

# 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  $\delta_k$  and  $\gamma_k$ . 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:**  $B_{k+1} = B^k$ ,  $\gamma_j = \gamma^j$ ,  $\delta_j = \delta^j$  and  $x_n = x^*$

**Claim 1:** For a quadratic function  $f(x) = (1/2)x^T H x + b^T x + c$ , 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  $\delta_0, \delta_1, \dots, \delta_{n-1}$  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  $j = 0, 1, \dots, k-1$ .

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

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)

Claim: (1) If we minimize  $f(x) = \frac{1}{2}x^T H x + b^T x + c$ , using rank-one correction, we attain the minimizer in at most  $n$  steps, i.e.,  $\boxed{x^{n+1} = x^*}$ .

(2)  $B^n = H^{-1}$ .

Proof:  $B^{k+1} \gamma^k = \delta^k \Rightarrow B^1 \gamma^0 = \delta^0, B^2 \gamma^1 = \delta^1, B^3 \gamma^2 = \delta^2, \dots, B^n \gamma^{n-1} = \delta^{n-1}$ .

$\Rightarrow B^2 \gamma^0 = \delta^0? \quad B^3 \gamma^1 = \delta^1? \quad \dots \quad B^n \gamma^{n-2} = \delta^{n-2}?$   
 $B^3 \gamma^0 = \delta^0? \quad B^n \gamma^{n-3} = \delta^{n-3}?$   
 $\vdots$   
 $B^n \gamma^0 = \delta^0?$

(Hereditary property)

We will show that  $B^1, B^2, \dots, B^n$  satisfy the Hereditary property, by mathematical induction.

- \*  $B^1 \gamma^0 = \delta^0$  is true.
- \* Assume  $B^k \gamma^j = \delta^j \quad \forall j = 0, 1, \dots, k-1$ .

We show that  $B^{k+1} \gamma^j = \delta^j \quad \forall j = 0, 1, \dots, k$ .

$$B^{k+1} \gamma^j = \left( B^k \gamma^j + \frac{(\delta^k - B^k \gamma^k)(\delta^k - B^k \gamma^k)^T \gamma^j}{(\delta^k - B^k \gamma^k)^T \gamma^k} \right) \quad \forall j = 0, 1, \dots, k-1.$$

$$= \delta^j + \frac{(\delta^k - B^k \gamma^k)}{(\delta^k - B^k \gamma^k)^T \gamma^k} (\delta^{kT} \gamma^j - \gamma^{kT} B^k \gamma^j)$$

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:

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

$$\text{Similarly, } \gamma_k = H\delta_k.$$

Now, let's substitute this into our expression:

$$\begin{aligned} \delta_k^T \gamma_j - \gamma_k^T B_k \gamma_j &= \delta_k^T (H\delta_j) - (H\delta_k)^T B_k \gamma_j \\ &= \delta_k^T H \delta_j - \delta_k^T H^T B_k \gamma_j \end{aligned}$$

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  $j = 0, 1, \dots, n-1$ .

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

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

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

$$B_n H [\delta_0 \mid \delta_1 \mid \dots \mid \delta_{n-1}] = [\delta_0 \mid \delta_1 \mid \dots \mid \delta_{n-1}]$$

If the vectors  $\delta_0, \delta_1, \dots, \delta_{n-1}$  are linearly independent, then the matrix  $[\delta_0 \mid \delta_1 \mid \dots \mid \delta_{n-1}]$  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)

Note that  $\eta^j = g^{j+1} - g^j = H x^{j+1} - H x^j = H(x^{j+1} - x^j) = H \delta^j$ .  
 So  $\delta^{kT} \eta^j - \eta^{kT} B^k \eta^j = \delta^{kT} H \delta^j - \delta^{kT} H \delta^j = 0$ .  
 $\therefore B^{k+1} \eta^j = \delta^j \quad \forall j = 0, \dots, k-1$ .  
 At  $j=k$ ,  $B^{k+1} \eta^k = \delta^k$  is the quasi-Newton condition.  
 Thus  $B^1, B^2, \dots, B^n$  satisfy the hereditary property.  
 $\therefore B^{k+1} \eta^j = \delta^j \quad \forall j = 0, 1, \dots, k$   
 $\therefore B^n \eta^j = \delta^j \quad \forall j = 0, 1, \dots, n-1$   
 $B^n [\eta^0 \mid \eta^1 \mid \eta^2 \mid \dots \mid \eta^{n-1}] = [\delta^0 \mid \delta^1 \mid \dots \mid \delta^{n-1}]$   
 $B^n H [\delta^0 \mid \delta^1 \mid \dots \mid \delta^{n-1}] = [\delta^0 \mid \delta^1 \mid \dots \mid \delta^{n-1}]$   
 If  $\delta^0, \delta^1, \dots, \delta^{n-1}$  are linearly independent, then  $B^n = H^{-1}$ .  
 Then,  $d^n = -B^n g^n = -H^{-1} g^n \Rightarrow$  Newton's method!  
 $\therefore x^{n+1} = x^*$ .

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  $\text{grad}_f(x)$ . 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:  $d_k = -B_k * \text{grad}_f(x_k)$
2. Perform a backtracking line search to find a step size  $\alpha_k$  that satisfies the Armijo condition:

$$f(x_k + \alpha d_k) > f(x_k) + c_1 * \alpha * \text{grad}_f(x_k)^T d_k$$

We initialize parameters, typically  $c_1=0.75$  and the reduction factor  $\rho=0.8$ .

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

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

$$\gamma_k = \text{grad}_f(x_{k+1}) - \text{grad}_f(x_k)$$

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 an important Python-specific detail in the code. The vectors  $\delta_k$  and  $\gamma_k$  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  $n \times n$  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  $n \times 1$  matrix).

`v_row = v[np.newaxis, :]` creates a row vector (a  $1 \times n$  matrix).

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

(Refer Slide Time 31:50)

The image displays two screenshots of a Jupyter Notebook interface, likely from a video lecture. The top screenshot shows the code for a Kalman filter implementation. The bottom screenshot shows the output of the code, which is a list of 31 rows of numerical data.

**Top Screenshot: Code**

```
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,d1=x+alpha*d,alpha*d
    gm=grad(x+alpha*d)-grad(x)
    B=B+(d1-B@gm)[None,:]*(d1-B@gm)[None,:]/(d1-B@gm).dot(gm)
    k+=1
    print("%d, (%1.3e,%1.3e), (%1.3e"%(k,x[0],x[1],np.linalg.norm(grad(x))))
```

**Bottom Screenshot: Output**

```
23, (1.749e-05, 1.749e-05), 4.947e-05
[3] 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

print((d1-B@gm)[None,:])
print((d1-B@gm)[None,:])
print((d1-B@gm)[None,:])

[[-2.64697796e-23]
 [-2.64697796e-23]]
[-2.64697796e-23 -2.64697796e-23]
[[7.00649232e-46 7.00649232e-46]
 [7.00649232e-46 7.00649232e-46]]

[ ] Start coding or generate with AI.
```

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