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

Working in real time
Buffers and arrays
Parameter control
Classes and objects
Analog and digital I/O
Filtering
Timing in real time
Circular buffers
State machines
MIDI
Block-based processing
Threads
Fixed point arithmetic
ARM assembly language

Music/audio topics

Oscillators
Samples
Wavetables
Control voltages
Gates and triggers
Filters
Metronomes and clocks
Delays and delay-based effects
Envelopes
ADSR
MIDI
Additive synthesis
Phase vocoders
Impulse reverb

Today

Lecture 17: Block-based processing

What you'll learn today:

Generating and processing windows of a signal
The Fast Fourier Transform (FFT)
Sending data to the Bela GUI

What you'll make today:

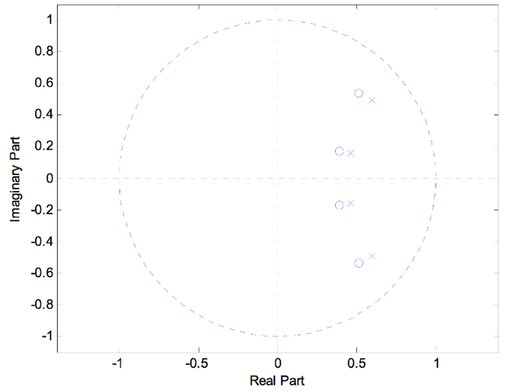
Two simple spectrum analysers with LEDs and GUI

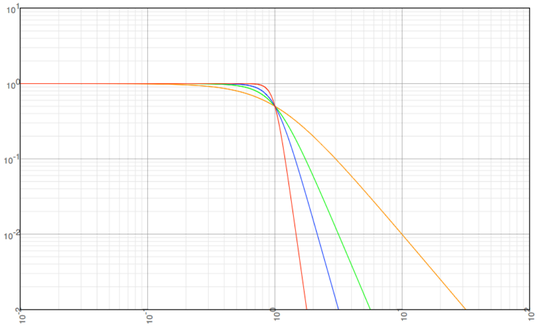
Companion materials:

github.com/BelaPlatform/bela-online-course

Theory and implementation

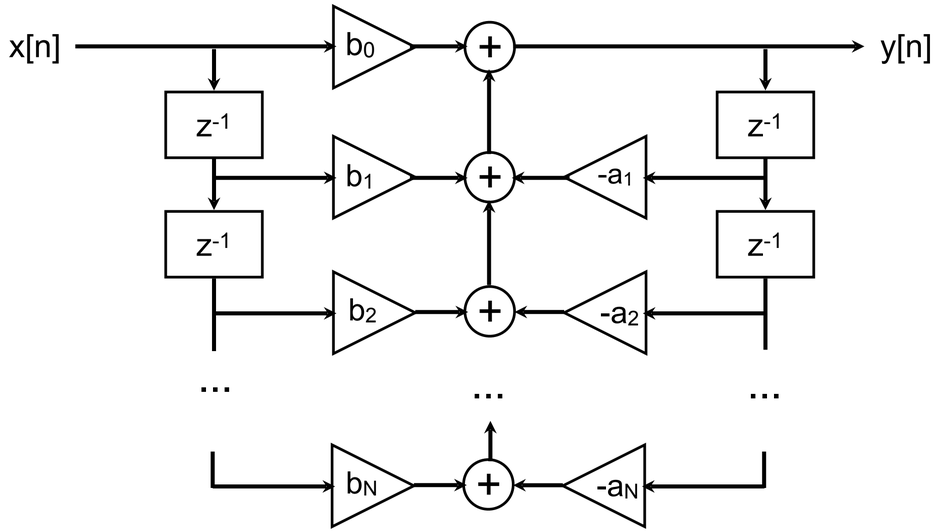
Specifications





Design

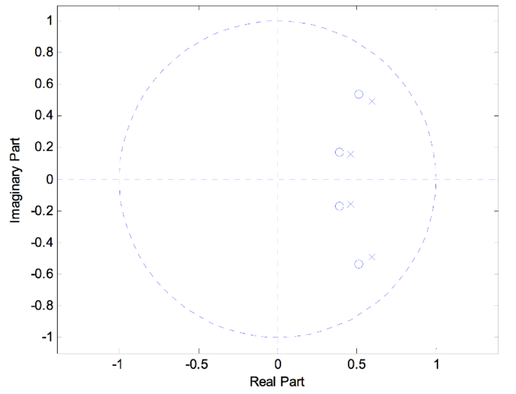
$$y[n] = \sum_{k=1}^{N} a_k y[n-k] + \sum_{k=0}^{M} b_k x[n-k]$$

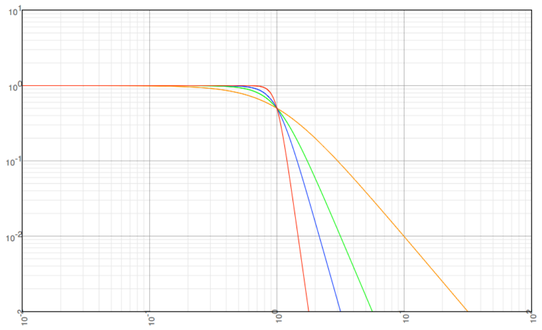


Typical digital signal processing course

Theory and implementation

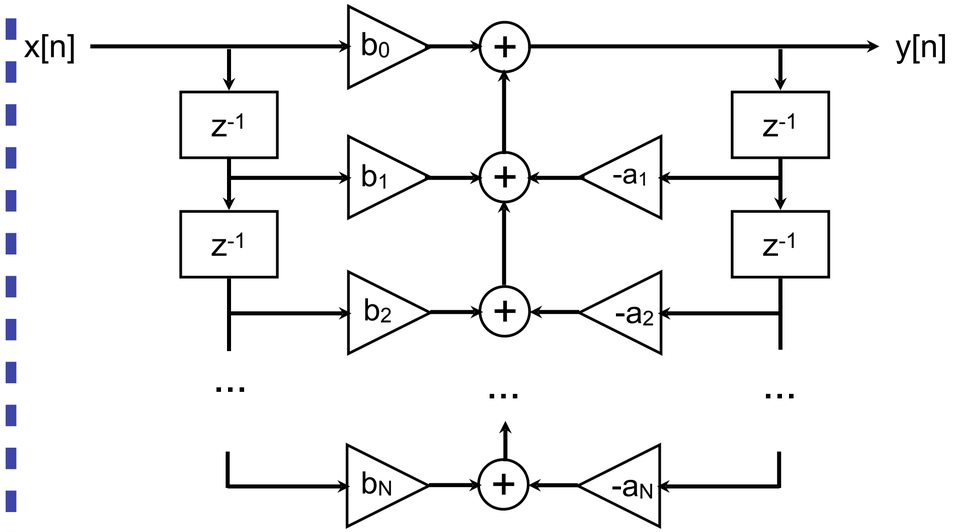
Specifications





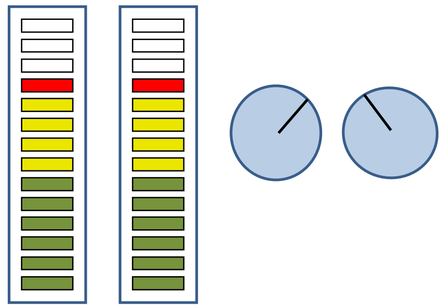
Design

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



Implementation

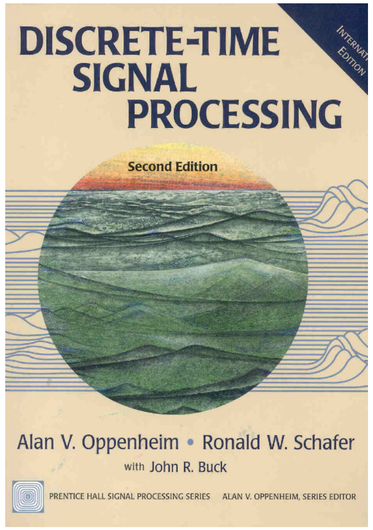
```
void render(BelaContext *context, void *
{
  /* Iterate over each channel */
  for(int channel = 0; channel < numChan
    /* Then iterate over each sample wit
    for(int n = 0; n < numSamples; n++)
      /* Calculate the sample... */
      float sample = gAmplitude *
                     sin(2.0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

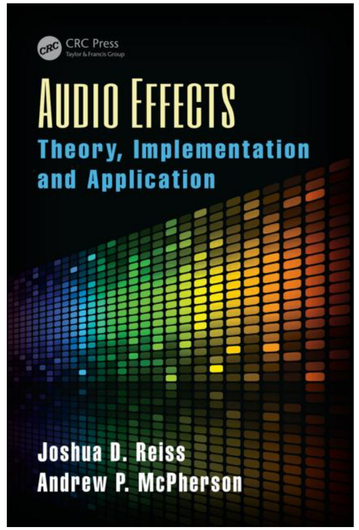


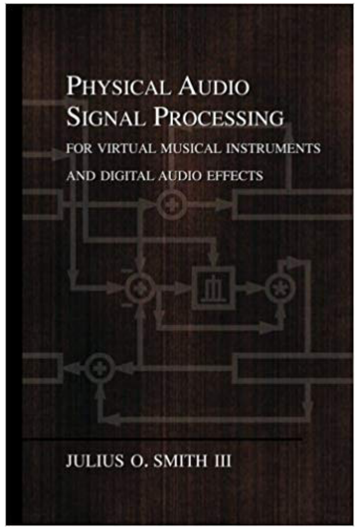
- We will turn signal processing theory into efficient C++ code
- There are plenty of excellent introductions to DSP mathematics

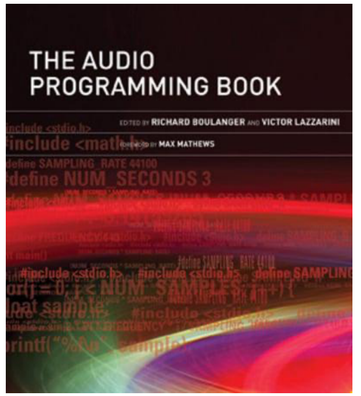
Books and references

- There are lots of good books and online resources about DSP and audio programming. For example:
  - Oppenheim, Schafer and Buck, *Discrete-Time Signal Processing*
  - Julius O. Smith, *Physical Audio Signal Processing*
  - Richard Boulanger and Victor Lazzarini (editors), *The Audio Programming Book*
  - Reiss and McPherson, *Audio Effects: Theory, Implementation and Application*
- See the GitHub companion site for links: **github.com/BelaPlatform/bela-online-course**

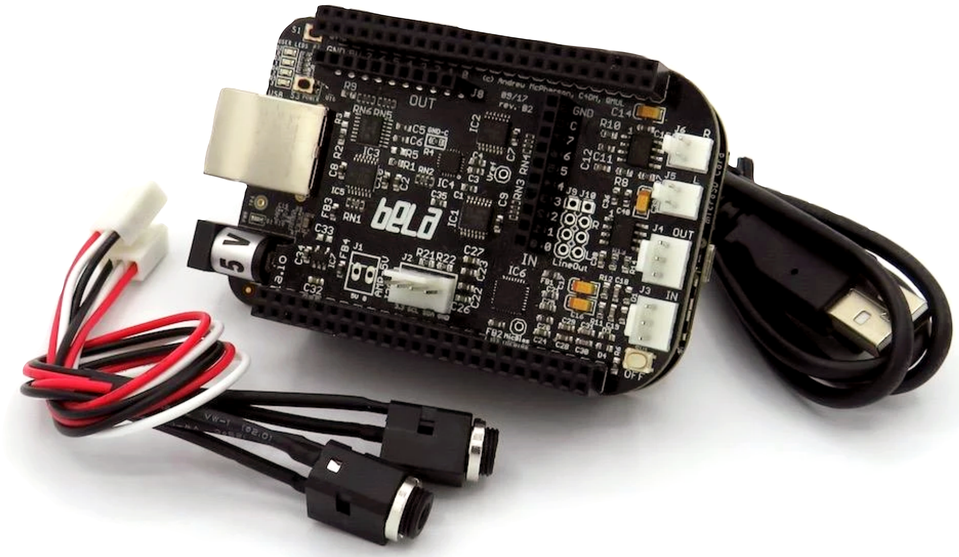






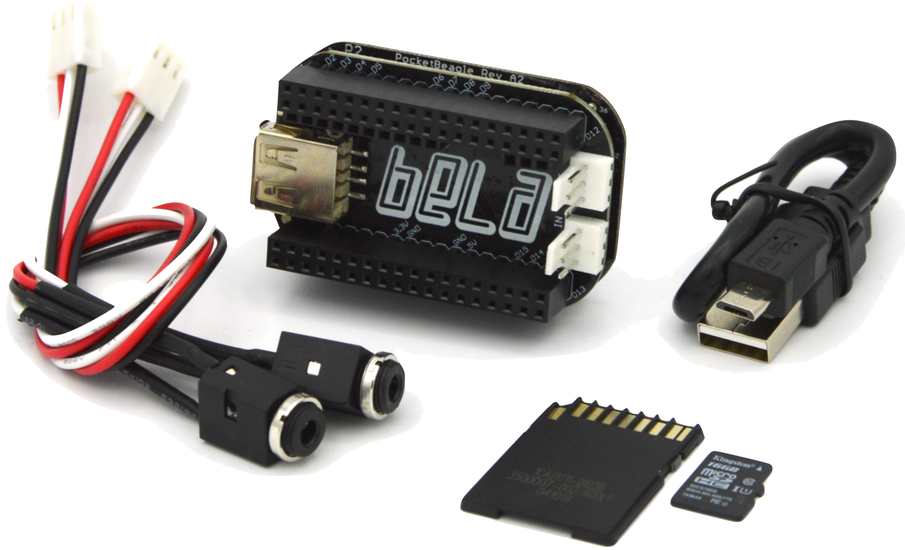


What you'll need



Bela Starter Kit

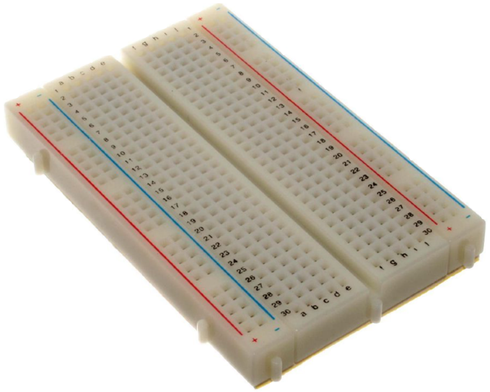
or

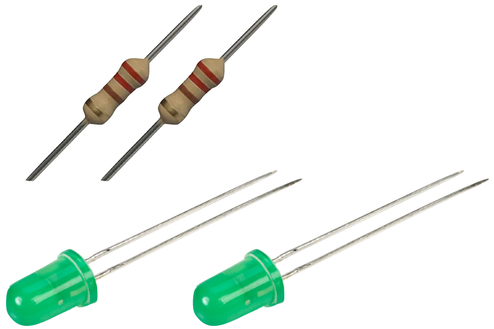


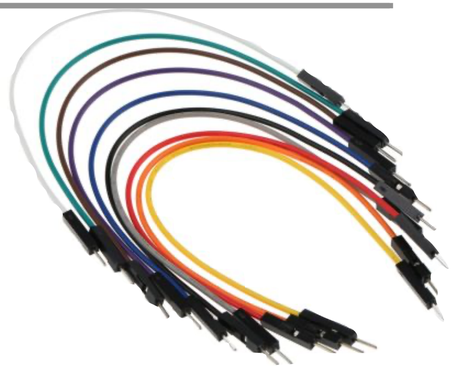
Bela Mini Starter Kit

[shop.bela.io]

Also needed for this lecture:

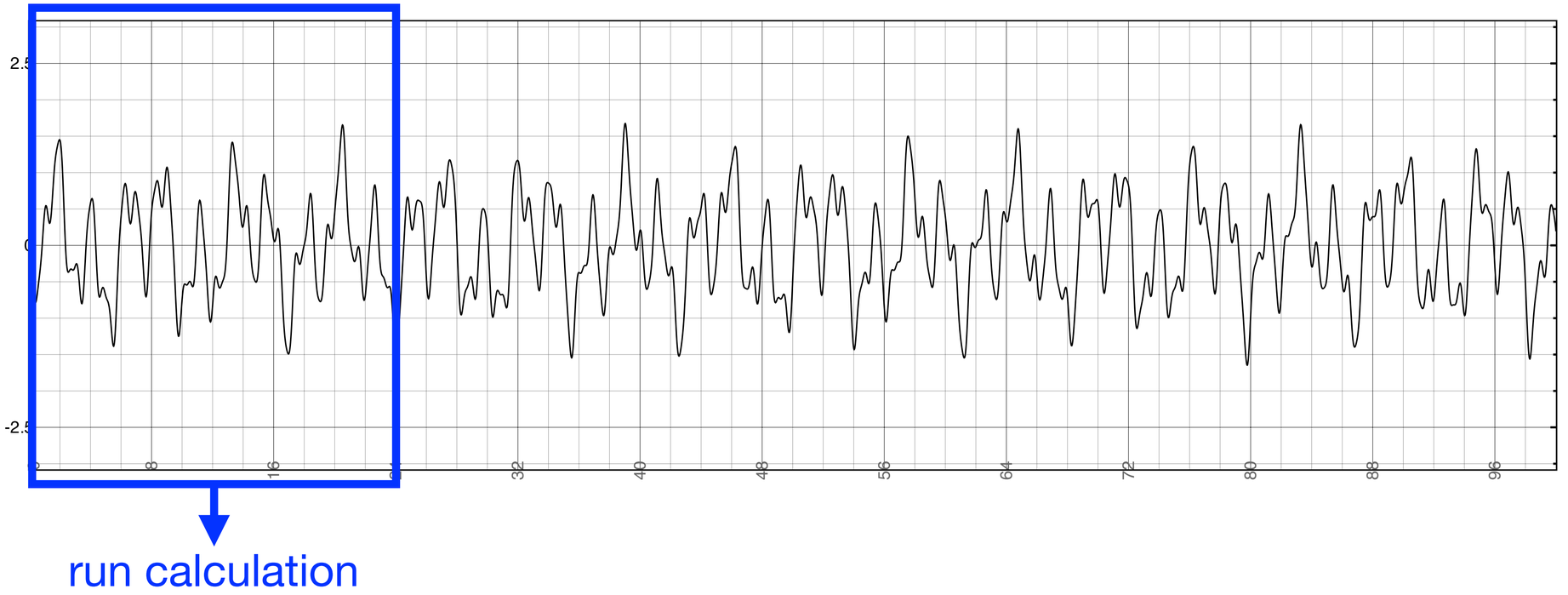






Block-based processing

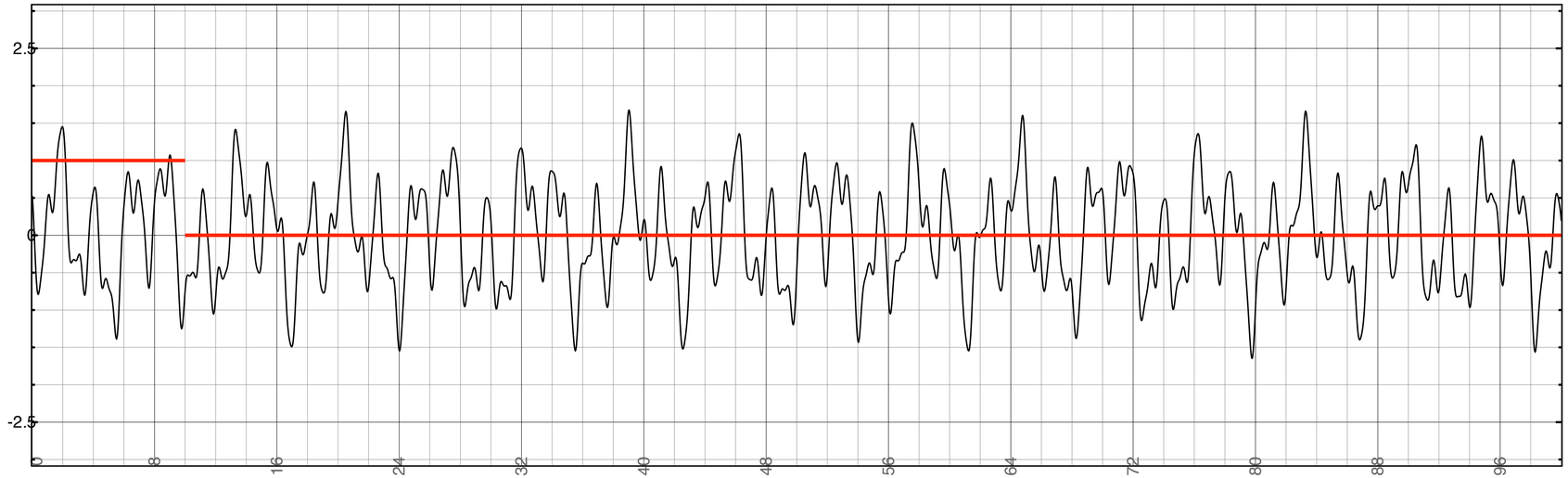
- Sometimes audio calculations need to run on a block of samples simultaneously
  - For example: wait until 512 samples arrive, then do something with all of them at once



- Compare this to filters, where we calculate a new output every sample
  - With block-based processing, we typically run the calculation periodically, not every sample

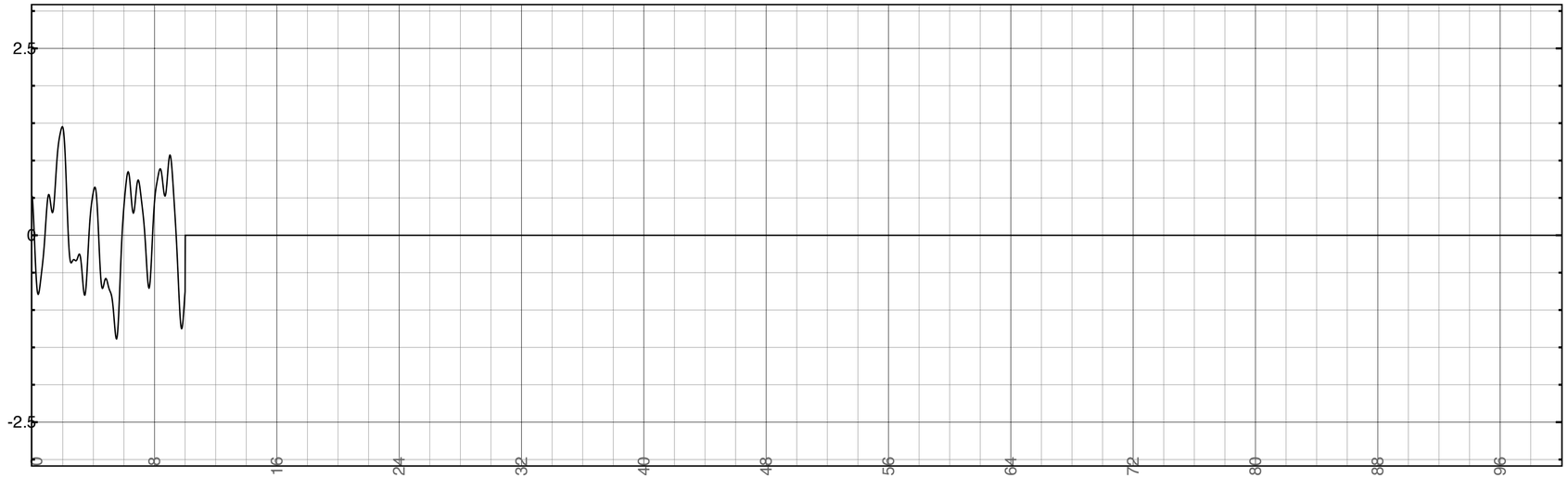
Block-based processing

- Block-based processing is not the same as non-realtime processing
  - We are not trying to run a calculation on the whole signal all at once
- Instead, we want to work with finite segments of the input signal
- How do we generate a block? Apply a window to the signal
  - Multiply by a function that's nonzero for a fixed length M samples
  - Window might have a shape: rectangular, triangular (Bartlett), Hamming, Blackman, etc.



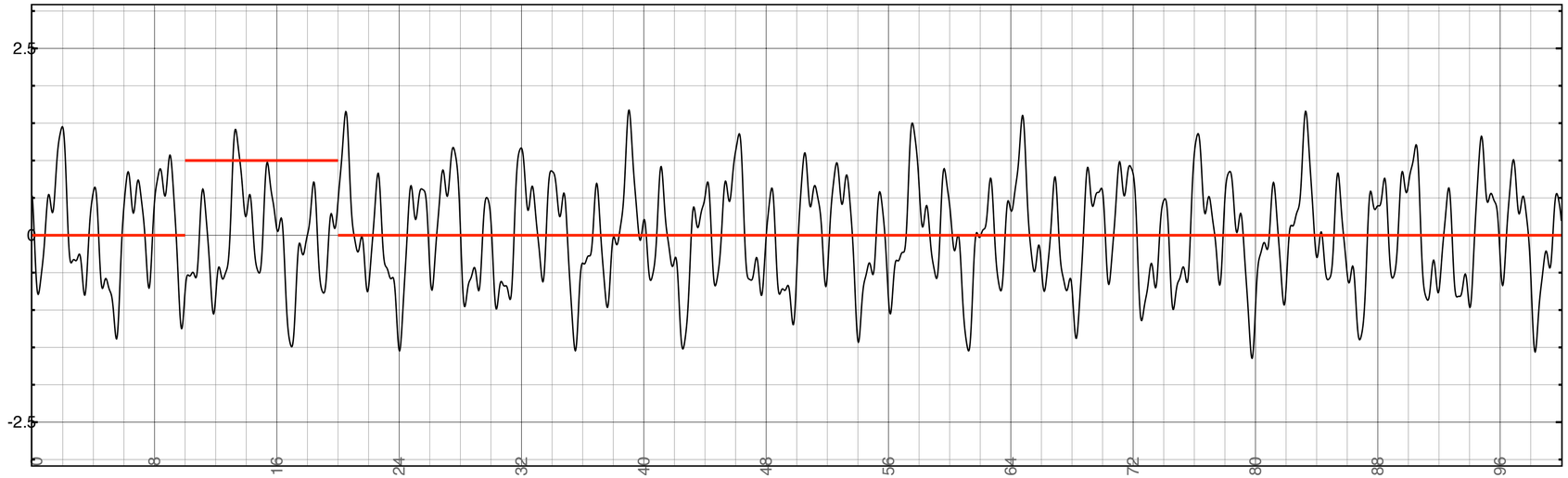
Block-based processing

- Block-based processing is not the same as non-realtime processing
  - We are not trying to run a calculation on the whole signal all at once
- Instead, we want to work with finite segments of the input signal
- How do we generate a block? Apply a window to the signal
  - Multiply by a function that's nonzero for a fixed length M samples
  - Window might have a shape: rectangular, triangular (Bartlett), Hamming, Blackman, etc.



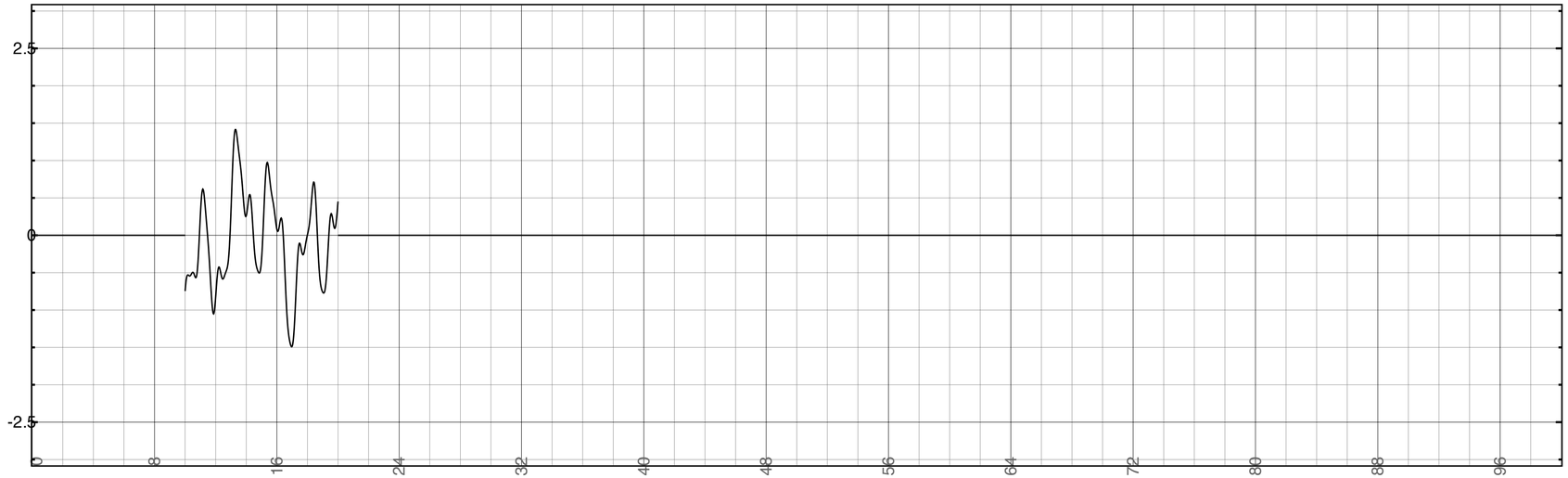
Block-based processing

- Block-based processing is not the same as non-realtime processing
  - We are not trying to run a calculation on the whole signal all at once
- Instead, we want to work with finite segments of the input signal
- How do we generate a block? Apply a window to the signal
  - Multiply by a function that's nonzero for a fixed length M samples
  - Window might have a shape: rectangular, triangular (Bartlett), Hamming, Blackman, etc.



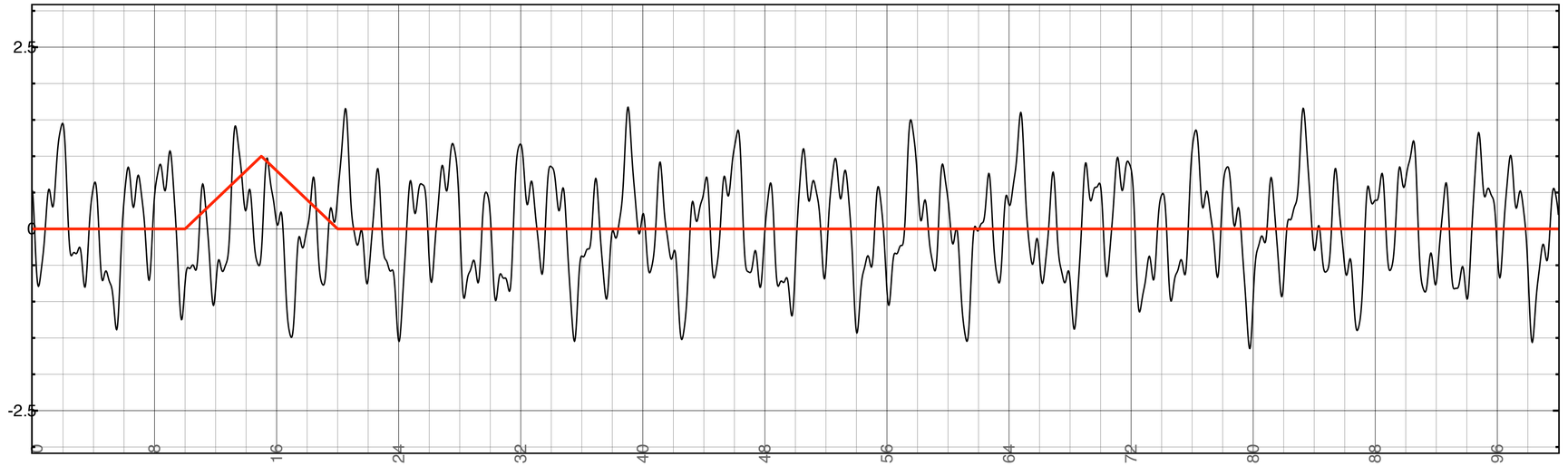
Block-based processing

- Block-based processing is not the same as non-realtime processing
  - We are not trying to run a calculation on the whole signal all at once
- Instead, we want to work with finite segments of the input signal
- How do we generate a block? Apply a window to the signal
  - Multiply by a function that's nonzero for a fixed length M samples
  - Window might have a shape: rectangular, triangular (Bartlett), Hamming, Blackman, etc.



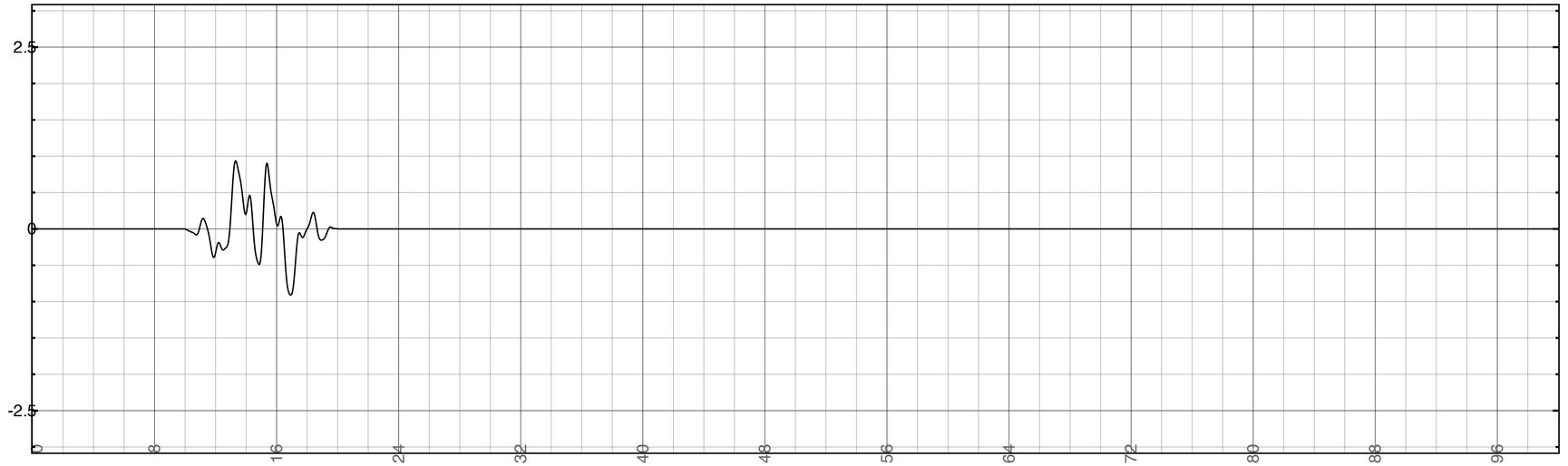
Block-based processing

- Block-based processing is not the same as non-realtime processing
  - We are not trying to run a calculation on the whole signal all at once
- Instead, we want to work with finite segments of the input signal
- How do we generate a block? Apply a window to the signal
  - Multiply by a function that's nonzero for a fixed length M samples
  - Window might have a shape: rectangular, triangular (Bartlett), Hamming, Blackman, etc.



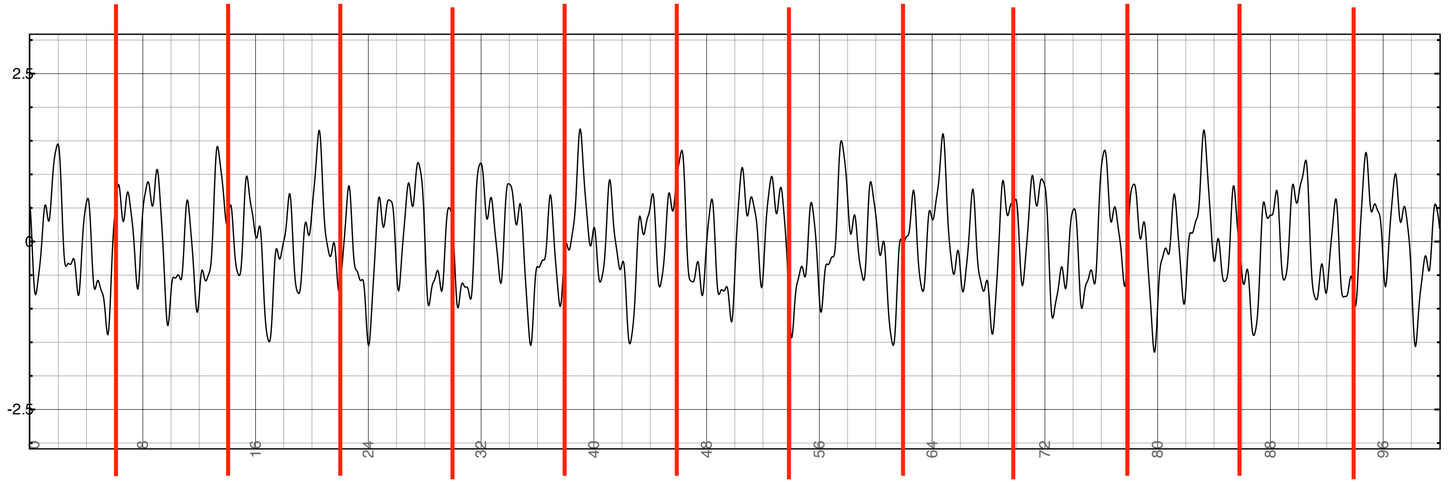
Block-based processing

- Block-based processing is not the same as non-realtime processing
  - We are not trying to run a calculation on the whole signal all at once
- Instead, we want to work with finite segments of the input signal
- How do we generate a block? Apply a window to the signal
  - Multiply by a function that's nonzero for a fixed length M samples
  - Window might have a shape: rectangular, triangular (Bartlett), Hamming, Blackman, etc.



Windowing a signal

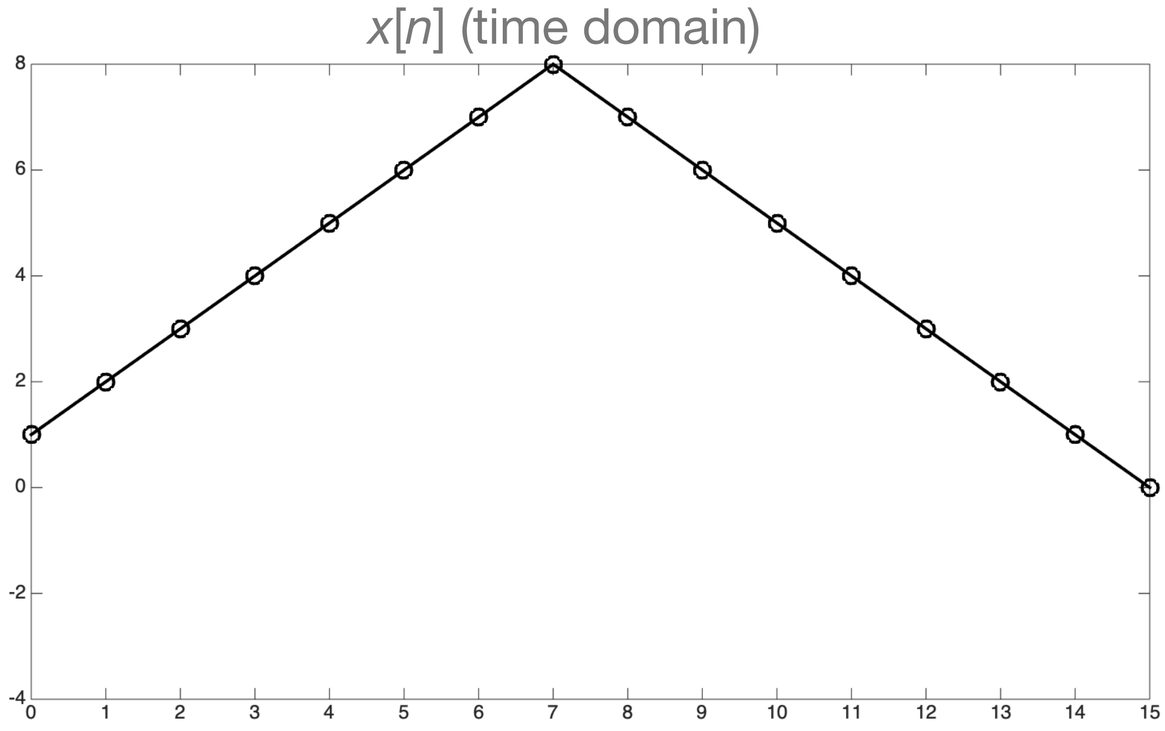
- Any arbitrarily-long signal can be divided into successive windows
  - This segmentation can be done in real time, as the signal comes in



- In the right circumstances, we can use the windows to reconstruct the signal
  - Need window functions to overlap so that they all add to a constant value
  - Constant Overlap-Add (COLA) criterion: we'll come back to this later
- We call this segmentation and reconstruction block-based processing
  - We work on an entire window (or block) of samples at once

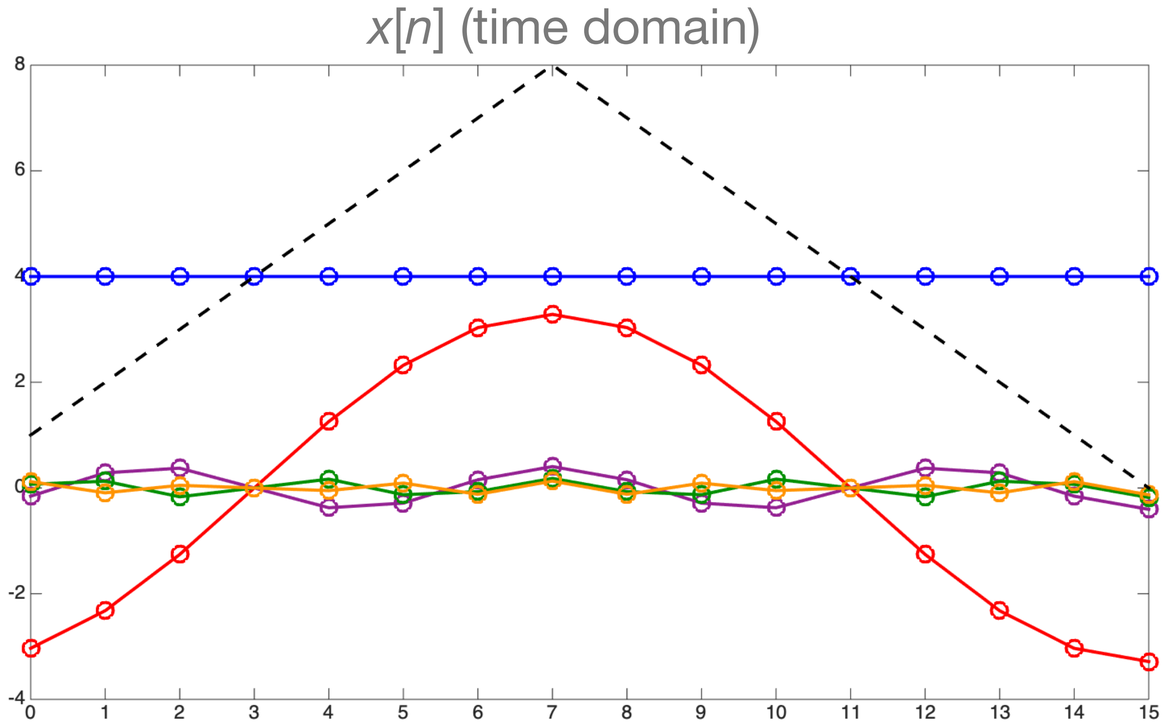
The Fast Fourier Transform

- Most common use of block-based processing: the Fast Fourier Transform (FFT)
  - Efficient computational algorithm for calculating the Discrete Fourier Transform (DFT)
  - The DFT is a mathematical formula for calculating the frequency content of a signal
- Any discrete-time signal of N points can be expressed as the sum of N sinusoids:



The Fast Fourier Transform

- Most common use of block-based processing: the Fast Fourier Transform (FFT)
  - Efficient computational algorithm for calculating the Discrete Fourier Transform (DFT)
  - The DFT is a mathematical formula for calculating the frequency content of a signal
- Any discrete-time signal of N points can be expressed as the sum of N sinusoids:

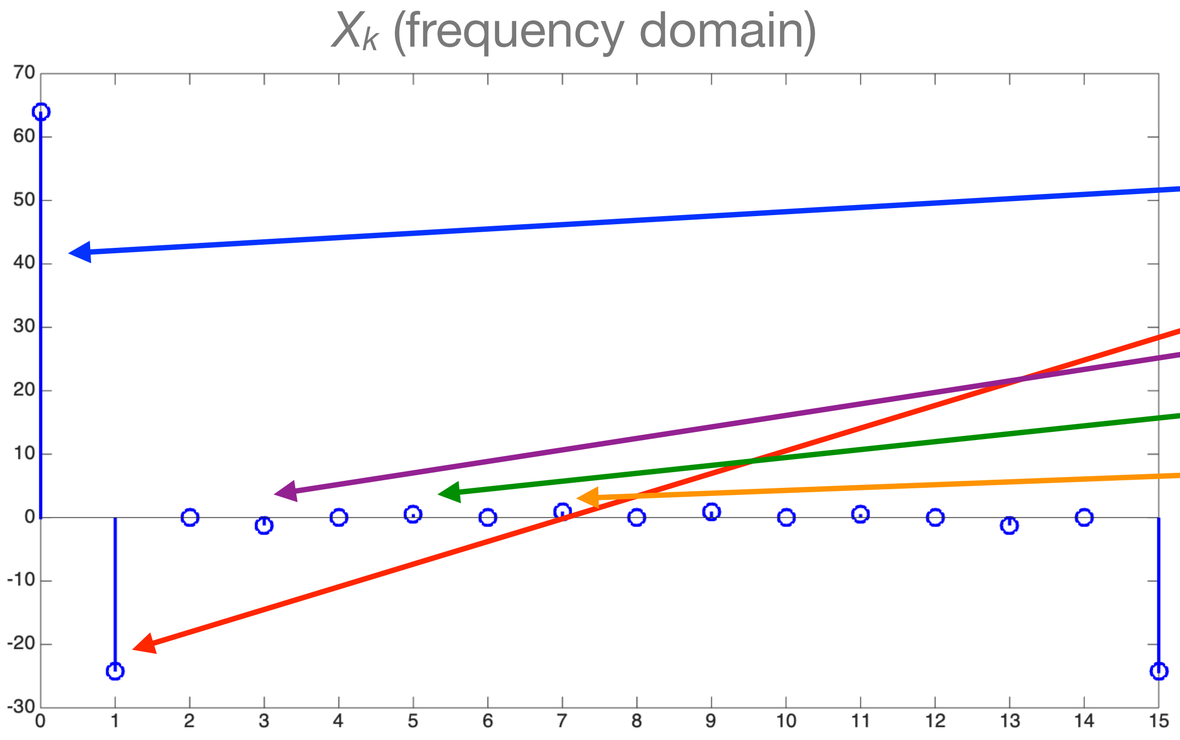


Frequency	Amplitude	Phase
0	4.00	0
π/8	1.64	-.875*π
3π/8	0.20	-.625*π
5π/8	0.09	-.375*π
7π/8	0.65	-.125*π

(other components are 0 or redundant)

The Fast Fourier Transform

- Most common use of block-based processing: the Fast Fourier Transform (FFT)
  - Efficient computational algorithm for calculating the Discrete Fourier Transform (DFT)
  - The DFT is a mathematical formula for calculating the frequency content of a signal
- Any discrete-time signal of N points can be expressed as the sum of N sinusoids:



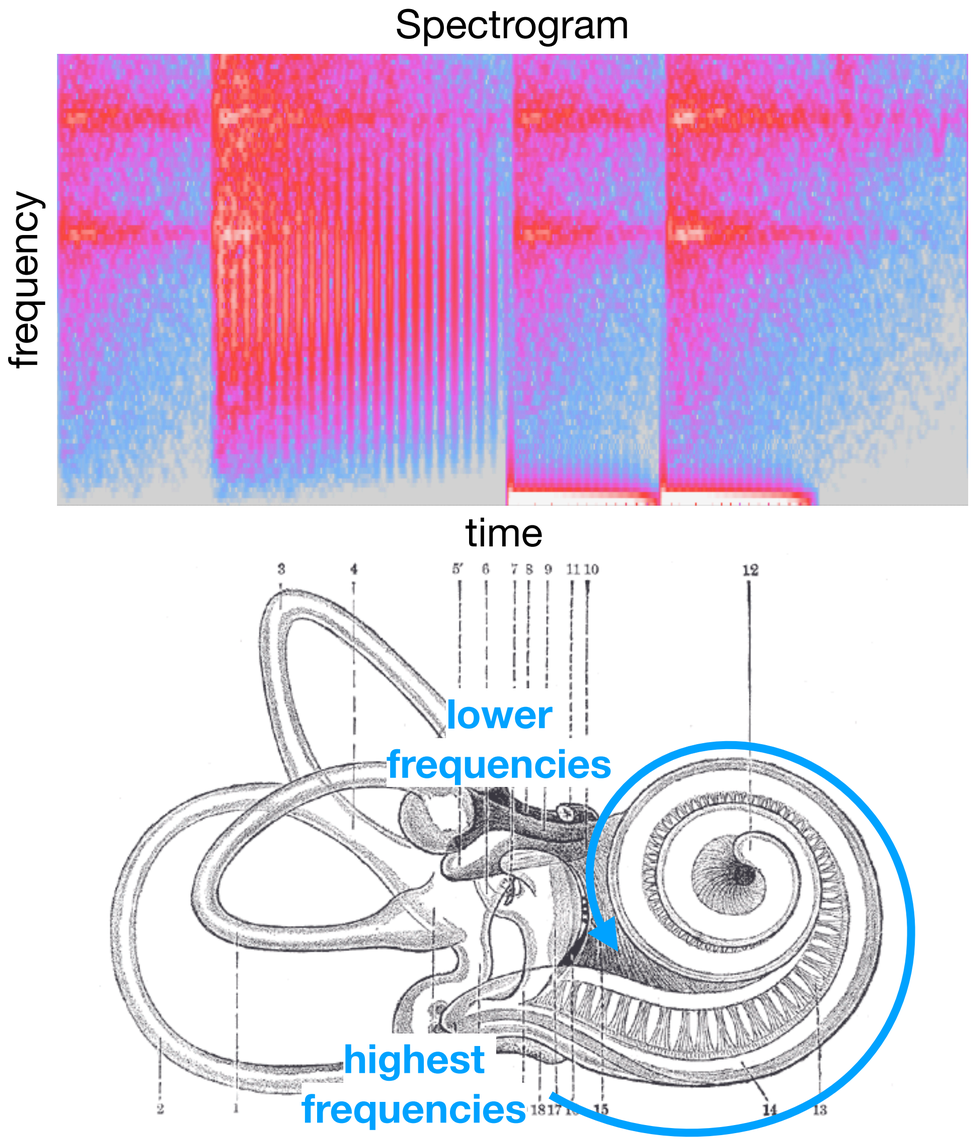
Frequency	Amplitude	Phase
0	4.00	0
π/8	1.64	-.875*π
3π/8	0.20	-.625*π
5π/8	0.09	-.375*π
7π/8	0.65	-.125*π

(other components are 0 or redundant)

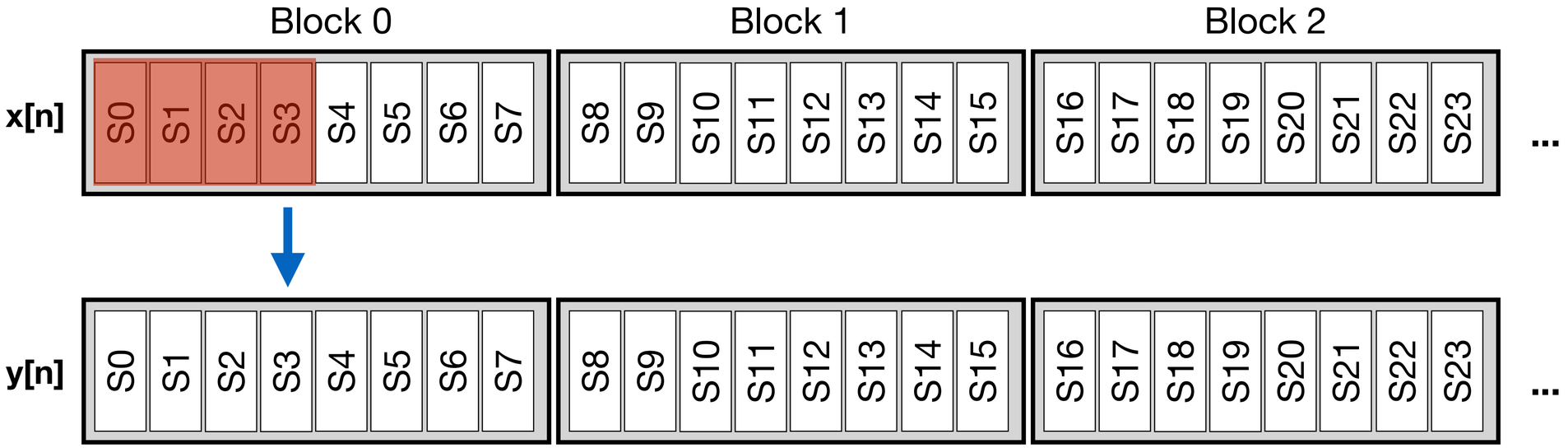
The DFT calculates the amplitudes and phases of each frequency component

Why do we care about the FFT?

- The FFT has transformed digital signal processing, including for audio
- Application 1: audio analysis
  - Calculate the spectrum of a signal (magnitude vs. frequency)
  - Or the spectrogram (how the spectrum changes over time)
  - Can extract higher-level audio features from the FFT
    - This is the basis of the field of music informatics
- Application 2: sonic transformation and resynthesis
  - Many useful audio effects are done in the frequency domain
    - Pitch shifting, denoising, cross-synthesis, ...
  - These are known as phase vocoder effects
- Interesting aside: the human cochlea (inner ear) acts like a kind of biological Fourier transform
  - Each location is sensitive to a different frequency



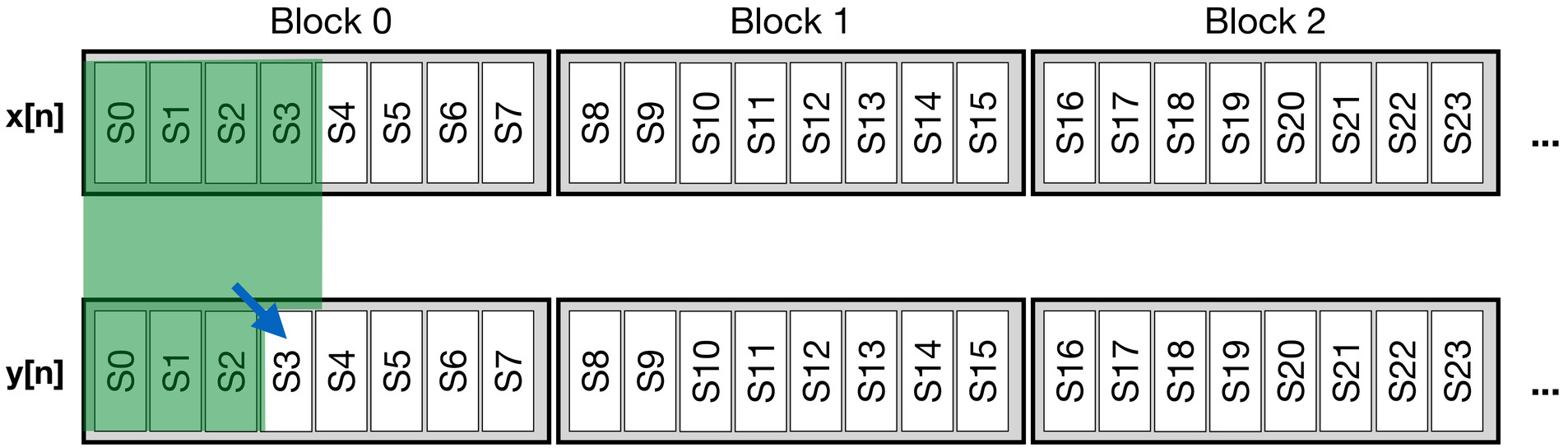
Filters vs. block-based processing



FIR Filter: new calculation each sample

- Important: Window size is not necessarily the system audio buffer size!
  - Might need multiple calls to `render()` to fill a single window
  - Or window might fill up in the middle of `render()`

Filters vs. block-based processing

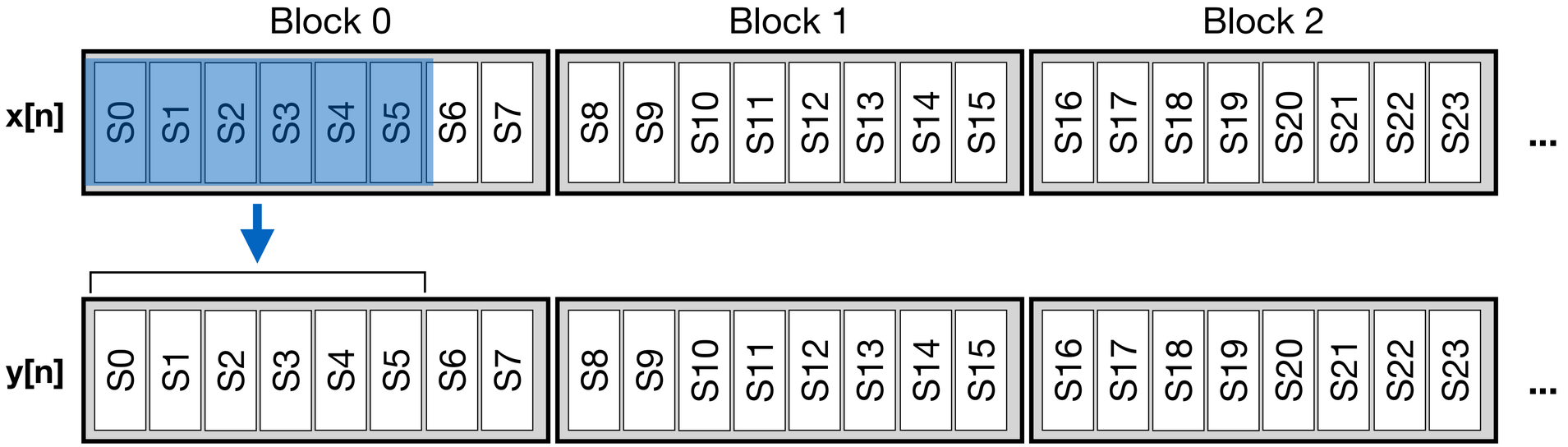


FIR Filter: new calculation each sample

IIR Filter: new calculation each sample

- Important: Window size is not necessarily the system audio buffer size!
  - Might need multiple calls to `render()` to fill a single window
  - Or window might fill up in the middle of `render()`

Filters vs. block-based processing



FIR Filter: new calculation each sample

IIR Filter: new calculation each sample

DFT: calculation once per block*

**note: size of DFT may be different than interval between calculations*

- Important: Window size is not necessarily the system audio buffer size!
  - Might need multiple calls to `render()` to fill a single window
  - Or window might fill up in the middle of `render()`

Real-time windowing for the FFT

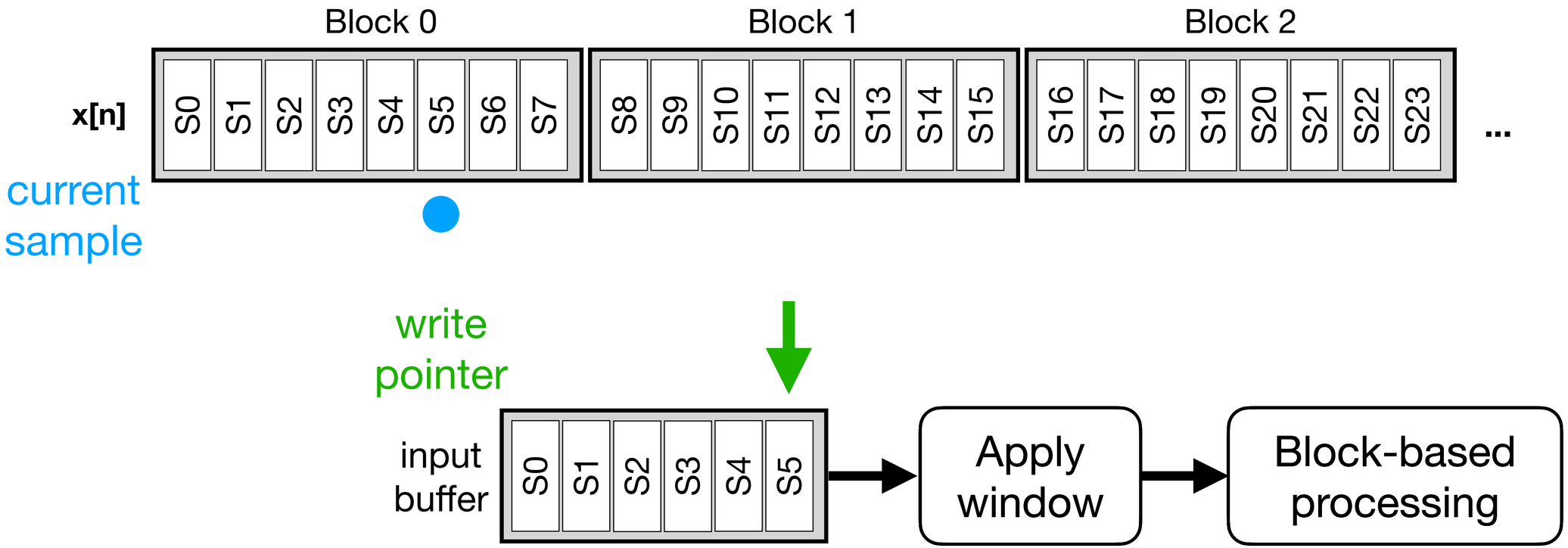
- We need a full window (i.e. block) of samples at once to pass to the FFT
- Suppose that we have a window size of N samples
- To handle FFT processing in real time:
  1. In our initialisation code, create a buffer (array) of type `float`, size N
     - If we have multiple audio channels, we'll need one buffer per channel
  2. In `render()`, as we process samples, store them in buffer and keep count
     - We need a global variable to keep track of how many samples are in the buffer
  3. When the buffer is full:
     - Multiply the contents of the buffer by the window function (if window is not rectangular)
     - Perform an FFT on the result
     - Handle any necessary processing in the frequency domain
  4. After the processing is complete, reset the counter to 0 to start filling the next window

Block-based input

- Again: window size is not necessarily the system buffer size
  - Therefore, we cannot just pass the raw audio buffer inside `BelaContext` to the FFT!
- Instead, we need:
  - Separate buffer to hold incoming samples
  - Counter or write pointer to keep track of how full the buffer is
  - Even though we're processing in discrete blocks, all the code still needs to go inside the `for()` loop in `render()`

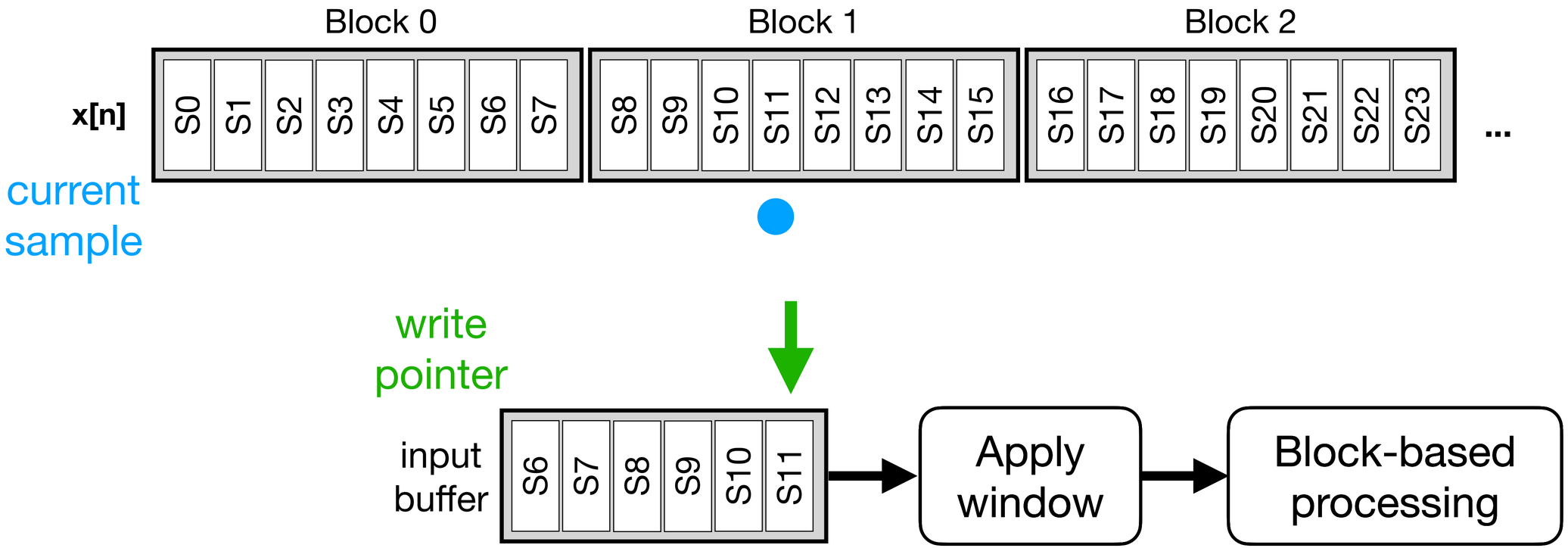
```
float gWindowBuffer[BUFFER_SIZE]; // Buffer for 1 window of samples
int gWindowBufferPointer = 0;     // Tells us how many samples in buffer

void render(BelaContext *context, void *userData) {
    // Important: nothing goes here; everything in the loop!

    for(int n = 0; n < context->audioFrames; n++) {
        // 1. Copy audio input into gWindowBuffer
        // 2. Increment gWindowBufferPointer
        // 3. Check if pointer has reached end of gWindowBuffer
        // 4. If yes, time to do the FFT and any other processing then reset pointer to 0
    }
}
```

Block-based input



- Start with a buffer of fixed size
  - Fill it up sample by sample
  - When it's full, apply the window function and pass it to the FFT (or other block based process)

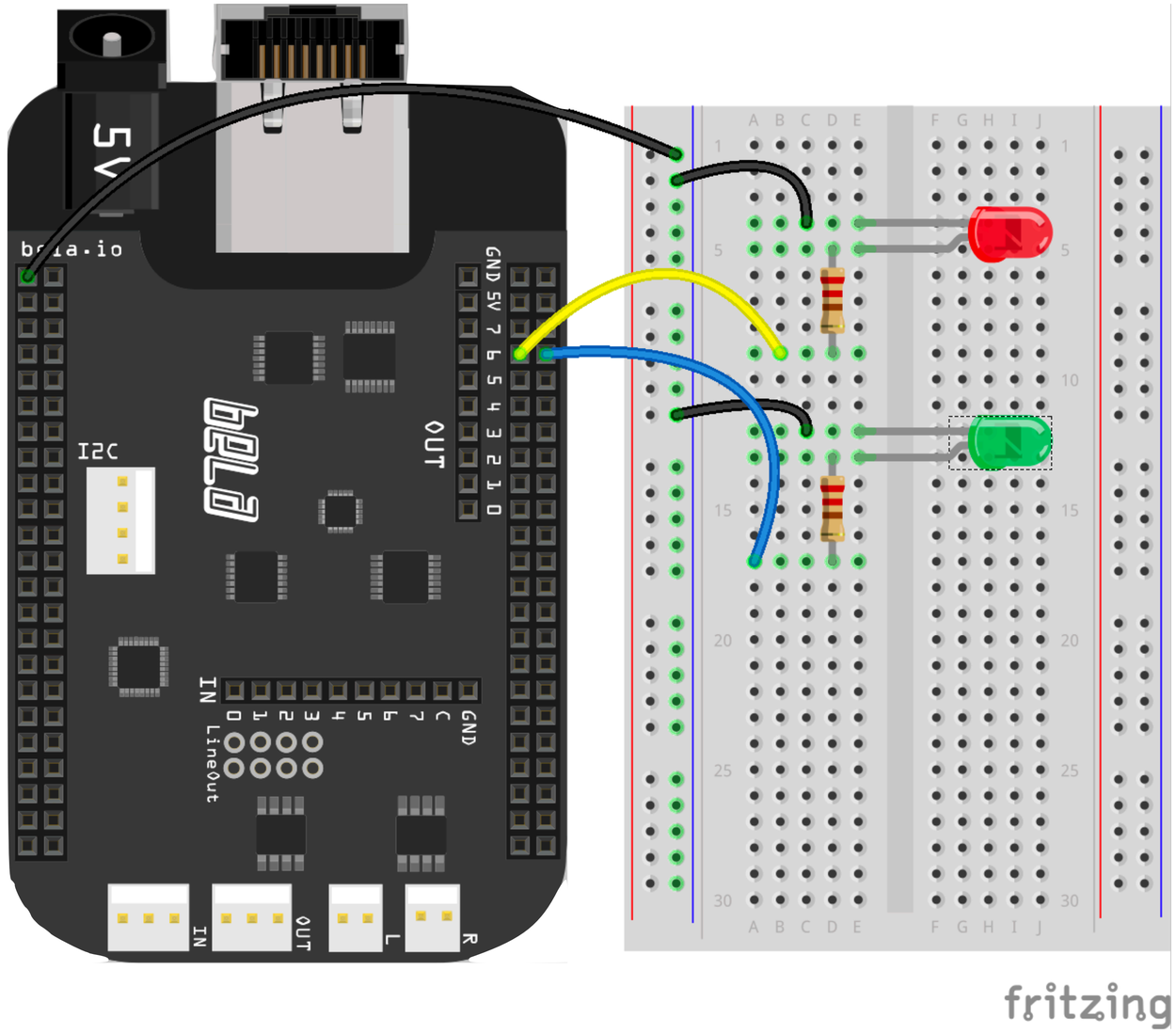
Block-based input



- Start with a buffer of fixed size
  - Fill it up sample by sample
  - When it's full, apply the window function and pass it to the FFT (or other block based process)

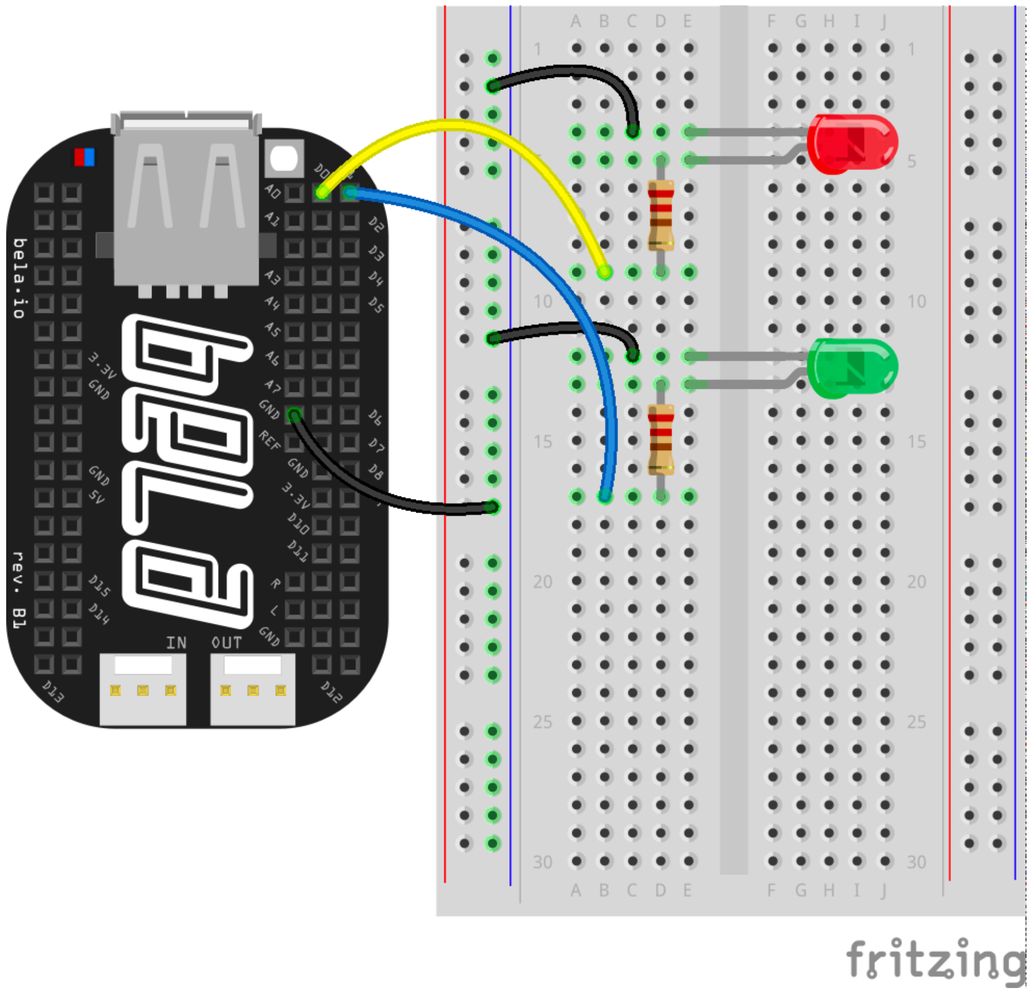
Block-based processing task

- Task: using fft-led project
  - Attach LEDs to digital outputs 0 and 1
  - Implement code to segment the incoming signal into windows
    - Rectangular windows are fine for now (no window function needed)



Block-based processing task

- Task: using fft-led project
  - Attach LEDs to digital outputs 0 and 1
  - Implement code to segment the incoming signal into windows
    - Rectangular windows are fine for now (no window function needed)



Block-based processing task

- Task: using fft-led project
  - Attach LEDs to digital outputs 0 and 1
  - Implement code to segment the incoming signal into windows
    - Rectangular windows are fine for now (no window function needed)
- Steps:
  1. Declare a buffer of size `gFftSize` as a global variable
     - Could use either an array or a `vector` of type `float`
  2. Keep a counter (also global) to record how many samples have been put in the buffer
     - Can also use this as a write pointer into the buffer
  3. Each iteration of the `for()` loop in `render()`, store the current sample in the buffer
     - When the counter reaches `gFftSize`, call the function `process_fft()`
  4. When finished with `process_fft()`, reset the counter and start again

The Bela GUI library

- Bela projects can have a GUI that renders in the browser
  - The GUI is typically written in **p5.js** (but other frameworks can be used)
  - Data can be sent from the Bela program to the browser via a WebSocket connection
  - Can also send data the other way, to control a Bela program from the GUI

The simple GUI from the `fft-led` project

- How to communicate with a p5.js GUI from your Bela C++ code:
  1. Include the GUI library: `#include <libraries/Gui/Gui.h>`
  2. Make an object of type `Gui`: `Gui gLedGui;`
  3. In `setup()`, initialise the `Gui` object: `gLedGui.setup(context->projectName);`
  4. Use the `Gui::sendBuffer()` method to send data to the GUI

The Bela GUI library: communication

- Sending data from C++ on Bela to JavaScript in the browser:

render.cpp

```
// Update the Bela GUI at a 100Hz frame rate
if(++gGuiUpdateCounter >= context->audioSampleRate/100) {
    gGuiUpdateCounter = 0;

    // Send low and high values in two slots of the buffer
    gGuiBuffer[0] = gLedLoOutput;
    gGuiBuffer[1] = gLedHiOutput;
    if(gLedGui.isConnected()) {
        gLedGui.sendBuffer(0, gGuiBuffer);
    }
}
```

limit the rate that we send to the GUI by counting samples

store all the data in a single buffer (here, array of `float`)

check if GUI running

first argument is buffer number (allows multiple pieces of data)

second argument can be any type, array or vector

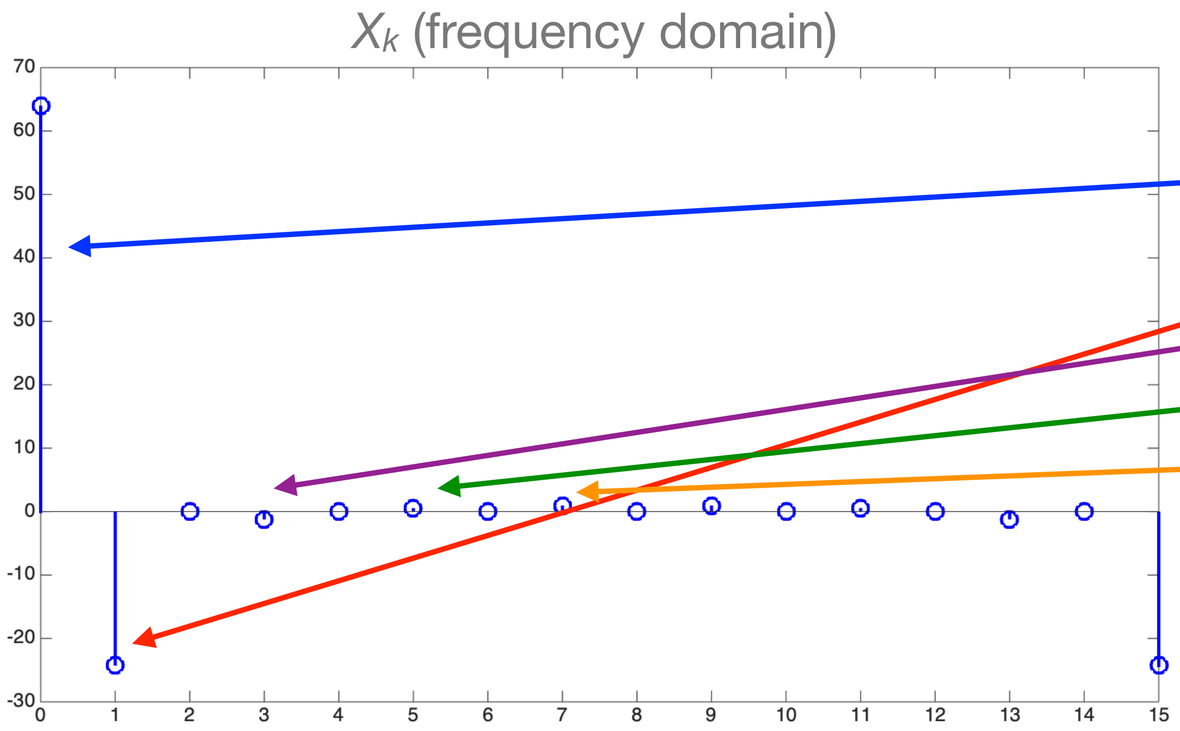
sketch.js

```
// Retrieve the data being sent from render.cpp
let ledLow = Bela.data.buffers[0][0];
let ledHigh = Bela.data.buffers[0][1];
```

this holds the contents of `gGuiBuffer` on the C++ side (because we sent it as buffer number 0)

The Fast Fourier Transform

- Most common use of block-based processing: the Fast Fourier Transform (FFT)
  - Efficient computational algorithm for calculating the Discrete Fourier Transform (DFT)
  - The DFT is a mathematical formula for calculating the frequency content of a signal
- Any discrete-time signal of N points can be expressed as the sum of N sinusoids:



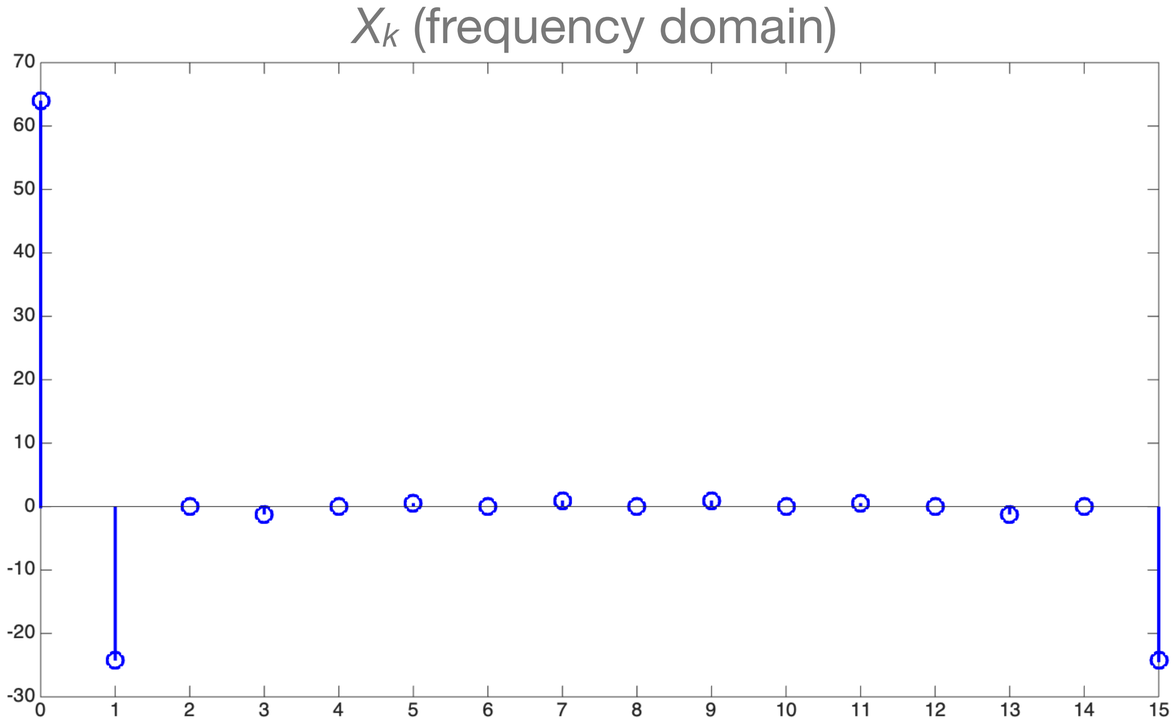
Frequency	Amplitude	Phase
0	4.00	0
π/8	1.64	-.875*π
3π/8	0.20	-.625*π
5π/8	0.09	-.375*π
7π/8	0.65	-.125*π

(other components are 0 or redundant)

The DFT calculates the amplitudes and phases of each frequency component

The Discrete Fourier Transform

- The DFT calculates the amplitudes and phases of each sinusoidal component
  - The frequencies are linearly spaced between 0 and 2π: $\frac{2\pi k}{N}$ for $0 \leq k < N$
  - Each amplitude/phase measurement is called a bin
  - Collectively, the N bins are called the frequency domain representation of the signal



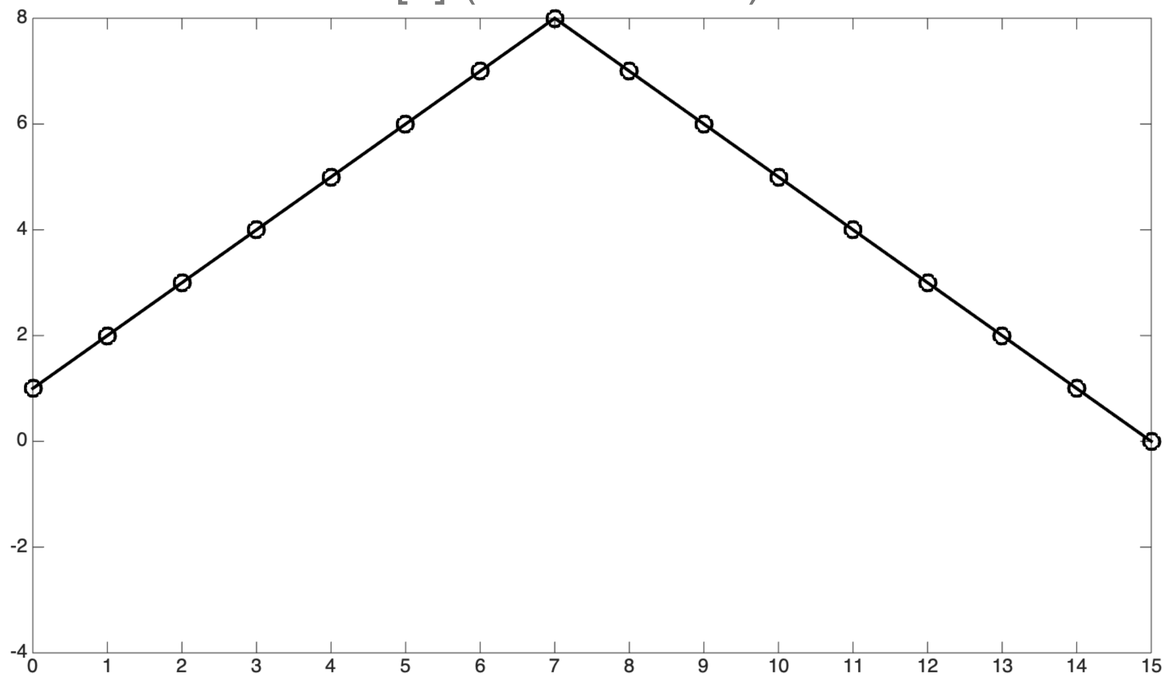
Frequency	Amplitude	Phase
0	4.00	0
π/8	1.64	-.875*π
3π/8	0.20	-.625*π
5π/8	0.09	-.375*π
7π/8	0.65	-.125*π

(other components are 0 or redundant)

The Discrete Fourier Transform

- Key point: N time samples ↔ N frequency samples
  - Two different mathematical perspectives on the same signal
  - The Inverse Discrete Fourier Transform (IDFT) converts from frequency domain to time domain
  - The DFT+IDFT is an exact reconstruction as long as signal length $M \leq$ DFT length N

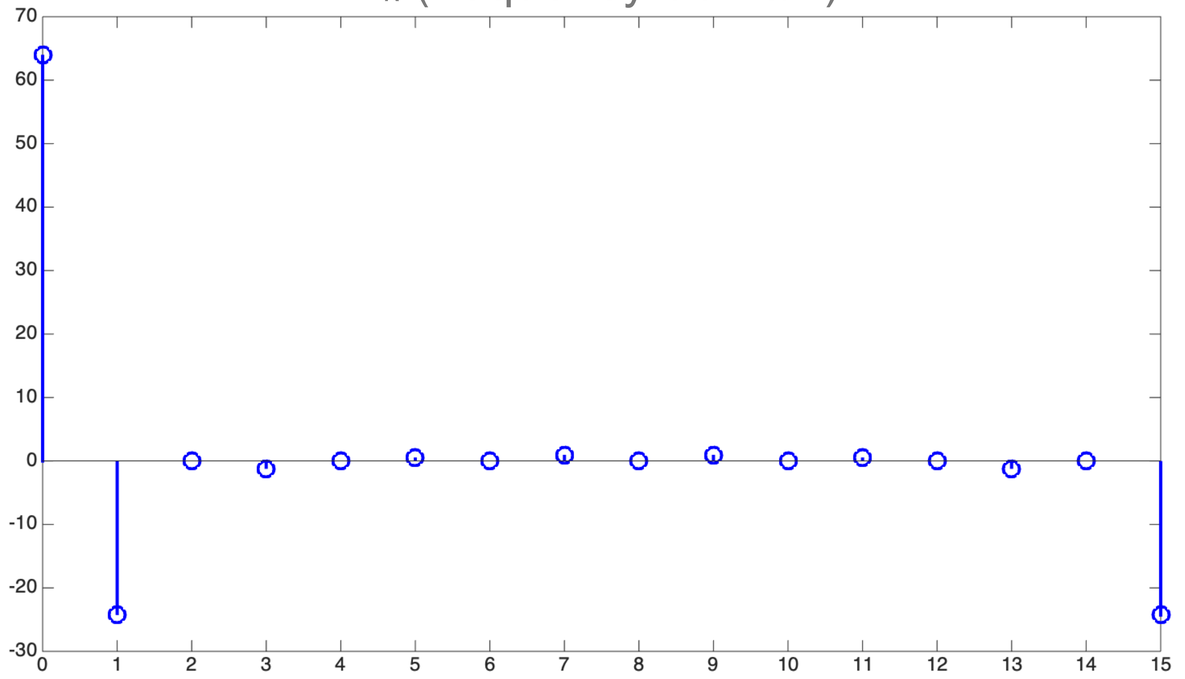
$x[n]$ (time domain)



DFT

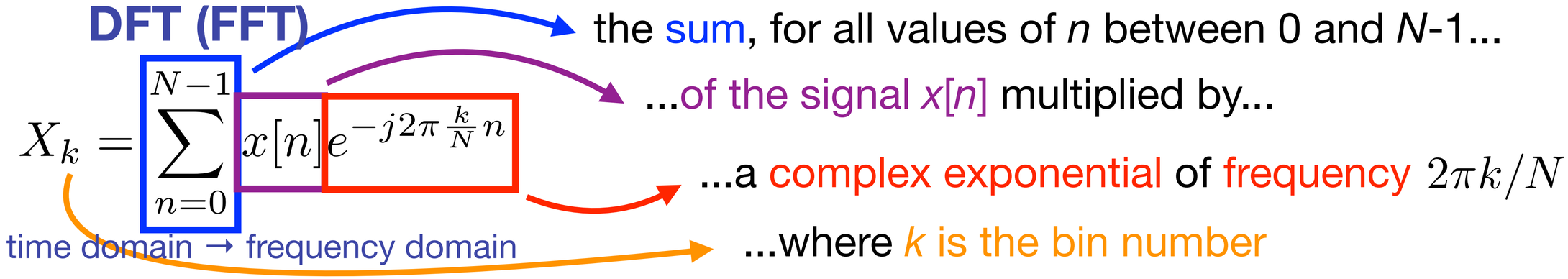
IDFT

X_k (frequency domain)



The Discrete Fourier Transform

- The mathematical basis of the Discrete Fourier Transform:



- N is the size of the DFT
  - For the FFT, N is almost always a power of 2 (the Cooley-Tukey algorithm)
- Each time sample *x*[*n*] is a real number
- Each frequency bin X_k is a complex number: $a + bj$ where $j = \sqrt{-1}$
  - Magnitude of the bin is given by: $\sqrt{a^2 + b^2}$
    - We usually use magnitude to plot a spectrum or spectrogram of a signal
  - Phase of the bin is given by: $\arctan(b/a)$

The Discrete Fourier Transform

- The mathematical basis of the Discrete Fourier Transform:

DFT (FFT)

$$X_k = \sum_{n=0}^{N-1} x[n] e^{-j2\pi \frac{k}{N} n}$$

time domain → frequency domain

IDFT (IFFT)

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{+j2\pi \frac{k}{N} n}$$

frequency domain → time domain

- N is the size of the DFT
  - For the FFT, N is almost always a power of 2 (the Cooley-Tukey algorithm)
- Each time sample *x*[*n*] is a real number
- Each frequency bin X_k is a complex number: $a + bj$ where $j = \sqrt{-1}$
  - Magnitude of the bin is given by: $\sqrt{a^2 + b^2}$
    - We usually use magnitude to plot a spectrum or spectrogram of a signal
  - Phase of the bin is given by: $\arctan(b/a)$

The Discrete Fourier Transform

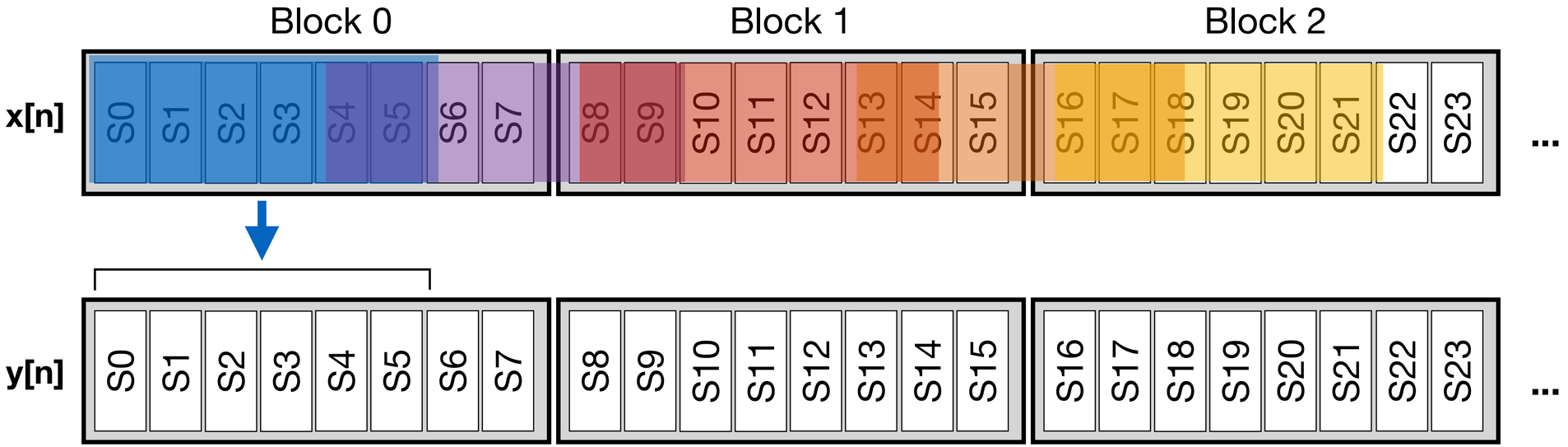
- Frequency bins are equally spaced between 0 and 2π:
  - Where, for sampled analog signals, 2π corresponds to the sample rate

$$X_k = \sum_{n=0}^{N-1} x[n] e^{-j2\pi \frac{k}{N} n} \qquad x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{+j2\pi \frac{k}{N} n}$$

$0 \leq k < N$

- But wait! Doesn't a frequency of π represent the Nyquist rate?
  - Why are there bins with frequency higher than π?
- The higher bins don't contain any unique information in our case:
  - The DFT works generally with complex signals, but audio signals are always real-valued
  - With a real signal, the frequency bins are conjugate symmetric: $X_k = X^*_{N-k}$
- Therefore, we only care about bins $0 \leq k \leq N/2$
  - This is why you'll see audio spectrum plots stop at the Nyquist frequency

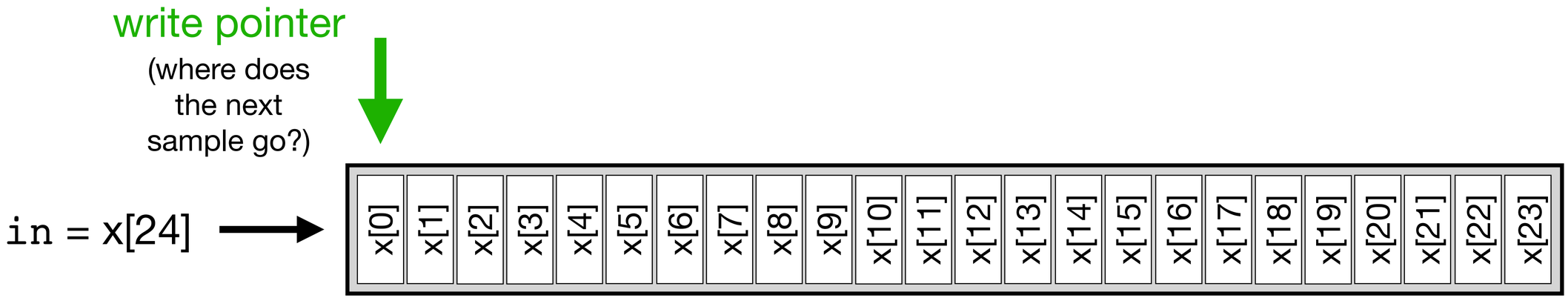
Overlapping blocks in real time



- As we'll see next lecture, we often want to work with overlapping blocks
  - This means block *k* shares some samples with block *k*-1
  - How do we adapt our windowing code to handle this?
- Easiest approach: keep a running history of the input samples
  - i.e. a buffer which always has the last M samples
  - What kind of structure have seen that does this? Circular buffer
  - At each hop, pass M samples from circular buffer to FFT

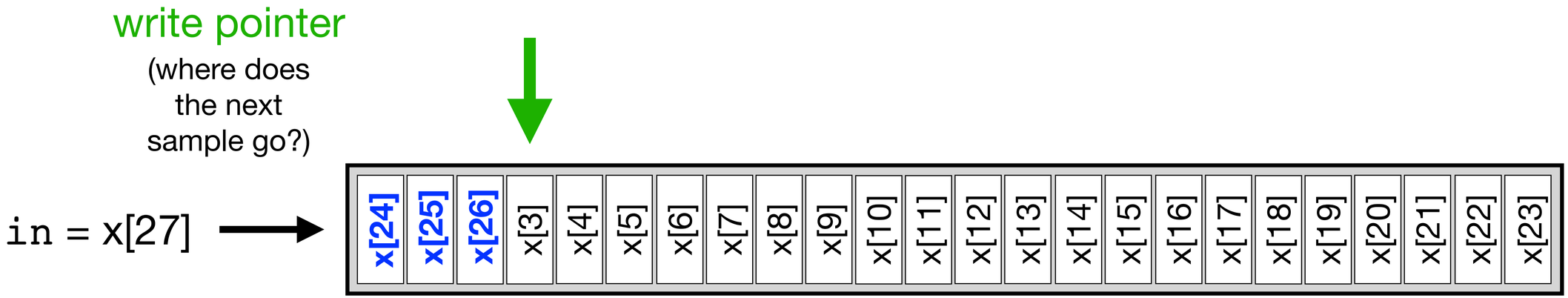
Review: circular buffering

- A circular buffer is a memory buffer (array) that acts like a loop
  - Write each new sample in the next location, from beginning to end
  - When we get to the end, go back to the beginning again
  - Keep track of the write pointer, which tells us which slot we write to next
- The buffer always ends up holding the N most recent samples
  - We just need to keep track of which sample is held where



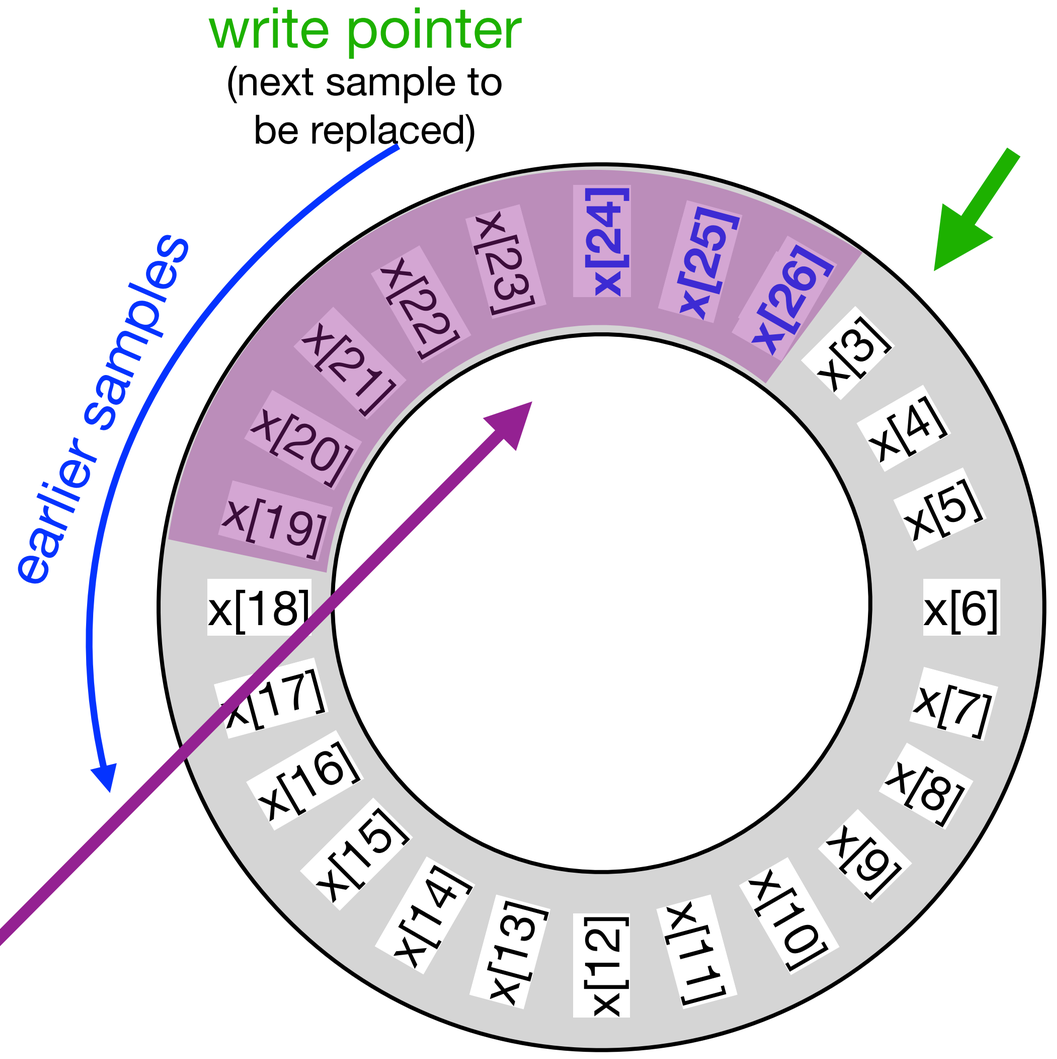
Review: circular buffering

- A circular buffer is a memory buffer (array) that acts like a loop
  - Write each new sample in the next location, from beginning to end
  - When we get to the end, go back to the beginning again
  - Keep track of the write pointer, which tells us which slot we write to next
- The buffer always ends up holding the N most recent samples
  - We just need to keep track of which sample is held where



Circular buffering and windowing

- Another equivalent view:
- The write pointer tells us where to find the front of the buffer
  - Points just past the most recent sample
    - i.e. it's the oldest sample in the buffer until it's replaced
  - To find earlier samples, look backward from the write pointer
- At any given time:
  - The buffer holds the N most recent samples
  - Each individual sample never moves until it is eventually replaced
  - As long as our buffer is big enough, we can always get the latest window of the signal



Circular buffer write pointer

- The circular buffer has two components:
  1. A region in memory (array) to store the samples
  2. A write pointer to keep track of where we are
- Remember, there is no functional beginning or end to a circular buffer!
- In code, we need to declare two (global) variables:

```
float gLastSamples[gBufferSize] = {0};
int gWritePointer = 0;
```

  - When we have a new sample, store it at the write pointer, then increment the pointer

```
gLastSamples[gWritePointer] = in;
gWritePointer++;
```

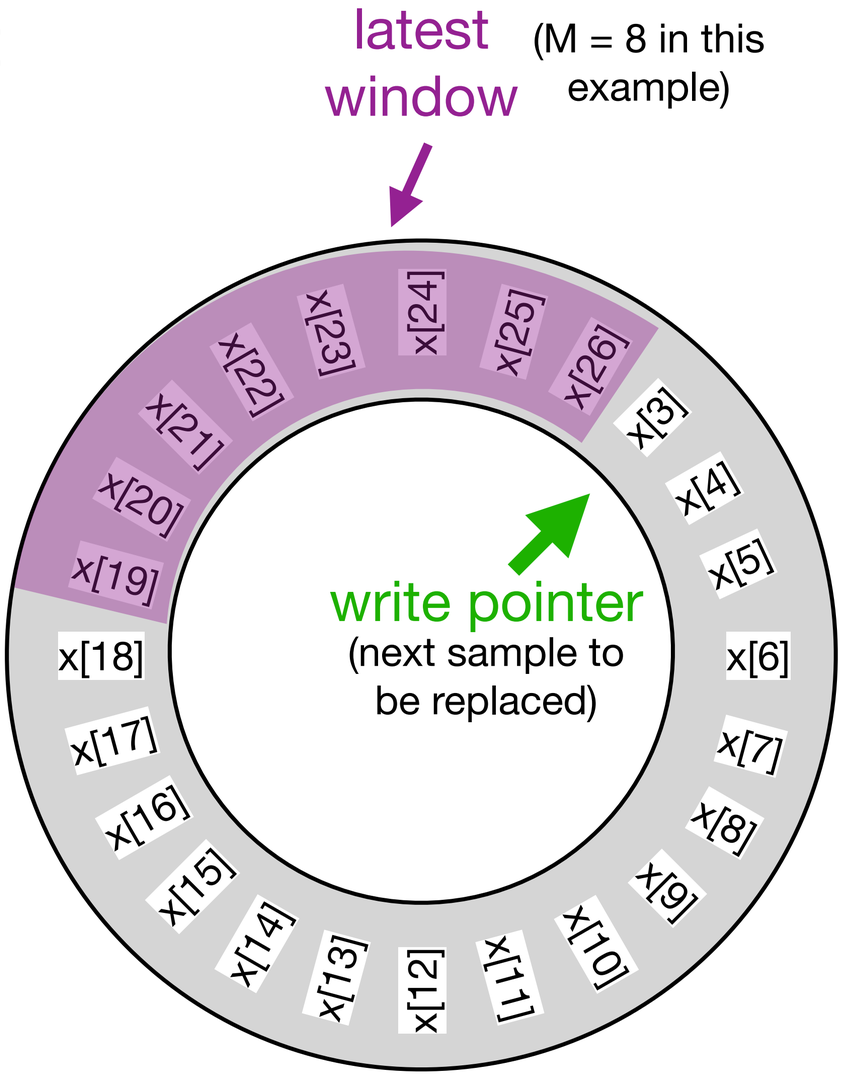
  - What else do we need to do?
    - Keep the write pointer in range

```
if(gWritePointer >= gBufferSize)
    gWritePointer = 0;
```

Overlap-add with a circular buffer

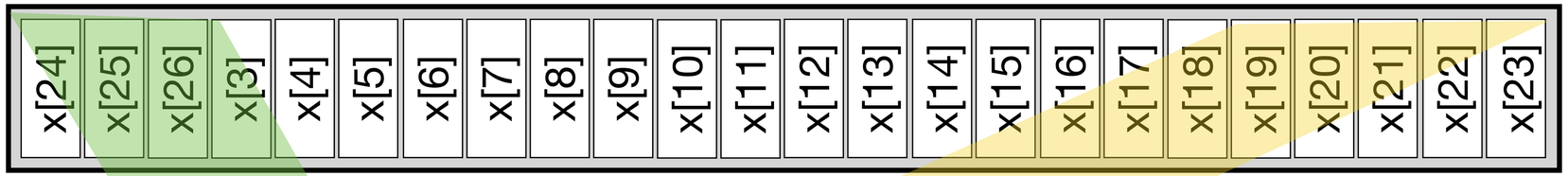
- We use a circular buffer to keep track of the input
- Necessary characteristics:
  - The buffer always holds (at least) the last M input samples (where M is the window size)
  - If using multiple channels, one buffer per channel
- Using the circular buffer:
  - Each iteration of the `for()` loop:
    - store input in buffer
    - increment the write pointer (wrapping as necessary)
    - increment the total count of samples stored
  - When the count reaches the hop size:
    - take one window from buffer, unwrap it, and pass it to FFT
    - in other words: copy it to a new buffer such that the oldest sample appears at index 0 of the new buffer

Important: this is not the same as when the write pointer wraps around!

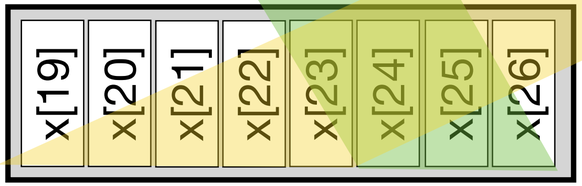


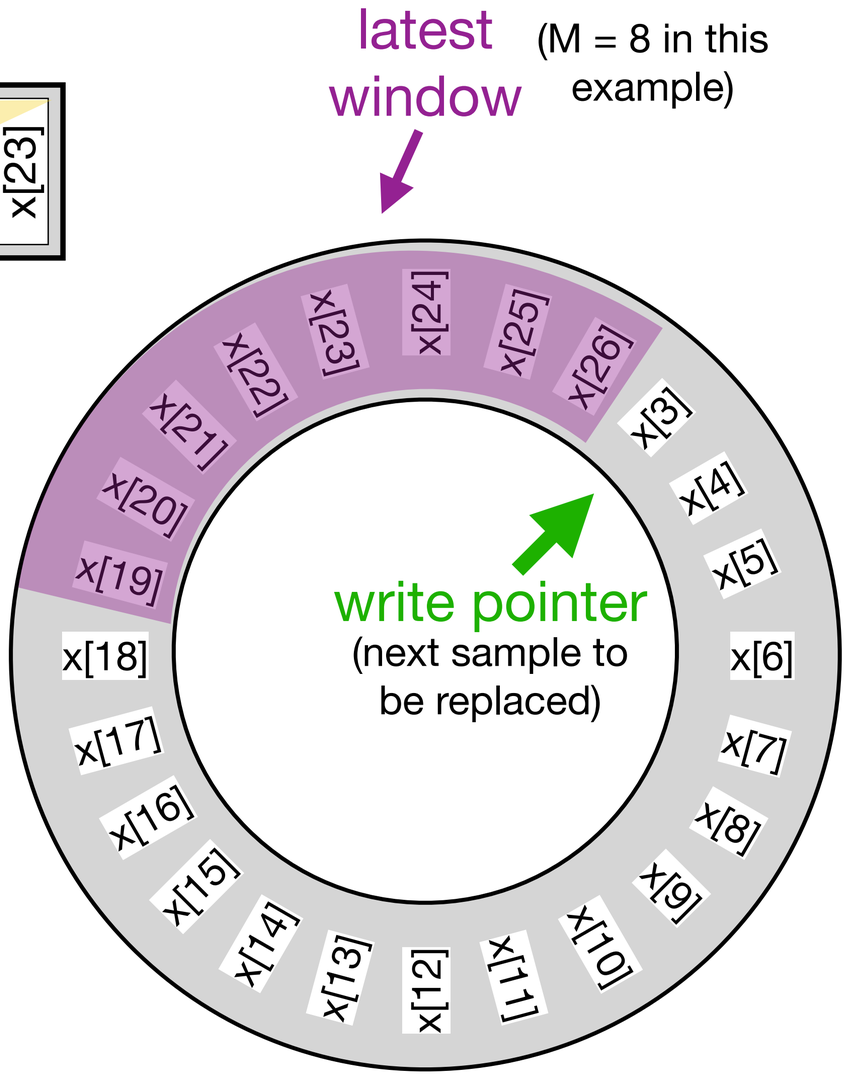
Unwrapping the circular buffer

How the circular buffer is actually stored in memory:



What we need to calculate the FFT (the latest 8 samples):





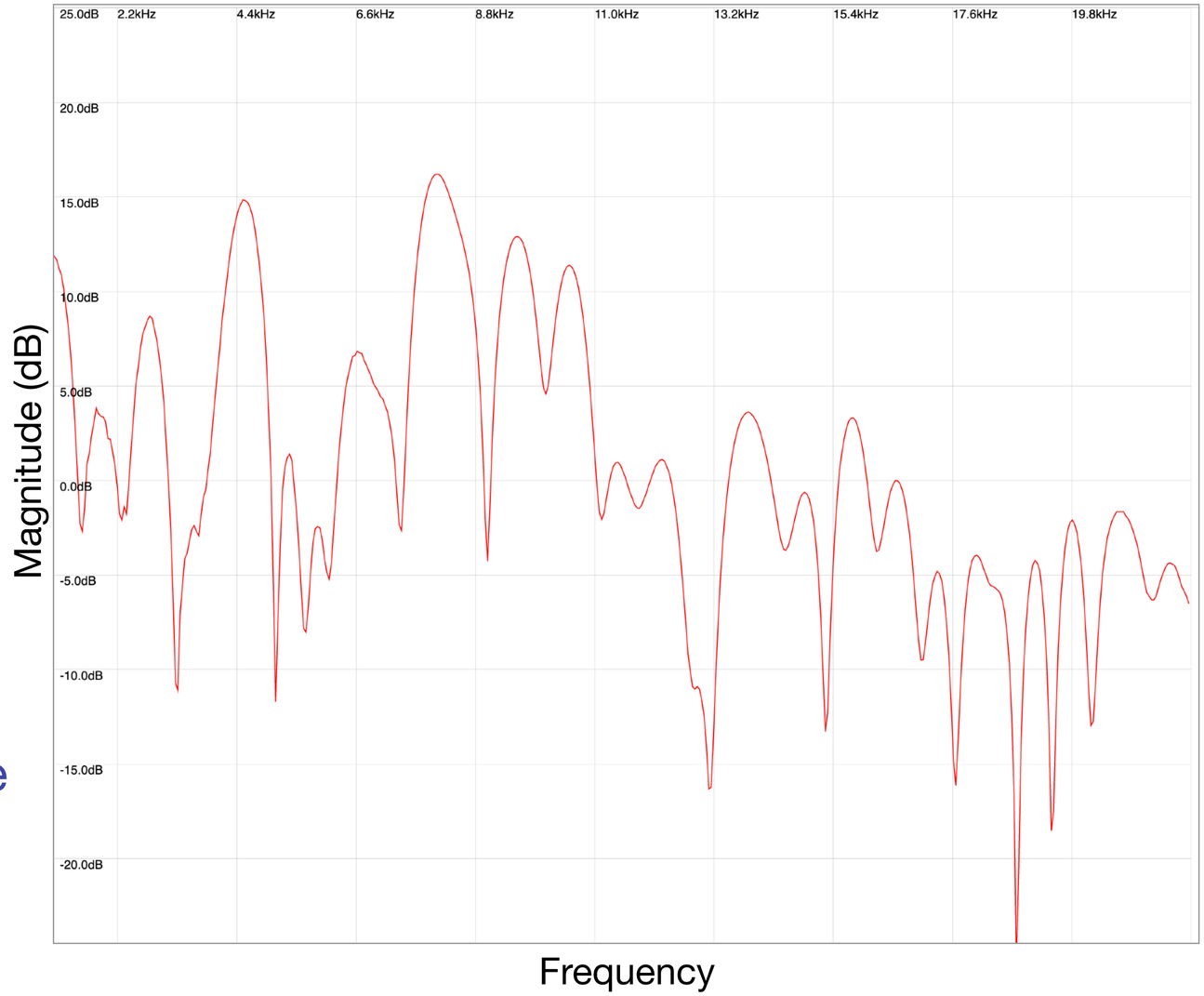
In other words: having assembled the samples into a circular buffer, on each hop it is helpful to copy them into a linear (standard) buffer to pass to the FFT library

Block-based processing: input summary

- Block-based processes are calculated at regular intervals, not every frame
  - Unlike filters, where the output is updated every frame
  - The block-based calculation typically involves a recent window of the signal
- Two relevant quantities:
  - Window size (block size): how long of a segment of the signal is used in the calculation?
  - Hop size: at what interval is the calculation performed?
    - The hop size will typically be no larger than the window size, and is commonly a fraction of window size
  - Neither of these quantities is the same as the real-time audio system buffer size!
    - Your block-based code should work identically regardless of audio buffer size
- We can use a circular buffer to keep track of a recent window of samples
  - Needs the usual array and write pointer
- We also keep a counter of how many frames have elapsed
  - Different from the write pointer of the circular buffer
  - When the counter reaches the hop size, we perform a new calculation

Plotting the audio spectrum

- The bins of the DFT can be used to plot the audio spectrum
  - Frequency on the X axis, magnitude on the Y axis (remember, bins are complex)
  - The Y axis is usually logarithmic (decibel scale)
  - Sometimes the X axis is logarithmic too
- We only plot the first half of the bins
  - The higher bins are the mirror image of the lower ones (conjugate symmetry)
  - For an N-sample DFT, bin N/2 corresponds to a frequency of π, which is the Nyquist rate
- The spectrum is typically a local snapshot, based on a window of signal

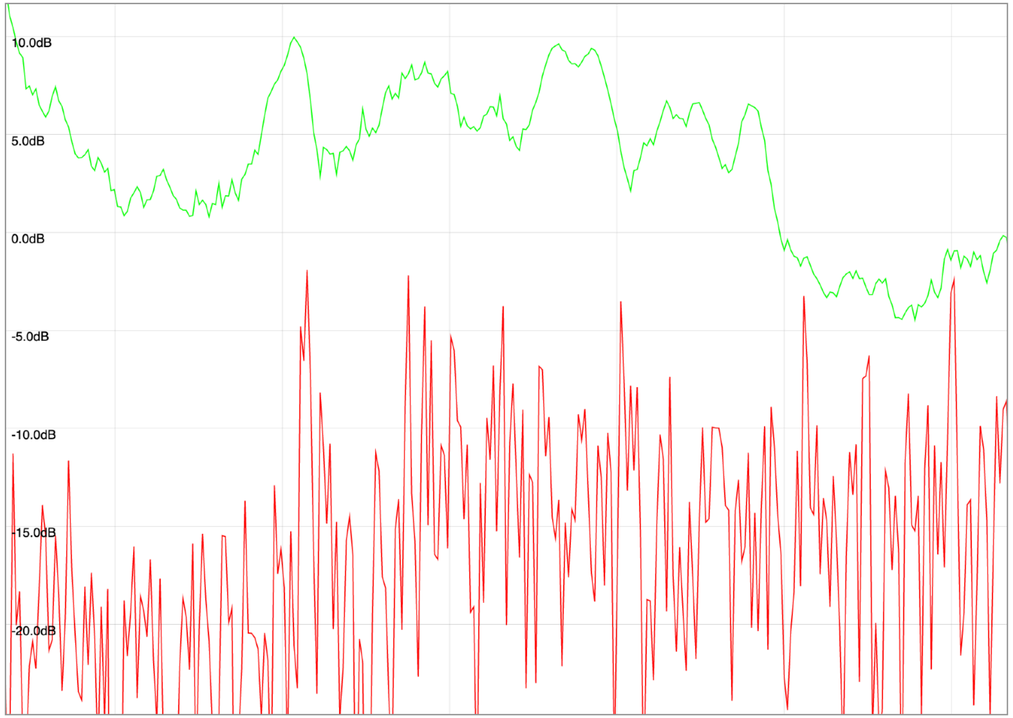


Circular buffer task

- Task: using fft-circular-buffer project (copying in drumloop.wav)
  - Implement the circular buffer in `render()`
  - See variables `gInputBuffer` and `gInputBufferPointer`
    - You still need both of these, and they work similarly to the last example, except...
    - `gInputBufferPointer` should not reset to 0 on each hop
      (it should wrap around whenever it hits the end of the buffer)
  - You need to use the additional global variable `gHopCounter`
    - Use this to keep track of how many samples have gone by,
      rather than using `gInputBufferPointer` to do this
    - When `gHopCounter` reaches the hop size, call `process_fft()` and reset the counter to 0
  - Implement the circular buffer in `render()`
  - Unwrap the buffer in `process_fft()`
    - See the code at the top of the function where the data is copied into the FFT data structure
    - Use modulo indexing here to copy the right samples

Peak detection

- Task 2: add peak detection to the plot
  - We'll calculate a second spectrum which stores the most recent peak value for each bin
  - Need to declare a persistent (global or static) array
    - This will remember the peak value for each frequency bin
    - Each hop, iterate through the bins:
      if the current value is greater than the peak,
      then update the peak
    - Multiply the peak by a decay factor (try 0.98) so the peaks decay over time
  - Send the peak spectrum to the GUI with buffer ID 1



- Task 3: also add smoothing to the peak spectrum
  - Calculate a local average of the nearest 11 bins (from bin *k*-5 to *k*+5)
  - Be careful not to run off the end of the array!

Keep in touch!

Social media:

@BelaPlatform

forum.bela.io

blog.bela.io

More resources and contact info at:

learn.bela.io/resources