Optimization Algorithms: Theory and Software Implementation

Prof. Thirumulanathan D

Department of Mathematics

Institute of IIT Kanpur

Lecture: 26

Hello everyone. This is the first lecture of the sixth week. We will be starting with quasi-Newton methods this week.

Until now, we have covered line search methods, both exact and backtracking line searches. To find the descent directions, we have discussed the gradient descent method, the popular Fletcher-Reeves conjugate gradient method, and the eigenvalue/eigenvector method. Last week, we discussed Newton's methods and variants like the modified Newton method and the damped Newton method.

We were able to rectify most of the issues that Newton's methods faced, except for one particular issue: the inversion of the Hessian matrix. The inversion of the Hessian matrix is computationally very expensive; it takes a lot of time if n is large. It is not scalable. That is the reason we move to quasi-Newton methods, which is what we will be learning this week.

You can see that variants of Newton's methods are named along with Newton itself. We saw damped Newton methods and modified Newton method. Now we have a few different quasi-Newton methods. We will see the popular quasi-Newton methods this week.

Where do we start? Please recall that for Newton's method, we considered the second-order Taylor's approximation of f:

$$f(x) \approx f(x_k) + \nabla f(x_k)^T(x - x_k) + \tfrac{1}{2}(x - x_k)^T\nabla^2 f(x_k)(x - x_k)$$

Notation: Hessian$=\nabla^2=\nabla_2$

We might have written this as g_k^T and H_k.

From this, if you recall what we did in Newton's method, we differentiated this with respect to x and evaluated it at x*, which is the minimizer of f. Differentiating the above equation and evaluating at x = x*, we get:

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

From here, we actually got the direction to be:

$$d_k = -H_k^{-1}g_k$$

This was the basis of Newton's method.

Now, how do we actually avoid the inverse in this step? If taking the inverse is costly and computationally expensive, how do we avoid this step H_k^{-1}? The idea is as follows. We consider that H_k^{-1} and H_{k+1}^{-1} are close to each other; they are not very different because $x_{k+1} = x_k + \alpha_k d_k$. The difference between x_{k+1} and x_k is not too much, so we expect the difference between H_k^{-1} and H_{k+1}^{-1} not to be too much either. We try to update H_k^{-1} to H_{k+1}^{-1}. That is how the quasi-Newton method comes into practice.

How to avoid inversion of the Hessian? The idea is to update H_k^{-1} to H_{k+1}^{-1}, assuming that they are not very different. I am just giving the heuristics; this is not a well-defined statement. I am saying "very different" and "update" to help you understand what is actually happening in the quasi-Newton method so that you can follow the different methods and algorithms.

Now, let us define $B_k = H_k^{-1}$. Then we have the second-order approximation using this equation:

$$f(x) \approx f(x_k) + g_k^T(x - x_k) + \tfrac{1}{2}(x - x_k)^T B_k^{-1}(x - x_k)$$

It should have been H_k, but instead I am writing B_k^{-1} now.

With this definition, in the idea above, I need to update B_{k+1} using B_k. I need to update B_k to B_{k+1}, just like how I update x_{k+1} as $x_k + \alpha_k d_k$. How do we do it? I will differentiate this particular equation (the last equation I have written) and evaluate it at $x = x_{k+1}$.

Differentiating the above equation and evaluating at $x = x_{k+1}$, we get:

$$\nabla f(x_{k+1}) = g_{k+1} = g_k + B_k^{-1}(x_{k+1} - x_k)$$

This is:

$$g_{k+1} - g_k = B_k^{-1}(x_{k+1} - x_k)$$

Define $\gamma_k = g_{k+1} - g_k$ and $\delta_k = x_{k+1} - x_k$. So we have:

$$\gamma_k = B_k^{-1}\,\delta_k$$

Or, equivalently:

$\delta_k = B_k\,\gamma_k$? Let's be careful. From $g_{k+1} - g_k = B_k^{-1}\,\delta_k$, multiplying both sides by B_k gives:

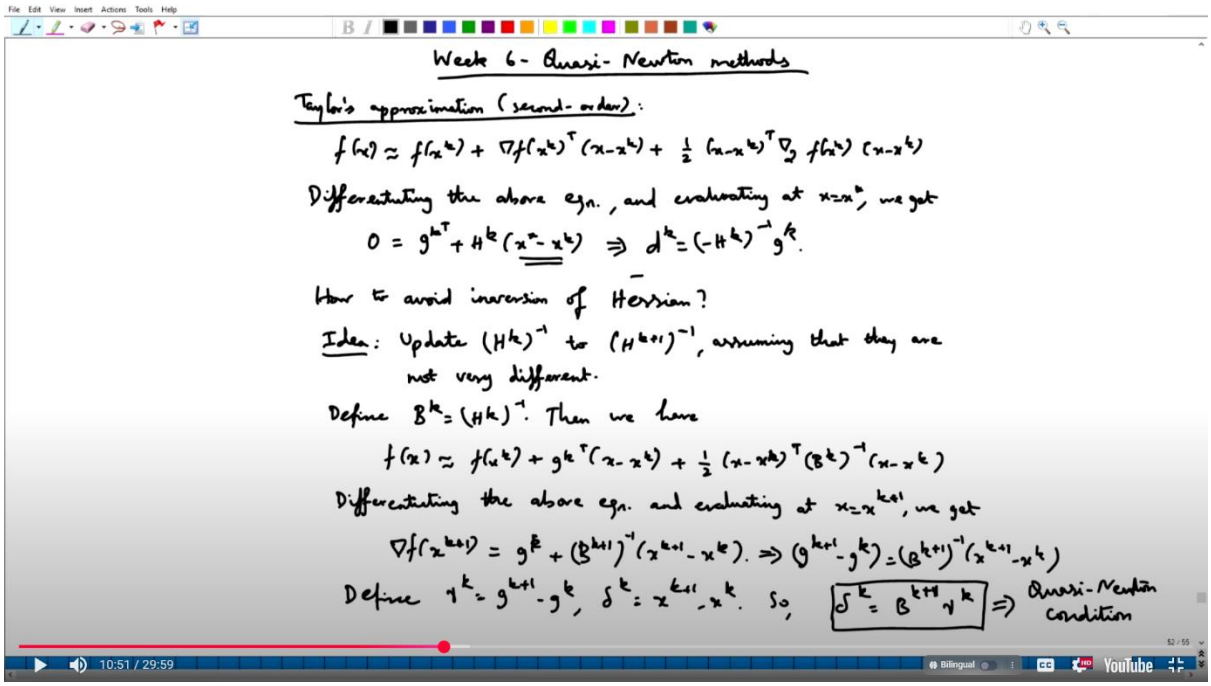
$$B_k\,(g_{k+1} - g_k) = \delta_k$$

So, $\delta_k = B_k\,\gamma_k$. But we want to update B_k to B_{k+1}. The condition we want for the new matrix is that it satisfies a similar secant equation. The standard quasi-Newton condition is:

$$\delta_k = B_{k+1}\,\gamma_k$$

This is called the **quasi-Newton condition**. When you update B_k to B_{k+1}, you must make sure that this quasi-Newton condition is satisfied. That is basically how we construct different quasi-Newton methods.

(Refer Slide Time 10:51)



This week, we will see three different quasi-Newton methods. One is called the rank one correction. The other is the DFP method (Davidon-Fletcher-Powell), named after the discoverers. The third one is the BFGS method (Broyden-Fletcher-Goldfarb-Shanno), named after the four optimizers who came up with these algorithms. We will see these three quasi-Newton methods, their conditions, differences in the algorithms, their performances, and so on.

We will start with rank one correction. We have to update B_{k+1} from B_k. As the name suggests, we will update this matrix B_{k+1} using a rank 1 matrix. Any rank 1 matrix can be written in the form $\beta\, u\, u^T$, where u is an n-length vector. So we update it as:

$$B_{k+1} = B_k + \beta\, u\, u^T$$

We need to find out what β and u are so that the quasi-Newton condition is satisfied. We want $B_{k+1}\, \gamma_k = \delta_k$.

Substituting the update:

$$(B_k + \beta\, u\, u^T)\, \gamma_k = \delta_k$$

$$B_k\, \gamma_k + \beta\, u\, (u^T\, \gamma_k) = \delta_k$$

Note that $u^T\, \gamma_k$ is a scalar. So:

$$\beta\, (u^T\, \gamma_k)\, u = \delta_k - B_k\, \gamma_k$$

Let us choose u to be $\delta_k - B_k\, \gamma_k$. In that case, we need $\beta\, (u^T\, \gamma_k)$ to be 1. Therefore, we have:

$$\beta = 1 / (u^T\, \gamma_k)$$

And $u = \delta_k - B_k\, \gamma_k$.

Therefore, we actually have:

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

This is what we call the rank one correction update. Whenever you update your B matrix using this particular equation (the boxed one), that is the rank one correction method.

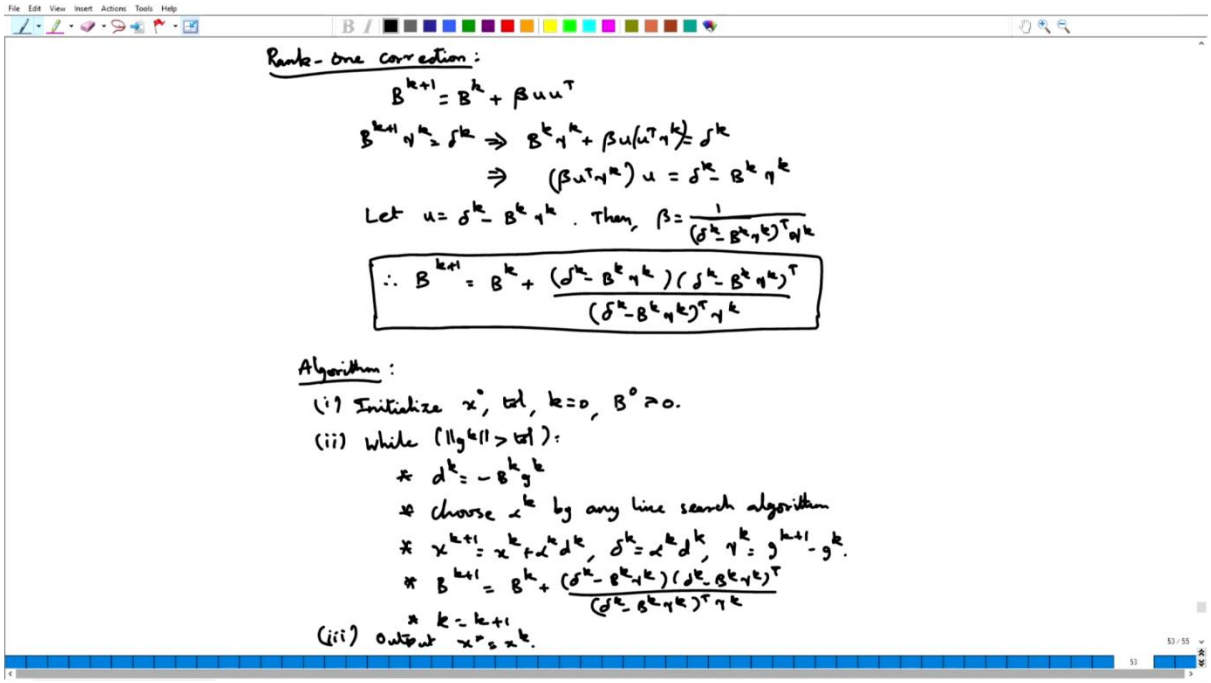
I will write down the algorithm fully. We first start by initializing x_0, a tolerance, and $k = 0$. Here we also have to initialize B_0. The starting point B_0 needs to be a symmetric positive definite matrix. It cannot be any arbitrary matrix. This is an extra step compared to the algorithms we discussed until now.

While $\|g_k\| >$ tolerance:

1. Compute the descent direction: $d_k = -B_k g_k$ (since H_k^{-1} is approximated by B_k).
2. Choose α_k by a line search algorithm (exact or backtracking).
3. Update the iterate: $x_{k+1} = x_k + \alpha_k d_k$.
4. Compute the differences: $\delta_k = x_{k+1} - x_k = \alpha_k d_k$, $\gamma_k = g_{k+1} - g_k$.
5. Update the matrix: $B_{k+1} = B_k + ((\delta_k - B_k \gamma_k) (\delta_k - B_k \gamma_k)^T) / ((\delta_k - B_k \gamma_k)^T \gamma_k)$.
6. Set $k = k + 1$.

Finally, output $x^* = x_k$.

(Refer Slide Time 20:04)



This is basically the rank one correction algorithm. The first thing we do after writing down an algorithm is to evaluate it for quadratic functions. That is how we derived the theories for many

methods earlier. You can recall that conjugate gradient methods (the eigenvector method or Fletcher-Reeves) find the answer in n steps for quadratic functions, and Newton's method finds it in one step if the Hessian of the quadratic function is positive definite.

Let us try the same with rank one correction. I claim that for quadratic functions, this converges in n steps. Let us write down the code for this particular algorithm and check whether the steps are followed.

This is rank one correction for quadratic functions. Import numpy. I would like to use a 5x5 positive definite matrix, like we did for conjugate gradient and Newton's method. Let H be a random matrix consisting of integers between -9 and 9, a 5x5 matrix. I want to make this positive definite, so I will consider $H H^T + I$.

Consider the quadratic function $f(x) = ½ x^T H x + b^T x$. b can be a random vector.

Define f(x) as 0.5 * np.dot(x, np.dot(H, x)) + np.dot(b, x).

The gradient grad(x) is np.dot(H, x) + b.

Now, do the initializations. Initialize x randomly. Initialize k = 0. Initialize B as the identity matrix I (which is positive definite).

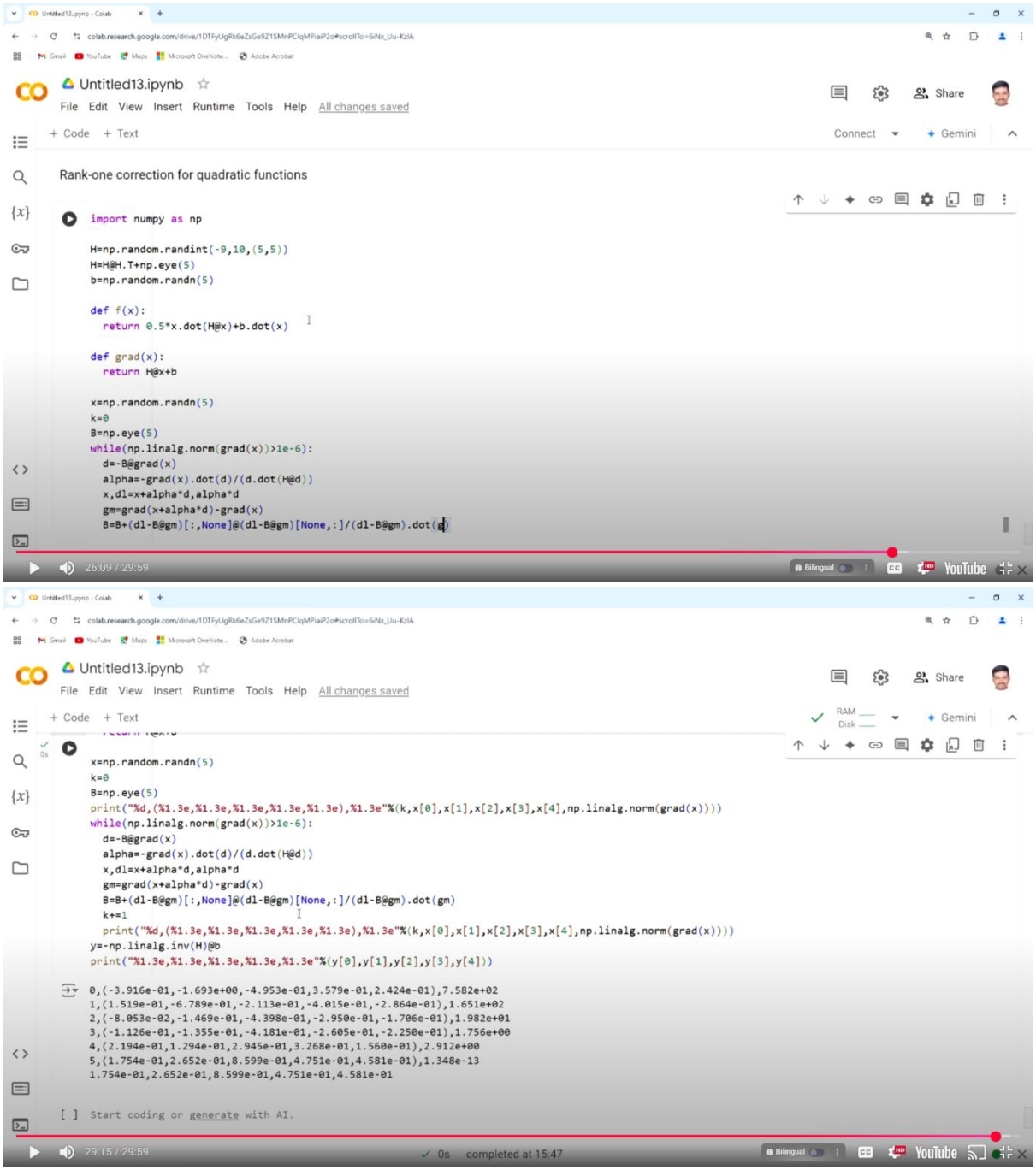
While np.linalg.norm(grad(x)) > tolerance (say 1e-6):

1. d = - np.dot(B, grad(x))
2. For a quadratic function, use exact line search: α = - (np.dot(grad(x), d)) / (np.dot(d, np.dot(H, d)))
3. Update x = x + α * d
4. Compute δ = α * d (which is x_new - x_old)
5. Compute γ = grad(x_new) - grad(x_old)
6. Update B: First compute the vector u = δ - np.dot(B, γ). Then compute the denominator denom = np.dot(u, γ). Then B = B + np.outer(u, u) / denom
7. k = k + 1

You can also print statements to track k, x, and the norm of the gradient.

The true solution is x* = -np.linalg.inv(H) @ b. You can verify that the algorithm finds this solution in n=5 steps. Note that I chose a random H (positive definite), a random b, and started with a random x. My B_0 was the identity matrix. Given that the function is quadratic, this rank one correction method found the answer in just n steps. No inversion of H has happened at all. This is just to verify that the answer is correct.

(Refer Slide Time 29:15)



So, no inversion has happened, but I have managed to find the answer for any quadratic function in n steps. That is how good the rank one correction method is. In the next lecture, we will understand more about rank one correction and then move to the other methods.

Thank you.