# 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

Today

## 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

# Lecture 12: Envelopes

## What you'll learn today:

Linear and exponential envelopes
Generating ramps by counting samples
Finding the envelope of an audio signal

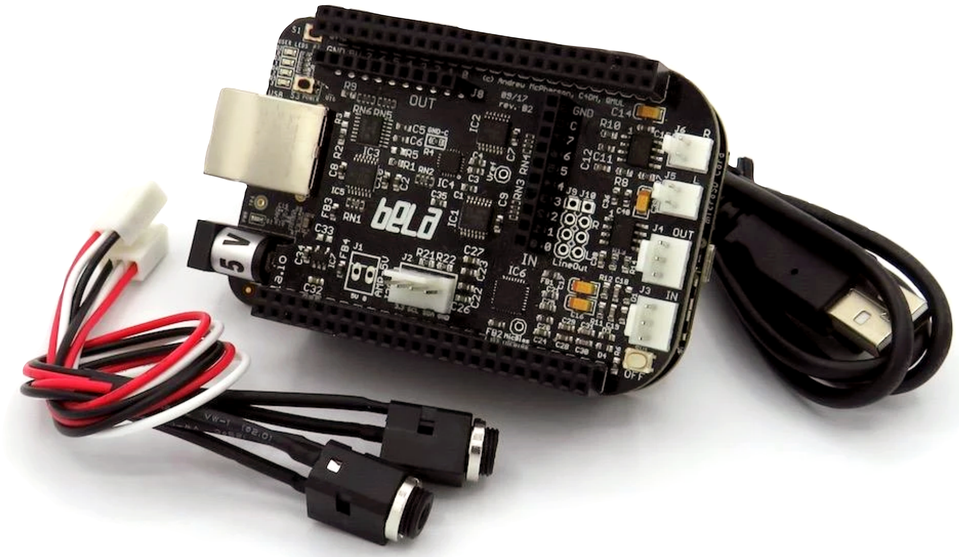
## What you'll make today:

Ramp-controlled filters, simple percussion, envelope follower

## Companion materials:

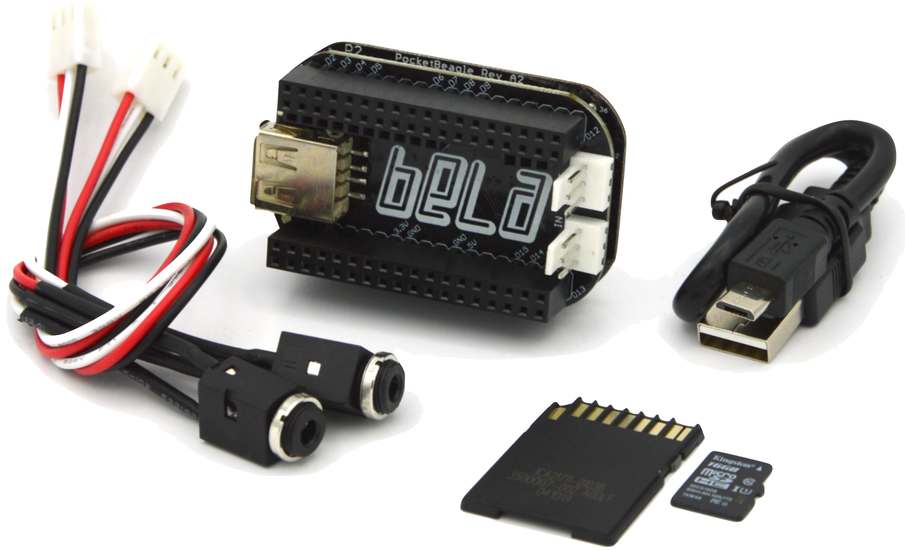
**github.com/BelaPlatform/bela-online-course**

# What you'll need



Bela Starter Kit

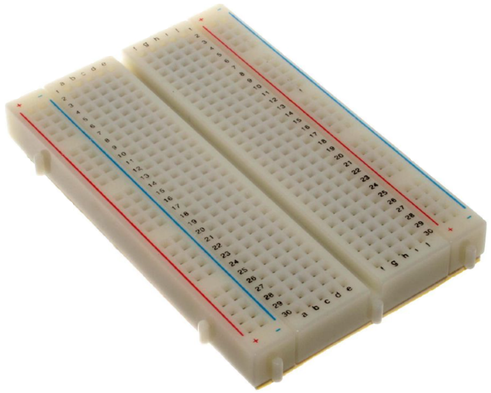
or

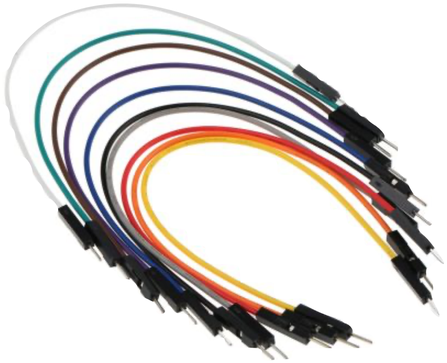


Bela Mini Starter Kit

[shop.bela.io]

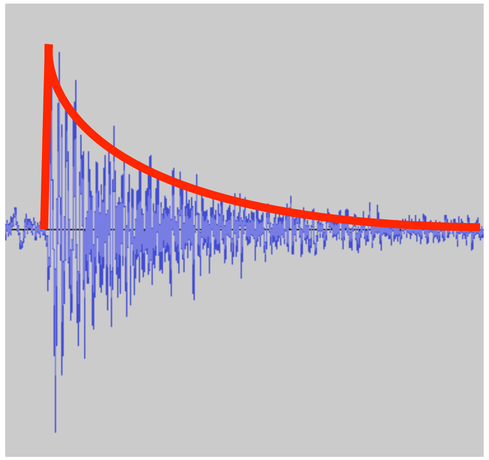
Optional for this lecture:





# Envelopes

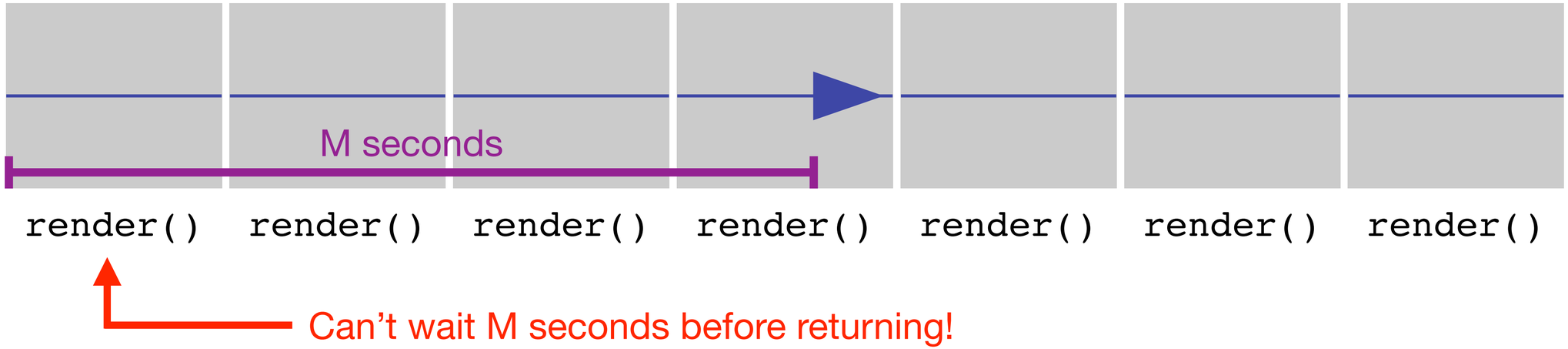
- The envelope of an audio signal is its amplitude profile over time
- Imagine tracing a line over the top of the audio signal over time:



  - Most percussive sounds have envelopes like this one
  - We can apply an envelope to an oscillator (or a noise source) to make a simple drum sound
- Several types of envelope:
  - Linear, exponential, ADSR (Attack, Decay, Sustain, Release), etc.

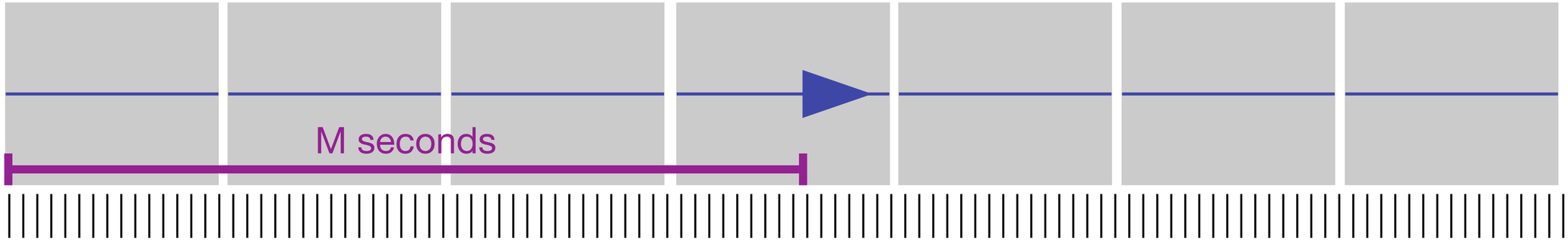
# Review: timing

- We are always processing signals one block at a time
  - Our `render()` function is only responsible for what happens in the next block
  - We have strict timing limitations on how long `render()` can take to finish
  - If we miss our deadlines, we don't get a delay, we get an underrun




- The time when the computation happens is not the same as the signal time
  - By the time `render()` is called, the input was already captured
  - On the other hand, the output signals we generate don't appear until the next block

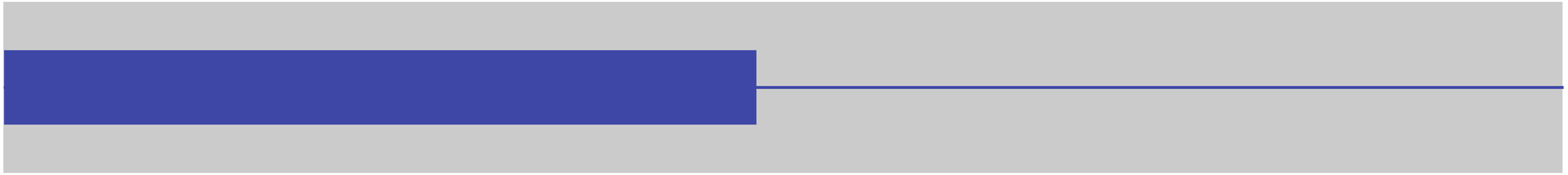
# Review: timing without delay




- Alternative to delay: count elapsed time as we go along
  - Perhaps we want something like the `millis()` function on Arduino, which gives the number of milliseconds since the board powered up
  - There are system calls on Linux that give us the clock time, though many of these are not safe to run in Xenomai primary mode (i.e. they would cause mode switches on Bela)
- What is a simpler way to keep track of elapsed time within a digital signal?
  - Count samples!
  - (elapsed time in seconds) = (elapsed samples) * (sample rate)
  - This also has the advantage of telling us the signal time rather than the computation time

# Review: intervals

"Generate a signal for the next M seconds and then stop"



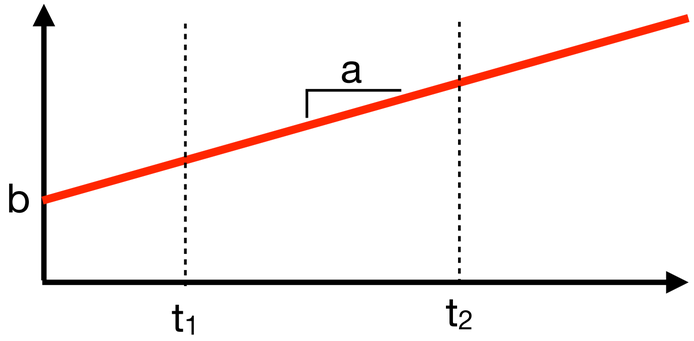
- The previous examples looked at a discrete trigger after a period of time
- What about having a signal persist for a certain interval?
  - For example, how do we make a pulse that lasts 0.1 seconds?
- Same idea as before: count samples until a certain number elapse
  - This time, instead of having a condition occur once when the count elapses, make the condition true every sample *until* the count elapses
  - e.g. inside the `for()` loop:

```
counter++;
if(counter < interval) {
    // Generate a signal
}
```

# Linear envelope

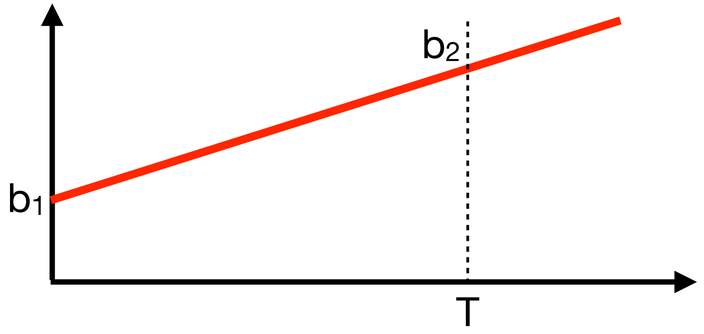
- A linear envelope consists of one or more straight line segments over time
  - Each segment has a constant slope between start and end points:

$$v(t) = at + b$$
$$t_1 \leq t < t_2$$




  - To take a more common formulation for real-time implementation, consider a segment that starts at a value of **$b_1$** at time **0**, and goes to a value of **$b_2$** by time **T**:

$$v(0) = b_1$$
$$v(T) = b_2$$
$$\longrightarrow v(t) = \frac{b_2 - b_1}{T}t + b_1$$

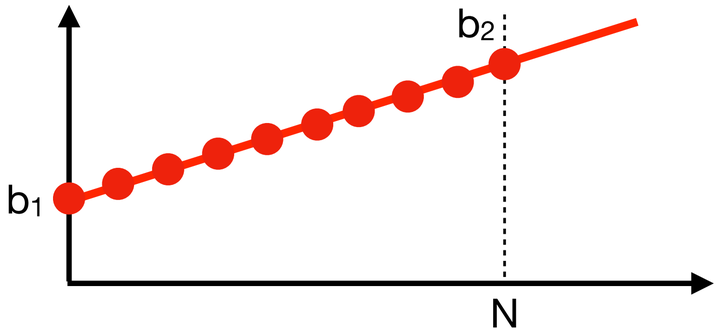



    - This is very similar to what `map()` does!

# Linear envelope

- Let's rewrite the line segment (or ramp) equation in discrete time:

$$v(t) = \frac{b_2 - b_1}{T}t + b_1 \longrightarrow v[n] = \frac{b_2 - b_1}{N}n + b_1$$




- How do we implement this line segment in real time?
  - It's easy to look at a graph over time to see where we want to go, but how do we get there one sample at a time?
  - In other words: what do we do here and now to get from sample `n` to sample `n+1`?

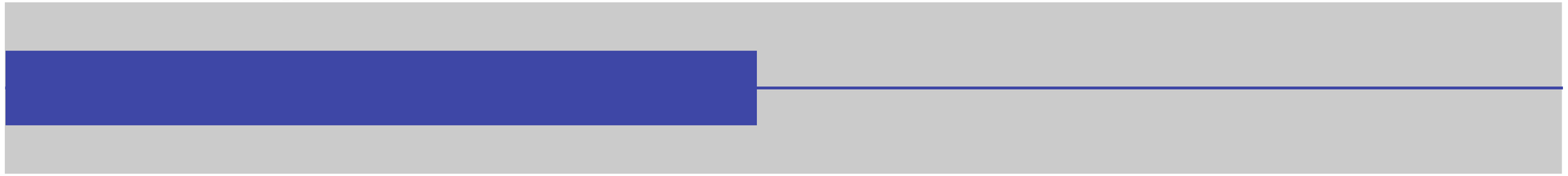
$$v[n+1] = v[n] + \frac{b_2 - b_1}{N}$$

    - Add a constant increment each time, until we reach the desired end point (in time or in level)

# Linear envelope (of finite length)

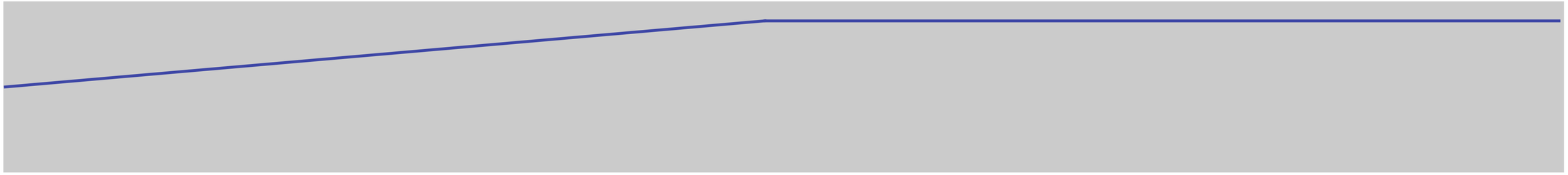
- Linear envelopes in digital signals typically have a finite length
  - Rather than increasing or decreasing forever, the line segment lasts some specific amount of time (i.e. some specific number of samples) and then stops
- The form is very similar to the case we just saw:

  “Generate a signal for the next M seconds and then stop”



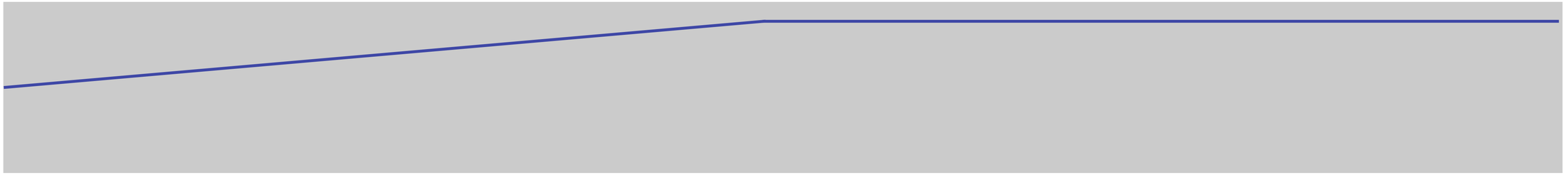
- More specifically in this case:

  “Generate a linear ramp for the next M seconds and then do something else”



# Real-time line segment generator

"Generate a linear ramp for the next M seconds and then do something else"



- As we saw before: count samples until a certain number elapse
  - This gives us the finite duration of the segment
  - As before, we need a condition (e.g. an `if()` statement) which is true for every sample *until* the count elapses
  - e.g. inside the `for()` loop:

```
counter++;
if(counter < interval) {
    // Generate a signal
}
```

- Within the segment, add a constant increment each sample:

$$v[0] = b_1 \quad v[n+1] = v[n] + \frac{b_2 - b_1}{N}$$

⟶ ramp from $b_1$ to $b_2$ over N samples

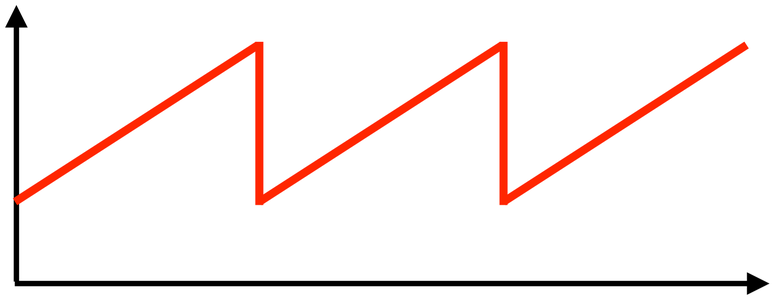
# Real-time line segment generator

$$v[0] = b_1 \quad v[n+1] = v[n] + \frac{b_2 - b_1}{N}$$ ⟶ ramp from $b_1$ to $b_2$ over N samples

- What we will need to implement this:
  - Variable 1: counter to measure how many samples have elapsed
  - Variable 2: current value of the output
  - Other values (might be constants or variables): N, $b_1$, $b_2$

These could be combined in some implementations: the current value implicitly tells you how much time has elapsed

- Task: using the ramp-filter project
  - Implement a linear envelope from 200 to 4000Hz over 2.0 seconds *(what is N above?)*
  - When the ramp finishes after 2.0 seconds, start over from 200Hz again
  - Use the envelope to control the cutoff frequency of the filter



# Linear envelope code

```
// Variables for linear envelope
const float kRampDuration = 2.0;
const float kFilterFrequencyMin = 200.0;
const float kFilterFrequencyMax = 4000.0;
float gFilterFrequency = kFilterFrequencyMin;
float gFilterFrequencyIncrement = 0;

// Oscillator and filter objects
Wavetable gOscillator;
Filter gFilter;

// setup() only runs one time
bool setup(BelaContext *context, void *userData)
{
    // Initialise ramp increment [other setup omitted]
    float rampDuration = kRampDuration *
                         context->audioSampleRate;
    gFilterFrequencyIncrement =
      (kFilterFrequencyMax - kFilterFrequencyMin) /
      rampDuration;

    return true;
}
```

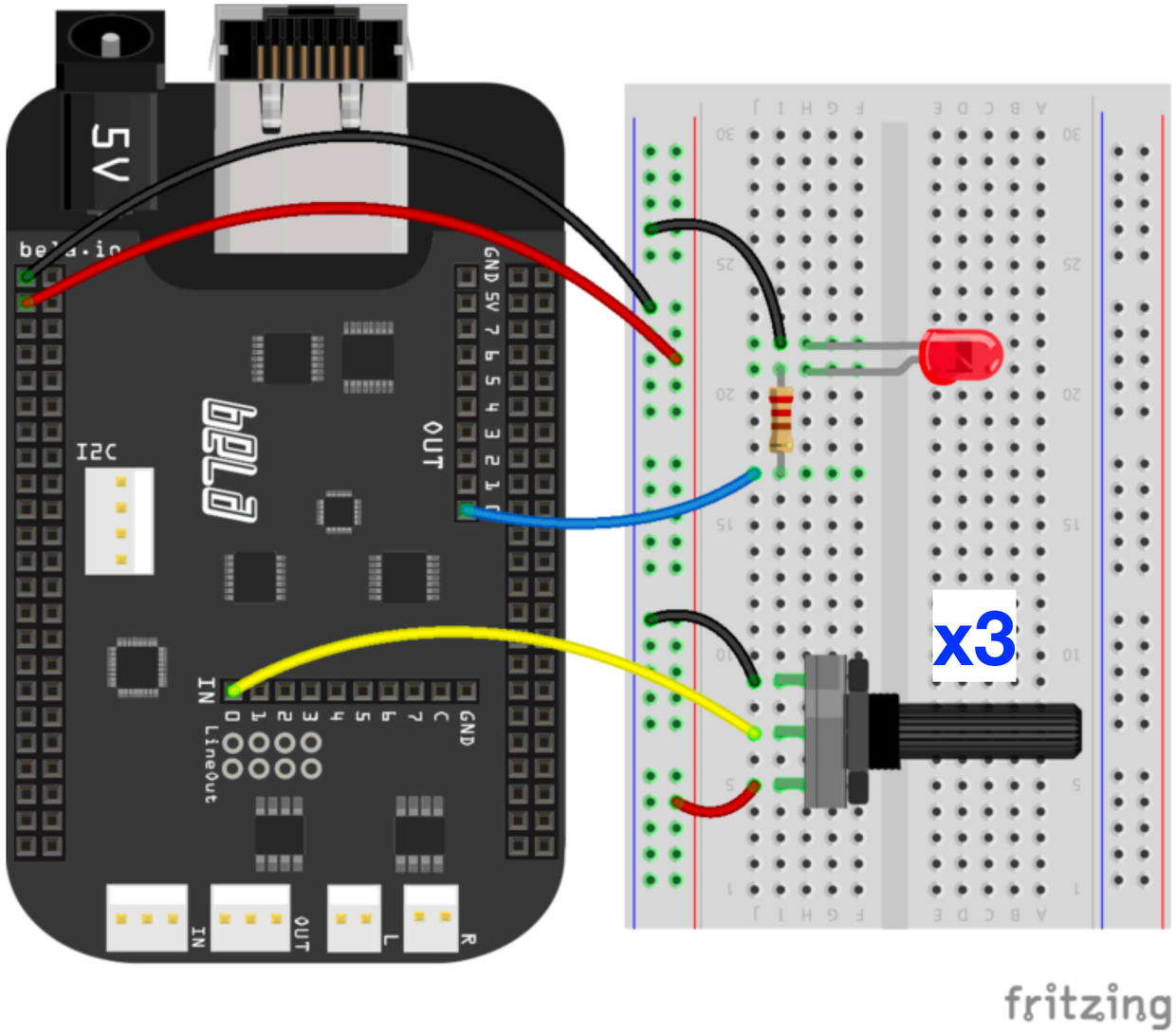
Calculate the duration of the segment and based on that, the increment $\frac{b_2 - b_1}{N}$

```
// render() is called every time there is a new block to calculate
void render(BelaContext *context, void *userData)
{
        // This for() loop goes through all the samples in the block
    for (unsigned int n = 0; n < context->audioFrames; n++) {
        // Increment the frequency to create a linear ramp
        gFilterFrequency += gFilterFrequencyIncrement;
        if(gFilterFrequency >= kFilterFrequencyMax)
            gFilterFrequency = kFilterFrequencyMin;

        // Update the filter frequency
        gFilter.setFrequency(gFilterFrequency);

        // Generate and filter the signal
        float in = gOscillator.process();
        float out = 0.2 * gFilter.process(in);

        // This part is done for you: store the sample in the
        // audio output buffer
        for(unsigned int channel = 0;
            channel < context->audioOutChannels; channel++) {
            audioWrite(context, n, channel, out);
        }
    }
}
```

Increment the frequency each cycle, reset when we reach the top

# Adding controls

- Let's add potentiometers to control some envelope parameters:
  - Ramp duration
  - Ramp maximum value
  - Oscillator frequency
- Alternatively, you could make a GUI to control these parameters
- Also add an LED to an analog output to show the envelope value
  - This only works on the original Bela
  - Use `map()` to rescale the envelope range between 0.2 to 1.0
    - We start from 0.2 because the LED needs a certain minimum voltage to light up



# Ramp generator: a class-based approach

- Let's make a C++ class to encapsulate the line segment (ramp) generator
- Key methods:
  - Tell the generator to start a line segment, specifying a destination and a time to get there

```
void Ramp::rampTo(float value, float time);
```

  - Return the next value in the line segment

```
float Ramp::process();
```

- When we reach the end of the segment, the value should stay there
- Two additional methods:
  - Tell the generator to immediately jump to a new value

```
void Ramp::setValue(float value);
```

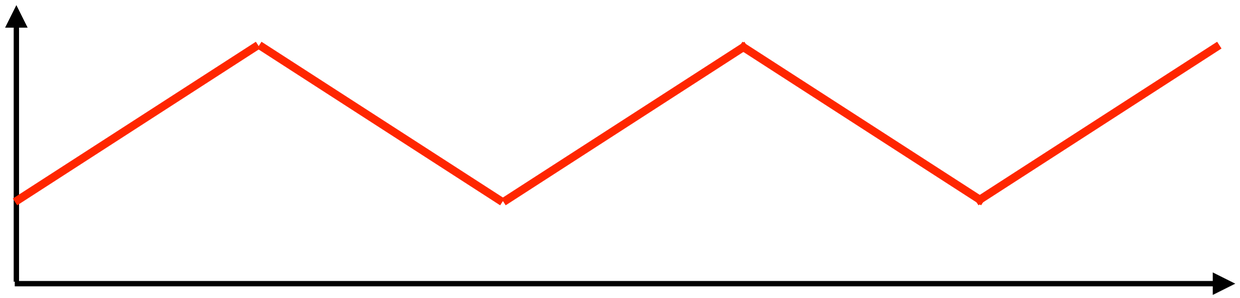
  - Ask whether the generator has reached a steady value (i.e. finished a line segment)

```
bool Ramp::finished();
```

- Try this now in the ramp-generator-class project. Fill in `Ramp.cpp`

# Multiple segments

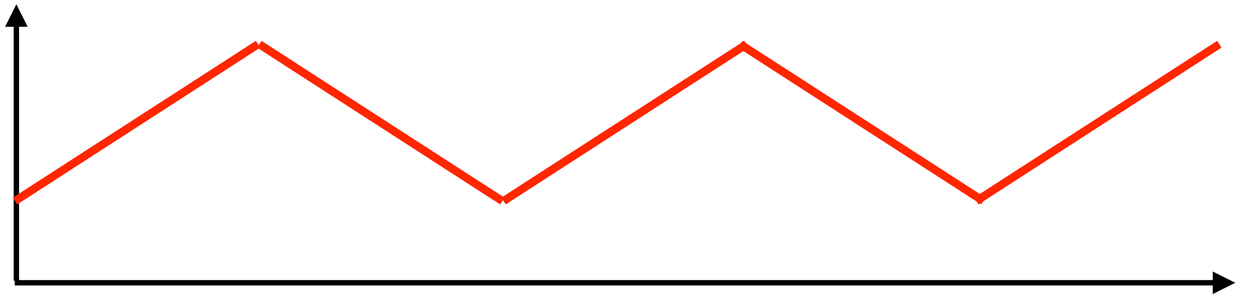
- Many practical envelopes involve a series of line segments
  - We will see this in more detail soon with the ADSR envelope
- Suppose we want to have alternating increasing and decreasing segments:



- We can use our ramp generator class, which already gives us:
  - Variable 1: counter to measure how many samples have elapsed
  - Variable 2: current value of the output
  - Variable 3: increment for how much to add to the value each time
- What else do we need to alternate increasing and decreasing ramps?
  - Variable 4: state variable to tell us which direction we're moving at the moment

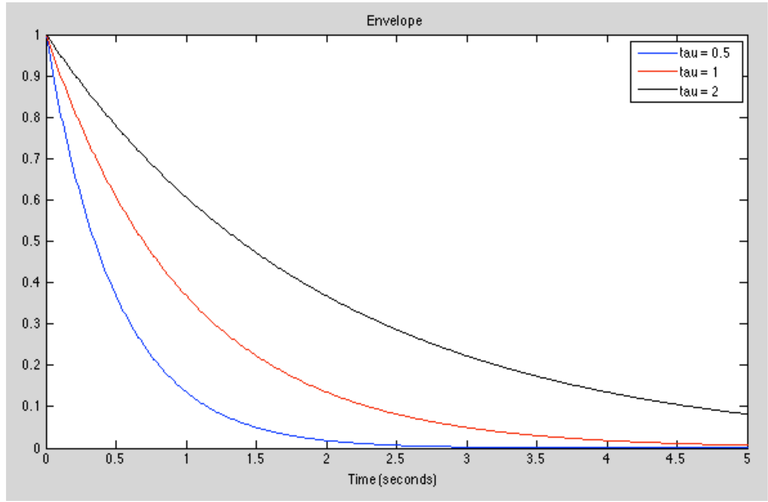
# Multiple segment task

- Task: using the ramp-generator-class project
  - Add a state variable of type `bool` to indicate which direction the ramp is going
  - Make the envelope alternate between increasing and decreasing line segments
  - Optionally, add controls to change the length of each segment independently



# Exponential envelope

- An exponential envelope gets smaller over time by a constant percentage
  - Rather than a constant slope in the linear envelope
  - Percussive instruments commonly have (approximately) exponential envelopes
- Equation in continuous time: $v(t) = e^{-t/\tau}$
  - $\tau$ is called the time constant
  - Larger values of $\tau$ mean a slower decay
  - Starts at a value of 1 when *t* = 0, asymptotically approaches 0 in long time




- Another mathematically equivalent form: $v(t) = a^{-t}$ where $a = e^{1/\tau}$
- How should we implement $v[n]$ in discrete time?

# Exponential envelope, real time

- How should we implement $v[n]$ in discrete time, also in real time?
- We could calculate samples of $v(t)$ every time
  - But this is inefficient: need to calculate an exponential function on every sample
    - Potentially many multiplies per sample
  - Better: $v[n] = b^n$ which means: $v[0] = 1, v[n] = bv[n-1]$
    - This is a recursive implementation that takes only one multiply per sample
  - How do we find *b* such that $v[n]$ is a sampled version of $v(t) = a^{-t}$
    - What more do we need to know? **Sample rate**

$$v[f_s] = v(1) \implies b^{f_s} = \frac{1}{a} \longrightarrow b = \sqrt[f_s]{1/a} = \sqrt[f_s]{e^{-1/\tau}}$$

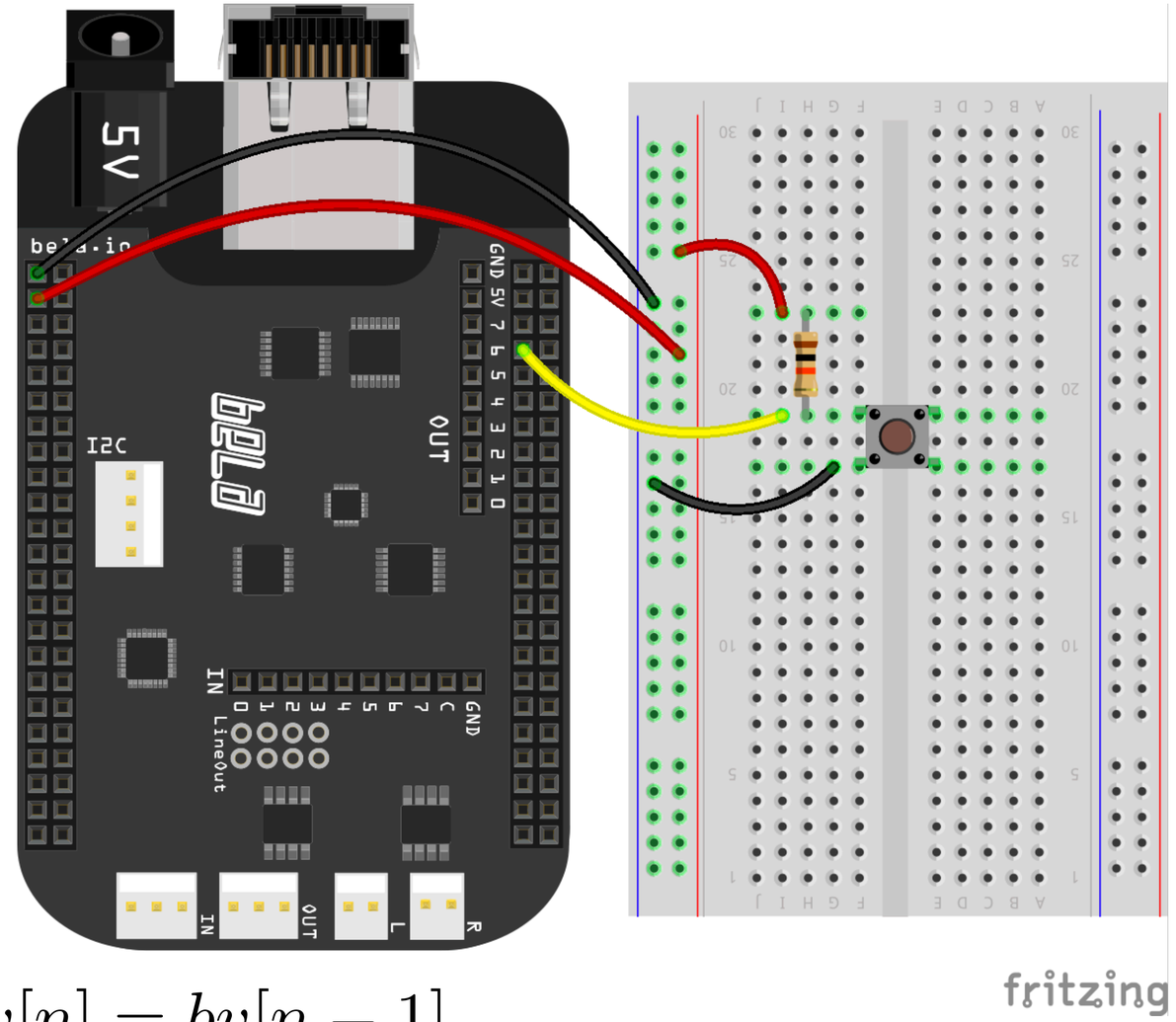
  - For example: $\tau$ = .5, $f_s$ = 1000Hz gives *b* = .998
  - What might be a problem with this approach?
  - What happens when $\tau$ and $f_s$ get larger?

# Exponential envelope precision

- We previously found: $b = \sqrt[f_s]{1/a} = \sqrt[f_s]{e^{-1/\tau}}$
- But as $f_s$ and $\tau$ get larger, $b$ approaches 1
  - τ = 1, $f_s$ = 44100 means $b$ = 0.9999773
  - τ = 10, $f_s$ = 44100 means $b$ = 0.9999977
  - τ = 10, $f_s$ = 96000 means $b$ = 0.9999990
  - Eventually, we won't have sufficient precision to distinguish $b$ from 1.0
- C `float` variable type is known as IEEE754 single precision
  - Uses 23 bits to store the fractional part
  - Precision is roughly $2^{-23} \approx .0000001$
  - Below this, won't be able to resolve small differences
  - This precision also affects every intermediate result in calculating $b^n$
  - Possible solution: use `double` type (46 bits to store fraction, but slower on Bela)

# Simple percussion

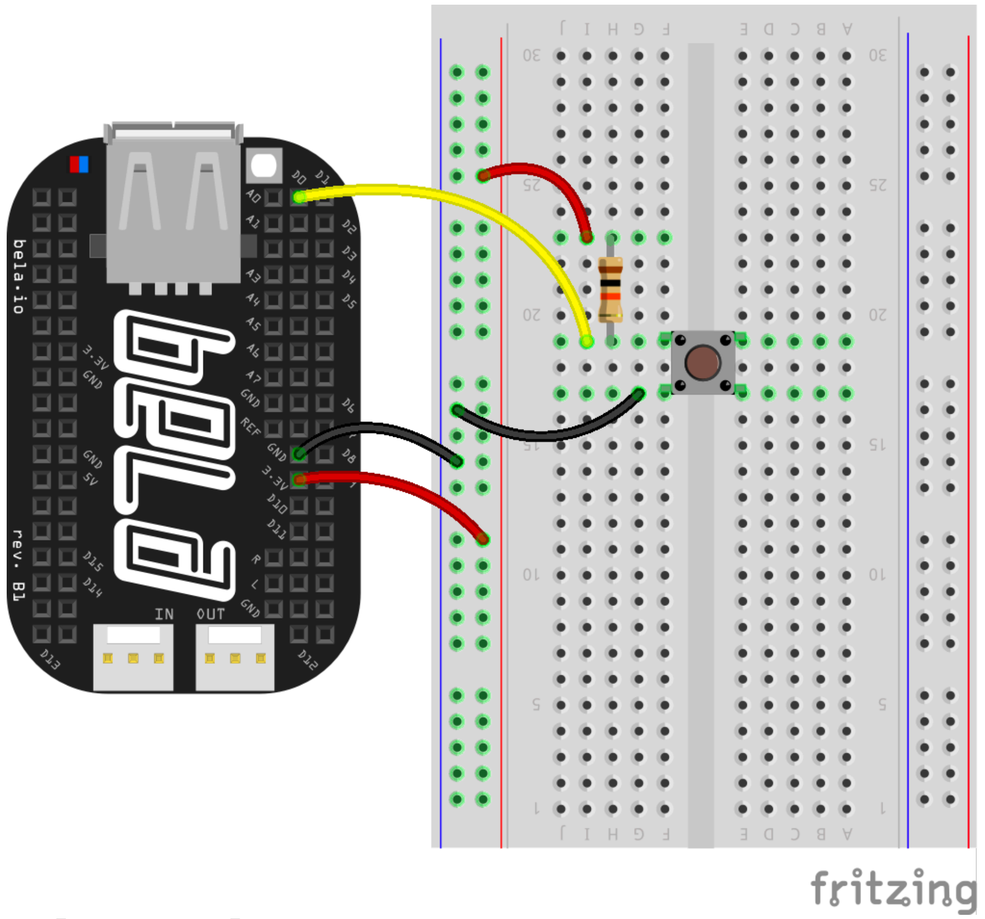
- What we need to store:
  - Current value of the envelope
  - Decay rate (the multiplier *b*)
- Task: using the percussion example
  - Wire up a button
  - When it's pushed, set the envelope value to 1.0
  - Each sample, multiply the value by the decay rate to produce an exponential envelope (use *a* = 100)
  - Use `pow()` to implement the calculation for *b*




$$b = \sqrt[f_s]{1/a} = \sqrt[f_s]{e^{-1/\tau}} \qquad v[0] = 1, v[n] = bv[n-1]$$

# Simple percussion

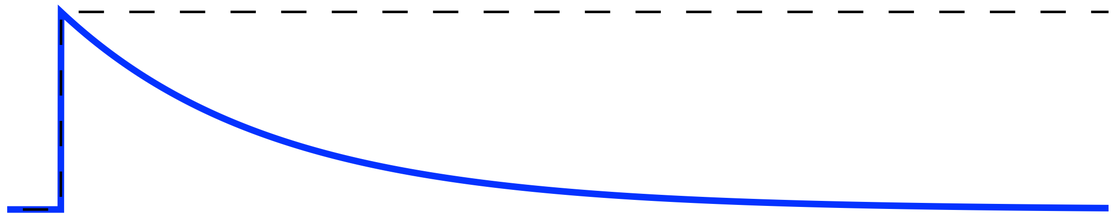
- What we need to store:
  - Current value of the envelope
  - Decay rate (the multiplier *b*)
- Task: using the percussion example
  - Wire up a button
  - When it's pushed, set the envelope value to 1.0
  - Each sample, multiply the value by the decay rate to produce an exponential envelope (use *a* = 100)
  - Use `pow()` to implement the calculation for *b*




$$b = \sqrt[f_s]{1/a} = \sqrt[f_s]{e^{-1/\tau}} \qquad v[0] = 1, v[n] = bv[n-1]$$

# Envelopes and filters

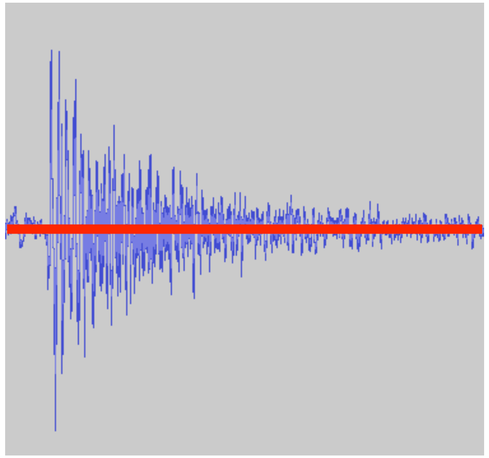
- Where have we seen an equation like this before? $v[n] = bv[n-1]$
- Similar to a 1st-order IIR filter: $y[n] = \alpha y[n-1] + (1-\alpha)x[n]$
- The exponential envelope is a 1st-order decay
  - It could be the step response of a 1st-order highpass filter



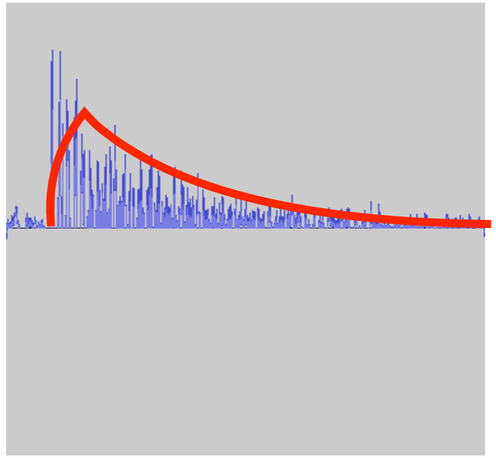
  - In fact, this is how exponential envelopes are typically implemented in analog circuits
  - For the parameter *a* = 100 in our last example, we have: $\tau = 1/\ln(a) \approx 0.22 \text{ seconds}$
- So (exponential) envelopes and filters are closely related
- This leads us to a different use of the term *envelope*...

# Calculating the envelope of a signal

- The envelope of an audio signal is its amplitude profile over time
  - Contrast this with the values of individual samples
- Given a signal, how do we calculate its envelope?
- We want to smooth out instantaneous variations
  - Sounds like we want a lowpass filter!
  - What's the (approximate) average value of the signal above? **0**
  - A simple lowpass filter won't work because the signal goes positive and negative



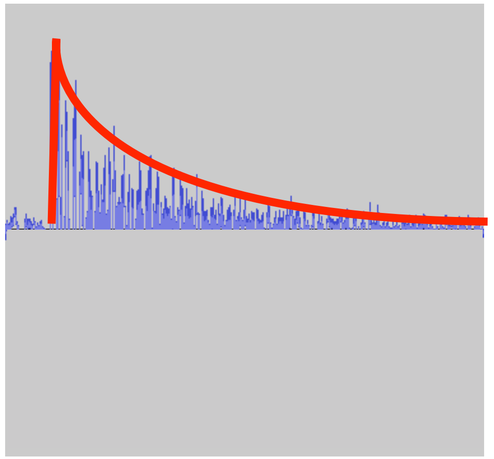
# Calculating the envelope of a signal

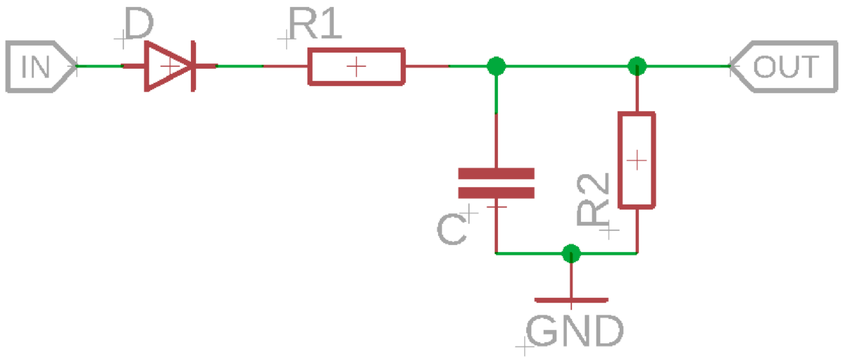


- The envelope of an audio signal is its amplitude profile over time
  - Contrast this with the values of individual samples
- Given a signal, how do we calculate its envelope?
- We want to smooth out instantaneous variations
  - Sounds like we want a lowpass filter!
  - What's the (approximate) average value of the signal above? **0**
  - A simple lowpass filter won't work because the signal goes positive and negative
- Instead, start with the absolute value of the signal
  - Then we can apply a lowpass filter to smooth out local variations
  - The lower the cutoff frequency, the less ripple, at the cost of slower response time

# Attack and release

- In the previous example, we respond slowly to both onsets and releases
- In many musical applications, we want a quick reaction to an onset, with a slower decay upon release
  - We can't do this with a single lowpass filter!
  - In analog, we use a diode peak detector circuit to have a fast attack and slow decay (choosing $R_1 \ll R_2$):






- A digital implementation needs two filters: one for attack, one for release
  - Both lowpass filters, but with different time constants
  - We will choose which one to use based on the level of the input

# Digital peak detector

- A digital implementation needs two filters: one for attack, one for release
  - Both lowpass filters, but with different time constants
  - Let's call them $\tau_a$ and $\tau_r$
  - Then we can calculate the coefficients of the filters:
  - This gives us the following piecewise equation:

$$b_a = \sqrt[f_s]{e^{-1/\tau_a}}$$
$$b_r = \sqrt[f_s]{e^{-1/\tau_r}}$$

$$y[n] = \begin{cases} b_a y[n-1] + (1-b_a)x[n] & x[n] \geq y[n-1] \\ b_r y[n-1] + (1-b_r)x[n] & x[n] < y[n-1] \end{cases}$$

← Attack: input greater than output

← Release: input less than output

  - Hint: use an `if()` statement to choose between the two cases

- Task: using the envelope-follower project, implement an auto-wah effect
  - Find the envelope of the input signal using the equation above
  - Rescale the range of the envelope to control the cutoff frequency of the filter
  - Use GUI controls for adjusting the attack and release time constants and the scaling

# Keep in touch!

Social media:

**@BelaPlatform**

**forum.bela.io**

**blog.bela.io**

More resources and contact info at:

**learn.bela.io/resources**