Uma Gentil Introdução ao SuperCollider

por Bruno Ruviaro

revisão 26 de julho de 2016
tradução: Rodolfo Valente e Bruno Ruviaro

Sumário

A Gentle Introduction to SuperCollider

Bruno Ruviaro

26 de julho de 2016

Parte I
O BÁSICO

1 Olá Mundo

Tudo pronto para criar seu primeiro programa em SuperCollider? Assumindo que você tem o SC aberto e funcionando à sua frente, abra um novo documento (menu File→New ("Arquivo→Novo") ou o atalho [ctrl+N]) e digite a seguinte linha:

```
1 "Olá Mundo".postln;
```

Posicione o cursor em qualquer lugar desta linha (não importa se início, meio ou fim). Pressione [ctrl+Enter] para executar o código. "Olá Mundo"aparece no seu Post window ("Janela de postagem"). Parabéns! Este foi seu primeiro programa em SuperCollider.

Figura 1: A interface IDE do SuperCollider.

DICA: Em todo este livro, ctrl (control) indica a tecla modificadora para atalhos de teclado que é usada nas plataformas Linux e Windows. No Mac OSX, use a tecla command (cmd).

A figura 1 mostra uma foto de tela do IDE (Integrated Development Environment ou "Ambiente de desenvolvimento integrado") do SuperCollider no momento em que é aberto. Vamos agora nos familiarizar com essa interface.

O que é o IDE do SuperCollider? É um "ambiente de programação multiplataforma desenvolvido especificamente para o SuperCollider (. . .), fácil de começar a usar, prático de lidar e dotado

de recursos poderosos para programadores experientes. Ele também é bastante personalizável. Roda igualmente bem e tem quase a mesma aparência no Mac OSX, Linux e Windows."*

As principais partes que você vê são o Editor de Código, o Navegador de Ajuda ("Help browser") e a Janela de Postagem ("Post window"). Se você não estiver vendo qualquer uma dessas quando abrir o Super Collider, simplesmente vá para o menu View→Docklets (é onde você pode exibir ou esconder cada uma delas). Há também a Barra de Status, sempre localizada no canto inferior direito da janela.

Sempre mantenha a Post window visível, mesmo se você ainda não entender todas as coisas que são mostradas ali. A Post window mostra as reações do programa aos seus comandos: resultados da execução de códigos, notificações diversas, avisos, erros, etc.

DICA: Você pode aumentar ou reduzir temporariamente o tamanho da fonte do editor com os atalhos [Ctrl++] e [Ctrl+-] (ou seja, a tecla control junto com as teclas de mais ou menos, respectivamente). Se você está em um laptop que não tem uma tecla + de verdade, use [Ctrl+shift+=].

2 Servidor e Linguagem

Na Barra de Status você pode ver as palavras "Interpreter"("Interpretador") e "Server"("Servidor"). Por definição, o Interpretador já vem ligado ("Active") na abertura do programa, enquanto o "Servidor"vem desligado (é isso o que todos aqueles zeros querem dizer). O que é o Interpretador é o que é o Servidor?

*Citado da documentação do SuperCollider: `http://doc.sccode.org/Guides/SCIde.html`. Visite esta página para aprender mais sobre a interface do IDE.

O SuperCollider, na realidade, é composto de dois aplicativos distintos: o servidor e a linguagem. O servidor é responsável por fazer sons. A linguagem (também chamada *cliente* ou *interpretador*) é usada para controlar o servidor. O primeiro é chamado scsynth ("SC-synthesizer") e o segundo, sclang ("SC-language"). A Barra de Status diz o status (ligado/desligado) de cada um destes dois componentes.

Não se preocupe se esta distinção não faz muito sentido para você agora. As duas coisas principais que você precisa saber por enquanto são:

1. Tudo o que você digita no SuperCollider está na linguagem do SuperCollider (o cliente): é onde você escreve e executa comandos, vendo os resultados na Post window.

2. Todo som que o SuperCollider faz está vindo do servidor—o "motor sonoro", por assim dizer—, controlado por você através da linguagem do SuperCollider.

2.1 Iniciando o Servidor

Seu programa "Olá Mundo"não produziu som algum: tudo aconteceu na linguagem e o servidor nem chegou a ser usado. O próximo exemplo produzirá som, então precisamos ter certeza de que o Servidor está ligado e funcionando.

O jeito mais fácil de iniciar o servidor é com o atalho [ctrl+B]. Alternativamente, você pode clicar nos zeros da Barra de Status: no menu que aparece, escolha a opção "Boot Server"("Iniciar Servidor"). Você observará alguma atividade na Post Window enquanto o servidor está iniciando. Depois que você tiver iniciado o servidor com sucesso, os números na Barra de Status vão ficar verdes. Você terá de fazer isso todas as vezes que iniciar o SC, mas apenas uma vez por sessão.

3 Sua primeira senoide

"Olá Mundo"é tradicionalmente o primeiro programa que as pessoas criam quando estão aprendendo uma nova linguagem de programação. Você já fez isso no SuperCollider.

Criar uma onda senoidal simples talvez seja o "Olá Mundo"das linguagens de programação para música. Vamos direto à senoide então. Digite e execute a seguinte linha de código. Cuidado—o volume pode ser alto. Abaixe todo o volume do seu computador, execute a linha e aumente o volume devagar.

```
1 {SinOsc.ar}.play;
```

Trata-se de uma senoide bela, suave, contínua e talvez um pouco entediante. Você pode parar o som com [ctrl+.] (ou seja, a tecla *control* junto com a tecla de *ponto final*). Memorize esta combinação de teclas, porque você a utilizará muito para interromper todo e qualquer som no SC.

DICA: Em uma linha separada, digite e execute **`s.volume.gui`** se você quiser um slider gráfico para controlar o volume da saída do SuperCollider.

Agora vamos tornar esta senoide um pouco mais interessante. Digite isto:

```
1 {SinOsc.ar(LFNoise0.kr(10).range(500, 1500), mul: 0.1)}.play;
```

Lembre-se, basta deixar o cursor em qualquer lugar da linha e apertar [ctrl+Enter] para executar. Ou, se preferir, você pode também selecionar toda a linha antes de executá-la.

DICA: Digitar os próprios exemplos é uma grande ferramenta de aprendizagem. Isso irá ajudar a criar confiança e familiaridade com a linguagem. Ao ler tutoriais em formato digital, você pode às vezes sentir uma certa preguiça e ficar tentado a copiar e colar o código dos exemplos. Evite fazer isso: você aprenderá melhor se digitar tudo por conta própria, ao menos nos primeiros estágios da sua aprendizagem com o SC.

4 Mensagens de erro

Não saiu nenhum som quando você rodou o último exemplo? Se isso aconteceu, provavelemente seu código tem um erro de digitação: um caracter errado, uma vírgula ou um parêntese a menos, etc. Quando algo acontece de errado no seu código, a Post window mostra uma mensagem de erro. Mensagens de erro podem ser longas e obscuras, mas não entre em pânico: com o tempo você aprenderá a lê-las. Veja abaixo um exemplo de uma mensagem de erro curta:

```
ERROR: Class not defined.
  in file 'selected text'
  line 1 char 19:

  {SinOsc.ar(LFNoise0.kr(12).range(400, 1600), mul: 0.01)}.play;

----------------------------------
nil
```

Esta mensagem de erro diz "Class not defined"(Classe não definida) e aponta a localização aproximada do erro ("line 1 char 19", ou seja: linha 1, caracter 19). Classes no SC são aquelas palavras azuis que começam com uma letra maiúscula (como `SinOsc` e `LFNoise0`). O que causou

o erro nesse exemplo foi que a pessoa digitou LFNoiseO com uma letra "O"maiúscula ao final. A classe correta é LFNoise0, com o número zero ao final. Parece até pegadinha de vestibular, mas é verdade. Como você pode ver, atenção aos detalhes é crucial.

Se você tem um erro no seu código, revise-o, mude o que for necessário e tente novamente até que ele esteja corrigido. Se você ainda não cometeu nenhum erro, experimente introduzir um para que você possa ver como é uma mensagem de erro (por exemplo, remova um ponto ou uma vírgula de um dos exemplos das senoides).

DICA: Aprender SuperCollider é como aprender uma outra língua como Alemão, Inglês ou Japonês... você tem que praticar falá-la o máximo possível, esforce-se em expandir seu vocabulário, preste atenção na gramática e na sintaxe e aprenda com seus erros. A pior coisa que pode acontecer é você travar o SuperCollider, o que é bem menos ruim do que pegar um ônibus errado em Nova Iorque por culpa de um erro de pronúncia na hora de pedir informações.

5 Mudando parâmetros

Segue abaixo um exemplo interessante adaptado do primeiro capítulo do The SuperCollider Book.* Da mesma forma que em exemplos anteriores, não se preocupe em entender tudo. Apenas aprecie o resultado sonoro e brinque com os números.

```
1 {RLPF.ar(Dust.ar([12, 15]), LFNoise1.ar([0.3, 0.2]).range(100, 3000), 0.02)}.play;
```

*Wilson, S. and Cottle, D. and Collins, N. (Editors). The SuperCollider Book, MIT Press, 2011, p. 5. Diversas coisas neste tutorial foram emprestadas, adaptadas ou inspiradas pelo excelente "Beginner's Tutorial"de David Cottle, que é o primeiro capítulo do livro, mas—diferentemente dele–aqui assumimos que o leitor tenha pouca familiaridade com computação musical, e apresentamos a família de Patterns como eixo principal da abordagem pedagógica.

Pare o som, mude alguns números e rode novamente. Por exemplo, o que acontece quando você substitui os números 12 e 15 por valores mais baixos, entre 1 e 5? Depois de `LFNoise1`, que tal se em vez de 0.3 e 0.2 você tentasse algo como 1 ou 2? Mude um de cada vez. Compare o novo som com o som anterior, escute as diferenças. Veja se você consegue entender qual número está controlando o quê. Esta é uma maneira divertida de explorar o SuperCollider: pegue um trecho de código que faça algo interessante e experimente com os parâmetros para criar variações. Mesmo se você não entender completamente a função exata de cada número, ainda assim pode encontrar resultados sonoros interessantes.

DICA: Como qualquer software, lembre-se de salvar frequentemente o seu trabalho com [ctrl+S]! Quanto estiver trabalhando em tutoriais como esse, você muitas vezes vai chegar a sons interessantes experimentando com os exemplos fornecidos. Quando você quiser guardar algo que gostou, copie o código em um novo documento e salve-o. Repare que todo arquivo do SuperCollider tem a extensão .scd, que quer dizer "SuperCollider Document".

6 Comentários

Todo o texto que aparece em vermelho no seu código é um *comentário*. Se você é novo em linguagens de programação, comentários são bastante úteis para documentar o seu código, tanto para você mesmo, quanto para outros que tenham de lê-lo depois. Qualquer linha começando com uma barra dupla é um comentário. Você pode escrever comentários logo depois de uma linha válida de código, pois a parte do comentário será ignorada quando você rodar. No SC, usamos um ponto-e-vírgula para indicar o fim de um enunciado válido.

```
1 2 + 5 + 10 - 5; // apenas fazendo contas
2
3 rrand(10, 20); // gerar um numero aleatório entre 10 e 20
```

Você pode rodar uma linha mesmo que o seu cursor estiver no meio de um comentário depois desta linha. A parte do comentário é ignorada. Os próximos dois parágrafos serão escritos como "comentários"apenas como exemplo.

```
1 // Você pode rapidamente transformar uma linha de código em comentário usando o
      atalho [ctrl+/].
2 "Algum código de SC aqui...".postln;
3 2 + 2;
4
5
6 // Se você escrever um comentário beeeem longo (uma única linha longa), seu texto
      vai ser quebrado em várias "linhas" que não vão começar com barra dupla. No
      entanto, tudo isso ainda conta como uma só linha de comentário.
7
8 /* Use "barra + asterisco" para começar um comentário longo com diversas linhas.
      Feche o trecho de comentário com "asterisco + barra". O atalho mencionado
      anteriormente também funciona para grandes trechos: simplesmente selecione as
      linhas de código que você quer "comentar" ("comment out", em inglês) e pressione
       [ctrl+/]. O mesmo serve para des—comentar ("uncomment").*/
```

7 Precedência

O SuperCollider segue a ordem de precedência da esquerda para a direita, independente da operação. Isso significa, por exemplo, que multiplicação *não* acontece primeiro:

```
1 // Na escola, o resultado era 9; no SC é 14:
2 5 + 2 * 2;
3 //Use parênteses para forçar uma ordem de operações específica:
4 5 + (2 * 2); // igual a 9.
```

Quando se combina mensagens e operações binárias, mensagens assumem precedência. Por exemplo, em `5 + 2.squared`, a elevação ao quadrado acontece primeiro.

8 A última coisa é sempre postada

Um detalhe pequeno mas útil para se entender: o SuperCollider tem como padrão sempre postar na Post window o resultado de qualquer operação que tenha sido a *última coisa a ser executada*. Isso explica porque o seu código "Olá Mundo"imprime duas vezes quando você o executa. Digite as próximas linhas em um novo documento, depois selecione tudo com [ctrl+A] e rode todas as linhas de uma vez:

```
1 "Primeira linha".postln;
2 "Segunda linha".postln;
3 (2 + 2).postln;
4 3 + 3;
5 "Fim".postln;
```

Todas as cinco linhas são executadas pelo SuperCollider. Você vê o resultado de `2 + 2` na Post window porque existia um pedido de `postln` explícito. O resultado de `3 + 3` foi calculado, mas não foi dada nenhuma instrução para postá-lo, então você não o vê na Post window. Depois do 3 + 3, é executado o comando da última linha (a palavra "Fim"é postada por conta da solicitação `postln`). Finalmente, como padrão, o resultado da última coisa a ser rodada é postado: neste caso, acabou sendo a palavra "Fim".

9 Blocos de código

É um pouco chato ter que ficar selecionando várias linhas de um código toda vez antes de rodá-lo. Uma maneira muito mais fácil de rodar uma porção de código ao mesmo tempo é criar um *bloco de código*: simplesmente coloque dentro de parênteses todas as linhas de código que você quer rodar juntas. Aqui está um exemplo:

```
1 (
2 // Um pequeno poema
3 "Hoje é domingo".postln;
4 "Pé de cachimbo".postln;
5 "O cachimbo é de ouro".postln;
6 "Bate no touro".postln;
7 )
```

Os parênteses nas linhas 1 e 7 delimitam o bloco de código. Desde que que o cursor esteja em qualquer lugar dentro dos parênteses, um único [ctrl+Enter] rodará as linhas para você (elas serão executadas em ordem de cima para baixo, mas isso é tão rápido que parece simultâneo).

Usar blocos de código poupa o trabalho de ter que selecionar todas as linhas novamente a cada vez que você quiser mudar algo e rodar novamente. Por exemplo, mude algumas das palavras entre aspas e pressione [ctrl+Enter] logo após fazer a mudança. Todo o bloco de código é rodado sem que você tenha que selecionar manualmente todas as linhas. Ao rodar o bloco, o SuperCollider mostra a seleção por um momento para dar uma indicação visual do que está sendo executado.

10 Como limpar a Post window

Este é um comando tão útil para maníacos por limpeza que merece uma seção só pra ele: [ctrl+shift+P]. Execute esta linha e experimente limpar a Post window em seguida:

```
100.do({"Imprima esta linha um monte de vezes...".scramble.postln});
```

Nem precisa agradecer.

11 Como gravar os sons do SuperCollider

Logo você vai querer começar a gravar a saída de som dos seus patches de SuperCollider. Eis um jeito rápido:

```
// GRAVAÇÃO RÁPIDA
// Começar a gravar:
s.record;
// Faça algum som bacana:
{Saw.ar(LFNoise0.kr([2, 3]).range(100, 2000), LFPulse.kr([4, 5]) * 0.1)}.play;
// Pare de gravar:
s.stopRecording;
// Opcional: GUI com botão de gravação, controle de volume, botão de mudo.
s.makeWindow;
```

A Post window mostra o caminho para a pasta onde o arquivo foi salvo. Vá até o arquivo, abra-o no Audacity ou um programa similar e verifique se o arquivo foi mesmo gravado. Para mais informações, veja o arquivo de Ajuda de "Server"(role para baixo até "Recording Support"). Também online em `http://doc.sccode.org/Classes/Server.html`.

12 Variáveis

Você pode guardar números, palavras, unidades geradoras, funções ou blocos inteiros de código em variáveis. Variáveis podem ser letras minúsculas simples ou palavras escolhidas por você. Usamos o sinal de igual (=) para "atribuir"variáveis. Rode estas linhas uma de cada vez e observe a Post window:

```
x = 10;
y = 660;
y; // confira o que está aqui dentro
x;
x + y;
y - x;
```

A primeira linha atribui o número 10 à variável `x`. A segunda linha coloca 660 na variável `y`. As próximas duas linhas provam que estas letras agora "contêm"estes números (os dados). Finalmente, as duas últimas linhas mostram que podemos usar as variáveis para fazer qualquer operação com os dados guardados nelas.

Letras minúsculas de `a` a `z` podem ser utilizadas a qualquer momento como variáveis no SuperCollider. A única letra simples que comumente não se usa é `s`, que por convenção é reservada para representar o Servidor. Qualquer coisa pode entrar em uma variável:

```
a = "Olá Mundo"; // uma cadeia de caracteres
b = [0, 1, 2, 3, 5]; // uma lista
c = Pbind(\note, Pwhite(0, 10), \dur, 0.1); // você aprenderá tudo sobre Pbind mais
    adiante, não se preocupe

// ...e agora você pode utilizá-las como você utilizaria os dados originais:
a.postln; // poste-a
b + 100; // faça contas
```

```
8 c.play; // toque aquele Pbind
9 d = b * 5; // pegue b, multiplique por 5 e atribua o resultado a uma nova variável
```

Muitas vezes fará mais sentido dar nomes melhores para suas variáveis, para ajudar a lembrar o que elas representam no seu código. Você pode usar um ∼ (til) para declarar uma variável com um nome mais longo. Note que não há espaço entre o til e o nome da variável.

```
1 ~minhasFreqs = [415, 220, 440, 880, 220, 990];
2 ~minhasDurs = [0.1, 0.2, 0.2, 0.5, 0.2, 0.1];
3
4 Pbind(\freq, Pseq(~minhasFreqs), \dur, Pseq(~minhasDurs)).play;
```

Nomes de variáveis devem começar com letras minúsculas depois do til. Você pode usar números, underscores e letras maiúsculas no meio do nome, somente não como primeiro caracter. Todos os caracteres têm de ser contíguos (sem espaços ou pontuação). Resumindo, atenha-se a letras e números e underscores ocasionais, evitando todos os outros caracteres ao nomear suas variáveis. ∼`minhasFreqs`, ∼`aMelhorSenoide` e ∼`banana_3` são nomes válidos. ∼`MinhasFreqs`, ∼`aMelhor&*#Senoide` e ∼`banana!!!` vão estragar o seu dia.

Há dois tipos de variáveis que você pode criar: variáveis "globais"e variáveis locais.

12.1 "Global"vs. Local

As variáveis que vimos até agora (letras minúsculas de `a` a `z` e aquelas começando com o caracter til (∼)) são genericamente chamadas "variáveis globais", porque uma vez declaradas, funcionarão "globalmente"em qualquer lugar no patch, em outros patches e mesmo em outros documentos do SC, até que você saia do SuperCollider.*

*Tecnicamente falando, variáveis começando com um til são chamadas variáveis de Ambiente ("Environment") e variáveis de letras minúsculas (de a a z) são chamadas variáveis do Interpretador. Iniciantes no SuperCollider

Variáveis locais, por outro lado, são declaradas com a palavra-chave específica `var` no início de uma linha. Você pode atribuir um valor inicial para uma variável no momento de sua declaração (`var tangerina = 4`). Variáveis locais apenas existem dentro do escopo de seu bloco de código.

Aqui está um exemplo simples comparando os dois tipos de variáveis. Execute linha a linha e observe a Post window.

```
1  // Variáveis de Ambiente
2  ~abacates = 4;
3  ~laranjas = 5;
4  ~tangerinas = 2;
5  ~bananas = 1;
6
7  ["Cítricos", ~laranjas + ~tangerinas];
8  ["Não-Cítricos", ~bananas + ~abacates];
9
10 // Variáveis locais: válidas apenas dentro do bloco de código.
11 // Execute o bloco uma vez e observe o Post window:
12 (
13 var jacas = 4, laranjas = 3, tangerinas = 8, bananas = 10;
14 ["Frutas cítricas", laranjas + tangerinas].postln;
15 ["Frutas Não-cítricas", bananas + jacas].postln;
16 "Fim".postln;
17 )
18
19 ~abacates; // variável ainda existe
20 jacas; // variável não existe mais
```

não precisam se preocupar com estas distinções, mas mantenha-as em mente para o futuro. O Capítulo 5 do The SuperCollider Book explica a diferença em detalhes.

12.2 Reatribuição

Uma última coisa útil de se entender sobre variáveis é que elas podem ser *reatribuídas*: você pode dar-lhes um novo valor a qualquer momento.

```
// Atribua uma variável
a = 10 + 3;
a.postln; // verifique
a = 999; // reatribua a variável (dê-lhe um novo valor)
a.postln; // verifique: o valor antigo se foi.
```

Uma prática muito comum que pode parecer um pouco confusa para iniciantes é quando *a própria variável é usada na sua própria reatribuição.* Dê uma olhada neste exemplo:

```
x = 10; // atribua 10 à variável x
x = x + 1; // atribua x + 1 à variável x
x.postln; // verifique
```

A maneira mais fácil de entender essa última linha é lê-la da seguinte forma: "pegue o valor atual da variável x, adicione 1 a ela e atribua este novo resultado à variável x". No fundo, não é tão complicado e você verá mais tarde como isso pode ser útil.*

*Este exemplo claramente demonstra que o sinal de igual, em programação, não é o mesmo sinal de igual que você aprendeu em matemática. Em matemática, $x = x + 1$ é impossível (um número não pode ser igual a si mesmo mais um). Já em uma linguagem de programação como o SuperCollider, o sinal de igual pode ser entendido como um tipo de ação: *pegue o resultado da expressão à direita do símbolo e o "atribua"à variável ao lado esquerdo.*

Parte II
PATTERNS

13 A família Pattern

Vamos experimentar algo diferente agora. Digite e rode esta linha de código:

```
1 Pbind(\degree, Pseries(0, 1, 30), \dur, 0.05).play;
```

13.1 Conheça o Pbind

`Pbind` é um membro da família Pattern ("padrão"ou "modelo") no SuperCollider. O P maiúsculo no `Pbind` e `Pseries` remete a *Pattern*; em breve teremos o prazer de conhecer outros membros da família. Por ora, vamos focar somente somente no `Pbind`. Experimente este exemplo reduzido ao mínimo:

```
1 Pbind(\degree, 0).play;
```

A única coisa que esta linha de código faz na vida é tocar a nota *dó central*, uma vez por segundo. A palavra-chave `\degree` se refere a graus de uma escala e o número 0 representa o primeiro grau da escala (uma escala de Dó Maior é subentendida, então esta é a própria nota dó). Note que o SuperCollider começa a contar as coisas do 0 e não do 1. Em uma linha simples como esta acima, as notas dó, ré, mi, fá, sol... seriam representadas pelos números 0, 1, 2, 3, 4... Tente mudar o número e perceba como a nota muda quando você reexecuta. Você também pode escolher notas abaixo do dó central (por exemplo, -2 é a nota lá abaixo do dó central). Resumindo, apenas imagine que o dó central do piano é 0 e conte teclas brancas para cima ou para baixo (números positivos ou negativos) para obter qualquer outra nota.

Agora brinque um pouco com a duração das notas. `Pbind` usa a palavra-chave `\dur` para especificar durações em segundos:

```
1 Pbind(\degree, 0, \dur, 0.5).play;
```

Claro que isso ainda é muito rígido e inflexível—sempre a mesma nota, sempre a mesma duração. Não se preocupe: as coisas vão esquentar logo, logo.

13.2 Pseq

Vamos agora tocar várias notas em sequência, como uma escala. Vamos também diminuir a duração das notas para 0.2 segundos.

```
1 Pbind(\degree, Pseq([0, 1, 2, 3, 4, 5, 6, 7], 1), \dur, 0.2).play;
```

Esta linha introduz um novo membro da família Pattern: `Pseq`. Como o nome sugere, este Pattern lida com sequências. Tudo que um `Pseq` precisa para tocar uma sequência é:

- uma lista de itens entre colchetes []
- um número de repetições.

No exemplo, a lista é `[0, 1, 2, 3, 4, 5, 6, 7]` e o número de repetições é `1`. Este `Pseq` simplesmente diz: "toque todos os itens da lista, na sequência, uma vez". Repare que estes dois elementos, lista e número de repetições, estão dentro dos parênteses que pertencem ao `Pseq` e são separados por uma vírgula.

Veja também onde o `Pseq` aparece no interior do `Pbind`: é colocado como valor de `\degree`. Isso é importante: em vez de fornecer um número único e fixo para o grau da escala (como no nosso primeiro `Pbind` simples), *estamos fornecendo todo um Pseq: uma receita para uma sequência de números.* Com isto em mente, podemos facilmente expandir esta ideia e usar outro Pseq para controlar também as durações.

```
Pbind(\degree, Pseq([0, 1, 2, 3, 4, 5, 6, 7], 5), \dur, Pseq([0.2, 0.1, 0.1, 0.2,
    0.2, 0.35], inf)).play;
```

O que está acontecendo neste exemplo? Primeiro, mudamos o número de repetições do primeiro `Pseq` para 5, de modo que toda a escala vai tocar cinco vezes. Segundo, substituímos o valor de `\dur`, antes fixo em 0.2, por um outro `Pseq`. Este novo `Pseq` tem uma lista de seis itens: `[0.2, 0.1, 0.1, 0.2, 0.2, 0.35]`. Estes números se tornam valores de duração das notas resultantes. O valor de `repeats` deste segundo `Pseq` é definido como `inf`, que quer dizer "infinito". Isso sgnifica que o `Pseq` não tem limite no número de vezes que ele pode repetir esta sequência. Então o `Pbind` toca para sempre? Não: ele para depois que o *outro* `Pseq` terminou seu trabalho, isto é, depois que a sequência de graus de escala foi tocada 5 vezes.

Finalmente, o exemplo tem um total de oito notas diferentes (a lista no primeiro `Pseq`), enquanto há apenas seis valores para duração (segundo `Pseq`). Quando você fornece sequências de tamanhos diferentes como estas, o `Pbind` simplesmente as faz circular o quanto for preciso.

Responda estas perguntas para praticar o que você aprendeu:

- Experimente o número 1 em vez de inf como o argumento `repeats` do segundo `Pseq`. O que acontece?
- Como você pode fazer este Pbind tocar para sempre?

Soluções estão no final do livro.[1]

13.3 Deixe seu código mais legível

Você deve ter percebido que a linha de código acima é um tanto longa. De fato, ela é tão longa que quebra em uma nova linha, mesmo que tecnicamente se trate de um enunciado único. Linhas de código longas podem ser confusas de ler. Para evitar isso, é uma prática comum quebrar o

código em várias linhas indentadas; com o objetivo de torná-lo o mais claro e inteligível possível. O mesmo `Pbind` pode ser escrito assim:

```
(
Pbind(
        \degree, Pseq([0, 1, 2, 3, 4, 5, 6, 7], 5),
        \dur, Pseq([0.2, 0.1, 0.1, 0.2, 0.2, 0.35], inf)
).play;
)
```

De agora em diante, adquira o hábito de escrever seus `Pbind`s deste jeito. Escrever código com uma aparência arrumada e bem organizada vai ajudá-lo bastante no aprendizado do SuperCollider.

Além disso, perceba que envolvemos este `Pbind` dentro de parênteses para criar um bloco de código (lembra-se da seção 9?): como ele não está mais em uma única linha, precisamos fazer isso para conseguir rodá-lo todo de uma vez. Lembre-se que o cursor precisa estar em algum lugar no interior do bloco antes de executá-lo.

13.4 Quatro maneiras de especificar alturas

`Pbind` aceita outras maneiras de especificar alturas, não apenas graus de escala.

- Se você quiser usar todas as doze notas cromáticas (teclas brancas e pretas do piano), você pode usar `\note` em vez de `\degree`. 0 continuará representando o dó central, mas agora os números incluem as teclas pretas do piano (0=dó central, 1=dó♯, 2=ré, etc).
- Se você preferir usar a numeração de notas MIDI, use `\midinote` (60=dó central, 61=dó♯, 62=ré, etc).
- Finalmente, se você quiser especificar frequências diretamente em Herz, use `\freq`.

Veja a figura 2 para uma comparação dos quatro métodos.

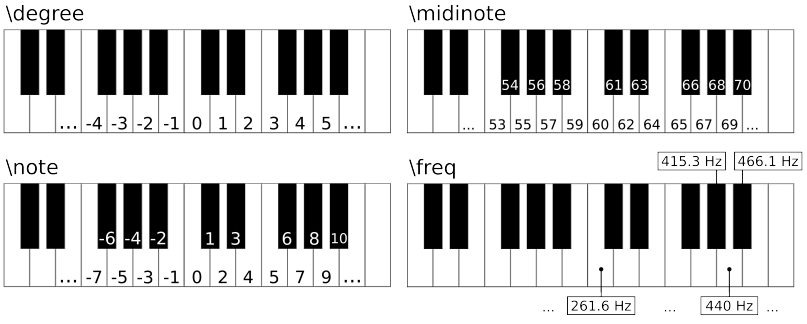

Figura 2: Comparando graus de escala, números de nota, notas MIDI e frequências

No próximo exemplo, quatro Pbinds vão tocar a mesma nota: o lá acima do dó central (lá 4).

```
1 Pbind(\degree, 5).play;
2 Pbind(\note, 9).play;
3 Pbind(\midinote, 69).play;
4 Pbind(\freq, 440).play;
```

DICA: Lembre-se que cada maneira de especificar alturas requer números em âmbitos diferentes. Uma lista de números como `[-1, 0, 1, 3]` faz sentido para `\degree` e `\note`, mas não faz sentido para `\midinote` ou `\freq`. A tabela abaixo compara alguns valores usando o teclado do piano como referência.

	lá 0 (nota mais grave do piano)	dó 4	lá 4	dó 5	dó 8
`\degree`	-23	0	5	7	21
`\note`	-39	0	9	12	48
`\midinote`	21	60	69	72	108
`\freq`	27.5	261.6	440	523.2	4186

13.5 Mais palavras-chave: amplitude e legato

O novo exemplo introduz duas novas palavras-chave: `\amp` e `\legato`, que definem a amplitude dos eventos e a quantidade de legato entre as notas. Perceba como o código fica bem mais fácil de ler, graças a uma boa indentação e distribuição em várias linhas. Parênteses externos (no topo e embaixo) são usados para delimitar um bloco de código a ser executado de uma vez.

```
(
Pbind(
        \degree, Pseq([0, −1, 2, −3, 4, −3, 7, 11, 4, 2, 0, −3], 5),
        \dur, Pseq([0.2, 0.1, 0.1], inf),
        \amp, Pseq([0.7, 0.5, 0.3, 0.2], inf),
        \legato, 0.4
).play;
)
```

`Pbind` tem muitas destas palavras-chave pré-definidas e, com o tempo, você aprenderá mais delas. Por agora, vamos nos centrar em apenas algumas—uma para altura (com as opções de `\degree`, `\note`, `\midinote` ou `\freq`), uma para durações (`\dur`), uma para amplitude (`\amp`) e uma para legato (`\legato`). Durações estão em tempos (neste caso, 1 batida por segundo, que é o padrão); a amplitude deve estar entre 0 e 1 (0 = silêncio, 1 = muito alto); e o legato funciona melhor com valores entre 0.1 e 1 (se você não tem certeza o que o legato faz,

simplesmente experimente o exemplo acima com 0.1, depois 0.2, depois 0.3, até chegar no 1 e ouça os resultados).

Tome o último exemplo como um ponto de partida e crie novos `Pbind`s. Mude a melodia. Introduza novas listas de durações e amplitudes. Experimente usar `\freq` para alturas. Lembre-se, você sempre pode usar um número fixo para qualquer um destes parâmetros, se quiser. Por exemplo, se você quer que todas as notas da sua melodia tenham 0.2 segundos de duração, não há porque escrever `Pseq[0.2, 0.2, 0.2, 0.2...`, nem mesmo `Pseq([0.2], inf)`—simplesmente remova toda a estrutura do `Pseq` e escreva 0.2 no lugar.

13.6 Prand

`Prand` é um parente próximo do `Pseq`. Ele também aceita uma lista e um número de repetições. Mas em vez de ir tocando a lista na sequência, `Prand` *escolhe um item aleatório da lista a cada vez*. Experimente:

```
(
Pbind(
        \degree, Prand([2, 3, 4, 5, 6], inf),
        \dur, 0.15,
        \amp, 0.2,
        \legato, 0.1
).play;
)
```

Substitua `Prand` pelo `Pseq` e compare os resultados. Agora experimente usar `Prand` para durações, amplitudes e legato.

13.7 Pwhite

Outro membro popular da família Pattern é o `Pwhite`. É um gerador de números aleatórios de distribuição uniforme (o nome vem de "white noise", ruído branco). Por exemplo, `Pwhite(100, 500)` irá fornecer números aleatórios entre 100 e 500.

```
(
Pbind(
        \freq, Pwhite(100, 500),
        \dur, Prand([0.15, 0.25, 0.3], inf),
        \amp, 0.2,
        \legato, 0.3
).trace.play;
)
```

O exemplo acima também mostra outro truque útil: a mensagem `trace` logo antes de `play`. Isso imprime na Post window os valores selecionados em cada evento. Muito útil para corrigir problemas ou simplesmente para entender o que está acontecendo!

Preste atenção nas diferenças entre `Pwhite` e `Prand`: mesmo que ambos tenham a ver com aleatoriedade, eles aceitam argumentos distintos e fazem coisas diferentes. Dentro dos parênteses do `Pwhite` você só precisa fornecer um limite mínimo e máximo: `Pwhite(mínimo, máximo)`. Números aleatórios serão escolhidos no interior deste âmbito. `Prand`, por outro lado, aceita uma lista de itens (necessariamente entre colchetes) e um número de repetições: `Prand([lista, de, itens], repetições)`. Itens aleatórios serão escolhidos *desta lista*.

Explore ambos e confirme que você entendeu completamente a diferença.

DICA: Um `Pwhite` com dois números inteiros vai gerar somente números inteiros. Por exemplo, `Pwhite(100, 500)` vai gerar números como 145, 568, 700, mas não 145.6, 450.32, etc. Se você quer incluir números decimais no resultado, escreva `Pwhite(100, 500.0)`. Isso é muito útil para, digamos, amplitudes: se você escreve `Pwhite(0, 1)` vai obter apenas 0 ou 1, mas escreva `Pwhite(0, 1.0)` e você terá todos os resultados intermediários.

Tente as seguintes perguntas para testar seu novo conhecimento:

a) Qual a diferença entre os resultados de `Pwhite(0, 10)` e `Prand([0, 4, 1, 5, 9, 10, 2, 3], inf)`?

b) Se você precisa de um fluxo de números inteiros escolhidos aleatoriamente entre 0 e 100, você poderia usar um `Prand`?

c) Qual a diferença entre os resultados de `Pwhite(0, 3)` e `Prand([0, 1, 2, 3], inf)`? E se você escrever `Pwhite(0, 3.0)`?

d) Rode os exemplos abaixo. Usamos `\note` em vez de `\degree` para tocar uma escala de dó menor (que inclui teclas pretas). A lista `[0, 2, 3, 5, 7, 8, 11, 12]` tem oito números dentro dela, correspondendo às notas dó, ré, mi♭, fá, sol, lá♭, si, dó, mas quantos eventos cada exemplo realmente toca? Por quê?

```
// Pseq
(
Pbind(
        \note, Pseq([0, 2, 3, 5, 7, 8, 11, 12], 4),
        \dur, 0.15;
).play;
)
```

```
8
9  // Pseq
10 (
11 Pbind(
12         \note, Prand([0, 2, 3, 5, 7, 8, 11, 12], 4),
13         \dur, 0.15;
14 ).play;
15 )
16
17 // Pwhite
18 (
19 Pbind(
20         \note, Pseq([0, 2, 3, 5, 7, 8, 11, 12], 4),
21         \dur, Pwhite(0.15, 0.5);
22 ).play;
23 )
```

Respostas ao final deste tutorial.[2]

DICA: Um Pbind para de tocar quando o Pattern interno mais curto tiver terminado de tocar (conforme determinado pelo argumento **repeats** de cada Pattern interno).

13.8 Expandindo seu vocabulário de Patterns

A partir de agora, você já deve ser capaz de escrever **Pbind**s simples por conta própria. Você sabe especificar alturas, durações, amplitudes, valores de legato e você sabe como embutir outros Patterns (**Pseq**, **Prand**, **Pwhite**) para gerar mudanças interessantes de parâmetros.

Esta seção irá expandir um pouco seu vocabulário de Patterns. Os exemplos abaixo introduzem seis novos membros da família Pattern. Tente descobrir por você mesmo o que eles fazem. Use as seguintes estratégias:

- Escute a melodia resultante; descreva e analise o que você ouve;
- Olhe para o nome do Pattern: ele sugere algo? (por exemplo, `Pshuf` pode lembrá-lo da palavra "shuffle", embaralhar);
- Olhe para os argumentos (números) dentro do novo Pattern;
- Use `.trace.play` como vimos antes para observar os valores sendo impressos na Post window;
- Finalmente, confirme suas especulações consultando os arquivos de Ajuda (selecione o nome do Pattern e aperte [ctrl+D] para abrir o arquivo de Ajuda correspondente).

```
1  // Expandindo seu vocabulário de Patterns
2
3  // Pser
4  (
5  Pbind(
6          \note, Pser([0, 2, 3, 5, 7, 8, 11, 12], 11),
7          \dur, 0.15;
8  ).play;
9  )
10
11 // Pxrand
12 // Compare com Prand e escute a diferença
13 (
14 p = Pbind(
```

```
    \note, Pxrand([0, 2, 3, 5, 7, 8, 11, 12], inf),
        \dur, 0.15;
).play;
)

// Pshuf
(
p = Pbind(
    \note, Pshuf([0, 2, 3, 5, 7, 8, 11, 12], 6),
        \dur, 0.15;
).play;
)

// Pslide
// Aceita 4 argumentos: lista, repetições, comprimento, deslocamento
(
Pbind(
        \note, Pslide([0, 2, 3, 5, 7, 8, 11, 12], 7, 3, 1),
        \dur, 0.15;
).play;
)

// Pseries
// Aceita três argumentos: início, razão, comprimento
(
Pbind(
    \note, Pseries(0, 2, 15),
        \dur, 0.15;
).play;
)

// Pgeom
```

```
// Aceita três argumentos: início, razão, comprimento
(
Pbind(
        \note, Pseq([0, 2, 3, 5, 7, 8, 11, 12], inf),
        \dur, Pgeom(0.1, 1.1, 25);
).play;
)

// Pn
(
Pbind(
        \note, Pseq([0, Pn(2, 3), 3, Pn(5, 3), 7, Pn(8, 3), 11, 12], 1),
        \dur, 0.15;
).play;
)
```

Pratique usar estes Patterns—você pode fazer muitas coisas com eles. `Pbind`s são como receitas para partituras musicais, com a vantagem que você não está limitado a escrever sequências fixas de notas e ritmos: você pode descrever processos de parâmetros musicais em constante mudança (às vezes isto é chamado "composição algorítmica"). E isso é apenas um aspecto das capacidades poderosas da família Pattern.

No futuro, quando você sentir a necessidade de mais objetos Pattern, consulte o "Practical Guide to Patterns"de James Harkins, disponível nos arquivos de Ajuda do SC.*

*Também online em `http://doc.sccode.org/Tutorials/A-Practical-Guide/PG_01_Introduction.html`

14 Mais truques com Patterns

14.1 Acordes

Quer escrever acordes dentro de `Pbinds`? Escreva-os como listas (valores separados por vírgula entre colchetes):

```
(
Pbind(
        \note, Pseq([[0, 3, 7], [2, 5, 8], [3, 7, 10], [5, 8, 12]], 3),
        \dur, 0.15
).play;
)
// Mais um recurso legal: "strum" ("dedilhar")
(
Pbind(
        \note, Pseq([[-7, 3, 7, 10], [0, 3, 5, 8]], 2),
        \dur, 1,
        \legato, 0.4,
        \strum, 0.1 // experimente 0, 0.1, 0.2, etc
).play;
)
```

14.2 Escalas

Quando estiver usando **\degree** ("grau") para especificar alturas, você pode acrescentar uma linha com a palavra-chave **\scale** ("escala") para escolher a escala (nota: isso só funciona em conjunção com **\degree**, não com **\note**, **\midinote** ou **\freq**):

```
(
Pbind(
```

```
        \scale, Scale.harmonicMinor,
        \degree, Pseq([0, 1, 2, 3, 4, 5, 6, 7], 1),
        \dur, 0.15;
).play;
)

// Execute esta linha para ver uma lista de todas as escalas disponíveis:
Scale.directory;

// Se você precisa de uma nota cromática entre dois graus da escala, faça assim:
(
Pbind(
        \degree, Pseq([0, 1, 2, 3, 3.1, 4], 1),
).play;
)

// O 3.1 acima quer dizer um passo cromático sobre o grau 3 da escala (neste caso,
// fá# acima do fá). Note que quando você não especifica
// explicitamente uma escala \scale, Scale.major é subentendida.
```

14.3 Transposição

Use a palavra-chave **\ctranspose** para realizar uma transposição cromática. Isso funciona em conjunto com **\degree**, **\note** e **\midinote**, mas não com **\freq**.

```
(
Pbind(
        \note, Pser([0, 2, 3, 5, 7, 8, 11, 12], 11),
        \ctranspose, 12, // transpor oitava acima (= 12 semitons)
        \dur, 0.15;
).play;
```

```
)
```

14.4 Microtons

Como escrever microtons:

```
// Microtons com \note e \midinote:
Pbind(\note, Pseq([0, 0.5, 1, 1.5, 1.75, 2], 1)).play;
Pbind(\midinote, Pseq([60, 69, 68.5, 60.25, 70], 1)).play;
```

14.5 Andamento

Os valores que você fornece para a palavra-chave `\dur` de um Pbind estão em *número de tempos*, isto é, 1 significa um tempo, 0.5 significa meio tempo e assim por diante. A não ser que você especifique outra coisa, o tempo padrão é 60 BPM (batidas por minuto). Para tocar em um andamento diferente, simplesmente crie um novo TempoClock. Aqui está um `Pbind` tocando a 120 batidas por minuto (BPM):

```
(
Pbind(\degree, Pseq([0, 0.1, 1, 2, 3, 4, 5, 6, 7]),
        \dur, 1;
).play(TempoClock(120/60)); // 120 batidas em 60 segundos: 120 BPM
)
```

Por acaso, você percebeu que o `Pseq` recebeu um único argumento (a lista)? Onde está o valor `repeats` que sempre veio após a lista? Você pode ouvir que o exemplo toca a sequência apenas uma vez, mas por quê? Está é uma propriedade comum a todos os Patterns (e também de muitos outros objetos dentro do SuperCollider): se você omite um argumento, será utilizado um valor

padrão que já vem embutido. Neste caso, o `repeats` padrão para `Pseq` é 1. Lembra-se do seu primeiro `Pbind` ridiculamente simples? Era um mero `Pbind(\degree, 0).play` e ele somente sabia tocar uma nota. Você não forneceu nenhuma informação para duração, amplitude, legato, etc. Nestes casos, o `Pbind` simplesmente usa os valores padrão.

14.6 Pausas

Veja abaixo como escrever pausas. O número dentro de `Rest` é a duração da pausa em tempos. Pausas podem estar em qualquer lugar no Pbind, não somente na linha `\dur`.

```
(
Pbind(
        \degree, Pwhite(0, 10),
        \dur, Pseq([0.1, 0.1, 0.3, 0.6, Rest(0.3), 0.25], inf);
).play;
)
```

14.7 Tocando dois ou mais Pbinds juntos

Para disparar alguns Pbinds simultaneamente, simplesmente ponha todos eles em um único bloco de código:

```
( // parêntese abrindo o bloco
Pbind(
        \freq, Pn(Pseries(110, 111, 10)),
        \dur, 1/2,
        \legato, Pwhite(0.1, 1)
).play;

Pbind(
```

```
        \freq, Pn(Pseries(220, 222, 10)),
        \dur, 1/4,
        \legato, Pwhite(0.1, 1)
).play;

Pbind(
        \freq, Pn(Pseries(330, 333, 10)),
        \dur, 1/6,
        \legato, 0.1
).play;
) // parêntese fechando o bloco
```

Para tocar Pbinds de uma maneira temporalmente ordenada (em vez de simplesmente executá-los manualmente um após o outro), você pode usar `{ }.fork` [N.T.: "garfo", uma maneira abreviada de criar rotinas de programação]:

```
// Exemplo básico de fork. Observe o Post window:
(
{
        "uma coisa".postln;
        2.wait;
        "outra coisa".postln;
        1.5.wait;
        "uma última coisa".postln;
}.fork;
)
// Um exemplo mais interessante:
(
t = TempoClock(76/60);
{
        Pbind(
                \note, Pseq([[4, 11], [6, 9]], 32),
```

```
                \dur, 1/6,
                \amp, Pseq([0.05, 0.03], inf)
        ).play(t);

        2.wait;

        Pbind(
                \note, Pseq([[−25, −13, −1], [−20, −8, 4], \rest], 3),
                \dur, Pseq([1, 1, Rest(1)], inf),
                \amp, 0.1,
                \legato, Pseq([0.4, 0.7, \rest], inf)
        ).play(t);

        2.75.wait;

        Pbind(
                \note, Pseq([23, 21, 25, 23, 21, 20, 18, 16, 20, 21, 23, 21], inf),
                \dur, Pseq([0.25, 0.75, 0.25, 1.75, 0.125, 0.125, 0.80, 0.20, 0.125,
                    0.125, 1], 1),
                \amp, 0.1,
                \legato, 0.5
        ).play(t);
}.fork(t);
)
```

Para formas avançadas de tocar **Pbind**s simultaneamente e em sequência, confira **Ppar** e **Pspawner**. Para saber mais sobre **fork**, veja o arquivo de Ajuda de **Routine**.

14.8 Usando variáveis

Na seção anterior, "Expandindo seu vocabulário de Patterns", você percebeu quantas vezes teve que digitar a mesma lista de notas [0, 2, 3, 5, 7, 8, 11, 12] em vários `Pbind`s? Não é muito eficiente ficar copiando a mesma coisa à mão, certo? Em programação, sempre que você se vir fazendo a mesma tarefa repetidas vezes, é provavelmente hora de adotar uma estratégia mais eficiente para chegar ao mesmo objetivo. Neste caso, podemos usar variáveis. Como você deve se lembrar, variáveis permitem que você se refira a qualquer conjunto de dados de uma maneira flexível e concisa (releia a seção 12 se necessário). Aqui está um exemplo:

```
// Usando muito a mesma sequência de números? Guarde-a em uma variável:
c = [0, 2, 3, 5, 7, 8, 11, 12];

// Agora você pode simplesmente se referir a ela
Pbind(\note, Pseq(c, 1), \dur, 0.15).play;
Pbind(\note, Prand(c, 6), \dur, 0.15).play;
Pbind(\note, Pslide(c, 5, 3, 1), \dur, 0.15).play;
```

Outro exemplo para praticar o uso de variáveis: digamos você queira tocar dois `Pbind`s simultaneamente. Um deles toca uma escala maior ascendente e o outro toca uma escala maior descedente uma oitava acima. Ambos usam a mesma lista de durações. Eis um jeito de escrever isso:

```
~escala = [0, 1, 2, 3, 4, 5, 6, 7];
~durs = [0.4, 0.2, 0.2, 0.4, 0.8, 0.2, 0.2, 0.2];
(
Pbind(
        \degree, Pseq(~escala),
        \dur, Pseq(~durs)
).play;

```

```
9  Pbind(
10         \degree, Pseq(~escala.reverse + 7),
11         \dur, Pseq(~durs)
12 ).play;
13 )
```

Alguns truques interessantes: graças às variáveis, reutilizamos a mesma lista de graus de escala e durações para ambos os `Pbind`s. Quisemos que a segunda escala fosse descendente e uma oitava acima da primeira. Para obter isso, simplesmente utilizamos a mensagem `.reverse` para inverter a ordem da lista (digite `~scale.reverse` em um nova linha e execute para ver exatamente o que ela faz). Depois, adicionamos 7 para transpô-la uma oitava acima (teste isso também para ver o resultado).* Tocamos dois `Pbind`s ao mesmo tempo fechando-os no mesmo bloco de código.

Exercício: crie um `Pbind` adicional dentro do código acima, para que você ouça três vozes simultâneas. Use ambas as variáveis (`~scale` e `~durs`) de alguma maneira diferente—por exemplo, use um outro Pattern que não seja o Pseq; mude o intervalo de transposição; inverta e/ou multiplique as durações; etc.

15 Iniciando e parando Pbinds independentemente

Está é uma dúvida muito comum que surge com `Pbind`s, especialmente os que rodam para sempre com `inf`: como posso parar e iniciar `Pbind`s quando eu quiser? A resposta envolverá o uso de variáveis e veremos um exemplo completo em breve, mas antes de chegarmos lá, precisamos entender um pouco melhor o que acontece quando você toca um `Pbind`.

*Poderíamos também ter usado `\ctranspose, 12` para obter a mesma transposição.

15.1 Pbind como uma partitura musical

Você pode imaginar o `Pbind` como uma espécie de partitura musical: é uma receita para fazer sons, um conjunto de instruções para realizar uma passagem musical. Para que a partitura se torne música, você precisa entregá-la a um intérprete: alguém que vai ler a partitura e fazer sons com base nessas instruções. Vamos separar conceitualmente estes dois momentos: a definição da partitura e a performance da mesma.

```
// Defina a partitura
(
p = Pbind(
        \midinote, Pseq([57, 62, 64, 65, 67, 69], inf),
        \dur, 1/7
); // sem .play aqui!
)

// Peça que a partitura seja tocada
p.play;
```

A variável `p` no exemplo acima simplesmente guarda a partitura—repare que o `Pbind` não tem uma mensagem `.play` logo após o fechamento dos parênteses. Nenhum som é produzido neste momento. O segundo momento é quando você pede ao SuperCollider que toque a partitura: `p.play`. É aqui que foi criado o "músico"("tocador") que realiza a partitura em som.

Um erro comum agora seria tentar `p.stop`, na esperança de fazer com que o "músico"(tocador) pare de tocar. Tente isso e verifique por si mesmo que não funciona deste jeito. Você entenderá o porquê nos próximos parágrafos.

15.2 EventStreamPlayer

Limpe a Post window com [ctrl+shift+P] (não é absolutamente necessário, mas por que não?) e rode `p.play` novamente. Olhe para a Post window e você verá que o resultado é algo chamado `EventStreamPlayer` ("Tocador de fluxo de eventos"). Toda vez que você chama um `.play` em um `Pbind`, o SuperCollider cria um tocador que realiza aquela ação: o `EventStreamPlayer` é isso. É como ter um pianista se materializando na sua frente toda vez que você disser "Eu quero que esta partitura seja tocada agora". Legal, né?

Bem, sim, exceto pelo fato de que depois que este tocador virtual anônimo aparece e começa seu trabalho, você não tem mais como se dirigir a ele—ele não tem nome. Em termos ligeiramente mais técnicos, você criou um objeto, mas você não tem como se referir a este objeto depois. Talvez neste ponto você já possa entender porque `p.stop` não funciona: é como tentar falar com a partitura em vez de falar com o tocador. A partitura (o `Pbind` armazenado na variável `p`) não sabe nada sobre começar ou parar: ela é só uma receita. O *tocador* é quem sabe ler a partitura, tocá-la, parar, "por favor volte pro começo", etc. Em outras palavras, você tem que falar com o `EventStreamPlayer`. Tudo que você precisa fazer é dar um nome a ele, ou seja, armanzená-lo em uma variável:

```
1 // Experimente estas linhas, uma por uma:
2 ~meuTocador = p.play;
3 ~meuTocador.stop;
4 ~meuTocador.resume;
5 ~meuTocador.stop.reset;
6 ~meuTocador.start;
7 ~meuTocador.stop;
```

Em resumo: chamando `.play` em um `Pbind` gera um `EventStreamPlayer`; e armazenando seu `EventStreamPlayer`s em variáveis permite que você os acesse mais tarde para iniciar e parar Patterns individualmente (sem precisar usar [ctrl+.], que interrompe tudo de uma vez).

15.3 Exemplo

Aqui temos um exemplo mais complexo para finalizar esta seção. A melodia principal é emprestada do "Álbum para a Juventude"de Tchaikovsky e uma melodia mais grave é adicionada em contraponto. A figura 3 mostra a passagem em notação musical.

```
// Defina a partitura
(
var minhasDurs = Pseq([Pn(1, 5), 3, Pn(1, 5), 3, Pn(1, 6), 1/2, 1/2, 1, 1, 3, 1, 3],
     inf) * 0.4;
~melodiaSuperior = Pbind(
        \midinote, Pseq([69, 74, 76, 77, 79, 81, Pseq([81, 79, 81, 82, 79, 81], 2),
            82, 81, 79, 77, 76, 74, 74], inf),
        \dur, minhasDurs
);
~melodiaInferior = Pbind(
        \midinote, Pseq([57, 62, 61, 60, 59, 58, 57, 55, 53, 52, 50, 49, 50, 52, 50,
             55, 53, 52, 53, 55, 57, 58, 61, 62, 62], inf),
        \dur, minhasDurs
);
)
// Toque as duas juntas:
(
~tocador1 = ~melodiaSuperior.play;
~tocador2 = ~melodiaInferior.play;
)
// Pare-os separadamente:
~tocador1.stop;
~tocador2.stop;
// Outras mensagens disponíveis
~tocador1.resume; // retomar
~tocador1.reset; // voltar do início
```

```
~tocador1.play;
~tocador1.start; // mesmo que .play
```

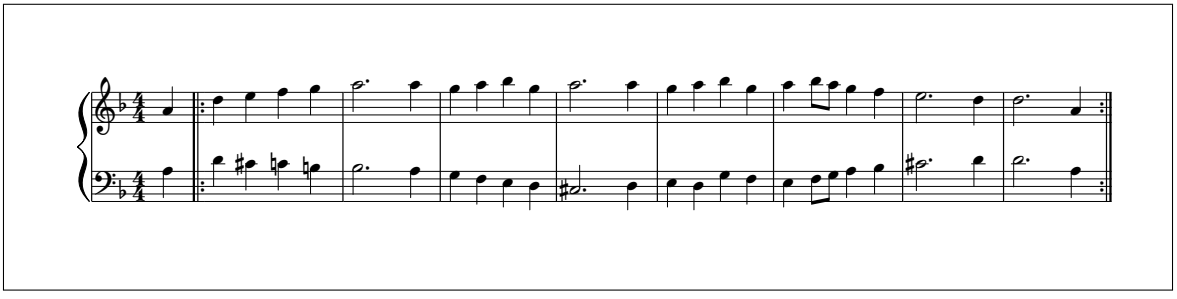

Figura 3: `Pbind` contraponto com uma melodia de Tchaikovsky

Primeiro, repare no uso de variáveis. Uma delas, `minhasDurs`, é uma variável local. Dá pra saber que é uma variável local porque não começa com um til (∼) e é declarada no início com a palavra-chave específica `var`. Esta variável contém um `Pseq` que será usado como `\dur` em ambos os `Pbind`s. `minhasDurs` é necessária somente no momento de definir a partitura, então faz sentido usar uma variável local para isso (embora uma variável global funcionasse igualmente bem). As outras variáveis que você vê no exemplo são variáveis globais—uma vez declaradas, são válidas em qualquer lugar nos seus patches de SuperCollider.

Segundo, repare na separação ente partitura e tocadores, como discutimos anteriormente. Quando os `Pbind`s são definidos, eles não são tocados no mesmo momento—não há `.play` imediatamente depois de fechar os parênteses. Depois que você roda o primeiro bloco de código, tudo o que você tem são duas definições de `Pbind` armazenadas nas variáveis `∼melodiaSuperior` e `∼melodiaInferior`. Elas ainda não fazem som—são apenas partituras. A linha `∼tocador1 = ∼melodiaSuperior.play` cria um `EventStreamPlayer` para cumprir a tarefa de tocar a melodia

superior e a este tocador é dado o nome ~tocador1. A mesma ideia vale para o ~tocador2. Graças a isso, podemos falar com cada tocador e pedi-lo para parar, iniciar, retomar, etc.

Mesmo correndo o risco de sermos chatos, vamos reiterar uma última vez:

- Um `Pbind` é só uma receita para fazer som, como uma partitura musical;
- Quando você chama a mensagem `play` em um `Pbind`, um objeto `EventStreamPlayer` é criado;
- Se você armazena este `EventStreamPlayer` em uma variável, você pode acessá-lo mais tarde para usar comandos como `stop` e `resume`.

Parte III
MAIS DETALHES SOBRE A LINGUAGEM

16 Objetos, classes, mensagens, argumentos

SuperCollider é uma linguagem de programação orientada a objetos, como Java ou C++. Está além do escopo deste tutorial explicar o que isso significa, então deixaremos você pesquisar isso na internet se tiver curiosidade. Aqui vamos apenas explicar alguns conceitos básicos que você precisa saber para entender melhor esta nova linguagem que você está aprendendo.

Tudo no SuperCollider é um *objeto*. Mesmo simples números são objetos no SC. Diferentes objetos se comportam de diferentes maneiras e armazenam diferentes tipos de informação. Você pode solicitar uma informação ou ação de um objeto enviando uma *mensagem* para ele. Quando você escreve algo como `2.squared`, a mensagem `squared` está sendo enviada para o objeto `2`, que a recebe (por isso vamos chamá-lo de "objeto recebedor", tradução do inglês "receiver object"). O ponto entre o objeto e a mensagem faz a conexão entre os dois. A propósito, mensagens também são chamadas *métodos*.

Objetos são especificados hierarquicamente em *classes*. O SuperCollider vem com uma imensa coleção de classes pré-definidas, cada uma com seu próprio conjunto de métodos.

Eis uma analogia pra ajudar a entender isso. Imaginemos que há uma classe abstrata de objetos chamada `Animal`. A classe Animal define alguns métodos (mensagens) comuns a todos os animais. Métodos como `mover`, `comer`, `dormir` fariam o Animal realizar uma ação específica. Daí poderíamos imaginar duas subclasses de Animal: uma chamada `Doméstico` outra chamada `Selvagem`. Cada uma destas subclasses poderia ter ainda mais subclasses derivadas destas (como `Cachorro` e `Gato`, derivados de `Doméstico`). Subclasses herdam todos os métodos de suas classes-mãe e implementam novos métodos próprios para atributos especializados. Por exemplo, tanto

o objeto Cachorro quanto Gato alegremente responderiam à mensagem `.comer`, herdada da classe Animal. `Cachorro.nome` e `Gato.nome` retornariam o nome do bicho: `nome` poderia ser um método comum a todos os objetos derivados da classe Doméstico. `Cachorro` tem um método `latir`, então você pode chamar `Cachorro.latir` e ele saberá o que fazer. `Gato.latir` retornaria uma mensagem de erro: `ERRO: Mensagem 'latir' não entendida.`

Em todos estes exemplos hipotéticos, as palavras começando com uma letra maiúscula são *classes* que representam *objetos*. As palavras e minúsculas depois do ponto são *mensagens* (ou *métodos*) que estão sendo enviadas para estes objetos. Mandar uma mensagem para um objeto sempre retorna algum tipo de informação. Finalmente, mensagens às vezes aceitam (ou mesmo exigem) *argumentos*. Argumentos são os dados que vêm entre parênteses logo depois de uma mensagem. Em `Gato.comer("sardinhas", 2)`, a mensagem `comer` está sendo enviada para `Gato` com algumas informações bem específicas: o que comer e em que quantidade. Às vezes você verá argumentos declarados explicitamente dentro de parênteses (palavras-chave terminando com dois pontos). Isso é muitas vezes útil para quem lê o código lembrar rapidamente a que o argumento se refere. `Cachorro.latir(volume: 10)` é mais autoexplicativo que apenas `Cachorro.latir(10)`.

OK—já basta desta explicação rapidinha sobre programação orientada a objetos. Vamos tentar alguns exemplos que você possa de fato rodar no SuperCollider. Rode uma linha após a outra e veja se você consegue identificar a mensagem, o objeto recebedor e o argumento (se houver). A estrutura básica é `Recebedor.mensagem(argumentos)`. Respostas ao final do livro.[3]

```
1 [1, 2, 3, "uau"].reverse;
2 "alô".dup(4);
3 3.1415.round(0.1); // note que o primeiro ponto é a separação decimal de 3.1415 [N.T
      .: Atenção: o SC separa casas decimais com ponto! Vírgulas têm outros usos.]
4 100.rand; // rode esta linha diversas vezes;
5 // Encadear mensagens é divertido:
6 100.0.rand.round(0.01).dup(4);
```

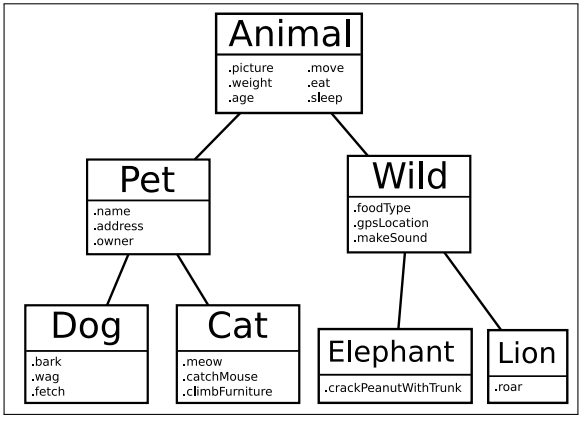

Figura 4: Hierarquia de classes hipotética.

17 Notação de objeto recebedor, notação funcional

Há mais de uma maneira de escrever expressões no SuperCollider. A que vimos acima é chamada *notação de objeto recebedor* ("receiver notation"): `100.rand`, na qual um ponto conecta o Objeto `100` à mensagem `rand`. Alternativamente, a mesmíssima coisa pode ser escrita assim: `rand(100)`. Isso é chamado *notação funcional* ("functional notation").

Você pode escrever das duas formas. Eis como isso funciona quando uma mensagem recebe dois ou mais argumentos.

```
1 5.dup(20);  // notação de objeto recebedor
2 dup(5, 20); // mesma coisa em notação funcional
3
4 3.1415.round(0.1); // notação de objeto recebedor
```

```
5 round(3.1415, 0.1); // notação funcional
```

Nos exemplos acima, você pode ler `dup(5, 20)` como "faça duplicatas do número 5 vinte vezes"e `round(3.1415, 0.1)` como "arredonde o número 3.1415 para uma casa decimal". Por sua vez, as versões em notação de objeto recebedor podem ser lidas como "Número 5, faça duplicatas de si mesmo vinte vezes!"(para `5.dup(20)`) e "Número 3.1415, arredonde-se para uma casa decimal!" (para `3.1415.round(0.1)`).

Resumindo: `Recebedor.mensagem(argumento)` é equivalente a `mensagem(Recebedor, argumento)`.

Escolher um estilo de escrita em vez do outro é uma questão de preferência pessoal e convenção. Às vezes um método pode ser mais claro que o outro. Qualquer que seja o estilo que você acabe preferindo (e tudo bem misturá-los), o importante é ser consistente. Uma convenção que é muito difundida entre usuários de SuperCollider é que classes (palavras que começam com letras maiúsculas) são quase sempre escritas como `Recebedor.mensagem(argumento)`. Por exemplo, você sempre verá `SinOsc.ar(440)`, mas quase nunca verá `ar(SinOsc, 440)`, embora ambas estejam corretas.

Exercício: reescreva a seguinte sentença usando apenas notação funcional:

`100.0.rand.round(0.01).dup(4);`

Solução ao final.[4]

18 Aninhamento

A solução do último exercício levou você a aninhar coisas uma dentro da outra, isso é, você colocou o rand dentro do round que por sua vez foi pra dentro do dup. David Cottle tem uma explicação excelente para aninhamento no The SuperCollider Book, então simplesmente o citaremos aqui.*

*Cottle, D. "Beginner's Tutorial." The SuperCollider Book, MIT Press, 2011, pp. 8-9.

Para explicar melhor a ideia de aninhamento, considere um exemplo hipotético no qual o SC vai preparar sua refeição. Para fazer isso você pode usar uma mensagem `servir`. *Os argumentos podem ser salada, prato principal e sobremesa. Mas apenas dizer* `servir(alface, peixe, banana)` *talvez não produza o resultado que você quer. Pra ter certeza que a comida seja bem feita, você pode explicar melhor os argumentos, substituindo-os por uma mensagem aninhada e alguns argumentos mais específicos.*

```
servir(misturar(alface, tomate, queijo), assar(peixe, 400, 20), bater(banana, sorvete))
```

o SC então não apenas servirá alface, peixe e banana, mas uma salada mista com alface, tomate e queijo; um peixe assado; e um sundae de banana. Estes comandos internos podem ser ainda mais explicados, aninhando uma mensagem(arg) para cada ingrediente: alface, tomate, queijo e assim por diante. Cada mensagem interna produz um resultado que por sua vez é usado como argumento pela mensagem exterior.

```
1  // Pseudo-código para fazer o jantar:
2  servir(
3          misturar(
4                  lavar(alface, água, 10),
5                  picar(tomate, pequeno),
6                  salpicar(escolher([gorgonzola, feta, gouda]))
7          ),
8          assar(pescar(lagoa, anzol, vara), 200, 20),
9          misturar(
10                 fatiar(descascar(banana), 20),
11                 cozinhar(misturar(leite, açúcar, amido), 200, 10)
12         )
13 );
```

Quando o aninhamento tem diversos níveis, podemos usar novas linhas e indentações para uma maior clareza. Algumas mensagens e argumentos podem permanecer na mesma linha, enquanto outras mensagens e argumentos podem distribuídos um em cada linha—o que quer seja mais claro. Cada nível de indentação deve indicar um nível de aninhamento. (Note que você pode ter qualquer quantidade de espaço em branco—novas linhas, tabulações e espaços—entre trechos de código.)

[No exemplo do jantar,] *agora pede-se ao programa de refeições que ele lave a alface em água por 10 minutos e pique o tomate em pequenos pedaços antes de misturá-los na travessa da salada e salpicá-los com queijo. Você também especificou onde pegar o peixe e pediu para assá-lo a 200 graus por 20 minutos antes de servir, e assim por diante. Para "ler"este estilo de código, você começa da mensagem aninhada mais interna e vai seguindo para fora camada por camada. Aqui está um exemplo alinhado de maneira a mostrar como a mensagem mais interna é aninhada dentro das outras mensagens.*

```
                exprand(1.0, 1000.0);
           dup({exprand(1.0, 1000.0)}, 100);
      sort(dup({exprand(1.0, 1000.0)}, 100));
round(sort(dup({exprand(1.0, 1000.0)}, 100)), 0.01);
```

O código abaixo é um outro exemplo de aninhamento. Responda às perguntas que se seguem. Você não precisa explicar o que os números estão fazendo—a tarefa é simplesmente idenftificar os argumentos em cada camada de aninhamento. (Este exemplo e as questões do exercício também são emprestadas e ligeiramente modificadas do tutorial do Cottle.)

```
// Aninhamento e indentação apropriada
(
{
```

```
        CombN.ar(
                SinOsc.ar(
                        midicps(
                                LFNoise1.ar(3, 24,
                                        LFSaw.ar([5, 5.123], 0, 3, 80)
                                )
                        ),
                        0, 0.4
                ),
                1, 0.3, 2)
}.play;
)
```

a) Qual número é o segundo argumento do `LFNoise1.ar`?

b) Qual o primeiro argumento do `LFSaw.ar`?

c) Qual o terceiro argumento do `LFNoise1.ar`?

d) O método `midicps` tem quantos argumentos?

e) Qual o terceiro argumento do `SinOsc.ar`?

f) Qual o segundo e terceiro argumentos do `CombN.ar`?

Confira as respostas ao final do livro.[5]

DICA: Se por qualquer motivo, seu código perdeu a indentação apropriada, simplesmente selecione tudo e vá para o menu Edit→Autoindent Line or Region ("Autoindentar Linha ou Região") e isto será consertado.

19 Fechamentos

Há quatro tipos de fechamento: `(parênteses)`, `[colchetes]`, `{chaves}` e `"aspas duplas"`.

Cada um que você abre, deve ser fechado mais adiante. Isso é chamado "balancear", ou seja, manter devida correspondência entre os pares de fechamento em todo o seu código.

O IDE do SuperCollider automaticamente indica o fechamento de parênteses (também de colchetes e chaves) quando você fecha um par— eles são destacados em vermelho. Se você clicar um parêntese que não tem um par de abertura/fechamento, você verá uma seleção em vermelho escuro indicando que algo está faltando. O balanceamento é uma maneira rápida de selecionar uma grande seção de código para ser executado, deletado ou para operações de copiar/colar. Você pode fazer um clique duplo em um parêntese de abertura ou fechamento para selecionar tudo o que está contido neles (o mesmo vale para colchetes e chaves).

19.1 Aspas duplas

Aspas duplas são usadas para definir uma sequência de caracteres (incluindo espaços) como uma coisa única. Estas são chamadas Strings ("cadeias"). Aspas simples criam Símbolos ("Symbols"), que são ligeiramente diferentes de Strings. Símbolos também podem ser criados com uma barra invertida imediatamente antes do texto. Portanto, `'queSimbolo'` and `\queSimbolo` são equivalentes.

```
1 "Isto aqui é uma string";
2 'simboloLegalDemais';
```

19.2 Parênteses

Parênteses podem ser usados para:

- englobar listas de argumentos: `rrand(0, 10);`
- forçar precedência: `5 + (10 * 4);`
- criar blocos de código (múltiplas linhas de código para serem rodadas simultaneamente).

19.3 Colchetes

Colchetes definem uma coleção de itens, como `[1, 2, 3, 4, "oi"]`. Estas são normalmente chamadas Arrays. Um array pode conter qualquer coisa: números, strings, funções, Patterns, etc. Arrays entendem mensagens como `reverse` ("inverter"), `scramble` ("embaralhar"), `mirror` ("espelhar"), `choose` ("escolher"), para dar alguns exemplos. Você também pode fazer operações matemáticas em arrays.

```
1 [1, 2, 3, 4, "oi"].scramble;
2 [1, 2, 3, 4, "oi"].mirror;
3 [1, 2, 3, 4].reverse + 10;
4 // converter notas MIDI para frequências em Hz
5 [60, 62, 64, 65, 67, 69, 71].midicps.round(0.1);
```

Mais sobre arrays em breve na seção 22.

19.4 Chaves

Chaves definem funções. Funções encapsulam algum tipo de operação ou tarefa que será provavelmente reutilizada múltiplas vezes, possivelmente retornando diferentes resultados a cada vez. O exemplo abaixo é do The SuperCollider Book:

```
1 exprand(1, 1000.0);
2 {exprand(1, 1000.0)};
```

David Cottle nos explica passo a passo esse exemplo: *"a primeira linha seleciona um número aleatório, que é mostrado na Post window. A segunda linha imprime um resultado bastante diferente: uma função. O que essa função faz? Ela seleciona um número aleatório. Como pode esta diferença afetar o código? Considere as linhas abaixo. A primeira escolhe um número aleatório e o duplica. A segunda executa cinco vezes a função-seletora-de-números-aleatórios e coleta os resultados em um array."**

```
1 rand(1000.0).dup(5);  // seleciona um número e o duplica
2 {rand(1000.0)}.dup(5);  // duplica a função de selecionar um número
3 {rand(1000.0)}.dup(5).round(0.1); // tudo o que foi feito acima, depois arredondando
      os valores
4 // um resultado semelhante a:
5 [rand(1000.0), rand(1000.0), rand(1000.0), rand(1000.0), rand(1000.0)]
```

Mais sobre funções em breve. Por ora, aqui está um resumo de todos os fechamentos possíveis:

Coleções `[lista, de, itens]`

Funções `{ operações a serem executadas }`

Strings `"palavras entre aspas"`

Símbolos `'aspasSimples'` ou precedidas de uma `\barraInvertida`

20 Condicionais: if/else e case

Se estiver chovendo, saio com um guarda-chuva. Se estiver sol, saio com meus óculos escuros. Nosso dia-a-dia está repleto desse tipo de tomada de decisão. Em programação, estes são os

*Cottle, D. "Beginner's Tutorial." The SuperCollider Book, MIT Press, 2011, p. 13.

momentos em que o seu código tem de testar alguma condição e realizar ações diferentes dependendo do resultado do teste (verdadeiro ou falso). Há muitos tipos de estruturas condicionais. Vamos dar uma olhada em dois casos simples: *if/else* ("se/senão") e *case* ("no caso de").

A sintaxe de um if/else no SC é: `if(condition, {true action}, {false action})`. A condição é um teste booleano, ou seja, precisa retornar um `true` ("verdadeiro") ou `false` ("falso"). Se o teste retorna verdadeiro, a primeira função é executada. Se não for verdadeiro, roda-se a segunda. Experimente:

```
1 // if / else
2 if(100 > 50, { "muito verdadeiro".postln }, { "muito falso".postln });
```

A tabela abaixo, emprestada do The SuperCollider Book*, apresenta alguns operadores booleanos comuns que podem ser usados. Note a distinção entre um sinal de igual simples (`x = 10`) e o sinal de igual duplo (`x == 10`). O simples significa "*atribua 10 à variável x*,"ao passo que o duplo significa "*é x igual a 10?*"Digite e rode alguns exemplos da coluna de verdadeiro ou falso e você verá parecerem os resultados `true` ou `false` aparecerem na Post window.

*Cottle, D. "Beginner's Tutorial."The SuperCollider Book, MIT Press, 2011, p. 33

Símbolo	Significado	Exemplo verdadeiro	Exemplo falso
`==`	igual a?	`10 == 10`	`10 == 99`
`!=`	diferente de?	`10 != 99`	`10 != 10`
`>`	maior que?	`10 > 5`	`10 > 99`
`<`	menor que?	`10 < 99`	`10 < 5`
`>=`	maior ou igual a?	`10 >= 10`, `10 >= 3`	`10 >= 99`
`<=`	menor ou igual a?	`10 <= 99`, `10 <= 10`	`10 <= 9`
`odd`	é ímpar?	`15.odd`	`16.odd`
`even`	é par?	`22.even`	`21.even`
`isInteger`	é um número inteiro?	`3.isInteger`	`3.1415.isInteger`
`isFloat`	é um número decimal?	`3.1415.isFloat`	`3.isFloat`
`and`	ambas as condições	`11.odd.and(12.even)`	`11.odd.and(13.even)`
`or`	uma das condições	`or(1.odd, 1.even)`	`or(2.odd, 1.even)`

As últimas duas linhas (`and`, `or`) mostram como escrever as expressões mais longas tanto em notação de objeto recebedor quanto em notação funcional.

Outra estrutura útil é `case`. Ela funciona definindo pares de funções a serem executadas em ordem até que um dos testes retorne verdadeiro:

```
case
{teste1} {ação1}
{teste2} {ação2}
{teste3} {ação3}
...
{testeN} {açãoN};
```

A expressão dentro de cada teste tem de retornar ou `true` ou `false`. Se o teste1 retorna falso, o programa ignora a ação1 e segue para o teste2. Se teste2 for um falso de novo, ação2

também é ignorada e seguimos para o teste3. Se este for verdadeiro, então a ação3 é executada e o `case` se encerra ali (mais nenhum teste ou ação é executado). Note que não há vírgulas entre as funções. Simplesmente use um ponto-e-vírgula no fim para marcar o final do enunciado `case`.

```
1  // case
2  (
3  ~num = -2;
4
5  case
6  {~num == 0} {"UAU".postln}
7  {~num == 1} {"UM!".postln}
8  {~num < 0} {"número negativo!".postln}
9  {true} {"em último caso".postln};
10 )
```

Tente modificar o código acima para obter todos os resultados possíveis. Perceba o truque útil (e opcional) na última linha de `case` no exemplo acima: como `true` sempre retorna verdadeiro, pode-se definir uma ação para acontecer "em último caso"que vai sempre ocorrer, mesmo no caso de todas as condições anteriores serem falsas.

Para saber mais, verifique o arquivo de ajuda de Control Structures ("Estruturas de Controle").

21 Funções

Quando você se pegar repetindo a mesma tarefa diversas vezes, talvez seja um bom momento para criar uma função reutilizável. Uma função, como você aprendeu na seção de Fechamentos, é algo escrito entre chaves. David Touretzky introduz a ideia de função da seguinte forma: "pense na função como uma caixa através da qual os dados passam. Essa função opera sobre os dados

de alguma forma e o resultado é o que passa para fora."*

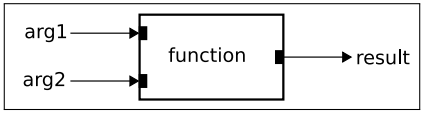

Figura 5: Ideia geral de uma função.

A primeira linha no exemplo abaixo define a função, atribuindo-a à variável `f`. A segunda linha coloca a função para trabalhar.

```
1 f = { 2 + 2 }; // define a função
2 f.value; // põe a função para trabalhar
```

A função acima não é lá muito útil, pois sabe fazer uma coisa só (somar 2 e 2). Normalmente a gente quer definir funções que possam dar diferentes resultados dependendo dos argumentos de entrada fornecidos. Nós usamos a palavra-chave **`arg`** para especificar as entradas que uma função vai aceitar. O exemplo abaixo é mais parecido com o desenho da figura 5.

```
1 f = {arg a, b; ["a mais b", a+b, "a vezes b", a*b].postln}; // define função
2 f.value(3, 7); // forneça dois números quaisquer como argumentos para a função
3 f.value(10, 14);
4
5 // Compare:
6 ~aleaTonto = rrand(0, 10); // não é uma função
7 ~aleaTonto.value; // execute diversas vezes
8 ~aleaTonto2 = {rrand(0, 10)}; // é uma função
```

*Touretzky, David. COMMON LISP: A Gentle Introduction to Symbolic Computation. The Benjamin/-Cummings Publishing Company, Inc, 1990, p. 1. Este é o livro que inspirou o título original em inglês deste tutorial.

```
~aleaTonto2.value; // execute diversas vezes
```

Como um último exemplo, aqui está uma função muito útil.

```
// Use esta função para decidir como passar seus dias de verão

(
~oQueFazer = {
var hoje, nomeDoDia, acoes;
        hoje = Date.getDate.dayOfWeek;
        nomeDoDia =
        case
        {hoje==0} {"Domingo"}
        {hoje==1} {"Segunda"}
        {hoje==2} {"Terça"}
        {hoje==3} {"Quarta"}
        {hoje==4} {"Quinta"}
        {hoje==5} {"Sexta"}
        {hoje==6} {"Sábado"};
        acoes = ["jogar bumerangue", "queda de braço", "subir escadas", "jogar
            xadrez", "basquete subaquático", "arremesso de ervilhas", "longa soneca"
            ];
        "Ah, " ++ nomeDoDia ++ "...! " ++ "Que ótimo dia para " ++ acoes.choose;
};
)

// Execute pela manhã
~oQueFazer.value;
```

DICA: Outra notação comum para declarar argumentos no início de uma Função é: `f = {|a, b| a + b}`. Isso é equivalente a `f = {arg a, b; a + b}`

22 Divirta-se com Arrays

Arrays [N.T.: em computação, o termo "array"pode ser traduzido como "vetor", mas é comum mantê-lo no original] são o tipo mais comum de coleção no SuperCollider. Toda vez que você escreve uma coleção de itens entre colchetes, como `[0, 1, 2]`, cria uma instância da classe `Array`. Frequentemente, você se verá manipulando arrays de diversas formas. Aqui está uma pequena seleção de métodos interessantes que arrays entendem:

```
1  // Criar um array
2  a = [10, 11, 12, 13, 14, 15, 16, 17];
3
4  a.reverse; // inverter
5  a.scramble; // embaralhar
6  a.choose; // escolher um elemento ao acaso
7  a.size; // retorna o tamanho do array
8  a.at(0); // recupera um item na posição especificada
9  a[0]; // o mesmo que acima
10 a.wrapAt(9); // recupera item na posição especificada, relendo do início se > a.size
11 ["wow", 99] ++ a; // concatena dois arrays em uma novo
12 a ++ \oi; // um Símbolo é entendido como um único caracter
13 a ++ 'oi'; // o mesmo que acima
14 a ++ "oi"; // uma cadeia ("String") é entendida como uma coleção de caracteres
15 a.add(44); // cria um novo array adicionando o novo elemento ao final
16 a.insert(5, "uau"); // insere "uau" na posição 5, empurra itens seguintes para a
       frente (retorna um novo array)
17 a; // rode isso e veja que nenhuma das operações acima mudou o array original
18 a.put(2, "oops"); // colocar "oops" no índice 2 (destrutivo; volte a rodar a linha
       acima para verificar)
19 a.permute(3); // permutar: item na posição 3 vai para a posição zero, e vice-versa
20 a.mirror; // faz um palíndromo
21 a.powerset; // retorna todas as possíveis combinações dos elementos do array
```

Você pode fazer contas com arrays:

```
[1, 2, 3, 4, 5] + 10;
[1, 2, 3, 4, 5] * 10;
([1, 2, 3, 4, 5] / 7).round(0.01); // note o uso de parênteses para precedência
x = 11; y = 12; // experimente algumas variáveis
[x, y, 9] * 100;
// mas garanta que você só faça contas com números de verdade
[1, 2, 3, 4, "oops", 11] + 10; // resultado estranho
```

22.1 Criando novos Arrays

Aqui há algumas maneiras de usar a classe `Array` para criar novas coleções:

```
// Progressão aritmética (definindo tamanho, início e razão da progressão)
Array.series(size: 6, start: 10, step: 3);
// Progressão geométrica (definindo tamanho, início e razão da progressão)
Array.geom(size: 10, start: 1, grow: 2);
// Compare as duas:
Array.series(7, 100, -10); // 7 itens; começando em 100, passo de -10
Array.geom(7, 100, 0.9); // 7 itens; começando em 100; multiplicar por 0.9 a cada
    vez
// Conheça o método .fill
Array.fill(10, "igual");
// Compare:
Array.fill(10, rrand(1, 10));
Array.fill(10, {rrand(1, 10)}); // a função é reexecutada 10 vezes
// A função definida como segundo arg do método .fill pode receber um contador como
    argumento padrão.
// O nome do argumento pode ser o que você quiser.
Array.fill(10, {arg contador; contador * 10});
// Por exemplo, gerando uma lista de frequências harmônicas:
```

```
Array.fill(10, {arg uau; uau+1 * 440});
// O método .newClear
a = Array.newClear(7); // cria um array vazio do tamanho desejado
a[3] = "uau"; // mesmo que a.put(3, "uau")
```

22.2 Aquele ponto de exclamação esquisito

É só uma questão de tempo até que você veja algo como `30!4` no código de alguém. Essa notação abreviada simplesmente cria uma array contendo o mesmo item um determinado número de vezes:

```
// Notação abreviada:
30!4;
"alô" ! 10;
// Dá o mesmo resultado que o seguinte:
30.dup(4);
"alô".dup(10);
// ou
Array.fill(4, 30);
Array.fill(10, "alô");
```

22.3 Os dois pontos entre parênteses

Aqui está mais uma sintaxe abreviada muito usada para criar arrays.

```
// Que diabo é isso?
(50..79);
// É um atalho para gerar um array com uma progressão numérica aritmética.
// O atalho acima tem o mesmo resultado que:
series(50, 51, 79);
// ou
```

```
Array.series(30, 50, 1);
// Para um passo (razão) diferente de 1, você pode fazer o seguinte:
(50, 53 .. 79); // de 3 em 3
// Mesmo resultado que:
series(50, 53, 79);
Array.series(10, 50, 3);
```

Note que cada comando implica um jeito de pensar ligeiramente diferente. O `(50..79)` permite que você pense assim: "*me dê um array de 50 a 79.*"Você não precisa necessariamente pensar a quantidade de itens que a lista vai conter. Por outro lado, **`Array.series`** permite pensar: "*me dê um array com 30 itens no total, começando a partir do 50.*"Você não precisa necessariamente pensar qual será o último número da série.

Também note que o atalho usa parênteses, não colchetes. O array resultante, naturalmente, virá entre colchetes.

22.4 Como "fazer"um Array

Muitas vezes você vai precisar realizar alguma operação com cada um dos itens de uma coleção. Você pode usar o método **`do`** ("faça") para isso:

```
~minhasFreqs = Array.fill(10, {rrand(440, 880)});

//Agora vamos fazer uma ação simples com cada um dos itens da lista:
~minhasFreqs.do({arg item, contador; ("Item " ++ contador ++ " é " ++ item ++ " Hz.
    A nota MIDI mais próxima é " ++ item.cpsmidi.round).postln});

// Se você não precisa do contador, use apenas um argumento:
~minhasFreqs.do({arg item; item.squared.postln});

// Claro que algo simples como o exemplo anterior poderia ser feito assim:
```

```
10 ~minhasFreqs.squared;
```

Em resumo: quando você processa um array com "do", você fornece uma função. A mensagem `"do"` vai cuidar de executar a função uma vez para cada item constante do array. A função pode receber dois argumentos por definição: o item da vez propriamente dito, e um contador que vai registrando o número de iterações já realizadas. Os nomes destes argumentos podem ser o que você quiser, mas estarão sempre nesta ordem: item, contador.

Veja também o método `collect`, que é muito similar ao `do`, mas retorna uma nova coleção com todos os resultados intermediários.

23 Obtendo Ajuda

Aprenda a usar bem os arquivos da Ajuda ("Help"). Muitas vezes, ao final de cada página da Ajuda, há exemplos úteis sobre o tópico em questão. Role a página para baixo para conferi-los, mesmo que (ou especialmente se) você não tenha entendido completamente as explicações do texto. Você pode rodar os exemplos diretamente de página de Ajuda do SuperCollider, ou você pode copiar e colar o código em uma nova janela para fuçar e experimentar mais com ele.

Selecione qualquer classe ou método válidos no seu código de SuperCollider (um duplo-clique na palavra irá selecioná-la) e aperte [ctrl+D] para abrir o arquivo de Help correspondente. Se você selecionar o nome de uma classe (por exemplo, `MouseX`), você será direcionado para o arquivo de Ajuda da classe.* Se você selecionar um método, você será direcionado a uma lista de todas as classes que entendem aquele método (por exemplo, peça ajuda do método `scramble`).

*Atenção: O SuperCollider vai mostrar em azul qualquer palavra que começa com uma letra maiúscula. Isso significa que a cor azul *não garante* que a palavra esteja livre de erros: por exemplo, se você digitar Sinosc (com o "o"minúsculo incorreto), ainda assim a palavra aparece em azul.

Outras maneiras de explorar os arquivos de Ajuda do SuperCollider são os links "Browse"("Navegar") e "Search"("Buscar"). Use o Browse para navegar os arquivos por categorias e o Search para pesquisar palavras em todos os arquivos de Ajuda. Nota importante sobre o Browser de Ajuda no IDE do SuperCollider:

- Use o campo superior direito (onde se lê "Find...") para procurar palavras específicas *dentro do arquivo de Help que está aberto* (da mesma maneira que você usaria um "find"para localizar algo em um website ou num documento de texto);
- Use o link "Search"(à direita de "Browse") para procurar texto *em todos os arquivos de Ajuda*.

Quando você abre o primeiro parêntese para adicionar argumentos de um método específico, o SC mostra uma pequena "dica de ajuda"para mostrar quais são os argumentos esperados. Por exemplo, digite o início de uma linha como mostrado na figura 6. Logo depois de abrir o primeiro parêntese, a dica mostra que os argumentos para um `SinOsc.ar` são `freq`, `phase`, `mul` e `add`. Também aparecem os valores padrão. Esta é exatamente a mesma informação que você obteria no arquivo de Ajuda do `SinOsc`. Se a dica de ajuda desapareceu, você pode trazê-la de volta com [ctrl+Shift+Espaço].

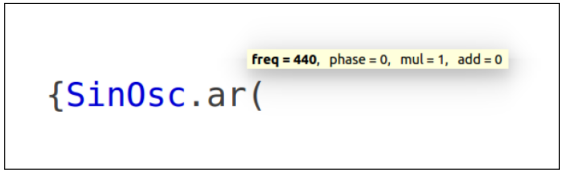

Figura 6: Informações úteis mostradas conforme você vai digitando.

Outro atalho: se você quiser explicitamente nomear seus argumentos (como `SinOsc.ar(freq: 890)`), experimente apertar a tecla Tab logo depois de abrir o parêntese. O SC vai autocompletar para você com o nome do argumento correto, na ordem, à medida que você digita (aperte Tab depois da vírgula para os nomes dos argumentos subsequentes).

DICA: Crie uma pasta com seus próprios "arquivos pessoais de ajuda". Sempre que você descobrir um novo truque ou aprender um novo objeto, escreva um exemplo simples com explicações em suas próprias palavras e salve-o para o futuro. Pode apostar que esses arquivos pessoais vão ser muito úteis depois de um mês ou um ano, quando você precisar lembrar como funciona tal ou qual objeto ou mensagem.

Os arquivos de Ajuda podem ser também consultados online: `http://doc.sccode.org/`.

Parte IV
SÍNTESE E PROCESSAMENTO SONORO

Você já aprendeu bastante coisas sobre o SuperCollider. Nas seções anteriores, este tutorial apresentou para você detalhes minuciosos sobre a própria linguagem, de variáveis a fechamentos e muito mais. Você também aprendeu como criar `Pbind`s interessantes, usando vários membros da família Pattern.

Esta parte do tutorial vai (finalmente!) apresentar síntese e processamento sonoros com o SuperCollider. Começaremos com o tópico das Unit Generators ("Unidades Geradoras"ou "Geradores Unitários", que daqui em diante chamaremos UGens). *

24 UGens

Você já viu algumas Unidades Geradoras (UGens) em ação nas seções 3 e 18. O que é uma UGen? Uma unidade geradora é um objeto que gera sinais sonoros ou sinais de controle. Estes sinais são sempre calculados no servidor. Há muitas classes de unidades geradoras, todas elas derivando da classe UGen. `SinOsc` e `LFNoise0` são exemplos de UGens. Para mais detalhes, veja os arquivos de Ajuda chamados "Unit Generators and Synths"e "Tour of UGens".

Quando você tocou seus Pbinds uns capítulos atrás, o som padrão era sempre o mesmo: um sintetizador simples semelhante a um piano. Este sintetizador é feito de uma combinação de unidades geradoras.† Você aprenderá como combinar unidades geradoras para criar todo tipo

*A maioria dos tutoriais começa diretamente com as UGens; nesta introdução ao SC, no entanto, escolhemos primeiro enfatizar a família Pattern (`Pbind` e sua turma) para uma abordagem pedagógica diferente.

†Como você usou `Pbind`s até aqui para fazer som no SuperCollider, pode ser tentador pensar: *"Entendi, então o `Pbind` é uma Unidade Geradora!"* Não é o caso. `Pbind` não é uma Unidade Geradora—ele é apenas uma receita

de instrumentos com sons sintéticos e processados. O próximo exemplo parte da nossa primeira senoide para criar um instrumento eletrônico que você pode tocar ao vivo com o mouse.

24.1 Controle do Mouse: Theremin instantâneo

Aqui está um sintetizador simples que você pode tocar em tempo real. É uma simulação do Theremin, um dos mais antigos instrumentos eletrônicos:

```
1 {SinOsc.ar(freq: MouseX.kr(300, 2500), mul: MouseY.kr(0, 1))}.play;
```

Se você não sabe o que é um Theremim, por favor pare tudo que está fazendo e procure por "Clara Rockmore Theremin"no YouTube. Depois volte aqui e tente tocar "O Cisne"com o seu Theremin de SuperCollider.

`SinOsc`, `MouseX` e `MouseY` são UGens. `SinOsc` está gerando o som da onda senoidal. Os outros dois estão captando o movimento do seu cursor na tela (X para o movimento horizontal e Y para o movimento vertical) e usando os números para alimentar os valores de frequência e amplitude da senoide. Bem simples, mas já bem divertido!

---

para fazer eventos musicais (partitura). *"Então o* ***`EventStreamPlayer`****, a coisa que aparece quando eu chamo* ***`play`*** *em um* ***`Pbind`****, ISSO deve ser uma UGen!"* A resposta ainda é não. O `EventStreamPlayer` é apenas o tocador, como um pianista, e o pianista não gera som. Continuando com esta comparação didática, o *instrumento piano* é a coisa que realmente vibra e produz som. Está é uma analogia mais adequada para a UGen: não é a partitura, não é o tocador: é o instrumento. Quando você fez música com `Pbind`s antes, o SC criava um `EventStreamPlayer` para tocar sua partitura com seu sintetizador de piano embutido. Você não tinha de se preocupar em criar o piano ou algo assim—o SuperCollider fez todo o trabalho nos bastidores para você. Aquele sintetizador de piano escondido é feito de uma combinação de algumas Unidades Geradoras.

24.2 Dente-de-serra e onda quadrada; gráfico e osciloscópio

O Theremin acima usou um oscilador senoidal. Há outras formas de onda que você pode usar para fazer som. Rode as linhas abaixo—elas usam o conveniente método `plot` ("gráfico")—para olhar a forma do `SinOsc` e compará-lo com `Saw` e `Pulse`. As linhas abaixo não produzem som—elas apenas permitem que você visualize um trecho da forma de onda.

```
1 { SinOsc.ar }.plot; // onda senoidal
2 { Saw.ar }.plot; // onda dente-de-serra
3 { Pulse.ar }.plot; // onda quadrada
```

Agora reescreva sua linha de Theremin, substituindo `SinOsc` por `Saw`, depois por `Pulse`. Escute como eles soam diferente. Finalmente, experimente `.scope` em vez de `.play` no seu código de Theremin e você poderá ver uma representação da forma de onda em tempo real (abrirá uma janela "Stethoscope", que na realidade é um osciloscópio).

25 Audio rate, control rate

É bastante fácil identificar uma UGen em um código de SuperCollider: elas são quase sempre seguidas pelas mensagens `.ar` ou `.kr`. Estas letras querem dizer Audio Rate ("taxa de áudio") e Control Rate ("taxa de controle"). Vamos ver o que isso significa.

Do arquivo de ajuda "Unit Generators and Synths"("Unidades geradoras e Sintetizadores"):

> Uma unidade geradora é criada ao se enviar uma mensagem `ar` ou `kr` para um objeto da classe da respectiva unidade geradora. A mensagem `ar` cria uma UGen que é executada na velocidade da taxa de áudio. A mensagem `kr` cria uma UGen executada na velocidade da taxa de controle. UGens de controle são usados para sinais de controle de baixa frequência ou que mudam lentamente. UGens de controle

produzem uma única amostra por ciclo de controle e, por isso, utilizam menos poder de processamento que UGens de áudio. *

Em outras palavras: quando você escreve `SinOsc.ar`, você está mandando a mensagem "taxa de áudio"para a UGen `SinOsc` UGen. Assumindo que seu computador esteja rodando na taxa de amostragem comum de 44100 Hz, este oscilador senoidal vai gerar 44100 amostras por segundo para serem enviadas ao alto-falante. Então escutamos uma onda senoidal.

Reflita por um momento sobre o que você acabou de ler: quando você manda a mensagem `ar` para uma UGen, você esta pedindo que que ela gere *quarenta e quatro mil e cem* números por segundo. Uma verdadeira enxurrada de números. Você escreve `{SinOsc.ar}.play` na linguagem e a linguagem comunica seu pedido ao servidor. O verdadeiro trabalho de gerar todas estas amostras é feito pelo servidor, o "motor sonoro"do SuperCollider.

Agora, quando você usa `kr` em vez de `ar`, o trabalho tambem é feito pelo servidor, mas há algumas diferenças:

1. A quantidade de números gerados por segundo com `.kr` é muito menor. `{SinOsc.ar}.play` gera 44100 números por segundo, enquanto `{SinOsc.kr}.play` produz perto de 700 números por segundo (se você tiver curiosidade, a quantidade exata é 44100 / 64, sendo que 64 é o chamado "período de controle".)

2. O sinal gerado com `kr` não vai para os alto-falantes. Em vez disso, é normalmente utilizado para controlar parâmetros de outros sinais—por exemplo, o `MouseX.kr` no seu theremim estava controlando a frequência de um `SinOsc`.

OK, então UGens são geradores de números incrivelmente velozes. Alguns destes números se tornam sinais sonoros; outros se tornam sinais de controle. Até aí, tudo bem. Mas que

*`http://doc.sccode.org/Guides/UGens-and-Synths.html`

números são estes, afinal de contas? Grandes? Pequenos? Positivos? Negativos? Na verdade, são números bem pequenos, muitas vezes entre -1 e +1. Às vezes somente entre 0 e 1. Todas as UGens podem ser divididas em duas categorias, de acordo com o âmbito númerico que elas geram: UGens unipolares e UGens bipolares.

UGens unipolares geram números entre 0 e 1.

UGens bipolares geram números entre -1 e +1.

25.1 O método `poll`

Vale a pena examinar a saída de algumas UGens para esclarecer isso. Não é uma boa ideia pedir pro SuperCollider imprimir milhares de números por segundo na Post window, mas podemos pedir que ele imprima algumas dezenas deles, só para termos uma ideia. Digite e rode as seguintes linhas, uma de cada vez (confirme que seu servidor esteja rodando), e observe a Post window:

```
1 // apenas observe a Post window (não vai fazer som)
2 {SinOsc.kr(1).poll}.play;
3 // pressione ctrl+ponto, daí rode a próxima linha:
4 {LFPulse.kr(1).poll}.play;
```

Os exemplos não produzem som algum porque estamos usando `kr`—o resultado é um sinal de controle, então nada é enviado para os alto-falantes. O objetivo aqui é apenas observar a saída típica de algumas UGens. A mensagem `poll` pega 10 números por segundo da saída do `SinOsc` e as imprime na Post window. O argumento 1 é a frequência da senoide, o que quer dizer que a onda senoidal vai demorar um segundo para completar um ciclo inteiro. Baseado no que você observou, o `SinOsc` é unipolar ou bipolar? E o `LFPulse`?[6]

Abaixe todo o volume antes de rodar a próxima linha, depois aumente-o devagar. Você vai ouvir uns cliques suaves.

```
1 {LFNoise0.ar(1).poll}.play;
```

Como mandamos uma mensagem `ar` para esta Ugen, ela está enviando 44100 amostras por segundo para a sua placa de som—é um sinal de áudio. `LFNoise0` é um gerador de ruído de baixa frequência ("Low Frequency Noise"). Cada amostra é um número aleatoriamente escolhido entre -1 e +1 (então é uma UGen bipolar). Com `poll` você está vendo somente dez deles por segundo. `LFNoise0.ar(1)` escolhe um novo número aleatório a cada segundo. Tudo isto é feito pelo servidor.

Pare os cliques com [ctrl+.] e experimente mudar a frequência de `LFNoise0`. Tente números como 3, 5, 10 e depois mais altos. Observe os números produzidos e ouça os resultados.

26 Argumentos de UGen

Quase sempre você vai precisar especificar os argumentos das UGens que você estiver usando. Você já viu isso antes: quando escrevemos `{SinOsc.ar(440)}.play`, o número `440` é um argumento para o `SinOsc.ar`; ele especifica a frequência que você vai ouvir. Você pode nomear os argumentos explicitamente, assim: `{SinOsc.ar(freq: 440, mul: 0.5)}.play`. Os nomes dos argumentos são `freq` e `mul` (note os dois pontos imediatamente após as palavras no código). O `mul` quer dizer "multiplicador"e é essencialmente a amplitude da forma de onda. Se você não prover nenhum valor para `mul`, o SuperCollider usa o valor padrão de 1 (amplitude máxima). Um valor como `mul: 0.5` significa multiplicar a forma de onda por meio, em outras palavras, ela vai tocar com metade da amplitude máxima.

No código do seu theremin, os argumentos `freq` e `mul` do `SinOsc` foram explicitamente nomeados. Você deve lembrar que `MouseX.kr(300, 2500)` foi usado para controlar a frequência do theremin. Nesse caso, o `MouseX.kr` recebeu dois argumentos: um limite mínimo e um máximo para seu âmbito de saída (300 e 2500, respectivamente). O mesmo vale para o `MouseY.kr(0, 1)`,

controlando a amplitude. Esses argumentos dentro as UGens de mouse não foram explicitamente nomeados, mas poderiam ter sido.

Como fazer pra descobrir que argumentos uma UGen aceita? Simplesmente vá para o arquivo de Ajuda correspondente: dê um duplo clique no nome da UGen para selecioná-la e aperte [ctrl+D] para abrir a página de documentação. Faça isso, digamos, com o MouseX. Depois da seção Description ("Descrição") você verá a seção Class Methods ("Métodos da Classe"). É ali que você vai descobrir que os argumentos do método `kr` são minval, maxval, warp e lag. Na mesma página, você também encontra a explicação sobre o que cada um deles significa.

Sempre que você não fornece um argumento, o SC vai usar os valores padrão que você vê no arquivo de Ajuda. Se você não nomear explicitamente os argumentos, você terá que fornecê-los na ordem exata mostrada no arquivo de Ajuda. Se você nomeá-los explicitamente, você pode colocá-los em qualquer ordem e até mesmo pular alguns do meio. Nomear explicitamente os argumentos é também uma boa ferramente de aprendizado, pois te ajuda a entender melhor o código. Um exemplo é dado abaixo.

```
1 // minval e maxval fornecidos na ordem, sem palavras-chave
2 {MouseX.kr(300, 2500).poll}.play;
3 // minval, maxval e lag fornecidos, warp foi pulado
4 {MouseX.kr(minval: 300, maxval: 2500, lag: 10).poll}.play;
```

27 Redimensionando âmbitos

A verdadeira festa começa quando você usa UGens para controlar os parâmetros de outras UGens. O exemplo do theremin fez exatamente isto. Agora você tem todas as ferramentas para entender exatamente o que está acontecendo em um dos exemplos da seção 3. As três últimas linhas do exemplo demonstram passo a passo como o `LFNoise0` é usado para controlar a frequência:

```
{SinOsc.ar(freq: LFNoise0.kr(10).range(500, 1500), mul: 0.1)}.play;

// Separando em partes:
{LFNoise0.kr(1).poll}.play; // veja um simples LFNoise0 em ação
{LFNoise0.kr(1).range(500, 1500).poll}.play; // agora com .range
{LFNoise0.kr(10).range(500, 1500).poll}.play; // agora mais rápido
```

27.1 Redimensione com o método `range`

O método **`range`** simplesmente redimensiona ("rescale") a saída de uma UGen. Lembre-se, **`LFNoise0`** produz números entre -1 e +1 (é uma UGen bipolar). Estes números brutos não seriam muito úteis para controlar frequência de uma senoide (precisamos de números em Hz que sejam perceptíveis na escala de audição humana). O **`.range`** pega esta saída entre -1 e +1 e a redimensiona entre qualquer valor mínimo e máximo que você fornecer como argumentos (neste caso, 500 e 1500). O número 10, que é o primeiro argumento do **`LFNoise0.kr`**, especifica a frequência da UGen: quantas vezes por segundos ela escolhe um novo número aleatório.

Resumindo: para usar uma UGen para controlar algum parâmetro de outra UGen, primeiro você tem que saber qual o âmbito numérico que você precisa. Os números serão usados como frequências? Você os quer entre, digamos, 10 e 1000? Ou são amplitudes? Talvez você queira que as amplitudes estejam entre 0.1 (suave) e 0.5 (metade do máximo)? Ou você está tentando controlar o número de harmônicos? Você quer que ele seja entre 5 e 19?

Uma vez que você definir o âmbito que você precisa, use o método **`.range`** para fazer com que a UGen controladora faça a coisa certa.

Exercício: escreva uma linha de código simples que toca uma onda senoidal, cuja frequência é controlada por um **`LFPulse.kr`** (forneça agumentos apropriados para ele). Então, use o método **`.range`** para redimensionar a saída do **`LFPulse`** para frequências que você queira escutar.

27.2 Redimensionando com `mul` e `add`

Agora você já sabe redimensionar a saída de UGens no servidor usando o método `.range`. A mesma coisa pode ser obtida em um nível mais fundamental usando os argumentos `mul` e `add`, que quase todas as UGens têm. O código abaixo mostra a equivalência entre as abordagens com `range` e de `mul/add`, ambos com uma UGen bipolar e uma UGen unipolar.

```
// Isto aqui:
{SinOsc.kr(1).range(100, 200).poll}.play;
// ...é o mesmo que:
{SinOsc.kr(1, mul: 50, add: 150).poll}.play;

// Isto aqui:
{LFPulse.kr(1).range(100, 200).poll}.play;
// ...dá na mesma que:
{LFPulse.kr(1, mul: 50, add: 100).poll}.play;
```

A figura 7 ajuda a visualizar como `mul` e `add` trabalham juntos redimensionando as saídas de UGens (um `SinOsc` é usado como demonstração).

27.3 `linlin` e sua turma

Para qualquer outro redimensionamento arbitrário de âmbitos, você pode usar os práticos métodos `linlin`, `linexp`, `explin` e `expexp`. Os nomes nos métodos dão uma dica do que eles fazem: converter um âmbito linear em outro âmbito linear (`linlin`), linear para exponencial (`linexp`), etc.

```
// Um punhado de números
a = [1, 2, 3, 4, 5, 6, 7];
// Redimensione para 0—127, linear para linear
a.linlin(1, 7, 0, 127).round(1);
```

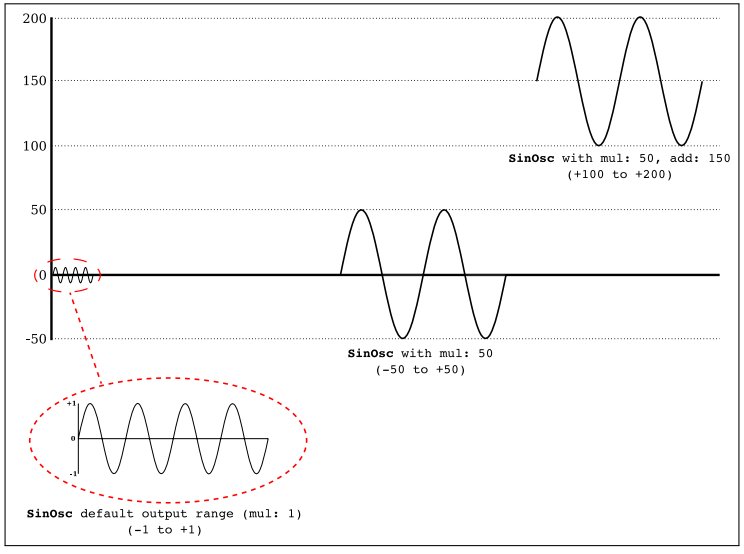

Figura 7: Redimensionando âmbitos de UGens com mul e add

```
5 // Redimensione para 0—127, linear para exponencial
6 a.linexp(1, 7, 0.01, 127).round(1); // não use zero para um âmbito exponencial
```

Para uma revisão acerca de linear e exponencial, procure online a diferença entre progressões aritméticas e geométricas. Brevemente, sequências lineares (aritméticas) são como "1, 2, 3, 4, 5, 6"or "3, 6, 9, 12, 15", etc; e sequências exponenciais (geométricas) são como "1, 2, 4, 8, 16, 32"or "3, 9, 27, 81, 243", etc.

28 Parando sintetizadores individualmente

Eis um jeito bastante comum de iniciar diversos sintetizadores e ser capaz de interrompê-los separadamente. O exemplo é autoexplicativo:

```
// Rode uma linha de cada vez (não pare o som entre elas):
a = { Saw.ar(LFNoise2.kr(8).range(1000, 2000), mul: 0.2) }.play;
b = { Saw.ar(LFNoise2.kr(7).range(100, 1000), mul: 0.2) }.play;
c = { Saw.ar(LFNoise0.kr(15).range(2000, 3000), mul: 0.1) }.play;
// Pare os sintetizadores individualmente:
a.free;
b.free;
c.free;
```

29 A mensagem `set`

Assim como com qualquer função (reveja a seção 21), argumentos especificados no início da sua função de sintetizador ficam acessíveis ao usuário. Isso permite que você modifique parâmetros em tempo real (enquanto o sintetizador está rodando). A mensagem **`set`** ("definir") é usada para este fim. Exemplo simples:

```
x = {arg freq = 440, amp = 0.1; SinOsc.ar(freq, 0, amp)}.play;
x.set(\freq, 778);
x.set(\amp, 0.5);
x.set(\freq, 920, \amp, 0.2);
x.free;
```

É um bom hábito fornecer valores pré-definidos (como 440 e 0.1 acima), de outra forma o sintetizador não vai tocar até que você defina um valor apropriado para os parâmetros 'vazios'.

30 Canais de Áudio ("Audio Buses")

Canais de áudio ("audio buses") são usados para rotear sinais de áudio. É como se fossem os canais de uma mesa de som. O SuperCollider tem 128 canais de áudio como padrão. Também existem canais de controle (para sinais de controle), mas por enquanto vamos nos concentrar só nos canais de áudio.*

Pressione [ctrl+M] para abrir a janela Meter ("Medidor"). Ela mostra os níveis de todas as entradas e saídas. A figura 8 mostra uma captura de tela dessa janela e sua correspondência com os canais padrão do SuperCollider. No SC, canais de áudio são numerados de 0 a 127. Os primeiros oito (0-7) são por definição reservados para serem os canais de saída da sua placa de som. Os próximos oito (8-15) são reservados para as entradas da sua placa de som. Todos os outros (16 a 127) estão livres para serem utilizados como se desejar, por exemplo, quando você precisa rotear sinais de áudio de uma UGen para outra.

30.1 As UGens`Out` e `In`

Agora experimente a seguinte linha de código:

```
1 {Out.ar(1, SinOsc.ar(440, 0, 0.1))}.play; // canal direito
```

A UGen `Out` cuida do roteamento de sinais para canais específicos.

O primeiro argumento para `Out` é o canal de destino, isto é, para onde você quer que o sinal vá. No exemplo acima, o número `1` significa que queremos mandar o sinal para o canal 1, que é o canal direito da sua placa de som.

O segundo argumento de `Out.ar` é o próprio sinal que você quer "escrever"neste canal. Pode ser uma única UGen ou uma combinação de UGens. No exemplo, é uma simples onda senoidal.

*Vamos dar uma rápida olhada em canais de controle na seção 41.

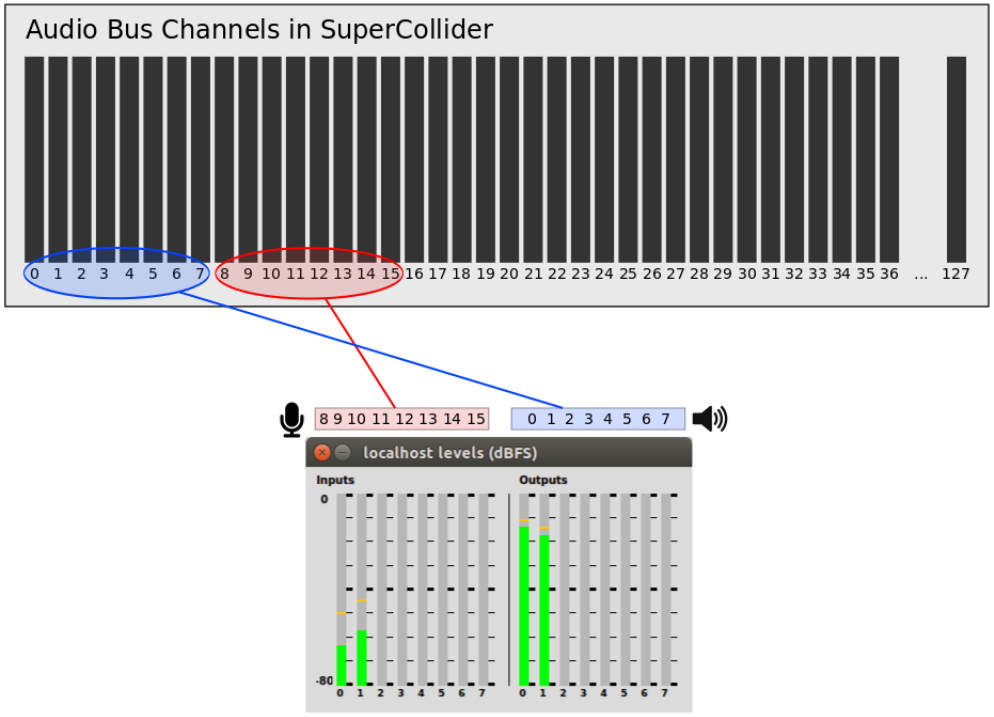

Figura 8: Canais de áudio e a janela Meter no SC.

Você deve ouvi-la somente no seu alto-falante direito (ou seu ouvido direito, se estiver usando fones de ouvido).

Com a janela Meter aberta e visível, vá ao código e mude o primeiro argumento de `Out.ar`: tente qualquer número entre 0 e 7. Observe os medidores. Você verá que o sinal vai para qualquer lugar que você mandá-lo.

DICA: É bastante provável que você tenha uma placa de som que só pode tocar dois canais (esquerdo e direito), então você somente escutará a senoide quando mandá-la para o canal 0 ou 1. Se você enviá-la para outros canais (3 a 7), você ainda poderá ver o medidor correspondente mostrando o sinal: o SC está de fato mandando som para aquele canal, mas a menos que você tenha uma placa de som de 8 canais, você não poderá ouvir a saída dos canais 3-7.

Um exemplo simples de um canal de áudio sendo usado para um efeito é mostrado abaixo.

```
1 // iniciar o efeito
2 f = {Out.ar(0, BPF.ar(in: In.ar(55), freq: MouseY.kr(1000, 5000), rq: 0.1))}.play;
3 // iniciar a fonte sonora
4 n = {Out.ar(55, WhiteNoise.ar(0.5))}.play;
```

A primeira linha declara um sintetizador (armazenado na variável `f`), consistindo em uma UGen de filtro (Band Pass Filter: "filtro passa-banda"). Um filtro passa-banda aceita qualquer som como entrada e *elimina todas as frequências exceto a região de frequência que você quer deixar passar*. `In.ar` é a UGen que usamos para ler de um canal de áudio. Portanto, com `In.ar(55)` sendo utilizado como entrada para o `BPF`, qualquer som que mandarmos para o canal 55 passará pelo filtro passa-banda. Note que o primeiro sintetizador, em um primeiro momento, não produz som algum: quando você roda a primera linha, o canal 55 ainda está vazio. Ele

somente produzirá som quando mandarmos algum audio para o canal 55, que é o que acontece na segunda linha.

A segunda linha cria um sintetizador e o armazena na variável `n`. Este sintetizador simplesmente gera ruído branco e o envia *não diretamente para os alto-falantes, mas sim para o canal de audio 55.* Este é precisamente o canal que nosso sintetizador de filtro está escutando, então assim que você rodar a segunda linha, você deve começar a ouvir o ruído branco sendo filtrado pelo sintetizador `f`.

Em resumo, o roteamento tem a seguinte configuração:

sintetizador de ruído → *canal 55* → *sintetizador de filtro*

A ordem de execução é importante. O exemplo anterior não funcionará se você rodar a fonte *antes* do efeito. Isso será discutido em mais detalhe na seção 42, "Ordem de Execução".

Uma última coisa: quando em exemplos anteriores você escreveu sintetizadores como `{ SinOsc.ar(440) }.play`, internamente o SC estava de fato executando `{ Out.ar(0, SinOsc.ar(440)) }.play`: se nada for dito, o SC assume que você queria mandar som para o canal 0, então ele automaticamente embute (ou "aninha") a primeira UGen em uma UGen `Out.ar(0, ...)`. Na realidade, há mais algumas coisas acontecendo nos bastidores, mas voltaremos a isso mais tarde (seção 39).

31 Entrada de Microfone

O exemplo abaixo mostra como você pode facilmente acessar a entrada da sua placa de som com a UGen `SoundIn`.*

*Sabendo que `In.ar` lê o sinal de qualquer canal indicado, e sabendo que as entradas da sua placa de som são por padrão assinaladas para os canais 8-15 do SC, você poderia escrever `In.ar(8)` para obter som do seu microfone. Isso funciona perfeitamente bem, mas `SoundIn.ar` é uma opção mais conveniente.

```
// Aviso: use fones de ouvido para evitar microfonia
{SoundIn.ar(0)}.play; // o mesmo que In.ar(8): recebe som do primeiro canal de
    entrada

// Versão estéreo
{SoundIn.ar([0, 1])}.play; // primeira e segunda entradas

// Um pouco de reverb só para animar?
{FreeVerb.ar(SoundIn.ar([0, 1]), mix: 0.5, room: 0.9)}.play;
```

32 Expansão Multicanal

Com sua janela Meter aberta—[ctrl+M]—, observe o seguinte.

```
{Out.ar(0, Saw.ar(freq: [440, 570], mul: Line.kr(0, 1, 10)))}.play;
```

Estamos utilizando uma simpática UGen `Line.kr` para aumentar a amplitude de 0 a 1 em 10 segundos. Legal. Mas há outras mágicas interessantes acontecendo aqui. Você percebeu que há dois canais de saída (esquerdo e direito)? Você ouviu que há uma nota diferente em cada canal? Que que estas duas notas vêm de uma *lista*—[440, 570]—que foi fornecida para o `Saw.ar` como argumento `freq`?

Isto se chama Expansão Multicanal.

Expansão multicanal em SuperCollider é uma maneira de usar arrays dentro de sintetizadores que às vezes parece magia negra. É uma das características mais poderosas e únicas do programa. E também uma das coisas que mais intriga as pessoas no início.

Em poucas palavras: se você utilizar um array como qualquer argumento de uma UGen, *todo o patch é duplicado*. O número de cópias criado é igual ao *número de itens no array*. Estas

duplicatas são enviadas para tantos *canais adjacentes* quantos forem necessários, começando pelo primeiro canal especificado como primeiro argumento de `Out.ar`.

No exemplo acima, temos `Out.ar(0, ... )`. O `freq` da onda dente-de-serra ("Saw") é um array de dois itens: `[440, 570]`. O que o SC faz? Ele "expande multicanal", criando duas cópias de todo o patch. A primeira cópia é uma onda dente-de-serra com uma frequência de 440 Hz, enviada para o canal 0 (canal esquerdo); a segunda cópia é uma dente-de-serra com uma frequência de 570 Hz, enviada para o canal 1 (canal direito)!

Vamos lá, verifique você mesmo. Mude estas duas frequências para qualquer outros valores que você quiser. Escute os resultados. Um vai para o canal esquerdo e o outro vai para o direito. Vá além e adicione uma terceira frequência para a lista (digamos, `[440, 570, 980]`). Observe a janela Meter. Você verá que as três primeiras saídas estão iluminadas (mas você só conseguirá ouvir a terceira se tiver uma placa de som multicanal).

Além disso: você pode usar arrays adicionais em outros argumentos da mesma UGen ou em argumentos de outras UGens no mesmo sintetizador. O SuperCollider vai tomar conta de tudo direitinho e gerar sintetizadores que usem todos os valores corretamente. Por exemplo: agora ambas as frequências [440, 570] estão crescendo suavemente ("fade in") de 0 a 1 em 10 segundos. Mas mude o código para `Line.kr(0, 1, [1, 15])` e você fará com que o som de 440 Hz demore 1 segundo para crescer e o de 570 Hz leve 15 segundos. Experimente.

Exercício: escute esta simulação de um "sinal de ocupado"de um telefone antigo. Ela usa a expansão multicanal para criar dois osciladores senoidais, cada um tocando uma frequência em um canal diferente. Faça o canal esquerdo pulsar 2 vezes por segundo e o canal direito pulsar 3 vezes por segundo.[7]

```
1 a = {Out.ar(0, SinOsc.ar(freq: [800, 880], mul: LFPulse.ar(2)))}.play;
2 a.free;
```

33 O objeto Bus

Aqui está um exemplo que utiliza tudo o que você aprendeu nas duas seções anteriores: canais de áudio e expansão multicanal.

```
// Rode isso primeiro ('ligar reverb' -- inicialmente você não vai ouvir nada)
r = {Out.ar(0, FreeVerb.ar(In.ar(55, 2), mix: 0.5, room: 0.9, mul: 0.4))}.play;

// Agora rode isto em seguida ('envie o som para o canal de reverb')
a = {Out.ar(55, SinOsc.ar([800, 880], mul: LFPulse.ar(2)))}.play;
a.free;
```

Graças à expansão multicanal, o som usa dois canais. Quando (no sintetizador `a`) enviamos o sinal para o canal 55, na verdade, dois canais estão sendo utilizados—o número 55 e o canal imediatamente adjacente, 56. No reverb (sintetizador `r`), indicamos com `In.ar(55, 2)` que queremos ler 2 canais, começando pelo canal 55, de modo que tanto 55 quanto 56 vão entrar no reverb. A saída do reverb também é expandida para dois canais, de maneira que o sintetizador `r` manda som para os canais 0 e 1 (canais esquerdo e direito da sua placa de som).

Esta escolha de canal (número 55) para conectar uma fonte sonora a um efeito foi arbitrária: poderia ter sido qualquer outro número entre 16 e 127 (lembre-se, canis 0-15 são reservados para a as saídas e entradas da placa de som). Seria muito inconveniente ter que ficar escolhendo e lembrando de números assim a todo momento. Quando os patches começarem a ficar mais complexos, imagine que pesadelo: "Qual foi mesmo o canal que escolhi para o reverb? Era 59 ou 95? E o canal do meu delay? Será que era 27? Não lembro..."e assim por diante.

O SuperCollider toma conta disso para você com objetos Bus. Nos exemplos acima, nós só definimos manualmente o tal do canal 55 como forma de demonstração. No dia-a-dia com o SuperCollider, você pode simplesmente usar o objeto Bus. O objeto Bus faz o trabalho de escolher um canal disponível para você e monitorá-lo. Eis como usá-lo:

```
// Criar o bus
~meuBus = Bus.audio(s, 2);
// Ligar o reverb: ler de ~meuBus (fonte sonora)
r = {Out.ar(0, FreeVerb.ar(In.ar(~meuBus, 2), mix: 0.5, room: 0.9, mul: 0.4))}.play;
// Alimente o ~meuBus com o som
b = {Out.ar(~meuBus, SinOsc.ar([800, 880], mul: LFPulse.ar(2)))}.play;
// Libere ambos os sintetizadores
r.free; b.free;
```

O primeiro argumento do `Bus.audio` é a variável `s`, que representa o servidor. O segundo argumento é quantos canais você precisa (2 no exemplo). Daí você armazena isso em uma variável com um nome qualquer (`~meuBus` no exemplo, mas poderia ser `~reverbBus`, `~fonte`, `~tangerina` ou o que fizer sentido para você dentro do seu patch). Depois disso, sempre que você precisar se referir àquele bus, é só usar a variável que você criou.

34 Pan

Panorâmica ou Pan é a distribuição de um sinal de áudio em um campo sonoro estéreo ou multicanal. O exemplo abaixo toca um sinal mono oscilando entre o canal esquerdo e o direito graças ao `Pan2`:*

```
p = {Pan2.ar(in: PinkNoise.ar, pos: SinOsc.kr(2), level: 0.1)}.play;
p.free;
```

No arquivo de Ajuda do `Pan2`, vemos que o argumento `pos` ("posição") requer um número entre -1 (esquerda) e +1 (direita), 0 sendo o centro. É por isso que você pode utilizar um `SinOsc`

*Para pan multicanal, dê uma olhada em `Pan4` e `PanAz`. Usuários avançados podem se interessar pelos plug-ins de SuperCollider para Ambisonics.

diretamente neste argumento: o oscilador senoidal é uma UGen bipolar, então ela gera números entre -1 e +1 por definição.

Agora vamos analisar um exemplo mais elaborado. Uma onda dente-de-serra passa por um filtro passa-banda muito estreito (`rq: 0.01`). Note o uso de variáveis locais para tornar modulares diferentes partes do código. Analise e tente entender o máximo que você puder no exemplo. Depois responda as perguntas a seguir.

```
(
x = {
        var lfn = LFNoise2.kr(1);
        var saw = Saw.ar(
                freq: 30,
                mul: LFPulse.kr(
                        freq: LFNoise1.kr(1).range(1, 10),
                        width: 0.1));
        var bpf = BPF.ar(in: saw, freq: lfn.range(500, 2500), rq: 0.01, mul: 20);
        Pan2.ar(in: bpf, pos: lfn);
}.play;
)
x.free;
```

Perguntas:

(a) A variável `lfn` é usada em dois lugares diferentes. Por quê? (Qual o resultado?)

(b) O que acontece se você mudar o argumento `mul:` do `BPF` de 20 para 10, 5 ou 1? Por que um número grande como 20 foi usado?

(c) Qual parte do código está controlando o ritmo?

Respostas ao final do livro.[8]

35 Mix e Splay

Este é um truque bacana. Você pode usar expansão multicanal para gerar sons complexos e depois mixá-los todos para mono ou estéreo com `Mix` ou `Splay`:

```
// saída com 5 canais (veja a janela Meter)
a = { SinOsc.ar([100, 300, 500, 700, 900], mul: 0.1) }.play;
a.free;
// Mixe em mono:
b = { Mix(SinOsc.ar([100, 300, 500, 700, 900], mul: 0.1)) }.play;
b.free;
// Mixe em estéreo (distribuição uniforme da esquerda para a direita)
c = { Splay.ar(SinOsc.ar([100, 300, 500, 700, 900], mul: 0.1)) }.play;
c.free
// Divirta-se com Splay:
(
d = {arg fundamental = 110;
        var harmonicos = [1, 2, 3, 4, 5, 6, 7, 8, 9];
        var som = BPF.ar(
                in: Saw.ar(32, LFPulse.ar(harmonicos, width: 0.1)),
                freq: harmonicos * fundamental,
                rq: 0.01,
                mul: 20);
        Splay.ar(som);
}.play;
)
d.set(\fundamental, 100); // mude a fundamental só porque é legal
d.free;
```

Você consegue ver a expansão multicanal em ação neste último exemplo de `Splay`? A única diferença é que o array é primeiro armazenado em uma variável (`harmonicos`) antes de ser

utilizada nas UGens. O array `harmonicos` tem 9 items, então o sintetizador irá se expandir para 9 canais. Então, um pouco antes de `.play`, `Splay` recebe o array de 9 canais e os mixa em estéreo, distribuindo os canais uniformemente da esquerda para a direita.*

`Mix` tem um outro truque interessante: o método `fill`. De uma só vez, ele cria um array de sintetizadores e os mixa em mono.

```
1  // Gerador instantâneo de clusters
2  c = { Mix.fill(16, {SinOsc.ar(rrand(100, 3000), mul: 0.01)}) }.play;
3  c.free;
4  // Uma nota com 12 parciais com amplitudes decrescentes
5  (
6  n = { Mix.fill(12, {arg contador;
7          var parcial = contador + 1; // queremos começar do 1, não do 0
8          SinOsc.ar(parcial * 440, mul: 1/parcial.squared) * 0.1
9          })
10 }.play;
11 FreqScope.new;
12 )
13 n.free;
```

Você fornece duas coisas para o `Mix.fill`: o tamanho do array e uma função (entre chaves) que será utilizada para preencher o array. No primeiro exemplo acima, `Mix.fill` executa a função 16 vezes. Note que a função inclui um componente variável: a frequência do oscilador senoidal poderá ser qualquer número entre 100 e 3000. Dezesseis senoides serão criadas, cada uma com uma frequência aleatória diferente. Todas elas serão mixadas em mono e você ouvirá o resultado no seu canal esquerdo. O segundo exemplo mostra que a função pode receber um

*A última linha antes do `.play` poderia ser explicitamente escrita como `Out.ar(0, Splay.ar(som))`. Lembre-se que o SuperCollider está gentilmente preenchendo as lacunas e inserindo aí um `Out.ar(0...)`—é assim que o sintetizador sabe que deve tocar nos canais esquerdo (bus 0) e direito (bus 1).

argumento de "contador"que monitora o número de iterações (como em `Array.fill`). Doze osciladores senoidais são gerados seguindo a série harmônica e mixados como uma única nota em mono.

36 Tocando um arquivo de áudio

Primeiro, você tem que carregar o arquivo de som em um buffer (uma espécie de "contêiner"). O segundo argumento para `Buffer.read` é o caminho ("path"), entre aspas duplas, indicando onde o seu arquivo de som está salvo. Esse caminho deve apontar para um arquivo WAV ou AIFF no seu computador. Depois que o buffer é carregado, simplesmente use a UGen `PlayBuf` para tocá-lo de diversas maneiras.

DICA: Um jeito rápido de obter o caminho correto para um arquivo de som no seu computador é arrastar o arquivo para um documento em branco do SuperCollider. O SC fornecerá automaticamente o caminho completo, já entre aspas duplas!

```
1  // Carregar arquivos nos buffers:
2  ~buf1 = Buffer.read(s, "/home/Music/wheels-mono.wav"); // um arquivo de som
3  ~buf2 = Buffer.read(s, "/home/Music/mussorgsky.wav"); // outro arquivo de som
4
5  // Tocar:
6  {PlayBuf.ar(1, ~buf1)}.play; // número de canais e buffer
7  {PlayBuf.ar(1, ~buf2)}.play;
8
9  // Obter informações sobre os arquivos:
10 [~buf1.bufnum, ~buf1.numChannels, ~buf1.path, ~buf1.numFrames];
11 [~buf2.bufnum, ~buf2.numChannels, ~buf2.path, ~buf2.numFrames];
```

```
12
13 // Mudar a velocidade de reprodução com 'rate'
14 {PlayBuf.ar(numChannels: 1, bufnum: ~buf1, rate: 2, loop: 1)}.play;
15 {PlayBuf.ar(1, ~buf1, 0.5, loop: 1)}.play; // tocar na metade da velocidade
16 {PlayBuf.ar(1, ~buf1, Line.kr(0.5, 2, 10), loop: 1)}.play; // acelerando
17 {PlayBuf.ar(1, ~buf1, MouseY.kr(0.5, 3), loop: 1)}.play; // controlando com mouse
18
19 // Inverter direção (ao contrário)
20 {PlayBuf.ar(1, ~buf2, -1, loop: 1)}.play; // inverter som
21 {PlayBuf.ar(1, ~buf2, -0.5, loop: 1)}.play; // tocar o som na metade da velocidade E
       ao contrário
```

37 Nós de sintetizador

Nos exemplos anteriores com `PlayBuf`, você teve que apertar [ctrl+.] depois de cada linha para parar o som. Em outros exemplos, você atribuiu o sintetizador a uma variável (como `x = {WhiteNoise.ar}.play`) para que você pudesse pará-lo diretamente com `x.free`.

Toda vez que você cria um sintetizador no SuperCollider, você sabe que ele roda no servidor, nosso "motor sonoro". Cada sintetizador que está rodando no servidor é representado por um *node* ("nó"ou "nódulo"). Podemos dar uma espiada nesta árvore de nós com o comando `s.plotTree`. Experimente. Uma janela chamada `NodeTree` ("Árvore de nós") vai abrir.

```
1 // abra a GUI
2 s.plotTree;
3 // rode estes um por um (não pare o som) e observe a Node Tree:
4 w = { SinOsc.ar(60.midicps, 0, 0.1) }.play;
5 x = { SinOsc.ar(64.midicps, 0, 0.1) }.play;
6 y = { SinOsc.ar(67.midicps, 0, 0.1) }.play;
7 z = { SinOsc.ar(71.midicps, 0, 0.1) }.play;
```

```
w.free;
x.free;
y.free;
z.free;
```

Cada retângulo que você vê na Node Tree é um nó de sintetizador. Cada sintetizador ganha um nome temporário (algo como temp_101, temp_102, etc.) e fica ali enquanto estiver rodando. Experimente agora tocar novamente as quatro senoides e aperte [ctrl+.] (observe a janela Node Tree). O atalho [ctrl+.] impiedosamente interrompe todos os nós que estão rodando no servidor. Por outro lado, com o método `.free`, você pode ser mais sutil e liberar nós específicos, um de cada vez.

Uma coisa importante de se perceber é que sintetizadores podem continuar a rodar no servidor mesmo que eles estejam gerando apenas silêncio. Eis um exemplo. A amplitude desta UGen `WhiteNoise` irá de 0.2 a 0 em dois segundos. Depois disso, não escutaremos nada. Mas você vê que o nó do sintetizador ainda está ali e não desaparecerá até que você o libere.

```
// Execute e observe a janela Node Tree window por alguns segundos
x = {WhiteNoise.ar(Line.kr(0.2, 0, 2))}.play;
x.free;
```

37.1 O glorioso doneAction: 2

Felizmente, há uma maneira de criar sintetizadores mais espertos neste sentido: por exemplo, não seria ótimo se pudéssemos pedir ao `Line.kr` para notificar o sintetizador quando ele tiver terminado seu trabalho (a rampa de 0.2 a 0), e então o sintetizador se liberasse automaticamente?

Insira o argumento `doneAction: 2` para resolver todos os nossos problemas.

Toque os exemplos abaixo e compare como eles se comportam com e sem doneAction: 2. Continue observando a Node Tree enquanto você roda as linhas.

```
// sem doneAction: 2
{WhiteNoise.ar(Line.kr(0.2, 0, 2))}.play;
{PlayBuf.ar(1, ~buf1)}.play; // PS.: isso presume que você anda tem seu arquivo de
    som carregado no ~buf1 da seção anterior

// com doneAction: 2
{WhiteNoise.ar(Line.kr(0.2, 0, 2, doneAction: 2))}.play;
{PlayBuf.ar(1, ~buf1, doneAction: 2)}.play;
```

Os sintetizadores com doneAction: 2 vão se liberar automaticamente logo que seu trabalho estiver feito (isto é, assim que a rampa do `Line.kr` tiver terminado no primeiro exemplo e logo que o `PlayBuf.ar` tiver terminado de tocar o arquivo de som no segundo exemplo). Confirme que você entendeu este conceito, pois ele será bastante útil na próxima seção: Envelopes.

38 Envelopes

Até agora, a maioria dos nossos exemplos foi de sons contínuos. Já está na hora de aprender a modelar o envelope de amplitude de um som. Um bom exemplo para começar é um envelope percussivo. Imagine um ataque em um prato suspenso. O tempo que o som leva para ir do silêncio à amplitude máxima é muito curto—alguns milissegundos talvez. Isto é chamado *tempo de ataque*. O tempo que leva para o som do prato diminuir da máxima amplitude de volta ao silêncio (zero) é um pouco mais longo, talvez alguns segundos. Isto é chamado o *tempo de repouso*.

Pense em um envelope de amplitude simplesmente como um número que muda ao longo do tempo e pode ser utilizado como o multiplicador (`mul`) de qualquer UGen que produz som. Estes números devem estar entre 0 (silêncio) e 1 (amplitude máxima), porque é assim que o SuperCollider entende a amplitude. Talvez agora você se dê conta de que o último exemplo já

continha um envelope de amplitude: em `{WhiteNoise.ar(Line.kr(0.2, 0, 2, doneAction: 2))}.play`, fazemos a amplitude do ruído branco ir de 0.2 a 0 em 2 segundos. Um `Line.kr`, no entanto, não é um tipo de envelope muito flexível.

`Env` é o objeto que você usará o tempo todo para definir vários tipos de envelopes. O `Env` tem muitos métodos úteis; só conseguiremos ver alguns deles aqui. Dê uma olhada na Ajuda do `Env` para aprender mais.

38.1 Env.perc

`Env.perc` é uma maneira prática de se obter um envelope percussivo. Ele aceita quatro argumentos: attackTime, releaseTime, level e curve (que podemos traduzir como: "tempoDeAtaque, tempoDeRepouso, nível e curva". No caso de envelopes de amplitude, "nível"é como se fosse o "volume máximo"que o seu envelope pode alcançar). Vejamos algumas formas típicas, ainda fora de qualquer sintetizador.

```
1 Env.perc.plot; // usando todos os argumentos padrão
2 Env.perc(0.5).plot; // attackTime: 0.5
3 Env.perc(attackTime: 0.3, releaseTime: 2, level: 0.4).plot;
4 Env.perc(0.3, 2, 0.4, 0).plot; // o mesmo que acima, mas curve:0 produz uma linha
      reta
```

Agora simplesmente o encaixamos dentro de um sintetizador:

```
1 {PinkNoise.ar(Env.perc.kr(doneAction: 2))}.play; // argumentos padrão do Env.perc
2 {PinkNoise.ar(Env.perc(0.5).kr(doneAction: 2))}.play;
3 {PinkNoise.ar(Env.perc(0.3, 2, 0.4).kr(2))}.play;
4 {PinkNoise.ar(Env.perc(0.3, 2, 0.4, 0).kr(2))}.play;
```

Tudo o que você tem de fazer é adicionar `.kr(doneAction: 2)` logo depois de `Env.perc` e pronto. Na verdade, neste caso você pode até remover a declaração explícita do argumento

doneAction e simplesmente ficar com `.kr(2)`. O `.kr` está esta dizendo para o SC rodar este envelope na velocidade da taxa de controle (como outros sinais de controle que vimos antes).

38.2 Env.triangle

`Env.triangle` recebe apenas dois argumentos: duração e nível de amplitude. Exemplos:

```
// Veja-o:
Env.triangle.plot;
// Ouça-o:
{SinOsc.ar([440, 442], mul: Env.triangle.kr(2))}.play;
// Aliás, um envelope pode ser um multiplicador em qualquer lugar do seu código:
{SinOsc.ar([440, 442]) * Env.triangle.kr(2)}.play;
```

38.3 Env.linen

`Env.linen` descreve um envelope linear com ataque, porção de sustentação e repouso. Você também pode especificar o nível de amplitude e tipo de curva. Exemplo:

```
// Veja-o:
Env.linen.plot;
// Ouça-o:
{SinOsc.ar([300, 350], mul: Env.linen(0.01, 2, 1, 0.2).kr(2))}.play;
```

38.4 Env.pairs

Quer algo ainda mais flexível? Com `Env.pairs` você pode ter envelopes com qualquer forma e duração que quiser. `Env.pairs` recebe dois argumentos: uma lista (array) de pares de [tempo, nível] e um tipo de curva (veja na Ajuda de Env todos os tipos de curva disponíveis).

```
(
{
        var env = Env.pairs([[0, 0], [0.4, 1], [1, 0.2], [1.1, 0.5], [2, 0]], \lin);
        env.plot;
        SinOsc.ar([440, 442], mul: env.kr(2));
}.play;
)
```

Leia o array de pares assim:

No tempo 0, esteja no nível 0;
No tempo 0.4, esteja no nível 1;
No tempo 1, esteja no nível 0.2;
No tempo 1.1, esteja no nível 0.5;
No tempo 2, esteja no nível 0;

38.4.1 Envelopes—não só para amplitude

Nada impede você de usar as estas mesmas formas para controlar algo que não seja amplitude do som. Você só precisa redimensioná-las para o âmbito numérico desejado. Por exemplo, você pode criar um envelope para controlar a mudança de frequências ao longo do tempo:

```
(
{
        var freqEnv = Env.pairs([[0, 100], [0.4, 1000], [0.9, 400], [1.1, 555], [2,
            440]], \lin);
        SinOsc.ar(freqEnv.kr, mul: 0.2);
}.play;
)
```

Envelopes são uma maneira poderosa de controlar qualquer parâmetro de um sintetizador que precisa variar ao longo do tempo.

38.5 Envelope ADSR

Todos os envelopes vistos até agora têm uma coisa em comum: eles têm uma duração fixa, pré-definida. Há situações, no entanto, em que este tipo de envelope não é adequado. Por exemplo, imagine que você está tocando em um teclado MIDI. O *ataque* da nota é disparado quando você pressiona a tecla. O *repouso*, quando você tira seu dedo da tecla. Mas a quantidade de tempo que você permanece com o dedo pressionando a tecla não é conhecido de antemão. O que precisamos neste caso é de um "envelope sustentado". Em outras palavras, depois da porção de ataque, o envelope precisa segurar a nota por uma quantidade de tempo indefinida e apenas disparar a porção de repouso depois de algum sinal, ou mensagem— isto é, o momento em que você "solta a tecla".

Um envelope ASR (Ataque, Sustentação, Repouso) se encaixa perfeitamente nesse caso. Uma variação mais popular é o envelope ADSR (Ataque, Decaimento, Sustentação, Repouso). Vamos dar uma olhada nos dois.

```
1  // ASR
2  // Toque nota ('aperte tecla')
3  // attackTime: 0.5 seconds, sustainLevel: 0.8, releaseTime: 3 seconds
4  x = {arg gate = 1, freq = 440; SinOsc.ar(freq: freq, mul: Env.asr(0.5, 0.8, 3).kr(
       doneAction: 2, gate: gate))}.play;
5  // Pare nota ('tirar dedo da tecla' — ativar a porção de repouso)
6  x.set(\gate, 0); // uma alternativa é x.release
7
8  // ADSR (ataque, decaimento, sutentação, repouso)
9  // Toque nota:
10 (
```

```
d = {arg gate = 1;
        var snd, env;
        env = Env.adsr(0.01, 0.4, 0.7, 2);
        snd = Splay.ar(BPF.ar(Saw.ar((32.1, 32.2..33)), LFNoise2.kr(12).range(100,
            1000), 0.05, 10));
        Out.ar(0, snd * env.kr(doneAction: 2, gate: gate));
}.play;
)
// Pare nota:
d.release; // isto é equivalente a d.set(\gate, 0);
```

Conceitos-chave:

Ataque ("Attack") O tempo (em segundos) que leva para ir do silêncio (zero) até o pico de amplitude

Decaimento ("Decay") O tempo (em segundos) que leva para ir do pico de amplitude para a amplitude de sustentação

Sustentação ("Sustain") A amplitude (entre 0 e 1) na qual a nota é sustentada (importante: isto não tem nada a ver com tempo)

Repouso ("Release") O tempo (em segundos) que leva para ir do nível de sustentação para o zero (silêncio).

Como envelopes sustentados não tem uma duração total conhecida de antemão, eles precisam de uma notificação tanto de quando começar (disparar o ataque) e quando parar (disparar o repouso). Esta notificação é chamada um *gate* ("portão"). O gate é o que diz para que o envelope se 'abra' (1) ou se 'feche' (0), portanto começando e terminando a nota.

Para que um envelope ASR ou ADSR funcione no seu sintetizador, você precisa declarar um argumento `gate`. Normalmente, o padrão é `gate = 1` porque você quer que o sintetizador comece a tocar assim que for instanciado (ativado). Quando você quer que o sintetizador pare, simplesmente mande uma mensagem `.release` ou `.set(\gate, 0)`: a porção de repouso do envelope será então disparada. Por exemplo, se seu tempo de repouso é 3, a nota vai levar três segundos para se extinguir completamente *a partir do momento em que você enviou a mensagem* `.set(\gate, 0)`.

38.6 EnvGen

Vale registrar que a construção que você aprendeu nesta seção para gerar envelopes é um atalho, como mostrado no código abaixo.

```
1 // Isso:
2 { SinOsc.ar * Env.perc.kr(doneAction: 2) }.play;
3 // ... é um atalho disso:
4 { SinOsc.ar * EnvGen.kr(Env.perc, doneAction: 2) }.play;
```

`EnvGen` é a UGen que de fato toca os envelopes segmentados ("breakpoint envelopes") definidos por `Env`. Para todos os propósitos práticos, você pode continuar a usar o atalho. Mas é útil saber que estas notações são equivalentes, já que você muitas vezes verá `EnvGen` sendo utilizado nos arquivos de Ajuda e outros exemplos online.

39 Definições de sintetizador

Até aqui, sem dificuldade alguma, pudemos *definir* sintetizadores e fazê-los *tocar* imediatamente. Além disso, a mensagem `.set` nos deu alguma flexibilidade para alterar os controles do sintetizador em tempo real. No entanto, há situações em que você pode querer definir seus sintetizadores

antes (sem tocá-los imediatamente) e tocá-los somente depois. Em essência, isso significa que temos de separar o momento de escrever a receita (a definição de sintetizador) do momento de assar o bolo (criar o som).

39.1 SynthDef e Synth

`SynthDef` é o que usamos para "escrever a receita"de um sintetizador. Depois você pode tocá-lo com `Synth`. Aqui está um exemplo simples.

```
1  // Definição de sintetizador com o objeto SynthDef
2  SynthDef("minhaSenoide1", {Out.ar(0, SinOsc.ar(770, 0, 0.1))}).add;
3  // Toque uma nota com o objeto Synth
4  x = Synth("minhaSenoide1");
5  x.free;
6
7  // Um exemplo ligeiramente mais flexível usando argumentos
8  // e um envelope com desligamento automático (doneAction: 2)
9  SynthDef("minhaSenoide2", {arg freq = 440, amp = 0.1;
10         var env = Env.perc(level: amp).kr(2);
11         var snd = SinOsc.ar(freq, 0, env);
12         Out.ar(0, snd);
13 }).add;
14
15 Synth("minhaSenoide2"); // usando os valores pré-definidos;
16 Synth("minhaSenoide2", [\freq, 770, \amp, 0.2]);
17 Synth("minhaSenoide2", [\freq, 415, \amp, 0.1]);
18 Synth("minhaSenoide2", [\freq, 346, \amp, 0.3]);
19 Synth("minhaSenoide2", [\freq, rrand(440, 880)]);
```

O primeiro argumento para a `SynthDef` é um nome para o sintetizador definido pelo usuário. O segundo argumento é uma função na qual você especifica um gráfico de UGens (assim é

chamada uma combinação de UGens). Note que você tem de usar `Out.ar` explicitamente para indicar para qual canal você quer enviar o sinal. Finalmente, a `SynthDef` recebe a mensagem `.add` ao final, que diz que você está a adicionando a uma coleção de sintetizadores que o SC conhece. Isso fica valendo até a hora que você fechar o SuperCollider.

Depois que você criar uma ou mais definições de sintetizador com `SynthDef`, você pode tocá-las com `Synth`: o primeiro argumento é o nome do sintetizador que você quer usar e o segundo argumento (opcional) é uma array com quaisquer parâmetros que você queira especificar (freq, amp, etc.)

39.2 Exemplo

Eis um exemplo mais longo. Depois que a SynthDef é adicionada, nós utilizamos um truque com uma array para criar um acode de 6 notas com alturas e amplitudes aleatórias. Cada sintetizador é armazenado em uma das posições da array, para que possamos desligá-los individualmente.

```
1  // Criar SynthDef
2  (
3  SynthDef("uau", {arg freq = 60, amp = 0.1, gate = 1, uaurelease = 3;
4          var chorus, fonte, filtromod, env, som;
5          chorus = Lag.kr(freq, 2) * LFNoise2.kr([0.4, 0.5, 0.7, 1, 2, 5, 10]).range
               (1, 1.02);
6          fonte = LFSaw.ar(chorus) * 0.5;
7          filtromod = SinOsc.kr(1/16).range(1, 10);
8          env = Env.asr(1, amp, uaurelease).kr(2, gate);
9          som = LPF.ar(in: fonte, freq: freq * filtromod, mul: env);
10 Out.ar(0, Splay.ar(som))
11 }).add;
12 )
13
14 // Observe a Node Tree
```

```
15 s.plotTree;
16
17 // Criar um acorde de 6 notas
18 a = Array.fill(6, {Synth("uau", [\freq, rrand(40, 70).midicps, \amp, rrand(0.1, 0.5)
       ])}); // tudo em uma única linha
19
20 // Encerrar notas uma por uma
21 a[0].set(\gate, 0);
22 a[1].set(\gate, 0);
23 a[2].set(\gate, 0);
24 a[3].set(\gate, 0);
25 a[4].set(\gate, 0);
26 a[5].set(\gate, 0);
27
28 // AVANÇADO: rode o acorde de 6 notas novamente e depois execute esta linha.
29 // Você consegue imaginar o que está acontecendo?
30 SystemClock.sched(0, {a[5.rand].set(\freq, rrand(40, 70).midicps); rrand(3, 10)});
```

Para ajudá-lo a entender o SynthDef acima:

- O som resultante é a soma de sete osciladores dentes-de-serra com afinações muito próximas passando por um filtro passa-baixa ("low pass").
- Estes sete osciladores são criador por expansão multicanal.
- O que é a variável `chorus`? É a frequência `freq` multiplicada por um `LFNoise2.kr`. Aqui começa a expansão multicanal, porque um array de 7 itens é fornecido como argumento para o LFNoise2. O resultado é que sete cópias do LFNoise2 são criadas, cada uma rodando a uma velocidade diferente retirada da lista `[0.4, 0.5, 0.7, 1, 2, 5, 10]`. Suas saídas são restritas ao âmbito de 1.0 a 1.02.

- Como um atributo extra, note que `freq` está empacotado em um `Lag.kr`. Sempre que uma nova frequência alimenta o Synth, a UGen Lag simplesmente cria uma rampa entre o valor velho e o valor novo. O "lag time"(duração da rampa), neste caso, é 2 segundos. Isso é o que causa o efeito de glissando que você ouve após rodar a última linha do exemplo.

- A fonte sonora `LFSaw.ar` recebe a variável `chorus` como sua frequência. Em um exemplo concreto: para um valor `freq` de 60 Hz, a variável `chorus` resultaria em uma expressão como

$$60 * [1.01, 1.009, 1.0, 1.02, 1.015, 1.004, 1.019]$$

  na qual os números da lista estariam constantemente subindo e descendo de acordo com as velocidades de cada LFNoise2. O resultado final é uma lista de sete frequências sempre deslizando entre 60 e 61.2 (60 * 1.02). Isso é chamado *efeito chorus*, por isso o nome da variável.

- Quando a variável `chorus` é usada como freq de `LFSaw.ar`, acontece expansão multicanal: temos agora sete ondas dentes-de-serra com frequências ligeiramente diferentes.

- A variável `filtromod` é só um oscilador senoidal movendo-se muito lentamente (1 ciclo a cada 16 segundos), com seu âmbito de saída escalonado para 1-10. Isso será usado para modular a frequência de corte do filtro passa-baixa.

- A variável `som` contém o filtro passa-baixa (LPF), que recebe `fonte` como entrada e atenua todas as frequências acima de sua frequência de corte. Este corte não é um valor fixo: ele é a expressão `freq * filtromod`. Então no exemplo, ao assumir freq = 60, isso torna-se um número entre 60 e 600. Lembre-se que filtromod é um número oscilando entre 1 e 10, de maneira que a multiplicação seria 60 * (1 a 10).

- `LPF` também expande multicanal para sete cópias. O envelope de amplitude `env` também é aplicado neste ponto.
- Finalmente, `Splay` pega essa array de sete canais e mixa em estéreo.

39.3 Nos bastidores

Este processo em duas etapas de primeiro criar o SynthDef (com um nome próprio) e depois chamar um Synth é o que o SC faz o tempo todo quando você escreve comandos simples como `{SinOsc.ar}.play`. SuperCollider desdobra isso em (a) criar um SynthDef temporário e (b) tocá-lo imediatamente (essa é a razão dos nomes temp_01, temp_02 que você vê na Post window). Tudo isso nos bastidores, para sua conveniência.

```
1  // Quando você faz isso:
2  {SinOsc.ar(440)}.play;
3  //O que o SC está fazendo é isso:
4  {Out.ar(0, SinOsc.ar(440))}.play;
5  // O que por sua vez na verdade é isso:
6  SynthDef("nomeTemporario", {Out.ar(0, SinOsc.ar(440))}).play;
7
8  // E todos eles são atalhos desta operação em duas etapas:
9  SynthDef("nomeTemporario", {Out.ar(0, SinOsc.ar(440))}).add; // criar a definição de
       um sintetizador
10 Synth("nomeTemporario"); // tocá-lo
```

40 Pbind pode tocar sua SynthDef

Uma das belezas de se criar seus sintetizadores como `SynthDefs` é que você pode usar `Pbind` para tocá-los.

Assumindo que a SynthDef `"uau"` ainda esteja armazenado na memória (deveria estar, a não ser que você tenha fechado e reaberto o SC depois do último exemplo), experimente os `Pbind`s abaixo:

```
(
Pbind(
        \instrument, "uau",
        \degree, Pwhite(-7, 7),
        \dur, Prand([0.125, 0.25], inf),
        \amp, Pwhite(0.5, 1),
        \uaurelease, 1
).play;
)

(
Pbind(
        \instrument, "uau",
        \scale, Pstutter(8, Pseq([
                Scale.lydian,
                Scale.major,
                Scale.mixolydian,
                Scale.minor,
                Scale.phrygian], inf)),
        \degree, Pseq([0, 1, 2, 3, 4, 5, 6, 7], inf),
        \dur, 0.2,
        \amp, Pwhite(0.5, 1),
        \uaurelease, 4,
        \legato, 0.1
).play;
)
```

Ao usar `Pbind` para tocar um das suas `SynthDef`s personalizados, esteja atento aos seguintes

pontos:

- Use a chave ("key") `\instrument` do `Pbind` para declarar o nome da sua `SynthDef`.
- Todos os argumentos da sua SynthDef são controláveis a partir do `Pbind`: simplesmente use-os como chaves do `Pbind`. Por exemplo, repare no argumento chamado `\uaurelease` utilizado acima. Esta não é uma das chaves padrão entendidas pelo `Pbind`—ela só existe na definição do sintetizador `uau` (o nome bobo foi escolhido de propósito).
- Para utilizar todas as facildades de conversão de alturas do `Pbind` (as chaves `\degree`, `\note` e `\midinote`), tenha certeza de que sua `SynthDef` tem um argumento de entrada `freq` (tem que ser escrito exatamente assim). Pbind fará os cálculos para você.
- Se for usar um envelope sustentado como `Env.adsr`, garanta que seu sintetizador tenha o argumento padrão `gate = 1` (`gate` tem que ser excrito exatamente assim, porque o `Pbind` o utiliza nos bastidores para parar as notas nos momentos certos).
- Se você não estiver usando um envelope sustentado, tenha certeza que sua SynthDef inclui um doneAction: 2 em uma UGen apropriada, para liberar automaticamente os nós de sintetizador no servidor.

Exercício: escreva um ou mais `Pbind`s para tocar a SynthDef `"pluck"` fornecida abaixo. Para o argumento `cordaAbafada`, tente valores entre 0.1 e 0.9. Faça com que seu `Pbind` toque uma sequência lenta de acordes. Tente arpejar os acordes com `\strum`.

```
1 (
2 SynthDef("pluck", {arg amp = 0.1, freq = 440, decaimento = 5, cordaAbafada = 0.1;
3 var env, som;
4 env = Env.linen(0, decaimento, 0).kr(doneAction: 2);
```

```
som = Pluck.ar(
        in: WhiteNoise.ar(amp),
        trig: Impulse.kr(0),
        maxdelaytime: 0.1,
        delaytime: freq.reciprocal,
        decaytime: decaimento,
        coef: cordaAbafada);
    Out.ar(0, [som, som]);
}).add;
)
```

41 Canais de Controle

Em uma seção anterior deste tutorial, falamos sobre canais de áudio ("Audio Buses") (seção 30) e o objeto Bus (seção 33). Naquele momento, escolhemos deixar de lado o tópico dos Canais de Controle ("Control Buses") para nos focarmos no conceito de roteamento de áudio.

Canais de controle no SuperCollider são para o roteamento de sinais de controle, não de áudio. Exceto por esta diferença, não há nenhuma outra distinção prática ou conceitual entre canais de áudio de de controle. Você cria e gerencia um canal de controle da mesma maneira que você faz com os canais de áudio, simplesmente usando `.kr` em vez de `.ar`. O SuperCollider tem 4096 canais de controle como padrão.

A primeira parte do exemplo abaixo usa um número de canal arbitrário apenas com a finalidade de demonstração. A segunda parte usa o objeto Bus, que é a maneira recomendada de criar canais.

```
// Escreva um sinal de controle no canal de controle 55
{Out.kr(55, LFNoise0.kr(1))}.play;
// Leia um sinal de controle do canal 55
```

```
{In.kr(55).poll}.play;

// Usando o objeto Bus
~meuCanalDeControle = Bus.control(s, 1);
{Out.kr(~meuCanalDeControle, LFNoise0.kr(5).range(440, 880))}.play;
{SinOsc.ar(freq: In.kr(~meuCanalDeControle))}.play;
```

O próximo exemplo mostra um único sinal de controle sendo utilizado para modular dois sintetizadores diferentes ao mesmo tempo. No sintetizador `Blip`, o sinal de controle é redimensionado para controlar o número de harmônicos entre 1 e 10. No segundo sintetizador, o mesmo sinal de controle é redimensionado para modular a frequência do oscilador `Pulse`.

```
// Crie o canal de controle
~meuControle = Bus.control(s, 1);

// Direcione o sinal de controle para o canal
c = {Out.kr(~meuControle, Pulse.kr(freq: MouseX.kr(1, 10), mul: MouseY.kr(0, 1)))}.
    play;

// Toque os sons que estão sendo controlados
// (mova o mouse para ouvir as mudanças)
(
{
        Blip.ar(
                freq: LFNoise0.kr([1/2, 1/3]).range(50, 60),
                numharm: In.kr(~meuControle).range(1, 10),
                mul: LFTri.kr([1/4, 1/6]).range(0, 0.1))
}.play;

{
        Splay.ar(
                Pulse.ar(
```

```
                        freq: LFNoise0.kr([1.4, 1, 1/2, 1/3]).range(100, 1000)
                        * In.kr(~meuControle).range(0.9, 1.1),
                        mul: SinOsc.ar([1/3, 1/2, 1/4, 1/8]).range(0, 0.03))
        )
}.play;
)

// Desligue o sinal de controle para comparar
c.free;
```

41.1 asMap

No próximo exemplo, o método **asMap** ("como mapa") é usado para mapear um canal de controle diretamente para um nó de um sintetizador que esteja rodando. Desta maneira, você não precisara sequer de um **In.kr** na definição do sintetizador.

```
// Crie uma SynthDef
SynthDef("simples", {arg freq = 440; Out.ar(0, SinOsc.ar(freq, mul: 0.2))}).add;
// Crie um canal de controle
~umCanal = Bus.control(s, 1);
~outroCanal = Bus.control(s, 1);
// Iniciar controles
{Out.kr(~umCanal, LFSaw.kr(1).range(100, 1000))}.play;
{Out.kr(~outroCanal, LFSaw.kr(2, mul: -1).range(500, 2000))}.play;
// Toque um nota
x = Synth("simples", [\freq, 800]);
x.set(\freq, ~umCanal.asMap);
x.set(\freq, ~outroCanal.asMap);
x.free;
```

42 Ordem de Execução

Quando discutíamos canais de áudio na seção 30, nós mencionamos a importância da ordem de execução. O código abaixo é uma versão expandida do exemplo de ruído filtrado daquela seção. Vamos agora explicar o conceito básico de ordem de execução, e demonstrar o por quê da sua importância.

```
1  // Criar um canal de áudio
2  ~fxBus = Bus.audio(s, 1);
3  ~masterBus = Bus.audio(s, 1);
4  // Criar SynthDefs
5  (
6  SynthDef("noise", {Out.ar(~fxBus, WhiteNoise.ar(0.5))}).add;
7  SynthDef("filter", {Out.ar(~masterBus, BPF.ar(in: In.ar(~fxBus), freq: MouseY.kr
       (1000, 5000), rq: 0.1))}).add;
8  SynthDef("masterOut", {arg amp = 1; Out.ar(0, In.ar(~masterBus) * Lag.kr(amp, 1))}).
       add;
9  )
10 // Abrir a janela Node Tree:
11 s.plotTree;
12 // Tocar os synths (observe a Node Tree)
13 m = Synth("masterOut");
14 f = Synth("filter");
15 n = Synth("noise");
16 // Volume master
17 m.set(\amp, 0.1);
```

Primeiro, dois canais de áudio são criados nas variáveis ~`fxbus` e ~`masterBus`. Depois, três `SynthDefs` são criadas:

- `"noise"` é uma fonte de ruído que envia ruído branco para o canal de efeitos;

- `"filter"` é um filtro passa-banda que toma como entrada o canal de efeitos, e envia o som processado para o canal master;
- `"masterOut"` pega o sinal do canal master, aplica um controle de volume simples, e envia o som final (com volume ajustado) para os alto-falantes.

Observe a janela Node Tree conforme você roda os synths em ordem.

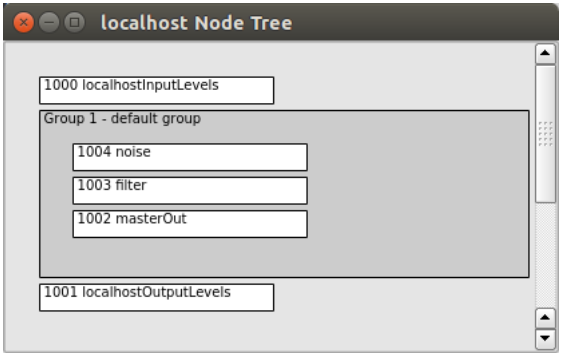

Figura 9: A janela Node Tree

Os nós de Synth na janela Node Tree são calculados *de cima pra baixo*. Os synths mais recentes são adicionados no topo da pilha. Na figura 9, você pode ver que `"noise"` está no topo, `"filter"` vem em segundo, e `"masterOut"` aparece por último. Esta é a ordem correta que nós queremos para este exemplo: lendo de cima para baixo, o ruído branco passa para o filtro, e o resultado do filtro passa para o canal master. Tente agora rodar o exemplo de novo, mas de trás pra frente (ou seja, rode as linhas na ordem `n`, `f`, and `m`. Você não vai ouvir nada, porque os sinais estão sendo calculados na ordem errada.

Executar as linhas certas na ordem correta é um método adequado em muitos casos, mas pode ser que isso fique um pouco complicado conforme o seu código vá ficando mais complexo. Para simplificar a tarefa, o SuperCollider permite que você defina explicitamente onde colocar os synths na janela Node Tree. Para tanto, usamos os argumentos `target` e `addAction`.

```
n = Synth("noise", addAction: 'addToHead');
m = Synth("masterOut", addAction: 'addToTail');
f = Synth("filter", target: n, addAction: 'addAfter');
```

Agora, não importa em que ordem você rodar as linhas acima, você pode ter certeza que os nós de cada synth vão ser colocados no lugar certo. O synth `"noise"` está sendo explicitamente adicionado na cabeça (topo) da janela Node Tree; o `"masterOut"` é adicionado na cauda (embaixo de todos os outros); e o `filter` é explicitamente adicionado logo depois do `n` (o synth de ruído).

42.1 Grupos

Conforme você começar a usar um monte de synths—alguns como fontes sonoras, outros para processamento, efeitos, seja lá o que você precisar—, pode ser uma boa ideia organizá-los em grupos. Aqui está um exemplo simples:

```
// Vá olhando tudo o que acontece na janela NodeTree
s.plotTree;

// Crie alguns canais de áudio
~reverbBus = Bus.audio(s, 2);
~masterBus = Bus.audio(s, 2);

// Defina os grupos
(
~sources = Group.new;
~effects = Group.new(~sources, \addAfter);
```

```
~master = Group.new(~effects, \addAfter);
)

// Rode todos os synths de uma vez
(
// Uma fonte de som qualquer
{
  Out.ar(~reverbBus, SinOsc.ar([800, 890])*LFPulse.ar(2)*0.1)
}.play(target: ~sources);

// Uma outra fonte sonora qualquer
{
  Out.ar(~reverbBus, WhiteNoise.ar(LFPulse.ar(2, 1/2, width: 0.05)*0.1))
}.play(target: ~sources);

// Uma pitada de reverb
{
  Out.ar(~masterBus, FreeVerb.ar(In.ar(~reverbBus, 2), mix: 0.5, room: 0.9))
}.play(target: ~effects);

// Um controle de volume com o mouse só porque é legal
{
  Out.ar(0, In.ar(~masterBus, 2) * MouseY.kr(0, 1))
}.play(target: ~master);
)
```

Para mais informações sobre ordem de execução, consulte os arquivos de ajuda “Synth”, “Order of Execution” e “Group.”

Parte V
E AGORA?

Se você leu e mais ou menos entendeu tudo neste tutorial até agora, você já não é mais um iniciante em SuperCollider! Cobrimos um monte de material, e daqui pra frente você tem todas as ferramentas básicas necessárias para começar a desenvolver seus projetos pessoais, e continuar a aprender mais por conta própria. As seções a seguir fornecem uma breve introdução a alguns tópicos comuns de dificuldade intermediária. A última seção oferece uma lista concisa de outros tutoriais e fontes de aprendizado.

43 MIDI

Uma apresentação aprofundada dos conceitos e truques de MIDI está além do escopo deste tutorial. Os exemplos abaixo assumem alguma familiaridade com dispositivos MIDI e são fornecidos apenas como uma introdução.

```
1  // Jeito rápido de conectar todos os dispositivos disponíveis ao SC
2  MIDIIn.connectAll;
3
4  // Jeito rápido de ver todas as mensagens MIDI que estão chegando
5  MIDIFunc.trace(true);
6  MIDIFunc.trace(false); // pare de rastrear
7
8  // Jeito rápido de inspecionar todas as entradas CC
9  MIDIdef.cc(\someCC, {arg a, b; [a, b].postln});
10
11 // Obter entrada somente do cc 7, canal 0
12 MIDIdef.cc(\algumControleEspecifico, {arg a, b; [a, b].postln}, ccNum: 7, chan: 0);
```

```
13
14 // Uma SynthDef para testes rápidos
15 SynthDef("rápido", {arg freq, amp; Out.ar(0, SinOsc.ar(freq) * Env.perc(level: amp).
       kr(2))}).add;
16
17 // Toque de um teclado ou pad de percussão
18 (
19 MIDIdef.noteOn(\algumTeclado, { arg vel, nota;
20         Synth("rápido", [\freq, nota.midicps, \amp, vel.linlin(0, 127, 0, 1)]);
21 });
22 )
23
24 // Criar um Pattern e o inicie pelo teclado
25 (
26 a = Pbind(
27         \instrument, "rápido",
28         \degree, Pwhite(0, 10, 5),
29         \amp, Pwhite(0.05, 0.2),
30         \dur, 0.1
31 );
32 )
33
34 // teste
35 a.play;
36
37 // Disparar o Pattern de um pad ou teclado
38 MIDIdef.noteOn(\algumTeclado, {arg vel, note; a.play});
```

Uma dúvida frequente é como administrar mensagens de "nota ligada"e "nota desligada"("note on"e "note off") para notas sustentadas. Em outras palavras, quando você utiliza um envelope ADSR, você quer que cada nota seja sustentada enquanto a tecla estiver pressionada. O estágio

de repouso ("release") inicia apenas quando o dedo solta a tecla correspondente (revise a seção sobre envelopes ADSR se necessário).

Para fazer isso, o SuperCollider simplesmente precisa monitorar qual nó de sintetizador corresponde a cada tecla. Podemos usar uma array para este fim, como demonstrado no exemplo abaixo.

```
 // Uma SynthDef com envelope ADSR
SynthDef("rápido2", {arg freq = 440, amp = 0.1, gate = 1;
        var snd, env;
        env = Env.adsr(0.01, 0.1, 0.3, 2, amp).kr(2, gate);
        snd = Saw.ar([freq, freq*1.5], env);
        Out.ar(0, snd)
}).add;

// Toque com um teclado MIDI

(
var arrayDeNotas = Array.newClear(128); // array com uma posição para cada nota MIDI
     possível

MIDIdef.noteOn(\minhaTeclaApertada, {arg vel, nota;
        arrayDeNotas[nota] = Synth("rápido2", [\freq, nota.midicps, \amp, vel.linlin
            (0, 127, 0, 1)]);
        ["NOTA LIGADA", nota].postln;
});

MIDIdef.noteOff(\minhaTeclaLiberada, {arg vel, nota;
        arrayDeNotas[nota].set(\gate, 0);
        ["NOTA DESLIGADA", nota].postln;
});
)
// PS. Garanta que as conexões MIDI do SC estejam ativas (MIDIIn.connectAll)
```

Para ajudar a entender o código acima:

- A SynthDef `"rápido2"` usa um envelope ADSR. O argumento `gate` é responsável por ligar e desligar as notas.
- Um Array chamado `"arrayDeNotas"` é criado para monitorar quais notas estão sendo tocadas. Os índices do array devem corresponder aos números das notas MIDI sendo tocadas.
- Toda vez que uma tecla é pressionada no teclado, um Synth começa a tocar (um nó de sintetizador é criado no servidor) e *a referência a este nó de sintetizador é armazenada em uma posição exclusiva na array*; o índice da array é simplesmente o próprio número de nota MIDI.
- Sempre que a tecla é liberada, a mensagem `.set(\gate, 0)` é enviada para o nó de sintetizador apropriado, recuperado da array através do número da nota.

Nesta curta demonstração de MIDI apenas discutimos como enviar informação MIDI *para* o SuperCollider. Para enviar mensagens MIDI *a partir* do SuperCollider para algum outro programa ou equipamento MIDI, dê uma olhada no arquivo de Ajuda de `MIDIOut`.

44 OSC

OSC (Open Sound Control ou "Controle de Som Aberto") é uma excelente maneira de comunicar qualquer tipo de mensagem entre diferentes programas ou diferentes computadores em uma rede. Em muitos caos, é uma alternativa muito mais flexível às mensagens MIDI. Não temos espaço

para explicar isso em mais detalhes aqui, mas o exemplo abaixo deve servir como um bom ponto de partida.

O objetivo desta demonstração é mandar mensagens OSC de um smartphone para seu computador ou de um computador para outro computador.

No computador receptor, rode este fragmento simples de código:

```
(
OSCdef(
        key: \seiLa,
        func: {arg ...args; args.postln},
        path: '/coisas')
)
```

Nota: pressionando [ctrl+.] interromperá o `OSCdef` e você não receberá mais mensagens.

44.1 Mandando OSC para um outro computador

Para esta demonstração, partimos do princípio que ambos os computadores estejam rodando o SuperCollider e conectados à mesma rede. Descubra o endereço IP do computador receptor e rode as seguintes linhas no computador emissor:

```
// Use isso na máquina que está mandando mensagens
~destino = NetAddr("127.0.0.1", 57120); // use o endereço IP correto para o
    computador de destino

~destino.sendMsg("/coisas", "aaloooo");
```

44.2 Mandando OSC de um smartphone

- Instale qualquer aplicativo de OSC no telefone (por exemplo, gyrosc);
- Entre o endereço IP do computador receptor no aplicativo OSC (como "target");
- Entre a porta de entrada do SuperCollider no app OSC (geralmente 57120);
- Verifique o caminho de mensagem ("message path") que o aplicativo usa para mandar OSC e mude o seu OSCdef de acordo;
- Tenha certeza que seu telefone está conectado à mesma rede sem fio que o computador

Desde que seu telefone esteja enviando mensagens para o caminho correto, você deve vê-las chegando no computador.

45 Quarks e plug-ins

Você pode aumentar a funcionalidade do SuperCollider adicionando classes e UGens criados por outros usuários. *Quarks* são pacotes de classes de SuperCollider, expandindo o que você pode fazer na linguagem do SuperCollider. *UGen plug-ins* são extensões para o servidor de síntese de áudio do SuperCollider.

Visite `http://supercollider.github.io/` para obter informações atualizadas sobre como adicionar plug-ins e quarks à sua instalação do SuperCollider. O arquivo de Ajuda "Using Quarks"é também um bom ponto de partida: `http://doc.sccode.org/Guides/UsingQuarks.html`. De qualquer documento do SuperCollider você pode rodar `Quarks.gui` para ver uma lista de todos os quarks disponíveis (ela abre em uma nova janela).

46 Referências Adicionais

Chegamos ao fim desta introdução ao SuperCollider. Algumas referências adicionais para estudo estão listadas aqui. Aproveite!

- Uma excelente série de tutoriais no YouTube por Eli Fieldsteel: `http://www.youtube.com/playlist?list=PLPYzvS8A_rTaNDweXe6PX4CXSGq4iEWYC`.
- O tutorial padrão para começar no SC com Scott Wilson e James Harkins, disponível online e incluído nos arquvos de Help: `http://doc.sccode.org/Tutorials/Getting-Started/00-Getting-Started-With-SC.html`
- Tutorial online de Nick Collins: `http://composerprogrammer.com/teaching/supercollider/sctutorial/tutorial.html`
- A lista de e-mails oficial do SuperCollider é a melhor maneira de conseguir uma ajuda amistosa de um grande grupo de usuários. Iniciantes são muito bem vindos para fazer perguntas nesta lista. Você pode se inscrever aqui: `http://www.birmingham.ac.uk/facilities/BEAST/research/supercollider/mailinglist.aspx`
- Descubra um grupo local de SuperCollider na sua cidade. A lista oficial de usuários do SC é a melhor maneira de descobrir se existe uma onde você mora. Se não há um grupo na sua área, comece um!
- Muitos exemplos interessantes de códigos podem ser encontrados aqui: `http://sccode.org/`. Crie uma conta e compartilhe seus códigos também.
- Grupo brasileiro de SuperCollider no Facebook: `https://www.facebook.com/groups/630981953617449/`.

- Já ouviu falar nos tweets de SuperCollider? `http://supercollider.github.io/community/sc140.html`

Notas

[1]Primeira pergunta: quando você usa o número 1 em vez de `inf` como o argumento `repeats` do segundo `Pseq`, o Pbind para depois que 6 notas foram tocadas (isto é, depois que uma sequência completa de valores de duração foi tocada). Segunda questão: para fazer o Pbind tocar para sempre, simplesmente use `inf` como valor para `repeats` em todos os Patterns internos.

[2]

a) `Pwhite(0, 10)` vai gerar qualquer número entre 0 e 10. `Prand([0, 4, 1, 5, 9, 10, 2, 3], inf)` vai escolher apenas números que estão contidos na lista; note que essa lista tem *alguns* números entre 0 e 10, mas não todos (6, 7, 8 não estão lá, então nunca vão aparecer neste `Prand`).
b) Tecnicamente você poderia usar um `Prand` se você fornecer uma lista com todos os números entre 0 e 100, mas faz mais sentido usar um `Pwhite` para esta tarefa: `Pwhite(0, 100)`.
c) `Prand([0, 1, 2, 3], inf)` escolhe itens da lista aleatoriamente. `Pwhite(0, 3)` chega ao mesmo resultado por outros meios: ele gera aleatoriamente números inteiros entre 0 e 3, o que acaba dando o mesmo leque de opções que o `Prand` acima. No entanto, se você escrever `Pwhite(0, 3.0)`, o resultado será diferente: como um dos argumentos de entrada do `Pwhite` está escrito como um decimal (3.0), esse Pwhite produzirá qualquer número decimal entre 0 e 3, como 0.154, 1.0, 1.45, 2.999.
d) O primeiro `Pbind` toca 32 notes (4 vezes a sequência de 8 notas). O segundo `Pbind` toca apenas 4 notas: quatro escolhas aleatórias retiradas da lista (lembre-se que o `Prand`, diferentemente do `Pseq`, não tem a obrigação de tocar todas as notas da lista: ele vai simplesmente escolher tantos números aleatórios quanto você pedir). O terceiro e último `Pbind` toca 32 notas, como o primeiro.

[3]Primeira linha: o Array `[1, 2, 3, "uau"]` é o objeto recebedor; `reverse` é a mensagem. Segunda linha: a String "alô"é o objeto recebedor; `dup` é a mensagem; `4` é o argumento para `dup`. Terceira linha: 3.1415 é o objeto recebedor; `round` é a mensagem; `0.1` é o argumento para `round`. Quarta linha: `100` é o objeto recebedor, `rand` é a mensagem. Última linha: `100.0` é o recebedor da mensagem `rand`, o resultado do qual é um número aleatório entre 0 e 100. Este número se torna o recebedor da mensagem `round` com o argumento `0.01`, assim o número aleatório é arredondado em duas casas decimais. Daí este resultado se torna o objeto recebedor da mensagem `dup` com o argumento `4`, que cria uma lista com quatro duplicatas daquele número.

[4]Reescrevendo usando apenas notação funcional: `dup(round(rand(100.0), 0.01), 4);`

[5] Respostas:

a) 24

b) [5, 5.123] (tanto números quanto colchetes)

c) Toda a linha do `LFSaw`

d) Somente um

e) 0.4

f) 1 e 0.3

[6] `SinOsc` é bipolar porque produz números entre 1- e +1. `LFPulse` é unipolar porque o âmbito de saída é 0-1 (repare que no caso do `LFPulse` em particular, somente "zeros"e "uns"são gerados, sem nenhum número intermediário)

[7] Solução: `a = {Out.ar(0, SinOsc.ar(freq: [800, 880], mul: LFPulse.ar([2, 3])))}.play;`

[8] (a) A variável `lfn` simplesmente armazena um `LFNoise2`. O papel do `LFNoise2` é gerar aleatoriamente um novo número (entre -1 e +1) a cada segundo e deslizar até ele a partir do último número (diferentemente do `LFNoise0`, que salta para o novo número imediatamente). O primeiro uso desta variável `lfn` é no argumento `freq` do BPF: `lfn.range(500, 2500)`. Isso pega os números entre -1 e +1 e os redimensiona para o âmbito 500-2500. Estes números são então usados como a frequência central do filtro. Estas frequências são as alturas que escutamos deslizando para cima e para baixo. Finalmente, `lfn` é usada novamente para controlar a posição do pan `Pan2`. Ela é usada diretamente (sem uma mensagem `.range`) porque os números já estão no âmbito que queremos (de -1 a +1). O resultado interessante disso é que associamos a mudança de frequência com a mudança de posição. Como? A cada segundo, `LFNoise2` começa a deslizar em direção a um novo número aleatório e isso se torna uma mudança sincronizada na frequência do filtro e na posição de pan. Se tivéssemos dois `LFNoise2` diferentes, um em cada lugar, as mudanças não teriam relação entre si (o que poderia ser bom também, mas o resultado sonoro é distinto).
(b) Um `mul:` de 1 soaria fraco demais. Como o filtro é bastante estreito, ele retira tanto do sinal original que a amplitude sofre uma queda exagerada. Precisamos então aumentar o sinal de volta para um âmbito razoavelmente audível, então é por isso que temos `mul: 20` ao final da linha do `BPF`.
(c) O ritmo é controlado pelo `LFPulse` que é o argumento `mul:` do `Saw`. A frequência do `LFPulse` (quantos pulsos por segundo) é controlada por um `LFNoise1` que produz números de 1 a 10 (interpolando entre eles). Tais números são as "quantas notas por segundo"deste patch.