Optimization Algorithms: Theory and Software Implementation

Prof. Thirumulanathan D

Department of Mathematics

Institute of IIT Kanpur

Lecture: 27

Hello everyone, this is the second lecture of week six. In the previous lecture, we learnt about the basics of quasi-Newton methods. The idea was to use these methods to implement a Newton-like approach but avoid the inversion of the Hessian matrix. The strategy is to maintain an approximation of the inverse, let's call it B_k, and then update B_k to B_{k+1} using a simple formula, assuming the change between successive approximations is small.

We looked at one such method: the rank-one correction. In this method, we update B_{k+1} by adding a rank-one matrix to B_k. We derived an expression for this update, which I have boxed here on the screen.

The algorithm is to start by initializing a positive definite matrix B_0. Then, you choose a descent direction, choose an appropriate step size using a line search, and then update B_{k+1} using the vectors δ_k and γ_k. That is the basic algorithm.

We also learned the Python code to implement this for quadratic functions. We generated a random 5x5 positive definite matrix H and a random vector b. We started from a random initial point and initialized B_0 as the identity matrix. We found that the algorithm converged to the solution in just 5 steps because the dimension n was 5.

You can try this for any higher value of n, for any arbitrary positive definite H, any arbitrary vector b, any arbitrary starting point x_0, and even any arbitrary starting matrix B_0 (as long as it is positive definite). For all these cases, you will see that you get the answer for a quadratic function in just n steps. This is a good property of the rank-one correction method.

I would like to indicate one more property. If I print the inverse of H, H^{-1}, and also print the final B matrix we obtained after n steps, you can see that both matrices are exactly the same. This means that when you start with a quadratic function and update the B matrix, in exactly n steps you reach the true inverse of the Hessian, H^{-1}, regardless of what positive definite matrix you started with.

We will now prove these two claims.

Note: $B_{k+1}= B^{k+1}$, $\gamma_j= \gamma^j$, $\delta_j= \delta^j$ and $x_n= x^n$

Claim 1: For a quadratic function $f(x) = (1/2)x^THx + b^Tx + c$, with H positive definite, using the rank-one correction method, we attain the minimizer x^* in at most n steps.

That is, $x_n = x^*$.

Claim 2: The final approximation matrix is exactly the inverse of the Hessian, i.e., $B_n = H^{-1}$.

We will prove these under the mild condition that the steps $\delta_0, \delta_1, ..., \delta_{n-1}$ are linearly independent.

Let's recall the quasi-Newton condition: $B_{k+1}\gamma_k = \delta_k$.

This gives us a set of equations:

$B_1\gamma_0 = \delta_0$

$B_2\gamma_1 = \delta_1$

...

$B_n\gamma_{n-1} = \delta_{n-1}$

A stronger, "hereditary" property is that later matrices satisfy the conditions of earlier steps. For example, does $B_2\gamma_0 = \delta_0$?

Does $B_3\gamma_1 = \delta_1$ and $B_3\gamma_0 = \delta_0$? And ultimately, does $B_n\gamma_j = \delta_j$ for all j from 0 to n-1?

We will prove this hereditary property by mathematical induction.

The base case is clear: $B_1\gamma_0 = \delta_0$ is true from the quasi-Newton condition.

Now, assume that $B_k\gamma_j = \delta_j$ for all j = 0, 1, ..., k-1.

We will show that $B_{k+1}\gamma_j = \delta_j$ for all j = 0, 1, ..., k.

The update formula is:

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

Now, let's compute $B_{k+1}\gamma_j$ for some j < k:

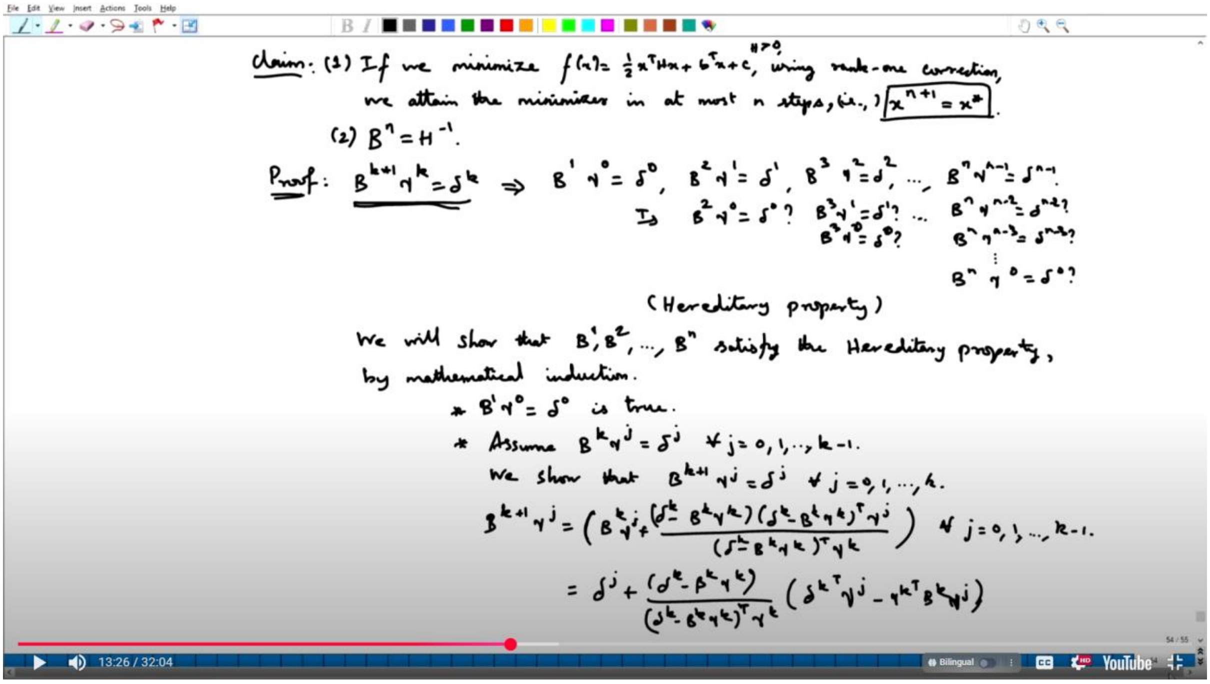
$B_{k+1}\gamma_j = B_k\gamma_j + [(\delta_k - B_k\gamma_k) (\delta_k - B_k\gamma_k)^T\gamma_j] / [(\delta_k - B_k\gamma_k)^T\gamma_k]$

By our induction hypothesis, $B_k\gamma_j = \delta_j$.

So, we have:

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

(Refer Slide Time 13:26)



We now focus on the numerator of the fraction, specifically the scalar term $(\delta_k - B_k\gamma_k)^T\gamma_j$.

$(\delta_k - B_k\gamma_k)^T\gamma_j = \delta_k^T\gamma_j - \gamma_k^T B_k\gamma_j$

Because we are dealing with a quadratic function, we know that $\gamma_j = g_{j+1} - g_j$. For our quadratic function, the gradient is $g(x) = Hx + b$. Therefore:

$\gamma_j = g_{j+1} - g_j = (Hx_{j+1} + b) - (Hx_j + b) = H(x_{j+1} - x_j) = H\delta_j$

Similarly, $\gamma_k = H\delta_k$.

Now, let's substitute this into our expression:

$\delta_k^T\gamma_j - \gamma_k^T B_k\gamma_j = \delta_k^T(H\delta_j) - (H\delta_k)^T B_k\gamma_j$

$= \delta_k^T H\delta_j - \delta_k^T H^T B_k\gamma_j$

Since H is symmetric ($H = H^T$), this becomes:

$= \delta_k^T H\delta_j - \delta_k^T H B_k\gamma_j$

From our induction hypothesis, we know $B_k\gamma_j = \delta_j$. Substituting this gives:

$= \delta_k^T H\delta_j - \delta_k^T H\delta_j = 0$

Therefore, the entire second term in the update formula is zero.

This means $B_{k+1}\gamma_j = \delta_j$ for all $j < k$.

For the case $j = k$, we already know $B_{k+1}\gamma_k = \delta_k$ from the quasi-Newton condition.

Thus, by induction, the hereditary property holds: $B_k\gamma_j = \delta_j$ for all $j < k$.

Now, let's apply this for $k = n$. We have:

$B_n\gamma_j = \delta_j$ for all $j = 0, 1, ..., n-1$.

We can write this in matrix form. Let's create matrices from these vectors:

$B_n [\gamma_0 | \gamma_1 | ... | \gamma_{n-1}] = [\delta_0 | \delta_1 | ... | \delta_{n-1}]$

But we know that $\gamma_j = H\delta_j$ for all j. Substituting this gives:

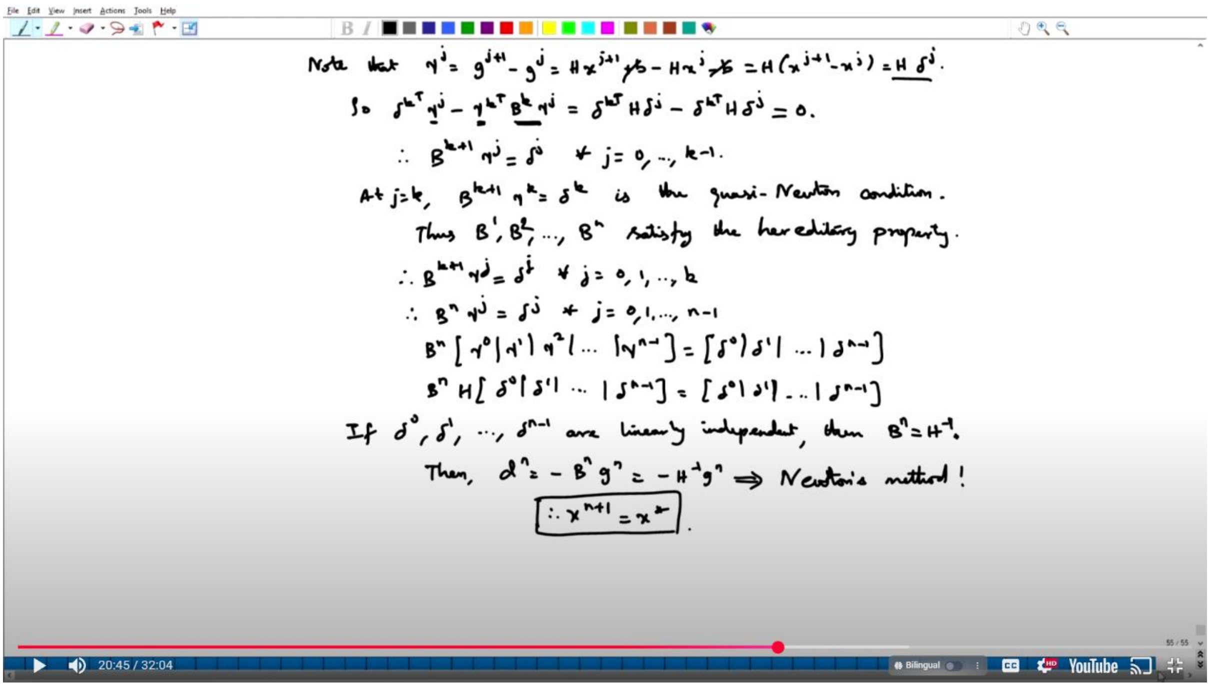
$B_n H [\delta_0 | \delta_1 | ... | \delta_{n-1}] = [\delta_0 | \delta_1 | ... | \delta_{n-1}]$

If the vectors $\delta_0, \delta_1, ..., \delta_{n-1}$ are linearly independent, then the matrix $[\delta_0 | \delta_1 | ... | \delta_{n-1}]$ is invertible. This allows us to conclude:

$B_n H = I$

Therefore, $B_n = H^{-1}$. This proves Claim 2. Now, if $B_n = H^{-1}$, then the search direction at step n is $d_n = -B_n g_n = -H^{-1}g_n$, which is the Newton direction. A Newton step from x_{n+1} will take us directly to the minimizer x^* that is $x_{n+1} = x^*$. This proves Claim 1.

(Refer Slide Time 20:45)



So, for any quadratic function in n dimensions, starting from any initial point and any initial positive definite matrix B_0, the rank-one correction method converges to the minimizer in n steps and exactly computes the inverse of the Hessian along the way without explicitly inverting it.

However, most functions we work with are not quadratic. What happens then? We will test the method on the non-quadratic function we have been using:

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

Let's implement the rank-one correction for this general function. We define the function f(x) and its gradient grad_f(x). We start from the point (1, 1). We initialize B_0 as the 2x2 identity matrix.

The algorithm is similar to the quadratic case:

For each iteration k:

1. Compute the descent direction: $d_k = -B_k$ * grad_f(x_k)
2. Perform a backtracking line search to find a step size α_k that satisfies the Armijo condition:

   f($x_k + \alpha d_k$) > f(x_k) + c_1 * α * grad_f(x_k)$^T d_k$

   We initialize parameters, typically c_1=0.75 and the reduction factor ρ=0.8.

3. Update the point: $x_{k+1} = x_k + \alpha_k d_k$
4. Compute the differences:

   $\delta_k = x_{k+1} - x_k$

   γ_k = grad_f(x_{k+1}) - grad_f(x_k)

5. Update the matrix B using the rank-one formula:

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

For our function, the algorithm took 31 steps to converge to the answer (0, 0).

Finally, I need to explain a important Python-specific detail in the code. The vectors δ_k and γ_k are stored as one-dimensional arrays. In Python, if a is a one-dimensional array, a.T (the transpose) is still the same one-dimensional array.

The update formula requires the outer product of the vector ($\delta_k - B_k\gamma_k$) with itself. This should result in an n x n matrix. However, if we treat these vectors as one-dimensional arrays, the operation (dl - B @ gm) * (dl - B @ gm).T would be interpreted as an element-wise multiplication or a dot product, resulting in a scalar, which is wrong.

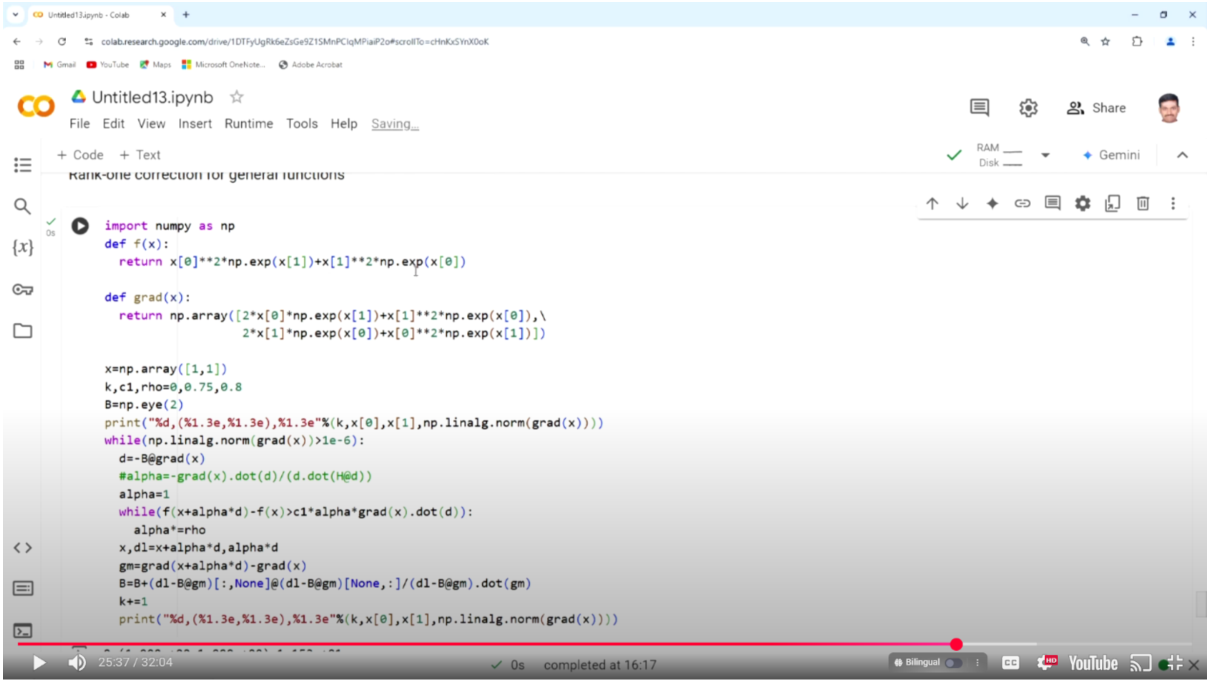
To fix this, we must explicitly convert these one-dimensional arrays into two-dimensional column and row vectors. We can do this using indexing with np.newaxis (or None).

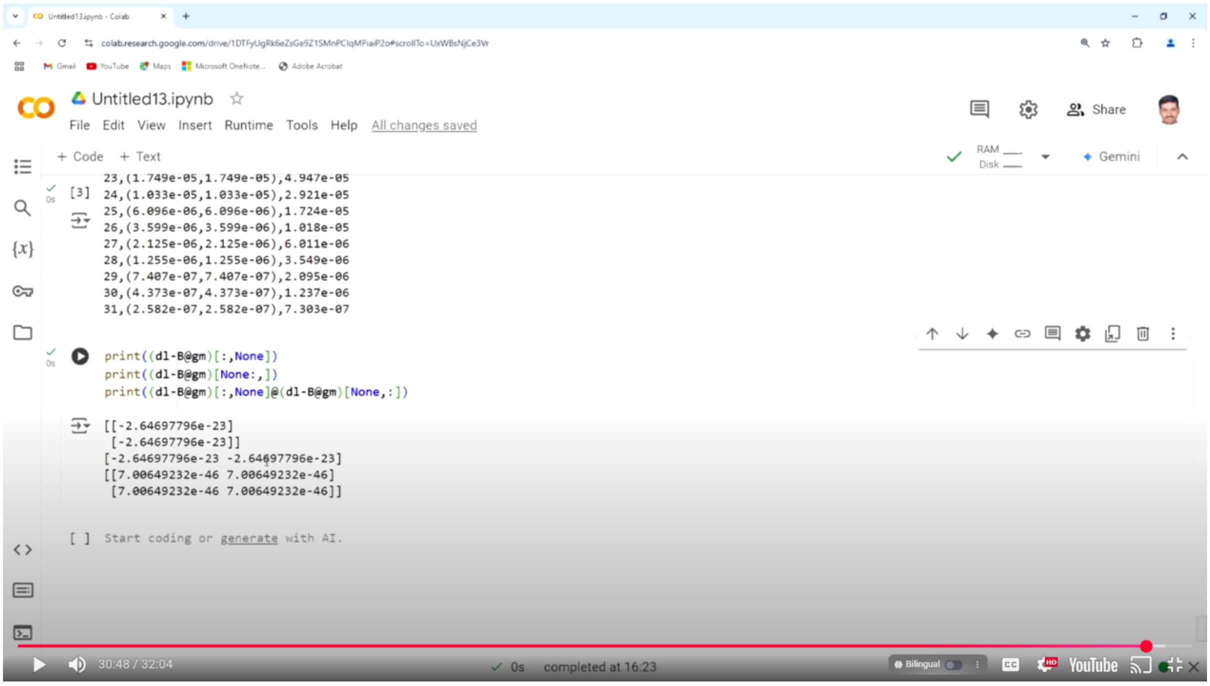
v_col = v[:, np.newaxis] creates a column vector (an n x 1 matrix).

v_row = v[np.newaxis, :] creates a row vector (a 1 x n matrix).

Then, the outer product v_col @ v_row correctly produces an n x n matrix. This is a crucial implementation detail to ensure the mathematical correctness of the algorithm.

(Refer Slide Time 31:50)





We will continue with this example in the next lecture. Thank you.