

Designing Sound in SuperCollider/Print version

From Wikibooks, the open-content textbooks collection

Note: the latest version of this book can be found at http://en.wikibooks.org/wiki/Designing_Sound_in_SuperCollider

This book is an independent project based on **Designing Sound** by Andy Farnell, all about the principles and techniques needed to design sound effects for real-time synthesis. The original book provides examples in the PureData (<http://puredata.info>) language - here we have re-created some of the examples using SuperCollider (<http://supercollider.sourceforge.net/>) .

The original book includes **much more** than what you see here - we're only recreating the examples and not the text! So in a sense this is *not a stand-alone book* and you'll get the most out of it if you have the original book with you. But we hope the examples are illustrative in themselves.

Any defects in the code we present should be assumed to be our own mistakes, and no reflection on Andy's fine book!

Table of contents

(Note: the numbering is done to match up with the chapter numbers in the original book.)

Tools & Technique

- 14. Schroeder reverb
- 17. Additive synthesis

Practicals: Artificial sounds

- 24. Pedestrians
- 25. Phone tones
- 26. DTMF
- 27. Alarms
- 28. Sirens

Practicals: Idiophonics

- 29. Telephone bell
- 30. Bouncing ball



- 31. Rolling can
- 32. Creaking door
- 33. Boing



Practicals: Nature

- 34. Fire
- 35. Bubbles
- 36. Running water
- 38. Rain

Tools & Technique

Fig 14.28: recirculating schroeder reverb

First start the server, if you haven't already:

```
s.boot;
```

Then so we have some source material, we'll load the standard sound file that comes bundled with SC:

```
b = Buffer.read(s, "sounds/allw1k01-44_1.aiff");
// Hear it raw:
b.play
```

Now here's the code which creates the reverb in a single Synth, with four separate delay lines cross-fertilising each other:

```
(
Ndef(\verb, {
  var input, output, delrd, sig, deltimes;

  // Choose which sort of input you want by (un)commenting these lines:
  input = Pan2.ar(PlayBuf.ar(1, b, loop: 0), -0.5); // buffer playback, panned
  //input = SoundIn.ar([0,1]); // TAKE CARE of feedback - use headphones
  //input = Dust2.ar([0.1, 0.01]); // Occasional clicks

  // Read our 4-channel delayed signals back from the feedback loop
  delrd = LocalIn.ar(4);

  // This will be our eventual output, which will also be recirculated
  output = input + delrd[[0,1]];

  // Cross-fertilise the four delay lines with each other:
  sig = [output[0]+output[1], output[0]-output[1], delrd[2]+delrd[3], delrd[2]-
  sig = [sig[0]+sig[2], sig[1]+sig[3], sig[0]-sig[2], sig[1]-sig[3]];
  // Attenuate the delayed signals so they decay:
  sig = sig * [0.4, 0.37, 0.333, 0.3];

  // Here we give delay times in milliseconds, convert to seconds,
  // then compensate with ControlDur for the one-block delay
  // which is always introduced when using the LocalIn/Out fdbk loop
  deltimes = [101, 143, 165, 177] * 0.001 - ControlDur.ir;
```


```

// Apply the delays and send the signals into the feedback loop
LocalOut.ar(DelayC.ar(sig, deltimes, deltimes));

// Now let's hear it:
Out.ar(0, output);

}).play
)

```



And here's an alternative way of doing exactly the same thing, this time using a matrix to represent the cross-mixing of the delayed streams. The single matrix replaces all those plusses and minusses so it's a neat way to represent the mixing - see which you find most readable.

```

(
Ndef(\verb, {
  var input, output, delrd, sig, deltimes;

  // Choose which sort of input you want by (un)commenting these lines:
  input = Pan2.ar(PlayBuf.ar(1, b, loop: 0), -0.5); // buffer playback, panned
  //input = SoundIn.ar([0,1]); // TAKE CARE of feedback - use headphones
  //input = Dust2.ar([0.1, 0.01]); // Occasional clicks

  // Read our 4-channel delayed signals back from the feedback loop
  delrd = LocalIn.ar(4);

  // This will be our eventual output, which will also be recirculated
  output = input + delrd[[0,1]];

  sig = output ++ delrd[[2,3]];
  // Cross-fertilise the four delay lines with each other:
  sig = ([ [1, 1, 1, 1],
           [1, -1, 1, -1],
           [1, 1, -1, -1],
           [1, -1, -1, 1]] * sig).sum;
  // Attenuate the delayed signals so they decay:
  sig = sig * [0.4, 0.37, 0.333, 0.3];

  // Here we give delay times in milliseconds, convert to seconds,
  // then compensate with ControlDur for the one-block delay
  // which is always introduced when using the LocalIn/Out fdbk loop
  deltimes = [101, 143, 165, 177] * 0.001 - ControlDur.ir;


  // Apply the delays and send the signals into the feedback loop
  LocalOut.ar(DelayC.ar(sig, deltimes, deltimes));

  // Now let's hear it:
  Out.ar(0, output);

}).play
)

// To stop it:
Ndef(\verb).free;

```



These SuperCollider code examples are by Dan Stowell, based on techniques and examples in *Designing Sound* by Andy Farnell.

Fig 17.4: Additive synthesis

Additive synthesis using eqn 17.2, taken from Moorer.

First start the server, if you haven't already - this time making sure it's the server which lets use the oscilloscopes:

```
Server.default = s = GUI.stethoscope.defaultServer;  
s.boot;
```

In the following we use ".sin" and ".cos" as mathematical operations, so we can stick closely to the original design - these are **much** less efficient than using SC's SinOsc oscillator, which is what most people would use for additive synthesis in SuperCollider.

In the original diagram, freq starts as 122, index as 0.42. Here we connect these parameters to the mouse - left/right for freq, up/down for harmonic decay.

The amplitude will vary with the decay coefficient, so practically, you'd want some kind of normalisation.

```
(  
x = {  
  
    var freq = MouseX.kr(100, 1000, 1) / SampleRate.ir;  
    var distance = 3.00;  
    var index = MouseY.kr(0.42, 0.99);  
    var theta, beta, num, denom, son;  
  
    // Two phasors which will ramp from zero to 2pi  
    theta = Phasor.ar(0, freq, 0, 2pi);  
    beta = Phasor.ar(0, freq * distance, 0, 2pi);  
  
    num = sin(theta) - (index * sin(theta - beta));  
    denom = 1 + index.squared - (2 * index * cos(beta));  
  
    son = num / denom;  
  
    Out.ar(0, Pan2.ar(son * 0.3));  
  
}.freqscope; // Use ".freqscope" or ".scope", both are illustrative.  
)
```

These SuperCollider code examples are by Dan Stowell, based on techniques and examples in *Designing Sound* by Andy Farnell.

Practicals: Artificial sounds

Fig 24.4: pedestrian crossing beeps (UK-style)

The LFPulse here is used to provide a sharp-edged on/off for the sound. (LFPulse outputs a low of zero

and a high of one, which is handy in this case so we don't need to re-scale it before we can use it as the multiplier.)

```
(
Ndef(\pedestrian, {
  SinOsc.ar(2500, 0, 0.2) * LFPulse.ar(5);
}).play
)
```

These SuperCollider code examples are by Dan Stowell, based on techniques and examples in *Designing Sound* by Andy Farnell.

Fig 25.2: CCITT dialing tone

```
(
Ndef(\dialtone, {
  // Note: the array here specifies two frequencies, so we get two separate ch
  // We sum the two channels so they combine into one signal - otherwise we
  // would hear one note on left, one note on right.
  Pan2.ar(SinOsc.ar([350, 440], 0, 0.2).sum)
}).play
)

// To stop:
Ndef(\dialtone).free;
```



Fig 25.3: Filter to approximate the transmission medium

```
// Doesn't make any sound in itself, but
// filters whatever we put in Ndef(\phonesource) and outputs that
(
Ndef(\transmed, {
  var sig = Ndef(\phonesource).ar.clip2(0.9);
  sig = BPF.ar(sig, 2000, 1/12);
  sig =
    BPF.ar(sig * 0.5, 400, 1/3)
    +
    (sig.clip2(0.4) * 0.15);
  HPF.ar(HPF.ar(sig, 90), 90) * 100;
}).play
)
```

Leave it running while you run the other blocks of code below...

Fig 25.4: The filter applied to the dialing tone

```
(
Ndef(\phonesource, {
  var onoff;
  onoff = if(MouseX.kr > 0.2, 1, 0);

```

```

        SinOsc.ar([350, 440], 0, onoff).sum * 0.2
    })
)

```

Fig 25.5a: Ringing tone

```

(
Ndef(\phonesource, {
    var onoff;
    onoff = LFPulse.ar(1/6, width: 1/3);
    SinOsc.ar([480, 440], 0, onoff).sum * 0.2
})
)

```

Fig 25.5b: Busy tone

```

(
Ndef(\phonesource, {
    var onoff;
    onoff = LPF.ar(LFPulse.ar(2), 100);
    SinOsc.ar([480, 620], 0, onoff).sum * 0.2
})
)

```

Fig 25.6: Pulse dialling, before DTMF

```

// First run this block to create the synth... (the filter from earlier on should st.
(
Ndef(\phonesource, { |t_trig=0, number=0|
    var onoff, trigs, son;
    number = if(number < 0.5, 10, number); // zero is represented by 10 clicks!
    onoff = Trig1.ar(t_trig, number * 0.1);
    trigs = Impulse.ar(10) * onoff;
    son = Trig1.ar(trigs, 0.04);
    son;
});
)
// ...then dial some numbers by repeatedly running this line:
Ndef(\phonesource).set(\t_trig, 1, \number, 10.rand.postln);

```

These SuperCollider code examples are by Dan Stowell, based on techniques and examples in *Designing Sound* by Andy Farnell.

26: DTMF dialling tones

```

(
// This data structure (like a "hashtable" or "associative array" in other languages,
// maps from a phone key to a pair of frequencies in Hz.
// We can push these frequencies to a synth.
~tbl = IdentityDictionary[
    $1 -> [[697, 1209]],
    $2 -> [[770, 1209]],

```

```

$3 -> [[852, 1209]],
$4 -> [[697, 1336]],
$5 -> [[770, 1336]],
$6 -> [[852, 1336]],
$7 -> [[697, 1477]],
$8 -> [[770, 1477]],
$9 -> [[852, 1477]],
$* -> [[697, 1633]],
$0 -> [[770, 1633]],
$# -> [[852, 1633]],
$A -> [[941, 1209]],
$B -> [[941, 1336]],
$C -> [[941, 1477]],
$D -> [[941, 1633]]
];

// Here we define a SynthDef which plays a single "number" at a time.
// Note that our strategy here is a bit different from the PD code in the book:
// there, a single pair of sine-wave oscillators was re-used for each number,
// whereas here, we create (and later free) an individual synth for each number.
SynthDef(\dtmf, {|freq=#[770, 1633], out=0, amp=0.2, gate=1|
  var son, env;
  son = SinOsc.ar(freq, 0, amp).sum;
  env = EnvGen.ar(Env.asr(0.001, 1, 0.001), gate, doneAction: 2);
  Out.ar(out, Pan2.ar(son * env * amp));
}).memStore;
)

// Check that it works:
x = Synth(\dtmf) // create
x.set(\gate, 0) // free

(
// This pattern generates a random "phone number" and dials it
Pbind(
  \instrument, \dtmf,
  \dur, 0.2, // or for more "human" timing, try Pwhite(0.2, 0.5, inf)
  \sustain, 0.15,
  \amp, 0.3,
  \freq, Prand(~tbl.asArray, 13)
).play;
)

(
// You could even dial a specific number:
Pbind(
  \instrument, \dtmf,
  \dur, 0.2, // or for more "human" timing, try Pwhite(0.2, 0.5, inf)
  \sustain, 0.15,
  \amp, 0.3,
  \freq, Pseq("012827743866".collectAs({|digit| ~tbl[digit] }, Array))
).play;
)

```

These SuperCollider code examples are by Dan Stowell, based on techniques and examples in *Designing Sound* by Andy Farnell.

Fig 27.2: Alarm with two alternating tones

Note that we have separated this into multiple lines for clarity. *Exercise:* rewrite it as a single line - and use array expansion rather than writing "SinOsc" twice.

```
(
Ndef(\alarm, {
  var tone1 = SinOsc.ar(600);
  var tone2 = SinOsc.ar(800);
  // We switch between the tones using LFPulse, but soften the crossfade with :
  var control = LPF.kr(LFPulse.kr(2), 70);
  var out = SelectX.ar(control, [tone1, tone2]);
  Pan2.ar(out * 0.1)
}).play
)
```




Fig 27.3: Alarm with three alternating tones

```
(
Ndef(\alarm, {
  var tone1 = SinOsc.ar(723);
  var tone2 = SinOsc.ar(932);
  var tone3 = SinOsc.ar(1012);
  // Stepper is perfect for stepping through the options:
  var control = LPF.kr(Stepper.kr(Impulse.kr(2), 0, 0, 2), 70);
  var out = SelectX.ar(control, [tone1, tone2, tone3]);
  Pan2.ar(out * 0.1)
}).play
)
```

Or we can write exactly the same thing a bit more generically using Demand units. The frequencies are simply given as an Array - change the values, or add new ones to the end:

```
(
Ndef(\alarm, {
  var freq, out;
  freq = Duty.kr(0.5, 0, Dseq([723, 932, 1012], inf));
  freq = LPF.kr(freq, 70);
  out = SinOsc.ar(freq);
  Pan2.ar(out * 0.1)
}).play
)
```

Fig 27.4: A choice of timbral settings

This using .cos and .sin to perform waveshaping - move the mouse left and right to choose from our 4 timbral options.

```
(
Ndef(\alarm, {
  var freq, out, operations;
  freq = Duty.kr(0.05, 0, Dseq([723, 932, 1012], inf));
  freq = LPF.kr(freq, 70);
  out = SinOsc.ar(freq);
  operations = [out, (out * pi).sin, (out * pi).cos, ((out+0.25) * pi).cos];
  out = Select.ar(MouseX.kr(0,4).poll, operations);
}
```



```

    Pan2.ar(out * 0.1)
  }).play
)

```

Exercise: alter the "Duty" line so that you can have different durations for each of the notes.(Hint: you can use Dseq to specify times just as easily as to specify frequencies.)

Fig 27.7: A multi-ringer

This SynthDef is capable of a wide variety of sequences, as you'll see below:

```

(
SynthDef(\dsaf_multialarm, {
  |length=0.05, freqs=#[600,800,600,800], timbre=1, repeats=inf|
  var freq, out, operations;
  freq = Duty.ar(length, 0, Dseq(freqs, repeats), doneAction: 2);
  freq = LPF.ar(freq, 70);
  out = LeakDC.ar(SinOsc.ar(freq));
  out = Select.ar(timbre, [out, (out * pi).sin, (out * pi).cos, ((out+0.25) * ]
  // NOTE: when writing a synthdef always remember the Out ugen!
  // (Handy shortcuts like Ndef and {}).play often add Out on your behalf)
  Out.ar(0, Pan2.ar(out * 0.1))
}).memStore;
)

```

...and now we can play it, using the sequences used in the book:

```

// happy blips
Synth(\dsaf_multialarm, [\length, 0.1, \freqs, [349, 0, 349, 0], \timbre, 1, \repeats, 2];
// affirmative
Synth(\dsaf_multialarm, [\length, 0.1, \freqs, [238, 0, 317, 0], \timbre, 2, \repeats, 2];
// activate
Synth(\dsaf_multialarm, [\length, 0.02, \freqs, [300, 125, 0, 0], \timbre, 2, \repeats, 2];
// invaders?
Synth(\dsaf_multialarm, [\length, 0.03, \freqs, [360, 238, 174, 158], \timbre, 1]);
// information
Synth(\dsaf_multialarm, [\length, 0.05, \freqs, [2000, 2010, 2000, 2010], \timbre, 1];
// message alert
Synth(\dsaf_multialarm, [\length, 0.15, \freqs, [619, 571, 365, 206], \timbre, 1, \repeats, 2];
// finished
Synth(\dsaf_multialarm, [\length, 0.15, \freqs, [365, 571, 619, 206], \timbre, 3, \repeats, 2];
// error code
Synth(\dsaf_multialarm, [\length, 0.01, \freqs, [1000, 0, 1000, 0], \timbre, 3, \repeats, 2];
// wronnnnnnnnnng
(
Pbind(
  \instrument, \dsaf_multialarm,
  \freqs, [[1000, 476, 159, 0]],
  \timbre, 2,
  \repeats, 25,
  \length, Pseq([0.003, 0.005]),
  \dur, 0.5
).play
)

```

Excercise: work out why some of these multialarm examples don't quite sound like Andy's audio examples! ;)

These SuperCollider code examples are by Dan Stowell, based on techniques and examples in *Designing Sound* by Andy Farnell.

28: US-style police siren

We'll use the internal server to make sure we can use the oscilloscope etc:

```
Server.default = s = Server.internal;  
s.boot;
```

Fig 28.6: imitation of capacitor charge/discharge

In SuperCollider we can provide an almost-direct physical model: the LFPulse represents a "raw" on/off signal before smoothing by capacitors, and the "lagud" provides exponential smoothing, with the handy feature of allowing different time-periods for the "on" and "off" convergence. This one-line example will plot the curve for you:

```
{LFPulse.ar(1, 0.99, 0.4).lagud(0.3, 0.7)}.plot(2)
```

The siren

Now let's use this technique both for the pitch curve, and for waveform synthesis.

```
(  
SynthDef(\dsaf_horn1, { |rate=0.1|  
  var freq = LFPulse.kr(rate, 0.99, 0.4).lagud(0.4 / rate, 0.6 / rate) * 800 +  
  var son = LFPulse.ar(freq, 0.99, 0.2).lagud(0.4 / freq, 0.6 / freq) * 2 - 1  
  
  // This filtering is a simple approximation of the plastic horn acoustics:  
  son = BPF.ar(son.clip2(0.2), 1500, 1/4) * 4;  
  
  // delay and reverb, to simulate the environment in which we hear the siren  
  son = son + DelayC.ar(son, 0.1, 0.1, 0.3);  
  son = son + FreeVerb.ar(son);  
  
  Out.ar(0, Pan2.ar(son * 0.4));  
}).memStore;  
)  
  
x = Synth(\dsaf_horn1);  
  
s.scope  
  
// Choose a rate  
x.set(\rate, 3);  
x.set(\rate, 0.1);
```

Excercise: instead of using the lagged-pulse implementation of the waveform, do as the book says and try using a simple triangle oscillator (LFTri) - this loses the physical-modelling "realism" of the

authentic circuit but will be more efficient, and reasonably similar. How much of an effect on the tone quality do you get?

These SuperCollider code examples are by Dan Stowell, based on techniques and examples in *Designing Sound* by Andy Farnell.

Practicals: Idiophonics

29 Old-style telephone bell

In the book, one partial of the resonance is modelled using a sinewave, modulated by an envelope that dies away on a quadratic-shaped curve. Groups of three such oscillators are then made.

Here we take advantage of a built-in unit in SuperCollider which does quite a lot of the work for us: Klank. A difference: it gives exponential rather than quadratic decay - which is not too bad since exponential decay is often what you want.

Fig 29.11: a simple metallic resonance with three pitches

```
(
{
  var son;
  son = Klank.ar(`[
    [521, 732, 934], // freqs
    [0.7, 0.45, 0.25], // amps
    [0.8, 0.8, 0.8] // ring times
  ]
  , Impulse.ar(1));
  Out.ar(0, Pan2.ar(son * 0.2));
}.play
)
```

Fig 29.12: a striker sound

As described in the book, a brief click is all we need to approximate this part of the sound.

```
(
x = {
  var son = WhiteNoise.ar(XLine.ar(1, 0.000001, 0.01, doneAction: 2)) * 0.1;
  Out.ar(0, Pan2.ar(son));
}.play
)
```


Fig 29.13: many resonators to create the various modes of the bell

```
(
```

```

SynthDef(\dsaf_phonebell1, { |freq=465, strength=1, decay=3|
  var son;
  son = Klank.ar(`[
    // frequency ratios
    [0.501, 1, 0.7, 2.002, 3, 9.6, 2.49, 11, 2.571, 3.05, 6.242, 12
    // amps
    [0.002,0.02,0.001, 0.008,0.02,0.004, 0.02,0.04,0.02, 0.005,0.05,0.05
    // ring times - "stutter" duplicates each entry threefold
    [1.2, 0.9, 0.25, 0.14, 0.07].stutter(3)
  ]
  , Impulse.ar(1), freq, 0, decay);
  Out.ar(0, Pan2.ar(son));
}).memStore
)
x = Synth(\dsaf_phonebell1, [\freq, 500]);

```



Note: **leave the bell playing** while you add the following effect:

Fig 29.14: resonant effects of bakelite phone casing

```

(
SynthDef(\dsaf_phonecase1, { |in=0, out=0, mix=0|
  var casein = In.ar(in, 2);

  var delayA = CombC.ar(casein, 0.00077, 0.00077, 0.1);
  var delayB = CombC.ar(delayA, 0.00088, 0.00088, 0.1);
  var bands = BPF.ar(delayB, [1243, 287, 431], 1/12).sum;
  var son = bands.clip2(0.3);

  ReplaceOut.ar(out, XFade2.ar(casein, son, mix))

}).memStore;
)
y = Synth(\dsaf_phonecase1, target: x, addAction: \addAfter);

```

If you have sc3.3 or later, you can visualise a frequency representation of the filter's effect:

```

GUI.stethoscope.defaultServer.boot;
(
// Notice that for this shortcut visualisation, we define the filter as a function to
// input as an argument, and returning its output (we don't use In.ar or Out.ar)
{ |casein|
  var delayA = CombC.ar(casein, 0.00077, 0.00077, 0.1);
  var delayB = CombC.ar(delayA, 0.00088, 0.00088, 0.1);
  var bands = BPF.ar(delayB, [1243, 287, 431], 1/12).sum;
  var son = bands.clip2(0.3);

  // Mouse to the LEFT means flat filter (no change), to the RIGHT means full i
  XFade2.ar(casein, son, MouseX.kr(-1,1));

}.scopeResponse
)

```




Fig 29.14: putting it all together

We will re-use the SynthDef from fig 29.14, so make sure you have run that so that the server has that SynthDef. (But stop the sound, if it is still running.)

```

// A tweaked version of the phonebell synthdef, to take an on/off from outside, and .
(
SynthDef(\dsaf_phonebell2, { |gate=1, freq=465, strength=1, decay=3, amp=1|
  var trigs, striker, son;
  trigs = Impulse.ar(14) * gate;
  striker = WhiteNoise.ar(EnvGen.ar(Env.perc(0.0000001, 0.01), trigs));
  son = Klank.ar([
    // frequency ratios
    [0.501, 1, 0.7, 2.002, 3, 9.6, 2.49, 11, 2.571, 3.05, 6.242, 12
    // amps
    [0.002, 0.02, 0.001, 0.008, 0.02, 0.004, 0.02, 0.04, 0.02, 0.005, 0.05, 0.05
    // ring times - "stutter" duplicates each entry threefold
    [1.2, 0.9, 0.25, 0.14, 0.07].stutter(3)
  ],
  , striker, freq, 0, decay);
  Out.ar(0, Pan2.ar(son * amp));
}).memStore
)

// We could launch the patch all at once, but let's do it bit-by-bit so we understand
// Here we start the phone bells constantly ringing. We put them in a group for convenience
~bellgroup = Group.new(s);
~bell1 = Synth(\dsaf_phonebell2, [\freq, 650], ~bellgroup);
~bell2 = Synth(\dsaf_phonebell2, [\freq, 653], ~bellgroup);

// Now we add the bakelite
y = Synth(\dsaf_phonecase1, [\mix, -0.65], target: ~bellgroup, addAction: \addAfter)

// OK, shush for now
~bellgroup.set(\gate, 0);

// Now let's turn them on and off in a telephone-like pattern.
// This could be done using a synth, but let's use a (client-side) pattern:
p = Pbind(\type, \set, \id, ~bellgroup.nodeID, \args, [\gate], \gate, Pseq([1,0], in:
p.stop

```



These SuperCollider code examples are by Dan Stowell, based on techniques and examples in *Designing Sound* by Andy Farnell.

Fig 30.2: Bouncing ball

```

(
Ndef(\bouncer, {

var bounceperiod, bouncetrigs, amp, fm, mainosc;

bounceperiod = Line.kr(0.3, 0, 3, doneAction: 2);

bouncetrigs = Impulse.kr(bounceperiod.reciprocal.min(30));

amp = EnvGen.ar(Env.perc(0.001, 0.0), bouncetrigs);
amp = Amplitude.ar(amp, 0, bounceperiod) * Line.kr(1, 0.05, 3);

fm =

```

```

    SinOsc.ar(120).range(0, Line.ar(1, 0, 3))
      +
    (amp * Line.ar(1, 0, 3).cubed * 130 + 80)
;

mainosc = SinOsc.ar(fm, pi/2);

amp * mainosc;
}).play
)

```

These SuperCollider code examples are by Dan Stowell, based on techniques and examples in *Designing Sound* by Andy Farnell.

Fig 31.3: simple approximation to a drink can

The four clear resonances are identified using a spectrogram of a can.

```

(
x = { |t_trig=0|
    // This line just creates a sharp little spike whenever we want:
    var strike = EnvGen.ar(Env.perc(0.0001, 0.001, 0.1), t_trig);
    // here's the resonances:
    var son = Ringz.ar(strike, [359, 426, 1748, 3150], 0.2).sum;
    // some distortion livens up the spectrum a little:
    son = HPF.ar(son.clip2(0.6), 300);
    son * 0.2
}.play;
)
x.set(\t_trig, 1); // Run this line to hit the can!

```

Fig 31.6: a repeating roll pattern

This simulates the regular turning of the drinks can. Since it's a control signal we'll plot it rather than play, for now.

```

(
~regularroll = { |rate = 1|
    // In the original code, Andy uses a master phase control,
    // wrapping and re-scaling it before differentiating, to produce
    // a repeating but disorderly set of impulses.
    // Here we do it differently - we use Impulse.kr to generate the
    // impulses directly.
    // We evaluate this function multiple times using .dup so that
    // we get a whole set of impulses with random phase positions.
    {
        Impulse.kr(rate, 1.0.rand, 1.0.bilinrand)
    }.dup(10).sum
};
~regularroll.plot(2);
)

```

Fig 31.7: ground bumping

```
(
// This signal contribution to rolling signature based on Mathias Rath's idea - see
// (ajf2009)
//
~irregularground = { |rate=10|
  var trigs = Dust.kr(rate);
  EnvGen.ar(
    Env([0,0,-1,1.5,-1,1,0], [0.1, 0.1, 0.001, 0.001, 0.1, 0.1], 'sine')
    trigs
  ) * 0.1
};
~irregularground.plot(4)
)
```



Fig 31.8: putting it all together

(You must have run figs 31.6 and 31.7 to get their functions defined.)

The sound of a can rolling a little and coming to a stop.

```
(
x = {
  var rate, strike, son;
  // rate of motion starts fast and tails off
  rate = XLine.kr(4, 0.001, 8, doneAction: 2);
  // This rate affects both the regular rolling, and the irregular ground cont.
  strike =
    ~irregularground.(rate*2) * 0.04
    +
    K2A.ar(~regularroll.(rate) * 0.1)
    ;
  // Force the strikes to die off in intensity:
  strike = strike * XLine.ar(1, 0.0001, 8);
  // And here are the tin-can resonances as in fig 31.3:
  son = Ringz.ar(strike, [359, 426, 1748, 3150], 0.2).sum;
  son = HPF.ar(son.clip2(0.6), 300);
  son * 0.2
}.play;
)
```



These SuperCollider code examples are by Dan Stowell, based on techniques and examples in *Designing Sound* by Andy Farnell.

Fig 32.3: formants for a wooden door

```
(
~woodfilter = { |input|
  var freqs, rqs, output;
  // Note: these freqs are as given in the diagram:

```

```

freqs = [62.5, 125, 250, 395, 560, 790];
// The Q values given in the diagram (we take reciprocal, since that's what i
rqs = 1 / [1, 1, 2, 2, 3, 3];
// in the text, andrew says that the freqs follow these ratios,
// which give a very different set of freqs...:
// freqs = 125 * [0.5, 1, 1.58, 2.24, 2.92, 2, 2.55, 3.16];

//Now let's apply the parallel bandpass filters, plus mix in a bit of the or.
output = BPF.ar(input, freqs, rqs).sum + (input*0.2);

};

// Now let's use this function in something - some dust impulses tap-tap-tapping on :
x = {
  Pan2.ar(~woodfilter.value( LPF.ar(Dust.ar(10), 10000) ))
}.play;
// Doesn't sound much like a door? Compare it to the raw tapping sound (edit the abov
)

```




Fig 32.4: stick-slip motion in response to applied force

```

(
~stickslip = { |force|
  var inMotion, slipEvents, forceBuildup, evtAmp, evtDecayTime, evts;
  force = force.lag(0.1); // smoothing to get rid of volatile control changes

  inMotion = force > 0.1; // static friction: nothing at all below a certain fo

  // slip events are generated at random with frequency proportional to force.
  // I originally used Dust to generate random events at a defined frequency, i
  // that lacks the slight "pitched" sound of the creaky door. Here we use Impu
  // to generate a frequency, but we add some noise to its frequency to try and
  // avoid it getting too perfectly regular.
  slipEvents = inMotion * Impulse.ar(force.linlin(0.1, 1, 1, 1/0.003) * LFDNois

  forceBuildup = Phasor.ar(slipEvents, 10 * SampleDur.ir, 0, inf).min(1);

  // Whenever a slip event happens we use Latch to capture the amount of
  // force that had built up.
  evtAmp = Latch.ar(Delay1.ar(forceBuildup.sqrt), slipEvents);
  evtDecayTime = evtAmp.sqrt;
  // The book applies square-root functions to shape the dynamic range of the e
  // Remember that square-root is computationally intensive, so for efficient
  // generation we might want to change it to (e.g.) a pre-calculated envelope

  // Now we generate the events
  evts = EnvGen.ar(Env.perc(0.001, 1), slipEvents, evtAmp, 0, evtDecayTime * 0
};
// Let's plot 4 seconds worth, with steadily increasing force.
// Events should appear more frequent but less violent as the plot progresses.
{~stickslip.value(Line.kr(0,1,4))}.plot(4);
)

```




Fig 32.5: parallel delays to simulate rectangular door frame


```
(
~squarepanel = { |input|
  var times, filt;
  // times in milliseconds, converted to seconds:
  times = [4.52, 5.06, 6.27, 8, 5.48, 7.14, 10.12, 16] * 0.001;
  filt = DelayC.ar(input, times, times).mean;
  filt = HPF.ar(filt, 125);
  filt * 4
};
)
```

Fig 32.6: putting it all together

The following re-uses the functions defined further up this page in combination, to create a door which *you can push yourself!*

Put the mouse over to the left of the screen before you run this code. Then move the mouse to the right and to the left, to control the amount of force on the door.

```
x = {~squarepanel.value(~woodfilter.value(~stickslip.value(MouseX.kr(0,1))))}.play
```

These SuperCollider code examples are by Dan Stowell, based on techniques and examples in *Designing Sound* by Andy Farnell.

Fig 33.2a: clamped-mode vibration

```
(
~clampedmodes = { |basefreq, env|

  var freqs, amps;

  freqs = [1, 6.267, 17.55, 34.39];
  amps = [0.5, 0.25, 0.125, 0.06125];

  Klank.ar(`[freqs, amps, 0.2], env, basefreq);
};

{~clampedmodes.(100, Impulse.ar(10))}.plot(1)
)
```

Fig 33.2b: free-mode vibration

```
(
~freemodes = { |input, basefreq=100, res=80|
  var filtfreqs;

  // The actual filter freqs take these harmonic relationships:
  filtfreqs = basefreq * [1, 2.7565, 5.40392, 8.93295, 13.3443, 18.6379];

  BPF.ar(input, filtfreqs, 1/res).sum * 10
};

{~freemodes.(LFSaw.ar(4))}.plot(1)
```

)

Motion of the twanged ruler

Here we use Env to create an asymmetric waveshape - when the value is below zero, the ruler is touching the table and so is practically "shorter". Therefore it has a higher frequency (shorter wavelength) in the lower cycle than in the upper cycle.

```
~rulerwave = Env([1, 0, -0.7, 0, 1], [0.3, 0.1, 0.1, 0.3], [4, -4, 4, -4]).asSignal(!
~rulerwave.plot;
// Here let's plot it running at a frequency that speeds up.
// This approximates the actual trajectory of motion of the end of the ruler:
{Osc.kr(~rulerwave.as(LocalBuf), XLine.kr(50, 100, 1), mul: XLine.kr(1, 0.001, 1))}.]

// Now, every time the wave passes zero in a downwards-going direction, that represe
// This code builds on the previous one to derive the thwacks - one at each downward
(
{
  var motion, thwacks, isDown;
  motion = Osc.ar(~rulerwave.as(LocalBuf), XLine.kr(50, 100, 1), mul: XLine.kr
  isDown = motion < 0;
  thwacks = Trig1.ar(isDown, 0) * (0-Slope.ar(motion)) * 0.01;
  thwacks = LPF.ar(thwacks, 500);
  [motion, isDown, thwacks]
}.plot(1)
)
```



Let's hear it

OK, so now we need to make sound from this data. The base frequency for the resonator is higher if isDown==true, since the effective length is shorter. So we need to modulate the base frequency at the same time as pushing the thwacks through the resonators.

```
(
{
  var motion, thwacks, isDown, basefreq;
  motion = Osc.ar(~rulerwave.as(LocalBuf), XLine.kr(10, 100, 1), mul: Line.kr(
  isDown = motion < 0;
  thwacks = Trig1.ar(isDown, 0) * (0-Slope.ar(motion)) * 0.01;
  thwacks = LPF.ar(thwacks, 500);

  basefreq = if(isDown, 289, 111);
  ~freemodes.value(thwacks, basefreq, 100)
  +
  ~clampedmodes.value(basefreq, thwacks);
}.play
)
```

// That was a model of a ruler-on-a-desk. The next one is... something else.

```
(
{
  var motion, thwacks, isDown, basefreq;
  motion = Osc.ar(~rulerwave.as(LocalBuf), 80, mul: Line.kr(1, 0.001, 1, doneAc
```

```

isDown = motion < 0;
thwacks = Trig1.ar(isDown, 0) * (0-Slope.ar(motion)) * 0.01;

basefreq = if(isDown, 289, 111) * Pulse.ar(10).exprange(0.9, 1.1);
~freemodes.value(thwacks, basefreq, 100)
      +
~clampedmodes.value(basefreq, thwacks);
}.play
)

```



These SuperCollider code examples are by Dan Stowell, based on techniques and examples in *Designing Sound* by Andy Farnell.

Practicals: Nature

Fig 34.3: simplest random hissing sound

```
{WhiteNoise.ar(LFNoise2.kr(1))}.play
```

Fig 34.4: square it

Squaring the noise gives it a more useful dynamic range, with rare loud bursts

```
{WhiteNoise.ar(LFNoise2.kr(1).squared)}.play
```

Fig 34.5: more

Add another .squared for even stronger effect on the dynamics, and use a high-pass filter to make the hiss more hissy

```
{HPF.ar(WhiteNoise.ar, 1000) * LFNoise2.kr(1).squared.squared}.play
```

Fig 34.6: a simple single crackle

```
{WhiteNoise.ar * Line.ar(1, 0, 0.02, doneAction: 2)}.play
```

Fig 34.7: many crackles

If we use a proper envelope generator instead of a simple line, we can retrigger it randomly with Dust to give random crackles of a controllable density

```
{WhiteNoise.ar * EnvGen.ar(Env.perc(0, 0.02, curve: 0), Dust.kr(1))}.play
```

Fig 34.8: more variation

Add more variation by modulating the duration and the filter frequency for each event

```
(
{
  var trigs, durscale, son, resfreq;
  trigs = Dust.kr(1);
  durscale = TRand.kr(1, 1.5, trigs); // vary duration between default 20ms and
  resfreq = TExpRand.kr(100, 1000, trigs); // different resonant frequency for
  son = WhiteNoise.ar * EnvGen.ar(Env.perc(0, 0.02, curve: 0), trigs, timeScale: durscale);
  son = son + BPF.ar(son, resfreq, 20);
}.play
)
```




Fig 34.9: woof

Simple attempt at the low "woofing" noise made by the flames themselves

```
{LPF.ar(WhiteNoise.ar, 30) * 100}.play
```

Fig 34.10: woosh

// another component we could use to build up a flamey sound

```
{BPF.ar(WhiteNoise.ar, 30, 0.2) * 20}.play
```

Fig 34.11: shape

Shaping the dynamic range and discarding some of the lower frequencies, allowing a little clipping to give a less static sound.

```
{LeakDC.ar(LeakDC.ar(BPF.ar(WhiteNoise.ar, 30, 0.2) * 50).clip2(0.9)) * 0.5}.play
```

Fig 34.12: putting it all together

```
(
~firegen = {
  var trigs, durscale, resfreq;
  var noise, hissing, crackles, lapping;
  // A common noise source
  noise = WhiteNoise.ar;
  // Hissing
  hissing = HPF.ar(noise, 1000) * LFNoise2.kr(1).squared.squared;
  // Crackle
  trigs = Dust.kr(1);
  durscale = TRand.kr(1, 1.5, trigs); // vary duration between default 20ms and
  resfreq = TExpRand.kr(100, 1000, trigs); // different resonant frequency for
  crackles = noise * EnvGen.ar(Env.perc(0, 0.02, curve: 0), trigs, timeScale: durscale);
  crackles = crackles + BPF.ar(crackles, resfreq, 20);
  // Flame
  lapping = LeakDC.ar(LeakDC.ar(BPF.ar(noise, 30, 0.2) * 50).clip2(0.9)) * 0.5;
  // Combine them:
  ([crackles, hissing, lapping] * [0.1, 0.3, 0.6]).sum * 3
};
~firegen.play
```



Fig 34.13: poly

Let's have four of the above, each filtered differently, for a composite effect

```
(
{
  BPF.ar(~firegen, 600, 1/0.2) +
  BPF.ar(~firegen, 1200, 1/0.6) +
  BPF.ar(~firegen, 2600, 1/0.4) +
  HPF.ar(~firegen, 1000)
}.play
)
```

These SuperCollider code examples are by Dan Stowell, based on techniques and examples in *Designing Sound* by Andy Farnell.

Fig 35.5: producing a repeating but random-seeming pattern of triggers

First we'll create a reusable synthdef that outputs triggers (but not sound):

```
(
SynthDef(\bubbletrigs, {|out=0, probability=0.5|
  var trigs, buf, a;
  // These two lines create a loop of zeroes
  // with some ones (i.e. triggers) placed at prime-number locations
  a = {0}.dup(200);
  [29, 37, 47, 67, 89, 113, 157, 197].do{|val| a[val] = 1};
  buf = a.as(LocalBuf);
  // playbuf by default will use the server's rate, but we want one item every
  trigs = PlayBuf.kr(1, buf, 0.015.reciprocal / (s.sampleRate / s.options.blockSize));
  // Randomly discard half of them, to remove too much obvious looping
  trigs = CoinGate.kr(probability, trigs);
  // Let's poll to watch the events appearing
  trigs.poll(trigs);
  Out.kr(out, trigs);
}).store
)
```

```
// Then we'll play it:
x = Synth(\bubbletrigs); // watch the post window to see the bubble events happening
x.free;
```



Fig 35.8: sound of a bubble

```
(
SynthDef(\bubblebub, { |out=0, t_trig=0, attack=0.01, decay=0.08, pitchcurvelen=0.1
  var pitch, son;
  amp = amp * EnvGen.ar(Env.perc(attack, decay).delay(0.003), t_trig, doneAct
```

```

pitch = freq * EnvGen.ar(Env.new([0,0,1],[0,1]).exprange(1, 2.718), t_trig, 1);
son = SinOsc.ar(pitch);
// high-pass to remove any lowpitched artifacts, scale amplitude
son = HPF.ar(son, 500) * amp * 10;
Out.ar(out, son);
}).store
)

x = Synth(\bubblebub);
x.set(\t_trig, 1); // run this line multiple times, to get multiple (very similar) bubbles
x.free;

```




Fig 35.9: four bubble systems, simply triggered at random.

```

(
s.bind{
// Here we'll create busses to hold the triggers, passing them from synth to
~maintrigbus = Bus.control(s, 1);
~bubtrigbus = Bus.control(s, 4);
// Note how we make sure things are running in the desired order, using \add
~trigs = Synth(\bubbletrigs, [\out: ~maintrigbus]);
// This reads the main trig and puts each trig on a randomly-chosen bus
~randomdistrib = {
var trig, which;
trig = In.kr(~maintrigbus);
which = TIRand.kr(0,3, trig);
// or try the Stepper instead of TIRand for "round-robin" selection:
// which = Stepper.kr(trig, 0, 0, 3);
which = which.poll(trig);
Out.kr(~bubtrigbus.index + which, trig);
}.play(target: ~trigs, addAction: \addAfter);


s.sync;

~bubs = [2400, 2600, 2500, 2700].collect{|afreq|
Synth(\bubblebub, [\freq, afreq], target: ~randomdistrib, addAction:
};

s.sync;

// "map" allows us to push the triggers from the control bus to the "t_trig"
~bubs.do{|bub, bubindex| bub.map(\t_trig, ~bubtrigbus.index + bubindex) };
};
)

```



Note, instead of using the "bubbletrigs" synth (which is a direct port of the pd example) we could use Patterns to trigger bubble synths. This is a different model for resource management: instead of having four always-running synths which re-trigger to create a new bubble, we create one synth whenever we need a bubble, and it frees itself after.

```


(
p = Pbind(
\instrument, \bubblebub,
// The commented version is a bit like the above timings...
// \dur, Pseq([29, 37, 47, 67, 89, 113, 157, 197, 200].differentiate * 0.015,
// ...but happily we have useful random-distrib generators. Ppoisson would be

```

```

    \dur, Pgauss(0.3, 0.2),
    \freq, Pwhite(0,1,inf).linexp(0,1, 1000, 3000),
    // doneAction of two allows the synths to free themselves. See "UGen-doneAc:
    \doneAction, 2
).play
)
p.stop

```



This next one's a bit more complex - we do as the book does and make smaller bubbles have (a) higher pitch (b) lower volume (c) shorter duration. To connect these values together we define a "sizefactor" and use Pkey to reuse it in each of the args.

```

(
p = Pbind(
    \instrument, \bubblebub,
    \sizefactor, Pwhite(0,1,inf),
    \dur, Pgauss(0.3, 0.2),
    \freq, Pkey(\sizefactor).linexp(0, 1, 1000, 3000),
    \amp, Pkey(\sizefactor).linlin(0, 1, 0.15, 0.04),
    \decay, Pkey(\sizefactor).linlin(0, 1, 0.05, 0.08),
    \doneAction, 2
).play
)
p.stop

```

These SuperCollider code examples are by Dan Stowell, based on techniques and examples in *Designing Sound* by Andy Farnell.

Fig 36.3: random-frequency sines

Not a physically realistic model, but notice how it has some of the same aural qualities:

```

x = {SinOsc.ar(LFNoise0.kr(170).range(800, 2400), 0, 0.3)}.play;
x.free;

```

Fig 36.4: rising pitches

Tweak the above so that the sounds drift in pitch, tending to rise:

```

(
x = {
    var trigs, freq;
    trigs = Dust.kr(170);
    freq =
        // Generally choose from a varied base freq
        TExpRand.kr(800, 2000, trigs)
        // Wobbly variation
        + LFNoise2.kr(20, mul: 300)
        // General tendency for upward rise
        + EnvGen.kr(Env.perc(1).range(0,17), trigs)
}
)

```

```

        SinOsc.ar(freq, 0, 0.3)
    }.play;
)
x.free;

// hmmm, let's try combining a few of these in parallel.
// do we sound like a river yet?
(
x = {
    var trigs, freq;
    6.collect{
        trigs = Dust.kr(170);
        freq =
            // Generally choose from a varied base freq
            TExpRand.kr(800, 2000, trigs)
            // Wobbly variation
            + LFNoise2.kr(20, mul: 300)
            // General tendency for upward rise
            + EnvGen.kr(Env.perc(1).range(0,17), trigs)
            ;
        SinOsc.ar(freq, 0, 0.3)
    }.mean
}.play;
)
x.free;

```

These SuperCollider code examples are by Dan Stowell, based on techniques and examples in *Designing Sound* by Andy Farnell.

Fig 38.1: Fluid sphere impact

The waveform produced by impact of fluid sphere on hard surface - we'll create it as a Function here so we can reuse it:

```

~dropletonhard = {|dur=0.0005| Env.perc(dur/2, dur/2, 1, [-4, 4])};
~dropletonhard.value.plot

```

Fig 38.3: obtaining a Gaussian noise distribution.

The Central Limit Theorem tells us that adding together independent noise sources will eventually give us gaussian noise. Twelve is certainly enough for auditory purposes.

```

x = { {WhiteNoise.ar}.dup(12).mean.dup * 0.5}.play;
x.free;

```

In the book, Andy says that it is more efficient to use two white noise sources and to create a gaussian shape using the Box-Muller transform. However, this involves some relatively heavy mathematical operations (log, cos, sqrt) so it's not actually obvious which approach is more efficient on a given system. Compare the CPU usage (and the audio result) of the above, with the following:

```

(

```



```

x = {
  var amount=MouseX.kr(0,1); // move mouse left/right to change amount
  var n1 = WhiteNoise.ar.abs * amount + (1-amount);
  var n2 = WhiteNoise.ar(2pi);
  var gaussian = sqrt(-2 * log(n1)) * cos(n2);
  gaussian.dup * 0.5
}.play;
)
x.free;

```

Fig 38.4: raindrop pressure on solid ground

```

(
x = {
  var gaus, osc;
  gaus = {WhiteNoise.ar}.dup(12).sum;
  gaus = LPF.ar(BPF.ar(gaus, 50, 1/0.4), 500);

  //
  osc = SinOsc.ar(gaus.linlin(-1, 1, 40, 80)) * gaus.squared * 10;
  osc = (osc - 0.35).max(0);

  2.do{
    osc = HPF.ar(osc, 500);
  };

  osc.dup
}.play
)

```

These SuperCollider code examples are by Dan Stowell, based on techniques and examples in *Designing Sound* by Andy Farnell.

Retrieved from "http://en.wikibooks.org/wiki/Designing_Sound_in_SuperCollider/Print_version"

- This page was last modified on 5 December 2009, at 15:29.
- Text is available under the Creative Commons Attribution/Share-Alike License; additional terms may apply. See Terms of Use for details.