ConvexOptimizationII-Lecture13

Instructor (Stephen Boyd):Great, I guess this means we've started. So today, we'll continue with the conjugate gradient stuff. So last time – let me just review sort of where we were. It was actually yesterday, but logic, I mean, logically – in fact. But we can pretend it's five days or whatever it would be.

So we're looking at solving symmetric positive definite systems of equations and this would come up in Newton's method, it comes up in, you know, interior point methods, least squares, all these sorts of things. And last time we talked about, I mean, the CG Method the basic idea is it's a method which solves Ax=b where A is positive definite. And – but it does so in a different way.

The ways you've seen before are factor solve methods. In fact, in those methods what you need is you actually need the matrix. So you actually – you pass a pointed to an array of numbers, roughly. And then you work on that.

What's extremely interested about the CG method is actually the way A is described is completely different. You do not give a set of numbers. In fact, in most of the interesting applications of CG you will never form or store the matrix A, ever, because, in fact, in most cases it's huge. It's some vast thing. That's kind of the point. Instead, what you need is simply a method for calculating A times a vector.

So what you really are gonna pass into a CG method is a pointer to a function that evaluates this thing. How it evaluates it is it's business, it's none of your business. That's sort of how – that's the idea.

Okay, well, last time we started looking at a little bit about this. We looked at two different measures of the error. So one measure is this number tao and it's the amount of decrease, F is that quadratic function you're minimizing, you've achieved divided by – sorry, this is the amount of the decrease, the sub optimality and decrease divided by the original sub optimality of decrease.

That's probably what you're interested in. But another one which is probably in many applications what's actually easier to get a handle on and all that sort of stuff is the residual. And the residual is nothing but, you know, b-Ax. It's basically how far are you from solving Ax-b.

Now the CG method, and actually many others, actually work with the idea of a Krylov subspace. And this is just to sort of rapidly review what we did last time.

The Krylov sequence of a set of vectors is defined this way. It's actually – you take this Krylov subspace and that's the stand of b/ab up to some Ak-1b. And that essentially means it's all vectors that can be written this way. It's B times a polynomial of A and that polynomials of degree K minus one. That's a subspace of dimension, oh, it could be K but it actually can be less. Actually, if it’s less than K then it means it you’ve solved the problem.

Okay, so XK is the minimum of this – this is the function you’re minimizing, this quadratic function on the Krylov subspace, it’s the minimizer of that. And the CG algorithm and several others generate the Krylov sequence. That’s actually the important part.

Now, in the Krylov – along the Krylov sequence obviously this quadratic function that you’re minimizing decreases. That’s obvious because, in fact, you’re minimizing over a bigger and bigger subspace in each step and that can’t get worse. Now the residual can actually increase, that’s not monotone.

Now it turns out if you run N steps you get X star, that follows from Cayley-Hamilton theorem. And the Kth iterate of the Krylov sequence is actually a polynomial of degree K-1 or less multiplied by B. Now the interesting part and the reason why these methods actually work really well – although the – by the way, there’s plenty of cases where you’re gonna do this for such a small number of steps that this is actually not that relevant.

There’s a two term recurrence and the two term recurrence is this, you can compute the next point in the Krylov sequence actually as a linear combination of not just the previous one but the previous one and the one before that and then there’s some coefficient here and we’ll see what the coefficients are later. Actually, they’re not that relevant. What matters is that they exist and are easily calculated.

Okay, so we’ve seen this and what I want to do now is look at the – to understand how the CG method works or how well it does. It’s extremely important to get a good intuitive feel for how it works.

When is it gonna work well? By the way, notice that you would never even say anything like that when you talk about, like, you know, direct methods. You wouldn’t say it’s good to talk about, you know, let’s say, “Well, here’s a 1000 by 1000 matrix. Oh, yeah, here’s one that’s gonna work well.” It always works well. You have positive definite matrix, you take a Cholesky factorization, it doesn’t depend on the data. I mean, to first and – to 0 and 1st order does not depend on the data.

So okay. With these though you need to know when is it gonna work well because that’s actually the key to all of these things. So the way to do this is essentially to diagonalize the system. Then when you diagonalize it’s kind of stupid because if A is diagonal and I ask you, “How do you solve Ax=b?” that’s easy. That’s just the inverse, A is diagonal.

So the solution if I diagonalize is actually just this, it’s just the transform B divided by the Ith entry in the transform A which is diagonal. This lambda thing here. So that’s very simple. But this is gonna give us an idea for when this works well. The optimal value is just this. It’s nothing but this thing. That’s that A inverse B star, A inverse B. That’s this thing here. Okay?

Okay, now the Krylov sequence in terms of Y is the same except now we can actually look very carefully at this thing because the polynomial of a matrix is really, really simple. The matrix is diagonal so a polynomial of it is simply the polynomial it’s a diagonal matrix and each entry is a polynomial of that entry in the diagonal which are the eiganvalues of A.

So you get very simple expressions, all in terms of the eiganvalues now. So it says that PK is the – one way to say it is that it’s the polynomial of degree less than K that minimizes this sum.

And notice it’s got some positive weights, none negative weights here. And then over here you can kind of see what’s going on. If you look carefully at this you really want – well, we’ll say what P should look like in a minute. P should look like, you know, one over lambda or something like that to make this work well.

Okay, so another way to write it, let’s just keep going down here is we’ll look at the second expression. The second expression says that this error is the minimum over – these are the positive weights, and then here you can see it’s lambda I times lambda IP of lambda I minus one. And in fact what that says if you can make P of lambda look like one over lambda on the eigenvalues of A you’re – in fact if you could have P of lambda I equals one over lambda I it’s done. This is zero and then that says that in fact this would have to equal F star. Okay?

So that’s the idea. And in fact, these – we saw already in the Cayley-Hamilton theorem that there’s an Nth degree polynomial which in fact we saw exactly what PN is. It has to do with a characteristic polynomial. It gives – it’s a polynomial that in all cases satisfies P. The P of lambda I equals one over lambda I and that’s why this conversion ten steps – sorry, in N steps.

Now what’s interesting here is this is gonna give us sort of the intuition about when this works well. There are lots of other ways to say it. I mean, one is to say – well, look. A polynomial – something that looks like that is a general polynomial of degree K with the value that if you plug in it’s probably if you plug in 0 this goes away, you get one.

I mean, I switched the sign on it. So that’s another way to say it is this way. There are lots of these. I won’t go into too many of the details but the important part are the conclusions. So here are the conclusions. If you can find a polynomial of degree K that starts at one that’s small on the spectrum of A then the Kth – then actually no matter what the right-hand side B is you’re F of XK minus F star is gonna be small.

And in particular this says that if the eigenvalues are clustered into groups, let’s say K groups, then YK is gonna be a really good approximate solution because if I had K clusters of eigenvalues I can take a Kth order polynomial and put it right through, let’s say, the center of those clusters or near the center of those clusters and then that polynomial would be really small on each of those clusters and we'll get a very good fit here.

Now there's another way to do well and that's to have YI star small for just a handful of things. So this says if your solution is approximate linear combination of K eigenvectors then YK is a good approximate solution. So that's another way to say it.

Notice that this statement is independent of the right-hand side and depends only on the matrix A. This one now depends on the right-hand side but doesn't depend on the clustering. It basically says if you're a linear combination of K eigenvectors then this must be a good solution.

So, okay. Now you can do things like this, this allows you – these are classical bounds. Classical bounds would be things like this; you would – suppose the only thing I told you about A was that its condition number is kappa? So I give you – let's say I can scale A. So I give you lambda min and lambda max and if I put a Chebyshev polynomial on there, that's a polynomial that's small uniformly across it on that interval, you end up with a conclusion that says this, it says this convergence measure that this thing goes down like that.

And this is actually – this allows you to – oh, by the way, a simple gradient method would have a kappa here and a kappa here. So if you just use the gradient method to minimize that function F you'd get kappa here and kappa here. And so you're supposed to say, "Wow, that's much better because you get square root here," or something. So that's the idea.

It turns out – this is interesting and that's fine but it turns out actually that where you want to use CG is where in fact, I mean, this is like many other things it's an upper bound and in fact, you usually get convergence. It's sort of much, much faster. Not to high accuracy but –

So let's do an example here. So these show you – this is the function Q. They all start at 1 here. It's a 7 by 7 matrix. Now I think it goes without saying that CG is not the method of choice for solving a 7 by 7 positive definite system. That's something – I guess the time to actually solve a 7 by 7 positive system is down in the – it's definitely sub microsecond, you know, obviously. So this is not the right way.

Nevertheless, this is sort of the – we'll just look at an example. So here are the eigenvalues of A, I don't know, it's one here, I guess it's around two, a couple down here, another cluster here and an outlier out there.

After one step of CG the optimal Q is this thing. By the way, it goes without saying that you don't ever produce these. I mean, if you want keep track of these polynomials that's fine, but you don't need to.

So this shows you that one and you can see it. It’s kind of doing its best. It’s not so small here nor is it small here and it’s pretty big there. The quadratic is this green one and you can see it kind of – it gets small near this cluster and it kind of splits this cluster and this cluster and goes near it so that you get things like this.

The cube term, I think that might be the purple one here, is the cubic term and you can see now cubic is nice because it’s sort of – you can see three clusters and it does just what you think. It goes down, goes right through this thing. It’s very small here, small here, small here, over here it’s very, you know, quite small, still small, still small, and then kind of goes down there and it’s small here. So actually you could expect that after three steps of this method you’re gonna get a very good solution.

The quartic I think is the red, maybe, I guess maybe that’s the red. I’m not sure. I guess that’s the quartic or something. And the quartic one as you can see goes through – is now actually picking off bits and pieces. It’s actually doing things like hitting both sides of it so it’s small on all three here. It’s small here and now it’s just nailing that one.

And then the seventh one is this one and that’s actually an interpolating polynomial so it’s 0 on all of them and that means that at step seven you get the exact answer. Okay? So, I mean, actually I’m anthropomorphizing this, obviously. So well, all that’s actually done is the Krylov sequence is computed. But this gives you a rough idea of how this works.

Okay, actually you’ll know shortly why it is that you need to know how well CG works because it’s gonna be your job to change coordinates to make CG work. We’ll see that in a minute.

Okay, so here’s the convergence and sure enough, you know, you start with that’s the full decrease, and you can see this sort of after five steps you’ve done very well and I guess after four you’ve done extremely well and so on. So that’s the picture. Okay, here’s the residual which in fact does go down monotonically. It didn’t have to but it did in this case. Now look at – I mean, that’s a fake example. Let’s look at a real one.

Here’s an example, it’s just a resistor network with 100,000 nodes. We just made – it’s a random graph so each node is connected to an average of ten other nodes. So, you know, some big complicated resistor network.

So, again, a million nonzeros in the matrix G. And I pick one node and I make that the ground. Okay? Then I’m gonna do the following, I’m gonna inject at each of the million nodes a uniform current, a current that’s chosen randomly uniform on 01 and the resistor values will be uniform on 01. Okay?

So that’s our – that’s the problem. It doesn’t matter. But in this case if you want to use a sparse Cholesky factorization – actually, before you ever do it you’ll know what happens because you can actually do the symbolic factorization on G and actually calculate the number of fill-ins. Okay?

So in this case it required – there was too much fill-in and I don't know – I don't know how big it would be but maybe, I don't know, 50 million, 100 million or something like that, nonzeros starting from 1 million. Okay?

So – oh, I should mention this, if the number of nonzeros goes up by a factor of a 2 you are to consider yourself lucky in a sparse matrix thing, right? And that means you must go and make an offering to the god that controls the heuristic of sparse orderings and of ordering in sparse matrix methods.

Ten, you know, that's okay. You're supposed to be happy or that's typical. You start getting to fill-in factors like 100 and stuff like that and that's because you didn't go make an offering the last time sparse matrices went your way.

So, all right. So in this case this problem you can't solve with a sparse Cholesky factorization and I actually shouldn't say that. I should say using the approximate minimum degree ordering method produces a Cholesky factor with too much fill-in.

Now, of course, on the big machine I actually could have done it and would have gotten the exact answer but it would have been really long and taken a lot of time. It might be that there's another heuristic ordering method that would work perfectly well here. I doubt it but anyway, there's lots of them.

Okay, so instead we'll use CG here. Now in this case I do form the matrix G explicitly and all I have to do at each step is I have to multiply by G. But that's just a sparse matrix vector multiply is a million nonzeros so I'm doing like a million flops per matrix vector multiply and that's a dominant effort of a CG iteration. So I don't know, how much time does that take?

Student: About a second.

Instructor (Stephen Boyd):A second. Man, we've got to work on you guys. This is not cool.

Student:[Inaudible].

Instructor (Stephen Boyd):What?

Student: Less than a millisecond?

Instructor (Stephen Boyd):What?

Student: Less than a millisecond.

Instructor (Stephen Boyd):Thank you, less than a millisecond. So around a – let's just say a millisecond and let's just get the order of magnitude right, millisecond. Okay? So matrix vector multiplies a millisecond here, roughly. Is that right? Thousandths of – yeah, sounds about right, right? Yeah, sure. Weird. Isn't that strange? Man, these computers are fast. Okay, that shocks me.

All right, so it's a millisecond, matrix vector multiply. And it might be a few milliseconds because of all sorts of, you know, issues with accessing memory and stuff like that. But if it's set up right and you're lucky it's on that order of magnitude.

Okay, so here's how CG runs and this is a residual here. So – and you can see that, well – for ten steps the residual actually goes up by a factor by 100 – generally considered not good. And then it goes back down again. But the wild thing is – the theory tell us the following, the theory says that if you run it one – what did I - was 100,000? Yeah.

The theory says if you run it 100,000, you know, the millisecond doesn't sound right to me but I'll have to think about that. I think that's – I have to do that for each one or something? Anyway, maybe it is right. It doesn't matter.

The theory tells you that K here runs out to 10 to the 5; we'll get the solution exactly. But the wild part is, is if you're not picky about super high accuracy you actually have a perfectly good solution in 50 steps.

Each step was a matrix vector multiply and if our estimate of a millisecond is right it means you just solved a very large, you know, diffusion equation, Poisson equation, whatever you want to call it. You just solved it in a quarter second. I mean, if we're right or, you know, something like a tenth of a second. Everybody see what's going on here? By the way, absolutely nothing in theory guaranteed that this would happen, absolutely nothing. Okay? It just did. And this is very common.

So – and this is kind of the cool part about CG is that in a shockingly small number of steps you often – what emerges is something that looks kind of like the solution.

In this case it doesn't look like the solution; it's actually, like, quite good. Okay? So that's the idea. So if you wanted to know at what point have you learned something new, you just did. You cannot – if you go just type this thing, you know, G\I or whatever, and let a sparse, you know, even if you have a big computer it's gonna take a long time and a lot of RAM. And then this'll just get a pretty good answer in 50 steps and just be way, way shorter. Okay?

Okay, so here is the CG algorithm. There are many variations on it. People seem to focus more on the algorithm itself than actually on the – what the algorithm produces. In my opinion it's much more important what the algorithm produces which is the Krylov sequence and as many – there are other methods to produce the Krylov sequence and they have different round up properties and things like that.

Let me show you what those are. So here's one and this is – instead of just sort of making up my own I just followed exactly one from a very good book on the subject by Kelly. So just to make it – not to invent new notation or anything, it's this.

And the only important part you need to know here is something like this; you maintain the square of the residual at each step. If the residual is small enough, you quit. So this quitting criterion epsilon is on the ratio, it's what we call ada; it's on the ratio of the current residual to the norm of B. That's this thing.

And what you do now is something like this. Your search direction is gonna be something like P and it's a combination of both the current residual and the previous search direction. Then all of the effort in here, well, not necessarily but in most cases is right there, this one thing right here. This is where you call the mult by A method is called right here.

Everything else if you look here is actually sort of an O of N or a blass level one or however you want to call it, call. So, for example, here you update a vector that's O of N, that's O of N, you have to calculate an inner product.

Actually, if you want to parallelize this that's the one that is really irritating. Everything else here can be done in a – is completely distributed. So the main effort here is this call to A. These other things I guess this is also has to be collected together so that's another one that would not be distributed easily but that's the algorithm.

By the way, these calculations can be rearranged like 50 different ways and so you get different versions of it. In exact arithmetic all of those ways will compute exactly the same sequence XK. With round off errors in there they can be different and you'll find people talking about one versus the other and this that and the other thing and you'll find all sorts of different flavors and things like that and people telling you one way is better than another and all that. Yeah.

Student: Is there anyway to know, like, if it will be faster than a theoretic convergence?

Instructor (Stephen Boyd):No, I think the theory just gives you, like, rough guidelines and basically says if you're – if the eigenvalues are clustered – for example, if they're tight, if the condition number of the matrix is small that condition number [inaudible] will tell you it's gonna nail it in 50 steps. Okay?

If they're spread out but have, you know, clusters or something like that, it's gonna nail it, that kind of thing. So in general you don't know. Kind of the worst things you can have are ones whose spectrum is sort of uniformly spread all over the place, right? That's the kind of the worst thing.

Now it also depends on the right-hand side. So if the right-hand side – the B, actually if that one – the worst thing that could happen is B can be sort of a uniform mixture of all of the eigenvectors and that would be kind of the worst thing to happen or something like that.

Okay, so let's talk about – so as I said at the beginning this is mostly interesting. I mean there's a fundamental difference between this and a direct method. In a direct method you give the matrix A, you give the coefficients to the algorithm literally. You pass an array or some data structure and you get the entries.

In CG you do not need that. All you need is a method that multiplies by your matrix A. There's nothing else you need. Okay? That actually in interesting cases that can be some, like, specialized hardware, it could actually carry out an actual experiment. I mean, it could do – it could solve the whole PDE, it could do insane things, right? Do a full up simulation of something, run Monte Carlo, it could be all sorts of weird stuff.

But it doesn't have to be a matrix, is the point. So here's some examples, why can't you do an efficient matrix spectrum multiply. This is kind of obvious, if A is sparse, for example, if it's sparse plus low rank now you better know the factorization here.

And if you want some numbers here you should think of A as like a million by a million. That's just kind of, you know, because if A is, I don't know, 10,000 or something this is kind of not worth it or something. Yeah, so you should – your mental image is should be that A's a million by million or 10 million by 10 million. That's a good number. A 100 million by 100 million.

These are the numbers you want to think about when you think of CG and how these things work. If you have products of easy to multiply matrices, that works. Fast transforms come up so FFT, Wavelet, DCT, these types of things, things like fast Gauss transform that actually is when A looks like this and you do this via multiple pull methods.

This is actually an extremely interesting one. Here's a matrix that is extremely – that is dense, well, half dense, and that this the inverse of a lower triangular matrix, in fact, even the inverse of a sparse lower triangular matrix, that's a great example.

So I give you a sparse lower triangular matrix, by the way, the inverse of that is generally completely dense. So what – and I couldn't even store – let's make it a million by million, in fact, let's make it a million by million banded.

The Cholesky factorization of a million by million banded, of course, I could solve that directly, but anyway. A million by million banded or something like that is actually gonna be banded. It's gonna have – it's small, it's easy to do. The inverse is gonna be a full lower triangular matrix. You can't even store it and it would be completely foolish to actually calculate the inverse matrix and then multiply each time.

But if I give you a vector and I ask you to multiply by the inverse, it means you have to solve a lower triangular set of equations and that you can do by back substitution. If it's a – I'm assuming – I'm taking the case where it's sparse.

Okay, so these are just examples. It's very good to think of examples like this. Okay, now couple things you can do. You can also shift things around. So if you have a guess X hat of the solution X star then, in fact, what you can do is actually – well, you can – this is the residual with your guess. If you solve Az equals this residual then your solution is actually gonna be just if you solve this you get the optimal Z and you add it to X hat and that's your solution.

What happens now is that XK now looks like this. It's actually X hat plus this and it's the argment of this quadratic over the shifted Krylov subspace so you shift by this X hat. And there's nothing you have to do to make this happen, all you need to do is initialize not with 0 in the first step but with X hat and everything will work.

Now this is also very important because this is good for worm start. So what this means is, if you need to solve, let's say a giant circuit equation, giant resister equation, if that's part of integrating a circuit, I mean, sorry, integrating a differential equation for a circuit you will step forward one couple of picoseconds or whatever and you'll solve a similar system. You can use the last thing you just calculated as your guess and this can often give you, like, stunningly good results. So as you will see an application of that in optimization soon.

Okay, but now we come to the real way it's used and I should say this. I should say that if you – in most cases if you simply run CG on a problem it won't work.

The theory says it will work but that's not relevant for several reasons. It's not relevant number one because you have no intention of doing a million – if you have a million dimensional system you have no intention of running a million steps because you don't have that much time. That's the first thing.

So to say that it works in practice means that it works in some very modest number of steps. Or a lot of people I remember hearing – I hear this in – here's something you hear on the street that you're doing on CG when you're doing – pretty well when you're doing square root N steps. Theory says you have to do N but the word on the streets is square root N should do the trick for you.

And if you think carefully about what that means it's just a rule-of-thumb it says if you have a million variables in a thousand steps you should have a pretty good solution. Okay? These are just – I'm just saying these – it could be slow, you know, but you shouldn't be shocked if it's on the order of squared N. That should be the target. So 100 million, 10,000, that kind of thing. Okay? These are the numbers.

Okay, and the other reason it's not that relevant is the following – the theory is that errors in – when you're doing conjugant gradients round off errors they actually add, it's unstable and the thing diverges, actually, very quickly.

So by itself CG generally – often will just fail. If you just have some problem and you run it, it's just gonna fail. Okay, so the trick in CG is actually changing coordinates and then running CG on the changed – the system in the changed coordinates. That's the trick.

That’s called preconditioning and now you know why it is that you needed to know when CG works because the key idea now is to change coordinates to make the spectrum of A, for example, to make the spectrum of A clustered or something like that or live in a small interval. So that’s the key.

So here’s the basic idea, you’re gonna change coordinates, you’re gonna use Y is the coordinates of X and some T expansion and what’s gonna happen is you’re gonna solve this equation here and then in the end set X stars to inverse Y.

Then sometimes people call T the preconditioner but then sometimes they call TT transpose the preconditioner and you get all combinations of preconditioner meaning T, T transpose, M, T inverse and T minus transpose. So there may be some other possibilities like M inverse but the point is that anytime anybody says preconditioner you have to look very carefully and figure out exactly what they mean.

Okay, so that’s the preconditioner. And in a – basically what happens is when you – if you rerun – if you just take CG it’s actually not that bad. You have to multiply by T and T transpose at each step. The other option is to multiply simply by M once and you don’t really need this final solve, in fact.

So this is called PCG. And by the way, this is exactly where you – this is why in these iterative methods you have room for personal expression, right? So if you’re doing – if you’re solving dense equations there is no room for personal expression. If you do anything other than get glass and run Atlas to [inaudible] or cash sizes and things like that and then write good code then it’s just not right.

There is no reason under any circumstances to do anything else. You go to sparse matrices – actually, in sparse matrices is there room for personal expression if you’re doing sparse matrix methods? Yeah, and what is it?

Student: Ordering.

Instructor (Stephen Boyd):Ordering, yeah. So there is some room for personal expression in sparse matrices and as to selection of ordering and that’s cool. Actually you can say, “Oh, I know my problem. I know a better ordering than approximate minimum degree is finding” or something like that. Or there’s exotic ordering methods. You can go look at Netus is a whole giant repository at Minnesota that’s got all sorts of partitioning methods and some of those are rumored to work really well.

Okay, but we move up the scale one step higher to iterative methods. Now, boy, is there room for personal expression and it is mostly expressed through the preconditioner. So, in fact, when you get into these things you’ll find everything is about finding a good preconditioner for your problem. And then you can go nuts and you can have simple preconditioners and complex ones, unbelievably complex ones and things like that. So that’s the – there’s a lot to do.

Okay, so the choice of preconditioner is this. Here's what you want. You want the following. You want a matrix T for which T transposed AT or a matrix M, let's say M. I want a matrix M for which the eigenvalues of MA are clustered, for example.

And here's one choice, how about this? The inverse. That'll do it because now the eigenvalues of MA are all 1; CG takes one step because the eigenvalues are all 1. It's kind of stupid though because you actually then have to, like, sort of invert the matrix or something like that.

So that doesn't make any sense. So here's what you really want. What you really want is a matrix M which somehow approximately is some kind of approximate – well, see approximate can be very crude. An approximate inverse of the matrix and yet is fast to multiply. That's what you want. Okay? So that's the essence of it. And so this is the idea. And the M – this M can be quite – you'll see approximate it like very generous here. I mean, you can be way, way off.

Here's some examples. The most famous one and often very effective is diagonal preconditioning. It's kind of stupid but actually you should try that always and immediately first because it often just works. So that's – you would not call the diagonal of the inverse of the matrix or whatever you would not call that an excellent approximation of the inverse. But you have to admit it's cheap to multiply and all that kind of stuff and it works quite well, amazing.

Here's another huge family, it's actually really cool and it works – it's called incomplete or approximate Cholesky factorization. And so here's how that works.

Now what I'm gonna do is I'm gonna compute a Cholesky factorization not of A but of some A hat. And the way you might do it is something – it's weird. It might work like this, you might run a Cholesky factorization, in fact, there's a whole field on this called incomplete – it's also called ILU, incomplete LU factorization. These are preconditions based on this.

And the way it would work is this, you take the matrix and you start doing a Cholesky factorization on it but you might decide if an entry is too small when you do the Cholesky factorization – you say, "Oh, screw it. Just forget it. I'm just gonna ignore it." Or if you're doing something and you're gonna fill in in various places which is the death of direct methods, you just say, "Forget it. I'll just ignore it."

So you end up calculating a sparse – well, it's a sparse Cholesky factor or something but it's definitely not the original matrix, right? So these methods can also work really, really well, these things.

And here will be an example; you can do the central KY band. That's a version of this. It's a version of this, too, where you just basically say, any fill in of L below some band I refuse to even – I won't even go there. So let's see, these are some obvious – anyway these are sort of the obvious things.

These can work really well and a good example would be something like this, if you have a problem where there's a natural – where something is ordered, for example, in space or in time like in signal processing or control you have time. Then what happens is, you know, things are connected locally, you know, states, transitions, signals and things like that and that leads to this banded system.

Now banded systems you can solve super fast, we all know that. But supposing you have a dynamic system or signal processing but a few things were kind of – you had a main bandwidth of, like, 10 or 20 and then a light sprinkling of little things all over the place everywhere else.

So for some weird reason in your problem, you know, the state here is related to the – it's just, I don't know, I'm just making this up. But the point is you could easily make up an example like that. Everybody see what I'm saying? So it's kind of banded plus a little light sprinkling of nonzero entries other places, okay?

So this would work unbelievably well. You simply do a banded – you simply ignore all the crap off some band, you do a Cholesky factorization on that, that's your preconditioner, and the only thing you're working the error to fix is all that little light sprinkling of entries that were outside that band. So that's the idea.

By the way, a lot of this stuff that I'm talking about, I mean, these are whole fields, I mean, this is the basis of all scientific computing, PDE's, all – so there's tons of people that know all of this stuff backwards and forwards. A lot of this stuff though hasn't diffused to other groups for some reason. So these are actually just really good things to know about.

Okay, so here's that same example with 100,000 nodes and a million resistor circuit and here it's just with diagonal preconditioning. I mean, it's kind of silly because it was already working unbelievably well but you can see this is what diagonal preconditioning does. Diagonal preconditioning actually is mostly useful not for this kind of speed up but for when this residual goes like this and goes like that and then it would go like this, okay.

Theory says it doesn't do that, that's all from round off error. So that how the – and in fact, here's a very common CG stopping criterion used on the street, this. You run CG until the answer starts getting worse then you stop, then you just say, "Well, that's it, sorry. Here's what it is."

So – and I know that's done and I know people who do that in image processing and computational astronomy and all sorts of things. They just – they run CG until it starts – that usually only gets it a couple hundred steps and then basically you can keep running CG but things are getting worse because of the numerical errors you've lost or foganailty and all sorts of things happen.

Okay, so here's the summary of CG. Actually, the summary's all that matters. So here it is, in theory, and that means with exact arithmetic, it's not particularly relevant. It converges to a solution and N steps, period.

That's not interesting or relevant because N you should think of is on the order of a million roughly and the whole point is you have no intension of doing a million steps. Your goal is to do it in a thousand steps, couple hundred, whatever. So that's kind of your goal. I mean, if you're really lucky, 10, 20, 30, these are the numbers you really want.

Okay, now the bad news is that if you – is that actual numeric round off errors actually makes this thing work much worse than you would predict. Now the good news though is that with luck that means a good spectrum of A you can get good approximate solution in much less than N steps. Right?

So now the other main difference with a CG type method or something like that is this, and this is very important, you never form or need the matrix A. Anyway, you can't store a million by million dense matrix anyway.

Well, I mean, maybe you could but the point is you don't want to. So the point is you don't need the matrix, you don't need the coefficients the way it works for direct methods where you pass in a data structure and it's got all the coefficients in it. Okay?

Here you don't need it. What you need only a method to do matrix multiplication. That means you have to rethink in your head all – everything you knew about linear algebra and you have to rethink to this model where what you really have is a matrix vector multiply oracle and that's it.

What the oracle does, how it's implemented is none of your business. You can call it, you can give a vector and it will come back with AZ. And by the way, you could – this would be very bad, but you could actually get the coefficients in A from an oracle. How would you do that actually?

Student: Multiplying by –

Instructor (Stephen Boyd):Yeah, EI.

Student:– EI.

Instructor (Stephen Boyd):Exactly. So you call the oracle with E1 and what comes back as the first column of A? Then you do it for E2, second column, and actually after EN you've got all of A. So then you could pack it in there and then call it LA pack, whatever. But that's not the point here.

Okay, and it's very important to fix in your mind how this is different from factor solve methods, okay? It's less reliable, it's data dependent. So, for example, that circuit I showed you that was with a random topology. I might take another circuit that's got like a pinch point or something in it, 10,000 iterations, nothing happens or it's even worse, so it diverged.

These are data dependent, okay? And this is not the case roughly speaking for direct methods. They're data dependent. And – but there is – you can either think this is the bad way or the good way, the bad way is this, these methods don't work in general unless you change coordinates with a clever preconditioner. That's the bad way.

The good way is to say is that CG methods have lots – I mean, the employment prospects are very positive because you don't just call a CG method, you need somebody to sit there, figure out the problem, try a bunch of preconditioners and tune things and find out what works. So that's – you can think of that a good way. There's room for considerable personal expression in CG methods.

On the other hand you can hardly complain, this is really your only method for solving a problem with a million, 10 million, 100 million variables. So if you made the mistake of going into field where that's needed then that's your problem.

Student: What – so if you have any operator form it's a very, very hard to get some of the preconditioners you mentioned before. You just use easier ones?

Instructor (Stephen Boyd):Yes, so that's – okay, yeah. So if A is an operator form usually you know the operate – I mean, typically you know the operator so it's not this thing where you don't even know what the oracle does. The theory says you could do that.

You know, to get – well, no. So for that you have to sneak around the oracle. It's a good point, you know, you have your – it's not a pure oracle protocol. Just to get to diagonal you'd sort of sneak – you'd send a messenger around in back of the protocol and say, "Would you mind telling me what your diagonal is?"

Normally a reasonable oracle will be willing to tell you the diagonal. So – but that's a very good point. If you have to find your diagonal by calls to the oracle you're screwed. Yeah, okay, that's – any other questions about this? Because we'll go on to the next – the topic which is actually just a – it's kind of obvious it's just applying this to optimization.

Okay, so the – these are called truncated Newton Methods. Actually, there's a lot of other names for it, we'll get to those in a bit. And they're called approximated, truncated and you can do this all the way up to interior point methods. So, let's see how this works. So here's Newton's method and in Newton's method you want to minimize a smooth convex function, second derivatives.

And so what you do is you form – you want to calculate the Newton step by solving this Newton system here. It's symmetric positive definite, that's symmetric positive definite. And you might do that using a Cholesky factorization. This would be the standard method, you know, for problems that are either dense with up to a couple thousands variables or problems that go up to 10,000 or even 100,000 something like that but where the sparcity is such that the Cholesky factorization can be carried out.

So that's what you do here. And then you do a backtracking line search on the function value. It could be on the function value or the norm, either way, and you'd stop basically based on the Newton decrement, that's this thing, or the norm of the gradient. So that would be your typical stopping criterion.

Okay, so an approximate or inexact Newton Method goes like this, instead of solving that Newton system exactly we're gonna solve it approximately. So – and the argument there is something like this, you don't really have to get the Newton direction exactly, that's kind of silly. In fact, for sort of convergence you only need a dissent direction.

Of course, the whole point – the advantage of the Newton Method is these – especially for these smooth convex functions is that it works so unbelievably well and you get your final quadratic conversions and things like that.

So what you really want is to trade off two things. What you want to do is you want to do a fast and sort of crappy job of approximately computing the Newton steps. You want to get a fast delta X. But if it's too crappy a job then the algorithm is actually gonna make slow progress.

By the way, as to whether or not it's gonna make progress at all it'll always make progress because you will see that almost any method for approximately solving a Newton step, every single step is gonna generate a direction which is a dissent direction.

And so convergence of the methods, at least theoretically is guaranteed. What you can lose is you'll lose things like quadratic terminal convergence and things like that. Now if you're solving gigantic problems you may not be interested in quadratic terminal convergence, you're interested in just solving it even to modest accuracy but in, you know, the amount of time, for example, less than a human lifetime. I mean, that's your – but again, this is your problem for solving such big problems. This is your fault, I should say.

Okay, so this is sort of the idea. Now here are some examples, one is this, is to simply replace the Hessian with a diagonal or another one is a band of this and use that as the Newton step because if that's diagonal or banded this can be solved incredibly efficiently. So that's one method.

Another method, and I think we even explored these in 364a a bit or maybe you did a homework problem on this or something. I can't remember. If you didn't, you should have. So – you did? Okay.

Another one is to factor, this is called the Shimansky Method is you factor the Hessian, every K iterations and then use that for a while. And, of course, that requires a method – a factor solve method.

Now the factor solve method – oh, that just reminds me, I have to say something. In a factor solve method if it's dense or in some dense factor solve methods there's a big difference between the factor and the solve. The factor typically goes like N cubed and the solve goes like N squared. Right?

So what that says is you get weird things you can say. You can say things like this, "I need to solve Ax=b." And you say, "No, actually I need to solve Ax=b, Ay=, you know, Ax2=b2, Ax3." You want to solve it multiple times with multiple right-hand sides.

In a dense factor solve method it's the same cost because the expensive part was the factorization. Once you've factorized the – you can solve – you put an investment in and you can now solve multiple versions of that problem very cheaply. That's the key behind this. Okay?

Iterative methods, nothing like that, none. Iterative – want to hear the cost of solving Ax=b by an iterative method if that's like C? Then the cost of solving Ax=b and Ax~=b~ along with the other one? 2C. There's no speed up. You simply call the solver again. Everybody see what I'm saying? So some things that you probably just got used to thinking about, for example, if you factor back solves are essentially free because they're in order less, now don't transfer to these iterative things.

Okay, so this is the Shimansky Method. It also works quite well. Okay, truncated Newton Method is very, very simple. It does this, you're gonna run either CG or PCG and you're gonna terminate early and in fact, it's very important to understand the key is not to terminate early, the key actually is to terminate way early. That's kind of the hope here.

That's – if you want to get stunning results it's gonna have to be something like that. Now, these also have lots of other names. They're called Newton Iterative Methods, they're called – it's also called Limited Memory Newton. It's also called Limited Memory BFGS. You end up with exactly the same – you have to go through all this horrible equations, everyone has their different notation system, but in the end you find out it's sort of the same method. So these methods have lots of names.

Okay, now in a situation like that the cost is not the cost – the cost per iteration is completely irrelevant. So the cost is measured by the number of CG – actually, it's the number of calls you make to the multiplied by the Hessian. That's actually – that's the exact cost is that. That's the number of CG steps.

And to make these things work well you're gonna need to tune the CG stopping criterion, you want to use just enough steps to get a good enough search direction. If you use too many CG steps then you're gonna get a nicer search direction but it's gonna take you longer to get there.

If you use way too many CG steps you're gonna basically be doing Newton's Method now. That's great because now, you know, whatever it is, 12 steps, it's all over. But each of those steps is gonna involve 4000 CG iterations or something. At the other extreme if you have too few steps you're basically doing gradient, it's gonna jam and it's gonna take a long time.

Okay, now it's, of course, less reliable than Newton's Method which is essentially completely reliable. But, you know, again, with good tuning, good preconditioner, a fast Hessian multiply and you need some luck on your problem, you can handle very large problem.

I should say that what – I should say something else about these methods. Whereas one can write a general purpose convex optimization solver and you've been using multiple ones, you've been using SDPT3, [inaudible], all these things.

If you think about it that's really very impressive. I mean basically they don't know what problem is gonna be thrown at them. They can be scaled horribly. I mean, you can make it too horrible it's not gonna work but they can be scaled horribly, all sorts of weird things. You can have weird sick, you know, flat feasible sets and all sorts of weird stuff people throw at these things every day and they do a pretty good job. They're general purpose. Okay?

So but when you get into these huge systems it's much more ad-hoc and ad-hoc means literally it means everything is built, you know, for this.

So, although people have tried to come up with sort of general purpose iterative methods and they're getting close with some things, I think, in some cases. But generally speaking what this – you're gonna need to tune stuff. You're gonna need to – I mean, it has to be for a particular problem.

You have to say, "I'm interested in total variation denoising." Or, "I'm interested in this pet estimation problem." You know, in medical imaging or something. I mean, it has to be a specific problem. "I'm interested in this flow – aircraft flow scheduling problem." Okay?

So having – now the good news is this, once you've fixed to a particular – it's, by the way, not a particular instance of the problem but the problem class. Once you've fixed to one of those things I am not aware of any case where these methods cannot be made to work.

Obviously that's not a general statement. I can't back that up by anything but every time I've ever looked at any problem, as long as you say it's a specific thing, like it's a network flow problem, it's a this problem, it's a gate sizing problem, it's a problem in finance or machine learning, these work always. They don't work in 15 minutes, they work in – it takes a while, right? Maybe not a while, so that's kind of the good news of these methods.

So – and it's kind of the answer to this, maybe once a day somebody comes up to me and whines, often by email actually, because they're not here. But they whine, they go, "I have my problem CVX, you know, I can't solve it" and blah, blah, blah. And okay, you're like, "It worked fine for 10,000 variables, it doesn't work for 100,000." And I'm like, "Well, first of all it's mat lab. It's made so you could rapidly prototype your problem. You need to solve 100,000 variables, you need to know how these things work and stuff like this."

So this is the answer to people like that. They're like, "No, I don't want to write my own software, it's too hard" and everything like that. I'm like, "Then don't solve problems with 100,000 variables." Just go away, don't bother me, or something.

So – but the bottom line is you want to solve a problem with 10 million variables, 100 million in a specific area, like a specific little area, it's gonna work basically. So it's gonna require some luck and it's not gonna happen in 15 minutes.

Okay, so here's truncated Newton Method. You'll do a backtracking line search on this, you can do a backtracking line search on F as well. The typical CG termination rule is gonna stop when this is your Newton System residual here, divided by the gradient, which by the way is the right-hand side. So this is exactly the ada that we had before. This is an okay one, it says – all of these things are kind of heuristic and stuff like that. So exactly what you use to stop maybe doesn't matter so much.

And what you'll do is you'll have simple – with simple rules you'll iter out at some constant number and you'll have this epsilon PCG, that's constant. And so this might be like .1. That would be a reasonable number there. You wouldn't want to put .01 and you sure as hell would not want to put 1E-4, 1E-4 says you're basically calculating the Newton direction and there's no need for that.

Well, maybe to get the thing up and running and verify your code works and stuff like that, you might start with a small problem and make this 1E-4 just to make sure you're actually calculating the Newton step and it should look then exactly like a Newton trace at that point. But you might want this to be .1.

So a more sophisticated method would adapt the maximum number or this thing is the algorithm proceeds and the argument would go something like this. It would say that, look, early on you don't need to solve the – the whole point of Newton is that it changes coordinates correctly in the end game so that if you're stuck in some sort of little canyon instead of bouncing across canyon walls as you would in the gradient method you're skewed towards a direct shot at the minimum. That is what Newton's Method is.

So you'd argue that actually Newton's Method at first you'd totally – I mean, in many cases you're wasting your time. So then you might have actually at first this might be, like, very – this might start .1 and then it might kind of get – go smaller or something and that might modulate depending on how close you are to the solution. There's a lot of lore on this and you can find this in books and things like that but a lot of it is just to play with it. So that's it.

You would find some theory on this and it’s, you know, it’s mildly interesting and things like that. I mean, this would – you’d find some – go find some thing from the ‘70s or ‘80s or something like that or in some book that would tell you, “If you did this you’d guarantee super linear convergence.” I guess my response to that is, you know, we’re not really trying to achieve super linear convergence; we’re trying to solve problems with 10 million variables.

Student: Right. So you [inaudible] tends to take, like, a long time. Is there –

Instructor (Stephen Boyd):It shouldn’t. Why would it take a long time?

Student: Because essentially, I guess, it just starts, like, has to do a lot of queries to get to the, like, I guess –

Instructor (Stephen Boyd):It shouldn’t.

Student:– it [inaudible] with your criteria for –

Instructor (Stephen Boyd):It should not.

Student: It should?

Instructor (Stephen Boyd):Yeah, a general rule of thumb is, you know, you said beta equals a half, so your step – every time you query you’re divided by equals two. If you do five or six of those that’s too many. And the average number of lines or steps you should be doing is like three or something, two, no more, often one point something.

So I’m suspicious of that. And I’m just telling you sort of what people experience. Yeah, that’s not – and in any case you have to evaluate F. You’re gonna evaluate it at every step here. Not every CG step, right? But every outer iteration you’re gonna evaluate the gradient of F and I haven’t added that in to the cost here.

So, yeah, that would come up. But, yeah, you shouldn’t be doing a lot of line search thing. Right? I mean, first of all, Newton’s best when you get into the end game you’re doing none. I mean, if it’s really Newton you should be doing none.

And that will translate over here. When you get down in the region of quadratic convergence with a method like this you shouldn’t be – even though you don’t have the exact with the Newton direction which is aiming you right to the solution, it’s a good enough direction that a full step should be taken.

So it’s actually kind of a bad sign if you’re doing more than a couple of Newton steps. And if later in the algorithm – I’m sorry, backtracking a second. Later in the algorithm if you’re doing lots of backtracking steps it doesn’t sound right. Is that a specific problem you’re thinking of?

Student: It’s just – yeah, I guess.

Instructor (Stephen Boyd):Okay, we’ll talk about it later. Okay, now you also have the question of CG initialization. One way is to initialize with delta X=0. It turns out in that case the first step if you go back and look at the CG thing is you – well, actually it’s very simple, you solve Ax=b. In the first step you minimize over the line span through B. B in this case is minus the gradient.

So the first – after one step of CG to solve a Newton Method if you start from 0 what pops out is a very interesting thing. It’s something – it’s along the negative gradient direction.

So if you take CG and you said N max=1 which means do only one step then it turns out that the steps popped out by CG in fact are scaled versions of the negative gradient. So – and in fact, you can prove things like this, this is – that might even be a homework problem on that in which case we’ll assign it to you or maybe – I forgot. Or maybe there’re – if there’s not a word problem maybe there should be one or something like that.

You can prove the following, that when you run Newton’s Method, sorry, CG to do an approximate Newton Method every step of CG takes your angle of your search direction closer to the Newton direction.

Everybody see what I’m saying? So let me show you the – how that works. So here is – let’s just make it quadratic. I mean, it doesn’t really matter. Here you are, you don’t know it’s quadratic, let’s say. And you are right here, okay? Now the negative gradient direction says search in that direction, right?

But the Newton direction says, “No, I think actually that’s a better step.” That’s – well, that’s Newton, right? If it’s quadratic it nails it in one step. And what you can show is this, that if you run CG starting from 0 your first step will be in this direction and every other step is gonna actually close – is simply – they’re gonna simply move over here in angle towards the Newton thing. Okay?

So everybody see the idea here? So basically you bend – you can even think of CG as sort of, you know, modifying your step taking into account more and more of the curvature, in fact. That’s a good model for what it is.

Okay, now another option is this, you can choose – you can actually use the previous one and do a warm start. Now one problem with that is that if you start with zero every step of CG it will be a dissent direction and this might not be the case here. But you can give it an advantage if you’re only gonna do ten steps, if that’s what your tuning suggestion, you can actually do really well by just using the previous one.

And the simple scheme is something like this, if the previous step is a dissent direction for the current point use it, initialize this way, otherwise you initialize from zero and these are just sort of schemes.

So let’s look at an example. It is L2-regularized logistic regression. So here’s my problem. It’s gonna look like this, and if you want to know what this is, is I am fitting the coefficients in a logistic model right here and there’s L1-regularization over here.

So – and what that means is you have a logistic model, you have binary Boolean outcome and I have a vector of features and I have a giant pile of data that says, “Here are the features and here’s the outcome like plus or minus 1.” You know, like the patient died or didn’t and I give you – or, you know, the stock price went up or it went down.

And I give you – let’s say, oh, to make it interesting a million features. Okay? So – and I give you as a thousand samples. Now obviously that’s horrendously over fit, you’re over fit by a 1000 to 1. If I give you a 1000 samples a thousand patient records and a million features.

So what the L – hey, wait a minute here. Sorry, I’m gonna have to go back and cut out that whole discussion. I thought that was L1. So, all right. We’ll just rewind and skip all that, sorry. We just gonna do L2-regularized. Actually, you can solve L1 – if you can do this one, you can do that one. So, all right. We’ll just do L2-regularized. Sorry, pardon me.

L1-regularized would allow you to do it with a million features. It would run and it would come back with 37 features and it would say, “Here are the 37 out of the million that look interesting. These are the ones that I can make a logistic model and predict these thousand medical outcomes well” or something like that. Okay?

Sorry, this is not that. This is just an example. Okay, all right. So the Hessian and gradient looked like this. I mean, they start looking very familiar, it’s [inaudible] equals DA plus 2 times lambda. Lambda is diagonal here. The gradient looks very similar.

And none of the details matter but the important point is all you have to be able to do is multiply by the Hessian, multiply the vector by the Hessian. When you do that you have to do this and the key is here you don’t even form this matrix because this thing it’s easy to make up cases where I can store A but I can’t store A transposed DA let alone can I even think about factorizing it, that’s a joke.

But this is a case where I can’t even store it let alone factorize it. But anyway, I make this – I multiply it this way so I need two methods, I need a method – by the way, this often corresponds to a forward model method and that’s a reverse or adjoint call. So I have a forward model call and an adjoint call and I’m gonna make two. For each I’m gonna make a forward model call and an adjoint call per CG step.

And we’ll just make an example here with 10,000 features and 20,000 samples. So here we just may have a problem where these XI’s have random sparcity pattern. They have around 10 nonzero entries and then in the nonzero entries it just shows at random. None of this matters. But you end up with about, I guess half a million nonzeros in this and we made this small enough that we could actually do the symbolic factorization just to know what we’re talking here.

And so the factorization gave us 30 million nonzeros in the Cholesky factor. Oh, by the way, that’s something we can handle but as you can see this method will be like just orders of magnitude different.

Okay, so the method is Newton which we’ll look at in a minute and then truncated Newton with various things for stopping. Basically we’re max iterating out because we’re asking for .01 percent error and here’s the convergence versus iteration.

So these are actually iterations and the Newton and the 250 step CG are right on top of each other. So the whole problem takes like – well, I mean, it depends what your accuracy is but, you know, it takes like 15 steps. This is what we expect from our friend Newton, right? This is exactly the kind of thing we expect.

Now, by the way, the cost of a Newton step versus the cost of 250 CG’s is like just orders of magnitude off. The Cholesky factor had 30 million nonzeros. Okay? The CG – the original problem had something like 100,000 nonzeros or something like that. So you’re just way, way, way off by orders of magnitude.

The time for these – you can see here that if you do only 50 CG steps you start losing a little bit, but not much. Now the ten is actually – this is very typical. What happens in the ten is you’re making excellent progress and then down here that’s this one, you actually – you stall. And the theory says that’ll keep going down but we don’t have time to wait for it. So that’s the idea.

Now the right way to do this is actually to look at the convergence versus cumulative CG steps and I see a totally different thing. Oh, by the way, Newton step would be like, I don’t know, probably over in my office over there, the first Newton step in terms of effort would probably be, I don’t know, we’d have to check but probably way, way, way, a kilometer that way. Okay? Just to put these things in perspective.

Now, let’s see here, what you can see if very, very cool. This guy that does ten CG steps, which is hardly gonna win at any prizes for, like, good approximation of the Newton step is doing, like, amazingly well and then it just jams.

By the way, if you’re happy with a gradient ten to the minus five or whatever, like, this is ridiculous. Cumulative number of CG steps is 100 and so you’re solving a problem that I don’t know how long it would take with the direct method but this one might be in the order of like 20-30 minutes or something like that. You’re now back to solving it in milliseconds. So or you’re just orders and orders of magnitude faster.

So here – oh, here's some time so we can actually get the times. So, let's see, so if you do 50 or 250 you're basically the same. So here's Newton Method with N max and it jams near this gradient 10 to the minus 6 but that often is just fine.

Oh, so here are the time, it's 1600 as opposed – 1600 seconds so it's not bad, it's half an hour, something like that, as opposed to four seconds, okay? So these numbers are – these are pretty good factors here. These are worth going after these factors. So these are just much, much, much faster methods.

That's – yeah, this one that jams or something like that. So but you're already way off. I mean, you're down in the sub, sub, sub minute range. If you just do diagonal PCG here's the same picture and I think now it's simple. Now there's absolutely no doubt what the best thing to do is and it's right here. You just do diagonally preconditioned CG to do your Newton method.

Ten steps. Now that's ridiculous. If someone says, "What are you doing?" You say, "I'm doing a crappy job on Newton's Method." And you say, "Really? How's it crappy?" And you go, "I'm using an iterative method to calculate the search direction." You say, "Well, what's your iterative method?"

You say, "I'm doing ten steps of CG and then I quit. Then whatever I have at that point I pass it to the hieroalgram to do a line search." Then you say, "Oh, do you have some theory that says that worked?" And you go, "Oh, yeah, if I were to do 100,000 steps in exact arithmetic I would have computed the Newton step."

So, I mean no one can possibly justify why ten steps is enough here. I mean, you just can't do it, so that's it. But these things work done now unbelievably well, and now that means you can scale. That means you can do a problem with 100 million variables like no problem.

So here are the numbers for this and they're kind of silly. It takes 1600 and then it goes down to 3 seconds or it would be 5 seconds as opposed to, like, 45 minutes or whatever that is. So these are real factors here.

And by the way, if you make the problem bigger this goes up much faster than – this will just go up linearly so you'll just – if you were to make it ten times bigger you could solve a problem – and this would be a minute and that would probably scale much worse than linearly.

So okay, and you can use this for many, many, many, many things. You can use it for barrier and primal-dual methods and I think what we're gonna do is quit here and sort of – and finish up this lecture next time. So we'll quit here.

[End of Audio]

Duration: 75 minutes