Optimization Algorithms: Theory and Software Implementation

Prof. Thirumulanathan D

Department of Mathematics

Institute of IIT Kanpur

Lecture: 28

Hello everyone. This is the third lecture of week 6. Recall that in the previous two lectures, we started with the basics of quasi-Newton methods. We derived the quasi-Newton condition, $B_{k+1}\gamma_k = \delta_k$. We then looked at an algorithm called rank-one correction, which updates B_k to B_{k+1} by adding a rank-one matrix.

In the previous lecture, we showed that if we are minimizing a quadratic function, this particular algorithm reaches H^{-1} in n steps, meaning B_n becomes H^{-1}. You can start with any positive definite matrix B_0.

We then applied the method to a non-quadratic function, starting from the point (1, 1). We attained the minimum (0, 0) in 31 steps. I also gave a brief illustration of the `np.newaxis` (or `None`) operation in Python, which is necessary to correctly compute outer products for the matrix update.

Now, let's recall an issue from previous weeks. If you start from the point $(-\sqrt{2}, -\sqrt{2})$, Newton's method converged to a saddle point at (-2, -2), while the Fletcher-Reeves conjugate gradient method reached the actual answer (0, 0). Let us see what happens with the rank-one correction quasi-Newton method.

You can see that this quasi-Newton method suffers from the same problem. It converges to the saddle point (-2, -2).

Recall the two variants we discussed for Newton's method: damped Newton's method and modified Newton's method. Damped Newton's method addressed the global convergence issue, but even it faced this problem of converging to a saddle point when starting from $(-\sqrt{2}, -\sqrt{2})$. This problem only disappeared with modified Newton's method, which uses eigenvalues to modify the Hessian.

However, finding eigenvalues and eigenvectors is computationally expensive, often as expensive as inverting a matrix. Since the purpose of quasi-Newton methods is to avoid such expensive operations, using eigenvalues defeats the purpose. Therefore, this issue of converging to critical points that are not minimizers is not rectified in these basic quasi-Newton methods. You have to bear with this problem.

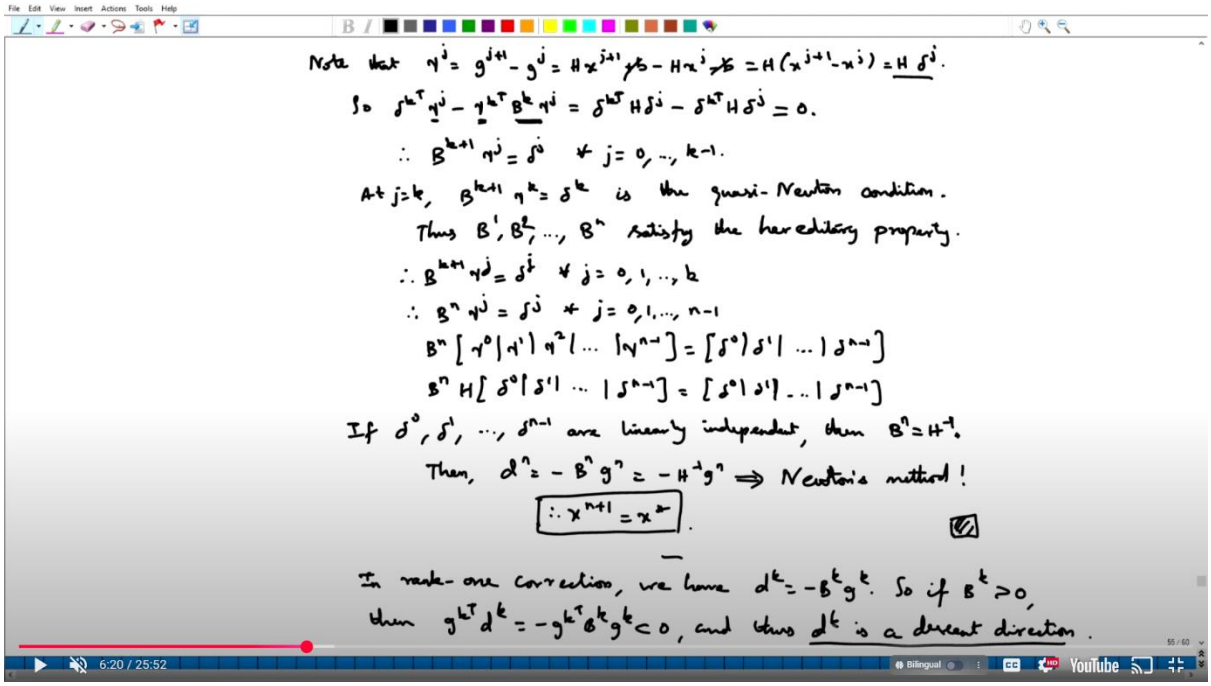
Of course, if the initial point is sufficiently close to the minimizer, like (-0.5, -0.5), the method still converges to (0, 0) in about 25 steps.

Another important issue is the positive definiteness of B_k.

So, what we had in Newton's method, right. We were checking if whether H, the resultant H_k that we get, is positive definite or not. If it is not positive definite then $-H_k^{-1}d_k$ need not be a descent direction. So, we will face that kind of a problem in rank one correction as well. If B_k does not turn out to be a positive definite matrix, then $-B_kg_k$ need not be a descent direction, so this is a proof that we had already done but coming back to the issue of B_k being positive definite specifically for non-quadratic functions, let us actually ask this question. In rank one correction we have $d_k = -B_kg_k$, if B_k were positive definite then

$g_k^Td_k=-g_k^TB_kg_k < 0$ and thus d_k is a descent direction.

(Refer Slide Time 6:20)



Unfortunately, if B_k is not positive definite, then $d_k = -B_kg_k$ may not be a descent direction. So, we might end up with $f(x_{k+1}) - f(x_k) > 0$. This is what happened in the example starting from $(-\sqrt{2}, -\sqrt{2})$; $-B_kg_k$ was not a descent direction.

Let's examine the update step: $B_{k+1} = B_k + [\ (\delta_k - B_k\gamma_k)(\delta_k - B_k\gamma_k)^T\ ] / [\ (\delta_k - B_k\gamma_k)^T\gamma_k\ ]$.

Suppose B_k is positive definite. The numerator, $(\delta_k - B_k\gamma_k)(\delta_k - B_k\gamma_k)^T$, is a positive semi-definite matrix. If this were the only term, B_{k+1} would be positive definite. The issue comes from the denominator, $(\delta_k - B_k\gamma_k)^T\gamma_k$.

It is possible for $(\delta_k - B_k\gamma_k)^T\gamma_k$ to be negative. If this happens, the entire term we are adding becomes negative semi-definite. Adding this to the positive definite B_k could result in a matrix that is negative definite or indefinite.

Another issue is if $(\delta_k - B_k\gamma_k)^T\gamma_k$ is close to zero, or zero itself. In this case, the update term blows up, and the model collapses.

These are real possibilities when minimizing non-quadratic functions with rank-one correction, which is why we cannot always use it as-is.

We will now consider the next method, called the DFP method. DFP stands for Davidon–Fletcher–Powell. Fletcher is a name you have heard from the Fletcher-Reeves method. Fletcher is also part of the next method we will discuss, BFGS (Broyden–Fletcher–Goldfarb–Shanno). This is more of a trivia, but note that Fletcher is involved in many of these algorithms.

The DFP method is a rank-two correction method. While rank-one correction added one rank-one matrix, DFP adds two such matrices. The update step in the DFP method is:

$B_{k+1} = B_k + \beta_1 u u^T + \beta_2 v v^T$

Here, u and v are vectors, and β_1 and β_2 are constants.

We must satisfy the quasi-Newton condition: $B_{k+1}\gamma_k = \delta_k$.

Substituting the update gives:

$B_k\gamma_k + \beta_1 (u^T\gamma_k) u + \beta_2 (v^T\gamma_k) v = \delta_k$

This can be rearranged as:

$\beta_1 (u^T\gamma_k) u + \beta_2 (v^T\gamma_k) v = \delta_k - B_k\gamma_k$

A standard choice is to set:

$u = \delta_k$

$v = -B_k\gamma_k$

To satisfy the equation, we then choose the constants so that:

$\beta_1 (u^T\gamma_k) = 1 \Rightarrow \beta_1 = 1 / (\delta_k^T\gamma_k)$

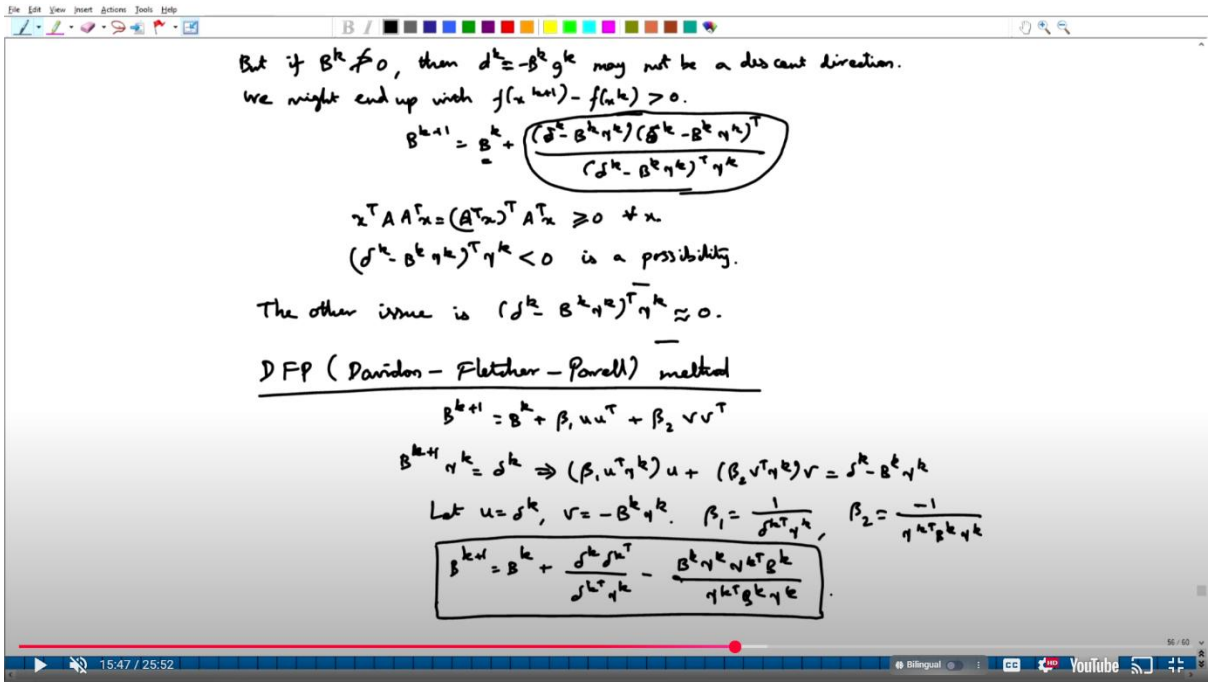
$\beta_2 (v^T\gamma_k) = 1 \Rightarrow \beta_2 = 1 / (\gamma_k^T B_k \gamma_k)$ (Note: $v^T\gamma_k = -\gamma_k^T B_k\gamma_k$, so the negative sign is absorbed)

Substituting these choices for u, v, β_1, and β_2 back into the update formula gives the final DFP update:

$B_{k+1} = B_k + (\delta_k \delta_k^T) / (\delta_k^T\gamma_k) - (B_k \gamma_k \gamma_k^T B_k) / (\gamma_k^T B_k \gamma_k)$

The algorithm is exactly the same as for rank-one correction, except for this update step for B_{k+1}.

(Refer Slide Time 16:00)



So, similarly here, you have a minus sign because the term vv^T is added and β_2 is negative, so the resultant is subtracted. You have $B_k\gamma_k$, and the transpose of that is $\gamma_k^T B_k^T$. But since B_k is symmetric, I will just write it as B_k, and the term becomes $(B_k\gamma_k)(B_k\gamma_k)^T / (\gamma_k^T B_k \gamma_k)$. So that is the update step.

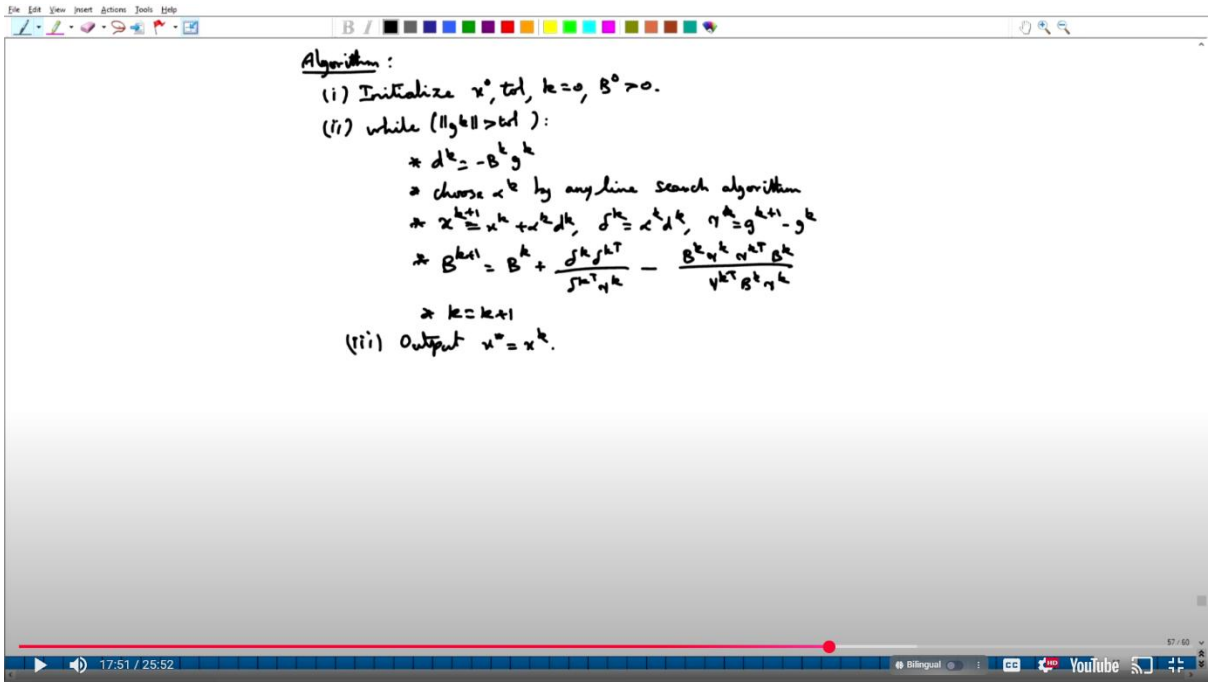
The method is conceptually straightforward. If you want the algorithm written out, it is exactly the same as the one for the rank-one correction, except for the update step for B_{k+1}. I will write it down clearly.

You initialize everything as in the rank-one correction method. Then, while the norm of the gradient, $||g_k||$, is greater than a specified tolerance, you perform the following steps:

1. Compute the search direction: $d_k = -B_k\, g_k$
2. Choose a step size α_k using a line search algorithm.
3. Update the iterate: $x_{k+1} = x_k + \alpha_k\, d_k$
4. Compute the change in position: $\delta_k = \alpha_k\, d_k$
5. Compute the change in gradient: $\gamma_k = g_{k+1} - g_k$
6. Update the Hessian approximation using the DFP formula: $B_{k+1} = B_k + (\delta_k\, \delta_k^T) / (\delta_k^T\, \gamma_k) - (B_k\, \gamma_k\, \gamma_k^T\, B_k) / (\gamma_k^T\, B_k\, \gamma_k)$
7. Set $k = k + 1$

After the loop terminates, the output is the solution $x^* = x_k$.

(Refer Slide Time 17:51)



Since only the update step for B_k is changing from the previous algorithm, applying this to examples will be very similar. The core of the work is in modifying the B update according to this new formula, while everything else remains the same.

Let's implement this for a quadratic function. We initialize everything as before: a random positive definite H, a random vector b, a random starting point x_0, and B_0 as the identity matrix. The only change is the B-update step, which now uses the DFP formula.

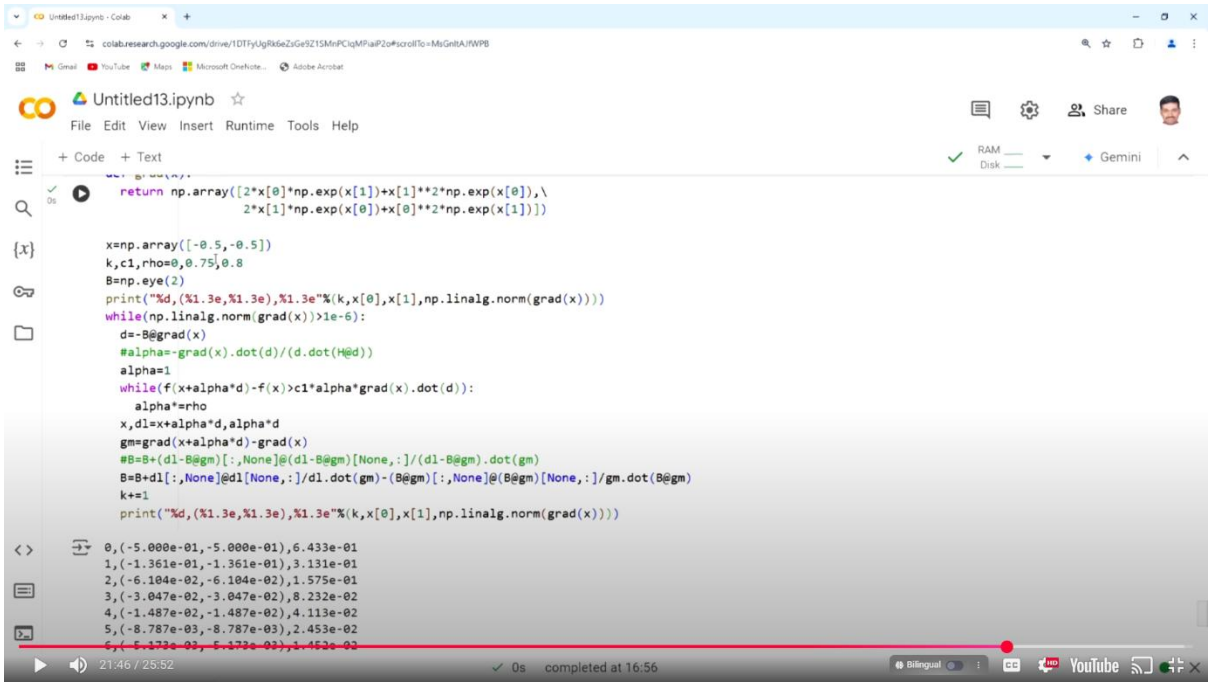
For the quadratic function, the DFP method also converges to the solution in n=5 steps, and the final B matrix equals H^{-1}.

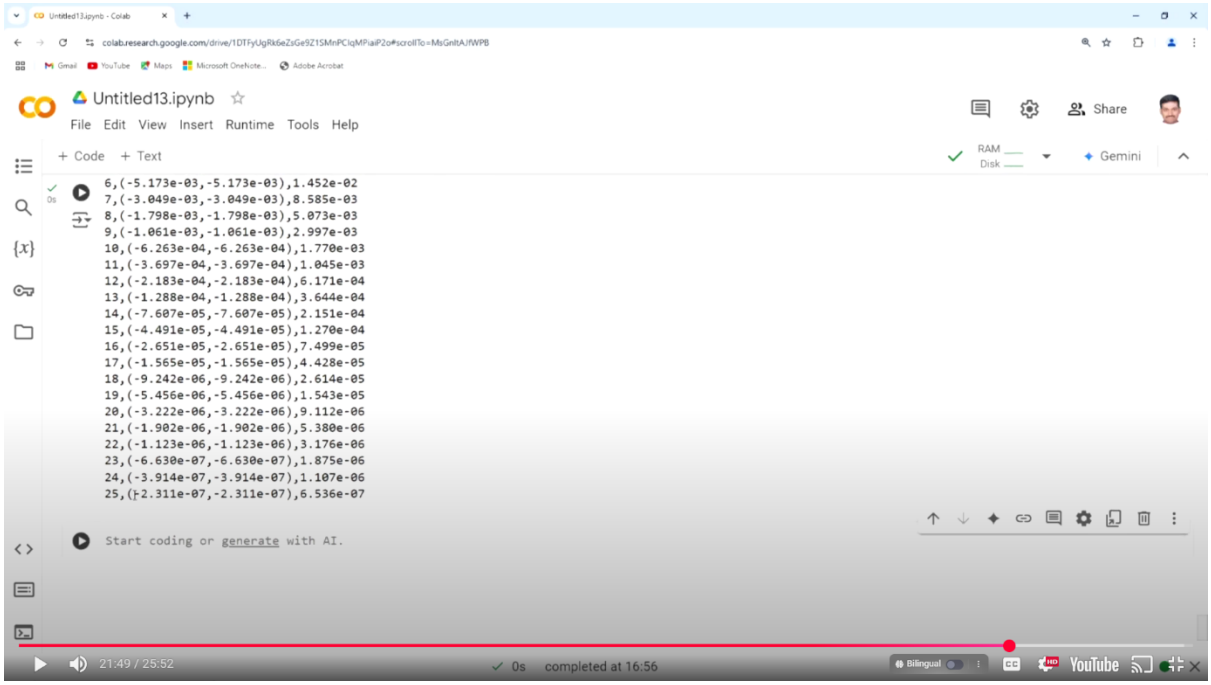
Now, let's test it on the general function

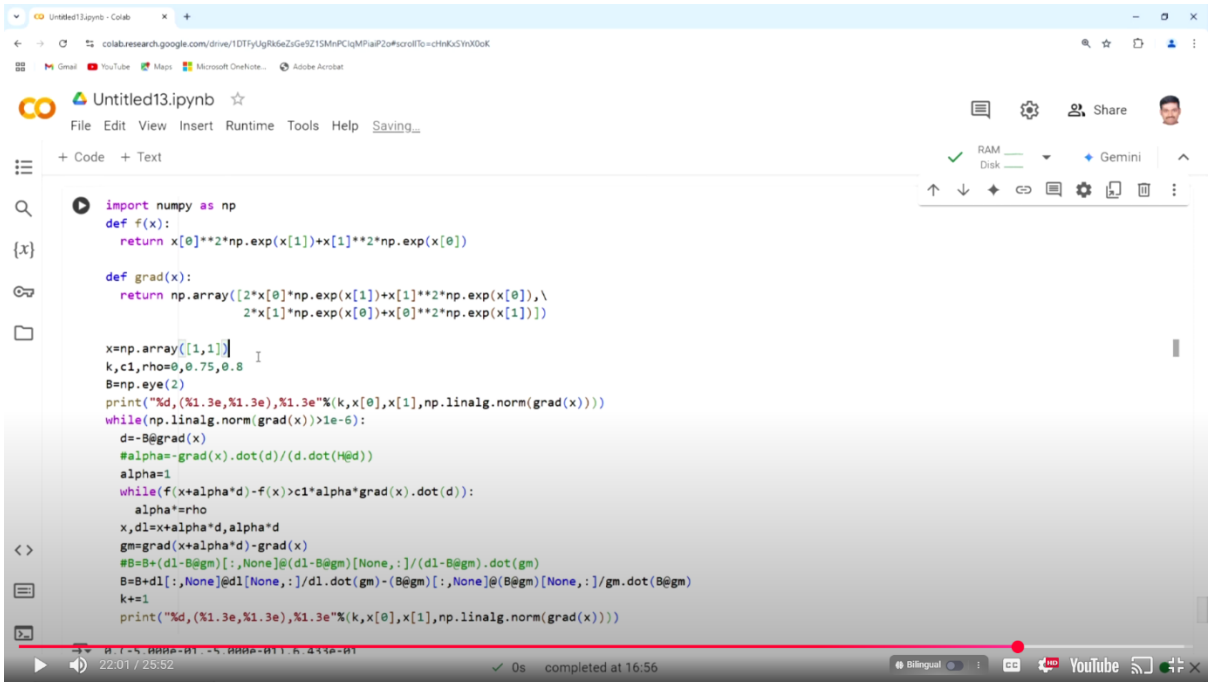
$f(x) = x_1^2 e^{x_2} + x_2^2 e^{x_1}$. Starting from (1, 1) or (-0.5, -0.5), the method converges to (0, 0) in a similar number of steps (e.g., 31).

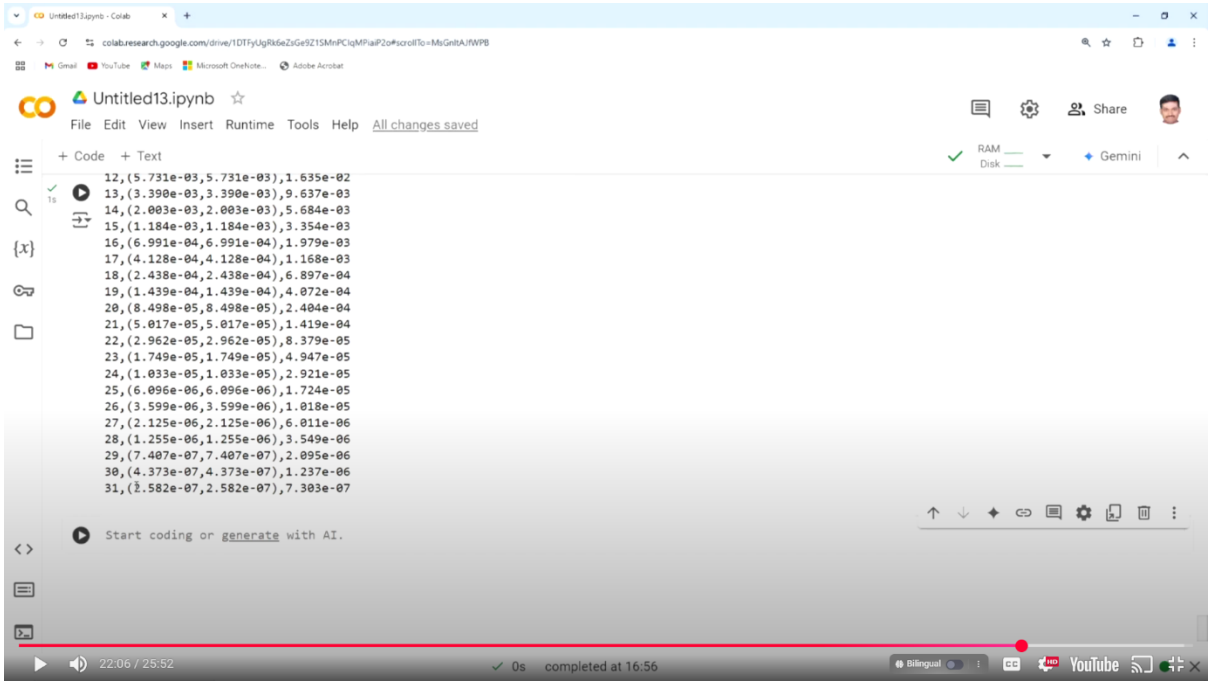
Now, let's return to the problematic starting point $(-\sqrt{2}, -\sqrt{2})$. The DFP method, like the others, converges to the saddle point (-2, -2). The performance, in this case, has not changed.

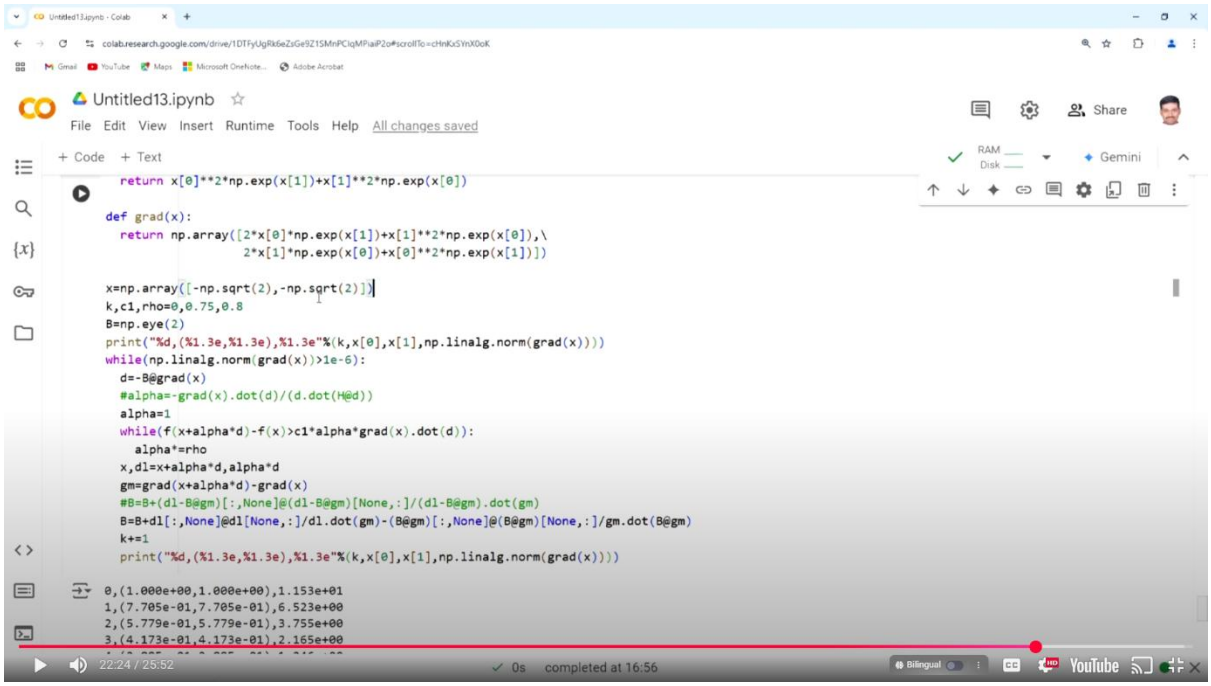
(Refer Slide Time 24:00)

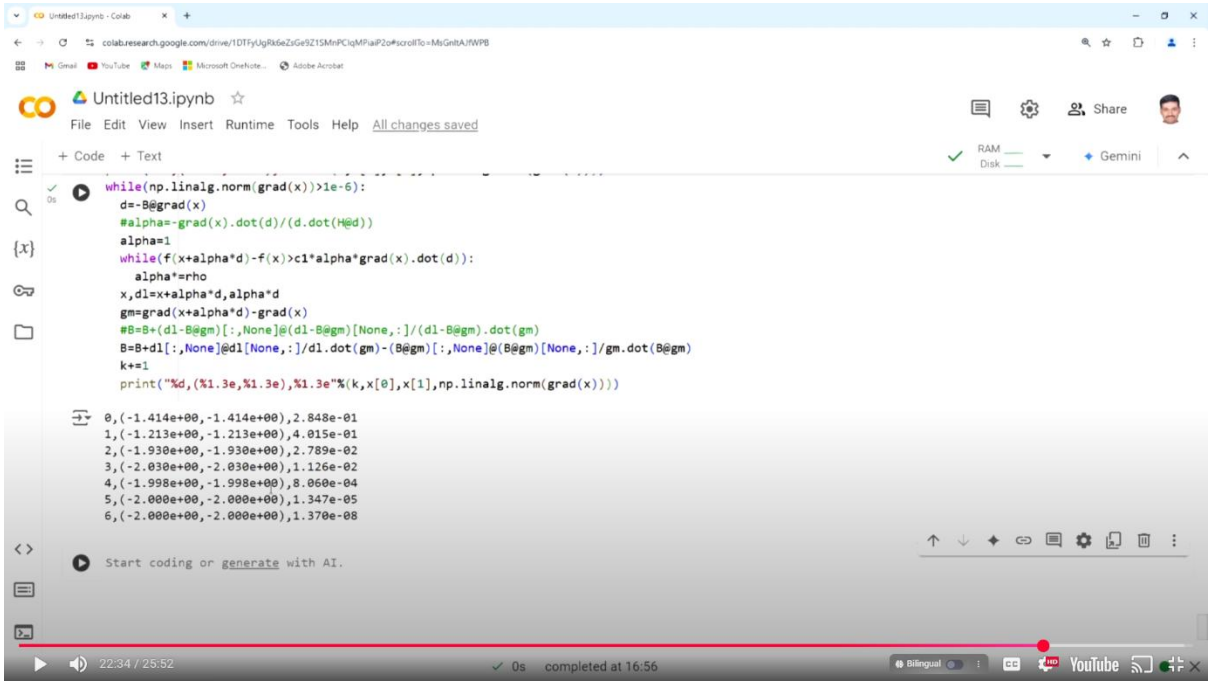

Untitled13.ipynb
File Edit View Insert Runtime Tools Help
Share
+ Code + Text
RAM
Disk
Gemini
return np.array([2*x[0]*np.exp(x[1])+x[1]**2*np.exp(x[0]),\
2*x[1]*np.exp(x[0])+x[0]**2*np.exp(x[1])])
x=np.array([-0.5,-0.5])
k,c1,rho=0,0.75,0.8
B=np.eye(2)
print("%d,(%1.3e,%1.3e),%1.3e"%(k,x[0],x[1],np.linalg.norm(grad(x))))
while(np.linalg.norm(grad(x))>1e-6):
d=-B@grad(x)
#alpha=-grad(x).dot(d)/(d.dot(H@d))
alpha=1
while(f(x+alpha*d)-f(x)>c1*alpha*grad(x).dot(d)):
alpha*=rho
x,dl=x+alpha*d,alpha*d
gm=grad(x+alpha*d)-grad(x)
#B=B+(dl-B@gm)[:,None]@(dl-B@gm)[None,:]/(dl-B@gm).dot(gm)
B=B+dl[:,None]@dl[None,:]/dl.dot(gm)-(B@gm)[:,None]@(B@gm)[None,:]/gm.dot(B@gm)
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),6.433e-01
1,(-1.361e-01,-1.361e-01),3.131e-01
2,(-6.104e-02,-6.104e-02),1.575e-01
3,(-3.047e-02,-3.047e-02),8.232e-02
4,(-1.487e-02,-1.487e-02),4.113e-02
5,(-8.787e-03,-8.787e-03),2.453e-02
21:46 / 25:52
0s completed at 16:56
Bilingual
YouTube


Untitled13.ipynb
File Edit View Insert Runtime Tools Help
Share
+ Code + Text
RAM
Disk
Gemini
6,(-5.173e-03,-5.173e-03),1.452e-02
7,(-3.049e-03,-3.049e-03),8.585e-03
8,(-1.798e-03,-1.798e-03),5.073e-03
9,(-1.061e-03,-1.061e-03),2.997e-03
10,(-6.263e-04,-6.263e-04),1.770e-03
11,(-3.697e-04,-3.697e-04),1.045e-03
12,(-2.183e-04,-2.183e-04),6.171e-04
13,(-1.288e-04,-1.288e-04),3.644e-04
14,(-7.607e-05,-7.607e-05),2.151e-04
15,(-4.491e-05,-4.491e-05),1.270e-04
16,(-2.651e-05,-2.651e-05),7.499e-05
17,(-1.565e-05,-1.565e-05),4.428e-05
18,(-9.242e-06,-9.242e-06),2.614e-05
19,(-5.456e-06,-5.456e-06),1.543e-05
20,(-3.222e-06,-3.222e-06),9.112e-06
21,(-1.902e-06,-1.902e-06),5.380e-06
22,(-1.123e-06,-1.123e-06),3.176e-06
23,(-6.630e-07,-6.630e-07),1.875e-06
24,(-3.914e-07,-3.914e-07),1.107e-06
25,(-2.311e-07,-2.311e-07),6.536e-07
Start coding or generate with AI.
21:49 / 25:52
0s completed at 16:56
Bilingual
YouTube


Untitled13.ipynb
File Edit View Insert Runtime Tools Help Saving...
Share
+ Code + Text
RAM Disk
Gemini
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])
k,c1,rho=0,0.75,0.8
B=np.eye(2)
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):
d=-B@grad(x)
#alpha=-grad(x).dot(d)/(d.dot(H@d))
alpha=1
while(f(x+alpha*d)-f(x)>c1*alpha*grad(x).dot(d)):
alpha*=rho
x,dl=x+alpha*d,alpha*d
gm=grad(x+alpha*d)-grad(x)
#B=B+(dl-B@gm)[:,None]@(dl-B@gm)[None,:]/(dl-B@gm).dot(gm)
B=B+dl[:,None]@dl[None,:]/dl.dot(gm)-(B@gm)[:,None]@(B@gm)[None,:]/gm.dot(B@gm)
k+=1
print("%d,(%1.3e,%1.3e),%1.3e"%(k,x[0],x[1],np.linalg.norm(grad(x))))
22:01 / 25:52
0s completed at 16:56
Bilingual
YouTube


Untitled13.ipynb
File Edit View Insert Runtime Tools Help All changes saved
Share
+ Code + Text
RAM Disk
Gemini
12,(5.731e-03,5.731e-03),1.635e-02
13,(3.390e-03,3.390e-03),9.637e-03
14,(2.003e-03,2.003e-03),5.684e-03
15,(1.184e-03,1.184e-03),3.354e-03
16,(6.991e-04,6.991e-04),1.979e-03
17,(4.128e-04,4.128e-04),1.168e-03
18,(2.438e-04,2.438e-04),6.897e-04
19,(1.439e-04,1.439e-04),4.072e-04
20,(8.498e-05,8.498e-05),2.404e-04
21,(5.017e-05,5.017e-05),1.419e-04
22,(2.962e-05,2.962e-05),8.379e-05
23,(1.749e-05,1.749e-05),4.947e-05
24,(1.033e-05,1.033e-05),2.921e-05
25,(6.096e-06,6.096e-06),1.724e-05
26,(3.599e-06,3.599e-06),1.018e-05
27,(2.125e-06,2.125e-06),6.011e-06
28,(1.255e-06,1.255e-06),3.549e-06
29,(7.407e-07,7.407e-07),2.095e-06
30,(4.373e-07,4.373e-07),1.237e-06
31,(2.582e-07,2.582e-07),7.303e-07
Start coding or generate with AI.
22:06 / 25:52
0s completed at 16:56
Bilingual
YouTube





So why did we introduce the DFP method? The reason is that we can mathematically prove that when using the DFP method with an exact line search (where α_k is found by exactly minimizing $f(x_k + \alpha d_k)$), the matrix B_k remains positive definite throughout the optimization.

This result does not necessarily hold for backtracking line search or other inexact line searches. But under exact line search, this property is guaranteed. We will show this proof and discuss the BFGS method in the next lecture.

Thank you.