ALAN OPPENHEIM: In the last lecture, we began the discussion of the computation of the discrete Fourier transform. And in particular, we developed a flow graph for one algorithm. The algorithm that we talked about last time, which is referred to as the decimation in time form of the fast Fourier transform algorithm, was derived by decomposing the original sequence into subsequences.

First, two subsequences consisting of the even numbered points and the odd numbered points, implementing an N-over-2-point DFT on the even numbered points and an N-over-2-point DFT on the odd numbered points, and then combining those results in an appropriate way to obtain the discrete Fourier transform of the overall sequence. And we saw, first of all, that there was a computational efficiency that resulted from doing this. And furthermore, we recognized that we could continue this decomposition and thereby achieve an even greater efficiency.

In particular, with N expressed as a power of 2, we could continue decomposing the N-over-2-point DFTs into N-over-4-point DFTs, the N-over-4-point DFTs into N-over-8-point DFTs, et cetera. The flow graph that resulted when we did this was the flow graph indicated here, which is the one that we derived last time. It consists essentially of a set of butterfly computations-- what we refer to as butterflies-- a multiplication on the bottom branch of the butterfly at the input to the butterfly, and then an addition and subtraction. And that basically is the type of computation that follows throughout the flow graph.

Now, this is a flow graph representing the computation of the discrete Fourier transform. And what is important about the flow graph are the nodes and the connections between the nodes and the transmittances on the branches that connect the nodes together. However, the flow graph in the form that is depicted here suggests a strategy for organizing the computation of the DFT. In particular, what we recognize is that the flow graph in this form tends, in a natural way, to suggest organizing the computation corresponding to a computation of an input array-- a computation on an input array-- computing a second array at this stage, from these points, computing this array, and from these points, finally computing the DFT output points.

If we do that and we in fact think of these points as corresponding to sequential memory registers, then we know, first of all, that the input points would be stored sequentially, not in their normal order, but in fact, in some altered order. And as you'll see in a few minutes, what this order corresponds to and will be referred to as is bit reversed order.

However, an additional point that I'd like to make first about this flow graph, and that is that it corresponds-- when thought of as a procedure for organizing the computation-- corresponds to an in-place computation of the discrete Fourier transform. Now, what I mean by that is the following.

Let's suppose that we think of this set of points as corresponding to sequential registers in memory and we'd like to compute from this set of points this second array that is the set of points here. What we notice is that because of the butterfly structure of this computation, it is these two points that are used to compute these two points in the next array. These points are used to compute these two points in the next array, et cetera, so that in fact, we can organize the computation by implementing the computation required on these two input points, storing the results in a corresponding set of memory locations, which in fact could be the original memory locations from which we took these data points.

The reason for that is that once we've used these two data points, we don't need them any longer. And so the result of this butterfly we could in fact store back in the original memory locations. And a few minutes of reflection on this flow graph in general would indicate that this is true throughout the computation.

It's clearly true in going from the first stage to the second stage. In proceeding from this stage to the next stage, we see again, for example in computing these two points, the only input points that we use are these two. So again we could think of storing the result of this computation for this butterfly, for example, back in the original memory locations.

So if we think of nodes on the same horizontal line as corresponding to identical memory registers, then we see that with the computation organized in the way that this flow graph suggests, that in fact the computation can be done in place. And we'll see that with some other alternative forms of the FFT algorithm, some alternative forms of this flow graph, there perhaps are other advantages, but for some of the other flow graphs, the computation can no longer be done in-place.

So first of all, this flow graph represents an in-place algorithm. Second of all, the input points are in some altered order, although the DFT output points come out in what is normal order. That is, sequential order if we think of these as sequential registers. Well, let's examine in a little more detail what the order is that is being generated here.

If we simply look at the memory registers and the corresponding data index from the previous flow graph, in storage registers 0 we're storing x of 0, in storage register 1 we're storing x of 4, storage register 2, x of 2, and et cetera. These are the storage register locations. These are the data indices.

If we write the storage register locations in binary and we write the data index in binary, what we see, interestingly enough, is that the data index is the bit reversed counterpart of the storage register index. So here, for example, we have 001, storage register 1. Here we have 100, which is data index 4. And this continues on throughout. 110 is storage register 6, bit reversed is 011, which is data index 3.

Now, it's not accidental that this developed in such a way that the data is in fact stored in a bit reversed order. The reason for that basically is because of the way in which we originally sorted the sequence into the even numbered points and the odd numbered points, then the even numbered of the even numbered points, and the odd numbered of the even numbered points, et cetera.

In particular, suppose that we considered a tree to carry out the sorting that we implemented in breaking the original sequence up into subsequences. Well, we started with the original sequence and we divided that into two subsequences, one corresponding to the even numbered points, the second corresponding to the odd numbered points.

If you look at the binary representation of the data index, you can identify the even numbered points by the points for which the least significant binary bit is a 0 and the odd numbered points for which the least significant binary bit is a 1. So we divided the original sequence into two subsequences. The top half all had 0 as the least significant bit. The bottom half all had 1 as the least significant bit.

Well, then for the even numbered points we wanted to divide those into the even numbered of the even numbered and the odd numbered of the even numbered. How would we do that? Well, we would do that by examining the next least significant bit.

If that's a 0, then we would identify the data as being an even numbered of the even numbered points. If it's a 1, it's an odd numbered of the even numbered points. So this data was sorted as we've indicated here.

And then, of course, we continue on looking at the bits from the right to the left. So sorting, then we could sort in the order that resulted by constructing a tree, as I've indicated here, and examining the bits from the right to the left.

Suppose that we wanted instead to sort the data, not as I've indicated here, but in normal order. In other words, we have a bucket of data. We want to sort the data so that the data index comes out in normal order. How would we do that with a similar type of tree structure?

Well, to decide whether the data index is in the top half or the bottom half we would look at the most significant bit. If the most significant bit is a 0, then the data is in the top half of the entire set of data. And if the most significant bit is a 1, the data is in the bottom half.

Next, we would proceed to the second most significant bit. If the second most significant bit is a 0, then we fall either into the top half of the top half or the top half of the bottom half. And we continue on like that going through a tree exactly identical to this, but examining the bits for sorting in normal order from the most significant bit down to the least significant bit.

So it is basically that in sorting the data as we sorted it to develop the flow graph that we developed, we examined the bits in exactly the reverse order that we would examine the bits if we wanted to sort the data normally. And consequently what results is data that's sorted in bit reversed order.

Incidentally, speaking of things being bit reversed, the view graph as I first got it back from drafting I thought you might enjoy looking at. And if you find your mind wandering during the rest of the lecture, you might try to figure out exactly what happened with this view graph.

The first inclination is that I simply have it turned around, so we can try that. Well, that doesn't quite do it. So maybe we should turn it over. And that doesn't quite do it. And in fact, I haven't yet been able to figure out a way of sorting this particular view graph in any normal order and I'd be curious as to whether you can figure out a way to straighten it out.

Well, in any case, what we have then is a sorting of the original data-- returning now to the flow graph that we saw previously-- a sorting of the original data in bit reversed order and the output resulting in normal order. Well, of course this isn't the only way to organize the computation of the discrete Fourier transform.

And in fact, an important aspect of the flow graph is as I stressed previously that what counts about the flow graph are the nodes and the way they're connected and not the way the flow graph is laid out in a sheet. When it's laid out in a sheet, we were able to apply some additional interpretation having to do with a way of organizing the computation from stage to stage. But it's important to recognize that if you take this flow graph, roll it up in a ball, as long as the right points are connected to the right points, then it still represents a computation of the discrete Fourier transform.

So in particular, we are free to rearrange this flow graph in any way that we would like to as long as we don't change the connections between the notes. Well, one possible thought is to rearrange the flow graph so that we avoid this problem of bit reversal on the input. We can think of rearranging these lines so that the input is changed to an input in normal order. And one way of doing that, in fact, is to simply take all of the horizontal lines and rearrange them so that the data is sorted correctly.

So we would leave this line where it is, we would take this line and move it up to here, this line would then stay where it is, this line would move up, and they would continue to be rearranged. The input would be then in normal order, the output, of course, would be disturbed, and we could anticipate exactly how it's going to be disturbed, in fact. The output then will come out in bit reversed order.

That flow graph is then as I've indicated here. And this then corresponds to a rearrangement of the previous flow graph, where we have rearranged the computation so that the input is now in normal order. The way in which we rearranged it, the output then comes out in bit reversed order.

Incidentally, on most of the following view graphs, I have indicated a reference to the figure in the text, which this view graph corresponds to, because in fact, many of these flow graphs will be going through quite a number of flow graphs. Many of them look very similar, so I thought perhaps a reference to the figure in the text will help you keep them organized.

So now we have a modification of the decimation in time form of the FFT algorithm, which is such that if we now think of this as the procedure for organizing the computation, the input is now in normal order, the output is in bit reversed order. Have we destroyed the fact that the original computation was an in-place computation? Well, in fact, we haven't, because if we look at this now the way that it's organized, again, thinking of this as sequential memory registers, then, again, if we, for example, select this butterfly, the computation of these two points requires only these two input points. So once we have used these two points, we can store them back in the original memory locations.

What makes the flow graph correspond to an in-place computation is the fact that the output nodes of the butterfly are horizontally adjacent to the input nodes of the butterfly. If that property of the flow graph is destroyed, then the flow graph will no longer correspond to an in-place computation if we think of all nodes on the same horizontal line as corresponding to identical memory registers, or equivalently, that vertical nodes correspond to sequential memory registers.

Well, there are a variety of other ways in which we can rearrange this flow graph. We can, of course, modify the flow graph so that we have not only normal order at the input, but so that we also have normal order at the output. And we can do that simply by demanding it. In other words, by rearranging these nodes, keeping these nodes in normal order, rearranging these nodes so that they also correspond to normal order. And the flow graph that results in that case is the one that I've indicated here.

So this is now the decimation in time form of the algorithm with the input sorted in normal order and the output also sorted in normal order. The computation still is represented by butterflies, but the butterflies are distorted. In other words, each of the butterflies no longer corresponds to the computation of horizontally adjacent nodes in the flow graph.

So in fact, this computation is no longer an in-place computation. Well, is it an in-place computation from this array to this one? As it happens, it is in the computation of the first array, the reason being that in each of the butterflies, again, the output nodes are horizontally adjacent to the input nodes.

But that property is lost after this array. And proceeding from this array to this array, we see that to compute these two points-- let's take this one and this one-- we need this point and this point. So we couldn't store then this answer back in here because we're going to need this point to compute some other output points, et cetera.

So this flow graph in this form has the disadvantage that the in-place computational aspects of the flow graph are lost although the input is in normal order and the output is in normal order. It has another disadvantage incidentally, and that is that the indexing is somewhat difficult in contrast to the two flow graphs that we've talked about so far where the indexing strategy is relatively straightforward from stage to stage.

Now, there's another rearrangement of this flow graph which has some advantages and also some disadvantages. First, let me return to the original flow graph that we began with, which consisted of the input sorted in bit reversed order and the output sorted in normal order.

The indexing in this flow graph is relatively straightforward in that you should notice that as you proceed from stage to stage, the width of the butterflies continues to double. So here the width of the butterfly is 1, here the width of the butterfly is 2-- or the height of the butterfly-- here the height of the butterfly is 4. If we were computing a larger order transform, then that procedure would continue.

But as you can imagine, if we were implementing this on a computer or with special purpose hardware, because of the fact that the data separation as we compute each butterfly increases or changes as we go from stage to stage, what is required is random access memory. We require random access memory because in each array we are not accessing data from sequential registers. We are going from this array to this one. After that, in computing a butterfly, we no longer are accessing sequential registers.

There is, however, a modification of this algorithm which is useful in the sense that it permits the computation without random access memory and in particular is organized so that it can utilize sequential memory, although it does not correspond to an in-place computation. And that form of the algorithm, I have indicated here, is a form of the algorithm that originally was proposed by singleton.

This then is a form of the decimation in time algorithm. It's a rearrangement of the flowchart so that sequential memory can be used rather than random access memory. Let me point out first of all that the input is in bit reversed order, the output is in normal order, and now let me explain in a little more detail why this flow graph allows the use of sequential memory rather than random access memory.

First of all, let me indicate that the organization of this flow graph is identical from stage to stage. In other words, if you think of how this piece of the flow graph looks in computing the first stage, it is exactly the same with regard to data access as computing this stage is in relation to the one before it, and this continues on through. So as opposed to the other forms of the algorithm, the indexing is identical from stage to stage in this form of the algorithm.

Now, to utilize sequential memory-- sequential memory might be disk memory or drum memory or shift register memory, for example. Generally, bulk memory is sequential memory rather than random access. Let's think of the original data first shuffled in bit reversed order and then stored in two separate memories. Let's call them M sub A and M sub B.

The first half of the data is stored in memory A. The second half of the data is stored in memory B. And then let's permit two additional sequential memory registers, memories M sub C and M sub D for storing the output of the computation.

Then the computation would proceed essentially as follows. We would access the first two data points from memory A, use those to compute the first point in this array, store that in memory C, and use that to compute the first point in the second half of this array and store that in memory D. Next, we would access the next two input points from memory A, add those-- we would do the required computation-- store the result in the next location in memory A, also store the second half of the butterfly output in the second register in memory D.

So as we proceed along, we access the first two points from memory A then the next two points from memory A, storing in memory C and memory D. When we have gone through the first half of the data, we now access the data from the second input memory, memory B, the first two points, do the required computation, and store in the next sequential registers in memory C and memory D, et cetera.

So the strategy in this case, then is that the input is stored half in memory A, half in memory B. The data is first accessed from memory A and we alternately put results in memories C and D. When we finish with memory A, we then proceed to memory B, access data sequentially, and continue to store the results alternating in memory C and D. Now that gives us the result at this stage.

To compute the next stage, we now start with the memory C and D as the inputs and store the results of the computation back into memories A and B. So in a sense, you can think of the computation then as involving four memory, four separate sequential memories, A, B, C, and D. We start with A and B, flush the data into memory C and D, then flush the results back.

And in fact, you can think of the computation much the same way as a slinky toy where you bounce the data back and forth between these two sets of sequential memories. That then is a form of the computation which is not an in-place computation. And furthermore, the input is bit reversed, although the output is in normal order, and it has the advantage, though, that the indexing is identical from stage to stage and consequently can utilize sequential memory rather than requiring random access memory.

Well, that then completes the discussion of the decimation in time form of the algorithm. There are, of course, a number of other variations on this and some of them are discussed in the text. What I'd like to proceed to now is a slightly different form of FFT algorithms which are really derived on a somewhat different basis than the decimation in time form of the algorithms. But we'll see as we finally continue the discussion that there is a very close relationship between the decimation in time algorithms as we've just talked about and the class of algorithms which I would now like to introduce, which are the decimation in frequency forms of the FFT algorithm.

Well, in particular, the notion in deriving the decimation and frequency forms of the FFT algorithm is, essentially, rather than breaking the input up into the even numbered and the odd numbered points, organize the computation so that we compute separately the even numbered and odd numbered output points. In particular, let's look again at the general form for the discrete Fourier transform x of k given by this sum and let's consider decomposing this into a sum over the first half of the input points and a sum over the second half of the input points.

Well, the sum over the second half of the input points we can rearrange somewhat differently by essentially implementing a substitution of variables so that the index on the sum is changed to an index from 0 to N over 2 minus 1. And if we do that, the sum that results then, the expression that results, is W sub N to the N over 2 k times the sum of x of n plus capital N over 2 times W sub N to the nk. And you can easily check that this is correct.

For example, for little n equal to capital N over 2 here, we have x of capital N over 2 W sub N to the capital N over 2 k. Here, for little n equals 0, we get W sub N to the capital N over 2 k times x of capital N over 2, and this term becomes 1. Just simply a substitution of variables.

Well, we recognize this term, W sub n to the N over 2 k, is equal to minus 1. And consequently, then we can rewrite this as I've indicated here, x of k is equal to a sum from n equals 0 to N over 2 minus 1 of x of n plus minus 1 to the k-- k being the index on the DFT-- times x of n plus capital N over 2 W sub N to the nk.

Well, it's tempting to look at this and say kind of in analogy with the decimation in time-- the steps we took in the decimation time algorithm, that this is an N-over-2-point DFT. It is a sum from 0 to N over 2 minus 1, it's a modified input sequence, but it's not, in fact, an N-over-2-

point DFT. And the reason that it isn't is that this is W sub N to the nk whereas if it was an N-over-2-point DFT, this would be W sub capital N over 2 to the nk.

However, let's look at these DFT points for two separate cases, one being that for which the index, the output index, is even and the second for which the output index is odd. The output index is even, then we can think of that as indexing through the DFT points with an argument x of 2r, where r ranges from 0 to N over 2 minus 1. And we then have this sum since k is even, minus 1 to the k is positive and we have then this sum that I've indicated here, whereas for k odd, we choose an index 2r plus 1, where again r ranges from 0 to N over 2 minus 1. And because k is odd, what results in the sum is a subtraction rather than an addition. And then we have substituting in also. We have W sub capital N to the little n times W sub capital N to the 2rn.

Now, finally, we can take advantage of the fact that W sub N to the 2rn can be written in terms of W sub capital N over 2. In particular, W sub capital N to the 2rn is equal to W sub capital N over 2 to the rn. And that follows as it did and we utilize that fact also in deriving the decimation in time form of the algorithm. It follows simply by substituting in to the expression for W sub capital N.

Finally utilizing this, we recognize that for the computation of the DFT for that the output index even, the computation can be reduced to a sum of a sequence g of n, where g of n is the sum of the first half and the last half of the input data points. That multiplied by W sub capital N over 2 to the rn.

Well, this is now an N-over-2-point discrete Fourier transform. It involves N-over-2-points and the powers of W involved are the appropriate powers of W for-- or rather the W is involved is the appropriate W for an N-over-2-point DFT. That's indicated by this subscript N over 2. So this is to compute the even numbered points.

To compute the odd numbered points, we have a similar expression. We have the sum of h of n times W sub capital N to the n, and that times W sub N over 2 to the rn, where h of n is the difference between the first half of the data points and the last half of the data points. So basically, following this strategy, what this says is that we can compute the discrete Fourier transform by forming a subsequence, which is the sum of the first and last half of the points, and computing the N-over-2-point DFT of that, and then forming a sequence, which is the difference of the first and the last half of the input points, multiplying that by W sub capital N to the little n, and computing the N-over-2-point DFT of that. And if you count up the number of operations involved, multiplications and additions, you will find that there is exactly the same computational efficiency implied in this decomposition as there was as we went through the decimation in time form of the algorithm.

So this then is the basis for the decimation in time form of the computation and it basically states that we would compute the discrete Fourier transform by first forming a subsequence, which we denoted as g, which we obtain by adding the first half of the input points to the last half of the input points, and implementing an N-over-2-point DFT of that to obtain the even numbered output points. And then we would subtract the first half of the input points from the last half of the input points, multiply that subsequence, h, by successive powers of W, compute an N-over-2-point DFT of that, and the result would then be the odd numbered output points.

Well, just as we did with the decimation in time form of the algorithm, we can continue this procedure. In other words, we can decompose the N-over-2-point DFTs into N-over-4-point DFTs by adding the first half of the input points here to the last half, et cetera. As we proceed through, we would then develop a flow graph in which we would compute first the even numbered of the even numbered output points and then the odd numbered of the even numbered output points, et cetera. So you can imagine that as we proceed through this, we'll get a flow graph.

Similar in many respects to the flow graph that we developed for the decimation in time form of the algorithm, and furthermore, the flow graph as it naturally develops this way will result in data, the DFT output, sorted in a bit reversed order. So continuing on then, here is the decomposition with the N-over-2-point DFTs broken into N-over-4-point DFTs, and we are now computing the even numbered of the even numbered output points first.

The N-over-4-point DFTs, if we consider the specific case of N equals 8, the N-over-4-point DFTs are just simply 2-point DFTs, which involve, as they did in the decimation in time form of the algorithm, just simply a computation involving an addition and a subtraction. In other words, just a simple butterfly. So the resulting flow graph based on pursuing this strategy through the entire computation is as I've indicated here, which is one form of the decimation in frequency form of the algorithm.

Notice that the input is in normal order, the output is in bit reversed order, the flow graph as it developed here, again, if we think of it as a computational strategy, a strategy for organizing the computation, again, corresponds to an in-place computation. It's an in-place computation because output nodes for a butterfly are horizontally adjacent to the input nodes of the butterfly.

In many respects, in fact, it looks somewhat like the decimation in time form of the algorithm when we sorted things such that the input was in normal order and the output was in bit reversed order. In fact, there is a difference between this class of algorithms and the decimation in time form of the algorithm. One difference that I would just draw your attention to quickly is the fact that the butterflies in the decimation in frequency form of the algorithm, as we've just been talking about it, involve additions and subtractions, and the multiplication by powers of W is implemented on the output of the butterfly. Whereas for the decimation in time form of the algorithm, the multiplication by a power of W was implemented at the input to the butterfly followed by an addition and subtraction.

So in fact, the decimation and frequency form of the algorithm as we've just developed it is somewhat different than the decimation in time form of the algorithm. As we'll see as we continue in the next lecture, there are modifications of this form of the algorithm just as there were for the decimation in time form of the algorithm. And we'll also see that in fact there is a very close relationship between these two forms of the algorithm, the relationship being suggested by properties of flow graphs that we've discussed in some of the previous lectures.

So at this point, then we have concluded our discussion of the decimation in time forms of the algorithm, we've introduced the decimation in frequency form of the algorithm. In the next lecture, what I would like to do is continue on a discussion of the decimation in frequency forms, in particular discussing alternative forms which are similar to the alternative forms that we discussed for decimation in time. Thank you.