ConvexOptimizationI-Lecture14

Instructor (Stephen Boyd):Just a quick poll: I just wanna make sure – how many people have tried the image interpolation problem? And how many actually had trouble with it? One, two – okay. Unspecified trouble, I think is what they're –

Student:[Inaudible].

Student: Yeah. I think it was – had something wrong with the norm function.

Instructor (Stephen Boyd):Okay. No, there's nothing wrong with the norm function.

Student: That was an exaggerated one.

Instructor (Stephen Boyd):Oh.

Student: Like, I was trying to do –

Student: Yeah, it gets a slightly different response than if you just do the sum of squares.

Student: Right.

Instructor (Stephen Boyd):Oh, you do?

Student: Yeah.

Student: Yeah.

Instructor (Stephen Boyd):How different?

Student:[Inaudible]. [Crosstalk]

Student:[Inaudible].

Instructor (Stephen Boyd):Really?

Student: Yeah. I mean, norm looks good, but it's not –

Instructor (Stephen Boyd):Wow. Okay. You know what you should do if you don't mind?

Student: What?

Instructor (Stephen Boyd):Send your code and your build number to the staff website. I mean, the full code and the CVX build number – which I should probably mention about how you're supposed to vote early and often or whatever. You may wanna download –

redownload this CVX stuff every couple of weeks. Not that anything really has changed or anything. No, no. It's things like there's a comment in the user guide that was wrong or there's something mid – tiny, tiny things. But just every two weeks, something like that, when you – you can think of it when you clear your cash or something on your browser. Just download another CV just for fun. So I –

Student: How do I know when to do it?

Instructor (Stephen Boyd):How do you what?

Student: Know when to do it?

Instructor (Stephen Boyd):It's easy enough. You just look and see what the build number is. It's fine.

And you don't have to, right? I mean, some of these – most of these builds are just, like, fixing commas. Okay? So it's not –

Student: Could we just rerun a [inaudible]?

Instructor (Stephen Boyd):You'll figure it out. Yeah. It'll work. It'll work fine.

Okay. But actually, please, if you actually do find what you believe to be an error – and some people have found a couple – actually, mostly in the documentation – actually, e-mail us. And if you're finding inconsistencies and numbers coming out, we – actually, we wanna know about it. So [inaudible] just passively go, "Okay. Now it's working," and move on. Please let us know. Some are not – can be explained, or there's nothing anyone can do about it. Others probably need to be addressed. So – Okay. Back to numerical linear algebra whirlwind tour. So last time, we got as far as the LU factorization, which is also – I mean, it's the same as Gaussian elimination. So this is Gaussian elimination, and it basically says this, "Any non-singular matrix you can factor as a permutation times a lower times an upper triangular matrix." Actually, this, of course, is highly non-unique. There's zillions of factorizations. But for the moment, we don't need – we're not gonna even cover why you'd pick one over another or something like that. That's a topic you would go into in detail in whole classes on this topic. But the point is that once you have affected this – once you've carried out this factorization, you solve AX equals B by, essentially, you factor, and then you do back substitutions or forward and backward substitutions. And a very important point about that is that once you've factored a matrix and the cost – if the matrix is dense and you're not gonna exploit any structure – is N cubed. The back solves are N squared. And actually, what this – now you know something that's not obvious at first sight. I mean, you just have to know how this is all done to really know this because it's kinda shocking. It says, "Basically, the cost of solving two sets of linear equations with the same matrix is actually equal." More or less – I mean, for all practical purposes, it's identical to the cost of solving one. Okay? And that's not obvious, I don't think, because if someone walked up to you on the street and said, "How much effort does it take to solve two sets of linear equations compared to how much it costs to solve one set of linear equations?" I think a normal person would probably answer, "Twice." That's wrong. The answer is the same. Okay? And that's where factor-solve method – there are other methods where the number is twice. It couldn't be any worse than twice, I suppose. Right? But the point is it's the same.

Student: Could you just put the two in one system? Is that what you're saying? Like, just to stack the –

Instructor (Stephen Boyd):Nope, nope. That would make it much worse. If you stacked it like that, you'd have a 2N by 2N –

Student: Yeah.

Instructor (Stephen Boyd):And 2N cubed – you'd be off by a factor of eight. So you'd be eight times more.

Student: So what was your argument for being two – less than two?

Instructor (Stephen Boyd):That's all. One factor, two back solves. Okay. Now we need to get to something very important. A lot of you probably haven't seen it. It's probably – it's one of the most important topics, which I believe is basically not covered because it falls between the cracks. It's covered somewhere deep into some class on the horrible fine details of numerical computing or something like that, I guess. I don't think it's well enough covered, at least from the people I hang out with – not enough of them know about it. And it has to do with them exploiting sparsity in numerical algebra. So if a matrix A is sparse, you can factor it as P1LUP2. Now here, you're doing both row and column permutations, and you might ask – I mean, you don't need the P2 for the existence of the factorization. All you need, in fact, is the existence of P1. You need P1 because there are matrices that don't have an LU factorization. Though the P2 here really means that there's lots and lots of possible factorizations. The question is why would you – why would you choose one over another? And I'll mention just two reasons – again, I'm compressing six weeks of a class into 60 seconds, but one reason is this: you would choose the permutations – the row and column permutations in your factorization. First, you'd choose it so that the – when you actually do the factorization, it's less sensitive to numerical round off errors because you're doing all this calculation in floating point arithmetic. Right? Actually, by the way, if you're doing this symbolically, then you don't need to – this first issue doesn't hold. But of course, 99.99999 percent of all numerical computing is in doubles – [inaudible] floating points. You have floating points. Okay. Now, there's another reason, and this is very, very important. It turns out that when you change the permutations – row and column permutations, the sparsity of L and U is affected. And in fact, in can be dramatically affected. So if you actually get the row and column orderings correctly – that's these two permutations – the LU that you will get here – the number of non-zeros in them when they're sparse – by the way, I should add something like this: that once P1 and P2 are fixed, then you normalize L by – or U by saying, for example, the diagonal entries of either L or U are one – then this becomes unique – the factorization at that point. Otherwise, there's some stupid things you could do, like for example multiply L by two and divide U by two. But once you – then you – then they become a canonical form. They're fixed, and so once you pick these, there's only once choice for L and U once you normalize either L or U. So in fact, the sparsity pattern will be determined by P1 and P2 here. And if you – if these are chosen correctly, you will have a number of non-zeros in L and U that is small. If you choose them poorly, it'll just be enormous. And this is the key to solving, for example, 100,000-variable, 100,000-equation problem: AX equals B. Obviously, you can't remotely do that if it were dense. You can't even store such a matrix. Okay? Period. And if you could store it – well, I guess if you – you could store it and, I mean, if – theoretically, you could, but it's a big deal, and you'd have to use a whole lot of your friends machines and all sorts of other stuff to do this. I mean, it's a big, big deal to do something like that. However, if that 100,000 by 100,000 matrix is sparse – let's say, for example, it has ten million non-zeros entries, which means roughly – this is all very rough. If you have 100,000 by 100,000, you have ten million things – it means you have about 100 – on average, 100 non-zeros per row and 100 non-zeros per column – just roughly. Well, I guess that's not rough. That's exact. When you say average – so what that means is something like this: it says that each variable only enters into – on average, each of your 100,000 variables enters into, on average, 100 equations. Each equation involves, on average, 100 variables. And by the way, if you still – if you think that's an easy problem to solve, it's not. So I mean – however, that can be solved extremely fast if P1 and P2 can be found so that these factors actually have a reasonable number of non-zeros. If you start with ten million non-zeros – if the L and U – if you're lucky, L and U will have about on the order of ten million, 20 million on a bad day. Thirty million would be – you're still no problem – just not even close to a problem. If you can't find P1 and P2 that end up with a sparse cell, then it's all over. But in many, many, many cases, there's gonna be very simple – shockingly simple and naïve algorithms. We'll find permutations that give you a good sparse factorization. Yeah?

Student: How do you know, when you're finding P1 and P2, whether it can be made less sparse?

Instructor (Stephen Boyd):You don't. You don't. You don't know. So what you do is you run – it depends on what you're doing. You run at – you'll actually attempt to solve it, and then you'll be told – I mean, it depends, but actually, it'll be really simple. If it just doesn't respond, like, for the – and you can look at your memory allocation and watch it go up and up and up and up, then you'll know it's not working.

But it's easy enough to know ahead of time. There's actually two passes – I'll talk about that in a minute. The first pass is called a symbolic factorization, and the symbolic factorization, you only look at the sparsity pattern of A, and you actually attempt to find P1 and P2.

In real systems, what would happen is this: the symbolic factorization would run, and by the time it's gotten to some upper limit, like 100 million or a billion, let's say – it depends on, of course, what machine you're doing this on. But after it gets to a billion, it might quit and just say, “There’s no point in moving forward because the L and U I’m finding already have 100 million entries.”

Something like that. So that’s how you do it in the real world. It’d be two phases for that. That’s how it would really work.

The way that would work in Matlab – which, again, let me emphasize Matlab has nothing to do with the numerics. It’s all done by stuff written by other people. They would just, I think, stop. I mean, you just write A/B, and it just wouldn’t respond. And that would be its way of telling you that it’s not finding a very good permutation.

I’ll say more about this in a minute.

One thing that’s gonna come up a lot for us is Cholesky Factorization, not surprising. So that’s when you factor a positive-definite matrix. So if you factor a positive-definite matrix, you can write it as LL transpose – L transpose – I mean, obviously, what this means is it’s an LU factorization.

Two things about it – you do not need a permutation. Every positive-definite matrix has an LU factorization. In fact, it’s got an LU factorization with L equals U.

By the way, just to warn you, there’s different styles for this. Other people that use – this is lower triangular. You can also do it as, I guess, let’s see – upper triangular – there’s an upper triangular factor. There’s different ones, so I guess you just have to check. In fact, who has recently used – what’s the chole in LA pack? That would be the standard. What does it [inaudible]?

Student:U.

Instructor (Stephen Boyd):In the upper triangular? Oh, okay. So sorry. We’re backwards from the world standard. The world standard would be upper triangular, but everything works this way.

So this is the Cholesky Factorization. It’s unique, so there’s a unique – and by the way, there’s no mystery to these factorizations. I’m not gonna talk about how they’re done, but there’s absolutely no mystery. I mean, you basically just write out the equations and equate coefficients. I mean, I don’t want you to think there’s some mysterious thing and you have to take this ten-week course to learn what a Cholesky Factorization – it’s really quite trivial.

To write down what it is – the formula for it is extremely simple. What you should know after our discussion last time in multiplying two matrices is that little formula we write down – that’s not how it’s done. It’s done by blocked and hierarchies and all that kind of stuff, which exploit memory and data flow and things like that.

So let me just show you what I mean here. If you write A11, A12 – you know what? That’s good enough. A22 – something like that. You simply write this out that this is L11, L21, L22 times the transpose of that, which is L11, L21 and then L22 – like that. And now, this is the Cholesky Factorization, and you just wanna find out what are these numbers? Well, okay. Let’s start here. Let’s take the 11 entry. We notice that this times that is – so we find L1 squared is A11. Everybody following this? So obviously, L11 is square root A11. Okay? That’s the first one. Now that you know what L11 is, you just do the same thing. You back substitute, and you look at the next row – would be this one times this one would be L11, L12 equals A12. And now you divide out. Do I have to do this? I don’t think I have to. I mean, so this is totally elementary for you. You just write it all out. And so the existence and formulas – in fact, lots of formulas, obviously all equivalent for the Cholesky Factorization – you can derive this way. But just – and then you could write, by the way, a little chole in C, and it would be about five lines. Again, like matrix multiply, but your chole would be beaten like crazy by the real ones, which would block it up into little blocks optimized for your register sizes and cashes and things like that. Okay. All right. So that’s the Cholesky Factorization. The way you solve – and here you get one third, and this is kinda predicted – the factor of two is totally irrelevant in flop counts. As we – as I mentioned last time, you can be off by a factor of 100 comparing real computation time versus flop count time. So two things with an equal number of flop counts, like for example, your amateur chole or your amateur mat molt will be beat – could be beat by a factor of 100 by one that does the right thing. So – and they have exactly equal number of flops. Actually, it could be more than 100. You could arrange it very nastily to – you arrange N to be just big enough to have a lot of, like, cash misses and things like that. And we could arrange for that number to be almost as big as we like. Okay. So when you do a Cholesky Factorization, not surprisingly, A is symmetric. A basically has about half the number of entries as a non-symmetric matrix. So the fact that the work is half is hardly surprising. But again, the factor of two is totally irrelevant. Nothing could be less relevant. Okay. So this is how you do Cholesky Factorization. And there is one thing I should say about this, which is good to know, and that is this: when you do a dense – when you do a non-symmetric LU factorization, the P is chosen primarily so that numerical round-off does not end up killing you as you carry out this factorization. It doesn’t end up – you end up factoring something, but it’s not – the product of the things you factor are not equal to the original matrix or something like that so your answer is totally wrong. So the P is chosen. That means that this P is often chosen, actually, at solve time. So when you – you look at – you have to look at the data in the matrix A to know whether you should do a – that’s called pivoting, by the way. By the way, you – obviously, if you – you don’t have to follow everything I’m saying, but it’s important to know some of these things. By the way, that slows down a code. Right? Because if it’s got branches and things like that – and then if it’s got to actually pivot and stuff like that, you have data movement. On a modern system, this can be very expensive. Okay? So –

Student: Why did you say [inaudible] fewer elements in the positive [inaudible] case? It seems like they’re the same size.

Instructor (Stephen Boyd):Yeah, they're the same size. But this is a general one. So yeah, if you get N squared entries to tell you what it is – this is symmetric. So you really only have half the increase.

Student:[Inaudible].

Instructor (Stephen Boyd):Okay. Now – so here, you would choose this – the P dynamically. So actually at – when you factor a specific matrix. Here, you don't need it, and in fact, it's actually well known that just straight Cholesky Factorization actually will – is quite fast. And actually, it's quite – it's stable numerically. In other words, it's known that this is not gonna lead to any horrible errors.

This means, for example, if this is small, like 100 by 100, you get something that's very, very fast. And also, it's got a bounded run – I mean, it's got a bounded run time. It just – it will be very fast. Right? Because there are no conditionals in it, for example. Okay.

Now we're talking about sparse Cholesky Factorization. That looks like this. Here, you of course, preserve sparsity by permuting rows and columns. Okay?

So of course, any permutation you pick – if – well, I can write it this way. P transpose AP equals LL transpose. So the other way to think of it is this: you take A, and your permute is – you carry out a permutation of its rows and columns – the same one. Then you carry on a Cholesky Factorization – by the way, it's absolutely unique – the Cholesky Factorization – provided you normalize by saying that the diagonals are positive. Otherwise, it's a stupid thing.

Question? Okay.

Okay. So this – so since you can – I can permute the As any way I like, and then – I'm sorry, compute the rows and columns of A and then carry out a Cholesky Factorization, when A is sparse, this choice of P is absolutely critical. So if you give me a sparse matrix, and I pick one P, there's just a Cholesky Factorization. Period. And that Cholesky Factorization will have a number of non-zeros in L. In fact, if you choose P randomly, L will basically be dense for all practical purposes. Okay?

And then it's all over. Especially if your matrix is 10,000 by 10,000, it's pretty much over at that point.

Yeah?

Student: Could you [inaudible] solve the equation using Cholesky? You need to know the matrix is positive-definite. Right?

Instructor (Stephen Boyd):That's correct. Right. So –

Student:[Inaudible]. [Crosstalk]

Instructor (Stephen Boyd):I can tell you exactly how that works. So in some cases, it’s positive-definite because, for example, it’s a set of normal equations. That’s very common. Or it’s gonna be something we’ll talk about later today. Maybe we’ll get to it – Newton equations. Right.

So a lot of times, it’s known. Now in fact, the question is what happened? Obviously, the matrix has a Cholesky Factorization. It’s positive-definite. In fact, if L is – especially if this is the case, it’s positive-definite. What happens if it’s not? That’s a good question. And here – I didn’t get very far in my Cholesky Factorization, but in fact, what would happen is this: at each step of a Cholesky Factorization, there’s a square root. Right? You really did the first one. The next one would be square root of – well, that’s basically the determinant. Okay? It’d be the square root of the determinant would be the next one.

What would happen is this: that square root – if all those square roots – if the arguments passed to them are all positive, then you will have a Cholesky Factorization. If that matrix is not positive-definite, one of these square roots – you’re gonna get a square root of a negative number. Okay?

And then several methods – there’s actually different Cholesky Factorizations modified. In fact, some – I forget the name of it, but there’s a type of Cholesky Factorization which you might want, and it works like this: right before it takes a square root, it has a test that says if this thing is negative, it throws an exception.

By the way, it can even be arranged to throw – to actually calculate a vector Z for which Z transpose AZ is negative, which of course is a certificate proving A is not positive – or less than or equal to zero. Whatever. So it would do that.

That’s one way. Unfortunately, a lot of the other ones – because these are built for speed – it’ll just throw some horrible exception from the square root or something like that.

So that’s – there’s a modified – in fact, Cholesky is, in fact, the best way to check if a matrix is positive-definite. You find that number. You don’t calculate the IG or anything like that. You just run a Cholesky. You don’t want a basic Cholesky, which will dump core, and then that will be your – then you have a system called – anyway. No. I’m joking.

But the way it would work is this: you’d run with this modified Cholesky that, when it encounters a square root of a negative number, does the graceful thing, which is to return telling you, “Cholesky didn’t succeed,” which is basically saying, “Not positive-definite.”

So that’s the best way to check.

Okay. Let’s go back to this one. This permutation. Now, in – so here’s the way this actually works, and I’ll explain how it works. The permutation here is generally not chosen in Cholesky Factorization. What I’m about to say is not quite true, but it’s very close and is a zeroth order statement – it's good enough. You don't choose the permutation in a sparks Cholesky Factorization

to enhance numerical properties of it.

Now technically, that's a lie, but the first order – it's pretty close to the truth. It's a perfectly – it's a zeroth order. It's a good statement.

You choose the P to get a sparse L. That's what you do. Okay? And what happens then is this: there's a whole – I mean, this actually – there's simple methods developed in the '70s where you would choose these Ps – and I mean, they have names like approximate minimum degree. Then they get very exotic. Then there'd be ones like Reverse Cuthill-Mckee ordering. Then you get to very exotic orderings.

So, that's all fine. That's all fine, but now I'll tell you as most of you would be consumers of sparse matrix methods – in fact, you don't know it, but you have already been very extensive consumers of sparse matrix methods. Okay? Because when you take your little baby problem with 300 variables and CVX expands it – I don't know if you've ever looked, but if you looked, you'd find some innocent little problem you're solving with ten assets or something like that – if you look at it, it'll all of a sudden say, "Solving this problem with 4400 variables."

Well, trust me. Oh, systems of – when it says 27 iterations, I promise you equations of the size 4400 by 4400 were solved each iteration. Okay? You would have waited a whole long time if this wasn't being done by sparse matrix methods. So you're already consumers – big time consumers of sparse matrix methods. So what I'll – what I wanna say about this is that even very simple – by the say, all the things that you did, although it's in the solver, which I don't know the details of – those were – the orderings were very simple ones. They were things like approximate minimum degree. I mean, just really simple, and for most sparsity patterns, this things obviously work fine. So that's what this is. Now the exact value really depends in a complicated way on the size of the problem, the number of zeros and the sparsity pattern. So actually, it's interesting to – I wanna distinguish the dense versus the sparse, so let's actually talk about that. So – and I'll just talk on a laptop little PC – something like that. Dense is, like, no problem. You can go up to, like, 3,000. I mean, just no problem. [Inaudible] scale something like N cubed. You already know that. And it'll just work. It'll start allocating more memory and start taking, like, a macroscopic time, but yeah. It'll just – it's gonna work. There's essentially no variance in the compute time. It's extremely reliable. It's gonna work. It's gonna fail if the matrix you pass in is singular or nearly singular, but then you shouldn't have been solving that equation anyway. So you can hardly blame the algorithm for it. So these things – for all practical purposes, the time taken to solve a 3,000 by 3,000 equation with 3,000 variables – the time taken is just deterministic. By the way, at the microscopic level, that's a big lie. But it's good enough. And the reason it's a big lie is very simple. Actually, depending on that matrix – the data in that matrix, different orderings and pivoting is gonna do – and if you have no pivoting, it might take less time or whatever. And you have a lot of pivoting and all that, it might take more. But it's absolutely second order – just not there. Okay? So that's the deal with – so there's no stress in solving a 3,000 by 3,000 system. Zero. Absolutely zero. It just works. You can say exactly how long it's gonna take ahead of time, and that's how long it's gonna take. Period. Now you go to sparse. Now – well, it depends. There's actually two passes. It depends on the sparsity pattern. So let's go to a typical example: 50,000 by 50,000 matrix. So you wanna solve AX equals B, 50,000 by 50,000. Now some people, you can make a small offering to the god of sparse matrix – orderings that some people say is effective. But the point is, there's no reason to believe that you can always solve 50,000 by 50,000 equations. Okay? And in fact, there's ones that you can't. And it just means that whatever sparse solver you're – whatever ordering method is being used, typically on your behalf – I mean, unless you do go into large problems. If you do large problems, you have to know all of this material – every detail. Okay? If you don't work on huge problems, then this is stuff you just have to vaguely be aware of on the periphery. Okay. So in this case, 50,000 by 50,000 – there's a little bit of stress. You type A/B, it might come back like that. You'll know examples where it'll come back like that soon. Okay? Or it might just grind to a halt, and you can go and watch the memory being allocated and allocated and allocated, and it just – at some point, you'll put it out of it's misery. So that's what – however, the interesting thing is this. Once that's done – if you do it in two steps – if you do a symbolic factorization first, you're gonna know. You'll know. Then after that, it's – for matrices of that sparsity pattern, it's absolutely fixed. Right? So this value that has a lot of implications in applications. If you write general purpose software that has to solve AX equals B where A is sparse, you don't know. Basically, you deal with it every time it's fresh. If I pass you an A matrix, the first thing you do is you do a symbolic factorization. You start going over it, and first, you look at it and you go, "Well, it's sparse. All right. Let's see what we can do with ordering." And then you do the ordering. You do the symbolic factorization. You know how many non-zeros there are gonna be. At that point, if you wanted to, you could announce how long it's gonna take to solve that problem because once you know the sparsity pattern of L, it's all over. You know everything. In fact, just the number of non-zeros is a good – question?

Student:[Inaudible] symbolic factorization?

Instructor (Stephen Boyd):Well first of all, that's in the appendix, which you should read. So let's start with that.

Student: Appendix B?

Instructor (Stephen Boyd):Well, yes. Okay. Some people are saying it's appendix B. It's one of the appendices. There's not too many, so a linear search through the appendices wouldn't – it's feasible, I think. So the appendices where this is covered in more detail, but you have to understand even that is like condensing a two-quarter sequence into 15 pages or something.

So – okay. So the interesting thing is if you're gonna solve a set of equations lots of times – like, suppose you wanna do medical imaging or you wanna do control or estimation or some machine learning problem or something like that, but you have to keep solving the same problem with the same sparsity structure. At that point, actually, you could invest a giant pile of time into getting a good ordering. So it could take you – you could run this whole communities that do nothing but calculate ordering. You could actually calculate your ordering for three days, and when you get – by the way, if you get an ordering that works and has sparse Cholesky Factorization, it's no one's business how you got it. Right? Like a lot of things, it's just – it's fine. So you could do that. And then, it can be fast. So – okay. All right. Well, that's an – oh, let me just summarize this. For dense matrices, totally predictable how much time it is. Sparse – depends on the sparsity pattern, depends on the method you or whatever is solving this equation on your behalf is using to do the ordering. By the way, if it fails on Matlab or something like that because it's using the simple approximate minimum degree, symmetric minimum degree, blah, blah, blah ordering, it could easily be that if you were to use a more sophisticated ordering, it would actually solve it. Just no problem. Right? So – okay. On the other hand, it is too bad that the sparsity pattern plays a role in these methods. On the other hand, hey. You wanna solve 100,000 by 100,000 equations? You can't be too picky. You can't say, "Wow. That's really irritating. Sometimes I can do it, and sometimes I can't." So anyway. All right. Enough on that. And finally, one more factorization we'll look at is the LDL transpose. So this is just for a general symmetric matrix. Here, the factorization is this. There's a permutation. There's an L and an L transpose. In fact, they can have ones on the diagonal. In fact, you can't – what you do here is D is actually a block – it's block diagonal, and the blocks are either one by one or two by two. The essential matrix is easily inverted, by the way. If I give you a block matrix where the blocks are one by one or two by two, can you solve DZ equals Y? Well, of course you can. Right? Because you either are dividing or solving a two by two system. So – okay. And it's the – it's sort of the same story. So I think I won't go into that. Now, we'll look at something that's actually – everything we've looked at so far is just so you can get a very crude understanding of what happens. There's nothing for you to do so far. Right? So in fact, most of this stuff is done – in most systems, this is done for you. You solve a – you call a sparse solver. You can do that in Matlab just by making sure A is sparse and typing A/B, just for example. Or the real thing would be to directly call something, like, from UMF back or spools or there's all sorts of open sores packages for doing this. And you're probably using them whether you know it or not. So you're certainly using it in Matlab. If you use R, you're using it. If – anyway. So – actually, I don't know. Does anyone know? Do you know what the sparse solver is in Packagen R?

Student:I don't have one.

Instructor (Stephen Boyd):No, come on.

Student: It's a separate package.

Instructor (Stephen Boyd):Than – okay. That's fine. But it has one. Come on.

Student: Yeah, but it's not like private [inaudible] language.

Instructor (Stephen Boyd):Well, that's okay. That's fine. That's – I mean, come on. It's an open sores thing. It doesn't matter then, right? So that's fine. Okay.

Now, we're gonna talk about something that is gonna involve your action. Everything so far has been – it's just me explaining what is done on your behalf by some of these systems. Now, we're gonna talk about something you actually need to know because you're gonna need to do it. So – and these are things, by the way, that will not be done for you. They cannot be done for you. You'll have to do them yourself. So this – now is the time – up till now, it's just sort of, "Here's how things work." Now you better listen because now there's gonna be stuff you have to understand and you'll actually have to do.

So we'll just start with this blocking out a set of equations. Really dumb – I mean, we just divide it up like this. You can divide it up any way you like, and we're just gonna do elimination. So it's gonna work like this: we'll take the first row – block row. It says A11 X1 plus A12 X2 equals B1. So I just – I solve that equation by multiplying on the left by A11 inverse, which I'm assuming is invertible. Obviously, a leading sub block of a non-singular matrix does not have to be non-singular. Right? That's obvious, as in 0111 is obviously non-singular, and its leading one by one block – its A11 is certainly not.

But assuming it is, you can eliminate X1, and you just get this. That's X1. You back substitute X1 into the second equation, and in the second equation, you get A21 X1. That's A21 A11 times this junk over here. And then that plus A22 X2 equals B2, and that equation looks like this.

Now, this guy you have seen before, although you don't remember it probably. Or maybe you do. That's the sure compliment. The sure compliment just comes up all the time, so you might as well get used to it because it comes up all the time.

By the way, if you have an operator called the sure compliment, that's what – the Cholesky Factorization is, like, two lines because you just keep calculating the sure compliment of kind of what's below you or something like that.

So that's the sure compliment times X2 is equal to this right hand side.

Now – so this suggests a method for solving this set of equations, and you do it this way. You first form A11 inverse A12 and A11 inverse B1 – and by the way, I have to say something about this because this is sort of already high-level language. When you say form this, how do – by the way, how do you do it? Do you actually calculate the inverse of A11 and – it depends on what A11 is. So what this really means is the following: you factor A11 once, and you back solve for B1 and then each column of A12. Okay?

So it's understood when you see this in pseudo-code that you know what you're doing and that you don't form this inverse and all that kinda stuff. It's just understood. That's what it means.

Student:[Inaudible] in Matlab because we can wait to do that?

Instructor (Stephen Boyd):Yeah.

Student: Can Matlab store that [inaudible]?

Instructor (Stephen Boyd):Yeah. So let me – this is coming up on your homework, which we'll put on the web server. That is if it's up. Sorry about the AFS outage yesterday. I don't know how many people noticed it. Anyone notice it yesterday? I think it raises our blood pressure a lot more than yours. So I gotta learn, actually, just to just kinda – it's like a power outage and just deal with it. But it makes you very nervous. That's my, like, nightmare – number six worst nightmare is final exam – AFS outage. Or something like that. I don't know.

Okay. What were we talking about? Fact solving. Yeah, yeah. Matlab. All right. Yes. Of course.

Okay. Let's suppose that you needed to – let's suppose we had to do a Cholesky Factorization just – okay. So the way you do it is this: So you're gonna factor a matrix A – okay. Here's what you do – and this will work. So first, I have to tell you a little bit about what a backslash does. Okay? Which you could figure out anyway.

So it does a lot of things. I'll just give you what's essential for what we're talking about now. Here's what this does. First of all, it checks if A is sparse or something. But even if A is full, it looks to see if A is upper or lower triangular. Okay? If A is upper lower triangular – actually, more than that. If it can be permuted to upper lower triangular – but let's say it's just – if it's upper lower triangular, it will actually be back solved. Okay?

So for example, that means if I write A is 3,000 by 3,000, that's a big enough matrix that you will see a macroscopic difference. And I type A/B if A is full. It's gonna take order N cubed. I don't know how long it'll take – 20 seconds, 15, 12 – I don't know. Something like that. Okay?

I mean, obviously, it depends on the machine you're on. But it's gonna take a macroscopic amount of time. You'll hit carriage return, and you will wait a little bit before you get the answer. Okay?

If A is 3,000 by 3,000 and lower triangular, what will happen when I type A/B?

Student:N squared?

Instructor (Stephen Boyd):Yes. It's N squared, and it'll be – I won't even see it. I'll just – I'll hit carriage return. It'll take more time to actually go through all the – to interpret it and to instruct X or whatever is drawing your window to type into that terminal. That's gonna take way more time than solving it.

So it'll go from, like, ten seconds to nothing. We can – in fact, we can guess how much faster it'll be. It's gonna be on the order of 3,000 times faster – that's by the order. But in fact, the numbers are different by a factor of two. So it'll be about 1,000 times faster. This is my guess.

Okay. So this is the first thing you need to know. Okay.

Now, suppose you wanna do – suppose you wanna solve AXI equals BI in Matlab where you have a bunch of these. Okay? So this would be intensely stupid. I mean, it should work, but anyway – that doesn't matter. If you did this – I'm just writing something that's in between – I don't even know what it is.

But anyway, if you did this here, this thing – what do you think it'll do? It'll just – it'll look at A fresh each time and go, "Great. Let's factor that guy." Right?

It'll even recalculate the permutations fresh every time. Okay? So this is not the way to do it. Okay?

If you do this, though, it will actually do the right thing. If you do that, it'll do it. I mean, this is nothing but a batch – that's a batch solve. That's all it is because the columns of A inverse B, where B is this thing, is nothing but A inverse B1 inverse B2 and so on. So you can call this a batch solve or whatever you want.

So if you write A/B, it'll do the right thing. It'll factorize A once. It'll go through the columns and back solve.

Actually, to tell you the truth, even that is not – that's actually not what it does. But let's pretend that's what it does. It doesn't do that, but it again blocks it out and calls an LA pack code. But these are fine details.

This'll do the right thing here. Okay?

So for example, what is the run time of this versus this? What's the run time of those two?

Student:[Inaudible].

Instructor (Stephen Boyd):Identical. Basically identical. It's not identical, but it's – basically, it's the same for all practical purposes. Okay? So that's an important thing to know.

Now, we can put all these together. And now suppose you need to factor A once, but then you don't have all the things you wanna back solve at the same time. Then you can't do this trick. It's not a trick. I mean – okay. Here's what you do. Now we'll get through it, and I'll show you how to do it.

You would do something like this. I'm just gonna do this approximately. By the way, I think this will give you the upper triangular version of L, but it hardly matters.

So you do this once. Okay? And the cost on this is N cubed. Is that the comment? I think it is. Okay, there. Okay? That's an N cubed thing. Okay?

So for 3,000 by 3,000, you're gonna – there'll be a microscopically visible delay when you do this. Okay?

If by then, do you – well, let's see how I can do this. You have LL transpose equals A. So A inverse B is L minus TL inverse B. Is that right? Yeah. So here's my – ready for my code now? If I do this in store L, anywhere else from now on – if I wanna solve AX equals B, I can do it super fast like this. Let's see if I can get this right.

There. There you go. I can execute this anywhere I like with different Bs, and it will execute super fast N squared. Do you know – do you see why? The reason is – I mean, there's a lot of interpretive overhead, so the whole thing – I mean, this is a bunch of stupid overhead. Basically, it looks at this – I mean, I'm just – this is just – this is the formula for that. Right?

So it looks at this, and it says, "Okay. Let's look at L." And it looks at L, and it pokes around for a while. And then after a while, it says, "Say. L is lower triangular. Oh. Okay. I'll call the back solve instead of doing blah, blah – instead of doing – factoring it. It's already factored," or something like that.

So we'll do this. Then this will be passed to this thing, and it'll look at that, and it'll analyze the matrix as if it's never seen it before and go, "Hey. Wait a minute. That thing is upper triangular. Oh. Okay. I'll just go ahead and –."

By the way, all of this thinking and – takes way longer than to actually do it, unless the matrix is a couple thousand by a couple thousand. Right?

So – and then it's fine. But the point is that this will now execute very fast. So I think I just answered your question.

Okay. All right. So let's – back to block elimination, which is what this is called. So what you do is you form A11 – and actually, the comment I just made is very important because it's about to come up right now.

So these two, you would solve by factorizing A11 once and then back solving on A12 concatenated with B1. So this line would translate into a very compact thing. Okay.

Then you form the sure compliment. Now, to form the sure compliment, the point is that you've already calculated these columns. So you would – this is just a matrix multiply here and a subtraction. So you form the sure compliment, and you form B tilde – that's this right-hand side guy here. This guy. No more of this inverse is appearing here, but you've already calculated the sub expressions, so this is fine.

Then you determine X2 by solving this equation. That's the sure compliment

times X2 B tilde, and then you get X1. Once you get X2, you go back and you replug X1 from wherever our formula was – here it is. Right here. From this guy. Okay?

Now, the interesting thing here is you have to do the following: you have to solve A11 X1 equals this. But guess what? A11 you already factored on step one, so this last – step four is actually a back solve. It is not a factorization – the back solve. Right?

So these are very important things.

Student:I have a question.

Instructor (Stephen Boyd):Yeah.

Student: Why does the sure compliment arise? Like, what's the common thread?

Instructor (Stephen Boyd):Oh.

Student: Is it just coincidence? Or is there, like [inaudible]. [Crosstalk]

Instructor (Stephen Boyd):No, it's not coincidence at all. Yeah, there's a physical reason for it. It has lots of – I think you did a homework problem on it, anyway. I think it's – it comes up when you solve one set of equations – if you have a block set of equations, and you solve one in terms of the other. That's basically where it comes up. So – and it comes up in a lot of physical things, too, like in networks and – it just comes up all the time in all sorts of things. PDEs – it just comes up everywhere, basically. So whenever you have some linear system, and you impose some boundary conditions on some of the variables, a sure compliment's gonna come along.

It comes up in statistics, too. I mean, this is – this happens to be the covariant – this is the error covariance of estimating X1 given X2 or the other way around. Do you remember which it is? One or the other. But I have a Galician joint variable, and I estimate X1 condition – if I look at the random variables expected – the condition expectation of X1 given X2. I look at the conditional covariance – it's this thing. So just – it just comes up everywhere.

So – okay. So if we work out this block elimination and see what happens, you have a big drum roll, and you work it out, and you think it's all very clever. When you total it up, you get something like this. And we're gonna do it in terms of factorizations and solves – back solves. You get something that looks like that. Okay? And we're going to assume, by the way, that the smaller system – the sure compliment one – we're gonna do by dense methods. So to pay for the dense method, it's gonna be N cubed.

So you get something that looks like this – N2 cubed. You get this. Now, if you just plug in the general matrix A, you – it comes out, actually, not down to the leading order, but of course, as it has to be down to the flop. It's identical to what we had before. So there's no savings. There's absolutely no reason to do this. Well, sorry – in terms of flop count for a dense matrix.

Now, the truth is this is exactly how it's done when you do blocking and stuff like that. This is how you get fast because you arrange for, like, the sure compliment block size to be just right for your registers and your L1 cash and all that kinda stuff. And you get that size just right, and this thing will give you a big speed up. But that's a secondary question.

Now, this is useful – when this is – block elimination is useful exactly when A11 has extra structure that can be exploited. Okay? Now, that's – when it does, this method is gonna speed you up like crazy. So let's look at some of those.

Here's an example. We'll do a couple of examples, actually, because they're very, very important.

If A11 is diagonal, okay? That means that your matrix looks like this. I'm just gonna draw a pattern like that. Okay? If it looks like that, then you're gonna do unbelievably well, and here's why. Let's just put some numbers on here. Let's put 10,000 here, and let's put 1,000 here. Okay? Something like that.

You're gonna do really, really well. So I have 11,000 equations, 11,000 variables. But it's blocked in such a way that A11 is diagonal. Okay?

And by the way, this has meaning – this will have meaning in any application which this arrives. It basically common variables, and everybody depends on them. Every equation, everybody – if you go – if you look at every variable, they all depend on those 1,000 common variables.

But then there's 10,000 variables that don't depend on each other. And all I'm doing is I'm providing the English description of this sparsity pattern. That's all I'm doing. Okay?

So if you see this – actually, from this class onward, if you see this, some big green lights should light up. This is what you want to see. If you see something like this, you will know this will scale to unimaginably large systems. Okay? And it'll scale very gracefully.

Now, the reason here is this. You just go back and look at what we're gonna save, and you can actually even determine – I lost my – oh, here it is right here. All right. So let's actually see what happens in this case. A11 is diagonal. Well, this is, like, super cheap right here. That's, again, a non-issue. Don't even think about it.

This – by the way, here's the joke. Forming S is now the dominating thing. So in fact, if you were to – the joke, as in this problem – if you were to actually do a – if you're gonna profile the code that solves this, you would find out that, most of the time, it would be done doing matrix multiplication. It would actually be forming this. Okay? That's gonna dominate completely, and in fact, it's gonna be something like – it'll be 1,000 by 10K by a 10K by 1,000. Matrix multiply – that's it.

In the end, you'll have to solve 1,000 by 1,000 system, but that's gonna be 1,000 squared, and this one was 1,000 times – 1,000 squared by 10,000 is ten times more. So we know exactly the number of flops it's gonna take. It's gonna be 1,000 – it's gonna be multiplying this matrix by one like that. That's it. And it will be seriously fast.

If you were to take this and pass in a matrix like this to – and just write this back slash something, it's not gonna work. I mean, basically, it'll just sit there and – actually, it'll very happily start factoring this – very happily. This will not be recognized – this thing. This structure will not be recognized, and it'll start factoring things and things like that. And most of the time, of course, it's multiplying zero by zero and adding it to zero and then storing it very carefully back and then accessing it again later – pulling it back in. But it's very happy to do so. No problem. It'll just do it, and it just will never finish. Okay?

So – by the way, this is called arrow structure. So it's good. It's a good thing. So from now on, arrow – by the way, this – let's do another example. Suppose, in fact, this 10K was ten 1K by 1K blocks. Okay? I can't fit ten in there, but you get the idea. Okay?

By the way, there's a story behind this. Basically, it says, "I have 11 groups of 1,000 variables each." Okay? One of those groups connects to everybody – that's this last group. The other ten only connect to that last guy. So – in fact, you should even visualize the following graph. Right? It looks like that except there's ten of these. Okay?

So each note here on the graph is a group of 1,000 variables. This group does not depend on that one, but one of them depends on all of them. How do you solve this? By block – I mean, you do block elimination, this'll be way, way fast because how do you do A11 inverse?

A11 is 10,000 by 10,000. What's the effort to compute A11 if it's ten 1,000 by 1,000 blocks? How do you actually – well, you don't calculate A11 numbers. Sorry. How do you calculate A11 inverse A12? How do you form this expression? How fast can you – what's the cost to factorize a block diagonal matrix?

Yeah, you factorize each block. So this is ten times 1,000 cubed divided by three. That beats like crazy 10,000 cubed over three. I mean, by a long shot. Okay? So that's very important.

Okay.

Student: So are – is the moral of the story don’t use A/ and just go through this algorithm in your own script?

Instructor (Stephen Boyd):No. Now I’ll tell you a little bit about this. Okay. So having said all that for this example, here’s what would happen. So – okay. So let me now, let’s talk practically about what would happen here. It would probably work. If you had a matrix whose sparsity pattern looked like this, okay? If you did this – now, let me just explain a few things. If you just did this /B, too bad. It’ll never work. And it’s 11,000 by 11,000 – never, never. Okay? However, if you gave, say, Matlab or some other system the hint that this was sparse – so if you say it’s sparse, all you’re saying is that there’s a bunch of zeros. There are a bunch of zeros. You see this giant triangle here – giant triangle here. These are all zeros. Okay? That’s a lot of zeros. It’s a lot of non-zeros, though, over here. Right? But if you do – then, actually, it is overwhelmingly likely that it won’t do a bad job because the method will actually get all of these guys first, and then – in the ordering, you will get something – it might not be – it might not get it exactly right, but it’s likely to do pretty well because this – so in fact, this arrow structure is one that a normal sparsity handling system is probably gonna recognize. Okay? So in this case, declared as sparse – do backslash – you can do your own experiments, but they’re not really very interesting experiments. They’re just experiments on what some people implemented in messages. I mean, it’s not that interesting. Right? I mean, the important part is if you do real problems like this, you’re calling these things directly. Then, of course, it’s your job to do this. But now, let me mention some others where it’s not obvious. Okay? Let’s do one. Here’s one. Let’s suppose you do medical imaging, for example, and suppose you do MRI or something like that. And you have to solve an equation that looks like this. Okay? Let’s make this like one million variables, and let’s make this 1,000 here. By the way, I have no idea what this is, but let’s just imagine that such a thing is there. And this one million by one million here – if this is, like, diagonal, you know what to do. We just did it. I’m gonna make this, though, a DFT. Okay? So it’s a discrete Fourier transform. That’s a dense matrix – completely dense. So you can form – well, you could attempt to form your million plus 1,000 by a million plus 1,000 matrix and actually attempt to solve it. It’s completely dense. Everyone agree? Totally dense. Okay. Those sparsity is gonna help you out – the gods of sparsity – you are on your own on this one. Okay. However, how fast can you solve this system? You can’t even store this, right? All you have to store is this part, right? This is pushing it, but that’s your fault for going into medical imaging. You do medical imaging, you’re not gonna do this on something with a couple of gigs of RAM, right? Obviously, we can work out what that is. But this is perfectly storable otherwise. You don’t store this block, obviously. How do you solve this?

Student:[Inaudible].

Instructor (Stephen Boyd):Yeah. Of course. So you look at this, and you go back, and you say, “Well, what do I have to do?”

And you go, “Well, I have to calculate A11 inverse a couple of times.”

By the way, you do not solve a discrete Fourier transform equation by – you can't even form – if it's a million by a million, you don't even form a factorization. It doesn't matter. You actually calculate by the inverse DFD. It's analogue N. So it's 20 times 20 times a million flops. It's way, way fast. Okay? Everybody got this?

So – by the way, these are not obvious things. They're known by lots of people, but probably more than an order of magnitude, more people who should know these things don't know them. So that's why I'm telling you these things.

So – okay. So this is very important to know. So – in fact, I cringe whenever I hear people very unsophisticated – in very unsophisticated ways talking about scaling. And it's like, "Well, I tried that. It's way too slow."

Well, anyway. By the way, the difference between this and just typing A/B – the technical name for that is – this time, they actually – that's called stupid or naïve linear algebra, and this is called smart linear algebra.

Smart linear algebra means you kinda know what you're doing, and you're exploiting structure at this level. That's what it means. So –

Student: How do you [inaudible] in English?

Instructor (Stephen Boyd):How do you communicate it – well, you – well, it might – it'll just end up being sparse. I mean, do you mean in Matlab, how do you do it?

There's a very bad way, and then there's a good way. You make it sparse in the beginning, and then it'll be preserved as you do the right operations on it.

You have to be very careful in Matlab because, like, one false move – one little character here, and boom. You have a dense matrix. You add identity to it. Now, adding identity to a sparse matrix, you got a sparse matrix. Right? Not in Matlab, you don't. So you have to know, "Oh, you shouldn't add IEYE, you have to add spy or something." I don't know. SPEYE – that would be the sparse identity.

Student:[Inaudible].

Instructor (Stephen Boyd):Right. Now, of course, that makes you wonder under what circumstances the identity not sparse, and I don't know of any. Maybe two by two case because then you could say, "Well, only 50 percent of the entries are non-zero."

I don't know. It's not my – but anyway. Okay.

So I think that's it. So basically, you wanna look for things – block things. If you look at a block, and you see a block that's easily solved, exploit it.

By the way, if what's easy to solve about it is it has to do with it's sparsity, you can just be lazy and let your sparse – let whatever – do your offering to the gods of sparse matrix factorization orderings – heuristics for the orderings, and hope that it works. But if it's dense or something like that, you're gonna have to do the work yourself. That comes up in a bunch of cases, one of which we're about to look at right now. But that's the idea.

Let's quickly talk for fun about some fast – what systems can you solve fast? I mean, we've already looked at some obvious ones: diagonal, lower triangular – we just talked about one a few minutes ago: block diagonal. How about, like, block lower triangular? Can you do that fast?

Yeah, of course you can because if it's block lower triangular, you can do back substitution at the highest level or whatever, and you'll have matrix inverses to do on the blocks. Okay?

We talked about another one, [inaudible] FD. Sorry – that's the algorithm. DFD – so discrete Fourier transform is the operation. [Inaudible] FD is am algorithm, which does it fast.

Others would be things like DCT – so Discrete Cosine Transform that you'd use in imaging. Anybody know others that are fast?

Student: Toplets?

Instructor (Stephen Boyd):Which one?

Student: Toplets.

Instructor (Stephen Boyd):Toplets, yeah. Toplets is a good one. So you have fast algorithms for toplets matrices, and they all have names. And usually, people are tortured by having to learn them every operation by operation like Levinson-Durbin is one of them, and – did they teach this in statistics? Probably. No. Okay, good. Fast – oh wait. It doesn't matter. For toplets? No. Okay.

Hunko matrices is another one, so it goes like this.

Student: Circulant.

Instructor (Stephen Boyd):Circulant – yeah, circulant. How fast can you solve a circulant?

Student:N log N.

Instructor (Stephen Boyd):N log N. Right because you take an FFT, multiply FFT back. So that – and circulant, of course, is the matrix is circular convolution. Convolution – well, that's toplets. We've already covered that.

There’s a bunch of others, and it’s fun to actually poke around and learn about others, like there’s one called the fast Gauss transform that comes up in machine learning. That’s, like, way fast.

So it’s – there’s a bunch of them that are, like, order N. There’s some other ones that come – they’re whole fields that can be described as fast methods of solving AX equals B.

For example, if you solve an elliptic PDE, you are solving a very specific AX equals B. Okay? And those can be done, like, some of them are light, and they can solve it insanely fast. It’s not easy to do. It is not trivial. But for example, if you go to someone and say, “Oh, I really – I have this grid. Like, I have an image,” let’s say. Right? For example, suppose you have an image, and all the constraints only connect a pixel and its immediate neighbors. And you say, “I have to solve a set of linear equations that looks like that.”

It’ll have a very characteristic look if you look at it. It’ll be, like, stripes and things like that. And if you do PDEs and things, you’ll just immediately say, “Ah. Okay.”

But in fact, solving that equation, that’s the one-one block in an image. Let’s do the image you’re doing or will do or have done in homework eight – seven. What are you doing now, anyway?

Student: Six.

Student: Six.

Instructor (Stephen Boyd):Oh. Well, that shows you what we’re thinking about. All right. So suppose we scale it up to a million by – 1K by 1K, so you have a million variables. You – actually, each step in that algorithm – the 20 steps you will solve – I promise you a million by million equation. Definitely.

It’s super duper sparse. Right? Because each row and column will only have, like, two and three entries because you’re connected to yourself. You’re connected – four entries. Whatever. You’re connected to your neighbor above, below, left, right. Right?

So super sparse. By the way, sparse orderings won’t do so well. It does okay for your homework – or it should, anyway. It does. But it won’t scale to bigger numbers very gracefully. However, if you look at it carefully, you’ll realize that, in fact, solving that set of equations – that big A11 block for that problem, you know what it is? Solving an elliptic PDE on a uniform grid.

Then, you go to your friends. It’s like a plus on equation or something like that. So you go to your friends who know about PDEs, and you say, “How do you solve that?”

And they fall down laughing. And they’re like, “Are you kidding? 1K by 1K? We precondition. Blah, blah, blah – Fourier transform – multi-level.”

I mean, whatever they do – there's probably some people here who do this. Okay? It's not easy. It's a whole field. Right? But the point is, they know how to do it. And then you – after they calm down, you say, "Great. Could you just give me the code that solves that?"

And then wherever you have to do A11 in the back substitution, you put that there. Now you're done.

So, okay. Okay. That's enough on that.

Let's look at a consequence of this. It's the idea of a structured matrix plus a low rank. This is actually very important. This is also extremely good to know. And this is also – this is something that you have to do. In other words, you better recognize this because no automatic system can do this. Period. So you have to do it.

So here it is. Suppose you need to solve A plus BC times X equals B. Normally, B and C are small. So you're supposed to think of B – this is low rank. This is, like, rank P. B is a skinny matrix. C is fat, and A is something that's quickly invertible. A could be the identity or diagonal – block diagonal, lower triangular. You take your pick. It could be solving a plus on equation. It doesn't matter. It's something. You can do fast circulant – who cares?

Okay. And you wanna solve this equation. Now, here's the problem. Generally speaking, whatever structure was good about A is almost certainly destroyed by adding – in fact, just add a dyad. Just add a little B, a little C to it, and it'll destroy anything. It'll – and of course, it'll destroy a DFD matrix, a DCT matrix, toplet structure is totally blown out of the way. Diagonal – forget it. You make an outer product, it'll spray it non-zeros everywhere. A block diagonal ruined – so that's kinda the idea.

Okay. By the way, once someone calculates this and hands you this matrix – all over. You can't – if they don't tell you it's A plus – once they multiply this out and hand it to you, it's all over. You'll never recover. You'll never get any speed out of it or anything. You're doing – you're stuck at N by N now.

Okay. So let's kinda do this. What you do is you do what we did in duality, which is you uneliminate. You introduce new variables. It's strange. So here's what you do. You uneliminate. You write this as AX plus B times CX, and you introduce a new variable called Y, which is CX. You uneliminate.

Then you write this as AX plus BY equals B, and then CX minus Y equals zero. That's that this is Y equals CX.

So, if you can – these two are completely equivalent. If you solve one, you can solve the other. Okay?

Now when you look at this, you should have the following urge: you should look at a matrix like that, and your eye – don't ignore this. Your eye should gravitate towards the minus I. I mean – well, yes. That's right. Okay. Sorry. Your eye should gravitate towards this minus identity. And it should look at that, and you should have an overwhelming urge to carry out block elimination because a matrices is easy to invert. They don't come simpler that I. Right?

So you should have an overwhelming urge to eliminate this. By the way, if you eliminate this I, what will – if you eliminate this thing, what will happen? You'll get the short cut, and you'll go, "Oh my God. This is great. This is fantastic. That's an I. I now know that if I see an easily inverted matrix appearing as a block in the diagonal, I know what to do."

You do it, you'll form the sure compliment. That's the sure compliment. Okay? So by the way, if the I were huge and A were small, great. But of course, this is [inaudible]. So if you eliminate this block first, you're actually back where you started.

So now the trick is – so you have to actually hold off on the urge to eliminate an easily inverted block. You must hold off. This is very important, and instead, you eliminate A. Okay?

So that's the idea. And actually, let me explain the reason. The reason is this: in this application, I is small. This is – let's say A is a million by a million, but B and C are each a million by ten and ten by a million. But anyway, that's what they are. In that case, this I is ten by ten. So it doesn't really help you to eliminate this. It's this million by million guy you wanna go after. And A, for example, could be a DFD – a discrete Fourier transform. Right? Or something like that. That's the one you wanna go after.

If you eliminate that, you get this: I plus C inverse B times Y equals this. And this allows you – this, we can actually handle efficiently. Why? Because you put A inverse B here. That's fast because, somehow, you have a fast method for doing A inverse. You form the – in this case, you form the smaller one, and then you solve that. And then this is sometimes written – this has got lots of names. It's called matrix inversion lemma, but it's got lots of other names. There's – if you're British, it's called Sherman-Morrison-Woodbury, and then any subset of those, depending on if you went to, like, Oxford or Cambridge or something like that. Okay. So that's one. And it's also called – let's see. Matrix inversion lemma, Sherman-Morrison-Woodbury – there's one more. I think it's if you came from MIT, there's some name you have for it special. It doesn't matter. You'll hear another name for it. So – what's it called? Anyway – I don't know. I'll think of it. It hardly matters. The point is you'll hear lots of people talk about it, and it's – sometimes, it's just written out as this formula: A plus BC inverse is equal to this, assuming all the inverses here are – A is invertible and A plus BC is invertible. You get this formula. And you would see this – I mean, here are the types of things you should know. It's a staple in signal processing. In fact, essentially, most signal processing relies on this. This is basically the one non-trivial fact of all signal processing up to about 1975, I'd say. It's just this. Same – by the way, this is also the trick for optimization. So let me explain what it is. In that case, B and C – I mean, the most classic case is something like this: diagonal plus low rank. So here – well, I can't use lower b. So PQ transpose – there you go. So, suppose you need to solve that – something like that. So diagonal plus low rank is a beautiful example of a matrix that, if you know what you're doing, you can solve equations diagonal plus low rank so fast it's scary. But if you don't know what you're doing, it's a total disaster. So this one, you can solve. It's actually – in fact, what is the order of solving that? Order N. It's just order N. It's – there's nothing here to do. It's just order N. So if someone says, "Don't you understand? I have to solve a million by million equation?" And you go, "Yeah. I understand." And they go, "It's totally dense. Dense! A million by a million! That's like 10 to the 12 entries in my matrix." And you go, "Yeah, I know. But you can solve it in about a half a second. Maybe less." Okay? But you have to know what you're doing to do this. Okay? Nothing will recognize this. By the way, you might ask, "Is it easy to recognize a matrix, which is diagonal plus low rank?" If it were easy to recognize – I guess you know the answer, by the way. I'm asking. But if it were easy to recognize, you can imagine some processing or some higher-level thing that would look at the matrix – it could think a while. If it's a big matrix, this might be worth thinking. So it can look at it for a little while and go, "Look at that. You're lucky. It's diagonal plus low rank." So what do you think? Is it easy to determine diagonal plus low rank? No. It's a famous problem in statistics. It's in lots of other – it comes up in lots of areas. It's impossible. Okay? Let me just ask this just to see if we're on the same page here. How about this: how about alpha I plus PQ transpose? Do you think is that right? Am I asking the right thing? I think I am. All right. By the way, I'm not sure I know the answer to this one, but let's try it anyway. This might – what I'm about to say – my example might be just for symmetric, but let's just try it anyway. How about a multiple of the identity plus rank one? Can you detect that in a matrix? How?

Student:[Inaudible]. [Crosstalk]

Instructor (Stephen Boyd):[Inaudible]. The eigen values will do it. Thank you. Thank you. What are the eigen values?

Student:[Inaudible].

Instructor (Stephen Boyd):You got it. Okay. And that's if and only if. Right. So in fact, this we can determine. This we can determine. Okay? Because one eigen value will be different, and all the others will be equal to alpha. Okay?

Student:[Inaudible].

Instructor (Stephen Boyd):Correct. Right. So then we say fantastic. We have a method of recognizing a multiple identity plus low rank. Okay? And we have to calculate – all we have to do is calculate the eigen values. And what's the cost of that?

N cubed. Actually, with a multiplier, it's about five or ten times more than just solving the stupid equation. Okay.

So – all right. Well, anybody had this? I mean, the main thing here's to think. But diagonal plus low rank? Forget it. So – okay.

So – all right. So that's the picture. And these are just sort of stunning – you get stunning speed-ups if you know this material.

By the way, I should say I went over this kind of fast, and a slightly less fast description is the appendix, which you should definitely read. And we're going to assume, moving forward, that you've read it and got it.

A lot of this, by the way, you can experiment with. I mean, you can experiment with it in Matlab. You have to be careful, though, because you're kind of experimenting half with – half of it is this method, and half is what Matlab actually sort of does and stuff like that.

So – okay. So I guess we'll quit right here.

[End of Audio]

Duration: 72 minutes