[MUSIC PLAYING]

ALAN OPPENHEIM: In several of the previous lessons, we've focused primarily on the class of linear shift-invariant systems that are representable by linear constant coefficient difference equations. And we've talked, for example, about methods for solving that class of equations. One of the reasons why a linear constant coefficient difference equations represent a particularly important class of discrete time systems is because of the fact that they have a particularly straightforward implementation in terms of digital components.

So in this lecture, and actually over the next several lectures, what I'd like to focus on is that class of linear shift-invariant systems and in particular the implementation of that class of systems, in other words, digital networks for the implementation of systems describable by linear constant coefficient difference equations.

Well, as we have defined previously, an n-th order linear constant coefficient difference equation essentially corresponds to a linear combination of delayed output sequences set equal to a linear combination of delayed input sequences. This is the general form for a linear constant coefficient difference equation, which as we discussed in one of the early lectures, actually Lesson 3, corresponds to an explicit input output description for the system. In other words, we can solve this equation by taking this set of terms over to the right-hand side of the equation. And the difference equation that results is a statement that says that y of n, the output sequence for the system, is a linear combination of past outputs plus a linear combination of delayed inputs.

Now, we also discussed at that point in one of the early lectures the fact that a linear constant coefficient difference equation is not a unique description for a system. In other words, we saw that this system, or a system that satisfies this equation, could correspond to a causal system. Or, in fact, it could correspond to a non-causal system, depending on the initial conditions or the boundary conditions that we impose on the equation. We could in particular impose the boundary conditions implied by the statement that the system is causal, or we could impose boundary conditions implied by the statement that the system is non-causal.

For this discussion, for the discussion in this lecture and actually over the next several

lectures, the assumption will always be that we're talking about a causal system. In other words, the boundary conditions that we're imposing is that for the input 0 for n less than, say, n1, the output will also be 0 for n less than n1. So we're implying a set of boundary conditions by virtue of the fact that we're assuming that we're talking about causal systems.

Now, in implementing a system describable by a difference equation of this form, what we see is that there are three basic operations that are involved. We have to somehow generate delayed sequences. We have to get y of n delayed. We have to get x of n delayed. So one of the basic operations is the operation of delay.

Furthermore, we need to multiply these delayed sequences by scalars and these delayed sequences by scalars. So there is the additional operation of multiplication.

And third, we have to accumulate these sums, or accumulate these products. And so the third operation that is required in the implementation of a linear constant coefficient difference equation is the operation of addition.

Consequently, if we were to think of basic digital network elements, we could choose as our basic elements the operations corresponding to delay, multiplication, and addition. In particular what we'd like to develop, or generate, is a graphical representation of the difference equation, which is what we'll refer to as the digital network. So in essence, what we want is a graphical representation of these three operations, which we'll combine together into a digital network.

One type of notation, which is essentially block diagram notation, corresponds to representing the operation of delay as a box. The input, of course, is the sequence that we want to delay, let's say, x of n. And the output then is that sequence delayed by one sample, x of n minus 1. And typically in the block diagram type of representation, this element, or block, is denoted with a z to the minus 1.

Well, what does z to the minus 1 have to do with the operation of delay? Simply that if we were to think of the z transform of the input sequence and the z transform of the output sequence, what we know is that those are related, in fact, by multiplication by z to the minus 1. So the implication in putting inside the box the notation z to the minus 1, the implication is that essentially what this block is doing is multiplying the z transform of the input sequence by z to the minus 1.

So that's one element. The second element that we need is the element for multiplication. And

typically, that's denoted simply by a line with an arrow on it.

The input to this branch is x of n. Or it could be, of course, y of n. The output of the branch is that multiplied by a scalar, let's say, a, the scalar a.

And this should be x of n. If we're talking about x of n in, we're talking about x of n out, or a times x of n out. And the branch transmittance then is the coefficient a. So this branch allows for coefficient multiplication.

And finally the third element that we want is an element that will perform addition. And that element is the noted in block diagram notation as a summer with two inputs and a single output. The input, let's say, x1 and x2, then the output would be x1 plus x2 of n or z, depending on how we choose to describe the sequence.

Now in the difference equation, of course, we had to accumulate a number of turns. In other words, there was more than one term that we were adding up. Typically, in either digital hardware or on a digital computer, when we're forming additions, the additions are formed in pairs. In other words, we accumulate a large number of terms by adding two terms together, then adding one to that and adding another one to that, etc. So it's convenient in that sense to think of the addition as always being a two input one output addition. Of course, if we accumulate three terms, then it would require two adders to do that.

So these then are the basic elements that correspond to a block diagram representation of a linear constant coefficient difference equation, or equivalently correspond to a digital network. And for example, if we had a first order difference equation, let's say the equation y of n is equal to a times y of minus 1 plus x of n, we can represent that equation graphically in terms of these elements as I've indicated here.

So what do we have? Well, we have as an input x of n, as an output y of n. We need in implementing this difference equation to obtain y of n minus 1. And that's obtained with the delay element.

y of n minus 1 is then multiplied by the coefficient a. So here, we have a times y of n minus 1. And then the difference equation states that y of n is the sum of that with the input. And so that goes into the adder. And out comes y of n.

So this then corresponds to a block diagram representation of the difference equation, or equivalently, a digital network in block diagram notation. This notation, in fact, is a very

common type of notation for graphically representing a difference equation, or digital network. There is an alternative notation, which has a number of important advantages, which is very much like the block diagram with just a few minor changes, which turn out in some instances to be convenient. And that is the graphical representation of the difference equation, or the digital network, in terms of signal flow graphs.

Well let me introduce the notation of signal flow graphs first in general and then more specifically as it applies to your constant coefficient difference equations. A signal flow graph is essentially by definition a network of directed branches, which are connected at nodes. So what that says is that in a signal flow graph we have what will refer to as nodes and directed branches, in other words branches with arrows on them. The nodes will be numbered. And so, for example, we would have a node j and a node k.

Furthermore, each node in the network has a variable associated with it. So that, for example the j node has the variable w sub j. The k node has the variable w sub k. And we have a branch going between the j-th node and the k-th node, which will be referred to as the jk-th branch. So essentially the numbering of the nodes allows us to refer to the nodes and also to refer to the branches, referring to the branches by talking about which nodes the branches go between.

The nodes that I've indicated here correspond to what we'll refer to as network nodes. There are also source nodes. And I've indicated one here.

A source node is a node that has no branches coming into it, only branches leaving it. And the opposite, namely a sink node, which is a node that has no branches going out of it and only branches coming into it. Basically, it's these type of nodes that allow us to get inputs into the network. It's this type of node that we use to represent outputs from the network.

Now, this defines essentially the notation. It doesn't yet specify what the algebraic construction of a signal flow graph is. So in particular, there are some rules that tell us how the node variables are related to inputs and outputs of branches or equivalently the inputs and outputs of other nodes.

A branch, the jk-th branch, which originate at node j, terminates at node k, has as its input, the variable of the node that it originates from. It also has an output. The output we will denote as v sub jk. That's the output of the jk-th branch. And what it is very simply in the most general

sense, the most general context, is some function of the input to the branch. So the output of a branch is some function of the input to the branch, where this function is different, of course, for different branches.

Finally, to get the values of the node variables, the node values are defined to be equal to the sum of the outputs of the branches entering the node. So that if we were to look up here, for example, this node variable would be equal to the output of this branch plus the output of that branch. And that's essentially the algebraic construction of flow graphs.

It's also convenient to separate notationally the output of the branches from an internal node to another internal node, or from a network node to a network node, from the output of those branches that originate at sources and go to network nodes. And the notation that we'll use is sub jk corresponding to the output of the branch that goes from the j-th source-- the sources being numbered differently from the network nodes-- going from the j-th source to the k-th network node.

OK, so what this says, and this equation in particular says, that algebraically the node variables are specified as the k-th node variable being equal to the sum over the internal nodes of the output of the branches from the network nodes to the k-th node plus the output of the branches from the j-th over all the source nodes, from the j-th source node, to the network node that we're focusing on. So this corresponds to the output of the branches from the network nodes. And this corresponds to the output of the branches from the source nodes.

Well, for linear constant coefficient difference equations, we don't need all of the generality that we've implied in the discussion so far. In particular, what we can concentrate on is a special set of signal flow graphs, namely signal flow graphs which are linear. And what we mean by linear is that the output of a branch is equal to the input to the branch multiplied by some function. So that it's a linear relationship between the branch output in the branch input.

In other words v sub jk, the output of the branch from the j-th node to the k-th node, is equal to some function-- this might be simply a scalar or it might be a function of z, for example, which it will often turn out to be-- multiplied by the input to that branch, in other words, the value of the j-th node. So that is then the additional constraint that we impose if we're talking about linear signal flow graphs, which we'll be concentrating on for the rest of the lecture.

Well, to see how a linear constant coefficient difference equation can be represented in terms of linear signal flow graphs, let's take the same example that we worked with previously in

terms of the block diagram notation, in other words, y of n is equal to a times y of n minus 1 plus x of n. Well, we have a node that's a source node. That's x of n. We have a node that's a sink node. That's y of n. That's what we'd like to get out of the network.

And presumably somehow we've gotten a network node whose value will be y of n minus 1. If we have that node, then what we want to form to get y of in is a times y of n minus 1. So we want y of minus 1 multiplied by a, which means that this branch has a transmittance, or a gain, which is a. And then because of the algebraic definition at nodes, the value of this node is the sum of this branch and the output of this branch. So if this branch has a gain of unity, then this node variable will have the value x of n plus a times y the minus 1.

Well, now we simply have to get from here all the way out to there. I've drawn it, the network, somewhat inefficiently. I can get from this node to this node simply with a unity gain. From this node to the output node again, with the unity gain.

And now the only question is-- of course, if this is y of n, and this node has value y of n also-- how do I get from this node to this node if I'm talking about a linear signal flow graph? Well, y of n minus 1, of course, isn't a scalar multiplied by y of n. But y of z, or the z transform rather of y of n minus 1 is actually a scalar or a function multiplied by the z transform of y of n.

So if I were to think of this rather than in terms of the sequences, if I were to think of this in terms of this transform, then in fact the z transform at this node would be z to the minus 1 times y of z. And consequently, thinking of it in terms of z transforms, the gain of this branch then is z to the minus 1.

So this, in fact does, correspond to multiplication by a function of z, as long as in the background what we remember is that we're essentially thinking of the flow graph as representing relationships between the z transform. Now, obviously, we wouldn't feel constrained to designate it that way all the time. In other words, I would feel very comfortable about drawing this flow graph and labeling this input as x of n, labeling this node is y of n, and this is y of n minus 1, and indicating that this branch has a transmittance of z the minus 1. Just simply in the background what we realize is it's not y of n that's multiplied by z to the minus 1. It's y of z that's multiplied by z to the minus 1.

Also, notationally, it's usually convenient, since very often we end up with branches that have unity gain, to choose the convention that if a branch doesn't have the game labeled on it, the implication is that what the gain is, in fact, is unity. So generally, actually we won't specify

explicitly unity gain. If there is a branch with unity gain, we simply won't label the gain.

One final point and that is that you can, I think, see-- you probably observed this for yourself already-- that actually this flow graph is drawn somewhat inefficiently as I've indicated here. Basically, I can take these two nodes and collapse them together. I chose to draw this flow graph this way so that it looks as much like the block diagram representation as possible.

And in fact, if you compare it back to the block diagram representation, it is basically the same except for the fact that we've chosen to represent this delay branch a little differently. And we've used the algebra of signal flow graphs to do the addition at nodes rather than explicitly putting in an adder. And that's really the only difference.

Now, what a flow graph or a block diagram is, as I've stressed, is a graphical representation of that difference equation. And what it in turn corresponds to is a graphical representation of a linear set of equations, a set of equations, which describe the relationships among the various variables or among the nodes in the digital network. If I were, for example, going to implement a difference equation or a single flow graph on a digital computer, what I would implement in effect is not the graphics of the signal flow graph. I would implement the algebra of the signal flow graph. In other words, I would implement the linear set of equations that the flow graph corresponds to.

Consequently, what I'd like to spend the remainder of the lecture on is focusing on the equivalent linear set of equations that the signal flow graph corresponds to, in other words, the matrix representation of a digital network. And the matrix representation is important from a number of points of view. One is, of course, that as I've just stressed it's important for the actual implementation of the digital network. It's also important in a sense that lots of properties of digital networks fall out from the properties of matrices. We won't, in fact, in this set of lectures be exploiting that particularly, but it is one of the reasons why the matrix representation of digital networks is also important to be aware of.

So let's then see what this graphical representation, or the linear single flow graph representation, corresponds to in terms of a set of linear equations, or in terms of a set of matrices. So we're considering then a general network. It has some internal nodes, branches, coming into them. I've indicated one source node and one sink node. More generally, there would be a set of source nodes and a set of sink nodes.

The algebraic equations that the flow graph corresponds to is a statement of how the node variables are related to each other. And just to remind you of how that's generated-- and I'm expressing this now in terms of the z transforms of the node variables-- the z transform of the k-th node variable is the sum over the network nodes of the branch outputs-- that's indicated here-- plus the sum over the source branch outputs.

And incidentally, I've assumed that it's just-- best to assume this because of notational convenience-- that there's a branch from every network node to every other network node. Obviously I can always get away with that. If that branch doesn't really exist, I simply set the gain of that branch equal to zero.

Now, furthermore, we have the relationship that defines what the branch output is in terms of the branch input or in terms of the node variables. So we have specifically the statement that v sub jk, the output of the jk-th branch, is some function of z multiplied by the branch input. And furthermore, the source branch output is some function, or scalar, multiplied by the source node value.

Well, if we substitute these two equations into the previous equation, which defined what the node values are, we end up with a statement that says that the node variables are equal to a linear combination of the node variables, the linear combination specified by the branch transmit answers, plus a linear combination of the source node values. And this then is a set of linear equations that, in essence, defines the digital network. This is a set of equations, of course, because we have an equation like this for each value of k as k runs over all the network nodes, in other words, for k starting from 1 and running up to n.

We can express the set of linear equations in matrix notation. If we think of a column vector corresponding to the node variables, then this equation says that the vector or column matrix of node variables is equal to a matrix of coefficients dictated by the branch transmittances multiplied again by the node variables plus a second matrix multiplied by the column of source node values.

And there is one peculiar thing that happens notationally here, which it's worthwhile to straighten out. It happens because of the way that we chose to write these equations that the branch transmittance values come in to the matrix, as I've indicated here. So the first row, for example, contains F 1 1, F2 through F sub N1. As you're well aware, typically in defining a matrix and matrix elements, the general convention is to refer to the matrix elements with the

first index corresponding to the row, the second index corresponding to the column. So the tendency is to refer to the matrix elements as A1 1, A 1 2 through A 1N, etc.

So that simply means that in writing this set of equations in compact matrix form, it's convenient to do that by referring to the transpose of a matrix of coefficients rather than directly to the matrix of coefficients itself. In other words, it's convenient when I write this set of equations, or a set of equations in compact matrix notation, to write it as I've indicated here. We have a column vector of node values, node variables, equal to a transpose coefficient matrix multiplied by the column vector of node variables. And then the transpose of the transmittances that connect the sources to the internal network nodes multiplied by the source vector, where the matrix F of z, the jk-th element of the matrix F of z is F sub jk of z.

I think that it takes just a few minutes of private reflection to see that this works out notationally. But it's this reason, it's to interface the matrix notation and the flow graph notation that makes it convenient to refer to the transpose, to a transpose matrix here, rather than omitting the transposition.

It's typical and notationally what will also stick with in representing a digital network with flow graph notation to restrict ourselves always to branches that are either simply co-efficient branches-- in other words, they correspond to multiplication by a scalar-- or they're simply delay branches. In other words, they have a transmittance which is z to the minus 1.

And with that restriction, it's convenient for a number of reasons, and in particular for developing some network properties, to consider decomposing this matrix equation in such a way that we separate out with the matrix F transpose of z the part of that matrix that doesn't involve any delay terms and the part of that matrix that does involve delay terms. In particular, we can write the transpose matrix F transposed of z. We can separate it into the sum of two matrices.

One matrix is simply a coefficient matrix. In other words, it involves no terms z to the minus the 1. The second matrix contains only terms involving z to the minus 1. And, of course, we can pull the z to the minus 1 outside. And so we have the delay terms isolated. In other words, we have a matrix of coefficients here, all of the terms multiplied by z to the minus 1.

So the matrix f of z can be separated into a coefficient matrix and a delay matrix. And if we now substitute that back into our original matrix equation, then we have the statement that says that the vector of node variables is a coefficient matrix multiplied by that vector, a delay

matrix multiplied by that vector. And then I'm assuming incidentally in this notation and, of course, it can be generalized quite easily that the sources are connected only with non-delay branches.

I have suppressed incidentally in the discussion up to this point, the additional statement that we need, the additional set of equations that we need, that tell us how finally we get from the set of internal network nodes to the network output. We have then an additional set of equations that specify what the sink node values are in terms of the internal network nodes. And often in referring to the matrix equations for a signal flow graph, sometimes we'll include this equation explicitly, and sometimes we'll tend to suppress it.

Now, looking at this equation where we've separated out the coefficient and delay branches, we can alternatively reinterpret that equation in the time domain. It's expressed here in terms of z transforms. In the time domain, then this corresponds again to a matrix set of equations stating that the column vector w of n is equal to a coefficient matrix multiplied by the present state-- in other words, the same column vector-- plus another matrix multiplied by that set of node variables, one iteration previously, and then, of course, an equation coupling the sources into that and finally an equation that generates the sync node outputs from the node variables.

But if you look at this, what this corresponds to, of course, is a set of first order difference equations, so that essentially what we've done, starting from our n-th order linear constant coefficient difference equation, through the signal flow graph representation, we have in essence decomposed that into a set of first order linear constant coefficient difference equations, of course, all of them coupled. And the discussion in the next several lectures will in fact involve what the various options are in terms of what various forms we can use, or are commonly used, in representing this large n-th order equation. But what we see is that the signal flow graph in essence breaks down the n-th order equation into a set of first order difference equations.

Well, let's just finally look at an example of how the set of equations might look for a single flow graph, a relatively simple example, and incidentally somewhat meaningless in the sense that it's not a flow graph that is motivated by anything other than something to draw here for discussion.

Well, all right, here's a linear signal flow graph. It has these unity gain branches. It has a branch with gain b and a branch with gain a and the two delay branches. According to the

algebra for the flow graph, this node variable is equal to B times this node variable. So that's this equation. This node variable is this one delayed plus this one delayed. That's this equation and this node-- I'm sorry, this node is a times w2 of n plus the source.

So those are the linear equations defining the internal network nodes. And this corresponds then to a matrix equation. Let me draw your attention to the fact, incidentally, that the way that I've numbered these nodes in fact is not the order in which I can compute them if I were to compute them successively.

Now, what I mean by that is suppose that in my numbering what I decide is that what the numbering means is that node 1 I'll compute first, node 2 I'll compute second, node 3 I'll compute third. To compute node 1, I need node 3. And I don't have node 3 if I'm going to compute that after node 1. So actually the order that I've specified here for the nodes is not an order in which I can compute the network. Or another way to say that is that this network as I've numbered the nodes is not computable.

But let's go on, and we'll look in fact at a rearrangement of the nodes that is computable. But let's see how the matrix equation looks.

Simply, this equation rewritten in terms of a set of matrices, then it comes out as I've indicated here. We have the matrix F sub c. This is the coefficient matrix. This is the delay matrix. So it involves a vector of only delayed sequences and then the input coupled in.

Let me point out incidentally-- just keep track of the fact that this matrix, the coefficient matrix, happens to have an element on or above the major diagonal. And that shouldn't have any particular meaning right now, but it will in a couple of minutes.

All right, let's consider a slightly different ordering for the network nodes. This is the same network. I now have ordered the node slightly differently. I'm calling this node 1, this node 2, and this node 3. And let's just inquire as to whether the nodes ordered as I've indicated here are now computable.

Well, to get node 1, I need w of n minus 1, which of course I got because that's from the previous iteration. I need w3 of n minus 1. And that I've got from the previous iteration. So this node can be computed.

To compute this node, I need the source, which I have. And I need w1 of n, which I just got. So this can be computed. And then to compute w3 of n, I need w2 of n, which I just computed.

And so, in fact, the nodes numbered in this order are computable, whereas the nodes numbered in the previous order are not computable.

If I look at the equations for this, they're, of course, the same equations as I had previously, except that some of the indices are changed. And so it simply involves changing some of the rows and columns. And what it comes out to look like, in fact, is what I've indicated here.

Again, the coefficient matrix-- the same elements appear, but the order is swapped around-- multiplied by the vector of current node states and then a delay matrix multiplied by the vector of delayed states, and then the input coupled in. And notice that in this case, we don't have any non-zero values on or above the main diagonal.

It turns out that you can show it, and it's not too difficult to do, you can show that a necessary and sufficient condition for the node numbering to correspond to a computable network, or to be a computer numbering, is that this coefficient matrix has to have the form that there are no non-zero elements on or above the major diagonal.

So often if the nodes are numbered in a non-computable order, the order can simply be rearranged so that it is computable. It does turn out incidentally that there are some networks for which no matter how you number the nodes, you can't number them in an order that makes the network computable.

Computability, as I've just said, is the statement that the notes can be numbered in such a way that it's 0 above or on the main diagonal. As you'll see as you work some problems in the study guide, a non-computable network, in essence, what happens is that we have a set of nodes in the network all of which are connected together by non-delay branches. And so you can see that what happens is to compute this one I need that one. But to compute that when, I need this one. But to compute this one, I need this. But to compute that, I need that. There's no way to get out of that loop.

So if there's a delay free loop in the network, it turns out that that corresponds to a non-computable network. Although it's important to stress that the non-computability doesn't mean that the set of equations can't be solved, the set of equations can still be solved, or the network can be rearranged. What it simply means is that non-computability means that we can't compute the node variables in sequence. We can't start with w1 of n and then go on the w2 of n and then w3 of n, etc.

Well, this issue of non-computability is just one brief glimpse in fact into the kinds of network properties that can be pulled out of either the signal flow graph representation of properties of signal flow graphs or out of the matrix representation. As I indicated at the beginning of the lecture, we won't be exploiting these kinds of properties. There is, in fact, a rather lengthy discussion of properties of digital networks in the text, which we won't be going into further.

In the next several lectures what we'll concentrate on are the more common digital network structures for implementing both infinite impulse response, or recursive digital filters, and finite impulse response, or non-recursive digital filters. Thank you.