C++ Real-Time Audio Programming with Bela

Dr Andrew McPherson

Centre for Digital Music
School of Electronic Engineering and Computer Science
Queen Mary University of London

Founder and Director, Bela

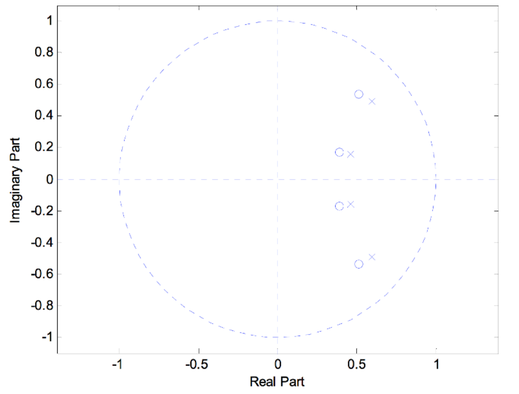
Course topics

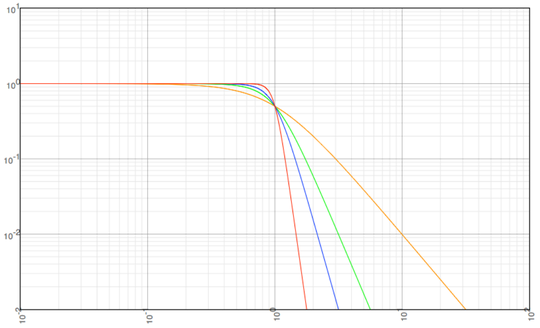
Programming topics	Music/audio topics
Working in real time	Oscillators
Buffers and arrays	Samples
Parameter control	Wavetables
Classes and objects	Control voltages
Analog and digital I/O	Gates and triggers
Filtering	Filters
Circular buffers	Delays and delay-based effects
Timing in real time	Metronomes and clocks
State machines	Envelopes
MIDI	ADSR
Block-based processing	MIDI
Threads	Additive synthesis
Fixed point arithmetic	Phase vocoders
ARM assembly language	Impulse reverb

Today

The course in a nutshell

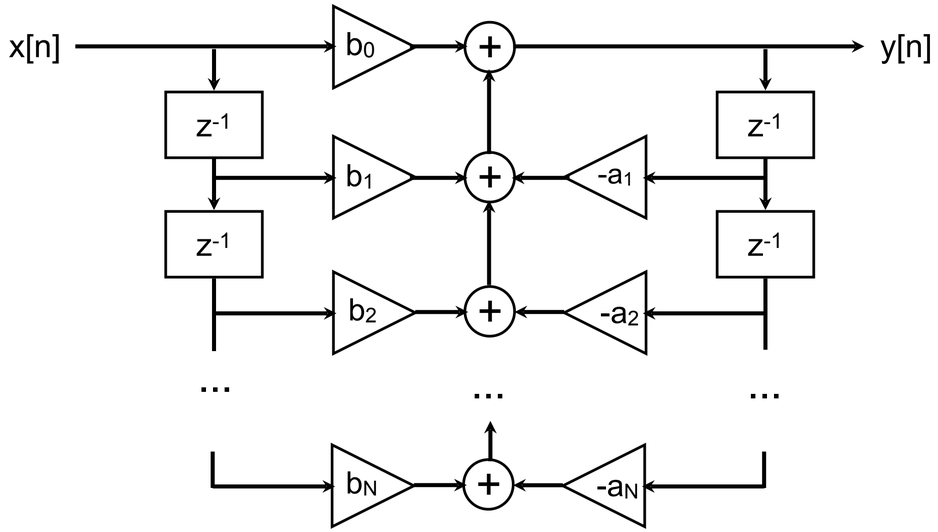
Specifications





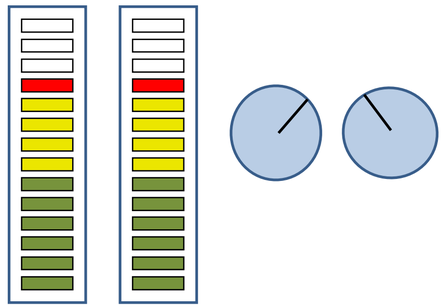
Design

$$y[n] = \sum_{k=1}^{N} a_k y[n-k] + \sum_{k=0}^{M} b_k x[n-k]$$



Implementation

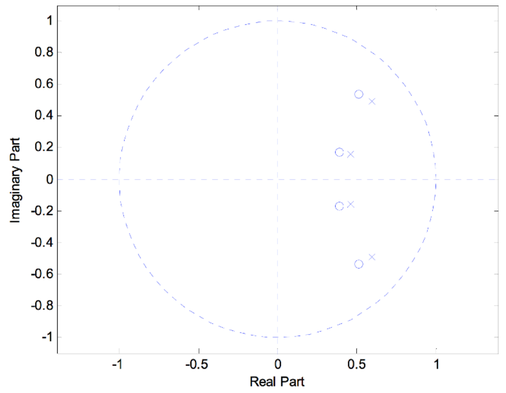
```
void render(BelaContext *context, void *
{
  /* Iterate over each channel */
  for(int channel = 0; channel < numChan
    /* Then iterate over each sample wit
    for(int n = 0; n < numSamples; n++)
      /* Calculate the sample... */
      float sample = gAmplitude *
                     sin(2.0 * M_PI * (f
                                 * gFrequ
```

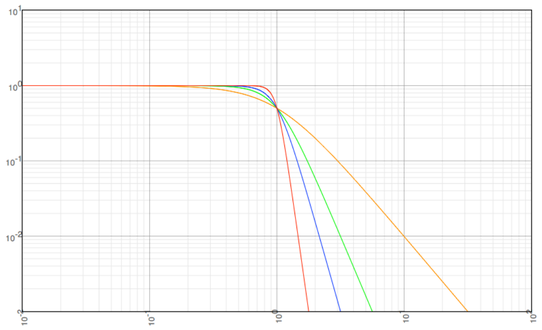


Typical digital signal processing course

The course in a nutshell

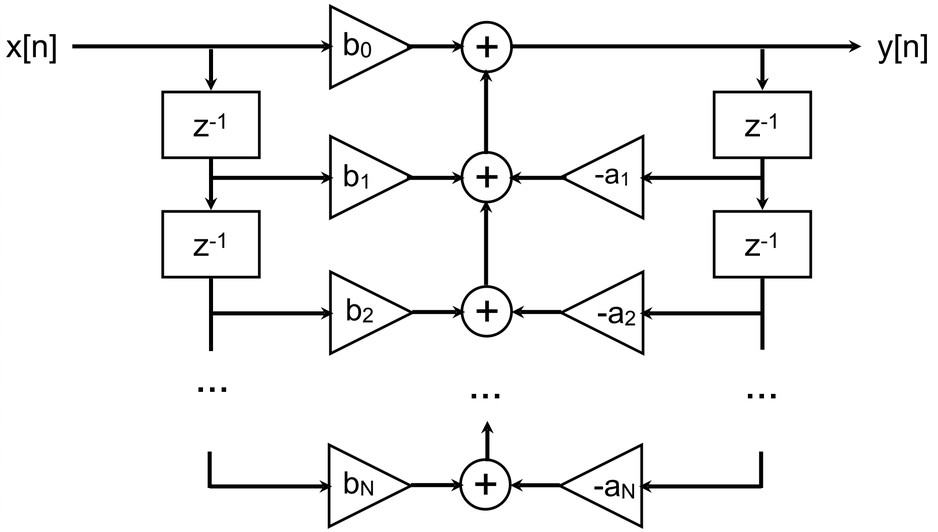
Specifications





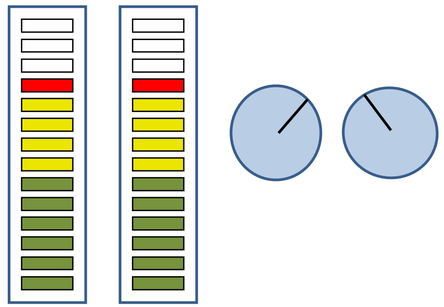
Design

$$y[n] = \sum_{k=1}^{N} a_k y[n-k] + \sum_{k=0}^{M} b_k x[n-k]$$



Implementation

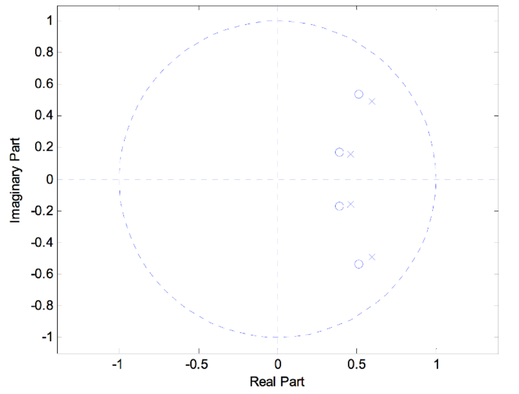
```
void render(BelaContext *context, void *
{
  /* Iterate over each channel */
  for(int channel = 0; channel < numChan
    /* Then iterate over each sample wit
    for(int n = 0; n < numSamples; n++)
      /* Calculate the sample... */
      float sample = gAmplitude *
                     sin(2.0 * M_PI * (f
                                 * gFrequ
```

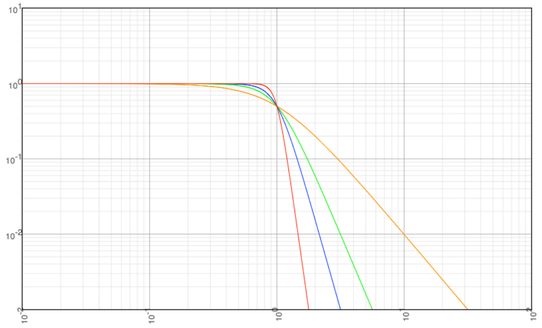


This course

The course in a nutshell

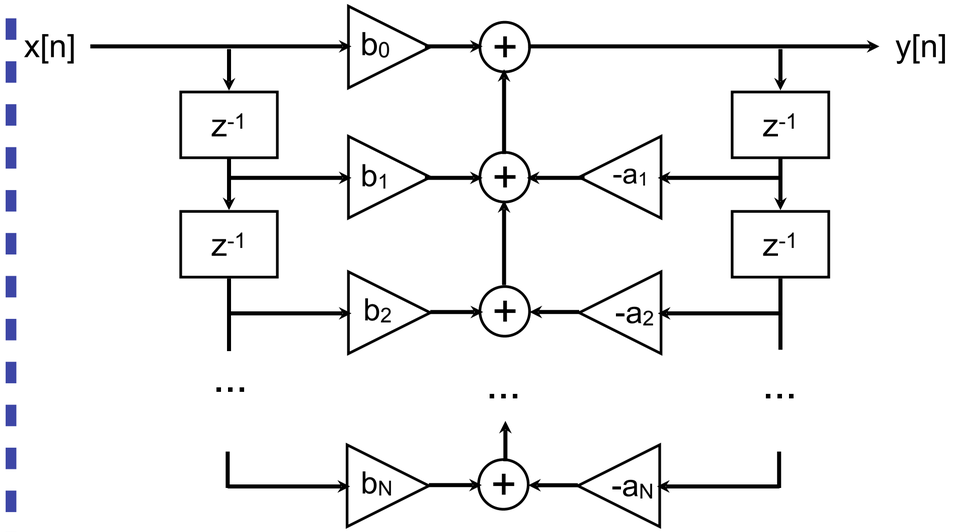
Specifications





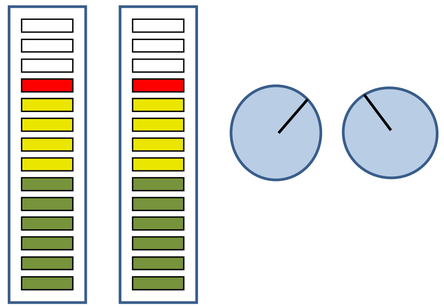
Design

$$y[n] = \sum_{k=1}^{N} a_k y[n-k] + \sum_{k=0}^{M} b_k x[n-k]$$



Implementation

```
void render(BelaContext *context, void *
{
  /* Iterate over each channel */
  for(int channel = 0; channel < numChan
    /* Then iterate over each sample wit
    for(int n = 0; n < numSamples; n++)
      /* Calculate the sample... */
      float sample = gAmplitude *
                     sin(2.0 * M_PI * (f
                                 * gFrequ
```



- We will turn signal processing theory into efficient C++ code
- We'll use Bela to process audio and sensor signals in real time

Lecture 8: Filters

What you'll learn today:

Basic theory of digital filters
Filter terminology: cutoff, bandwidth, Q, order
Turning filter equations into code

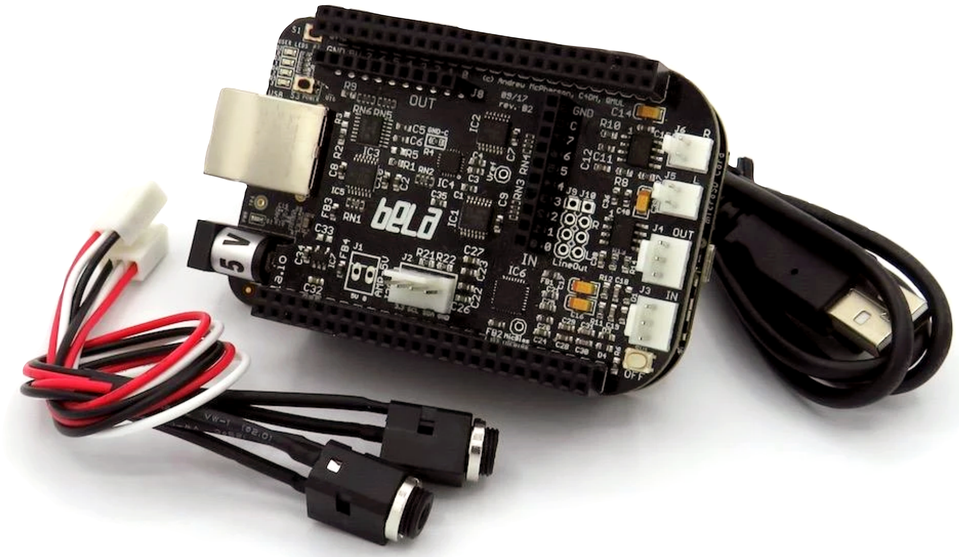
What you'll make today:

An adjustable resonant lowpass filter

Companion materials:

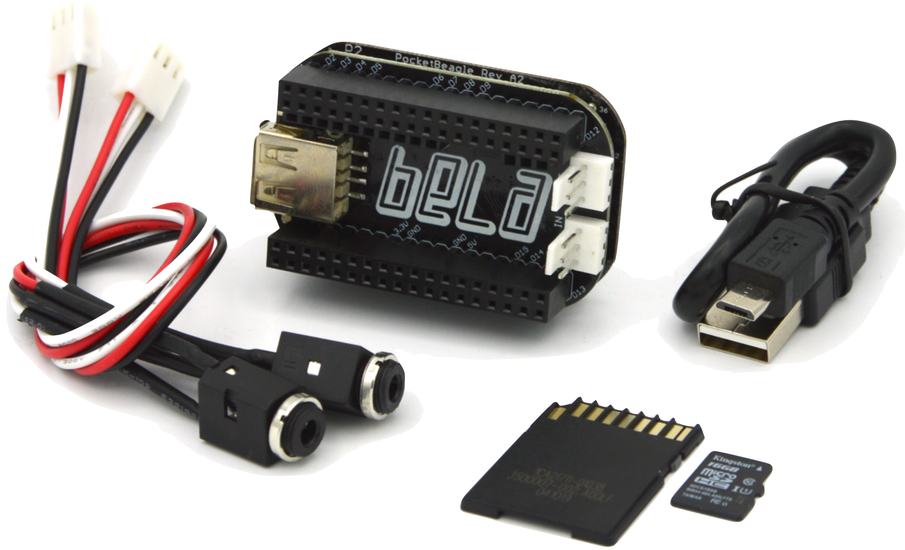
github.com/BelaPlatform/bela-online-course

What you'll need



Bela Starter Kit

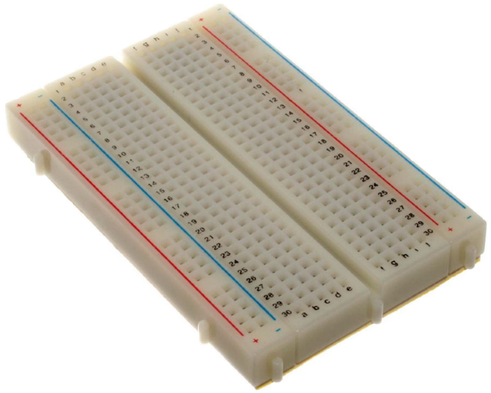
or

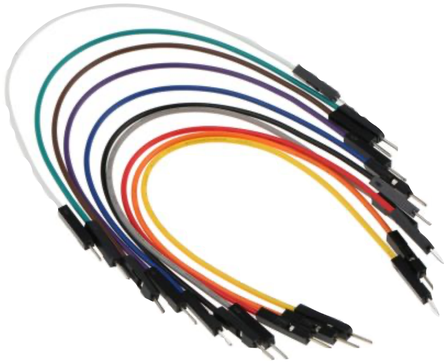


Bela Mini Starter Kit

[shop.bela.io]

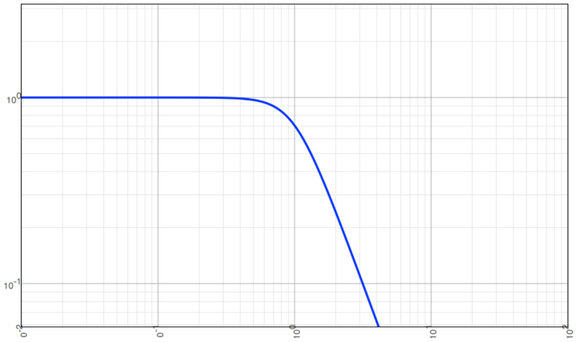
Recommended for some lectures:



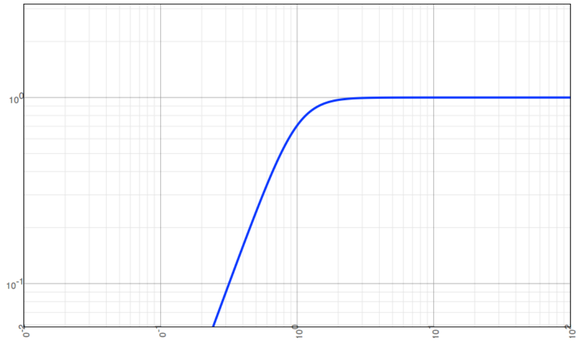


Filters

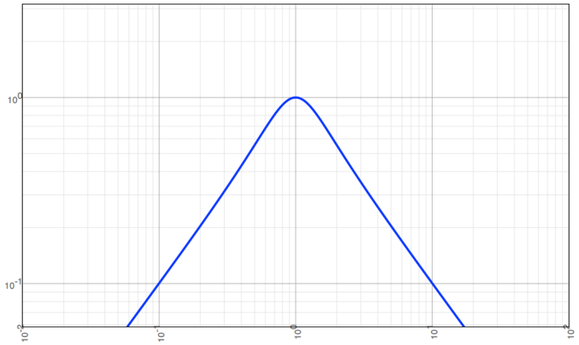
- Filters are a fundamental building block of DSP systems
  - They alter the frequency and phase content (and therefore the waveforms) of signals
- There are a huge number of filter types and applications
  - Some common types in audio:



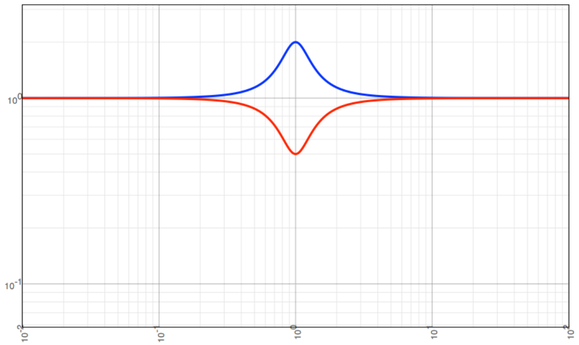
Lowpass



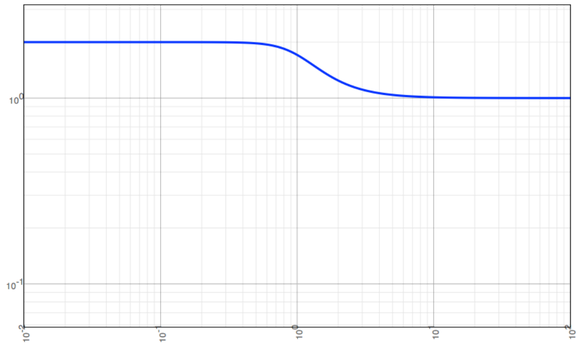
Highpass



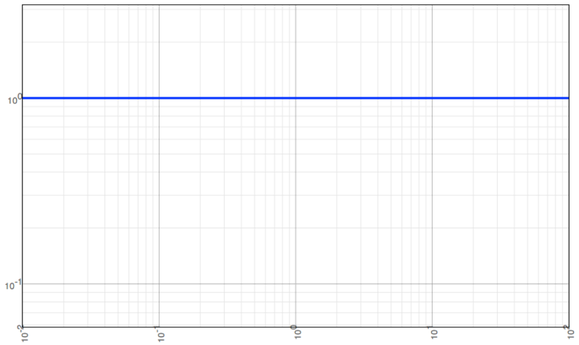
Bandpass



Peak / Notch



Shelving



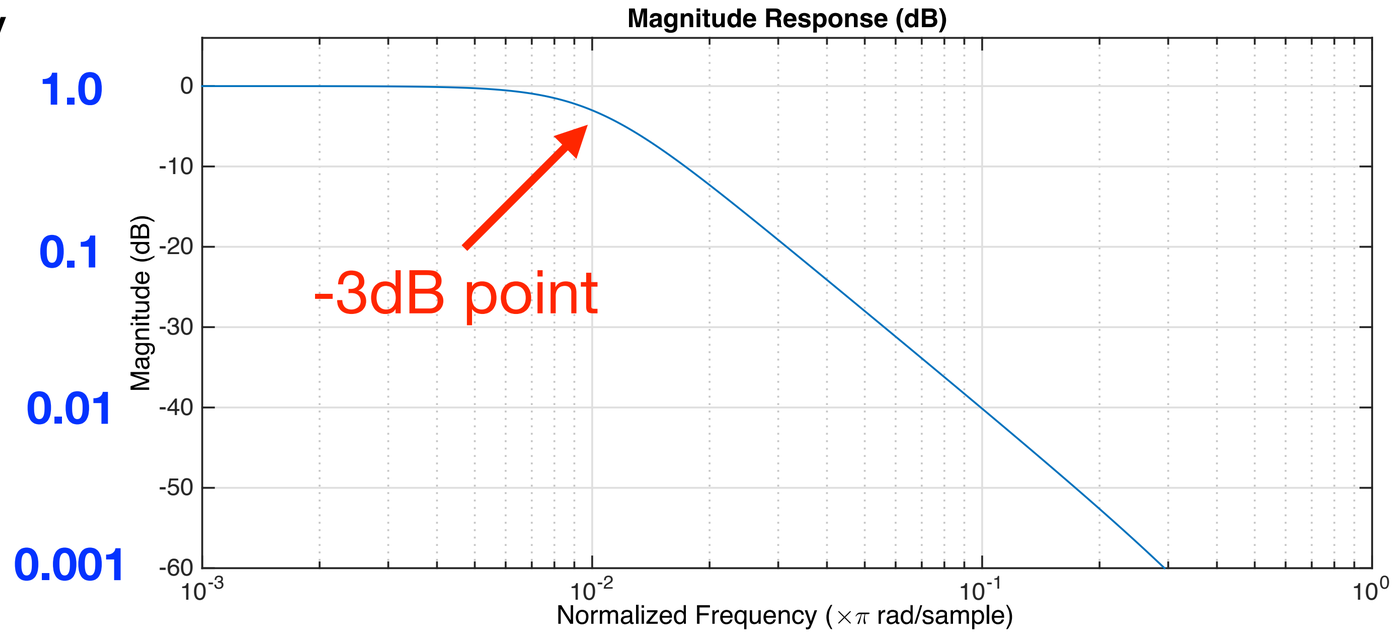
Allpass

Decibel amplitude scale

- The decibel scale is a logarithmic scale used across science and engineering to express ratios between values
  - For a given ratio *A*, we could express the same thing in linear or decibel terms:

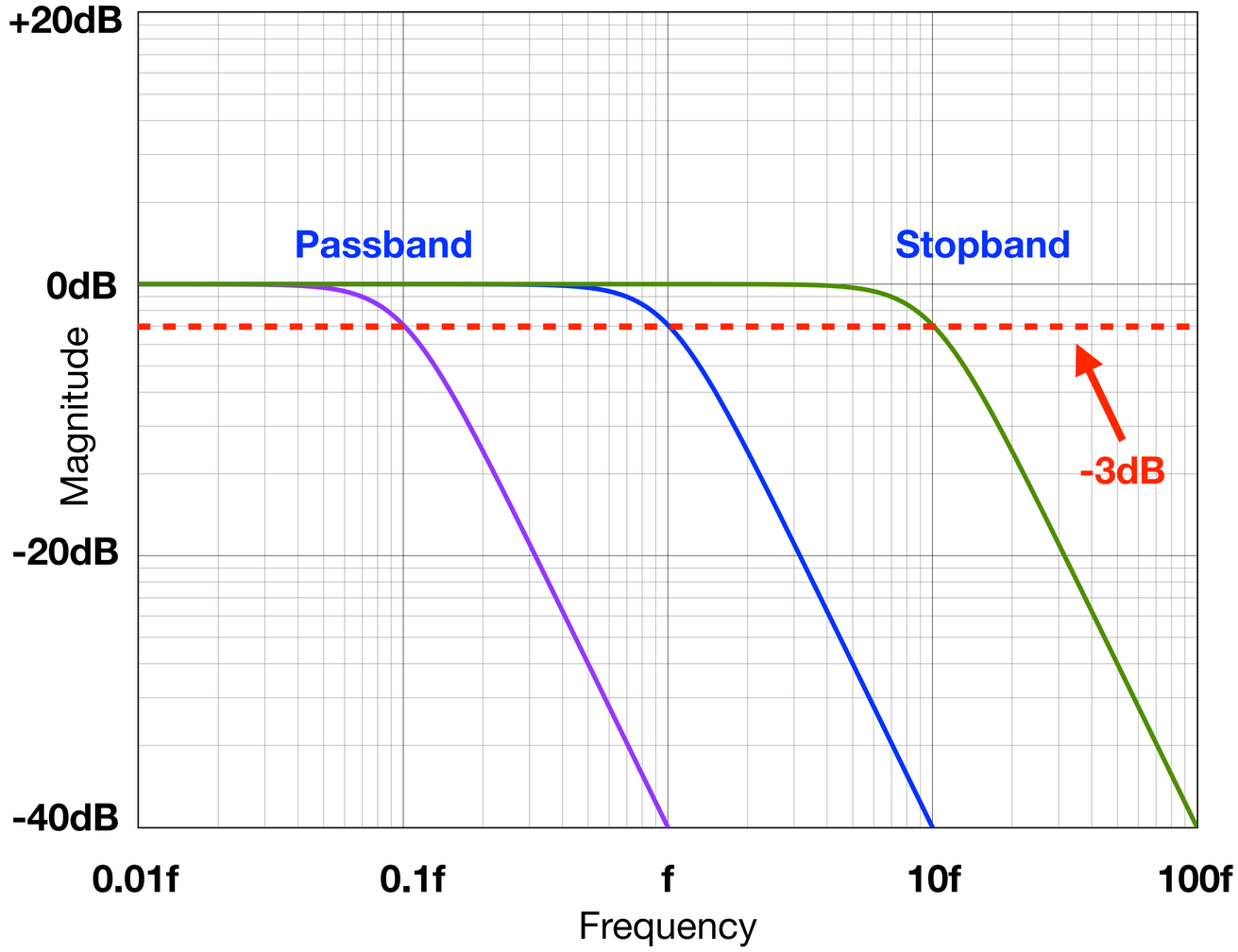
$$A_{dB} = 20\log_{10}(A_{linear}) \quad \longleftrightarrow \quad A_{linear} = 10^{\frac{A_{dB}}{20}}$$

- Decibel scales are often used in specifying filter magnitude response
  - By convention, the critical frequency of many standard filter types is the -3dB point
  - What is -3dB as a linear ratio?
    - $10^{\frac{-3}{20}} \approx .707$
    - More precisely, we are often interested in where the response drops by a factor of $\sqrt{2}$



Filter terminology

- Cutoff frequency
  - Transition between the passband and stopband
  - Often specified as the -3dB point
    - The point where the magnitude response is down 3dB compared to the passband level
    - Not necessarily a raw gain of -3dB!
  - For a bandpass, peak or notch filter, we might talk about centre frequency instead of cutoff frequency

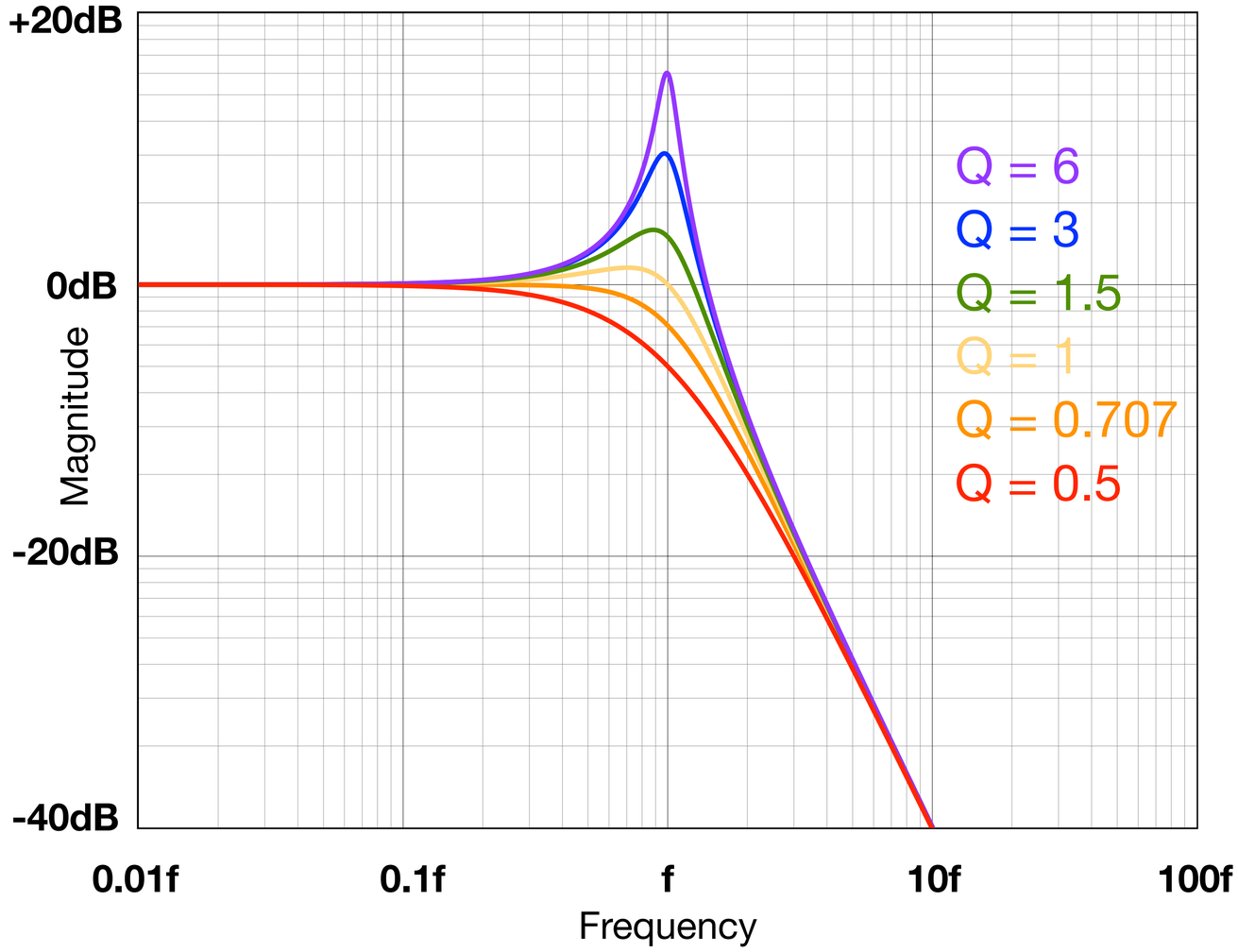


Filter terminology

- Bandwidth
  - For bandpass filters or filters with a resonant peak: bandwidth is the width of the peak
  - In bandpass filters, bandwidth measured between frequencies where gain is -3dB less than peak
- Q factor
  - Defined as the centre frequency divided by the bandwidth:

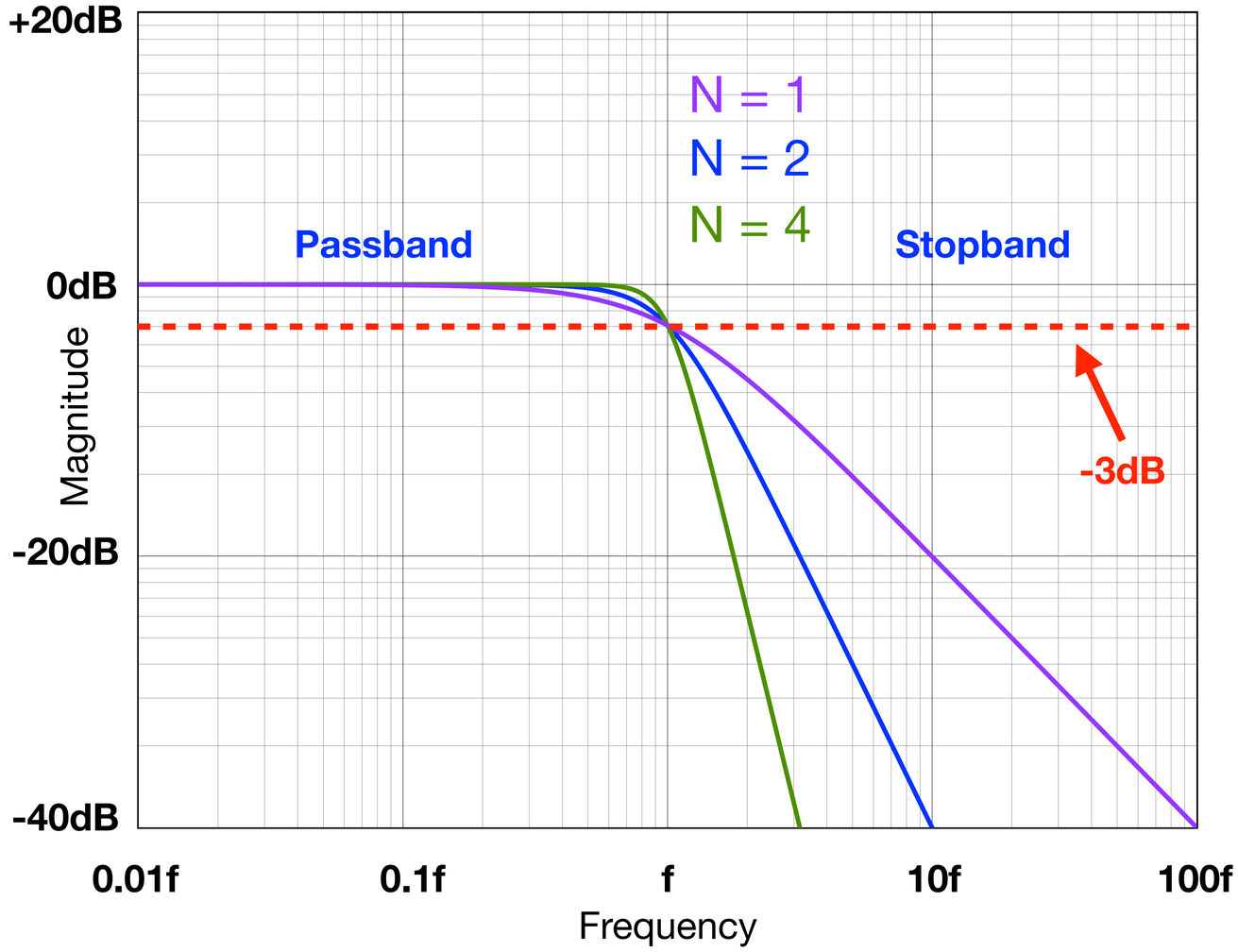
$$Q = \frac{f_c}{\Delta f}$$

  - Low/highpass filters have a Q related to the sharpness of the resonant peak



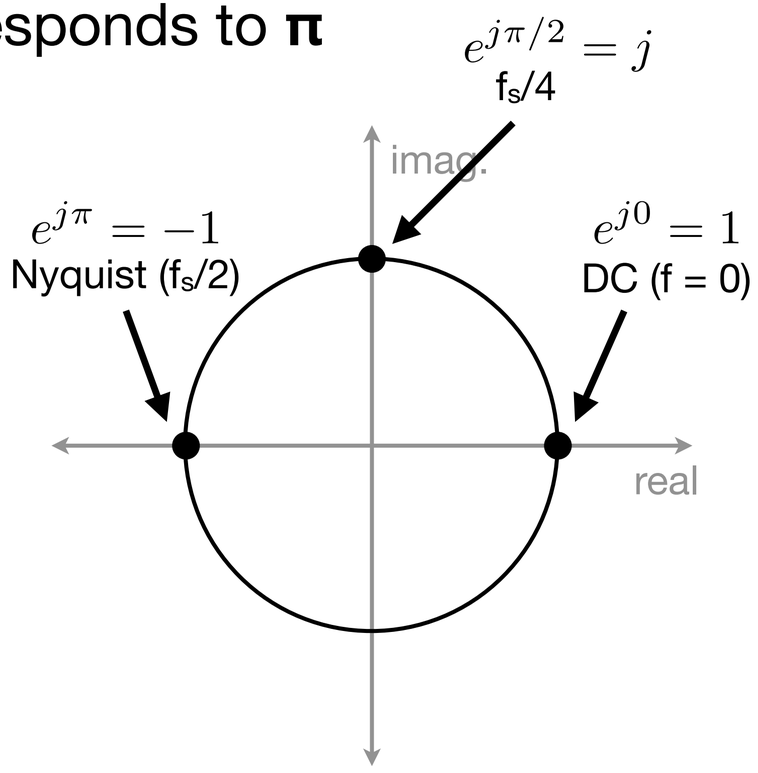
Filter terminology

- Order
  - Determines the steepness of the filter rolloff in the stopband
  - For order *N*, the rolloff is:
    - 6**N* dB/octave, or
    - 20**N* dB/decade
  - In discrete time, the number of taps (previous inputs/outputs) determines the order
    - For example, a 2nd-order IIR filter will use the previous 2 inputs and 2 outputs
  - Any higher-order filter can be built out of second-order sections



Frequency in discrete time

- In continuous time, we represent frequency in Hertz (cycles per second)
- In discrete time, we have a different mathematical formulation for frequency
  - Frequency normalised between **0** and **2π** (2π corresponds to the sample rate)
  - The Nyquist rate, the highest representable frequency, corresponds to **π**
- We will often see frequency-domain representations of discrete-time filters written in terms of ***z***
  - This is based on the *z*-transform (see references in the companion materials)
  - Frequency in the *z*-domain is mapped to the unit circle
  - $z = e^{j\omega}$ where ω (range 0-2π) is the discrete-time frequency
  - If you're given a filter *H*(*z*) and you want to find out its response at a given frequency:
    - Plug in $z = e^{j\omega}$ and calculate the magnitude and phase



Filter equation

- In signal processing, we typically deal with linear time-invariant (LTI) filters
- These filters are specified by their coefficients:

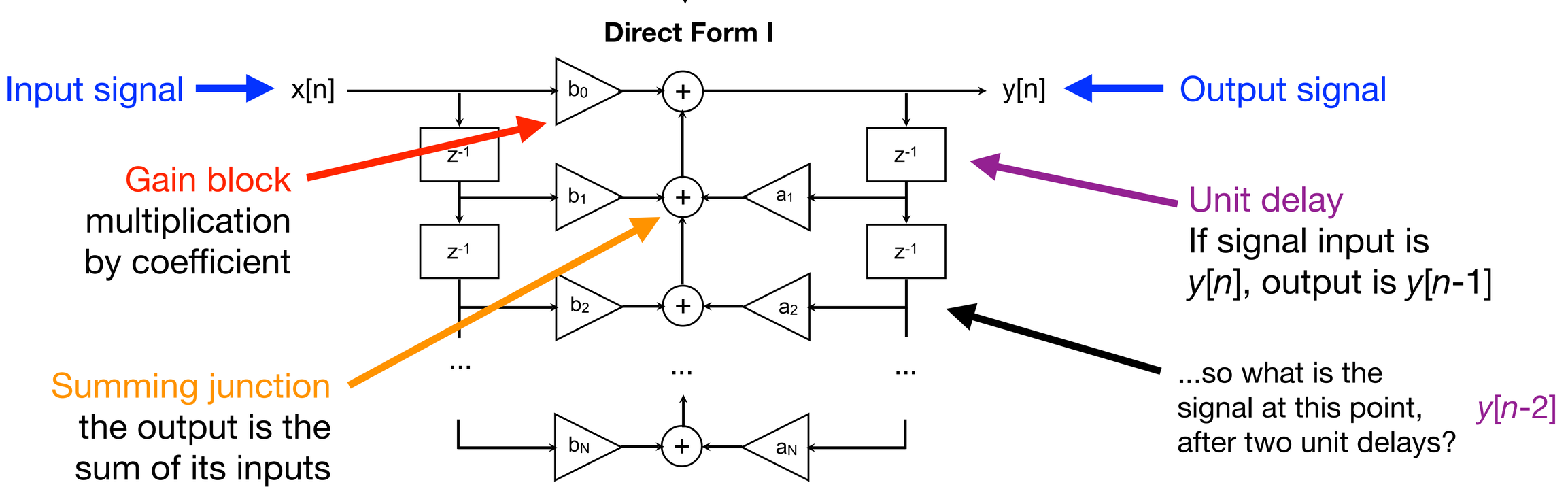
$$y[n] = \sum_{k=1}^{N} a_k y[n-k] + \sum_{k=0}^{M} b_k x[n-k]$$

  - The current output sample is the weighted sum of previous inputs and outputs
  - The coefficients b_k specify the weights of previous inputs
  - The coefficients a_k specify the weights of previous outputs
  - Going from specifications to coefficients is a topic for a DSP class
- Given a set of coefficients, how do we implement a filter in code?
  - Assume the coefficients are calculated for us, and stay the same over time
  - For each new sample, we need to calculate the equation above

Filter block diagram

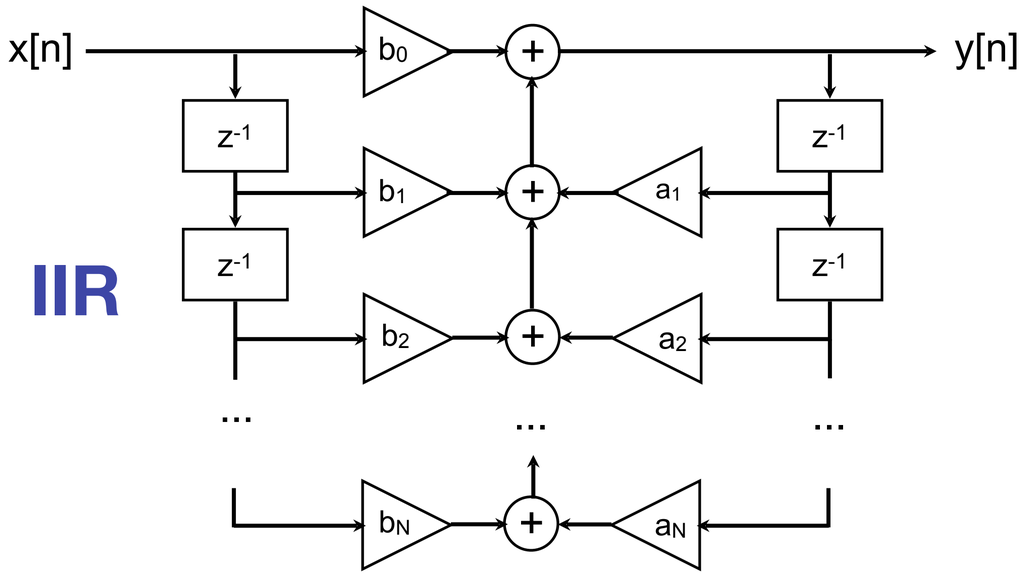
- Filter equations can also be written as block diagrams:

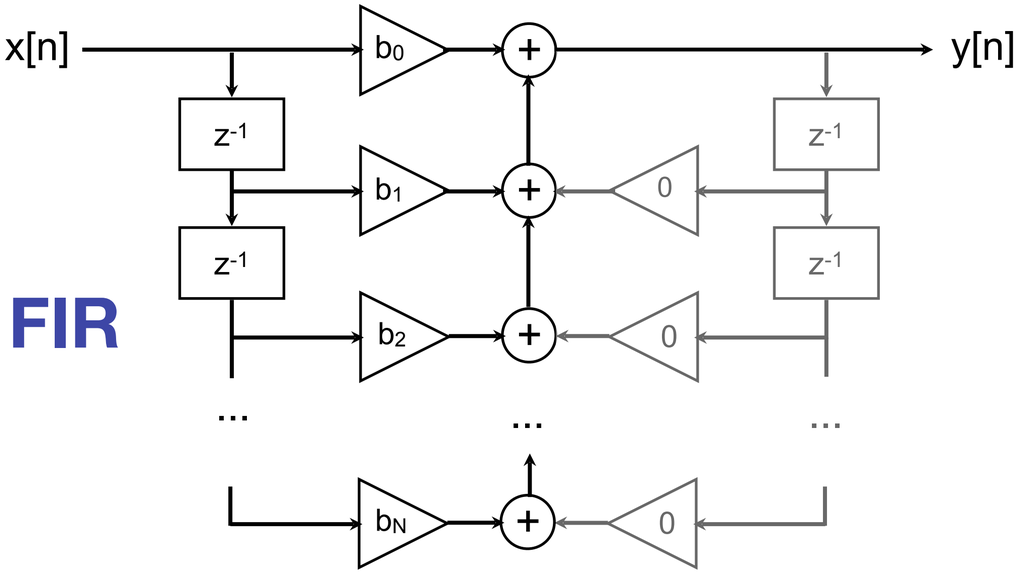
$$y[n] = \sum_{k=1}^{N} a_k y[n-k] + \sum_{k=0}^{M} b_k x[n-k]$$



IIR and FIR filters

- If a filter depends on its previous outputs as well as previous inputs, it is an Infinite Impulse Response (IIR) filter
  - Feedback means that after an impulse comes in, the output will never return exactly to 0





- If a filter depends only on its previous inputs (all $a_k = 0$), it is a Finite Impulse Response (FIR) filter
  - With an impulse input, the output will return to 0 after a finite number of samples

Filter example

- Let's implement a simple first-order difference filter: $y[n] = x[n] - x[n-1]$
  - Current output equals the current input minus the previous input
  - Is this FIR or IIR?
- DSP theory review: is this a high-pass or low-pass filter?
  - Intuitively: what happens if *x*[*n*] is a constant value?
  - Let's look at the z-transform:

$$Y(z) = X(z) - z^{-1}X(z) \longrightarrow H(z) = Y(z)/X(z) = 1 - z^{-1} = \frac{z-1}{z}$$

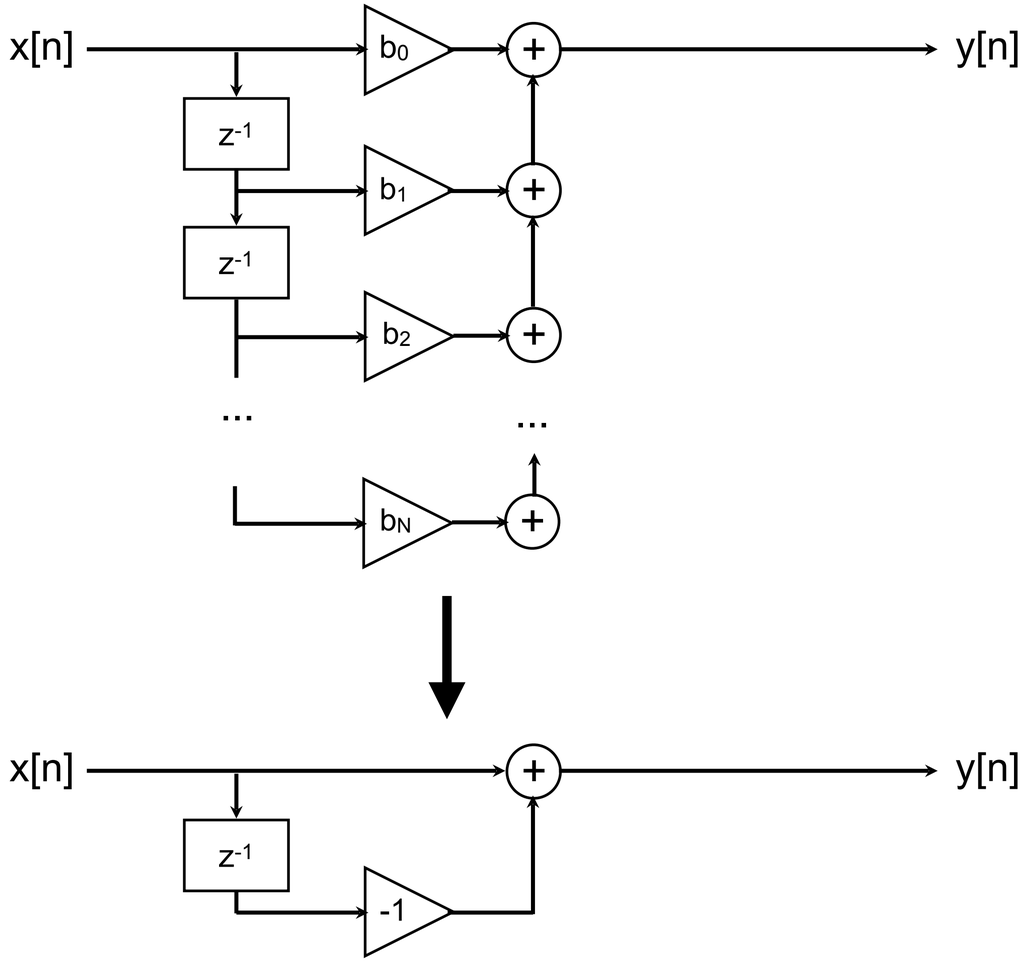
  - There's a zero at $z = 1$. What frequency is that?

$$e^{j\omega} = 1 \implies \omega = 0$$

  - So it's a very simple high-pass filter

Filter example

- We want to implement this filter: $y[n] = x[n] - x[n-1]$
- Block diagram for an FIR filter:
- What are the coefficients b_n?
  - $b_0 = 1$
  - $b_1 = -1$
- Task: in the sample-player-filter project, implement this filter in code
  - What information do you need to keep track of?
    - Previous value of $x[n]$
    - Use a global variable



Filter code

```
float gLastSample = 0;
```

global variable to hold *x*[*n* - 1]

```
for(unsigned int n = 0; n < context->audioFrames; n++) {
    // Read the next sample from the buffer
    float in = gPlayer.process();

    // y[n] = x[n] - x[n-1]
    float out = in - gLastSample;
    gLastSample = in;

    for(unsigned int channel = 0; channel < context->audioOutChannels; channel++) {
       // Write the sample to every audio output channel
       audioWrite(context, n, channel, out);
    }
}
```

so that the **next** time it holds x[n-1]

set `gLastSample` to x[n]...

Filter code

```
for(unsigned int n = 0; n < context->audioFrames; n++) {
    // Read the next sample from the buffer
    float in = gPlayer.process();

    // y[n] = x[n] - x[n-1]
    float out = in - gLastSample;
    gLastSample = in;

    // ....
}
```

n	in	gLastSample	out
0 (begin)	x[0]	0	x[0] - 0
0 (end)	x[0]	x[0]	x[0] - 0
1 (begin)	x[1]	x[0]	x[1] - x[0]
1 (end)	x[1]	x[1]	x[1] - x[0]
2 (begin)	x[2]	x[1]	x[2] - x[1]
2 (end)	x[2]	x[2]	x[2] - x[1]

Filter example 2

- This time, let's start from a frequency-domain function: $\frac{Y(z)}{X(z)} = \frac{1-\alpha}{1-\alpha z^{-1}}$
  - Rewrite as: $Y(z) - \alpha z^{-1} Y(z) = (1-\alpha)X$
  - Convert to time domain (z^{-N} turns into an N sample delay)

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

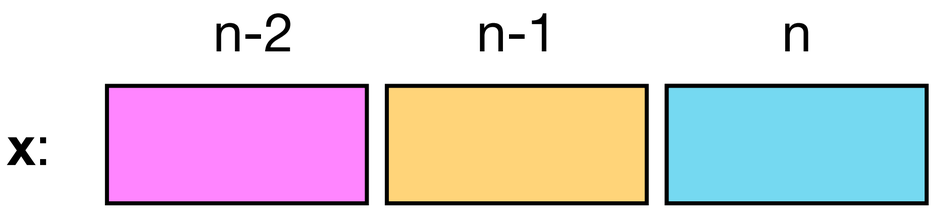
$$\downarrow$$

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

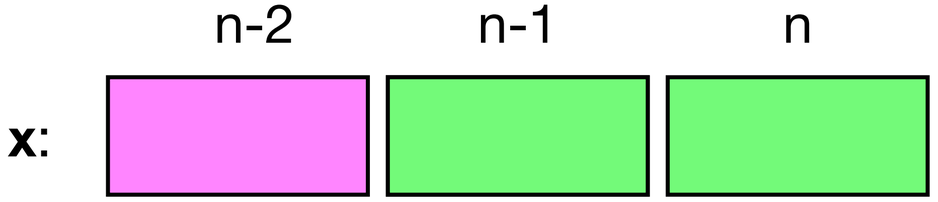
  - This is a first-order IIR lowpass filter
  - What values should we choose for α to get a low cutoff frequency?
    - Close to (but less than) 1.0 - try 0.99, for example
- Task: implement this using the sample-player-filter example
  - What previous value(s) do we need to keep track of this time?

Second-order filters

- Now let's implement a general second-order IIR filter
  - We can use 2nd-order ("biquad") sections to implement any type and order of IIR filter
- We won't focus in these lectures on how to calculate filter coefficients
  - See the companion materials for references
  - Instead, given a set of coefficients, how do we implement the equations as code?
- How many previous values of *x* and *y* do we need for a 2nd-order IIR filter?
  - 2 each. So how many global variables? 4
  - How do we keep track of *x*[*n*-2] and *y*[*n*-2]?
    - Analogously to what we did before: copy *x*[*n*-1] and *y*[*n*-1] at the end of the loop
    - Be careful about the order!



NO:



Second-order filter task

- Task: in the resonant-lowpass example
  - Implement the filter calculation in `render()`
  - Declare any global variables you need to remember the previous state
  - Coefficients for a resonant lowpass are calculated for you in `calculate_coefficients()`
  - Once it works, add GUI controls or analog inputs for frequency and Q
- Possible point of confusion: the sign of the *a* coefficients
  - Sometimes you will see the IIR filter equation written like this:

$$y[n] = \sum_{k=0}^{N} b_k x[n-k] + \sum_{k=1}^{N} a_k y[n-k]$$

  - Sometimes (including in this example), it's like this:

$$y[n] = \sum_{k=0}^{N} b_k x[n-k] - \sum_{k=1}^{N} a_k y[n-k]$$

Filters: from analog to digital

- The IIR filters we most commonly use in DSP are based on analog prototypes
  - Start with a continuous-time model (expressed in terms of its Laplace transform)
  - Transform it to the discrete-time domain, making a digital approximation of the analog behaviour
- Useful tool: the bilinear transformation
  - Start with a continuous-time filter $H_a(s)$
  - Perform this substitution:
    - where *T* is sampling period $s = \frac{2}{T}\frac{1 - z^{-1}}{1 + z^{-1}}$
  - Now implement the digital filter: $H_d(z) = H_a\left(\frac{2}{T}\frac{1 - z^{-1}}{1 + z^{-1}}\right)$
  - More info (Julius O. Smith, *Physical Audio Signal Processing*)
    - https://ccrma.stanford.edu/~jos/pasp/Bilinear_Transformation.html
- Bela provides the Biquad library for implementing common 2nd-order filters
  - Based on open-source code by Nigel Redmon (earlevel.com)

Keep in touch!

Social media:

@BelaPlatform

forum.bela.io

blog.bela.io

More resources and contact info at:

learn.bela.io/resources