Optimization Algorithms: Theory and Software Implementation

Prof. Thirumulanathan D

Department of Mathematics

Institute of IIT Kanpur

Lecture: 21

Hello everyone, we will now start with Newton's method in this fifth week.

So far, we have learned about line search algorithms, both exact and backtracking.

We have also learned about algorithms that choose descent directions, specifically the gradient descent algorithm and different forms of conjugate gradient algorithms.

This included one method using eigenvectors and the Fletcher-Reeves algorithm, for which we saw implementations for both quadratic and general functions.

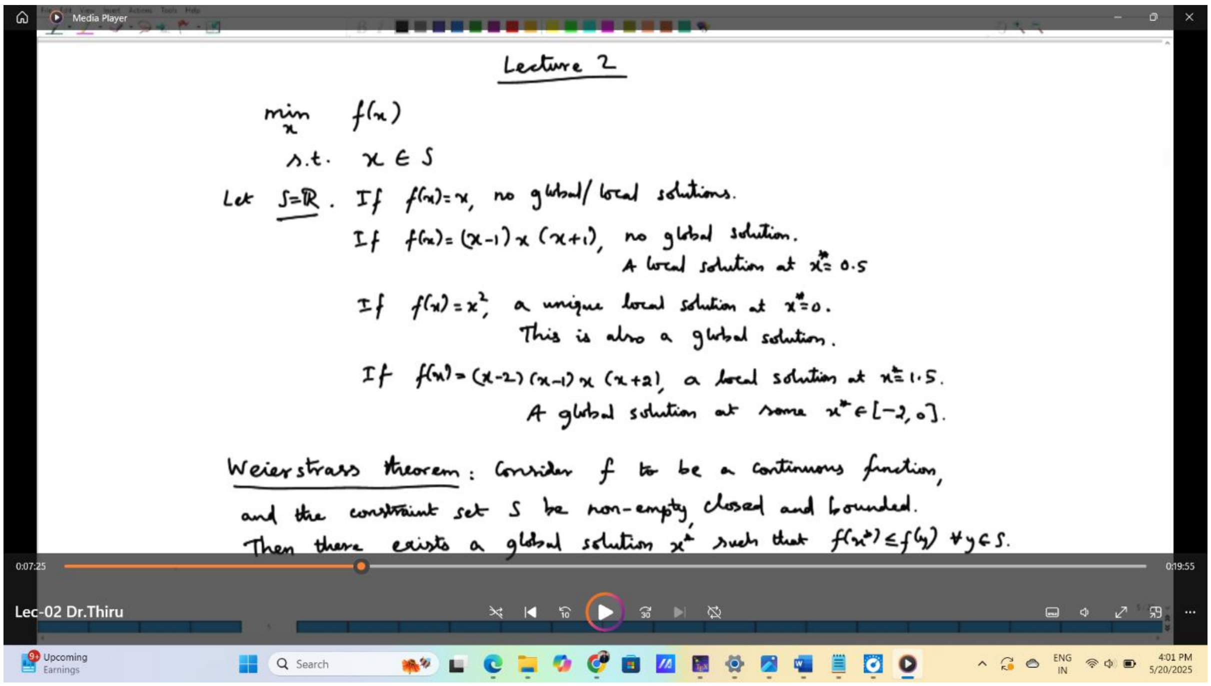
Before we start with Newton's method, I would like to revisit the Fletcher-Reeves algorithm for general functions by providing a complete code example.

We will use the function:

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

We will implement the algorithm using a framework from our previous codes, specifically the gradient descent with backtracking line search.

(Refer Slide Time 2:43)



Lecture 2

$\min_x \; f(x)$

s.t. $x \in S$

Let $S=\mathbb{R}$. If $f(x)=x$, no global/local solutions.

If $f(x)=(x-1)x(x+1)$, no global solution.

A local solution at $x^* = 0.5$

If $f(x)=x^2$, a unique local solution at $x^*=0$.

This is also a global solution.

If $f(x)=(x-2)(x-1)x(x+2)$, a local solution at $x^* \approx 1.5$.

A global solution at some $x^* \in [-2,0]$.

Weierstrass theorem: Consider f to be a continuous function, and the constraint set S be non-empty, closed and bounded. Then there exists a global solution x^* such that $f(x^*) \le f(y) \; \forall y \in S$.

We will define the function f(x) and its gradient grad(x).

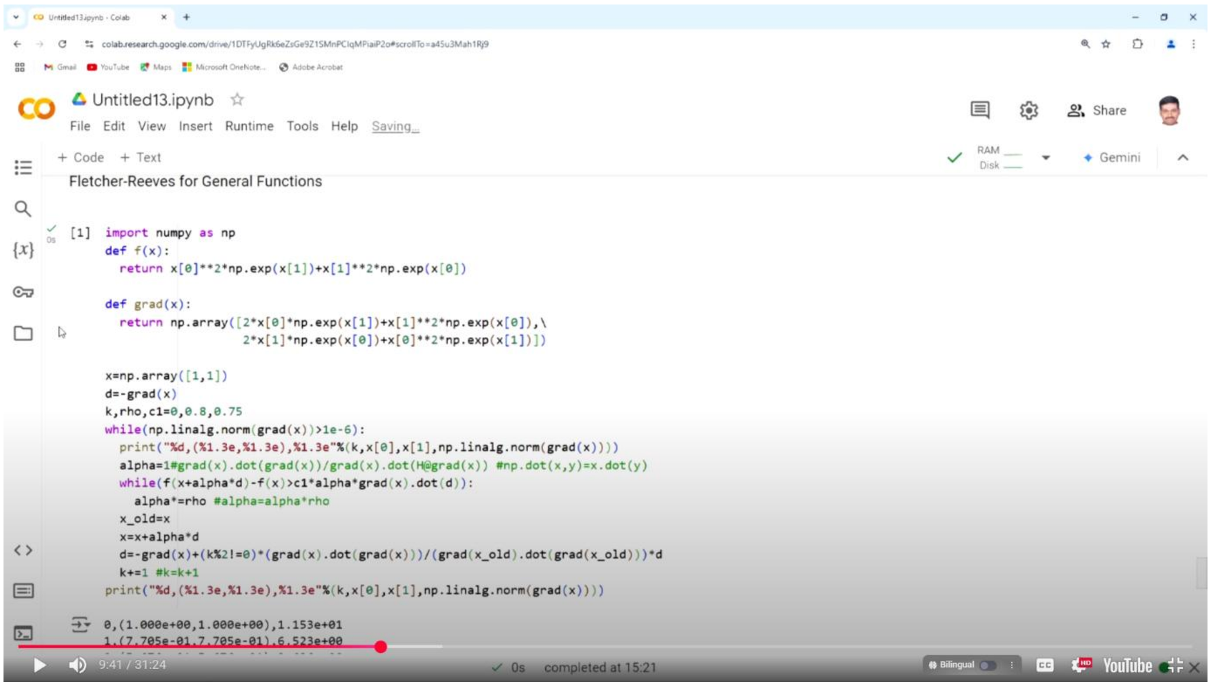
Here is the Python code for the Fletcher-Reeves algorithm with backtracking line search:

Python

```
import numpy as np
# Define the function and its gradient
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])
# Algorithm parameters
x = np.array([1.0, 1.0])  # Initial point
tol = 1e-5
k = 0
max_iter = 1000
rho = 0.5
c1 = 1e-4
# Initialize the first direction
g = grad(x)
d = -g

# Main optimization loop
while np.linalg.norm(g) > tol and k < max_iter:
    # Backtracking line search
    alpha = 1.0
    while f(x + alpha*d) > f(x) + c1 * alpha * np.dot(g, d):
        alpha = rho * alpha
    # Update the point
    x_old = x
    g_old = g
    x = x + alpha * d
    g = grad(x)
    # Fletcher-Reeves beta calculation and direction update
    beta = np.dot(g, g) / np.dot(g_old, g_old)
    # Check if we need to reset the direction (every n steps, n=2 here)
    if k % 2 != 0:
        d = -g + beta * d
    else:
        d = -g  # Reset to steepest descent
    k += 1
print(f"Solution: {x}")
print(f"Number of iterations: {k}")
```

This code should find the minimum at (0, 0) in about 30 steps.

(Refer Slide Time 9:41)

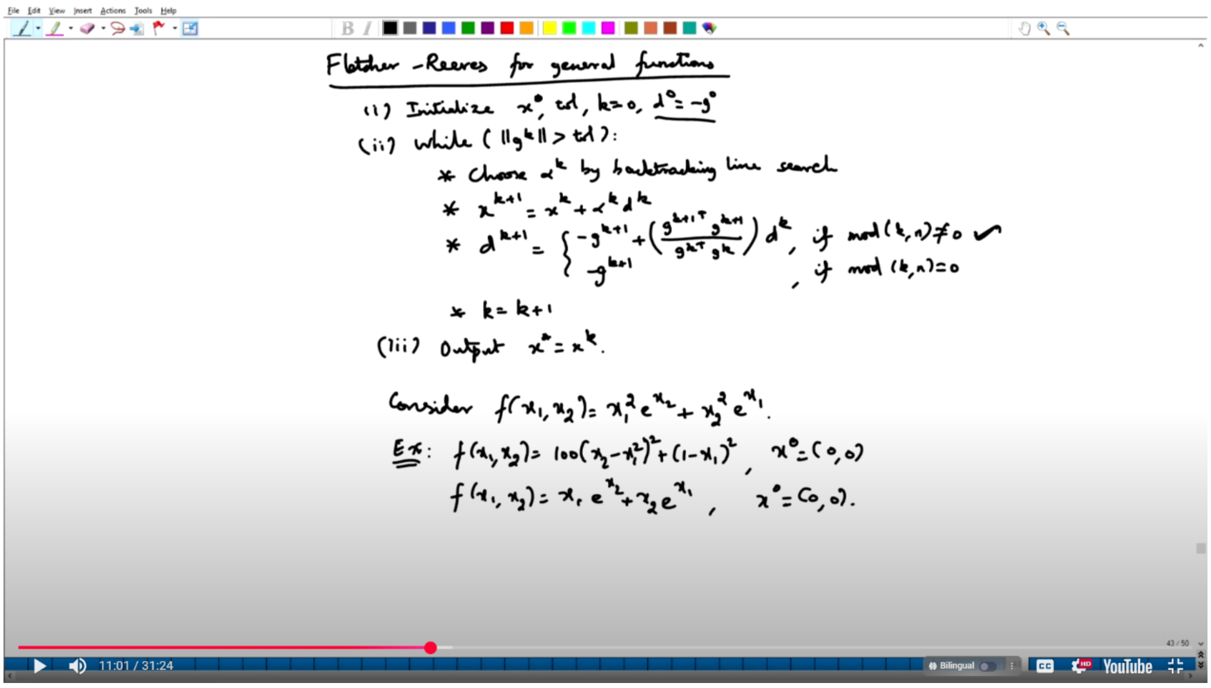


Fletcher-Reeves for General Functions

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

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

    x=np.array([1,1])
    d=-grad(x)
    k,rho,c1=0,0.8,0.75
    while(np.linalg.norm(grad(x))>1e-6):
      print("%d,(%1.3e,%1.3e),%1.3e"%(k,x[0],x[1],np.linalg.norm(grad(x))))
      alpha=1#grad(x).dot(grad(x))/grad(x).dot(H@grad(x)) #np.dot(x,y)=x.dot(y)
      while(f(x+alpha*d)-f(x)>c1*alpha*grad(x).dot(d)):
        alpha*=rho #alpha=alpha*rho
      x_old=x
      x=x+alpha*d
      d=-grad(x)+(k%2!=0)*(grad(x).dot(grad(x)))/(grad(x_old).dot(grad(x_old)))*d
      k+=1 #k=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),1.153e+01
    1,(7.705e-01,7.705e-01),6.523e+00
```

You can modify the function f(x), grad(x), and the initial point to test other functions, like the Rosenbrock function.

(Refer Slide Time 11:01)



Fletcher-Reeves for general functions

```
(i) Initialize $x^0$, tol, $k=0$, $d^0 = -g^0$
(ii) while ( $\|g^k\| > tol$ ):
        * Choose $\alpha^k$ by backtracking line search
        * $x^{k+1} = x^k + \alpha^k d^k$
        * $d^{k+1} = \begin{cases} -g^{k+1} + \left(\frac{g^{k+1^T} g^{k+1}}{g^{k^T} g^k}\right) d^k, & \text{if } \mathrm{mod}(k,n) \neq 0 \checkmark \\ -g^{k+1}, & \text{if } \mathrm{mod}(k,n) = 0 \end{cases}$
        * $k = k+1$
(iii) Output $x^* = x^k$.
```

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

Ex: $f(x_1, x_2) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$, $x^0 = (0,0)$

$f(x_1, x_2) = x_1 e^{x_2} + x_2 e^{x_1}$, $x^0 = (0,0)$.

With this, we conclude our discussion on the conjugate gradient algorithms.

Now, let's start with **Newton's method.**

The key difference between Newton's method and the previous methods (gradient descent, conjugate gradient) lies in the approximation of the function we use to find the descent direction.

Until now, all our algorithms were based on the first-order Taylor approximation:

$f(x) \approx f(x_k) + \nabla f(x_k)^T (x - x_k)$

Newton's method is constructed using the *second-order* Taylor approximation:

$f(x) \approx f(x_k) + \nabla f(x_k)^T (x - x_k) + (1/2)(x - x_k)^T \nabla^2 f(x_k)(x - x_k)$

Let's see how we derive the method from this.

Let x* be the minimizer of this local quadratic approximation.

We find it by setting the derivative of the approximation to zero:

$\nabla f(x_k) + \nabla^2 f(x_k)(x^* - x_k) = 0$

Solving for x*:

$x^* - x_k = -(\nabla^2 f(x_k))^{-1} \nabla f(x_k)$

We introduce shorthand notation:

$g_k = \nabla f(x_k)$ (Gradient)

$H_k = \nabla^2 f(x_k)$ (Hessian)

This gives us the Newton step:

$x^* = x_k - H_k^{-1} g_k$

Therefore, in the algorithm, we choose:

Descent direction: $d_k = -H_k^{-1} g_k$

Step size: $\alpha_k = 1$

The Newton's method algorithm is very simple:

```
1. Initialize $x_0$, set $k = 0$
2. While $\|g_k\|$ > tolerance:
   a. Compute $g_k = \nabla f(x_k)$
   b. Compute $H_k = \nabla^2 f(x_k)$
   c. Solve $H_k d_k = -g_k$ for $d_k$ (instead of explicitly inverting $H_k$)
   d. $x_{k+1} = x_k + d_k$
   e. $k = k + 1$
3. Output $x^* = x_k$
```

A crucial question is: Is $d_k = -H_k^{-1} g_k$ always a descent direction?

A direction is descent if $\nabla f(x_k)^T d_k < 0$.

Let's check:

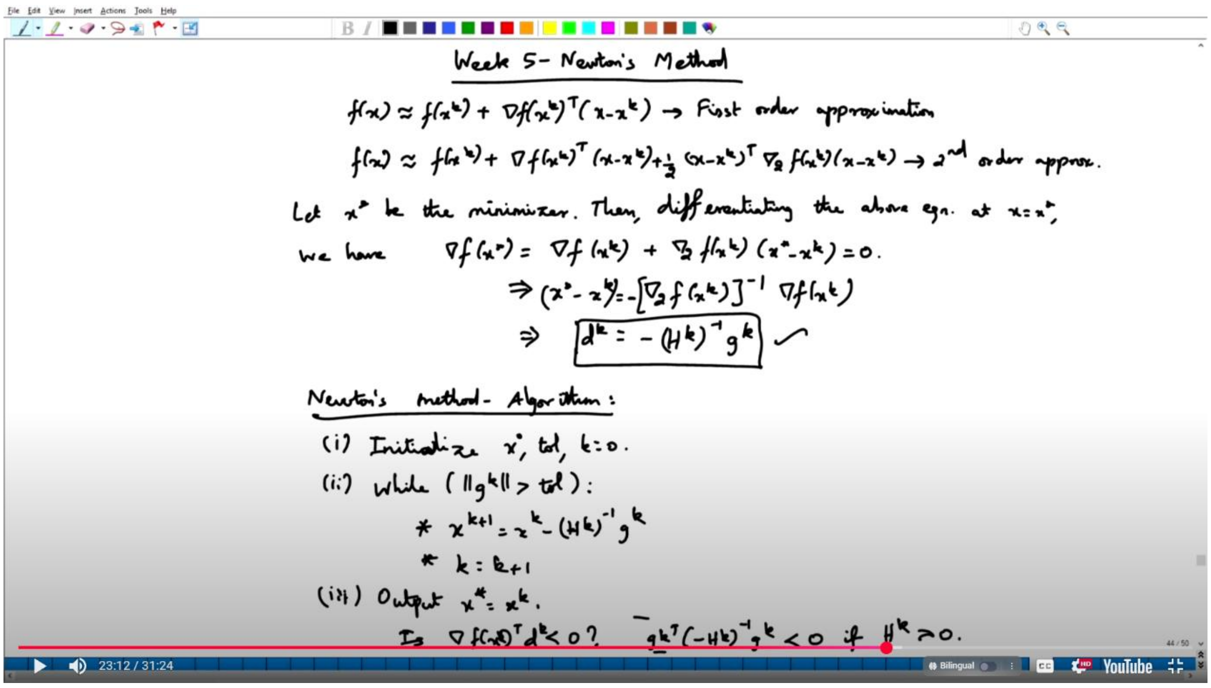
$\nabla f(x_k)^T d_k = g_k^T (-H_k^{-1} g_k) = -g_k^T H_k^{-1} g_k$

This expression is negative only if H_k^{-1} is positive definite (which is true if H_k is positive definite).

Therefore, Newton's method provides a descent direction if the Hessian H_k is positive definite at each iteration.

This is always true for convex functions.

(Refer Slide Time 23:12)



Week 5- Newton's Method

$f(x) \approx f(x^k) + \nabla f(x^k)^T(x-x^k) \rightarrow$ First order approximation

$f(x) \approx f(x^k) + \nabla f(x^k)^T(x-x^k) + \frac{1}{2}(x-x^k)^T \nabla_2 f(x^k)(x-x^k) \rightarrow 2^{nd}$ order approx.

Let x^* be the minimizer. Then, differentiating the above eqn. at $x=x^*$,
we have $\nabla f(x^*) = \nabla f(x^k) + \nabla_2 f(x^k)(x^*-x^k) = 0.$

$$\Rightarrow (x^*-x^k) = -[\nabla_2 f(x^k)]^{-1} \nabla f(x^k)$$

$$\Rightarrow \boxed{d^k = -(H^k)^{-1} g^k} \checkmark$$

Newton's method- Algorithm:

```
(i) Initialize $x^0$, tol, $k=0$.
(ii) While ($\|g^k\| >$ tol):
        * $x^{k+1} = x^k - (H^k)^{-1} g^k$
        * $k = k+1$
(iii) Output $x^* = x^k$.
```

Is $\nabla f(x^k)^T d^k < 0$? $\underline{g}^{k^T}(-H^k)^{-1} g^k < 0$ if $H^k \succ 0$.

Now, let's look at an example. We will consider a quadratic function:

$f(x) = (1/2)x^T Hx + b^T x + c$, with H positive definite.

For this function:

$\nabla f(x) = Hx + b$

$\nabla^2 f(x) = H$ (constant)

Let's apply Newton's method theoretically from any initial point x_0:

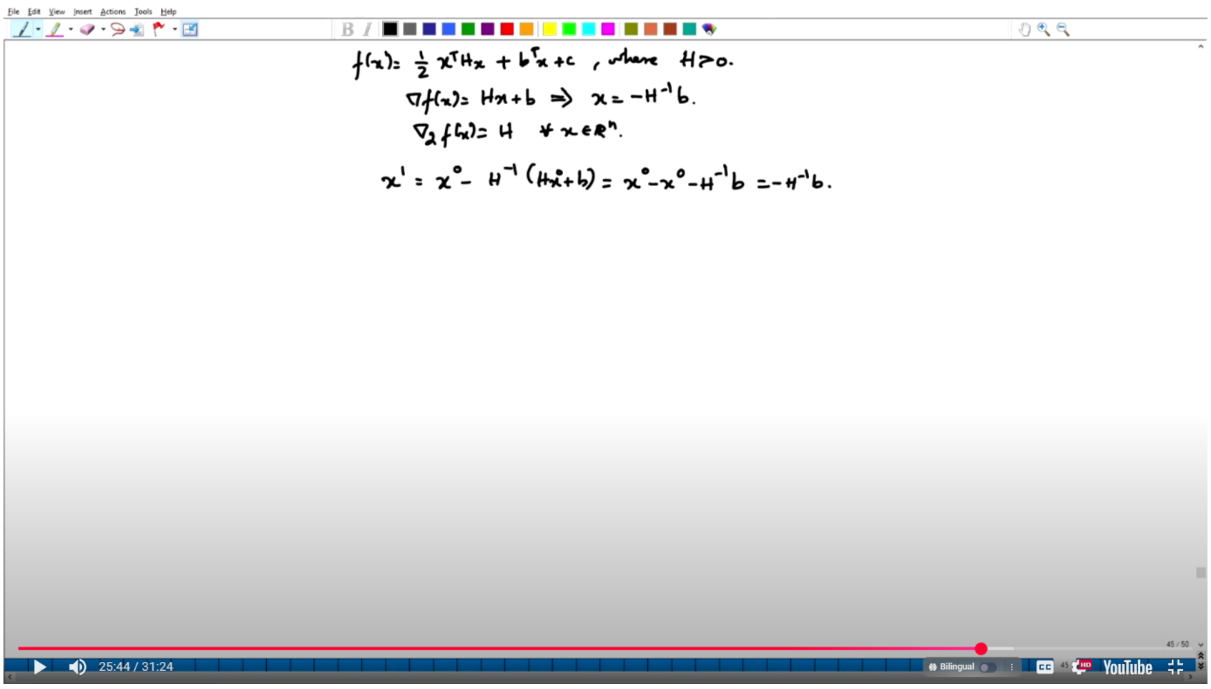
$x_1 = x_0 - H^{-1}(Hx_0 + b)$

$x_1 = x_0 - x_0 - H^{-1}b$

$x_1 = -H^{-1}b$

This is the exact solution! Newton's method finds the minimum of a quadratic function in a single step, regardless of the initial point or the condition number of H.

(Refer Slide Time 25:44)



$f(x) = \frac{1}{2} x^T H x + b^T x + c$, where $H > 0$.

$\nabla f(x) = Hx + b \Rightarrow x = -H^{-1} b$.

$\nabla_2 f(x) = H \quad \forall x \in R^n$.

$x^1 = x^0 - H^{-1}(Hx^0 + b) = x^0 - x^0 - H^{-1} b = -H^{-1} b$.

Let's verify this with a Python code example:

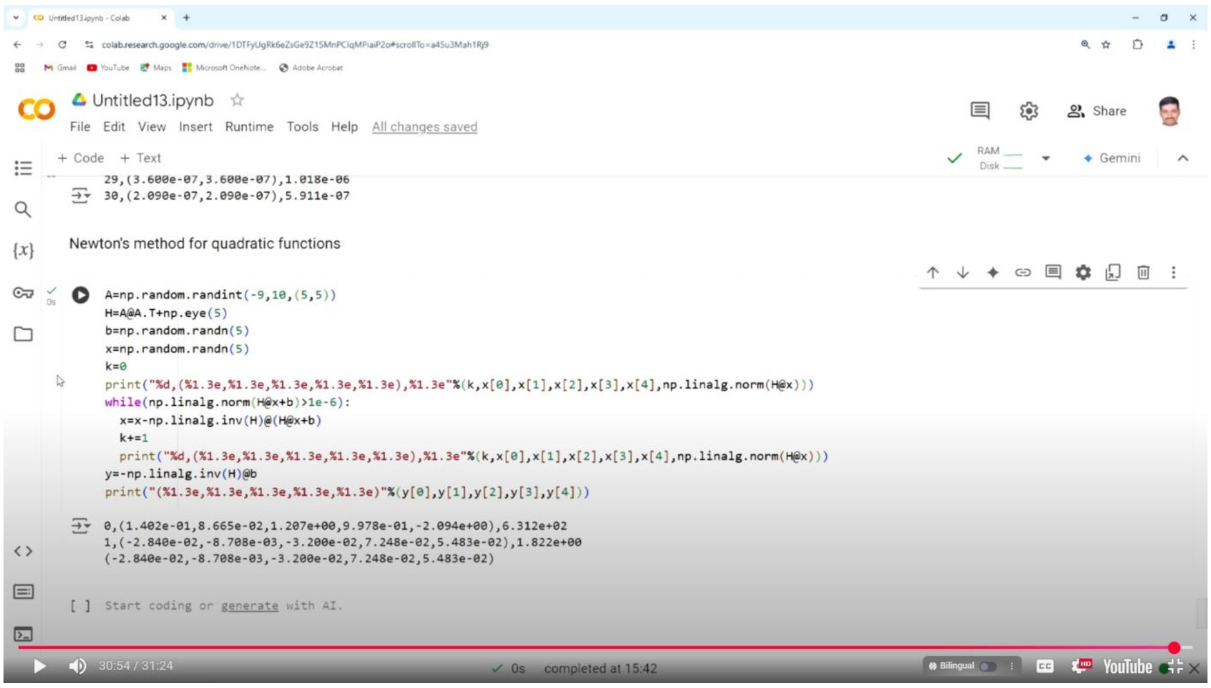
Python

```
import numpy as np

# Generate a random positive definite matrix H

n = 5

A = np.random.rand(n, n)

H = A @ A.T + np.eye(n)  # A*A^T + I is positive definite

# Generate a random vector b

b = np.random.rand(n)

# Choose a random initial point x0

x0 = np.random.rand(n)

# Newton's step

x1 = x0 - np.linalg.inv(H) @ (H @ x0 + b)

# Calculate the true solution for comparison

true_solution = -np.linalg.inv(H) @ b

print("Initial point x0:\n", x0)

print("\nNewton's step result x1:\n", x1)

print("\nTrue solution -H^{-1}b:\n", true_solution)
```

print("\nIs x1 equal to the true solution?", np.allclose(x1, true_solution))

This code will show that `x1` is indeed equal to the true solution, demonstrating the effectiveness of Newton's method for quadratic functions.

(Refer Slide Time 30:54)



```
29,(3.600e-07,3.600e-07),1.018e-06
30,(2.090e-07,2.090e-07),5.911e-07
```

Newton's method for quadratic functions

```
A=np.random.randint(-9,10,(5,5))
H=A@A.T+np.eye(5)
b=np.random.randn(5)
x=np.random.randn(5)
k=0
print("%d,(%1.3e,%1.3e,%1.3e,%1.3e,%1.3e),%1.3e"%(k,x[0],x[1],x[2],x[3],x[4],np.linalg.norm(H@x)))
while(np.linalg.norm(H@x+b)>1e-6):
  x=x-np.linalg.inv(H)@(H@x+b)
  k+=1
  print("%d,(%1.3e,%1.3e,%1.3e,%1.3e,%1.3e),%1.3e"%(k,x[0],x[1],x[2],x[3],x[4],np.linalg.norm(H@x)))
y=-np.linalg.inv(H)@b
print("(%1.3e,%1.3e,%1.3e,%1.3e,%1.3e)"%(y[0],y[1],y[2],y[3],y[4]))
```

```
0,(1.402e-01,8.665e-02,1.207e+00,9.978e-01,-2.094e+00),6.312e+02
1,(-2.840e-02,-8.708e-03,-3.200e-02,7.248e-02,5.483e-02),1.822e+00
(-2.840e-02,-8.708e-03,-3.200e-02,7.248e-02,5.483e-02)
```

This makes Newton's method appear very powerful. We will discuss its properties and potential issues in the next lecture. Thank you.