Optimization Algorithms: Theory and Software Implementation

Prof. Thirumulanathan D

Department of Mathematics

Institute of IIT Kanpur

Lecture: 22

Hello everyone, this is the second lecture of week five. We will continue our discussion on Newton's method.

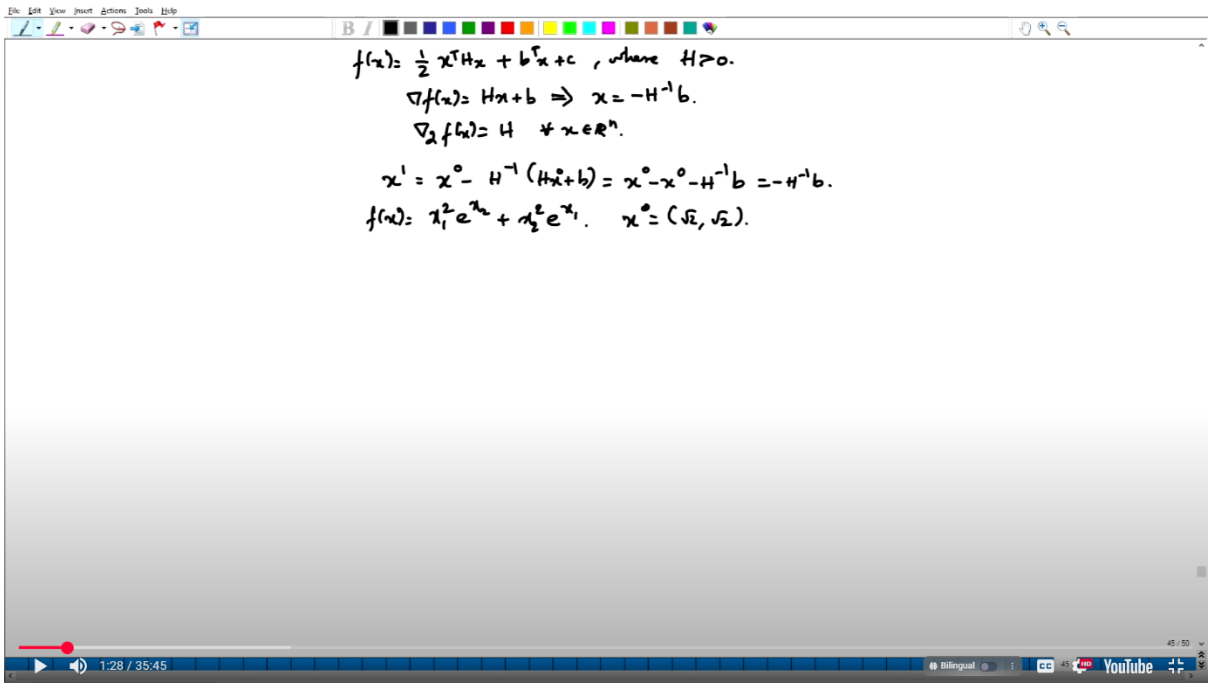
In the last lecture, we learned the general form of the Newton's method algorithm and looked at an example with a quadratic function. We proved analytically that for a quadratic function, Newton's method finds the solution in a single step, and we also demonstrated this with a Python code.

Now, we will look at another example before moving on to discuss the properties and potential issues with Newton's method. We will again consider the function:

$f(x_1, x_2) = x_1^2 e^{x_2} + x_2^2 e^{x_1}$

and use the initial point $x_0 = (\sqrt{2}, \sqrt{2})$.

(Refer Slide Time 1:28)



We will implement this using a Python code. For this, we need to define the function f(x), its gradient grad(x), and its Hessian matrix H(x).

Here is the Python code for Newton's method for this general function:

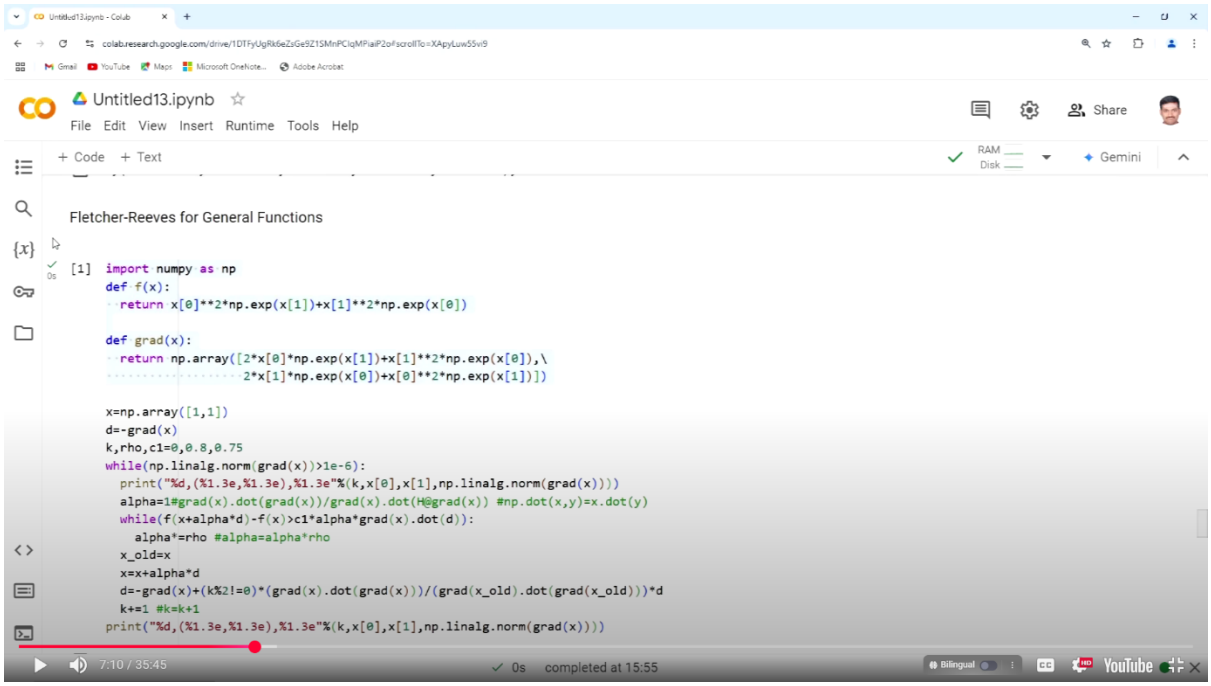
Python

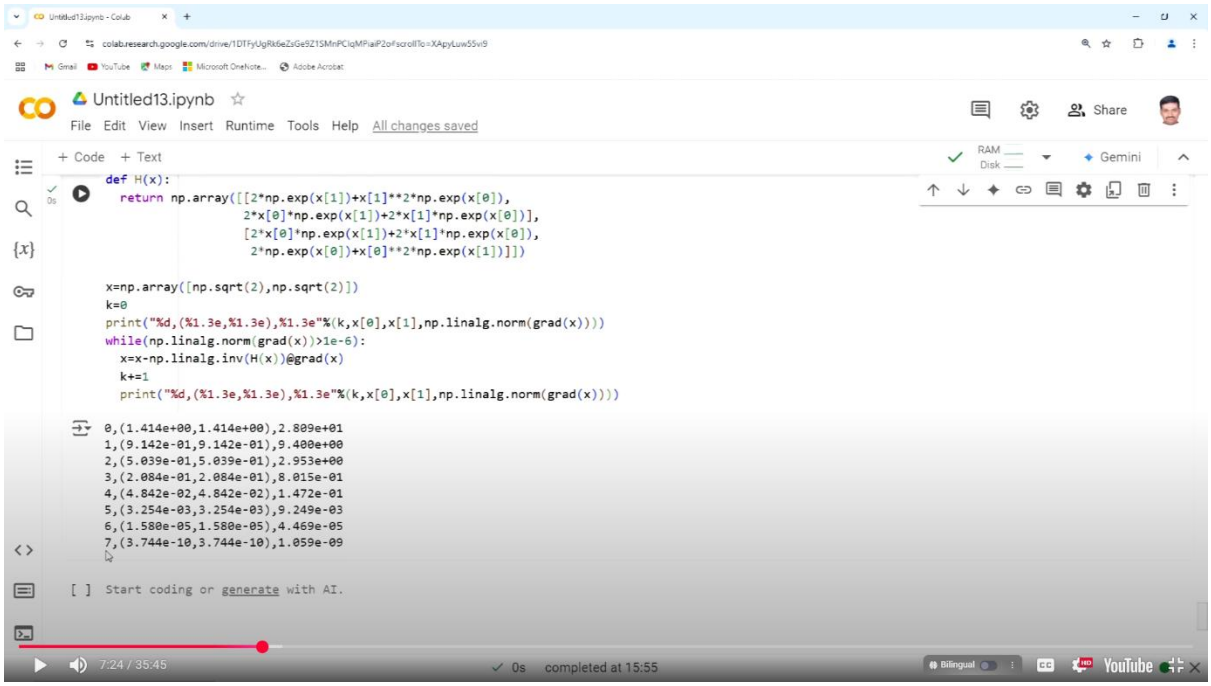
```
import numpy as np
# Define the function, its gradient, and its Hessian
def f(x):
    return x[0]**2 * np.exp(x[1]) + x[1]**2 * np.exp(x[0])
def grad(x):
    g1 = 2*x[0]*np.exp(x[1]) + x[1]**2*np.exp(x[0])
    g2 = x[0]**2*np.exp(x[1]) + 2*x[1]*np.exp(x[0])
    return np.array([g1, g2])
def H(x):
    h11 = 2*np.exp(x[1]) + x[1]**2*np.exp(x[0])
    h12 = 2*x[0]*np.exp(x[1]) + 2*x[1]*np.exp(x[0])
    h21 = 2*x[0]*np.exp(x[1]) + 2*x[1]*np.exp(x[0])
    h22 = 2*np.exp(x[0]) + x[0]**2*np.exp(x[1])
    return np.array([[h11, h12], [h21, h22]])

# Algorithm parameters
x = np.array([np.sqrt(2), np.sqrt(2)])  # Initial point
tol = 1e-6
k = 0
max_iter = 100
print(f"k = {k}: x = {x}, f(x) = {f(x)}")
# Main optimization loop
while np.linalg.norm(grad(x)) > tol and k < max_iter:
    g = grad(x)
    H_inv = np.linalg.inv(H(x))
    d = -H_inv @ g  # Newton direction
    # Newton update (step size alpha = 1)
    x = x + d
```

```
    k += 1
    print(f"k = {k}: x = {x}, f(x) = {f(x)}")
print(f"\nSolution: {x}")
print(f"Number of iterations: {k}")
```

This code converges to the solution (0, 0) in about 7 iterations. In previous lectures, the conjugate gradient (Fletcher-Reeves) and gradient descent methods took about 30 steps for the same function. This demonstrates that Newton's method is significantly faster for many functions.

(refer Slide Time 7:24)





The reason for this speed is the order of convergence. The algorithms we learned before (gradient descent, conjugate gradient) have a linear rate of convergence (order p = 1). Newton's method has a quadratic rate of convergence (order p = 2).

The order of convergence p is defined by the following limit behavior for large k:

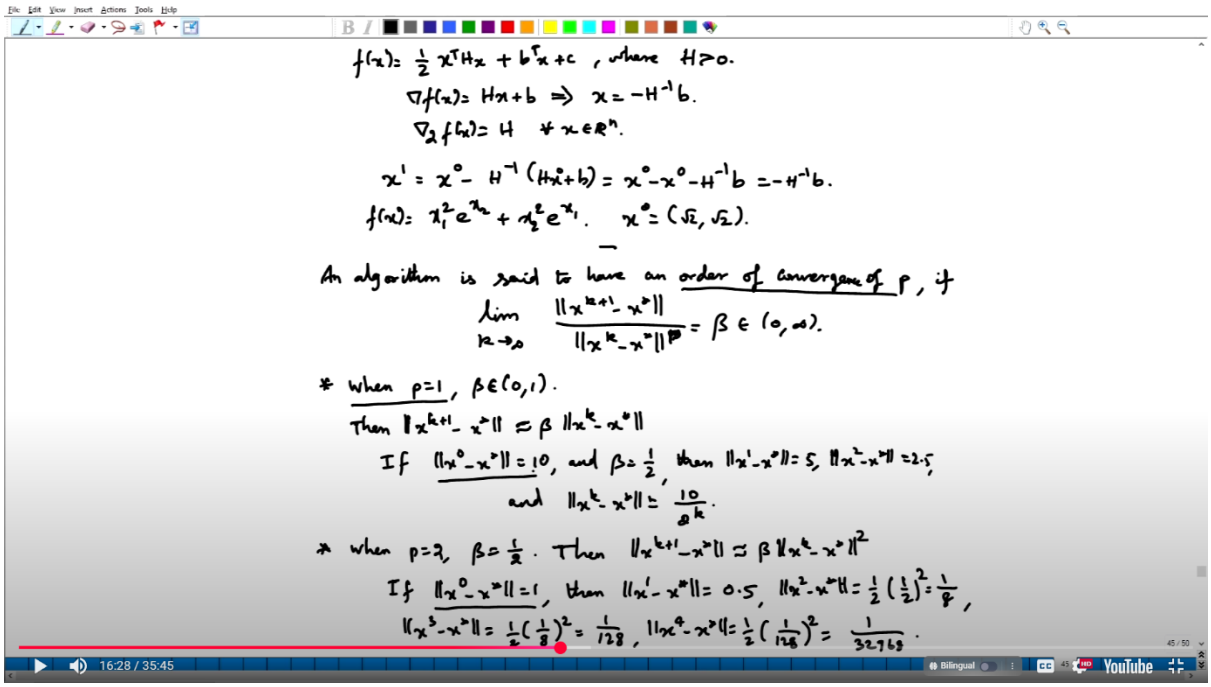
$\|x_{k+1} - x^*\| \leq \beta * \|x_k - x^*\|^p$

where β is a constant and x^* is the true solution.

* **Linear Convergence (p=1):** The error decreases exponentially. If $\|x_k - x^*\| = 10$ and $\beta=0.5$, the errors become 5, 2.5, 1.25, 0.625, ...

* **Quadratic Convergence (p=2):** The error decreases much faster. If $\|x_k - x^*\| = 1$ and $\beta=0.5$, the errors become 0.5, 0.125, 0.0078, ... The number of correct digits roughly doubles with each step.

(Refer Slide Time 16:28)



However, this immense speed comes with significant drawbacks:

1. **No Global Convergence**: Newton's method is only locally convergent. This means it will converge quadratically only if the initial guess x_0 is already sufficiently close to the solution x^*. If started too far away, it may not converge at all, or even diverge, as the quadratic model becomes a poor approximation of the function.

2. **Descent Direction Not Guaranteed:** The Newton direction $d_k = -H_k^{-1}g_k$ is a descent direction only if the Hessian H_k is positive definite. For non-convex functions, the Hessian may not be positive definite at some points, causing the algorithm to potentially increase the function value.

3. **Computational Expense:** Inverting the Hessian matrix H_k (or solving the linear system $H_k d_k = -g_k$) in each iteration is computationally expensive, especially for high-dimensional problems (e.g., in machine learning with thousands of variables). The Hessian might also be singular (non-invertible).

To illustrate the first issue (no global convergence), consider the function:

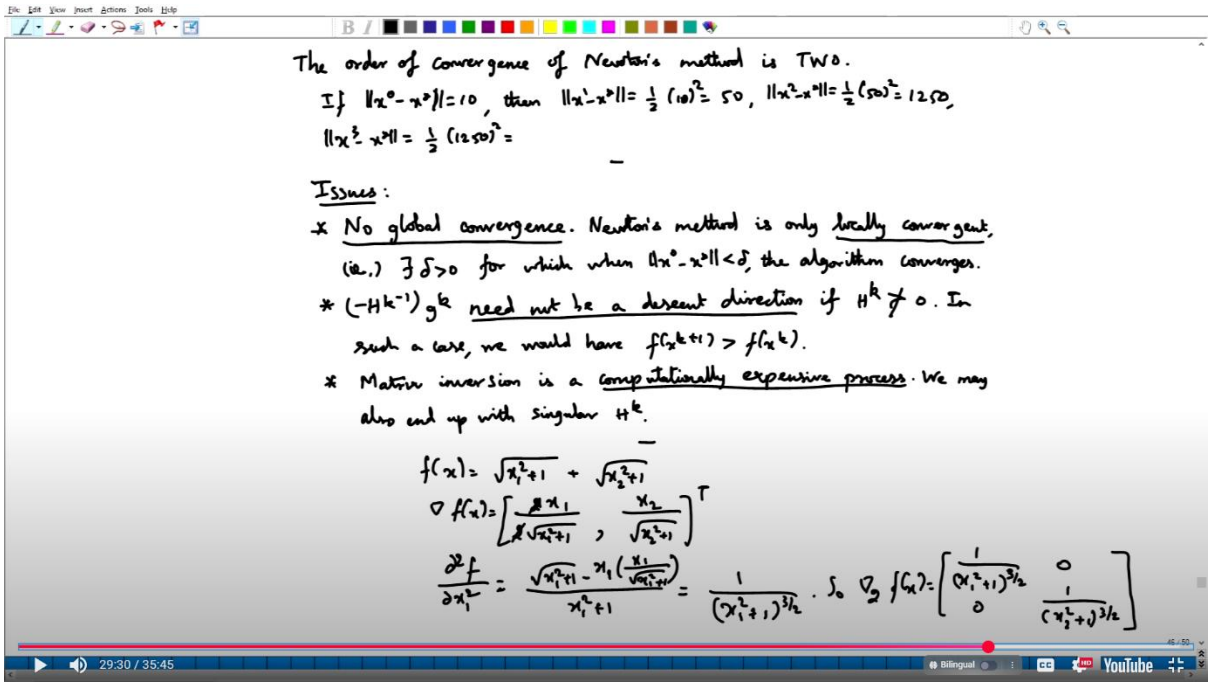
$f(x) = \sqrt{(x_1^2 + 1)} + \sqrt{(x_2^2 + 1)}$

Its minimum is at (0, 0). Its gradient and Hessian are:

$\nabla f(x) = [\ x_1/\sqrt{(x_1^2+1)},\ x_2/\sqrt{(x_2^2+1)}\ ]$

$H(x) = \text{diag}(\ 1/(x_1^2+1)^{(3/2)},\ 1/(x_2^2+1)^{(3/2)}\ )$ (a diagonal matrix)

(Refer Slide Time 29:10)



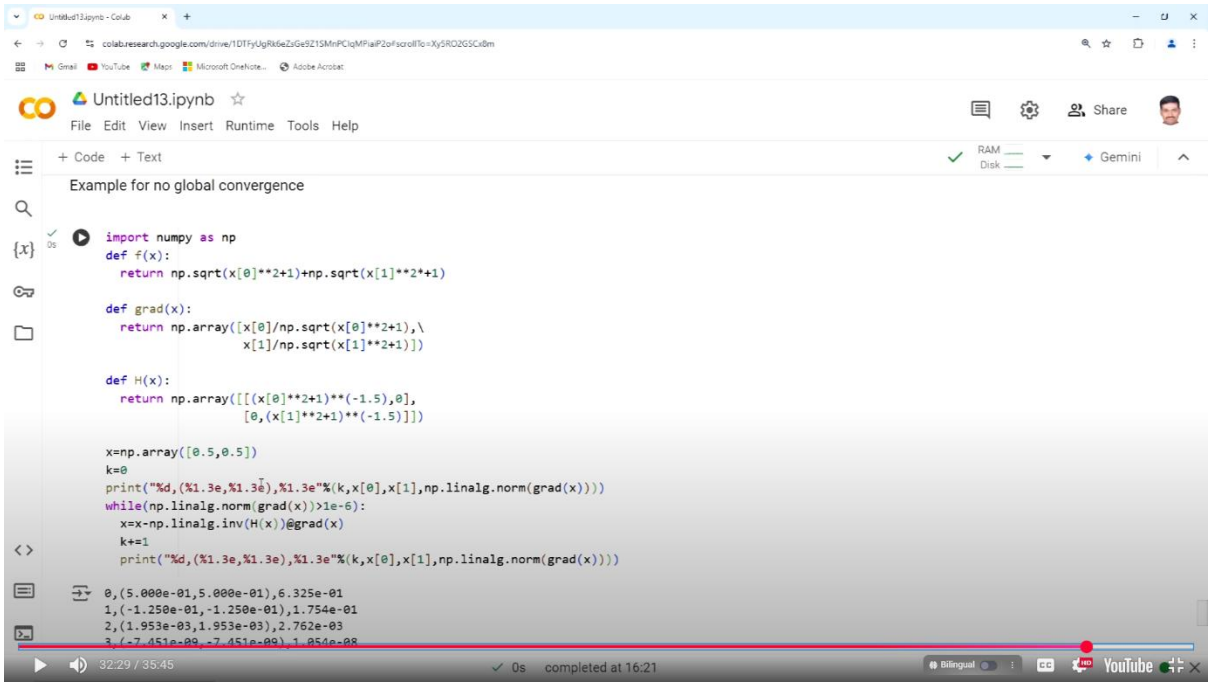
If you run Newton's method on this function:

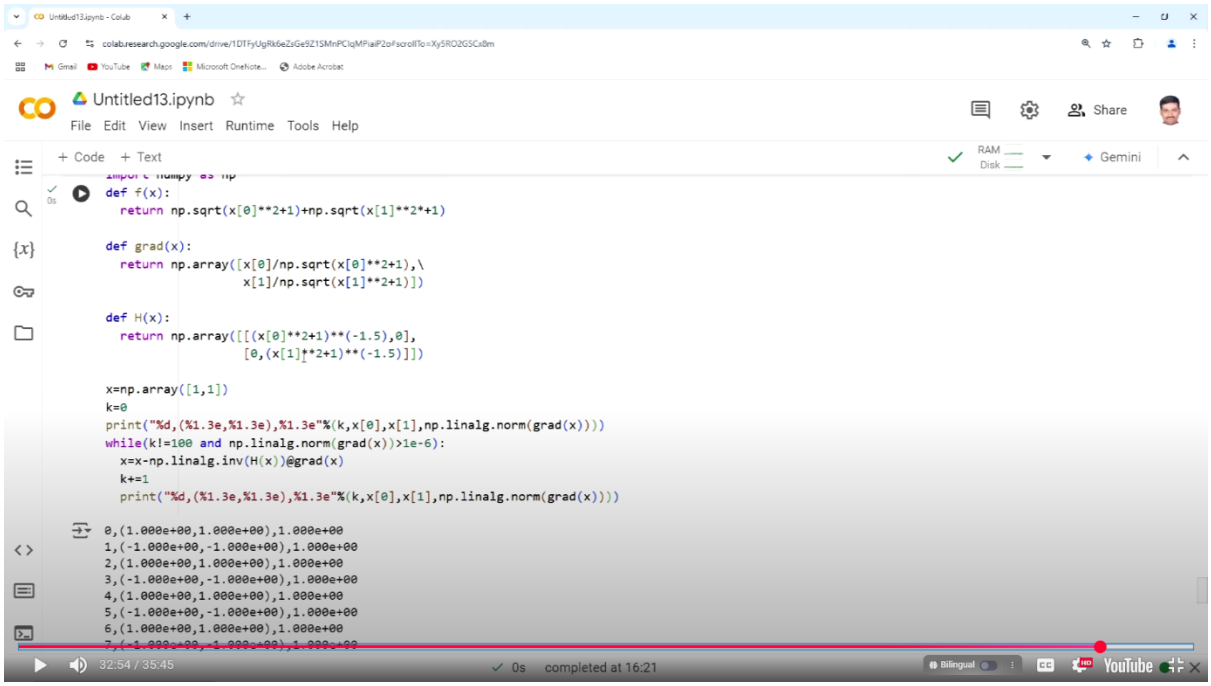
Python

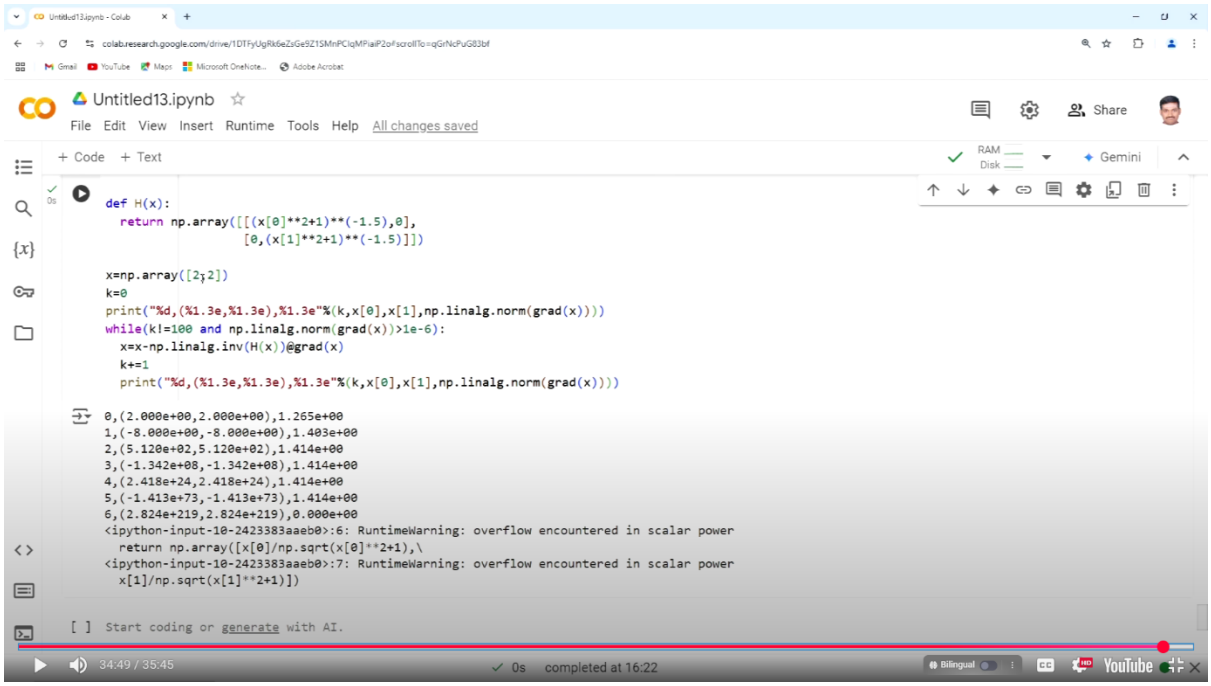
* Starting from (0.5, 0.5) (close to the solution), it converges quickly to (0, 0).

* Starting from (2, 2) (farther away), the algorithm can diverge, sending the iterates to very large values, demonstrating the lack of global convergence.

(Refer Slide Time 34:49)


Untitled13.ipynb
File Edit View Insert Runtime Tools Help
Share
+ Code + Text
RAM
Disk
Gemini
Example for no global convergence
import numpy as np
def f(x):
return np.sqrt(x[0]**2+1)+np.sqrt(x[1]**2+1)
def grad(x):
return np.array([x[0]/np.sqrt(x[0]**2+1),\
x[1]/np.sqrt(x[1]**2+1)])
def H(x):
return np.array([[(x[0]**2+1)**(-1.5),0],
[0,(x[1]**2+1)**(-1.5)]])
x=np.array([0.5,0.5])
k=0
print("%d,(%1.3e,%1.3e),%1.3e"%(k,x[0],x[1],np.linalg.norm(grad(x))))
while(np.linalg.norm(grad(x))>1e-6):
x=x-np.linalg.inv(H(x))@grad(x)
k+=1
print("%d,(%1.3e,%1.3e),%1.3e"%(k,x[0],x[1],np.linalg.norm(grad(x))))
0,(5.000e-01,5.000e-01),6.325e-01
1,(-1.250e-01,-1.250e-01),1.754e-01
2,(1.953e-03,1.953e-03),2.762e-03
32:29 / 35:45
0s completed at 16:21
YouTube


Untitled13.ipynb
File Edit View Insert Runtime Tools Help
Share
+ Code + Text
RAM
Disk
Gemini
def f(x):
return np.sqrt(x[0]**2+1)+np.sqrt(x[1]**2+1)
def grad(x):
return np.array([x[0]/np.sqrt(x[0]**2+1),\
x[1]/np.sqrt(x[1]**2+1)])
def H(x):
return np.array([[(x[0]**2+1)**(-1.5),0],
[0,(x[1]**2+1)**(-1.5)]])
x=np.array([1,1])
k=0
print("%d,(%1.3e,%1.3e),%1.3e"%(k,x[0],x[1],np.linalg.norm(grad(x))))
while(k!=100 and np.linalg.norm(grad(x))>1e-6):
x=x-np.linalg.inv(H(x))@grad(x)
k+=1
print("%d,(%1.3e,%1.3e),%1.3e"%(k,x[0],x[1],np.linalg.norm(grad(x))))
0,(1.000e+00,1.000e+00),1.000e+00
1,(-1.000e+00,-1.000e+00),1.000e+00
2,(1.000e+00,1.000e+00),1.000e+00
3,(-1.000e+00,-1.000e+00),1.000e+00
4,(1.000e+00,1.000e+00),1.000e+00
5,(-1.000e+00,-1.000e+00),1.000e+00
6,(1.000e+00,1.000e+00),1.000e+00
32:54 / 35:45
0s completed at 16:21
YouTube



In the next lecture, we will look at examples for the other issues (non-descent directions and singular Hessians). Thank you.