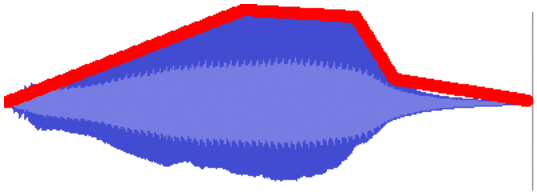
MUSC 208 Winter 2014
John Ellinger Carleton College

Fun With Envelopes

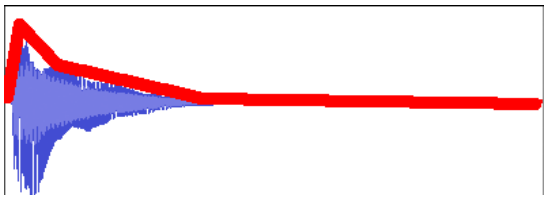
The term envelope in digital audio is generally applied to the overall amplitude shape of a single musical note. The envelopes of three common instruments are outlined in red.

Horn

Cello



Guitar



Performers on wind and string instruments like the the Horn and Cello can control the shape of the amplitude envelope. Performers on percussion and plucked or struck strings like the snare drum, piano, and guitar have no control over the shape of the amplitude envelope. It's built into the instrument.

Why Use Envelopes

Enter and run this code. You should hear four notes played.

```
// fourNotes1.ck
// MUSC 208 Lab 9
SinOsc s => dac;
.25 => s.gain;

Std.mtof( 60 ) => s.freq; // Middle C
1::second => now;

Std.mtof( 62 ) => s.freq; // D
1::second => now;

Std.mtof( 64 ) => s.freq; // E
1::second => now;

Std.mtof( 60 ) => s.freq; // Middle C
1::second => now;
```

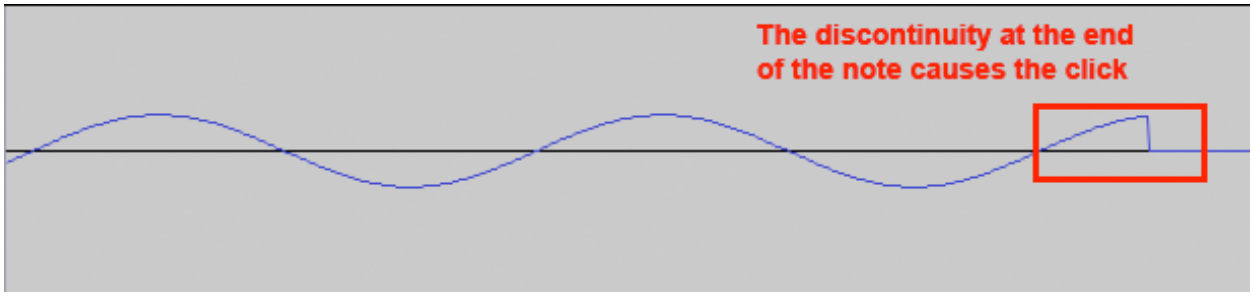
Change all four notes to Middle C and play again.

```
// fourNotes2.ck
// MUSC 208 Lab 9
SinOsc s => dac;
.25 => s.gain;

Std.mtof( 60 ) => s.freq; // Middle C
1::second => now;
Std.mtof( 60 ) => s.freq; // Middle C
1::second => now;
Std.mtof( 60 ) => s.freq; // Middle C
1::second => now;
Std.mtof( 60 ) => s.freq; // Middle C
1::second => now;
```

Did you hear all four notes? No, you heard one continuous frequency for 4 seconds. Let's see if we can turn the notes on and off by changing on and off times and on and off gains.

```
// fourNotes3.ck
// MUSC 208 Lab 9
SinOsc s => dac;
dac => WvOut w => blackhole;
w.wavFilename( me.sourceDir() + "/fourNotes3.wav" );

// Note on 3/4 second, Note off 1/4 second
750::ms => dur onTime;
1::second - 750::ms => dur offTime;

// Note on gain of .25, Note off gain of zero
.25 => float onGain;
0 => float offGain;

Std.mtof( 60 ) => s.freq; // Middle C
onGain => s.gain;
onTime => now;
offGain => s.gain;
offTime => now;

Std.mtof( 60 ) => s.freq; // Middle C
onGain => s.gain;
onTime => now;
offGain => s.gain;
offTime => now;

Std.mtof( 60 ) => s.freq; // Middle C
onGain => s.gain;
onTime => now;
offGain => s.gain;
offTime => now;

Std.mtof( 60 ) => s.freq; // Middle C
onGain => s.gain;
onTime => now;
offGain => s.gain;
offTime => now;

// workaround to automatically close file on remove-shred
null @=> w;
```

This sort of worked but produced clicks between the notes.



Try setting the off frequency to zero at the end of the note to remove clicks.

```
// fourNotes4.ck
// MUSC 208 Lab 9
SinOsc s => dac;

// Note on 3/4 second, Note off 1/4 second
750::ms => dur onTime;
1::second - 750::ms => dur offTime;
// Note on gain of .25, Note off gain of zero
.25 => float onGain;
0 => float offGain;
// Note off frequency of zero
0.0 => float offFreq;

for ( 0 => int ix; ix < 4; ix++ )
{
    Std.mtof( 60 ) => s.freq; // Middle C
    onGain => s.gain;
    onTime => now;
    offFreq => s.freq;
    offGain => s.gain;
    offTime => now;
}
```

Didn't seem to help. Clicks are still present.

ChucK Envelope Class To The Rescue

[ugen]: **Envelope** (STK Import)

- *STK envelope base class.*
- *see examples: envelope.ck*

```
This class implements a simple envelope
generator which is capable of ramping to
a target value by a specified \e rate.
It also responds to simple \e keyOn and
\e keyOff messages, ramping to 1.0 on
keyOn and to 0.0 on keyOff.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.
```

(control parameters)

- **.keyOn** - (int , WRITE only) - *ramp to 1.0*
- **.keyOff** - (int , WRITE only) - *ramp to 0.0*
- **.target** - (float , READ/WRITE) - *ramp to arbitrary value.*
- **.time** - (float , READ/WRITE) - *time to reach target (in seconds)*
- **.duration** - (dur , READ/WRITE) - *duration to reach target*
- **.rate** - (float , READ/WRITE) - *rate of change*
- **.value** - (float , READ/WRITE) - *set immediate value*

http://chuck.cs.princeton.edu/doc/program/ugen_full.html#Envelope

Envelope 1

Uses the keyOn and keyOff methods. No clicks.

```
// envelope1.ck
// MUSC 208 Lab 9
SinOsc s => Envelope env => dac;

// Note on 3/4 second, Note off 1/4 second
750::ms => dur onTime;
1::second - 750::ms => dur offTime;

for ( 0 => int ix; ix < 4; ix++ )
{
    Std.mtof( 60 ) => s.freq;
    env.keyOn();
    onTime => now;
    // turn note off
    env.keyOff();
    offTime => now;
}
```

Envelope 2

Use the duration and target methods to change the envelope.

```
// envelope2.ck
// John Ellinger Music 208 Winter 2014
SinOsc s => Envelope env => dac;
// Note on 3/4 second, Note off 1/4 second
750::ms => dur onTime;
1::second - 750::ms => dur offTime;

// play four quarter notes
for ( 0 => int ix; ix < 4; ix++ )
{
    Std.mtof( 60 ) => s.freq;
    // turn note on
    .25 => env.target; // equivalent to s.gain
    onTime => now;
    // turn note off
    0.0 => env.target;
    offTime => now;
}
```

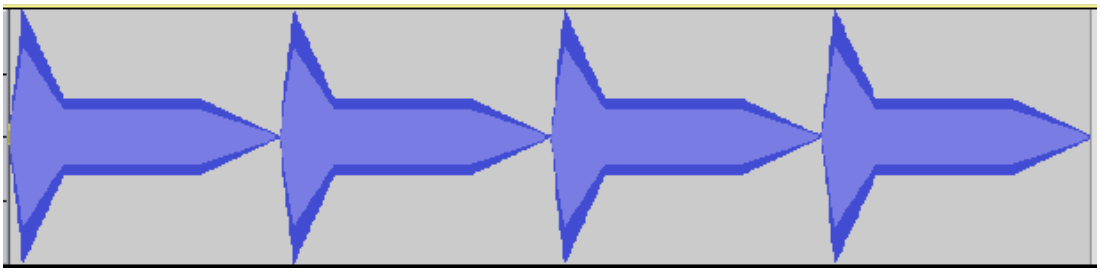
Envelope 3

Create an ADSR envelope. The ADSR envelope problem in Homework 5-6 treated the envelope like a wind or string instrument where the shape of the envelope was controlled as a percentage of the overall length of the note. This solution is not be appropriate for piano and guitar because their envelopes have a sharp attack and rapid decay that is not affected by how long the note lasts.

Enter and run this code.

```
// envelope3.ck
// John Ellinger Music 208 Winter 2014
SinOsc s => Envelope env => dac;
dac => WvOut w => blackhole;
w.wavFilename( me.sourceDir() + "/envelope3.wav" );

// Total duration of ADSR 1::second
50::ms => dur Attack;
150::ms => dur Decay;
500::ms => dur Sustain;
300::ms => dur Release;

// play four quarter notes
for ( 0 => int ix; ix < 4; ix++ )
{
    Std.mtof( 60 ) => s.freq;
    // turn note on
    1.0 => env.target; // equivalent to s.gain
    Attack => env.duration;
    Attack => now;
    .3 => env.target;
    Decay => env.duration;
    Decay => now;
    .3 => env.target;
    Sustain => env.duration;
    Sustain => now;
    0.0 => env.target;
    Release => env.duration;
    Release => now;
}
```

Waveform viewed in Audacity



ChucK's ADSR Class

[ugen]: **ADSR** (STK Import)

- *STK ADSR envelope class.*
- *see examples: adsr.ck*

```
This Envelope subclass implements a
traditional ADSR (Attack, Decay,
Sustain, Release) envelope.  It
responds to simple keyOn and keyOff
messages, keeping track of its state.
The \e state = ADSR::DONE after the
envelope value reaches 0.0 in the
ADSR::RELEASE state.

by Perry R. Cook and Gary P. Scavone, 1995 - 2002.
```

extends Envelope

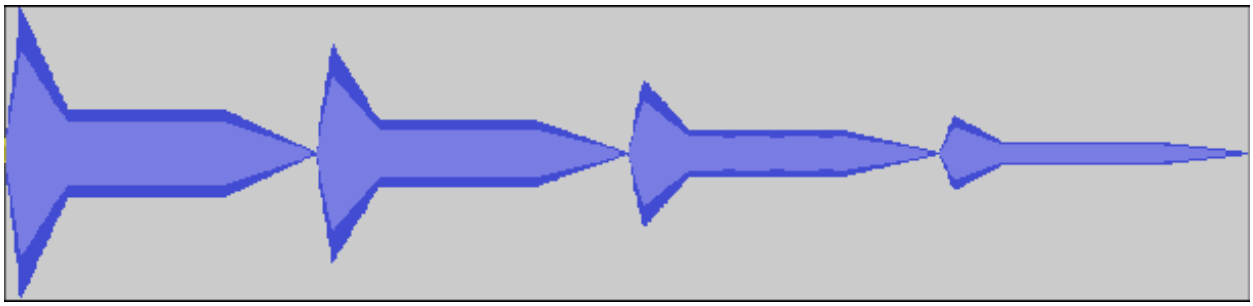
(control parameters)

- **.keyOn** - (int , WRITE only) - *start the attack for non-zero values*
- **.keyOff** - (int , WRITE only) - *start release for non-zero values*
- **.attackTime** - (dur , READ/WRITE) - *attack time*
- **.attackRate** - (float , READ/WRITE) - *attack rate*
- **.decayTime** - (dur , READ/WRITE) - *decay time*
- **.decayRate** - (float , READ/WRITE) - *decay rate*
- **.sustainLevel** - (float , READ/WRITE) - *sustain level*
- **.releaseTime** - (dur , READ/WRITE) - *release time*
- **.releaseRate** - (float , READ/WRITE) - *release rate*
- **.state** - (int , READ only) - *attack=0, decay=1 , sustain=2, release=3, done=4*
- **.set** - (dur, dur, float, dur) - *set A, D, S, and R all at once*

http://chuck.stanford.edu/doc/program/ugen_full.html#Envelope

ADSR Example 1

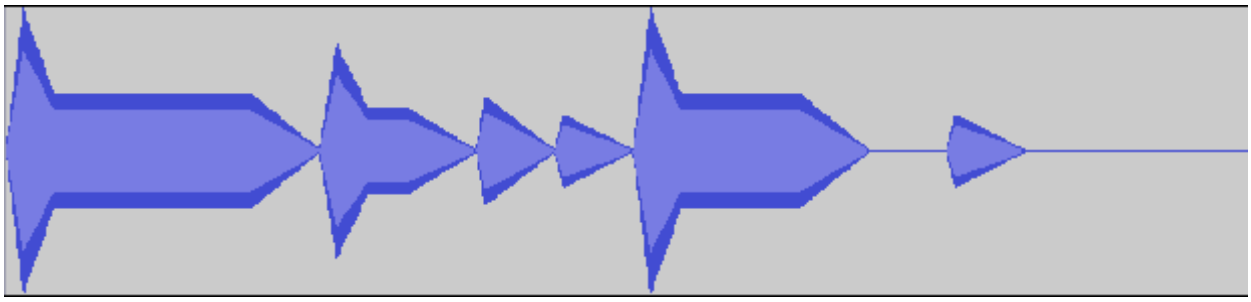
The attack, decay, and release values in ChucK's ADSR envelope class are durations in seconds. The sustain value is a float acts as a percentage of the current gain value. ADSR example 1 plays four notes decreasing in volume, all using the same envelope shape.



```
// adsr1.ck
// John Ellinger Music 208 Winter 2014
SinOsc s => ADSR adsr => dac;
dac => WvOut w => blackhole;
w.wavFilename( me.sourceDir() + "/adsr1.wav" );

// Total duration of ADSR 1::second
50::ms => dur atk;
150::ms => dur dk;
0.3 => float susLvl; // amplitude 0 - 1.0
300::ms => dur rls;
adsr.set( atk, dk, susLvl, rls );

1.0 => s.gain;
0.25 => float gainDelta;
for ( 0 => int ix; ix < 4; ix++ )
{

    Std.mtof( 60 ) => s.freq;
    if ( ix > 0 )
        s.gain() - gainDelta => s.gain;
    1000::ms => dur noteDur;
    adsr.keyOn();
    noteDur - rls => now;
    adsr.keyOff();
    rls => now;
}
```

ADSR Example 2

ADSR example 2 repeats a one measure rhythm four times. The notes are of different durations but all notes use the same envelope shape.

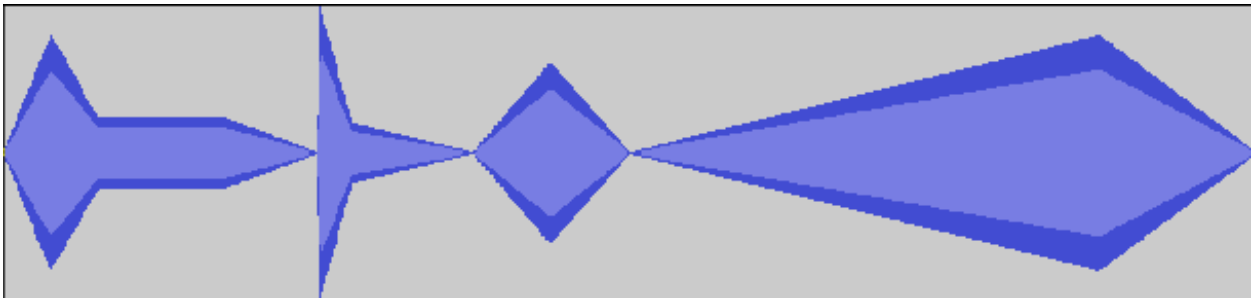


```
// adsr2.ck
// John Ellinger Music 208 Winter 2014
SinOsc s => ADSR adsr => dac;
dac => WvOut w => blackhole;
w.wavFilename( me.sourceDir() + "/adsr2.wav" );

// set adsr values
50::ms => dur atk;
100::ms => dur dk;
0.4 => float susLvl; // amplitude as percent of s.gain
225::ms => dur rls;
adsr.set( atk, dk, susLvl, rls );

0 => int count;
while ( count < 4 ) // repeat four times
{
    Std.mtof( 60 ) => s.freq;
    1.0 => s.gain;
    1000::ms => dur noteDur; // quarter note at tempo of 60
    adsr.keyOn();
    noteDur - rls => now;
    adsr.keyOff();
    rls => now;

    Std.mtof( 60 ) => s.freq;
    0.75 => s.gain;
    500::ms => noteDur; // eighth note at tempo of 60
```

```
    adsr.keyOn();
    noteDur - rls => now;
    adsr.keyOff();
    rls => now;

    Std.mtof( 60 ) => s.freq;
    0.75 => s.gain;
    250::ms => noteDur; // eighth note at tempo of 60
    adsr.keyOn();
    noteDur - rls => now;
    adsr.keyOff();
    rls => now;

    Std.mtof( 60 ) => s.freq;
    0.5 => s.gain;
    250::ms => noteDur; // eighth note at tempo of 60
    adsr.keyOn();
    noteDur - rls => now;
    adsr.keyOff();
    rls => now;

    Std.mtof( 60 ) => s.freq;
    1.0 => s.gain;
    750::ms => noteDur; // quarter note at tempo of 60
    adsr.keyOn();
    noteDur - rls => now;
    adsr.keyOff();
    rls => now;
    250::ms => now;  // time is multiples of 1000 here

    Std.mtof( 60 ) => s.freq;
    0.5 => s.gain;
    250::ms => noteDur; // quarter note at tempo of 60
    adsr.keyOn();
    noteDur - rls => now;
    adsr.keyOff();
    rls => now;
    750::ms => now;  // time is multiples of 1000 here
    count++;
}
```

ADSR Example 3

ADSR example 3 plays four notes of different durations and different envelope shapes. See if you can create them.



Fill in the ? mark values in the code below to create these four envelope shapes.

```
// adsr4.ck
// John Ellinger Music 208 Winter 2014
SinOsc s => ADSR adsr => dac;
dac => WvOut w => blackhole;
w.wavFilename( me.sourceDir() + "/adsr4.wav" );

// Note 1
Std.mtof( 60 ) => s.freq;
1000::ms => dur noteDur; // quarter note at tempo of 60
300::ms => dur rls;
? => s.gain;
adsr.set(  ?::ms, ?::ms, ?, rls );
adsr.keyOn();
noteDur - rls => now;
adsr.keyOff();
rls => now;

// Note 2
Std.mtof( 60 ) => s.freq;
500::ms => noteDur; // eighth note at tempo of 60
390::ms => rls;
? => s.gain;
adsr.set(  ?::ms, ?::ms, ?, rls );adsr.keyOn();
noteDur - rls => now;
adsr.keyOff();
rls => now;
```

```
// Note 3
Std.mtof( 60 ) => s.freq;
500::ms => noteDur; // eighth note at tempo of 60
250::ms => rls;
? => s.gain;
adsr.set(  ?::ms, ?::ms, ?, rls );adsr.keyOn();
noteDur - rls => now;
adsr.keyOff();
rls => now;

// Note 4
Std.mtof( 60 ) => s.freq;
2000::ms => noteDur; // half note at tempo of 60
500::ms => rls;
? => s.gain;
adsr.set(  ?::ms, ?::ms, ?, rls );
adsr.keyOn();
noteDur - rls => now;
adsr.keyOff();
rls => now;
```

The Natural Decay Envelope

Human hearing responds to sound levels (amplitude) in an exponential manner. Envelope segments using exponential curves sound natural to the ear.

The exponential natural decay function is found in many natural phenomenon and is given by this formula.

$$f(n) = e^{-\frac{Kn}{N}}$$

where e is the base of natural logarithms, K is a positive integer called the time constant, n is a single data pointr, and N is the total number of data points.

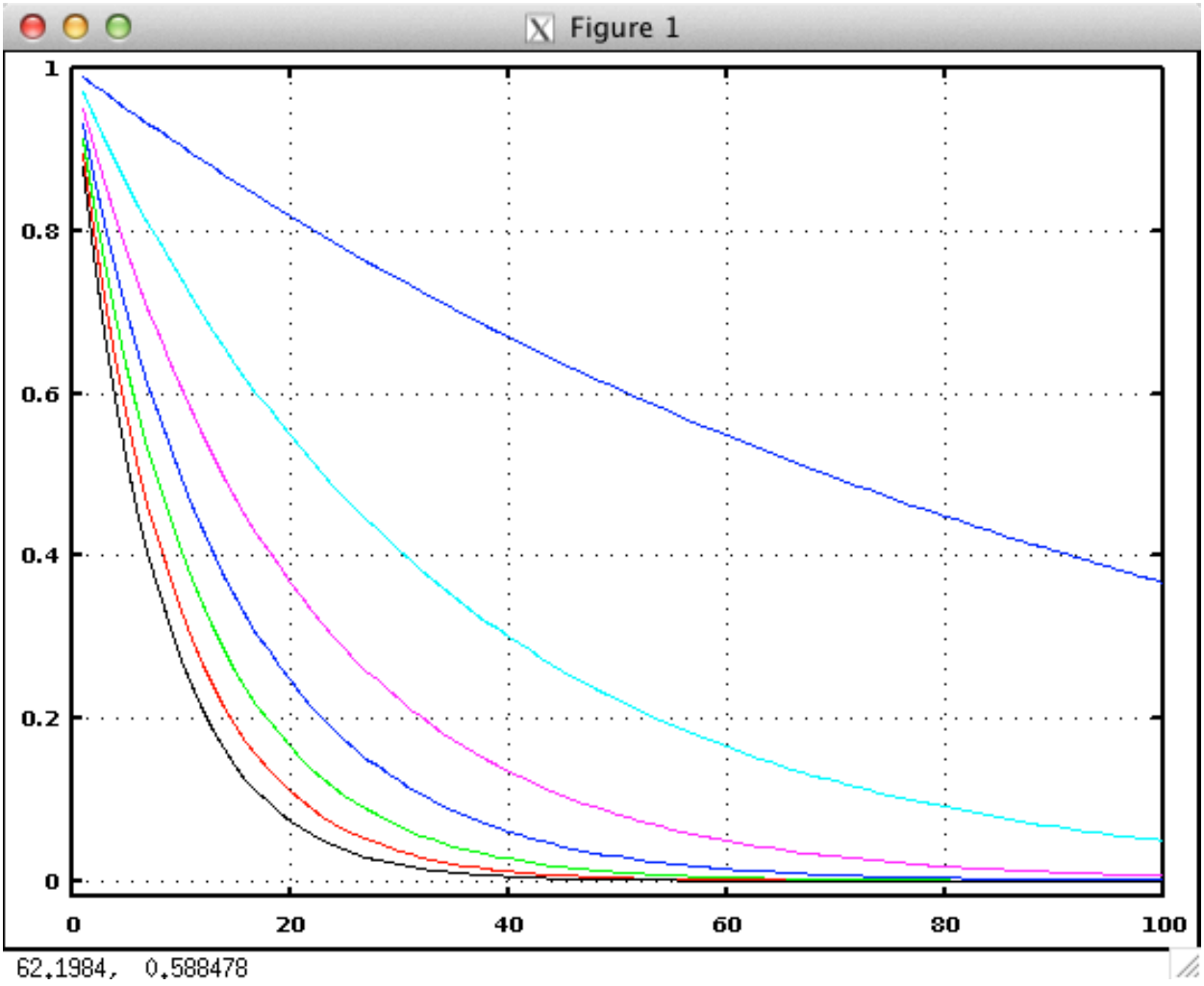
Plot The Natural Decay Function in Octave

Open octave. Create a new naturalDecayPlots.m file. Enter this code and run it in octave.

```
## naturalDecayPlots
## Author: John Ellinger <je@jemac.mibac.lan>
## Created: 2014-02-03

K = 1;
n = 1:100;
N = length( n );

clf; // clear figure
hold; // don't erase when showing next plot
grid; // show grid
axis( [0 100 -0.02 1.0] ); // set axis [xLo xHi yLo yHi]

x = exp( -1 * n / N );
plot( x );

x = exp( -3 * n / N );
plot( x, "c" );
x = exp( -5 * n / N );
plot( x, "m" );
```

```
x = exp( -7 * n / N );
plot( x, "b" );

x = exp( -9 * n / N );
plot( x, "g" );

x = exp( -11 * n / N );
plot( x, "r" );

x = exp( -13 * n / N );
plot( x, "k" );
```

You should see these plots. Notice that time constants 1, 3, and 5 never quite reach zero. In digital audio terms that means a note is not completely silenced.



Natural Decay Envelope in ChucK

NaturalDecayEnv1.ck uses an Impulse to create a sine wave and uses the Math.exp() function to create the natural decay function. The sine value and the decay value at each sample point are multiplied together to create an output sample.

[function]: float **exp** (float **x**);

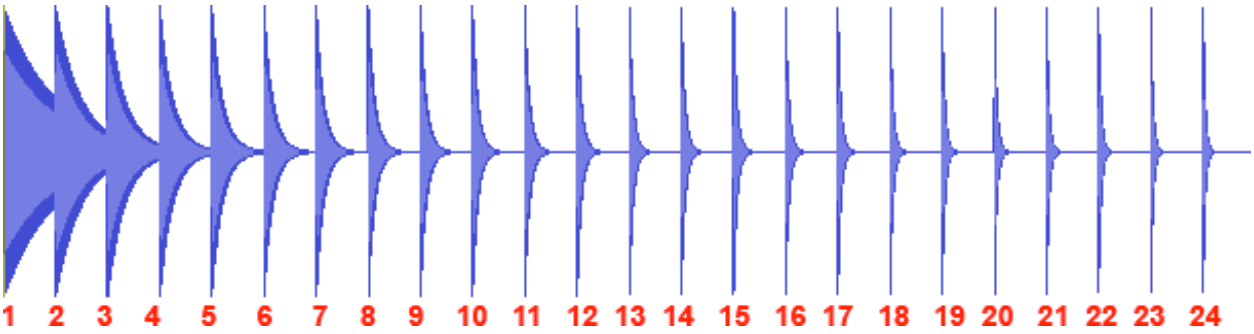
- *computes e^**x**, the base-e exponential of **x***

Enter and run this code

```
// naturalDecayEnv1.ck
// John Ellinger, Music 208, Winter 2014

Impulse imp => dac => WvOut w => blackhole;
w.wavFilename( me.sourceDir() + "naturalDecay1.wav" );

second / samp => float SR; // samples per second
samp / second => float T;   // seconds per sample
float sinValue;
float dkayValue;

1 => int timeConstant;
while ( timeConstant < 25 )
{
    for ( 0 => int n; n < SR; n++ )
    {
        Math.sin( 2 * Math.PI * 440 * n * T ) => sinValue;
        Math.exp( -1 * timeConstant * n * T ) => dkayValue;
        sinValue * dkayValue => imp.next;
        1::samp => now;
    }
    timeConstant++;
}
```

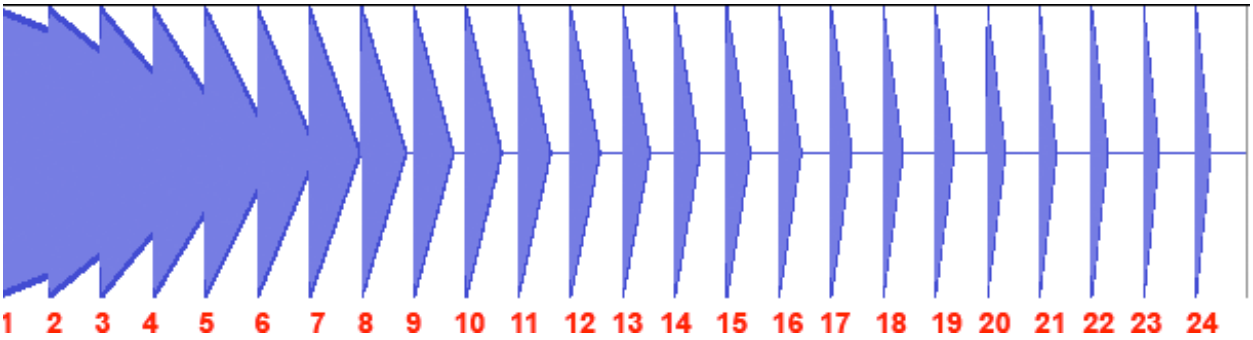
Audacity Amplitude View

Open the naturalDecay1.wav in Audacity. As the time constant increases the duration of the note gets shorter and shorter.

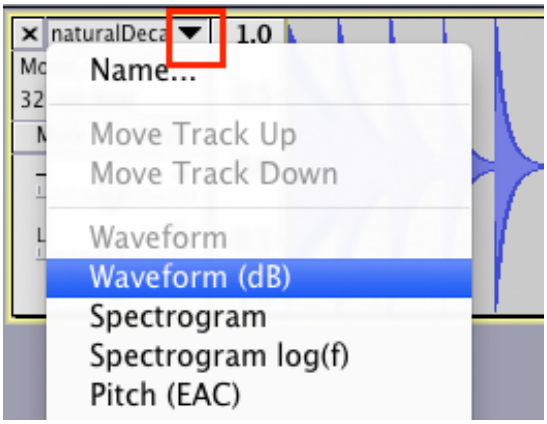


Audacity Decibel View

When the amplitude is viewed on the decibel scale it is apparent that you need a time constant of *7* or greater to ensure a silence between notes.



Audacity dB popup menu.



Natural Decay Envelope 2

NaturalDecayEnv2.ck uses a SinOsc and an Envelope to create the decay. Enter and run this code. Experiment with different time constants.

```
// naturalDecayEnv2.ck

// John Ellinger Music 208 Winter 2014
SinOsc s => Envelope e => dac;
440 => s.freq;

1.0 / 44100.0 => float T;
5 => int timeConstant;
1 => int n;

now + 1::second => time later;
while ( now < later )
{
    Math.exp( -1 * timeConstant * n * T ) => e.value;
    1::samp => now;
    n++;
}
```

Frequency Sweep Using An Envelope

This code is similar to the frequency sweep problem from Homework 5-6 but solves by directly incrementing the frequency at each sample. It does this by using the envelope to create a ramp with 88200 samples starting at 220 and ending at 880. Blackhole pulls the envelope values one sample at a time and assigns the envelope value to the SinOsc s.freq. The code is very short and compact.

```
// frequencySweep.ck
// www.cs.princeton.edu/courses/archive/spring12/cos314/handouts/Envelopes101.pdf

SinOsc s => dac;
Envelope e => blackhole;

// .value - ( float , READ/WRITE ) - set immediate value
220 => e.value;
// .target - ( float , READ/WRITE ) - ramp to arbitrary value.
880 => e.target; //  to 880Hz
// .duration - ( dur , READ/WRITE ) - duration to reach target
2::second => e.duration;

now + e.duration() => time later; //swoop for 1 second
while (now < later)
{
    // update frequency on every sample
    e.value() => s.freq;
    1::samp => now;
}
```

Do it again this time sweeping down.

Change the start and end frequencies.

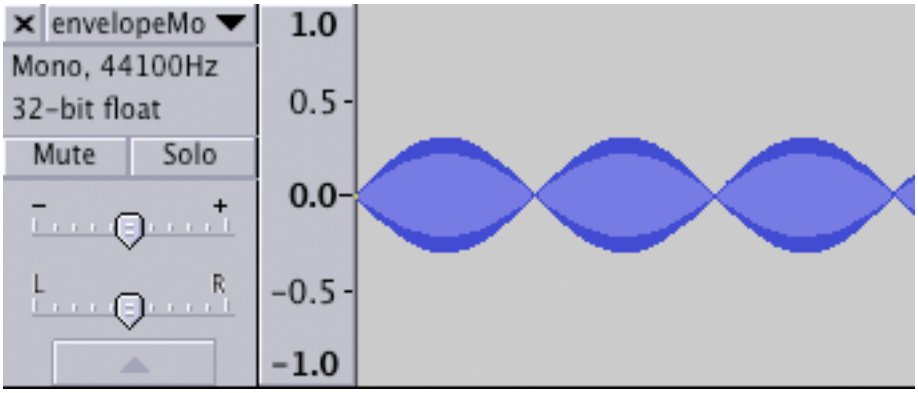
Envelope Tremelo is Amplitude Modulation

This code uses a Low Frequency Oscillator as an envelope creating amplitude modulation with three cycles per note.

```
// envelopeTremelo.ck
// John Ellinger Music 208 Winter 2014

function void playNoteWithTremelo( int midiNote, dur millis )
{
    SinOsc s => Envelope e => dac;
    Std.mtof( midiNote ) => s.freq;
    0.5 => s.gain;
    SinOsc lfo => blackhole;
    millis / samp => float noteDurationInSamples;
    // this will produce three tremelo cycles per note
    second / ( noteDurationInSamples / 1.5 )::samp => lfo.freq;

    for ( 0 => int n; n < noteDurationInSamples; n++ )
    {
        lfo.last() => e.value;
        1::samp => now;
    }
    e.keyOff();
    // unchuck (disconnect) from dac
    // comment it out to hear why
    e =< s =< dac;
// order important following causes clicks
//  s =< e =< dac;
}

dac => WvOut w => blackhole;
w.wavFilename( me.sourceDir() + "/envelopeTremelo.wav" );

650::ms => dur eighth;
```

```
playNoteWithTremelo( 64, eighth );
playNoteWithTremelo( 64, eighth );
playNoteWithTremelo( 65, eighth );
playNoteWithTremelo( 67, eighth );
playNoteWithTremelo( 67, eighth);
playNoteWithTremelo( 65, eighth );
playNoteWithTremelo( 64, eighth );
playNoteWithTremelo( 62, eighth );
playNoteWithTremelo( 60, eighth );
playNoteWithTremelo( 60, eighth );
playNoteWithTremelo( 62, eighth );
playNoteWithTremelo( 64, eighth );
playNoteWithTremelo( 64, eighth );
playNoteWithTremelo( 62, eighth );
playNoteWithTremelo( 62, eighth * 2 );
```

Single Note Viewed in Audacity



Envelope Pitch Bend

Use two envelopes, one for keyOn and keyOff and another for LFO created pitch bend.

```
// envelopeModulation3.ck
// John Ellinger, Music 208, Spring2013
SinOsc s => Envelope e => dac;
Envelope ePitch => blackhole;

500::ms =>  dur noteDur;
200::ms =>  dur noteRls;
noteDur * .85 => dur rlsDur;

function void playNoteWithPitchBend( int midiNote )
{
    SinOsc lfo => blackhole;
    // bend up
    Std.mtof( midiNote ) => float startFreq;
    Math.pow( -1, n ) => float sign;
    Std.mtof( midiNote + sign * 1.1 ) => float endFreq;
    startFreq => ePitch.value;
    endFreq => ePitch.target;

    now + (noteDur - rlsDur) => time later;
    1 => e.keyOn;
    while (now < later)
    {
        ePitch.value() => s.freq;
        1::samp => now;
    }

    //  bend down
    startFreq => ePitch.target;
    now + rlsDur => later;
    while (now < later)
    {
        ePitch.value() => s.freq;
        1::samp => now;
    }
```

```
    // release withoug click
    rlsDur => e.duration;
    e.keyOff();
    noteRls => now;
}

playNoteWithPitchBend( 64 );
playNoteWithPitchBend( 64 );
playNoteWithPitchBend( 65 );
playNoteWithPitchBend( 67 );
playNoteWithPitchBend( 67 );
playNoteWithPitchBend( 65 );
playNoteWithPitchBend( 64 );
playNoteWithPitchBend( 62 );
playNoteWithPitchBend( 60 );
playNoteWithPitchBend( 60 );
playNoteWithPitchBend( 62 );
playNoteWithPitchBend( 64 );
playNoteWithPitchBend( 64 );
playNoteWithPitchBend( 62 );
playNoteWithPitchBend( 62 );
```

Experiments

Change the ePitchRls time

```
noteDur * .85 => dur ePitchRls;
```

Change the endFreq by adjusting the amount added to midiNote. For example +12 would be an octave swing. 1.1 is slightly more than a half step.

```
Std.mtof( midiNote + 1.1 ) => float endFreq;
```

Change the direction of the endFreq swing.

```
Std.mtof( midiNote - 1.1 ) => float endFreq;
```

See if you can figure out a way to alternate the direction of the swing + - + - + - .