Distributed Optimization and Machine Learning

Prof. Mayank Baranwal

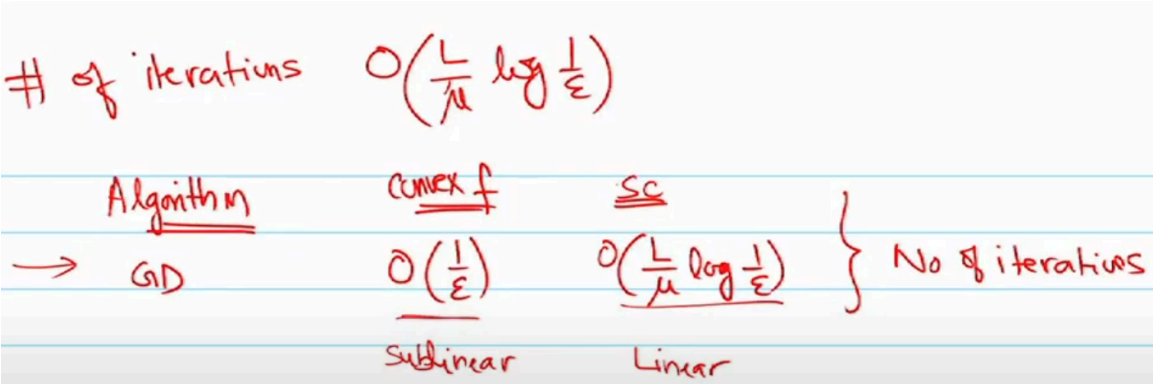
Computer Science & Engineering, Electrical Engineering, Mathematics

Indian Institute of Technology Bombay

Week-4

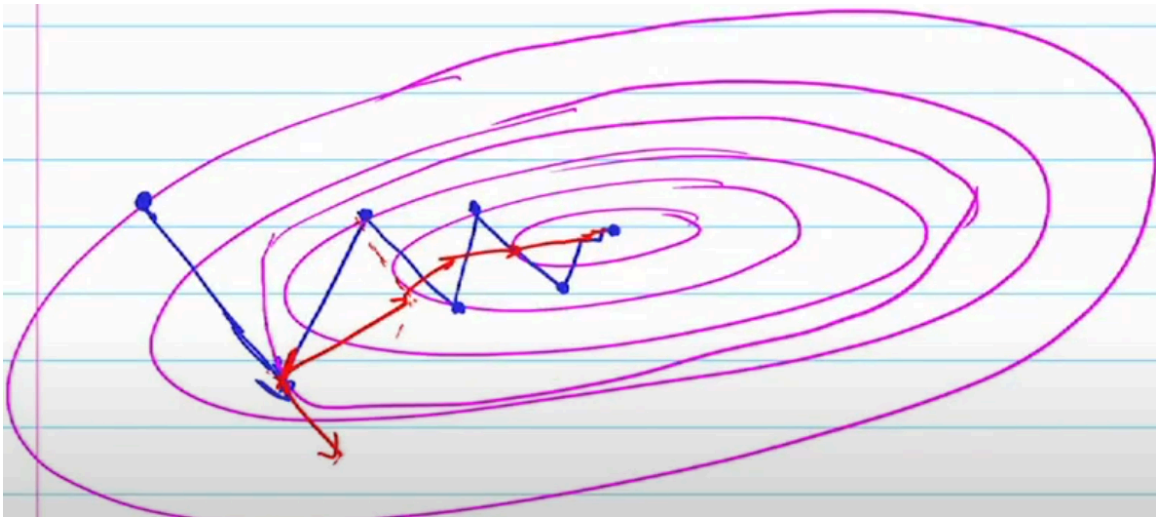
Lecture-13: Accelerate the convergence even further

Now, the question is can you accelerate this even further ok. So, can you accelerate like can you can you find a better rate than l over mu log 1 over epsilon. What did we discuss other than gradient descent? What kind of algorithms did we like sort of qualitatively discuss in the last lecture? So, remember we were looking at. So, what is one shortcoming of gradient descent? So, one of the shortcomings of gradient descent is. So, let us say we start somewhere over here and gradient descent points us in a specific direction which is normal to this level curve right. So, that is let us say it is going to be somewhere over here and let me draw the level curve here as well.



something like this. Now, what is the gradient in this direction? May be the gradient in this direction then will be pointing somewhere over here and then it may be pointing somewhere over here, over here, over here and so on until it converges to the optimal value or optimal solution. So, what is one shortcoming do you see with this kind of approach? The path to convergence is very zigzaggy right and the idea is can we improve on this and this may tell us why it would require more iterations because you are oscillating a lot around the optimal solution instead of doing. So, if you can have a smoother convergence maybe it would require fewer iterations right and that is the intuition behind maybe some other algorithm may do better than simple gradient descent and the idea is.

to use something called momentum. So, momentum as I mean as the name suggests, you are still maintaining on the momentum that you had in the previous iteration. So, in this in the in case of gradient descent what happens? We completely sort of discard what happened in the previous iteration right and that basically that means you kill all the momentum that you had in a specific direction which was decreasing the function value, you completely discard it and then we start it fresh. Instead of doing so, if we let us say use a little bit of momentum as well as the gradient in the new direction. Now, if I look at the resultant direction of movement from here it will then be here and then here and then here and so on.



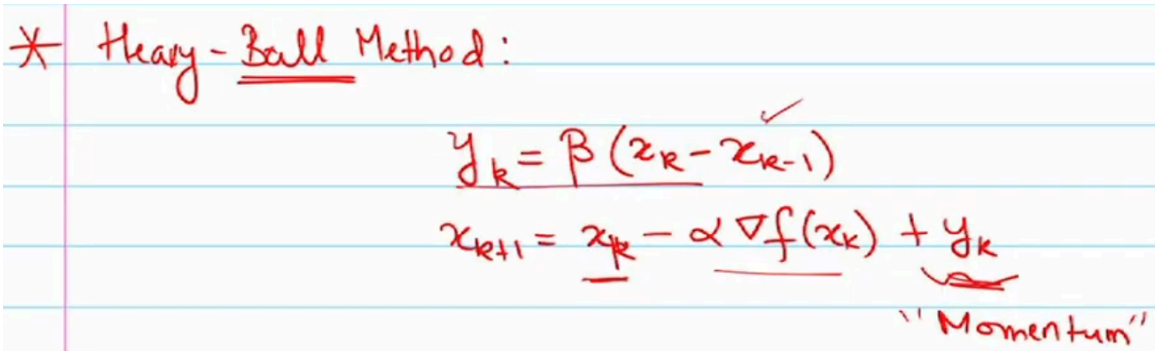
So, this would be a much nicer way to converge to the like a smoother way to converge to the optimal solution just by simply using the momentum that you had used that you had in the previous iteration right. and that basically gives us gives rise to two like accelerated methods. So, the question is further and the answer is yes and basically by using momentum right. So, if you want to use momentum, I mean, so there is no free lunch right, like if you want to use momentum, then what is the shortcoming of this momentum based approach? Restoring the previous direction as well right. So, that is one shortcoming of momentum based method, but if convergence is the priority and then it may make sense to use momentum right.

So, there are two momentum based algorithms that are very popular, one is the Polyak's Heavy Ball method which we going to denote this by H B which is heavy ball and the other is Nesterov's accelerated gradient method ok. I think this was in some 1964 if I am not wrong and this was in 80s and both are sort of very popularly like very popular so momentum based algorithms. So, both these algorithms they use momentum in different ways. So, in this case for instance gradient at this point is calculated entirely using where you arrived right. So, this is the new value of x k and you compute the gradient at this point.

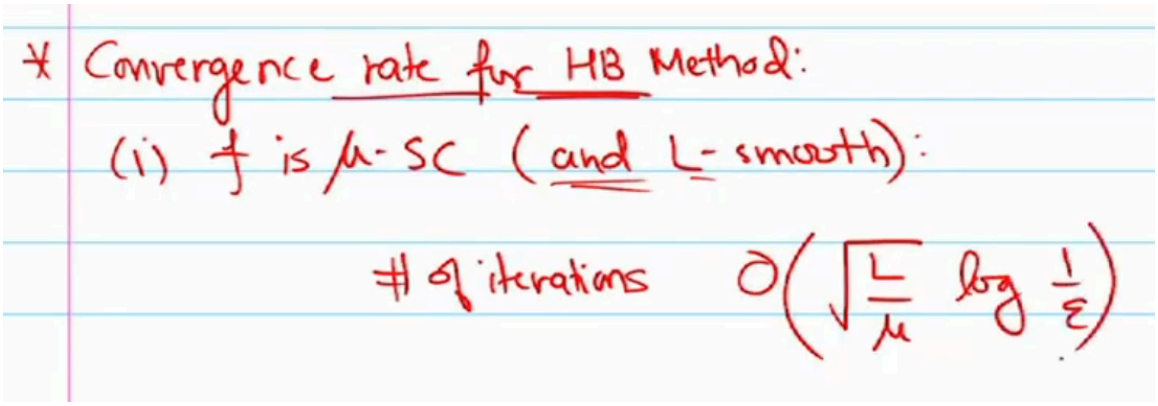
you take the momentum in the previous step and then you move in the resultant direction. Whereas, in Nesterov's accelerated gradient method you first compute the resultant direction and then on the resultant direction you actually then compute the gradient on the not on this point but let us say the resultant direction points you to some other direction some other some other x and on that x you compute the gradient. So, you first come have the like you compute the resultant vector and then compute the gradient here you compute the gradient and then take the combination of the gradient and the previous momentum to move. So, that is the difference I think it will be become clearer once we write the equation. So, let me start with heavy ball method.

So, in heavy ball method we define an additional term yk which is the difference between current xk and the previous xk and then xk plus 1. So, this is nothing but the momentum term right. So, you compute the momentum based on your current x k and the previous x k, you know which direction you had moved in the previous iteration. So, you take that direction. So, you take this direction, let us say you have moved in this direction.

So, you take this direction, you take the gradient direction and then you move in the resultant direction, resultant of this gradient term and this particular term and that is the heavy ball method. So, this is your momentum term. So, really you need to store an additional x k minus 1 here right and that is the difference because earlier when we were doing gradient descent we just needed to store x k and we were supposed to evaluate gradient on x k whereas, now we also need to store x k minus 1. So, it is a second order method unlike the previous gradient descent which was the first order method ok. So, let us look at the convergence rate.



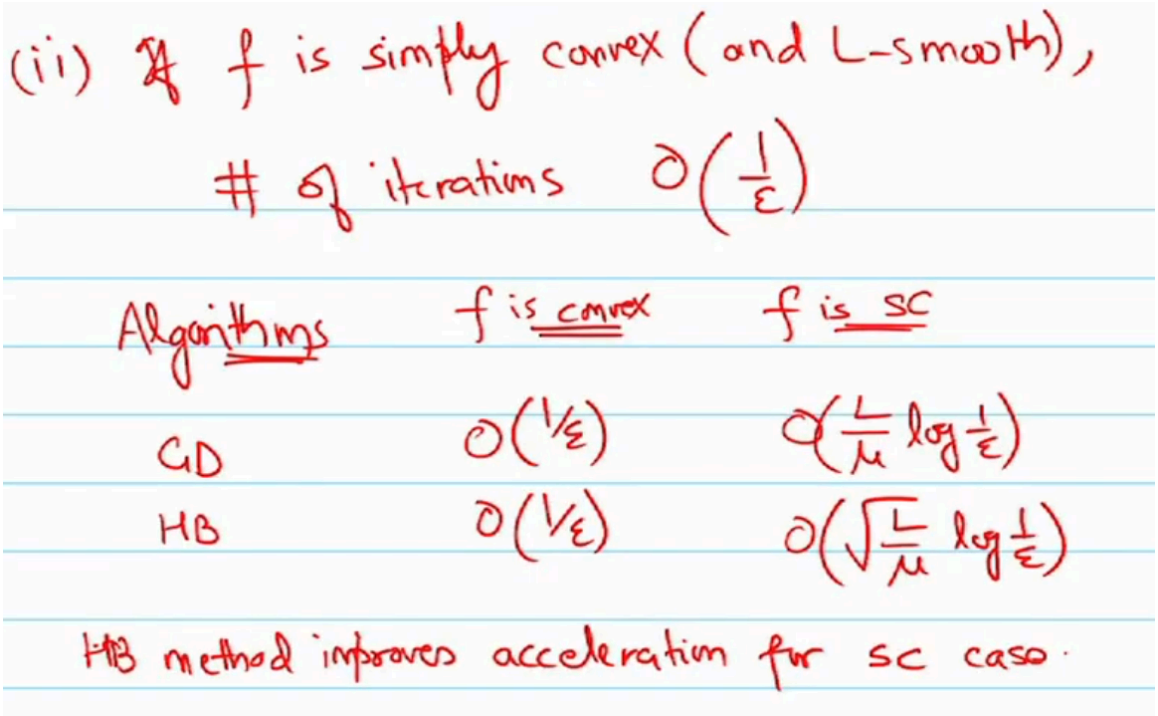
So, if f so there are two cases just like in the previous gradient descent case we considered f to be strongly convex and just simply convex. So, if f is mu strongly convex and of course, we assume that f is else smooth. So, as I said if you are working with discrete time algorithms you always assume that the f is like function is L smooth otherwise it would be nearly impossible to derive any convergence result. So, what do we mean by like a function is L smooth? That means your gradients are like gradients are Lipschitz right. So, that means they do not change arbitrary as you move along.



So, that means you can bound your like every like every time you jump from x k to x k plus 1, you can bound the number of like basically by the amount that you have basically changed right. For discrete time algorithms it is always I mean L smoothness is kind of implicit in order to show any convergence guarantees. And the additional assumption that we can make is either whether it is strongly convex or convex. So, in this case number of iterations this is. So, instead of L over mu now you get order square root L over mu log 1 over epsilon.

So, in gradient descent what was the number of iterations that was order l over mu log 1 over epsilon and now you get square root l over mu ok. However, if you assume that if f is simply convex that is and also you assume that f is L smooth then number of iterations it turns out to be the same order 1 over epsilon. So, for L smooth for convex case heavy ball method is no better than gradient descent, but for strongly convex case heavy ball method is better than is faster than the gradient descent. So, that is the bottom line of it ok. So, had it not been the case then I mean people would have simply worked with heavy ball method.

But the fact that accelerated gradient method was there that means this new method is may not be significantly better than the gradient descent right and that is why there was a need to develop an accelerated gradient method. But to summarize if I again now look at the algorithms. So, one was gradient descent and the other is heavy ball and you have f is convex. and f is strongly convex. So, in this case you get in both cases the number of iterations required is order 1 over epsilon for a strongly convex case and it basically improves right.



proves acceleration for strongly convex scales ok. So, we are not going to be I mean the analysis is very similar just as we had for gradient descent. So, I am not going to be doing the proof, I will just share the proof in the like over teams. So, you can just take a look at it ok. Now, the question is So, while I mean it is it is a good setting to have over here I mean we do not make any progress over here right.

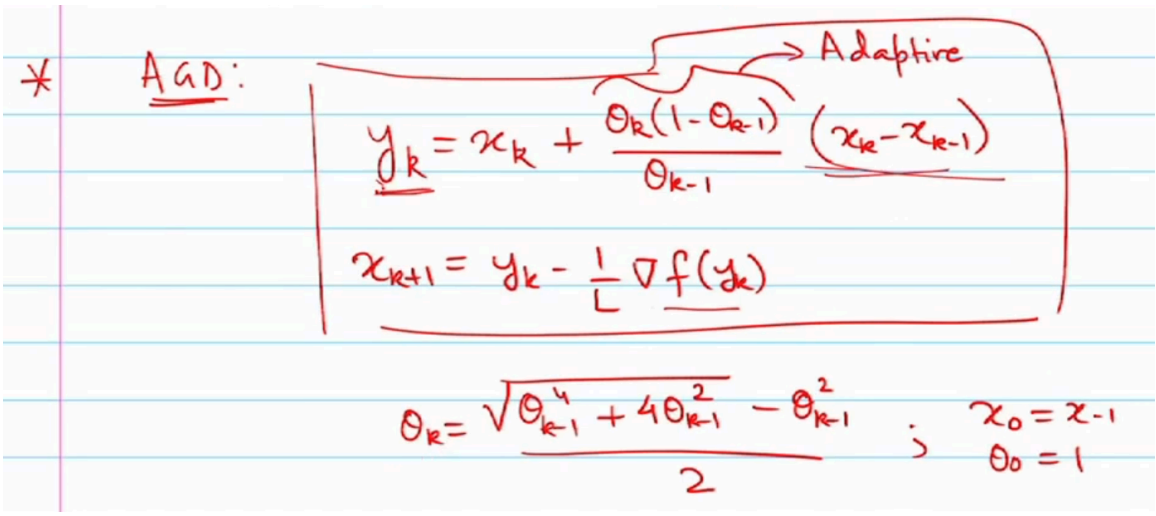
So, the question is can we can we improve this even further and that is when the accelerated gradient method sort of plays a role. And as I said there are two types of momentum that you can use. One is first you compute the momentum vector, you compute the gradient separately and then take the resultant like your final sort of descent direction is basically a combination of the gradient vector and the momentum vector. The other way out is you first compute your new sort of pseudo x k and on that new x k you compute the gradient on that value right and then you basically do a simple gradient descent. So, there are two ways to go about it.

So, this is not simple gradient descent because to gradient descent you are adding momentum whereas you can do a simple gradient descent on this particular value. So, that is the difference. So, the update rule is somewhat more involved. So, accelerated gradient method you compute the momentum vector So, this in some sense this is like an adaptive sort of learning rate for the momentum term. So, not only you are computing this you are using this momentum x k minus x k minus 1.

So, that means you are using the momentum from the previous step to find a new pseudo

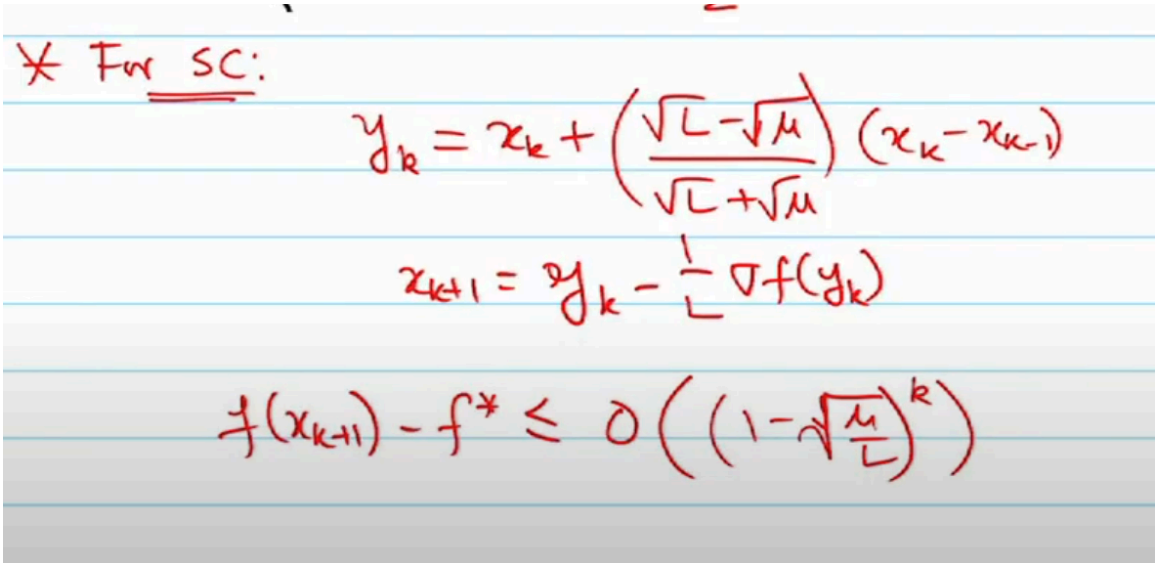
point where you are going to be computing the gradient and then using the and then you will do the gradient descent on that particular point. So, you will compute the gradient on this new f y k and then you will just update your x k plus 1 as a gradient descent as if you are I mean that is that was your x k to start with. So, the momentum is kind of built into this. So, that is the difference between how these two algorithms are different and so, let me first write this term. So, this is the accelerated gradient method and what is theta k? So, theta k is basically defined to be again the proof of this particular result is more involved.

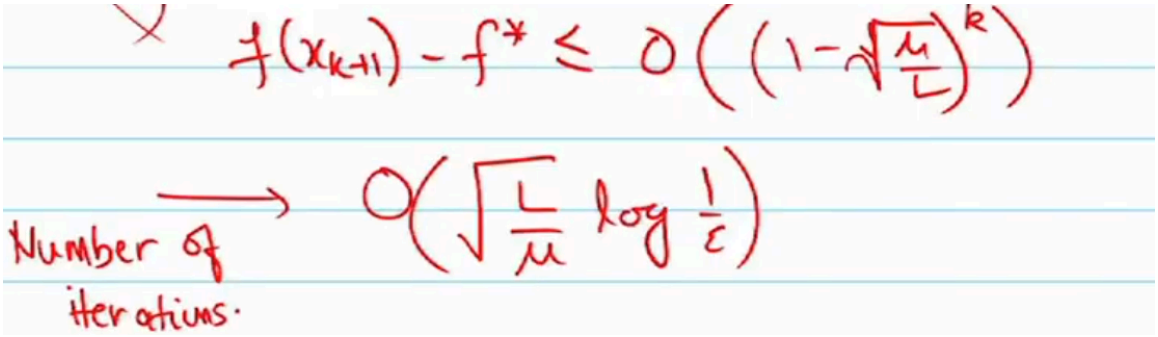
So, I will I will not do this in class and share it separately, but this is what ok. So, this is so, that is how you update your theta k ok and if you use this particular algorithm for strongly convex case. So, whether for strongly convex case this algorithm acts as. So, this is the result that you are going to be deriving in your homework. This is for I mean if I make it strongly convex and you would see that is a natural way to derive this particular result.



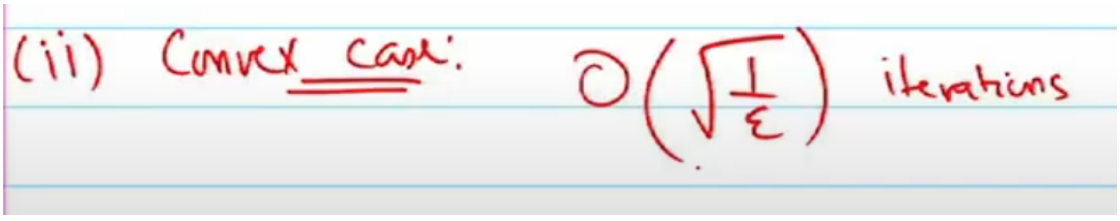
So, if you know l and mu this one is y k and you can show that with this kind of update rule f of x k plus 1 minus f star that is you get the same square root mu over L ok. So,

first result is for For just for no, for strongly convex function. For strongly convex.





For strongly convex. Yeah. No, so this is for the strongly convex case, this is what the update rule would look like for strongly convex case. So, in terms of rate of convergence, this would be I mean you can show that number of iterations required, this again turns out to be order log 1 over epsilon. So, this is the number of iterations. which is same as the heavy ball method for strongly convex case. But if I consider the convex case, then this basically you get order square root 1 over epsilon iterations.



So, if I now look at these like if I compare the different algorithms. So, when f is convex simply convex heavy ball gives you the same rate as these gradient descent. Whereas, this particular accelerated gradient method this gives you order of square root 1 over epsilon. So, that means even fewer number of iterations, but for the when f is strongly convex the rate like basically the number of iterations required are same are in the same order as your heavy ball method. So, if I were to summarize all these different algorithms, sorry

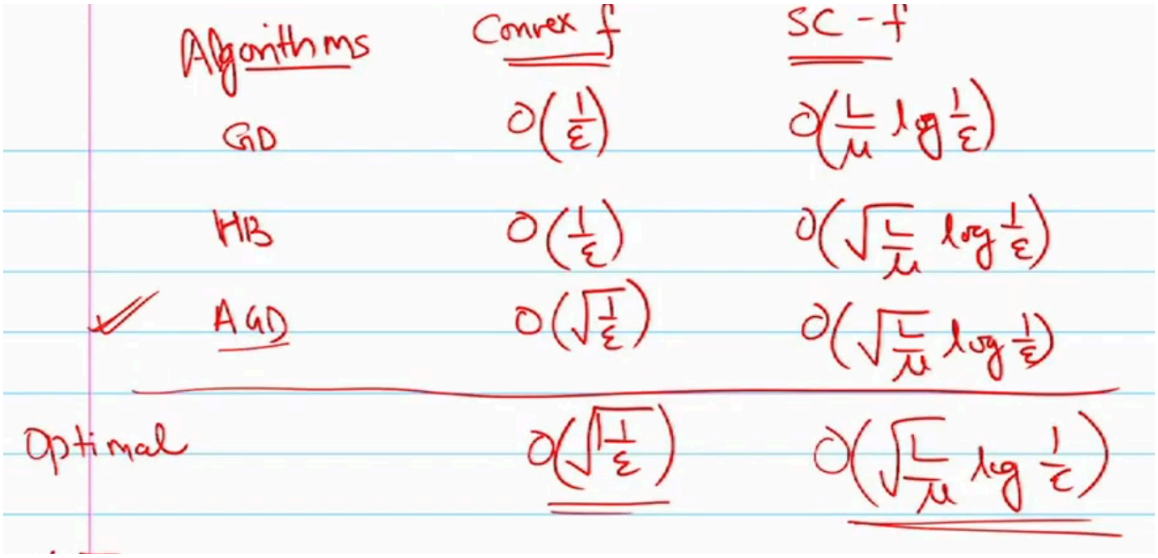
because for evaluating y 1 or y naught you would be.

So, the starting value y naught would be dependent on x naught and x naught minus 1 right x naught and x minus 1. So, you have gradient descent. So, in the convex case you get number of iterations to be order 1 over epsilon, strongly convex it is L over mu log 1 over epsilon, heavy wall. You do not improve for the convex if, but you do improve for the strongly convex case. And when I look at the accelerated gradient this you improve further.

So, you make it square root 1 over epsilon number of iterations and this is in the same order as a b ball method. Another question is which one is optimal like I mean. So, different algorithms have different rates and obviously, this seems to be a better choice than any other algorithm in this case, but what can we continue to do right like I mean maybe we can improve this even further. But then not only Nestorov came up with this accelerated gradient method, he also showed that you cannot improve any further. In fact, he designed a class of optimization problems for strongly convex case and he showed that no matter what kind of algorithm you choose, if it is a gradient simple, gradient based algorithm, you would require at least these many iterations.

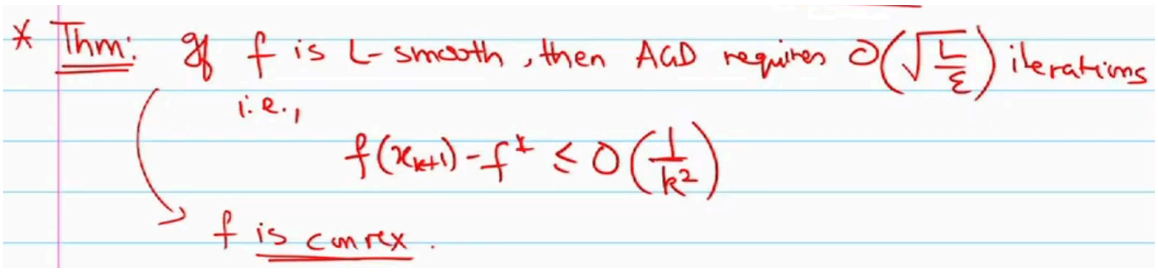
So, he came up with a family of optimization problems for which you cannot improve any better than this. So, not only this is fastest among all three, this is also optimal. So, that means both these rates that are obtained by accelerated gradient method or both these number of iterations that are obtained by accelerated gradient methods, these are also optimal. So, you cannot improve any further. And he basically came up with a class of problems where you would require at least these many and that is one of his very celebrated contribution other than also coming up with this algorithm.

You can I mean he has a very nice book on optimization by the way I will also share the link to that particular text. So, you can read up there on that all right. So, just to summarize. So, with accelerated gradient method.



So, let me write down the theorem. And this is from where you get the order 1 over square root epsilon kind of I mean in some text you would see that it is written as L over epsilon as you see that there is also an L term even in the other theorems right. So, either you can write this in terms of L over epsilon or I mean 1 over epsilon it is the same thing. So, I can in depending on which book you are following insert in some text you would see written I mean it written because it is in the same order.

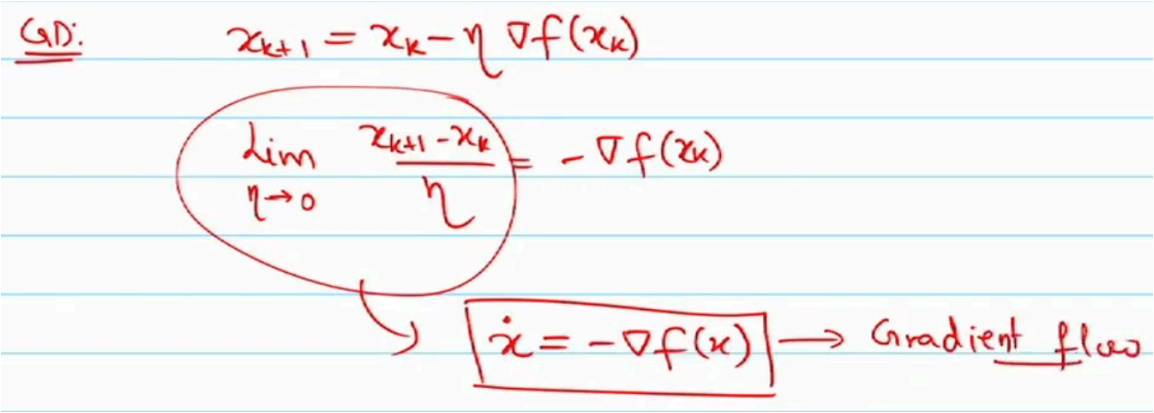
So, it does not matter. So, f is convex yeah. So, I mean again this I mean in this course we are going to be assuming f is convex unless we specify otherwise. So, the only case where it is not convex is when it is an convex function, but still satisfies PL inequality. So, that would be the only setting where we do not assume f to be convex. So, yeah just to be more precise f is convex as well. So, this is the number of iterations required for different algorithms ok.



So, this pretty much sums up our discussion on discrete time algorithms. And we are going to revisit discrete time algorithms in the context of distributed optimization, but this is all we wanted to cover on simple discrete time convex optimization. So, in the subsequent lectures we will take a little departure from how we sort of look at these algorithms and we are going to relate this analysis of optimization algorithms from continuous time dynamical systems point of view. And the idea is let us say I consider a simple algorithm like this which is gradient descent and as I had indicated in previous

lectures as well. So, this is a typical construct of gradient descent and in the limiting case when eta or this step size is almost 0 ok.

So, this term what is this term becomes? So, it is almost like x dot right. So, we could get something like. So, the limiting behavior of gradient descent it follows something called gradient flow And it is much easier to analyze algorithm for this form. So, as some of you who I mean who may have taken courses on in control theory or stability theory, let us say f of x is half x square. What is gradient of f? Just x, right? So, gradient of f is x and the dynamical system that you end up with is x dot is minus x, right? And this we know is exponentially stable.

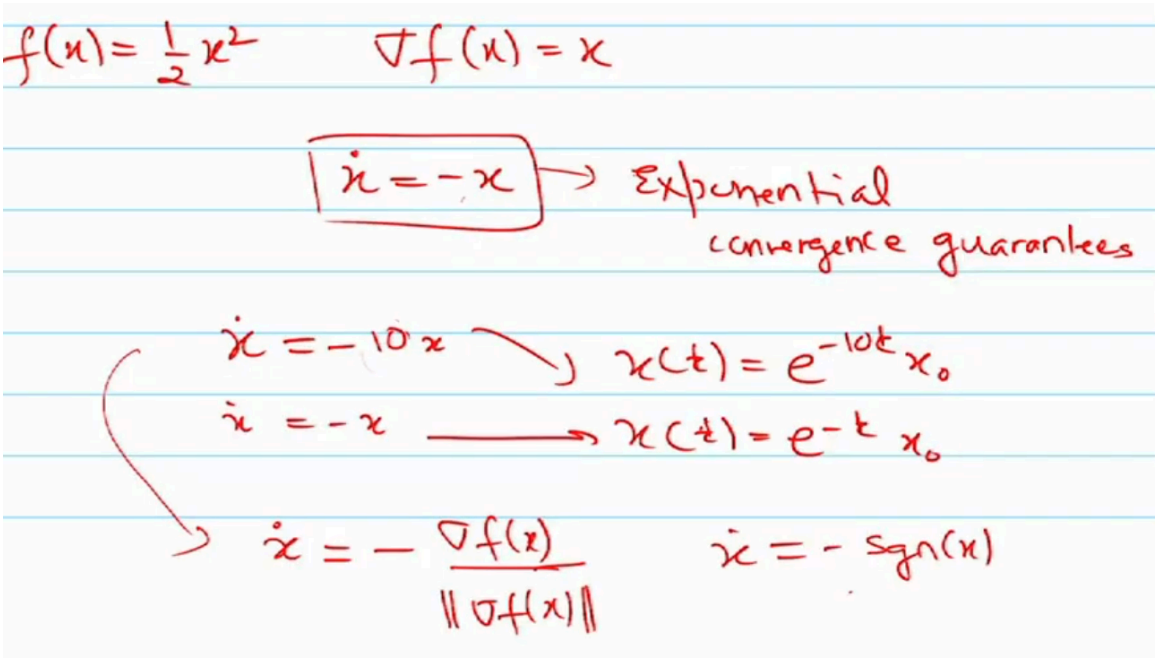


The equilibrium 0 is exponentially stable. So, you can immediately provide exponential convergence guarantees right, for this I mean using this continuous time analog. What is one caveat that you should keep in mind when you look at continuous time dynamical system? And that is that has to do with the fact Let us say I use, I run these two dynamical system, one is x dot is minus 10 x and x dot is minus x, ok. So, which one do you think is faster? The first one, right? If this one goes like x t is e to the negative 10 t times x naught and in this case x t is e to the t times x naught. So, if I just arbitrarily increase the like maybe add a superficially large coefficient in front, we can make it converge even faster right. But you know from your I mean from typical like experience that if you try and implement simple gradient descent with a very large step size, it is unstable.

So, the kind of instabilities that you see in discrete setting, you would not see in the continuous time setting. And while it is important from the point of view of analysis, it is not really useful from the point of view of implementation. So, but then what do we achieve? We can leverage a lot of tools that have been developed for continuous time dynamical systems. And we can try and at least come up with different maybe more sort of creative ways to come up with new algorithms, which may have like which may have nice empirical performance when implemented in discrete time. Whereas, one caveat that I said already is you can arbitrarily make continuous time dynamical system converge

faster, but that is not the case with discrete system.

So, you would also see in due course of time that one of the requirements for instance when deriving rate of convergence for discrete time algorithms is that we needed f to be not just convex, but also else smooth. Here we I mean in fact for continuous time setting because we can make these dynamical systems arbitrarily fast, we do not need to make such kind of assumption. So, that is these are like sort of two different things that we are going to be looking at. Other ways to for instance accelerate this dynamical system is what if I instead of using this let us say this is my dynamical system and instead of using this I divide this by the norm of this right. So, in continuous time case what does this become? So, this almost x lies x dot is minus sgn of x ok.



So, if you are on the positive side you will get a. So, no matter how close you are to the optimal solution or how close or how far you always have a definitive plus or minus 1 kind of gradients right. And that means you can converge much faster even though you would oscillate a lot once you close once you are closer to the optimal solution you would oscillate a lot, but then you can I mean convert arbitrarily fast if you are close to the optimal solution. So, as I said you can accelerate things even faster, but then these kind of algorithms these kind of creative ways to design algorithms which you can also analyze using Lyapunov theory that basically gives us ideas about how to design equivalent algorithms in discrete time setting with the hope that eventually it will also translate the kind of performance guarantees that you have in continuous time dynamical system those will also translate to the discrete case. Thank you very much.