MITOCW | watch?v=mUpwOQ0w2vk

ALAN OPPENHEIM: Last time we introduced the notion of digital networks and the general topic of digital network theory. There, of course, are lots of directions that that discussion can proceed in, but during this set of lectures, we won't be talking in any more detail about the general issues of digital network theory. In this lecture and the next lecture, I would like to consider, in particular, some of the more common structures that are used for implementing digital filters-- first for the case of infinite impulse response filters, which we'll discuss in this lecture, and then in the next lecture some of the more common structures for finite impulse response digital filters.

To begin the discussion, let's consider the most general form, once again, for the transfer function of a digital filter where we are assuming that the system function H of z is a rational function in z to the minus 1. We recall from our previous discussions that a rational transfer function of this form corresponds to a linear constant coefficient difference equation, and in particular, the difference equation corresponding to this system function is the difference equation that I've indicated here. The coefficients in the numerator corresponding to the polynomial that represents the 0's of the transfer function are identical to the coefficients applied to the delayed values of the input, and the coefficients in the denominator corresponding to the coefficients for the polynomial representing the poles are the same coefficients, which, in the difference equation, correspond to the weights applied to delayed values of the output.

So this is the general form of a transfer function for assuming that the system is representable by a linear constant coefficient difference equation, and the difference equation corresponding to this transfer function is as I've indicated here. Incidentally, let me stress as I indicated in the last lecture that our assumption will be throughout these discussions that we are discussing-- considering a causal system. In other words, the region of convergence that we would associate with this transfer function is the region of convergence that would correspond to a causal system. In other words, the region of convergence is outside the outermost pole.

There are a variety of ways in which we can rewrite a transfer function of this form. One possible way of writing this transfer function is as I've indicated at the bottom. That is an expression corresponding to the product of two functions-- the first representing the polynomial for the 0's, and the second representing the polynomial for the poles. What this suggests is that we can imagine implementing this system by implementing a system which realizes this transfer function, and cascading that-- the cascade leading to the product of the system functions-- cascading that with a system that implements this system function.

What that corresponds to in the implementation of the difference equation is first implementing in terms of multipliers, delays, and adders as we talked about last time-- implementing first the linear combination of the weighted delayed input values, and then using that as the input to a system which implements the weighted delayed output values. So implementing this system function with this function first in cascade with this corresponds to implementing this difference equation where we can imagine denoting this first sum as x1 of n-- implementing x1 of n-- implementing the function x1 of n, and then using that as the input to a system which is represented by the difference equation y of n equals x1 of n plus this sum.

Carrying that out, the digital network that results is as I've indicated here, where we have, first of all, the linear combination of weighted delayed input values. So here is x of n, x of n minus 1, x of n minus 2, down through x of n minus capital N, where I'm assuming in drawing this that capital M is equal to capital N. This first block then implements this summation to form x1 of n, and then the second system which this has cascaded with has as an input x1 of n and added to it weighted delayed values of the output. So here is y of n, y of n minus 1, down through y of n minus capital N. We see the coefficients a1, a2, through a sub capital N. Clearly in this implementation this block corresponds to implementing the zeros of the system, and this block corresponds to implementing the poles of the system. So as we've factored the transfer function into the 0's followed by the poles, we have this system implementing the 0's followed by this system implementing the poles.

Well, this, of course, is one implementation of the difference equation, but in fact, there are a variety of ways in which we can manipulate the transfer function, or, equivalently, in which we can manipulate the difference equation, which will lead to other structures for implementing the system besides the structure that I've indicated here. Well, let's consider one simple way of manipulating this system to generate another structure.

We recognize this as two systems in cascade. They both implement linear shift invariant systems, and we know that two linear shift invariant systems in cascade can be cascaded in either order without affecting the overall transfer function of the system. So we can imagine just simply breaking the system at this point, interchanging the order in which these two systems are cascaded, and obviously what that leads to is a second implementation of the same difference equation. That implementation in particular, whereas this one has the zeros first followed by the poles-- interchanging the order of those two will result in the poles implemented first, followed by the 0's, and the system that results is what I've indicated on this next view graph.

So this system is identical to the other one. It's clearly identical in terms of the overall transfer function, and what I've done simply is just to interchange the order in which the 0's and the poles are implemented. Well, that manipulation that is breaking that system and interchanging the order in which the systems are cascaded can be interpreted in terms of either a manipulation on the transfer function or a manipulation on the difference equation, and to indicate what that corresponds to let's return to the general difference equation as we had it on the first view graph-- the transfer function, where now, rather than cascading this system first and this system second, we've simply interchanged the order in which those two systems are cascaded. That's interpreting this operation in terms of the transfer function.

To interpret it in terms of the difference equation is slightly more involved, but basically and very quickly what it involves is first implementing the difference equation in which we consider that the input is just x of n rather than a weighted sum of delayed x of n's to implement to implement the output y1 of n. And then since the input is, in fact, a linear combination of weighted delayed inputs, the corresponding output is the same linear combination of weighted delayed outputs. That essentially is derived from using properties of linear shift invariant systems that we talked about and some of the early lectures.

Well, returning to the network that resulted by interchanging the order of these two systems, one of the questions we can ask, of course, is whether there is any advantage to implementing this system rather than implementing the first system that we derived. And in answering that, one thing that we notice about this system is that there are two parallel branches here with corresponding delays. Now, what does that mean? Well, if we consider this output to be y1 of n-- it's the y1 of n that we had defined in the previous slide-- the value appearing here is y1 of n minus 1, but the value appearing here is y1 of n minus 1. The value appearing here is y1 of n minus 2 and appearing here is y1 of n minus 2. And in fact, following this chain down, what we observe is that the output of this delay is exactly the same as the output of this delay.

Well, if that's the case, then in fact there obviously, if we think about an implementation, is no reason to separately store this delayed output and this delayed output since they're the same. In other words, we can collapse these delays together, and the network that results when we do that is the network that I indicate here, where all that I've done in going from the previous network to this one is simply collapse the delays together, taking advantage of the fact that their outputs were identical.

Now, in drawing a network, of course, it doesn't particularly matter whether we conserve z to the minus 1's or equivalently whether we collapse a network when we can take advantage of the fact that the output of two delays is the same, but clearly in terms of implementation of a digital filter either in terms of a program or in terms of special purpose hardware, obviously clearly there's an advantage to reducing the number of delay registers that are required, because you see each z to the minus 1 that appears in the structure requires in the implementation a register to store the value-- in other words, to hold it for the next iteration.

So in this structure, as it's implemented here, we have n delay registers, where again I'm assuming that capital M was equal to capital N. There are N delay registers, whereas in the first network that we generated-- the network corresponding to the 0's first and then the poles-- the there were two M delay registers.

In general, a digital filter structure that has the minimum number of delay registers-- and you can show that the minimum number required is the greater of M or N, or since we're considering M equal to N, the minimum number is N. The structure that has only that minimum number and no more is generally referred to as a canonic structure. So the structure that I've indicated here is a canonic structure. It has the minimum number of delays, but in fact, it's not the only canonic structure. There are a large variety of canonic structures, and in fact, there's a canonic structure that is similar to the first structure that we derived in the sense that it also has-- is implemented with the 0's first, followed by the poles.

Let me remind you again that this system as it's implemented is basically a cascade of the system poles. Those are the a's-- or the polynomial that this implements corresponds to the poles-- followed by an implementation of the 0's, and it's the b's that control the 0's. Well, to generate another canonic structure we can take advantage of a theorem that, in fact, is a very powerful theorem in dealing with filter structures-- the theorem, which is referred to as the transposition theorem.

What the transposition theorem says is that, if we have a network that implements a transfer function, and if we simply reverse the direction of all of the branches in the network and we interchange the input and the output, then the transfer function that results is exactly the same. So it says take the network, reverse the direction of the branches, put the input where the output was, take the output where the input was, and what you find is that the transfer function of the system is exactly the same.

Well, let me illustrate this theorem. We won't incidentally prove the theorem, although in the notes in the text at the end of the chapter there, in fact, is a proof of the transposition theorem, but let me illustrate the transposition theorem-- first with a simple example that makes it appear to be a trivial theorem, and then with another example that suggests that perhaps the theorem is less obvious than it would at first appear.

Well, to illustrate the transposition theorem, let's begin with a simple network-- just a simple first order network, two coefficient branches and a delay branch. The transposition theorem says that we want first of all to reverse the direction of all of the branches. So this branch gets turned around, again, with a gain of unity. This branch gets turned around with a gain of c. This branch gets turned around with a gain of a. The delay branch is turned around. This branch, which has a gain of unity, is turned around. Put the input where the output was, and take the output from where the input was.

So the transpose of this network is the network that I've indicated here, and now of course we can redraw this network by putting the input on the left hand side and taking the output on the right hand side. That is taking the same network and just flipping it over-- flipping it over because we tend to have a convention that the input is coming in from the left and the output is going out at the right. If we do that, just taking this network and simply flipping it over, we have x of n coming in through a unity gain. This delay has now ended up on the left hand side, and you can verify in a straightforward way that these branches are now correct if we just take this network and flip it over. And is it true that the transfer function of this network is identical to the transfer function of this network? Well, you should be able to see by inspection that in fact it is true.

In fact, if you compare this network to this one, what's the only difference? The only difference is that this delay, instead of being here, ended up on the other side of the coefficient multiplier. And obviously since these two in cascade implement a times z to the minus 1, it doesn't matter whether I do the multiplication by a first and then delay or the reverse. So applying the transposition theorem to this simple example, we see that obviously for this example the transposition theorem works.

Well, let's try it on a slightly more complicated example not to verify that it works, but just again to emphasize how the transposition is implemented. Here I have an example in which I have a canonic first order system. This implements one 0 and one pole. Here is the implementation of the pole and the implementation of the 0. This, in fact, is the first order counterpart of the canonic structure that I showed several view graphs ago. And so it's one pole, and that's implemented through this loop, one 0, and that's implemented through this loop. And these, of course, are unity gain since I put no amplitude on them.

And now to apply the transposition theorem to this network, again, we interchange the-- we reverse the direction of all of the arrows, and you can see that I've done that in all of these branches. The delay is likewise reversed. The a is reversed, and the b is reversed. I put the input in where the output was. I take the output out where the input was, and then the transposition theorem says that this first order system implements exactly the same transfer function as this first order system does.

Well, again, we can redraw this by taking again the input at the left hand side, the output at the right hand side that corresponds to just taking this and flipping it over. In fact, I could do that by taking the view graph and just flipping it over, and the result then, just flipping this over, is the system that I've indicated here-- x of n in at the left hand side, y of n out at the right hand side. And now in comparing these two there are some changes that took place. In particular, we notice that the direction of the delayed branch is reversed. Furthermore, whereas this system implemented the pole first followed by the 0, this system implements the 0 first, followed by the pole.

Is this still a canonic structure? Well, of course it's a canonic structure because it only has one delay, and obviously, in fact, transposing a network couldn't possibly affect the number of delays in the network so that, if we begin with a canonic structure and apply the transposition theorem to it, we must end up with a canonic structure also. Well, it shouldn't be obvious-- or, at least it isn't obvious to me by inspection-- that this system and this system have the same transfer function, but in fact you can verify that in a very simple and straightforward way by simply calculating what the transfer functions of these two systems are.

Well, returning then to the general canonic structure that we had, we can generate a second canonic structure by applying the transposition theorem to this structure. That is reversing the directions of all of the arrows, putting the input in here, and taking the output out there, and then to keep our convention of the input in at the left, the output out at the right, flip that over to generate a second canonic structure, which is the transpose of this structure. And if we do that, the two changes to focus on is that the direction of the delayed branches is reversed, and furthermore, the system will implement the 0's first, followed by the poles. In fact, the network that results is what I've indicated here. This is then the transposed version of the structure that I just showed on the last view graph. The 0's are implemented here. The poles are implemented here, and the direction of the delayed branches is reversed, but again, this is a canonic form structure.

So some structures are canonic form, and some aren't. The first one, in fact, that we developed wasn't canonic form in the sense that it had more delays than were absolutely necessary. The last two structures that we've shown are canonic form structures in that they have the minimum number of delays. All of these structures are referred to as direct form structures-- direct form because they are structures that involve as coefficients in them the same coefficients that are present in the difference equation describing the overall system.

Recall again that these were the coefficients in the difference equation which were applied to the delayed values of the input, and these were the coefficients in the difference equation that were applied to delayed values of the output. And this structure and the other structures involving the coefficients in that form are often referred to as direct form structures. Well, these structures are fine for implementing the difference equation, although there are other structures-- actually there are essentially an infinite variety of structures, but there are some other structures that in some situations are better to use than the direct form structures, and two of the more common, which I'd like to introduce now, are the cascade structure and the parallel structure.

The cascade structure is developed basically by factoring the transfer function of the system into a product of second order sections or second order factors. In particular, we have, again, the general form of the transfer function-- H of z as the numerator polynomial for the 0's, a denominator polynomial for the poles. We can factor the numerator polynomial into a product of first order polynomials and the denominator polynomial into a product of first order polynomials. In general, of course, these factors will be complex, and these factors will be complex.

We can combine together the complex conjugate 0 pairs and the complex conjugate pole pairs so that in fact as a general cascade form it's often convenient to think of a factorization of each of these polynomials into second order polynomials rather than first order polynomials.

Carrying that out, we end up with a representation of the transfer function in a form as I've indicated here-- a second order numerator polynomial and a second order denominator polynomial, and, of course, it's the product of these that we use to implement this overall transfer function with some constant multiplier out in front, which is required essentially because we've normalized these polynomials to have a leading coefficient of unity.

Well, first of all, why might we want to do this? There actually are a variety of reasons for perhaps wanting to consider an implementation of the transfer function in terms of a cascade of lower order systems than the general n-th order system. One of the more common reasons, which I'll have more to say about actually at the end of the next lecture, but I'd like to at least allude to it now, is the fact that any time we implement a system on a digital computer or with special purpose hardware we're faced with the problem that these coefficients can't be represented exactly.

If we have, let's say, an 18 bit fixed point register, we're restricted to truncating or rounding these coefficients to 18 bits. If we implement a filter and special purpose hardware, we might want the coefficient registers to be as low as 4, or 5, or 6, or 10 bits. Obviously, the more bits, the more expensive the hardware implementation is. And the statement which I'll make and justify in a little more detail at the end of the next lecture is that, if I implement the poles of the system through a high order polynomial, errors in the coefficients lead to large errors in the pole locations as compared with an implementation of the poles in terms of low order polynomials. That is, basically the sensitivity of pole locations to errors in the coefficients is higher the higher the order of the polynomial.

Consequently, if that indeed is an issue for the filter implementation, then it is better to implement a system as a cascade of lower order systems or lower order polynomials then as one large high order polynomial. So factoring this transfer function into a cascade of second order sections, what this leads to is an implementation of the system as a cascade of second order systems, and we, again, have the choice of implementing each second order section in a variety of ways corresponding to the various direct forms that we've talked about previously.

One implementation, which is a canonic direct form, is the implementation that I indicate here, where this is alpha 1,1, the coefficient alpha 1,1, the coefficient beta 1,1. It's a little hard to see where each of these coefficients goes, but this is a canonic form implementation of a second order section that has two poles and two 0's. So as I've implemented it, I've implemented it with the poles first, followed by the 0's, and then this is one second order piece that's in cascade with the next pole 0 pair, with the next pole 0 pair, et cetera.

So this is a cascade of second order sections, and, of course, I can generate other cascade forms was one possibility by just simply applying the transposition theorem to this cascade, and basically what would result is that each of these delay branches would be reversed in direction, and the 0's would be implemented first, followed by the poles. Generally what's meant though by the cascade structure, and you can see again that there are a variety of cascade structures depending on how you choose to implement the pole 0 pairs-- what is generally meant by the cascade structure or a cascade structure is an implementation of the transfer function is a cascade of second order sections where the second order sections can be implemented in a variety of ways.

Another structure which is like the cascade structure in that it implements the poles in terms of low order sections, but is different in the way effectively that it realizes the zeros is the so-called parallel form structure. And the parallel form structure can be implemented by-- can be derived by expanding the transfer function of the system in terms of a partial fraction expansion. That is, we can expand this transfer function in terms of-- and let's assume first of all the capital M is less than capital N. If capital M were less than capital N, we can expand this simply as a sum of residues together with first order poles, or in general since the poles are complex, we can imagine factoring this in terms of first order terms corresponding to the real poles, and then second order terms where we combine together first order terms which are complex conjugates so that we have second order terms of this form. If capital M is less than capital N, those are the only two kinds of terms that would result in a partial fraction expansion. If capital M is greater than or equal to capital N, then we'll have additional terms corresponding simply to weighted powers of z to the minus 1.

Now, as in the cascade form, generally the parallel form structure is considered to be one where even if we have real poles we combine two of the real poles together to implement the system in terms of second order sections. If we do that, then the parallel form expansion for the transfer function is what I've indicated here, where combining two terms of this form together, or, equivalently, looking at expansions of this form, we have a numerator polynomial implementing a single 0 and a denominator polynomial implementing two poles. And then depending on whether M is greater than capital N or not, there might be additional terms involving simple weighted values of z to the minus-- weighted powers of z to the minus 1.

Let me stress that there are some differences between this and the cascade form obviously.

One of the differences is that the sections used for implementing the filter consist of one 0 plus two poles, and then the output is formed not as a cascade of sections of that type, but as a sum of the outputs of sections of that type since H of z here is expressed as a sum of second order sections, whereas in the cascade form it is expressed as a product of second order sections.

Well, the general filter structure that results I've indicated here for the case in which we have three second order sections, and again, I'm assuming that capital M is equal to capital N so that we have one branch, which is just simply a coefficient branch. If capital M was one more than capital N, we would have in addition to that one delay branch. And then we have second order sections as I've indicated here, but the second order sections implement only a single 0 and a pair of poles-- a single 0 and a pair of poles.

Now, why might you want to use a parallel form implementation instead of a cascade form implementation? Well, there actually are several reasons. One of the most common, in fact, though, is that sometimes in applying filter design techniques the filter design parameters are automatically generated in a parallel form. That is, rather than being generated either as a ratio of polynomials or as poles and 0's, it might be generated in terms of residues and poles. In that case, of course, it's very straightforward to go to a parallel form implementation rather than a cascade form implementation. Basically the difference between the two forms is that the cascade form implements in terms of low order sections poles and 0's of the system, whereas the parallel form implementation is effectively an implementation of the system in terms of the poles. The poles are controlled in the same way as they were in the cascade structure, but poles and residues rather than poles and 0's.

Again, with the parallel form structure, we can generate other parallel form structures by considering other ways of implementing the second order sections. One possibility is to apply the transposition theorem, and in fact, there are other implementations of second order sections, and so we can talk about a variety of parallel form structures, but generally when we refer to a parallel form or the parallel form structures, we generally mean a parallel realization of poles in terms of second order sections and a weighting applied that correspond to the residues.

Now, the topic of digital filter structures is, in fact, a very complicated topic. There are a lot of other filter structures which can be used for implementing recursive filters, or non-recursive filters, or a finite impulse response, or infinite impulse response. There are structures referred to as continued fraction structures. There are structures referred to as interpolation structures. There are structures referred to as lattice structures, and ladder structures, et cetera. There are a large variety of structures, and in fact, one of the very important issues currently is the design and development of structures and the comparison of structures in particular focusing on the trade-offs-- that is, the advantages and disadvantages-- between various structures.

The structures that I've introduced here-- that is, the direct form, the canonic form, cascade, and parallel-- are the most common structures which are used. They, in fact, tend to hold up very well and seem to be for a variety of reasons some of the more advantageous structures. While I've presented this discussion from a general point of view and tended to focus on infinite impulse response transfer functions, obviously these structures can be applied to finite impulse response or infinite impulse response systems. In the next lecture, I'd like to continue the discussion of structures by focusing specifically on finite impulse response systems and directing our attention to some specific structures that apply only to find an impulse response and take advantage of some particular aspects of finite impulse response systems. Thank you.