[MUSIC PLAYING]

ALAN OPPENHEIM: In this lecture, we will begin the discussion of the computation of the discrete Fourier transform. This is an extremely important topic in this set of lectures for a number of reasons. One of the reasons is that the principal algorithm that we'll be discussing, the algorithm, which we'll refer to as the fast Fourier transform algorithm, relies, for its derivation, on many of the properties of the discrete Fourier transform. And so, in fact, developing this algorithm, provides us with an opportunity to exercise some of the intuition and some of the results related to discrete Fourier transforms.

Of much more practical importance is the fact that the fast Fourier transform algorithm, which we'll be presenting, is a highly efficient algorithm for computing the discrete Fourier transform. And in fact, as we'll see, it is orders of magnitude more efficient than a straightforward direct computation of the discrete Fourier transform.

Now, We've indicated in previous lectures, and you can well imagine that this is true, that there are many contexts in which we would want to explicitly compute the discrete Fourier transform. For example, obviously, in spectral analysis of discrete time data, we would want to use the computation of the discrete Fourier transform. And we've seen also that the discrete Fourier transform can be used for the implementation of finite impulse response digital filters, or more generally, can be used to implement a convolution.

It will turn out that because of the efficiency of the algorithm that we'll be discussing in these lectures, it, in fact, in many cases is more efficient to implement the convolution by explicitly computing a discrete Fourier transform, multiplying the transforms together, and then Inverse transforming the result. Of course, keeping track of the fact that that implements a circular convolution and doing the necessary things that need to be done if we're interested in a linear convolution.

Now, I'd like, just briefly, to remark that this algorithm, this-- actually, it's a class of algorithms, which I'll refer to as the fast Fourier transform algorithms, were a major landmark in the development of the field of digital signal processing in its present form. The algorithm, one of the forms of the algorithm, was disclosed in a paper by Cooley and Tukey in 1965. And it turns

out, though, that the algorithm has, in a variety of forms, been reinvented several times over the past 20 or 30 years. So it is mainly this class of algorithms that we'll be talking about in the next several lectures. And I stress again that one of the primary reasons why this is so important is because of the rather dramatic improvements in efficiency that it offers over just a straightforward computation of the discrete Fourier transform.

Well, let's look specifically at what's involved in the computation of the discrete Fourier transform. The discrete Fourier transform, as we know, is given by the sum of x sub n times W sub n to the nk, where W sub n is the complex exponential, e to the minus j, 2 pi over capital n. So it is this expression that we want to compute. Or equivalently, there is the inverse discrete Fourier transform given by 1 over capital N times the sum of x of k, the DFT, times W sub N to the minus nk.

Now, hopefully you recall from the discussion of the discrete Fourier transform, that the inverse DFT is really the same as the discrete Fourier transform if we reverse the result in N, et cetera. In other words, computation of the inverse DFT is really essentially the same as the computation of the DFT. So whereas we'll be referring the discussion to the computation of the discrete Fourier transform, essentially everything that we say applies also to a computation of the inverse DFT.

Well, let's look, first of all, at what would be involved in just a straightforward computation of the discrete Fourier transform. Basically what this expression says is to take x of n, multiply it by the appropriate powers of W, and then sum the result from 0 to capital N minus 1. And so, we basically have that set of operations to carry out for each value of k for which we're computing the discrete Fourier transform.

Well, if we think of x of n as being complex, which in the most general case it will be, then we have in the direct computation of the DFT, we have to form the product, x of n times W sub N to the nk. And what that involves is one complex multiplication. This is a complex number. That's a complex number. Multiplying the two is then one complex multiply, or equivalently, four real multiplies and two real adds.

OK, well now that is to form one term in the sum. We have n terms altogether in the sum, or capital N minus one additions, complex additions, to implement in the sum. So to compute the DFT for one value of k will require capital N complex multiplications and capital N minus 1 complex additions. Well to compute the discrete Fourier transform for all capital N values of k

requires then n squared complex multiplies and n times n minus 1 complex additions. Or basically if capital N is any reasonable number, we can approximate this as N squared. And so, approximately, the computation involved is on the order of capital N squared complex multiplies and adds.

Now, a few remarks here. One, first of all, is that in counting up the number of complex multiplies and adds, we didn't particularly take account of multiplies by unity. For example, when k is equal to 0, W sub N to the nk is equal to unity, and we, in fact, wouldn't have to carry out a multiplication. But there are a small enough number of those cases in comparison to all of the multiplies involved so that this is generally an indication of what the dependence of the computation is on N.

Second of all, and this will occur throughout this next set of lectures, we are tying a discussion of the computational complexity or efficiency to counting the number of complex multiplies and adds that are involved in the computation of x of k, and not particularly focusing on other issues, such as indexing, bookkeeping, et cetera. So in discussing the computational complexity or efficiency, we'll tend to refer to the number of multiplications and adds that are involved in implementing the transform.

Well, we have then a direct computation, which involves on the order of n squared complex multiplies and adds. And the important point then is its dependency on N, namely N squared. There are a variety of algorithms that have been known for some time, in fact, and ones that you can generate yourself by taking advantage of various symmetry properties of these complex exponentials. To reduce the constant of proportionality on the computation, that is, to make the computation perhaps not involve N squared multiplies and adds but only 1/2 N squared or 1/4 N squared. But still, in cases like that, the algorithms involved basically fooling around with a constant of proportionality out in front, whereas the main dependence on N, the size of the transform, remains the same.

And you can, I think, convince yourself rather quickly with some fairly standard computers that if, in fact, the computation to be done is proportional to N squared, then very large transforms certainly aren't possible in very reasonable amounts of time. For example, if N is on the order of 1,000, then we're talking about on the order of a million complex multiplies and adds. And if we assume that one of those took, say, two microseconds or three microseconds to implement, or even one microsecond, then clearly we're talking about computation on the order of seconds. And you can convince yourself rather quickly that the computation grows

very fast as N increases from 1,000 to 2,000 to 4,000, et cetera.

All right, so direct computation, the dependency is an n squared type of dependency. And this is in contrast to the primary algorithm that we want to discuss, which is the fast Fourier transform algorithm. Basically, the fast Fourier transform algorithm is directed at computing the discrete Fourier transform for a choice of N, the size of the transform, highly composite. That is, N is, in general-- of course, any N can be represented as a product of primes. And as we'll see, the larger the number of primes involved in the decomposition of N, the greater the efficiency of the fast Fourier transform as an algorithm for computing the discrete Fourier transform.

In particular, we'll see that for N, a product of primes, using the fast Fourier transform algorithm, the number of complex multiplies and additions involved will turn out to be proportional not to N squared as it is in the direct computation, but proportional to N times a sum of the primes involved in that decomposition of N. So the greater the number of primes that we can decompose N in and the smaller each of those primes, the higher the efficiency of the fast Fourier transform algorithm will be. Well,

The most highly composite that we can make N, and at the same time keeping the primes, p, as small as possible, is to choose N to be a power of 2. So that N is of the form 2 the nu, where nu is the log to the base 2 of N. And it will turn out, in that case, that the complex multiplies and additions is proportional to N times log to the base 2 N, rather than, in the direct computation, proportional to N squared.

Let me incidentally comment that although I've referred to these p's as primes, they, in fact, don't have to be primes. The important issue, as we'll see, is that N be representable as a product of numbers. And it's not particularly important that those numbers turn out to be primes.

Well, OK, so this then is the kind of efficiency that we will get out of the fast Fourier transform algorithm. And let me begin the discussion of the algorithm by concentrating on the case in which N is a power of 2. Well, basically, the idea behind the fast Fourier transform algorithm is to decompose the computation of the discrete Fourier transform into a set of subsequences. And this is-- this generates a class of forms for the algorithm, which are referred to as decimation-in-time forms of the algorithm.

In particular, suppose that we consider decomposing the original sequence x of n into two

subsequences. One sequence, subsequence, consists of the even-numbered points of x of n. And the second subsequence consists of the odd-numbered points of x of n. Well, let's just do that and see where it gets us. We can then rewrite the discrete Fourier transform as a sum over n even of x of n times W sub N to the nk plus the sum over n odd of x of n W sub N to the nk. Simply pulling out from this the even-numbered values and the odd-numbered values.

All right, now by a substitution of variables, we can substitute for n 2 r in the even-numbered terms. And we can substitute n equals 2 r plus 1 for the odd-number terms, in which case, when we convert these two sums over r, the sum on r will now range from 0 to capital N over 2 minus 1. As r runs from 0 to capital N over 2 minus 1, 2 r ranges over the even-numbered terms, and 2 r plus 1 ranges over the odd-numbered terms.

All right, so making the substitution of variables then, this sum is converted into a sum from r equals 0 to N over 2 minus 1. x of 2r W sub N to the 2rk plus a sum-- those are the even-numbered terms-- plus a sum over the odd-numbered terms from 0 to n over 2 minus 1. x of 2r plus 1, the odd-numbered points, times W sub N to the 2r plus 1 times k.

Well, let's look at this factor. W sub N to the 2r plus 1 times k, that's equal to W sub N to the k, because of the 1, and W sub N to the 2rk. So we can, first of all, break this term up into a product of these two terms. And now, let's look at this one, W sub N to the 2rk. Well, W sub N squared is equal to e to the minus j 2 pi over capital N squared. So the factor of 2. If we take the 2 down in the denominator and associate it with the N, we can rewrite this as e to the minus j 2 pi over N/2. So what W sub N squared is equal to is W sub capital N over 2. In other words, the W that we would associate with an N/2 point discrete Fourier transform.

Finally, substituting this together with this statement back into this expression, we have, for x of k, the sum from 0 to N/2 minus 1, x of 2r W sub N over 2 to the rk plus W sub N to the k times the sum of x of 2r plus 1 W sub N over 2 to the rk. Well, what this looks like is an N/2 point discrete Fourier transform. And what this looks like is an N/2 point discrete Fourier transform. And in fact, that's what they are.

This is an N/2 point DFT of the even-numbered points in a sequence x of n. This is an N/2 point of the odd-numbered points in the sequence x of n. So this says we can compute x of k by computing an N/2 point DFT, another N/2 point DFT, and combining those two together with this factor out in front, W sub N to the k.

Now, it's not particularly obvious yet that, in fact, in doing this, we've saved something in computation. But let's look at what this implies in terms of the computational steps in a little more detail. So here we have, again, X of k as an N/2 point DFT involving the even-numbered points in x of n plus an N/2 point DFT involving the odd-numbered points in x of n. Those are both N/2 point DFTs. This one, if we refer to as G of k, this one, if we refer to as H of k, then the computation of the DFT, X of k is G of k plus W sub N to the k times H of k.

All right, let's see what this means in terms of computation, computational complexity. We have to compute an N/2 point DFT. We just argued that an endpoint DFT involves N squared complex multiplies and adds. So the computation of G of k involves N/2 squared complex multiplies and adds. The computation of H of k involves, likewise, N/2 squared complex multiplies and adds. So we have then 2 times N/2 squared multiplies and adds due to the computation of these two N/2 point DFT.

In addition, we have to combine these together by multiplying by W sub N to the k. That involves capital N complex multiplies and adds, because k ranges from 0 to N minus 1. In other words, k ranges over N different values. So there are N complex multiplies involved in this, and then N complex adds in combining these together.

So in addition to the computation of the two N/2 point DFTs, we have the combination of them. And the resulting computation involves N plus N squared over 2 complex multiplies and adds. And you can convince yourself on the back of an envelope very quickly that this number will always turn out to be less than N squared, except, I think, for some trivial cases like N equals 1 or N equals 2.

Now, one minor point to be made-- in combining these two together, I commented that k has to run from 0 to capital N minus 1. That is, there are N different values that k ranges over, so doesn't that say that in computing G of k, I can't just compute it from k equals 0 to n over 2 minus 1 as I normally would in computing a DFT? Don't I have to compute it from 0 all the way up to N or all the way up to N minus 1? Well, sure I do. I do, because those are the values that I need in here and in here.

But what does that imply in terms of computation? Well, we know that an N/2 point DFT is periodic in k, so that if we compute only the first N/2 points, we know that the next N/2 points are going to look exactly the same. And so, there's no point in computing them. So the point then is that in computing G of k for this computation, even though we want k to run from 0 to N

minus 1, we only need to explicitly do this computation for k from 0 to N over 2 minus 1. And then we simply use those values over again for the second range from k from N/2 to N minus 1. And that, by the way, is the essential key in the fast Fourier transform algorithm. It's basically that periodicity in the discrete Fourier transform that will lead to the kind of computational efficiency that's about to result.

All right, so let's look then at how we would actually do the computation of the discrete Fourier transform. We know then that we will break the computation into a computation involving the even-numbered points and x of n and the odd-numbered points and x of n. Compute an N/2 point discrete Fourier transform to obtain the sequence G of k, another N/2 point discrete Fourier transform to compute the sequence H of k, and then combine the results together by multiplying H of k by W sub N to the k and adding that to G of k.

I'll illustrate the computation, incidentally, with choosing N equal to 8, simply because I have to choose N equal to something. And it turns out, interestingly enough, that N equals 8 is sufficiently general, so that you can infer the generality for N equal to an arbitrary power of 2 just from the computational issues that we'll raise for this particular case. All right, so we have N/2 point DFT of the even-numbered points, the N/2 point DFT of the odd-numbered points.

And now, to compute the output points, the discrete Fourier transform, X of k, we then want to take G of 0 and add it to H of 0 multiplied by W sub N to the 0. So using flow graph notation, we have G of 0 added to H of 0 multiplied by W sub N so the 0. For x of 1, we want G of 1 added to H of 1 multiplied by W sub N to the 1. For x of 2, we want G of 2 plus H of 2 multiplied by W sub N squared. And for x of 3, likewise, G of 3 added to H of 3 multiplied by W sub N cubed. So that computation, then, gives us x of 0 through x of 3.

For x of 4, we want G of 4 plus W sub N to the 4 times H of 4. But what's G of 4? G of 4 is equal to G of 0, because of the fact that G of k is periodic with period 4. This is a four-point DFT that we're computing. So G of k is periodic with period 4. And consequently, H of 4 and G of 4 are equal to H of 0 and G of 0.

So to compute the next four points, we can likewise use the outputs of these N/2 point DFTs. To compute x of 4, we want G of 4, which is equal to G of 0, because of the periodicity, plus W sub N to 4 times H of 4. H of 4 is equal to H of 0. So we have then for x of 4, G of 0 plus W sub N to 4 times H of 0. Likewise, for x of 5, we have G of 5, or G of 1, plus W sub N to the 5th times H of 5, or H of 1 because of the periodicity, and likewise, down through the remaining

four points.

So we showed previous-- on the previous view graph, the computation for the first four points. And here we have the computation for the last four points. Putting those two together then, the computation for the eight-point DFT is, as I've indicated here, the even-numbered points into an N/2 point DFT, the odd-numbered points into an N/2 point DFT, and the results combined together according to the computation that we've shown here.

Now, a couple of things that I'd like to draw your attention to about the flow graph as it's developing. First of all, note that the flow graph, as we're looking at it here, involves a computation, which if we were to just pull out the computation of these two points, or these two points, et cetera, it has the structure of a set of butterfly-looking flow graphs. And, in fact, we'll find it convenient to refer to these as butterflies in the computation.

The second thing that I'd like to draw your attention to is that the multiplications by powers of W always are applied to the branches which come from this bottom transform. And that's because we had the expression that X of k was equal to G of k plus W sub N to the k times H of k, and this was the H of k. So it's in the bottom branch of these butterflies that we see always the multiplication by these powers of W. And I say that, I want to point that out hit now, because it's somewhat difficult in drawing these flow graphs to always get these powers of W close enough to the arrow so that it's clear where the computation, where the multiplication needs to be carried out.

All right, so here's where we're at. We had counted out previously the number of complex multiplies and adds involved, and we found that it was on the order of N plus 2 times and over 2 squared, which is less than N squared. And consequently, this is a more efficient way to compute the discrete Fourier transform than simply a direct computation.

However, like this assumes that we're going to compute these N/2 point DFTs as a direct computation. But in fact, if N/2 is a composite number, in particular a power of 2, which it is if N was a power of 2, then we can apply the same strategy to the computation of these N/2 point DFTs. So we can break these N/2 point DFT computations into-- each one of these-- into two N/4 point DFTs plus some additional arithmetic in order to form the N/2 point DFTs.

So each one of the N/2 point DFTs can then be decomposed into an N/4 point DFT computation, into two N/4 point DFT computations. In particular, we had, recall, for the top half of the previous view graph, the even-numbered points in the original sequence. All right, now

we want to compute the DFT of those even-numbered points. So we can go through exactly the same steps. We can break that into its even-numbered points and its odd numbered points.

So we have the even numbered of the even-numbered points into an N/4 point DFT, the odd numbered of the even-numbered points into a second N/4 point DFT, and then those results are combined in exactly the same way as the N/2 point DFTs were combined in the previous view graph, but now with N/2 as the subscript rather than N as the subscript, because here what we're computing is an N/2 point DFT. However, in drawing the flow graph, we can re-express the powers of W sub N over 2 in terms of powers of W sub N, because of the fact that W sub N squared is equal to W sub N over 2 or vice versa.

All right, so we have this then apply to the even-numbered points and to the odd-numbered points. And when we do that, then the flow graph that results is as I've indicated here. We have the even numbered of the even-numbered points and the odd numbered of the even-numbered points. We have the even numbered of the odd-numbered points and the odd numbered of the odd-numbered points. Those are each going into N/4 point DFTs.

Here is then the sequence G of k. I'm sorry. Here is the sequence G of k. Here is the sequence H of k. This then implements the computation of the N/2 point DFTs that we had before. And then we have the output DFT obtained by combining those. So this flow graph results from simply carrying the previous strategy back one step to implementing the N/2 point DFTs.

All right, well, now we have the computation of N/4 point DFTs. And we can do exactly the same thing with the N/4 point DFTs, except that for this particular example, N/4-- if N is equal to 8, N/4 is equal to 2. The computation of the two-point DFTs is a relatively straightforward thing. But in general, if N wasn't equal to 8, if it was a more general power of 2, we could likewise apply this strategy to the computation of the N/4 point DFTs, the N/8 point DFTs, and over 16 point DFTs, et cetera, until we got to the computation of two-point DFTs.

All right, so just simply inserting the computation for these two-point DFTs, which involves simply adding-- well, adding x of 4 to x of 0, multiplying x of 4 by W sub N to the 4 and adding it to x of 0. The resulting flow graph for the entire computation is as I show here, where we have now the input points. We have the output points. Here we're computing the two-point DFTs, combining those together to get the four-point DFTs, combining those together to get

the eight-point DFTs, et cetera.

And I'll remind you again that the basic computation that's involved throughout these stages is the butterfly computation. We see it here, we see it here, and we see it here. And always, the multiplication by powers of W is applied to the branches coming from the bottom node of the butterfly. And that's because of the fact that X of k is expressed as G of k plus W sub N to the k times H of k. And it's always the H of Ks that are what are being computed in the bottom half as we've been developing this flow graph. It will turn out that that's not true for other forms of the algorithm, but for the form that we're developing here it is.

All right, well, having carried this all the way back to the two-point DFTs, let's see what this has accomplished for us in terms of computational efficiency. We began with N equal to a power of 2. We thought about computing a direct discrete Fourier transform, but that involved N squared complex multiplies and adds. Instead of that, we broke the computation into two N/2 point DFTs plus some diddling to fit those together. The computation of the two N/2 point DFTs involved complex multiplies and adds equal to 2, because there were two of them, times N/2 squared plus N for the diddles-- the diddles being involved in fitting those two N/2 point DFTs together. And this, for N greater than two, is a bigger number than that.

All right, then we said, well, why stop there? Let's break these two N/2 point DFTs into four N/4 point DFTs. That meant then that this N/2 squared, which came from a direct computation of the DFT, gets replaced instead by two N/4 point DFTs. So we have 2 times N/4 squared plus N/2, which are the diddles involved in, again, fitting those together.

So the N/2 squared, which involved computing one N/2 point DFT gets replaced by this when we go to the N/4 point DFTs. And consequently, substituting this in there, the computation is-- instead of that, it's N/4 squared times 4 plus N plus N. This N coming from that 2 times this N/2.

Well, we can go through this again. We broke the N/4 point DFTs into N/8 point DFTs. Therefore, the N/4 squared got replaced by N/8 squared twice, plus N/4 to fit those together. That gets substituted in for this N/4 squared. We get 8 times N/8 squared. We pick up another N, which is this 4 times that N/4. And you can see that each time we do this, this number gets smaller, and we keep picking up another factor of N added in.

Well, if you track this through, what you find is that for N equal to a power of 2, in fact, the computation then comes out proportional to N plus N plus N nu times, where nu is log for the

base 2 of N. And so, the number of complex multiplies and adds involved in implementing the discrete Fourier transform using the fast Fourier transform algorithm is proportional to N times nu rather than proportional to N squared. A rather dramatic savings.

OK, well there is one-- there are a number of modifications that we're going to want to make to this basic structure for the computation as we go through this lecture and the next several lectures. The first is to introduce a fairly straightforward simplification by, first of all, again, reminding you that the basic computation involved always, from one stage to the next, is always a butterfly type of computation with multiplications by powers of W applied to the signal from the bottom node.

And you should notice, and it's rather straightforward to verify why this is true, that in each of these butterflies, the separation in powers of W between the factor going to the top node at the end of the butterfly and the factor going to the bottom node in the butterfly, they are always separated by N/2. So that means that the basic computation in the computation of the DFT, as I illustrated on the previous flow graph, is a butterfly, where the characteristic of it is that it's multiplication by unity from the top node, and multiplication by some power of W from the bottom to the top node, and a power of W displaced from this one by N/2 from the bottom node to the bottom node.

Well, let's decompose this power of W into W sub N to the r times W sub N to the N/2. W sub N to the N/2. Well, what is that? That's e to the minus j 2 pi over capitalized N-- that's W sub N-- raised to the N/2 power. The capital N's cancel out. The 2's cancel out. We're left with e to the minus j pi, which is equal to minus 1. So then, in fact, this factor, or equivalently, this factor, is equal to minus 1 or minus W sub N to the r.

Consequently, we could redraw the butterfly so that rather than implementing that multiplication in both the branch going to the top node and the branch going to the bottom node, we simply multiply the value coming into the butterfly by W sub N to the r, and then add it to this value at the top node, and subtract it from this value at the bottom node. So we could redraw our original flow graph by replacing the butterfly computation in the previous flow graph by this butterfly computation, where the multiplication by the power of W is extracted from inside the butterfly, so that basically we reduce the number of multiplications by W that we have to implement by a factor of 2.

The flow graph that results when we do that we have here then, where the powers of W are

now slid out of the butterfly to the entering node to the butterfly. And we now have, rather than multiplication by powers of W in the butterfly, each butterfly computation consists of an addition and a subtraction. So the basic computation now is a multiplication by a power of W followed by an addition and a subtraction. And this follows all the way through a multiplication by a power of W and then an addition and a subtraction, this minus 1 being associated with this line. The minus 1's, of course, always occur on the branches coming from the bottom node of the butterfly. That's consistent with everything that we've been saying throughout this discussion.

All right, so this then is one form of an algorithm, referred to as the fast Fourier transform algorithm, for implementing the computation of the discrete Fourier transform. As the algorithm developed, it developed somewhat naturally in a flow graph form. And in fact, we can interpret this flow graph as a computational recipe for computing the discrete Fourier transform.

There are a few things that I would like to draw your attention to. One primary one in particular at this point, that is that we notice that the input sequence values, because of the way that we derive this algorithm, the input sequence values got rearranged in a somewhat funny order at the input to the flow graph. Although, they come out of the flow graph in a sequential order, 0, 1, 2, 3, et cetera. Over here, it's an order of 0, 4, 2, 6 1, 5, 3, 7.

It will turn out that that order has a certain amount of significance. In fact, as we'll see, as we continue this discussion in the next lecture, that this is in fact a bit reversed order. It corresponds to rearranging the order of the bits in the binary representation of the indices. And furthermore, it's possible to interpret this flow graph in terms, not just of the computation of the discrete Fourier transform, but it also suggests how to arrange the memory, that is, the storage of the input sequence and the resulting storage of the output sequence. There are some efficiencies that result when we do that. And that's one of the first topics that we'll begin with next time.

So in the next lecture then, we will continue on the development of the fast Fourier transform algorithm, picking up from the point that we've reached so far, which is-- the point that we've reached is to have exposed one form of the fast Fourier transform algorithm. There are some important points to be made about that form. And then a variety of modifications of that form that have some advantages, and, as it will turn out, also some disadvantages. Thank you.

[MUSIC PLAYING]