# A Practical Guide to Patterns

**H. James Harkins**

## Introduction

Patterns are one of the most powerful elements of the SuperCollider language, but in some ways they can be difficult to approach using only the class-oriented help files. These documents seek to bridge the gap, explaining the conceptual background behind patterns, describing the usage of specific Pattern classes, and proceeding into examples of practical musical tasks written as patterns.

**Contents**



The **pattern cookbook** is a set of examples with explanations.



**Reference material**



**Why patterns?**

Patterns describe calculations without explicitly stating every step. They are a higher-level representation of a computational task. While patterns are not ideally suited for every type of calculation, when they are appropriate they free the user from worrying about every detail of the process. Using patterns, one writes *what* is supposed to happen, rather than *how* to accomplish it.

In SuperCollider, patterns are best for tasks that need to produce sequences, or streams, of information. Often these are numbers, but they don't have to be -- patterns can generate any kind of object.

For a simple example, let's count upward starting from 0. We don't know how high we will need to count; we just know that every time we ask for values, we should get a continually increasing series.

Writing everything out, it looks like this. Routine is used because this is a control structure that can interrupt what it's doing and remember where it was, so that it can pick up again at exactly that point. You can get some numbers out of it, and call it again later and it will keep counting from the last number returned. (This is an example of a Stream. You can find more about Streams in Streams-Patterns-Events1.)

```
a = Routine {
	var	i = 0;
	loop {
		i.yield;
		i = i + 1;
	};
};

a.nextN(10);
```

SuperCollider's built-in control structures allow some simplification.

```
a = Routine {
	(0..).do { |i|
		i.yield;
	};
};

a.nextN(10);
```

But wouldn't it be nice just to say, "Give me an infinite series of numbers starting with 0, increasing by 1"? With Pseries, you can. (Here, keyword addressing of the arguments is used for clarity, but 'start', 'step' and 'length' can be omitted.)

```
a = Pseries(start: 0, step: 1, length: inf).asStream;

a.nextN(10);
```

What are the advantages of the pattern representation?

- It's shorter.
- It's tested and it works. You don't have to debug how Pseries works (whereas, if you write a Routine, you might make a mistake and then have to find it.)
- With the Routine -- especially if it's complicated -- you will have to decipher it when you come back to the code later. The Pattern states the purpose right there in the code.

What are some disadvantages?

- Patterns are a new vocabulary to learn. Until you know a critical mass of them, it can be hard to trust them. That's the purpose of this guide!
- If there isn't a pattern that does quite what you want, then it might take some ingenuity to combine patterns into new designs. (Custom behaviors can always be written using Proutine.)

Using patterns for sequencing might seem to be an advanced usage, but for many uses they are easier than the equivalent code written out step by step. They can serve as a bridge for new and advanced users alike, to represent a musical conception more directly with less connective tissue explicitly stated.

The first step in learning a new language is vocabulary, so the next chapter will concentrate on foundational patterns to generate data streams of nearly every sort.

**Patterns versus Streams**

Some context that is important to keep in mind throughout this discussion is the difference between patterns and streams. In the most general terms:

*Patterns define behavior; streams execute it.*

A pattern is like a blueprint for a building, showing how all the parts fit together. The building doesn't exist until the contractors go and do what the plans specify. When a stream is made from a pattern, it follows the plans laid out in the pattern's blueprint. Rendering the plans into a real-world result does not change the blueprint in any way, but to get the result, the stream has to go through different states.

A pattern is supposed to describe behavior, and in general, evaluating the pattern (by way of a stream) should not change anything in the Pattern object itself. In computer science terms, patterns are *stateless*; their definition does not change over time. The stream is what keeps track of where we are in the pattern's evaluation.

This explains an easy "gotcha" with patterns -- forgetting to turn the pattern into a stream doesn't get the expected result. Since a pattern doesn't have any concept of a current state, calling 'next' on it is meaningless, so 'next' does what it does for most objects: return the receiver object itself. The method 'asStream' creates the stream conforming to the pattern's specification, and calling 'next' on the stream advances to its next state and returns the new value.

```
p = Pseries(0, 1, 10);
p.next;      // always returns the Pseries, not actual numbers

q = p.asStream;
q.next;      // calling this repeatedly gets the desired increasing integers
```

There is a concrete benefit to this strict division of labor. Since the stream does not modify the original pattern, any number of streams can be made from the same blueprint. All of those streams maintain their own independent states, and they can operate concurrently without interfering with each other.

```
r = p.asStream;
r.next;      // starts from zero, even though q already gave out some numbers

q.next;      // resumes where q left off, with no effect from getting values from r

[q.next, r.next]    // and so on...
```

Bear these points in mind as we move to the next subject: getting basic types of data (deterministic and random) out of patterns.

Next: PG_02_Basic_Vocabulary

## Basic Vocabulary: Generating values

Before getting to the really cool things patterns can do, we need to build up a basic vocabulary. We'll start with some words, then move into phrases in the next tutorial.

Some of the patterns will be demonstrated with a Pbind construct. This is a taste of things to come -- sequencing sonic events using patterns. Don't worry about how Pbind works just yet... all in good time.

Let's start with a very quick reference of some basic patterns. More complete descriptions follow this list. The list might seem long at first, but concentrate your attention on patterns marked with a star. Those are the most basic, and commonly used. Again, the purpose is to start learning the vocabulary of patterns -- like learning new words when studying a human language.

This document describes a lot of patterns, but what I call "primary patterns" are the most important. If you are new to patterns, concentrate on these first. You can always come back and look at the rest later.

For more information on any of these patterns, select the class name and use the help key for your editor to open its help file.

#### Quick reference

#### Primary Patterns

- ***Pseq(list, repeats, offset):** Play through the entire list 'repeats' times. Like list.do.
- ***Prand(list, repeats):** Choose items from the list randomly (same as list.choose).
- ***Pxrand(list, repeats):** Choose randomly, but never repeat the same item twice.
- ***Pshuf(list, repeats):** Shuffle the list in random order, and use the same random order 'repeats' times.
- ***Pwrand(list, weights, repeats):** Choose randomly by weighted probabilities (like list.wchoose(weights)).

- ***Pseries(start, step, length):** Arithmetic series (addition).
- ***Pgeom(start, grow, length):** Geometric series (multiplication).

- ***Pwhite(lo, hi, length):** Random numbers, equal distribution ("white noise"). Like rrand(lo, hi).
- ***Pexprand(lo, hi, length):** Random numbers, exponential distribution. Like exprand(lo, hi).
- ***Pbrown(lo, hi, step, length):** Brownian motion, arithmetic scale (addition).

- ***Pfunc(nextFunc, resetFunc):** Get the stream values from a user-supplied function.
- ***Pfuncn(func, repeats):** Get values from the function, but stop after 'repeats' items.
- ***Proutine(routineFunc):** Use the function like a routine. The function should return values using .yield or .embedInStream.

#### Additional List Patterns

- **Pser(list, repeats, offset):** Play through the list as many times as needed, but output only 'repeats' items.
- **Pslide(list, repeats, len, step, start, wrapAtEnd):** Play overlapping segments from the list.

- **Pwalk(list, stepPattern, directionPattern, startPos):** Random walk over the list.

- **Place(list, repeats, offset):** Interlace any arrays found in the main list.
- **Ppatlace(list, repeats, offset):** Interlace any patterns found in the main list.
- **Ptuple(list, repeats):** Collect the list items into an array as the return value.

#### Additional Random Number Generators

- **Pgbrown(lo, hi, step, length):** Brownian motion, geometric scale (multiplication).

- **Pbeta(lo, hi, prob1, prob2, length):** Beta distribution, where prob1 = α and prob2 = β.
- **Pcauchy(mean, spread, length):** Cauchy distribution.
- **Pgauss(mean, dev, length):** Guassian (normal) distribution.
- **Phprand(lo, hi, length):** Returns the greater of two equal-distribution random numbers.
- **Plprand(lo, hi, length):** Returns the lesser of two equal-distribution random numbers.

- **Pmeanrand(lo, hi, length):** Returns the average of two equal-distribution random numbers, i.e., (x + y) / 2.
- **Ppoisson(mean, length):** Poisson distribution.

- **Pprob(distribution, lo, hi, length, tableSize):** Arbitrary distribution, based on a probability table.

**Functional descriptions of patterns**

**List Patterns**

The most obvious thing one would want to do with a pattern is to give it a list of values and have it read them out in order. You have a couple of choices, which differ in their handling of the 'repeats' parameter.

**Pseq(list, repeats, offset):** Play through the entire list 'repeats' times.
**Pser(list, repeats, offset):** Play through the list as many times as needed, but output only 'repeats' items.

```
Pseq(#[1, 2, 3], 4).asStream.all;      // 12 items = 4 repeats * 3 items
Pser(#[1, 2, 3], 4).asStream.all; // 4 items only
```

Pseq is an obvious choice for streaming out known pitch and rhythm values.

```
(
Pbind(
      \degree, Pseq(#[0, 0, 4, 4, 5, 5, 4], 1),
      \dur, Pseq(#[0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 1], 1)
).play;
)
```

A variation, Pslide, plays overlapping segments of the input list.

**Pslide(list, repeats, len, step, start, wrapAtEnd)**
repeats: number of segments
len: length of each segment
step: is how far to step the start of each segment from previous.
start: what index to start at.
wrapAtEnd: if true (default), indexing wraps around if goes past beginning or end. If false, the pattern stops if it hits a nil element or goes outside the list bounds.

If step == 1, then the first segment is at 'start', the second at 'start' + 1, and so on.

```
Pslide(#[1, 2, 3, 4, 5, 6, 7, 8], 10, 3, 1, 0, false).asStream.all;

// or, to show the segments as separate arrays
Pslide(#[1, 2, 3, 4, 5, 6, 7, 8], 10, 3, 1, 0, false).clump(3).asStream.all;

// Flock of Seagulls!
(
Pbind(
      \degree, Pslide((-6, -4 .. 12), 8, 3, 1, 0),
      \dur, Pseq(#[0.1, 0.1, 0.2], inf),
      \sustain, 0.15
).play;
)
```

**Random-order list patterns**

**Prand(list, repeats):** Choose items from the list randomly (same as list.choose).

```
// Prand: given scale degrees (pentatonic) with equal probability of each
(
Pbind(
      \degree, Prand([0, 1, 2, 4, 5], inf),
      \dur, 0.25
).play;
)
```

**Pxrand(list, repeats):** Choose randomly, but never repeat the same item twice in immediate succession.

```
// Pxrand: same as above but never repeats a pitch twice in a row
(
Pbind(
	\degree, Pxrand([0, 1, 2, 4, 5], inf),
	\dur, 0.25
).play;
)
```

**Pshuf(list, repeats):** Shuffle the list in random order, and use the same random order 'repeats' times.

```
// Pshuf: randomly ordered once and repeated
(
Pbind(
	\degree, Pshuf([0, 1, 2, 4, 5], inf),
	\dur, 0.25
).play;
)
```

**Pwrand(list, weights, repeats):** Choose randomly, according to weighted probabilities (same as list.wchoose(weights)).

```
// Pwrand: these probabilities favor triadic notes from scale degrees
(
Pbind(
	\degree, Pwrand((0..7), [4, 1, 3, 1, 3, 2, 1].normalizeSum, inf),
	\dur, 0.25
).play;
)
```

**Pwalk(list, stepPattern, directionPattern, startPos):** Random walk over the list. This pattern is a bit more complicated; see its help file for details.

**Interlacing values and making arrays**

These are opposing operations: interlacing means splitting arrays and merging them into a stream of single values, and arrays can be made out of single-value streams as well.

**Place(list, repeats, offset):** Take one from each item in the main array item in succession. Hard to explain, easier to see:

```
Place([0, [1, 2], [3, 4, 5]], 3).asStream.all;
--> [ 0, 1, 3, 0, 2, 4, 0, 1, 5 ]
```

If we turn this into a matrix and read vertically, the original arrays are clearly visible:

```
Place([0, [1, 2], [3, 4, 5]], 3).clump(3).do(_.postln);

[ 0, 1, 3 ]// leftmost column: 0 from first Place item
[ 0, 2, 4 ]// second column: alternates between 1 and 2, from second Place item
[ 0, 1, 5 ]// third column: 3, 4, 5 from third Place item
```

**Ppatlace(list, repeats, offset):** Take one value from each sub-pattern in order.

```
// Hanon exercise
(
Pbind(
	\degree, Ppatlace([
		Pseries(0, 1, 8),		// first, third etc. notes
		Pseries(2, 1, 7)		// second, fourth etc. notes
	], inf),
	\dur, 0.25
).play;
)
```

That's also a taste of things to come: Patterns can be nested.

**Ptuple(list, repeats):** Get one value from each item in the array, and return all of them as an array of values.

```
// Chords
// \degree receives [7, 9, 4], then [6, 7, 4] successively, expanded to chords on the server
(
Pbind(
	\degree, Ptuple([
		Pseries(7, -1, 8),
		Pseq([9, 7, 7, 7, 4, 4, 2, 2], 1),
		Pseq([4, 4, 4, 2, 2, 0, 0, -3], 1)
	], 1),
	\dur, 1
).play;
)
```

**Generating values**

**Arithmetic and geometric series**

Now, let's move to patterns that produce values mathematically, without using a predefined list.

**Pseries(start, step, length):** Arithmetic series, successively adding 'step' to the starting value, returning a total of 'length' items.
**Pgeom(start, grow, length):** Geometric series, successively multiplying the current value by 'grow'.

```
// Use Pseries for a scale and Pgeom for an accelerando
(
Pbind(
	\degree, Pseries(-7, 1, 15),
	\dur, Pgeom(0.5, 0.89140193218427, 15)
).play;
)
```

**Third-party extension alert:** If you want an arithmetic or geometric series to start at one number and end at another specific number, the step size/multiplier must be calculated from the endpoints and the number of items desired. The ddwPatterns quark includes a convenience method, **fromEndpoints**, for both Pseries and Pgeom that performs this calculation. It's necessary to give an exact number of repeats, at least two and less than infinity.

```
p = Pgeom.fromEndpoints(0.5, 0.1, 15);  // error if ddwPatterns not installed
p.postcs;

Prints:
Pgeom(0.5, 0.89140193218427, 15)
```

**Random numbers and probability distributions**

- **Pwhite(lo, hi, length):** Produces 'length' random numbers with equal distribution ('white' refers to white noise).
- **Pexprand(lo, hi, length):** Same, but the random numbers have an exponential distribution, favoring lower numbers. This is good for frequencies, and also durations (because you need more notes with a shorter duration to balance the weight of longer notes).
- **Pbrown(lo, hi, step, length):** Brownian motion. Each value adds a random step to the previous value, where the step has an equal distribution between -step and +step.
- **Pgbrown(lo, hi, step, length):** Brownian motion on a geometric scale. Each value multiplies a random step factor to the previous value.

- **Pbeta(lo, hi, prob1, prob2, length):** Beta distribution, where prob1 = α and prob2 = β.
- **Pcauchy(mean, spread, length):** Cauchy distribution.
- **Pgauss(mean, dev, length):** Guassian (normal) distribution.
- **Phprand(lo, hi, length):** Returns the greater of two equal-distribution random numbers.
- **Plprand(lo, hi, length):** Returns the lesser of two equal-distribution random numbers.
- **Pmeanrand(lo, hi, length):** Returns the average of two equal-distribution random numbers, i.e., (x + y) / 2.
- **Ppoisson(mean, length):** Poisson distribution.

- **Pprob(distribution, lo, hi, length, tableSize):** Given an array of relative probabilities across the desired range (a histogram) representing an arbitrary distribution, generates random numbers corresponding to that distribution.

To see a distribution, make a histogram out of it.

```
Pmeanrand(0.0, 1.0, inf).asStream.nextN(10000).histo(200, 0.0, 1.0).plot;
```

**Catchall Patterns**

Not everything is pre-written as a pattern class. These patterns let you embed custom logic.

**Pfunc(nextFunc, resetFunc):** The next value is the return value from evaluating nextFunc. If .reset is called on a stream made from this pattern, resetFunc is evaluated. The stream will run indefinitely until nextFunc returns nil.

**Pfuncn(func, repeats):** Like Pfunc, output values come from evaluating the function. Pfuncn, however, returns exactly 'repeats' values and then stops. The default number of repeats is 1.

**Proutine(routineFunc):** Use the routineFunc in a routine. The stream's output values are whatever this function .yields. Proutine ends when it yields nil.

Next, we'll look at the central pattern for audio sequencing: **Pbind**.

Previous: PG_01_Introduction
Next: PG_03_What_Is_Pbind

## What's that Pbind thing?

Some of the examples in the last tutorial played notes using Pbind, and you might be wondering how it works in general and what else you can do with it.

In the most general sense, Pbind is just a way to give names to values coming out of the types of patterns we just saw. When you ask a Pbind stream for its next value, the result is an object called an Event. Like a Dictionary (which is a superclass of Event), an event is a set of "key-value pairs": each value is named by a key.

```
e = (freq: 440, dur: 0.5);     // an Event

e.at(\freq)                    // access a value by name
e[\freq]
e.freq              // See IdentityDictionary help for more on this usage

e.put(\freq, 880);  // Change a value by name
e[\freq] = 660;
e.freq = 220;

e.put(\amp, 0.6);  // Add a new value into the event
e.put(\dur, nil);              // Remove a value
```

A Pbind is defined by a list of pairs: keys associated with the patterns that will supply the values for the events.

Things get interesting when the names associated with Pbind's sub-patterns are also SynthDef arguments. Then it becomes possible to play new Synths with Pbind, and feed their inputs with different values on each event.

**Building an event, one key at a time**

We can look at the return values from a Pbind by calling next on the stream. Note that it's necessary to pass an empty event into *next*, so that Pbind has somewhere to put the values.

```
(
p = Pbind(
        \degree, Pseq(#[0, 0, 4, 4, 5, 5, 4], 1),
        \dur, Pseq(#[0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 1], 1)
).asStream;        // remember, you have to make a stream out of the pattern before using it
)

p.next(Event.new);                    // shorter: p.next(())

// Output:
( 'degree': 0, 'dur': 0.5 )
( 'degree': 0, 'dur': 0.5 )
( 'degree': 4, 'dur': 0.5 )
( 'degree': 4, 'dur': 0.5 )
```

The return events show us what Pbind really does. Each time the next value is requested, it goes through each key-pattern pair and gets the next value from each pattern (actually streams, but Pbind makes streams out of the sub patterns internally). Each value gets put into the event, using the associated key.

For the first event, the first key is 'degree' and the value is 0. This is placed into the event before moving to the next pair: the event in transition contains ( 'degree': 0 ). Then the next key supplies 0.5 for 'dur', and since there are no more pairs, the event is complete: ( 'degree': 0, 'dur': 0.5 ).

```
// User does:
p.next(Event.new);

// SuperCollider processes:
1. \degree stream returns 0
2. Put it in the Event: ( 'degree': 0 )
3. \dur stream returns 0.5
4. Put it in the Event: ( 'degree': 0, 'dur': 0.5 )
5. Return the result event.
```

**Note:** Dictionaries in SuperCollider are *unordered* collections. Even though Pbind processes its child streams in the order given, the results can display the keys and values in any order. This does not affect the behavior of playing Events, as we will soon see.

### Event, .play and event prototypes

So far we haven't seen anything that produces a note, just data processing: fetching values from patterns and stitching them together into events. The notes come from the difference between Events and regular Dictionaries: Events can do things when you **.play** them.

```
( 'degree': 0, 'dur': 0.5 ).play;
```

The action that the event will take is defined in an "event prototype." The prototype must include a function for the 'play' key; this function is executed when .play is called on the event. Also, optionally the prototype can contain default values for a wide variety of parameters.

Pbind doesn't do much without an event prototype. Fortunately, you don't have to write the prototype on your own. There is a default event, accessed by Event.default, that includes functions for many different server-messaging tasks. If no specific action is requested, the normal action is to play a Synth. That's why playing a Pbind, as in the previous tutorial, with only 'degree' and 'dur' patterns produced notes: each event produces at least one synth by default, and the default event prototype knows how to convert scale degrees into frequencies and 'dur' (duration) into note lengths.

When a pattern is played, an object called EventStreamPlayer is created. This object reads out the events one by one from the pattern's stream (using a given event prototype as the base), and calls 'play' on each. The 'delta' value in the event determines how many beats to wait until the next event. Play continues until the pattern stops producing events, or you call .stop on the EventStreamPlayer. (Note that calling .stop on the pattern does nothing. Patterns are stateless and cannot play or stop by themselves.)

**To sum up so far:** A Pbind's stream generates Events. When an Event is played, it does some work that usually makes noise on the server. This work is defined in an event prototype. The Event class provides a default event prototype that includes powerful options to create and manipulate objects on the server.

#### Useful Pbind variant: Pmono

Pbind plays separate notes by default. Sometimes, you might need a pattern to act more like a monophonic synthesizer, where it plays just one Synth node and changes its values with each event. If Pbind normally corresponds to Synth.new or /s_new, Pmono corresponds to aSynth.set or /n_set.

Compare the sound of these patterns. Pbind produces an attack on every note, while Pmono glides from pitch to pitch.

```
p = Pbind(\degree, Pwhite(0, 7, inf), \dur, 0.25, \legato, 1).play;
p.stop;

p = Pmono(\default, \degree, Pwhite(0, 7, inf), \dur, 0.25).play;
p.stop;
```

Articulating phrases is possible with Pmono by chaining several Pmono patterns together in a row, or by using PmonoArtic.

### Connecting Event values to SynthDef inputs

Most SynthDefs have Control inputs, usually defined by arguments to the UGen function. For example, the default SynthDef (declared in Event.sc) defines five inputs: out, freq, amp, pan and gate.

```
SynthDef(\default, { arg out=0, freq=440, amp=0.1, pan=0, gate=1;
	var z;
	z = LPF.ar(
				Mix.new(VarSaw.ar(freq + [0, Rand(-0.4,0.0), Rand(0.0,0.4)], 0, 0.3)),
				XLine.kr(Rand(4000,5000), Rand(2500,3200), 1)
		) * Linen.kr(gate, 0.01, 0.7, 0.3, 2);
	OffsetOut.ar(out, Pan2.ar(z, pan, amp));
}, [\ir]);
```

When an event plays a synth, any values stored in the event under the same name as a SynthDef input will be passed to the new synth. Compare the following:

```
// Similar to Synth(\default, [freq: 293.3333, amp: 0.2, pan: -0.7])
(freq: 293.3333, amp: 0.2, pan: -0.7).play;

// Similar to Synth(\default, [freq: 440, amp: 0.1, pan: 0.7])
(freq: 440, amp: 0.1, pan: 0.7).play;
```

This leads to a key point: **The names that you use for patterns in Pbind should correspond to the arguments in the SynthDef being played.** The Pbind pattern names determine the names for values in the resulting Event, and those values are sent to the corresponding Synth control inputs.

The SynthDef to play is named by the 'instrument' key. To play a pattern using a different Synth, simply name it in the pattern.

```
SynthDef(\harpsi, { |outbus = 0, freq = 440, amp = 0.1, gate = 1|
        var out;
        out = EnvGen.ar(Env.adsr, gate, doneAction: 2) * amp *
                Pulse.ar(freq, 0.25, 0.75);
        Out.ar(outbus, out ! 2);
}).memStore;       // see below for more on .memStore

p = Pbind(
                // Use \harpsi, not \default
        \instrument, \harpsi,
        \degree, Pseries(0, 1, 8),
        \dur, 0.25
).play;
```

It's actually an oversimplification to say that the Pbind names should always match up to SynthDef arguments.

- A Pbind can use some values in the event for intermediate calculations (see PG_06g_Data_Sharing). If these intermediate values have names not found in the SynthDef, they are not sent to the server. There is no requirement that every item in an Event must correspond to a SynthDef control.
- The default event prototype performs some automatic conversions. You might have noticed that the examples so far use 'degree' to specify pitch, but the default SynthDef being played does not have a degree argument. It works because the default event converts degree into 'freq', which is an argument. The most important conversions are for pitch and timing. Timing is simple; pitch is more elaborate. See PG_07_Value_Conversions for an explanation of these automatic calculations.

**Don't send or load SynthDefs; use .memStore or .store instead**

To send only the relevant values to the new Synth, the Event needs to know what controls exist in the SynthDef. This is done by a library of descriptors for SynthDefs; the descriptor is a SynthDesc, and the library is a SynthDescLib. The normal methods -- .send(s), .load(s) -- to communicate a SynthDef to the server do not enter it into the library. As a result, SynthDefs sent this way will not work properly with Pbind. Instead, use different methods that *store* the SynthDef into the library.

```
// Save into the library, write a .scsyndef file, and load it on the server
SynthDef(...).store;

// Save into the library and send the SynthDef to the server (no .scsyndef file)
// Make sure the server is booted before doing this
SynthDef(...).memStore;
```

~~.load(s)~~ --> .store
~~.send(s)~~ --> .memStore

**"Rest" events**

The convention to include a rest in the middle of an event pattern is to set the \freq key to a Symbol. Commonly this is \rest, but a backslash by itself is enough to suppress the note on the server. Ligeti's "touches bloquées" technique could be written this way (see PG_06e_Language_Control for an explanation of the conditional Pif):

```
(
// first, pitches ascending by 1-3 semitones, until 2 octaves are reached
```

```
var	pitches = Pseries(0, Pconst(24, Pwhite(1, 3, inf)), inf).asStream.all,
		// randomly block 1/3 of those
	mask = pitches.scramble[0 .. pitches.size div: 3];

p = Pbind(
	\arpeg, Pseq(pitches[ .. pitches.size - 2] ++ pitches.reverse[ .. pitches.size - 2], inf),
		// if the note is found in the mask array, replace it with \rest
		// then that note does not sound
	\note, Pif(Pfunc { |event| mask.includes(event[\arpeg]) }, \rest, Pkey(\arpeg)),
	\octave, 4,
	\dur, 0.125
).play;
)

p.stop;
```

If it's the \freq key that determines whether the event as a rest or not, why does it work to use it with \note? As noted, keys like \degree, \note, and \midinote are automatically converted into frequency. The math operations that perform the conversion preserve Symbols intact -- e.g., \rest + 1 == \rest. So the \rest value is passed all the way through the chain of conversions so that \freq in the event ends up receiving \rest.

Note that it doesn't matter if the SynthDef has a 'freq' argument. It's the event, on the *client* side, that looks to this key to determine whether to play the note or not. If it is a rest, the server is not involved at all.

## Writing SynthDefs for patterns

SynthDefs should have a couple of specific features to work well with patterns.

### Synths should release themselves

The default event prototype relies on the synth to remove itself from the server when it's finished. This can be done in several ways:

- (Most typical) A gated envelope with a releasing doneAction ( >= 2) in the envelope generator (see UGen-doneActions for a complete list). The \harpsi SynthDef above uses this technique. A gated envelope specifies a release node or uses one of the predefined sustaining envelope types: Env.asr, Env.adsr, Env.dadsr. The Env help file offers more detail on gated envelopes.
- Linen.kr, which is a shortcut for EnvGen.kr(Env([0, susLevel, 0], [attackTime, releaseTime], \lin, releaseNode: 1), gate, doneAction: [2 or higher]). The default SynthDef uses this. The doneAction should be at least 2 to release the node.
    - **Note:** If the release is controlled by a gate, the gate must be represented by the synth argument 'gate'; standard event prototypes expect to be able to control the synth's release using this argument. Also, make sure the gate's default value is greater than 0. Otherwise, the envelope will never start and you will both hear nothing and watch synths pile up on the server.
- Fixed-duration envelopes (no gate).

### Argument name prefixes

One other subtle point about synth argument names. In a SynthDef, argument names can have the prefix **t**_ to indicate a "trigger control," or **i**_ for an "initial rate" control (meaning that it holds the value set when the Synth is first played). This is described in SynthDef help. Pbind and its cousins should leave out the prefixes, e.g.:

```
(
SynthDef(\trig_demo, { |freq, gate = 1, t_trig = 1|   // t_trig here
	var	env = Decay2.kr(t_trig, 0.01, 0.1),
		sig = SinOsc.ar(freq, 0, env)
			* Linen.kr(gate, 0.01, 0.1, 0.1, doneAction: 2);
	Out.ar(0, sig ! 2)
}).memStore;
)

(
p = Pmono(\trig_demo,
	\freq, Pexprand(200, 800, inf),
	\trig, 1,	// note that this is NOT t_trig -- just \trig
```

```
        \delta, 0.125
).play;
)

p.stop;
```

Previous: PG_02_Basic_Vocabulary
Next: PG_04_Words_to_Phrases

## From words to phrases

A single pattern by itself is not so exciting. But patterns can be used together to get more complex results.

**Patterns within list patterns**

We saw list patterns (Pseq, Prand, etc.) that returned numbers from a preset list, either in the order given or rearranged randomly. The list may also include other patterns. When a list pattern encounters another pattern in its list, the inner pattern is *embedded* into the stream. That is, the inner pattern takes over until it runs out of values to return. Then, control returns to the outer list pattern. This is like calling a function in the middle of another function.

There is no preset limit to the number of levels of embedding.

If a single pattern is like a word, a list pattern that uses other patterns could be more like a sentence or phrase. You can alternate between different behaviors, either in a predictable order as in the example below, or randomly by using one of the random-order list patterns.

```
// Scale segments, in the sequence: up, up, down (repeat)
(
TempoClock.default.tempo = 1;
p = Pbind(
	\degree, Pseq([
		Pseries({ rrand(0, 7) }, 1, { rrand(4, 8) }),	// up (step = 1)
		Pseries({ rrand(0, 7) }, 1, { rrand(4, 8) }),	// up (step = 1)
		Pseries({ rrand(7, 14) }, -1, { rrand(4, 8) })	// down (step = -1)
	], inf),
	\dur, 0.125
).play;
)

p.stop;
```

But it gets even more fun -- list patterns don't care whether they're enclosing value patterns (as in the previous example) or event patterns. That means you can write a set of Pbind-style patterns, each one representing a phrase, and string them together. This next example is longer, but that's only because of a larger number of phrase patterns. The structure is very simple, though: **Pxrand([Pbind(), Pmono(), Pmono()...], inf)**. Some of the phrases are written with Pmono to slide between notes.

```
(
SynthDef(\bass, { |freq = 440, gate = 1, amp = 0.5, slideTime = 0.17, ffreq = 1100, width =
0.15,
		detune = 1.005, preamp = 4|
	var	sig,
		env = Env.adsr(0.01, 0.3, 0.4, 0.1);
	freq = Lag.kr(freq, slideTime);
	sig = Mix(VarSaw.ar([freq, freq * detune], 0, width, preamp)).distort * amp
		* EnvGen.kr(env, gate, doneAction: 2);
	sig = LPF.ar(sig, ffreq);
	Out.ar(0, sig ! 2)
}).memStore;
)

(
TempoClock.default.tempo = 132/60;
p = Pxrand([
	Pbind(			// repeated notes
		\instrument, \bass,
		\midinote, 36,
		\dur, Pseq([0.75, 0.25, 0.25, 0.25, 0.5], 1),
		\legato, Pseq([0.9, 0.3, 0.3, 0.3, 0.3], 1),
		\amp, 0.5, \detune, 1.005
	),
	Pmono(\bass,		// octave jump
		\midinote, Pseq([36, 48, 36], 1),
		\dur, Pseq([0.25, 0.25, 0.5], 1),
```

```
                \amp, 0.5, \detune, 1.005
        ),
        Pmono(\bass,                    // tritone jump
                \midinote, Pseq([36, 42, 41, 33], 1),
                \dur, Pseq([0.25, 0.25, 0.25, 0.75], 1),
                \amp, 0.5, \detune, 1.005
        ),
        Pmono(\bass,                    // diminished triad
                \midinote, Pseq([36, 39, 36, 42], 1),
                \dur, Pseq([0.25, 0.5, 0.25, 0.5], 1),
                \amp, 0.5, \detune, 1.005
        )
], inf).play(quant: 1);
)

p.stop;
```

**Shortcut notation:** Just like you can concatenate arrays with ++, you can also concatenate patterns the same way. Writing **pattern1 ++ pattern2** is the same as writing **Pseq([pattern1, pattern2], 1)**.

**Some ways to string together patterns**

**Sequentially:** Each sub-pattern follows the next in the same order every time. Use Pseq or Pser.
**Randomized order:** Sub-patterns in completely random order (Prand), random order with no repeats (Pxrand), or random order according to a set of probabilities (Pwrand). Pshuf creates one random ordering and uses it repeatedly.

**Direct array indexing:** Patterns can be chosen in arbitrary order by index. This gives you more control than Pwrand. Both Pindex and Pswitch can be used for this.

```
// scale degree segments, every fifth choice is odd-numbered only (descending)
(
var     n = 10,
        scaleSegments = Array.fill(n, { |i|
                if(i.odd) {
                        Pseries(11, -1, rrand(5, 10))
                } {
                        Pseries(rrand(-4, 4), 1, i+2)
                }
        });

TempoClock.default.tempo = 1;
p = Pbind(
        \degree, Pswitch(scaleSegments, Pseq([Pwhite(0, n-1, 4), Pwhite(0, n-1, 1).select(_.odd)], inf)),
        \dur, 0.125
).play;
)

p.stop;
```

**Finite state machine (Pfsm, Pdfsm):** A finite state machine is a way of associating an item with its possible successors. It is closer to a "grammar" than purely random selection. Pfsm defines a finite state machine as a set of possible "entry points," followed by a list of the possible "states" of the machine and, for each state, a list of the possible states that may follow the current state. States may be single values or patterns, meaning that phrases can be linked to other phrases that "make sense" in succession (and unwanted transitions can be prevented).

If this sounds a bit like a Markov chain, that's because the Pfsm implementation is a special case of a Markov chain where there is an equal probability of choosing the next state from the valid successors. In a Markov chain, the probabilities are weighted according to analysis of a real-world data stream.

The Pfsm help file includes very good examples of organizing single values and pattern phrases. Also see PG_Cookbook06_Phrase_Network for an application of Pfsm to generate a corny jazz solo.

Pdfsm stands for "deterministic finite state machine," where there is no random selection.

**Third-party extension alert:** A good Markov chain implementation for SuperCollider exists in the MathLib quark.

**Library of named sub-patterns**

One very effective way to manage phrases is to make a library, or more precisely Dictionary, of sub-patterns, and then call them up one at a time. Psym is the pattern to do this. The advantage here is that you can store the phrases in a separate place, while the pattern that you actually play is much simpler and describes the musical intent at a much higher level.

```
// Uses the bass SynthDef above
(
~phrases = (
	repeated: Pbind(
		\instrument, \bass,
		\midinote, 36,
		\dur, Pseq([0.75, 0.25, 0.25, 0.25, 0.5], 1),
		\legato, Pseq([0.9, 0.3, 0.3, 0.3, 0.3], 1),
		\amp, 0.5, \detune, 1.005
	),
	octave: Pmono(\bass,
		\midinote, Pseq([36, 48, 36], 1),
		\dur, Pseq([0.25, 0.25, 0.5], 1),
		\amp, 0.5, \detune, 1.005
	),
	tritone: Pmono(\bass,
		\midinote, Pseq([36, 42, 41, 33], 1),
		\dur, Pseq([0.25, 0.25, 0.25, 0.75], 1),
		\amp, 0.5, \detune, 1.005
	),
	dim: Pmono(\bass,
		\midinote, Pseq([36, 39, 36, 42], 1),
		\dur, Pseq([0.25, 0.5, 0.25, 0.5], 1),
		\amp, 0.5, \detune, 1.005
	)
);

TempoClock.default.tempo = 128/60;

// the higher level control pattern is really simple now
p = Psym(Pxrand(#[repeated, octave, tritone, dim], inf), ~phrases).play;
)

p.stop;
```

A complicated pattern with lots of embedding can be hard to read because it's more work to separate note-level details from the larger structure. The pattern choosing the phrases -- **Pxrand(#[repeated, octave, tritone, dim], inf)** -- is self-explanatory, however, and Psym fills in the details transparently.

**Note:** Because of some special handling needed for event patterns, there are two versions of Psym. Psym handles event patterns, while Pnsym is for value patterns. Think of it this way: Pbind can be contained within Psym, but it contains Pnsym.

	( Psym ( Pbind ( Pnsym ) ) )

**Good:**

	**Psym**(**, (pattern1: **Pbind**(**))
	**Pbind**(\someValue, **Pnsym**(**, (pattern1: Pwhite(**)))

**Bad:**

	**Pbind**(\someValue, **Psym**(**, (pattern1: Pwhite(**)))

**Switching between patterns for individual values**

In the examples above, if a list pattern encounters another pattern in its input values, the subpattern is embedded in its entirety before the list pattern is allowed to continue. Sometimes you might want to get just one value out of the subpattern, and then choose a different subpattern on the next event. Pswitch, Psym and Pnsym have cousins that do exactly this: Pswitch1, Psym1 and Pnsym1.

```
// random pitches in two distinct ranges; use a coin toss to decide which for this event
// 70% low, 30% high
(
TempoClock.default.tempo = 1;
p = Pbind(
	\degree, Pswitch1([Pwhite(7, 14, inf), Pwhite(-7, 0, inf)], Pfunc { 0.7.coin.binaryValue }),
	\dur, 0.25
).play;
)

p.stop;
```

Compare to the following:

```
(
p = Pbind(
	\degree, Pswitch([Pwhite(7, 14, inf), Pwhite(-7, 0, inf)], Pfunc { 0.7.coin.binaryValue }),
	\dur, 0.25
).play;
)

p.stop;
```

With Pswitch, one of the items is chosen from the list and keeps playing until it's finished. But the length of both Pwhite patterns is infinite, so whichever one is chosen first retains control. Pswitch1 does the coin toss on every event and embeds just one item.

Psym1 and Pnsym1 behave similarly, choosing the name to look up the pattern for each event.

**Related: Conditional patterns**

Pif supports this kind of structure: If the next value from a Boolean pattern is true, return the next item from pattern A, otherwise take it from pattern B. Another way to write the Pswitch1 example is to use a Boolean test directly on Pwhite, instead of writing a Pfunc for the coin toss. This might be clearer to read. However, this works only when there are two alternatives. Pswitch1 and Psym1 allow any number of choices.

```
(
TempoClock.default.tempo = 1;
p = Pbind(
		// translation: if(0.7.coin) { rrand(-7, 0) } { rrand(7, 14 }
	\degree, Pif(Pwhite(0.0, 1.0, inf) < 0.7, Pwhite(-7, 0, inf), Pwhite(7, 14, inf)),
	\dur, 0.25
).play;
)

p.stop;
```

We will see in PG_06e_Language_Control that Pif can be used on values that were previously calculated in the Pbind. It adds considerably to the intelligence Pbind can manage, when its value streams are aware of other values in the event.

Previous: PG_03_What_Is_Pbind
Next: PG_05_Math_on_Patterns

## Math on patterns

Often, there is not a pattern that delivers exactly the desired result by itself. But, other operations can be applied to patterns, to manipulate one pattern's output and turn it into something else.

Some of these operations look like things you would do to an array, but there is a critical difference. Doing math on an array performs the operation on every array item all at once. By contrast, patterns are "lazy" -- they evaluate one value at the time, only when asked, and they only do as much as they need to do to deliver the next value. An operation on a pattern produces another pattern that remembers the work that is to be done. Making a stream out of the composite pattern creates the structure to perform the operation upon request.

For example, multiplying a pattern by a number produces a "binary operator pattern": Pbinop. Looking at the Pbinop's variables reveals everything that is needed to reconstruct the operation on demand.

```
p = Pwhite(1, 5, inf) * 2;      // a Pbinop

p.operator          // == '*'
p.a                 // == a Pwhite
p.b                 // == 2
```

In other words, the multiplication here produces not the result of a single multiplication, but a template for an infinite stream of multiplications to follow.

**Math on patterns**

Not only can patterns generate numbers, but they also support all the standard math operators: unary (abs, reciprocal, etc.), binary (+, -, *, /, **, min, max, etc.) and n-ary (clip, wrap, fold, linlin, linexp, etc.) operators are all valid with patterns.

```
// Random integers, 1-5
Pwhite(1, 5, inf).asStream.nextN(10);

// Random integers 1-5, multiplied by two gives even integers 2-10
(Pwhite(1, 5, inf) * 2).asStream.nextN(10);

// Random integers 1-5, multiplied by 1/4 gives multiples of 1/4 between 0.25 and 1.25
(Pwhite(1, 5, inf) * 0.25).asStream.nextN(10);

// Random integers 1-5, with the sign (positive or negative) randomly chosen
(Pwhite(1, 5, inf) * Prand(#[-1, 1], inf)).asStream.nextN(10);
```

If a binary operation occurs on two patterns, every time a value is requested from the resulting stream, both of the component streams are asked for a value, and the operator applies to those results. If either stream ends, the binary operator stream also ends.

```
// The resulting stream has two values, because the shorter operand stream has two values.
(Pseq([10, 9, 8], 1) + Pseq([1, 2], 1)).do { |x| x.postln };
```

The binary operator adverb '.x' is supported with patterns. (See Adverbs.) This adverb is like a nested loop: in streamA +.x streamB, the first value of streamA is added to every value of streamB in succession, then the second value of streamA is added to every streamB value, and so on. This is an easy way to transpose a pattern to different levels successively.

```
// Play a major-7th arpeggio, transposed to different scale degrees
// Pwhite is the transposer; Pseq is the chord
// The chord is like an "inner loop"
(
p = Pbind(
        \midinote, Pwhite(48, 72, inf) +.x Pseq(#[0, 4, 7, 11], 1),
        \dur, 0.125
).play;
)

p.stop;
```

**Collection operations on patterns**

Some of the things you can do to arrays also work with patterns.

**collect(func):** Applies the function to each return value from the pattern. Good for generic transformations.
**select(func):** Preserve values from the output stream that pass the Boolean test; discard the rest.
**reject(func):** Discard values from the output stream that pass the test; return the rest to the user.

```
// Arbitrary/custom operation: Turn each number into a two-digit hex string
Pwhite(0, 255, 20).collect({ |x| x.asHexString(2) }).do { |x| x.postln };

// Keep odd numbers in the result (which is now less than 20 items)
Pwhite(0, 255, 20).select({ |x| x.odd }).do { |x| x.postln };

// Throw out odd numbers in the result
Pwhite(0, 255, 20).reject({ |x| x.odd }).do { |x| x.postln };
```

**clump(n):** Calling .clump on an array turns a flat array into a multilevel array. Similarly, .clump on a pattern gets *n* values from the pattern at once and returns all of them as an array. *n* can be a number or a numeric pattern.
**flatten(levels):** The reverse operation: if a pattern returns an array, its values will be output one by one.

```
// A flat stream becomes an array of 4-item arrays
Pwhite(0, 255, 20).clump(4).do { |x| x.postln };

	// a two-dimensional array
Array.fill(5, { Array.fill(4, { rrand(1, 5) }) });

	// a pattern reading that array in sequence
p = Pseq(Array.fill(5, { Array.fill(4, { rrand(1, 5) }) }), 1);

	// the pattern returns several arrays
p.do { |x| x.postln };

	// flattening the pattern returns a one-dimensional stream of numbers
p.flatten.do { |x| x.postln };
```

**drop(n):** Discard the first n values, and return whatever is left.

```
Pseries(1, 1, 20).drop(5).do { |x| x.postln };
```

**differentiate:** Return the difference between successive values: second - first, third - second, and so on.

```
Array.geom(20, 1, 1.01).differentiate;
Pgeom(1, 1.01, 20).differentiate.do { |x| x.postln };
```

### Miscellaneous calculation patterns

These are some other numeric calculations that don't exactly fall in the category of math operators.

- **Pavaroh(pattern, aroh, avaroh, stepsPerOctave):** Convert scale degrees to note numbers, with separate ascending and descending scale patterns. Originally written for Indian ragas, it also works well for the western melodic minor scale.
- **PdegreeToKey(pattern, scale, stepsPerOctave):** Given a pattern yielding scale degrees, convert the degrees into note numbers according to the provided scale and steps per octave. This is done automatically when you use the 'degree' event key, but there might be cases where you would want to do some further math on the note numbers, and it might be necessary to make the conversion explicit.
- **Pdiff(pattern):** Returns the difference between the source stream's latest and previous values. Among other uses, this can measure whether a stream is ascending or descending. This is the underlying implementation of the 'differentiate' method discussed just above.
- **Prorate(proportion, pattern):** Splits up a number from 'pattern' according to proportion(s) given by the 'proportion' pattern. This is tricky to explain briefly; see the help file for some good examples.

```
// Swing notes with Prorate
(
p = Pbind(
	\degree, Pseries(4, Pwhite(-2, 2, inf).reject({ |x| x == 0 }), inf).fold(-7, 11),
	\dur, Prorate(0.6, 0.5)		// actually yields 0.3, 0.2, 0.3, 0.2...
```

).play;
)

p.stop;

**Calculations based on other event values**

In a Pbind, normally the patterns for the various keys calculate independently. But it's possible for one or more child patterns to depend on the result of another pattern inside the same Pbind. This is done with Pkey, described in PG_06g_Data_Sharing.

Previous: PG_04_Words_to_Phrases
Next: PG_060_Filter_Patterns

## Filter patterns

Just like filter UGens modify an input signal, filter patterns modify the stream of values coming from a pattern.

We have already seen some operations that modify a stream of values: math operators (which render as Punop, Pbinop or Pnaryop patterns) and certain collection methods (mainly collect, select and reject). Filter pattern classes can do some other surprising and useful things.

All filter patterns take at least one source pattern, providing the values/events to be filtered. Some filter patterns are designed for value patterns, others for event patterns. A handful work equally well with both single values and events.

Following is a categorized overview. See the separate category documents for more detail.

**Repetition and Constraint patterns**

- **Pclutch(pattern, connected):** If the 'connected' pattern is true, Pclutch returns the next value from 'pattern'. If 'connected' is false, the previous pattern value is repeated. It's like a clutch in a car: when engaged, the pattern moves forward; when disconnected, it stays where it is.
- **Pn(pattern, repeats):** Embeds the source pattern 'repeats' times: simple repetition. This also works on single values: Pn(1, 5) outputs the number 1 5 times.
- **Pstutter(n, pattern):** 'n' and 'pattern' are both patterns. Each value from 'pattern' is repeated 'n' times. If 'n' is a pattern, each value can be repeated a different number of times.
- **PdurStutter(n, pattern):** Like Pstutter, except the pattern value is divided by the number of repeats (so that the total time for the repeat cycle is the duration value from the source pattern).

- **Pfin(count, pattern):** Returns exactly 'count' values from the source pattern, then stops.
- **Pconst(sum, pattern, tolerance):** Output numbers until the sum reaches a predefined limit. The last output value is adjusted so that the sum matches the limit exactly.
- **Pfindur(dur, pattern, tolerance):** Like Pconst, but applying the "constrain" behavior to the event's rhythmic values. The source pattern runs up to the specified duration, then stops. This is very useful if you know how long a musical behavior should go on, but the number of events to fill up that time is not known.
- **Psync(pattern, quant, maxdur, tolerance):** Like Pfindur, but does not have a fixed duration limit. Instead, it plays until either it reaches maxdur (in which case it behaves like Pfindur, adjusting the last event so the total duration matches maxdur), or the pattern stops early and the last event is rounded up to the next integer multiple of quant.

**Time-based patterns**

- **Ptime(repeats):** Returns the amount of time elapsed since embedding.
- **Pstep(levels, durs, repeats):** Repeat a 'level' value for its corresponding duration, then move to the next.
- **Pseg(levels, durs, curves, repeats):** Similar to Pstep, but interpolates to the next value instead of stepping abruptly at the end of the duration. Interpolation is linear by default, but any envelope segment curve can be used. Levels, durs and curves should be patterns.
    - Related: Use of Env as a pattern.

**Adding values into event patterns**
**(Or, "Pattern Composition")**

- **Pbindf(pattern, pairs):** Adds new key-value pairs onto a pre-existing Pbind-style pattern.
- **Pchain(patterns):** Chains separate Pbind-style patterns together, so that all their key-value pairs go into the same event.

**Parallelizing event patterns**

- **Ppar(list, repeats):** Start each of the event patterns in the 'list' at the same time. When the last one finishes, the Ppar also stops. If repeats > 1, all the subpatterns start over again from the beginning.
- **Ptpar(list, repeats):** Here, the list consists of [timeOffset0, pattern0, timeOffset1, pattern1...]. Each pattern starts after the number of beats given as its time offset. The patterns can start at different times relative to each other.
- **Pgpar(list, repeats):** Like Ppar, but it creates a separate group for each subpattern.
- **Pgtpar(list, repeats):** This is supposed to be like Ptpar with separate groups for the sub patterns, but the class is currently broken.

- **Pspawner(routineFunc):** The function is used to make a routine. A Spawner object gets passed into this routine, and this object is used to add or remove streams to/from the parallel stream. New patterns can be added in sequence or in parallel.
- **Pspawn(pattern, spawnProtoEvent):** Supports most of the features of Pspawner, but uses a pattern to control the Spawner object instead of a routine function.

**Language control methods**

Some patterns mimic language-style control methods: conditionals (Pif), loops (Pwhile) and error cleanup (Pprotect).

- **Pif(condition, iftrue, iffalse, default):** Evaluates a pattern 'condition' that returns true or false. Then, one value is taken from the true or false branch before going back to evaluate the condition again. The 'default' value or pattern comes into play when the true or false branch stops producing values (returns nil). If the default is not given, Pif returns control to the parent upon nil from either branch.
- **Pseed(randSeed, pattern):** Random number generators depend on seed values; setting a specific seed produces a repeatable stream of pseudorandom numbers. Pseed sets the random seed before embedding 'pattern', effectively restarting the random number generator at the start of the pattern.
- **Pprotect(pattern, func):** Like the 'protect' error handling method, if an error occurs while getting the next value from the pattern, the function will be evaluated before the error interrupts execution.
- **Ptrace(pattern, key, printStream, prefix):** For debugging, Ptrace prints every return value. Is your pattern really doing what you think? This will tell you. A Ptrace is created automatically by the 'trace' message: aPattern.trace(key, printStream, prefix) --> Ptrace(aPattern, key, printStream, prefix).
- **Pwhile(func, pattern):** Like while: as long as the function evaluates to true, the pattern is embedded. The function is checked once at the beginning and thereafter when the pattern comes to an end. If it's applied to an infinite pattern, there's no looping because the pattern never gives control back.

**Server control methods**

- **Pbus(pattern, dur, fadeTime, numChannels, rate):** Creates a private group and bus for the synths played by the pattern. The group and bus are released when the pattern stops. Useful for isolating signals from different patterns.
- **Pgroup(pattern):** Creates a private group (without private bus) for the pattern's synths.

- **Pfx(pattern, fxname, pairs)**
- **Pfxb(pattern, fxname, pairs):** Both of these patterns play an effect synth at the tail of the target group. This synth should read from the bus identified by the 'out' argument, and write the processed signal onto the same bus using either ReplaceOut or XOut. Pfx uses whatever bus and group are specified in the incoming event. Pfxb allocates a separate bus and group for the effect and the pattern.

- **Pproto(makeFunction, pattern, cleanupFunc):** Allocate resources on the server and add references to them into the event prototype used to play 'pattern'. When the pattern stops (or is stopped), the resources can be removed automatically.

**Data sharing**

- **Pkey(key):** Read the 'key' in the input event, making previously-calculated values available for other streams.
- **Penvir(envir, pattern, independent):** Run the pattern inside a given environment.
- **Pfset(func, pattern):** Assign default values into the input event before getting each result event out of the given pattern.
- **Plambda(pattern, scope):** Creates a "function scope" into which values are assigned using Plet, and from which values are retrieved with Pget. Pget is somewhat like Pkey, except that its scope is strictly internal, hidden from the caller. With Pkey, the source values remain present in the event returned to the caller.

Previous: PG_05_Math_on_Patterns
Next: PG_06a_Repetition_Contraint_Patterns

## Repetition and Constraint patterns

These are essentially flow of control patterns. Each one takes a source pattern and repeats values from it, or stops the stream early based on a preset constraint.

**Repetition patterns**

These patterns allow you to repeat single values, or (in the case of Pn) entire patterns.

- **Pclutch(pattern, connected):** If the 'connected' pattern is true, Pclutch returns the next value from 'pattern'. If 'connected' is false, the previous pattern value is repeated. It's like a clutch in a car: when engaged, the pattern moves forward; when disconnected, it stays where it is.
- **Pn(pattern, repeats):** Embeds the source pattern 'repeats' times: simple repetition. This also works on single values: Pn(1, 5) outputs the number 1 5 times.
- **Pstutter(n, pattern):** 'n' and 'pattern' are both patterns. Each value from 'pattern' is repeated 'n' times. If 'n' is a pattern, each value can be repeated a different number of times.
- **PdurStutter(n, pattern):** Like Pstutter, except the pattern value is divided by the number of repeats (so that the total time for the repeat cycle is the duration value from the source pattern).

See also Pstep, described in PG_06b_Time_Based_Patterns. Pstep can be used like Pstutter, but repetition is controlled by time rather than number of repeats per item.

```
// play repeated notes with a different rhythmic value per new pitch
// using Pstutter
p = Pbind(
                // making 'n' a separate stream so that degree and dur can share it
        \n, Pwhite(3, 10, inf),
        \degree, Pstutter(Pkey(\n), Pwhite(-4, 11, inf)),
        \dur, Pstutter(Pkey(\n), Pwhite(0.1, 0.4, inf)),
        \legato, 0.3
).play;

p.stop;


// using Pfin / Pn
// Pn loops the Pbind infinitely
// Plazy builds a new Pbind for each iteration
// Pfin cuts off the Pbind when it's time for a new value

p = Pn(
        Plazy {
                Pbind(
                        \degree, Pfin(rrand(3, 10), rrand(-4, 11)),
                        \dur, rrand(0.1, 0.4)
                )
        },
        inf
).play;

// using Pclutch
// the rule is, when degree changes, dur should change also
// if Pdiff returns 0, degree has not changed
// so here, nonzero Pdiff values "connect" the clutch and allow a new dur to be generated
// otherwise the old one is held
p = Pbind(
        \degree, Pstutter(Pwhite(3, 10, inf), Pwhite(-4, 11, inf)),
        \dur, Pclutch(Pwhite(0.1, 0.4, inf), Pdiff(Pkey(\degree)).abs > 0),
        \legato, 0.3
).play;
```

**Constraint (or interruption) patterns**

Instead of prolonging a stream by repetition, these patterns use different methods to stop a stream dynamically. They are especially useful for modularizing pattern construction. One section of your code might be responsible for generating numeric or event patterns. By using constraint patterns, that part of the code doesn't have to know how many events or how long to play. It can just return an infinite pattern, and another part of the code can wrap it in one of these to stop it based on the appropriate condition.

- **Pfin(count, pattern):** Returns exactly 'count' values from the source pattern, then stops. (Pfin has a cousin, Pfinval, that is deprecated.)
- **Pconst(sum, pattern, tolerance):** Output numbers until the sum goes over a predefined limit. The last output value is adjusted so that the sum matches the limit exactly.
- **Pfindur(dur, pattern, tolerance):** Like Pconst, but applying the "constrain" behavior to the event's rhythmic values. The source pattern runs up to the specified duration, then stops. This is very useful if you know how long a musical behavior should go on, but the number of events to fill up that time is not known.

```
// Two variants on the same thing
// Use Pconst or Pfindur to create 4-beat segments with randomized rhythm
// Pconst and Pfindur both can ensure the total rhythm doesn't go above 4.0

p = Pn(Pbind(
                    // always a low C on the downbeat
        \degree, Pseq([-7, Pwhite(0, 11, inf)], 1),
        \dur, Pconst(4, Pwhite(1, 4, inf) * 0.25)
), inf).play;

p.stop;

p = Pn(Pfindur(4, Pbind(
        \degree, Pseq([-7, Pwhite(0, 11, inf)], 1),
        \dur, Pwhite(1, 4, inf) * 0.25
)), inf).play;

p.stop;
```

- **Psync(pattern, quant, maxdur, tolerance):** Like Pfindur, but does not have a fixed duration limit. Instead, it plays until either it reaches maxdur (in which case it behaves like Pfindur, adjusting the last event so the total duration matches maxdur), or the pattern stops early and the last event is rounded up to the next integer multiple of quant. This is hard to explain; a couple of examples might make it clearer.

```
(
// in this case, the pattern stops by reaching maxdur
// elapsed time = 4
var     startTime;
p = (Psync(Pbind(
        \dur, 0.25,         // total duration = infinite
        \time, Pfunc { startTime = startTime ? (thisThread.clock.beats.debug("time")) }
), 1, 4) ++ Pfuncn({
        thisThread.clock.beats.debug("finish time");
        (thisThread.clock.beats - startTime).debug("elapsed");
        nil
}, 1)).play;
)

(
// in this case, the pattern stops itself before maxdur (4)
// the Pbind's duration (1.25) gets rounded up to 2 (next multiple of 1)
var     startTime;
p = (Psync(Pbind(
        \dur, Pn(0.25, 5),  // total duration = 0.25 * 5 = 1.25
        \time, Pfunc { startTime = startTime ? (thisThread.clock.beats.debug("time")) }
), 1, 4) ++ Pfuncn({
        thisThread.clock.beats.debug("finish time");
        (thisThread.clock.beats - startTime).debug("elapsed");
        nil
}, 1)).play;
)
```

Previous: PG_060_Filter_Patterns
Next: PG_06b_Time_Based_Patterns

## Time-based patterns

"Time-based patterns" here are value patterns that use time as part of their calculation. Event patterns are naturally time-driven when played on a clock. (Technically it's possible to request events from an event stream without running it in an EventStreamPlayer, but this is not typical usage.)

Most of these patterns work by remembering the clock's current time at the moment the pattern is embedded into a value stream. The time value used for calculation is, then, the clock's time at the moment of evaluation minus the starting time -- that is, the number of beats elapsed since the patterns started embedding. If the pattern is embedded several times, the starting time is also reset so that the pattern begins again from the beginning.

There is nothing to prevent using these patterns outside of a scheduling context. In these documents, that context would be an event pattern played on a clock, but streams made from these patterns can be used in scheduled routines or functions as well. Only a scheduling context can ensure precise timing of requests for values.

- **Ptime(repeats):** Returns the amount of time elapsed since embedding. One nice trick with this pattern is to stop a value stream/pattern after a certain amount of time. This Pif pattern will use the true branch for exactly 4 beats after the first value is requested. After that, the condition will be false and Pif reverts to the false branch, which is nil. That causes the stream to stop. (This is like Pfindur for event patterns, but Pif/Ptime works for value patterns as well.)

```
// This is a really useful trick: like Pfindur but for value patterns
Pif(Ptime(inf) < 4.0, Pwhite(-4, 11, inf));

(
p = Pbind(
	\degree, Pif(Ptime(inf) < 4.0, Pwhite(-4, 11, inf)),
	\dur, 0.25
).play;
)
```

- **Pstep(levels, durs, repeats):** Repeat a 'level' value for its corresponding duration, then move to the next.
- **Pseg(levels, durs, curves, repeats):** Similar to Pstep, but interpolates to the next value instead of stepping abruptly at the end of the duration. Interpolation is linear by default, but any envelope segment curve can be used. Levels, durs and curves should be patterns.

```
// curve is 5 - here's what the curve looks like, ascending first then descending
Env(#[0, 1, 0], #[1, 1], 5).plot;

(
p = Pbind(
		// using \note b/c Pseg will give fractional note numbers
		// can't use \degree because it handles non-integers differently
	\note, Pseg(
		Pwhite(-7, 19, inf),		// chromatic note numbers
			// alternate version for diatonic numbers
			// PdegreeToKey does the same conversion as \degree --> \note
//		PdegreeToKey(Pwhite(-4, 11, inf), Pkey(\scale), 12),
		Pwhite(1, 4, inf) * 0.5,
		5, inf),
	\dur, 0.125
).play;
)

p.stop;
```

### Using envelopes as patterns

Env supports the stream protocol: 'asStream' turns an Env into a stream, and timed values can be obtained from it using 'next'. The envelope stream returns the value the envelope would have at the elapsed time, in the same way .at(time) returns the envelope value at the specified time.

```
e = Env.linen(1, 1, 1);
e.at(2);		// == 1
e.at(2.5);		// == 0.5
```

```
// print envelope values
r = fork {
	var	stream = e.asStream;
	12.do({
		stream.next.postln;
		0.25.wait;
	});
};

// Use an envelope to pan notes from left to right and back
p = Pbind(
	\degree, Pwhite(-4, 11, 32),
	\pan, Env(#[-1, 1, -1], #[2, 2], \sin),
	\dur, 0.125
).play;

p.stop;
```

The releaseNode and loopNode envelope parameters do not take effect, because they are meaningful only when used in a Synth with a gated EnvGen.

When the envelope ends, the stream will hold the final level indefinitely. The Pif(Ptime(inf) < totalTime, Env(...)) trick can make it stop instead.

```
// Use an envelope to pan notes from left to right and back
// Plays one cycle
(
p = Pbind(
		// change to inf: we don't need to know exactly how many events are needed
	\degree, Pwhite(-4, 11, inf),
	\pan, Pif(Ptime(inf) <= 4.0, Env(#[-1, 1, -1], #[2, 2], \sin)),
	\dur, 0.125
).play;
)

p.stop;

// To keep looping the envelope, wrap Pif inside Pn
(
p = Pbind(
	\degree, Pwhite(-4, 11, inf),
	\pan, Pn(Pif(Ptime(inf) <= 4.0, Env(#[-1, 1, -1], #[2, 2], \sin)), inf),
	\dur, 0.125
).play;
)

p.stop;
```

Previous: PG_06a_Repetition_Contraint_Patterns
Next: PG_06c_Composition_of_Patterns

## Adding values to a base event pattern
## (Or, “Pattern Composition”)

One way to use patterns is to write everything into the pattern up front. This has the advantage of clarity and ease of understanding. Another way is to modularize the behavior by creating smaller, simpler patterns and combining their results into single events that have keys and values from all the component patterns.

This is related to the computer science concept of "function composition," in which a complex calculation can be written not as a single large function, but as several smaller functions that are then chained together into a single function. Since Functions are normal objects in SuperCollider, it's easy to do an operation on a function that returns a composite function (which may then be used like any other function). http://en.wikipedia.org/wiki/Function_composition_(computer_science)

In mathematics, the · operator represents function composition.

f(x) = x + 1
g(x) = x * 2

g · f = g(f(x)) = (x + 1) * 2

g · f means to evaluate f first, then pass its result to g. The · operator is written as <> in SuperCollider.

```
f = { |x| x + 1 };
g = { |x| x * 2 };

h = (g <> f);
--> a Function

h.value(1);
--> 4  // == (1+1) * 2

(f <> g).value(1)
--> 3  // == (1*2) + 1

// g · f == g(f(x)) -- f is evaluated first, and its result is passed to g
g.value(f.value(1));
--> 4
```

Event patterns can be similarly composed.

- **Pbindf(pattern, pairs):** Adds new key-value pairs onto a pre-existing Pbind-style pattern. Pbindf(Pbind(\a, patternA), \b, patternB, \c, patternC) gets the same result as Pbind(\a, patternA, \b, patternB, \c, patternC).
- **Pchain(patterns):** Chains separate Pbind-style patterns together, so that all their key-value pairs go into the same event. For example, if one part of your code creates a Pbind instance a = Pbind(\a, patternA) and another part creates b = Pbind(\b, patternB, \c, patternC), you could append \b and \c into the \a result using Pchain(b, a). The subpatterns evaluate in reverse order, in keeping with function composition notation.

For musical purposes, you could have one part of your code create a pattern defining rhythm and another part defining pitch material, then combine them with Pchain.

```
~rhythm = Pbind(
        \dur, Pwrand(#[0.125, 0.25, 0.5], #[0.3, 0.5, 0.2], inf),
        \legato, Pwrand(#[0.1, 0.6, 1.01], #[0.1, 0.3, 0.6], inf)
);
~melody = Pbind(
        \degree, Pwhite(-4, 11, inf)
);

p = Pchain(~melody, ~rhythm).play;
p.stop;
```

That in itself has some good potential for algorithmic composition. Introducing EventPatternProxy into the mix makes it possible to swap different melody and rhythm components in and out on the fly, with no interruption. We can even change the type of pattern (Pbind, Pmono, PmonoArtic) with no ill effect.

```
~rhythm = EventPatternProxy(Pbind(
```

```
        \dur, Pwrand(#[0.125, 0.25, 0.5], #[0.3, 0.5, 0.2], inf),
        \legato, Pwrand(#[0.1, 0.6, 1.01], #[0.1, 0.3, 0.6], inf)
));

~melody = EventPatternProxy(Pbind(
        \degree, Pwhite(-4, 11, inf)
));

p = Pchain(~melody, ~rhythm).play;

~melody.source = PmonoArtic(\default, \degree, Pseries(4, Prand(#[-1, 1], inf), inf).fold(-4, 11));

~melody.source = Pbind(\degree, Pseries(4, Pwrand(#[-2, -1, 1, 2], #[0.3, 0.2, 0.2, 0.3], inf), inf).fold(-4, 11));

p.stop;
```

**Pset and cousins**

A group of pattern classes allow single event keys to be overwritten, or one addition or multiplication to be performed. Pkey, in combination with the Pchain or Pbindf "pattern composition" classes, can do everything the following classes can do (though this alternate notation may be more convenient in certain cases).

**Pset(name, value, pattern):** Get one event from 'pattern', and then put the next value from the 'value' pattern into the 'name' key. If the source pattern specifies a value for the same name, the value from the source will be replaced with the new one.
**Padd(name, value, pattern):** After getting the next event, replace the 'name' value with its existing value + the next number from 'value'.
**Pmul(name, value, pattern):** After getting the next event, replace the 'name' value with its existing value * the next number from 'value'.

These patterns remain in the library mainly for reasons of backward compatibility, since their behavior can be replicated easily using Pbindf.

**Pset**(name, value, pattern) == **Pbindf**(pattern, **name, value**)
**Padd**(name, value, pattern) == **Pbindf**(pattern, **name, Pkey(name) + value**)
**Pmul**(name, value, pattern) == **Pbindf**(pattern, name, **Pkey(name) * value**)

Psetpre, Paddpre, and Pmulpre reverse the order of evaluation. Pchain is able to duplicate this functionality.

**Psetpre(name, value, pattern):** Get the next 'value' and put it into the event prototype before evaluating 'pattern'.

**Psetpre**(name, value, pattern) == **Pchain**(pattern, **Pbind(name, value)**);
**Paddpre**(name, value, pattern) == **Pchain**(pattern, **Pbind(name, Pkey(name) + value)**);
Similar for Pmulpre

A third group -- Psetp, Paddp, Pmulp -- behave slightly differently, nesting pattern evaluation.

Previous: PG_06b_Time_Based_Patterns
Next: PG_06d_Parallel_Patterns

## Parallelizing event patterns

There are a couple of different ways to have several patterns playing at the same time. The most obvious is to play them separately. The patterns' events get scheduled independently on their clock(s) and run concurrently. None of these patterns need to have any knowledge of the others. One advantage of this approach is that the patterns can be stopped and started independently.

The other is to combine them into a parallel stream. The result is a single pattern object that can be played or stopped only as one unit. Some degree of interactive control is lost, but there are times when merging several patterns is necessary -- for instance, converting a pattern into a Score object for NRT rendering.

- **Ppar(list, repeats):** Start each of the event patterns in the 'list' at the same time. When the last one finishes, the Ppar also stops. If repeats > 1, all the subpatterns start over again from the beginning.
- **Ptpar(list, repeats):** Here, the list consists of [timeOffset0, pattern0, timeOffset1, pattern1...]. Each pattern starts after the number of beats given as its time offset. The patterns can start at different times relative to each other.
- **Pgpar(list, repeats):** Like Ppar, but it creates a separate group for each subpattern.
- **Pgtpar(list, repeats):** This is like Ptpar with separate groups for the subpatterns.

An excellent example of Ppar and Pseq used together to structure an entire piece (Kraftwerk's "Spacelab") can be found in examples/pieces/spacelab.scd.

**Dynamic parallelizing**

Ppar and its cousins are good for a fixed set of parallel patterns -- that is, you need to know in advance how many patterns will be parallelized. Once the parallel pattern starts, there is no way to add more streams to it. To keep adding streams, use Pspawner and Pspawn. For the purpose of this overview, some basic features will be illustrated in a couple of simple examples. These classes have more capabilities; refer to their help files for specifics.

- **Pspawner(routineFunc):** The function is run in a Routine. A Spawner object gets passed into this Routine, and this object is used to add or remove streams to/from the parallel stream. New patterns can be added in sequence or in parallel.
- **Pspawn(pattern, spawnProtoEvent):** Supports most of the features of Pspawner, but uses a pattern to control the Spawner object instead of a Routine function.

This example uses Pspawner to trigger overlapping scale segments at different speeds. Unlike Ppar, which could handle a fixed number before stopping, Pspawner can keep going indefinitely.

```
(
p = Pspawner({ |sp|          // sp = the Spawner object
        loop {
                        // run a new pattern in parallel
                        // the pattern is finite
                        // after a few events, it stops and the Pspawner forgets about it
                sp.par(Pbind(
                        \degree, Pseries(rrand(-5, 7), #[-1, 1].choose, rrand(4, 7)),
                        \pan, rrand(-1.0, 1.0),
                        \dur, rrand(0.1, 0.3)          // duration is chosen once for each pattern
                ));
                        // tell the Spawner to wait a bit before the next pattern goes
                        // DO NOT use numBeats.wait for this!
                        // Everything must go through the Spawner
                sp.wait(rrand(1, 4) * 0.25);
        }
}).play;
)

p.stop;
```

The same, written using Pspawn:

```
(
p = Pspawn(Pbind(
        \method, \par,              // embed patterns in parallel
                // generate the subpattern in a Pfunc (so there's a new pattern each time)
                // Pfunc returns the pattern without rendering the stream
```

```
		// -- important for Pspawn
		// See the Pspawn helpfile for other ways to embed patterns
	\pattern, Pfunc {
		Pbind(
			\degree, Pseries(rrand(-5, 7), #[-1, 1].choose, rrand(4, 7)),
			\pan, rrand(-1.0, 1.0),
			\dur, rrand(0.1, 0.3)		// duration is chosen once for each pattern
		)
	},
		// The \delta key is used automatically for the spawner.wait() call
	\delta, Pwhite(1, 4, inf) * 0.25
)).play;
)

p.stop;
```

Previous: PG_06c_Composition_of_Patterns
Next: PG_06e_Language_Control

## Language control methods

Some patterns mimic language-style control methods: conditionals (Pif), loops (Pwhile) and error cleanup (Pprotect).

- **Pif(condition, iftrue, iffalse, default):** Evaluates a pattern 'condition' that returns true or false. Then, one value is taken from the true or false branch before going back to evaluate the condition again. The 'default' value or pattern comes into play when the true or false branch stops producing values (returns nil). If the default is not given, Pif returns control to the parent upon nil from either branch.

```
p = Pbind(
	\degree, Pwhite(0, 11, inf),
		// odd numbered scale degrees get a shorter rhythmic value
	\dur, Pif(Pkey(\degree).odd, 0.25, 0.5)
).play;

p.stop;
```

- **Pseed(randSeed, pattern):** Random number generators depend on seed values; setting a specific seed produces a repeatable stream of pseudorandom numbers. Pseed sets the random seed before embedding 'pattern', effectively restarting the random number generator at the start of the pattern.

```
p = Pbind(
		// the random seed is generated once, when creating the Pattern object
		// so the same random seed is used every time whenever this pattern object plays
	\degree, Pseed(0x7FFFFFFF.rand, Pseries({ rrand(-7, 0) }, Pwhite(1, 3, inf), { rrand(4, 10) })),
	\dur, 0.25
);

q = p.play;  // uses one seed
q.stop;

r = p.play;  // uses the same seed
r.stop;

// reexecute the p = Pbind... and the seed will be different
```

- **Pprotect(pattern, func):** Like the 'protect' error handling method, if an error occurs while getting the next value from the pattern, the function will be evaluated before the error interrupts execution.
- **Ptrace(pattern, key, printStream, prefix):** For debugging, Ptrace prints every return value. Is your pattern really doing what you think? This will tell you. A Ptrace is created automatically by the 'trace' message: aPattern.trace(key, printStream, prefix) --> Ptrace(aPattern, key, printStream, prefix).
- **Pwhile(func, pattern):** Like while: as long as the function evaluates to true, the pattern is embedded. The function is checked once at the beginning and thereafter when the pattern comes to an end. If it's applied to an infinite pattern, there's no looping because the pattern never gives control back.

```
// Pwhile and Ptrace
(
~go = true;
p = Pwhile({ ~go }, Pbind(
	\degree, Pseries({ rrand(-7, 0) }, Pwhite(1, 3, inf), { rrand(4, 10) })
		.trace(prefix: "degree: "),
	\dur, 0.25
)).play;
)

~go = false;	// will stop the whole pattern when the Pbind comes to an end
```

Previous: PG_06d_Parallel_Patterns
Next: PG_06f_Server_Control

## Server control methods

A handful of filter patterns can isolate signals on a private bus and/or group, and also apply effect synths. A nice feature is that resources allocated at the beginning of the pattern are removed at the end. This is especially useful for effects, where you don't want to have a lot of effect synths left over taking up CPU but not processing audio.

- **Pbus(pattern, dur, fadeTime, numChannels, rate):** Creates a private group and bus for the synths played by the pattern. The group and bus are released when the pattern stops. Useful for isolating signals from different patterns.
- **Pgroup(pattern):** Creates a private group (without private bus) for the pattern's synths.

- **Pfx(pattern, fxname, pairs)**
- **Pfxb(pattern, fxname, pairs):** Both of these patterns play an effect synth at the tail of the target group. This synth should read from the bus identified by the 'out' argument, and write the processed signal onto the same bus using either ReplaceOut or XOut. Pfx uses whatever bus and group are specified in the incoming event. Pfxb allocates a separate bus and group for the effect and the pattern.

There are a lot of permutations when it comes to signal routing and effect management, too many to discuss in depth here. Some of the main scenarios are:

- Separate effects that should apply individually: the patterns and effects should be isolated on separate buses. Pfxb handles this isolation automatically: two patterns like Pfxb(Pbind(...), \fxname, \effectargName, value, \name, value...) will play on separate buses and their signals will not interfere with each other.

- Effects that should apply as a chain: both effects should use the same bus, and the effect patterns should be nested to string them together. The outermost effect should use Pfxb to allocate a separate group and bus for this signal chain; inner ones should use Pfx to piggyback on the existing bus.

```
Pfxb(
	Pfx(
		(event pattern here),
		\synthDefNameOfFirstEffectInChain,
		(argument list for the first effect),
	)
	\synthDefNameOfSecondEffectInChain,
	(argument list for the second effect)
).play;
```

More complex arrangements are possible through nesting, and parallelizing Pfx or Pfxb patterns using Ppar and its cousins.

This example uses Pfxb to isolate a pair of separately-sounding patterns on different buses, and to pass the two signals' streams through separate volume controls. The effect synth, for volume, is kept deliberately simple for the example, but of course it can do any kind of signal processing you like.

It might seem odd at first to use a gated envelope for an effect, but this is important to keep the signal's integrity. If the gate is not there, the effect synth will be n_free'd (brutally cut off), probably before the nodes played by the source pattern have finished. In this case it would produce a sudden, brief jump in volume at the end. The gate, combined with the one-second release in the envelope, keeps the effect synth around long enough to allow its source synths to become silent first.

Remember that streams made from patterns don't expose their internals. That means you can't adjust the parameters of an effect synth directly, because you have no way to find out what its node ID is. The example addresses this problem by allocating a couple of control buses for the amplitude values, and mapping the volume synths to those control buses. Then the little GUI needs only to update the control bus values.

```
// Demonstrates how Pfxb isolates signals on different buses
// The fx synth is a simple volume control here
// but it could be more complex

(
SynthDef(\volumeCtl, { |out, amp = 1, gate = 1|
	var	sig = In.ar(out, 2) * amp;
	sig = sig * EnvGen.kr(Env(#[1, 1, 0], #[1, 1], -3, releaseNode: 1), gate, doneAction:
2);
	ReplaceOut.ar(out, sig)
}).memStore;
```

```
~vbus1 = Bus.control(s, 1).set(0.5);
~vbus2 = Bus.control(s, 1).set(0.5);

~window = GUI.window.new("mixers", Rect(10, 100, 320, 60));
~window.view.decorator = FlowLayout(~window.view.bounds, 2@2);
EZSlider(~window, 310@20, "low part", \amp, { |ez| ~vbus1.set(ez.value) }, 0.5);
~window.view.decorator.nextLine;
EZSlider(~window, 310@20, "high part", \amp, { |ez| ~vbus2.set(ez.value) }, 0.5);
~window.front.onClose_({ ~vbus1.free; ~vbus2.free });
)

(
p = Ppar([
	Pfxb(Pbind(
		\degree, Pseq([0, 7, 4, 3, 9, 5, 1, 4], inf),
		\octave, 4,
		\dur, 0.5
	), \volumeCtl, \amp, ~vbus1.asMap),	// map to control bus here
	Pfxb(Pbind(
		\degree, Pwhite(0, 11, inf),
		\dur, 0.25
	), \volumeCtl, \amp, ~vbus2.asMap)	// ... and here
]).play;
)

p.stop;
```

**Third-party extension alert:** Pfx and its cousins work on the philosophy that a signal routing arrangement should be created as needed (when its subpattern is playing) and removed immediately when the pattern is finished. Another approach is to treat signal routing and effects as a persistent infrastructure, created and destroyed under the user's control (not the pattern's). JITLib's proxy system offers some support for this. MixerChannels (in the ddwMixerChannel quark) are a more explicit way. Any pattern can be played on a MixerChannel: **aMixer.play(aPattern)**.

**Pproto: Allocating other resources for the duration of a pattern**

It's also possible to load sound file or wavetable buffers or play synths as part of the preparation to run a Pbind-style pattern. When the Pbind stops, those resources would be removed automatically from the server.

The mechanism to do this is a bit unlike most of the other protocols to use the server in SuperCollider. To create the resources, Pproto takes a function in which one or more Event objects contain the instructions to create them. These events should use specific event types, described in Pproto's help file. The pattern is able to clean up the resources because each event has an associated cleanup action (see the event types with cleanup class). Thus, Pproto needs only to remember the events representing the resources, and execute their cleanup actions at the end.

The Pproto help file has several complex examples that are worth reading. Here is just one simple case that loads the standard a11wlk01.wav sound file and plays fragments from it.

```
(
SynthDef(\playbuf, { |bufnum, start, dur = 1, amp = 0.2, out|
	var	sig = PlayBuf.ar(1, bufnum, BufRateScale.ir(bufnum), 0, start);
	sig = sig * amp * EnvGen.kr(Env.linen(0.01, dur, 0.01), doneAction: 2);
	Out.ar(out, sig ! 2)
}).memStore;
)

(
TempoClock.default.tempo = 1;
p = Pproto({
	~buf = (type: \allocRead, path: "sounds/a11wlk01.wav").yield;
}, Pbind(
	\instrument, \playbuf,
		// access resources in the protoevent by Pkey
	\bufnum, Pkey(\buf),
	\dur, Pwhite(1, 4, inf) * 0.25,
		// upper bound of Pwhite is based on duration
```

```
            // so that start + (dur * samplerate) never goes past the buffer's end
        \start, Pwhite(0, 188893 - (Pkey(\dur) * 44100), inf)
)).play;
)

// shows a buffer number allocated ('true' ContiguousBlock)
s.bufferAllocator.debug;

p.stop;

s.bufferAllocator.debug;   // after stop, the buffer is gone
```

Previous: PG_06e_Language_Control
Next: PG_06g_Data_Sharing

## Sharing data between patterns

So far, we've seen patterns that are independent of each other. A single Pbind works on its own information, which is not available to other Pbinds. Also, for instance, the 'degree' pattern in a Pbind is not aware of what the 'dur' pattern is doing. Making these data available adds musical intelligence.

There are a couple of distinct ways to transmit information from one pattern into another. The first, simpler, technique is to read values from the current event that is already being processed. The second is to pass information from one event pattern into a separate event pattern. Since both are event patterns, they produce different result events and the first technique does not apply.

**Reading values from the current event**

Within a Pbind, value patterns can easily read from other values that have already been placed into the event. The **Pkey** pattern looks up the key in the event currently being processed and returns its value. From there, you can do any other pattern-friendly operation on it: filter patterns, math operators, etc.

**Pkey(key):** Read the 'key' in the input event. Outputs values until the input event doesn't contain the key (i.e., the value is nil). There is no 'repeats' argument. If you need to limit the number of values, wrap Pkey in Pfin.

```
p = Pkey(\a).asStream;

// The input value is an event with \a = 2, \b = 3; 'next' result is 2
p.next((a: 2, b: 3));

// We can do math on the input event too
p = (Pkey(\a) * Pkey(\b)).asStream;
p.next((a: 2, b: 3));          // returns 6 == 2 * 3
```

In this simple example, staccato vs. legato is calculated based on scale degree: lower notes are longer and higher notes are shorter. That only scratches the surface of this technique!

Be aware that Pkey can only look backward to keys stated earlier in the Pbind definition. Pbind processes the keys in the order given. In the example, it would not work to put 'legato' first and have it refer to 'degree' coming later, because the degree value is not available yet.

```
// something simple - the higher the note, the shorter the length
(
p = Pbind(
        \degree, Pseq([Pseries(-7, 1, 14), Pseries(7, -1, 14)], inf),
        \dur, 0.25,
                // \degree is EARLIER in the Pbind
        \legato, Pkey(\degree).linexp(-7, 7, 2.0, 0.05)
).play;
)

p.stop;
```

**Other information storage patterns**

These patterns represent three different strategies to persist information from one pattern and make it available to others.

**Penvir(envir, pattern, independent):** The Streams that evaluate patterns are usually Routines, and Routines have the special feature of remembering the Environment that was in force the last time it yielded, and restoring the same environment the next time it's awakened. Penvir establishes an environment in which 'pattern' will run. The environment can be initialized with values, or it could be empty at first and populated by elements of its pattern. The environment is separate from the event being processed (actually, the pattern could be either an event or value pattern). Access to the environment depends on function-driven patterns: Pfunc, Pfuncn, Proutine, .collect, .select, .reject, and similar.

The 'independent' flag specifies whether the environment will be kept separate for each stream made from the Penvir. If true (the default), whenever the Penvir is embedded in a stream, a new environment is created that inherits the initial values provided by 'envir'. If false, the same environment is used for every stream. In that case, the same environment could also be used in different Penvir patterns, and modifications of the environment by one Penvir would carry over to all the others -- hence its usefulness for sharing data.

**Pfset(func, pattern, cleanupFunc):** When embedded, Pfset creates an environment and populates it using environment variable assignments in the provided function. For every 'next' call, the values in the preset environment are inserted into the event prototype before evaluating the child pattern. This is one way to set defaults for the pattern. It could also be used to load objects on the server, although this takes some care because the object would be reloaded every time the Pfset is played and you are responsible for freeing objects created this way in the cleanupFunc. (Pproto is another way; see PG_06f_Server_Control.)

```
(
SynthDef(\playbuf, { |bufnum, start, dur = 1, amp = 0.2, out|
	var	sig = PlayBuf.ar(1, bufnum, BufRateScale.ir(bufnum), 0, start);
	sig = sig * amp * EnvGen.kr(Env.linen(0.01, dur, 0.01), doneAction: 2);
	Out.ar(out, sig ! 2)
}).memStore;
)

(
TempoClock.default.tempo = 1;
p = Pfset({
	~buf = Buffer.read(s, "sounds/a11wlk01.wav");
	0.2.yield;	// sync seems to be incompatible with patterns
	~bufFrames = ~buf.numFrames;
}, Pbind(
	\instrument, \playbuf,
		// access resources in the protoevent by Pkey
	\bufnum, Pkey(\buf),
	\dur, Pwhite(1, 4, inf) * 0.25,
		// upper bound of Pwhite is based on duration
		// so that start + (dur * samplerate) never goes past the buffer's end
	\start, Pwhite(0, Pkey(\bufFrames) - (Pkey(\dur) * 44100), inf)
), { defer(inEnvir { "freeing buffer".postln; ~buf.free }, 1.1) }).play;
)

// shows a buffer number allocated ('true' ContiguousBlock)
s.bufferAllocator.debug;

p.stop;
```

**Plambda(pattern, scope):** Maintains an 'eventScope' environment, that is attached to events while they're being processed. Values can be assigned into the event scope using Plet(key, pattern, return), and read from scope using Pget(key, default, repeats). Pget is somewhat similar to Pkey, but it has a 'repeats' argument controlling the number of return values as well as a 'default' that will be used if the given key is not found in the event scope.

A unique feature of Plambda/Plet/Pget is the ability for Plet to assign one value to the event scope and return another value to the main event simultaneously. Plet assigns the value from its 'pattern' into the event scope. The 'return' argument is optional; if provided, it gives the value to return back to Pbind.

Plambda removes the eventScope before returning the final event to the caller. You can see the scope by tracing the inner pattern.

```
p = Plambda(
	Pbind(
		\a, Plet(\z, Pseries(0, 1, inf), Pseries(100, -1, inf)),
		\b, Pget(\z, 0, inf) * 2
	).trace(key: \eventScope, prefix: "\nscope: ")
).asStream;

p.next(());
```

Something similar can be done with Pkey, by using intermediate values in the event that don't correspond to any SynthDef control names. There's no harm in having extra values in the event that its synth will not use; only the required ones are sent to the server. Often this is simpler than Plambda, but there might be cases where Plambda is the only way.

```
p = Pbind(
	\z, Pseries(0, 1, inf),
```

```
	\a, Pseries(100, -1, inf),
	\b, Pkey(\z) * 2
).asStream;

p.nextN(5, ()).do(_.postln);
```

**Communicating values between separate event patterns**

Passing values from one Pbind to another takes a couple of little tricks. First is to store completed events in an accessible location. Neither the Pattern nor the EventStreamPlayer save the finished events; but, calling 'collect' on the pattern attaches a custom action to perform on every result event. Here, we save the event into an environment variable, but it could go into the global library, a declared variable or any other data structure.

Second, we have to ensure that the source pattern is evaluated before any client patterns that depend on the source's value. The only way to do this is to schedule the source slightly earlier, because items scheduled at the same time on any clock can execute in any order. (There is no priority mechanism to make one thread always run first.) But, this scheduling requirement should not affect audio timing.

The solution is a timing offset mechanism, which delays the sound of an event by a given number of beats. In the example, the bass pattern is scheduled 0.1 beats before whole-numbered beats (while the chord pattern runs exactly on whole-numbered beats). The bass pattern operates with a timing offset of 0.1, delaying the sound so that it occurs on integer beats. Both patterns sound together in the server, even though their timing is different in the client.

| Beat | Client timing | Server timing |
|---|---|---|
| 0.9 | Bass event calculated | (bass event delayed by 0.1, nothing happens here) |
| 1.0 | Chord event calculated | Both bass and chord make sound |

```
(
TempoClock.default.tempo = 1;

~bass = Pbind(
	\degree, Pwhite(0, 7, inf),
	\octave, 3,  // down 2 octaves
	\dur, Pwhite(1, 4, inf),
	\legato, 1,
	\amp, 0.2
).collect({ |event|
	~lastBassEvent = event;
}).play(quant: Quant(quant: 1, timingOffset: 0.1));

// shorter form for the Quant object: #[1, 0, 0.1]

~chords = Pbind(
	\topNote, Pseries(7, Prand(#[-2, -1, 1, 2], inf), inf).fold(2, 14),
	\bassTriadNotes, Pfunc { ~lastBassEvent[\degree] } + #[0, 2, 4],
		// merge triad into topnote
		// raises triad notes to the highest octave lower than top note
		// div: is integer division, so x div: 7 * 7 means the next lower multiple of 7
	\merge, (Pkey(\topNote) - Pkey(\bassTriadNotes)) div: 7 * 7 + Pkey(\bassTriadNotes),
		// add topNote to the array if not already there
	\degree, Pfunc { |ev|
		if(ev[\merge].detect({ |item| item == ev[\topNote] }).isNil) {
			ev[\merge] ++ ev[\topNote]
		} {
			ev[\merge]
		}
	},
	\dur, Pwrand([Pseq([0.5, Pwhite(1, 3, 1), 0.5], 1), 1, 2, 3], #[1, 3, 2, 2].normalizeSum, inf),
	\amp, 0.05
).play(quant: 1);
)
```

```
~bass.stop;
~chords.stop;
```

The chord pattern demonstrates some of the ways higher-level logic can be expressed in patterns. The goal is to transpose the notes of the root position triad over the bass note by octave so that the notes all fall within the octave beneath a top note (chosen by stepwise motion). Pkey(\topNote) - Pkey(\bassTriadNotes) gives the number of transposition steps to bring the triad notes up to the top note; then the transposition steps are truncated to the next lower octave (x div: 7 is integer division producing an octave number; multiplying by 7 gives the number of scale degrees for that octave).

```
f = { |topNote, triad|
        var     x;
        x = (topNote - triad).debug("initial transposition steps");
        x = (x div: 7).debug("octaves to transpose");
        x = (x * 7).debug("steps to transpose");
        x + triad
};

f.value(7, #[0, 2, 4]);
--> [ 7, 2, 4 ] (first inversion triad)
```

Then the transposed array is checked to see if the top note is already a member. If not, it's added so that the melody will always be present.

Note that lazy operations on patterns define most of this behavior; only the conditional array check had to be written as a function.

The above example breaks one of the design principles of patterns. Ideally, it should be possible to play a single pattern object many times simultaneously without the different streams interfering with each other. Saving the bass note in one environment variable means that concurrent streams would not work together because they can't both use the same environment variable at the same time. The above approach does, however, allow the two patterns to be stopped and started independently, and new bass-dependent patterns to be added at any time. In some musical scenarios, this kind of flexibility is more important than respecting the pattern design ideal.

It is possible, using Ptpar and Penvir, to create independent environments for event storage as part of the pattern itself. By default, Penvir creates a new copy of its environment for each stream, guaranteeing independence. While the pattern is running, ~lastBassEvent = event saves the event in the stream's copy of the storage environment, and it's available to both Pbinds because both are under control of Penvir (indirectly through Ptpar).

```
(
p = Penvir((), Ptpar([
        0.0, Pbind(
                \degree, Pwhite(0, 7, inf),
                \octave, 3,  // down 2 octaves
                \dur, Pwhite(1, 4, inf),
                \legato, 1,
                \amp, 0.2,
                \timingOffset, 0.1
        ).collect({ |event|
                ~lastBassEvent = event;
        }),
        0.1, Pbind(
                \topNote, Pseries(7, Prand(#[-2, -1, 1, 2], inf), inf).fold(2, 14),
                \bassTriadNotes, Pfunc { ~lastBassEvent[\degree] } + #[0, 2, 4],
                \merge, (Pkey(\topNote) - Pkey(\bassTriadNotes)) div: 7 * 7 +
Pkey(\bassTriadNotes),
                \degree, Pfunc { |ev|
                        if(ev[\merge].detect({ |item| item == ev[\topNote] }).isNil) {
                                ev[\merge] ++ ev[\topNote]
                        } {
                                ev[\merge]
                        }
                },
                \dur, Pwrand([Pseq([0.5, Pwhite(1, 3, 1), 0.5], 1), 1, 2, 3], #[1, 3, 2, 2].normalizeSum, inf),
                \amp, 0.05
        )
```

```
])).play;
)

p.stop;
```

Previous: PG_06f_Server_Control
Next: PG_07_Value_Conversions

## Pitch and rhythm conversions in the default event

Using the default event prototype, pitch and rhythm can be specified in Pbind at different levels depending on the musical requirement. The default event prototype includes logic to convert higher-level abstractions into the physical parameters that are useful for synthesis.

The descriptions below start with the ending value that will actually be used, following up with the other values that are used in the calculations: e.g., \delta is based on \dur and \stretch. The calculations may be bypassed by providing another value for the calculated item. If your pattern specifies \delta directly, \dur and \stretch are ignored.

Note also that there is no obligation to use these constructs. The default event prototype is not meant to enforce one model of pitch or rhythm over any other; it simply provides these options, which you may use if they suit the task, or ignore or override if your task calls for something else entirely.

**Timing conversions**

Rhythm is based on \delta and \sustain event keys. Both of these can be calculated from higher-level abstractions: \dur, \stretch and \legato.

**delta:** The number of beats until the next event. You can give the delta pattern directly, or the default event prototype can calculate it for you based on other values:

**dur:** Duration of this event.
**stretch:** A multiplier for duration: *delta = dur * stretch*.

**sustain:** How many beats to hold this note. After \sustain beats, a release message will be sent to the synth node setting its 'gate' control to 0. Your SynthDef should use 'gate' in an EnvGen based on a sustaining envelope (see Env), and the EnvGen should have a doneAction (UGen-doneActions) that releases the synth at the end. You can give the sustain pattern directly, or the default event prototype can calculate it for you based on:

**legato:** A fraction of the event's duration for which the synth should sustain. 1.0 means this synth will release exactly at the onset of the next; 0.5 means the last half of the duration will be a rest. Values greater than 1.0 produce overlapping notes. *sustain = dur * legato * stretch*.

**Pitch conversions**

Pitch handling in the default event is rich, with a large number of options. To use events, it is not necessary to understand all of those options. As the examples have shown, a note-playing pattern produces sensible results even specifying only 'degree'. The other parameters allow you to control how the event gets from \degree to the frequency that is finally passed to the new synth. The default event prototype includes reasonable defaults for all of these.

To go from the highest level of abstraction down:

- **\degree** represents a scale degree. Fractional scale degrees support accidentals: adding 0.1 to an integer scale degree raises the corresponding chromatic note number by a semitone, and subtracting 0.1 lowers the chromatic note number. 0.2 raises or lowers by two semitones, and so on.
- **\note** is a chromatic note index, calculated from \degree based on a **\scale** and modal transposition (**\mtranspose**, scale degrees to raise or lower the note). **\note** is in equal-tempered units of any number of steps to the octave (**\stepsPerOctave**).
- **\midinote** is a 12ET conversion of **\note**, transposed into the right **\octave** and applying gamut transposition (**\gtranspose**, given in stepsPerOctave units). If \stepsPerOctave is anything other than 12, the non-12ET units are scaled into 12 \midinote units per octave.
- **\freq** is calculated from \midinote by 'midicps'. A chromatic transposition in 12ET units (**\ctranspose**) is added.

Most note-playing SynthDefs use 'freq' as an argument. If desired, they may use 'midinote', 'note' or even 'degree'.

To simplify into rules of thumb:

- If your material is organized around scales or modes, use **\degree**.
  - If the scale has different ascending and descending patterns, use **\note** in your Pbind, with the filter pattern Pavaroh.
- If your material is organized around equal divisions of the octave (not necessarily 12 divisions), use **\note** (and **\stepsPerOctave** for equal temperament other than 12 notes).
- If your material is organized around MIDI note numbers (or 12-tone equal temperament), **\midinote** will also work.
- If you prefer to give frequencies directly in Hz, use **\freq**.

Following is a complete description of all elements of the pitch system. Feel free to use the ones that are of interest, and ignore the rest.

**freq:** Frequency in Hz. May be given directly, or calculated based on the following. Pitch may be expressed at any one of several levels. Only one need be used at a time. For instance, if you write pitch in terms of scale degrees, the note, MIDI note and frequency values are calculated automatically for you.
**ctranspose:** Chromatic transposition, in 12ET units. Added to midinote.
**midinote:** MIDI note number; 12 MIDI notes = one octave. This may be fractional if needed. Calculated based on:
**root:** The scale root, given in 12ET MIDI note increments.
**octave:** The octave number for 'note' = 0. The default is 5, mapping note 0 onto MIDI note 60.
**stepsPerOctave:** How many 'note' units map onto the octave. Supports non-12ET temperaments.
**gtranspose:** Non-12ET transposition, in 'note' units. Added to note.
**note:** The note number, in any division of the octave. 0 is the scale root. Calculated based on:
**degree:** Scale degree.
**scale:** Mapping of scale degrees onto semitones. Major, for instance, is [0, 2, 4, 5, 7, 9, 11].
**stepsPerOctave:** (Same as above.)
**mtranspose:** Modal transposition; added to degree.

See also the Scale class for a repository of scale configurations, and the possibility of non-ET tuning.

```
(
// approximate a major scale with a 19TET chromatic scale
p = Pbind(
	\scale, #[0, 3, 6, 8, 11, 14, 17],
	\stepsPerOctave, 19,
	\degree, Pwhite(0, 7, inf),
	\dur, 0.125,
	\legato, Pexprand(0.2, 6.0, inf)
).play;
)

p.stop;
```

**Amplitude conversion**

Finally, you can specify amplitude as **\db** or **\amp**. If it's given as \db, the amplitude will be calculated automatically using .dbamp.

Previous: PG_06g_Data_Sharing
Next: PG_Cookbook01_Basic_Sequencing

## Cookbook: Sequencing basics

### Playing a predefined note sequence

The following are three different ways of playing the same famous fugue subject.

The first is brute force, listing all the scale degrees mechanically in order. The second and third recognize the A pedal point and use interlacing operations to insert the pedal notes in between the changing notes.

The example demonstrates the use of the 'scale' and 'root' event keys to define tonality. Root = 2 is D, and the scale defines a natural minor mode. The degree sequence also uses accidentals. Subtracting 0.1 from an integer scale degree flattens the note by a semitone; adding 0.1 raises by a semitone. -0.9 is 0.1 higher than -1; a natural minor scale degree below the tonic is a flat 7, and a half step higher than that is the leading tone.

```
(
TempoClock.default.tempo = 84/60;

p = Pbind(
        \scale, #[0, 2, 3, 5, 7, 8, 10],
        \root, 2,
        \degree, Pseq(#[rest, 4, 3, 4, 2, 4, 1, 4, 0, 4, -0.9, 4, 0, 4, 1, 4, 2, 4,
                -3, 4, -1.9, 4, -0.9, 4, 0, 4, -0.9, 4, 0, 4, 1, 4, 2], 1),
        \dur, 0.25
).play;
)

(
p = Pbind(
        \scale, #[0, 2, 3, 5, 7, 8, 10],
        \root, 2,
        \degree, Place([#[rest, 3, 2, 1, 0, -0.9, 0, 1, 2, -3, -1.9, -0.9, 0, -0.9, 0, 1, 2],
                (4 ! 16) ++ \rest], 17),
        \dur, 0.25
).play;
)

(
p = Pbind(
        \scale, #[0, 2, 3, 5, 7, 8, 10],
        \root, 2,
        \degree, Ppatlace([Pseq(#[rest, 3, 2, 1, 0, -0.9, 0, 1, 2, -3, -1.9, -0.9, 0, -0.9, 0, 1, 2], 1),
                Pn(4, 16)], inf),
        \dur, 0.25
).play;
)
```

### “Multichannel” expansion

In a SynthDef, using an array as the input to a UGen expands the UGen into an array of UGens (see MultiChannel). Something similar happens in patterns. Normally a value sent to a Synth node should be a single number, but if it's an array instead, the pattern expands the event to produce *multiple synth nodes* instead of just one.

The 'degree' pattern applies a set of chord intervals to a melody that's always on top. It's a compound pattern, Pseries(...) + Prand(...), where Pseries returns a single number and Prand returns an array. As with regular math operations, a number plus an array is an array. If the current Pseries value is 7 and Prand returns [0, -3, -5], the result is [7, 4, 2] and you would hear a C major triad in first inversion.

```
(
p = Pbind(
	\degree, Pseries(7, Pwhite(1, 3, inf) * Prand(#[-1, 1], inf), inf).fold(0, 14)
		+ Prand(#[[0, -2, -4], [0, -3, -5], [0, -2, -5], [0, -1, -4]], inf),
	\dur, Pwrand(#[1, 0.5], #[0.8, 0.2], inf)
).play;
)

p.stop;
```

**Using custom SynthDefs (including unpitched SynthDefs)**

Patterns have special features to support several styles of pitch organization, but those features are strictly optional. Here we play a SynthDef that has no frequency argument whatsoever.

Note the use of **memStore** to prepare the SynthDef. Without it, most of the SynthDef inputs would not be recognized and the pattern would not send values to them.

It's worth noting that the pattern runs in beats, whose real duration in seconds depends on the clock's tempo. The SynthDef, however, always measures time in seconds. This example keeps things simple by setting the clock to 1 beat per second. If the tempo needs to be something else, though, the 'time' key should be divided by the tempo:

```
	\time, Pkey(\delta) / Pfunc { thisThread.clock.tempo },
```

```
(
b = Buffer.read(s, "sounds/a11wlk01.wav");

SynthDef(\stretchedFragments, { |out, bufnum, start, time = 1, stretch = 1, amp = 1,
attack = 0.01, decay = 0.05|
	var	sig = PlayBuf.ar(1, bufnum, rate: stretch.reciprocal, startPos: start, doneAction:2);
	sig = PitchShift.ar(sig, pitchRatio: stretch)
		* EnvGen.kr(Env.linen(attack, time, decay), doneAction: 2);
	Out.ar(out, sig ! 2)
}).memStore;	// note memStore! Without this, arguments won't work
)

(
TempoClock.default.tempo = 1;

p = Pbind(
	\instrument, \stretchedFragments,
	\bufnum, b,
	\start, Pwhite(0, (b.numFrames * 0.7).asInteger, inf),
	\delta, Pexprand(0.2, 1.5, inf),
	\time, Pkey(\delta),
	\stretch, Pexprand(1.0, 4.0, inf),
	\amp, 0.5,
	\attack, 0.1,
	\decay, 0.2
).play;
)

p.stop;
b.free;	// be tidy! remember to clean up your Buffer
```

Previous: PG_07_Value_Conversions
Next: PG_Cookbook02_Manipulating_Patterns

## Manipulating pattern data

### Merging (interleaving) independent streams

Suppose you wanted a pattern that generated pitches in a lower range 70% of the time, and a higher range the other 30%. For purely random patterns, this is simple because the pattern for each range has no memory (the next value does not depend on the previous value in any perceptible way). The random number generator patterns (Pwhite) return one element before yielding control back to the "selector" (Pwrand).

```
\degree, Pwrand([Pwhite(-7, 11, 1), Pwhite(7, 18, 1)], #[0.7, 0.3], inf)
```

This does not work if the ranges need to keep their own integrity. For that, Pnsym1 is ideal. We create a dictionary with named patterns, each of which maintain their own streams. Then we choose randomly between their names, picking one value from whichever stream is chosen this cycle.

This use of Pseries is essentially a random walk among scale degrees. It has more linear continuity than the equal distribution generated by Pwhite. Even though the higher range interrupts from time to time, the continuity should still be audible.

```
(
var	melodies = (
		lowMelody: Pseries(4, Prand(#[-2, -1, 1, 2], inf), inf).fold(-7, 11),
		highMelody: Pseries(14, Prand(#[-3, -2, 2, 3], inf), inf).fold(7, 18)
	);

p = Pbind(
	\degree, Pnsym1(Pwrand(#[lowMelody, highMelody], [0.7, 0.3], inf), melodies),
	\dur, Pwrand(#[0.25, 0.5], #[0.4, 0.6], inf)
).play;
)

p.stop;
```

### Reading an array forward or backward arbitrarily

Here's an interesting one. We have an array of possible output values, and we want the pattern to move forward or backward through the array depending on some kind of user input.

There is actually a pattern that handles this already, based on the standard programming concept of a (random) walk. In a random walk, there is an "observer" who is at a position within an array. The observer moves randomly by some number of steps forward or backward. In the SuperCollider pattern implementation, Pwalk, the steps don't have to be random. So here, we determine the step size from a slider.

In general, GUI objects should not be used for data storage. The approach here is to save the step size into a variable, and then refer to that variable in the Pwalk pattern.

```
(
var	pitches = (0..14),	// replace with other pitches you want
	move = 0,
	window, slider;

window = GUI.window.new("Mouse Transport", Rect(5, 100, 500, 50));
slider = GUI.slider.new(window, Rect(5, 5, 490, 20))
	.action_({ |view|
		move = (view.value * 4 - 2).round;
	})
	.value_(0.5);
window.front;
```

```
p = Pbind(
            // Pfunc is the direction to move through the array
            // it could be anything
            //   - could read from MIDI or HID and convert it into a step
            //   - could be a GUI control, as it is here
        \degree, Pwalk(pitches, Pfunc { move }, 1, 7),
        \dur, 0.25
).play;
)

p.stop;
```

**Third-party extension alert:** The Pwalk pattern shown here moves forward and backward in a preset array. To do the same thing with the results of a pattern, see **Pscratch** in the ddwPatterns quark. An especially fun use of Pscratch is to use it on an event pattern like Pbind, skipping around in a series of fully realized note events.

**Changing Pbind value patterns on the fly**

Patterns are converted into streams to generate values (or events). By design, there is no way to access the internal state of the stream. This means, for Pbind and similar patterns, the streams producing values for the event keys are invisible. So, it isn't possible to reach inside the stream and change them while the pattern is playing.

What we can do instead is base the Pbind on **pattern proxies** -- objects that take the place of a pattern. The PatternProxy is a single object that creates a single stream within Pbind, but it looks for its values to the pattern and stream contained inside the proxy. Changing the proxy's pattern replaces the stream, without having to touch the Pbind's closed box.

In the first example, pattern proxies are held in environment variables, and they can be manipulated through those variables.

```
(
~degree = PatternProxy(Pn(Pseries(0, 1, 8), inf));
~dur = PatternProxy(Pn(0.25, inf));

p = Pbind(
        \degree, ~degree,
        \dur, ~dur
).play;
)

~degree.source = (Pexprand(1, 8, inf) - 1).round;

~dur.source = Pwrand(#[0.25, 0.5, 0.75], #[0.5, 0.3, 0.2], inf);

p.stop;
```

Another way is to use Pdefn, which is a global namespace of proxies for value patterns. (Because of the different requirements for handling values and event patterns, there are two namespaces: Pdef for event patterns like Pbind, and Pdefn for value patterns such as \degree and \dur here.) Storage is all taken care of for you, no need for variables of your own.

```
(
Pdefn(\degree, Pn(Pseries(0, 1, 8), inf));
Pdefn(\dur, Pn(0.25, inf));

p = Pbind(
        \degree, Pdefn(\degree),
```

```
        \dur, Pdefn(\dur)
).play;
)

Pdefn(\degree, (Pexprand(1, 8, inf) - 1).round);

Pdefn(\dur, Pwrand(#[0.25, 0.5, 0.75], #[0.5, 0.3, 0.2], inf));

p.stop;
```

**Third-party extension alert:** The ddwChucklib quark defines a third way of doing this, using object prototyping (based on Environments) to create objects that encapsulate all the information needed to perform a musical behavior. Patterns stored in the prototype's variables are automatically available as pattern proxies to the object's pattern, making it easier to create complex, malleable "processes" which can be replicated as separate objects that don't interfere with each other. It's a step toward object-oriented modeling of musical behaviors without requiring hardcoded classes that are specific to one piece or another.

Previous: PG_Cookbook01_Basic_Sequencing
Next: PG_Cookbook03_External_Control

## Pattern control by external device

### Control of parameters by MIDI or HID

To get pattern values from a HID device, use the Phid pattern.

For MIDI, the best approach is to save an incoming value into a variable, and then use Pfunc to access the variable for each event.

```
(
~legato = 1;
c = CCResponder({ |src, chan, num, value|
	~legato = value.linlin(0, 127, 0.1, 2.5)
}, num: 1); // num: 1 means modwheel
)

(
p = Pbind(
	\degree, Pwhite(-7, 12, inf),
	\dur, Pwrand([0.25, Pn(0.125, 2)], #[0.8, 0.2], inf),
	\legato, Pfunc { ~legato }    // retrieves value set by MIDI control
).play;
)

p.stop;
c.remove;
```

If Pfunc { } is bothersome in the Pbind, a PatternProxy or Pdefn could also serve the purpose.

```
(
~legato = PatternProxy(1);
c = CCResponder({ |src, chan, num, value|
	~legato.source = value.linlin(0, 127, 0.1, 2.5)
}, num: 1);
)

(
p = Pbind(
	\degree, Pwhite(-7, 12, inf),
	\dur, Pwrand([0.25, Pn(0.125, 2)], #[0.8, 0.2], inf),
	\legato, ~legato
).play;
)

p.stop;
c.remove;
```

### Triggering patterns by external control

Issuing 'play' to a pattern can occur in an action function for many different kinds of objects: GUI, MIDI, OSCresponder, HID actions. This allows triggering patterns from a variety of interfaces.

It's very unlikely that an action function would be triggered exactly in sync with a clock. If the pattern being played needs to run in time with other patterns, use the 'quant' argument to control its starting time (see Quant).

### Triggering a pattern by a GUI

```
(
```

```
var	pattern = Pbind(
		\degree, Pseries(7, Pwhite(1, 3, inf) * Prand(#[-1, 1], inf), inf).fold(0, 14)
			+ Prand(#[[0, -2, -4], [0, -3, -5], [0, -2, -5], [0, -1, -4]], inf),
		\dur, Pwrand(#[1, 0.5], #[0.8, 0.2], inf)
	),
	player, window;

window = GUI.window.new("Pattern trigger", Rect(5, 100, 150, 100))
		// onClose is fairly important
		// without it, closing the window could leave the pattern playing
	.onClose_({ player.stop });
GUI.button.new(window, Rect(5, 5, 140, 90))
	.states_([["Pattern GO"], ["Pattern STOP"]])
	.font_(GUI.font.new("Helvetica", 18))
	.action_({ |button|
		if(button.value == 1 and: { player.isNil or: { player.isPlaying.not } }) {
			player = pattern.play;
		} {
			player.stop;
			button.value = 0;
		};
	});
window.front;
)

p.stop;
```

**Triggering a pattern by MIDI**

```
(
var	pattern = Pbind(
		\degree, Pseries(7, Pwhite(1, 3, inf) * Prand(#[-1, 1], inf), inf).fold(0, 14)
			+ Prand(#[[0, -2, -4], [0, -3, -5], [0, -2, -5], [0, -1, -4]], inf),
		\dur, Pwrand(#[1, 0.5], #[0.8, 0.2], inf)
	),
	player;

~noteOnResp = NoteOnResponder({
	if(player.isNil or: { player.isPlaying.not }) {
		player = pattern.play;
	} {
		player.stop;
	};
// num: 60 limits this responder to listen to middle-C only
// but it will pick up that note from any port, any channel
}, num: 60);
)

// when done
~noteOnResp.remove;
```

**Triggering a pattern by signal amplitude**

Triggering a pattern based on audio amplitude is a bit trickier -- not because it's harder to play the pattern, but because identifying when the trigger should happen is more involved. The most straightforward way in SuperCollider is to use the Amplitude UGen to get the volume of the input signal and compare it to a threshold. Volume can fluctuate rapidly, so the 'releaseTime' argument of Amplitude is set to a high value. This makes the measured amplitude fall more slowly toward the baseline, preventing triggers from being sent too close together.

The actual threshold depends on the incoming signal, so the example pops up a quick and dirty window to see the measured amplitude and set the threshold and decay accordingly. The synth listens by default to the first hardware input bus, but you can change it the following in the code to use a different input bus:

```
inbus: s.options.numOutputBusChannels
```

In this configuration, the first trigger starts the pattern and the second trigger stops it. You might want the pattern to play while the input signal is above the threshold, and stop when the signal drops to a quieter level. The comparison amp >= thresh can send a trigger only when the signal goes from softer to lower, so if we want the pattern to stop when the signal becomes quiet, we need to send a trigger when crossing the threshold in both directions.

```
var	amp = Amplitude.kr(In.ar(inbus, 1), attackTime: 0.01, releaseTime: decay),
		trig = HPZ1.kr(amp >= thresh);
SendTrig.kr(trig.abs, 1, trig);
```

HPZ1 is positive if its input rises and negative if it falls. Triggering based on the absolute value, then, sends the trigger on any change. The client responding to the trigger might need to know the direction of change, so we send HPZ1's value back and the client can decide which action to take based on the sign of this value.

For this example, triggers are measured only when the signal rises above the threshold.

```
(
var	pattern = Pbind(
		\degree, Pseries(7, Pwhite(1, 3, inf) * Prand(#[-1, 1], inf), inf).fold(0, 14)
			+ Prand(#[[0, -2, -4], [0, -3, -5], [0, -2, -5], [0, -1, -4]], inf),
		\dur, Pwrand(#[1, 0.5], #[0.8, 0.2], inf)
	),
	player;

// Quicky GUI to tune threshold and decay times
~w = Window("threshold setting", Rect(15, 100, 300, 100))
	.onClose_({
		~ampSynth.free;
		~ampUpdater.remove;
		~oscTrigResp.remove;
		player.stop;
	});
~w.view.decorator = FlowLayout(~w.view.bounds, 2@2, 2@2);
~ampView = EZSlider(~w, 295@20, "amplitude", \amp, labelWidth: 80, numberWidth: 60);
~ampView.sliderView.canFocus_(false).enabled_(false);
~ampView.numberView.canFocus_(false).enabled_(false);
StaticText(~w, 295@5).background_(Color.gray);
~threshView = EZSlider(~w, 295@30, "threshold", \amp, action: { |ez|
	~ampSynth.set(\thresh, ez.value);
}, initVal: 0.4, labelWidth: 80, numberWidth: 60);
~decayView = EZSlider(~w, 295@30, "decay", #[0.1, 100, \exp], action: { |ez|
	~ampSynth.set(\decay, ez.value);
}, initVal: 80.0, labelWidth: 80, numberWidth: 60);

~w.front;

~ampSynth = SynthDef(\ampSynth, { |inbus, thresh = 0.8, decay = 1|
	var	amp = Amplitude.kr(In.ar(inbus, 1), attackTime: 0.01, releaseTime: decay);
		// this trigger (id==0) is to update the gui only
	SendTrig.kr(Impulse.kr(10), 0, amp);
		// this trigger gets sent only when amplitude crosses threshold
	SendTrig.kr(amp >= thresh, 1, 1);
}).play(args: [inbus: s.options.numOutputBusChannels, thresh: ~threshView.value, decay: ~decayView.value]);
```

```
~ampUpdater = OSCpathResponder(s.addr, ['/tr', ~ampSynth.nodeID, 0], { |time, resp, msg|
	defer { ~ampView.value = msg[3] }
}).add;

~oscTrigResp = OSCpathResponder(s.addr, ['/tr', ~ampSynth.nodeID, 1], { |time, resp, msg|
	if(player.isNil or: { player.isPlaying.not }) {
		player = pattern.play;
	} {
		player.stop;
	};
}).add;
)
```

Previous: PG_Cookbook02_Manipulating_Patterns
Next: PG_Cookbook04_Sending_MIDI

## Sending MIDI out with Patterns

Sending MIDI is the job of the MIDIOut class and the \midi event type. A MIDIOut is created to talk to a hardware device; the MIDI channel is provided when an event is played. MIDIOut's newByName makes it easier to identify a device.

The \midi event type supports the following commands, chosen in the event by the \midicmd key: \allNotesOff, \bend, \control, \noteOff, \noteOn, \polyTouch, \program, \smpte, \songPtr, \sysex, \touch. The default is \noteOn. When playing a note (noteOn), by default the corresponding noteOff message will be sent after the note's sustain time.

If you want to synchronize events played by a MIDI device and events played by the SuperCollider server, the MIDIOut object's latency must match the server latency. You can set the latency any time to affect all future events.

```
MIDIClient.init;  // if not already done

(
        // substitute your own device here
var     mOut = MIDIOut.newByName("FastLane USB", "Port A").latency_(Server.default.latency);

p = Pbind(
        \type, \midi,
                // this line is optional b/c noteOn is the default
                // just making it explicit here
        \midicmd, \noteOn,
        \midiout, mOut, // must provide the MIDI target here
        \chan, 0,
                // degree is converted to midinote, not just frequency
        \degree, Pwhite(-7, 12, inf),
        \dur, Pwrand([0.25, Pn(0.125, 2)], #[0.8, 0.2], inf),
        \legato, sin(Ptime(inf) * 0.5).linexp(-1, 1, 1/3, 3),
        \amp, Pexprand(0.5, 1.0, inf)
).play(quant: 1);
)

p.stop;
```

Previous:  PG_Cookbook03_External_Control
Next:  PG_Cookbook05_Using_Samples

## Using samples

### Playing a pattern in time with a sampled loop

A deceptively complex requirement... here, we will loop the a11wlk01.wav sample between 0.404561 and 3.185917 seconds (chosen for its surprisingly accurate four-beat groove), and overlay synthesized bells emphasizing the meter.

It might be tempting to loop a PlayBuf so that the loop runs automatically on the server, but it can easily drift out of sync with the client (because of slight deviations in the actual sample rate). Instead, the example defines a SynthDef that plays exactly one repetition of the loop, and repeatedly triggers it once per bar.

The primary bell pattern accents the downbeat and follows with a randomly generated rhythm. The catch is that we have no assurance that the Pwrand \dur pattern will add up to exactly 4 beats. The Pfindur ("finite duration") pattern cuts off the inner Pbind after 4 beats. This would stop the pattern, except Pn repeats the Pfindur infinitely, placing the accent in the right place every time.

The loop actually starts with a half-beat anacrusis, so Ptpar delays the bell patterns by 0.5 beats.

```
(
b = Buffer.read(s, "sounds/a11wlk01.wav");

// one loop segment
SynthDef(\oneLoop, { |out, bufnum, start, time, amp|
	var	sig = PlayBuf.ar(1, bufnum, startPos: start, loop: 0),
		env = EnvGen.kr(Env.linen(0.01, time, 0.05, level: amp), doneAction: 2);
	Out.ar(out, (sig * env) ! 2)
}).memStore;

SynthDef(\bell, { |out, accent = 0, amp = 0.1, decayScale = 1|
	var	exc = PinkNoise.ar(amp)
			* Decay2.kr(Impulse.kr(0), 0.01, 0.05),
		sig = Klank.ar(`[
			{ ExpRand(400, 1600) } ! 4,
			1 ! 4,
			{ ExpRand(0.1, 0.4) } ! 4
		], exc, freqscale: accent + 1, decayscale: decayScale);
	DetectSilence.ar(sig, doneAction: 2);
	Out.ar(out, sig ! 2)
}).memStore;
)

(
TempoClock.default.tempo = 0.35953685899971 * 4;

p = Ptpar([
	0, Pbind(
		\instrument, \oneLoop,
		\bufnum, b,
		\amp, 0.4,
		\start, 17841,
		\time, 0.35953685899971.reciprocal,
		\dur, 4
	),
	0.5, Pn(
		Pfindur(4,
			Pbind(
				\instrument, \bell,
				\accent, Pseq([2, Pn(0, inf)], 1),
				\amp, Pseq([0.3, Pn(0.1, inf)], 1),
```

```
                                \decayScale, Pseq([6, Pn(1, inf)], 1),
                                \dur, Pwrand(#[0.25, 0.5, 0.75, 1], #[2, 3, 1, 1].normalizeSum, inf)
                        )
                ),
        inf),
        0.5, Pbind(
                \instrument, \bell,
                \accent, -0.6,
                \amp, 0.2,
                \decayScale, 0.1,
                \dur, 1
        )
], 1).play;
)

p.stop;
```

The use of Ptpar above means that you could stop or start only the whole ball of wax at once, with no control over the three layers. It's no more difficult to play the layers in the independent event stream players, using the quant argument to ensure the proper synchronization. See the Quant help file for details on specifying the onset time of a pattern.

```
(
TempoClock.default.tempo = 0.35953685899971 * 4;

p = Pbind(
        \instrument, \oneLoop,
        \bufnum, b,
        \amp, 0.4,
        \start, 17841,
        \time, 0.35953685899971.reciprocal,
        \dur, 4
).play(quant: [4, 3.5]);

q = Pn(
        Pfindur(4,
                Pbind(
                        \instrument, \bell,
                        \accent, Pseq([2, Pn(0, inf)], 1),
                        \amp, Pseq([0.3, Pn(0.1, inf)], 1),
                        \decayScale, Pseq([6, Pn(1, inf)], 1),
                        \dur, Pwrand(#[0.25, 0.5, 0.75, 1], #[2, 3, 1, 1].normalizeSum, inf)
                )
        ),
inf).play(quant: [4, 4]);

r = Pbind(
        \instrument, \bell,
        \accent, -0.6,
        \amp, 0.2,
        \decayScale, 0.1,
        \dur, 1
).play(quant: [4, 4]);
)

[p, q, r].do(_.stop);

b.free;
```

**Using audio samples to play pitched material**

To use an instrument sample in a pattern, you need a SynthDef that plays the sample at a given rate. Here we will use PlayBuf, which doesn't allow looping over a specific region. For that, Phasor and BufRd are probably the best choice. (Third-party extension alert: LoopBuf by Lance Putnam is an alternative - click to download [osx]: http://www.uweb.ucsb.edu/~ljputnam/files/sc3/LoopBuf.zip.)

Frequency is controlled by the rate parameter. The sample plays at a given frequency at normal rate, so to play a specific frequency, **frequency / baseFrequency** gives you the required rate.

The first example makes a custom protoEvent that calculates rate, as \freq, based on the base frequency. It uses one sample, so it would be best for patterns that will play in a narrow range. Since there isn't an instrument sample in the SuperCollider distribution, we will record a frequency-modulation sample into a buffer before running the pattern.

```
// make a sound sample
(
var	recorder;
fork {
	b = Buffer.alloc(s, 44100 * 2, 1);
	s.sync;
	recorder = { |freq = 440|
		var	initPulse = Impulse.kr(0),
			mod = SinOsc.ar(freq) * Decay2.kr(initPulse, 0.01, 3) * 5,
			car = SinOsc.ar(freq + (mod*freq)) * Decay2.kr(initPulse, 0.01, 2.0);
		RecordBuf.ar(car, b, loop: 0, doneAction: 2);
		car ! 2
	}.play;
	OSCpathResponder(s.addr, ['/n_end', recorder.nodeID], { |time, resp, msg|
		"done recording".postln;
		resp.remove;
	}).add;
};
SynthDef(\sampler, { |out, bufnum, freq = 1, amp = 1|
	var	sig = PlayBuf.ar(1, bufnum, rate: freq, doneAction: 2) * amp;
	Out.ar(out, sig ! 2)
}).memStore;
)

(
// WAIT for "done recording" message before doing this
var	samplerEvent = Event.default.put(\freq, { ~midinote.midicps / ~sampleBaseFreq });

TempoClock.default.tempo = 1;
p = Pbind(
	\degree, Pwhite(0, 12, inf),
	\dur, Pwrand([0.25, Pn(0.125, 2)], #[0.8, 0.2], inf),
	\amp, Pexprand(0.1, 0.5, inf),
	\sampleBaseFreq, 440,
	\instrument, \sampler,
	\bufnum, b
).play(protoEvent: samplerEvent);
)

p.stop;
b.free;
```

**Multi-sampled instruments**

To extend the sampler's range using multiple samples and ensure smooth transitions between frequency ranges, the SynthDef should crossfade between adjacent buffers. A hybrid approach is used here, where Pbind calculates the lower buffer number to use and the SynthDef calculates the crossfade strength. (The calculations could be structured differently, either putting more of them into the SynthDef for convenience in the pattern, or loading them into the pattern and keeping the SynthDef as lean as possible.)

MIDI note numbers are used for these calculations because it's a linear frequency scale and linear interpolation is easier than the exponential interpolation that would be required when using Hz. Assuming a sorted array, indexInBetween gives the fractional index using linear interpolation. If you need to use frequency in Hz, use this function in place of indexInBetween.

```
	f = { |val, array|
		var a, b, div;
		var i = array.indexOfGreaterThan(val);
		if(i.isNil) { array.size - 1 } {
			if(i == 0) { i } {
				a = array[i-1]; b = array[i];
				div = b / a;
				if(div == 1) { i } {
					// log() / log() == log(val/a) at base (b/a)
					// which is the inverse of exponential
interpolation
					log(val / a) / log(div) + i - 1
				}
			}
		};
	};
```

But that function isn't needed for this example:

```
(
var	bufCount;
~midinotes = (39, 46 .. 88);
bufCount = ~midinotes.size;

fork {
		// record the samples at different frequencies
	b = Buffer.allocConsecutive(~midinotes.size, s, 44100 * 2, 1);
	SynthDef(\sampleSource, { |freq = 440, bufnum|
		var	initPulse = Impulse.kr(0),
			mod = SinOsc.ar(freq) * Decay2.kr(initPulse, 0.01, 3) * 5,
			car = SinOsc.ar(freq + (mod*freq)) * Decay2.kr(initPulse, 0.01, 2.0);
		RecordBuf.ar(car, bufnum, loop: 0, doneAction: 2);
	}).send(s);
	s.sync;
		// record all 8 buffers concurrently
	b.do({ |buf, i|
		Synth(\sampleSource, [freq: ~midinotes[i].midicps, bufnum: buf]);
	});
};
OSCresponderNode(s.addr, '/n_end', { |t, r, m|
	bufCount = bufCount - 1;
	if(bufCount == 0) {
		"done recording".postln;
		r.remove;
	};
}).add;

SynthDef(\multiSampler, { |out, bufnum, bufBase, baseFreqBuf, freq = 440, amp = 1|
	var	buf1 = bufnum.floor,
```

```
                buf2 = buf1 + 1,
                xfade = (bufnum - buf1).madd(2, -1),
                basefreqs = Index.kr(baseFreqBuf, [buf1, buf2]),
                playbufs = PlayBuf.ar(1, bufBase + [buf1, buf2], freq / basefreqs, loop: 0, doneAction: 2),
                sig = XFade2.ar(playbufs[0], playbufs[1], xfade, amp);
        Out.ar(out, sig ! 2)
}).memStore;

~baseBuf = Buffer.alloc(s, ~midinotes.size, 1, { |buf| buf.setnMsg(0, ~midinotes.midicps) });
)

(
TempoClock.default.tempo = 1;
p = Pbind(
        \instrument, \multiSampler,
        \bufBase, b.first,
        \baseFreqBuf, ~baseBuf,
        \degree, Pseries(0, Prand(#[-2, -1, 1, 2], inf), inf).fold(-11, 11),
        \dur, Pwrand([0.25, Pn(0.125, 2)], #[0.8, 0.2], inf),
        \amp, Pexprand(0.1, 0.5, inf),
                // some important conversions
                // identify the buffer numbers to read
        \freq, Pfunc { |ev| ev.use(ev[\freq]) },
        \bufnum, Pfunc({ |ev| ~midinotes.indexInBetween(ev[\freq].cpsmidi) })
                .clip(0, ~midinotes.size - 1.001)
).play;
)

p.stop;
b.do(_.free); ~baseBuf.free;
```

Previous: PG_Cookbook04_Sending_MIDI
Next: PG_Cookbook06_Phrase_Network

## Sequencing by a network of phrases
### Articulating notes with PmonoArtic

Two for one here!

Most conventional synthesizers have a mode where playing a note while the previous note is still sustaining slides from one note to the other. The PmonoArtic pattern does this based on the event's sustain value. The delta value is the number of beats until the next event; sustain is the number of beats until the note releases. If sustain is shorter than delta, the note should cut off early and the next event should produce a new synth.

The example uses Pfsm (finite state machine) to arrange a set of predefined phrases in a partially randomized order. Each phrase is followed by a list pointing to the phrases that could legitimately follow the current phrase. That is, it might make musical sense to go from phrase 1 to phrase 2, but not from 1 to 3. Defining the successors for 1 appropriately makes sure that a nonsense transition will not be made.

This is a long example, but it's only because there are lots of phrases. The structure is very simple: just a set of phrases chosen in succession by Pfsm.

**Third-party extension alert:** In this example, the selection of the next phrase is explicitly weighted by repeating array elements, such as #[1, 1, 1, 1, 2, 2, 3, 3, 4, 4, 5]. A more elegant way to do this is using the **WeighBag** class in the MathLib quark.

```
// the following are equivalent:
a = #[1, 1, 1, 1, 2, 2, 3, 3, 4, 4, 5];
({ a.choose } ! 100).histo(5, 1, 5);

a = WeighBag.with((1..5), #[4, 2, 2, 2, 1]);
({ a.choose } ! 100).histo(5, 1, 5);
```

**Example:**

```
(
// this SynthDef has a strong attack, emphasizing the articulation
SynthDef(\sawpulse, { |out, freq = 440, gate = 0.5, plfofreq = 6, mw = 0, ffreq =
2000, rq = 0.3, freqlag = 0.05, amp = 1|
        var sig, plfo, fcurve;
        plfo = SinOsc.kr(plfofreq, mul:mw, add:1);
        freq = Lag.kr(freq, freqlag) * plfo;
        fcurve = EnvGen.kr(Env.adsr(0, 0.3, 0.1, 20), gate);
        fcurve = (fcurve - 1).madd(0.7, 1) * ffreq;
        sig = Mix.ar([Pulse.ar(freq, 0.9), Saw.ar(freq*1.007)]);
        sig = RLPF.ar(sig, fcurve, rq)
                * EnvGen.kr(Env.adsr(0.04, 0.2, 0.6, 0.1), gate, doneAction:2)
                * amp;
        Out.ar(out, sig ! 2)
}).memStore;
)

(
TempoClock.default.tempo = 128/60;

// Pmul does only one thing here: take ~amp from each event
// and replace it with ~amp * 0.4
p = Pmul(\amp, 0.4, Pfsm([
        #[0, 3, 1],                 // starting places
        PmonoArtic(\sawpulse,
                \midinote, Pseq([78, 81, 78, 76, 78, 76, 72, 71, 69, 66], 1),
                \dur, Pseq(#[0.25, 1.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25], 1),
                \sustain, Pseq(#[0.3, 1.2, 0.3, 0.2, 0.3, 0.2, 0.3, 0.2, 0.3, 0.2],1 ),
```

```
        \amp, Pseq(#[1, 0.5, 0.75, 0.5, 0.75, 0.5, 0.75, 0.5, 0.75, 0.5], 1),
        \mw, Pseq([0, 0.03, Pseq(#[0], inf)], 1)
), #[1, 2, 3, 4, 7],

PmonoArtic(\sawpulse,
        \midinote, Pseq([64, 66, 69, 71, 72, 73], 1),
        \dur, Pseq(#[0.25], 6),
        \sustain, Pseq(#[0.3, 0.2, 0.2, 0.2, 0.3, 0.2], 1),
        \amp, Pseq(#[1, 0.5, 0.5, 0.5, 0.5, 0.5], 1),
        \mw, 0
), #[1, 1, 1, 1, 2, 2, 3, 3, 4, 4, 5],

PmonoArtic(\sawpulse,
        \midinote, Pseq([69, 71, 69, 66, 64, 69, 71, 69], 1),
        \dur, Pseq(#[0.125, 0.625, 0.25, 0.25, 0.25, 0.25, 0.25, 0.75], 1),
        \sustain, Pseq(#[0.2, 0.64, 0.2, 0.2, 0.2, 0.3, 0.3, 0.75], 1),
        \amp, Pseq(#[0.5, 0.75, 0.5, 0.5, 0.5, 1, 0.5, 0.5], 1),
        \mw, 0
), #[0, 1, 1, 1, 1, 3, 3, 3, 3, 5],

PmonoArtic(\sawpulse,
        \midinote, Pseq([72, 73, 76, 72, 71, 69, 66, 71, 69], 1),
        \dur, Pseq(#[0.25, 0.25, 0.25, 0.083, 0.083, 0.084, 0.25, 0.25, 0.25], 1),
        \sustain, Pseq(#[0.3, 0.2, 0.2, 0.1, 0.07, 0.07, 0.2, 0.3, 0.2], 1),
        \amp, Pseq(#[1, 0.5, 0.5, 1, 0.3, 0.3, 0.75, 0.75, 0.5], 1),
        \mw, 0
), #[1, 1, 1, 1, 3, 3, 4, 4, 4],

PmonoArtic(\sawpulse,
        \midinote, Pseq([64, 66, 69, 71, 72, 73, 71, 69, 66, 71, 69, 66, 64, 69], 1),
        \dur, Pseq(#[0.25, 0.25, 0.25, 0.25, 0.125, 0.375, 0.166, 0.166, 0.168,
                        0.5, 0.166, 0.166, 0.168, 0.5], 1),
        \sustain, Pseq(#[0.3, 0.2, 0.2, 0.2, 0.14, 0.4, 0.2, 0.2, 0.2, 0.6, 0.2, 0.2, 0.2, 0.5],1),
        \amp, Pseq(#[0.5, 0.5, 0.6, 0.8, 1, 0.5, 0.5, 0.5, 0.5, 1,
                0.5, 0.5, 0.5, 0.45], 1),
        \mw, 0
), #[0, 1, 1, 1, 1, 3, 3, 5],

PmonoArtic(\sawpulse,
        \midinote, Pseq([72, 73, 76, 78, 81, 78, 83, 81, 84, 85], 1),
        \dur, Pseq(#[0.25, 0.25, 0.25, 0.25, 0.5, 0.5, 0.5, 0.5, 0.125, 1.125], 1),
        \sustain, Pseq(#[0.3, 0.2, 0.2, 0.2, 0.95, 0.25, 0.95, 0.25, 0.2, 1.13], 1),
        \amp, Pseq(#[0.7, 0.5, 0.5, 0.5, 0.7, 0.5, 0.8, 0.5, 1, 0.5], 1),
        \mw, Pseq([Pseq(#[0], 9), 0.03], 1)
), #[6, 6, 6, 8, 9, 10, 10, 10, 10, 11, 11, 13, 13],

PmonoArtic(\sawpulse,
        \midinote, Pseq([83, 81, 78, 83, 81, 78, 76, 72, 73, 78, 72, 72, 71], 1),
        \dur, Pseq(#[0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25,
                        0.25, 2], 1),
        \sustain, Pseq(#[0.3, 0.3, 0.2, 0.3, 0.3, 0.3, 0.2, 0.3, 0.2, 0.3, 0.2, 0.3, 2], 1),
        \amp, Pseq(#[0.5, 0.5, 0.5, 0.8, 0.5, 0.5, 0.5, 0.8, 0.5, 0.8, 0.5,
                        1, 0.4], 1),
        \mw, Pseq([Pseq([0], 12), 0.03], 1)
), #[0, 7, 7, 7, 7, 7, 3, 3, 3, 3],

PmonoArtic(\sawpulse,
        \midinote, Pseq([69, 71, 72, 71, 69, 66, 64, 69, 71], 1),
        \dur, Pseq(#[0.25, 0.25, 0.25, 0.25, 0.166, 0.167, 0.167, 0.25, 0.25], 1),
        \sustain, Pseq(#[0.2, 0.2, 0.3, 0.2, 0.2, 0.2, 0.14, 0.3, 0.2], 1),
```

```
        \amp, Pseq(#[0.5, 0.5, 0.8, 0.5, 0.5, 0.5, 0.5, 0.8, 0.5], 1)
), #[3, 3, 3, 4, 4, 5],

PmonoArtic(\sawpulse,
        \midinote, Pseq([84, 85, 84, 84, 88, 84, 83, 81, 83, 81, 78, 76, 81, 83], 1),
        \dur, Pseq(#[0.125, 0.535, 0.67, 1.92, 0.25, 0.166, 0.167, 0.167,
                        0.25, 0.25, 0.25, 0.25, 0.25, 0.25], 1),
        \sustain, Pseq(#[0.2, 3.12, 0.2, 0.2, 0.2, 0.2, 0.2, 0.15, 0.3, 0.2, 0.2, 0.2,
                        0.3, 0.2], 1),
        \amp, Pseq(#[1, 0.8, 0.8, 0.8, 1, 1, 0.8, 0.8, 1, 0.8, 0.8, 0.8,
                        1, 0.8], 1),
        \mw, Pseq([0, 0.005, 0.005, 0.06, Pseq(#[0], 10)], 1)
), #[10, 10, 10, 11, 11, 11, 11, 12, 12, 12],

        // same as #4, 8va
PmonoArtic(\sawpulse,
        \midinote, Pseq(([64, 66, 69, 71, 72, 73, 71, 69, 66, 71, 69, 66, 64, 69]+12), 1),
        \dur, Pseq(#[0.25, 0.25, 0.25, 0.25, 0.125, 0.375, 0.166, 0.166, 0.168,
                        0.5, 0.166, 0.166, 0.168, 0.5], 1),
        \sustain, Pseq(#[0.3, 0.2, 0.2, 0.2, 0.14, 0.4, 0.2, 0.2, 0.2, 0.6, 0.2, 0.2, 0.2, 0.5],1),
        \amp, Pseq(#[0.5, 0.5, 0.6, 0.8, 1, 0.5, 0.5, 0.5, 0.5, 1,
                0.5, 0.5, 0.5, 0.45], 1),
        \mw, 0
), #[11, 11, 11, 11, 11, 12, 12],

PmonoArtic(\sawpulse,
        \midinote, Pseq([81, 84, 83, 81, 78, 76, 81, 83], 1),
        \dur, Pseq(#[0.25], 8),
        \sustain, Pseq(#[0.2, 0.3, 0.3, 0.2, 0.3, 0.2, 0.3, 0.2], 1),
        \amp, Pseq(#[0.5, 1, 0.5, 0.5, 0.6, 0.5, 0.8, 0.5], 1),
        \mw, 0
), #[0, 9, 9, 11, 11, 12, 12, 12, 12, 12],

        // same as #1, 8va
PmonoArtic(\sawpulse,
        \midinote, Pseq(([64, 66, 69, 71, 72, 73]+12), 1),
        \dur, Pseq(#[0.25], 6),
        \sustain, Pseq(#[0.3, 0.2, 0.2, 0.2, 0.3, 0.2], 1),
        \amp, Pseq(#[1, 0.5, 0.5, 0.5, 0.5, 0.5], 1),
        \mw, 0
), #[6, 6, 8, 9, 9, 9, 9, 10, 10, 10, 10, 13, 13, 13],

PmonoArtic(\sawpulse,
        \midinote, Pseq([78, 81, 83, 78, 83, 84, 78, 84, 85], 1),
        \dur, Pseq(#[0.25, 0.25, 0.5, 0.25, 0.25, 0.5, 0.25, 0.25, 1.75], 1),
        \sustain, Pseq(#[0.2, 0.3, 0.2, 0.2, 0.3, 0.2, 0.2, 0.3, 1.75], 1),
        \amp, Pseq(#[0.4, 0.8, 0.5, 0.4, 0.8, 0.5, 0.4, 1, 0.8], 1),
        \mw, Pseq([Pseq([0], 8), 0.03], 1)
), #[8, 13, 13],

PmonoArtic(\sawpulse,
        \midinote, Pseq([88, 84, 83, 81, 83, 81, 78, 76, 81, 83], 1),
        \dur, Pseq(#[0.25, 0.166, 0.167, 0.167,
                        0.25, 0.25, 0.25, 0.25, 0.25, 0.25], 1),
        \sustain, Pseq(#[0.2, 0.2, 0.2, 0.15, 0.3, 0.2, 0.2, 0.2,
                        0.3, 0.2], 1),
        \amp, Pseq(#[1, 1, 0.8, 0.8, 1, 0.8, 0.8, 0.8,
                        1, 0.8], 1),
        \mw, 0
), #[10]
```

```
], inf)).play;
)

p.stop;
```

Previous: PG_Cookbook05_Using_Samples
Next: PG_Cookbook07_Rhythmic_Variations

## Creating variations on a base rhythmic pattern

Normally patterns are stateless objects. This would seem to rule out the possibility of making on-the-fly changes to the material that pattern is playing. Indeed, modifying an existing pattern object is tricky and not always appropriate (because that approach cannot confine its changes to the one stream making the changes).

**Plazy** offers an alternate approach: use a function to generate a new pattern object periodically, and play these patterns in succession, one by one. (Plazy embeds just one pattern; wrapping Plazy in Pn does it many times.)

The logic in this example is a bit more involved: for each measure, start with arrays containing the basic rhythmic pattern for each part (kick drum, snare and hi-hat) and insert ornamental notes with different amplitudes and durations. Arrays hold the rhythmic data because this type of rhythm generation calls for awareness of the entire bar (future), whereas patterns generally don't look ahead.

This suggests an object for data storage that will also encapsulate the unique logic for each part. We saw earlier that Penvir maintains a distinct environment for each stream made from the pattern. In other words, Penvir allows more complicated behavior to be modeled using an object that encapsulates both custom logic and the data on which it will operate.

The specific ornaments to be added are slightly different for the three parts, so there are three environments. Some functions are shared; rather than copy and paste them into each environment, we put them into a separate environment and use that as the parent of the environment for each drum part.

Most of the logic is in the drum parts' environments, and consist mostly of straightforward array manipulations. Let's unpack the pattern that uses the environments to generate notes:

```
~kik = Penvir(~kikEnvir, Pn(Plazy({
	~init.value;
	~addNotes.value;
	Pbindf(
		Pbind(
			\instrument, \kik,
			\preamp, 0.4,
			\dur, 0.25,
			*(~pbindPairs.value(#[amp, decay2]))
		),
		\freq, Pif(Pkey(\amp) > 0, 1, \rest)
	)
}), inf)).play(quant: 4);
```

**Penvir(~kikEnvir, ...)** : Tell the enclosed pattern to run inside the kick drum's environment.

**Pn(..., inf)** : Repeat the enclosed pattern (Plazy) an infinite number of times.

**Plazy({ ... })** : The function can do anything it likes, as long as it returns some kind of pattern. The first two lines of the function do the hard work, especially ~addNotes.value, calling into the environment to use the rhythm generator code. This changes the data in the environment, which then get plugged into Pbind in the ~pbindPairs.value() line. That pattern will play through; when it ends, Plazy gives control back to its parent -- Pn, which repeats Plazy.

**Pbindf(..., \freq, ...)** : Pbindf adds new values into events coming from a different pattern. This usage is to take advantage of a fact about the default event. If the \freq key is a symbol (rather than a number or array), the event represents a rest and nothing will play on the server. It doesn't matter whether or not the SynthDef has a 'freq' control; a symbol in this space produces a rest. Here it's a simple conditional to produce a rest when amp == 0.

**Pbind(...)** : The meat of the notes: SynthDef name, general parameters, and rhythmic values from the environment. (The * syntax explains the need for Pbindf. The \freq expression must follow the pbindPairs result, but it isn't possible to put additional arguments after *(...). Pbindf allows the inner Pbind to be closed while still accepting additional values.)

**Third-party extension alert:** This type of hybrid between pattern-style flow of control and object-oriented modeling is powerful but has some limitations, mainly difficulty with inheritance (subclassing). The ddwChucklib quark (which depends on ddwPrototype) expands the object-oriented modeling possibilities while supporting patterns' ability to work with data external to a pattern itself.

```
(
// this kick drum doesn't sound so good on cheap speakers
// but if your monitors have decent bass, it's electro-licious
SynthDef(\kik, { |basefreq = 50, ratio = 7, sweeptime = 0.05, preamp = 1, amp = 1,
		decay1 = 0.3, decay1L = 0.8, decay2 = 0.15, out|
	var	fcurve = EnvGen.kr(Env([basefreq * ratio, basefreq], [sweeptime], \exp)),
		env = EnvGen.kr(Env([1, decay1L, 0], [decay1, decay2], -4), doneAction: 2),
		sig = SinOsc.ar(fcurve, 0.5pi, preamp).distort * env * amp;
	Out.ar(out, sig ! 2)
}).memStore;

SynthDef(\kraftySnr, { |amp = 1, freq = 2000, rq = 3, decay = 0.3, pan, out|
	var	sig = PinkNoise.ar(amp),
		env = EnvGen.kr(Env.perc(0.01, decay), doneAction: 2);
	sig = BPF.ar(sig, freq, rq, env);
	Out.ar(out, Pan2.ar(sig, pan))
}).memStore;

~commonFuncs = (
		// save starting time, to recognize the last bar of a 4-bar cycle
	init: {
		if(~startTime.isNil) { ~startTime = thisThread.clock.beats };
	},
		// convert the rhythm arrays into patterns
	pbindPairs: { |keys|
		var	pairs = Array(keys.size * 2);
		keys.do({ |key|
			if(key.envirGet.notNil) { pairs.add(key).add(Pseq(key.envirGet, 1)) };
		});
		pairs
	},
		// identify rests in the rhythm array
		// (to know where to stick notes in)
	getRestIndices: { |array|
		var	result = Array(array.size);
		array.do({ |item, i|
			if(item == 0) { result.add(i) }
		});
		result
	}
);
)

(
TempoClock.default.tempo = 104 / 60;

~kikEnvir = (
	parent: ~commonFuncs,
		// rhythm pattern that is constant in each bar
	baseAmp: #[1, 0, 0, 0,  0, 0, 0.7, 0,  0, 1, 0, 0,  0, 0, 0, 0] * 0.5,
	baseDecay: #[0.15, 0, 0, 0,  0, 0, 0.15, 0,  0, 0.15, 0, 0,  0, 0, 0, 0],
	addNotes: {
		var	beat16pos = (thisThread.clock.beats - ~startTime) % 16,
			available = ~getRestIndices.(~baseAmp);
```

```
		~amp = ~baseAmp.copy;
		~decay2 = ~baseDecay.copy;
			// if last bar of 4beat cycle, do busier fills
		if(beat16pos.inclusivelyBetween(12, 16)) {
			available.scramble[..rrand(5, 10)].do({ |index|
					// crescendo
				~amp[index] = index.linexp(0, 15, 0.2, 0.5);
				~decay2[index] = 0.15;
			});
		} {
			available.scramble[..rrand(0, 2)].do({ |index|
				~amp[index] = rrand(0.15, 0.3);
				~decay2[index] = rrand(0.05, 0.1);
			});
		}
	}
);

~snrEnvir = (
	parent: ~commonFuncs,
	baseAmp: #[0, 0, 0, 0,  1, 0, 0, 0,  0, 0, 0, 0,  1, 0, 0, 0] * 1.5,
	baseDecay: #[0, 0, 0, 0,  0.7, 0, 0, 0,  0, 0, 0, 0,  0.4, 0, 0, 0],
	addNotes: {
		var	beat16pos = (thisThread.clock.beats - ~startTime) % 16,
			available = ~getRestIndices.(~baseAmp),
			choice;
		~amp = ~baseAmp.copy;
		~decay = ~baseDecay.copy;
		if(beat16pos.inclusivelyBetween(12, 16)) {
			available.scramble[..rrand(5, 9)].do({ |index|
				~amp[index] = index.linexp(0, 15, 0.5, 1.8);
				~decay[index] = rrand(0.2, 0.4);
			});
		} {
			available.scramble[..rrand(1, 3)].do({ |index|
				~amp[index] = rrand(0.15, 0.3);
				~decay[index] = rrand(0.2, 0.4);
			});
		}
	}
);

~hhEnvir = (
	parent: ~commonFuncs,
	baseAmp: 15 ! 16,
	baseDelta: 0.25 ! 16,
	addNotes: {
		var	beat16pos = (thisThread.clock.beats - ~startTime) % 16,
			available = (0..15),
			toAdd;
			// if last bar of 4beat cycle, do busier fills
		~amp = ~baseAmp.copy;
		~dur = ~baseDelta.copy;
		if(beat16pos.inclusivelyBetween(12, 16)) {
			toAdd = available.scramble[..rrand(2, 5)]
		} {
			toAdd = available.scramble[..rrand(0, 1)]
		};
		toAdd.do({ |index|
			~amp[index] = ~doubleTimeAmps;
```

```
                        ~dur[index] = ~doubleTimeDurs;
                });
        },
        doubleTimeAmps: Pseq(#[15, 10], 1),
        doubleTimeDurs: Pn(0.125, 2)
);


~kik = Penvir(~kikEnvir, Pn(Plazy({
        ~init.value;
        ~addNotes.value;
        Pbindf(
                Pbind(
                        \instrument, \kik,
                        \preamp, 0.4,
                        \dur, 0.25,
                        *(~pbindPairs.value(#[amp, decay2]))
                ),
                        // default Event checks \freq --
                        // if a symbol like \rest or even just \,
                        // the event is a rest and no synth will be played
                \freq, Pif(Pkey(\amp) > 0, 1, \rest)
        )
}), inf)).play(quant: 4);

~snr = Penvir(~snrEnvir, Pn(Plazy({
        ~init.value;
        ~addNotes.value;
        Pbindf(
                Pbind(
                        \instrument, \kraftySnr,
                        \dur, 0.25,
                        *(~pbindPairs.value(#[amp, decay]))
                ),
                \freq, Pif(Pkey(\amp) > 0, 5000, \rest)
        )
}), inf)).play(quant: 4);

~hh = Penvir(~hhEnvir, Pn(Plazy({
        ~init.value;
        ~addNotes.value;
        Pbindf(
                Pbind(
                        \instrument, \kraftySnr,
                        \rq, 0.06,
                        \amp, 15,
                        \decay, 0.04,
                        *(~pbindPairs.value(#[amp, dur]))
                ),
                \freq, Pif(Pkey(\amp) > 0, 12000, \rest)
        )
}), inf)).play(quant: 4);
)

// stop just before barline
t = TempoClock.default;
t.schedAbs(t.nextTimeOnGrid(4, -0.001), {
        [~kik, ~snr, ~hh].do(_.stop);
});
```

Previous: PG_Cookbook06_Phrase_Network
Next: PG_Ref01_Pattern_Internals

## Inner workings of patterns

### Patterns as streams

As noted, patterns by themselves don't do much. They have to be turned into streams first; then, values are requested from a stream, not from the pattern.

For most patterns, the stream is an instance of Routine. Routines (formally known in computer science as "coroutines") are important because they can yield control back to the caller but still remember exactly where they were, so they can resume in the middle on the next call without having to start over. A few exceptional patterns use FuncStream, which is simply a wrapper around a function that allows a function to act like a stream by responding to 'next' and other Stream methods.

Every pattern class must respond to 'asStream'; however, most patterns do not directly implement asStream. Instead, they use the generic asStream implementation from Pattern.

```
asStream { ^Routine({ arg inval; this.embedInStream(inval) }) }
```

This line creates a Routine whose job is simply to embed the pattern into its stream. "Embedding" means for the pattern to do its assigned work, and return control to the parent level when it's finished. When a simple pattern finishes, its parent level is the Routine itself. After 'embedInStream' returns, there is nothing else for the Routine to do, so that stream is over; it can only yield nil thereafter.

```
p = Pseries(0, 1, 3).asStream;    // this will yield exactly 3 values
4.do { p.next.postln };           // 4th value is nil
```

We saw that list patterns can contain other patterns, and that the inner patterns are treated like "subroutines." List patterns do this by calling embedInStream on their list items. Most objects are embedded into the stream just by yielding the object:

```
        // in Object
embedInStream { ^this.yield; }
```

But if the item is a pattern itself, control enters into the subpattern and stays there until the subpattern ends. Then control goes back to the list pattern to get the next item, which is embedded and so on.

```
p = Pseq([Pseries(0, 1, 3), Pgeom(10, 2, 3)], 1).asStream;

p.next;  // Pseq is embedded; first item is Pseries(0...), also embedded
            // Control is now in the Pseries
p.next;  // second item from Pseries
p.next;  // third item from Pseries
p.next;  // no more Pseries items; control goes back to Pseq
            // Pseq gets the next item (Pgeom) and embeds it, yielding 10
p.next;  // second item from Pgeom
p.next;  // third item from Pgeom
p.next;  // no more Pgeom items; Pseq has no more items, so it returns to Routine
            // Routine has nothing left to do, so the result is nil
```

To write a new pattern class, then, the bare minimum required is:

- **Instance variables** for the pattern's parameters
- A ***new** method to initialize those variables
- An **embedInStream** method to do the pattern's work

One of the simpler pattern definitions in the main library is Prand:

```
Prand : ListPattern {
        embedInStream { arg inval;
```

```
                var item;

                repeats.value.do({ arg i;
                        item = list.at(list.size.rand);
                        inval = item.embedInStream(inval);
                });
                ^inval;
        }
}
```

This definition doesn't show the instance variables or *new method. Where are they? They are inherited from the superclass, ListPattern.

```
ListPattern : Pattern {
        var <>list, <>repeats=1;

        *new { arg list, repeats=1;
                if (list.size > 0) {
                        ^super.new.list_(list).repeats_(repeats)
                }{
                        Error("ListPattern (" ++ this.name ++ ") requires a non-empty
collection; received "
                                ++ list ++ ".").throw;
                }
        }
        // some misc. methods omitted in this document
}
```

Because of this inheritance, Prand simply expresses its behavior as a 'do' loop, choosing 'repeats' items randomly from the list and embedding them into the stream. When the loop is finished, the method returns the input value (see below).

**Streams' input values (inval, inevent)**

Before discussing input values in patterns, let's take a step back and discuss how it works for Routines.

Routine's 'next' method takes one argument, which is passed into the stream (Routine). The catch is that the routine doesn't start over from the beginning -- if it did, it would lose its unique advantage of remembering its position and resuming on demand. So it isn't sufficient to receive the argument using the routine function's argument variable.

In reality, when a Routine yields a value, its execution is interrupted after calling 'yield', but before 'yield' returns. Then, when the Routine is asked for its next value, execution resumes by providing a return value from the 'yield' method. (This behavior isn't visible in the SuperCollider code in the class library; 'yield' is a primitive in the C++ backend, which is how it's able to do something that is otherwise impossible in the language.)

For a quick example, consider a routine that is supposed to multiply the input value by two. First, the wrong way, assuming that everything is done by the function argument 'inval'. In reality, the first inval to come in is 1. Since nothing in the routine changes the value of inval, the routine yields the same value each time.

```
r = Routine({ |inval|
        loop {
                yield(inval * 2)
        }
});

(1..3).do { |x| r.next(x).postln };
```

If, instead, the routine saves the result of 'yield' into the inval variable, the routine becomes aware of the successive input values and returns the expected results.

```
r = Routine({ |inval|
	loop {
			// here is where the 2nd, 3rd, 4th etc. input values come in
		inval = yield(inval * 2);
	}
});

(1..3).do { |x| r.next(x).postln };
```

This convention -- receiving the first input value as an argument, and subsequent input values as a result of a method call -- holds true for the embedInStream method in patterns also. The rules are:

- embedInStream takes **one argument**, which is the first input value.
- When the pattern needs to yield a value directly, or embed an item into the stream, it receives the next input value as the result of 'yield' or 'embedInStream': **inval = output.yield** or **output.embedInStream(inval)**.
- When the pattern exits, it must return the last input value, so that the parent pattern will get the input value as the result of its embedInStream call: **^inval**.

By following these rules, embedInStream becomes a near twin of yield. Both do essentially the same thing: spit values out to the user, and come back with the next input value. The only difference is that yield can return only one object to the 'next' caller, while embedInStream can yield several in succession.

Take a moment to go back and look at how Prand's embedInStream method does it.

**embedInStream vs. asStream + next**

If a pattern class needs to use values from another pattern, should it evaluate that pattern using embedInStream, or should it make a separate stream (asStream) and pull values from that stream using 'next'? Both approaches are used in the class library.

embedInStream turns control over to the subpattern completely. The outer pattern is effectively suspended until the subpattern gives control back. This is the intended behavior of most list patterns, for example. There is no opportunity for the parent to do anything to the value yielded back to the caller.

This pattern demonstrates what it means to give control over to the subpattern. The first pattern in the Pseq list is infinite; consequently, the second subpattern will never execute because the infinite pattern never gives control back to Pseq.

```
p = Pseq([Pwhite(0, 9, inf), Pwhite(100, 109, inf)], 1).asStream;
p.nextN(20);	// no matter how long you do this, it'll never be > 9!
```

asStream should be used if the parent pattern needs to perform some other operation on the yield value before yielding, or if it needs to keep track of multiple child streams at the same time. For instance, Pdiff takes the difference between the current value and last value. Since the subtraction comes between evaluating the child pattern and yielding the difference, the child pattern must be used as a stream.

```
Pdiff : FilterPattern {
	embedInStream { arg event;
			// here is the stream!
		var stream = pattern.asStream;
		var next, prev = stream.next(event);
		while {
			next = stream.next(event);
			next.notNil;
```

```
                }{
                        // and here is the return value
                        event = (next - prev).yield;
                        prev = next;
                }
                ^event
        }
}
```

**Writing patterns: other factors**

Pattern objects are supposed to be *stateless*, meaning that the pattern object itself should undergo no changes based on any stream running the pattern. (There are some exceptions, such as Ppatmod, which exists specifically to perform some modification on a pattern object. But, even this special case makes a separate copy of the pattern to be modified for each stream; the original pattern is insulated from the streams' behavior.) *Be very careful if you're thinking about breaking this rule*, and before doing so, think about whether there might be another way to accomplish the goal without breaking it.

Because of this rule, *all variables reflecting the state of a particular stream should be local to the embedInStream method*. If you look through existing pattern classes for examples, you will see in virtually every case that embedInStream does not alter the instance variables defined in the class. It uses them as parameters, but does not change them. Anything that changes while a stream is being evaluated is a local method variable.

To initialize the pattern's parameters (instance variables), typical practice in the library is to give getter and setter methods to all instance variables, and use the setters in the *new method (or, use ^super.newCopyArgs(...)). It's not typical to have an init method populate the instance variables. E.g.,

```
Pn : FilterPattern {
        var <>repeats;
        *new { arg pattern, repeats=inf;
                        // setter method used here for repeats
                ^super.new(pattern).repeats_(repeats)
        }
...
}
```

Consider carefully whether a parameter can change in each 'next' call. If so, make a stream from that parameter and call .next(inval) on it for each iteration. Parameters that should not change, such as number of repeats, should call .value(inval) so that a function may be given. Pwhite demonstrates both of these features.

**Exercise for the reader:** Why does Pwhite(0.0, 1.0, inf) work, even with the asStream and next calls?

```
Pwhite : Pattern {
        var <>lo, <>hi, <>length;
        *new { arg lo=0.0, hi=1.0, length=inf;
                ^super.newCopyArgs(lo, hi, length)
        }
        storeArgs { ^[lo,hi,length] }
        embedInStream { arg inval;
                        // lo and hi streams
                var loStr = lo.asStream;
                var hiStr = hi.asStream;
                var hiVal, loVal;
                        // length.value -- functions allowed for length
                        // e.g., Pwhite could give a random number of values for each embed
                length.value.do({
                        hiVal = hiStr.next(inval);
                        loVal = loStr.next(inval);
                        if(hiVal.isNil or: { loVal.isNil }) { ^inval };
```

```
                inval = rrand(loVal, hiVal).yield;
        });
        ^inval;
    }
}

// the plot rises b/c the lo and hi values increase on every 'next' value
Pwhite(Pseries(0.0, 0.01, inf), Pseries(0.2, 0.01, inf), inf).asStream.nextN(200).plot;
```

**Cleaning up event pattern resources**

Some event patterns create server or other objects that need to be explicitly removed when they come to a stop. This is handled by the **EventStreamCleanup** object. This class stores a set of functions that will run at the pattern's end. It also uses special keys in the current event to communicate cleanup functions upward to parent patterns, and ultimately to the EventStreamPlayer that executes the events.

Basic usage involves 4 stages:

```
    embedInStream { |inval|
        var     outputEvent;

        // #1 - make the EventStreamCleanup instance
        var   cleanup = EventStreamCleanup.new;

        // #2 - make persistent resource, and add cleanup function
        // could be some kind of resource other than a Synth
        synth = (... make the Synth here...);
        cleanup.addFunction(inval, { |flag|
            if(flag) {
                synth.release
            };
        });

        loop {
            outputEvent = (... get output event...);

            // #4 - cleanup.exit
            if(outputEvent.isNil) { ^cleanup.exit(inval) };

            // #3 - update the EventStreamCleanup before yield
            cleanup.update(outputEvent);
            outputEvent.yield;
        }
    }
```

1. The embedInStream method should create its own instance of EventStreamCleanup. (Alternately, it may receive the cleanup object as the second argument, but it should not assume that the cleanup object will be passed in. It should always check for its existence and create the instance if needed. Note that the pattern should also reimplement asStream as shown.) It's much simpler for the pattern just to create its own instance.

2. When the pattern creates the objects that will need to be cleaned up, it should also use the **addFunction** method on the EventStreamCleanup with a function that will remove the resource(s). (The example above is deliberately oversimplified. In practice, attention to the timing of server actions is important. Several pattern classes give good examples of how to do this, e.g., Pmono, Pfx.)

   The flag should be used when removing Synth or Group nodes. Normally the flag is true; but, if the pattern's EventStreamPlayer gets stopped by cmd-., the nodes will already be gone from the server. If your

function tries to remove them again, the user will see FAILURE messages from the server and then get confused, thinking that she did something wrong when in fact the error is preventable in the class.

3. Before calling .yield with the return event, also call **cleanup.update(outputEvent)**.

4. When embedInStream returns control back to the parent, normally this is done with ^inval. When an EventStreamCleanup is involved, it should be **^cleanup.exit(inval)**. This executes the cleanup functions and also removes them from EventStreamCleanups at any parent level.

**When does a pattern need an EventStreamCleanup?**

If the pattern creates something on the server (bus, group, synth, buffer etc.), it must use an EventStreamCleanup as shown to make sure those resources are properly garbage collected.

Or, if there is a chance of the pattern stopping before one or more child patterns has stopped on its own, EventStreamCleanup is important so that the pattern is aware of cleanup actions from the children. For example, in a construction like ~~Pfindur~~(10, ~~Pmono~~(name, pairs...)), Pmono may continue for more than 10 beats, in which case Pfindur will cut it off. The Pmono needs to end its synth, but it doesn't know that a pattern higher up in the chain is making it stop. It becomes the parent's responsibility to clean up after the children. As illustrated above, EventStreamCleanup handles this with only minimal intrusion into normal pattern logic.

Previous: PG_Cookbook07_Rhythmic_Variations