Optimization Algorithms: Theory and Software Implementation

Prof. Thirumulanathan D

Department of Mathematics

Institute of IIT Kanpur

Lecture: 34

Hello everyone. This is the fourth lecture in week 7. Recall that in the last three lectures we learned about the quadratic penalty method, an algorithm for finding critical points in constrained optimization problems, specifically non-linear programming problems.

We discussed that the penalty function for a constrained optimization problem takes a specific form: the objective function plus γ times the squared equality constraints. For problems with only inequality constraints, the quadratic penalty function is defined as:

$Q(x, \gamma) = f(x) + \gamma \sum_{i=1}^{p} [\max(0, g_i(x))]^2$

This formulation ensures that when $g_i(x) \leq 0$, the penalty is 0, and when $g_i(x) > 0$, the penalty is positive. An alternative notation is:

$Q(x, \gamma) = f(x) + \gamma \sum_{i=1}^{p} [g_i(x)_+]^2$

where the plus subscript denotes the positive part ($\max(0, g_i(x))$).

The gradient of this penalty function is:

$\nabla Q(x, \gamma) = \nabla f(x) + 2\gamma \sum_{i=1}^{p} \max(0, g_i(x)) \nabla g_i(x)$

This gradient is well-defined and continuous. However, the Hessian presents challenges. At points where $g_i(x) = 0$, the second derivative has different left and right limits, making $Q(x, \gamma)$ not twice differentiable. Despite this, when using Newton's method, the approximation:

$\nabla^2 Q(x, \gamma) \approx \nabla^2 f(x) + 2\gamma \sum_{i=1}^{p} [g_i(x) \nabla^2 g_i(x) + \nabla g_i(x) \nabla g_i(x)^T]$ (Note here $\nabla^2 = \nabla_2 =$Hessian)

often yields correct results in practice.

Now, let's consider an example:

Minimize $f(x) = \frac{1}{2}(x_1 - 3)^2 + \frac{1}{2}(x_2 - 2)^2$

Subject to:

$x_1 \geq x_2$

$x_1 + x_2 \leq 1$

$x_2 \geq 0$

To express these in standard form ($g_i(x) \leq 0$):

$g_1(x) = x_2 - x_1 \leq 0$

$g_2(x) = x_1 + x_2 - 1 \leq 0$

$g_3(x) = -x_2 \leq 0$

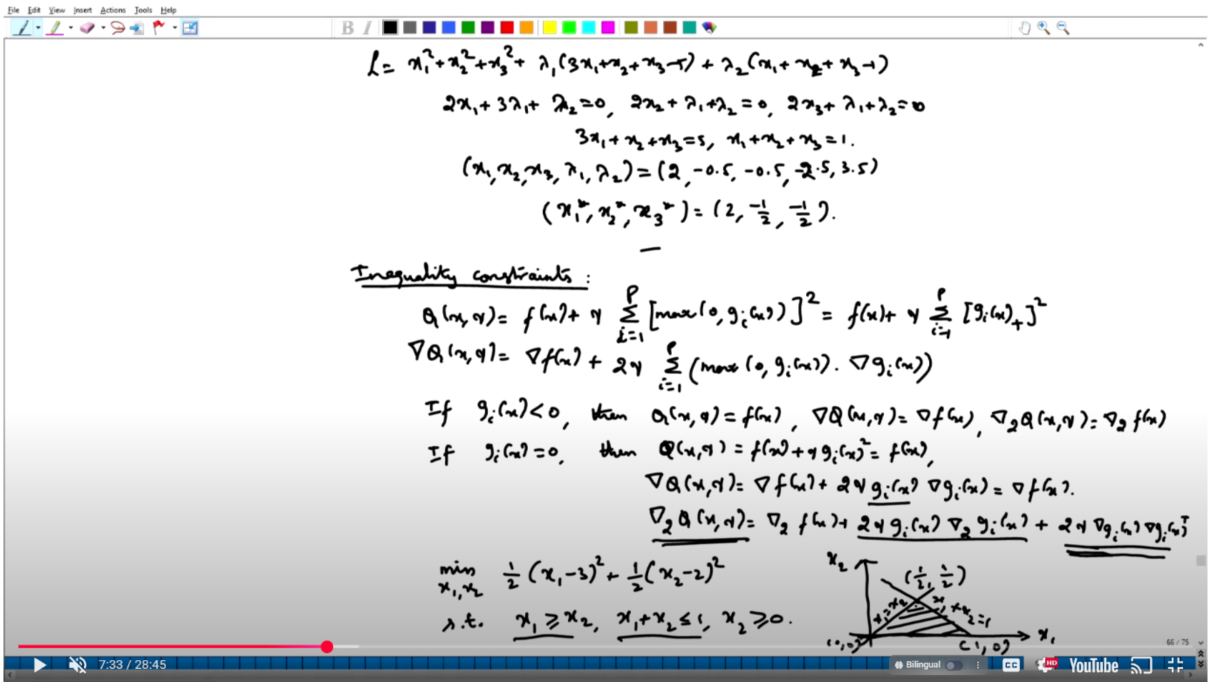
Geometrically, the constraint set forms a triangle with vertices at:

(0, 0) - intersection of $x_1 = x_2$ and $x_2 = 0$

(1, 0) - intersection of $x_1 + x_2 = 1$ and $x_2 = 0$

(½, ½) - intersection of $x_1 = x_2$ and $x_1 + x_2 = 1$

(Refer Slide Time 7:33)



$$\mathcal{L} = x_1^2 + x_2^2 + x_3^2 + \lambda_1(3x_1 + x_2 + x_3 - 5) + \lambda_2(x_1 + x_2 + x_3 - 1)$$

$$2x_1 + 3\lambda_1 + \lambda_2 = 0, \quad 2x_2 + \lambda_1 + \lambda_2 = 0, \quad 2x_3 + \lambda_1 + \lambda_2 = 0$$

$$3x_1 + x_2 + x_3 = 5, \quad x_1 + x_2 + x_3 = 1.$$

$$(x_1, x_2, x_3, \lambda_1, \lambda_2) = (2, -0.5, -0.5, -2.5, 3.5)$$

$$(x_1^*, x_2^*, x_3^*) = (2, -\tfrac{1}{2}, -\tfrac{1}{2}).$$

<u>Inequality constraints</u>:

$$Q(x,\eta) = f(x) + \eta \sum_{i=1}^{p} [\max(0, g_i(x))]^2 = f(x) + \eta \sum_{i=1}^{p} [g_i(x)_+]^2$$

$$\nabla Q(x,\eta) = \nabla f(x) + 2\eta \sum_{i=1}^{p} (\max(0, g_i(x)) \cdot \nabla g_i(x))$$

If $g_i(x) < 0$, then $Q(x,\eta) = f(x)$, $\nabla Q(x,\eta) = \nabla f(x)$, $\nabla_2 Q(x,\eta) = \nabla_2 f(x)$

If $g_i(x) = 0$, then $Q(x,\eta) = f(x) + \eta g_i(x)^2 = f(x)$,

$$\nabla Q(x,\eta) = \nabla f(x) + 2\eta \underline{g_i(x)} \nabla g_i(x) = \nabla f(x).$$

$$\underline{\nabla_2 Q(x,\eta)} = \nabla_2 f(x) + \underline{2\eta g_i(x) \nabla_2 g_i(x)} + \underline{2\eta \nabla g_i(x) \nabla g_i(x)^T}$$

$$\min_{x_1, x_2} \frac{1}{2}(x_1 - 3)^2 + \frac{1}{2}(x_2 - 2)^2$$

$$\text{s.t.} \quad \underline{x_1 \geq x_2}, \quad \underline{x_1 + x_2 \leq 1}, \quad x_2 \geq 0.$$

The objective function represents the squared distance from the point (3, 2). Visual inspection shows that (1, 0) is the closest point in the feasible region to (3, 2), making it the optimal solution $x^* = (1, 0)$.

We are solving the optimization problem using the quadratic penalty method for inequality constraints. The problem is:

Minimize $f(x) = 1/2*(x_1 - 3)^2 + 1/2*(x_2 - 2)^2$

Subject to:

$x_1 \geq x_2 \rightarrow g_1(x) = x_2 - x_1 \leq 0$

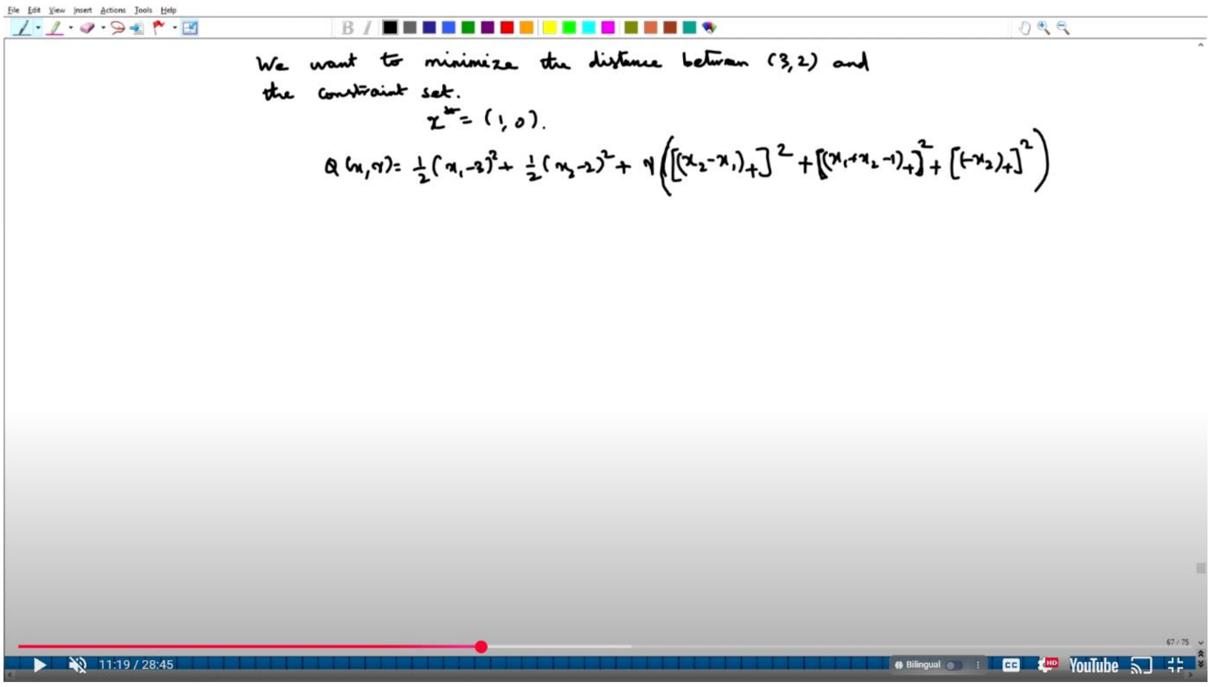
$x_1 + x_2 \leq 1 \rightarrow g_2(x) = x_1 + x_2 - 1 \leq 0$

$x_2 \geq 0 \rightarrow g_3(x) = -x_2 \leq 0$

The quadratic penalty function is defined as:

$Q(x, \gamma) = f(x) + \gamma * [\max(0, g_1(x))^2 + \max(0, g_2(x))^2 + \max(0, g_3(x))^2]$

(Refer Slide Time)



We want to minimize the distance between (3, 2) and the constraint set.

$$x^* = (1, 0).$$

$$Q(x,\gamma) = \frac{1}{2}(x_1-3)^2 + \frac{1}{2}(x_2-2)^2 + \gamma\left([(x_2-x_1)_+]^2 + [(x_1+x_2-1)_+]^2 + [(-x_2)_+]^2\right)$$

We begin by writing the code for the quadratic penalty method applied to inequality constraints. We start with the code for equality constraints and modify it accordingly.

The objective function is defined as: $f(x) = 0.5 * (x_0 - 3)^2 + 0.5 * (x_1 - 2)^2$

The inequality constraints are given by: $g(x) = [x_1 - x_0, x_0 + x_1 - 1, -x_1]$

The Jacobian matrix of the constraints, denoted $\nabla g(x)$, is: $\nabla g(x) = [[-1, 1], [1, 1], [0, -1]]$

For the penalty function, we only penalize when a constraint is violated, i.e., when $g_i(x) > 0$. This is implemented using the element-wise operation: $(g(x) > 0) * g(x)$. This results in a vector where each element is $g_i(x)$ if $g_i(x) > 0$, and 0 otherwise.

For example, if $g(x) = [1, -2, 4]$, then $(g(x) > 0)$ yields [True, False, True], which is treated as [1, 0, 1] in numerical operations. The multiplication $(g(x) > 0) * g(x)$ then gives [1, 0, 4].

The quadratic penalty function for inequalities is:

$Q(x, \gamma) = f(x) + \gamma * \text{sum}([(g(x) > 0) * g(x)]^2)$

The gradient of Q is computed as: $\nabla Q(x, \gamma) = \nabla f(x) + 2\gamma * [(g(x) > 0) * g(x)] @ \nabla g(x)$

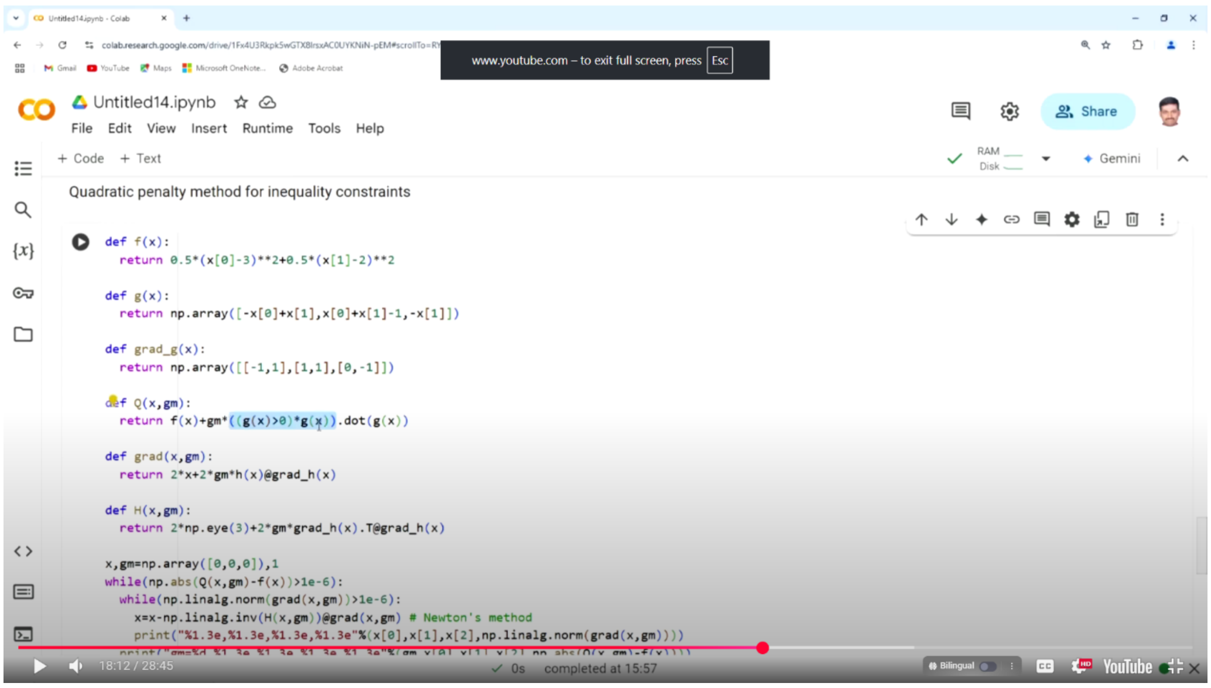
The Hessian of Q is: $\nabla^2 Q(x, \gamma) = \nabla^2 f(x) + 2\gamma * \nabla g(x)^T @ \text{diag}(g(x) > 0) @ \nabla g(x)$.

Here, $\text{diag}(g(x) > 0)$ is a diagonal matrix with ones where $g_i(x) > 0$ and zeros otherwise.

For our specific problem, $\nabla f(x) = [x_0 - 3, x_1 - 2]$ and $\nabla^2 f(x)$ is the 2x2 identity matrix.

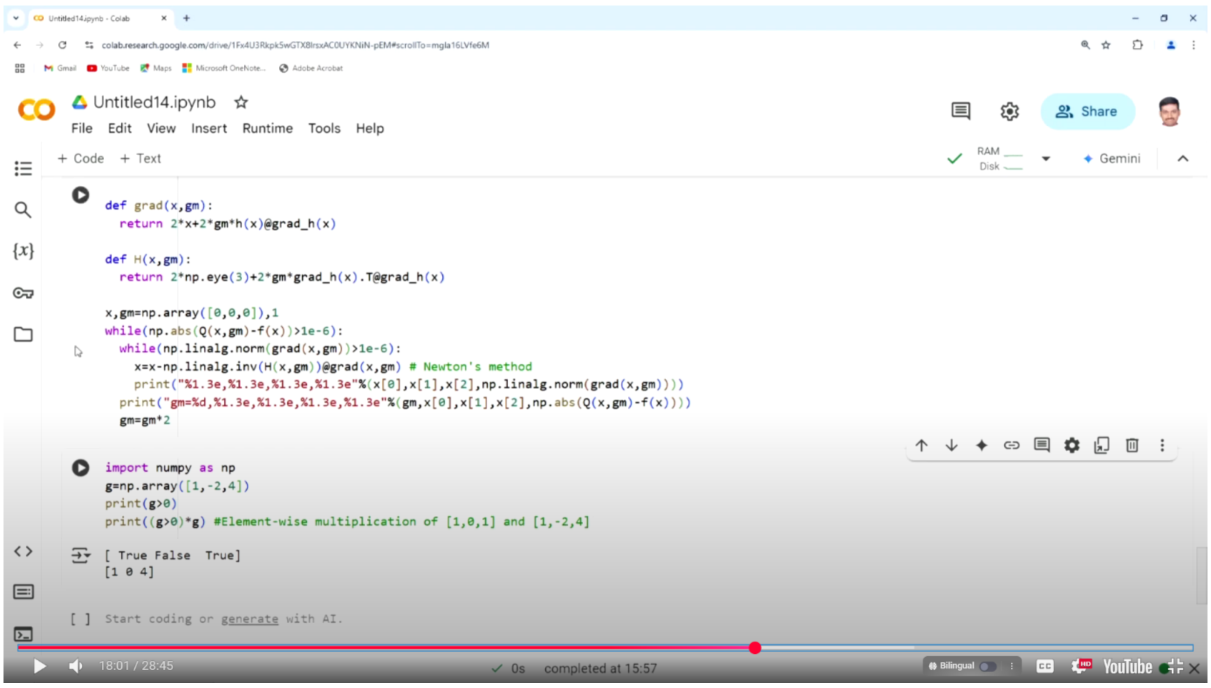
We initialize $x^0 = [3, 2]$ and increment γ iteratively. For each γ, we minimize $Q(x, \gamma)$ using Newton's method. With γ set to 2^{21} (approximately 2 million), the solution [1, 0] is obtained after 22 iterations, using a stopping tolerance of 1e-6.

(Refer Slides Time 21:00)



Quadratic penalty method for inequality constraints

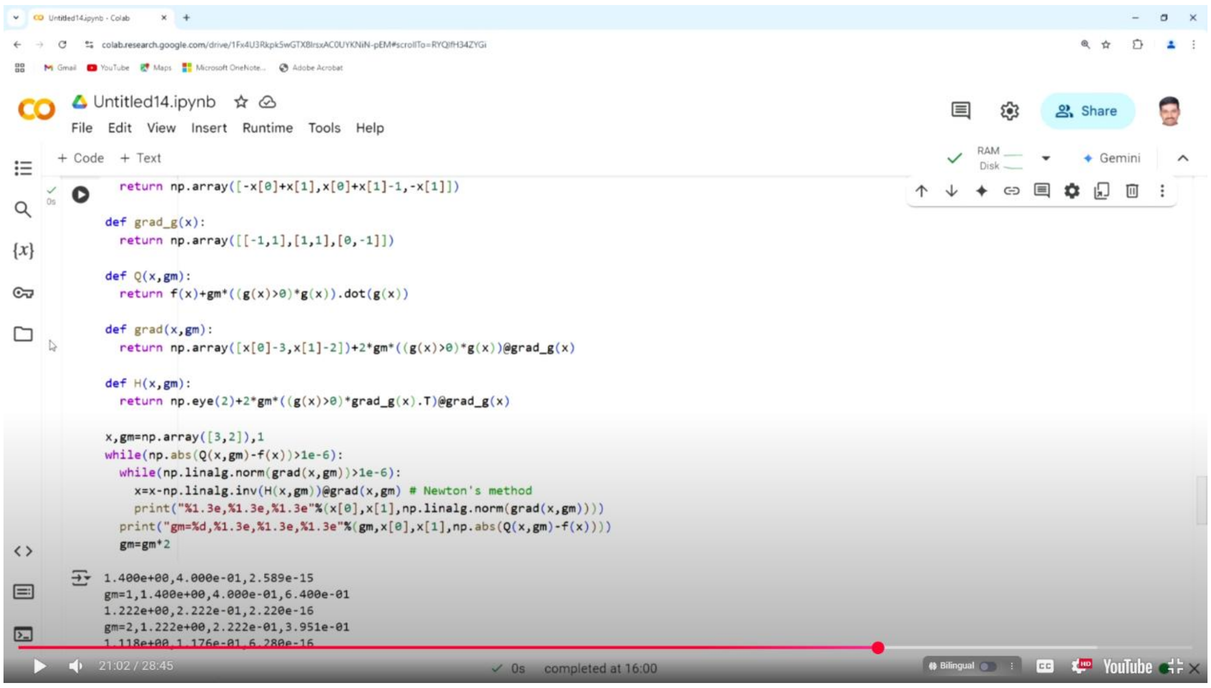
```
def f(x):
  return 0.5*(x[0]-3)**2+0.5*(x[1]-2)**2

def g(x):
  return np.array([-x[0]+x[1],x[0]+x[1]-1,-x[1]])

def grad_g(x):
  return np.array([[-1,1],[1,1],[0,-1]])

def Q(x,gm):
  return f(x)+gm*((g(x)>0)*g(x)).dot(g(x))

def grad(x,gm):
  return 2*x+2*gm*h(x)@grad_h(x)

def H(x,gm):
  return 2*np.eye(3)+2*gm*grad_h(x).T@grad_h(x)

x,gm=np.array([0,0,0]),1
while(np.abs(Q(x,gm)-f(x))>1e-6):
  while(np.linalg.norm(grad(x,gm))>1e-6):
    x=x-np.linalg.inv(H(x,gm))@grad(x,gm) # Newton's method
    print("%1.3e,%1.3e,%1.3e,%1.3e"%(x[0],x[1],x[2],np.linalg.norm(grad(x,gm))))
```



```
def grad(x,gm):
  return 2*x+2*gm*h(x)@grad_h(x)

def H(x,gm):
  return 2*np.eye(3)+2*gm*grad_h(x).T@grad_h(x)

x,gm=np.array([0,0,0]),1
while(np.abs(Q(x,gm)-f(x))>1e-6):
  while(np.linalg.norm(grad(x,gm))>1e-6):
    x=x-np.linalg.inv(H(x,gm))@grad(x,gm) # Newton's method
    print("%1.3e,%1.3e,%1.3e,%1.3e"%(x[0],x[1],x[2],np.linalg.norm(grad(x,gm))))
  print("gm=%d,%1.3e,%1.3e,%1.3e,%1.3e"%(gm,x[0],x[1],x[2],np.abs(Q(x,gm)-f(x))))
  gm=gm*2
```

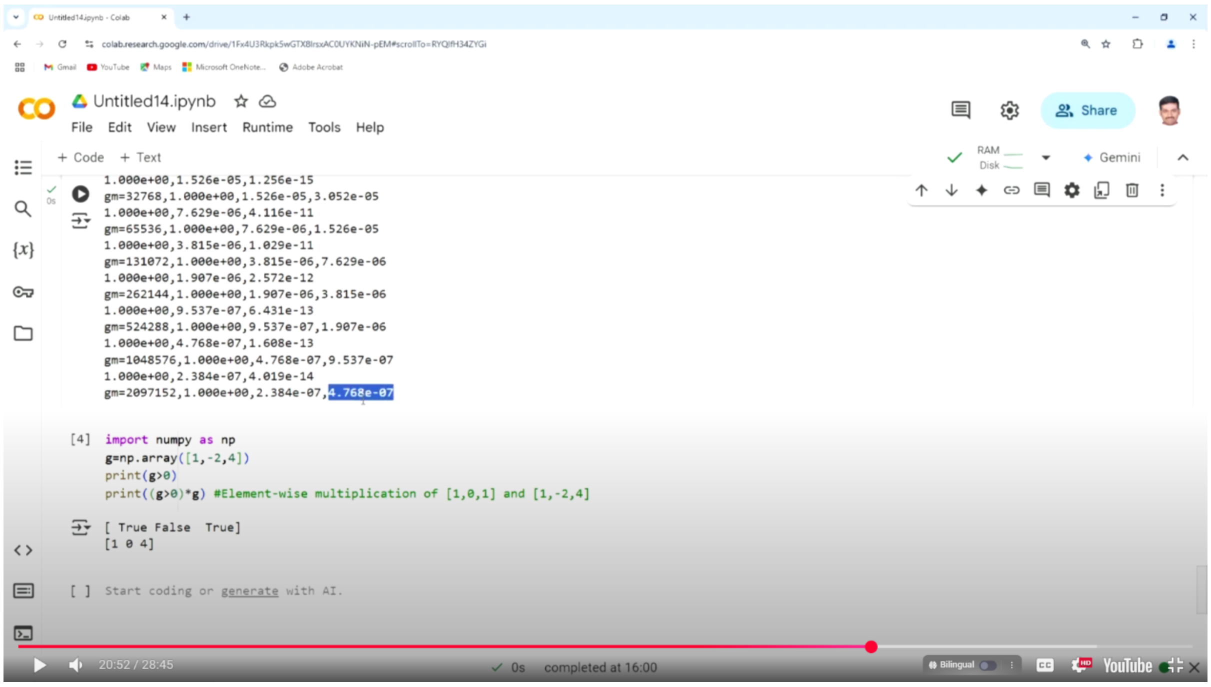
```
import numpy as np
g=np.array([1,-2,4])
print(g>0)
print((g>0)*g) #Element-wise multiplication of [1,0,1] and [1,-2,4]
```

```
[ True False  True]
[1 0 4]
```



```
  return np.array([-x[0]+x[1],x[0]+x[1]-1,-x[1]])

def grad_g(x):
  return np.array([[-1,1],[1,1],[0,-1]])

def Q(x,gm):
  return f(x)+gm*((g(x)>0)*g(x)).dot(g(x))

def grad(x,gm):
  return np.array([x[0]-3,x[1]-2])+2*gm*((g(x)>0)*g(x))@grad_g(x)

def H(x,gm):
  return np.eye(2)+2*gm*((g(x)>0)*grad_g(x).T)@grad_g(x)

x,gm=np.array([3,2]),1
while(np.abs(Q(x,gm)-f(x))>1e-6):
  while(np.linalg.norm(grad(x,gm))>1e-6):
    x=x-np.linalg.inv(H(x,gm))@grad(x,gm) # Newton's method
    print("%1.3e,%1.3e,%1.3e"%(x[0],x[1],np.linalg.norm(grad(x,gm))))
  print("gm=%d,%1.3e,%1.3e,%1.3e"%(gm,x[0],x[1],np.abs(Q(x,gm)-f(x))))
  gm=gm*2
```

```
1.400e+00,4.000e-01,2.589e-15
gm=1,1.400e+00,4.000e-01,6.400e-01
1.222e+00,2.222e-01,2.220e-16
gm=2,1.222e+00,2.222e-01,3.951e-01
1.118e+00,1.176e-01,6.280e-16
```



```
1.000e+00,1.526e-05,1.256e-15
gm=32768,1.000e+00,1.526e-05,3.052e-05
1.000e+00,7.629e-06,4.116e-11
gm=65536,1.000e+00,7.629e-06,1.526e-05
1.000e+00,3.815e-06,1.029e-11
gm=131072,1.000e+00,3.815e-06,7.629e-06
1.000e+00,1.907e-06,2.572e-12
gm=262144,1.000e+00,1.907e-06,3.815e-06
1.000e+00,9.537e-07,6.431e-13
gm=524288,1.000e+00,9.537e-07,1.907e-06
1.000e+00,4.768e-07,1.608e-13
gm=1048576,1.000e+00,4.768e-07,9.537e-07
1.000e+00,2.384e-07,4.019e-14
gm=2097152,1.000e+00,2.384e-07,4.768e-07
```

```
[4] import numpy as np
    g=np.array([1,-2,4])
    print(g>0)
    print((g>0)*g) #Element-wise multiplication of [1,0,1] and [1,-2,4]
```

```
[ True False  True]
[1 0 4]
```

```
[ ] Start coding or generate with AI.
```

Newton's method may fail if the Hessian is not positive definite, particularly when some $g_i(x) = 0$. In such cases, alternative methods like **steepest descent or conjugate gradient** can be employed.

For steepest descent, we define parameters α (step size), c_1 (Armijo constant), and ρ (backtracking factor). The search direction is $d = -\nabla Q(x, \gamma)$. We use backtracking line search to find α satisfying the Armijo condition: $Q(x + \alpha d, \gamma) \leq Q(x, \gamma) + c_1 * \alpha * \nabla Q(x, \gamma)^T d$

Then update $x = x + \alpha d$.

Steepest descent converges slowly when the Hessian has a high condition number, which occurs for large γ. A looser tolerance (e.g., 10^{-2}) can reduce the number of inner iterations.

To implement the steepest descent method, we modify the inner loop of the algorithm. We define the following parameters:

- α: Step size 1.
- c_1: Armijo constant (typically 0.75).
- ρ: Backtracking factor (typically 0.8).

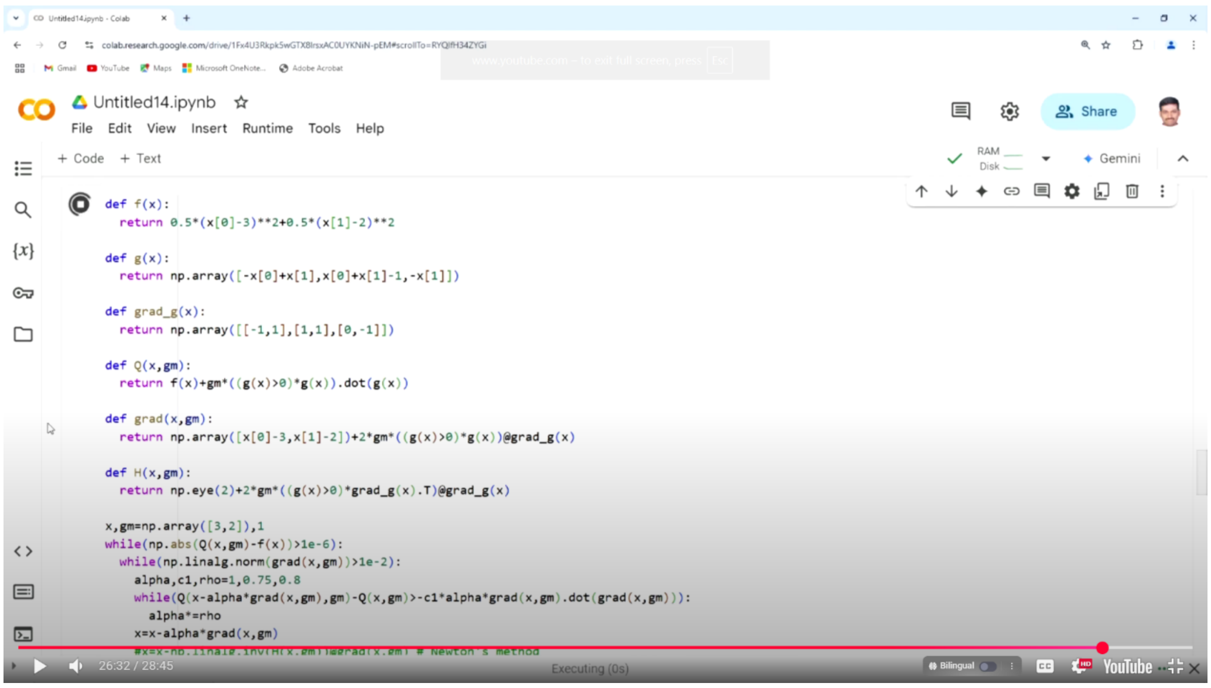
The search direction is the negative gradient of the penalty function: $d = -\nabla Q(x, \gamma)$

We use backtracking line search to find a step size α that satisfies the Armijo condition: $Q(x + \alpha d, \gamma) \leq Q(x, \gamma) + c_1 \cdot \alpha \cdot \nabla Q(x, \gamma)^T d$

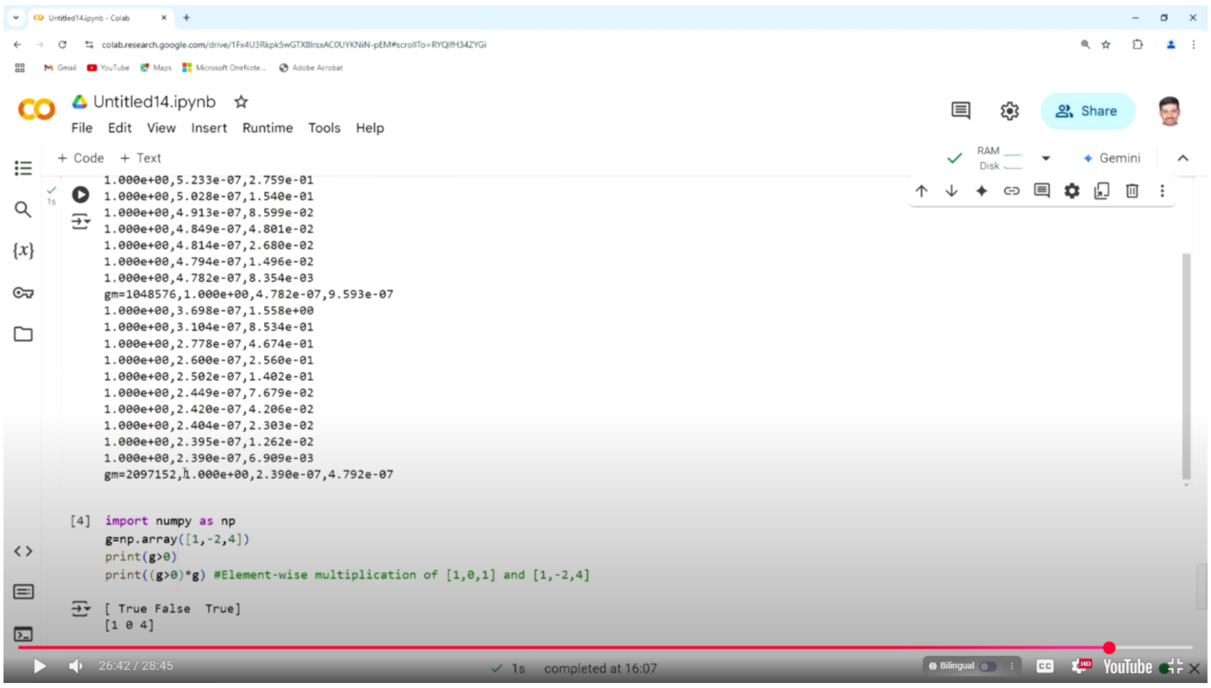
Once α is found, we update x: $x = x - \alpha \cdot \nabla Q(x, \gamma)$

However, the steepest descent method converges slowly when the condition number of the Hessian is high. This occurs for large values of γ, where the condition number increases with γ. To mitigate this, we can use a looser tolerance (e.g., 10^{-2}) for the inner loop stopping criterion.

For example, with $\gamma = 2^{21}$ (approximately 2 million), the steepest descent method requires many iterations for each γ, whereas Newton's method typically converges in one iteration per γ.



```
def f(x):
  return 0.5*(x[0]-3)**2+0.5*(x[1]-2)**2

def g(x):
  return np.array([-x[0]+x[1],x[0]+x[1]-1,-x[1]])

def grad_g(x):
  return np.array([[-1,1],[1,1],[0,-1]])

def Q(x,gm):
  return f(x)+gm*((g(x)>0)*g(x)).dot(g(x))

def grad(x,gm):
  return np.array([x[0]-3,x[1]-2])+2*gm*((g(x)>0)*g(x))@grad_g(x)

def H(x,gm):
  return np.eye(2)+2*gm*((g(x)>0)*grad_g(x).T)@grad_g(x)

x,gm=np.array([3,2]),1
while(np.abs(Q(x,gm)-f(x))>1e-6):
  while(np.linalg.norm(grad(x,gm))>1e-2):
    alpha,c1,rho=1,0.75,0.8
    while(Q(x-alpha*grad(x,gm),gm)-Q(x,gm)>-c1*alpha*grad(x,gm).dot(grad(x,gm))):
      alpha*=rho
    x=x-alpha*grad(x,gm)
    #x=x-np.linalg.inv(H(x,gm))@grad(x,gm) # Newton's method
```



```
1.000e+00,5.233e-07,2.759e-01
1.000e+00,5.028e-07,1.540e-01
1.000e+00,4.913e-07,8.599e-02
1.000e+00,4.849e-07,4.801e-02
1.000e+00,4.814e-07,2.680e-02
1.000e+00,4.794e-07,1.496e-02
1.000e+00,4.782e-07,8.354e-03
gm=1048576,1.000e+00,4.782e-07,9.593e-07
1.000e+00,3.698e-07,1.558e+00
1.000e+00,3.104e-07,8.534e-01
1.000e+00,2.778e-07,4.674e-01
1.000e+00,2.600e-07,2.560e-01
1.000e+00,2.502e-07,1.402e-01
1.000e+00,2.449e-07,7.679e-02
1.000e+00,2.420e-07,4.206e-02
1.000e+00,2.404e-07,2.303e-02
1.000e+00,2.395e-07,1.262e-02
1.000e+00,2.390e-07,6.909e-03
gm=2097152,1.000e+00,2.390e-07,4.792e-07
```

```
[4] import numpy as np
    g=np.array([1,-2,4])
    print(g>0)
    print((g>0)*g) #Element-wise multiplication of [1,0,1] and [1,-2,4]
```

```
[ True False  True]
[1 0 4]
```

We start from the point [3, 2]. So, fortunately, we have obtained the answer, and it is the same value of γ with [1, 0] as the solution. However, it is evident that for each value of γ, a significant number of iterations are required. This is clearly visible from the recorded results.

In contrast, if Newton's method had been used, each value of γ would have required at most one iteration. For instance, even with γ as small as 1e-2 or 1e-6, Newton's method converges in just one step.

This highlights a well-known limitation of gradient descent: when the condition number of the problem is not close to 1, the number of iterations in steepest descent increases substantially.

The farther the condition number is from 1, the slower the convergence. Newton's method does not suffer from this issue, but it has its own challenges, such as computational cost and sensitivity to initial conditions.

The choice between these methods should be based on the specific application and problem characteristics. This concludes the current discussion. We will proceed with the quadratic penalty method in the next lecture. Thank you.