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 14: ADSR

What you'll learn today:

The ADSR (attack-decay-sustain-release) envelope
Implementation using state machines and line segments
Class-based implementations

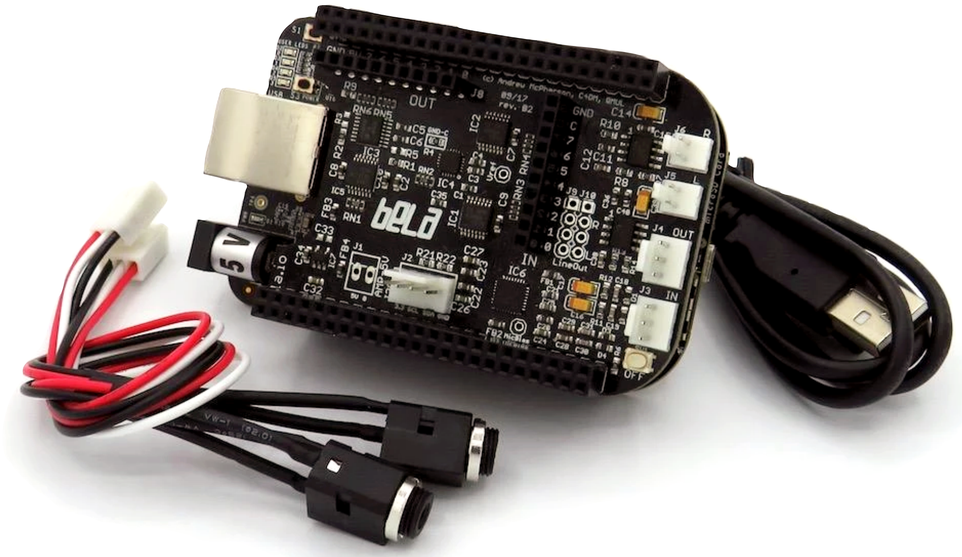
What you'll make today:

ADSR generators with linear or exponential segments

Companion materials:

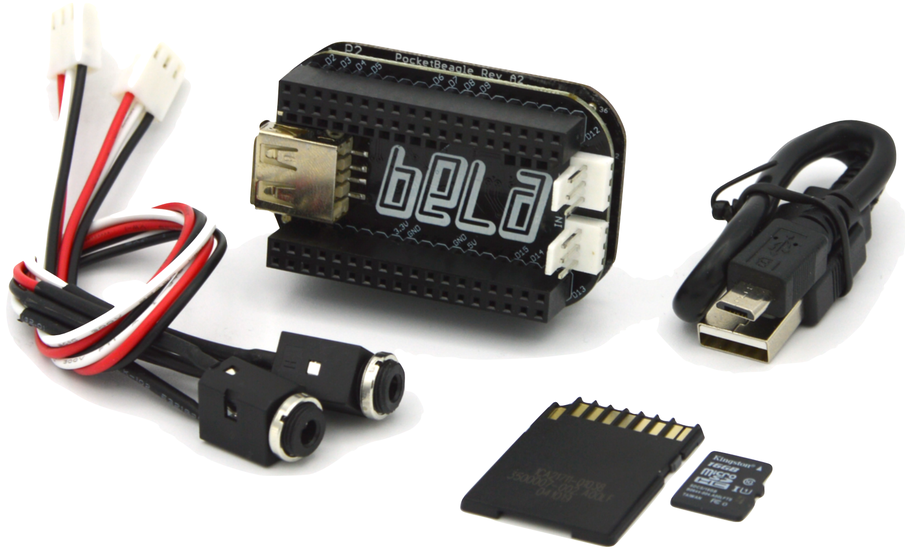
github.com/BelaPlatform/bela-online-course

What you'll need



Bela Starter Kit

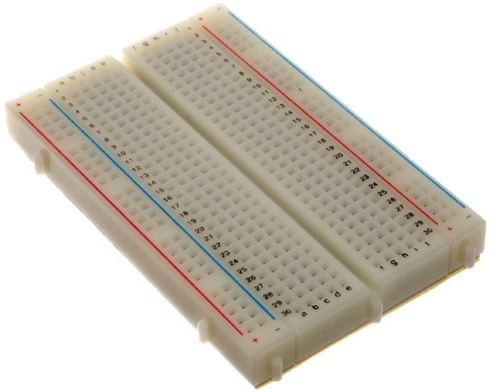
or

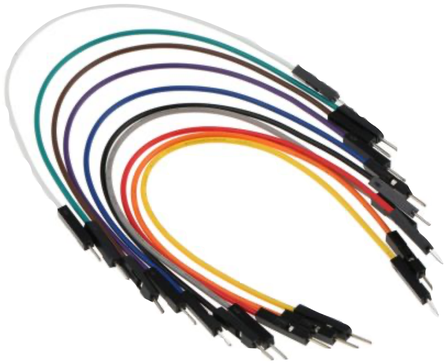


Bela Mini Starter Kit

[shop.bela.io]

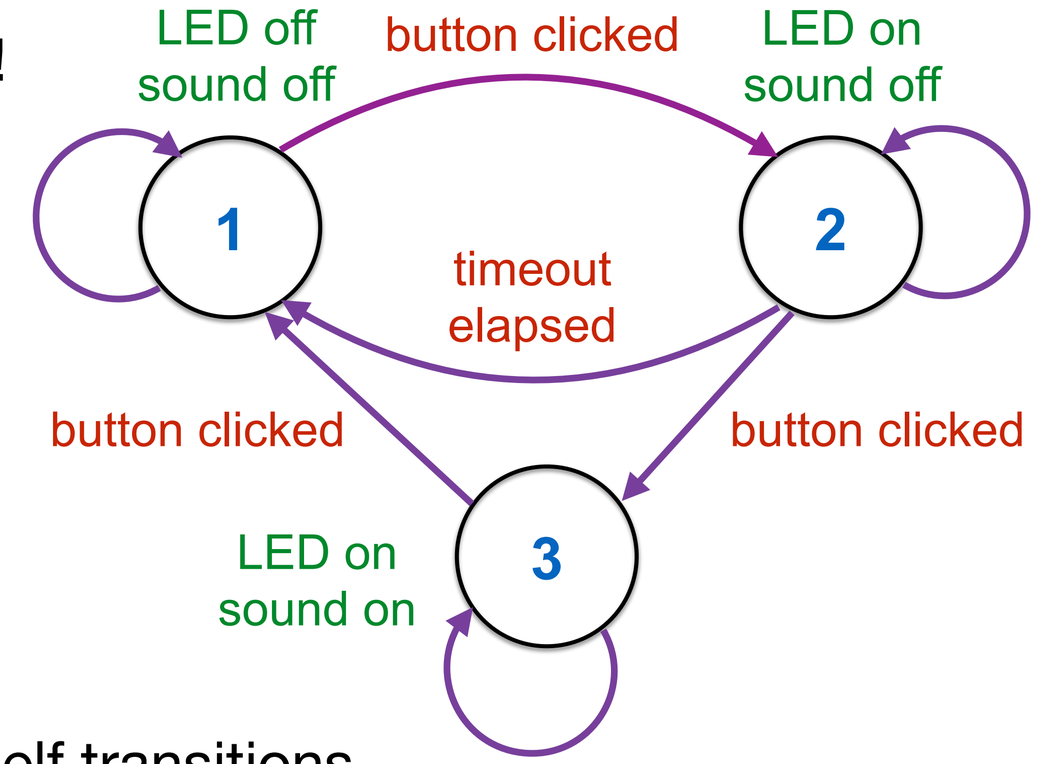
Also needed for this lecture:





Review: state machines

- A finite state machine (or FSM or state machine) is a simple mathematical model of computation
  - Lots of useful systems can be modelled this way!
- Four requirements:
  1. Set of **states** that the system can be in
  2. Set of **transitions** between those states
  3. Set of **inputs** which determine the transitions
  4. Set of **outputs** generated by the machine
- Only a finite number of states, and behaviour is deterministic
  - Transitions can be from one state to another, or self transitions



State machine implementation

Practically speaking, coding a state machine needs:

1. One or more variables to keep track of the state
   - If using a state machine in `render()` or across different functions, you will need global variables to remember the state
   - Could use a single variable to hold all possible states
     - e.g. assign a unique number to each state
     - the variable holds the number of whichever state we're in
   - Or might be conceptually clearer to use multiple variables
     - e.g. a boolean for whether you're in a particular state (on/off for example), plus another variable to say which of several "on" states you might be in
2. A set of `if()` statements to check the state
   - Check which state the system is in, using the variables above
   - Take actions and look for transitions accordingly

```
int gState = 0;
```

```
if(gState == 0) {
    // Actions and transitions
    // for state 0
}
else if(gState == 1) {
    // Actions and transitions
    // for state 1
}
// etc.
```

State machine implementation

- For clarity, it can be useful to name each state
  - As opposed to just giving each one a number
  - Define constants at the top of the code file to associate each name with a number
- C/C++ `enum` keyword is useful here
  - enumerate: define lots of (integer) constants at once
  - Unless otherwise specified, the constants are assigned numbers in increasing order
  - After you define the names, never use the numbers directly in your code
  - Here, the "k" prefix means constant: a coding style, not a requirement

```
enum {
    kStateOpen = 0,
    kStateClosing,  // = 1
    kStateClosed,   // = 2
    kStateOpening   // = 3
};

int gState = kStateOpen;

void myFunction() {
    if(gState == kStateOpen) {
        // ...
    }
    else if(gState == kStateClosing) {
        // ...
    }
    // etc.
}
```

State machine implementation

Practically speaking, coding a state machine needs:

3. Other `if()` statements using inputs to decide on transitions between states
   - Look at the inputs, decide which state to go to next
   - Self-transitions (no state change) are possible, and may even be the most frequent event
   - Remember, time can be one of your inputs
     - Can measured elapsed time by counting samples
     - Remember to reset the counter when needed (e.g. when changing state)

Changing state is easy!
Just need to assign a new value into the state variable, and next time something different will happen

```
if(gState == kStateOpen) {
    motorOff(); // Action for this state
    if(buttonWasPushed) {
        gState = kStateClosing; // Transition
    }
}
else if(gState == kStateClosing) {
    motorDown(); // Action for this state
    if(doorHasClosed) {
        gState = kStateClosed;
    }
}
// etc.
```

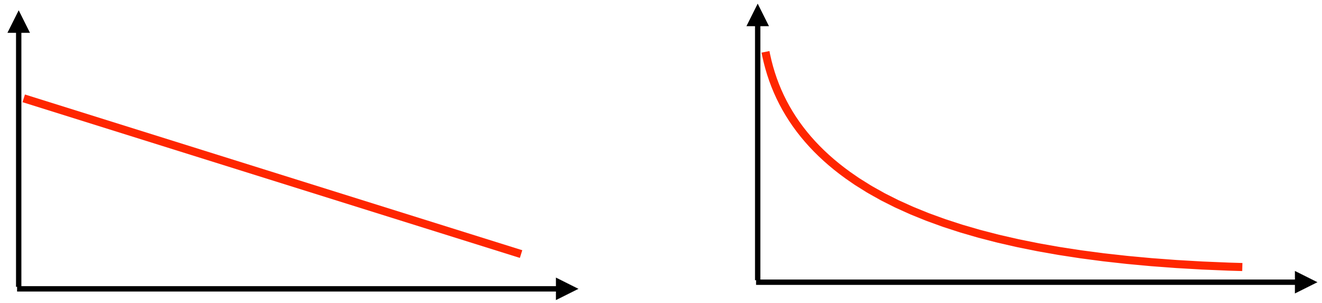
State machine implementation

Practically speaking, coding a state machine needs:

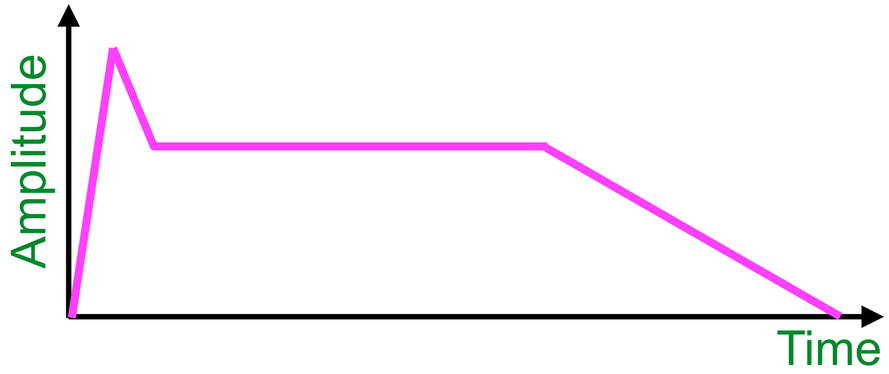
4. Clear planning on when to check for transitions
   - In digital logic, state machine is often driven by a clock
   - Edges of the clock determine when machine checks its state and when it can transition
   - If using real-time audio, audio frame clock will work
     - Check for actions and transitions on each iteration of the `for()` loop in `render()`
     - Each frame, perform actions for a particular state, then decide what state to go to for the next frame
   - Also consider which actions should be performed once on entry to a state (or on transition, or exit), and which should happen every cycle we're in a state

ADSR envelopes

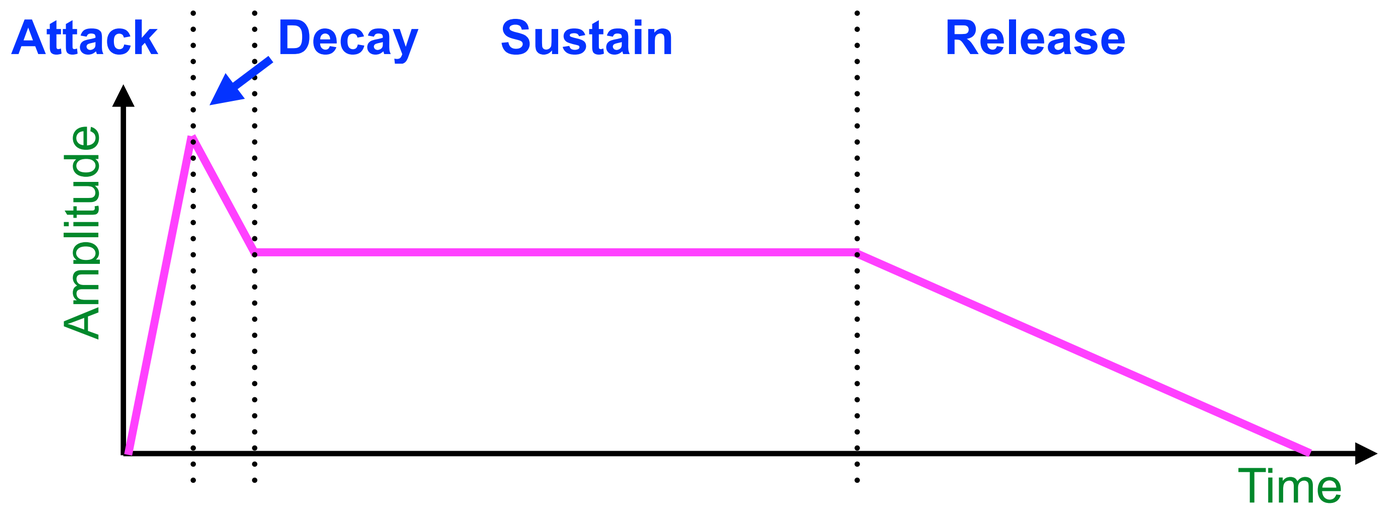
- Envelopes are used to control many different parameters in musical synthesis
  - Frequency, amplitude, filter, ...
  - We previously saw examples of linear and exponential envelopes



- A common and flexible form envelope is the ADSR:
  - **A**ttack, **D**ecay, **S**ustain, **R**elease
  - Simulates the amplitude profile of some acoustic instruments

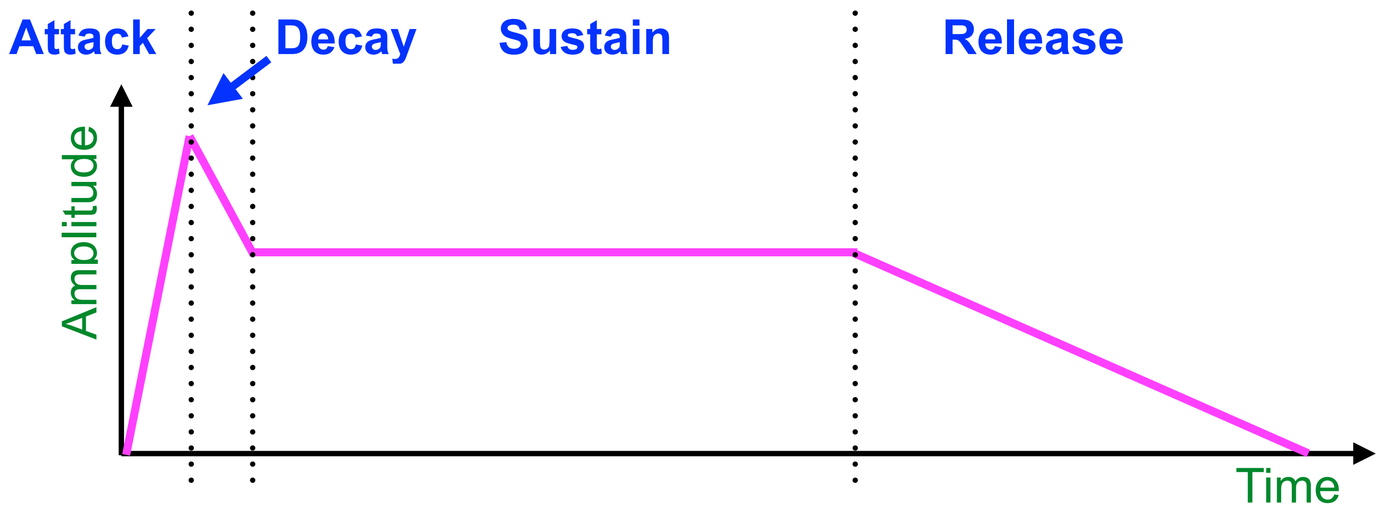


ADSR envelopes



- Four phases (states) to an ADSR envelope:
  - Attack: level rises from 0 to maximum value (typically 1)
  - Decay: level drops from 1 to the sustain level (between 0 and 1)
  - Sustain: level remains constant at the sustain level
  - Release: level drops to 0 over a specified period of time
- The segments are often linear, can also be exponential

ADSR envelope parameters

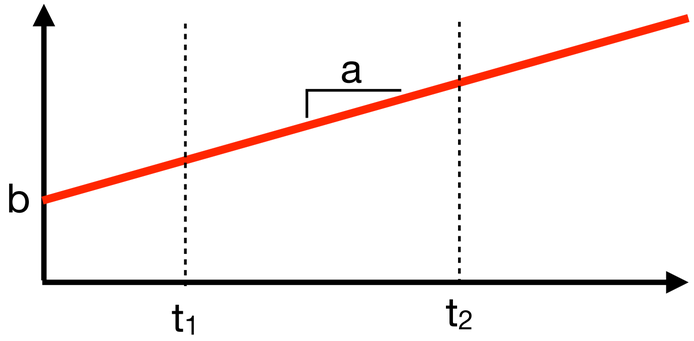


- Four parameters control the ADSR envelope:
  - Attack time: how long to take in the attack state
  - Decay time: how long to take in the decay state
  - Sustain level: level at which to hold the note
  - Release time: how long to take to drop to 0
- Typically a note onset initiates the attack
  - The envelope remains in sustain state until note release

Review: linear envelope

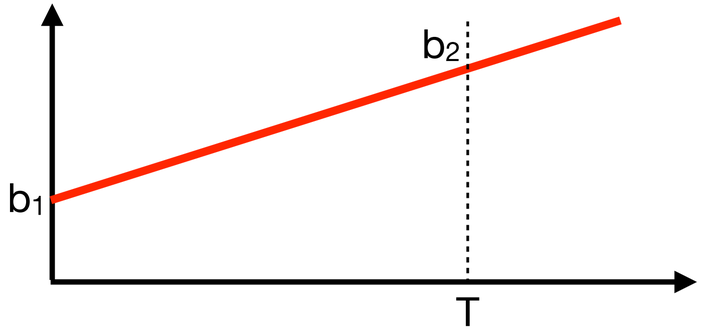
- A linear envelope consists of one or more straight line segments over time
  - Each segment has a constant slope between start and end points:

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



  - 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**:

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

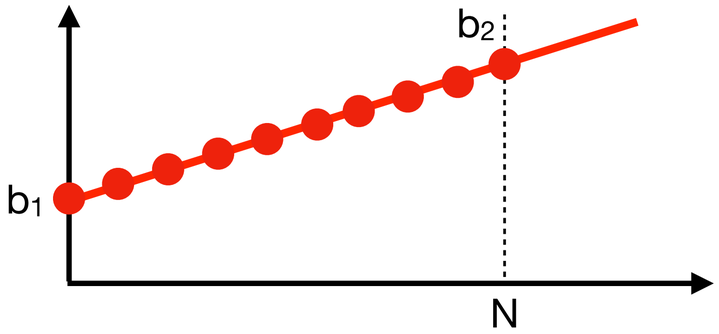


    - This is very similar to what `map()` does!

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)

How do we generate the ramps?

- We made 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

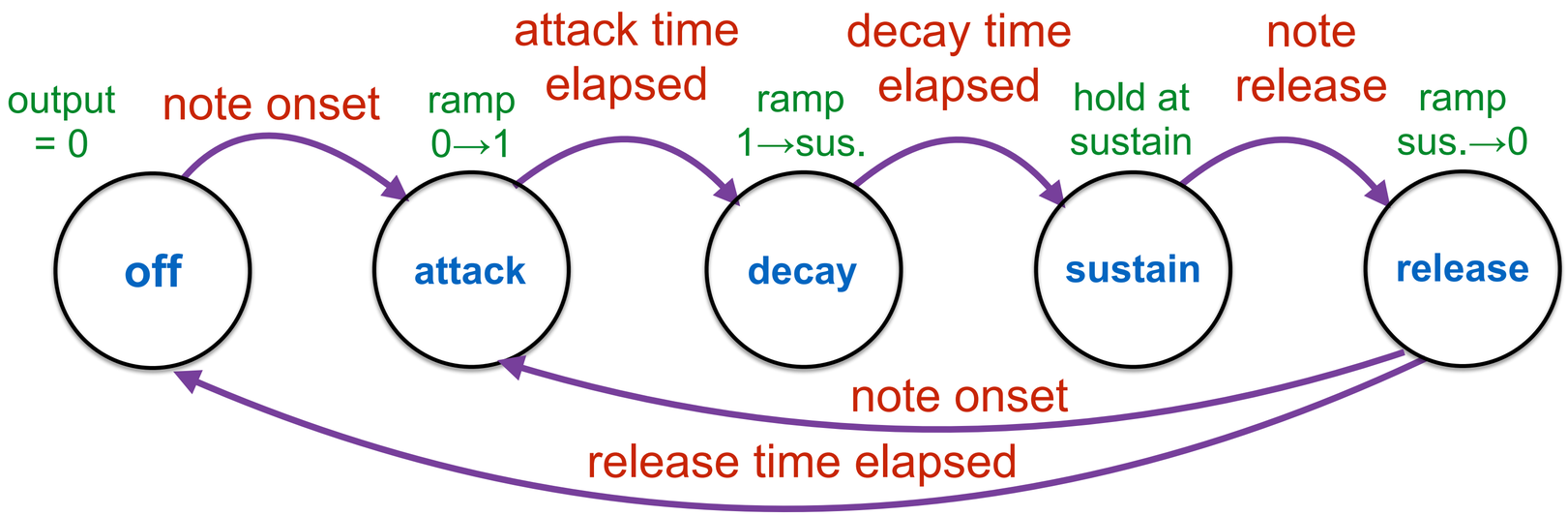
```
void Ramp::setValue(float value);
```

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

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

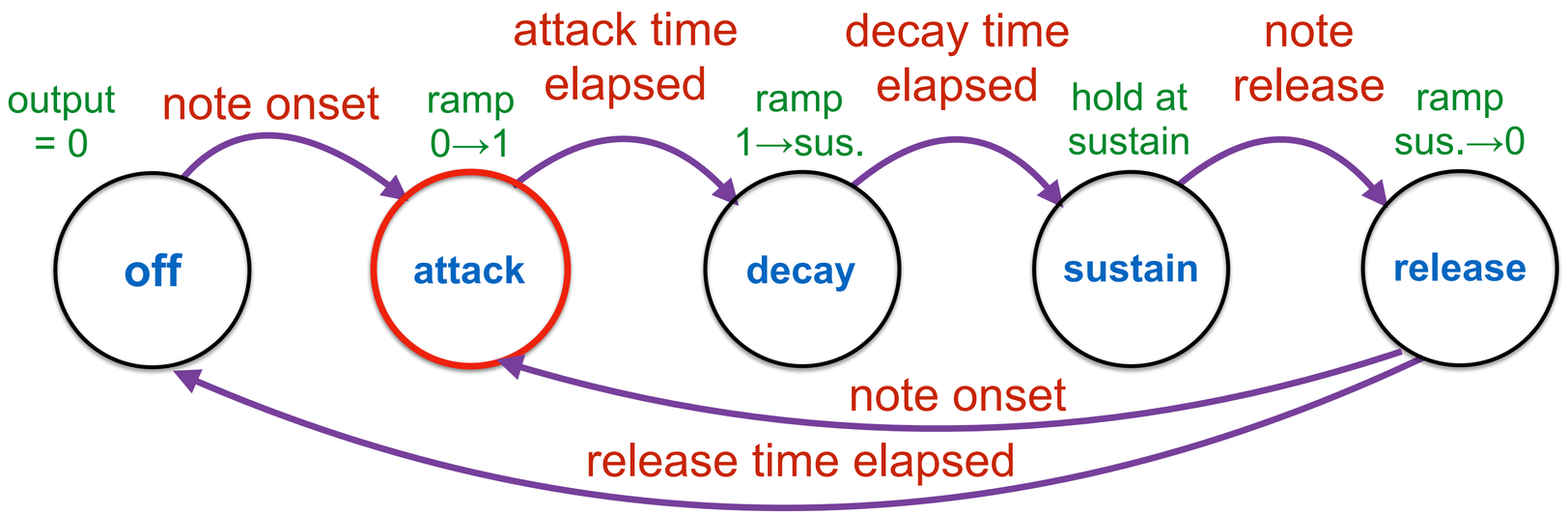
- See Lecture 12 for implementation details

ADSR as a state machine



- How many states? **5:** A, D, S, R, off
- What transitions? Mostly a cycle: off→A, A→D, D→S, S→R, R→off
  - Sometimes also R→A or S→A if a note is retriggered
- What inputs? Note on/off, elapsed time (count samples)
- What outputs? Line segments

ADSR: attack state



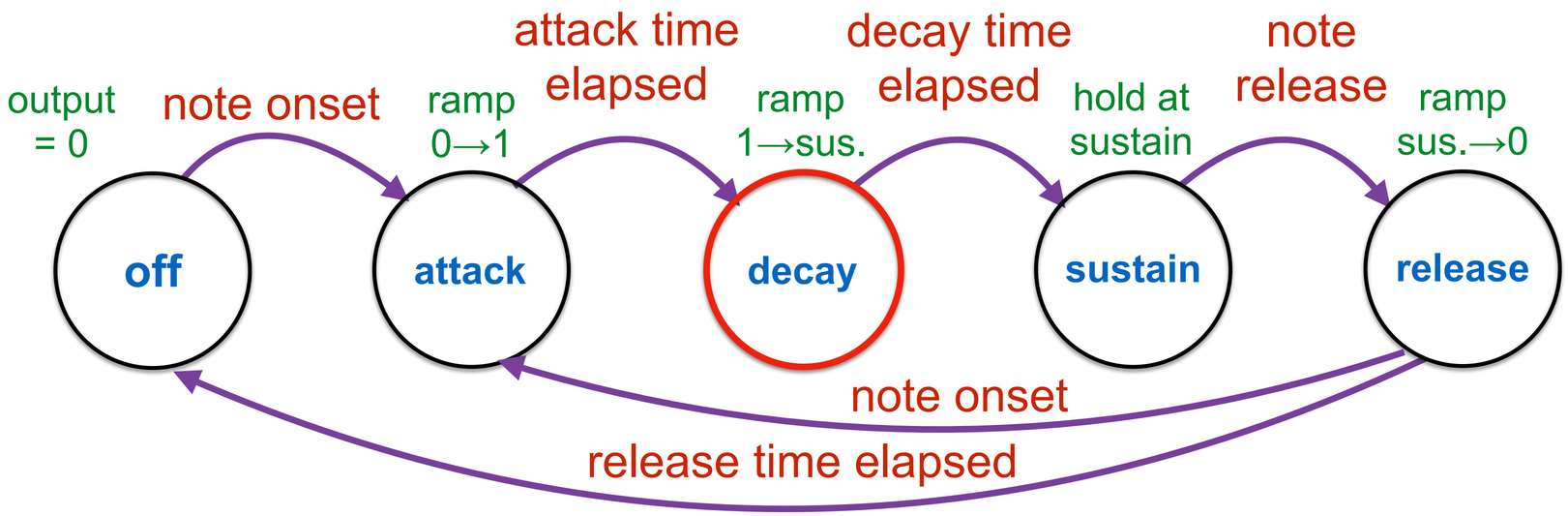
- Triggered by: note onset
- Behaviour: linear ramp from starting value (b_1) to 1.0 (b_2)
  - If we were previously in the off state, ramp starts from 0
  - If we were previously in the sustain or release states, ramp starts from wherever it was on entering the attack state
  - Duration of ramp (N) is given by the attack time parameter
- Calculate the increment on entering the state (or use the `Ramp` class)

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

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

increment

ADSR: decay state



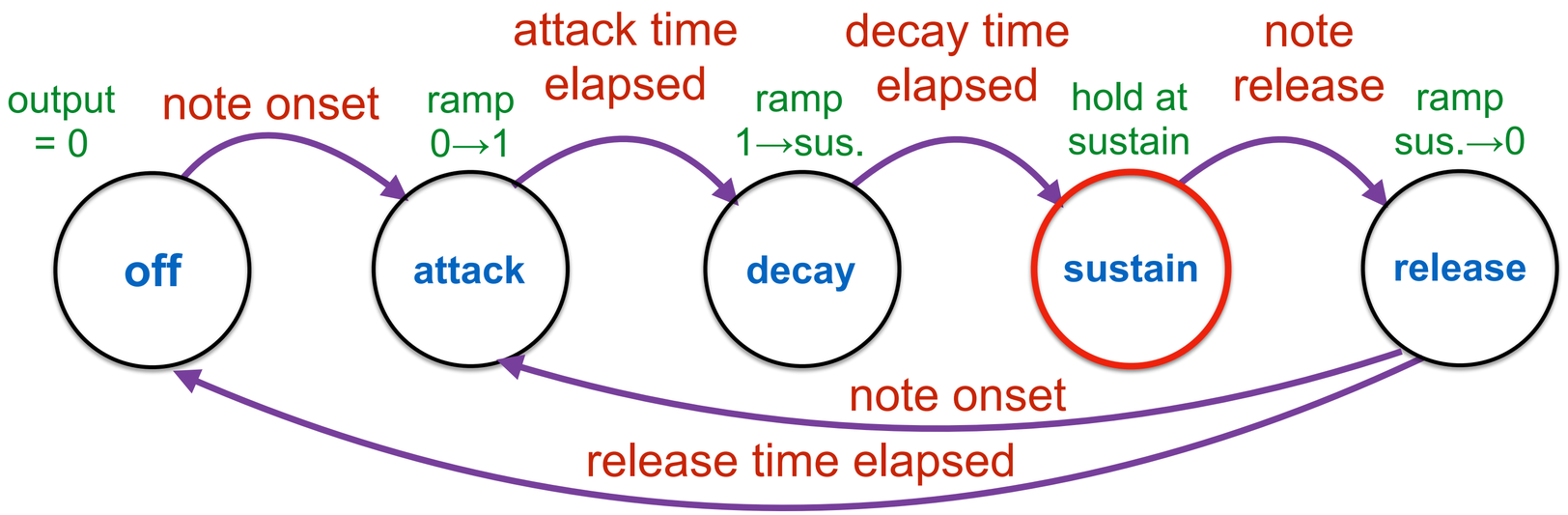
- Triggered by: ramp reaching 1.0 in the attack state
  - Alternatively, by attack time elapsing
- Behaviour: linear ramp from 1.0 (b_1) to sustain level (b_2)
  - We can ignore note on / off messages during this time (though some implementations might to release on note off)
  - Duration of ramp (N) is given by the decay time parameter
- As before, calculate the increment on entry or use `Ramp`

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

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

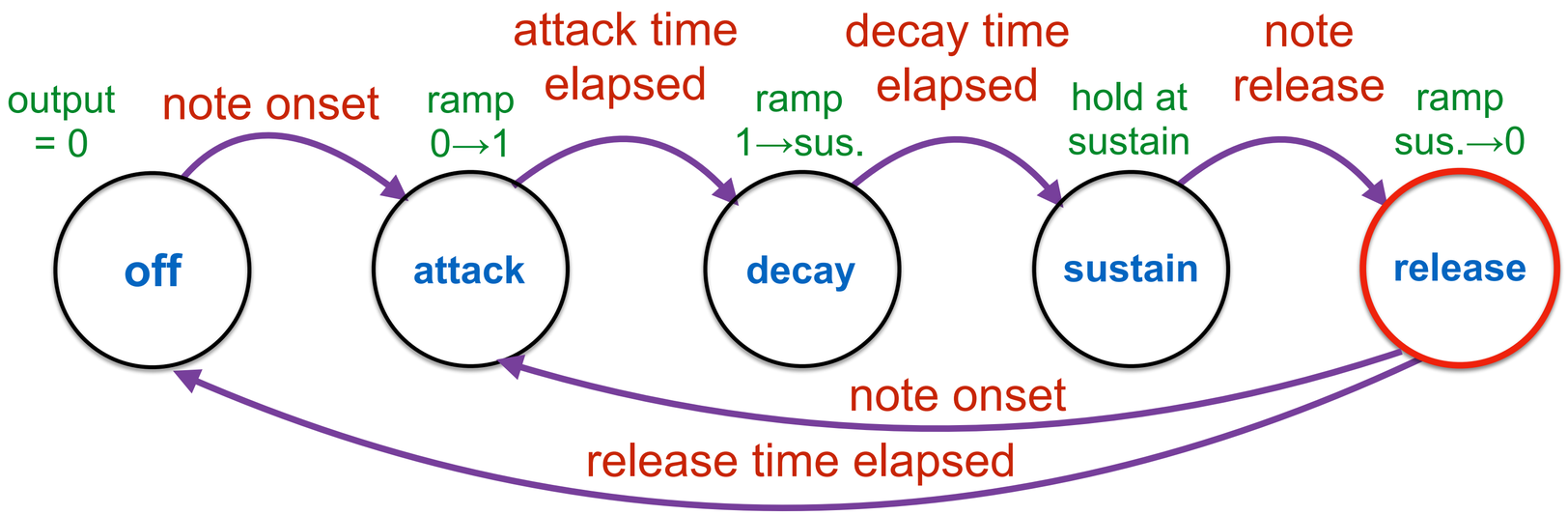
increment

ADSR: sustain state



- Triggered by: ramp reaching the sustain level in the decay state
  - Alternatively, by decay time elapsing
- Behaviour: constant value at the sustain level
  - Wait for a note release (note off) message
  - No need to count samples here; we could wait indefinitely
- In this state, a new note onset could go straight back to attack

ADSR: release state



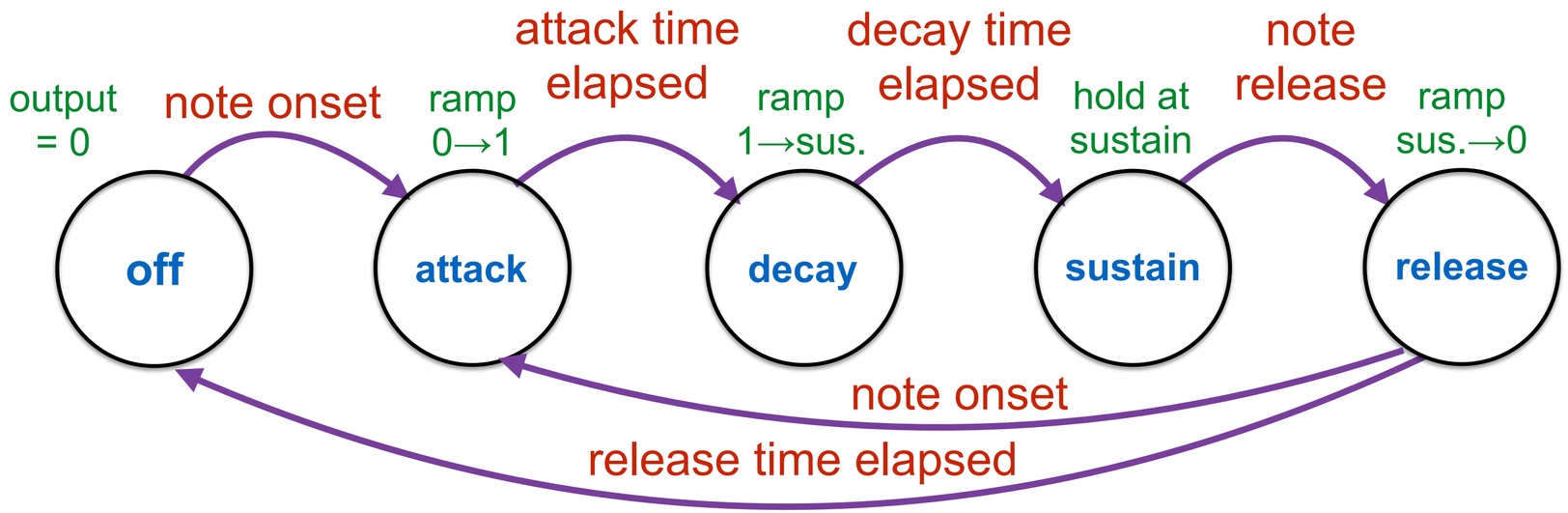
- Triggered by: note release message
- Behaviour: linear ramp from sustain level (b_1) to 0 (b_2)
  - Duration of ramp (N) is given by the release time parameter
  - If we get another note on message, go back to attack
- Calculate the increment or use `Ramp`
  - When the ramp finishes (reaches 0), go to the off state

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

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

increment

ADSR as a state machine



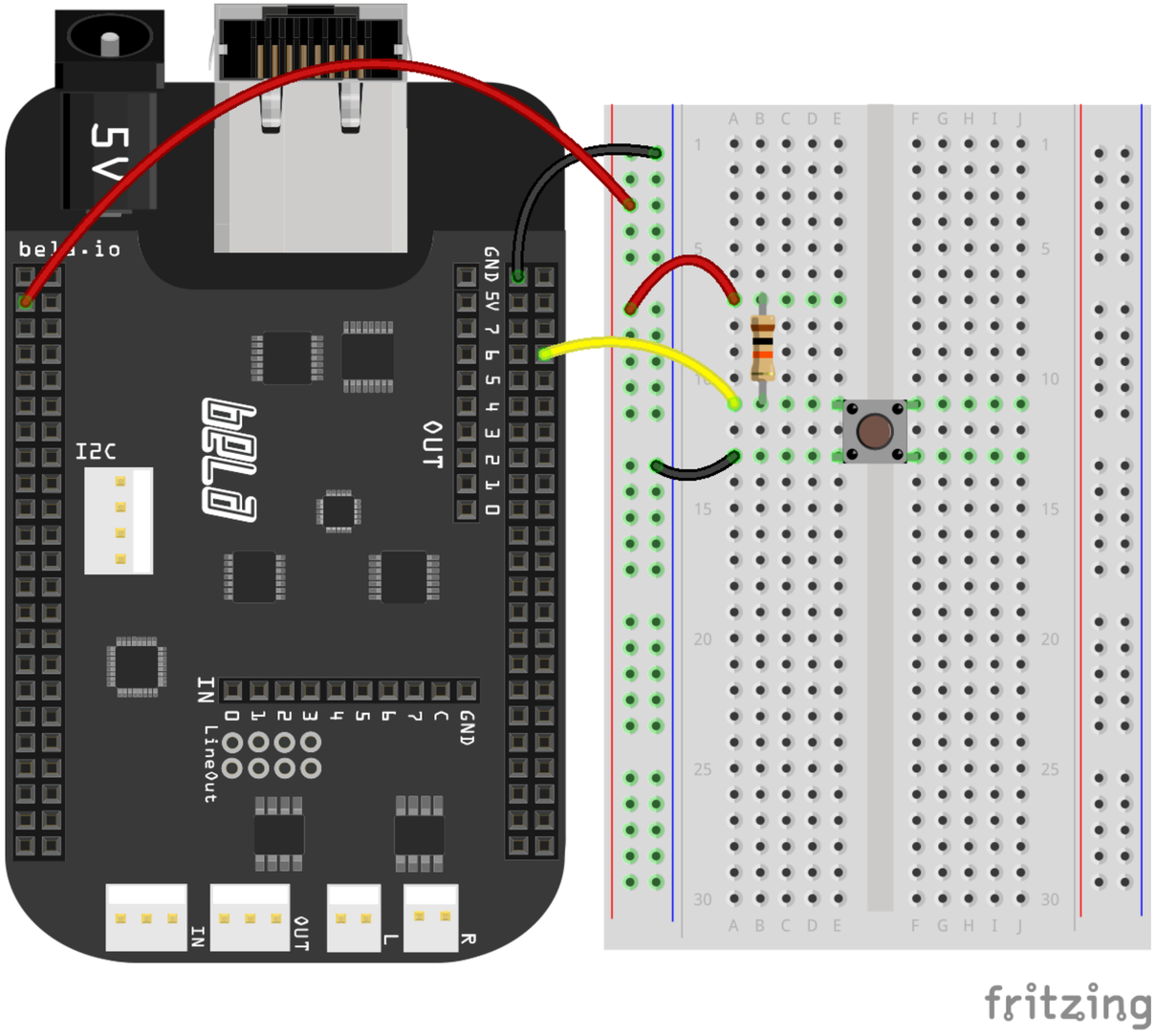
$$v[n] = \frac{b_2 - b_1}{N} n + b_1$$

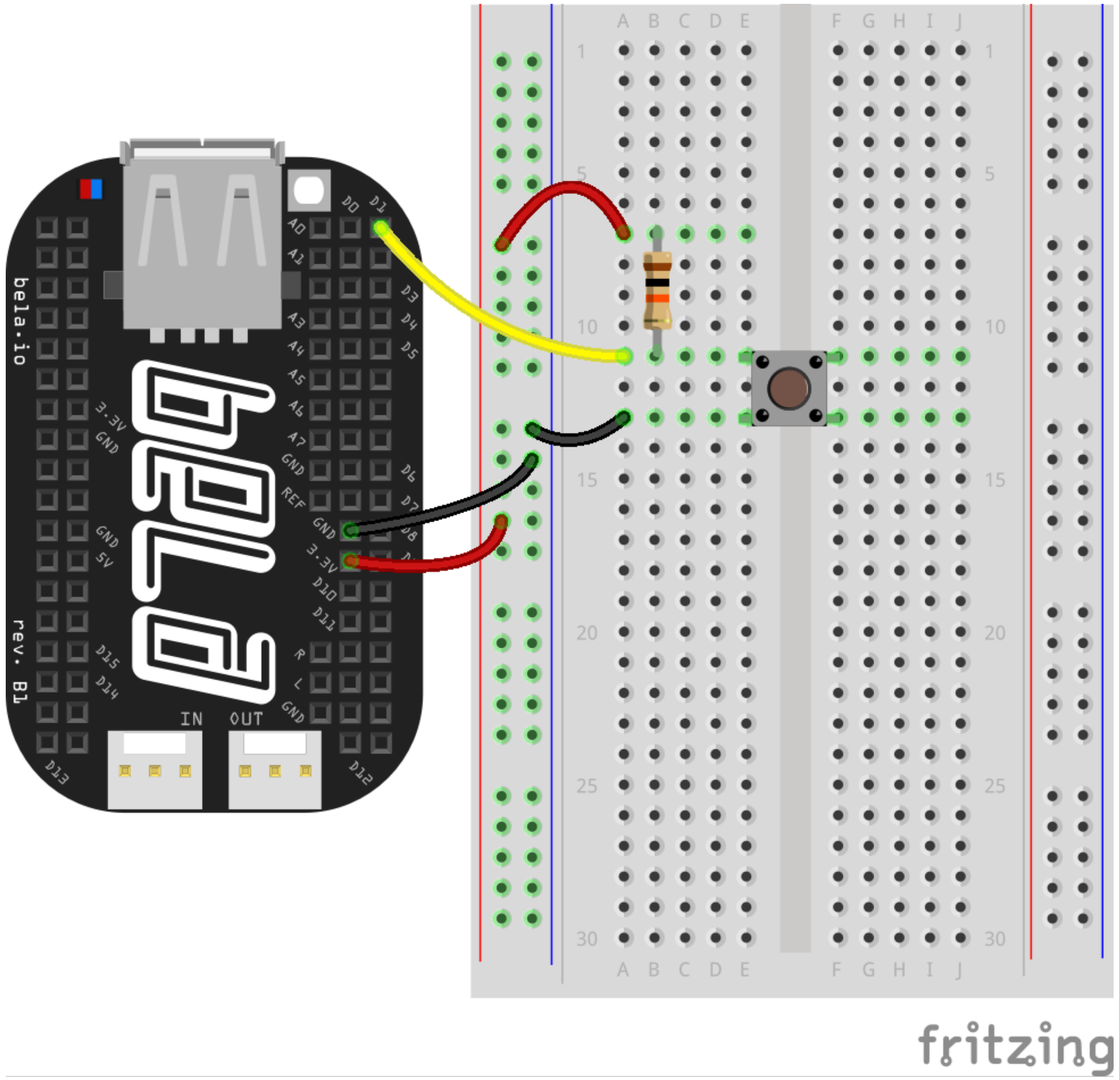
$$v[n+1] = v[n] + \boxed{\frac{b_2 - b_1}{N}}$$

increment

- Task: using the adsr example, implement the ADSR state machine
  - Use the button to trigger the notes (pressed = note on; released = note off)
  - Use the output of the ADSR to control the amplitude of the output
  - Parameters can be changed with browser sliders
  - See the states defined for you in the `enum` statement at the top of the file
  - Each time the state changes, call `Ramp::rampTo()`

ADSR as a state machine





ADSR class

- Next, let's encapsulate the ADSR envelope into its own class
- As with our other audio classes, we want a method to call every audio frame:

```
float ADSR::process();
```

  - This method should return the next sample of the envelope, regardless of its state
  - It will also advance the envelope and possibly change state (A→D, D→S, etc.)
- We also need methods to trigger and release the envelope:

```
void ADSR::trigger();
```

  - Move to the Attack state if we're not already there

```
void ADSR::release();
```

  - Move to the Release state if we're not already there
- Also need methods for setting (and getting) the parameters, e.g.:

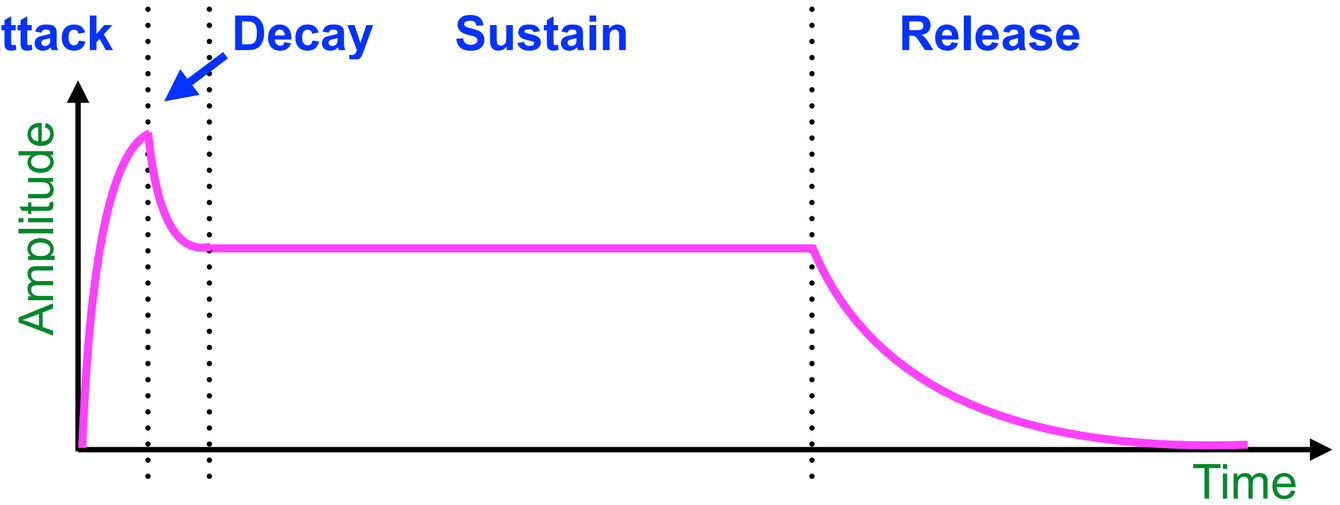
```
void ADSR::setAttackTime(float time);
```

ADSR class

- The `ADSR` class should contain a `Ramp` object to implement the segments
  - It should also involve most of the code you wrote in `render()` before
  - The states can be declared inside the class, rather than the top of the file
- Once finished, it's easy to make multiple `ADSR` objects!
  - For example, many synths have an ADSR on the filter frequency as well as amplitude
- Task: in the adsr-class example
  - Take a look at the methods defined in `ADSR.h`
  - Implement the methods in `ADSR.cpp`
  - In `render.cpp`, use the `ADSR` object to control the amplitude of the waveform
  - Once that works, then make a second ADSR object for filter frequency
    - Don't forget to normalise filter frequency to a sensible range (not 0 to 1!)
    - Use the `Filter` class to implement a resonant filter (see Lecture 12)
  - Optionally, give the two ADSRs separate parameters in the GUI

Exponential ADSR

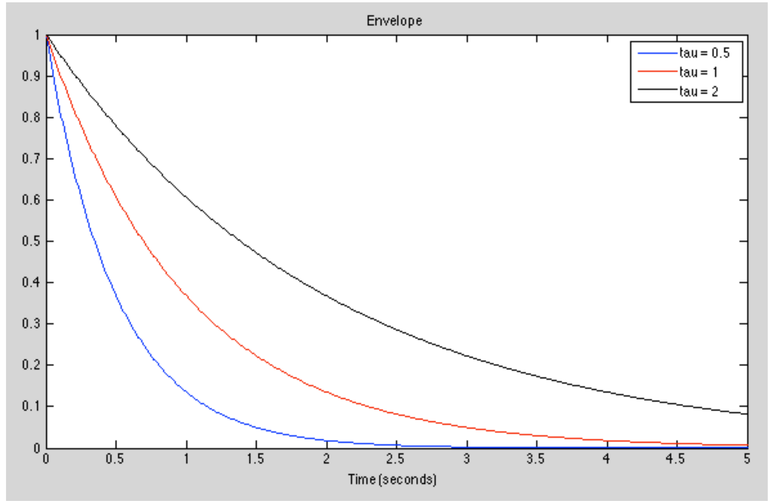
- Our previous ADSR implementation used linear segments
- It's also possible to implement ADSR with exponential segments
  - Particularly during decay and release, this may sound more natural for amplitude control



- What information do we need to implement one exponential segment?
  - Starting value
  - Destination value
  - Time constant

Review: 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: $v(t) = e^{-t/\tau}$
  - τ is called the time constant
  - Larger values of τ 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 $a = e^{1/\tau}$

Exponential envelope, real time

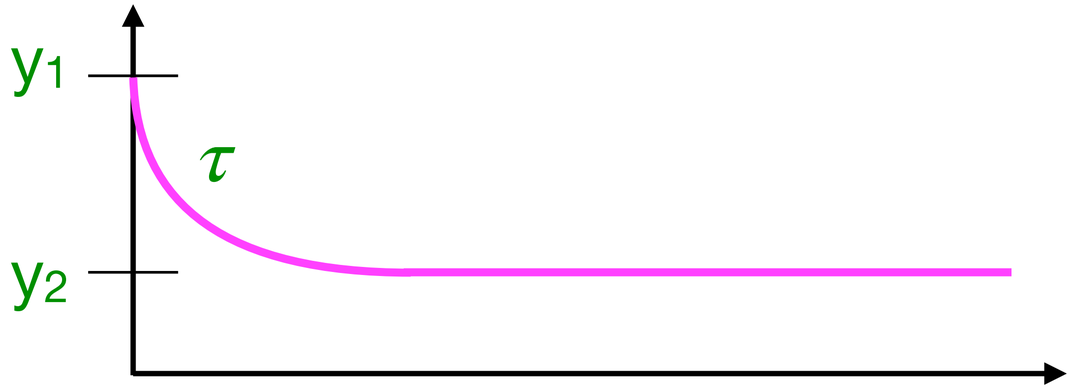
- How should we implement $v[n]$ in discrete time, also in real time?
- We could calculate samples of $v(t)$ every time
  - But this is inefficient: need to calculate an exponential function on every sample
    - Potentially many multiplies per sample
  - Better: $v[n] = b^n$ which means: $v[0] = 1, v[n] = bv[n-1]$
    - 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**

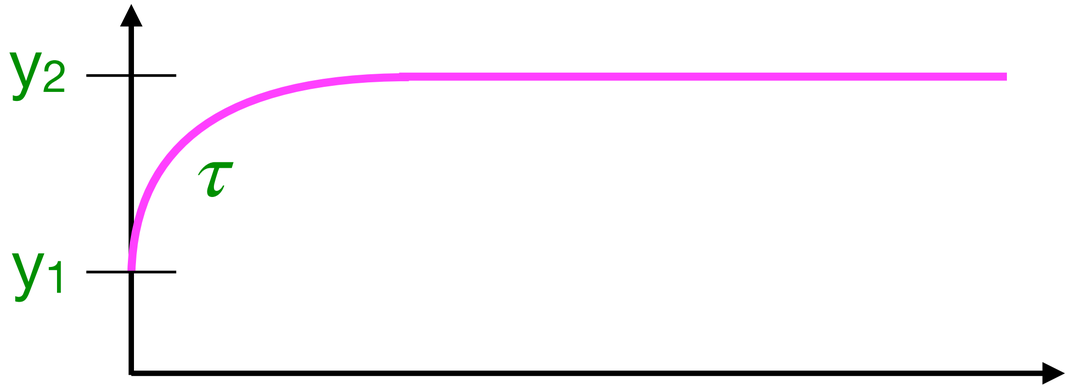
$$v[f_s] = v(1) \implies b^{f_s} = \frac{1}{a} \longrightarrow b = \sqrt[f_s]{1/a} = \sqrt[f_s]{e^{-1/\tau}}$$

  - For example: τ = .5, f_s = 1000Hz gives *b* = .998
  - For reasons of precision, it is often useful to use type `double` rather than `float`

Exponential envelope endpoints

- How do we make the exponential envelope decay to a value other than 0?





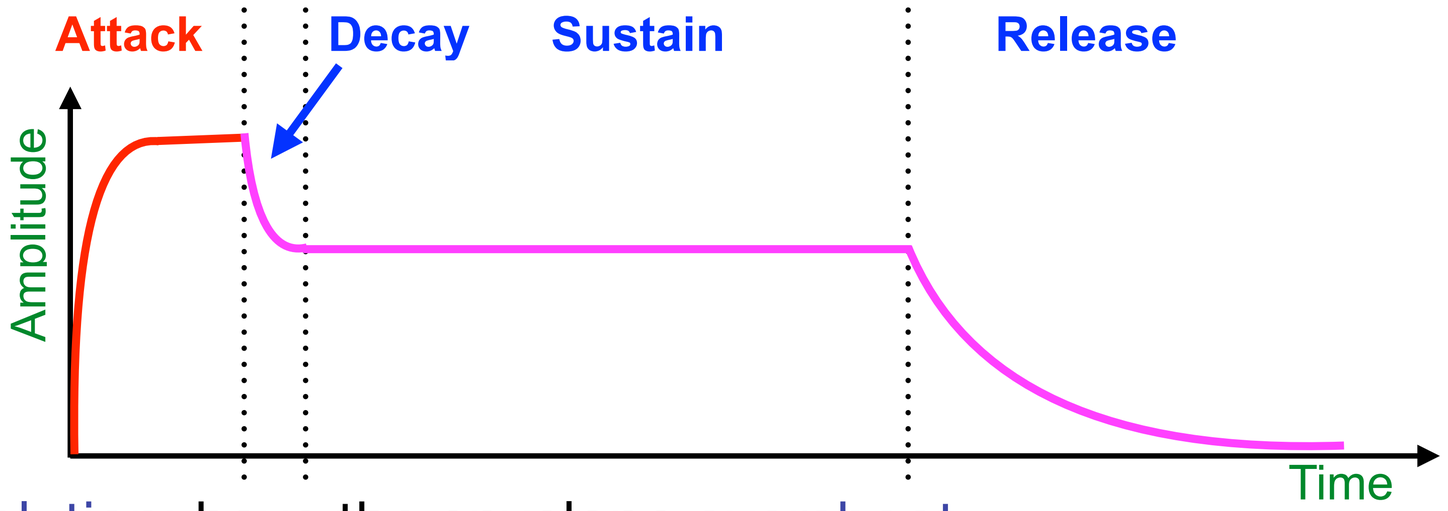
- Separate the signal into two components:
  - Constant offset y_2
  - Decaying component starting at a value of y_1 - y_2

$$v[0] = y_1 - y_2$$
$$v[n] = bv[n-1]$$
$$y[n] = y_2 + v[n]$$

- Similar idea for rising segments
  - Notice that y_1 - y_2 is negative, so decaying toward 0 means increasing in value
  - We can use the same equations

Exponential envelope endpoints

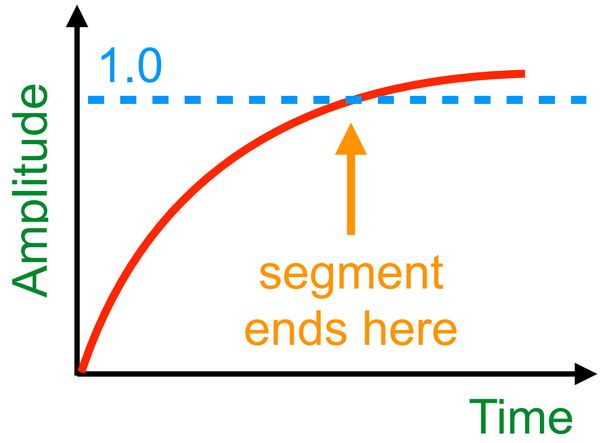
- Problem: exponential envelopes never reach their endpoint!
  - They only asymptotically approach it
  - This could be a problem for knowing when an ADSR segment is finished
- The problem is most critical for the Attack state
  - We don't want a shape like this:



  - Solution: have the envelope overshoot
  - Aim for a target higher than 1.0, and then switch states when the value crosses 1.0

Exponential envelope endpoints

- How much overshoot should we introduce per segment?
  - The larger the overshoot, the more linear the segment becomes
  - For the attack segment, let's say a ratio of 1.01 (1% beyond target) (but you can experiment with different values)



- For the decay and release segments, overshoot isn't strictly required at all
  - These ADSR segments don't actually need to have an endpoint!
  - We could remain in the decay state until the transition to release (skipping the sustain state)
  - We could remain in the release state until the next attack segment (skipping the off state)
- There's a subtle efficiency advantage to having the segment end
  - As the decaying exponential component approaches 0, it will eventually become small enough that to represent it with a `float` requires a denormal value
    - the value is smaller than the smallest available exponent in the 32-bit value
  - Some FPUs process denormals slowly, so it's better to avoid them if we can

Exponential ADSR implementation

- Let's create a new class: `ExponentialSegment`
  - It will have all the same methods as `Ramp`, and we'll use it in place of `Ramp` within `ADSR`
  - This could later let us explore class inheritance (though we won't do that today)
- The `process()` method will work differently for an exponential segment
- We will need to calculate internal time constants based on segment length
  - How should we calculate τ to reach a value 1.0/1.01 in *T* seconds?

$$y(t) = 1.01(1 - e^{-T/\tau}) = 1.0 \rightarrow e^{-T/\tau} = 1 - \frac{1.0}{1.01} \rightarrow \tau = \frac{-T}{\ln(1 - \frac{1.0}{1.01})} \approx .217T$$

  - Now back to discrete time using the formula: $b = \sqrt[f_s]{e^{-1/\tau}}$

$$v[0] = y_1 - y_2 \quad \text{where} \quad y_1 = 0, y_2 = 1.01$$
$$v[n] = bv[n-1]$$
$$y[n] = y_2 + v[n]$$

Exponential ADSR implementation

- Task: in the adsr-class example, create the `ExponentialSegment` class
  - It should have all the same methods as `Ramp`
  - The implementation of `process()` and `rampTo()` will be significantly different
  - Also provide an extra argument to `rampTo()` to specify the overshoot ratio
- Use these equations:

$$v[0] = y_1 - y_2$$
$$v[n] = bv[n-1]$$
$$y[n] = y_2 + v[n]$$

$$b = \sqrt[f_s]{e^{-1/\tau}} \qquad \tau = \frac{-T}{\ln(1 - 1/R)}$$

where *R* is the overshoot ratio
(1.01 for attack, 1.001 for decay/release)

- You'll need at least four instance variables:
  - *v*[*n*]: the value of the decaying exponential
  - *b*: the multiplier of *v*[*n*] each frame, calculated based on length and overshoot of each segment
  - y_2: the asymptote of the segment (including the overshoot; e.g. y_2 = 1.01 for attack)
  - Target value for the segment (not including the overshoot), for knowing when segment ends
- Don't forget to update `ADSR.h` to use the `ExponentialSegment` class

Keep in touch!

Social media:

@BelaPlatform

forum.bela.io

blog.bela.io

More resources and contact info at:

learn.bela.io/resources