Optimization Algorithms: Theory and Software Implementation

Prof. Thirumulanathan D

Department of Mathematics

Institute of IIT Kanpur

Lecture: 32

Hello everyone, this is the second lecture of week 7. We are learning about constrained optimization algorithms. In the last lecture, I gave a brief recap of what constrained optimization is and its classification into linear and non-linear problems. We began discussing how to frame an algorithm for solving a non-linear optimization problem.

If you recall, we started with a function $\tilde{Q}(x)$ which is defined as:

$\tilde{Q}(x) = f(x)$ if x belongs to the constraint set S

$\tilde{Q}(x) =$ infinity if x does not belong to S

This definition makes the constrained problem equivalent to an unconstrained minimization of $\tilde{Q}(x)$. However, the critical problem is that $\tilde{Q}(x)$ is not smooth or differentiable, which prevents us from using standard gradient-based algorithms.

To overcome this, we now define a new, slightly different function, $Q(x, \gamma)$. Let's first consider a simpler problem to illustrate this approach:

Minimize f(x)

subject to $h(x) = 0$

This is a problem with a single equality constraint and no inequality constraints (m=1, p=0).

For this problem, we define the new function $Q(x, \gamma)$ as:

$Q(x, \gamma) = f(x) + \gamma (h(x))^2$

What is the meaning of this function?

* When x is in the constraint set S (i.e., $h(x) = 0$), then $Q(x, \gamma) = f(x) + \gamma * 0 = f(x)$. So, on the feasible set, Q matches the original objective.

* When x is not in S (i.e., $h(x) \neq 0$), then $Q(x, \gamma) = f(x) + \gamma (h(x))^2$. The term $\gamma (h(x))^2$ acts as a penalty for violating the constraint. The larger the violation $(h(x))^2$, the higher the penalty. The parameter γ controls how severe this penalty is.

This new function $Q(x, \gamma)$ is smooth and differentiable if f and h are smooth, allowing us to use unconstrained optimization methods on it.

The parameter γ in the function $Q(x, \gamma) = f(x) + \gamma (h(x))^2$ is a non-negative quantity called the penalty parameter.

The term $\gamma\ (h(x))^2$ is called the penalty function.

Specifically, because it uses the square of the constraint violation, it is known as a quadratic penalty function.

The logic behind this function is:

* When x is in the constraint set S (i.e., $h(x) = 0$), the penalty term is zero, so $Q(x, \gamma) = f(x)$.

* When x moves away from the constraint set (i.e., $h(x) \neq 0$), the function levies a penalty. The farther x is from satisfying the constraint, the larger the value of $(h(x))^2$ becomes, and thus the larger the penalty ($\gamma\ (h(x))^2$) that is added to f(x).

Geometrically, for a problem like minimizing $x_1 + x_2$ subject to $x_1^2 + x_2^2 = 1$ (a unit circle), the penalty function modifies the landscape. On the circle's boundary, Q equals the original function f. Everywhere else—inside or outside the circle—the value of Q is f(x) + penalty. This penalty increases with the distance from the boundary.

Ideally, we would want the penalty to be infinite outside the feasible set, which is what the original function $\tilde{Q}(x)$ did. Indeed, we can see that:

$\lim_{(\gamma \to \infty)} Q (x, \gamma) = \tilde{Q}(x)$

For any finite γ, minimizing **Q (x, γ) is not exactly equivalent to minimizing f(x)** subject to h(x)=0. However, the key advantage is that for any positive, finite value of γ, the function Q(x, γ) is differentiable (and twice differentiable if f and h are), unlike $\tilde{Q}(x)$.

Therefore, for a sufficiently high value of the penalty parameter γ, the solution to the unconstrained problem of minimizing Q(x, γ) will be very close to the solution of the original constrained problem. This allows us to substitute the original constrained optimization problem with a sequence of unconstrained problems where we minimize this quadratic penalty function, gradually increasing γ to force the solution towards the feasible region. This is the essence of the quadratic penalty method. The function $Q(x, \gamma) = f(x) + \gamma\ (h(x))^2$ is called the quadratic penalty function for the constrained optimization problem. The method that minimizes this function is called the **quadratic penalty method**.

Here is how the method works algorithmically:

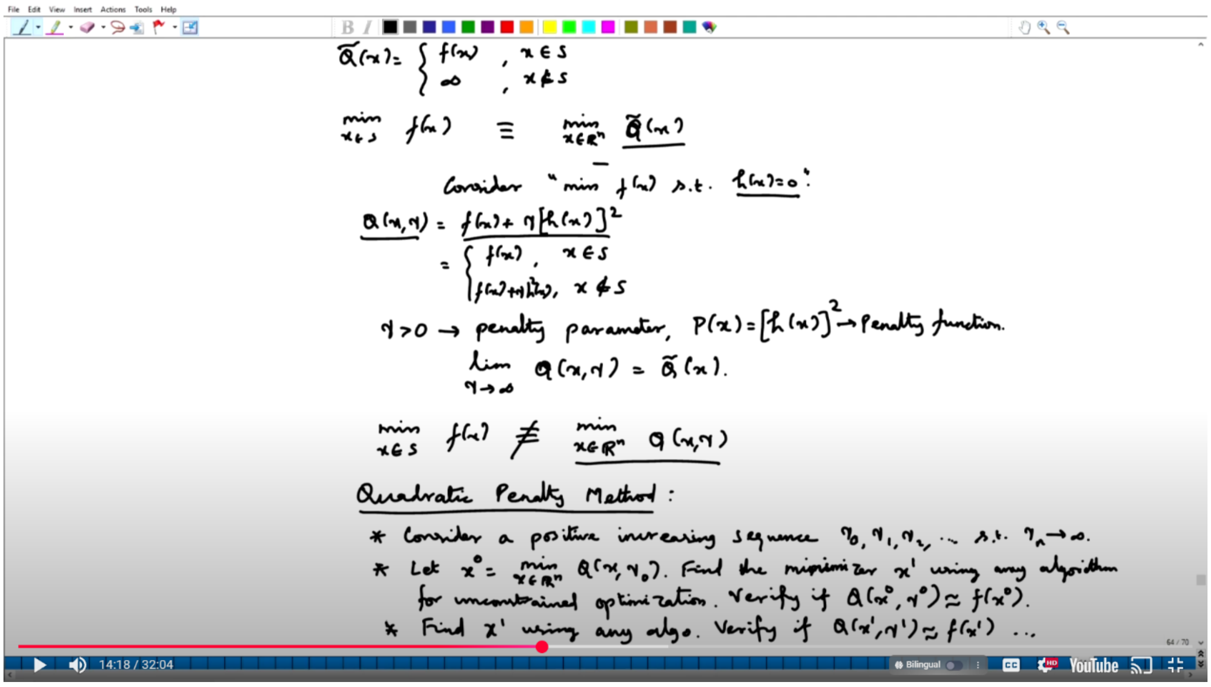
1. **Initialize:** Choose a starting penalty parameter $\gamma_0 > 0$. Select a positive, strictly increasing sequence $\{\gamma_k\}$ such that $\gamma_k \to \infty$ as $k \to \infty$. Choose a small tolerance value ($\varepsilon > 0$) to determine convergence. Set k = 0.

2. **Iterative Minimization:** Begin a loop. For the current penalty parameter γ_k, solve the unconstrained optimization problem:

   $x_k = \arg\min_{(x \in R^n)} Q(x, \gamma_k) = \arg\min_{(x \in R^n)} [\ f(x) + \gamma_k\ (h(x))^2\ ]$

   Use any standard unconstrained optimization algorithm (e.g., gradient descent, Newton's method) to find this minimizer, x_k.

(Refer Slide Time 14:20)



$$\tilde{Q}(x) = \begin{cases} f(x) & , x \in S \\ \infty & , x \notin S \end{cases}$$

$$\min_{x \in S} f(x) \equiv \min_{x \in \mathbb{R}^n} \underline{\tilde{Q}(x)}$$

Consider "$\min f(x)$ s.t. $\underline{h(x)=0}$".

$$\underline{Q(x,\gamma)} = \underline{f(x) + \gamma[h(x)]^2}$$
$$= \begin{cases} f(x), & x \in S \\ f(x)+\gamma h^2(x), & x \notin S \end{cases}$$

$\gamma > 0 \rightarrow$ penalty parameter, $P(x) = [h(x)]^2 \rightarrow$ Penalty function.

$$\lim_{\gamma \to \infty} Q(x,\gamma) = \tilde{Q}(x).$$

$$\min_{x \in S} f(x) \not\equiv \underline{\min_{x \in \mathbb{R}^n} Q(x,\gamma)}$$

Quadratic Penalty Method :

* Consider a positive increasing sequence $\gamma_0, \gamma_1, \gamma_2, \ldots$ s.t. $\gamma_n \to \infty$.
* Let $x^0 = \min_{x \in \mathbb{R}^n} Q(x, \gamma_0)$. Find the minimizer x' using any algorithm for unconstrained optimization. Verify if $Q(x^0, \gamma^0) \simeq f(x^0)$.
* Find x' using any algo. Verify if $Q(x', \gamma') \simeq f(x')$ ...

3. **Check Convergence:** Evaluate the constraint violation at the current solution, x_k. This is done by checking the value of the penalty term. Specifically, check if:

$$| Q(x_k, \gamma_k) - f(x_k) | < \varepsilon$$

Since $Q(x_k, \gamma_k) - f(x_k) = \gamma_k (h(x_k))^2$, this condition is equivalent to checking if the squared constraint violation $(h(x_k))^2$ is very close to zero. If it is, then x_k is very near the feasible set ($h(x)=0$), and the algorithm stops.

4. **Update and Repeat: If the convergence condition is not met:**

   * Increase the penalty parameter: set $k = k + 1$ and proceed to the next, larger value in the sequence, γ_k.

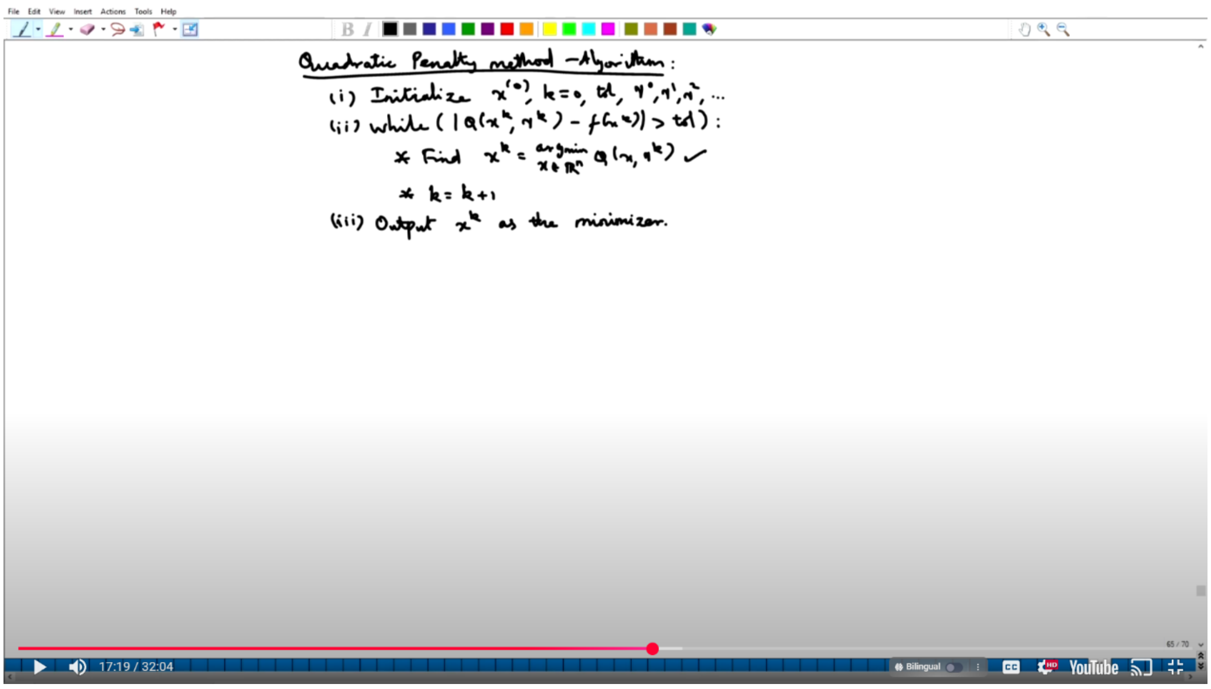
   * Return to Step 2 and solve the new unconstrained problem with this stronger penalty.

5. **Termination:** Once the convergence condition is satisfied, output the current solution x_k as the approximate minimizer of the original constrained problem.

In summary, the algorithm creates a sequence of unconstrained problems with increasingly harsh penalties for constraint violation. The solutions to these problems form a sequence of points $\{x_k\}$ that, under the right conditions, converge to the solution of the original constrained optimization problem as the penalty parameter γ_k tends to infinity.

Of course. Here is the explanation and algorithm for the Quadratic Penalty Method, written in the requested format with proper subscripts and superscripts, followed by a structured Python code implementation.

(Refer Slide Time 17:30)



<u>Quadratic Penalty method – Algorithm</u>:

```
(i) Initialize $x^{(0)}$, $k=0$, tol, $\gamma^0, \gamma^1, \gamma^2, \dots$
(ii) while ( $|Q(x^k, \gamma^k) - f(x^k)| >$ tol ):
        * Find $x^k = \underset{x \in \mathbb{R}^n}{\arg\min}\, Q(x, \gamma^k)$ ✓
        * $k = k+1$
(iii) Output $x^k$ as the minimizer.
```

Quadratic Penalty Method for a Constrained Optimization Problem

Problem Statement:

Minimize the objective function:

$f(\mathbf{x}) = x_1 + x_2$

subject to the equality constraint:

$c(\mathbf{x}) = x_1^2 + x_2^2 - 1 = 0$

So, let us start with [-1, -1], k = 0. We will put tolerance = 1e-6. And So, $\gamma_0 = 1$, $\gamma_1 = 2$, $\gamma_2 = 4$ and $\gamma_k = 2\,\gamma^{k-1}$.

Quadratic Penalty Function:

The method converts the constrained problem into a sequence of unconstrained minimization problems. For a given penalty parameter $\gamma_k > 0$, we minimize the function $Q(\mathbf{x}, \gamma_k)$:

$Q(\mathbf{x}, \gamma_k) = f(\mathbf{x}) + \gamma_k * (c(\mathbf{x}))^2$

Substituting our specific functions:

$Q(\mathbf{x}, \gamma_k) = (x_0 + x_1) + \gamma_k * (x_0^2 + x_1^2 - 1)^2$

(Note: In code, x[0] corresponds to x_0 or x_1, and x[1] corresponds to x_1 or x_2)

Gradient of the Penalty Function (∇Q):

$\nabla f(\mathbf{x}) = [1, 1]^T$

$\nabla[\gamma_k (c(\mathbf{x}))^2] = 2 * \gamma_k * c(\mathbf{x}) * \nabla c(\mathbf{x}) = 2 * \gamma_k * (x_0^2 + x_1^2 - 1) * [2x_0, 2x_1]^T = 4 * \gamma_k * (x_0^2 + x_1^2 - 1) * [x_0, x_1]^T$

Therefore, the full gradient is:

$\nabla Q(\mathbf{x}, \gamma_k) = [1 + 4\gamma_k x_0(x_0^2 + x_1^2 - 1),\ 1 + 4\gamma_k x_1(x_0^2 + x_1^2 - 1)]^T$

Hessian of the Penalty Function ($\nabla^2 Q$):

The Hessian of f(**x**) is a zero matrix. The Hessian of the penalty term is derived by differentiating ∇Q again.

* Element (0,0): $\partial^2 Q/\partial x_0^2 = 4\gamma_k(x_0^2 + x_1^2 - 1) + 8\gamma_k x_0^2$
* Element (0,1) and (1,0): $\partial^2 Q/\partial x_0 \partial x_1 = 8\gamma_k x_0 x_1$
* Element (1,1): $\partial^2 Q/\partial x_1^2 = 4\gamma_k(x_0^2 + x_1^2 - 1) + 8\gamma_k x_1^2$

Thus, the Hessian matrix is:

$$\nabla^2 Q(\mathbf{x}, \gamma_k) = \gamma_k * [\ [4(x_0^2+x_1^2-1) + 8x_0^2,\ 8x_0x_1\ ],$$
$$[8x_0x_1,\quad 4(x_0^2+x_1^2-1) + 8x_1^2]\ ]$$

Algorithm Pseudocode:

```
1. Initialize: Choose initial point $\mathbf{x}_0$, initial penalty parameter $\gamma_0 > 0$ (e.g., $\gamma_0 = 1$), tolerance $\epsilon > 0$ (e.g., $\epsilon = 10^{-6}$), and a scaling factor $\beta > 1$ (e.g., $\beta = 2$). Set $k = 0$.
2. Inner Loop (Unconstrained Minimization):
    * Using an algorithm like Newton's Method, find an approximate minimizer $\mathbf{x}_k^*$ of the unconstrained problem min $Q(\mathbf{x}, \gamma_k)$, using $\mathbf{x}_k$ as the starting point.
    * $\mathbf{x}_{k+1} = \text{argmin}( Q(\mathbf{x}, \gamma_k) )$ (approximately)
3. Stopping Check: If the constraint violation is small, e.g., $|c(\mathbf{x}_{k+1})| < \epsilon$, stop. $\mathbf{x}_{k+1}$ is the approximate solution.
4. Update: If not stopped, increase the penalty parameter: $\gamma_{k+1} = \beta * \gamma_k$. Set $k = k + 1$ and go back to Step 2.
```

Python Implementation Using Newton's Method

Python

```
import numpy as np

# Define the original objective function f(x)
def f(x):
    return x[0] + x[1]

# Define the equality constraint c(x) = 0
def constraint(x):
    return x[0]**2 + x[1]**2 - 1
# Define the Quadratic Penalty Function Q(x, gamma)
def Q(x, gm):
    return f(x) + gm * (constraint(x))**2
```
# Define the Gradient of Q(x, gamma)
```
def grad_Q(x, gm):
    # Gradient of f(x) is [1, 1]
    grad_f = np.array([1, 1])
    # Precompute the common term: (x0^2 + x1^2 - 1)
    c_val = constraint(x)
    # Gradient of the penalty term: 4 * gm * c_val * [x0, x1]
    grad_penalty = 4 * gm * c_val * x
    return grad_f + grad_penalty
```
# Define the Hessian of Q(x, gamma)
```
def hess_Q(x, gm):
    # Precompute common terms
    x0_sq = x[0]**2
    x1_sq = x[1]**2
    c_val = x0_sq + x1_sq - 1  # constraint(x)
    # Calculate each element of the 2x2 Hessian matrix
    h00 = 4 * gm * c_val + 8 * gm * x0_sq
    h01 = 8 * gm * x[0] * x[1]
    h11 = 4 * gm * c_val + 8 * gm * x1_sq
    # Construct the symmetric Hessian matrix
    H = np.array([[h00, h01],
                  [h01, h11]])
    return H
```

# Newton's Method for unconstrained minimization of Q

```
def newtons_method_for_Q(x0, gm, tol=1e-6, max_iter=100):
    x = x0.copy()
    for i in range(max_iter):
        g = grad_Q(x, gm)
        H = hess_Q(x, gm)
        # Newton step: solve H * d = -g
        try:
            d = np.linalg.solve(H, -g)
        except np.linalg.LinAlgError:
            print("Hessian is singular. Stopping.")
            break
        x_new = x + d
        # Check for convergence (step length is small)
        if np.linalg.norm(d) < tol:
            break
        x = x_new
    return x
# Main Quadratic Penalty Method Algorithm
# Initialize parameters
x = np.array([-1.0, -1.0]) # Initial guess
gamma_k = 1                  # Initial penalty parameter γ₀
beta = 2                     # Scaling factor for γ
tol_outer = 1e-6             # Outer loop tolerance
max_outer_iters = 20         # Prevent infinite loop
print(f"Starting Quadratic Penalty Method with x0 = {x}, γ0 = {gamma_k}")
print("k | gamma_k | x_k[0] | x_k[1] | |constraint(x_k)|")
print("-" * 55)
for k in range(max_outer_iters):
    # 1. Solve the unconstrained subproblem for current gamma_k
    x_sol = newtons_method_for_Q(x, gamma_k)
    # 2. Calculate the constraint violation
    constraint_violation = np.abs(constraint(x_sol))
    # Print current iteration data

print(f"{k:2d}|{gamma_k:6.1f}|{x_sol[0]:+6.4f}|{x_sol[1]:+6.4f}|{constraint_violation:12.6e}")
    # 3. Check stopping criterion
    if constraint_violation < tol_outer:
        print("\nConvergence achieved! Constraint violation is below tolerance.")
        break
    # 4. Update for next iteration: increase gamma and warm-start from current solution
    gamma_k = beta * gamma_k
    x = x_sol # Use the current solution as the starting point for the next subproblem
else:
    print("\nStopped after maximum iterations.")
# Output the final result
print(f"\n*** Final Solution ***")
print(f"x* ≈ [{x_sol[0]:.6f}, {x_sol[1]:.6f}]")
print(f"f(x*) = {f(x_sol):.6f}")
print(f"Constraint c(x*)={constraint(x_sol):.6e}(|c(x*)|<{tol_outer}is{np.abs(constraint(x_so)) < tol_outer})")
# The true solution for this problem is x* = [-√2/2, -√2/2] ≈ [-0.707107, -0.707107], f(x*) = -√2 ≈ -1.414213
```

We are minimizing the function $f(\mathbf{x}) = x_1 + x_2$ subject to the constraint $c(\mathbf{x}) = x_1^2 + x_2^2 - 1 = 0$ using the quadratic penalty method.

The quadratic penalty function is defined as:

$$Q(\mathbf{x}, \gamma) = f(\mathbf{x}) + \gamma\cdot(c(\mathbf{x}))^2 = (x_0 + x_1) + \gamma\cdot(x_0^2 + x_1^2 - 1)^2$$

The gradient of Q is:

$$\nabla Q(\mathbf{x}, \gamma) = [1 + 4\gamma x_0(x_0^2 + x_1^2 - 1),\ 1 + 4\gamma x_1(x_0^2 + x_1^2 - 1)]^T$$

The Hessian of Q is:

$\nabla^2 Q(\mathbf{x}, \gamma) = [[4\gamma(x_0^2+x_1^2-1) + 8\gamma x_0^2, 8\gamma x_0 x_1],$

$[8\gamma x_0 x_1, 4\gamma(x_0^2+x_1^2-1) + 8\gamma x_1^2]]$

Algorithm:

```
1. Initialize x = [-1, -1], $\gamma = 1$, tolerance = 1e-6
2. While the penalty term $\gamma\cdot(c(\mathbf{x}))^2$ > tolerance:
  a. Use Newton's method to minimize $Q(\mathbf{x}, \gamma)$:
    While $\|\nabla Q(\mathbf{x}, \gamma)\| > 1e-6$:
      x = x - $[\nabla^2 Q(\mathbf{x}, \gamma)]^{-1}\cdot\nabla Q(\mathbf{x}, \gamma)$
  b. Print $\gamma$, x[0], x[1], and the penalty term
  c. Update $\gamma = 2\cdot\gamma$
```

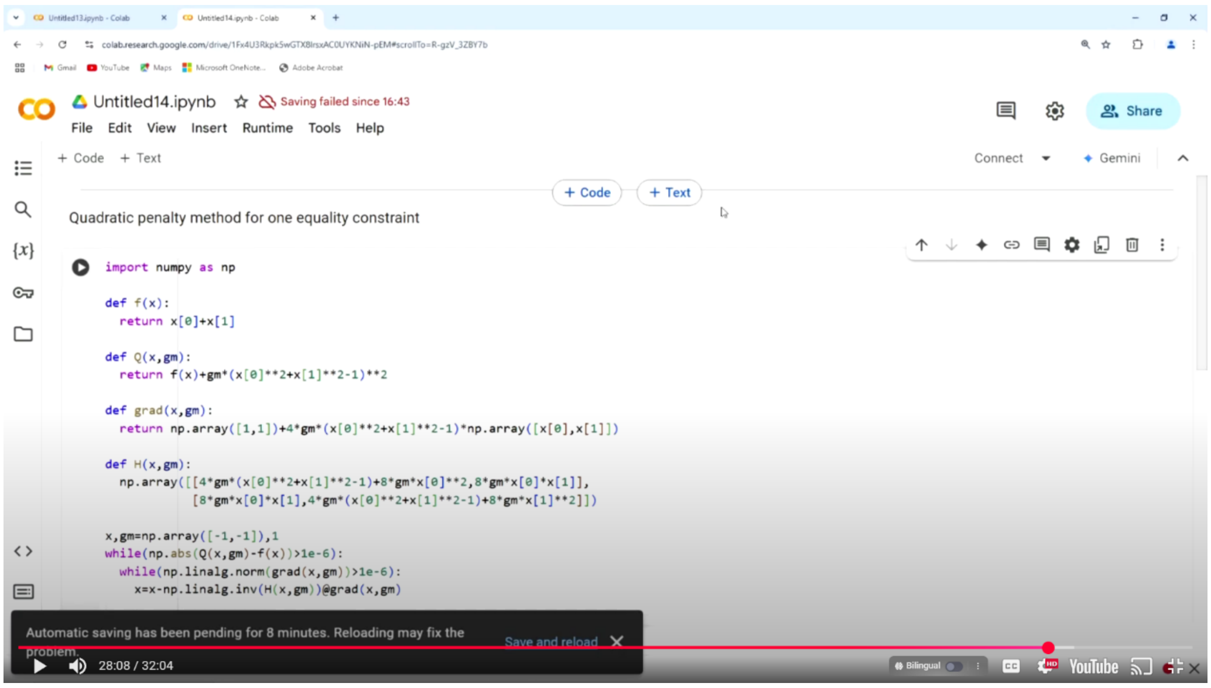
Python implementation:

Python

```python
import numpy as np
def f(x):
    return x[0] + x[1]
def constraint(x):
    return x[0]**2 + x[1]**2 - 1


def Q(x, gamma):
    return f(x) + gamma * (constraint(x))**2
def grad_Q(x, gamma):
    c_val = constraint(x)
    grad_penalty = 4 * gamma * c_val * x
    return np.array([1, 1]) + grad_penalty
def hess_Q(x, gamma):
    c_val = constraint(x)
    h00 = 4*gamma*c_val + 8*gamma*x[0]**2
    h11 = 4*gamma*c_val + 8*gamma*x[1]**2
    h01 = 8*gamma*x[0]*x[1]
    return np.array([[h00, h01], [h01, h11]])
# Initialize parameters
x = np.array([-1.0, -1.0])
gamma = 1
tolerance = 1e-6
max_iters = 20
print("γk| x[0] | x[1] | γk·(c(x))²")
print("------------------------------------------")
for k in range(max_iters):
    # Newton's method for inner optimization
    for inner_iter in range(100):
        g = grad_Q(x, gamma)
        H = hess_Q(x, gamma)
        dx = np.linalg.solve(H, -g)
        x = x + dx
        if np.linalg.norm(g) < 1e-6:
            break
    penalty_term = gamma * (constraint(x))**2
    print(f"{gamma:<6} | {x[0]:+8.6f} | {x[1]:+8.6f} | {penalty_term:.6e}")
    if penalty_term < tolerance:
        break
    gamma *= 2
print(f"\nFinal solution: x = [{x[0]:.6f}, {x[1]:.6f}]")
print(f"Theoretical optimum: x = [-1/√2, -1/√2] ≈ [-0.707107, -0.707107]")
```

The algorithm starts with $\gamma_0 = 1$ and doubles the penalty parameter each iteration ($\gamma_k = 2^k$). After 19 iterations, γ reaches $2^{18} = 262144$, and the solution converges to $x \approx [-1/\sqrt{2}, -1/\sqrt{2}]$ with the constraint violation below the tolerance of 1e-6.

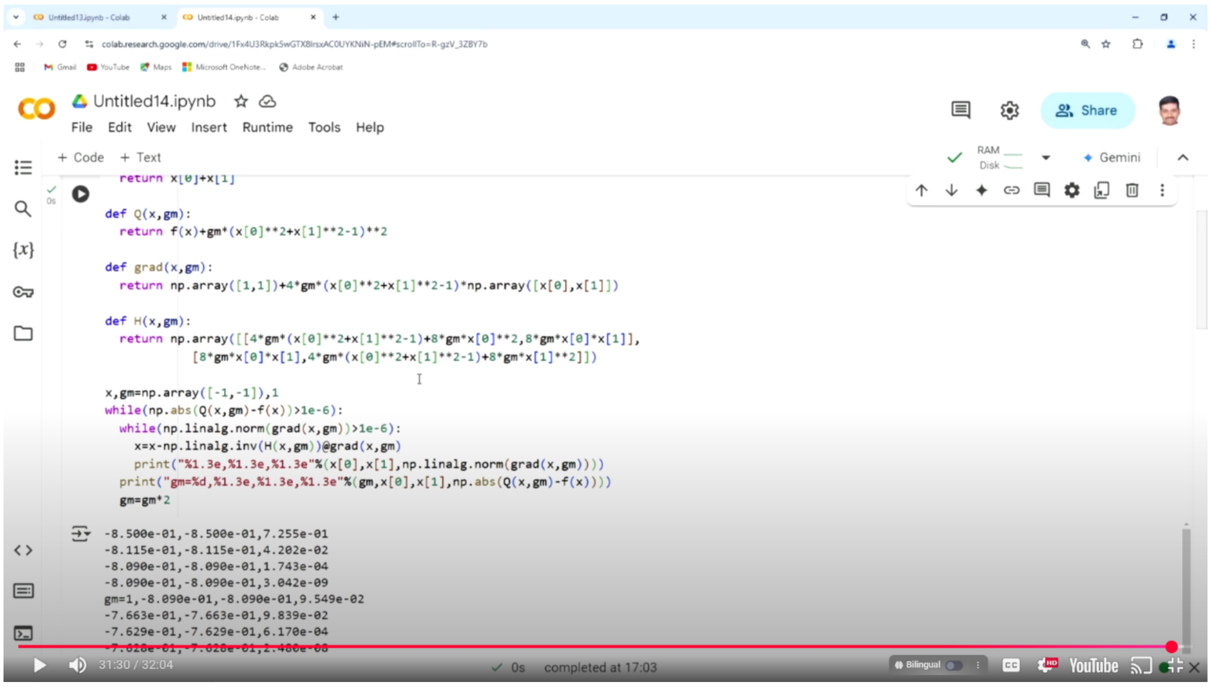
(Refer Slide Time 31:30)


Untitled14.ipynb
Saving failed since 16:43
File Edit View Insert Runtime Tools Help
+ Code + Text
Share
Connect
Gemini
28:08 / 32:04

Quadratic penalty method for one equality constraint

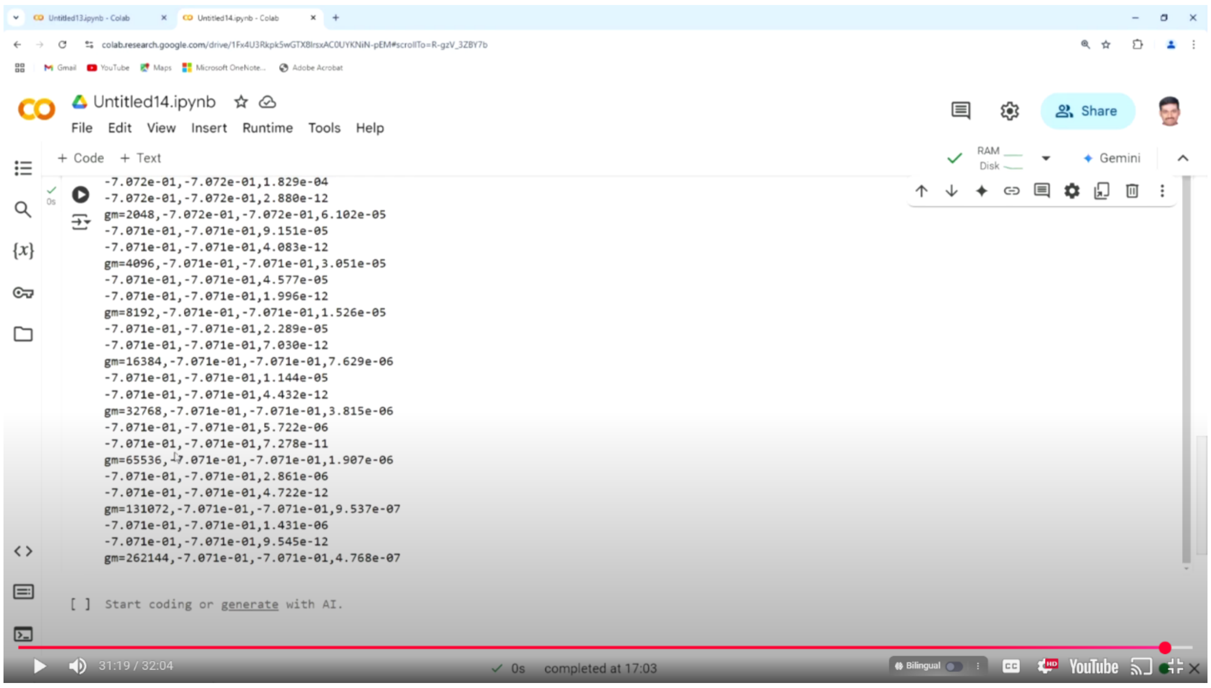
```
import numpy as np

def f(x):
  return x[0]+x[1]

def Q(x,gm):
  return f(x)+gm*(x[0]**2+x[1]**2-1)**2

def grad(x,gm):
  return np.array([1,1])+4*gm*(x[0]**2+x[1]**2-1)*np.array([x[0],x[1]])

def H(x,gm):
  np.array([[4*gm*(x[0]**2+x[1]**2-1)+8*gm*x[0]**2,8*gm*x[0]*x[1]],
            [8*gm*x[0]*x[1],4*gm*(x[0]**2+x[1]**2-1)+8*gm*x[1]**2]])

x,gm=np.array([-1,-1]),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)
```

Automatic saving has been pending for 8 minutes. Reloading may fix the problem. Save and reload



```
  return x[0]+x[1]

def Q(x,gm):
  return f(x)+gm*(x[0]**2+x[1]**2-1)**2

def grad(x,gm):
  return np.array([1,1])+4*gm*(x[0]**2+x[1]**2-1)*np.array([x[0],x[1]])

def H(x,gm):
  return np.array([[4*gm*(x[0]**2+x[1]**2-1)+8*gm*x[0]**2,8*gm*x[0]*x[1]],
            [8*gm*x[0]*x[1],4*gm*(x[0]**2+x[1]**2-1)+8*gm*x[1]**2]])

x,gm=np.array([-1,-1]),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)
    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
```

```
-8.500e-01,-8.500e-01,7.255e-01
-8.115e-01,-8.115e-01,4.202e-02
-8.090e-01,-8.090e-01,1.743e-04
-8.090e-01,-8.090e-01,3.042e-09
gm=1,-8.090e-01,-8.090e-01,9.549e-02
-7.663e-01,-7.663e-01,9.839e-02
-7.629e-01,-7.629e-01,6.170e-04
-7.628e-01,-7.628e-01,2.480e-08
```



```
-7.072e-01,-7.072e-01,1.829e-04
-7.072e-01,-7.072e-01,2.880e-12
gm=2048,-7.072e-01,-7.072e-01,6.102e-05
-7.071e-01,-7.071e-01,9.151e-05
-7.071e-01,-7.071e-01,4.083e-12
gm=4096,-7.071e-01,-7.071e-01,3.051e-05
-7.071e-01,-7.071e-01,4.577e-05
-7.071e-01,-7.071e-01,1.996e-12
gm=8192,-7.071e-01,-7.071e-01,1.526e-05
-7.071e-01,-7.071e-01,2.289e-05
-7.071e-01,-7.071e-01,7.030e-12
gm=16384,-7.071e-01,-7.071e-01,7.629e-06
-7.071e-01,-7.071e-01,1.144e-05
-7.071e-01,-7.071e-01,4.432e-12
gm=32768,-7.071e-01,-7.071e-01,3.815e-06
-7.071e-01,-7.071e-01,5.722e-06
-7.071e-01,-7.071e-01,7.278e-11
gm=65536,-7.071e-01,-7.071e-01,1.907e-06
-7.071e-01,-7.071e-01,2.861e-06
-7.071e-01,-7.071e-01,4.722e-12
gm=131072,-7.071e-01,-7.071e-01,9.537e-07
-7.071e-01,-7.071e-01,1.431e-06
-7.071e-01,-7.071e-01,9.545e-12
gm=262144,-7.071e-01,-7.071e-01,4.768e-07

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

For problems with multiple equality or inequality constraints, the penalty function would be extended to include all constraints, with appropriate modifications to the gradient and Hessian calculations. This will be covered in the next lecture. Thank You