[SQUEAKING]

[RUSTLING]

[CLICKING]

BERTHOLD HORN: So we've talked about edge detection. And we've talked about finding objects in two dimensional images. And, in particular, we discussed the Pad Quick patent, which provides an efficient way of doing that by building a model based on some training image and then using probes to collect evidence about a match and building a score and then looking for extrema in that multidimensional score surface, where the multiple dimensions were the pose-- translation, and rotation, scaling, et cetera.

And it was a great improvement over what came before, which was blob analysis, binary template matching, normalized correlation, half transform. Well, now we're going to talk about another patent in that category of finding things in images. And also, here the emphasis is more on inspection. That is-- if you read the abstract, it's all about inspection.

But, actually, the patent also deals with position and orientation. And what's different from the previous one is that the previous one quantized the pose space in order to do a complete search. And this one assumes you've already got a rough idea of where things are. And it's just going to improve them. So there's an incremental adjustment that will give you a very accurate position.

And you'll be happy to know we're not going to go through this in as much detail, particularly because quite a bit of it is based on what we've already done. So there it is. This one's called PatMax. And Bill Silver says that was motivated by two ideas. The one was that we're maximizing some kind of energy. It's an iterative approach. It's a least squares approach in a way.

And each step, which is called an attraction step in the patent, improves the fit. And then you stop when the fit is good enough. Or in the preferred embodiment, you stop after four steps. So that's the idea here.

And the other motivation for using that Max was Maxwell. So he knew about electromagnetics. And his intuition for this approach was based on forces between magnetics-- well electrostatic components and, in particular, dipoles. And, well, it turns out that intuition is completely wrong because those do not have the properties he wants in here.

And, in fact, a much better analogy would be springs-- mechanical analog. So the idea is if you're trying to fit two things together and you can identify corresponding parts, imagine connecting them with a spring, that they'll attract each other. And what the system will do after a while is minimize the energy.

It'll rotate and translate and do whatever so that the springs are as least extended from their rest position. And we'll explore that a little bit further. But in the patent, it's all about fields and stuff like that.

So the usual stuff-- application, dates, and prior patents, and the inventors, and the signee, and references. And you'll see here the first reference is Hough. That's the one we talked about last time.

And then there's a whole abstract, which is mostly about inspection. But, really, the patent is a large part about alignment. And here are more patterns and other publications. And if you read through them, you'll see that some of them are from the old AI Lab.

And here's figure one. So figure one looks a little bit like a figure we've seen before. There's a training image. And there's some training process, which generates a pattern. We called that a model before.

So they purposefully changed the terminology to avoid confusion. So what we call probes before are now called dipoles. But they are the same thing. They're a position and a direction and a weight.

So training in the previous patent meant that we created a model. And the model consists of probes. And the probes are created by each detection. So that's training.

And similarly, here, training here is going to be an edge detection process that produces edge dipoles and a field, a two-dimensional vector field. And that's training. Then you have a runtime image.

And at runtime, you're performing a attraction process that iteratively finds a good pose, assuming that you have a starting pose. So this one assumes that you already have done something to get an approximation. For example, you could run Pad Quick.

And why iterative? Well, because it's formulated as a least squares problem. And if we're lucky, the least squares problem has a closed form solution. And in this course, we try and set things up that way.

But in the real world, that's typically not the case. And then you solve it iteratively. And so you stop iterations when you think you have a good enough answer.

So you have a starting pose and you have the final pose. There's also a thing called a client map. So little detail-- if you want to work in a Cartesian coordinate system, well, your pixel positions may not be on a square grid. So the client map basically maps whatever you've got onto a nice square grid. And that's important because a lot of cameras do not have equal spacing in x and y.

And then there's an RMS error, which you can use to decide whether it's a good fit or not. And since this is about inspection, there are two more measures which you can use to decide whether or not the object is in good shape or not. And we get out evaluated things, which are used in inspection in this diagram. OK, let's go on to the next one.

So when I first saw this figure I thought, oh, OK, they forgot to actually put anything into this figure. But that's to illustrate the client map. So suppose your camera has pixels that are rectangular. Then you need to map from that to a square pixel array. So just minor detail.

OK, so this is the training. We have a training image. We do the feature detection, edge detection. There are a bunch of parameters that control that, such as the resolution you're working at, how far are you going to downsample? And it produces a field dipole list, which are basically the probes that we talked about in the other patent-- position, direction, and a weight, and maybe more. But more importantly, it produces a field.

So the idea is going to be that when you put down the runtime image, it's going to have features that you would like to align with features of the training image. And they are going to be attracted by features of the training image. So you're going to have this vector field, which basically tells you how hard to pull to try and get alignment.

And this is going to happen for all of the features. So there are going to be a lot of competing pools, which hopefully overall produce some coherent result. If your image is over to the right, those field values will be negative and will pull things to the left-- some kind of translation.

If your image is rotated, then they're going to be field values that have directions that are different in different quadrants. And overall, will have a talk that will improve the alignment. So that's the purpose of the field. And that's going to be something that's on the pixel grid.

That is going to be-- that's new. That wasn't in the previous one. And that also brings up another point, which is that the pose in the previous patent was a mapping from the model to the image because the model had the probes in it and you plonk the probes down on the runtime image. Here, we're going to go the other way around.

We're going to run feature detection on the runtime image and map it back to the field. Why? Well, because mapping a discrete number of things is much cheaper than transforming a whole image.

So in the previous case, we didn't want to take the runtime image-- and you can rotate it and translate it and make a new image. But, of course, that's a very expensive process. So instead, we took the discrete thing, the model, and we mapped it on top of that.

Here, we're going the other way. We're having discrete result edges from the runtime image. And we're going to map it on top of this field. Now we could, of course, rotate, translate, scale the field. But again, that would be very expensive.

So the pose here is the other way around but the same idea-- mostly translation, rotation, and scaling, although they allow for other things as well. OK, the part that's new and interesting is how the field is generated. And we'll go into that in some detail. There are many steps. And, essentially, the idea is to produce something that will draw you towards the alignment where things in the runtime image match up with things in the training image.

This should look very familiar. That's low level. That's our edge detection stuff with some small changes-- left out the [INAUDIBLE] thing here. Just assume that you have some Cartesian to polar conversion-- and then added parameters that control the process, because in different steps of the computation, you might want to change things, like how much you subsample the image, what kind of low-pass filter you use, what method you use for peak detection and interpolation. But other than that, that's just the same old story.

So in the training image, we compute these dipoles-- field dipoles they're called. And as I said, they're just really edge fragments. And this is the data structure for them.

And there are flags that tell you whether you're near a corner of the object. There are flags where the contrast is positive or negative. And there's a gradient direction. And, optionally, there's a link to the next field dipole.

So this has to do with the initialization. So what you do is you have this array, which is going to contain the vector field. And the vectors don't cover the whole array. And the reason is that when you get very far away from an edge, it's very unlikely that it really has anything to do with-- so here you've got an edge in the runtime image. And there's a edge in the training image that's far away.

Your belief that those two belong together is reduced when they're far away. So at some point, you just cut off and say, forget it. And so there's a part of the vector field which is going to contain these special markers saying terra incognita-- we don't know what's going on here. So don't have that contribute as evidence to the score.

And so you start off, basically, by wiping out the whole array and setting that special value everywhere. So if you don't get to impose a value, that's because you were too far away from an edge. And a lot of this low level stuff is the same.

This, you may remember, is all about linking them up into chains and then removing the weak chains-- short weak chains. And so here we have the starting values imposed on that field. So we've now taken the pixels in the field that directly correspond to edge points that we know about, field dipoles. And we've given them a value. And the value corresponds to the distance from the edge and the direction.

So it's not just an array of numbers. There's a direction to them and a length. So each of them is a little two vector. And so each of these squares that's been filled in has associated with it one of these field values that tells you how far away you are from the edge. So that's initialization.

And now you have to fill in the rest of it. So for squares that are anywhere near an edge, like within four pixels or whatever the threshold is that you pick, you have to somehow figure out how far away they are from the edge. This is very similar to something that's used often in machine vision called the distance map, which is a little simpler.

So a distance map is just an array that tells you for every pixel how far away it is from an edge. This is different because you actually give not just the distance but the direction. But the way you fill it in is very similar.

So it's an iterative process. You seed it. You put in the actual edge points.

And then you go one pixel away-- all the pixels that touch a pixel that's already been filled in. And the question then is, well, what value do you put there? Now, if we were using Manhattan distance, where it's just the sum of x and y distances, then you could do this accurately.

You can always accurately compute incrementally how far something is away from an edge. But since we're using Euclidean distance, which makes more sense here, that's actually not trivial. And since in addition, we're using direction, that makes it more complicated also.

So here are examples that explain how you compute it. So up there, there's an edge, 906. And we've already filled in 901-- that square. We know the vector that goes over to 906. And now we're looking at the block 900 and saying, OK, what do we fill in here?

And we should fill in something because we're touching a pixel that has been filled in-- namely 902. And so we can start off with the value of 902 and adjust it. And you can see there's a bit of geometry there.

We can actually figure out what that distance, 912, is if we know what the distance 904 is and that angle and so on. I'll bore you the trigonometry. But it's pretty straightforward.

Same over there-- we have that pixel now touching on a corner. But that's still considered connected. So one of them has been filled in, namely the pixel 932 has a vector, 934, that goes to the edge. And now we're filling in the pixel 930 by extending the distance that we have from the other pixel and so on.

And so that's the process. And we iterate a number of times until we decide that now we're so far from the edge that it's probably not a good idea to consider these edges to match. So we're sort of building-- one way of visualizing it, if we forget about the fact that the vectors, is as a kind of groove along each edge. So it goes up as you go away from the edge.

That's the distance map. So the distance map is this terrain, which at any place tells you how far you are from the edge. And now when you plunk down an edge from the runtime image, you want it to, by gravity, get pushed down, to run down the slope. Now, with one edge, of course, it'll just run through the bottom. But with lots of edges, they're all going to have their own local influence of where they should go.

And hopefully, it'll be somewhat connected so that you end up with the whole system settling into a lower energy state. And this is just slightly more sophisticated because we actually have a direction not just a height. And then it gets more interesting when we get to places where there's a collision, where we have information from more than one neighboring pixel.

And if they are similar enough, then you can take some kind of weighted average, try and improve the result. It's always noisy. So if you get more than one answer, you can improve things by combining them.

But when the angles get too large, then you have to say, well, that's not-- there's a problem. We're at a corner. So here's an example of the first step after seeding.

So we've seeded it with the edges that are in there. And now we've extended it by going one pixel away from those seeds. And each of those vectors is the force.

And it tells you how far you are from the edge and the direction to the edge. And in the process, we'll end up looking at points on the edge that are not the original points from edge detection, just as in the other case, we started off with points on a grid that were produced by that edge detection step. But then we interpolated our own points because we wanted a certain number of points per unit distance. They're not necessarily based on the pixel spacing.

Similarly here, the tips of these arrows don't point at the original edge pixels. They point somewhere else. And we record that.

Now, for some purposes, the vector field is the force, is all you use. But they also optionally allow you to record more information, such as what is the closest field dipole. So then you have options on matching.

Now, here there are no conflicts yet. And now we've taken the next step. We've moved outwards-- again, one pixel from where we were. And we filled in another bunch of pixels now with longer vectors, larger force. Because we are further away from the destination, there should be a larger pull.

It's a little bit like a spring. We're pulling more when the match is over a bigger gap. And the x marks the first place where there's a conflict, because we could have extended this vector here to fill in this pixel. Or we could have extended this one. And they obviously have very different ideas about where the edges. And so we mark that as a corner position, which will then be not contributing to the process later.

So you don't want to just blindly use this map. You want additional information. And that additional information could be a contrast direction. So that's the simplest one.

Or you might actually look at the directions of the vectors. How close is the runtime gradient match up with the direction of the vector in the field? Or how close is the direction of the gradient in the runtime image with the nearest field dipole gradient, because when you get over here, that direction may be different from this one. So they leave open all of these options.

And the process is going to be the same. You're trying to minimize the energy in the system. And you can see how this is better model with the mechanical system where you have springs.

So suppose you know that there's a runtime typo here that matches this model dipole. Then you can think of as being a spring between the two with zero rest length. So that when you pull it out, it'll want to shorten. And so the minimum energy state is the solution.

The one thing to keep in mind is that as we've mentioned many, many times. With an edge, often, we know accurately how things work in one direction and not the other. So over here when we're talking about a match with an edge, we can't really say that we are matching a particular point on the edge. We just know that it's matching that edge.

And so the idea of the spring isn't also completely accurate. What we'd really want is a slightly more sophisticated model that has a little carriage that can run along the edge. So there's a spring, yes. But we don't really care where the other end of it goes. So we could have a little mechanical device that can slide along the edge. And so that mechanical analog is basically the actual algorithm, not dipoles and electrostatic fields and stuff like that, even though that's the language used in the patent.

OK, so we talked about how to get the training image. Now we use it. We have the runtime image. We do the feature detection on the runtime image. That produces an image dipole list.

We plunk it down on top of the field. And that creates many, many little forces-- one for every feature that we detected. And they are going to adjust the pose.

Now, in the simplest case, if we only have translation, it's easy to visualize this. If the runtime image is off to the right of the model image, it's going to get pulled back to the left because all of the springs are extend it to the right. So that's very easy to visualize.

If it's rotated, it's a little bit harder to visualize. But again, you're stretching the springs out now tangent to a circle rather than radially. And they're all pulling in the same direction, either clockwise or anti-clockwise.

And so the equilibrium position is when you've let it that mechanical system go and adjust itself. We can think of size in a similar way, where if the size is wrong-- suppose that the runtime image is bigger than the model image. Well, all of the springs are going to be stretched outwards and by an amount that increases with radius.

And, again, that system will adjust itself by scaling the transformation until those springs are relaxed. And I guess in this patent, the emphasis is on what do you do then. And you compute clutter and coverage, which are measures of how good the runtime image matches the model and how good the model matches the runtime image.

And there are various ways of evaluating the result. Ideally, the coverage will be 100% and the clutter will be 0. OK, this is more about how the image dipole gets attracted to the pattern boundary.

So, overall, we're setting up a large least square system. So we're saying that each runtime dipole has a certain force exerted on it in a certain direction. And we are going to move all of them in a systematic way to try and reduce the tension, to reduce the energy in the system.

And, typically, the movement allowed will be translation, rotation, and scaling in this case. So if you write it all out, it's a huge least square system. And there's no closed form solution. But there's a natural way of computing it which involves a bunch of accumulators.

So let's see if we get there. This goes on and on for a long time. So this is the accumulator array, which is basically the upper triangle of a symmetric matrix. So with a symmetric matrix, we only need to keep the diagonal and everything to one side.

And these are sums of-- so W is the weight. We already talked about assigning weights based on various properties that might indicate whether a match is particularly good or not. And then positions in the image-- and so these are all what we call moments, things like the integral of something times x to the i times y to the j.

So you can see that we need quite a bunch of them. I forget. I think the matrix is 6 by 6. It depends on how many degrees of freedom you allow.

If you allow translation, rotation, and scaling, it's, I guess, 4. But if you allow for all 6 of the general affine linear transformation, then it'll be 6. And so you run through the image. And, of course, you can actually do it in parallel, if you're so inclined because they don't interact. You're just collecting evidence everywhere and adding it up.

And it produces an overall force. And then you adjust. So, again, as I mentioned, I'm not going to go into as much detail here as in the other patent. So these are simple cases. So the top one here is translation only.

And this one is translation and rotation. Don't be scared by tensor. As far as the patent is concerned, the tensor is just a multidimensional array, including array with one dimension or two dimensions and so on.

So it builds up to allow more and more degrees of freedom and emotional results. And that's not the end of it, because it's assuming-- it's locally linearized about the current operating point. So you don't get the final solution to the least squares problem. But you get a large improvement.

And in the preferred embodiment, after four steps, it's good enough. But of course, you can use any criteria you like for termination. Let's see if there are any others here that-- here's an interesting detail. There's a doubly linked list.

So the field dipoles make it easy to move in either direction along the edge because they are represented in the W linked list. So here's multiscale. So for speed, it's good to work at low resolution and get an-- so you have a starting pose. And then you get a low resolution pose at high speed using a low resolution trained patent. And then you use that as a starting pose for the high resolution stage.

So they only show two stages of this. You could do more. And also in this case, you don't need as high quality starting pose as you do if you just have one stage of this process.

Lots of flow charts. Let's see. Was there anything in here I wanted to point out? This is fun to read because you're thinking of an image of, I don't know, a printed circuit board or something. "Digital images are formed by many devices and used for many practical purposes. Devices include TV cameras operating on visible or infrared light, line scan sensors, flying spot scanners, electron microscope, X-ray devices, including CT scanners, magnetic resonance images, and others are found in industrial automation, medical diagnosis, satellite imaging," blah, blah, blah.

So that's just the lawyer trying to generalize this as much as possible. So if afterwards you say, oh, but I didn't use it for looking at a conveyor belt, they can say, well, it doesn't say you need to restrict it to that. And it explains the problems with previous methods and why this is the best thing since sliced bread. And then it gives all the details. And then finally you get to the claims.

Let's just take a quick look of the claims. Oh, here we go. What is claimed is geometric pattern matching method for refining an estimate of a true pose of an object in a runtime image, the method comprising-- and then it has these parts-- generating a low-resolution model pattern and so on. It's interesting how this first claim doesn't actually say much at all about the method.

So it doesn't give details. So it's very broad. And it's likely to be knocked down if there ever was a dispute because receiving a runtime image-- well, just about anything is going to receive a runtime image. Using a low-resolution model pattern, blah, blah, blah-- well, multiscale. Everyone does multiscale-- and so on.

So this is very, very general, trying to cover everything. And the downside of it is that if someone ever challenges it, it's probably going to be covered by prior art. And that's why you have all the dependent claims. So claim one is very general.

If you're lucky, it'll cover everything. But there's a good chance of being knocked down. So then you have claim two, which is dependent on claim one, and also includes producing a low-resolution error value, producing a low-resolution aggregate clutter value, and producing a low-resolution aggregate coverage value. So those are all the values used in inspection.

And then three is dependent on one wherein the low-resolution error value is a low-resolution root-mean-square error value. So you could use sum of absolute values instead. But this one specifically says root-mean-square. And then four depends on two and so on. And this one has an inordinate number of claims. It' goes all the way down to 55.

But that's probably all we want to know about that, unlike the Pad Quick-- so Pad Quick, it searched a complete polar space, admittedly at low resolution. So it didn't need to have a first guess. This one does require a first guess.

And it has a capture range. That is there's a region in polar space where, if you plunk down the initial value of somewhere in that region, it will end up giving you the result you want. Then we mentioned that the pose here is the other way around.

And that's for computational reasons. We don't want to transform a whole array of either gray values, edge values, or whatever, or fields. So it's the other way around from Pad Quick. And if you read it, there's a repeated emphasis on how they're trying to avoid thresholding. They're trying to avoid quantization at the pixel level. So everything is subpixel.

And I think part of that is to illustrate the quality of the result you can expect and partly to avoid prior art because the prior art was all pretty much pixel based and wasn't accurate to subpixel accuracy. Then there's a physical analog, which, as I mentioned, calling these things dipoles is misguided. I mean, do you know what the law is between two dipoles?

I mean, it's a mess because it depends on not just the separation between them but the angle and orientation in free space. Plus, forget dipoles. Think of just electrostatic charges. Well, if you bring them together, you have an infinite amount of energy. There's a singularity.

And so, wait, we're bringing these things together, the runtime edges on top of the training edges? So there's a singularity. So that doesn't work. But the mechanical analog works very well. It's exactly what they implemented. And it involves springs with that one little change that the springs have to be-- one end of the spring has to be able to slide on the edge.

So let's see. Description not tied to the pixel grid. And I guess we've been over what exactly the field is and the optional additional items.

OK, then how it use starting evidence-- so we collect evidence, as before, about whether this is a good alignment or not. And they have different ways of assigning weights and that there's this kind of threshold.

If you're too far away from the edge, then you don't know what's going on. And there's a kind of unpleasant discontinuity there, where you pull the spring out. And up to a certain separation, it gets stronger and stronger and stronger. And then suddenly, it breaks.

So that is one reason that you can't find a closed form solution, because it's a nonlinear problem. It's not a simple linear least squares problem. And so they don't really address that. There's some argument about some kind of fuzzy logic thing, but it's not really pursued. And apparently, in practice, it's not a big issue.

Now, this maximum distance that you allow for the separation should depend on how close you are to the answer. And so it should depend on what stage of the iteration you're at. So as you iterate, you get closer and closer to having the runtime image aligned with the model image. And so at some point, it just should be pretty well aligned.

And so you can cut off at a lower value. So the field is a fixed thing. And it has a cut off. But then during the iteration, you can additionally throw out matches that have too large a distance between the runtime edge. the

Talks lead to rotation. Forces lead to translation. So the matching, the inspection-- it's kind of a weird pattern in that the abstract is all about inspection. But then you read the specification, it's all about figuring out what the transformation is.

So the inspection part, sort of tagged on at the end, is based on two things. The one missing features. There are things that are in the trained image that have no match in the runtime image-- and on extra features, things that are in the runtime image that are not in the pattern.

And depending on what you are doing, one of those two measures will help you decide whether the object passes the inspection or not. There's also the notion of clutter, which are image dipoles with low weight, presumably things due to the background texture and so on. Multiscale-- blah, blah, blah. We've seen that. OK, the only part we really haven't-- well, we have sort of gone through it in the diagrams, which is the generation of the field.

Let's talk a little bit about that. And let me start off by talking about the simplified version, where we just have distance. So what does that look like?

So suppose that I have a single point. And there's no directionality to it. Then distance field, of course, is just a set of concentric circles. And then suppose that I have a circle.

So I have some edge that goes around in a circle. Well, then, of course, the distance field is just a bunch of circles, except now we're going inwards as well. And then if I have an edge, it's going to look like that.

And it gets interesting when I have a corner. So if I have a corner-- and then on the interior-- so in computing this, everything is nice and cozy, except right in the corner. Different things happen there. And so that's a potential problem with straightforward simple methods of filling this in.

Now, we are not working in the continuous world. So we're working in a discrete world. And so we have to figure out how to propagate these values. And, as I mentioned, if we were working with Manhattan distance, it would be simple because we can add up the changes in x and the changes in y. And we'd always have an accurate result.

So if this one is one of the features, then this is one away. That's one away. That's two away.

So Manhattan distance is very easy. Euclidean is not. And there's actually a whole bunch of papers on how to do Euclidean in an approximate way that's still good enough and is also very fast. I mean, you can always do it the slow way by, basically-- if you add a pixel, look at all of the other pixels and see which one is the closest and calculate the square root of the delta x squared plus delta y squared.

But, of course, that's very expensive. What you want to do is incrementally, as you fill in this distance field, just add another layer of pixels to it. So we won't talk about those. But just to know that that's one of those problems, like, what's a good way of sorting? There's this problem of what's a good way of computing the distance transform.

And, as I mentioned, in our case, we actually have vectors. And the figures in the patent make it pretty clear what to do. I mean, the equations are messy. But the figures pretty clearly tell you what to do.

OK. Then we have all of these little local forces. And we talked about adding them up. So one thing we can do is-- so the Fi's are forces at individual dipoles in the runtime image. And the Wi's are weights. And the weights can be all sorts of things according to the patent.

They can be predefined. They can depend on the magnitude of the gradient. So you believe things that have a strong gradient more than things that don't and so on.

They can depend on how well the runtime gradient lines up with the field vector. They can depend on how well the runtime gradient lines up with the nearest field dipole and so on. So lots of versions. But in general, you want to do something like this.

And so that's pretty obvious. We're basically just taking a weighted sum of these forces. And this is going to provide the translation. So we move the alignment based on that. And then what about rotation?

Well, very similar. We just take a torque around some center. So that's going to give us a rotation. And so I can just conveniently take the cross product of the vector from some center-- the rotation is going to be about some center. Most conveniently, it's the center of the image. So ri is measured from there.

And this is the torque that's exerted by all of those springs. And that will tend to make the runtime image move. So this is a little bit funny because-- no, it shouldn't be-- it's not a vector. So torque is a scalar, which is rotating. It's one degree of freedom.

But over here, we've got the cross product of two vectors. So what's going on there? Well, being a little sloppy with a notation, but the cross product of those two-- they're both in the plane. And so their cross product is going to be sticking out of the plane.

So if I want it to be pedantic, I could say that. Take the z component-- there's only a z component. But make it into a scalar, basically. And then we can have scaling, I suppose, although that's less common.

So that's-- if you want to reference the patent, I guess that's figure 21 and 22. And may the force be with you. Sorry, I couldn't resist that.

OK, nearest boundary point, alignment, energy minimization-- yeah. OK.

So we'll have a couple of little sidebars here. One of them is distance to a line. And we already mentioned that. But since it comes up so much, I want to go over that again.

It depends, of course, on your notation for a line. So, once again, let me propagandize my favorite. OK, so first of all, we can rotate into a coordinate system that's lined up with the line.

So we get x prime is x plus theta plus y sine theta. And then y prime is-- so that's step one. And then step two is let's move the origin to there. So we'll call this x double prime and y double prime.

So x double prime is just x prime. And y double prime is y prime minus rho. So y double prime is minus x sine theta plus y plus theta minus rho. And the line is the locus of y double prime equals 0. That's by construction.

And so we can write the equation of the line as x sine theta minus y cos theta plus rho is 0. And that parameterization doesn't have any singularities. It doesn't blow up when the line is going straight up or straight across or anything.

It's not redundant. It has two parameters, rho and theta, which is what you need because the family of lines in the plane is a two parameter family. And it's pretty handy. For example, suppose that we want to do some line fitting.

So suppose in our image we have a bunch of edge points. And we're trying to find with high accuracy a line that fits them very well. So what we could do is minimize sigma of the distance squared.

So what are our parameters for our rho and theta? Because by using this notation, this is the distance. Well, or maybe negative. But we're taking the square. So who cares?

And, of course, it's a calculus problem. So we take the derivative and set it equal to 0. And let's start with d rho. Then we're going to get 2 times-- so forget the 2.

And now the sine theta we can bring outside the summation. Sine theta sigma x minus cos theta sigma yi plus rho sigma 1. And if we have n points, we can divide through by n. And then we get-- right? Because the stigma of 1 is n. And so if you divide by n, we just get 1.

And so what does this say? So x-bar is the average, is 1/n. And, of course, y-bar correspondingly.

And so this says that there's a relationship between-- so it doesn't give us rho yet. It doesn't give us theta. But there's a strong relationship between them. And this is that the line has to go through the centroid, because if I plug in-- if I plug in the coordinates of the centroid, x-bar, y-bar, I get 0. And that's the definition of points on the line-- so the centroid is on the line.

So at that point, I might do a sensible thing of moving all my coordinates to the centroid. So I know that's going to be important point. So I can subtract that out. And then if I-- so I'm still trying to minimize this thing here.

And if you plug in-- so turn this the other way around. OK, so now I go into this equation. And I plug in-- for xi and yi, I plug in these expressions. Then it turns out that the centroid cancels out because we have this property. We're going to get x-bar sine theta minus y-bar cosine theta plus rho.

And we know that's 0. So we can get rid of that. OK, so now we're ready to do the final step, which is to take the derivative with respect to theta and set that equal to 0. And let's see.

After we gather terms-- so that's-- what we're going to get is 2 times this expression times it's derivative. And if you multiply all that out, you get this. And, well, that's going to-- so what's this? This is one half sine 2 theta.

And this, of course, is cos 2 theta. So I have an expression that involves sine of twice the angle and cosine of twice the angle. So I basically can do an atan2-- atan2-- of-- OK, so that's the sine. So that's going to be this one. 2 times sigma xi yi comma sigma.

So we use atan2 so we don't have the ambiguity about which quadrant we're in. And we don't have things blowing up-- no, sorry. That looks kind of a mess. But, obviously, this is the sine part, if you like. And this is the cosine part of the tangent. So we just take those two quantities.

And the E square solution has the benefit that it's independent of coordinate system choice. What do I mean by that? It means if I have a different coordinate system, different position, different rotations, of course I'm going to get a different answer. But in that coordinate system, that will be the same line.

Why do I make a fuss of that? Well, because that's not true if you fit y equals mx plus c. Now, there are circumstances where fitting y equals mx plus c makes sense if you happen to not have an error in x, if you assume that all the error in fitting is in y.

So x is some quantity you have complete control over. Y is something you measure. Then fitting y equals mx plus c makes sense.

But here we're talking about image coordinates. There's nothing different about x and y or different rotations and so on. So that's a method that could be used, for example, in some of the patent work we've talked about where we're dealing with edges and we're trying to combine short edge fragments into longer edge fragments.

So let's look at-- why don't we look at another patent? So the last one in this series is, again, kind of a subsidiary deal. it has to do with the multiscale business, namely, how do we efficiently-- so we keep on talking about the multiple scales. But how do we efficiently do that? Because, potentially, the computation is very expensive.

OK, so let's-- OK, right, because convolution is expensive. If we have a picture with n pixels and we have a kernel with m pixels, then computing the convolution is something the order of n times m. So with 10 million pixels or whatever, if you kernel is reasonably large, it's going to take a very long time.

So there's a lot of interest in finding shortcuts, finding tricks that make that job easier. And here's one. So, again, the last one we're going to look at from Bill Silver at Cognex. And this is about, basically, efficiently computing filters for doing multiscale.

And so the picture here is-- nth order piecewise polynomial kernel. So the trick here is we're going to approximate whatever kernel you like with something that's a spline-- piecewise polynomial. And it's an nth order spline.

And we then take the n plus first difference. So if you think about it, each of those pieces is nth order polynomial. And if you take the n plus first derivative, you get what?

Take a second-- order polynomial, take the second derivative-- you get-- 0. OK. And why is that good? Well, it's easy to convolve with 0. So what's the trick?

Well, the thing is that we you splice together the polynomials, you may have a discontinuity, not perhaps in the value but perhaps in the derivative, or maybe not in the value and the derivative but in the second derivative and so on. The result is that it's sparse. No matter what you do, maybe a little complicated at those transitions. But large parts of it are 0.

So if you do the convolution, your kernel has very small support. And so it's very efficient. It's very cheap to compute. So let's see. What do we got?

Filter parameters-- so you can pick different filters based on this. Then you get your signal. And this is the 1D version.

You sample it. You convolve it with that now sparse function. But then you have to undo that. And so you do the n plus first sum, which is the inverse of the n plus first difference.

And finally you normalize. And there's your filtered signal. So that's the basic idea.

Now, there's some things hidden in there that are pretty important-- first of all, that the n plus first sum is the inverse of the n plus first difference. But that sort of makes sense because one-- if you keep on-- left to right-- adding things up, that's the exact inverse of stepping through and subtracting neighbors in either order. So it makes sense on one difference. And then you can convince yourself that, well, these operations commute.

So if I have-- let's call them D and S. So let's suppose that this is the identity. And then what we want to convince ourselves that this is also the identity-- D times D, S times S.

So we take two differences. And then we take two sums. Well, because these operations commute, I can write this as DS DS. I switch these around.

And, of course, I know that this is the identity. And that's the identity. And I'm done. And that generalizes to n plus 1 differences and n plus 1 sums.

So that's one idea in there that we can sparsify a convolution and make it cheaper and that we can undo that sparsification by taking sums. And if you go through the algebra of figuring out how many operations, you always come out-- well, pretty much always come out ahead. That's one idea.

The other idea is that we're kind of mixing up convolutions and differentiation. And that's a slightly more subtle point. But we can think of integration and differentiation as convolutions.

So let's think about integration. So what would you convolve with to get an integral? Instead of ones, right? But all the way from minus infinity to plus infinity or-- so when we compute the integral-- so we have f of x. And we want the integral of f of x at some particular place-- let's say x prime.

Then we're integrating from minus infinity up to x prime. And we're ignoring the rest of f of x. So that's a little bit like taking a step function, multiplying by a step function-- well, flipped, right? Because we're multiplying by this and then adding the result.

So we'll talk about that a little bit more. So without that being mentioned in the patent, it's depending critically on the fact that derivative operators can be treated as convolutions-- not convolutions with real functions because we need impulses and stuff. But if we allow impulses and distributions instead of functions, then we can treat them.

And once we do that, we have all of the power of convolution and all of the nice properties of convolution. So, for example, convolutions commute. Why is that? Well, because the Fourier transform of convolution, as you may remember-- 603-- is the product in the transform space, right? And products commute.

So if convolution converts into a product and the product commutes, that must mean that that deconvolution commutes. Similarly, associativity-- if I've got A times B times C, I can think of that either as multiplying A times B first and then taking the result and multiplying by C. Or I can think of it as multiplying B by C first and then multiplying A by that.

Well, since, again, the transform of convolution is multiplication, the same must apply to convolution. So we can switch things around, which is what I did over here. Well, no. Here, I used commutivity.

But we're are also going to need associativity. So we can associate in convolution, which is going to be very powerful, and allow us to switch things around between the signal and the filter and so on. OK, so let's continue with this. So that's the same picture. The patent examiner just liked that and put it on the first page.

And let's look at what we have here. So the sort of smooth curve up here could be a kernel for some sort of smoothing filter, perhaps an attempt to roughly low-pass filter a signal or approximate a Gaussian or something. But it turns out that this is composed of segments that are each quadratic.

And so now we take a difference-- or in the continuous word, derivative. And what do we get? Are these straight lines?

Why are they straight lines? Because this was second order. So if we take a derivative, it'll be first order. But it's not sparse. We're not done.

So now I repeat that. We take another derivative. And because this is just a ramp, we get a constant value. And then this is a ramp going down. So we get a constant value on second derivative.

And that's a ramp-- get a constant value on third derivative. We're not done yet. Remember n plus first. So we have to do one more derivative.

Well, what's the derivative of a constant? It's 0. So all along these long stretches, potentially-- I mean, this is-- in order to fit into the figure, they've made it relatively compact. But you could imagine we could do this over a longer extent.

All of these areas, we get nothing. We get 0, which is what we want. It makes it sparse.

But at the transitions, at the places where these quadratic things were spliced together, we do get a result. And, amazingly, there are only two non-zero values. There's only that value in that value. So instead of having to convolve with something that 20 non-zero values, we convolve involved with 2, which is really cheap. Plus, they're the same magnitude. So we don't even have to multiply. We just subtract.

But then we don't get the answer we want. We get the third difference of the answer we want. So now we take the result. And we just left to right sum it up-- once, twice, three times.

Just like first difference is such a simple operation, this is also a simple operation. We start with 0 at the left. And we just keep on adding the values and writing them into an array.

And then we do that with that new array-- same thing. And then we do with the results of that. And if we've done it three times, we're done. So there's a certain cost to that.

But it's nothing compared to the cost of doing the convolution with the original function that had large support, had non-zero values for many points. So this is kind of a textbook example of this method because it's giving us a very high compression. The result after taking the third difference is very small.

Let's see-- first, second, third. Yeah, third difference. OK, so that's the basic idea. And here's another one.

Oh, wait is that the same one? Yes, that's the same one. And so here's the circuit for doing that. So here's the one-dimensional function. Think of it as a row in the image. And there's a sampling operator, which picks out every fifth value-- four of them.

And those are the results of that third difference of the convolution function. And then we convolve-- sorry, I think I screwed up here because there's non-zero values at the end as well. So there's this value here and that value.

So there are actually four non-zero values in this example. And two of them are magnitude three, and two of them are magnitude one. So that's why we're reading our four values.

And the outer ones are multiplied by plus n minus 1. The inner ones are multiplied by plus or minus 3. And we add them up. And that produces a new one-dimensional stream of numbers. And then we push it through three accumulators. And out comes the result. Let's see.

And so that's just a picture of a accumulator, where we keep the old value and we add in the new value. I'm not sure why we need a diagram for that. And that's the actual convolution operation. And then we can-- often we want to control the bandwidth. We want to control how much we reduce the resolution of the image.

Well, here we can do that very simply by changing the spacing between those four non-zero values. So we can stretch out that spline horizontally or compress it just by changing S. And so here we're looking at the ith pixel and the i minus S plus 1 pixel and the i minus twice S plus 1 pixel and the i minus 3 times S plus 1 pixel.

So we can change the scale very easily. And notice there's no change in the amount of computation, whereas if you were using the full original spline, or whatever it's an approximation of, like a Gaussian, then as you increase the size, of course the computation goes up. So here, since we're only doing any work where the parts of the spine spliced together, that doesn't change. So we can easily control the filter parameter.

Here's another one. Let's goes through this one. So this one-- we have this curve up here that's, I think, a parabola upside down. We take the first difference-- so the second order is first difference is first order. And we get this line. And then we take the second difference, and because this has a constant slope, of course, we get a constant value for the second difference.

We're still not done because it's still not sparse. We take the third difference. And, of course, we get 0 all the way in here. Now, this time, the ends are more complicated because the discontinuity-- there's a discontinuity not just in higher derivative but in the first derivative.

It's continuous in the function itself at the ends. But the first derivative has a discontinuity. And that means that the second and third derivative are going to be more complicated.

So we have these spikes at the end. And there are two non-zero values in each of these spikes at the two ends. So it's just another example of-- so if, for example, you decide that the Gaussian is a good filter for you to use because of its whatever transformed properties, then you can fit a spine to it.

And instead of doing the expensive computation-- or suppose that you say, OK, I know that the ideal low-pass filter has a sync function in the spatial domain. Well, the sync function, for a start, goes on forever. So that's going to be a problem.

But also, it doesn't have a [INAUDIBLE] support. It's non-zero almost everywhere. So I can fit a spline to it. And we'll be doing that.

And that's a particular-- of importance for us because we want to work multiple scales. So we have to subsample. Nyquist tells us-- as does Shannon and whoever invented it first. It wasn't Nyquist-- that we have to be careful that the sampling produces aliasing artifacts unless we've cut out the high frequency content before we sampled. And so we have to be able to at least approximately low-pass filter our images.

And so, how do you do that? Well, you can convolve with the sync function. But that's very expensive to even approximate directly.

If we can approximate the sync function with one of these splines, then we can make that reasonable premise. I'm not sure there's-- OK, there are more examples. It doesn't have to be just a lump.

So here we have a function that has a positive and a negative peak. Maybe that's an approximation to a first derivative operator. And again, we take the first derivative. We get the ramps, second derivative. We get constant, third derivative. We get zeros and with something at each of the transitions where the spines are stuck together.

And we reduce the computation dramatically because, in this case, we have one, two, three non-zero values. Oh, and there's is-- OK, five. Five non-zero values. And this is just about clever details on how to do the sampling efficiently.

And that's in the y direction. Now, we can do the same in the x direction. We can do the same in the y direction. And the idea would be that we run the convolution in the x direction. And then we produce a new image. And then we run the convolution in the y direction and produce the final result.

And that's one way to proceed. What this diagram shows is that you can do better by combining those two operations. So you do still need some intermediate memory. But you don't need enough memory for a whole image. You only need a memory large enough for the convolution in the other direction.

So here we go through the y convolution using the methods we just described. And we keep a few rows. And then we run the x convolution on that, each of these rows.

And so that kind of brings up another issue, which is it seems very efficient to do 1D convolutions. But to 2D convolutions are expensive. And so there's an open problem, which is can you use this clever idea of sparsifying the convolution in 2D. And I don't know-- might be a thesis topic in there.

But what you can do, meantime, is say, in some cases, maybe we can approximate a 2D convolution by a cascade of a 1D convolution in x and a 1D convolution in y. And we look at that as well. And it goes into why this is important. It talks about the Canny edge operator and other edge operators and multiscale and blah, blah, blah. This is much shorter than the other patents. But it's pretty much just using what I describe.

Just for fun, let's look at the claims. A method for digitally processing a one-dimensional signal, the method convolving the one-dimensional signal with the function that is the n plus first difference of an nth order discrete piecewise polynomial kernel so as to provide a second one dimensional digital signal, n being at least two and so on. And then perform discrete integration n plus 1 times to produce the final result.

OK. So there's some sort of things we need to fill in there that we haven't talked about. You may be wondering why I brought this. So this is a calcite crystal-- please don't drop-- which is almost optical quality.

And I got this in one of those crystal shops. You know how crystals work on your mind. You buy this and you become famous or rich or something. But this is used in optics.

And the reason is that it's birefringent. If you look through it at some surface, you will see two copies of the picture. And why are the two copies? Well, as you know, materials have refractive index, which causes light rays to be deflected by Snell's law.

Well, this material is unusual in that it has two refractive indices depending on the polarization. So since light in the room is sort of randomly polarized-- there are both types around. Both are deflected but by different degrees. So you get two copies of everything as you move the crystal.

So why is that relevant? Well that's a trick that's used in cameras. So in cameras, we have a problem with aliasing because we're sampling and we can't guarantee that the image has been low-pass filtered. And so one of the tricks that's used is to have a very, very thin layer of this material, which causes two copies of the image to appear very close together, a few microns apart.

And that's not a perfect low-pass filter. But it suppresses high frequency content. And together with other items, it overall makes the sampling better. Now, if you are willing to pay and take a chance, you can send your fancy schmancy digital SLR camera to a place that will remove this filter because-- geez, it's removing my high frequency content. I want higher resolution.

And, yes, you do that. And your camera has, apparently, higher resolution. But then heaven forbid you look at something like your jersey pullover or something with a fine pattern on it. And all of a sudden, you'll see all this interference, moire patterns.

So, yes, it's cutting high frequency content. But there's a good reason for cutting the high frequency content. And, for example, when you're brought in front of a television camera, in the old days, they used to tell you not to wear a striped tie or to wear a dress that has narrow, closely spaced stripes because you're going to end up not just with aliasing of the kind we usually see, but it's going to mess up with a color as well. So you're going to get funny colored stripes and so on.

And since then, they've improved the low-pass filtering before sampling so that this is not such a big issue anymore. But you can find a lot of videos of lectures at MIT where someone didn't pay attention to this. And instead of watching what's going on, you're watching the person's shirt change colors and shapes.

OK, so anyway, we'll talk about that next time, about what has to be done in the camera to suppress these aliasing effects. And that's the material that we use there.