ConvexOptimizationI-Lecture13

Instructor (Stephen Boyd):Okay, so let me start with an announcement. Yesterday's section, which is to say the section which would have been yesterday had yesterday not been a holiday, is actually rescheduled to Wednesday. Now I think it's [inaudible], but of course the website is where the – you'll find something that's more accurate. I was gonna say the truth, but it's merely more accurate. So that'll be tomorrow.

These will be online, the sections, but they're actually only online at the SCPD website, so which I figured by now I'm sure you figured out. And then I wanted to make another announcement that was just sort of general. The TA's and I spend a lot of time like putting together the homework problems, and, of course, we're starting to think about the final, not that you should, but we have to. Anyway, so I just wanted to let people know if you have a – if there's a topic you're interested in and we haven't covered it or something, just let us know.

I mean not that we're gonna change things, but if there's an application area, wireless, signal processing, communications. It's gotta be something everybody can get, but you – where the entire setup takes one paragraph and no more. But if you have – if there's an application area you're interested in and we're not covering it because I guess for some reason, I don't know. We keep doing log optimal finance and I forget what.

We keep falling back on in various maximum likelihood problems or something like that. So, but if there's areas that you're interested in, just grab a TA or me and let us know and maybe we'll do something about it. So, in fact, if any of you have a question, maybe we'll end up putting it on the final or something like that if we can put your research project on the final or something like that.

Okay. We'll continue with our whirlwind tour of applications in – they're geometric. Last time we started looking at this, at the basic linear discrimination problem. By the way, this is very old. This goes into the 40s, this topic. And we ask the question when can a bunch of points labeled? I've labeled them by changing the symbol from X to Y. So I have binary labels on data in our end. And the question is when can they be separated by a hyperplane? This problem goes back to the 40s.

Embarrassingly, for many people who worked on this, it was very strange that, in fact, for years, decades, it persisted. The people did not know this could be solved by linear programming. I mean some did always from like day one. Others did not for a very long time. So that's probably even true today that there are still people who don't know this is the most elementary linear programming.

Anyway, so the question was when can you separate, and we did this trick. This trick you're gonna see a bunch of times. You have strict inequalities here. Now, of course, in general strict inequalities, in many cases, there's no practical distinction between strict and non-strict inequalities, in many cases. In this case, it's – there is a huge difference. If these are non-strict, well, then A = B = 0 works always. So here it's actually – the strict is serious. It changes the problem.

And the trick here is to notice that these are homogeneous inequalities in A and B. And so they can be – these strict inequalities can be replaced by non-strict inequalities, but with a gap, so 1 and -1, okay. So this is a trick. You need to know it because you're gonna see it a bunch of times. It's how to handle strict inequalities that really do matter, strict inequalities in homogeneous problems.

Student: What did you [inaudible]?

Instructor (Stephen Boyd):If they're not homogeneous, well, depend – no, you can't use this trick, that's for sure. So you have to argue specially in that case what to do, I mean in each case, case-by-case basis. It could be that they're not relevant; I mean that the difference is not.

Student:[Inaudible] just –

Instructor (Stephen Boyd):You can do that, but you'd have to argue what you're doing is right, yeah. So that's what you'd have to do. Okay, actually, we'll talk a little bit about when we get to something called phase one programming, methods for determining feasibility. So we'll get to that.

Okay, now you can also – I mean, of course, the set of separating hyperplanes, that is to say the set of AB that separate two sets of data, provided they separate, provided they separate, the set of AB is a convex cone. In fact, it's a open polyhedral cone, okay. So it's a open polyhedral cone, or if you like, down here in this parameterization, it's a closed cone, but this has got the – this is in force to separation. Actually, it's no longer a cone now. It's just a convex set.

So then that means that if – once they separate, you can determine – you can go – you can now optimize something about the hyperplane. Probably the most obvious one is to make a hyperplane that sort of has the greatest distance between it and the other ones or something like that. So one way to say that is to work out the thickest slab that lies between the sets, and the way that works is this. If you have two hyperplanes, for example, well, the set defined by terms Z + B is 1 and then minus 1, the distance between them is 2 over the norm of A, right.

And that in general, A transposed Z + B is the signed distance to the hyperplane A. It's the signed distance provided you'd divide by norm A, 2 norm of A. It's the signed distance to the hyperplane A transposed Z + B = 0, okay. So that's what that is. So the thickness of this is 2 over the norm of A, so that's the – so then if you want to maximize the margin here, that – if you want to maximize this, it's the same as minimizing norm A2 and so you end up minimizing norm – ½ norm A2. The ½ is irrelevant subject to these now inequalities to the 1's and -1's.

So this is – and this is a QP and it's a QP in A and B and it will give you the thickest slab that separates two sets of points, so provided they are separable. Otherwise, this is infeasible if they can't be separated. Now, by the way, geometrically there's something that should be kind of clear here and we'll see that it comes out of duality anyway. It's this. If this is the thickest slab, so this problem is basically put a slab in between the two sets of points and then thicken it until you can't thicken it anymore.

Well, it turns out that is – turns out that's exactly the same as the following. You calculate the convex hull of the two categories of points like this, like that. And you calculate the minimum distance between these two convex hulls, okay. Actually, that's an upper bound on this maximum, and, in fact, it's equal to – Jim, I'm just – this is just arguing intuitively. The thickness of the maximum slab – so finding the thickest slab between two sets of points, that fits between two sets of points, number 1 and number 2, finding the smallest distance between the convex hulls of the points.

Geometrically, it looks like those should be duals. They should have equal value and so on. And, of course, that pops straight out of duality. So roughly speaking, it says that separation problems, maximum separation problems, thickness – you know, when you want to maximize, the thicknesses, the duals of these typically are distance problems, distances between convex hulls, oh, and vice-versa. So what this means is when you're solving, for example, a minimum distance problem, the dual will always have an interpretation of somehow putting a fixed slab in between things and vice-versa.

Okay, let's work out how that works. Well, we'll just go back to this problem, sorry, the QP here. So we go to this problem here, max origin problem, and we're gonna work out the dual of this problem. And I'll just – I won't go through the details now because you can do this on your own. You end up with a a Lagrange multiplier, lambda and mu.

These are the Lagrange multiplier for these inequalities and these, these lambda and mu for these. And you end up with something that looks like this. You want to maximize one transposed lambda plus one transposed mu, some of the lambdas plus some of the mus, subject to this second order cone inequality here, a single one. Then the sum should be equal and they should be positive, okay.

Now the optimal value of this is equal to the optimal value of this problem. All right. So here I'll change variables to theta I divided by 1 transposed theta. What I have to look at, you have to realize here that lots of things here are homogeneous here. So if I change to theta I's, which is the normalized lambdas and the normalized lambdas and then gamma I's are normalized mu's. You end up – and then you take T as this variable here. You invert the objective, so instead of that, we'll minimize 1 over the objective. We'll call that T. Then you end up with minimized T subject to this. And if you'll look at this carefully, you'll see exactly what it is.

These two say that theta is a probability distribution or a set of weights. Therefore, that is a general point in the convex hull of the XI's. That's a general point in the convex hull for the YI's. This is the difference of the two of them in norm less than T. We're minimizing T, so that’s the same as minimizing this thing. So it’s a distance between the two convex hulls problem, so that’s what you get here. So again, I didn’t – I don’t expect you to be able to follow any of this. Just the idea is just to be able to see that they are related by duality.

Okay, now mention a couple of things, some of which are, well, both highly hyped and highly useful. We’ll get to some of that in a minute. So we can talk about approximate linear separation of non-separable sets. So actually, in many applications, that’s quite useful. And if you want to put an application in your mind, let’s let me give you one. You take email and you take a training set and you flag it as spam or not. In other words, a person flags it. It just said, “That’s spam. That’s not.” And you take the email and you calculate various features.

A feature could be anything, so that’s what an expert in this kind of thing would do is go and work out what the features would be. It might be something – it doesn’t even matter what the feature is. It might be via currents or not of a term. It might be the frequency of a current. It might be the distance between two terms. It could be all sorts of weird stuff in it, but an expert will go analyze the text, not just text, but also technical things like maybe where it came from and all that kind of stuff.

And you’ll make a feature vector, and the feature vector might be, let’s say, 10,000 long. So each – remember back to exactly this thing. So these are now vectors in our 10,000 or something, or roughly. It doesn’t matter, 1,000. That number doesn’t matter. It’s not 5. There’s lot of – you have lots of features here, okay.

And what you’re hoping is that, in fact, there’d be a hyperplane that would separate the spam from the non-spam. I mean that would kind of – and then it’s great because a new email comes in. You calculate the features rapidly and simply find out what side of the hyperplane you’re on. And that’s your prediction as to whether it’s spam.

So everybody, this is the application, okay. So, in fact, you’re not gonna get a separating hyperplane. It’s essentially impossible. So what’s gonna happen is a couple are gonna be over here, right, that you’re just gonna miss on some. Real data sets are gonna have a couple of points here and a couple of points here. You still want to deal with it. You want to come up with something that, I mean, there’s lots of ways you can imagine approximate separation.

The simplest would be this. Okay, the most natural would be this. I give you two sets of data and actually you verify it by solving the linear discrimination problem, and in fact, they are not separable by hyperplane. That’s fine. Then what you want to do is this, and you might ask, please find me the hyperplane that minimizes the number of missed classifications. I mean that’s just totally obvious that that would be the kind of thing you’d want. It makes perfect sense, right? You’d say, “I have, you know, 30,000 emails, and if, in fact, I misclassify .1 percent, for example, that’s still a bunch of them.”

That would probably be okay. That would probably be quite good, so that's – but it'd be fine. The idea that you'd get all 30,000 exactly right is basically generally not possible. Yeah.

Student: Do you [inaudible] the size of an airplane?

Instructor (Stephen Boyd):Oh, absolutely. We're gonna get to that momentarily, yeah. So you could do – yeah. Actually, one of the big questions or big pictures or conclusion is it turns out things you can do in a low dimension space with all sorts of curved separators and things like that, it turns out a very good way to do those is to blow them up into a huge dimensional space and do linear separation there. So that's why you hear a lot of people talking about focusing a lot on linear separation. It's generally linear separation in a much larger space, right, which actually could involve non-linear stuff.

For example, one of the features in this space could be something like the product of the number of occurrence of one word and another. Given our application, I won't say those words, especially because we're being taped right now. But, for example, pairs of – so in that case, you actually – you think you're doing something that's linear in a high dimension space, but, in fact, you're really doing something non-linear because some of the features are products of others or whatever, okay.

I don't know if I answered that question because it got less clear actually. Anyway, let's move on. So you want to minimize the number of misclassifications. Well, it turns out that's impossible. That's gonna be hard to do. Actually, you know what? I'm not entirely certain of that, so let's just say I believe that's hard, that's a hard problem, so is minimizing the number of misclassifications. However, we have a very good eristic, which you know now, and it's based on these L1 type ideas.

So you know that, for example, if you minimize an L1 norm of something, you're typically gonna get something that's got – that's sparse. And so this is an exceedingly good eristic for minimizing cardinality, which is a number of non-zero elements. So here's what you do. I'm gonna put in a little fudge factor that's UI.

Now, if UI is – these are gonna be positive, by the way. The UNV are positive. If I put in a – if U is 0, if UI is 0, it means that inequality is satisfied. That means everything's cool. You're on the right side of the hyperplane. You're directly classified. But I'll put in fudge factors and they'll be positive, non-negative. If they're 0, it means no fudge factor was needed for that point being on the right side of the – actually, not hyperplane, but slab. That's right. It's on the right side of the slab if UI is 0, or if VI is 0, it's on the right side of the slab. Everybody got it?

Now you – if you minimize the sum of those what you'd really like to do perhaps is minimize, in fact, something like this, the number of non 0 used in these because any time you threw in a fudge factor, it means you failed to meet that – to classify that one correctly. You failed to classify that one correctly and you need a fudge factor to help you classify it or whatever. So if you minimize the sum like this, and by the way, these could be weighted, but this is gonna be a very good eristic for linear separation of non-separable sets.

And indeed, if you solve this problem, it does really well, and you get something that will typically misclassify a very small – a small number of points. Is it the minimum number of points? No, we don't know and probably don't know. By the way, I don't know what this is called the machine learning, this particular one. Does this have like a – usually since they imagine they invented everything they would have a name for – somebody in machine learning, is there a name for this one?

Student: Support detriments.

Instructor (Stephen Boyd):And isn't this the limit of the support vector machine at one end or something? They probably – I would imagine they'd make up some fancy name for it and then write multiple books on it, but we'll get to that momentarily. It's just this, okay. This is the support vector machine, but it's a limiting case of one, okay. So this is an example where the sets are not separable and you can see it's done quite a good job of doing this. Of course, all these pictures are silly because the applications of these, I don't have to tell you, are not for separating clouds of points in R2 because your eyeball is excellent at that, okay.

This is for distinguishing binary tagged data in dimensions like 10,000 and up where your eyeball is not so great. So that's what this is for. Okay, this leads us to the support vector classifier, and here there's actually – you make it a buy criterion problem. And the buy criterion problem is this. You want to – this is sort of the fudge factor here. But in fact, here you can also trade off the width of this slab, and if you do that, you end up with something like this. This is one version of what's called support vector. Oh, and this is just so pedantic. It's called SVM by people in machine learning and things. Anyways, remarkable.

And so, I mean, it's nothing. It's just two things. This is basically the inverse width of – it's two times the inverse width of the separating slab. This is an extremely well known, totally elementary eristic for making this sparse, meaning please misclassify few. And you get this, which is nothing but a QP, all right. That didn't stop people from writing like multiple books, all of which are more or less incomprehensible. Many of them are completely incomprehensible, but anyway, so it's called support vector machine.

Don't you like that? A machine, a support vector machine. It's great, isn't it. It's kind of what – okay, so it makes it sound fancy and so on. Okay, so that's it, and here is, I think, the same data set. It is indeed, but here's the same data set. Here we put – let's see what we did. We put less weight on gamma here and what that meant is we spent more effort making a small. You make a small, 2 over norm A is the width, so this thing opened up like that and I got this one. Actually, it's a bit different. Let's see, I don't know if I can overlay them, but well, I can't see it, but maybe you can if I go from there to there.

Yeah, it twisted around a little bit or something like that, but okay. Now, even though it's sort of hyped, I have to say the following. These things work unbelievable well. So, in fact, things like spam filters and things like that, of course, you have to have the right features, but generally speaking, these things work like really, really well, which kind of make sense. Things that are over hyped and then don't deliver kind of go away. This is not going away, so I assure you of that, so okay.

Somebody asked about linear versus non-linear discrimination. You can do anything. I mean you can easily separate things by non-linear function. For example, you might linearly – you might have a function which is itself a – you would consider a linear – a subspace, in fact, of functions. Now that's exactly what we did here. So far we do linear classification, which you might really call affine classification. The class of functions you look at are affine.

Here they could be anything you like. It wouldn't really matter. It doesn't – what matters is the following, is that the function family that you look at is linear in the parameters. That's all that matters, in fact. Then everything works. You can do discrimination with anything you like. They could be polynomials. They could be all sorts of crazy things. Well, I mean, for example, you could make them gaucians and things like that and you could do discrimination that way, and gaucians parameterized with different widths and different positions. And you could get quite fancy and do separation that way.

But as far as we're concerned, they're all very simple. The base problem is always a linear program, so period. And just as an example to see what you can do, suppose you said quadratic discrimination. That basically says you parameterize F by P, Q, and R. And so you're searching a solution of these inequalities. Now the variables here are P, Q, and R, so, in fact, these inequalities in P, Q, and R are what? They're what? What kind of inequality is that in P, Q, and R?

Student: It's linear.

Instructor (Stephen Boyd):It's linear. It's just linear. So this is nothing but a set of linear inequalities in P, Q, and R here, but obviously now I can add other constraints. If I make P positive, definite, or something like that, then, in fact, it means that this one will be – well, negative definite. It means that this one is actually an ellipsoid. So I can add the constraint. You know, please make me an ellipsoid that covers my data and cover – and doesn't contain as many of these.

And, of course, once you get the basic idea, we could even make a – let's see, how would you call that? A support ellipsoid machine, SEM, the new rage in machine learning. You can – so you can say, "Well, what if you find me an ellipsoid that trades off, so size of the ellipsoid versus number of misclassified or their blah, blah, blah." And we could easily make this up this way. It'd be extremely simple to do, right, to do that. And then you'd get approximate ellipsoidal separation and things like that. But let's just try a few – I'll try a few things on you and see what you think.

What if I said suppose the dimension is huge and I only want to look at, for example, P's that are banded with bandwidth 5? And I – will that work? Did I solve separation by a P, which is banded with a width of 5? Sure, yeah, it's just a subspace. It's no problem, right. So those kinds of things will work, okay.

Here's another example where there's a bunch of data here and this is separation by a fourth degree polynomial. For example, there's no third degree polynomial that separates them. So you could solve by quasi convex optimization, clearly the problem if someone says, "Do you find the minimum degree polynomial to separate these sets?" Actually, more generally, if I gave you any – if I set a basis elements, but ordered, so gave you an ordered set of basis elements like F1, F2, F3, and so on, and then I said, "Find the minimum number of these taken in order to separate the points."

That's quasi convex because you just do – you solve an LP feasibility problem with some number of them and then you know whether you can separate with that number of basis elements or not, okay. So that's that, that's the idea.

Okay, we'll move on and look at one last sort of topic in a family, problems in geometric optimization problems. And these are placement and facility locations. Again, there's tons of varieties on these things. There's whole books on these, so the idea here is just to give you the flavor of that.

So placement and facility location problems generally look like this. You have a bunch of points or coordinates. Now often, these are in R2. Sometimes they're in R3. And they're – but they're often not in higher dimensions, but there's no reason they couldn't be. So they could be in R2 or they could be in R3 or they could be in R4, which would be something like space time or something like that.

Then what happens is this. You have a bunch of points. We'll just make it R2. A bunch of points are given and others are variables. So, for example, what happens is you're given – these are – well, I'll make these fixed, and then the variables are these. So you can think of these. There's many names for these. These are sometimes called anchors, depending on the application because they're the ones that are fixed and don't move, and then these are sort of the free points. They move around, okay. Oh, the other name for these in circuits is pins, for example, or something like that, fixed cells or pins or something like that.

And what happens is for each pair of points here, you have a cost function that is a function of the pair of distance. In fact, usually it's a function of a norm of the difference, but it could just be a function of the pair. And I'll give you some examples here. Let's do circuits, for example. So in circuits we're placing cells here and what I will do is this. These are fixed. There are maybe pin – these are external pins or other blocks or whatever that you're gonna – that are just – whose position is fixed. You're doing placement.

And these are cells that can move around, and I have a net list, meaning I have connections in between a bunch of these things that look like this, something like that. And, in fact, each of these might have a weight on the edge, which would be the number of wires that connect one cell to another, okay. And the absence of an edge means there's no wires connecting it or something like that. So, for example, there's no wires directly connecting this cell and this one, for example.

And now your goal is to position these to minimize, for example, the total wire length. That's a very common problem. And, in fact, the wire length would typically be done – I mean if you care about this, would be done in a L1 norm because you do vertical, horizontal routing of wires in a circuit, for example. And there's all sorts of things that make this more complicated and so on. This could be sort of a radio relay network or something like that and the distance between two nodes could be the following. I'm sorry, the cost could be the following.

It could be the power required to establish that wireless link, right? And that would be, for example, a function of the distance, for example. And in that case, I'd say, "Please move my mobile forwarding stations in such a way that the total power used by the network is minimized, for example." I mean I'm just making up examples. They just go on and on and on like this. Or you could say, "I have a distribution of a commodity. These are the warehouses. Each of these lengths could have a – each of these could have a weight, which is the amount of traffic that goes between these two places, and you could ask to put this in such a way that the total amount of freight times miles is minimized, just for example.

But okay, so that's a typical placement problem. Okay, now these, of course, are all convex, these. By the way, the minute you start doing placement, there's all sorts of other stuff that immediately pushes it out of the convex realm, and I'll just mention a few of those just for fun. One is if you're doing placement in circuits. There's another rather important constraint. That is that cells should not sit on top of each other, so that's not an overlapped constraint. That's not convex by any, not even remotely because you're placing cells. So that's one problem.

And, of course, in all the other ones, they're silly. It turns out, for example, if you have to go – if you want to look at the power required to transmit from two points, it might not be just a function of the distance. It might depend on the terrain and things like that. Then this gets very – then it's clearly, again, non-convex and so on. But first cuts are typically convex like this.

So here's just a simple example. So we're gonna minimize a function of 2 norm of the distance of the pairs. We have six free points, and I think the anchor points are these around here, and there's 27 lengths here, and here are the six free points. So if you minimize the sum of the norms, you get this picture here. If you minimize the sum of the squares of the norms, you get this. And you can actually see what's happened here.

Oh, by the way, you’re not gonna get stupid with these things. You’re not gonna get stupid placements, right? For example, this node is not gonna be – no one is gonna place themselves outside the convex hull of the anchors, for example, right? Because moving it just to the boundary will decrease this. You’re not gonna get this point sort of way over there, right.

Oh, I should mention one other application of these things. One is just drawing or visualization, so that’s a perfectly good example. So, for example, I might have drawing your visualization would go like – you know, I’d have a bunch of points. And I would have some measure of sort of a similarity. And you want similar things. You want to plot this on a screen or on a piece of paper, and you want similar things to be close to each other. So what I have is a dissimilarity measure or something like that.

And so I want to place the points of the similar things close and dissimilar things are far away, just for example. And this is the kind of thing that would do that. It would cluster similar things and so on. Okay, so here you can see what’s happened is here you have a lot of – in fact, I should mention something here. If you minimize the sum of these norms, that is just like an L1 type thing. So what’s gonna happen is a bunch of these are gonna be equal.

So if I have six free points, I think – let’s see. Did I actually have – one, two, three, four, five, six? Okay, so I believe, but I can’t see closely enough here. My guess is that two of those, at least one point is sitting exactly on top of another. They cluster. So if you minimize sums of norms, they will cluster. It’s just like an L1 type problem. They’ll cluster here, okay.

And you will have some long ones. And if you look at the histogram of interconnection lengths, you’ll see you have some wires that are as long as or lengths that are as long as 1.5 and so on. And you have – let’s see. Well, so we may have one that’s 0. There’s at least one that’s very small, might be actually exactly 1. My guess is it’s exactly 0. In the – when you go to squares, what happens is these long lengths irritate you.

Well, they irritate you squared compared to just linear irritation, and so this spreads out so that the longer lengths are shorter, which is better, and you have bigger distances in the middle like this. By the way, the name for this is quadratic placement. This is called quadratic placement and used in a couple of different fields. And then you could do something like you’d have the fourth power and that will spread it out even more.

Let me ask you a couple questions here. We’ll do it this way. What if H was this? You get a free ride from 0 to 0.3 and then it grows linearly. What do you think the placement’s gonna look like?

Student: It’s gonna be a lot of 0.3.

Instructor (Stephen Boyd):Yeah. It says basically distance is free up to .3. So what’s gonna happen is when you solve this they’re actually gonna spread apart. And a lot of them are gonna be just exactly .3 away from each other, right? Some of them will be longer because they may have to be. Some will be shorter because it doesn't matter, but a lot of them will be about .3 away from each other, so that's what will happen here.

So, all the same ideas once you start understanding about how to construct these functions. Everything comes into play. I mean I could even do this. I could then make this go to plus infinity at .5, and then, of course, no wire length will be less than .5. It might be infeasible, but if possible, it will arrange it no wire length is less than – if there are wires, no wire length is more than .5, for example.

Okay, that finishes up this whirlwind tour. You should be reading the book as well because it goes into much more detail on these and the variations and things like that because, first of all, I'm going fast, and so anyway, you should be looking at the book.

Okay, we're actually gonna start a – I mean that actually sort of finishes one section of the class, although it's not really because it'll just go on forever now. That's our whirlwind tour of applications and we're just gonna – we'll just be doing applications, as I said, from now on, no exceptions, until and through the final. We'll just do lots of applications.

But we're gonna start a new section of the course, and it's actually on the algorithms, so how do you solve these things, about how, I don't know, the stuff, the codes you've been using. How do they work? What do they do? And we'll go through all of that. You'll end up knowing not stuff at the very – at the ultra high end, but actually well enough to implement that work like shockingly well. And then some applications work better than sort of generic custom.

So we're also gonna cover – actually, the stuff we're gonna work on – we're gonna start today is extremely useful. Basically for anybody taking this class should know the material here. Probably some people do. Actually, how many people have actually had a class on like numeric or linear algebra and some of the things like that? So just a handful, okay.

So everyone should know this material, just period, end of story. It doesn't matter if you're the most committed theorist, makes absolutely no difference. There's no excuse not to know this material. So that's my view of it.

And actually it's just a few things will get you very far. That's the problem. If you get books or take a whole course on it, it's just too much stuff in too much detail. And it's so much stuff and so much detail that by the end you've been so pounded with all this material that in the end you can't even remember what the main points were. So we'll do it so lightly and at such a high level that you'll have no option but to realize what the main points are because we're not going into any others, and we'll barely go into them.

So, okay, so this is our – and let me also just put this in perspective. You will know – actually, won't take very long. You will know how to, for example, solve a second [inaudible] program or an STP. I don’t, a couple of weeks, that’s all. You’ll implement one absolutely from scratch and it’ll work and it’ll work actually like really well, okay. So and it’s not a big deal. We’re talking like 30 lines of code. It’s just nothing. It will be competitive with something that’s got thousands and thousands of person hours of development in it, but you’d be shocked at how well it’s gonna work.

So let me say a little bit about how it works. Oh, by the way, I should say my position also on all this material. So when I talk to a lot of people, so a lot of people who use optimization are sort of very naïve about it. Everyone is fascinated by this. It’s just all the [inaudible]. That’s all they care about. So you mentioned something and I’ll go somewhere. It’s above noxious, but I’ll say something like, “That’s an SOCP.” And they’d say, “Oh, yeah. Oh, great, sure. What’s that?”

But then immediately focus in on like how do you solve an SOCP, in which case I’ll – and I think sort of the correct response to how do you solve an SOCP is it’s none of your business. It’s technology. It’s just not your business. SOCP’s can be solved very well. There’s a thriving community of people who do it. There’s lot of algorithms worked out, tons of papers on it, open source software. There’s also commercial software if you prefer to pay for it.

I mean, so it’s sort of not – but everyone just wants to worry about how you do it. Now, of course, if you solve giant problems, that might be necessary, obviously, right. If you solve – so this is why I don’t recommend you go into either machine learning or image processing or medical imagine. These are just – if you’re already in one of these fields, sorry. It’s too late because that means you are gonna have to know how to write your own things because you can’t just use generic stuff to solve problems with 10 million variables.

If you’re in another field, good, stay there, with a modest number of variables like 10,000, maybe a couple hundred thousand, something like that. And also, avoid fields with real-time constraints. So any field where someone can say to you, “That’s great, but that’s way not fast enough.” Then again, you should avoid that field. Again, this is only if you have not chosen a field to go into, but all right. Where am I going with this? I have no idea.

Okay, no, I just remembered where I was going, yes, yes, of course. Oh, so I want to say that – basically what I want to say is this whole section of the class on numerical stuff is good to know it, but remember, most people will focus on this. If you take a class like this at some other institutions that I won’t name. Well, I would name it, but I’m not going to. You will find the entire class, basically from the beginning to the end, is sort of on algorithms. I mean this is weird.

This would be like taking entire class on like TCPIP, right? I mean you can do it and somebody has to do it, right, otherwise the rest of us can’t use it. But there’s just something weird about thinking that’s what’s important to teach about this material. What’s important to teach about it is actually the applications and the modeling and all that kind of stuff. This is way secondary. But how do you say we're gonna now launch into it?

And let me say what he big picture. So the big picture is this. Let's say you're gonna solve – we're gonna solve non-linear constrained, inequality constrained problems, okay. We're gonna build our way there. And the way that's gonna work is this. We're gonna build it up by each – we're gonna solve each more sophisticated layer by solving a sequence of ones at the next layer. There are just – I'm gonna give you the big picture right now. This is how it's gonna work.

You're gonna – so let's just take an LP, okay. So let's just take – here's an LP. And the way we're gonna solve an LP is you're gonna solve a sequence of quadratic – sorry, of non-linear, but smooth minimization problems. That's what you're gonna do. That's how we're gonna solve an LP is you're gonna solve 20 non-linear, smooth, non-linear minimization problems with no inequality constraints. That's what you're gonna do, okay.

How are you gonna solve these? You're gonna solve each of these by solving a sequence of, in fact, quadratic minimization problems, okay. That's how you're gonna solve these. This is gonna be Newton's Method, by the way. The name for this is an interior point method. Then this is Newton's Method that converts solving a – minimizing a general non-linear convex function to one where you solve a quadratic minimization.

Now, by the way, quadratic minimization is very interesting because if I ask you to minimize a complex quadratic subject to the quality constraint, the solution involves solving linear equations. So this is basically – that's a sequence of linear equations, okay. So what it basically says is the following. When you solve an LP, yeah, you'll do about 20 of these or something like that, and it doesn't real – let's see. Is that about right? No, even less, sorry.

You'll do like ten of these, and in each of these you'll do five of these. And so the point is that what'll really happen is if someone profiles your LP software code, it will discover that it's doing basically only thing. It's solving linear equations and it's doing so 50 times. I'm just giving you the big picture. But I'm going to say that the whole foundation rests on linear equations.

Oh, by the way, we could go lower because linear equations rest on multiplication and this sort of stuff, and you can go down, and that – then we can actually talk about floating point calculations and blah, blah, blah. So you get – I mean we could keep going, but we'll just stop right here. So okay, so this is why solving linear equations is important, so all right.

So we'll start by talking about sort of matrix structure and algorithm complexity. So how does this work? We'll just look at just AX = B with a square matrix, okay. Now, for general methods, the cost of solving AX = B grows like N cubed. And that's just a – that' approximate for some reasons I'll say in a minute. And just for fun, I mean, last night I typed into my laptop, and so how long?

By the way, N cubed when N is 1,000 is not a small number. It's 10 to the 9. Roughly, it's on the order 10 to the 9. Floating point operations go down. How long do you think it took for me to solve AX = B? This is on my laptop, which is two years old last night.

Student:A few seconds.

Instructor (Stephen Boyd):What?

Student:A few seconds.

Instructor (Stephen Boyd):Okay, that's a good number. Any other guesses? Yeah, I mean the actual order of magnitude, you're in the ballpark. That's a lot. To write 1,000 cubes is a lot, okay. So the answer is half a second, okay. So that means on a modern machine right now it would be much – it would be substantially less, but okay. So, I mean, I – you want to get some rough – I mean that's actually why all of this is possible is because basically computers right now are way, way fast. I mean amazingly fast, okay. So I said about half a second.

Oh, by the way, for fun I went to three – I went from 1,000 to 3,000. It should have gone up by a factor 27. Is that right? No, 8? Sorry, three, I did three. I went 1,000 to 3,000 for fun. Should have gone up by a factor of 27. It actually went up only by a factor of 20, but roughly that was it. And I dropped it down to 100.

By the way, what's the prediction? If N = 1,000 took .5 seconds on my laptop, how about N = 100? By the way, the number you're about to work out, it's a very important number actually, and it's something that is not appreciated at all, I mean like throughout the world. So what's the number? I mean if the scale's like N cubed, what's the number?

Student:[Inaudible].

Instructor (Stephen Boyd):Yeah. It's 500 microseconds, okay. Now just to – if it's fine, I think we should probably have like a moment of silence to reflect on what this means. This is unbelievable what this means. That solving a set of 100 linear equations with 100 variables, just the data, there's 10,000 data points. Did I get that right? I mean that's just to describe the matrix, okay. This is not a small think, okay. That goes down in 500 microseconds. That's predicted. And the actual number, by the way, was about, I guess, a millisecond or something, okay.

So these numbers are just amazing. And this is just – this is with all the stupid mat lab overhead and all that kind of stuff. So this is – a mat lab, of course, has nothing whatsoever to do with the actual algorithms that we run. The algorithms actually run in mat lab. They do all the numerics all open source public domains called LA Pac. So don't ever confuse the two.

Okay, so these are just amazing numbers, but they are. So it's just worth – I mean I know everyone knows these things or actually, I know most people don't, by the way. If I go around and hang out with other professors who do stuff that involves like – you'd be shocked at the numbers people would guess like this. I mean well respected people I won't name. I've had people guess recently things like minutes for this, so minutes for a thousand, right.

So, of course, it's not their fault. If you kind of don't pay any attention and a decade goes by, Moore's law propagated by one decade, you know, I'm sure I've done that too. I could say, "Oh, boy. You could probably do that in about a minute." And anyway, you can say that in a couple of classes and you're wrong pretty quickly.

So okay, all right. Back to this. Okay, that's [inaudible]. Now here's the part that's important. I mean this is basically – you should get a rough idea. Everyone should know these numbers. There's no excuse not to. It's silly. Now here's the really cool part and here's actually what we're gonna cover in the next lecture, and this is very important to know. It makes these numbers even more shocking, way more shocking.

If A is structured like banded sparse couplets and we're gonna see a lot of other types, some that people would not recognize, most people, okay. In that case, it's way, way faster. Now, for example, let's just do a quick example. Suppose A is tridiagonal. Anybody know? Someone who's done numerical computing will know how fast you can solve a tridiagonal system. How fast?

Student: It's institution.

Instructor (Stephen Boyd):It's actually N.

Student: Oh, N.

Instructor (Stephen Boyd):It's N if it's banded, it's N. So what that means is, for example, solving a million equations with a million unknowns with a banded matrix, it's like that, okay. I mean these are not secrets or anything, okay. The people who do this all the time know it, actually, but other people should know it. You should know it. There's no reason for you not to know this kind of stuff, okay.

Actually, what will really matter is then we'll make all the neural connections between the application and these things and then you'll look at something like an optimal control problem or signal processing. You'll look at it. All the neural connections will go through and you can say, "That problem I can solve with a million variables, that signal processing problem. That image restoration problem, I can solve that with a lot of variables because blah, blah, blah banded or such and such." So we'll get to all these things.

Okay, so how do you estimate the – an algorithm for solving AX = B? Well, this is really from the 60s, so it's a bit sort of behind, but it's still a useful concept, although you shouldn’t take it too carefully. It’s a flop count. So flop count is something like a floating point operation. It means it’s something like this. It depends on how you do the accounting, and, in fact, it doesn’t really matter anymore because it’s not that accurate. But it basically means something like an addition, subtraction, right, something like an addition, a subtraction, a multiplication. And it also – something they throw in something like they amortize a little bit of indexing or something like that. So it’s just a very crude number.

By the way, it was not a crude number in the 60s and 70s and 80s, maybe early 80s. It was not a crude number because a floating point operation would actually often be like shipped off to a separate chip or something like that. I mean it was like a big deal to do a floating point operation. So counting floating point operations then made a lot of sense and actually would give you a pretty accurate idea of how long something would run.

These days, of course, it’s really close to meaningless. I mean it’s kind of silly. What matters now, obviously, is things like cache misses, locality of reference in memory. Things are deeply pipelined. All sorts of stuff is going down at the same time. So it – floating point, I mean, you can write down two algorithms, analyze the number of floating point operations, and they can run, let’s say, 1,000 times off in speed.

This, by the way, hints that floating point – counting floating point operations is not such a super good measure anymore, that they could be off by a factor of 1,000. Still, it’s roughly useful, but it means don’t take a number seriously. If someone walks up to you on the street and says, “I have a new algorithm, and it’s – you know, the usual one is like 1/3 N cubed. Mine is like .2 N cubed.” Then you can just laugh and walk away or something because it’s, I mean, so far off below the threshold of significance, it doesn’t matter.

Okay, so here’s what people do. Again, it’s from the 60s and it’s not quite right, but it’s still a useful idea. What you do is you count up roughly the number of operations required to do something like solve AX = B. And that will generally, in the problems we’re interested in, it will be a polynomial function of the problem dimensions. You may be estimating stuff, that’s fine. And you simplify by keeping only the leading terms.

So you’d say, “That’s an N cubed algorithm, or that’s an N to the 2.5 algorithm, or it’s fan, or linear, it would be – it’s just order M.” Okay, so as I said, this is not an accurate predictor of computation on modern computers, although, by the way, that’s what I was doing here. Later, I’ll actually tell you exactly what was happening in these two cases. And actually, the fact that I was predicting within a factor of two or three is pretty good, but actually only because it was all optimized, but we’ll get to that later. But it’s still useful as a rough measure, estimate of complexity.

Okay, so some of this is pretty basic, and I should mention something about this. I think everyone should know about it. We’re not gonna get into this, but it’s just to give you a glimpse of below where we’re gonna go in the class at the lower level stuff. Actually, all of this, the linear algebra we're talking about now is done in something call LA Pac, which is linear algebra package which is open source software.

It's among the most debugged software anywhere in the world because it's used by lots and lots and lots of smart people. It's written by very smart people, totally open source, and it's basically what – it's everything you do if you use Mat Lab, Octave, anything else, it's gonna use LA Pac. We'll get into some others, so, and I presume that's true of R as well, but I don't. Do you know? Yeah, there we go. So it's R as well.

Okay, so this is what – and basically below LA Pac is something lower called blast. Again, this is just – you know, it's not part of this class, but it's just if you want to look below the level we're gonna go, this is – these things are very good to know about. Blast is basic linear algebra subroutines, okay. And they distinguish blast level one, two, and three. So blast one, blast two, blast three. You'll hear things like this.

If you don't hear things like this, you're probably not hanging out with the right people, by the way, just let – so if you hear – if most of your friends are talking about squared in, login, complexity, you should at least get some friends that talk about these things just to kind of get a more balanced view of the world.

Okay, so blast level one is generally like things like vector vector operations and things like that. So these are separated logically this way. So these are actually blast level one. So here this would be things like calculating inner product. Well, how many flops does that take? Well, you have to multiply the pairs and then add them all up, and it's 2N – 1 and people just say, you know, 2N or they just say N. I mean, and by the way, some people count flops a factor of two off. So some people say a flop is actually kind of a multiplication and an addition plus a little bit of indexing.

So basically the usage of the word flop is inconsistent by a factor of two immediately, which is fine because it doesn't predict anything more than a factor of two anyway now, nowadays. So you would just say this is order. You would just say this is N. If you add two vectors, I mean, this is silly. It's N flops. What you want to start thinking about here is what happens when there's structure. There's not much structure you can have in a vector.

Well, there is one. You can have a sparse vector. If you had a sparse vector, then how would you calculate the inner product between two? Suppose my vectors are each 100 million long, but each of them only has 1,000 non-zeros stored in some reasonable data structure, right. I mean let's just make this simple. We're not gonna do an exotic one. How would you calculate the inner product between two?

You wouldn't ask each one in turn, "Please can I have XI? Please can I have YI? Multiply the numbers, and then plus equals that onto your accumulator or whatever, right?" You would not do that. What would you do?

Student: Fix it where I'd [inaudible].

Instructor (Stephen Boyd):Yeah. You find the common ones. So basically it would be boising fast. But as a sample, I mean a stupid, simple example of where structure comes to make not a small – we're not talking about small increases here, right. We're talking about dramatic increases in efficiency, right.

Student: How does knowing something's structure help you in actually doing these computations? Doesn't it take some work to actually figure out what's zero?

Instructor (Stephen Boyd):Right, that's an excellent question. Okay, and we're gonna get to exactly that question, excellent. If I don't answer it today sensibly, I'll answer it on Thursday. The question was, you know, how does knowing something's structure help. I'll tell you the answer right now roughly. It may not make much sense. It's this. Some structure will – is actually – can be, in some cases, automatically handled for you. Some types of sparcity patterns and things like that will be automatically recognized, depending on the software system you're using.

If you're writing sort of high level stuff, you know, like in Python or Mat Lab, some structure will be recognized. Other structure won't be, okay. And that's the one that will be the interesting part. If you're writing actually production code or stuff that actually works for something large, it – knowing the structure is there helps you this way because now guess who's gonna write the code for it? You. So we'll get to that.

Now the low level stuff, you don't have to do. A lot of the structures though will already be things they can do. That's just – sparcity is not one of them. There's no absolute standard now, but for a lot of other things there will be. There's actually code low level, so you should know about it. All right, let's move on.

Matrix vector multiply. Here if A is dense or you treat it as dense, then you want to do a matrix vector multiply. You just work out how many. It's basically 2MN flops because you're basically calculating interproduct with each row of an index, and that's what it is. It's that. If A is sparse, if a matrix is sparse, and it's got a round capital N non-zero elements, then I mean that's all you're gonna have to multiply by here because you're gonna just calculate those and that gives you this.

Now if A is given a different data structure, like, for example, suppose A is given as a low rank factorization. Suppose A is – it doesn't matter. A could be like a million by million, but I'd give it as a rank 10 factorization. Note, that would allow me to store A because there's no way you're storing a million by million dense matrix anyway. You can't do it.

But I could store A as two million by ten matrices. That's nothing. And here the key is not – is to store it like this and to multiply A by X by first calculating B transpose X, which gives you ten numbers, and then forming U times X, which gives you million. And that's fast. That's gonna be 2P times N. In this case, if that was a million, this would be something on the order of 10, 20 million flops, which, as you know now, goes down way, way fast as opposed to something that just wouldn't work even remotely.

By the way, to go back to your question about knowing structure and so on, there is no system in the world that will recognize for you low rank structure. So if you, anywhere in your code, write something equals U times B like this, there's nothing, there's no system right now that would check and say, "Silly guy, that's actually rank 10." Anyway, it wouldn't work anyway here because if you formed U times V where there are a million by million, it's all over anyway. It's just nothing. It'll stop when there's no more virtual memory left or something like – it'll just stop at some point.

So low rank structure is an example of structure that is not discoverable. A sparcity is sort of discoverable and have generic methods that do it, so okay. Matrix product, that's C = AB. Now here the number's approximately MPN. So it's the product of the sizes like this. That's what it is if it's large. Now if A or B is sparse, it can be much, much less.

By the way, this is – it's not at all obvious, actually, when you have sparse matrices. If you really want to work out good data structures for storing and manipulating sparse matrices, it's quite complicated. And a lot of the data structures you think of immediately are not that great. I mean the most obvious one a person might think of would be a key val pair, triplets, well, key val pairs, and it would look like this. That's how you might imagine is a whole bunch of things like this to be key and then val paired, you know, something like that. That's how you might imagine. They're probably the most natural for a person.

Seems like that's a very poor data structure for this, and so the ones that are actually used vary from simple, but sort of practical. That would be – you'll hear column compressed. What's the last term on that? Column compressed format, or something like that. You get column compressed format, and it's just some weird data structure that is kind of like a data structure that's a cross between simple and close enough to machines that you get something that's – but then you get all sorts of exotic data structures for storing sparse matrices, which I won't go into. But if you look at this area, you'd find out about them, okay.

Actually, this is a good time for me to point something out about this because it's a very good thing to know. Let's take matrix matrix multiply. Okay, everybody here knows how to write a code for this like in C. It's nothing, right? All you do is you take A and B, any language, doesn't matter. You take A and B, and you're gonna calculate CIJ, or let's do this. You say for I = 1 to N, for K = let's make them all end by N because I don't really care. Values 1 to N and for K = 1 to N, you're gonna calculate CIK. And so you simply walk across A, down B, and add these up.

So you get like six, eight lines of C, something like – everyone could write that right here, right. And you compile it and you will have a little function called [inaudible] A, B. Everybody got it? And it will multiply matrices, you know. It's not exactly a sophisticated computation you're carrying out, right?

Okay, so now let me ask this. I know I mentioned this is 263 if you took that class, but let me ask you. Can you imagine, there doesn't look like there's a whole lot of room for creativity in the [inaudible] thing. I mean if you were like scanning some code and wanted to know where to optimize something. You probably wouldn't take [inaudible] and the ensuing like five lines of C, two of which are comments, as something that we would optimize. I mean I'm just – does this sound reasonable to everybody?

Yeah, so here's the weird part. If you use the LA Pac version of this, it could easily be ten times faster than yours and could definitely be 100 times faster. Now, by the way, there's no structure being exploited. Total number of flops that go down in each case, identical. It's whatever it is. In a two man matrices, it's like N cubed or something, okay. Everybody, you have to let this sink in what I just said because it's ridiculous. Everybody see what I'm saying? Now I'm gonna ask you this. This is just a fun question. Why? What does LA Pac do that you didn't in your six line C program?

Student: Block.

Instructor (Stephen Boyd):It blocks, okay. So, in fact, what LA Pac did is it will actually break this up into blocks and then multiply blocks at the same time. Why does that work? Because some blocks are optimized. The size is optimized perfectly to do register register calculations, okay. And the next block is optimized for your level one cache and so on, so forth, okay.

By the way, if you don't know what I'm talking about, that's fine, but whether you know what I'm talking about or not, you must know the following, that when you get into numerical computing, there's things that just look like, you know, if you don't know about this and someone tells you that there's a good and a bad version of something that multiplies two matrices, right. It's just not obvious that it even makes any sense, but it's absolutely the case on modern computers, so that's how it works, so okay. And I'll say more about that later, but that's – this is the first thing that maybe if you don't know about this field, this is not obvious.

Okay, so let's talk about linear equations that are easy to solve. We'll start there. So well, look, if A is diagonal, it's very easy to solve AX = B because you just simply divide by the diagonal entries, so that's nothing, and that takes M flops, okay. If a matrix is lower triangular, that's the first one that's not totally obvious. That's, you know, it looks like this, A11, A21, A22, A31, A32, A33. You know, this thing times X1, X2, X3 equals B1, B2, B3, like that, okay.

How do you solve it? Well, you first go after X1 because this first equation is A11, X1 equals B1. You divide and get it. Once you know X1, you go to the second line and it says A3, A21, X1, but you just calculated X1, so you substitute it back in there, okay. So you actually calculate X1 and then you substitute it forward. This is called the forward substitution algorithm, okay. And what happens in this case is you successively calculate X1, X2, X3, X4. Once you've gotten down to here and you know everybody above here, it means that you know these guys, you multiplied by these, and that's just a number. And it goes on the other side and you divide, so it's very simple.

By the way, this thing works for block methods as well. These could be – the A's could be block lower triangular. You could also do – now, I can't use this. This code wouldn't work, but had I written this the right way, as A22 inverse here. Then my notation would have overloaded correctly in the block case, okay.

So this is – now if you do this if it's dense, this takes N squared flops with – this takes N squared flops here to work because you work out. It's basically, you know how these work. It takes like one flop. Then it takes three, then five, then seven. And you add those numbers up and you get N squared, okay, because you go, you keep adding anyway.

I'm sure you know how that works, okay. By the way, if A is sparse, then this forward substitution basically depends on the number of non-zeros in A, very, very fast, okay, if A is sparse. So forward substitution becomes fast. If a matrix is upper triangular, you just solve from the last element up and that's called backwards substitution, okay.

Well, there's lots of other. I mean you can have an orthogonal matrix. If a matrix were orthogonal, then basically the inverse is transposed, so you multiply by the transpose, and that's for general. But, in fact, often in orthogonal matrix is stored with a different data structure.

One common data structure, in fact, would be a product, just for example, of matrices of this form. That's a very common data structure used to describe an orthogonal matrix. And this – you have norm U equals 1. If you do that, then to multiply A transposed here by a vector, by the way, that's a perfect example of a matrix vector multiply.

This matrix, of course, in general, is dense, right. So if you form this outer product, it's gonna have – it's gonna spray non-zeros over the whole matrix, right. So that'll be a dense matrix and the matrix vector multiply will cost you order and squared. I think I got that right, okay. But here, if you exploit the fact that this is identity plus rank one and calculate it this way when you're past a vector, you first multiply by U. You calculate an inner product, then multiply by U, and then subtract that from your original one. It's an order N.

So matrix vector multiply for this structure of matrix is gone down from N squared to N here, okay. Another example would be a permutation matrix and it's actually – this is a great one. It's interesting. If you have a permutation matrix, so in the traditional 1960s term, you'd say there's no flops because you don't do any floating point calculation. You're just doing index tricking.

The funny part is actually nowadays this would be – this is exactly what you don't want. This is – I mean if you're actually calculate, it's like data movement and stuff like that. So this – in modern times, this would be a relatively expensive operation and it runs up according to the metric we're using zero bill because it's permutation.

So, okay, now I can tell you about the big picture of how you solves AX = B. Well, the general scheme is this. You take the matrix A and you factor it as a product of simple matrices. These can be diagonal permutations, orthogonal, upper triangular, lower triangular, block upper triangular, and these kinds of things. So you factor it this way. And then you compute A inverse B as the product of these inverses in reverse order like that, okay. You multiply it by B.

So you actually – let me just put the parenthesis in the right place. You do this, then that, and so on, up to that, okay. So you factor and then you solve from the – you solve actually from the first one backwards is what you do, something. I think I have it right, something like that. That's how you do it because each of these operations you can do fast, so that's how you do this.

Okay, so and in each case you – by the – people write things like A inverse B because that's what you write in math, but often, when people say, "Compute A inverse B," that's just sort of understood to mean no that you actually compute A inverse and multiply it by A. It means that you call a method, which is something like solve A, B. But it's just weird to write that, so people write A inverse B, and it's simply known that you wouldn't actually compute it.

You wouldn't take the pseudo code literally and actually form A inverse multiplied by B, okay. So, okay, and actually, once you know this, you already know something that is not obvious. Well, you already know one other thing that's not obvious, that basically LA Pac will beat the hell out of your six line C program or can. That's another thing that's, I believe, not obvious to multiple two matrix. Do something as stupid as multiply two matrices, LA Pac properly configured and optimized will beat you like by a wide, possibly very wide margin.

Okay, second thing that's not obvious, again, if you don't know this material, and it has huge consequences, and it's this, and it's really very cool. It's this. Suppose you want to solve a set of equations with the same matrix, but multiple right-hand sides, okay. That comes up like all over the place. Now those, for example, I don't know. AX = B really represents solving a set of circuit equations. B is a set of current sources injected into a circuit. X is the set of potentials that at the nodes in the circuit. So that's what AX = B means.

All right. In that case, it says suppose you want to – and basically AX equals – solving X = B is doing a circuit – is a circuit simulator. It basically says, "Inject these current. If I inject these currents in the circuit, please tell me what the potentials would be at the nodes." That's what solving X = B is. It's a sim.

Now once you know about factor solve methods, you know the following. If you have multiple sims, so you're saying, "You know what? I would like to know what's the potential distribution in the circuit with this current excitation and also this one and also this one and that one too, okay." The naïve way is simply to call the sim code four times, okay. But, in fact, that's silly because the factorization on the matrix only needs to be done once.

So once you have actually factored a matrix into easily solved systems, the – that is amortized across all the solves you do. And so it is just a totally weird thing. You can go and do this is mat lab, although just because it's using a LA Pac. You can do the most amazing thing. You can just go and you can time that. A has to be reasonable, otherwise it's too fast and the interpreted overhead. You know, so you could time that and then you could time this.

What's the time difference gonna be for dense matrices? I can tell you that the factor [inaudible] and the back solve you now know is N squared. What's the time difference between these two gonna be? What's it gonna do? It will actually do the right thing here. What will it do?

It will factor A once and then do two back substitutions, back solves, one with B1 and one with B2. So this is an order N cubed algorithm. Actually, or I should say quite specifically, it's N cubed plus something N squared. And this would be N cubed plus like 2N squared. What's the difference between these two? They're the same and you will, in fact, find that. So it's the most ridiculous thing.

If someone walks up to you on the street and says, "I need to solve once set of 1,000 variables, 1,000 equations." And they say, "You know what? Actually, I have ten. Same set of equations, ten right-hand sides. Please solve it for me." Guess what the time is? Identical. It's zero. It's insignificant. It's the same. So again, this is not obvious, but now you know it, and it has to do with the factor solve method. I mean this will be more clear when we get to actually how this is done, but that's it.

These things are just like not – you know, if someone tells you that it's – to calculate A inverse B1 up to A inverse BK, is that then if K is less than or less than N, like if it's order less than N. In this case, like if it's ten, then this – the cost is actually identical to calculating just one. It's weird. These are not – I mean, yeah, these are not obvious things and I think a lot of people don't know them. It just kind of doesn't make any sense.

So the cost of multiple solves is one factorization plus M solves, okay. We'll get to that. We'll see. Okay, now let's talk about the factorizations. I won't get very far, but I'll say a little bit about it. The most famous factorization actually just comes from gaucian elimination and it's this. Any non-singular matrix you can factor into a permutation, a lower triangular, and an upper triangular matrix period. And, in fact, if you want, you can even insist that the – either the L or the U has ones on the diagonal.

So this permutation – I mean this factorization is by no means unique, nor, by the way, are we gonna look at how it's done. If you take the ten week version of the thing we're gonna cover in a lecture in a quarter, you would know all the details of how you actually calculate this. Now the permutation matrix is absolutely required. So I can actually easily make up a matrix A. I shouldn't have said that because I actually can't. If I had to do it right now, I probably would fail. No, I might be able to do it, but I'm not gonna try.

You can easily make up a matrix A that's non-singular, but which cannot be written as LU. So you may have to permute, in this case rows, to make it – so the P here is not an extra thing. Actually, the P, if you take one of these classes on numerical methods, the P is chosen for two things. First of all, there's the mathematical issue that it's just not true that all non-singular matrices have an LU factorization without the P, number 1. Number 2, it's chosen so that round-off errors obtained when you calculate these things don't kill you. So that's the real reason that the P is used, but okay.

Now the cost of that factorization is 2/3 N cubed flops. Okay, so here's how you solve equations and it's roughly – actually, I mean it's not roughly. This is kind of – this is sort of what mat lab does by calling LA Pac codes, I might add. It works like this. Yeah, the X = B. A is non-singular. So it first carries out a PLU, but it's called an LU factorization. The P is understood. So if someone says it's an LU factorization, they mean PLU.

So an LU – you do an LU factorization, that's 2/3 N cubed flops. And then what you do is you – well, you start by solving P. You permute the right-hand side. Then you do forward substitution and backwards, and each of these cause N squared flops. So this is negligible, well, except if N is like six or eight. But then, in that case, these things are – different rules come into play when N is six or eight.

So, but you know if N is whatever, 100 or 50 or more, then it's just – basically it's just order N cubed. So the cost of solving, now you know the following. For dense equations, the cost of solving one set of linear equations is N cubed. As a single data point, my laptop last night, N = 1,000, half a second, okay.

And the cost of solving eight sets of linear equations with the same coefficient matrix would be on my laptop what? Yeah, it'd be basically unmeasurably – it would be you couldn't measure the difference. It would be the same. And the reason is you do one factorization and then once you've got the factorization, you do back solves ten times or something like that.

So, okay, so I think we'll quit here and then continue on Thursday. Let me remind you. The section that would have been yesterday is tomorrow.

[End of Audio]

Duration: 77 minutes