

Course material

Course:

Digital Signal Processing II

Video:

2.2.5.a Implementation of digital filters

Concepts (extracted from automatically generated subtitles):

Memory cell. New input sample. First use. Output sample. C code implementation of the leaky integrator. Moving average. Real-time. Digital signal processing. Delay operator. First call. Next slide. Constant value. Second order section. Capital m.. Output of the second delay.



[to video sequence search](#)
(within Digital Signal Processing II.)



[to video](#)

Center for Digital Education. More educational support material here:

<https://exoset.epfl.ch/booc/>

Overview:

- ▶ Algorithms for CCDE's
- ▶ Block diagram
- ▶ Real-time processing

In spite of the sometimes abstract mathematical derivations that we've carried out so far, we should remember that digital signal processing is a very practical discipline and we want to be able to implement everything that we have derived so far in very concrete terms on a computing platform. So today we will look at some algorithms for implementation of constant coefficient difference equations, we will study some block diagrams and we will look at the caveats when performing real-time processing. So here's an old friend in a slightly different form, this is a C code implementation of the leaky

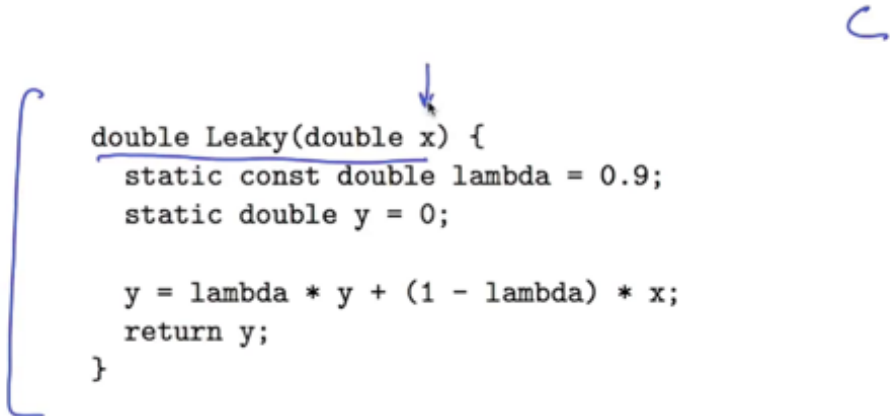
notes

summary

0m 0s



An old friend



```
double Leaky(double x) {  
    static const double lambda = 0.9;  
    static double y = 0;  
  
    y = lambda * y + (1 - lambda) * x;  
    return y;  
}
```

integrator and the way we implement the leaky integrator is by defining a function that we call each time a new input sample arrives that produces an output sample. So here's a function that takes one argument, a double x, and inside the function we have some initialization and then we have the computation part.

notes

summary

0m 32s



An old friend

↓

```
double Leaky(double x) {  
    static const double lambda = 0.9;  
    static double y = 0;  
  
    y = lambda * y + (1 - lambda) * x;  
    return y;  
}
```

Now when it comes to the initialization we first define the value for lambda, this is a constant value, and then we define a local variable y that we will

notes

summary

0m 52s



An old friend

↓

```
double Leaky(double x) {  
    static const double lambda = 0.9;  
    static double y = 0;  
  
    y = lambda * y + (1 - lambda) * x;  
    return y;  
}
```

use to hold the previous value of the output. Remember the CCDE for the leaky integrator is simply $y[n] = \lambda y[n-1] + (1 - \lambda)x[n]$

notes

summary

1m 3s



An old friend

↓

```
double Leaky(double x) {  
    static const double lambda = 0.9;  
    static double y = 0;  
  
    y = lambda * y + (1 - lambda) * x;  
    return y;  
}
```

$$y[n] = \lambda y[n-1] + (1-\lambda) x[n]$$

lambda x of n. So we need to remember the previous value of the output when we

notes

summary

1m 13s



An old friend

C

```
double Leaky(double x) {  
    static const double lambda = 0.9;  
    static double y = 0;  
    y = lambda * y + (1 - lambda) * x;  
    return y;  
}
```

$$y[n] = \lambda y[n-1] + (1-\lambda) x[n]$$

call the function. However we also need to first initialize this memory cell because the first time we will call the function we will assume that the filter

notes

summary

1m 25s



An old friend

C

```
double Leaky(double x) {  
    static const double lambda = 0.9;  
    static double y = 0; ✓  
    y = lambda * y + (1 - lambda) * x;  
    return y;  
}
```

$$y[n] = \lambda y[n-1] + (1-\lambda) x[n]$$

was at rest and therefore all the output was zero from the first call to minus infinity. This is a very important point when we deal with practical implementation. And then the computation part is just a straightforward

notes

summary

1m 32s



An old friend

↓

```
double Leaky(double x) {  
    static const double lambda = 0.9;  
    static double y = 0; ✓  
    y = lambda * y + (1 - lambda) * x;  
    return y; ✓  
}
```

$$y[n] = \lambda y[n-1] + (1-\lambda) x[n]$$

translation in C code of the CCDE. We multiply the output by lambda and we sum the scaled input and then we store the output value already in the memory

notes

summary

1m 48s



An old friend

↓

```
double Leaky(double x) {  
    static const double lambda = 0.9;  
    static double y = 0; ✓  
    y = lambda * y + (1 - lambda) * x;  
    return y;  
}
```

$$y[n] = \lambda y[n-1] + (1-\lambda) x[n]$$

cell before we return and we return the value there. We can test whether this works or not, we write a little main function that calls the function

notes

summary

1m 58s



Testing the code

```
int main() {  
    int n;  
    for (n = 0; n < 20; n++)  
        printf("%.4f ", Leaky(n==0 ? 1.0 : 0.0));  
}
```

$\delta[n]$

```
0.0500 0.0475 0.0451 0.0429 0.0407 0.0387 0.0368 0.0349 0.0332 0.0315  
0.0299 0.0284 0.0270 0.0257 0.0244 0.0232 0.0220 0.0209 0.0199 0.0189
```

repeatedly and as the input passes the delta function. This is a cheap way to write a delta function when n is equal to zero we pass one to the function otherwise we pass zero. And when we run this program we get this output here

notes

summary

2m 6s



Testing the code

```
int main() {  
    int n;  
    for (n = 0; n < 20; n++)  
        printf("%.4f ", Leaky(n==0 ? 1.0 : 0.0));  
}
```

$\delta[n]$

0.0500 0.0475 0.0451 0.0429 0.0407 0.0387 0.0368 0.0349 0.0332 0.0315
0.0299 0.0284 0.0270 0.0257 0.0244 0.0232 0.0220 0.0209 0.0199 0.0189



which is really the exponentially decaying impulse response that we expect. The key points that we learn from implementing the Leaky Integrator in

notes

summary

2m 18s



Key points

- ▶ we need a "memory cell" to store previous output ✓
- ▶ we need to initialize the storage before first use ✓
- ▶ we need 2 multiplications and one addition per output sample ✓

practice is that one we need a memory cell to store the previous output. We also need to initialize this memory cell before we use the filter for the first time and in terms of computation we see that we need two multiplications and one addition per output sample. Now we implemented this in C but the language

notes

summary

2m 32s



Key points

- ▶ we need a "memory cell" to store previous output ✓
- ▶ we need to initialize the storage before first use ✓
- ▶ we need 2 multiplications and one addition per output sample ✓

is really irrelevant when it comes to implementing a digital filter and this

notes

summary

2m 48s



Some like it OOP

```
class Leaky:
    def __init__(self, lmb):
        self.lmb=lmb
        self.y=0

    def compute(self, x):
        res = []
        for v in x:
            self.y = self.lmb * self.y + (1 - self.lmb) * v
            res.append(self.y)
        return res
```

is really the power of the DSP paradigm. So here is the same guy implemented in Python. Python is an object-oriented programming language so we can do

notes

summary

2m 49s



Some like it OOP

```
class Leaky:  
    def __init__(self, lmb):  
        self.lmb=lmb  
        self.y=0  
  
    def compute(self, x):  
        res = []  
        for v in x:  
            self.y = self.lmb * self.y + (1 - self.lmb) * v  
            res.append(self.y)  
        return res
```

something a little bit more elegant than what we did before. We can define a class that we can parameterize with different values of lambda and therefore we can instantiate different Leaky Integrators with different values of the leaking factor. So here the initialization procedure gets its own lines of code and what we do is we initialize lambda to a value that we pass and we initialize the memory cell that we need to have as well. And then finally we can compute the output of the filter since we're in Python it's very

notes

summary

3m 1s



Some like it OOP

```
class Leaky:  
    def __init__(self, lmb):  
        self.lmb=lmb  
        self.y=0  
  
    def compute(self, x):  
        res = []  
        for v in x:  
            self.y = self.lmb * self.y + (1 - self.lmb) * v  
            res.append(self.y)  
        return res
```

easy to pass an array here rather than a single value and so we iterate over the

notes

summary

3m 36s



Some like it OOP

```
class Leaky:  
    def __init__(self, lmb):  
        self.lmb=lmb  
        self.y=0  
  
    def compute(self, x):  
        res = []  
        for v in x:  
            self.y = self.lmb * self.y + (1 - self.lmb) * v  
            res.append(self.y)  
        return res
```

array and compute the output for each member of the array. And what we produce in the end is another array that contains the output values. So again we can test this we instantiate a Leaky Integrator with a value of 0.95 for

notes

summary

3m 40s



Testing the code

$\delta(n)$

```
>>> from leaky import Leaky
>>> L=Leaky(0.95)
>>> print L.compute([0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0])
[0.0, 0.0, 0.0, 0.0, 0.0500000000000000, 0.0475000000000000,
 0.0451250000000000, 0.0428687500000000, 0.0407253125000000,
 0.038689046875000, 0.0367545945312500]
>>>
```

lambda and then we compute the Leaky Integrator for a signal which is again the delta signal and to show that we can place our origin wherever we want we

notes

summary

3m 55s



Testing the code

```
>>> from leaky import Leaky
>>> L=Leaky(0.95)
>>> print L.compute([0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0])
[0.0, 0.0, 0.0, 0.0, 0.05000000000000000, 0.04750000000000000,
 0.04512500000000000, 0.04286875000000000, 0.04072531250000000,
 0.038689046875000, 0.0367545945312500]
>>>
```



actually provide a few zeros before the time n equal to 0. And the output gives us exactly what we expect some zero points before we hit 1 and then the

notes

summary

4m 10s



```
double MA(double x) {
    static const int M = 10;
    static double z[M];
    static int ix = -1;

    int n;
    double avg = 0;
```

```
    if (ix == -1) {
        for (n = 0; n < M; n++)
            z[n] = 0;
        ix = 0;
    }

    z[ix] = x;
    ix = (ix + 1) % M;

    for (n = 0; n < M; n++)
        avg += z[n];
    return avg / M;
}
```

exponentially decaying impulse response that we expect. We can try to do the same with another old friend the moving average whose the CD is this one and if we try to implement the moving average in C using the same paradigm as before we define a function that we call each time a new input sample arrives and then we do some initialization here. So we have to decide on the length of the moving average this will be a constant m . Once we have decided on the length we have to allocate an array of memory cells that contains the input samples that we're going to use to compute the average and we also have to define an index on this array of memory cells. We initialize this index at minus 1 this is a trick that will be clear in a second. This is standard C bookkeeping and these lines here are still part of the initialization procedure. So remember ix

notes

summary

4m 25s



```
double MA(double x) {
    static const int M = 10;
    static double z[M];
    static int ix = -1;

    int n;
    double avg = 0;
```

```
    if (ix == -1) {
        for (n = 0; n < M; n++)
            z[n] = 0;
        ix = 0;
    }

    z[ix] = x;
    ix = (ix + 1) % M;

    for (n = 0; n < M; n++)
        avg += z[n];
    return avg / M;
}
```

is initialized to minus 1 so the first time and the first time only that we call the function we will enter this part of the code and this part of the code simply sets the memory cells to 0 for first use after which ix is initialized to 0 and will never be minus 1 again. This part here simply stores the current value into the memory buffer and updates the index of the buffer modulo

notes

summary

5m 17s



```
double MA(double x) {  
    static const int M = 10;  
    static double z[M];  
    static int ix = -1;  
  
    int n;  
    double avg = 0;
```

```
    if (ix == -1) {  
        for (n = 0; n < M; n++)  
            z[n] = 0;  
        ix = 0;  
    }  
  
    z[ix] = x;  
    ix = (ix + 1) % M;  
  
    for (n = 0; n < M; n++)  
        avg += z[n];  
    return avg / M;  
}
```

capital M. We will see how that works in the next slide and finally this part computes the moving average simply by summing all the elements in the array and normalizing by capital M. The modulo M operation we saw before implements

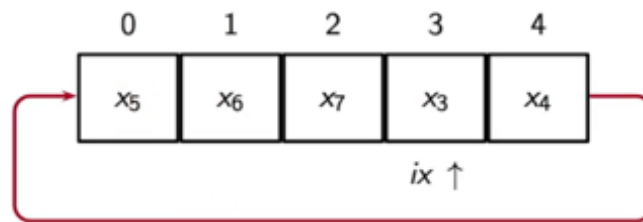
notes

summary

5m 42s



The circular buffer



what is called a circular buffer which is a neat trick used in most DSP applications. Suppose I need to store five values of the input so I define five memory cells like so which I initialize to 0 and then as the sample starts coming in I start putting them into the memory cells. So I use an index ix to keep track of the next free cell for use and so I initialize the thing at 0 and the first sample will go in here. The next step the sample will go on the next cell and I will update the index to the next free cell and so on and so forth. When I get to the last cell and I fill the buffer I update ix modulo N so that it will point back to the first cell of the array and as you can see now my array is full with the first five samples that have come in. At the next step I will still need five samples but I can forget the oldest sample and so I will put x of 5 into the position number 0 and start filling the array once again. The implementation of the moving average teaches us a set of lessons that are

notes

summary

5m 58s



Key points

- ▶ we now need M memory cells to store previous input values ✓
- ▶ we need to initialize the storage before first use
- ▶ we need 1 division and M additions per output sample

very close to what we learned with the leaky integrator. In this case we need M memory cells to implement the filter. The memory requirements are higher for the moving average. We need to initialize the cells once again before we first use

notes

summary

7m 6s



Key points

- ▶ we now need M memory cells to store previous input values ✓
- ▶ we need to initialize the storage before first use ✓
- ▶ we need 1 division and M additions per output sample ✓

them and for each output sample we'll need now M additions and one division. So when M grows you can see that both memory and computational requirements grow as well. As I said the details of the programming language that we choose

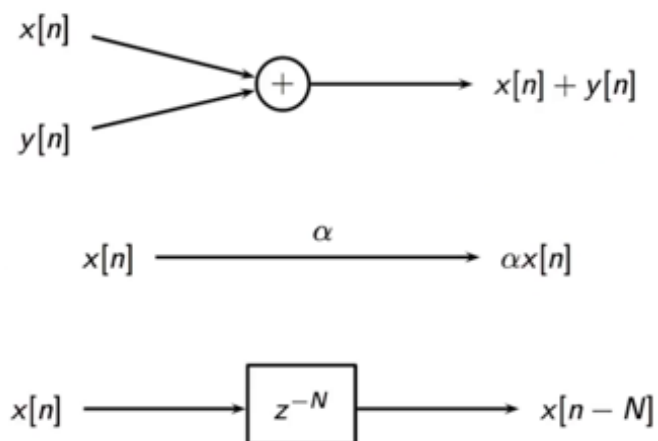
notes

summary

7m 22s



We can abstract from the implementation



to implement a filter are completely irrelevant and we can talk about

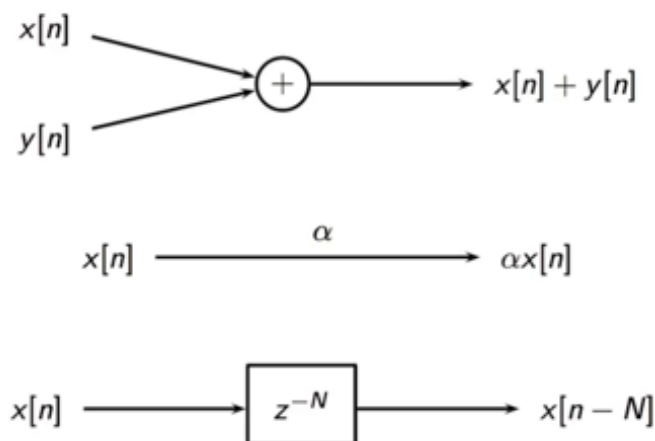
notes

summary

7m 36s



We can abstract from the implementation



algorithms and even derive some optimizations just by taking into account the abstract blocks that are used to implement digital filters and

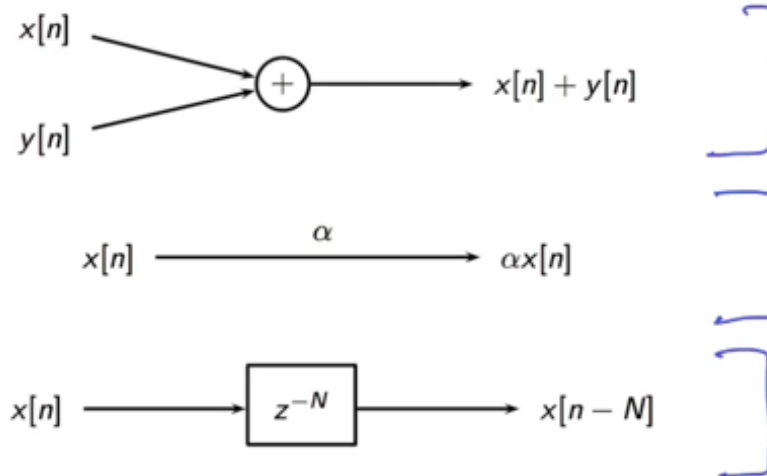
notes

summary

7m 37s



We can abstract from the implementation



remember these blocks are the addition between samples, the multiplication of samples by a scalar and the delay operator. With these three building blocks we've seen this a little bit before in module 2 we can describe any constant coefficient difference equation. So here's for instance the leaky integrator

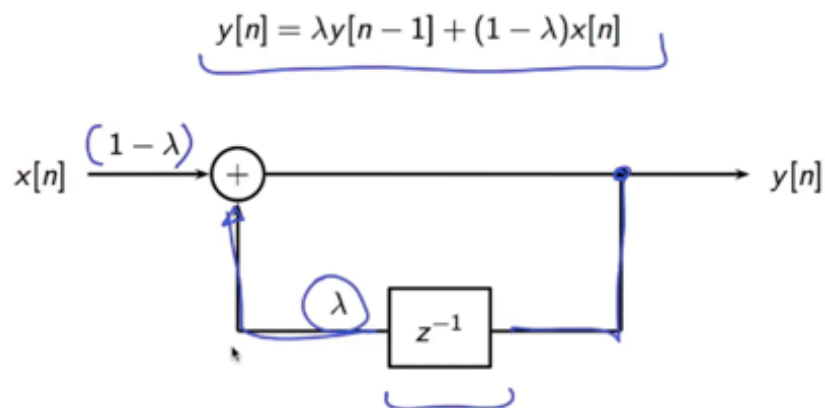
notes

summary

7m 52s



Leaky Integrator



once again and we can draw this block diagram where we can see that the input is being scaled by 1 minus lambda and the output is fed back into the input via a delay and a factor lambda. We can do the same with the moving average so

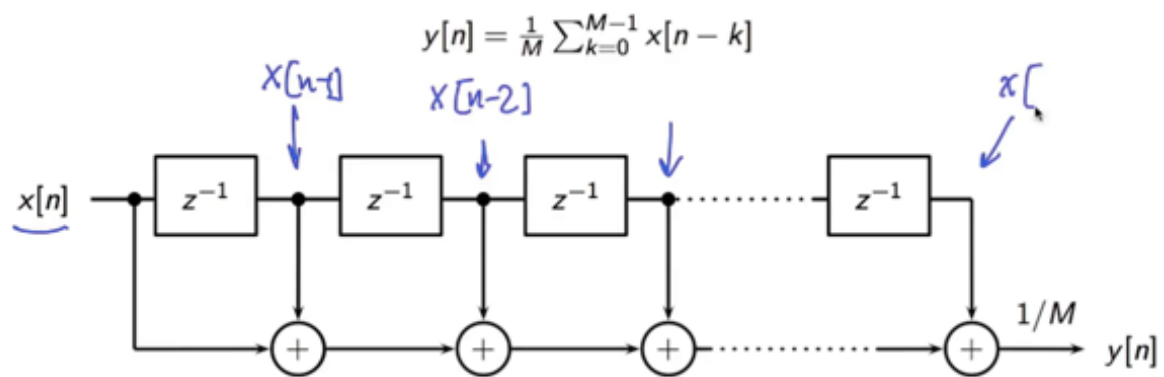
notes

summary

8m 10s



Moving Average



here we have a series of delays that will store past values of the input up to n minus m plus 1. So here you have the input, here you have one delay and so the output of this delay will be x of n minus 1. Here the output of the second delay is x of n minus 2 and so on and so forth until the last delay where you have x of n minus m plus 1. All these delayed samples are summed together and finally

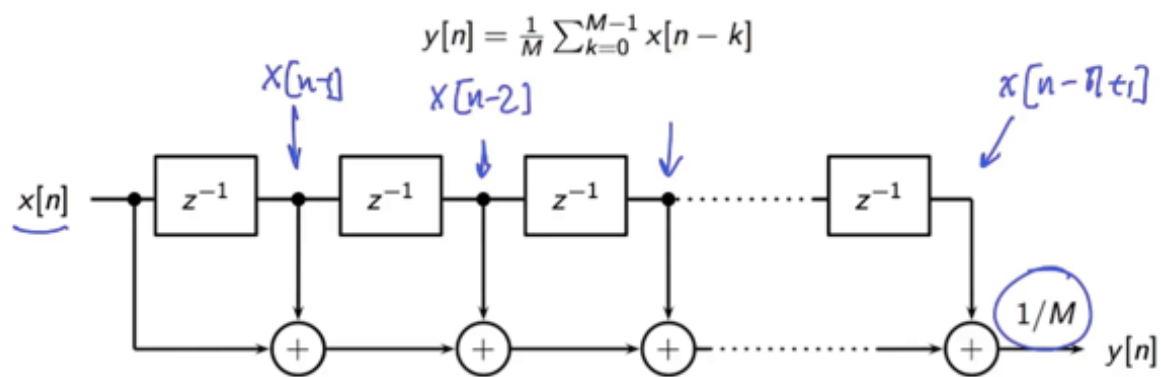
notes

summary

8m 28s



Moving Average



they're scaled by 1 over m. Another interesting structure that you're likely

notes

summary

8m 56s



The second-order section



$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 - a_1 z^{-1} - a_2 z^{-2}} = \frac{B(z)}{A(z)}$$

to encounter a lot in signal processing is the second order section. This is a transfer function where both the numerator and denominator are polynomials

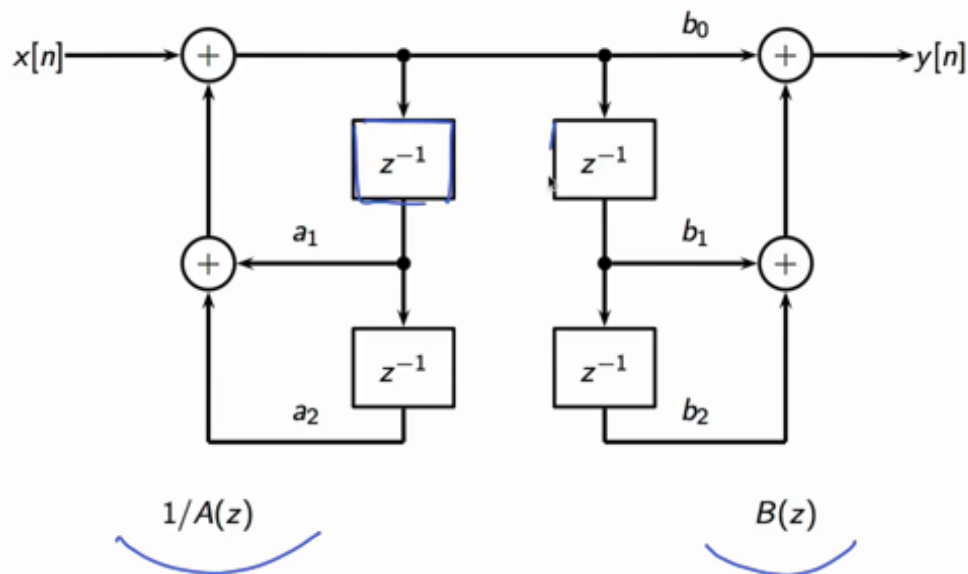
notes

summary

9m 5s



Second-order section, direct form I, inverted order



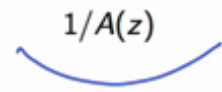
of degree 2 in z to the minus 1. So we'll have two poles and two zeros. We can implement the second order section in a very straightforward manner just by translating into block diagram language the structure of the transfer function. So here you have the numerator of the transfer function, delays that delay the input and scale it by the three coefficients of the numerator and here you have a feedback loop that takes the output, delays it, scales it by the coefficients of the denominator and then feeds it back into the system. So here's the interesting part. Because of the commutativity of the convolution we can actually invert the order of the two sections. We can put the denominator first followed by the numerator. So we first filter by $1/A(z)$ and then filter by $B(z)$. So nothing has changed in terms of the output but now if you look at the structure you see that the contents of this delay cells are going to be exactly

notes

summary

9m 13s

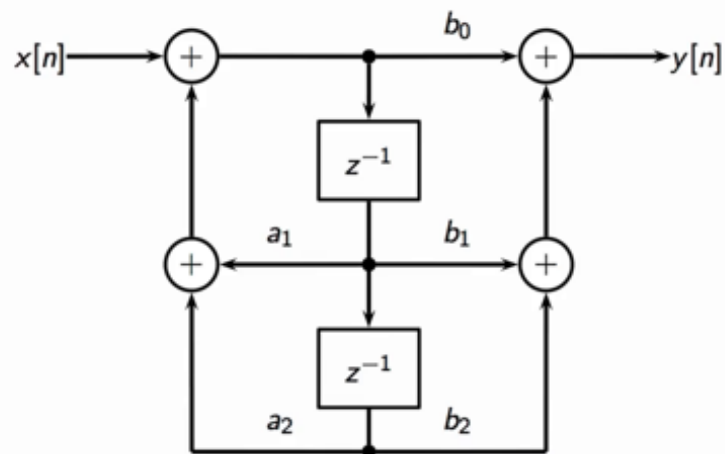




notes

10m 17s





notes

10m 28s



Real-time processing - speed

If input samples arrive every T seconds,

- ▶ each output sample must be computed in at most T seconds
- ▶ number of operations becomes important (IIR vs FIR)
- ▶ some common tricks:
 - circular buffers are size 2^K (mod operation faster)
 - exploit the parallelism some processor operations (fetch and compute)

signal processing. So if the input samples arrive every capital T seconds where capital T can be very very small, for instance in audio capital T is

notes

summary

10m 49s



Real-time processing - speed

$$T = \frac{1}{44000} \text{ sec}$$

If input samples arrive every T seconds,

- ▶ each output sample must be computed in at most T seconds
- ▶ number of operations becomes important (IIR vs FIR)
- ▶ some common tricks:
 - circular buffers are size 2^K (mod operation faster)
 - exploit the parallelism some processor operations (fetch and compute)

typically 1 over 44,000 seconds, then each output sample must be computed in at most capital T seconds. So the number of operations becomes very important and

notes

summary

11m 1s



Real-time processing - speed

$$T = \frac{1}{44000} \text{ sec}$$

If input samples arrive every T seconds,

- ▶ each output sample must be computed in at most T seconds
- ▶ number of operations becomes important (IIR vs FIR)
- ▶ some common tricks:
 - circular buffers are size 2^K (mod operation faster)
 - exploit the parallelism some processor operations (fetch and compute)

your choice of filter will depend on that. For instance using an IIR versus an FIR. Some common tricks in applied signal processing is using circular buffers of size 2 to the power of K for a suitable value of K so that the modular operation is a simple bitmask or you can exploit the parallelism of some processors that allow you to fetch the data and compute a multiplication and addition in parallel in order to speed up the pipeline. Another problem with real systems is accuracy. Processing units will always have a finite

notes

summary

11m 12s



END OF MODULE 5.8

precision even if they use floating point. So overflow and underflow are a real problem. You can use more complex filtering structures to combat some of the accuracy problems and other tricks involved using double precision inside the filtering loop. So for instance if your architecture is 32 bits inside a filtering loop you will use a 64-bit arithmetic. You can split the filter into low order sections or you can use floating point which although slower than the fixed point can help combat some of the overflow issues.

notes

summary

11m 43s

