KAREN WILLCOX: OK everybody, let's get started. So if you came in late, project three is here. It's graded in here. You're going to have project two back tomorrow. Yeah.

The project threes were really good. The one thing that seemed to trip up-- well, there was a few things that tripped up some people, but the one thing that seemed to trip up a few people was just the computation of the number of samples given the Monte Carlo simulation to achieve the required stated accuracy of plus or minus .01 on the probability estimate.

So some people got it really well. Some people got it, but their explanations were a little bit muddled in some of their words. And then some people didn't get it quite right. And I think it's a really important thing, and it's really important thing to make sure it's very clear in your mind.

And one thing that I would recommend is that whenever you're talking about mean and variance or mean and standard variation that you make sure it's very clear in your mind of which random variable you're talking about, because just saying mean or just saying variance doesn't necessarily tell me or you anything when there's lots of variances floating around.

In this particular example, there's the variance of the output of the simulation that we're trying to estimate, which is how far the ball flies. So how far the ball flies is a random variable that's got its own variance. But then the probability estimates themselves have got variances because the probability estimates are random variables because you're estimating them using Monte Carlo, which is a random process.

So when you're asked to get the probability estimate to within plus or minus 0.01 with 99% confidence, then that says we want to look at the variance of the probability estimate. How much does the probability estimate itself vary? And the variance of that estimator-- so what's that estimator? Let's call it p hat is the estimator of the probability. Well, we know that saying by the central limit theorem, it's distributed as a normal distribution with mean equal to the probability that we're actually trying to estimate, and variance of that probability p of a, 1 minus p of a over n. Yup.

AUDIENCE: [INAUDIBLE]

KAREN WILLCOX: We'll also put the lights on. That will help. Do you mind going into my office and grabbing some of the big chalk that's on-- great, thanks. I hate the little chalk. We'll get some bigger chalk. Can you see a bit better now? So this is the variance, but it's the variance of what? It's the variance of this estimator, p hat of a, and this is the mean of p hat of a.

So again, just any time you write variance, variance of what? Variance of what random variable? And I sort of got the feeling some people who knew it had to be this thing, but with defining this thing as the variance, and then just randomly or magically dividing by n, but make sure that this is really clear in your mind that the estimator itself is a random variable. And then in order to figure out how many samples you take, we've got to control the variance of the estimator. And then because we know this thing is normally distributed, and we want 99% confidence, that's plus or minus 3 sigma or plus or minus 2.58, some of you computed, so we need to make sure that this sigma, square root of this times 3 or times 2.58 falls within plus or minus-- is less than 0.01.

AUDIENCE: [INAUDIBLE]

KAREN WILLCOX: Less than here?

AUDIENCE: Yeah.

KAREN WILLCOX: So that's through the central limit theorem, because we know that we can write the estimate of the probability-- remember, we could write it as 1 over n, the sum from i equals 1 to n of i of ai.

AUDIENCE: [INAUDIBLE]

KAREN WILLCOX: Yeah, so this is another important thing to keep in mind is the output itself doesn't have to be normally distributed, but the estimator for the probability does have a normal distribution. And the reason it does is because we can write it as-- this is like a sample mean of an indicator function, something that's either 0 or 1. And so by central limit theorem, this thing for big enough n has a normal distribution.

The same is true for the estimator for the mean, because that's 1 over n, the sum from i equal 1 to n-- then we called this thing y, but I think we called it y bar in the notes. The distribution of y bar, the estimator for the mean also has a normal distribution. So remember, we saw estimators for mean, for probability, and for variance. These two have normal distributions

regardless of the distribution of the output, but the variance doesn't. Is that clear?

And again, that's why I think it's important to say variance of what, or distribution of what. This is the distribution of the random variable, which is the estimator of the probability. It's sort of got nothing-- it's got nothing to do with the variance of the range of the ball in this case, or even the shape of the distribution. But what it does depend on is the probability we're trying to estimate. And so we use this result to figure out how many samples to take, but we don't have an estimate for this guy.

So a trick that helps people realize is that we know the worst case for this. So what value of p will give us the biggest variance? p equal 1/2? If p is 1/2, this 1/2 times 1/2 is 0.25. That's the biggest value that p1 minus p can take. There's 0 at either end and 0.25 in the middle.

So one strategy is to compute this using p equal 1/2, which is the worst case, and I would just do plus or minus 3 sigma. It's a little bit conservative, and that gives you something like 22,500 is the n that you need. And then you know you'll be OK for anything. And it turns out one of them, the probability came out to be like 0.52 or something. And that would be the worst case, and you'd be a little bit conservative here, but that's fine. You'd still be meeting your plus or minus 0.01.

So just make sure make sure that that's really clear in your mind as you go through how you take all these different pieces that you're clear about, the distribution and what is normally distributed and what isn't, and then you can work from this variance to identifying the plus or minus 3 sigma to give 99% confidence and then translate that into an n.

OK. So if you didn't get it quite right, take a look at it. And then if it's not clear, I'm going to do sort of a summary of everything on Wednesday, and we can go through it again in more detail then if you want to. And I'll also post some solutions tonight or tomorrow. But any other questions?

OK. So today we're going to finish talking about unconstrained optimization. We'll just pick up where we left off on last Wednesday. And we'll talk a lot about how we compute gradients. We might talk a bit about the 1D search if we have time. And then we'll finish up by talking about response surface modelling.

If you remember where we left off last Wednesday, so first up, this is a basic optimization problem we've been talking about. Remember, minimize some objective j that's a function of

design variables x, and we have x1, x2, up to xn design variables. Parameters p subject to inequality constraints that we write as dJ is less than or equal to 0, and equality constraints that we write as hk equal to 0. And in general, we might have m1 inequality constraints and m2 equality constraints, and then we might also have bounds on the design space.

So we saw this last Wednesday. And we also talked about the general process of optimization. And remember, I said think of it as a landscape where the directions in the landscape are the design variables, and the height of the landscape represents the objective function. If you're maximizing, your goal is to get to the top of the highest mountain or highest hill in the landscape. If you're minimizing, your goal is to get to the bottom of the lowest valley.

And the way that that works is that we start with some initial guess that we denote x with a superscript 0. That's the initial guess. We look around. For a gradient-based method, we're going to also compute the gradient of the objective of where we're standing in the landscape. Look around, find the search direction. Once we've found that search direction, look along that search direction. Perform what's called the 1D search and take a step alpha. Compute a new design, figure out whether we've converged, and keep going.

So we're walking around the landscape iterating with our design variables. And what we talked about on Wednesday was the steepest descent method, which says always choose the search direction to be the negative of gradient of j. So that's the direction of steepest descent. And we also saw conjugate gradient, where we modify the search direction according to where we have come from. And remember that we saw that steepest descent-- in this plot where the objective function is represented with the contours, the steepest descent direction is always perpendicular to the contours. And we saw that as we approached the optimum, steepest descent starts to take this zig-zagging path with really, really small steps. Whereas conjugate gradient, which modifies each search direction depending on the previous search direction, is able to converge much more quickly. And in this case, it's just [? a quick consideration. ?]

OK, so that's where we are stopped on Wednesday. So I want to mention just really quickly a couple other methods. So if I go back actually, we talked about first order method that use just gradient information, and that was steepest descent and conjugate gradient. So there are also second order method that use not just the first derivative, but also second derivative information. And remember, we also derived on Wednesday that the second derivative of j is an n by n matrix. That was the Hessian matrix, the matrix of second derivative that handled the pure second on the diagonal, and then all the cross terms on the off diagonals.

So how does Newton's method work? Well, if you think about a Taylor series-- and again, we derived this on Wednesday. The Taylor series in the case of a scalar objective that depends on a vector of design variables, so x is an n-dimensional vector. If we expand j of x about the point j of x0, then the first term is the gradient times delta x. Delta x is x minus x0. And then remember, the second term was this quadratic term that looked like delta x transposed times the Hessian times delta x. Remember, we did that on Wednesday.

And this is an approximation, because we're truncating all the third order terms and higher. So we can use this Taylor series expansion, and we can say, differentiate both sides. So grad J of x can be approximated by grad J of x0 plus the Hessian evaluated at x0 times delta x. Yeah. And what are we looking for? We're looking for a place with the gradient of j is 0. Remember, that was the condition for finding a minimum or a maximum, or it could also be a saddle point.

So if we set this left hand side equal to 0, then what's left is this condition here. And so now we can rearrange and say that the delta x, the step that we take is going to be the inverse of the Hessian at the point we're standing at times the gradient of the objective at that point with a minus sign. OK, so remember [INAUDIBLE] in the steepest descent method, we said that the search direction was just equal to minus grad J. And we're writing that x at q plus 1 is equal to x at q plus alpha q-- let me write that as a subscript-- times [? x ?] q.

OK. So that's what we've been saying is that the new guess of the design is the old one plus [INAUDIBLE] term. In the steepest descent method, where it's just minus gradient of J. So now you can see with Newton's method, we have that the delta x, which is this guy here, the alpha q times [? this ?] [? direction ?] [? s ?] q is the negative of the inverse of the Hessian times gradient of J.

AUDIENCE: [INAUDIBLE]

KAREN WILLCOX: Yeah, so we'll have to check that when we converge, when we get to-- we'll check that on convergence. Yeah. Typically you don't check it as you go along, but it actually depends a little bit on the algorithm. The next ones that you'll see, you'll see that the desire for Hessian to be positive definite is worked in.

OK, so how does Newton's method work? This says S, but it really should be alpha times S in the way I defined the update in delta x. So maybe you can see that if you're trying to optimize a quadratic function, this extension is exact. There's no approximation here. And in that case,

Newton's method would converge in one step. So that means that if I was trying to optimize some kind of a quadratic function, or the landscape would be like a bowl, no matter where I started, I would get to the optimum in one step with Newton's method.

But if the function's not actually quadratic, then what we're basically doing is computing an approximation of the function here as a quadratic. So however it would look, it would look maybe something like this, we'd be approximating here. We would solve Newton's method. We would jump to there. We would create a new approximation locally that might look like this, and then we would come in and eventually we would converge. So if J is not quadratic, you just keep performing Taylor series, getting a new point, and then repeating until it's converged.

So this turns out to be a really efficient technique if you start near the solution, if you're starting sort of close to one of the local bowls of attraction if you're minimizing. But the problem is that you can see that now we need not just gradients, but we need also the second derivative. I'm going to talk in a second about how to estimate the gradients even if is actually in many cases getting this Hessian ends up to be much too expensive.

AUDIENCE: [INAUDIBLE]

KAREN WILLCOX: How do we define convergence?

AUDIENCE: [INAUDIBLE]. Yeah, so for an unconstrained problem, remember we're looking for two conditions. We're looking for the gradient of j equal to 0. And we're looking for Hessian to be positive definite. Or positive semi-definite depending on the uniqueness.

That's in theory. It turns out that in implementation, it can be a little bit tricky to monitor convergence. So often you would look for the sequence of the design variables to stop changing. In other words, when the updates become really small. And that's when you would terminate, and then you would try to check the optimality condition.

So let's talk about how to estimate the gradient. And this actually will connect back to some stuff that you saw earlier in the semester. So we're onto number two, which is computing gradients. So, generally we're going to need definitely grad j, which remember was the partial derivatives of j with respect to all of the design variables. And we might need the second derivatives. If we wanted to do the Newton method, then we might also need the second derivatives. And I won't write it out, but again, remember that was the n by n matrix of second order partials.

So methods to compute gradient. Any ideas? Finite difference, yup. So that's finite difference, which we'll talk about. That's probably the most commonly used in practice that people are using. Any other ideas?

If you could, what would be the most reliable way to get gradient? Differentiate, yeah. So analytical gradients, if you can. If you have a code, or if you have a model that analytic, that's all functions you can differentiate, then differentiate. Analytic gradients is definitely the way to go.

Have you guys heard of the adjoint method, that Professor Wong talked about? [INAUDIBLE]. So this is a really powerful way to get gradients when you have particularly CFD models or [? panelement ?] models that have got lots and lots and lots of design variables. So like if you're trying to do shape optimization of an aircraft wing and your design variables were all the surface points or all the properties of the sections in a wing, so you have hundreds and hundreds of design variables, people use what's the adjoint method to come up with gradients.

Has anybody heard of automatic differentiation? So this is something that's becoming kind of increasingly popular, where people write these automatic differentiation codes, and basically you take your code-- so you take your code that takes in x and puts out J, and you feed it through this automatic differentiator, and it gives you back a code that takes in x and gives you grad J. Sort of somewhere at the intersection of computer science and computational science, I guess. They're write these codes that will go through your code and differentiate it line by line by line, and use the chain rule, and basically automatically differentiate it.

And so there's an example of this in [? EETS ?]. There's a big model called the MIT GCM, which is the MIT Global Circulation Model. I don't know if anybody's heard of it. Nicholas, you work over in [? EETS ?], right? Have you--

AUDIENCE: [INAUDIBLE]

KAREN WILLCOX: MIT's been building for a long time this really massive community model that's doing a lot of modeling of the ocean and atmosphere and interactions for climate change. And so they've used automatic differentiation so that you can take this massive model and come up with gradients. And it's symbolic differentiation too, right? Isn't it Mathematica?

There's a lot of different ways you can come up with gradients. [? And there's ?] [? another one ?] [INAUDIBLE] something called the complex step method, which I'm afraid we don't

have time to talk about. It's a pretty neat trick. I can't remember the formula. Do you know the formula, Alex?

AUDIENCE: [INAUDIBLE]

KAREN WILLCOX: Yeah. I don't want to write it down and get it wrong, but if you can take your model, your J of x and put in a complex x-- maybe you can find the formula. Anyway, it's a really neat trick where you can pretend that your variables are complex, and then it turns out that the gradient ends up being the imaginary part. So that's also neat.

And if your code's written in Fortran, this can be a neat trick. Do you guys even know Fortran? You know, Professor [? Jolley ?] still writes all of his codes in Fortran.

AUDIENCE: [INAUDIBLE]

KAREN WILLCOX: You're painfully aware? So in Fortran, you can define complex variables.

AUDIENCE: [INAUDIBLE]

KAREN WILLCOX: The imaginary part of J, x of [? pi ?] times [? a ?] [? h ?] is equal to the gradient?

AUDIENCE: [INAUDIBLE]

KAREN WILLCOX: Divided by [? J ?] is equal to the gradient. Yeah, so [INAUDIBLE]. It turns out if you take a complex step, if you take your variable x and add a complex step, and then divide by that h, it turns out the imaginary part of that result ends up giving you the gradient.

OK, but anyway. Finite differences is the most commonly used because it's sort of the general purpose one. So let's just look at how finite differences work. And these formulas should look somewhat familiar, although you saw them in a different setting. Remember, you saw finite differences when we were approximating partial derivatives in the [? PEs ?], right? Remember way back? You're look at the convection equation or the convection diffusion equation, where you had things like du dx, or [INAUDIBLE] squared [INAUDIBLE] squared, and you used finite differences?

The exact same idea can be applied now to find the gradient of J with respect to design variables x. So for example, if we were looking for dJ dx i at some point, let's just call it x0, then we would just take J and we'd have x1, x2, all the way, and we would have xi plus delta xi. So we're going to perturb the ith design variable. And then we would have xi plus 1 all the

way up to xn. They would all be fixed.

And we're going to differentiate that with the baseline point, xi up to xn. And all that's going to be divided by delta xi. And so this thing here is the numerator. So this is the baseline point, the x0 would be right here. So there's the baseline point.

And then if we want to estimate the gradient in the direction xi, we just take xi and we perturb it by delta xi, take the difference between the function [INAUDIBLE] perturbed point minus the baseline divided by delta xi. Yeah. So again, it looks pretty similar to what you saw when we talked about finite differences earlier. So this would be a one-sided difference.

AUDIENCE: [INAUDIBLE]

KAREN WILLCOX: Different ways of computing the finite difference? Yeah, exactly. So that's a one-sided. You could also do a two-sided difference with the same partial. So it would be J dx i evaluated at the point x0 could be J x1, x2, to xi plus delta xi minus J x1, x2, xi minus delta xi. All divided out by 2. [INAUDIBLE] central difference. So you can evaluate the derivative at the same point using the point in one side, or you could use one little step forward, one little step back, and do it like a central difference.

And if you wanted to do higher order, you could take more points. But now here's the catch. So if we want to estimate [? compute ?] to compute grad J at whatever the current point is in the design space, how many function calls do we need if we use a one-sided difference? How many times do we need to run our model? 2n if we were to use this one. How about this one? Well, almost 2n. n plus 1? So 1 for the baseline and then it's going to be a little step in x1, a little step in x2, a little step in x3, all the way through xn. So with a one-sided, we would need n plus 1. And for a two-sided, we would just need 2n. We don't have the baseline. Function evaluations.

Now remember, each function evaluation in the course of your simulation code, which could take minutes or hours or even days. So if you have 100 design variables, each time you want a gradient, you're going to have to do of the order of 100 calls to your function. And remember, you're going to need gradients at each point in the design space when you're trying to figure out where to move.

So it can get expensive pretty quickly. How about if you wanted to compute the Hessian by finite differences? Well, we didn't write down the formula, but you remember the formula for

the second derivative, right? If we were computing that guy, then it would be J, and we would do xi-- probably don't want to write out all the arguments. xi minus 2J at the nominal point minus J doing xi minus-- do you remember that formula? Divided by delta x squared. The central finite difference for the second order derivative. Do you guys remember that? Am I confusing you, or is it OK to not write out all the x's? Is that all right? Yeah.

So to compute second derivative, first of all you're going to need the point in front and the point behind. But how many entries in the matrix?

AUDIENCE: [INAUDIBLE]

KAREN WILLCOX: Yeah, it's n by n. It turns out it's symmetric, so you don't need-- remember it's symmetric, so you only need to compute the diagonal and the upper triangle. It still turns out to be n plus 1 over 2 entries, and then you need the extra function evaluation, so then you would need two for each of these. So it basically ends up being [INAUDIBLE] n squared.

And so if you have 100 design variables, that's 100 squared, 10,000 calls to your function to get a Hessian matrix of the order every iteration. Which is why I say the Newton method is powerful, but if you're doing gradients by finite difference, almost it means you can't afford to use the Newton method.

But the good news is that MATLAB takes care of all of this for you. We can have a look at how MATLAB does that for you. Is this clear how you estimate the gradients? Yup? So let me pull this. So I showed you already on Wednesday. [INAUDIBLE] So I showed you already on Wednesday this little [? piece ?]. You might remember when we looked at the landscape and we saw the little asterisks climbing around. So I showed you [INAUDIBLE], but here's the call.

So remember I talked about [? Effman ?] unc, which is the unconstrained gradient-based optimization method in MATLAB. And I also talked about [? Effman ?] search, which I said was the [INAUDIBLE] simplex. Remember that was the triangles that were flipping around and going through the design space. So the thing with [? Naldemede ?] simplex is you don't need gradients. It doesn't need gradients. It just does these three triangles and all these rules to contract them. It's only ever looking at the three points in order in the design.

So if you have a function that's not differentiable, you can still use [? Effman ?] search. But if you have something that's not differentiable and you try to use it [? Effman ?] unc, MATLAB's going to be trying to compute gradients-- not second derivatives but first derivatives-- and

could get itself into trouble.

But the really nice thing with these MATLAB functions is that you basically supply a function, that given x computes J. And you supply an initial guess, an initial point in the landscape, and that's actually the only thing you have to give it. If you don't want to give it anything more, MATLAB will take care of everything else for you. You can give it just that black box function, given x computes J, and an initial guess.

If you know a little bit more about what's going on, you can also choose to manage options. So you can do things like turn on see more information about the iteration. You can play with tolerances. You can even specify exactly which methods you want to use. So here's the documentation for [? Effman ?] unc. So you see there, there's just the basic call, giving it a function and an initial guess.

So you see all these different options in here. So you can have the option to supply a gradient. So if you had a function that was analytic and you could differentiate it, it would be a really good idea to give that gradient to MATLAB as well as giving it the function. Or if you would use automatic differentiation and come up with a code that computed gradients, and you're able to specify that as an additional option.

And last thing I just want to show you is the methods. Check it in here. I'm not sure if it's telling us what the methods are. [INAUDIBLE] and have a look, see what method it is.

So again, I didn't talk about them, but there's a class of methods called quasi Newton methods, which in a way are kind of like the best of both worlds between the first order and the second order methods. So if you remember here back in the Newton's method, [INAUDIBLE] design space to be the inverse of the Hessian times the gradient. But then we said that [? computation ?] was really too expensive.

So what a quasi Newton method does is that it basically introduces an approximation for the Hessian. That's this A thing here. And it builds up this approximation using the gradients that it's computing anyway. So don't worry about the details of what is on here, because it's a little bit beyond what I want you guys to be comfortable with. But more or less every time we compute a gradient, we can also use differences between gradients to get approximations to Hessians. And I'm pretty sure that's the method that's sitting underneath here in MATLAB.

AUDIENCE: [INAUDIBLE]

KAREN WILLCOX: [INAUDIBLE]. OK.

AUDIENCE: [INAUDIBLE]

KAREN WILLCOX: So we're not guaranteed because it depends on the problem. So if you have a problem where there's a direction where the curvature is flat, then you're going to have a singular Hessian. And then you just have to be careful.

AUDIENCE: [INAUDIBLE] Yeah. If J were linear in any design variable, when you differentiate twice you would get 0. And you have to be very careful if you have [INAUDIBLE] inverse of the Hessian. But there are ways you can handle that, more advanced optimizations.

OK. So I'm not going to talk about constrained methods, but I did want to just show you briefly, and show you also in MATLAB how constrained methods work. So there's something called the [? buns ?] function. Actually the optimization community loves these little simple problems. Remember I talked about [? Rosenbrot ?] the other day and called it the banana function.

So there's another one called the [? buns ?] function, which is actually a constrained problem. So it turns out this is the objective, which I know doesn't really mean anything, but the difference of this one is it's got constraints. And so if we look at the visualization, here's the design space, design variables x1 and x2. And now the contours, they're called contours of the objective function. And then the constraints are saying the solution has to lie above this one, has to lie to the left of this one, and has to lie above this constraint.

So again, when I was talking about the landscape, remember I said think of the constraints as being like the keep-out regions, telling you where in the design space you can be. And so in this case our possible design region is this thing here traced out by the constraints.

So it turns out this problem has two local solutions. One is also the global solution. If we try and minimize, the global solution is actually up here in the corner of the design space, which you can see from the colors. It's the bluest color. But there's also a local optimum that sits here on this constraint down here at about 50, 20. And I should say that there are bounds on the design variables, which is why this global solution sits up here, because the design variables are at their bounds.

So that's a simple constrained problem. And I just want to show you running the algorithm. So [? fman ?] [? con ?] is the MATLAB function that does constrained optimization. And again, it's

really super simple to use. At a minimum, all you have to do is give it a function that given x returns J of x, give it an initial guess to start with, and give it a function that given x returns the constraints, the g of x or the h of x if you have them. So given x, return J of x, return g of x and h of x, and an initial guess.

And there are lots more options that you can set if you want to. And you can look at lots more things, but in the simplest implementation-- and then MATLAB will take care of finite differencing for you to do the gradient, the step size, everything. Everything you have to worry about. But of course, again, if you have an efficient way of computing gradients, you have the option to specify the gradients and make things run a lot more reliably and a lot more efficiently.

So just go back up here, so this one's actually using-- this is a quasi Newton method. There's also a Newton method that [INAUDIBLE]. I think there are actually three levels of optimization in [? fman ?] [? con ?], two different kinds of quasi Newton method, and there's a Newton method that's in there. In order to run the Newton method, you have to actually supply a gradient for it, so that it doesn't do all that finite differencing thing for the Hessian.

And what we can look at, what we're looking at here are different initial conditions. So here's that [? buns ?] function that I was showing you. And in this particular case, we initialize here, and what's being ported are the steps that the optimizer's taking in the design space. I'm not going to talk about the algorithms at all, but just to get a sense that when you have constraints, things are much more difficult, because it's not just about going downhill. But here we could just go downhill, and you can see more or less this looks kind of like a steepest descent direction.

But then when you get here and you're right kind of actually on the wrong side of the constraint, you can't just go downhill, because you've got to worry about the constraint. And you can see what it's doing is it's kind of following the constraint and skipping around. So things get a lot harder when you have constraints, because you have to worry about descent, but also about satisfying the constraints. But again, if you take optimization class you'll learn lots of stuff there.

AUDIENCE: [INAUDIBLE]

KAREN WILLCOX: Yeah. So it depends on the algorithm. Some algorithms will allow it to violate the constraints a little bit along the way. Some will not. And there are all lots of different ways of handling

constraints. There are things called penalty methods. There are things called barrier methods. There are interior point-- the barrier methods related to interior point methods that force things to stay away inside the constraints. It just depends on what algorithm you use.

But that sort of relates to the next part that I want to show you, which is now when you think about convergence, so here is the objective function value. This is the iteration. So you can see the objective is coming down, but you don't just want to worry about what's happening now with the objective. You also have to look at the constraint violation. And here it [? skips ?] [? forward ?]. It looks like we found a pretty good design, but actually the constraint is violated. So this would be like an aircraft wing that exceeds the maximum stress constraint, which would be no good. And so that's why we have to keep going. And actually you can see a little bit of an increase here in the objective to get the constraint violation satisfied.

And I'll say that when we try to apply these kinds of methods to realistic aircraft design problems that have lots of variables, they have even lots and lots of constraints. And satisfying constraints can be sometimes just about the most challenging thing for the optimizer. So it can often find good solutions, but then it turns out that they don't necessarily satisfy the constraints.

You see another one here. In this case, we started with an infeasible design, one that violated these constraints, or violated this constraint. So you see first of all the optimizer tries to get it back to the feasible region. It tries to make sure the constraint is satisfied. And then it starts searching. And actually most of these designs are infeasible. It's only at the end that we come here to the local optimum. And so again, there's the function value. And you can see we start off violating the constraint. We satisfy it, and then we go back outside the constraint, and then eventually come back to it at the end.

And then the last one starts again over here, where two constraints are violated. And you can see even though two constraints are violated, this one's actually able to get to that solution pretty easily. One thing you notice is none of the ones that I started actually found the global optimum. They all converge to the local optimum. And again, if you like to think more in 3D, remember that landscape where there was the big mountain and the two smaller peaks. In this case, we started here, we started here, and we started here, and they all ended up over here. Probably we would have to start on the other side of this ridge here in order to get up into the global optimum.

OK. So I know you don't really know very much about constraint optimization, but you could all be [? dangerous ?] [? utilize ?] for the next year if you wanted to be. It's actually a really powerful thing if you're trying to do a design problem, or you have simulation code and you want to explore what's going on with the parameters. The toolbox in MATLAB is really good. And then if you say for grad school, you can take the graduate class that I teach that goes into this stuff in much, much more detail.

AUDIENCE: Is there anything MATLAB, like other--

KAREN WILLCOX: Yeah. Absolutely. So there's all kind of optimization toolboxes. I mentioned Professor Johnson in [INAUDIBLE], where he's implemented a lot of some of the less mainstream optimization methods. That's nice. And then there's all kinds of other toolboxes, things like CPLEX that a lot of optimization people use. What are some of the big ones? I would say though MATLAB has really started to sort of dominate the market in a way because MATLAB's optimization toolbox has become really good, especially in the last 10 years. And I think the gradient pace, they have the state of the art, the Newton methods with all the bells and whistles on them.

And then there are also a lot of user-contributed algorithms. I'll show you. Let me see if I can run it, something that I don't approve of.

AUDIENCE: [INAUDIBLE]

KAREN WILLCOX: That's right. Yeah. I gave you the--

AUDIENCE: [INAUDIBLE]

KAREN WILLCOX: I gave you the link to that in the last lecture notes, to Professor Johnson's website. And actually since you said Python, also pyOpt. Right now pyOpt is starting to create all the optimization algorithms in Python.

AUDIENCE: [INAUDIBLE]

KAREN WILLCOX: What about [? Julian ?]?

AUDIENCE: [INAUDIBLE]

KAREN WILLCOX: Is that Professor Edelman's stuff?

AUDIENCE: [INAUDIBLE]

KAREN WILLCOX: Oh, once you graduate? But that's part of their model is for you to convince your employers to-- but that's why you stay in academia is you get perpetual very cheap MATLAB. It's one of the great perks of being an academic.

AUDIENCE: [INAUDIBLE]

KAREN WILLCOX: So I want to show you something again of which I definitely don't approve, but again this triggered in my mind because I said user-contributed toolboxes. There's a whole class of optimization-- I'm going to use the word "optimization" in quotes-- methods that don't use gradient, that don't really have any convergence theory. A couple of them have some theories associated with them. They're typically called heuristic algorithms because they use rules to search the design space.

And one of the most popular in aerospace engineering is something called a genetic algorithm, which basically uses all the rules of natural selection, and mimics the rules of natural selection to optimize the design space. So this is the function that we were looking at on Wednesday. And remember, we did a gradient-based method, or we did the [? naldamete ?] simplex, where we had a little asterisk, but with our initial guess, and then the asterisk marched and found the top of the hill, or got to one of the tops of the hills depending on where it started.

So a genetic algorithm is a population-based method, which means you don't have one design on each iteration, but you have many. And I don't know how many there are in this population, maybe 20 or 30. So there are 20 or 30 initial guesses, and that's the population. And when I set it running, what you're going to see is that each iteration, it's evolving this population.

So if there are, let's say there are 30 there, on the next iteration it's going to be 30 more and then 30 more and 30 more, and the way it's evolving it's taking designs and then the parents and they're having children, then it's figuring out which children to keep and which ones to not keep. And it's introducing a mutation that causes designs to jump around. So there's a whole sort of thing. And Professor [? Divic ?] knows a lot about genetic algorithms. So if you're interested, you could ask him. But let me set it running, and you'll see you'll see what I mean.

AUDIENCE: [INAUDIBLE]

KAREN WILLCOX: Yeah, the heuristic is all the things about-- you take designs and you have to encode them with chromosomes. And then there are heuristics about the different mating strategies and

then the mutations. So there's a lot of-- and at the end of the day, you don't have guarantees about optimality. But if you have a problem where you can compute gradients, maybe where the models are not differentiable, and you just kind of want to spray points everywhere and [? work ?] as well as you can, maybe something like this is OK. I wouldn't call it optimization.

AUDIENCE: [INAUDIBLE]

KAREN WILLCOX: Yeah, if you want to avoid local-- but I think you avoid local minima by not getting to any minimum at all. They work great for when you have 2D. But you see the idea how dramatically different it is. There's simulated annealing which does have some theory associated with it. There's particle swarm optimization. There's ant colonies. There's lots of analogies with nature. So there's this whole other class of optimization algorithms which can be kind of fun. OK. And so if you wanted to find those, you could probably go find other toolboxes as well.

OK, so I think I'll skip over the 1D search. It's not all that important. But I want to spend the last little bit just talking about surrogate modeling and reduced [INAUDIBLE] models. So maybe you have a sense that when we run an optimization algorithm, we have to call the code lots of times. Every time we want to evaluate the objective or the constraints, and any time we want to get a gradient, we may have to call it n times. Also, you've seen in the Monte Carlo simulation that you had to call the code 22,000 times when you were running the simulations for each ball, each pitch that you were analyzing.

So it turns out that when you have CFD models or finite element models or realistic models of aircraft or spacecraft or whatever it is you're modeling, it just gets too expensive. And so we use what is called surrogate model. So the word "surrogate" means that we're going to replace the real model with an approximation. And the idea is that the approximation should be something that's much cheaper than the model that we're trying to optimize, but hopefully it's good enough so that we can make good design decisions and find good designs, and then maybe go back and analyze them with our more sophisticated model.

So I like to think of surrogate models in these three different categories. There are data fit models like regression models, and we'll talk a little bit more about these ones. There are simplified physics models, which actually we use in engineering all the time, that maybe we want to design according to a Navier-Stokes equation of, in this case, of a supersonic business jet. But we could also use like a paddle method, or even just a simple empirical very high-level model of the aircraft so we can simplify the physics.

And then there are things that are called projection-based reduced model, which is basically a mathematical way of coming up with simpler models. This little diagram here is supposed to represent an x dot equal x plus b u. If you guys didn't see state space systems [INAUDIBLE] anymore, right? No. So mathematical ways for reducing systems that I won't talk about. But I do lots of research in that third category.

So if we just want to think a little bit about the data fit category, so the idea is going to be that we're going to sample our simulation at some number of design points. So we might use one of the design of experiments method that we talked about last week to somehow select some points. So we're going to run our expensive model at a bunch of points, and then we're going to fit a model using that sample information. We could try to fit a model everywhere, or we might want to do more of a local fit. And then you could also think about updating the model when you get new points.

But I want to talk specifically about response surface modeling, and see how one would come up with a surrogate model using a response surface. Because again, that's where we'll use some of the mathematical techniques that you guys should have seen before.

So we skipped over number three. We didn't talk about the 1D search. And number four was really just a brief intro to what is a surrogate model, and what are the different times. So we're at number five, on a response surface model.

OK, so the setup is that we're going to have-- let's just draw the [? block ?] diagram. So this is our expensive model where we can supply x, and out comes J of x. Therefore with a constraint problem, it would also be g and h coming out of there. So that's our expensive model. And basically what we want to do is come up with a surrogate model said that when we supply x, what's going to come out will be some J hat, where this guy is hopefully a good approximation of J.

And one simple but pretty powerful and commonly used way to do that is with what's called a response surface model. So response surface model, or RSM. So if we wanted to do just a first order RSM, then what we're going to say is that this J hat, which is a function of x, is just going to be some constant, let's call it a0, plus the sum from i equal 1 to n of some ai times xi. In other words, it's just a linear model. It's a constant plus a constant a1 times x1, plus a2 times x2. So it's going to replace the expensive model with a linear approximation. That would be a first order response surface model.

Turns out this is not such a good idea, and we'll look at it graphically in a second and you'll see why. So more often than not, people used a second order or a quadratic response surface. And it's the same idea that the approximate model they had is going to be, again, a constant, and then it's going to be the linear term, so the sum from i equal 1 to n of ai xi's, and then it's going to be all the quadratic terms, so i equals 1 to n, j equals 1 to n of b i J x i x J. And actually maybe this should be J equal i to n.

So again, we're just replacing our J of x with a model, and in this case, second order case, we're saying the model is quadratic. It can have a constant term, it can linear terms, and then it can have all these quadratic terms in the xi, xj, xi squared, x1 squared, x1, x2.

AUDIENCE: [INAUDIBLE]

KAREN WILLCOX: Well, because we don't need 1, 2 and 2, 1. Yeah. I mean, you can write 1 there if you wanted to, but because x1, x2 is the same as x2, x1, we would just lump them. [INAUDIBLE]

OK, so here we've got that the ai's are unknown coefficients. And here we have the ai's and the bij's are unknown coefficients. So we've said that we want to approximate our model with a linear or a quadratic function. Now the question is how do we come up with coefficients for that approximation. And that's where this sampling comes into play.

What we're going to do is we're going to generate samples using our favorite design of experiments method, or maybe we're just going to run Monte Carlo and sample randomly. We're going to generate a bunch of samples, and then we're going to use least squares fitting to try to estimate those coefficients and come up with the response surface model.

The next step is going to be how to determine the ai and the bij coefficients. So let's say that we have generated n samples, and each sample is going to have-- we'll call it j-- there's going to be a pair consisting of the design variables that we sampled, the value of x that we sampled, and that's again the n by 1 vector, and then the corresponding value of the objective. So we'll have x1, j1, we'll have x2, j2, all the way up to xn, jn. So I ran my expensive model n times. So I just collected all the data. I collected all the conditions that I ran it at, and I collected all the results that I got out of it. Is that clear? Yeah?

Let's think about this guy here, so just a first order [INAUDIBLE] J of x equal a0 plus the sum from i equal 1 to n of ai xi. How many pieces of data do we need at a minimum? How many unknown coefficients are there in the model? n plus 1. Little n plus 1. So we need big N to be

at least little n plus 1. But in m what we would do is we would generate more than little n plus 1, and then we would do a regression. We would do a least squares fit. You guys did least squares fir somewhere, yes? 1806? Did you do it anywhere else? [INAUDIBLE]

AUDIENCE: [INAUDIBLE]

KAREN WILLCOX: High school. Did everybody here remember their high school regression? No?

AUDIENCE: [INAUDIBLE]

KAREN WILLCOX: You can do them on your calculator? MATLAB backslash, like the most amazing operator? OK. So, well, let's think about what it would look like. So I always like to think of the least squares fitting from just the matrix system. Let me draw it over here so there's plenty of room.

This is a matrix system, and if we have exactly the right number of data points, the matrix will be square. And if we have more data points than unknowns, the matrix will have more rows than columns, and it will be an overdetermined system, and you would do a least square solution.

So we'll set up a matrix system. First of all, the unknowns, so the things we're solving for are these [INAUDIBLE] a0, a1, a2, down to a n, little n. Those are the unknowns. And then each row in the matrix system, this is going to correspond one of the sample points.

So the first sample point says that x-- when I feed it in x superscript one into the model, I get out J1. So here's J1, and here's the expansion. And when I feed in x1, so the first thing I do is I get 1 times a0 plus a1 times x1 in the first sample point. So this guy here is going to be x1,1 where the superscript is the sample number, and the subscript is the design variable number. Yup. And that's x2,1, xn,1. Is everybody with me? I just wrote down the equation 1 times a0 plus the first component of sample 1 times a1 plus the second component of sample 1 times a2, plus, plus, plus the nth component of sample a1 times a n equals to J1.

[INAUDIBLE] exactly does this equation apply to the first [? here ?], and I could write down so that this is the first sample, which we called x superscript 1. So then I would do the second sample, and it would just look like this. x2,1, x2,2, xn,2, and that's going to be equal to J2. And then I would keep going.

So I'm always going to have 1's in this column, because that 1 is always going to multiply a0. Down here eventually I'm going to have x, first component of the nth sample, and here I'll have

x, nth component of the big Nth sample, and everything in between. Yup. So this is a big N by little n plus 1 matrix. This is a little n plus 1 vector, and this is a big N vector. Yeah.

And if we have exactly little n plus 1 samples, again, this matrix would be square, and we would just invert it, and it would give us the coefficients. If we have more samples than we have unknowns, then again it's just a least squares solution, which you can do with matrix with MATLAB's backslash, or you can remember the formula involving ax equals b, and it's overdetermined, then you do a transposed ax equals a transposed b, and then take the inverse of this, and take it to the lefthand side. [INAUDIBLE]

OK, and if we're doing a quadratic response surface, so if I was setting J hat of x is this plus this plus the term, the b ij xi xj, how would the system change? What would I have to do? Would there be more unknowns? Yeah, so there would be the b1,1, b1,2, b1,3, all the b's. So more unknown means more columns in the matrix. So what would be-- if I put b1,1 here, what entry would go in here?

AUDIENCE: [INAUDIBLE]

KAREN WILLCOX: X1,1 squared, because in the expansion, it's b1,1 times x1 squared. And we would be evaluating that for the first sample. So we'd end up with x1,1 squared x1,1 times x2,1. You'd end up with all the quadratic terms matched out. So it looks just the same.

The regression's not really-- sometimes it seems more complicated, but I think if you just write out the matrix system and think about it as an overdetermined set of equations, it becomes clearer. And then you can also see we're going to have more unknowns. We're going to add more columns. If we want it to stay overdetermined, or at least determined, we would have to get more samples, which is what we talked about.

So let me show you how that works. Is this clear? Is this notation clear?

AUDIENCE: [INAUDIBLE]

KAREN WILLCOX: Say it a bit louder.

AUDIENCE: [INAUDIBLE]

KAREN WILLCOX: Yeah, that's right. And in fact, using a first order response surface is a really bad idea, and we'll see it graphically. So it turns out linear approximations in optimization are not a good

idea. Maybe you can think about why that would be with a linear approximation.

You think about what we're doing, we've got this landscape, we build a linear approximation and say "optimize." Find a new point, and build a new linear approximation. But where is the linear approximation always going to send you? It's always going to send you far away in some direction, kind of by definition.

So I build a linear approximation here. If it's sloping this way, it's going to take you all the way to that side of the design space. If it's sloping the other way, so it's always going to send you [INAUDIBLE]. So you could combine it with a strategy that-- that's what a trust region does.

But actually it turns out that quadratic approximation, because they embed the information of curvature, which we know is really important for optimality conditions, are far, far better choices and more commonly used. Put up on [? stellar ?], it goes through what we just wrote on the board. But Let me just get rid of that horrible genetic algorithm and get to the right directory.

So I have a code here that is going to build some response surface. It's the same demo. It's this peak problem, the landscape problem. So it's going to grab some samples. Maybe I'll just do it and we can look at the code. And we're going to build some response surfaces so you can-- so in this first example, again, here's our landscape, variable x1, variable x2, objective function.

We happen to generate these [AUDIO OUT]. Maybe they were random. Maybe we had some kind of design of experiments method to do it. And three points, two dimensions, what can we fit? What kind of a response surface can we fit? Linear one, first order? n plus 1? We don't have enough to fit a quadratic. So there is-- in effect, it's determined. It's three unknowns, constant, slope in the x direction, slope in the x2 direction.

So there's the response surface. It's clean. It goes exactly through those three points. And you see kind of the problem that [? Tyrone ?] was asking about, which is if we use this optimization, it's going to send us-- and we maximize, it's going to send us over into the corner of the design space. Clearly it's not a good approximation.

So we could generate more points. So now we have these six points. And now that we have six points, we could still set a linear model, but now it would be an overdetermined system. The matrix has got six rows, because there are six sample points. And it's got three columns

because there are still three degrees of freedom, the constant, the slope in the x1 direction, and the slope the x2 direction.

So now you still have a linear model, but you can see that it doesn't go through the points. And in fact, it's the least squares is like the line of best fit, but we're in multiple dimensions. So it's like the plane of best fit. Three of the points are above, and the other three want to be sitting somewhere underneath here. And maybe if you've taken 1806-- they're sitting underneath here. Maybe if you've taken 1806, you know that this solution is going to be the one that minimizes the sum of the squares of the two norm distance. So it is the line of best fit in that optimal sense. That's what the least squares fit is.

OK. So now we have six points, which turns out to be n times n plus 1 over 2, which is how many points we need to fit a quadratic model. So we have six points. And you think about the quadratic model, what do you have? You have the constant, a0, you have the linear terms, a1 and a2, and then you have of b1,1, [INAUDIBLE] squared. You have b2,2 for x2 squared, and then you have b1,2 for x1 x2.

You have six degrees of freedom to fit a quadratic model in 2D. I wrote out the expansion. So six points, six degrees of freedom. Again, our matrix system would be 6 by 6, so it would be uniquely determined. And this is what the quadratic response surface looks like. And as you can see, it's going exactly through those points.

OK, so now we're starting to get a little bit of curvature. With a quadratic model, we can only ever have one minimum or one maximum for the models that look like this, whereas we know that the design space we're trying to approximate has got multiple blobs. But a quadratic model;s not ever going to capture these multiple hills in this case.

I mentioned really briefly on one of the slides that we could think about adapting the model. So maybe as we're going through the optimization, maybe we can make a genetic algorithm, even though we're not supposed to. We're figuring out that this is the interesting region in the design space, so rather than having our points spread out everywhere, we might be saying let's kind of only on the points that have got good performance that we've found so far.

So see all these points around here-- [INAUDIBLE] this one does too because it's on the side of that other little hill that sits over here. So if you build a response surface using those points, maybe you can see now that the model's really pretty terrible over here, but it's actually starting to be a pretty good approximation of what's going on locally around those points.

And that's a common strategy would be to build these local quadratic models in regions where you've sampled and use them to make progress, and then to keep updating them. And even though a quadratic model seems really simple for a real problem, that strategy of adapting points and building local models [AUDIO OUT] updating the model as we go is actually a really, really powerful one.

And of course if we had more points-- I don't think I've got one that's got more points. If you had more than six points, than the quadratic model again would be the quadratic model of this fit, where it wouldn't go through all the points, but it would cut through them on average equally. OK, questions?

So that's a pretty, I think, neat use of regression that's actually very powerful and is used a lot in practice. So that's one of the things that you need to be able to do. So have a basic understanding of how a design problem can be posed as an optimization problem. What do I mean when I say constraint? What I mean when I say objective function? Have a basic understanding of the steps, and the gradient-based constrained optimization algorithms are looking for the direction and how the different methods do that.

Be able to estimate a gradient, a first order or second order derivative using finite differences. You could already do that. You just did it in the context of PDEs. Now we're talking about design problems. And understand how you could construct a polynomial response surface, either linear or quadratic, using least squares regression.

Actually, we didn't talk about the quality of fit. Do you guys remember what's the quality of fit metric for regression models? r squared. Yeah, so r squared tells you how well your surface approximates your data points. But you have to be careful because it doesn't tell you anything about how good it might be in regions where you didn't sample.

All right. Final questions? You guys feel like you've learned enough in this class already? No? OK. So on Wednesday, I will do a review of finite-- I think I'm going to focus only on finite element methods and then the probabilistic and optimization function.

If you have specific questions or topics that you want me to cover in more detail than others, then either bring them to class or you can e-mail me ahead of time so that I can prepare a little bit more. But that will be a good chance to ask questions about things that are confusing before the final. All right. See everybody on Wednesday.