Optimization Algorithms: Theory and Software Implementation

Prof. Thirumulanathan D

Department of Mathematics

Institute of IIT Kanpur

Lecture: 16

Hello everyone, so this will be the beginning of the fourth week of this course on optimization algorithms. In the first three weeks, we learned about an introduction to optimization, then in the second week, an introduction to Python coding, and in the last week, we learned why iterative algorithms are needed to solve optimization problems. We also learned about line search algorithms where you find the step size α, and we started with the gradient descent algorithm. We will continue with the gradient descent algorithm.

If you recall where we stopped last week, we choose the negative of the gradient of f at x_k as the descent direction.

We started with two examples: one was the very simple example

$x_1^2 + x_2^2$,

and for any starting point, we arrived at the answer in just one step. Just as we were beginning to feel happy that this would happen for many other functions, we gave another example where the domain was just $\mathbb{R}^2$ and the function was quadratic as well, but the number of steps taken to converge was about 30 or 31. The end of the last lecture was regarding the condition number. The condition number of any quadratic function depends on the matrix H.

If λ_1 is the smallest eigenvalue of H and λ_n is the largest eigenvalue of H, then λ_1/λ_n is defined as the condition number.

I gave this result: when λ_1/λ_n is approximately 1, then the gradient descent algorithm converges to x* very fast.

For example, in the case of $x_1^2 + x_2^2$, H is [[2, 0], [0, 2]],

which means $\lambda_1 = \lambda_2 = 2$.

So, $\lambda_1/\lambda_2 = 1$, and the gradient descent algorithm converged in just one iteration.

But when λ_1/λ_n is much less than 1, like in the other example where the eigenvalues were roughly 8.6 and 1.4, then λ_1/λ_2 was more than 6, and gradient descent converged much more slowly.

You might still have the question about where this λ_1/λ_n comes from.

We saw two examples: in one, λ_1/λ_2 was 1, and in the other, it was less than 1/6, and this result held. But why is this result true?

I am going to mathematically derive why λ_1/λ_n has such a big impact on the number of iterations, the speed of convergence of the algorithm.

I will first define an error function.

The error function is given by $(1/2)(x_k - x^*)^T H(x_k - x^*)$, where x^* is the solution.

Here, $x^* = -H^{-1}b$ is the solution to the problem min over x of $(1/2)x^T Hx + b^T x + c$.

You take a general quadratic function and minimize over x.

Differentiating and equating to zero, you get $Hx + b = 0$, so $x^* = -H^{-1}b$.

Let us define an error function for any x, say x_k, the value of x at the k-th iteration of the gradient descent algorithm:

$E(x_k) = (1/2)(x_k - x^*)^T H(x_k - x^*)$.

Expanding this, we have $(1/2)x_k^T Hx_k - x_k^T Hx^* + (1/2)x^{*T} Hx^*$.

Since $x^* = -H^{-1}b$, we can write this as $(1/2)x_k^T Hx_k + x_k^T b + (1/2)x^{*T} Hx^*$.

Adding and subtracting c, this is just $f(x_k)$ plus some constant.

Note that this constant does not depend on x_k.

So the error function is nothing but $f(x_k)$ plus some constant value.

We will understand why this function is needed in some time, but note that $E(x^*) = 0$.

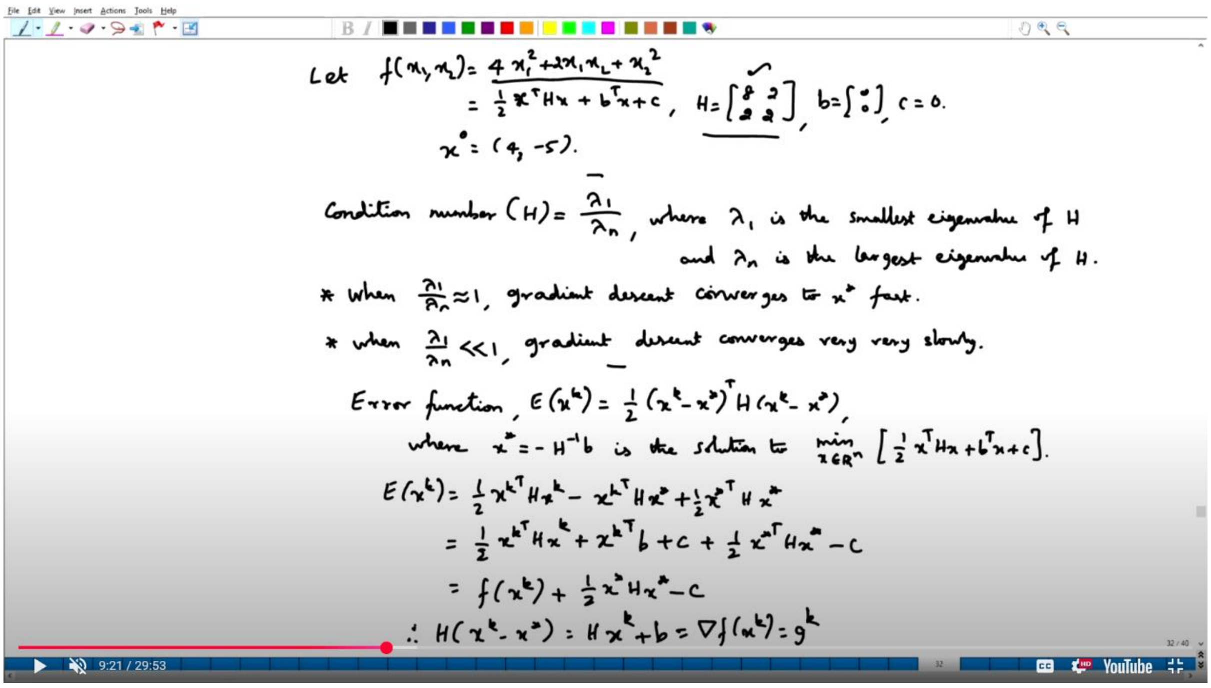
$f(x^*)$ need not be 0 if b and c are non-zero, but $E(x^*)$ is certainly 0 because it is $(1/2)*0^T H*0$.

Another observation is that $H(x_k - x^*) = Hx_k - Hx^*$.

But $Hx^* = -b$, so $Hx_k - (-b) = Hx_k + b = \nabla f(x_k) = g_k$.

So, $H(x_k - x^*) = g_k$. This is another property we will use in our proof.

(Refer Slide Time 9:25)



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

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

$x^0 = (4, -5)$.

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.

Error function, $E(x^k) = \frac{1}{2}(x^k - x^*)^T H (x^k - x^*)$,

where $x^* = -H^{-1}b$ is the solution to $\min_{x \in \mathbb{R}^n} \left[\frac{1}{2}x^T H x + b^T x + c\right]$.

$$E(x^k) = \frac{1}{2}x^{k^T} H x^k - x^{k^T} H x^* + \frac{1}{2}x^{*^T} H x^*$$

$$= \frac{1}{2}x^{k^T} H x^k + x^{k^T} b + c + \frac{1}{2}x^{*^T} H x^* - c$$

$$= f(x^k) + \frac{1}{2}x^* H x^* - c$$

$$\therefore H(x^k - x^*) = Hx^k + b = \nabla f(x^k) = g^k$$

We will now compute the quantity $[E(x_k) - E(x_{k+1})] / E(x_k)$.

So, $E(x_k) - E(x_{k+1}) = (1/2)(x_k - x^*)^T H(x_k - x^*) - (1/2)(x_{k+1} - x^*)^T H(x_{k+1} - x^*)$.

In the gradient descent algorithm, $x_{k+1} = x_k - \alpha_k g_k$.

Substituting that, we have:

$= (1/2)(x_k - x^*)^T H(x_k - x^*) - (1/2)(x_k - \alpha_k g_k - x^*)^T H(x_k - \alpha_k g_k - x^*)$

$= (1/2)[\ (x_k - x^*)^T H(x_k - x^*) - (x_k - x^* - \alpha_k g_k)^T H(x_k - x^* - \alpha_k g_k)\ ]$

Expanding the second term:

$= (x_k - x^*)^T H(x_k - x^*) - 2\alpha_k g_k^T H(x_k - x^*) + \alpha_k^2 g_k^T H g_k$

So, the difference becomes:

$(1/2)[\ 2\alpha_k g_k^T H(x_k - x^*) - \alpha_k^2 g_k^T H g_k\ ] = \alpha_k g_k^T H(x_k - x^*) - (\alpha_k^2/2) g_k^T H g_k$

But $H(x_k - x^*) = g_k$, so this becomes $\alpha_k g_k^T g_k - (\alpha_k^2/2) g_k^T H g_k$.

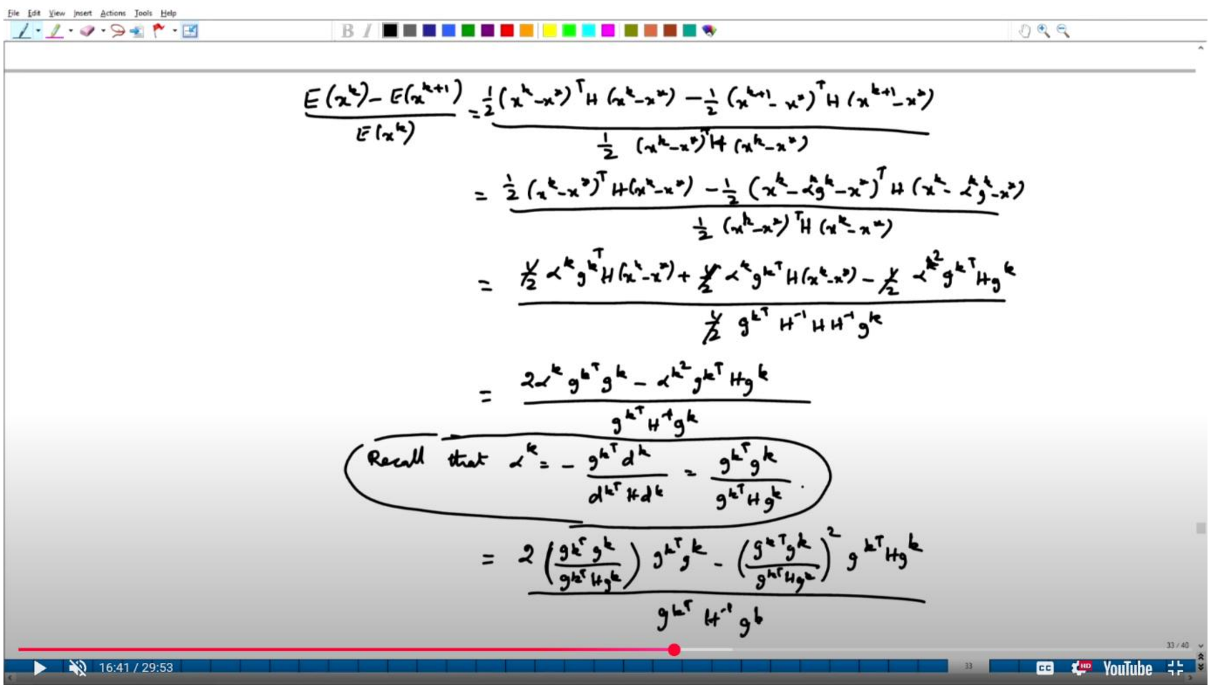
The denominator is $E(x_k) = (1/2)(x_k - x^*)^T H(x_k - x^*) = (1/2) g_k^T H^{-1} g_k$ (since $x_k - x^* = H^{-1} g_k$).

So, $[E(x_k) - E(x_{k+1})] / E(x_k) = [\ \alpha_k g_k^T g_k - (\alpha_k^2/2) g_k^T H g_k\ ] / [\ (1/2) g_k^T H^{-1} g_k\ ] = [\ 2\alpha_k g_k^T g_k - \alpha_k^2 g_k^T H g_k\ ] / [\ g_k^T H^{-1} g_k\ ]$

Now, recall that in exact line search for quadratic functions, $\alpha_k = (g_k^T g_k) / (g_k^T H g_k)$. Substituting this:

$= [\ 2^*(g_k^T g_k)^2/(g_k^T H g_k) - (g_k^T g_k)^2/(g_k^T H g_k)\ ] / [\ g_k^T H^{-1} g_k\ ]$

(Refer Slide Time 16:41)



$$\frac{E(x^k)-E(x^{k+1})}{E(x^k)} = \frac{\frac{1}{2}(x^k-x^*)^T H (x^k-x^*) - \frac{1}{2}(x^{k+1}-x^*)^T H (x^{k+1}-x^*)}{\frac{1}{2}(x^k-x^*)^T H (x^k-x^*)}$$

$$= \frac{\frac{1}{2}(x^k-x^*)^T H (x^k-x^*) - \frac{1}{2}(x^k-\alpha^k g^k-x^*)^T H (x^k-\alpha^k g^k-x^*)}{\frac{1}{2}(x^k-x^*)^T H (x^k-x^*)}$$

$$= \frac{\cancel{\tfrac{1}{2}}\alpha^k g^{kT} H(x^k-x^*) + \cancel{\tfrac{1}{2}}\alpha^k g^{kT} H(x^k-x^*) - \cancel{\tfrac{1}{2}}\alpha^{k^2} g^{kT} H g^k}{\cancel{\tfrac{1}{2}}\, g^{kT} H^{-1} H H^{-1} g^k}$$

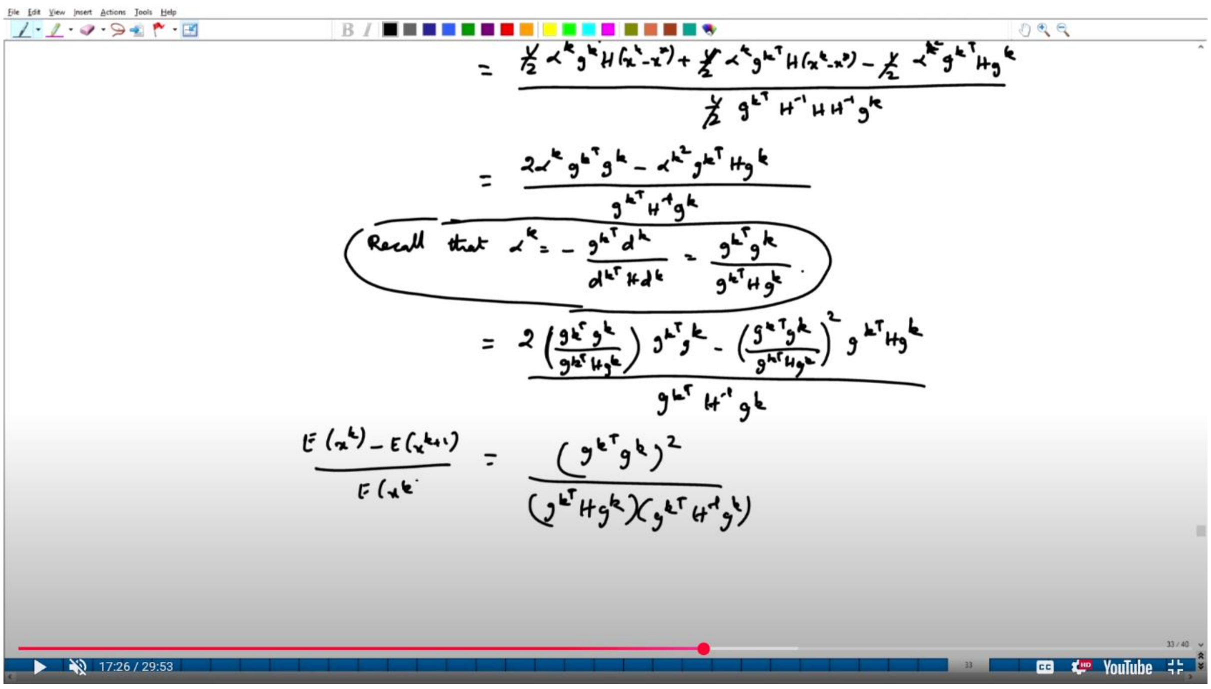
$$= \frac{2\alpha^k g^{kT} g^k - \alpha^{k^2} g^{kT} H g^k}{g^{kT} H^{-1} g^k}$$

Recall that $\alpha^k = -\frac{g^{kT} d^k}{d^{kT} H d^k} = \frac{g^{kT} g^k}{g^{kT} H g^k}$.

$$= \frac{2\left(\frac{g^{kT} g^k}{g^{kT} H g^k}\right) g^{kT} g^k - \left(\frac{g^{kT} g^k}{g^{kT} H g^k}\right)^2 g^{kT} H g^k}{g^{kT} H^{-1} g^k}$$

= [$(g_k^T g_k)^2 / (g_k^T H g_k)$] / [$g_k^T H^{-1} g_k$] = $(g_k^T g_k)^2$ / [$(g_k^T H g_k)(g_k^T H^{-1} g_k)$]

(Refer Slide Time 17:26)



$$= \frac{\cancel{\tfrac{1}{2}}\alpha^k g^{kT} H(x^k-x^*) + \cancel{\tfrac{1}{2}}\alpha^k g^{kT} H(x^k-x^*) - \cancel{\tfrac{1}{2}}\alpha^{k^2} g^{kT} H g^k}{\cancel{\tfrac{1}{2}}\, g^{kT} H^{-1} H H^{-1} g^k}$$

$$= \frac{2\alpha^k g^{kT} g^k - \alpha^{k^2} g^{kT} H g^k}{g^{kT} H^{-1} g^k}$$

Recall that $\alpha^k = -\frac{g^{kT} d^k}{d^{kT} H d^k} = \frac{g^{kT} g^k}{g^{kT} H g^k}$.

$$= \frac{2\left(\frac{g^{kT} g^k}{g^{kT} H g^k}\right) g^{kT} g^k - \left(\frac{g^{kT} g^k}{g^{kT} H g^k}\right)^2 g^{kT} H g^k}{g^{kT} H^{-1} g^k}$$

$$\frac{E(x^k)-E(x^{k+1})}{E(x^k)} = \frac{(g^{kT} g^k)^2}{(g^{kT} H g^k)(g^{kT} H^{-1} g^k)}$$

Now, we use the Kantorovich inequality.

For any positive definite matrix H and vector $x \neq 0$, we have:

$(x^T x)^2$ / [$(x^T H x)(x^T H^{-1} x)$] $\geq 4\lambda_1\lambda_n / (\lambda_1 + \lambda_n)^2$

where λ_1 and λ_n are the smallest and largest eigenvalues of H.

Applying this to $x = g_k$, we get:

$[E(x_k) - E(x_{k+1})] / E(x_k) \geq 4\lambda_1\lambda_n / (\lambda_1 + \lambda_n)^2$

Therefore, $E(x_{k+1}) / E(x_k) \leq 1 - 4\lambda_1\lambda_n / (\lambda_1 + \lambda_n)^2 = [(\lambda_1 + \lambda_n)^2 - 4\lambda_1\lambda_n] / (\lambda_1 + \lambda_n)^2 = (\lambda_n - \lambda_1)^2 / (\lambda_1 + \lambda_n)^2$

So, $E(x_{k+1}) \leq [(\lambda_n - \lambda_1) / (\lambda_1 + \lambda_n)]^2 E(x_k)$

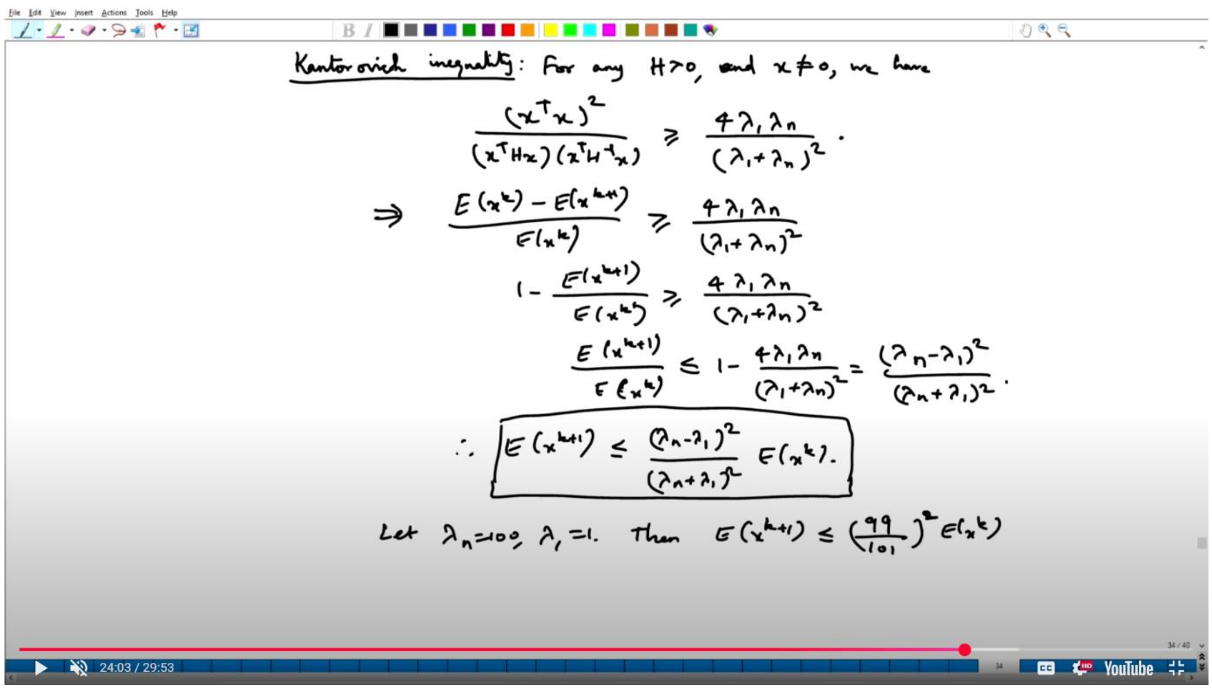
This tells us that the error function decreases by a factor of $[(\lambda_n - \lambda_1) / (\lambda_1 + \lambda_n)]^2$ each iteration. We want the error function to go to 0. The quicker x_k gets nearer to x^*, the better.

If $\lambda_n = \lambda_1$, meaning all eigenvalues are equal, then $E(x_{k+1}) \leq 0$, so $E(x_1) = 0$. We reach the solution in one step.

But if λ_n is much larger than λ_1, for example, $\lambda_n = 100$ and $\lambda_1 = 1$, then $E(x_{k+1}) \leq (99/101)^2 E(x_k) \approx 0.9606 E(x_k)$.

The error decreases very slowly each iteration.

(Refer Slide Time 24:03)



Kantorovich inequality: For any $H > 0$, and $x \neq 0$, we have

$$\frac{(x^T x)^2}{(x^T H x)(x^T H^{-1} x)} \geq \frac{4\lambda_1\lambda_n}{(\lambda_1+\lambda_n)^2}.$$

$$\Rightarrow \frac{E(x^k) - E(x^{k+1})}{E(x^k)} \geq \frac{4\lambda_1\lambda_n}{(\lambda_1+\lambda_n)^2}$$

$$1 - \frac{E(x^{k+1})}{E(x^k)} \geq \frac{4\lambda_1\lambda_n}{(\lambda_1+\lambda_n)^2}$$

$$\frac{E(x^{k+1})}{E(x^k)} \leq 1 - \frac{4\lambda_1\lambda_n}{(\lambda_1+\lambda_n)^2} = \frac{(\lambda_n-\lambda_1)^2}{(\lambda_n+\lambda_1)^2}.$$

$$\therefore \boxed{E(x^{k+1}) \leq \frac{(\lambda_n-\lambda_1)^2}{(\lambda_n+\lambda_1)^2} E(x^k).}$$

Let $\lambda_n = 100$, $\lambda_1 = 1$. Then $E(x^{k+1}) \leq \left(\frac{99}{101}\right)^2 E(x^k)$

This is the mathematical proof of why the condition number λ_1/λ_n affects the convergence speed. When λ_1/λ_n is close to 1, gradient descent converges very fast. When λ_1/λ_n is much less than 1, convergence is very slow.

I will illustrate this with code. For $x_1^2 + x_2^2$, $\lambda_1/\lambda_n = 1$, and it converges in one step.

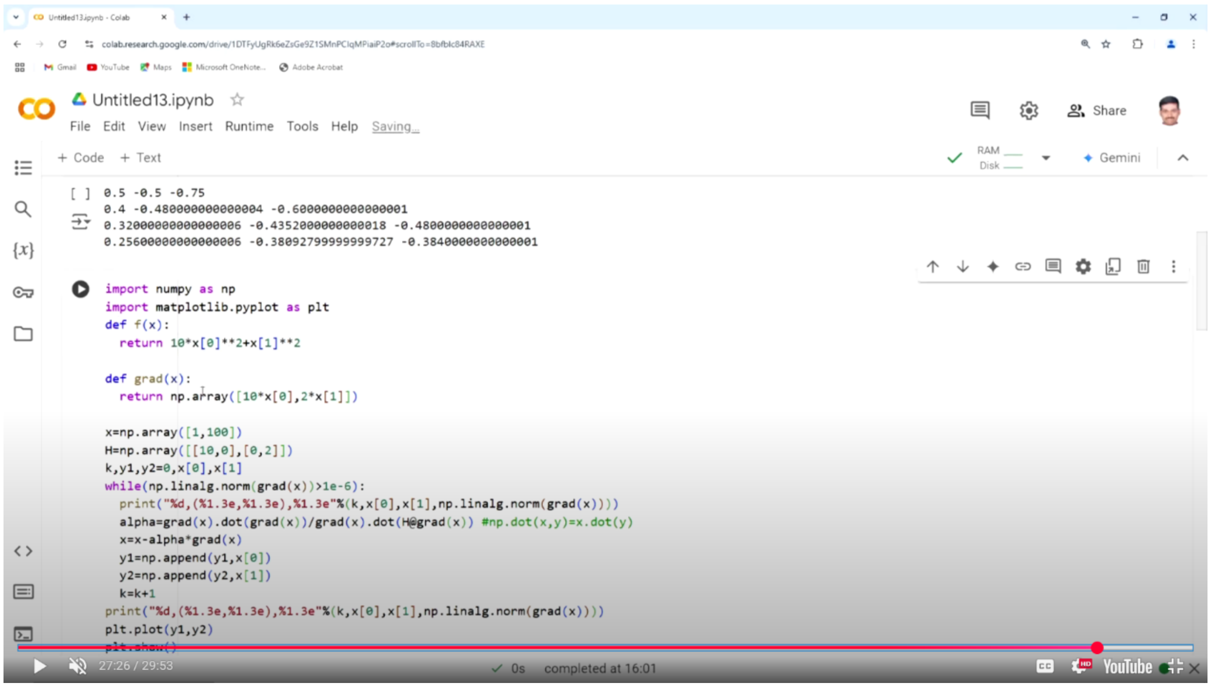
If we change the function to $2x_1^2 + x_2^2$, the condition number changes, and it takes more steps. For example, with initial point (1, 100):

- For $2x_1^2 + x_2^2$, it takes about 5 steps.
- For $5x_1^2 + x_2^2$, it takes about 8 steps.
- For $10x_1^2 + x_2^2$, it takes about 16 steps.
- For $25x_1^2 + x_2^2$, it takes about 68 steps.
- For $50x_1^2 + x_2^2$, it takes about 316 steps.
- For $100x_1^2 + x_2^2$, it takes about 973 steps.

You can see the zigzagging path the algorithm takes from (1, 100) to (0, 0).

The function is just $100x_1^2 + x_2^2$, but the algorithm takes 973 steps because $\lambda_1/\lambda_n = 1/100$ is very small, making convergence very slow.

(Refer Slide Time 27:30-29:00)

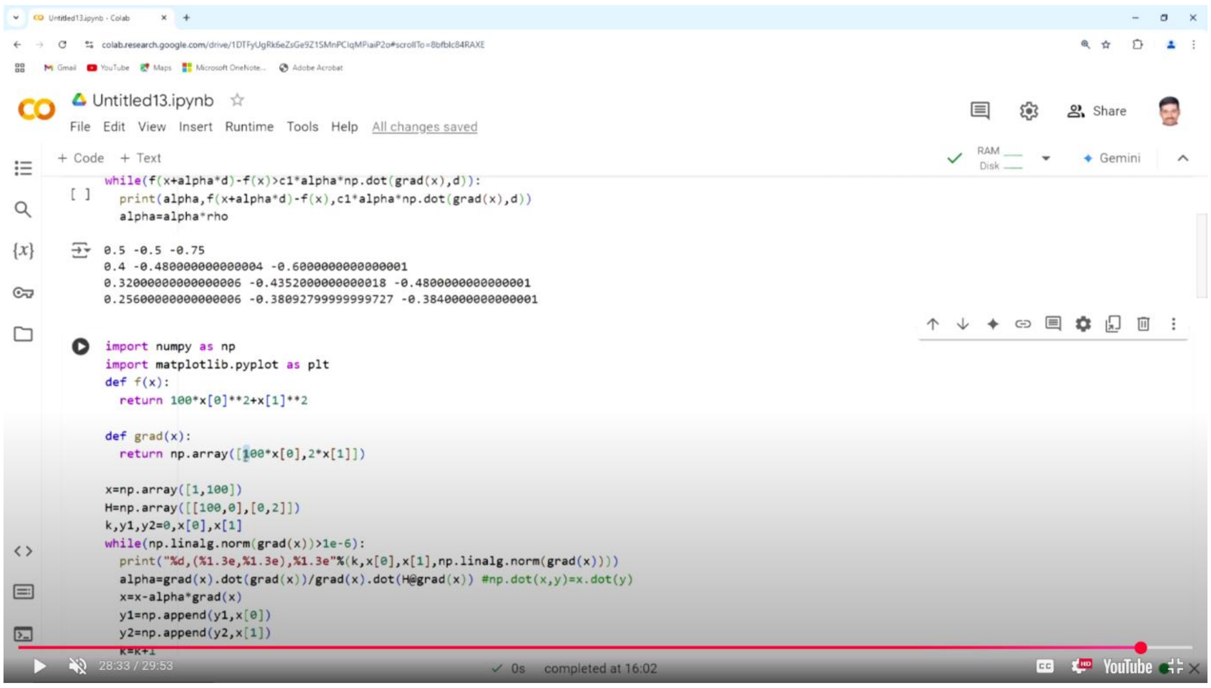


```
0.5 -0.5 -0.75
0.4 -0.480000000000004 -0.6000000000000001
0.32000000000000006 -0.4352000000000018 -0.4800000000000001
0.25600000000000006 -0.38092799999999727 -0.3840000000000001
```

```python
import numpy as np
import matplotlib.pyplot as plt
def f(x):
  return 10*x[0]**2+x[1]**2

def grad(x):
  return np.array([10*x[0],2*x[1]])

x=np.array([1,100])
H=np.array([[10,0],[0,2]])
k,y1,y2=0,x[0],x[1]
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=grad(x).dot(grad(x))/grad(x).dot(H@grad(x)) #np.dot(x,y)=x.dot(y)
  x=x-alpha*grad(x)
  y1=np.append(y1,x[0])
  y2=np.append(y2,x[1])
  k=k+1
print("%d,(%1.3e,%1.3e),%1.3e"%(k,x[0],x[1],np.linalg.norm(grad(x))))
plt.plot(y1,y2)
plt.show()
```



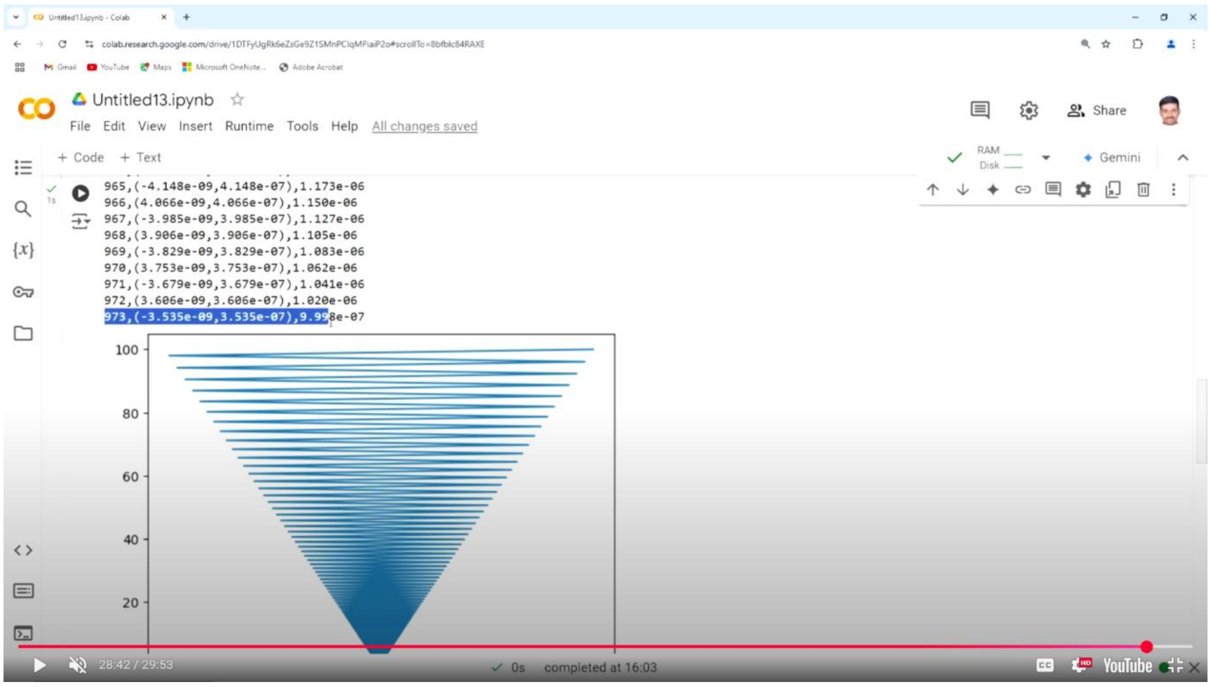
```
while(f(x+alpha*d)-f(x)>c1*alpha*np.dot(grad(x),d)):
  print(alpha,f(x+alpha*d)-f(x),c1*alpha*np.dot(grad(x),d))
  alpha=alpha*rho
```

```
0.5 -0.5 -0.75
0.4 -0.480000000000004 -0.6000000000000001
0.32000000000000006 -0.4352000000000018 -0.4800000000000001
0.25600000000000006 -0.38092799999999727 -0.3840000000000001
```

```
import numpy as np
import matplotlib.pyplot as plt
def f(x):
  return 100*x[0]**2+x[1]**2

def grad(x):
  return np.array([100*x[0],2*x[1]])

x=np.array([1,100])
H=np.array([[100,0],[0,2]])
k,y1,y2=0,x[0],x[1]
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=grad(x).dot(grad(x))/grad(x).dot(H@grad(x)) #np.dot(x,y)=x.dot(y)
  x=x-alpha*grad(x)
  y1=np.append(y1,x[0])
  y2=np.append(y2,x[1])
  k=k+1
```



```
965,(-4.148e-09,4.148e-07),1.173e-06
966,(4.066e-09,4.066e-07),1.150e-06
967,(-3.985e-09,3.985e-07),1.127e-06
968,(3.906e-09,3.906e-07),1.105e-06
969,(-3.829e-09,3.829e-07),1.083e-06
970,(3.753e-09,3.753e-07),1.062e-06
971,(-3.679e-09,3.679e-07),1.041e-06
972,(3.606e-09,3.606e-07),1.020e-06
973,(-3.535e-09,3.535e-07),9.998e-07
```

We will continue the discussion on this algorithm in the next lecture. Thank you.