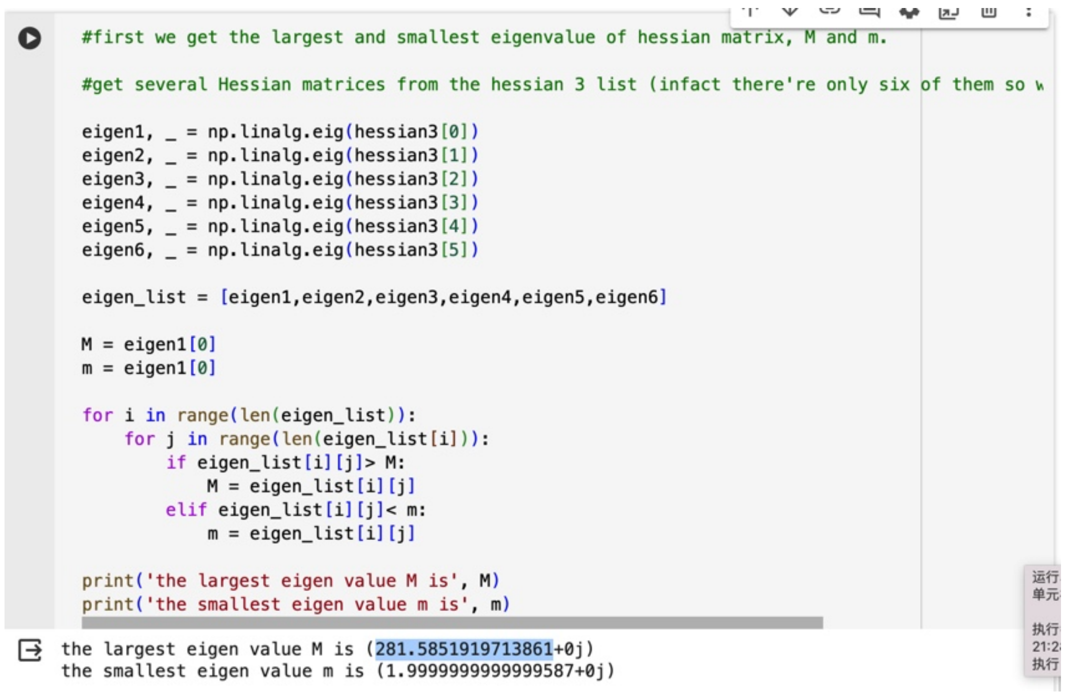


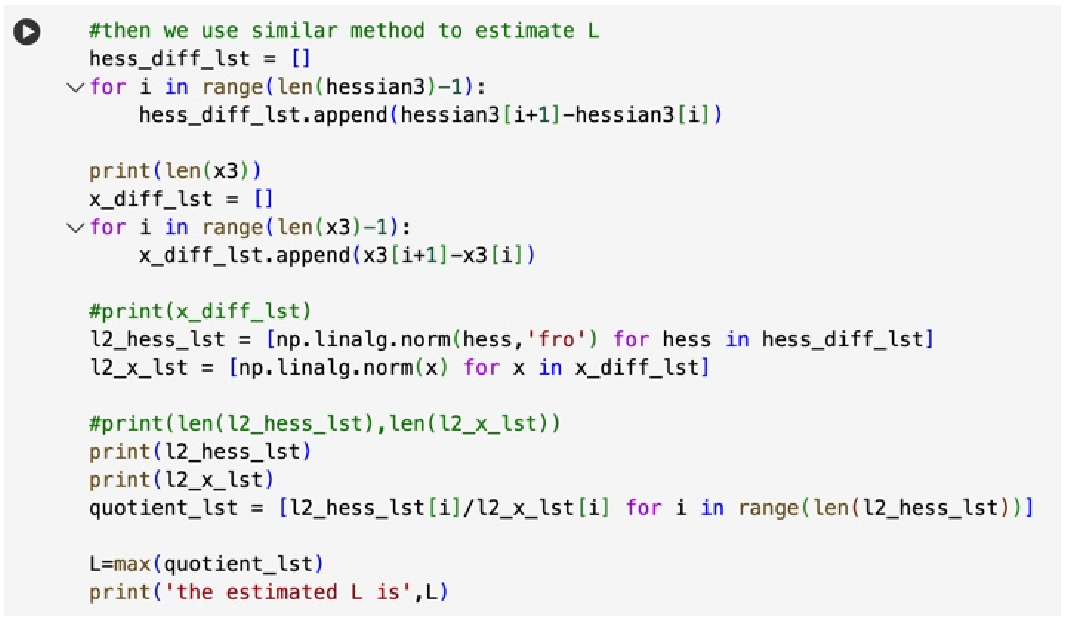
Figure 13: Plot of gradient norm compared to the original gradient method



```
#first we get the largest and smallest eigenvalue of hessian matrix, M and m.

#get several Hessian matrices from the hessian 3 list (infact there're only six of them so w

eigen1, _ = np.linalg.eig(hessian3[0])
eigen2, _ = np.linalg.eig(hessian3[1])
eigen3, _ = np.linalg.eig(hessian3[2])
eigen4, _ = np.linalg.eig(hessian3[3])
eigen5, _ = np.linalg.eig(hessian3[4])
eigen6, _ = np.linalg.eig(hessian3[5])

eigen_list = [eigen1,eigen2,eigen3,eigen4,eigen5,eigen6]

M = eigen1[0]
m = eigen1[0]

for i in range(len(eigen_list)):
    for j in range(len(eigen_list[i])):
        if eigen_list[i][j]> M:
            M = eigen_list[i][j]
        elif eigen_list[i][j]< m:
            m = eigen_list[i][j]

print('the largest eigen value M is', M)
print('the smallest eigen value m is', m)
```

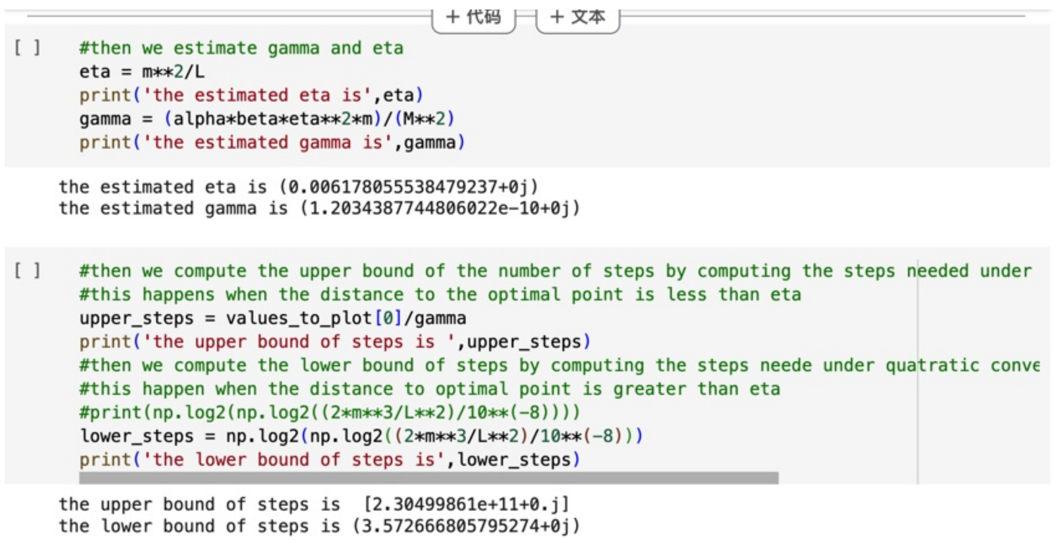
```
the largest eigen value M is (281.5851919713861+0j)
the smallest eigen value m is (1.9999999999995587+0j)
```

Figure 14: The calculation of M and m



```
#then we use similar method to estimate L
hess_diff_lst = []
for i in range(len(hessian3)-1):
    hess_diff_lst.append(hessian3[i+1]-hessian3[i])

print(len(x3))
x_diff_lst = []
for i in range(len(x3)-1):
    x_diff_lst.append(x3[i+1]-x3[i])

#print(x_diff_lst)
l2_hess_lst = [np.linalg.norm(hess,'fro') for hess in hess_diff_lst]
l2_x_lst = [np.linalg.norm(x) for x in x_diff_lst]

#print(len(l2_hess_lst),len(l2_x_lst))
print(l2_hess_lst)
print(l2_x_lst)
quotient_lst = [l2_hess_lst[i]/l2_x_lst[i] for i in range(len(l2_hess_lst))]

L=max(quotient_lst)
print('the estimated L is',L)
```

Figure 15: The calculation of L



```
#then we estimate gamma and eta
eta = m**2/L
print('the estimated eta is',eta)
gamma = (alpha*beta*eta**2*m)/(M**2)
print('the estimated gamma is',gamma)

the estimated eta is (0.006178055538479237+0j)
the estimated gamma is (1.2034387744806022e-10+0j)
```

```
#then we compute the upper bound of the number of steps by computing the steps needed under
#this happens when the distance to the optimal point is less than eta
upper_steps = values_to_plot[0]/gamma
print('the upper bound of steps is ',upper_steps)
#then we compute the lower bound of steps by computing the steps neede under quatratic conve
#this happen when the distance to optimal point is greater than eta
#print(np.log2(np.log2((2*m**3/L**2)/10**(-8))))
lower_steps = np.log2(np.log2((2*m**3/L**2)/10**(-8)))
print('the lower bound of steps is',lower_steps)

the upper bound of steps is  [2.30499861e+11+0.j]
the lower bound of steps is (3.572666805795274+0j)
```

Figure 16: The estimation of η, γ and the final lower and upper bound of iteration step numbers