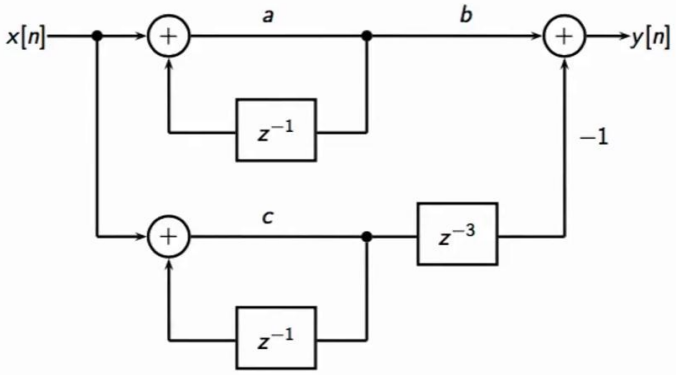
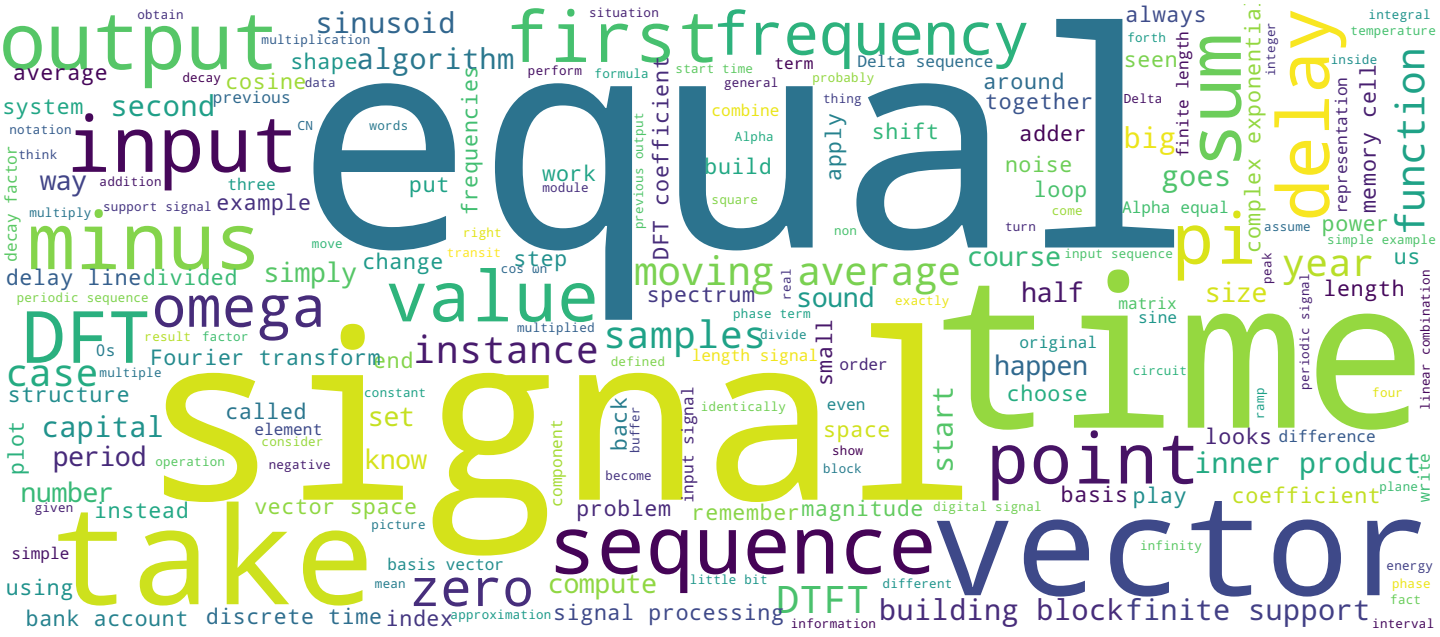
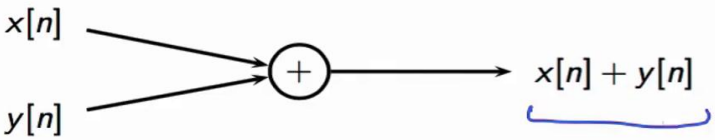
DSP as Meccano





EPFL

Building Blocks: Adder



3

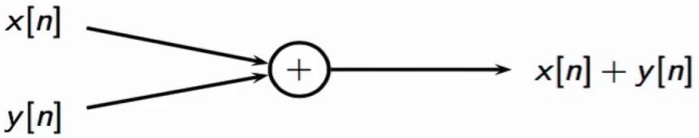
Perhaps some of you are familiar with the Meccano toy. You open the box, you have a bunch of little metal parts, some nuts, some bolts, and then you can use your imagination to create fanciful little contraptions. Digital signal processing is a little bit like that. You have a few fundamental building blocks, and if you are creative, you can combine them to build arbitrarily complex circuitry that performs analysis or synthesis. In this module, we will see what the building blocks are, and we will see how we can combine them to explain some simple phenomena like the accruing of bank interest or to synthesise musical sounds. Let's review the blocks in turn we have seen before. The first one is the adder. It takes two sequences as the inputs and creates a third sequence, which is the sum of the input.

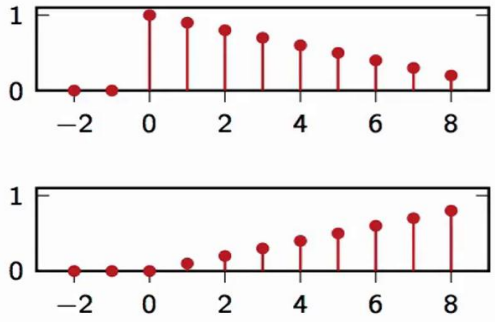
Notes

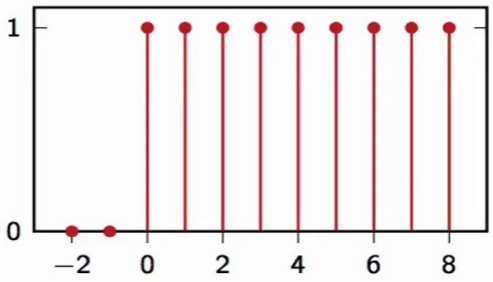
Summary

0m 00s

Building Blocks: Adder







3

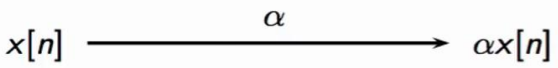
As an example, imagine the first sequence is a decreasing ramp like so, and the second sequence is an increasing ramp like so. If you sum them together, you get a constant sequence. This is actually a general result when the slopes of the ramps are the negative of each other.

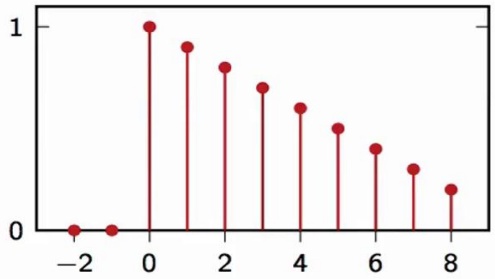
Notes

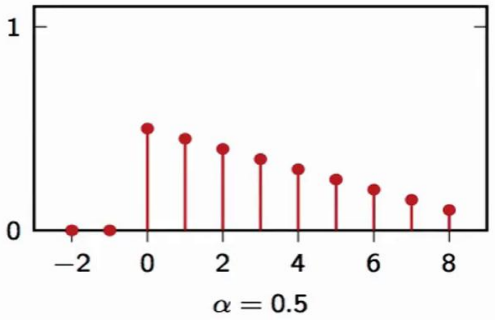
Summary

0m 48s

Building Blocks: Multiplier







4

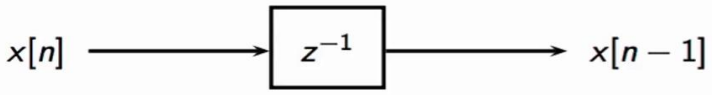
The second building block is the multiplier. It takes one sequence as the input and provides a scaled version of that input as the output. For instance, the same decreasing ramp is shown here, and if the factor is one-half, what you get at the output is the same shape but scaled by one-half.

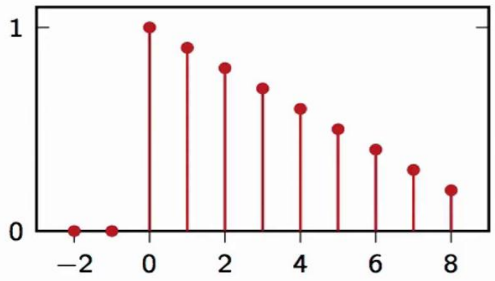
Notes

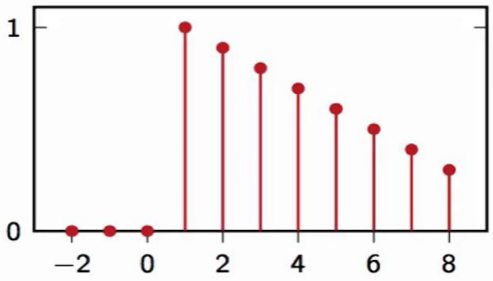
Summary

1m 05s

Building Blocks: Unit Delay







5

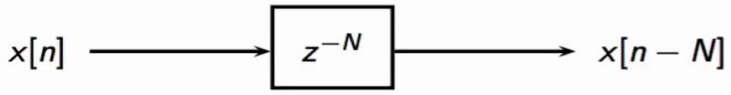
The third building block is the delay. Here, we see the unit delay. We take an input sequence here, and what we get at the output is the same sequence delayed by one sample. The notation z^{-1} inside this block might seem a little obscure, but it will be clear in a few weeks. You have to think of this block as a memory cell, as a buffer, that keeps the current value in storage and returns you instead the previous value. If we apply the delay to our usual decrease in ramp, we get that the delayed by 1 sequence will look like so. Here we are assuming that all the sequences are finite support signals. There are 0s everywhere outside of the shown portion of the signal.

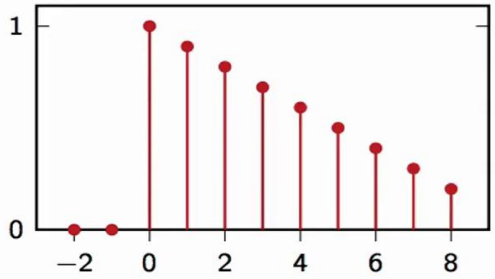
Notes

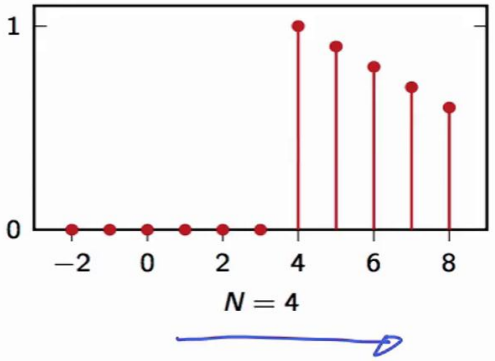
Summary

1m 23s

Building Blocks: Arbitrary Delay







6

A simple generalisation of the delay is the arbitrary delay where we delay the input by N samples. Here, the memory cell is actually N memory cells. You can think of it as a circular buffer where you store the current sample and return the sample that you have gotten N steps before. If we apply this to the usual signal and use a delay of 4, we have a shift to the right by 4 samples. If N is negative, then we would have a shift to the left, and we would say that we are anticipating the signal rather than delaying it. Of course, anticipation implies that you know the future. This is not really the case in general, but if your data is already stored on disk, imagine your sequence is, for instance, a recorded musical sound, then you can probably go backwards or forwards arbitrarily.

Notes

Summary

2m 09s

The 2-point Moving Average

- simple average:

$$m = \frac{a+b}{2}$$

- moving average: take a "local" average

$$y[n] = \frac{x[n]+x[n-1]}{2}$$

7

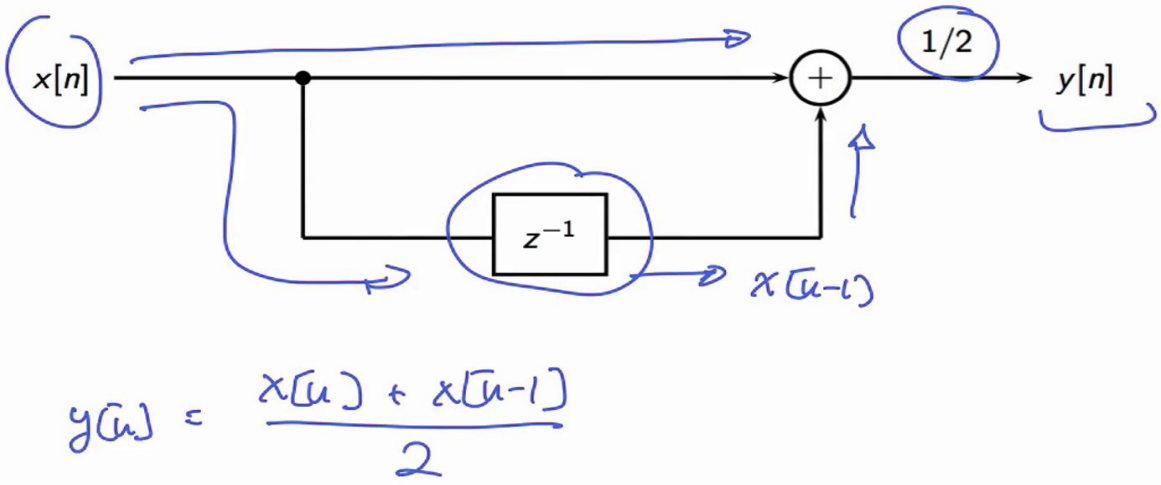
Now, let's see how you can use these building blocks to perform a very simple operation on a sequence, computing the moving average. The simple average of two numbers, we don't need to define it, it's simply the sum of the two numbers divided by 2. A moving average is a local average that you perform every step of the way in a sequence. You take the current sample, you take the previous sample, you sum them together, and you divide them by 2. How would you implement this? Here, we could write some lines of code in your favourite programming languages, but instead, we will use the graphical notation that we have introduced before. We will use these building blocks, these Meccano blocks, so that the structure shows you an abstract implementation of the algorithm that you can then convert in your favourite language. This is a very powerful descriptive paradigm for signal processing that we will use all through the course.

Notes

Summary

3m 01s

The 2-point Moving Average DSP Blocks



$$y[n] = \frac{x[n] + x[n-1]}{2}$$

8

Here's how we would represent the moving average with building blocks. Remember, y[n] is equal to x[n] plus x[n-1] divided by 2. Here, we have x[n]. Here, we have an adder. We need to get x[n-1]. We get that via a delay operator, so outside here, x[n] goes in here. Here, we get x[n-1]. We put it here, x[n] also goes through here. Here is the sum of the two, and here, we multiply by one-half, and here we have the output. This is the representation with building blocks of the moving average. Let's apply this to a signal and see how that works.

Notes

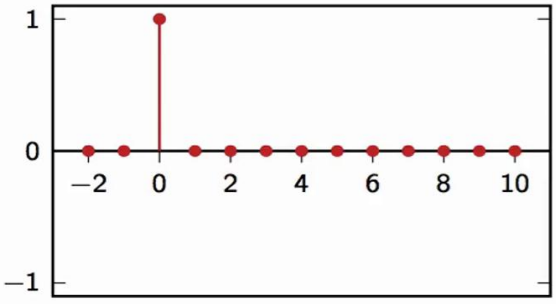
Summary

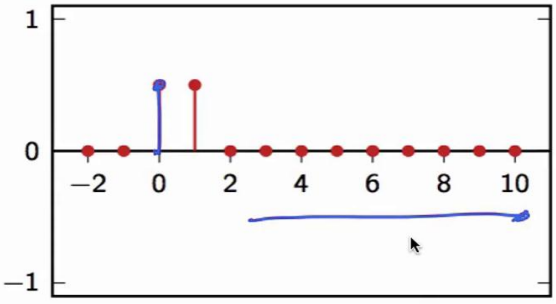
3m 53s

Let's average...

$x[n] = \delta[n]$

$y[1] = \frac{x[1] + x[0]}{2} = \frac{1}{2}$





$y[0] = \frac{x[0] + x[-1]}{2} = \frac{1}{2}$

9

Let's pick the simplest possible discrete-time signal, the Delta sequence, which is 0 everywhere except for n equal to 0, where the value is 1. If we apply the moving average to the signal, nothing really happens before n reaches 0 because for all negative values of n, the input is 0, and the average of two 0s will be 0. When n is equal to 0, then we have that y[0] will be equal to x[0] plus x[-1] divided by 2. x[0], we know that is equal to 1, x[-1] is 0, and so this is equal to one-half, and in 0, the output will be equal to 0.5. In 1, y[1] will be equal to x[1] plus x[1-1], x[0] divided by 2. And again, this is equal to 0, this is equal to 1, and this will be equal to one-half. Then, for all values of n greater than 1, we will have that the two inputs that we use in the average are 0, and so the output will be identically 0 for eternity.

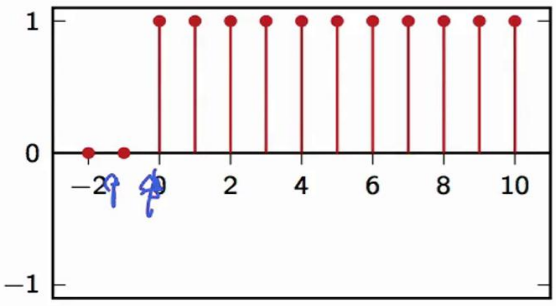
Notes

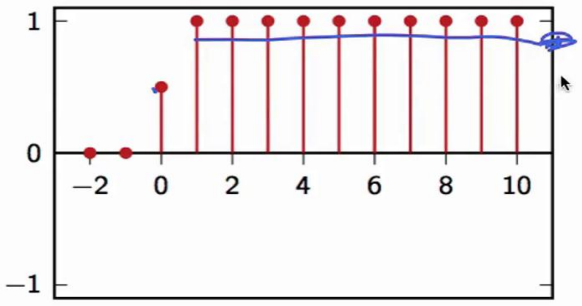
Summary

4m 34s

Let's average...

$x[n] = u[n]$





10

Another example, the unit step. What happens here is that, again, nothing happens for negative values of the index. The first time something moves is when n is equal to 0, at which point the first output sample will be exactly like in the case of the Delta sequence, the average of 0 here and 1 here, and we will get 0.5. Then, for n equal to 1 onwards, we will be averaging two equal values, and so of course, the average will be equal to the value itself, and we will have 1 for all values of the index greater than 1.

Notes

Summary

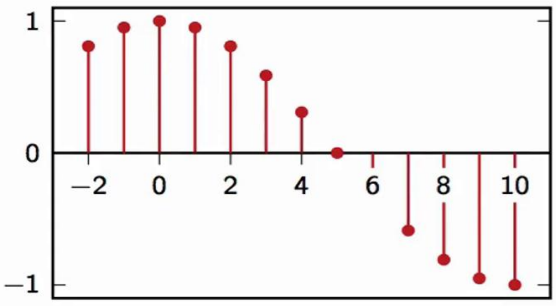
5m 39s

Let's average...

$x[n] = \cos(\omega n), \quad \omega = \pi/10$

$y[n] = \frac{\cos \omega n - \cos \omega(n-1)}{2}$

$= \cos(\omega n + \theta)$



11

Let's now try to compute the moving average of a sinusoid. Here, we have cos(ωn) with ω equal to π/10, but the frequency is immaterial. To see what the moving average of this signal is, we will have to use a little bit of trigonometry. In general, the output will be equal to cos(ωn) minus cos(ω(n-1)) divided by 2. We can use trigonometric formulas to show that this is equal to cos(ωn) plus a phase term Theta. Surprisingly, the output of a moving average applied to a sinusoid is a sinusoid at the same frequency of the input, just a phase term is added to the shape.

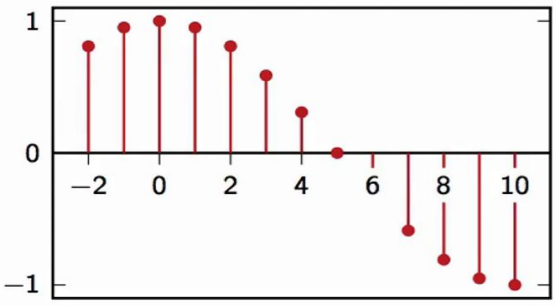
Notes

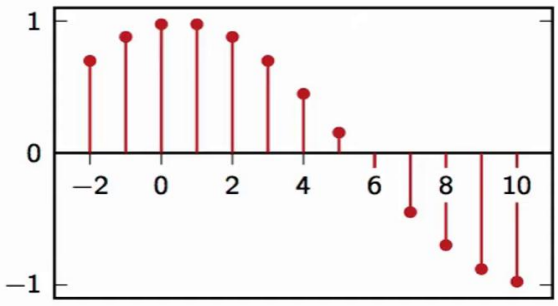
Summary

6m 15s

Let's average...

$x[n] = \cos(\omega n), \quad \omega = \pi/10$





11

This is actually a general result that holds whenever we apply a linear transformation to a sinusoidal input.

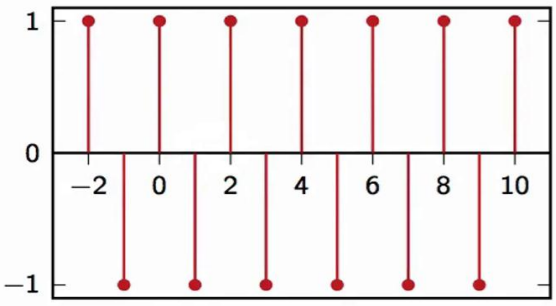
Notes

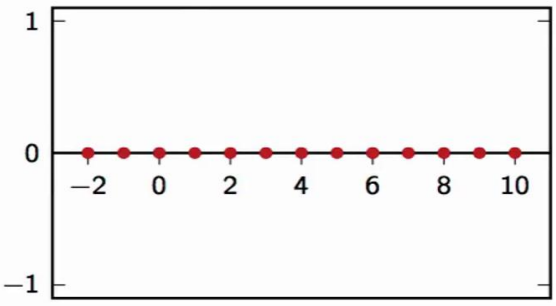
Summary

6m 59s

Let's average...

$x[n] = (-1)^n$





12

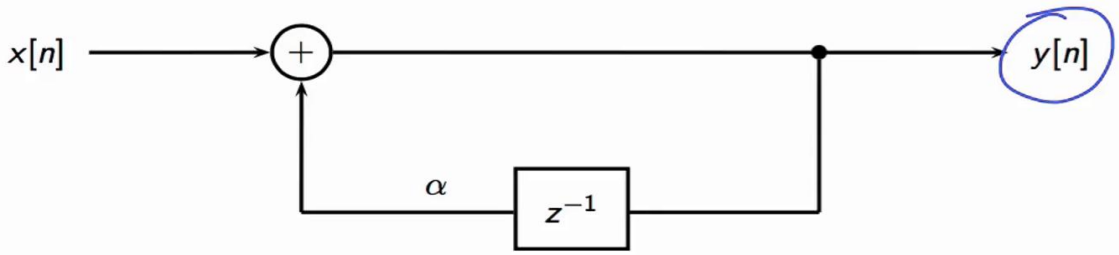
Another sequence could be the alternating sequence where we switch continuously between +1 and -1. At this point, when we average, we're always doing a local average of two values with opposite signs, so their average will be 0, and therefore, the output sequence will be identically 0.

Notes

Summary

7m 05s

What if we reverse the loop?



$$y[n] = x[n] + \alpha y[n-1]$$

$\alpha \in \mathbb{R}$

13

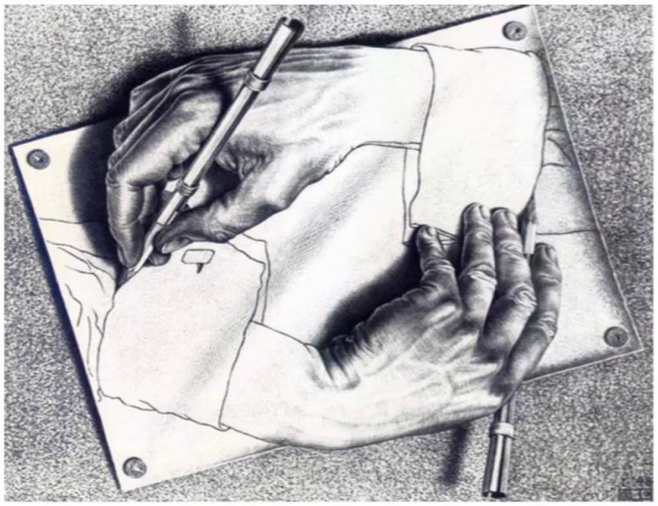
The moving average is a very simple piece of circuitry, but remember, it's like the Meccano toy. We can take these pieces and change the way we combine them. A legitimate question is: what happens if here we reverse the loop? You can see that in the flow of signals in the moving average circuitry, everything goes forward. What if we reverse one of the branches and obtain something like this? In this case, the signal goes in, finds an adder, then goes through the adder, and then goes back via delay to the adder. We have reversed the loop and created a feedback path from the output because the signal here is equal to the output y[n], back to the input section of the circuit. The equation that describes the transformation operated by this circuit is the following. We have that the output at time n is equal to the input at time n, plus Alpha, where Alpha is simply a scaling factor in this case, times the output at the preceding step. We're reusing the previous output rather than using a previous input sample.

Notes

Summary

7m 23s

A powerful concept: recursion



14

Whenever we use previous output samples in our formula, we say that the algorithm is recursive, and perhaps, this picture by Escher captures really the essence of recursion and its fundamental problem. If we need the previous output samples, where do we start?

Notes

Summary

8m 28s

How we solve the chicken-and-egg problem

Zero Initial Conditions

- set a start time (usually $n_0 = 0$)
- assume input and output are zero for *all time* before n_0

15

This is the chicken and egg problem of recursion, and we solve it in signal processing by setting 0 initial conditions. In other words, we set a start time for the computation of our algorithm, we assume that all inputs and outputs are identically 0 for all values of the index before the start time, and we say that at that point, usually, the start time is n equal to 0, all the buffers, all the memory cells in our circuitry are filled with 0s. With that, we can start the process.

Notes

Summary

8m 45s

A simple model for banking

A simple equation to describe compound interest:

- constant interest/borrowing rate of 5% per year
- interest accrues on Dec 31
- deposits/withdrawals during year n: $x[n]$
- balance at year n:

$$y[n] = 1.05\, y[n-1] + x[n]$$

16

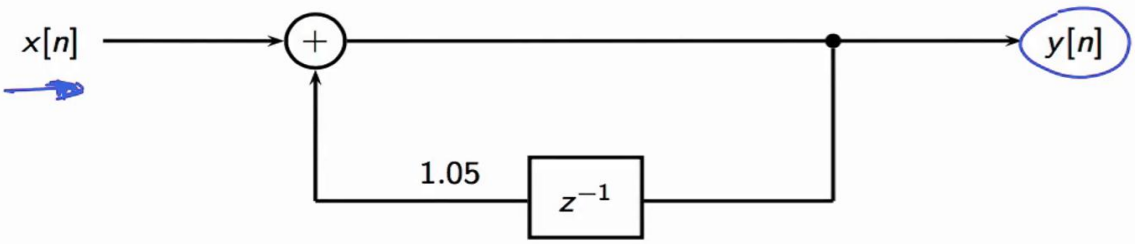
Let's exemplify recursion by explaining how interest accrues in your bank account. It's a very simple model for banking. Let's assume that the interest is constant, and it's 5%, which is rather generous, and it's accrued once per year on December 31st. The deposit and withdrawals during the year are modeled by an input sequence x[n], so x[n] for every year will be positive if you have deposited money in your bank account and negative if you have withdrawn money, and the balance at year n will be given by 1.05 times the capital you had in your bank account in the previous year plus the balance of deposits and withdrawals.

Notes

Summary

9m 17s

Accumulation of interest: first-order recursion



$$y[n] = 1.05\, y[n-1] + x[n]$$

17

If we draw this banking algorithm as a DSP circuit, we have the standard feedback loop with a delay of 1. Here, we have our operations that we perform on our bank account during the year, and the output here is the total sum that we'd find at the end of the year in our bank account.

Notes

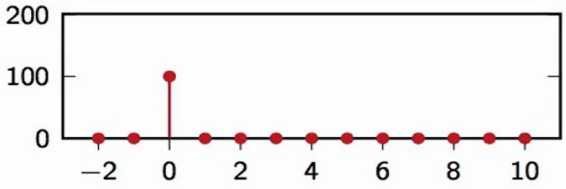
Summary

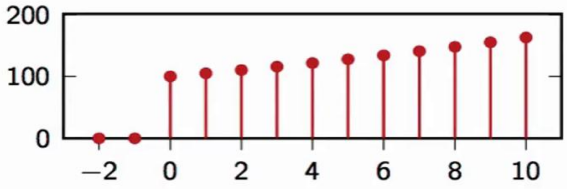
10m 04s

Example: the one-time investment

$x[n] = 100\,\delta[n]$

- $y[0] = 100$
- $y[1] = 105$
- $y[2] = 110.25$, $y[3] = 115.7625$ etc.
- In general: $y[n] = (1.05)^n 100\,u[n]$





18

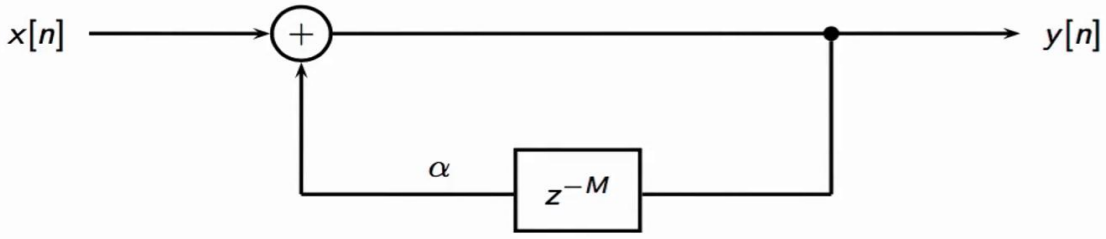
A simple example is the one-time investment. Suppose that the year is 0, you put 100 units of cash in your bank account, then the capital at year 0 is 100. At year one, you will have accumulated 5% interest on the capital at that year. Since we are not touching the capital at the end of the first year, we will have 105 units. At the second year, we will have accrued 5% on the accrued capital, so 5% of 105, and so y[2] will be 110.25, y[3] will be the previous sum times 1.05% and so on and so forth. If you work out the math, you see that the capital, if you don't touch it, will increase exponentially as 100, your initial investment, times 1.05 to the power of n, so it's an exponential growth, of course, the ratio of the exponent is very small, so you will not get rich very quick.

Notes

Summary

10m 23s

An interesting generalization



$$y[n] = \alpha\, y[n - M] + x[n]$$

19

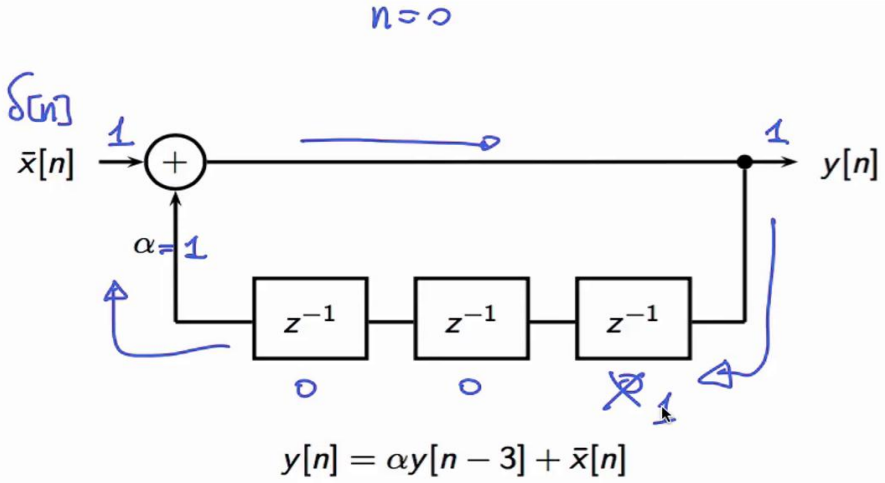
Let's go back to the Meccano board and do a little substitution, replace in the feedback loop the delay, that was a delay by 1 by a delay by M, so that the output will be now Alpha times the output computed M steps before plus the input.

Notes

Summary

11m 28s

Creating loops



$$y[n] = \alpha y[n-3] + \bar{x}[n]$$

20

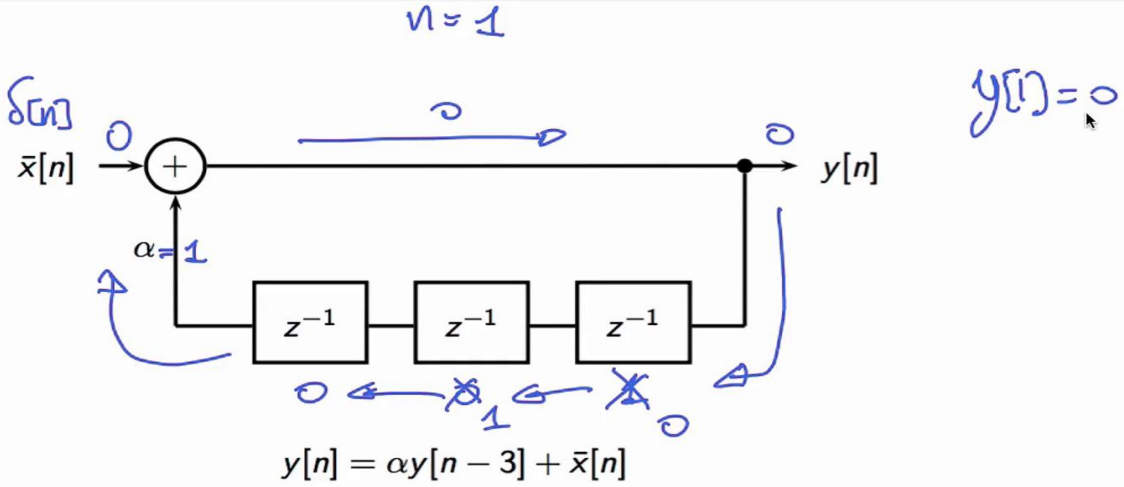
This is really equivalent to replacing the single delay by a cascade of three delay blocks, three memory cells, and we can use this structure now to create loops. Let's see how this works by using a Delta sequence as the input and Alpha equal to 1. The initial conditions are 0, and the delays are filled with 0s. When n is equal to 0, our input will be equal to 1, and so we have that a 1 transits in the system. It will be here, here. It will be at the output here. It will replace the content of the first delay line, and y[0] will be equal to 1. Now we're at n equal to 1.

Notes

Summary

11m 48s

Creating loops



20

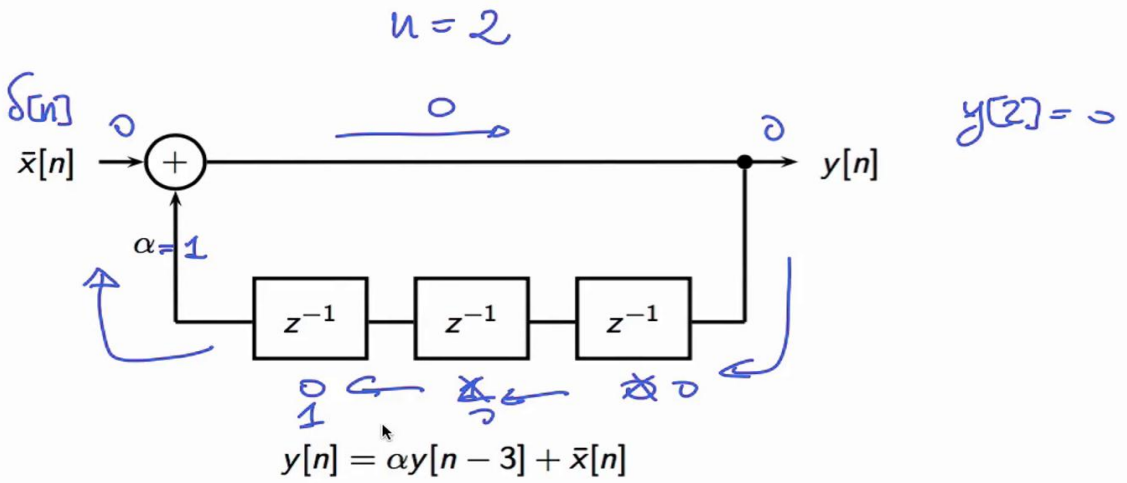
The situation now is like so. For n equal to 1, everything advances. The input will be 0, so it will be 0 here. There will be a 0 here coming from this delay. This will be 0. This 0 will transit into the delay line. This will be replaced by 0. This will advance and put a 1 here. This will advance and put a 0 here. In the end, we have that y[1] is equal to 0, and the delay line situation will be 0, 1, 0.

Notes

Summary

12m 31s

Creating loops



$$y[n] = \alpha y[n-3] + \bar{x}[n]$$

20

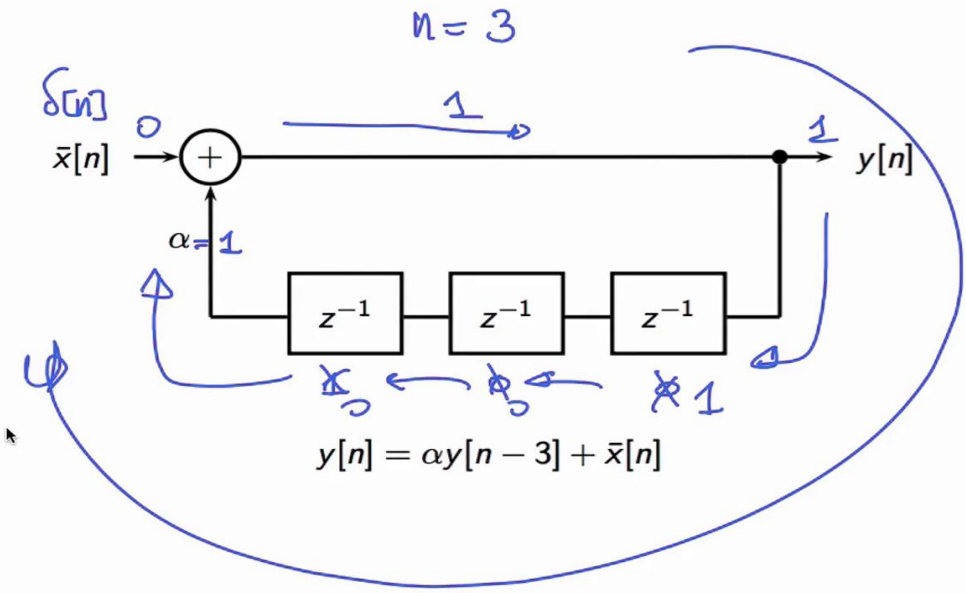
The next step for n equal to 2, we have a 0 here because, of course, x[2] is equal to 0 that will transit everywhere. A 0 will come from this delay. y[2] will be equal to 0, and so this 0 will go in here. This will be 0, 0 and 1 because everything has been shifted by 1.

Notes

Summary

13m 03s

Creating loops



20

The next step, y[3], this will be always 0 from now on, but now the 1 that was in the delay here will transit up, will sum 0 and 1 here, and so the output will be equal to 1. The 1 will also go into the delay here, and everything will be shifted back. In this way, we have created a loop that will periodically put out a 1 every three samples.

Notes

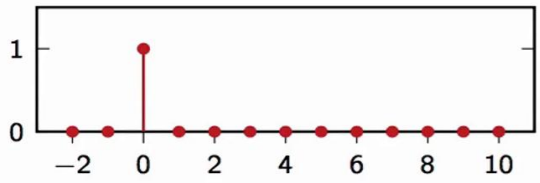
Summary

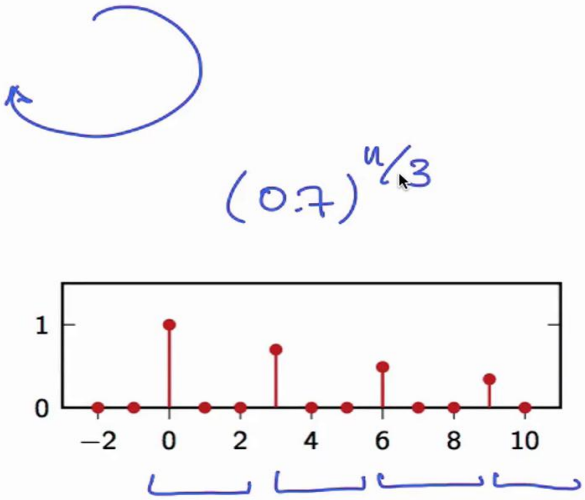
13m 24s

Example

$M = 3, \alpha = 0.7, x[n] = \delta[n]$

- $y[0] = 1, y[1] = 0, y[2] = 0$
- $y[3] = 0.7, y[4] = 0, y[5] = 0$
- $y[6] = 0.7^2, y[7] = 0, y[8] = 0$, etc.





21

We can experiment with a loop generator by changing some of the parameters in the structure. For instance, let's pick Alpha here equal to 0.7. Let's leave the delay at 3, and the input sequence equal to the Delta sequence. The first time we go around the loop, we'll get the same output as in the previous example, so y[0] is equal to 1, y[1] is equal to 0, and y[2] equal to 0. For n equal to 3, the value 1 that has transited through the delay line will pop out, and it will be multiplied by 0.7. So we get y[3] equal to 0.7, and then we get two 0s again. Then when we go around the loop once more, the 0.7 that has transited through the delay line will be multiplied by 0.7 again, and we have 0.7^2. The resultant output is a pseudo-periodic sequence in the sense that we have the same three sample pattern, but every time we go through the loop, the amplitude of the non-0 sample in the pattern is 0.7 to the power of n over 3.

Notes

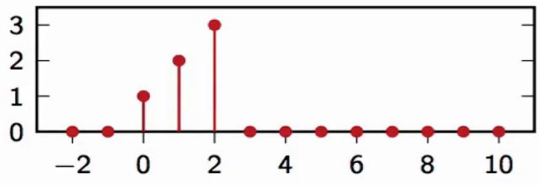
Summary

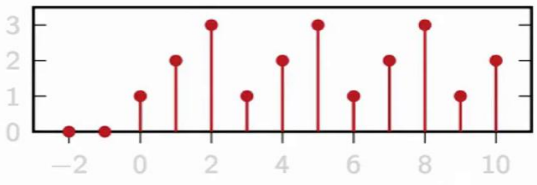
13m 50s

Example

$M = 3,\ \alpha = 1,\ x[n] = \delta[n] + 2\,\delta[n-1] + 3\,\delta[n-2]$

- $y[0] = 1,\ y[1] = 2,\ y[2] = 3$
- $y[3] = 1,\ y[4] = 2,\ y[5] = 3$
- $y[6] = 1,\ y[7] = 2,\ y[8] = 3$, etc.





22

Another way to play with the structure is to keep Alpha equal to 1, so no attenuation, and use an input signal that is a finite support signal whose length is equal to the length of the buffer. I encourage you to do this with pen and paper, and you will see that if you match the length of the input to the length of the buffer, what you get is a truly periodic sequence where the initial pattern gets repeated an infinite number of times, starting at 0.

Notes

Summary

14m 58s

We can make music with that!

- build a recursion loop with a delay of M
- choose a signal $\bar{x}[n]$ that is nonzero only for $0 \leq n < M$
- choose a decay factor
- input $\bar{x}[n]$ to the system
- play the output

23

Interestingly, we can use this simple structure to build an elementary music synthesiser that, in spite of its simplicity, can produce quite interesting simulations of real musical instruments. We will build a recursion loop with a delay of M samples. We will choose an input signal that is finite support and that is non-0, only between 0 and M minus 1. The length of the support coincides with the length of the delay line. We will choose a decay factor, either 1 for no decay or less than 1 to have a decay envelope for the node. We will input this finite support sequence to the system and play the output using the strategy that we have seen in the previous module.

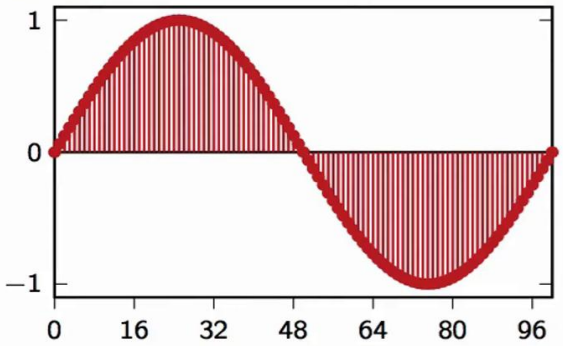
Notes

Summary

15m 27s

Playing a sine wave

$M = 100$, $\alpha = 1$, $\bar{x}[n] = \sin(2\pi\, n/100)$ for $0 \leq n < 100$ and zero elsewhere



$\omega = \frac{2\pi}{100}$

24

Let's start with a very simple example. We will choose M equal to 100 samples, Alpha equal to 1, so there will be no decay, and we will prepare an input signal which is just one full period of a sinusoid. We take a sinusoid at a frequency Omega equal to 2Pi over 100. A full period will complete over 100 samples.

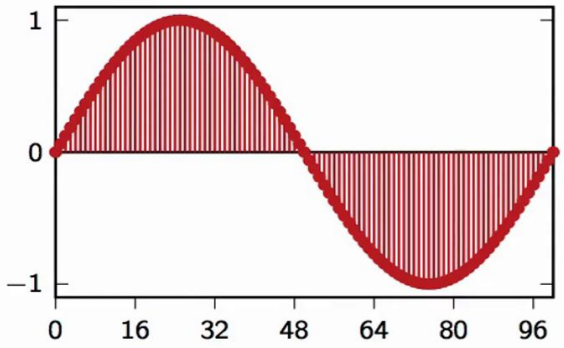
Notes

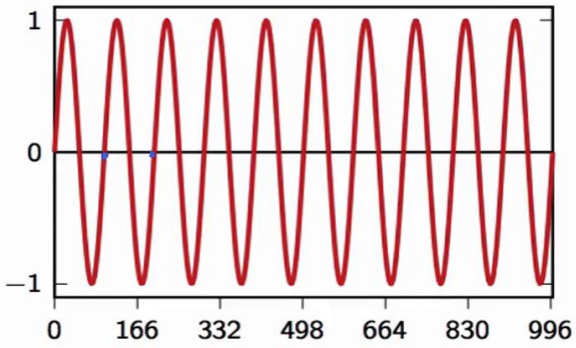
Summary

16m 13s

Playing a sine wave

$M = 100$, $\alpha = 1$, $\bar{x}[n] = \sin(2\pi\, n/100)$ for $0 \leq n < 100$ and zero elsewhere





$F_s = 48\,kHz$ → 480Hz

24

We will input the sequence to the system and we will get the following waveform where you see the periods are joined perfectly every 100 samples. If we play this waveform using a sampling frequency of 48 kilohertz, we will have 48,000 samples per seconds. The pattern repeats every 100 samples and so the resulting pitch that we will perceive will be equal to 480 hertz. This is how it sounds. This is not an extremely interesting sound, so let's try to introduce some realism.

Notes

Summary

16m 37s

Introducing some realism

- M controls frequency (pitch)
- α controls envelope (decay)
- $\bar{x}[n]$ controls color (timbre)

25

Remember, M, the length of the delay buffer, controls the frequency, the pitch of the perceived sound. Alpha controls the envelope, whether we have decay or not, and the initial samples of our finite support signal will control the timbre, the colour of the sound.

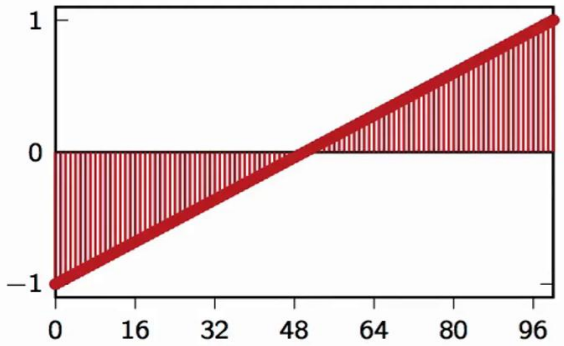
Notes

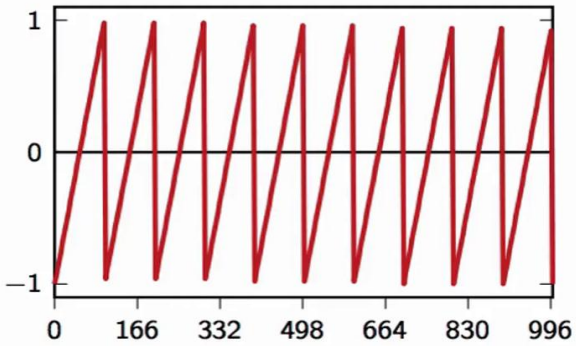
Summary

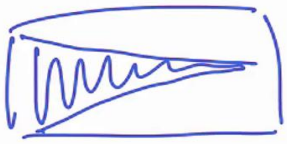
17m 12s

A proto-violin

$M = 100$, $\alpha = 0.95$, $\bar{x}[n]$: zero-mean sawtooth wave between 0 and 99, zero elsewhere







26

Let's try and change the timbre by, instead of using a sine wave, using what is called a sawtooth wave. The sawtooth wave tries to capture the mechanics of a violin. When you drag a bow across the string of a violin, the little teeth in the bow's strings are picking the string, displacing it, and then when the tension gets too big, the string jumps back in the original position. Here we have a situation where we model the displacement and then a sudden return to the original position. Each period will just be a linearly increasing ramp from -1 to 1. We build this 100-sample ramp that goes from -1 to 1. We will pick a decay factor of 0.95. We will put this into the recursion loop, and we'll get a pseudoperiodic signal that looks like so. Because the decaying factor is very close to 1, you're not going to be able to see the envelope in this small window of samples, but if we were to plot the signal on a longer scale, it would look like this. Now if we play the sound, we get a completely different colour. That wasn't really the best violin sound you've ever heard, but this algorithm actually works remarkably well to simulate the sound of plucked strings.

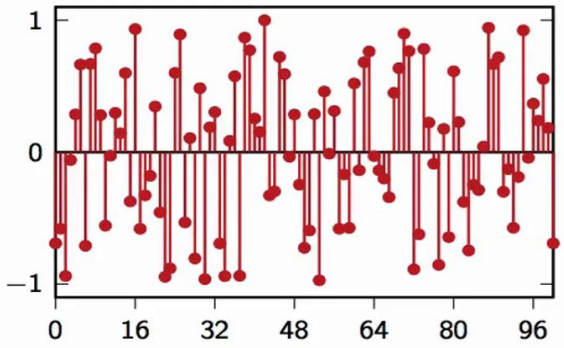
Notes

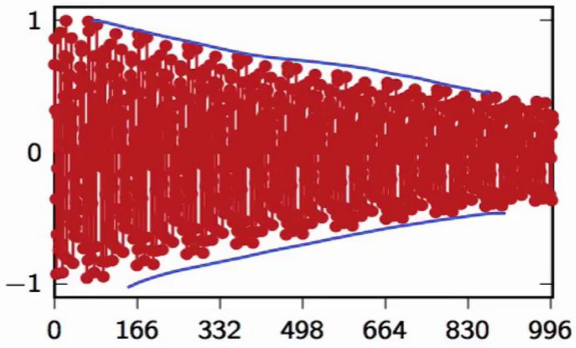
Summary

17m 32s

The Karplus-Strong Algorithm

$M = 100$, $\alpha = 0.9$, $\bar{x}[n]$: 100 random values between 0 and 99, zero elsewhere





27

This is actually the context in which it was invented by Karplus and Strong, and they found out that the best way to initialize the algorithm, the best way to fill the finite support of the input signal, is to use random values compared to a sufficiently fast decay factor. Here, we will pick Alpha equal to 0.9. We will use a hundred random values between -1 and 1 over the finite support of the input, and run the algorithm to obtain a waveform that looks like this. Here, you can see the decay factor because we're plotting the signal on a longer window. The sound is this, and it's remarkably similar to a harpsichord.

Notes

Summary

18m 59s