Optimization Algorithms: Theory and Software Implementation

Prof. Thirumulanathan D

Department of Mathematics

Institute of IIT Kanpur

Lecture: 15

Hello everyone. This is the fifth lecture in week three. In the previous lecture, we were learning about the steepest descent algorithm, also known as the gradient descent algorithm. This method involves choosing the descent direction, d_k, to be equal to the negative of the gradient of the function f at the point x_k. We concluded the last lecture by presenting the algorithm.

We will now begin with some examples.

Let us start with the familiar function: $f(x_1, x_2) = x_1^2 + x_2^2$, and let the initial point x_0 be any arbitrary value $(x_{1,0}, x_{2,0})$.

This is a quadratic function, which can be written in the form

$(1/2)x^T Hx + b^T x + c$,

where H is the 2x2 identity matrix multiplied by 2, b is the zero vector, and c is zero.

Note $\mathbf{x_{n,k}=x_n^k}$

Given this quadratic form, we can find the step size α_k using the exact line search formula. We start by computing the descent direction d_0.

The gradient of f is $[2x_1, 2x_2]$, so at x_0 it is $[2x_{1,0}, 2x_{2,0}]$.

Therefore, d_0 is the negative gradient: $[-2x_{1,0}, -2x_{2,0}]$.

The formula for the step size is

$\alpha_0 = -(d_0^T g_0) / (d_0^T H d_0)$.

Since d_0 is $-g_0$, the numerator becomes $(d_0^T g_0) = (-g_0)^T g_0 = -g_0^T g_0$.

Therefore, the negative sign in the formula gives $- (-g_0^T g_0) = g_0^T g_0$ for the numerator.

The denominator is $d_0^T H d_0$.

Because H is 2I, and d_0 is $-g_0$, this becomes $(-g_0)^T (2I) (-g_0) = g_0^T (2I) g_0 = 2 (g_0^T g_0)$.

Thus, $\alpha_0 = (g_0^T g_0) / (2 * g_0^T g_0) = 1/2$.

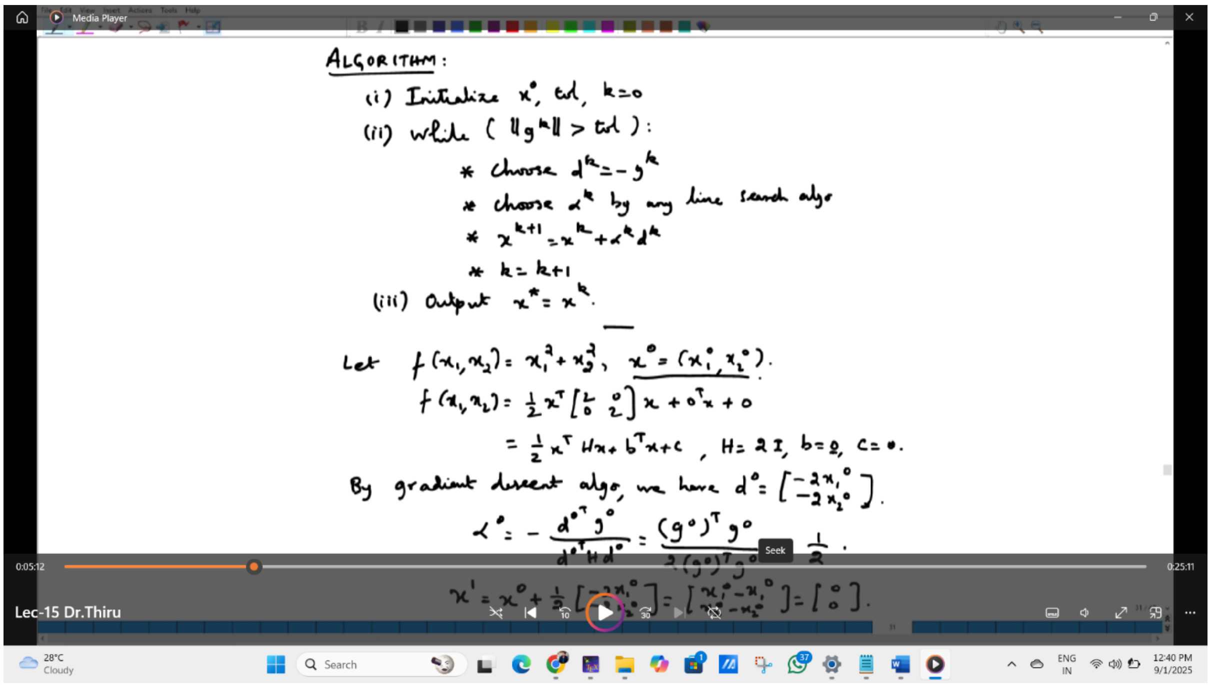
This result holds for any initial point x_0.

The next iterate, x_1, is computed as

$x_0 + \alpha_0 * d_0 = [x_{1,0}, x_{2,0}] + (1/2)*[-2x_{10}, -2x_{20}] = [x_{1,0} - x_{1,0}, x_{2,0} - x_{2,0}] = [0, 0]$.

Consequently, the algorithm converges to the minimum [0, 0] in a single step for this function.

(Refer Slide Time 5:10)



ALGORITHM:

```
(i) Initialize $x^0$, tol, $k=0$
(ii) while ( $\|g^k\| > tol$ ):
        * choose $d^k = -g^k$
        * choose $\alpha^k$ by any line search algo
        * $x^{k+1} = x^k + \alpha^k d^k$
        * $k = k+1$
(iii) Output $x^* = x^k$.
```

Let $f(x_1,x_2) = x_1^2 + x_2^2$, $\underline{x^0 = (x_1^0, x_2^0)}$.

$$f(x_1,x_2) = \frac{1}{2} x^T \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} x + 0^T x + 0$$

$$= \frac{1}{2} x^T H x + b^T x + c, \quad H = 2I, \; b = 0, \; c = 0.$$

By gradient descent algo, we have $d^0 = \begin{bmatrix} -2x_1^0 \\ -2x_2^0 \end{bmatrix}$.

$$\alpha^0 = -\frac{d^{0T} g^0}{d^{0T} H d^0} = \frac{(g^0)^T g^0}{2 (g^0)^T g^0} = \frac{1}{2}.$$

$$x^1 = x^0 + \frac{1}{2}\begin{bmatrix} -2x_1^0 \\ -2x_2^0 \end{bmatrix} = \begin{bmatrix} x_1^0 - x_1^0 \\ x_2^0 - x_2^0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

This example might suggest that the algorithm always converges quickly. However, this is not true even for all quadratic functions.

Consider another quadratic function: $f(x_1, x_2) = 4x_1^2 + 2x_1x_2 + x_2^2$.

This can be written as $(1/2)x^THx$ with H = [[8, 2], [2, 2]], b = [0, 0], and c = 0.

We will implement the gradient descent algorithm in Python for this function, starting from the initial point $x_0 = [4, -5]$.

First, we define the function and its gradient.

The gradient is computed as $\nabla f = [\partial f/\partial x_1, \partial f/\partial x_2] = [8*x_1 + 2*x_2, 2*x_1 + 2*x_2]$.

We will also define the Hessian matrix H = [[8, 2], [2, 2]].

The algorithm uses a while loop that continues until the norm of the gradient is very small (e.g., less than 1e-6).

In each iteration, the step size α_k for this quadratic function is calculated using the formula:

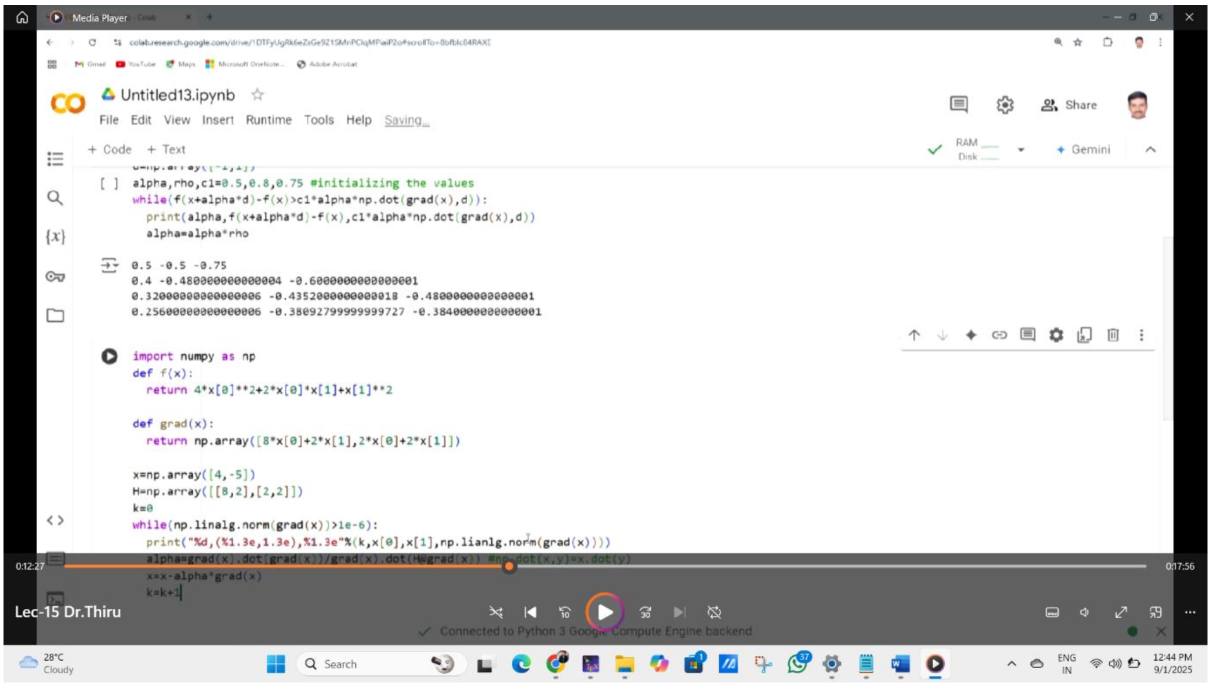
$\alpha_k = (g_k^T g_k) / (g_k^T H g_k)$, where g_k is the gradient at x_k.

The Python code for this is as follows:

Python

```
import numpy as np

# Define the function and its gradient

def f(x):

    return 4*x[0]**2 + 2*x[0]*x[1] + x[1]**2

def grad(x):

    return np.array([8*x[0] + 2*x[1], 2*x[0] + 2*x[1]])

# Define the Hessian

H = np.array([[8, 2], [2, 2]])

# Initial point

x = np.array([4.0, -5.0])

k = 0

# For storing the trajectory for plotting

x_trajectory = [x.copy()]

# Gradient descent algorithm

while np.linalg.norm(grad(x)) > 1e-6:

    g = grad(x)

    # Compute step size alpha

    alpha = np.dot(g, g) / np.dot(g, np.dot(H, g))

    # Update x

    x = x - alpha * g

    k=k +1
```

(Refer Slide Time 12:27)


Media Player
Untitled13.ipynb
File Edit View Insert Runtime Tools Help Saving...
+ Code + Text
Share
Gemini
0:12:27
0:17:56
Lec-15 Dr.Thiru
Connected to Python 3 Google Compute Engine backend

```
alpha,rho,c1=0.5,0.8,0.75 #initializing the values
while(f(x+alpha*d)-f(x)>c1*alpha*np.dot(grad(x),d)):
  print(alpha,f(x+alpha*d)-f(x),c1*alpha*np.dot(grad(x),d))
  alpha=alpha*rho
```

```
0.5 -0.5 -0.75
0.4 -0.48000000000000004 -0.6000000000000001
0.32000000000000006 -0.4352000000000018 -0.4800000000000001
0.25600000000000006 -0.38092799999999727 -0.3840000000000001
```

```
import numpy as np
def f(x):
  return 4*x[0]**2+2*x[0]*x[1]+x[1]**2

def grad(x):
  return np.array([8*x[0]+2*x[1],2*x[0]+2*x[1]])

x=np.array([4,-5])
H=np.array([[8,2],[2,2]])
k=0
while(np.linalg.norm(grad(x))>1e-6):
  print("%d,(%1.3e,1.3e),%1.3e"%(k,x[0],x[1],np.lianlg.norm(grad(x))))
  alpha=grad(x).dot(grad(x))/grad(x).dot(H@grad(x)) #np.dot(x,y)=x.dot(y)
  x=x-alpha*grad(x)
  k=k+1
```

x_trajectory.append(x.copy())

# Print iteration details

print(f"Iteration {k}: x = [{x[0]:.6f}, {x[1]:.6f}], Grad Norm = {np.linalg.norm(g):.6f}")

print(f"\nConverged to minimum at [0, 0] in {k} iterations.")

Running this algorithm shows that it takes approximately 30 iterations to converge to [0, 0], unlike the previous example which converged in one step. To visualize the convergence path, we plot the sequence of points.

To create a plot in Python, we use the matplotlib.pyplot library.

For instance, to plot e^x over the range [-5, 5], we define an array x using `x = np.arange(-5, 5, 0.1)`, compute y = np.exp(x), and then use `plt.plot(x, y)` followed by `plt.show()`.

Similarly, we can plot the trajectory of the gradient descent algorithm for the second function. We extract the x_1 and x_2 coordinates from the stored `x_trajectory` list and plot them.

Python

import matplotlib.pyplot as plt

# Convert the trajectory list into arrays for plotting

x1_vals = [point[0] for point in x_trajectory]

x2_vals = [point[1] for point in x_trajectory]

plt.plot(x1_vals, x2_vals, 'o-')

plt.xlabel('x_1')

plt.ylabel('x₂')

plt.title('Gradient Descent Trajectory for f(x₁, x₂) = 4x₁² + 2x₁x₂ + x₂²')

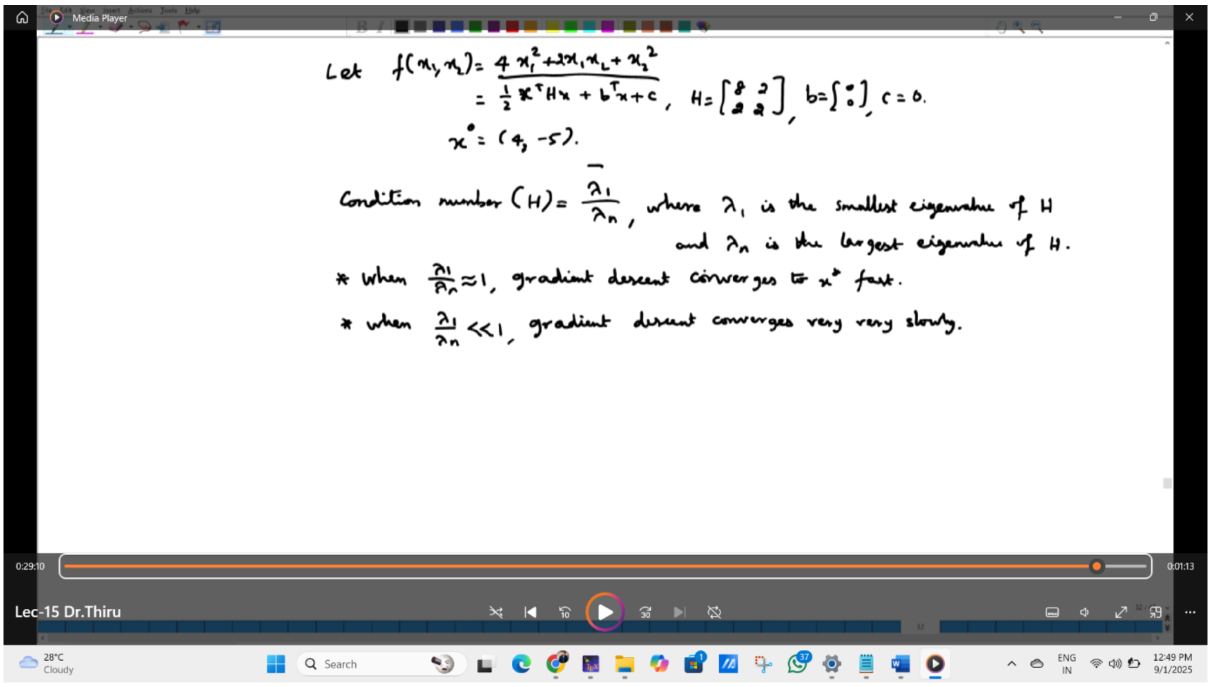
plt.grid(True)

plt.show()

The resulting path shows a zigzag pattern instead of a straight line from the start to the optimum. This indicates inefficient convergence behaviour.

The performance difference is explained by the condition number of the Hessian matrix H. The condition number is defined as the ratio of the smallest eigenvalue (λ_1) to the largest eigenvalue (λ_n) of H. When this ratio is close to 1, gradient descent converges quickly. When it is much smaller than 1, convergence is slow.

(Refer Slide Time 29:10)



Let $f(x_1, x_2) = \underline{4x_1^2 + 2x_1x_2 + x_2^2}$

$= \frac{1}{2}x^T H x + b^T x + c$, $H = \begin{bmatrix} 8 & 2 \\ 2 & 2 \end{bmatrix}$, $b = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, $c = 0$.

$x^* = (4, -5)$.

Condition number $(H) = \frac{\lambda_1}{\lambda_n}$, where λ_1 is the smallest eigenvalue of H and λ_n is the largest eigenvalue of H.

* when $\frac{\lambda_1}{\lambda_n} \approx 1$, gradient descent converges to x^* fast.
* when $\frac{\lambda_1}{\lambda_n} << 1$, gradient descent converges very very slowly.

For the first function, H = 2I, so its eigenvalues are both 2.

The condition number is $\lambda_1 / \lambda_n = 2/2 = 1$, leading to immediate convergence.

For the second function, the eigenvalues of H are found by solving the characteristic equation $\det(H - \lambda I) = 0$.

For H = [[8, 2], [2, 2]], the characteristic equation is:

$(8-\lambda)(2-\lambda) - (2)(2) = 0$

$\lambda^2 - 10\lambda + 12 = 0$

The eigenvalues are $\lambda = (10 \pm \sqrt{100 - 48}) / 2 = (10 \pm \sqrt{52}) / 2 \approx (10 \pm 7.211) / 2$.

Thus, $\lambda_n \approx (10 + 7.211)/2 \approx 8.6055$ and $\lambda_1 \approx (10 - 7.211)/2 \approx 1.3945$.

The condition number is approximately $\lambda_1 / \lambda_n \approx 1.3945 / 8.6055 \approx 0.162$.

This small condition number explains the slow convergence observed.

In the next class, we will analyze why the condition number is so important for the convergence rate of the gradient descent algorithm. Thank you.