JAMES SWAN: OK. Should we begin? Let me remind you, we switched topics. We transitioned from linear algebra, solving systems of linear equations, to solving systems of nonlinear equations. And it turns out, linear algebra is at the core of the way that we're going to solve these equations.

We need iterative approaches. These problems are complicated. We don't know how many solutions there could be. We have no idea where those solutions could be located. We have no exact ways of finding them. We use iterative methods to transform non-linear equations into simpler problems, right? Iterates of systems of linear equations.

And the key to that was the Newton-Raphson method. So I'm going to pick up where we left off with the Newton-Raphson method, and we're going find out ways of being less Newton-Raphson-y in order to overcome some difficulties with the method, shortcomings of the method. There are a number of them that have to be overcome in various ways.

And you sort of choose these so-called quasi- Newton-Raphson methods as you need them. OK, so you'll find out. You try to solve a problem. And the Newton-Raphson method presents some difficulty, you might resort to a quasi Newton-Raphson method instead.

Built into MATLAB is non-linear equations solver. fsolve. OK, it's going to happily solve systems of nonlinear equations for you, and it's going to use this methodology to do it. It's going to use various aspects of these quasi- Newton-Raphson methods to do it. I'll sort of point out places where fsolve will take from our lecture and implement them for you.

It will even use some more complicated methods that we'll talk about later on in the context of optimization . Somebody asked an interesting question, which is how many of these nonlinear equations am I going to want to solve at once? Right?

Like I have a system of these equations. What does a big system of nonlinear equations look like? And just like with linear equations, it's as big as you can imagine. So one case you could think about is trying to solve, for example, the steady Navia-Stokes equations. That's a nonlinear partial differential equation for the velocity field and the pressure in a fluid.

And a at Reynolds number, that non-linearity is going to present itself in terms of inertial terms that may even dominate the flow characteristics in many places. We'll learn ways of discretizing partial differential equations like that. And so then, at each point in the fluid we're interested in, we're going to have a non-linear equation that we have to solve.

So there's going to be a system of these non-linear equations that are coupled together. How many points are there going to be? That's up to you, OK? And so you're going to need methods like this to solve that. It sounds very complicated. So a lot of times in fluid mechanics, we have better ways of going about doing it. But in principle, we've have any number of nonlinear equations that we want to solve.

We discussed last time, the new Newton-Raphson method, which was based around the idea of linearization. We have these nonlinear equations. We don't know what to do with them. So let's linearize them, right?

If we have some guess for the solution, which isn't perfect, but it's our best possible guess. Let's look at the function and find a linearized form of the function and see where that linearized form has an intercept. And we just have an Ansatz. We guess that this is a better solution than the one we had before. And we iterate.

It turns out you can prove that this sort of a strategy-- this Newton-Raphson strategy is locally convergent. If I start with a guess sufficiently close to the root, you can prove mathematically that this procedure will terminate with a solution at the root, right? It's going to approach after an infinite number of iterates, the root.

That's wonderful. It's locally convergent, not globally convergent. So this is one of those problems that we discussed. Take a second here, right? Here's your Newton-Raphson formula. You've got it on your slides. Take a second here and-- this is sort of interesting. Derive the Babylonian method, right?

Turns out the Babylonians didn't know anything about Newton-Raphson but they had some good guesses for how to find square roots, right? Find the roots of an equation like this. See that you understand the new Newton-Raphson method by deriving the Babylonian method, right? The iterative method for finding the square root of s as the root of this equation. Can you do it?

[SIDE CONVERSATION]

JAMES SWAN: Yes, you know how to do this, right? So calculate the derivative. The derivative is 2x. Here's our formula for the iterative method. Right? So it's f of x over f prime of x. That sets the magnitude of the stop. The direction is minus this magnitude. It's in one d, so we either go left or we go right.

Minus sets the direction. We add that to our previous guess. And we have our new iterate, right? You substitute f and f prime, and you can simplify this down to the Babylonian method, which said take the average of x and s over x. If I'm at the root, both of these should be square root of s. And this quantity should be zero exactly, right? And you'll get your solution.

So that's the Babylonian method, right? It's just an extension of the Newton-Raphson method. It was pretty good back in the day, right? Quadratic convergence to the square root of a number. I mentioned early on computers got really good in computing square roots at one point, because somebody did something kind of magic.

They came up with a scheme for getting good initial guesses for the square root. This iterative method has to start with some initial guess. If it starts far away, it'll take more iterations to get there. It'll get there, but it's going to take more iterations to get there. That's undesirable if you're trying to do fast calculations.

So somebody came up with some magic scheme, using floating point mathematics, right? They masked some of the bits in the digits of these numbers. A special number to mask those bits. They found that using optimization, it turns out.

And they got really good initial guesses, and then it would take one or two iterations with the Newton-Raphson method to get 16 digits of accuracy. That's pretty good. But good initial guesses are important. We'll talk about that next week on Wednesday. Where do those good initial guesses come from?

But sometimes we don't have those available to us. So what are some other ways that we can improve the Newton-Raphson method? That will be the topic of today's lecture. What's the Newton-Raphson method look like graphically in many dimensions. We talked about this Jacobian. Right, when we're trying to find the roots of a non-linear equation where our function has more than one dimension-- let's say it has two dimensions. So we have an f 1 and an f 2.

And our unknowns our x 1 and x 2, they live in the x1 x2 plane, right?

f1 might be this say bowl-shaped function. I've sketched out in red, right? It's three dimensional. It's some surface here. Right? We have some initial guess for the solution. We go up to the function, and we find a linearization of it, which is not a line but a plane. And that plane intersects the x 1, x 2 plane at a line.

And our next best guess is going to live somewhere on this line. Where on this line depends on the linearization of f 2. Right? So we got to draw the same picture for f 2, but I'm not going to do that for you. So let's say, this is where the equivalent line from f 2 intersects the line from f 1, right?

So the two linearizations intersect here. That's our next best guess. We go back up to the curve. We find the plane that's tangent to the curve. We figure out where it intersects. The x 1 x 2 plane. That's a line. We find the point on the line that's our next best guess, and continue.

Finding that intersection in the plane is the act of computing Jacobian inverse times f. OK? If we project down to just the x 1 x 2 plane, and we draw the curves where f 1 equals 0, and f 2 equals zero, right? Then each of these iterates, we start with an initial guess. We find the planes that are tangent to these curves, or to these surfaces. And where they intersect the x 1 x 2 plane.

Those give us these lines. And the intersection of the lines give us our next approximation. And so our function steps along in the x1 and x2 plane. It takes some path through that plane. And eventually it will approach this locally unique solution. So that's what this iterative method is doing, right? It's navigating this multidimensional space, right? It moves where it has to to satisfy these linearized equations, right? Producing ever better approximations for a root.

Start close. It'll converge fast. How fast? Quadratically. And you can prove this. I'll prove it in 1D. You might think about the multidimensional case, but I'll show you in one dimension. So the Newton-Raphson method said, xi plus 1 is equal to xi minus f of xi over f prime of xi.

I'm going to subtract the root, the exact root from both sides of this equation. So this is the absolute error in the i plus 1 approximation. It's equal to this. And we're going to do a little trick, OK?

The value of the function at the root is exactly equal to zero, and I'm going to expand this as a

Taylor series, about the point xy. So f of xi plus f prime of xi times x star minus xi, plus this second order term as well. Plus cubic terms in this Taylor expansion, right? All of those need to sum up and be equal to 0, Because f of x star by definition is zero. x star is the root.

And buried in this expression here is a quantity which can be related to xi minus f of xi over f prime minus x star. It's right here, right? xi minus x star, xi minus x star. I've got to divide through by f prime. Divide through by f prime, and I get f over f prime. That's this guy here.

Those things are equal in magnitude then, to this second order term here. So they are equal in magnitude to 1/2, the second derivative of f, divided by f prime, times xi minus x star squared.

And then these cubic terms, well, they're still around. But they're going to be small as I get close to the actual root. So they're negligible, right? Compared to these second order terms, they can be neglected. And you should convince yourself that I can apply some of the norm properties that we used before, OK? To the absolute value. The absolute values is the norm of a scalar.

So these norm properties tell me that this quantity has to be less than or equal to, right? This ratio of derivatives multiplied by the absolute error in step i squared. And I'll divide by that absolute error in step i squared.

So taking the limit is i goes to infinity, this ratio here is bound by a constant. This is a definition for the rate of convergence. It says I take the absolute error in step i plus 1. I divide it by the absolute error in step i squared. And it will always be smaller than some constant, as i goes to infinity.

So it converges quadratically, right? If the relative error in step i was order 10 to the minus 1, then the relative error in step i plus 1 will be order 10 to the minus 2. Because they got to be bound by this constant.

If the relative error in step i was 10 to the minus 2, the relative error in step i plus 1, has got to be order 10 to the minus 4, or smaller, right? Because I square the quantity down here. I get to double the number of accurate digits with each iteration.

And this will hold so long as the derivative evaluated at the root is not equal to zero. If the derivative evaluated at the root is equal to zero, this analysis wasn't really valid. You can't divide by zero in various places, OK? It turns out the same thing is true if we do the multidimensional case.

I'll leave it to you to investigate that case. I think it's interesting for you to try and explore that. It follows the 1D model I showed you before. But the absolute error in iterate i plus 1, divided by the absolute error in iterate i-- here's a small typo here. Cross out that plus 1, right? The absolute error in iterate i squared is going to be bound by a constant.

And this will be true so long as the determinant at the Jacobian at the root is not equal to zero. We know the determinant of the Jacobian plays the role of the derivative in the 1D case. When the Jacobian is singular, you can show that linear convergence is going to occur instead. So it will still converge. It's not necessarily a problem that the Jacobian becomes singular at the root. But you're going to lose your rate of quadratic convergence.

And this rate of convergence is only guaranteed if we start sufficiently close to the root. So good initial guesses, that's important. We have a locally convergent method. Bad initial guesses? Well, who knows where this iterative method is going to go. There's nothing to guarantee that it's going to converge even. Right It may run away someplace.

Here are a few examples of where things can go wrong. So if I have a local minima or maxima, I might have an iterate where I evaluate the linearization, and it tells me my next best approximation is on the other side of this minima or maxima.

And then I go up, and I get the linearization here. And it tells me, oh, my next best approximation is on the other side. And this method could bounce back and forth in here for as long as we sit and wait. It's locally convergent, not globally convergent. It can get hung up in situations like this.

Asymptotes are a problem. I have an asymptote, which presumably has an effective root somewhere out here at infinity. Well, my solution would like to follow the linearization, the successive linearizations all the way out along this asymptote, right? So my iterates may blow up in an uncontrolled fashion.

You can also end up with funny cases where our Newton-Raphson steps continually overshoot the roots. So they can be functions who have a power loss scaling right near the root, such that the derivative doesn't exist. OK?

So here the derivative of this thing, if s is smaller than 1, and x equals zero, it won't exist, right? There isn't a derivative that's defined there. And in those cases, you can often wind up with overshoot. So I'll take a linearization, and I'll shoot over the root. And I'll go up and I'll take my next linearization, I'll shoot back on the other side of the root.

And depending on the power of s associated with this function, it may diverge, right? I may get further and further away from the root, or it may slowly converge towards that root. But it can be problematic.

Here's another problem that crops up. Sometimes people talk about basins of attraction. So here's a two-dimensional, non-linear equation I want to find the roots for. It's cubic in nature, so it's got three roots, which are indicated by the stars in the x1 x 2 plane. And I've taken a number of different initial guesses from all over the plane and I've asked-- given that initial guess, using the Newton-Raphson method, which root do I find?

So if you see a dark blue color like this, that means initial guesses there found this root. If you see a medium blue color, that means they found this root. See a light blue color, that means they found this root. And this is a relatively simple function, relatively low dimension, but the plane here is tilled by-- it's not tiled. It's filled with a fractal. These basins of attraction are fractal in nature.

Which means that I could think that I'm starting with a solution rate here that should converge to this green root because it's close. But it actually goes over here. And if I change that initial guess by a little bit, it actually pops up to this root over here instead.

It's quite difficult to predict which solution you're going to converge to.

Yes?

AUDIENCE: And in this case, you knew how many roots there are.

JAMES SWAN: Yes.

AUDIENCE: Often you wouldn't know. So you find one, and you're happy. Right? You're happy because [INAUDIBLE] physical. Might be the wrong one.

JAMES SWAN: So this the problem. I think this is about the minimum level of complexity you need. Which is not very complex at all in a function to get these sorts of basins of attraction. Polynomial equations are ones that really suffer from this especially, but it's a problem in general. You often don't know.

I'll show you quasi Newton-Raphson methods that help fix some of these problems. How about other problems? It's good to know where the weaknesses are. Newton-Raphson sounds great, but where are the weaknesses?

Let's see. The Jacobian-- might not be easy to calculate analytically, right? So far we've written down analytical forms for the Jacobian. We've had simple functions. But maybe it's not easy to calculate analytically. You should think about what are the sources for this function, f of x, that we're trying to find the roots for.

Also we got to invert the Jacobian, and we know that's a matrix. And matrices which have a lot of dimensions in them are complicated to invert. There's a huge amount of complexity, computational complexity, in doing those inversions. It can take a long time to do them.

It may undesirable to have to constantly be solving a system of linear equations. So might think about some options for mitigating this. Sometimes it won't converge at all? Or to the nearest root. This is this overshoot, or basin of attraction problem.

And we'll talk about these modifications to correct these issues. They come with a penalty though. OK? So Newton-Raphson was based around the idea of linearization. If we modify that linearization, we're going to lose some of these great benefits of the Newton-Raphson method, namely that it's quadratically convergent, right?

We're going to make some changes to the method, and it's not going to converge quadratically anymore. It's going to slow down, but maybe we'll be able to rein in the method and make it converge either to the roots we want it to converge to or converge more reliably than it would before. Maybe we'll be able to actually do the calculation faster, even though it may require more iterations. Maybe we can make each iteration much faster using some of these methods. OK so here are the three things that we're going to talk about. We're going to talk about approximating the Jacobian with finite differences. We're talking about Broyden's method for approximating the inverse of the Jacobian. And we're going to talk about something called damped Newton-Raphson methods. Those will be the three topics of the day.

So here's what I said before. Analytical calculations of Jacobian requires analytical formulas for f. And for functions of a few dimensions, right? These calculations are not too tough. For functions of many dimensions, this is tedious at best. Error prone, at worst.

Think about even something like 10 equations for 10 unknowns. If your error rate is 1%, well, you're shot. There's a pretty good chance that you missed one element of the Jacobian. You made a mistake somewhere in there. And now you're not doing Newton-Raphson You're doing some other iterative method that isn't the one that you intended.

There are a lot of times where you-- maybe you have an analytical formula for some of these f's, but not all of them. So where can these functionalities come from? We've seen some cases, where you have physical models. Thermodynamic models that you can write down by hand. But where are other places that these functions come from? Ideas?

AUDIENCE: [INAUDIBLE]

JAMES SWAN: Oh, good.

AUDIENCE: [INAUDIBLE]

JAMES SWAN: Beautiful. So this is going to be the most common case, right? Maybe you want to use some sort of simulation code, right? To model something. It's somebody else's simulation code. They're an expert at doing finite element modeling. But the output is this f that you're interested, and the input to the simulation are these x's. And you want to find the roots associated with this problem that you're solving via the simulation code, right?

This is pretty important being able to connect different pieces of software together. Well, there's no analytical formula for f there. OK? You're shot. So it may come from results of simulations. This is extremely common. It could come from interpretation of data. So you may have a bunch of data that's being generated by some physical measurement or a process, either continuously or you just have a data set that's available to you.

But these function values are often not, they're not things that you know analytically. It may also be the case that, oh, man, even Aspen, you're going to wind up solving systems of nonlinear equations. It's going to use the Newton-Raphson method. Aspen's going to have lots of these formulas in it for functions.

Whose going in by hand and computing the derivatives of all these functions for aspen? MATLAB has a nonlinear equation solver in it. You give it the function, and it'll find the root of the equation, given a guess. It's going to use the Newton-Raphson method.

Whose computing the Jacobian for MATLAB? You can. You can compute it by hand, and give it an input. Sometimes that's a really good thing to do. But sometimes, we don't have that available to us. So we need alternative ways of computing the Jacobian. The simplest one is a finite difference approximation.

So you recall the definition of the derivative. It's the limit of this difference, f of x plus epsilon minus f of x divided by epsilon, as epsilon goes to zero. There's an error in this approximation for the derivative with a finite value for epsilon, which is proportional to epsilon.

So choose a small value of epsilon. You'll get a good approximation for the derivative. It turns out the accuracy depends on epsilon, but kind of in a non-intuitive way. And here's a simple example. So let's compute the derivative of f of x equals e the x, which is e to the x. Let's evaluate it at x equals 1.

So

F prime of 1 is e the 1, which should approximately be e to the 1 plus epsilon minus e to the 1 over epsilon. And here I've done this calculation. And I've asked, what's the absolute error in this calculation by taking the difference between this and this, for different values of epsilon.

You can see initially, as epsilon gets smaller, the absolute error goes down in proportion to epsilon. 10 to the minus 3, 10 to the minus 3. 10 to the minus 4, 10 to the minus 4. 10 to the minus 8, 10 to the minus 8. 10 to the minus 9. 10 to the minus 7. 10 to the minus 10. And 10 to the minus 6. So it went down, and it came back up. But that's not what this formula told us should happen, right? Yes?

AUDIENCE: So just to be sure. That term in that column on the right?

JAMES SWAN: Yes?

AUDIENCE: It says exponential 1, but it represents the approximation?

JAMES SWAN: Exponential 1 is exponential 1. f prime of 1 is our approximation here.

AUDIENCE: Oh, OK.

JAMES SWAN: Sorry that that's unclear. Yes, so this is the absolute error in this approximation. So it goes down, and then it goes up. Is that clear now? Good. OK, why does it go down? It goes down because our definition of the derivative says it should go down.

At some point, I've actually got to do these calculations with high enough accuracy to be able to perceive the difference between e to the 1 plus, 10 to the minus 9, and e to the 1. So there is a truncation error in the calculation of this difference that reduces my accuracy at a certain level.

There's a heuristic you can use here, OK? You want to set this epsilon when you do this finite difference approximation, to be the square root of the machine precision times the magnitude of x, the point at which you're trying to calculate this derivative. So that's, usually we're double precision. So this is something like 10 to the minus 8 times the magnitude of x. That's pretty good. That holds true here. OK? You can test it out on some other functions.

If x is 0, or very small. We don't want a relative tolerance. We've got to choose an absolute tolerance instead. Just like we talked about with the step norm criteria. So one has to be a little bit careful in how you implement this. But this is a good guide, OK? A good way to think about how the error is going to go down, and where it's going to start to come back up. Make sense? Good.

OK, so how do you compute elements of the Jacobian then? Well, those are all just partial derivatives of the function with respect to one of the unknown variables. So partial f i with respect to x j is just f i at x plus some epsilon deviation of x in its j-th component only.

So this is like a unit vector in the j direction, or associated with the j-th element of this vector. Minus f i of x divided by this epsilon. Equivalently, you'd have to do this for f i. You can compute all the columns of the Jacobian very quickly by calling f of x plus epsilon minus f of x over epsilon.

Just evaluate your vector-valued function at these different x's. Take the difference, and that will give you column j of your Jacobian. So how many function evaluations does it take to calculate the Jacobian at a single point? How many times do I have to evaluate my function? Yeah?

AUDIENCE: 2 n.

JAMES SWAN: 2n, right. So if I have n, if I have n elements to x, I've got to make two function calls per column of j. There's going to be n columns in j. So 2n function evaluations to compute the Jacobian at a single point. Is that really true though?

Not quite. f of x is f of x. I don't have to compute it every time. I just compute f of x once. So it's really like n plus 1 that I have to do, right? N plus a function evaluations to compute this thing. I actually got to compute them though. Function evaluations may be really expensive.

Suppose you're doing some sort of complicated simulation, like a finite element simulation. Maybe it takes minutes to generate a function evaluation. So it can be expensive to compute the Jacobian in this way. Just be expensive to compute the Jacobian. How is approximation of Jacobian going to affect the convergence?

What's going to happen to the rate of convergence of our method? It's going to go down, right? It's probably not going to be linear. It's not going to be quadratic. It's going to be some super linear factor.

It's going to depend on how accurate the Jacobian is. How sensitive the function is near the root. But it's going to reduce the accuracy of the method, or the convergence rate of the method by a little bit. That's OK.

So this is what MATLAB does. It uses a finite difference approximation for your Jacobian. When you give it a function, and you don't tell it the Jacobian explicitly. Here's an example of how to implement this yourself. So I've got to have some function that does whatever this function is supposed to do.

It takes as input x and it gives an output f. And then the Jacobian, right? It's a matrix. So we initialize this matrix. We loop over each of the columns. We compute the displacement right? The deviation from x for each of these. And then we compute this difference and divide it by epsilon.

I haven't done everything perfect here, right? Here's an extra function evaluation. I could just calculate the value of the function at x before doing the loop. I've also only used a relative tolerance here. I'm going to be in trouble if xi is 0. It's going to be a problem with this algorithm.

These are the little details one has to pay attention to. But it's a simple enough calculation to do. Loop over the columns, right? Compute these differences. Divide by epsilon. You have your approximation for the Jacobian. I've got to do that at every iteration, right? Every time x is updated, I've got to recompute my Jacobian. That's it though.

All right, that's one way of approximating a Jacobian. There s a method that's used in one dimension called the Secant method. It's a special case of the Newton-Raphson method and uses a coarser approximation for the derivative. It says, I was taking these steps from xi minus 1 to xi. And I knew the function values there.

Maybe I should just compute the slope of the line that goes through those points, and say, that's my approximation for the derivative. Why not? I have the data available to me. It seems like a sensible thing to do. So we replace f prime at x1 with f of xi minus f of x i minus 1.

Down here we put xi minus xi minus 1 up here. That's our approximation for the derivative, or the inverse of the derivative. This can work, it can work just fine. Can it be extended to many dimensions? That's an interesting question, though? This is simple. In many dimensions, not so obvious right?

If I know xi, xi minus 1. f of xi, f of si minus 1. Can I approximate the Jacobian? What do you think? Does it strike you as though there might be some fundamental difficulty to doing that? Yeah?

AUDIENCE: Could you approximate the gradient? [INAUDIBLE] gradient of f at x.

JAMES SWAN: OK.

AUDIENCE: But I'm sure if you whether you can go backwards from the gradient in the Jacobian.

JAMES SWAN: OK. So, let's-- go ahead.

AUDIENCE: Perhaps the difficulty is, I mean when they're just single values--

JAMES SWAN: Yeah.

AUDIENCE: You can think of [INAUDIBLE] derivative, right?

JAMES SWAN: Yeah.

AUDIENCE: [INAUDIBLE] get really big, you get a vector of a function at xi, a vector of a function of xi minus 1 or whatever. Vectors of these x's. And so if you're [INAUDIBLE]

JAMES SWAN: Yeah, so how do I divide these things? That's a good question. The Jacobian-- how much information content is in the Jacobian? Or how many independent quantities are built into the

Jacobian?

AUDIENCE: [INAUDIBLE]

JAMES SWAN: And squared. And how much data do I have to work with here? You know, order n data. To figure out order n squared quantities. This is the division problem you're describing, right? So it seems like this is an underdetermined sort of problem. And it is. OK?

So there isn't a direct analog to the Secant method in dimensions. We can write down something that makes sense. So this is the 1D Secant approximation. That the value of the derivative multiplied by the step between i minus 1 and i is approximated by the difference in the values of the function.

The equivalent is the value of the Jacobian multiplied by the step between i minus 1 and i is equal to the difference between the values of the functions. But now this is an equation for n squared elements of the Jacobian, in terms of n elements of the function, right? So it's massively, massively underdetermined. OK?

Here we have an equation for-- we have one equation for one unknown. The derivative, right? Think about how it was moving through space before, right? The difference here, xi minus 1, that's some sort of linear path that I'm moving along through space. How am I supposed to figure out what the tangent curves to all these functions are from this linear path through multidimensional space, right? That's not going to work.

So there's underdetermined problems. It's not so-- that's not so bad, actually. Right? Doesn't mean there's no solution. In fact, it means there's all a lot of solutions. So we can pick whichever one we think is suitable. And Broyden's method is a method for picking one of these potential solutions to this underdetermined problem.

We don't have enough information to calculate the Jacobian exactly. But maybe we can construct a suitable approximation for it. And here's what's done. So here's the Secant approximation. It says the Jacobian times the step size, or the Newton-Raphson step, should be the difference in the functions.

And Newton's method for x i, said xi minus xi minus 1 was equal-- times the Jacobian, was equal to minus f of xi minus 1. This is just Newton's method. Invert the Jacobian, and put it on the other side of the equation. Broyden's method said, i-- there's a trick here. Take the difference between these things.

I get the same left-hand side on both of these equations. So take the difference, and I can figure out how the Jacobian should change from one step to the next. So maybe I have a good approximation to the Jacobian at xi minus i, I might be able to use this still underdetermined problem to figure out how to update that Jacobian, right?

So Broyden's method is what's referred to as the rank one update. You should convince yourself that letting the Jacobian at xi minus the Jacobian at xi minus 1 be equal to this is one possible solution of this underdetermined equation. There are others. This is one possible solution. It turns out to be a good one to choose.

So there's an iterative approximation now for the Jacobian. Does this strategy make sense? It's a little weird, right? There's something tricky here. You got to know to do this. Right, so somebody has to have in mind already that they're looking for differences in the Jacobian that they're going to update over time.

So this tells me the Jacobian, how the Jacobian is updated. Really we need the Jacobian inverse, and the reason for choosing this rank one update approximation is it's possible to write the inverse of j of xi in terms of the inverse of j at xi minus 1 when this update formula is true.

So it's something called the Sherman Morrison Formula, which says the inverse of a matrix plus the dyadic product of two vectors can be written in this form. We don't need to derive this, but this is true. This matrix plus dyadic product is exactly this. We have dyadic product between f and the step from xi minus 1 to x.

And so we can apply that Sherman Morrison Formula to the rank one update. And not only can we update the Jacobian iteratively, but we can update the Jacobian inverse. So if I know j inverse at some previous time, I know j inverse at some later time too. I don't have to compute these things. I don't have to solve these systems of equations, right?

I just update this matrix. Update this matrix, and I can very rapidly do these computations. So not only do we have an iterative formula for the steps, right? From x 0 to x1 to x 2, all the way up to our converged solution, but we can have a formula for the inverse of the Jacobian.

We give up accuracy. But that's paid for in terms of the amount of time we have to spend doing these calculations. Does it pay off? It depends on the problem, right? We try to solve problems in different ways. This is a pretty common way to approximate the Jacobian. Questions about this? No. OK.

Broyden's method. All right, here's the last one. The Damped Newton-Raphson method. We'll do this in one dimension. So the Newton-Raphson method, Newton and Raphson told us, take a step from xi to xi plus 1 that is this big. xi to xi plus 1, it's this big. Sometimes you'll take that step, and you'll find that the value of the function at xi plus 1 is even bigger than the value of the function at xi.

There was nothing about the Newton-Raphson method that told us the function value was always going to be decreasing. But actually, our goal is to make the function value go to 0 in absolute value. So it seems like this step, not a very good one, right? What are Newton and Raphson thinking here. This is not a good idea. The function value went up.

Far from a root, OK? The Newton-Raphson method is going to give these sorts of erratic responses. Who knows what direction it's going to go? And it's only locally convergent. It tells us a direction to move in, but it doesn't always give the right sort of magnitude associated with that step. And so you take these steps and you can find out the value of your function, the normed value of your functions. It's bigger than where you started. It seems like you're getting further away from the root.

Our ultimate goal is to drive this norm to 0. So steps like that you might even call unacceptable. Right? Why would I ever take a step in that direction? Maybe I should use a different method. When I take a step that's so big my function value grows in norm value.

So what one does, oftentimes, is introduce a damping factor, right? We said that this ratio, or equivalently, the Jacobian inverse times the value of the function, gives us the right direction to step in. But how big a step should we take?

It's clear a step like this is a good one. It reduced the value of the function. And it's better than the one we took before, which was given by the linear approximation. So if I draw the tangent line, it intercepts here. If I take a step in this direction, but I reduce the slope by having some damping factor that's smaller than 1, I get closer to the root.

Ideally we'd like to choose that damping factor to be the one that minimizes the value of the function at xi plus 1. So it's the argument that minimizes the value of the function at xi plus 1 or at xi minus alpha, f over f prime.

Solving that optimization problem, what's hard as finding the root itself. So ideally this is true. But practically you're not going to be able to do it. So we have to come up with some approximate methods of solving this optimization problem. Actually we don't even care about getting it exact. We know Newton-Raphson does a pretty good job. We want some sort of guess that's respectable for this alpha so that we get close to this root.

Once we get close, we'll probably choose alpha equal to 1. We'll just take the Newton-Raphson steps all the way down to the root. So here it is in many dimensions. Modify the Newton-Raphson step by some value alpha, choose alpha to be the argument that minimizes the norm value of the function at xi plus 1.

Here's one way of doing this. So this is called the Armijo line search. See? Line search. Start by letting alpha equal to 1. Take the full Newton-Raphson step, and check. Was the value of my function smaller than where I started? If it is, let's take the step. It's getting us-- we're accomplishing our goal. We re reducing the value of the function in norm. Maybe we're headed towards z. That's good. Accept it.

If no, let's replace alpha with alpha over 2. Let's take a shorter step. We take a shorter step, and we repeat. Right? Take the shorter step. Check whether the value of the function with the shorter step is acceptable. If yes, let's take it, and let's move on. And if no, replace alpha with alpha over 2, and continue. So we have our step size every time.

We don't just have to have it. We could choose different factors to reduce it by. But we try to take shorter and shorter steps until we accomplish our goal of having a function which is smaller in norm at our next iterate than where we were before.

It's got-- the function value will be reduced. The Newton-Raphson method picks a direction that wants to bring the function value closer to 0. We linearize the function, and we found the direction we needed to go to make that linearization go to 0. So there is a step size for which the function value will be reduced.

And because of that, this Armijo line search of the Damped Newton-Raphson method is actually globally convergent, right? The iterative method will terminate. You can guarantee it. Here's what it looks like graphically.

I take my big step, my alpha equals 1 step. I check the value of the function. It's bigger in absolute value than where I started. So I go back. I take half that step size. OK? I look at the value of the function. It's still bigger. Let's reject it, and go back.

I take half that step size again. The value of the function here is now smaller in absolute value. So I accept it. And I put myself pretty close to the root. So it's convergent, globally convergent. That's nice. It's not globally convergent to roots, which is a pain. But it's globally convergent. It will terminate eventually. You'll get to a point where you won't be able to advance your steps any further.

It may converge to minima or maxima of a function. Or it may converge to roots. But it will converge. I showed you this example before with basins of attraction.

So here we have different basins of attraction. They're all colored in. They show you which roots you approach. Here I've applied the Damped Newton-Raphson method to the same system of equations. And you can see the basins of attraction are shrunk because of the damping. What happens when you're very close to places where the determinant of the Jacobian is singular, you take all sorts of wild steps.

You go to places where the value of the function is bigger than where you started. And then you've got to step down from there to try to find the root. Who knows where those locations are? It's a very complicated, geometrically complicated space that you're moving through. And the Damped Newton-Raphson method is forcing the steps to always reduce the value of the function, so they reduce the size of these basins of attraction.

So this is often a nice way to supplement the Newton-Raphson method when your guesses aren't very good to begin with. When you start to get close to root you're, always just going to accept alpha equals 1. The first step will be the best step, and then you'll converge very rapidly to the solution. Do we have to do any extra work actually to do this Damped Newton-Raphson method. Does it require extra calculations?

What do you think? A lot of extra-- a lot of extra calculation? How many extra calculations does it require? Of course it requires extra. How many?

AUDIENCE: [INAUDIBLE]

JAMES SWAN: What do you think?

AUDIENCE: [INAUDIBLE]

JAMES SWAN: It's-- that much is true. So let's talk about taking one step. How many more-- how many more calculations do I have to pay to do this sort of a step? Or even write the multidimensional step?

For each of these times around this loop, do I have to recompute? Do I have to solve the system of equations? No. Right? You precompute this, right? This is the basic Newton-Raphson step. You compute that first. You've got to do it once. And then it's pretty cheap after that. I've got to do some extra function evaluations, but I don't actually have to solve the system of equations.

Remember this is order n cubed. If we solve it exactly, maybe order n squared or order n if we do it iteratively. And the Jacobian is sparse somehow, and we know about it sparsity pattern. This is expensive. Function evaluations, those are order n to compute. Relatively cheap by comparison.

So you compute your initial step. That's expensive. But all of this down here is pretty cheap. Yeah?

AUDIENCE: You're also assuming that your function evaluations are reasonably true.

JAMES SWAN: This is true.

AUDIENCE: [INAUDIBLE]

JAMES SWAN: It's true. Well, Jacobian is also very expensive to compute then too. So, if--

AUDIENCE: [INAUDIBLE]

JAMES SWAN: Sure, sure, sure. No, I don't disagree. I think one has to pick the method you're going to use to suit the problem. But turns out this doesn't involve much extra calculation. So by default, for example, fsolve in MATLAB is going to do this for you. Or some version of this. it's going to try to take steps that aren't too big. It will limit the step size for you, so that it keeps the value of the function reducing in magnitude. It's a pretty good general strategy. Yes?

AUDIENCE: [INAUDIBLE] so why do we just pick one value for [INAUDIBLE]

JAMES SWAN: I see. So why-- ask that one more time. This is a good question. Can you say it a little louder so everyone can hear?

AUDIENCE: So why, instead of having just one value of alpha and not having several values of alpha [INAUDIBLE]

JAMES SWAN: I see. So the question is, yeah, we used a scalar alpha here, right? If we wanted to, we could reduce the step size and also change direction. We would use a matrix to do that, instead, right? It would transform the step and change its direction. And maybe we would choose different alphas along different directions, for example. So diagonal matrix with different alphas.

We could potentially do that. we're probably going to need some extra information to decide how to set the scaling in different directions. One thing we know for sure is that the Newton-Raphson step will reduce the value of the function. If we take a small enough step size, it will bring the value of the function down.

We know that because we did the Taylor Expansion of the function to determine that step size. And that Taylor expansion was going to be-- that Taylor expansion is nearly exact in the limit of very, very small step sizes. So there will always be some small step in this direction, which will reduce the value of the function.

In other directions, we may reduce the value of the function faster. We don't know which directions to choose, OK? Actually I shouldn't say that. When we take very small step sizes in this direction, it's reducing the value of the function fastest. There isn't a faster direction to go in.

When we take impossibly small, vanishingly small step sizes. But in principle, if I had some extra information on the problem, I might be able to choose step sizes along different directions. I may know that one of these directions is more ill-behaved than the other ones. And choose a different damping factor for it. That's a possibility.

But we actually have to know something about the details of the problem we're trying to solve if we're going to do-- it's a wonderful question. I mean, you could think about ways of making this more, potentially more robust. I'll show you an alternative way of doing this when we talk about optimization. In optimization we'll do-- we'll solve systems of nonlinear equations to solve these optimization problems.

There's another way of doing the same sort of strategy that's more along what you're describing. Maybe there's a different direction to choose instead that could be preferable. This is something called the dogleg method. Great question. Anything else? No.

So globally convergent, right? Converges to roots, local minima or maxima. There are other modifications that are possible. We'll talk about them in optimization. There's always a penalty to doing this. The penalty is in the rate of convergence. So it will converge more slowly. But maybe you speed the calculations along anyways, right?

Maybe it requires fewer iterations overall to get there because you tame the locally convergent properties of the Newton-Raphson method. Or you shortcut some of the expensive calculations, like getting your Jacobian or calculating your Jacobian inverse. All right?

So Monday we're going to review sort of topics up til now. Professor Green will run the lecture on Monday. And then after that, we'll pick up with optimization, which will follow right on from what we've done so far. Thanks.