

Optimization Algorithms: Theory and Software Implementation

Prof. Thirumulanathan D

Department of Mathematics

Institute of IIT Kanpur

Lecture: 25

Hello everyone, this is the fifth lecture of week five. In the previous lecture, we looked at two variants of Newton's method. One is the damped Newton's method, which takes care of issues regarding a lack of global convergence. The other method is the modified Newton's method, which takes care of issues where $-H_k^{-1}\nabla f_k$ might not be a descent direction or where the Hessian matrix H_k turns out to be singular. So, all of those issues can be rectified by either using the damped Newton's method or the modified Newton's method.

We still have one more issue. Let me point that out to you. We took care of the global convergence problem. We took care of the descent direction problem. We also took care of the singular H_k problem. We still have the issue that matrix inversion is a computationally expensive process. Yes, it is a computationally expensive process, and you can also see that in the damped Newton's method and modified Newton's method we still have this computationally intensive process of inverting either H_k or $(H_k + \lambda I)$. So, we have not escaped this computationally expensive process with these small modifications.

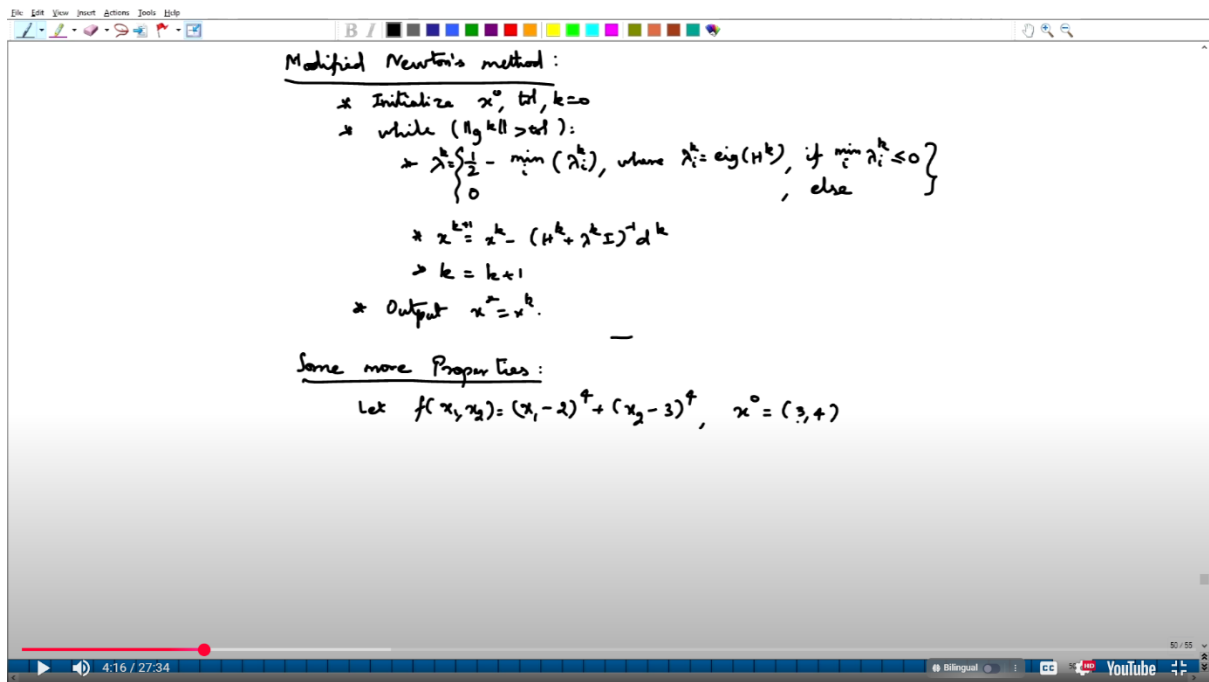
How do we take care of this issue? We actually use something called the quasi-Newton method. But that is a topic for next week. So in week six, we will discuss various quasi-Newton methods where you will not have the matrix inversion process. Without the matrix inversion process, you are actually going to solve the optimization problem. We will shift that to next week; this being the last lecture of this week on Newton's method, we will actually discuss some more properties of Newton's method.

Let me start with a particular problem. Let

$$f(x_1, x_2) = (x_1 - 2)^4 + (x_2 - 3)^4.$$

Just by looking at this, you would know that the minimum of f occurs at $(2, 3)$. Let me try to solve this using Newton's method where $x^{(0)}$ is $(3, 4)$.

(Refer Slide Time 4:16)



Let us try and check if we get something out of this particular example. I will just write this as Newton's method with higher-order functions. Of course, we want to just experiment with Newton's method, so I will remove all the damped and modified Newton's method parts. I just copied this, but I will remove λ .

Now this is the Newton's method, and I will modify the code to accommodate this problem: $(x_1 - 2)^4 + (x_2 - 3)^4$.

The gradient will be $[4(x_1 - 2)^3, 4(x_2 - 3)^3]$ and

Hessian = $[[12(x_1 - 2)^2, 0], [0, 12(x_2 - 3)^2]]$.

The answer is (2,3), so we will start with (3,4) just to see what happens.

I had written the values of the gradient and Hessian slightly incorrectly; this is the actual gradient of x and Hessian of x . You can see that it got to the answer (2, 3); there is a slight error (2.005, 3.005) but that is because of the tolerance of $1e^{-6}$. So, you might think that after seeing gradient descent and, say, the conjugate gradient Fletcher-Reeves algorithm where it takes 30 or 40 steps, 13 seems to be a smaller number. That is correct, but for Newton's method, 13 is actually a high number. Usually, you can see that many algorithms converge in a much smaller number of steps, in single digits like 4, 7, or 6. If it is 13, it is actually on the higher side. That is just to indicate to you that 13 is not a small number. So if it takes a little more iterations, we should try and figure out what the reason is.

(Refer Slide Time 8:06)

```

import numpy as np
def f(x):
    return (x[0]-2)**4+(x[1]-3)**4

def grad(x):
    return np.array([4*(x[0]-2)**3, 4*(x[1]-3)**3])

def H(x):
    return np.array([[12*(x[0]-2)**2, 0], [0, 12*(x[1]-3)**2]])

x=np.array([3,4])
k=0
print("%d, (%1.3e, %1.3e), (%1.3e, %1.3e)"%(k, x[0], x[1], f(x), np.linalg.norm(grad(x))))
while(np.linalg.norm(grad(x))>1e-6):
    x=x-np.linalg.inv(H(x))@grad(x)
    k+=1
    print("%d, (%1.3e, %1.3e), (%1.3e, %1.3e)"%(k, x[0], x[1], f(x), np.linalg.norm(grad(x))))

```

0, (3.000e+00, 4.000e+00), 2.000e+00, 5.657e+00
1, (2.667e+00, 3.667e+00), 3.951e-01, 1.676e+00
2, (2.444e+00, 3.444e+00), 7.804e-02, 4.966e-01
3, (2.296e+00, 3.296e+00), 1.541e-02, 1.471e-01
4, (2.198e+00, 3.198e+00), 3.045e-03, 4.360e-02
5, (2.132e+00, 3.132e+00), 6.015e-04, 1.292e-02

We will do that, but before that, I will show you a few more things. Let me actually increase this power. This is a fourth-order equation, a quartic equation. Let us say I put the order of the function as 10. So that is

$$f(x) = (x_0 - 2)^{10} + (x_1 - 3)^{10}.$$

The gradient will be $[10(x_0 - 2)^9, 10(x_1 - 3)^9]$ and

$$\text{Hessian} = \begin{bmatrix} 90(x_0 - 2)^8 & 0 \\ 0 & 90(x_1 - 3)^8 \end{bmatrix}.$$

I can start with (3, 4). Let us see where this is going to take us. It has taken us to the answer in 16 steps, but you can see that the error is slightly large. That is also not because of a problem with the algorithm; it is because of the tolerance we have chosen. Even $(0.15)^9$, the whole square plus another $(0.15)^9$ squared, the square root actually crosses $1e^{-6}$, that is the reason. That is not the problem, but you can see that the number of iterations has increased.

(Refer Slide Time

```

import numpy as np
def f(x):
    return (x[0]-2)**10*(x[1]-3)**10

def grad(x):
    return np.array([10*(x[0]-2)**9, 10*(x[1]-3)**9])

def H(x):
    return np.array([[90*(x[0]-2)**8, 0], [0, 90*(x[1]-3)**8]])

x=np.array([3,4])
k=0
print("%d, (%1.3e, %1.3e), (%1.3e, %1.3e)"%(k, x[0], x[1], f(x), np.linalg.norm(grad(x))))
while(np.linalg.norm(grad(x))>1e-6):
    x=x-np.linalg.inv(H(x))@grad(x)
    k+=1
    print("%d, (%1.3e, %1.3e), (%1.3e, %1.3e)"%(k, x[0], x[1], f(x), np.linalg.norm(grad(x))))

```

0, (3.000e+00, 4.000e+00), 2.000e+00, 1.414e+01
1, (2.889e+00, 3.889e+00), 6.159e-01, 4.899e+00
2, (2.790e+00, 3.790e+00), 1.897e-01, 1.697e+00
3, (2.702e+00, 3.702e+00), 5.841e-02, 5.880e-01
4, (2.624e+00, 3.624e+00), 1.799e-02, 2.037e-01
5, (2.555e+00, 3.555e+00), 5.539e-03, 7.058e-02

Now, let me increase this to 100.

The gradient will be $[100(x_0 - 2)^9, 100(x_1 - 3)^9]$ and

$$\text{Hessian} = \begin{bmatrix} 9900(x_0 - 2)^8 & 0 \\ 0 & 9900(x_1 - 3)^8 \end{bmatrix}.$$

Let us see how many steps it takes.

It has increased, and you can see that the answer is (2.825, 3.825).

(Refer Slide Time 10:18)

```

import numpy as np
def f(x):
    return (x[0]-2)**100*(x[1]-3)**100

def grad(x):
    return np.array([100*(x[0]-2)**99, 100*(x[1]-3)**99])

def H(x):
    return np.array([[9900*(x[0]-2)**98, 0], [0, 9900*(x[1]-3)**98]])

x=np.array([3,4])
k=0
print("%d, (%1.3e, %1.3e), (%1.3e, %1.3e)"%(k, x[0], x[1], f(x), np.linalg.norm(grad(x))))
while(np.linalg.norm(grad(x))>1e-6):
    x=x-np.linalg.inv(H(x))@grad(x)
    k+=1
    print("%d, (%1.3e, %1.3e), (%1.3e, %1.3e)"%(k, x[0], x[1], f(x), np.linalg.norm(grad(x))))

```

0, (3.000e+00, 4.000e+00), 2.000e+00, 1.414e+02
1, (2.825e+00, 3.825e+00), 7.240e-01, 3.270e+01

What I want to mention to you is that somehow if the order of the function increases—so instead of 2, 3, or 4, if it increases to 10, 15, 20, or 100—the number of steps it takes is usually on the higher side. It is not just a problem with Newton's method. If you want me to do this for, say, Fletcher-Reeves, I will just use this same code and just change $f(x)$ and $\text{grad } f(x)$, and I will also change the starting point to (3, 4). Let me do it with the power of 4 to show you what is happening. You can see that Fletcher-Reeves actually takes 2649 iterations just to compute the answer for a quartic equation.

(Refer Slide Time 12:03)

```
19, (2.825e+00, 3.825e+00), 8.389e-09, 7.194e-07

import numpy as np
def f(x):
    return (x[0]-2)**4+(x[1]-3)**4

def grad(x):
    return np.array([4*(x[0]-2)**3, 4*(x[1]-3)**3])

x=np.array([3,4])
d=grad(x)
k,rho,c1=0,0.8,0.75
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=1#grad(x).dot(grad(x))/grad(x).dot(H*grad(x)) #np.dot(x,y)=x.dot(y)
    while(f(x+alpha*d)-f(x)>c1*alpha*grad(x).dot(d)):
        alpha*=rho #alpha=alpha*rho
    x_old=x
    x=x+alpha*d
    d=grad(x)+(k%2!=0)*(grad(x).dot(grad(x))/(grad(x_old).dot(grad(x_old)))*d
    k+=1 #k=k+1
    print("%d, (%1.3e,%1.3e), %1.3e"%(k,x[0],x[1],np.linalg.norm(grad(x))))

2592, (2.006e+00, 3.006e+00), 1.033e-06
2593, (2.006e+00, 3.006e+00), 1.032e-06
2594, (2.006e+00, 3.006e+00), 1.032e-06
2595, (2.006e+00, 3.006e+00), 1.031e-06
```

```
2625, (2.006e+00, 3.006e+00), 1.014e-06
2626, (2.006e+00, 3.006e+00), 1.013e-06
2627, (2.006e+00, 3.006e+00), 1.012e-06
2628, (2.006e+00, 3.006e+00), 1.012e-06
2629, (2.006e+00, 3.006e+00), 1.011e-06
2630, (2.006e+00, 3.006e+00), 1.011e-06
2631, (2.006e+00, 3.006e+00), 1.010e-06
2632, (2.006e+00, 3.006e+00), 1.010e-06
2633, (2.006e+00, 3.006e+00), 1.009e-06
2634, (2.006e+00, 3.006e+00), 1.009e-06
2635, (2.006e+00, 3.006e+00), 1.008e-06
2636, (2.006e+00, 3.006e+00), 1.007e-06
2637, (2.006e+00, 3.006e+00), 1.007e-06
2638, (2.006e+00, 3.006e+00), 1.006e-06
2639, (2.006e+00, 3.006e+00), 1.005e-06
2640, (2.006e+00, 3.006e+00), 1.005e-06
2641, (2.006e+00, 3.006e+00), 1.004e-06
2642, (2.006e+00, 3.006e+00), 1.004e-06
2643, (2.006e+00, 3.006e+00), 1.003e-06
2644, (2.006e+00, 3.006e+00), 1.003e-06
2645, (2.006e+00, 3.006e+00), 1.002e-06
2646, (2.006e+00, 3.006e+00), 1.002e-06
2647, (2.006e+00, 3.006e+00), 1.001e-06
2648, (2.006e+00, 3.006e+00), 1.001e-06
2649, (2.006e+00, 3.006e+00), 9.998e-07
```

Similarly, if I change it to 6, it has taken 9701 iterations.

So, what is the moral of the story? It is that when you have a higher-order function—where a higher-order function alone is not a problem, but when you have multiple roots at the same point, like for example, $(x - 1)^3 (x - 7)^{10}$ —what does it mean? It means that there are 10 roots at the point 7 itself. When that happens, the algorithms that we are working on actually slow down.

What is the reason? I will explain it both analytically and graphically. To explain it analytically, let me take a simpler example. Let

$$f(x) = (x - a)^n,$$

where n is a positive integer. If you look at Newton's method, you start with some $x^{(0)}$. You will have

$$x^{(1)} = x^{(0)} - (f'(x^{(0)}))^{-1} f(x^{(0)}).$$

We can compute both of these very easily.

$$f(x) = n(x - a)^{n-1} \text{ and } f'(x) = n(n - 1)(x - a)^{n-2} \text{ at } x^{(0)}.$$

$$\text{So, } x^{(1)} = x^{(0)} - [n(n - 1)(x^{(0)} - a)^{n-2}]^{-1} * [n(x^{(0)} - a)^{n-1}].$$

$$\text{That is equal to } x^{(0)} - (x^{(0)} - a)/(n - 1).$$

$$\text{To be precise, I can simplify this as } ((n - 2)/(n - 1)) x^{(0)} + a/(n - 1).$$

Let us compute $x^{(2)}$. The process would be the same; you can verify that.

$$\text{It would be } ((n - 2)/(n - 1)) x^{(1)} + a/(n - 1).$$

If you write this again, you will have

$$x^{(2)} = ((n - 2)/(n - 1))^2 x^{(0)} + (a/(n - 1)) [((n - 2)/(n - 1)) + 1].$$

(Refer Slide Time 16:38)

Modified Newton's method:

- * Initialize x^0 , tol, $k=0$
- * while ($\|g^k\| > \text{tol}$):
 - * $\lambda^k = \begin{cases} \frac{1}{2} - \min(\lambda_i^k) \end{cases}$, where $\lambda_i^k = \text{eig}(H^k)$, if $\min \lambda_i^k \leq 0$, else
 - * $x^{k+1} = x^k - (H^k + \lambda^k I)^{-1} d^k$
 - * $k = k+1$
- * Output $x^* = x^k$.

Some more Properties:

Let $f(x_1, x_2) = (x_1 - 2)^4 + (x_2 - 3)^4$, $x^0 = (3, 4)$

Let $f(x) = (x - a)^n$ where $n \in \mathbb{Z}_+$

$$x^1 = x^0 - \frac{1}{f'(x^0)} f'(x^0) = x^0 - \frac{x(x-a)^{n-1}}{x(n-1)(x-a)^{n-2}} = x^0 - \frac{x-a}{n-1}$$

$$= \left(\frac{n-2}{n-1}\right)x^0 + \frac{a}{n-1}$$

$$x^2 = \left(\frac{n-2}{n-1}\right)x^1 + \frac{a}{n-1} = \left(\frac{n-2}{n-1}\right)^2 x^0 + \frac{a}{n-1} \left[\frac{n-2}{n-1} + 1 \right]$$

Now, $x^{(3)} = ((n-2)/(n-1)) x^{(2)} + a/(n-1)$,

which is $((n-2)/(n-1))^3 x^{(0)} + (a/(n-1)) [((n-2)/(n-1))^2 + ((n-2)/(n-1)) + 1]$.

In general,

$$x^{(k)} = ((n-2)/(n-1))^k x^{(0)} + (a/(n-1)) \sum_{j=0}^{k-1} ((n-2)/(n-1))^j.$$

You can see that this is a geometric progression.

The sum of the geometric progression will be

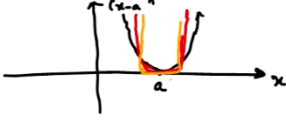
$$((n-2)/(n-1))^k x^{(0)} + (a/(n-1)) * [1 - ((n-2)/(n-1))^k] / [1 - ((n-2)/(n-1))].$$

The denominator $[1 - ((n-2)/(n-1))] = 1/(n-1)$.

This and the $1/(n-1)$ cancel, so you have

$$x^{(k)} = ((n-2)/(n-1))^k x^{(0)} + a [1 - ((n-2)/(n-1))^k].$$

(Refer Slide Time 22:11)

$$\begin{aligned}
 x^3 &= \left(\frac{n-2}{n-1}\right)x^2 + \frac{a}{n-1} = \left(\frac{n-2}{n-1}\right)^3 x^0 + \frac{a}{n-1} \left[\left(\frac{n-2}{n-1}\right)^2 + \left(\frac{n-2}{n-1}\right) + 1 \right] \\
 \vdots \\
 x^k &= \left(\frac{n-2}{n-1}\right)^k x^0 + \frac{a}{n-1} \left[\left(\frac{n-2}{n-1}\right)^{k-1} + \dots + 1 \right] = \left(\frac{n-2}{n-1}\right)^k x^0 + \frac{a}{n-1} \left[\frac{1 - \left(\frac{n-2}{n-1}\right)^k}{1 - \frac{n-2}{n-1}} \right] \\
 &= \left(\frac{n-2}{n-1}\right)^k x^0 + a \left(1 - \left(\frac{n-2}{n-1}\right)^k \right).
 \end{aligned}$$


You can see that as k tends to infinity, this term goes to 0 because the ratio is less than 1, and this term also goes to 0, so you converge to a , which is the right answer. But then what is the problem? The problem is the convergence to a . The first step is $a[1 - (n-2)/(n-1)]$.

The next step is $a[1 - ((n-2)/(n-1))^2]$. The third step is $a[1 - ((n-2)/(n-1))^3]$, and so on.

What happens when n becomes large? When n becomes 10 or 100, the movement towards a happens very, very slowly.

It is like $(1-x)^k$, where if x is low, the rate of increase is very, very slow. That is the reason; when you increase the value of n , the convergence to a gets slower and slower. This is a phenomenon that occurs whenever you have multiple roots at a particular point.

This is a similar issue to what we saw in gradient descent, where if the eigenvalues of the Hessian matrix are far apart, the convergence becomes slow. In a similar way, the convergence becomes slow in Newton's method and other methods when you have multiple roots at a particular point.

I will also explain this graphically. Consider this curve. Let us say it has multiple roots at the point a . Let us say this is $(x-a)^n$.

Suppose n is equal to 2. You will have a curve which looks like a parabola. Suppose n is equal to 4; the curve will look flatter at the bottom. Suppose if you have n equal to 10 or something, here it will be even flatter. As n increases, you can see that around a , the curve actually becomes very flat. It gets flatter and flatter. If the curve is very flat, then the inverse of the Hessian gets problematic. Because that is like having a second-order derivative that is close to 0, so H^{-1} suffers from this issue. The issue is pronounced as n becomes higher and higher.

So, as I said, this is an issue with every method, including Newton's method. But this is just a phenomenon that when you have multiple roots at a particular point, it is good that you detect that particular thing rather than attempting to correct it with Newton's method or any other method.

I wanted to indicate this behaviour before closing the discussion on Newton's method. Before ending this lecture, I will give an overview of what we have learned this week. We started with the general form of Newton's algorithm, which is to choose d_k to be $-H_k^{-1}\nabla f_k$ and α_k to be 1.

It had many good properties: for quadratic functions, it just converged in one step, and for other functions, it converges with an order of convergence equal to 2, that is, quadratic order of convergence, while the earlier methods had linear order of convergence. But it also suffered from quite a few issues, like a lack of global convergence, the direction we take might not be a descent direction, matrix inversion is computationally expensive, and we could end up with a singular H_k ; we saw quite a few examples of that.

We also showed that this Newton's method is locally convergent and the order of convergence is quadratic; that is the proof. To get around these issues—for getting around the lack of global convergence property—we use the damped Newton's method, which is just choosing α_k using backtracking line search. If H_k is not positive definite, we get around it by using the modified Newton's method, where instead of $-H_k^{-1}\nabla f_k$ you use $-(H_k + \lambda_k I)^{-1}\nabla f_k$, where λ_k was found from the eigenvalues of H_k , the minimum eigenvalue of H_k to be precise.

That got around the issues of a singular H_k and $-H_k^{-1}\nabla f_k$ not being a descent direction.

Finally, we have also discussed a property when $f(x)$ has multiple roots at a particular point. We saw examples where the convergence is slower, and that occurs basically because of the flat nature of the curve around a that causes such problems when you have multiple roots at a, and it gets pronounced when the number of roots is higher.

As a final comment, please recall that we are still stuck with one issue, which is the inversion of H at each step. This is a computationally expensive process that has to be done in every step, which is a significant drawback. How do we get around it? We get around it using quasi-Newton methods, which is what we are going to see in the next lecture.

Thank you.