In [1]:
```
import numpy as np
import time
```

In [2]:
```
# Function rosembrock to be optimized
def rosembrock_func(x):
    return 100 * (x[1][0] - x[0][0] ** 2) ** 2 + (1 - x[0][0]) ** 2
```

In [3]:
```
# Gradient of the function rosembrock to be optimized: delta f(x) = [df/dx[0],
def grad_func_rosembrock(x):
    return np.array([-400 * x[0][0] * (x[1][0] - x[0][0] ** 2) - 2 * (1 - x[0]
```

In [4]:
```
x0 = np.array([[1.2], [1.2]])
```

In [5]:
```
# Hessian of the function rosembrock to be optimized: delta^2 f(x) = [[d^2f/dx
def hessian_rosembrock(x):
    return np.array([[1200 * x[0][0] ** 2 - 400 * x[1][0] + 2, -400 * x[0][0]]
```

In [6]:
```
print(x0)
print(rosembrock_func(x0))
print(grad_func_rosembrock(x0))
print(hessian_rosembrock(x0))
```

```
[[1.2]
 [1.2]]
5.8
[[115.6]
 [-48. ]]
[[1250. -480.]
 [-480.  200.]]
```

In [7]:
```
# Backtracking line search
def backtracking_Rosembrock(x0, f, grad_f, descent_direction, alpha=0.3, beta=
    x = x0
    t = 1
    while f(x + t* descent_direction) > f(x) + alpha * t * np.dot(grad_f(x).T,
        t *= beta
    return t
```

1 Gradient descent method with back tracking linesearch

In [8]:
```
# Gradient descent method with Δx = - ∇f(x)
def gradient_descent(x0, f, grad_f, line_search, epsilon=1e-4, max_iter=1000):
    x = x0
    for i in range(max_iter+1):
        print(f"Epoch {i}: x = {x}, f(x) = {f(x)}")
        if np.linalg.norm(grad_f(x)) < epsilon:
            break
        descent_direction = -grad_f(x)
        t = line_search(x, f, grad_f, descent_direction)
        x += t * descent_direction
    return f(x)
```

In [9]:
```
print("First starting point: [1.2, 1.2]\n")
x0 = np.array([[1.2], [1.2]])
gradient_descent(x0, rosembrock_func, grad_func_rosembrock, backtracking_Rosem
```

```
Epoch 992: x = [[1.06004643]
 [1.12380551]], f(x) = 0.0036067206130375646
Epoch 993: x = [[1.05990196]
 [1.12376409]], f(x) = 0.003602077847273066
Epoch 994: x = [[1.05997523]
 [1.12362021]], f(x) = 0.0035975569375998615
Epoch 995: x = [[1.05983733]
 [1.1235977 ]], f(x) = 0.0035922392206629546
Epoch 996: x = [[1.05989908]
 [1.12343206]], f(x) = 0.003588111454070411
Epoch 997: x = [[1.05974388]
 [1.12341782]], f(x) = 0.003582344029701538
Epoch 998: x = [[1.05980854]
 [1.12327827]], f(x) = 0.0035777688896228114
Epoch 999: x = [[1.05964616]
 [1.12324572]], f(x) = 0.0035733254283960196
Epoch 1000: x = [[1.05972113]
 [1.12312324]], f(x) = 0.003567921126687234
```

Out[9]: 0.003563270857269817

In [10]:
```
print("Second starting point: [-1.2, 1]\n")
x0 = np.array([[-1.2], [1]])
gradient_descent(x0, rosembrock_func, grad_func_rosembrock, backtracking_Rosem
```

```
Epoch 992: x = [[0.93118282]
 [0.86662527]], f(x) = 0.004758478539458868
Epoch 993: x = [[0.93108676]
 [0.86685554]], f(x) = 0.0047494835111339334
Epoch 994: x = [[0.93130508]
 [0.86688146]], f(x) = 0.0047390342761011355
Epoch 995: x = [[0.93123404]
 [0.86709795]], f(x) = 0.004729735314962753
Epoch 996: x = [[0.93147751]
 [0.86714577]], f(x) = 0.004720792244680951
Epoch 997: x = [[0.93137894]
 [0.86734097]], f(x) = 0.0047104313506956255
Epoch 998: x = [[0.93159749]
 [0.86740178]], f(x) = 0.004701191053918959
Epoch 999: x = [[0.93150291]
 [0.86763008]], f(x) = 0.004692308143537733
Epoch 1000: x = [[0.93171918]
 [0.86765623]], f(x) = 0.004682019824241827
```

Out[10]:
```
0.0046728427256420004
```

2 Newton method with back tracking line search

In [11]:
```
import numpy as np

x = np.array([[2, 3],
              [4, 5]])
y = np.linalg.inv(x)

result = np.dot(x, y)
print(result)
```

```
[[1. 0.]
 [0. 1.]]
```

In [12]:
```
# Newton method with Δx = - (∇²f(x))⁻¹ ∇f(x)
def newton_method(x0, f, grad_f, hessian, line_search, epsilon=1e-4, max_iter=
    x = x0
    for i in range(max_iter+1):
        print(f"Epoch {i}: x = {x}, f(x) = {f(x)}")
        # Stopping criteria
        # norm of delta < epsilon
        if np.linalg.norm(grad_f(x)) < epsilon:
            break
        # lambda2 / 2 <= epsilon with lambda2 = ∇f(x)ᵀ (∇²f(x))⁻¹ ∇f(x)
        lambda2 = np.dot(grad_f(x).T, np.dot(np.linalg.inv(hessian(x)), grad_f
        if lambda2 / 2 <= epsilon:
            break
        descent_direction = -np.dot(np.linalg.inv(hessian(x)), grad_f(x))

        t = line_search(x, f, grad_f, descent_direction)
        x += t * descent_direction
    return f(x)
```

In [13]:
```
print("First starting point: [1.2, 1.2]\n")
x0 = np.array([[1.2], [1.2]])
newton_method(x0, rosembrock_func, grad_func_rosembrock, hessian_rosembrock, b
```

```
First starting point: [1.2, 1.2]

Epoch 0: x = [[1.2]
 [1.2]], f(x) = 5.8
Epoch 1: x = [[1.19591837]
 [1.43020408]], f(x) = 0.038384034418534184
Epoch 2: x = [[1.09594128]
 [1.19108374]], f(x) = 0.01921182605186693
Epoch 3: x = [[1.06396842]
 [1.13100653]], f(x) = 0.004196460661754682
Epoch 4: x = [[1.01085848]
 [1.01901419]], f(x) = 0.0009135219903405465
Epoch 5: x = [[1.00391631]
 [1.00779976]], f(x) = 1.55697292474431e-05
```

Out[13]:
```
1.55697292474431e-05
```

In [14]:
```
print("Second starting point: [-1.2, 1]\n")
x0 = np.array([[-1.2], [1]])
newton_method(x0, rosembrock_func, grad_func_rosembrock, hessian_rosembrock, b
```

```
Second starting point: [-1.2, 1]

Epoch 0: x = [[-1.2]
 [ 1. ]], f(x) = 24.199999999999996
Epoch 1: x = [[-1.1752809 ]
 [ 1.38067416]], f(x) = 4.731884325266608
Epoch 2: x = [[-0.91511382]
 [ 0.76921738]], f(x) = 4.13300226320981
Epoch 3: x = [[-0.7843285 ]
 [ 0.59806639]], f(x) = 3.2130856128676224
Epoch 4: x = [[-0.52602031]
 [ 0.20381651]], f(x) = 2.8598998169973844
Epoch 5: x = [[-0.42804883]
 [ 0.1736274 ]], f(x) = 2.048536419892932
Epoch 6: x = [[-0.22770892]
 [ 0.00604837]], f(x) = 1.7170605076532184
Epoch 7: x = [[-0.10687852]
 [-0.00317697]], f(x) = 1.2464960184633609
Epoch 8: x = [[ 0.11901522]
 [-0.03978336]], f(x) = 1.0671726284538807
Epoch 9: x = [[0.19374083]
 [0.03195159]], f(x) = 0.6531718577280331
Epoch 10: x = [[0.38875591]
 [0.11037533]], f(x) = 0.5397231243685873
Epoch 11: x = [[0.45555003]
 [0.20306437]], f(x) = 0.29841622839711246
Epoch 12: x = [[0.60286268]
 [0.33956521]], f(x) = 0.21473492952674286
Epoch 13: x = [[0.67162341]
 [0.44634997]], f(x) = 0.11006661799842264
Epoch 14: x = [[0.80664616]
 [0.63150127]], f(x) = 0.07416048107314221
Epoch 15: x = [[0.84663372]
 [0.71518965]], f(x) = 0.023776898332548767
Epoch 16: x = [[0.939597  ]
 [0.87388054]], f(x) = 0.011680217404296728
Epoch 17: x = [[0.96122825]
 [0.92349184]], f(x) = 0.0015251423987508415
Epoch 18: x = [[0.99668214]
 [0.99211832]], f(x) = 0.00016900762708l6276
Epoch 19: x = [[0.99933347]
 [0.99866035]], f(x) = 4.4920548861222697e-07
```

Out[14]: 4.4920548861222697e-07

3 Compare the convergence speed of the two algorithms.

We can see that the first method - gradient descent costs us 1000 iterations or more to get the

x = [[1.2] [1.2]]
Epoch 1000: x = [[1.05972113] [1.12312324]], f(x) = 0.003567921126687234

x = [[-1,2] [1]]
Epoch 1000: x = [[0.93171918] [0.86765623]], f(x) = 0.004682019824241827

Otherwise, the second method - newton_method - is more efficient than the first one - gradient_descent - because it converges in less iterations and get more efficient answer as you can see below:

x = [[1.2] [1.2]]
Epoch 5: x = [[1.00391631] [1.00779976]], f(x) = 1.55697292474431e-05

x = [[-1,2] [1]]
Epoch 19: x = [[0.99933347] [0.99866035]], f(x) = 4.4920548861222697e-07