# 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 | Filters |
| Analog and digital I/O | Control voltages |
| Filtering | Gates and triggers |
| 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

# Lecture 4: Parameter control

## What you'll learn today:

Using the Bela GUI to control projects from the browser
Linear and logarithmic scales

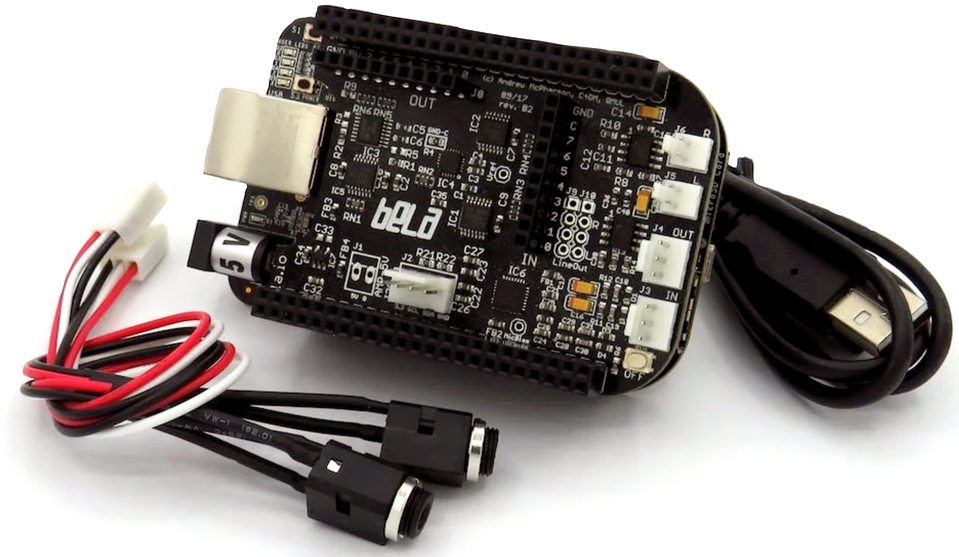
## What you'll make today:

Multiple oscillators with detuning

## Companion materials:

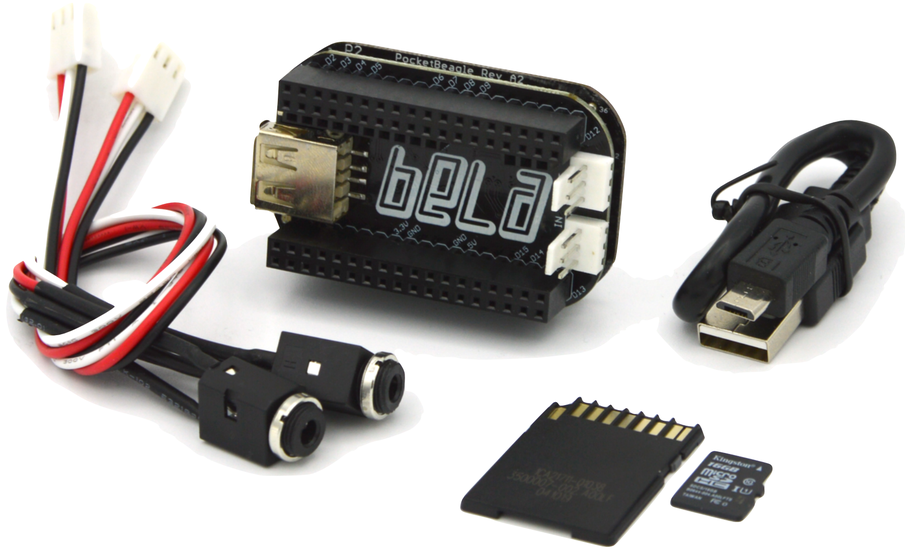
**github.com/BelaPlatform/bela-online-course**

# What you'll need



Bela Starter Kit

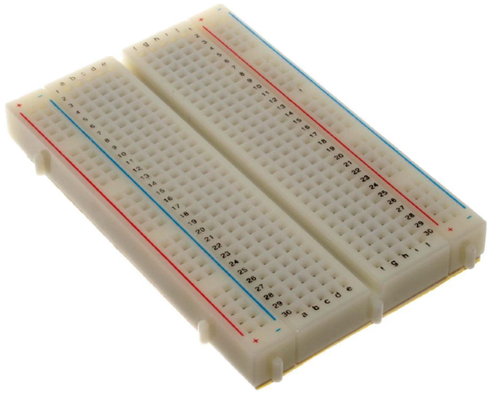
or

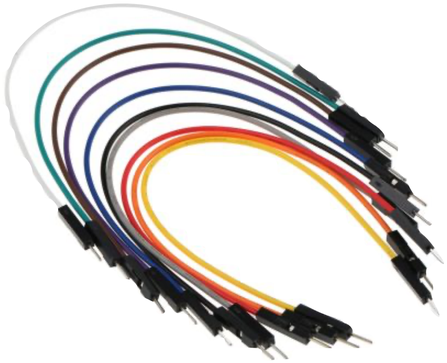


Bela Mini Starter Kit

[shop.bela.io]

Recommended for some lectures:

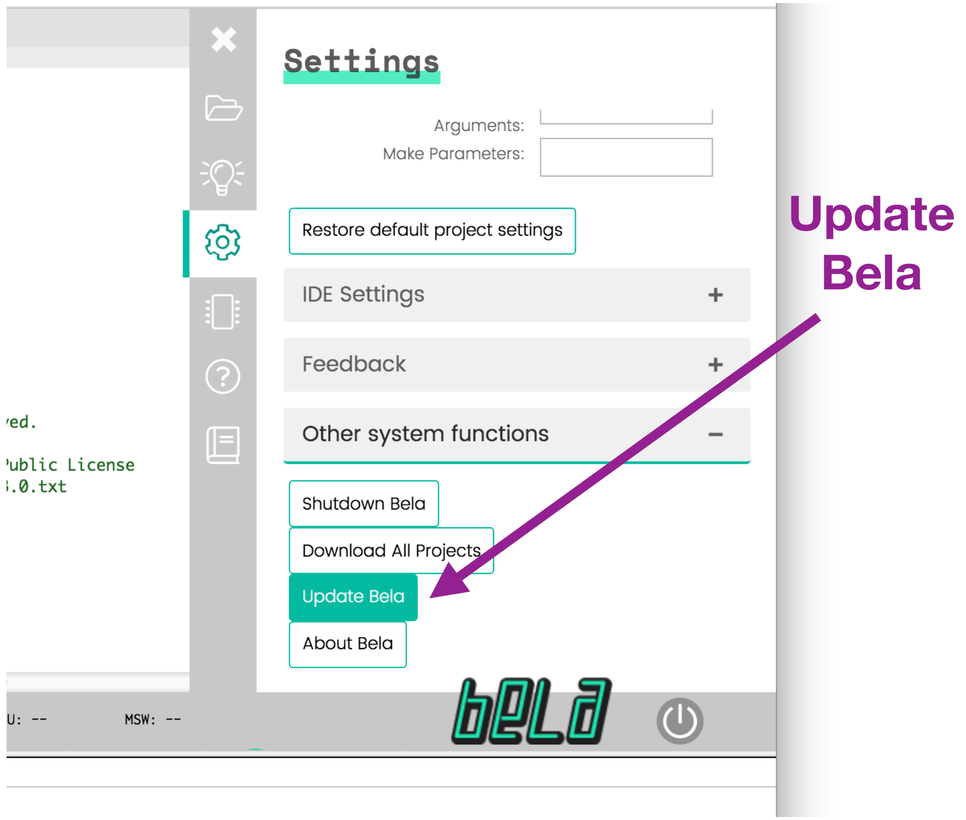




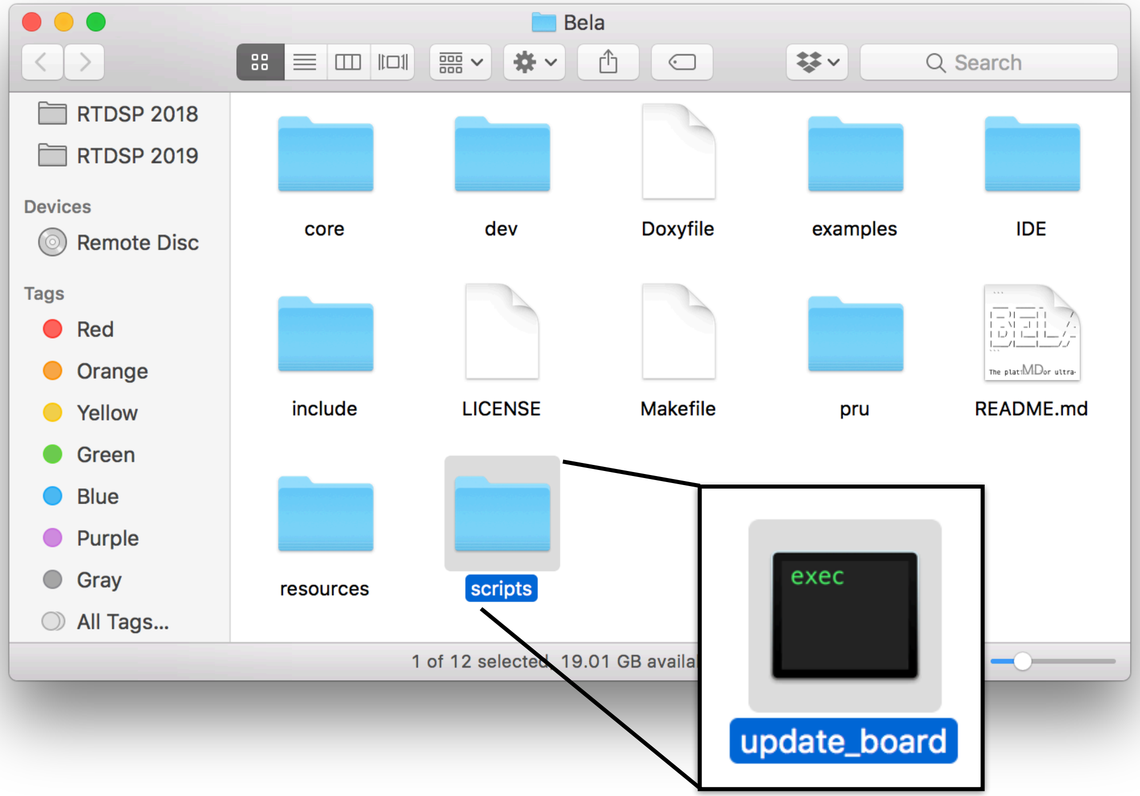
# Updating Bela

- Download the latest Bela core software from: **learn.bela.io/update**
- Two ways to update the board:

**By IDE** (recommended)

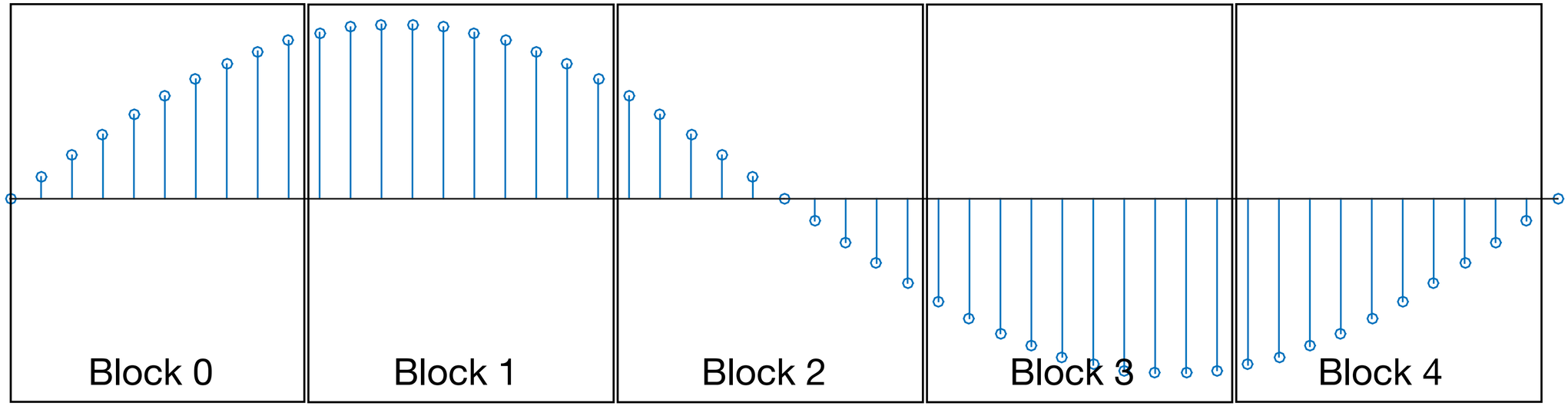



**By Script** (Mac and Linux)
unzip the archive



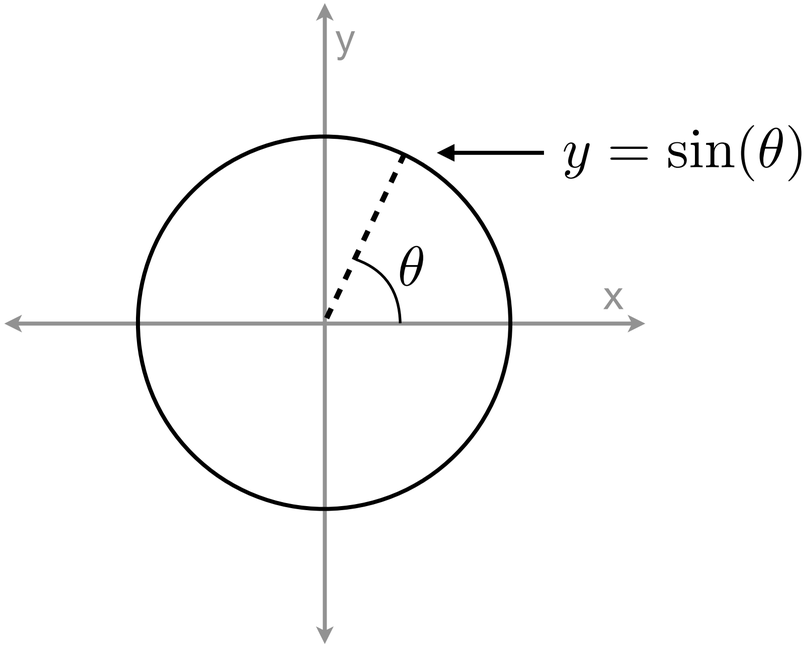
# Real-time sine oscillator

- In Lecture 1, we made a real-time sine oscillator that generated one block at a time:



# Oscillator frequency and phase

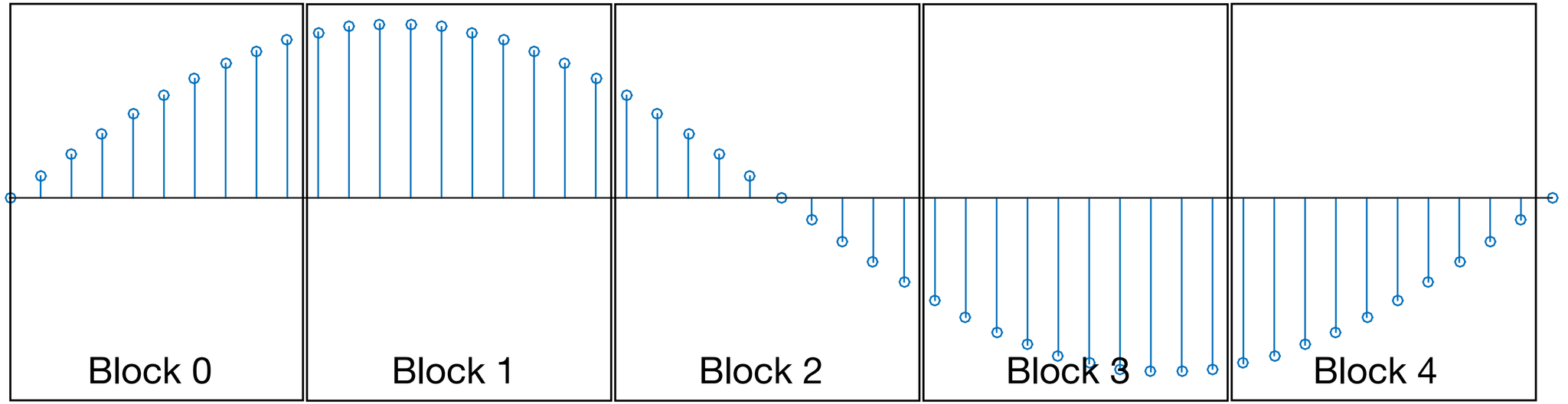
- The argument to the $\sin()$ function is known as the phase
  - Possible range of (unique) values: 0 to 2π
  - Analogous to the y-coordinate on the unit circle for a particular angle




- What does frequency correspond to in this analogy?
  - How fast we spin around the circle: the derivative of the phase

# Real-time sine oscillator

- In Lecture 1, we made a real-time sine oscillator that generated one block at a time:

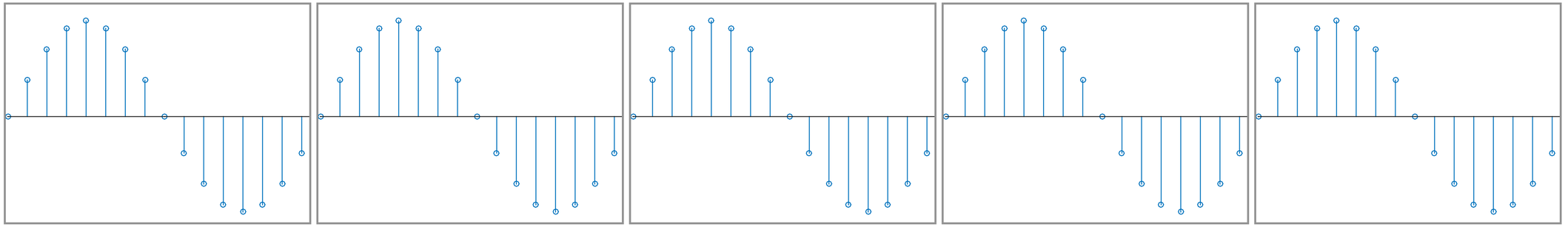



- We had to remember the phase between calls to `render()`
- The phase updated at each frame based on the frequency of the oscillator:

$$\phi = \phi + 2\pi f / f_s$$

# Wavetable oscillators

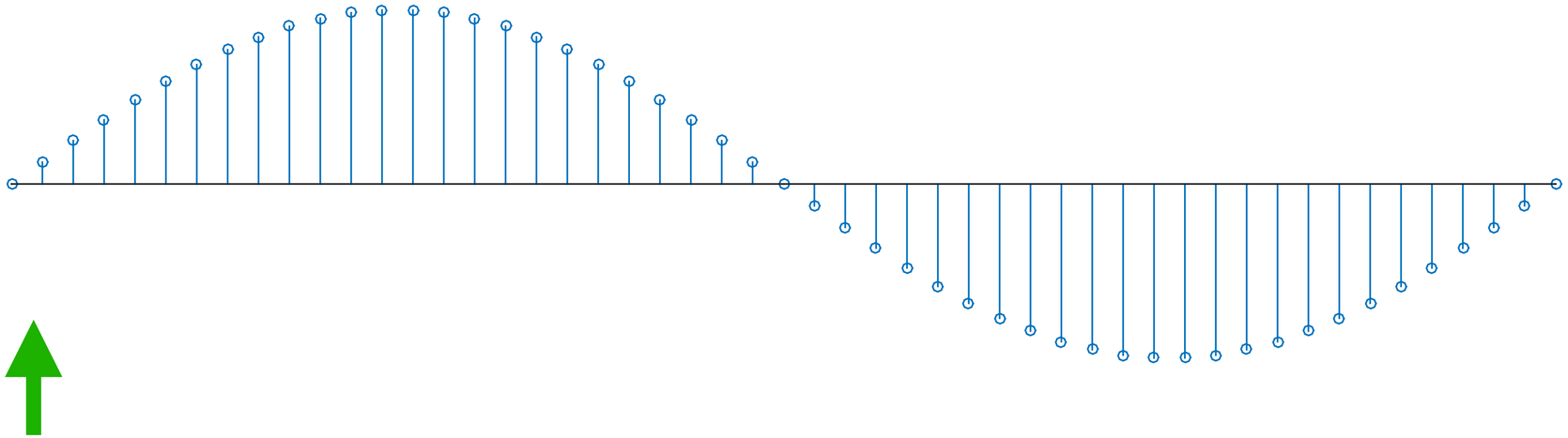
- A wavetable oscillator plays a repeating buffer of sound in a loop:

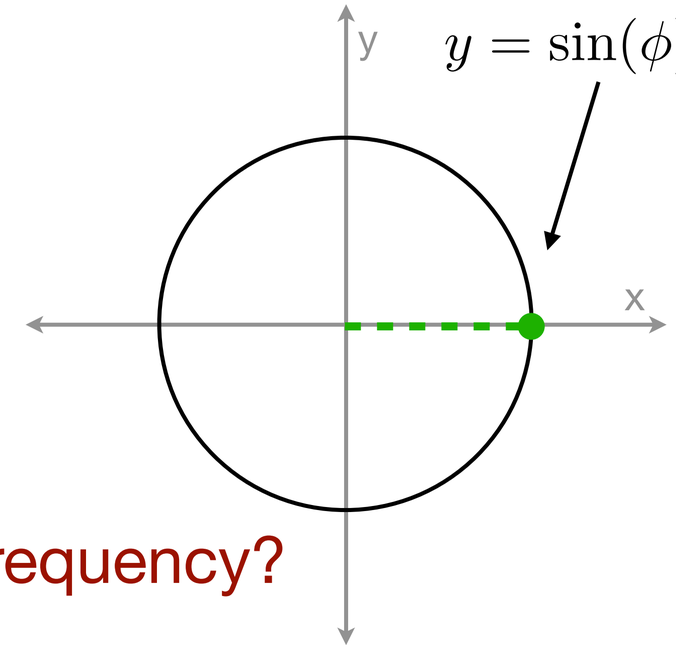


  - Effectively, a look-up table of values
  - As we saw in Lecture 3, we can use interpolation to get a higher-quality output
- Advantages of wavetables over using functions like `sin()`:
  - Less computation time per sample, because we can calculate the table at the beginning
  - More flexibility on the waveform (which could be nearly any shape)

# Wavetable phase and frequency

- We need to keep track of where we are in the wavetable
  - i.e. keep track of the read pointer in the source buffer
  - We'll need to remember this between calls to `render()`, like before
- The read pointer is analogous to the phase in our previous oscillator:



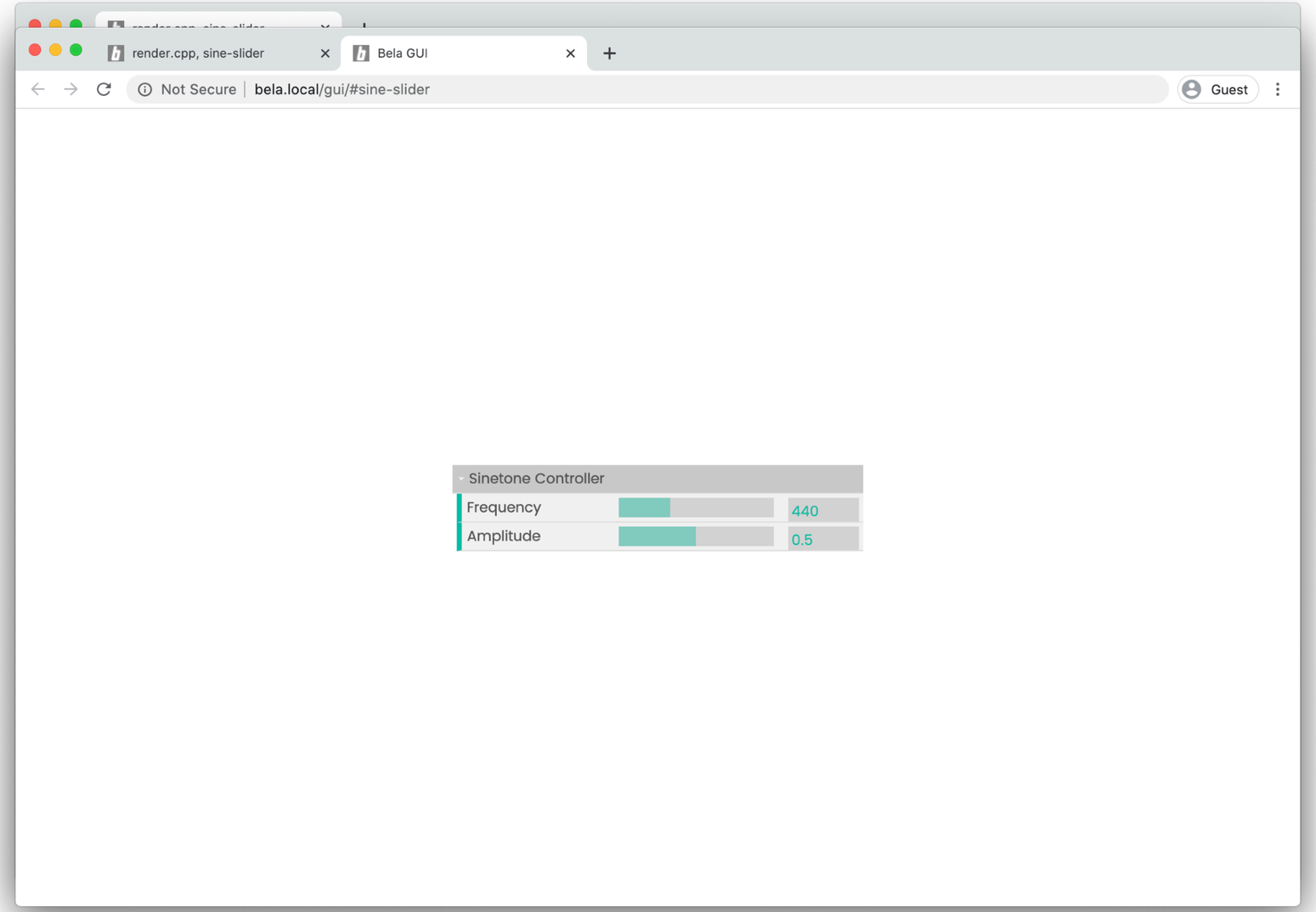



- If the read pointer corresponds to phase, what determines frequency?
  - Frequency is the derivative (rate of change) of phase
  - Therefore, frequency is given by rate of change of the read pointer

# Adding controls

- Open the sine-slider example from the companion materials: **github.com/BelaPlatform/bela-online-course**

- Run the program, then click the GUI button
- Adjusting the sliders should change the sound



# GUI controls: initialisation

- Bela provides a simple browser-based GUI where you can create sliders to control parameters
- To use it:
  1. Include `Gui.h` and `GuiController.h`
  2. Make global `Gui` and `GuiController` variables
  3. Initialise the sliders you want in `setup()`
  4. Get the parameters in `render()`

```
#include <libraries/Gui/Gui.h>
#include <libraries/GuiController/GuiController.h>

// Browser-based GUI to adjust parameters
Gui gui;
GuiController controller;

// setup() only runs one time
bool setup(BelaContext *context, void *userData)
{
    // Set up the GUI
    gui.setup(context->projectName);
    controller.setup(&gui, "Sinetone Controller");

    // Arguments: name, default, minimum, maximum, increment
    controller.addSlider("Frequency", 440, 220, 880, 0);
    controller.addSlider("Amplitude", 0.5, 0, 1, 0);

    return true;
}
```

# GUI controls: parameter query

frequency and amplitude
are no longer global variables
*(why does this still work?)*

we query the sliders
once per block (not every sample)

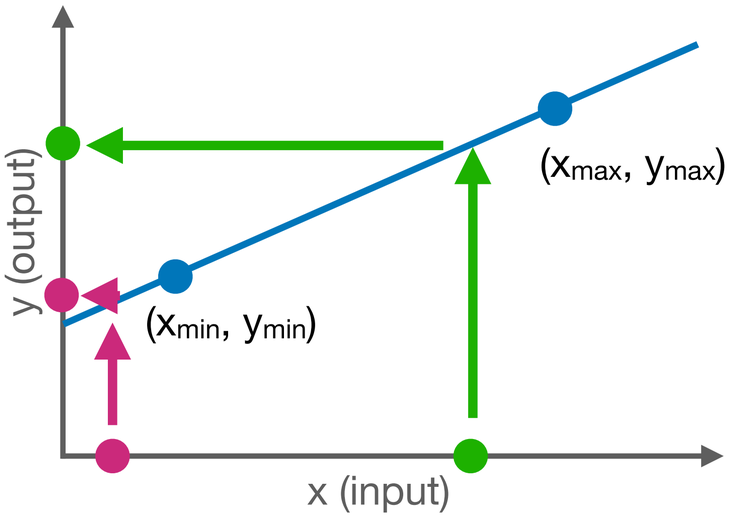
```
// render() is called every time there is a new block to calculate
void render(BelaContext *context, void *userData)
{
    float frequency = controller.getSliderValue(0); // Frequency is first slider
    float amplitude = controller.getSliderValue(1); // Amplitude is second slider

    // This for() loop goes through all the samples in the block
    for (unsigned int n = 0; n < context->audioFrames; n++) {
       // Increment the phase by one sample's worth at this frequency
       gPhase += 2.0 * M_PI * frequency / context->audioSampleRate;
       if(gPhase >= 2.0 * M_PI)
           gPhase -= 2.0 * M_PI;

       // Calculate a sample of the sine wave
       float out = amplitude * sin(gPhase);

                    ...
```

# Rescaling ranges

- It's common to want to rescale the range of a signal
  - For example, use a control with a 0-1 range to set the frequency of an oscillator
- Bela provides some useful functions for adjusting ranges
- `map()` rescales the input `x` from one range to another:

```
float map(float x, float in_min, float in_max, float out_min, float out_max)
{
  return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
}
```

  - Equivalently: $y = y_{min} + (y_{max} - y_{min})\frac{x - x_{min}}{x_{max} - x_{min}}$
  - Values outside the input range will be extrapolated
  - Task: rescale the frequency range from 220-880 to a different range (say 110-440)



# Constraining ranges

- Sometimes we want to limit the range of a signal
  - For example, by convention, audio signals should always be between -1 and 1
- `constrain()` clips the input `x` to stay within the specified range:

```
float constrain(float x, float min_val, float max_val)
{
   if(x < min_val) return min_val;
   if(x > max_val) return max_val;
   return x;
}
```

  - Values within the range will be unchanged
  - Values above or below will be constrained to the maximum or minimum

# Decibel amplitude scale

- The `map()` function performs linear rescaling
  - However, many real-world phenomena are not linear!
- The loudness of a sound is logarithmic with respect to its raw amplitude
  - Human hearing is complex, and perceived loudness also depends on frequency and other factors, but this is the basic relationship
- 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:

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

- For example, a ratio of 0.1 (1/10) can be written as -20dB
- What is 0.01 (1/100) in dB?
- What is +60dB as a linear ratio?

# Reference points for decibel scales

- Remember: the decibel scale is for ratios between two quantities
  - In every case using dBs, we should ask: "relative to what?"
- In digital signal processing, it's common to see the term dBFS
  - Decibels Full-Scale
  - The maximum value (e.g. 1.0) is defined as 0 dBFS
  - Smaller values (e.g. 0.1, 0.01) correspond to negative numbers (-20dB, -40dB)
  - This is the most common use of dB if no other suffix is specified
- In analog audio systems, we'll also see dBV and dBu
  - 0 dBV = 1.0V
    - e.g. -20dBV = 0.1V, etc.
  - 0 dBu = 0.775V (= 1mW into a standard 600Ω load)
- Decibels are also used in acoustics for sound pressure level (SPL)
  - The formula is slightly different (based on $10*\log_{10}$ rather than $20*\log_{10}$) because it's measuring power rather than amplitude; we won't go into this now

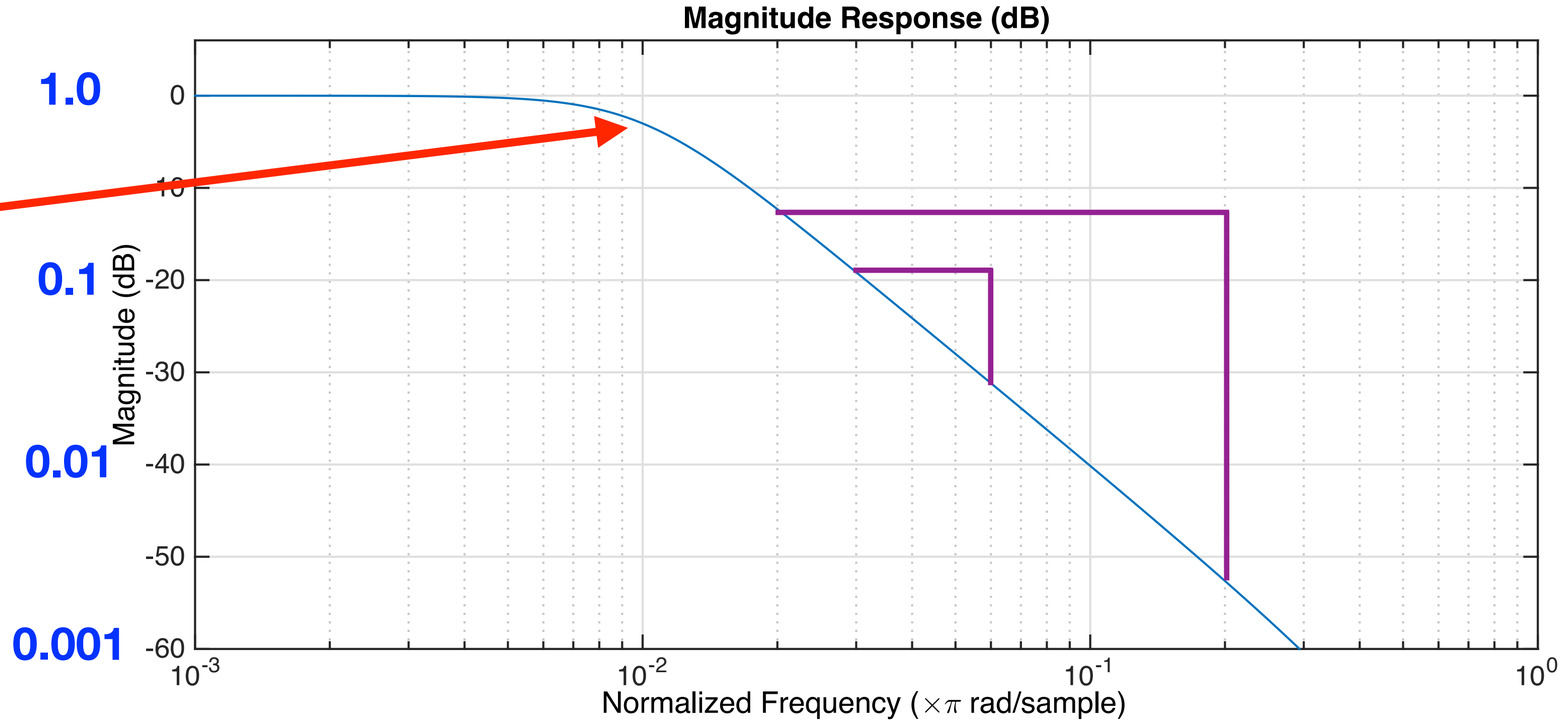
# Uses of decibel amplitudes

- Decibels are used to label most gain controls and level meters
- Decibel scales are often used in specifying filter magnitude response
  - By convention, the critical frequency of many standard filter types is the -3dB point
  - The order of a filter determines the rolloff, measured in dB/octave or dB/decade

Butterworth low-pass filter

$f_c = 0.01\pi$

-3dB at this point




N = 2

12 dB/oct

=

40 dB/dec

# Decibel scale task

- Task: in the sine-slider project, implement a decibel scale for amplitude
  - Make the range go from -40 dBFS to 0 dBFS
  - Option 1: change the range of the amplitude slider to -40 to 0
  - Option 2: use the `map()` function to rescale the range 0-1 to go from -40 to 0
  - In either case, implement the decibel formula to calculate the linear value of amplitude that we use to scale the signal: $A_{linear} = 10^{\frac{A_{dB}}{20}}$
    - Hint: `pow(x,y)` implements $x^y$

# Multiple oscillators

- On analog synths, it's common to use 2 or more oscillators for a single voice
  - The oscillators are tuned nearly, but not exactly, to the same frequency
  - Slight detuning results in beat frequencies according to the mathematical relationship:

$$\sin(x) + \sin(y) = 2\sin(\frac{x+y}{2})\cos(\frac{x-y}{2})$$

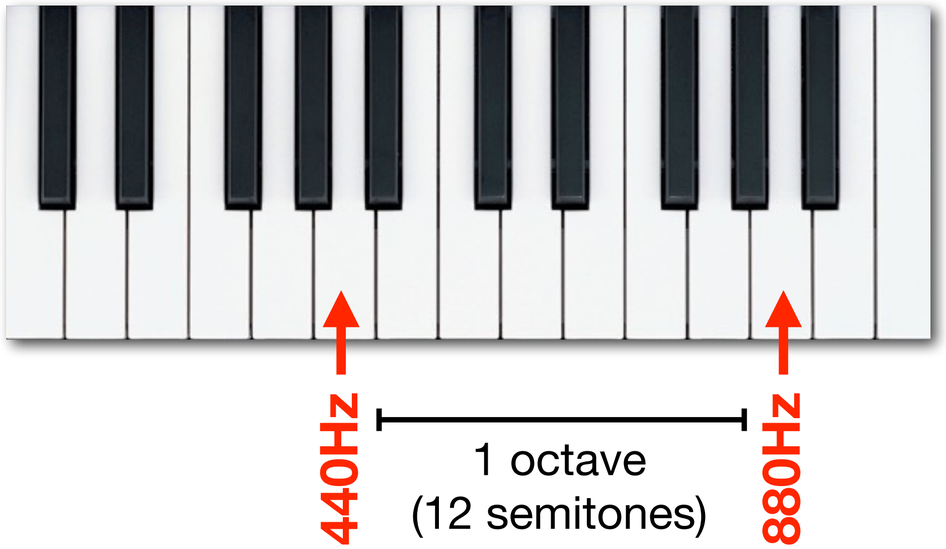
- Practically speaking, we want 2 controls:
  - One for setting the centre frequency of both oscillators (same as before)
  - One for changing the detuning (the slight difference between the oscillators)
  - If the centre frequency is *f* and the detuning amount is *r*, we should have:

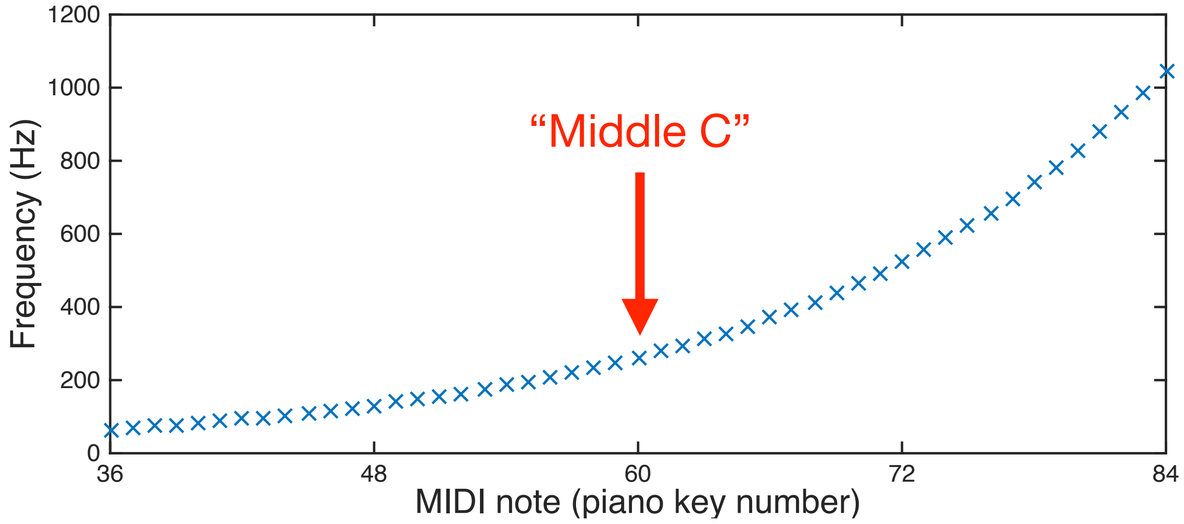
$$f_1 = (1+r)f \qquad f_2 = (1-r)f$$

    - In this case, *r* = 0 corresponds to no detuning (both oscillators the same frequency)
- Task: using the wavetable-slider example, implement two detuned oscillators
  - Add another GUI control for the detuning ratio

# Frequency and pitch

- The perceived pitch of a sound is related to its frequency
- Pitch perception also operates on a logarithmic scale
  - Musical intervals are defined as frequency ratios
    - An octave is a 2:1 ratio; a perfect fifth is 3:2, a fourth is 4:3, etc.
  - To get from frequency to notes on a keyboard, we need a logarithmic calculation
  - However, by convention, this calculation is different from decibels



# From musical pitch to frequency

- In audio synthesis, we often want to convert from pitch ("note") to frequency
  - We want a linearised scale: an increment of *x* is always the same musical interval
  - On analog synths, a common standard is 1V/octave
    - i.e. the control signal going up 1V makes the pitch go up by an octave (what happens to the frequency?)
- Given a reference frequency $f_{ref}$ and an input *V* measured in octaves:
  - $f_{out} = f_{ref} 2^{V}$ *(octaves to frequency)*
  - What if *V* was measured in semitones instead? (12 semitones to the octave)
    - How do I convert from semitones to octaves?
  - $f_{out} = f_{ref} 2^{V/12}$ *(semitones to frequency)*
- Task: in the wavetable-slider project:
  - Change the frequency slider to be called "pitch", with a range of 0-24 semitones
  - Implement the frequency calculation above with $f_{ref} = 110\text{Hz}$
  - This should result in a more even pitch change as you move the slider

# MIDI note number

- MIDI (Musical Instrument Digital Interface) is a standard protocol for controllers and synths
  - It defines a note number for each key of the piano keyboard, numbered sequentially from low to high
  - By definition, middle C is note number 60
    - It has frequency 261.63Hz
  - The A above middle C is note number 69
    - It has frequency 440Hz (a commonly used reference, sometimes called "A440")


- How should we write the formula to go from MIDI note number to frequency?
  - For a note number *N*, how do we calculate number of semitones above A440? *N* - 69
    - Number of octaves? (*N* - 69) / 12
  - If note 69 corresponds to $f_{ref} = 440\text{Hz}$ how do we calculate the frequency for note N?

$$f_{out} = 440 \cdot 2^{\frac{N-69}{12}}$$ *(MIDI note number to frequency)*

# Quantising pitch

$$f_{out} = 440 \cdot 2^{\frac{N-69}{12}}$$ *(MIDI note number to frequency)*

- When using MIDI note numbers, each integer corresponds to a piano key
  - We can create a chromatic scale using only integer values
  - We can also feed in fractional note numbers to the formula, even if the MIDI standard doesn't allow this
- In the GUI, we can control the increment of a parameter:

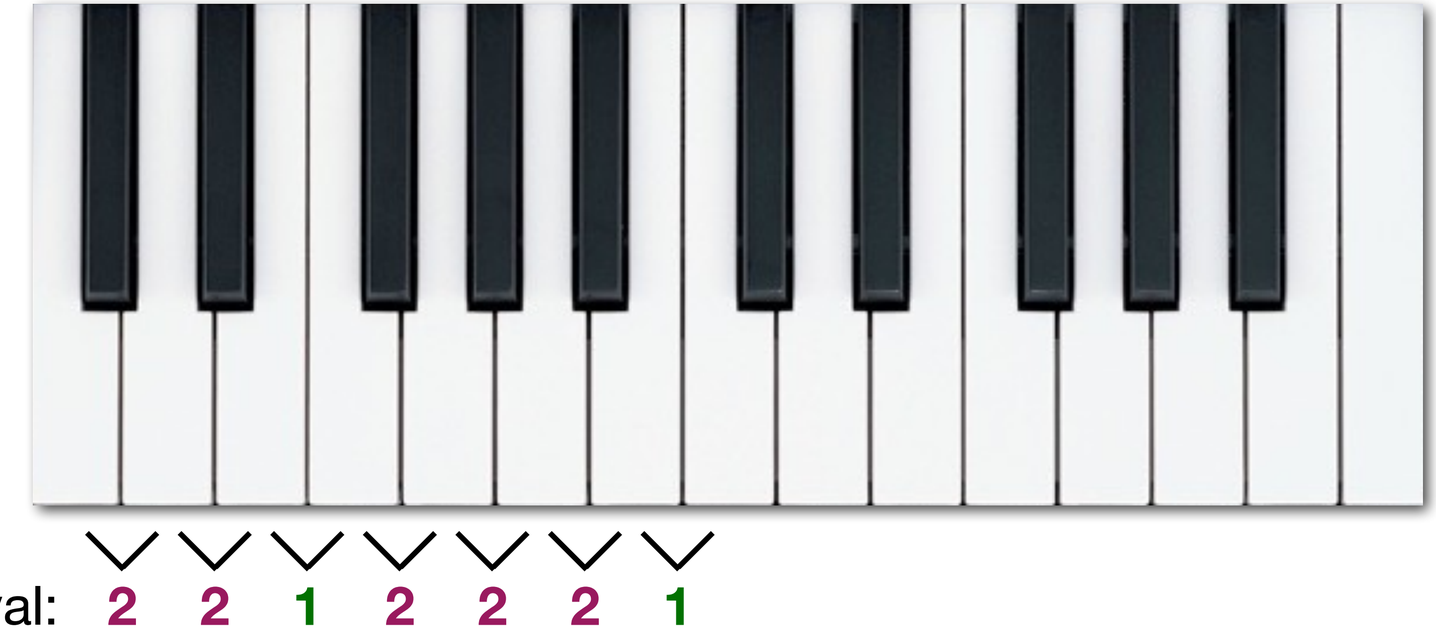
```
controller.addSlider("Note Number", 60, 48, 72, 0);
```

**Increment**

  - Default of 0 means a continuous range (no quantisation)
  - Set increment to 1 to get a chromatic scale
  - Set increment to 0.5 to hear quarter tones!

# Bonus task: diatonic scales

- What if we wanted a different musical scale?
  - Diatonic scales have 7 notes to the octave in a characteristic pattern:




  - We can't implement this with a single value of increment
- Alternatively: use an array to implement the diatonic scale
  - Store the MIDI note numbers in an array, then use the GUI control as an index to the array

# Keep in touch!

Social media:

**@BelaPlatform**

**forum.bela.io**

**blog.bela.io**

More resources and contact info at:

**learn.bela.io/resources**