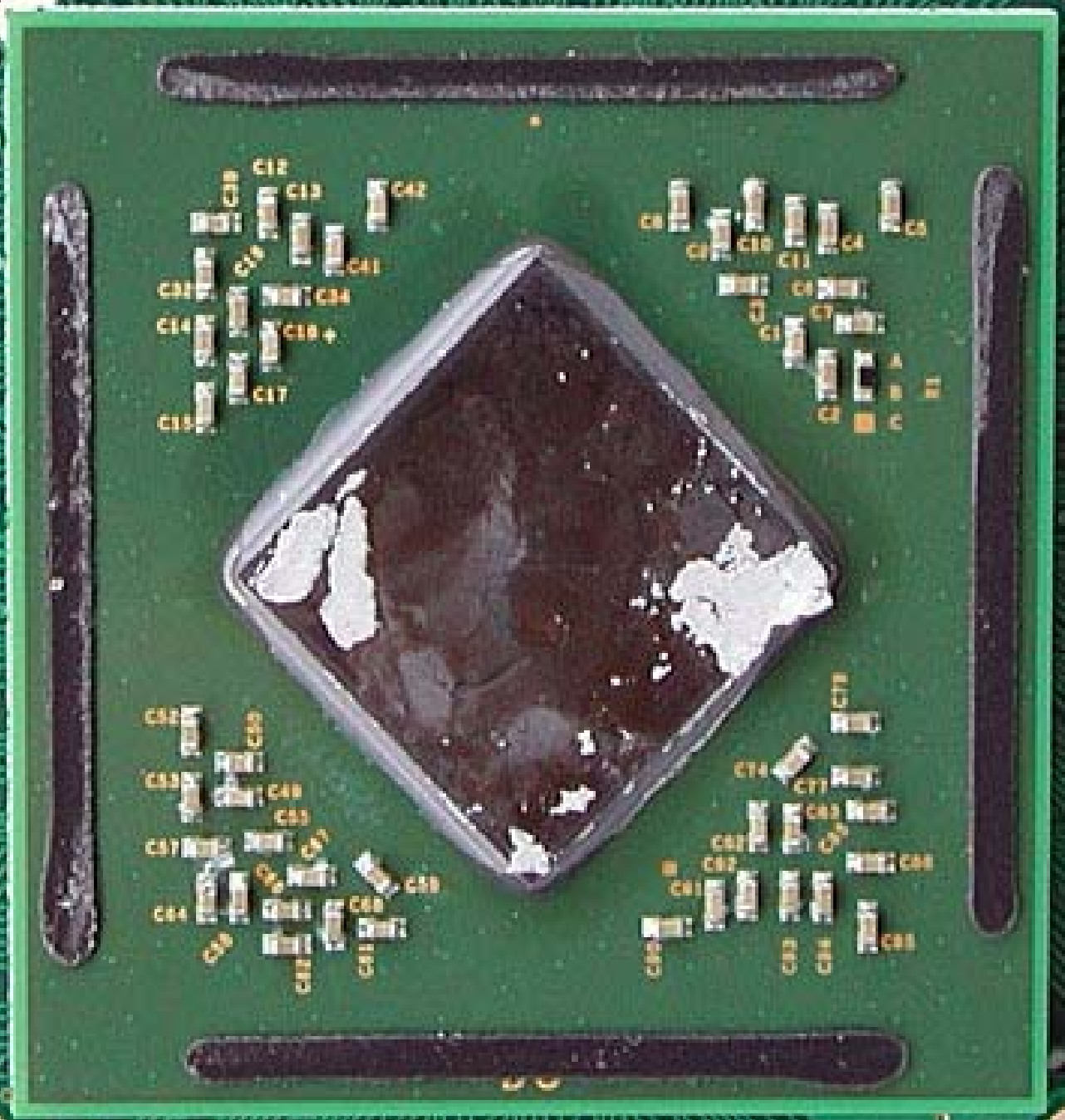
Guia Prático de CSound

Robson Cozendey



Guia Prático de
CSound

Robson Cozendey

Sumário

Capítulo 1
Primeiros Instrumentos

Csound tem uma capacidade de criação de sons apenas limitada pelo programador, e dada a abrangência do programa iremos nesse capítulo aprender sua estrutura e sintaxe, usando exemplos simples a partir de senóides.

Configurando o Sistema

Explicaremos nessa seção como configurar o Csound para ser executado no Windows e no Unix.

Antes de executar o Csound no Windows, as variáveis de ambiente **OPCODEDIR** e **OPCODEDIR64** devem estar definidas com o caminho das bibliotecas do Csound. Se o Csound está instalado por exemplo em *c:\csound*, tipicamente devem ser definidas as variáveis de ambiente como:

```
OPCODEDIR=c:\csound\plugins
OPCODEDIR64=c:\csound\plugins64
```

Além disso, o diretório *bin* de Csound, no nosso exemplo *c:\csound\bin*, deve estar na variável de ambiente **PATH**. Por exemplo, no Windows poderíamos ter:

```
PATH=%PATH%;C:\csound5\bin
```

Para realizarmos essa configuração no Unix, devemos copiar o executável *csound* para */usr/local/bin* e as bibliotecas do Csound devem ser copiadas para */usr/local/lib*.

A Estrutura dos Arquivos-Fonte

Csound tem uma estrutura em relação aos arquivos bastante versátil. Existem dois métodos para se executar Csound, podemos usar o modelo de dois arquivos-fonte ou o modelo de um único arquivo-fonte.

Nesse primeiro contato com o programa, para entendermos melhor como ele funciona, iremos usar o modelo de dois arquivos-fonte separados. Nesse modelo são usados dois arquivos para que o Csound renderize um arquivo de saída de áudio, esses dois arquivos são o arquivo de *orchestra*, que contém as definições dos instrumentos, e o arquivo de *score*, que contém a "partitura" da música, com suas notas, durações, intensidades, etc. O arquivo de saída é do tipo *.wav* ou *.aiff*, para execuções que não são em tempo real, ou a saída é diretamente nos alto-falantes, quando não especificarmos um arquivo de saída.

Convencionalmente quando vai-se executar o Csound com esses dois arquivos-fonte, é usado o mesmo nome para ambos, diferenciando-os pelas extensões *.orc* para o arquivo orchestra e *.sco* para o arquivo score. Por exemplo, se quisermos o arquivo de saída *seno.wav* a partir dos arquivos *seno.orc* e *seno.sco*, digitaremos algo como:

```
csound –o seno.wav seno.orc seno.sco
```

Com a opção *–o* dizemos qual arquivo de saída desejamos, nesse caso o arquivo de saída será *seno.wav,* e em seguida dizemos qual o arquivo-fonte da orchestra, *seno.orc,* e o arquivo-fonte do score, *seno.sco.*.

Primeiro exemplo: *seno.wav*

Nesse primeiro exemplo queremos apenas produzir uma onda de seno na frequência do Lá do diapasão, 440Hz, e através do exemplo poderemos ver como os arquivos de orchestra e score conversam entre si, para produzir o arquivo de saída. Aqui o arquivo score dirá:

1. A forma de onda inicial a ser usada, nesse caso uma senóide.
2. O instrumento a ser usado em cada nota
3. O tempo de início de cada nota
4. A duração de cada nota

Como você pode ver, o arquivo de score fornece os *parâmetros* que o arquivo de orchestra usará, e o arquivo de orchestra define como esses parâmetros serão usados para efetivamente gerar o som de saída. Aqui está o arquivo de score com sua sintaxe:

```
/* seno.sco */
; Tabela #1: uma simples onda de seno usando GEN10.
f 1 0 16384 10 1

; Toca o instrumento #1 por 2 segundos, começando em 0 segundos
i 1 0 2
e
/* seno.sco */
```

Fig. 1: *seno.sco,* o arquivo score para uma senóide

A sintaxe do arquivo *score*

Na fig.1 temos os comentários entre "/*" e "*/", ou após ";". Na terceira linha temos o que é chamado de f-table, ou *function table*, que é uma tabela indicando a forma de onda que os instrumentos poderão usar mais tarde. Nesse caso essa tabela conterá os valores para a onda de seno. A sintaxe de uma f-table é:

```
f índice  tempo de início  tamanho  GEN#  parâmtro1 parâmetro2 ...
```

Na Fig.1 temos essa sintaxe na linha:

```
f 1 0 16384 10 1
```

Então olhando para a linha acima podemos ver que a f-table tem índice 1, é carregada após 0 segundos, tem o tamanho de 16384 amostras (ou pontos), usa a rotina GEN10 para gerar a onda que será armazenada na tabela, e tem o número 1 como primeiro parâmetro para a GEN10.

Todas as f-table como a acima são preenchidas com formas de onda gerada pelas rotinas GEN do Csound, que neste presente momento vão de GEN1 até GEN52, e para usá-las basta especificar o seu número no quarto parâmetro da declaração da f-table. Cada rotina GEN gera uma forma de onda diferente. As mais importantes, GEN1 e GEN10, respectivamente carregam um arquivo de áudio ou senóides em uma tabela.

Estamos portanto usando a GEN10 no exemplo acima para preencher os 16384 pontos da f-table com os valores do seno. O parâmetro 1 que vem após o 10 diz que deve ser usada apenas uma onda de seno na tabela, e não vários harmônicos sobrepostos.

Vale ressaltar que o tamanho das f-tables são sempre em potências de 2, i.e., 1024, 2048, 4096, 8192, 16384 pontos, etc.

Após as declarações das f-tables temos a sintaxe da declaração de nota, que é:

```
; p1                      p2      p3        p4           p5
i número do instrumento   início  duração   parâmetro1   parâmetro2   ...
```

Novamente na Fig.1 temos essa declaração na linha:

```
i 1 0 2
```

Portanto na sexta linha da Fig.1 temos que o instrumento 1 será tocado por 2 segundos, começando em 0 segundos, imediatamente no início da música.

Finalmente a letra “e” indica que chegamos ao final do arquivo score. Vejamos agora como essas declarações serão usadas no arquivo de orchestra, onde o som é produzido.

A sintaxe do arquivo *orchestra*

Na fig. 2 temos a listagem do arquivo orchestra, que será usado em conjunto com o arquivo score visto acima, para nosso primeiro instrumento:

```
/* seno.orc */
; Inicializa as variáveis globais.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrumento #1.
instr 1
  ; Amplitude do sinal
  kamp = 30000
  ; Frequência
  kcps = 440
  ; Número da f-table.
  ifn = 1

  ; Toca com amplitude 30000 e frequência de 440 Hz a onda do seno
  ; armazenada na tabela 1.
  a1 oscil kamp, kcps, ifn

  ; Manda o som armazenado em a1 para o arquivo de saída, seno.wav
  out a1
endin
/* seno.orc */
```

Fig. 2 : *seno.orc* o arquivo orchestra para uma senóide

O cabeçalho no arquivo orchestra

A listagem começa com o cabeçalho, presente em todo aquivo orchestra. Aqui declaramos as variáveis globais *sample rate* (taxa de amostragem) como **sr** = 44100 amostras por segundo, *kontrol rate* (taxa de controle) como **kr** = 4410 amostras por

segundo, e a razão sample rate dividido por control rate como **ksmps** = 10, e finalmente declaramos o número de canais como **nchnls** = 1, ou seja, o som será mono.

O Csound faz uma diferença entre taxa de amostragem **sr** e taxa de controle **kr**. A taxa de amostragem **sr** é a quantidade de vezes por segundo que pontos de uma tabela são gerados para produzir o áudio da onda contida nessa tabela. A taxa de controle **kr** é a quantidade de vezes por segundo que uma onda é modificada em um instrumento, tipicamente bem menor que a taxa de amostragem. Finalmente em **ksmps** temos a razão entre **sr** e **kr**.

Então não precisamos que a cada ponto lido da tabela essa onda seja alterada, mas como no exemplo acima, essa onda pode ser modificada a cada 10 pontos lidos, já que a taxa de modificação **kr** é a taxa de controle que é 10 vezes menor que a taxa de amostragem, o que economiza em muito o processamento da onda e não afeta a percepção de quem ouve.

Nomenclatura de Variáveis

As variáveis usadas na orchestra podem ser estáticas ou dinâmicas, locais ou globais, dependendo do prefixo que possuem. Veremos em mais detalhes a seguir o que isso significa. No nosso aquivo orchestra temos o opcode *oscil* usado como:

```
a1 oscil kamp, kcps, ifn
```

No Csound as variáveis podem ter qualquer nome, mas devem sempre começar com os prefixos "**k**"," **a**", "**i**", ou "**g",** e isto tem significado para o interpretador.

As variáveis com o prefixo "**i**", como em **ifn**, tem espaço para apenas um valor, isto é, não são tabelas. É uma variável tradicional, como em outras linguagens de programação.

Variáveis com o prefixo "**k**", como em **kamp**, são variáveis dinâmicas que são atualizadas automaticamente com a frequência da taxa de controle, em nosso exemplo ela é lida 4410 vezes por segundo, avançando um ponto na variável a cada leitura. Essas variáveis são ditas **k-rate**, pois seguem a taxa de controle, como no exemplo de **kr** = 4410. Tecnicamente essas variáveis são simples tabelas, e Csound avança lendo **kr** pontos dessa tabela por segundo, e elas são criadas com tamanho suficiente para durar durante toda a nota, fornecendo novos pontos. São normalmente usadas para controlar e modificar parâmetros de uma onda, como por exemplo alterando a amplitude dessa onda ao longo da duração da nota. Aqui **kamp** não tem um valor fixo, mas pode ser uma tabela com valores entre 0 e 30000. Então o oscilador irá reproduzir uma onda que começa com amplitude 0 e cresce até a amplitude final 30000.

Variáveis com o prefixo "**a**", como em **a1**, são variáveis dinâmicas do tipo **a-rate**, e são similares às variáveis **k-rate,** mas são atualizadas segundo a taxa de amostragem, em nosso exemplo, na taxa de **sr** = 44100 vezes por segundo. Pela sua elevada fidelidade de som e alto custo computacional, são usadas apenas para sinais de áudio, ou o sinal final que viremos a ouvir.

Podemos ainda multiplicar, por exemplo, uma váriável **k-rate** de 100 pontos por uma variável **a-rate** de 1000 pontos, e elas gerarão uma variável **a-rate** como uma tabela de 1000 pontos modificada a cada 10 pontos.

Todas essas váriáveis são locais e existem apenas dentro do instrumento. Se quisermos que uma variável seja global e vista por todos os instrumentos, devemos declará-la no cabeçalho da seção orchestra, e usar o prefixo "**g"** antes de qualquer outro prefixo. Nossas variáveis usadas acima ficariam então **gifn, gkamp** e **ga1,** e deveriam estar definidas no cabeçalho da orchestra.

Declaração do Instrumento

Logo depois do cabeçalho vem a declaração de início do nosso primeiro instrumento com a palavra-chave **instr** seguida do número do instrumento, nesse caso 1.

Por questão de legibilidade usamos aqui uma boa prática: definimos os parâmetros **kamp, kcps** e **ifn** como variáveis antes de usá-los no opcode **oscil**. O opcode **oscil** tem como parâmetros respectivamente a amplitude, a frequência e a forma de onda a ser usada. Como vimos antes em **oscil** os parâmetros **kamp** e **kcps** são variáveis de controle, e **ifn** é um parâmetro de inicialização.

No arquivo orchestra vemos que a amplitude foi definida como 30000, a frequência como 440 Hz e a forma de onda é indicada através do índice 1 da f-table. Como já armazenamos no arquivo score uma senóide na f-table 1, o opcode *oscil* aqui apenas processará essa forma de onda para ter amplitude 30000 e frequência 440 Hz, uma nota de Lá no diapasão. No lado esquerdo a onda de saída é armazenada na variável **a1 .**

Na linha seguinte o opcode **out** grava a onda armazenada em **a1** no arquivo de saída especificado na linha de comando, *seno.wav*, ou manda o sinal diretamente para as caixas de som, se nenhum arquivo de saída fôr especificado na linha de comando. Finalmente no arquivo orchestra temos a declaração de fim de instrumento, **endin**.

Então ao executarmos esses dois arquivos obteremos um arquivo *seno.wav* que contém o som de uma senóide a 440 Hz tocada por 2 segundos. Apesar da simplicidade desse primeiro exemplo, ele teve o propósito de esclarecer várias questões em relação ao que é feito no arquivo score e no arquivo orchestra. Note que o mesmo arquivo orchestra pode ser usado para vários arquivos scores diferentes, um mesmo instrumento tocando várias partituras diferentes.

Parâmetros de Performance e de Inicialização

Como dito acima em **oscil** os parâmetros **kamp** e **kcps** são dinâmicos, isso quer dizer na terminologia do Csound que eles são parâmetros de *performance*, ou seja, podem ser mudados em tempo de execução. O outro tipo possível de parâmetros são os parâmetros de *inicialização,* que podem ter apenas um valor e são definidos assim que a nota é iniciada. No caso de **oscil**, o parâmetro **ifn** é de inicialização.

Integrando os arquivos *orchestra* e *score:* o arquivo unificado .CSD

Falamos no início do capítulo que havia duas maneiras de executar o Csound: com dois arquivos-fonte orchestra e score separados, ou com um único arquivo-fonte. Vamos ver agora a segunda maneira.

À primeira vista o uso de arquivos orchestra e score é bastante intuitivo, pois são isolados as duas principais etapas de renderização do Csound. Entretanto o uso constante do programa, fez com que fôsse desenvolvido o formato de um único arquivo, com seções para a orchestra e para o score, tornando-se assim um formato mais compacto e fácil de se organizar quando se tem muitos arquivos.

Trata-se do formato .**CSD**, que contém em um só arquivo três seções: **CsOptions** para opções de linha de comando, **CsInstruments**, para os instrumentos que eram armazenados no arquivo orchestra, e **CsScore,** a seção que conterá o que era armazenado no arquivo score. Assim, nosso exemplo anterior *seno.orc* e *seno.sco* se tornariam um só arquivo CSD, como vemos na figura 3:

```
<CsoundSynthesizer>

<CsOptions>

-o seno.wav

</CsOptions>

<CsInstruments>
; Inicializa as variáveis globais.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrumento #1.
instr 1
  ; Amplitude do sinal
  kamp = 30000
  ; Frequência
  kcps = 440
  ; Número da f-table.
  ifn = 1

  ; Toca com amplitude 30000 e frequência de 440 Hz a onda do seno
  ; armazenada na tabela 1.
  a1 oscil kamp, kcps, ifn

  ; Manda o som armazenado em a1 para o arquivo de saída, seno.wav
  out a1
endin

</CsInstruments>

<CsScore>

; Tabela #1: uma simples onda de seno usando GEN10.
f 1 0 16384 10 1

; Toca o instrumento #1 por 2 segundos, começando em 0 segundos
i 1 0 2
e

</CsScore>
```

```
</CsoundSynthesizer>
```

Fig.3: *seno.csd*, nosso primeiro arquivo csd, nesse caso a versão exata em *seno.csd* dos arquivos anteriores *seno.orc* e *seno.sco*.

Note que além da orchestra e do score ficarem encapsulados nas seções **CsInstruments** e **CsScore**, agora a nossa antiga opção de linha de comando "-o seno.wav" fica dentro da seção **CsOptions**, e nossa linha de comando para executar o arquivo se reduz para:

```
csound seno.csd
```

Daqui para frente usaremos apenas arquivos CSD, mas tenham em mente que muitos usuários ainda usam os formatos orchestra e score, e esse formato é compatível com qualquer versão de Csound.

Especificação de parâmetros e conversão de valores

No arquivo anterior *seno.csd* as únicas informações que passamos do score para a orchestra são o número do instrumento, o tempo de início da nota e sua duração. Esses três parâmetros são obrigatórios em qualquer score, e são referenciados na seção orchestra respectivamente por **p1, p2** e **p3**. Assim se quisermos que uma variável **idur** contenha a duração da nota que está sendo executada, basta declarar dentro da seção orchestra a linha:

```
idur = p3
```

Esses três primeiros parâmetros são reservados pelo Csound, assim **p1** sempre é o número do intrumento, **p2** o tempo de início da nota em segundos e **p3** a duração em segundos. Mas podemos definir qualquer número de parâmetros adicionais, começando por **p4**.

Em *seno.csd* definimos a frequência do oscilador dentro da seção orchestra através da declaração **kcps =** 440. Seria melhor se nosso instrumento pudesse tocar qualquer frequência, e pudéssemos especificar essa frequência na seção score, dando uma frequência diferente a cada nota do instrumento.

Até aqui temos usado sempre a frequência fixa de Lá 4, isto é, 440 Hz, mas usaremos agora a frequência como um parâmetro dado pelo score. Se tornaria muito trabalhoso definir as notas musicais pela frequência em Hertz, então usaremos o conversor de valores **cpspch,** que tem como parâmetro a nota em *octave-point-pitch-class*, que é um formato intuitivo para definir notas.

Nesse formato valores inteiros como 7.00, 8.00, 9.00 definem a oitava em que se está, e as duas casas decimais definem a nota dentro da oitava. Temos por exemplo que 7.00 é o Dó central (C3), e que 7.01 é o Dó sustenido nessa oitava (C3#), e 7.02 é o Ré (D3), e etc. Assim cada centésimo a mais corresponde a um semitom acima, e portanto nesse exemplo para cobrir as doze notas dessa oitava haverá a variação entre 7.00 e 7.11. Pode-se especificar também milésimos como 7.015 para a nota um quarto de tom entre C#3 e D3, em composições microtonais.

Veja como ficará nosso arquivo *cpspch.csd*, usando essa notação para diferentes notas renderizadas pelo instrumento. Usando a notação *octave-point-pitch-*

class tocaremos a escala de Dó Maior em nosso oscilador, e dentro da orchestra faremos a conversão dessa notação em Hertz usando **cpspch.**

```
<CsoundSynthesizer>

<CsOptions>

-o cpspch.wav
</CsOptions>

<CsInstruments>
; Inicializa as variáveis globais.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrumento #1.
instr 1
  ; Amplitude do sinal
  kamp = 30000
  ; Frequência especificada pelo score
  kcps = cpspch(p4)
  ; Número da f-table.
  ifn = 1

  ; Toca com amplitude 30000 e frequência dada pelo score a onda do seno
  ; armazenada na tabela 1.
  a1 oscil kamp, kcps, ifn

  ; Manda o som armazenado em a1 para os alto-falantes
  out a1
endin

</CsInstruments>

<CsScore>

; Tabela #1: uma simples onda de seno usando GEN10.
f 1 0 16384 10 1

; Toca o instrumento #1 com cada nota durando um décimo de segundo e
; percorrendo toda a escala de Dó maior.
i 1 0 0.1 8.00
i 1 0.1 0.1 8.02
i 1 0.2 0.1 8.04
i 1 0.3 0.1 8.05
i 1 0.4 0.1 8.07
i 1 0.5 0.1 8.09
i 1 0.6 0.1 8.11
i 1 0.7 0.1 9.00
e

</CsScore>

</CsoundSynthesizer>
```

Fig. 4: *cpspch.csd*

Capítulo 2

A série harmônica

Os exemplos anteriores foram úteis para entendermos como funciona Csound, mas vamos agora aplicar algumas técnicas para ir tornando nossos instrumentos mais interessantes.

Normalmente quando uma nota é tocada os instrumentos acústicos produzem a chamada *série harmônica*, que é uma onda composta da frequência fundamental acompanhada por várias frequências a partir desta. Se tocarmos a nota Dó em um piano, tipicamente as frequências produzidas serão a fundamental Dó acompanhada de seus harmônicos:

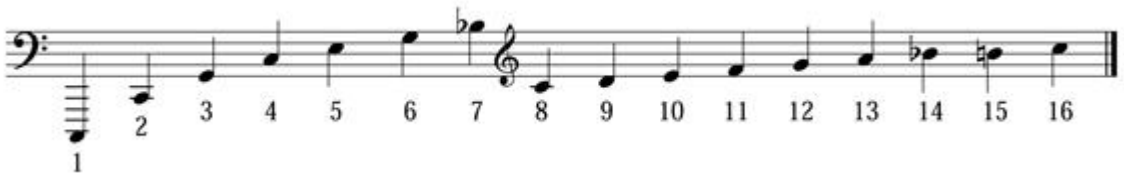


Fig. 5: A série harmônica

Essas frequências que soam juntas com a frequência fundamental Dó, não são notadas como frequências separadas, mas é o que dá o timbre ao som quando uma única nota é tocada. Praticamente todos os instrumentos acústicos possuem a mesma série harmônica, entretanto o que diferencia o timbre de cada um é a intensidade com que cada harmônico, ou frequência superior, é reproduzido. Uma flauta doce tem pouca intensidade em seus harmônicos, gerando um som mais puro, enquanto que um saxofone tem maior intensidade nos harmônicos superiores, produzindo um timbre mais complexo.

Gerando a série harmônica a partir de GEN10 e GEN11

Existem vários opcodes em Csound que tiram proveito da série harmônica para produzir sons mais complexos, entre eles GEN10, GEN11 e o opcode **buzz**. Se olharmos para a forma como a GEN10 foi utilizada no nosso exemplo anterior veremos que ela foi escrita com um único parâmetro depois da especificação 10:

```
f 1 0 16384 10 1
```

Na verdade dissemos com esse último parâmetro 1 que apenas a frequência fundamental deveria ser produzida. Vejamos agora um exemplo onde teremos os primeiros 5 harmônicos da série harmônica reproduzidos, todos com intensidade 1:

```
<CsoundSynthesizer>

<CsOptions>

-o gen10.wav

</CsOptions>
```

```
<CsInstruments>
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

instr 1
  kamp = 30000
  kcps = 440
  ifn = 1

  ; Toca com amplitude 30000 e frequência de 440 Hz a onda dos 5
  ; primeiros harmônicos armazenados na tabela 1.
  a1 oscil kamp, kcps, ifn

  out a1
endin

</CsInstruments>

<CsScore>

; Tabela #1: Primeiros 5 harmônicos com GEN10, todos com a mesma
; intensidade.
f 1 0 16384 10 1 1 1 1 1

; Toca o instrumento #1 por 3 segundos, começando em 0 segundos
i 1 0 3
e

</CsScore>

</CsoundSynthesizer>
```

Fig. 6: *GEN10.csd,* O arquivo csd para uma série harmônica usando GEN10

Se quiséssemos apenas seis harmônicos, mas que não tivesse o segundo harmônico, que o terceiro harmônico tivesse a metade da intensidade da fundamental, e todos os harmônicos seguintes um quarto da intensidade, usaríamos:

```
f 1 0 16384 10 1 0 0.5 0.25 0.25 0.25 0.25
```

A principal diferença entre GEN10 e GEN11 é que em GEN10 definimos a intensidade de cada harmônico, enquanto em GEN11 podemos definir todos os harmônicos com a mesma intensidade dizendo apenas o número de harmônicos. Se em GEN10 usamos a linha de comando:

```
f 1 0 16384 10 1 1 1 1 1
```

para os cinco primeiros harmônicos com a mesma intensidade, podemos criar a mesma f-table com GEN11 através da linha:

```
f 1 0 16384 11 5
```

Gerando a série harmônica a partir do opcode *buzz*

Uma alternativa para se produzir o som da série harmônica, é gerá-lo na seção orchestra, a partir do opcode **buzz**, ao invés de gerá-lo no arquivo score através de GEN10.

Se olharmos no *Csound Reference Manual*, veremos que a sintaxe de buzz é:

```
ares buzz xamp, xcps, knh, ifn [, iphs]
```

Onde os parâmetros dados são:

. xamp é a amplitude de saída

. xcps é a frequência

. knh é o número de harmônicos a ser reproduzido

. ifn é o índice de uma f-table que contém a onda do seno

. iphs (opcional) é a fase em que as ondas de seno começarão

Os parâmteros **ifn** e **iphs** são de tempo de inicialização, os outros parâmetros são de tempo de performance. Aqui temos o seu uso no arquivo *buzz.csd:*

```
<CsoundSynthesizer>

<CsOptions>

-o buzz.wav

</CsOptions>

<CsInstruments>
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

instr 1
  kamp = 5000
  kcps = 440
  ifn = 1
  ; Número de harmônicos
  knh = 5

  ; Toca com amplitude 5000, a partir da frequência fundamental de
  ; 440 Hz, a onda dos 5 primeiros harmônicos do seno
  a1 buzz kamp, kcps, knh, ifn

  out a1
endin

</CsInstruments>

<CsScore>

  ; Tabela #1: uma simples onda de seno usando GEN10.
  f 1 0 16384 10 1

  ; Toca o instrumento #1 por 2 segundos, começando em 0 segundos
  i 1 0 2
  e

</CsScore>

</CsoundSynthesizer>
```

Fig. 5: *buzz.csd*, um arquivo csd utilizando o opcode buzz para gerar a série harmônica

Podemos então ouvir a mesma onda gerada antes por GEN10, agora gerada por buzz, através da linha de comando:

```
csound buzz.csd
```

A primeira vista parece que os opcodes GEN10, GEN11 e buzz são quase idênticos, mas uma de suas diferenças fundamentais é que ondas geradas em uma f-table como nas rotinas GEN são estáticas, enquando que a onda gerada pelo opcode buzz é dinâmica, pode ser alterada a qualquer momento da execução. Podemos, por exemplo, começar com 10 harmônicos no opcode buzz e a cada segundo diminuirmos em um os harmônicos, o que é impossível de se fazer com a f-table.

Capítulo 3

Ataque e Decaimento

Em nenhum instrumento real o som se mantém com uma amplitude constante, normalmente temos variações de intensidade, tipicamente mais altos no início e silenciando ao longo do tempo. Para tornar os instrumentos mais factíveis, podemos controlar essa amplitude usando as funções de *envelope*.

Uma função de envelope é um formato de onda que delinea a amplitude que a onda de saída deve ter. De uma certa forma, ela *empacota* a amplitude da onda de saída como um envelope real.

No Csound para usarmos o efeito de ataque e decaimento em uma nota, precisamos que a onda de saída seja modulada por um envelope. Um tipo de função de envelope muito usado para simular instrumentos acústicos é o ADSR, *attack-decay-sustain-release*. Como em um instrumento real, essa onda tem um ataque rápido, depois um leve decaimento, suntenta por algum tempo, e então silencia.

Funções de envelope usando o opcode *linseg*

Como vimos antes o parâmetro de amplitude **kamp** de **buzz** é um parâmetro de performance, então para aplicarmos o ADSR basta modularmos a amplitude de saída de **buzz**. Usaremos uma função de envelope no parâmetro **kenv**, ao invés de usarmos um valor fixo, modulando assim a amplitude de **buzz**.

```
<CsoundSynthesizer>

<CsOptions>

-o linseg.wav

</CsOptions>

<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

instr 1

  ; Amplitude do sinal dada pelo envelope de saída de linseg
  kenv linseg 0, p3/5, 5000, p3/5, 2500, p3 * 2/5, 2500, p3/5, 0

  ; Frequência fundamental
  kcps = 440
  ; Número da f-table.
  ifn = 1
  ; Número de harmônicos
  knh = 5

  ; Toca com amplitude variável kenv, a partir da frequência
```

```
  ; fundamental de 440 Hz, a onda dos 5 primeiros harmônicos do seno
  a1 buzz kenv, kcps, knh, ifn

  ; Manda o som armazenado em a1 para o arquivo de saída, linseg.wav
  out a1
endin

</CsInstruments>

<CsScore>

; Tabela #1: uma simples onda de seno.
f 1 0 16384 10 1

; Toca o instrumento #1 por 5 segundos, começando em 0 segundos
i 1 0 5
e

</CsScore>

</CsoundSynthesizer>
```

Fig.1: *linseg.csd*, o uso de linseg para gerar um envelope de amplitude

Aqui temos o mesmo arquivo *buzz.csd*, exceto que agora a amplitude é dada pelo opcode **linseg**, que traça segmentos de linha entre os parâmetros dados. A sintaxe de **linseg** é:

```
kres linseg ia, idur1, ib [, idur2] [, ic] [...]
```

onde os parâmetros de inicialização são dados por:

. ia é o valor do primeiro ponto
. idur1 é a duração do segmento de linha entre o primeiro e o segundo ponto
. ib é o valor do segundo ponto
. idur2 é a duração do segmento de linha entre entre o segundo e terceiro ponto.
. ic é o valor do terceiro ponto

e etc. Não há parâmetros de performance em **linseg**, apenas parâmetros de inicialização.

Então olhando para a seção orchestra de linseg.csd, vemos a inicialização de **kenv**:

```
kenv linseg 0, p3/5, 5000, p3/5, 2500, p3 * 2/5, 2500, p3/5, 0
```

Temos que p3 é a duração total da nota dada pelo arquivo score. Então o primeiro segmento de linha será um ataque do valor 0 ao valor 5000 com duração de um quinto da duração da nota, o segundo segmento será um decaimento do valor 5000 para o valor 2500, depois há uma sustentação no valor 2500, e finalmente um release para o valor 0 na última parte.

Usamos a seção **CsScore** de *linseg.csd* para tocar a nota por 5 segundos, para ser possível perceber nitidamente as variações de amplitude dadas por **linseg**. Executando o Csound para obter o arquivo de saída *linseg.wav*, nota-se como há um ataque inicial e logo depois um decaimento.

Se quisermos que o envelope tenha um único segmento de linha, existe o opcode bem mais simples **line**, cuja sintaxe é:

```
kres line ia, idur1, ib
```

que simplesmente traça um segmento de linha entre **ia** e **ib** com duração **idur1**.

Funções de envelope usando f-tables e o opcode *oscil*

Muitas vezes não queremos gerar o envelope na seção **CsInstruments**, como no exemplo anterior, mas queremos gerá-lo como uma forma de onda em uma f-table na seção **CsScore**. Podemos usar a função GEN07 para gerar a f-table de envelope. A sintaxe de GEN07 é:

```
f  #    time    size   7   a   n1   b   n2   c  ...
```

Aqui os parâmetros a, b, c, etc são valores que a onda deve alcançar através de um segmento de linha, e n1, n2, etc é a quantidade de pontos que este segmento de linha deve ter entre um valor e outro. Se declararmos por exemplo a f-table como:

```
f 1 0 16384 7 0 4096 5000 4096 2500 4096 2500 4096 0
```

ela será um segmento de linha de 0 até 5000 com 4096 pontos, em seguida outro segmento de linha de 5000 até 2500 com 4096 pontos, e etc. Todos os segmentos nesses exemplo tiveram 4096 pontos, podendo ser outros valores, mas a soma dos tamanhos deve ser igual ao tamanho total, aqui 16384 pontos. Se a soma tiver tamanho diferente do total, a onda será truncada ou preenchida com zeros.

Vamos então usar essa declaração de f-table na seção **CsScore** de nosso arquivo *gen07.csd*, como veremos na figura 2.

Queremos então gerar um envelope à partir da onda que está na na f-table 2, e através dela controlar a amplitude da f-table 1, que é uma senóide.

Temos que ter em mente que a f-table 2 não vai modular apenas um comprimento de onda da f-table 1, mas vai modular a onda *por quanto tempo a nota durar*. Então se a nota durar 10 segundos, temos que "esticar" a f-table 2 para ocupar 10 segundos e assim modular toda a onda.

Felizmente, isso pode ser feito com um simples cálculo na declaração de **oscil**, usando então:

```
kenv oscil kamp, 1/p3, ienv
```

teremos então em **kenv** a forma de onda da f-table 2 com exatamente **p3** segundos de duração. Se especificarmos por exemplo no arquivo score que a nota terá 10 segundos, p3 = 10, então para que o envelope tenha também 10 segundos precisamos gerar com o opcode oscil uma onda com período de 10 segundos. Se queremos que **kenv** tenha período 10, precisamos apenas especificar em **oscil** que sua frequência seja 1/10, ou seja, basta definir a frequência em **oscil** como 1/**p3** para gerar o envelope com exatamente a duração da nota.

O nosso arquivo *gen07.csd* então ficará:

```
<CsoundSynthesizer>

<CsOptions>
```

```
-o gen07.wav

</CsOptions>

<CsInstruments>

; Inicializa as variáveis globais.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrumento #1.
instr 1

  ; Frequência fundamental
  kcps = 440
  ; Número da f-table do seno.
  ifn = 1
  ; Número de harmônicos
  knh = 5
  ; Número da f-table de envelope
  ienv = 2

  ; Gera o envelope com amplitude 5000, duração de p3 segundos e a
  ; forma de onda armazenada na f-table 2.
  kenv oscil 5000, 1/p3, ienv

  ; Toca com amplitude variável de envelope kenv, a partir da
  ; frequência fundamental de 440 Hz, a onda dos 5 primeiros
  ; harmônicos do seno
  a1 buzz kenv, kcps, knh, ifn

  ; Manda o som armazenado em a1 para o arquivo de saída, gen07.wav
  out a1
endin

</CsInstruments>

<CsScore>

; Tabela #1: uma onda de seno.
f 1 0 16384 10 1

; Tabela #2: função de envelope usando GEN7.
f 2 0 16384 7 0 4096 5000 4096 2500 4096 2500 4096 0

; Toca o instrumento #1 por 5 segundos, começando em 0 segundos
i 1 0 5
e

</CsScore>

</CsoundSynthesizer>
```

Fig. 2: *gen07.csd*

Aqui usamos **kenv** para modular a saída do opcode **buzz**, mas poderíamos ter usado kenv de uma outra maneira produzindo o mesmo efeito:

```
a1 buzz 1, kcps, knh, ifn
out a1*kenv
```

Ou seja, pegamos a saída **a1** de **buzz**, sem ter a amplitude modulada, e a modulamos multiplicando **a1** por **kenv**, que tecnicamente é o que o Csound faz para modular a amplitude. Apesar de ambos os métodos terem efeitos idênticos, quando possível é melhor usar o método da Fig.2, por ser mais claro e legível.

Funções de envelope usando o opcode *envlpx*

Até aqui usamos opcodes que não são específicos para funções de envelope, como **linseg** e **oscil**. Entretanto existe uma classe de opcodes projetadas especificamente para produzir funções de envelope, e o mais importante deles é **envlpx**. A sintaxe de **envlpx** é:

```
kres envlpx kamp, irise, idur, idec, ifn, iatss, iatdec [, ixmod]
```

Onde os parâmetros de inicialização são:

. kamp é a amplitude de saída

. irise é o tempo de ataque da função

. idur é o tempo total do envelope

. idec é o tempo de decaimento

. ifn é o índice da forma de onda do ataque

. iatss é a abreviação para *attenuation of steady state***,** é o fator de atenuação para o leve decaimento que existe na fase de sustentação da nota. O valor 1 diz que a sustentação deve ser absoluta, enquanto para valores maiores que 1 há um aumento da amplitude do sinal e para valores entre 0 e 1 há um decaimento durante o tempo de sustentação da nota. Como a atenuação é exponencial, não se deve usar o valor zero.

. iatdec é o fator de atenuação na etapa de decaimento da nota, esse valor deve ser positivo e normalmente é da ordem de 0.01, para não haver um corte no final da nota.

. ixmod é opcional, o default é zero. Para valores positivos existe um retardamento no ataque ou decaimento definido em **iatss**, enquanto para valores negativos há uma aceleração, nesse período de sustentação da nota.

Vamos usar uma maneira mais simples de **envlpx**, utilizando como forma de onda de ataque uma reta, e usaremos a atenuação na sustentação constante. Esse será nosso arquivo *envlpx.csd*:

```
<CsoundSynthesizer>

<CsOptions>

-o envlpx.wav

</CsOptions>

<CsInstruments>

sr = 44100
kr = 4410
```

```
ksmps = 10
nchnls = 1

; Instrumento #1.
instr 1

  ; Frequência fundamental
  kcps = 440
  ; Número da f-table do seno.
  ifn = 1
  ; Número de harmônicos
  knh = 5

  ; Gera o envelope com amplitude 5000, duração de p3 segundos,
  ; sendo p3/5 segundos de ataque e p3*2/5 de decaimento, com a
  ; forma de onda de ataque armazenada na f-table 2.
  kamp = 5000
  irise = p3/5
  idur = p3
  idec = p3*2/5
  ienv = 2
  iatss = 1
  iatdec = 0.01

  ; Cria o envelope de amplitude
  kenv envlpx kamp, irise, idur, idec, ienv, iatss, iatdec


  ; Toca com amplitude variável de envelope kenv, a partir da
  ; frequência fundamental de 440 Hz, a onda dos 5 primeiros
  ; harmônicos
  a1 buzz kenv, kcps, knh, ifn

  ; Manda o som armazenado em a1 para o arquivo de saída, envlpx.wav
  out a1
endin

</CsInstruments>

<CsScore>

; Tabela #1, uma onda de seno.
f 1 0 16384 10 1
; Tabela #2, um envelope ascendente.
f 2 0 129 7 0 128 1

; Toca o instrumento #1 por 5 segundos, começando em 0 segundos
i 1 0 5
e

</CsScore>

</CsoundSynthesizer>
```

Fig.5: *envlpx.csd*

Na seção **CsScore** tocamos o instrumento por 5 segundos, e definimos a segunda f-table como a forma da onda de ataque de **envlpx**, que será uma reta ascendente usando a nossa já conhecida função GEN07.

Ao gerar o arquivo *envlpx.wav*, notamos uma variação de intensidade mais suave devido ao decaimento exponencial.

Funções de envelope e a Modulação de Anel

Um tipo especial de modulação de envelope é a modulação de anel ou *ring modulation*. Trata-se de uma das formas mais simples de síntese, e sua definição é simplesmente quando multiplicamos um sinal de entrada por um segundo sinal modulador, como fazemos no envelope. Entretanto quando a função moduladora não tem uma frequência com período igual à duração da nota, como uma função normal de envelope, mas possui uma frequência própria, ela passa a ser uma moduladora de anel.

Enquanto ao somamos dois sinais mantemos seus dois espectros de frequência, isso não ocorre na multiplicação. Quando multiplicamos dois sinais com frequências diferentes surgem novos harmônicos que não estavam presentes antes em nenhum dos dois sinais, e outros harmônicos desaparecem.

A seguir veremos um tipo de modulação que é uma aproximação da modulação de anel, já que o sinal modulador será de baixa frequência e atualizado apenas pela taxa de controle k-rate. Nele temos uma onda inicial cuja forma é um pulso com um oitavo de duração e sete oitavos de silêncio, armazenados na f-table 1 . Sobrepomos então oito versões dessa onda na saída, cada uma com uma defasagem de 45 graus em relação a anterior, e cada uma com variações aleatórias na amplitude, que alterarão bastante o espectro de frequências do sinal original.

Nesse instrumento usamos o opcode gerador de números aleatórios **randi**, e a sintaxe de **randi** que usaremos é apenas:

```
kres randi kamp, kcps
```

Em **kamp** temos a amplitude máxima que os números aleatórios assumirão, positiva e negativa, e **kcps** é a frequência na qual esses números são gerados. O sufixo **i** em **randi** significa que a cada novo ponto gerado ele é interpolado com o anterior, gerando uma curva contínua entre números subsequentes.

Veremos também agora um outro artifício muito usado que é o *de-click*, uma onda de envelope que retira possíveis cliques da onda de saída atenuando o início e o fim dessa onda. Isso será feito através das linhas:

```
kdclick  linseg  0, .02, 1, p3-.04, 1, .02, 0
out      aout*iamp*kdclick
```

A seguir temos o programa, que é uma versão do instrumento originalmente criado por Steven Cook, com variações aleatórias de amplitude e suas diferenças de fase. Essas variações de baixa frequência será nosso sinal modulador, e alterará drasticamente o espectro de frequências do sinal, dando um belo efeito etéreo e não-harmônico.

```
<CsoundSynthesizer>

<CsOptions>

-o randi.wav

</CsOptions>

<CsInstruments>

sr     = 44100
kr     = 4410
ksmps  = 10
nchnls = 1
```

```
instr    1

iamp   = 30000
icps   = cpspch(p4)
ifn    = 1

kdclick  linseg  0, .02, 1, p3-.04, 1, .02, 0

k1       randi  1, 01
k2       randi  1, 02
k3       randi  1, 03
k4       randi  1, 04
k5       randi  1, 05
k6       randi  1, 06
k7       randi  1, 07
k8       randi  1, 08

a1       oscil  k1, icps, ifn
a2       oscil  k2, icps, ifn, .125
a3       oscil  k3, icps, ifn, .25
a4       oscil  k4, icps, ifn, .375
a5       oscil  k5, icps, ifn, .5
a6       oscil  k6, icps, ifn, .625
a7       oscil  k7, icps, ifn, .75
a8       oscil  k8, icps, ifn, .875

aout     sum  a1, a2, a3, a4, a5, a6, a7, a8

out      aout*iamp*kdclick

endin


</CsInstruments>

<CsScore>

f1 0 4096 7 1 256 0 3584 0 256 1 ; Pulso com um oitavo da duração.

i1    0  4  05.00
e

</CsScore>

</CsoundSynthesizer>
```

Fig.6: *randi.csd*

Capítulo 4
Modelagem Física

Neste capítulo daremos veremos como o Csound emula instrumentos acústicos através da síntese por modelagem física. É importante notar que o Csound não foi desenvolvido com o propósito de emular sons reais, mas sim para criar novos sons sintetizados, portanto os opcodes a seguir são uma exceção à regra, e tem características próprias, diversas de um instrumento replicado por samples. Veremos primeiro o opcode mais usado dentre eles, **pluck,** e em seguida passaremos por **repluck, wgpluck, wgpluck2, wgbow, wgbowedbar, wgbrass, wgclar** e **wgflute.**

Cordas vibrantes

Primeiro vejamos como é a sintaxe dos opcodes de cordas vibrantes e o que eles tëm em comum. A sintaxe de **pluck** que usaremos é:

ares **pluck** kamp, kcps, icps, ifn, imeth

Em **imeth** temos o método de decaimento usado na corda, e usaremos apenas o método 1 que é de média simples entre os harmônicos iniciais e finais.

Ifn nos diz qual tabela deverá ser usada como onda inicial antes do decaimento, se **ifn =** 0 uma onda inicial de ruído aleatório é utilizada. Para simular uma corda real é melhor usar **ifn** = 0, pois o ruído inicial é rico em harmônicos o que acentua o processo de decaimento para uma onda apenas com harmônicos inteiros, como acontece quando uma corda real é tocada.

Kcps determina a frequência fundamental, a partir da qual é determinada quais parcias são harmônicas e devem ser mantidas, e quais parciais são inarmônicas e devem ser atenuadas. A única exigência é que **kcps** atinja no mínimo o valor de **icps**, pois para valores menores que **icps,** o período da onda não caberá no buffer.

Icps não afeta a frequência, apenas é usada para determinar de antemão o tamanho do buffer para armazenar um ciclo de onda da frequência fundamental, e atenuar subsequentemente as parciais inarmônicas dentro desse buffer. Se quisermos que o buffer tenha, ao invés de apenas um ciclo de onda, vários ciclos de onda da fundamental, basta usarmos uma frequência em **icps** abaixo da fundamental real determinada por **kcps.**

Um trecho de código que usa **pluck** vem a seguir:

```
instr 1
  kamp = 20000
  icps = cpspch(p4)
  kcps linseg icps, p3, icps*2
  ifn = 0
  imeth = 1

  a1 pluck kamp, kcps, icps, ifn, imeth

  out a1
endin
```

Podemos comparar **pluck** com seu parente mais complexo, **repluck**, criado por John ffitch, cuja sintaxe é:

ares **repluck** iplk, kamp, icps, kpick, krefl, axcite

Se em **pluck** podemos variar a frequência, em **repluck** não haverá essa possibilidade, mas em compensação podemos definir em que ponto tocaremos a corda e em que local estará colocado o captador. Isso é feito respectivamente através de **iplk** para o ponto em que a corda será tocada e **kpick** para o local do captador**,** que podem conter valores entre 0 e 1.

Em **krefl** definimos o tempo que a corda se manterá vibrando (i.e., a velocidade do decaimento), e deve conter um valor estritamente entre 0 e 1, mas diferente de 0 e de 1. Quanto maior **krefl,** maior a perda de amplitude e a velocidade do decaimento.

Finalmente em **axcite** temos a onda que excitará a corda, e preferencialmente deve ser uma onda de baixíssima frequência, algo entre 1Hz e 10Hz, para a corda ser excitada poucas vezes ao longo da duração da nota.

Um trecho de código típico de **repluck** está abaixo, onde usamos **axcite** oscilando apenas uma vez ao longo de toda a duração da nota:

```
instr 2
  iplk = 0.75
  kamp = 30000
  icps = cpspch(p4)
  kpick = 0.75
  krefl = 0.5
  axcite oscil 1, 1/p3, 1

  a1 repluck iplk, kamp, icps, kpick, krefl, axcite

  out a1
endin
```

Vamos ver agora o restante da família de opcodes que utiliza modelagem física por wave guide, desenvolvidos primeiro por Perry Cook e depois codificados para o Csound por John ffitch. O primeiro grupo para cordas vibrantes são **wgpluck** e **wgpluck2.** Abaixo temos a sintaxe de **wgpluck2,** que é uma versão simplificada de **repluck**:

ares **wgpluck2** iplk, kamp, icps, kpick, krefl

Temos em **wgpluck2** os mesmos parâmetros de **repluck,** utilizando a mesma faixa de valor, mas desta vez sem a onda de excitação **axcite.** Abaixo temos um trecho de código de **wgpluck2:**

```
instr 3
  iplk = 0.25
  kamp = 30000
  icps = cpspch(p4)
  kpick = 0.25
  krefl = 0.7

  a1 wgpluck2 iplk, kamp, icps, kpick, krefl

  out a1
endin
```

Chegamos agora a **wgpluck,** que é uma versão mais complexa de **repluck** e **wgpluck2.** A sintaxe de **wgpluck** é:

ares **wgpluck** icps, iamp, kpick, iplk, idamp, ifilt, axcite

Esse opcode é semelhante a **repluck** e **wgpluck2**, pois além dos parâmetros de utilização idêntica, temos **idamp** que é uma nova versão de **krefl**, que para valores maiores o decaimento é mais rápido, mas com a novidade que para valores negativos de **idamp** há um acréscimo do sinal com o tempo. Vale lembrar que ao contrário de **krefl, idamp** não é normalizado, podendo assumir qualquer valor positivo ou negativo.

Em **ifilt** temos um filtro na ponte (ponto onde as cordas são presas ao corpo do instrumento) em que definimos a partir de quais harmônicos a onda será atenuada, valores altos para uma atenuação primeiro de frequências altas e valores baixos para que haja no início uma atenuação das baixas frequências.

Um trecho de código utilizando as particularidades de **wgpluck** vem abaixo:

```
instr 4
  icps = cpspch(p4)
  iamp = 10000
  kpick = 0.5
  iplk = 0
  idamp = -10
  ifilt = 1000
  axcite oscil 1, 1/p3, 1

  a1 wgpluck icps, iamp, kpick, iplk, idamp, ifilt, axcite

  out a1
endin
```

A seguir temos o código completo com os quatro instrumentos e o score dando as frequências a serem tocadas.

```
<CsoundSynthesizer>

<CsOptions>

-o string.wav

</CsOptions>

<CsInstruments>

instr 1
  kamp = 20000
  icps = cpspch(p4)
  kcps linseg icps, p3, icps*2
  ifn = 0
  imeth = 1

  a1 pluck kamp, kcps, icps, ifn, imeth

  out a1
endin

instr 2
  iplk = 0.75
```

```
  kamp = 30000
  icps = cpspch(p4)
  kpick = 0.75
  krefl = 0.5
  axcite oscil 1, 1/p3, 1

  a1 repluck iplk, kamp, icps, kpick, krefl, axcite

  out a1
endin

instr 3
  iplk = 0.25
  kamp = 30000
  icps = cpspch(p4)
  kpick = 0.25
  krefl = 0.7

  a1 wgpluck2 iplk, kamp, icps, kpick, krefl

  out a1
endin

instr 4
  icps = cpspch(p4)
  iamp = 10000
  kpick = 0.5
  iplk = 0
  idamp = -10
  ifilt = 1000
  axcite oscil 1, 1/p3, 1

  a1 wgpluck icps, iamp, kpick, iplk, idamp, ifilt, axcite

  out a1
endin

</CsInstruments>

<CsScore>

f1 0 16384 10 1

i1 0 5 7.09
i2 5 5 7.09
i3 10 5 7.09
i4 15 5 7.09

e

</CsScore>

</CsoundSynthesizer>
```

Fig.1: *string.csd*

Instrumentos de arco

No Csound temos a modelagem de instrumentos de arco através do opcode **wgbow.** A sintaxe de **wgbow** a ser usada é:

```
ares wgbow kamp, kfreq, kpres, krat, kvibf, kvamp, ifn
```

Em **kamp** e **kfreq** temos a amplitude e a frequência de nota. Em **kpres** a pressão do arco sobre a corda, e deve variar entre 1 e 5, e em **krat** o ponto em que o arco está sobre a corda, que deve variar entre 0.025 e 0.23.

Embutido no opcode temos um vibrato, controlado pelos parâmetros **kvibf, kvamp** e **ifn.** Em **kvibf** temos a frequência do vibrato, que deve ser um valor baixo, usualmente entre 1 e 12Hz, em **kvamp** temos a amplitude do vibrato, ou seja, em que porcentagem o vibrato alterará a frequência original, e deve ser um valor baixíssimo, em torno de um centésimo. Finalmente em **ifn** temos a forma de onda do vibrato, usualmente uma f-table com uma senóide. A seguir o código usando **wgbow:**

```
<CsoundSynthesizer>

<CsOptions>

-o wgbow.wav

</CsOptions>

<CsInstruments>
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

instr 1
  kamp = 30000
  kfreq = cpspch(p4)
  kpres = 3
  krat = 0.13
  kvibf = 4
  kvamp = 0.01
  ifn = 1

  a1 wgbow kamp, kfreq, kpres, krat, kvibf, kvamp, ifn
  out a1
endin

</CsInstruments>

<CsScore>

f 1 0 128 10 1

i 1 0 5 8.09
e

</CsScore>

</CsoundSynthesizer>
```

Fig.2: *wgbow.csd*

A barra percutida

O opcode seguinte, para uma barra percutida, é **wgbowedbar**, e a sintaxe que usaremos é:

```
ares wgbowedbar kamp, kfreq, kpos, kbowpres, kgain \
               [, iconst] [, itvel] [, ibowpos]
```

Como em **wgbow, kpos** determina a posição do arco, mas agora pode variar entre 0 e 1, e **kbowpress** determina a pressão do arco, com valores entre 0.6 e 0.7. Em **kgain** há o ganho de saída do opcode, que deve variar em torno de 0.9 e 1.

De fato, **wgbowedbar** é um opcode muito sensível, e é preciso fazer um ajuste fino entre **kbowpres** e **kgain** para se obter uma saída satisfatória. Poucos décimos de mudança nesses parâmetros é a diferença entre uma microfonia completamente descontrolada e o silêncio absoluto. Abaixo temos o código usando **wgbowedbar,** com os parâmetros devidamente ajustados**:**

```
<CsoundSynthesizer>

<CsOptions>

-o wgbowedbar.wav

</CsOptions>

<CsInstruments>
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

instr 1
  kamp = 30000
  kfreq = cpspch(p4)
  kpos = 0.5
  kbowpres = 0.7
  kgain = 0.995

  a1 wgbowedbar kamp, kfreq, kpos, kbowpres, kgain
  out a1
endin

</CsInstruments>

<CsScore>

f 1 0 128 10 1

i 1 0 5 8.09
e

</CsScore>

</CsoundSynthesizer>
```

Fig.3: *wgbowedbar.csd*

Instrumentos de sopro

Vejamos agora os opcodes de modelagem física para os instrumentos de sopro, que tem semelhanças de parâmetro com **wgbow.** Mais especificamente veremos **wgbrass, wgclar** e **wgflute.** A sintaxe que usaremos de **wgbrass** é:

```
ares wgbrass kamp, kfreq, ktens, iatt, kvibf, kvamp, ifn
```

Kamp e **kfreq** são a amplitude e a frequência, **ktens** é semelhante a **kbowpres**, nos diz a pressão que o instrumentista faria ao soprar, e deve conter valores em torno de 0.4. Em seguida vem **iatt,** o único parâmetro realmente novo, que nos diz o

tempo de ataque da nota até atingir pressão total. **Kvibf, kvamp** e **ifn** fazem parte do vibrato embutido, como em **wgbow**.

Todos os opcodes da seção de sopros estão em nível experimental, e tem-se que ter uma atenção especial com o vibrato. É recomendável que se use um valor baixíssimo para frequência e amplitude do vibrato, ou o som sofrerá cortes. Um trecho de código para **wgbrass** vem abaixo:

```
instr 1
  kamp = 30000
  kfreq = cpspch(p4)
  ktens = 0.4
  iatt = 0.1
  kvibf = 1/(2*p3)
  kvamp = 0.2
  ifn = 1

  a1 wgbrass kamp, kfreq, ktens, iatt, kvibf, kvamp, ifn
  out a1
endin
```

A seguir temos **wgclar,** a variação para clarineta de **wgbrass**, e a sintaxe que usaremos é:

```
ares wgclar kamp, kfreq, kstiff, iatt, idetk, kngain, kvibf, kvamp, \
           ifn
```

A maioria dos parâmetros aqui são herdados de **wgbrass.** Os parâmetros que já tínhamos visto antes são **kamp** e **kfreq** para amplitude e frequência, **iatt** para tempo de ataque, e **kvibf, kvamp** e **ifn** para a frequência, amplitude e forma de onda do vibrato.

Vejamos a seguir os parâmetros novos. **Kstiff** é a força que o ar sai da clarineta, e deve conter valores negativos entre -0.1 e -0.5, sendo maior e mais ruidoso conforme valores mais negativos são definidos. **Idetk** é o intervalo de tempo entre parar de soprar o instrumento e parar de ser emitido som, através de um rápido decaimento, e valores em torno de 0.1 segundos são geralmente usados. **Kngain** é o ganho que deve sofrer o ruído que acompanha o som da clarineta, normalmente com valores de no máximo 0.5.

A seguir temos um trecho de código com **wgclar**:

```
instr 2
  kamp = 30000
  kfreq = cpspch(p4)
  kstiff = -0.4
  iatt = 0.1
  idetk = 0.1
  kngain = 0.4
  kvibf = 1/(2*p3)
  kvamp = 0.2
  ifn = 1

  a1 wgclar kamp, kfreq, kstiff, iatt, idetk, kngain, kvibf, kvamp,\
            ifn
  out a1
endin
```

E como último instrumento da seção de sopros, temos **wgflute**, que emula uma flauta, e sua sintaxe reduzida é:

```
ares wgflute kamp, kfreq, kjet, iatt, idetk, kngain, kvibf, kvamp, ifn
```

Aqui o único novo parâmetro é **kjet,** que é similar a **ktens, kbowpres** e **kstiff**, e controla a força do jato de ar, e deve conter valores entre 0.1 e 0.5. Os parâmetros restantes são como vistos acima em **wgclar.**

Um trecho de código para **wgflute** é:

```
instr 3
  kamp = 30000
  kfreq = cpspch(p4)
  kjet = 0.3
  iatt = 0.1
  idetk = 0.1
  kngain = 0.05
  kvibf = 1/(2*p3)
  kvamp = 0.05
  ifn = 1

  a1 wgflute kamp, kfreq, kjet, iatt, idetk, kngain, kvibf, kvamp, ifn
  out a1
endin
```

A seguir temos o código completo dos três instumentos de sopro, tocados pelo score:

```
<CsoundSynthesizer>

<CsOptions>

-o wind.wav

</CsOptions>

<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

instr 1
  kamp = 30000
  kfreq = cpspch(p4)
  ktens = 0.4
  iatt = 0.1
  kvibf = 1/(2*p3)
  kvamp = 0.2
  ifn = 1

  a1 wgbrass kamp, kfreq, ktens, iatt, kvibf, kvamp, ifn
  out a1
endin

instr 2
  kamp = 30000
  kfreq = cpspch(p4)
  kstiff = -0.4
  iatt = 0.1
  idetk = 0.1
  kngain = 0.4
  kvibf = 1/(2*p3)
  kvamp = 0.2
  ifn = 1
```

```
  a1 wgclar kamp, kfreq, kstiff, iatt, idetk, kngain, kvibf, kvamp, ifn

  out a1
endin

instr 3
  kamp = 30000
  kfreq = cpspch(p4)
  kjet = 0.3
  iatt = 0.1
  idetk = 0.1
  kngain = 0.05
  kvibf = 1/(2*p3)
  kvamp = 0.05
  ifn = 1

  a1 wgflute kamp, kfreq, kjet, iatt, idetk, kngain, kvibf, kvamp, ifn
  out a1
endin

</CsInstruments>

<CsScore>

f1 0 16384 10 1

i1 0 5 8.09
i2 5 5 8.09
i3 10 5 8.09

</CsScore>

</CsoundSynthesizer>
```

Fig. 4: *wind.csd*

Capítulo 5
Efeitos

No capítulo 3 vimos a modulação de amplitude através das funções de envelope, e aqui veremos a alteração de frequências através dos efeitos de chorus, vibrato, reverb, distrorção, flanger e um oscilador FM. Veremos que a medida que usamos mais efeitos, o som ficará mais complexo e interessante. Nos exemplos a seguir você criará sons que sempre ouviu a partir de pacotes comerciais, com uma versatilidade muito superior dada à abrangência e parametrização de Csound.

O efeito *chorus*

Um efeito muito usado é o chorus, que simula a execução de vários instrumentos em uníssono. Existe algumas técnicas para transformar o som de um único instrumento, em algo tocado por vários instrumentos juntos.

Um dos métodos para alcançar esse efeito é a combinação de várias ondas com suas frequências ligeiramente diferentes para gerar o efeito de chorus. O segundo método é alterar levemente o atraso dos instrumentos entre si.

Para termos esse atraso entre os instrumentos vamos usar o par de opcodes **delayr** e **delayw.** O funcionamente desses opcodes é simples, primeiro criamos uma linha de atraso através da declaração:

```
ares delayr idlt
```

Com **idlt** especificando o tempo de atraso dessa linha, e qualquer sinal que seja escrito nela sofrerá esse atraso antes de chegar a **ares**. Depois de criada essa linha, podemos escrever nela através de **delayw**, como em:

```
delayw asig
```

E **asig** será escrito em **ares** com o atraso **idlt.** Sempre que houver uma declaração **delayw**, ela se referirá ao **delayr** anterior mais próximo.

Podemos ver na fig. 1 a definição de dois instrumentos: o primeiro é a combinação do sinal de três opcodes **buzz** com frequências alteradas e depois atrasados para gerar o efeito chorus, e depois somamos as três ondas para gerar a onda de saída. O segundo é análogo, mas usamos o opcode **vco,** criado por Hans Mikelson, que é um oscilador capaz de gerar ondas de vários tipos. A sintaxe de **vco** é:

```
ares vco xamp, xcps, iwave, kpw [, ifn]
    [, imaxd] [, ileak] [, inyx] [, iphs] [, iskip]
```

Xamp e **xcps** são a amplitude e frequência, em seguida temos **iwave**, que define a forma da onda de acordo com os índices: 1 gera uma onda em forma de rampa, 2 gera uma onda quadrada, 3 uma onda triangular. **Kpw** depende de **iwave,** e define a declividade se **iwave** =3, ou a largura do pulso se **iwave** = 2. Usaremos **kpw** = 1 para obter uma onda triangular perfeita quando **iwave** é 3.

Os parâmetros **ifn, imaxd, ileak, inyx, iphs, iskip** são opcionais e não serão usados.

Em nosso segundo instrumento usaremos três opcodes **vco** com a frequência levemente alterada e iremos mixá-los, como fizemos com **buzz** no primeiro instrumento, para alcançar o efeito de chorus.

```
<CsoundSynthesizer>

<CsOptions>

-o chorus.wav

</CsOptions>

<CsInstruments>

; Inicializa as variáveis globais.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrumento #1, três buzz para gerar chorus
instr 1

  ; Gera o envelope com amplitude 5000, duração de p3 segundos,
  ; sendo p3/5 segundos de ataque, p3*2/5 de sustentação e p3*2/5 de
  ; decaimento, 0.1 é usado pois 0 é ilegal em exponenciais
  iamp = 5000
  irise = p3/5
  isus = p3*2/5
  idec = p3*2/5
  kenv expseg 0.1, irise, iamp, isus, iamp, idec, 0.1

  ; Cria o envelope de alteração de frequência
  kfreq linseg 0, p3, 0.01

  ; Toca com amplitude variável de envelope kenv, a partir da
  ; frequência fundamental de 440 Hz, a onda dos 5 primeiros
  ; harmônicos em 3 frequências ligeiramente diferentes
  ; Aqui há uma transformação de 440Hz +- 1% em 440 Hz novamente.

  ; Frequência fundamental
  kcps = 440
  ; Número de harmônicos
  knh = 5
  ; Número da f-table do seno.
  ifn = 1
  a1 buzz kenv, kcps, knh, ifn
  a2 buzz kenv, kcps*(0.99+kfreq), knh, ifn
  a3 buzz kenv, kcps*(1.01-kfreq), knh, ifn

  ; Dá um leve atraso entre as três ondas
  adel1 = a1
  adel2 delayr 0.1
  delayw a2
  adel3 delayr 0.05
  delayw a3

  ; soma as três ondas na resultante
  asig = adel1 + adel2 +adel3

  ; Manda o som armazenado em asig para o arquivo de saída,
  ; chorus.wav
  out asig
endin
```

```
; Instrumento #2, três vco para gerar chorus
instr 2

  ; Gera o envelope com amplitude 5000, duração de p3 segundos,
  ; sendo p3/5 segundos de ataque, p3*2/5 de sustentação e p3*2/5 de
  ; decaimento, 0.1 é usado pois 0 é ilegal em exponenciais
  iamp = 5000
  irise = p3/5
  isus = p3*2/5
  idec = p3*2/5
  kenv expseg 0.1, irise, iamp, isus, iamp, idec, 0.1

  ; Toca com amplitude variável de envelope kenv, a partir da
  ; frequência fundamental de 440 Hz, a onda em 3 frequências
  ; levemente diferentes, com 3 formas de onda possíveis.

  ; Frequência fundamental
  kcps = 440
  ; Forma de onda
  iwave = p4
  ; Alterador de declividade do vco
  kpw = 1
  a1 vco kenv, kcps, iwave, kpw
  a2 vco kenv, kcps*0.99, iwave, kpw
  a3 vco kenv, kcps*1.01, iwave, kpw

  ; Dá um leve atraso entre as três ondas
  adel1 = a1
  adel2 delayr 0.1
  delayw a2
  adel3 delayr 0.05
  delayw a3

  ; soma as três ondas na resultante
  asig = adel1 + adel2 +adel3

  ; Manda o som armazenado em asig para o arquivo de saída,
  ; chorus.wav
  out asig
endin

</CsInstruments>

<CsScore>

; Tabela #1, uma onda de seno.
f 1 0 16384 10 1

; Tabela #2, um envelope ascendente.
f 2 0 129 7 0 128 1

; Toca o instrumento #1 por 20 segundos, começando em 0 segundos
i 1 0 20

; Toca o instrumento #2 por 5 segundos, começando em 21 segundos
; com forma de onda 1, rampa.
i 2 21 5 1
; com forma de onda 2, quadrado.
i 2 27 5 2
; com forma de onda 3, triângulo.
i 2 33 5 3
e

</CsScore>

</CsSynhesizer>
```

Fig.1: *chorus.csd*

No instrumento 1 nós usamos um envelope **kfreq** para alterar dinamicamente a frequência dos opcodes **buzz**, começando com frequências bem acima e abaixo de 440 Hz e caminhando para o uníssono. Nesse exemplo vimos as três ondas combinadas, assim como as ondas produzidas pelo instrumento 2 com os opcodes **vco.** Em nossa seção CsScore houve mais tempo para a primeira nota, para observarmos todas as nuances na alteração das frequências por **kfreq**, e então tocamos mais três notas consecutivas, cada uma com uma forma de onda diferente, todas com o efeito chorus.

Nesse exemplo usamos o opcode **expseg**, criado por Gabriel Maldonado, que tem exatamente a mesma sintaxe de **linseg** que já vimos, apenas traçando dessa vez segmentos exponenciais.

O efeito *vibrato*

O efeito vibrato são leves e rápidas alterações em torno de uma frequência fundamental, como se tensionássemos rapidamente várias vezes uma corda, sem que a frequência percebida se altere drasticamente. É exatamente isso que faremos na orchestra a seguir, modularemos a frequência de **buzz** com um sinal oscilatório que alterará a frequência em torno das notas dadas pelo score.

Aqui usaremos o opcode **linen** como gerador de envelope. A sintaxe de linen é:

```
kres linen kamp, irise, idur, idec
```

Onde **kamp** é a amplitude, **irise** o tempo de ataque, **idur** a duração total do envelope**,** e **idec** o tempo de decaimento**.**

```
<CsoundSynthesizer>

<CsOptions>

-o vibrato.wav

</CsOptions>

<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

instr 1
  ; A frequência é dada pelo quarto parâmetro no score
  kcps = cpspch(p4)

  ; envelope de amplitude
  kamp = 5000
  idur = p3
  iatk = p3/5
  idec = p3*2/5
  kenv linen kamp, iatk, idur, idec

  ; a variação máxima da frequência será de um centésimo
  kdepth = kcps/100
  ; velocidade de oscilação da frequência
  kfreq = 4
  ; envelope do vibrato
```

```
  ifn = 1
  kvib oscil kdepth, kfreq, ifn

  knh = 5
  a1 buzz kenv, kcps+kvib, knh, ifn

  out a1
endin

</CsInstruments>

<CsScore>

f 1 0 16384 10 1

i  1  0   1  6.04
i  1  1   1  6.04
i  1  2   1  6.05
i  1  3   1  6.07
i  1  4   1  6.07
i  1  5   1  6.05
i  1  6   1  6.04
i  1  7   1  6.02
i  1  8   1  6.00
i  1  9   1  6.00
i  1  10  1  6.02
i  1  11  1  6.04
i  1  12  2  6.04
i  1  14  2  6.02
e

</CsScore>

</CsoundSynthesizer>
```

Fig.2: *vibrato.csd*

Osciladores FM

Uma técnica de síntese com efeito interessante é a onda modulada por frequência, ou FM (*frequency modulated)*, em que temos dois osciladores, onde a saída de um modula a frequência do outro. Felizmente em Csound não precisaremos usar dois osciladores, pois um único opcode faz todo o trabalho, **foscil.** A sintaxe de **foscil** é:

```
ares foscil xamp, kcps, xcar, xmod, kndx, ifn [, iphs]
```

Onde os parâmetros são:

. xamp é a amplitude.

. kcps é uma frequência de base, que será o denominador comum entre os parâmetros **xcar** e **xmod**.

. xcar é o valor pelo qual se multiplica **kcps** para se obter a frequência da onda portadora (*carrier*)

. xmod é o valor pelo qual se multiplica **kcps** para obter a frequência da onda moduladora. A frequência percebida por nós é dada pela frequência da portadora, quando esta é menor que a moduladora.

. kndx é o índice de modulação, que determina a distribuição de amplitude para cada harmônico gerado na modulação, e normalmente varia no tempo entre os valores 0 e 4.

. ifn é o índice da f-table que conterá uma senóide

. iphs é opcional, com valor default 0, e define a fase inicial da onda em **ifn**, podendo conter valores entre 0 e 1.

Veremos agora um exemplo em que usamos apenas a modulação por frequência para gerar a onda de saída.

```
<CsoundSynthesizer>

<CsOptions>

-o foscil.wav

</CsOptions>

<CsInstruments>

; Inicializa as variáveis globais
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrumento #1 - uma forma de onda FM variável
instr 1

  ; cria o envelope de amplitude
  iamp = 10000
  iatk = p3/3
  idur = p3
  idec = p3/3
  kenv  linen   iamp, iatk, idur, idec

  ; define o índice de modulação variável
  ibegin = p5
  iend = p6
  kndx  expon   ibegin, idur, iend

  ; gera a onda a partir do oscilador FM
  icps = cpspch(p4)
  icar = 1
  imod = 1
  ifn = 1
  a1   foscil   kenv, icps, icar, imod, kndx, ifn
  out a1

endin

</CsInstruments>

<CsScore>

f 1  0 4096 10   1

i 1     0        4      7.09             1      30
i 1     5        4      6.09             1      60
i 1     10       9      5.09             60     1
e

</CsScore>

</CsoundSynthesizer>
```

Fig. 3: *foscil.csd*, um oscilador FM de índice de modulação variável

Aqui a onda portadora e a moduladora tem a mesma frequência, dada pelo quarto parâmetro do score, mas o índice de modulação varia entre valores dados pelos dois últimos parâmetros.

Flanger

Originalmente o efeito de flanger era obtido através de vários delays variáveis colocados em cascata. Csound tem um opcode específico para esse efeito, **flanger,** criado por Gabriel Maldonado, e sua sintaxe é:

```
ares flanger asig, adel, kfeedback [, imaxd]
```

Os parâmetros de **flanger** são **asig**, que é o sinal de entrada em que será aplicado o flanger, **adel**, que é o delay variável que controla o flanger e não pode exceder **imaxd**, e **kfeedback**, que é a quantidade de feedback desejada, e deve estar entre 0 e 1.

O único parâmetro opcional é **imaxd,** e diz o delay máximo em segundos, e esse parâmetro é usado para que o opcode saiba a quantidade de memória necessária na inicialização.

Como fonte de som inicial usaremos o opcode **pluck**, visto no Capítulo 4, que simula uma corda vibrante com decaimento natural.

Abaixo temos três efeitos de flanger diferentes obtidos através da variação dos parâmetros de entrada do opcode **flanger**, dando uma perspectiva do que pode ser alcançado com esse opcode.

As alterações dos parâmetros fôram reproduzidas e destacadas dentro de cada instrumento, ao contrário de serem passadas pelo score, para que a compreensão seja mais direta.

```
<CsoundSynthesizer>

<CsOptions>

-o flanger.wav

</CsOptions>

<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

instr 1
  iamp = 10000
  ifqc = cpspch(p4)
  ifn = 0
  imeth = 1
  asig pluck iamp, ifqc, ifqc, ifn, imeth

  adel line 0, p3, 0.01
  kfeedback = 0.7

  aflang flanger asig, adel, kfeedback

  a1 clip aflang, 1, 30000
  out a1
endin
```

```
instr 2
  iamp = 10000
  ifqc = cpspch(p4)
  ifn = 0
  imeth = 1
  asig pluck iamp, ifqc, ifqc, ifn, imeth

  adel line 0, p3, 0.01
  kfeedback = 0.85

  aflang flanger asig, adel, kfeedback

  a1 clip aflang, 1, 30000
  out a1
endin

instr 3
  iamp = 10000
  ifqc = cpspch(p4)
  ifn = 0
  imeth = 1
  asig pluck iamp, ifqc, ifqc, ifn, imeth

  adel line 0, p3, 0.05
  kfeedback = 0.7

  aflang flanger asig, adel, kfeedback

  a1 clip aflang, 1, 30000
  out a1
endin

</CsInstruments>

<CsScore>

i 1 0 1 8.05
i 1 1 1 8.02
i 1 2 1 8.05
i 1 3 1 8.04
i 1 4 1 8.05
i 1 5 1 8.07
i 1 6 1 8.05
i 1 7 1 8.07
i 1 8 1 8.04
i 1 9 1 8.09
i 1 10 1 8.04
i 1 11 1 8.02
s

i 2 0 1 8.05
i 2 1 1 8.02
i 2 2 1 8.05
i 2 3 1 8.04
i 2 4 1 8.05
i 2 5 1 8.07
i 2 6 1 8.05
i 2 7 1 8.07
i 2 8 1 8.04
i 2 9 1 8.09
i 2 10 1 8.04
i 2 11 1 8.02
s

i 3 0 1 8.05
i 3 1 1 8.02
i 3 2 1 8.05
```

```
i 3 3 1 8.04
i 3 4 1 8.05
i 3 5 1 8.07
i 3 6 1 8.05
i 3 7 1 8.07
i 3 8 1 8.04
i 3 9 1 8.09
i 3 10 1 8.04
i 3 11 1 8.02
e

</CsScore>

</CsoundSynthesizer>
```

Fig.4: *flanger.csd*

Reverberação

Em Csound existe uma série de opcodes que dão eco ou delay a uma onda de entrada. Quando esse atraso é perceptível, podemos distinguir o efeito de delay, e quando esse atraso é pequeno e se torna imperceptível ao ouvido humano, alcançamos o efeito de reverb.

Vamos começar pelo opcode **reverb,** criado por William "Pete" Moss, por se tratar do mais simples e direto opcode de reverberação. Como veremos depois, existem vários parâmetros envolvidos no processo do efeito de reverb, entretanto esses parâmetros são pré-configurados em **reverb** para dar uma resposta de reverberações comuns à uma sala média. Isso se nota pela pouca quantidade de parâmetros de **reverb:**

```
ares reverb asig, krvt [, iskip]
```

Asig é o sinal de entrada e **krvt** é o tempo pelo qual cada *sample* será reverberado.

O parâmetro **iskip** é opcional e diz se será aproveitada a última reverberação dada por outro instrumento no início desta nota. O default é 0, isto é, o som começa apenas com o sinal original, sem aproveitar reverberações anteriores. Tecnicamente, isso diz se o *buffer* usado para a reverberação é esvaziado no início da execução do opcode ou é mantido com o sinal que estava anteriormente.

Precisamos antes fazer algumas considerações em como implementar o reverb em uma orchestra. A solução imediata e intuitiva usando **reverb** seria implementar toda a geração do sinal e do reverb dentro do mesmo instrumento, como abaixo:

```
instr 1
  krvt = 1.5                  ; tempo enquanto vai haver reverberação
  a1 oscil 30000, 440, 1      ; sinal de entrada
  arev reverb a1, krvt        ; sinal reverberado
  out arev
endin
```

Entretanto essa implementação tem um problema grave, pois assim que a nota que ativou o instrumento termina, no mesmo instante a reverberação cessa. Se nossa nota fôr durar, por exemplo, um décimo de segundo, não poderemos perceber o eco pois assim que esse décimo passar não haverá mais reverberação, isto é, precisamos que o reverb esteja ativado mesmo quando o instrumento não está mais sendo executado.

Para alcançar esse objetivo, vamos implementar um instrumento separado apenas para o reverb, que fique ativo durante toda a duração do score. Nessa nova

implementação usaríamos uma variável global **gamix** para passar o sinal dos instrumentos para o reverb, e ficaria algo como:

```
gamix init 0

instr 1
  a1 oscil 30000, 440, 1
  out a1
  gamix = gamix + a1
endin

instr 99
  krvt = 1
  arev reverb gamix, krvt
  out arev
  gamix = 0
endin
```

Note que no primeiro instrumento **gamix** é somada ao sinal do instrumento. Apesar de no exemplo haver apenas um instrumento antes do reverb por simplicidade, em outros códigos isso garantiria que o sinal de todos os instrumentos antes do reverb estariam em **gamix**, para esse sinal ser então reverberado no último instrumento.

Aqui quando o primeiro instrumento pára, o sinal de entrada **gamix** cessa mas continua sendo reverberado pelo segundo instrumento. No instante em que **instr 1** é desativado, o sinal de entrada **gamix** também deve se igualar a zero, caso contrário **gamix** geraria um sinal de entrada eterno mesmo quando **instr 1** foi desligado, e você obteria um sinal com amplitude tendendo a infinito, obtendo na mensagem *samples out of range* um número muito grande de amostras que ultrapassaram a amplitude limite. Temos apenas de lembrar de numerar os instrumentos de reverb com a maior numeração da orchestra, para garantir que eles sempre sejam executados por último.

Em síntese, essa abordagem de instrumentos distintos, reinicializando sempre **gamix** para zero, garante que haverá a reverberação necessária e não além. Esse modelo de programação será usado em todos os opcodes de eco que usaremos daqui em diante.

Para tornar nosso leque mais variado, podemos aplicar o reverb sobre um sample de áudio. Para isso usamos o opcode **loscil,** que aplica uma frequência escolhida sobre um sample, depois que lhe informamos em que frequência base o sample está. A sintaxe de **loscil** que usaremos é:

```
ares loscil xamp, kcps, ifn [, ibas]
```

onde **xamp** é a amplitude, **kcps** a frequência desejada, **ifn** a f-table que contém o arquivo de áudio, e **ibas** a frequência fundamental da onda contida no arquivo, que será modificada para chegar a **kcps**. Abaixo vem um trecho de código, aplicando o reverb sobre uma onda com **loscil**:

```
instr 2
  kamp = 10000
  kcps = 1
  ifn = 2
  ibas = 1
  a1 loscil kamp, kcps, ifn, ibas
  out a1
  gamix = gamix + a1
endin

instr 99
```

```
  krvt = p4
  a99 reverb gamix, krvt
  out a99
  gamix = 0
endin
```

E finalmente temos o programa completo, usando um mesmo reverb, que fica ativo durante todo o score, para os dois instrumentos.

```
<CsoundSynthesizer>

<CsOptions>

-o reverb.wav

</CsOptions>

<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

gamix init 0

instr 1
  kamp = 10000
  kcps = cpspch(p4)
  ifn = 1
  a1 oscil kamp, kcps, ifn
  out a1
  gamix = gamix + a1
endin

instr 2
  kamp = 10000
  kcps = 1
  ifn = 2
  ibas = 1
  a1 loscil kamp, kcps, ifn, ibas
  out a1
  gamix = gamix + a1
endin

instr 99
  krvt = p4
  a99 reverb gamix, krvt
  out a99
  gamix = 0
endin

</CsInstruments>

<CsScore>

f 1 0 16384 10 1

f 2 0 524288 1 "female.aif" 0 4 0

i 1 0 .1 8.05
i 1 1 .1 8.02
i 1 2 .1 8.05
i 1 3 .1 8.04
i 1 4 .1 8.05
```

```
i 1 5 .1 8.07
i 1 6 .1 8.05
i 1 7 .1 8.07
i 1 8 .1 8.04
i 1 9 .1 8.09
i 99 0  10  1

i 2  10 6
i 99 10 10  1

i 2  20 6
i 99 20 10  3

i 2  30 6
i 99 30 10  10

e

</CsScore>

</CsoundSynthesizer>
```

Fig.4: *reverb.csd*

Reverberação usando *comb* e *vcomb*

Tradicionalmente a codificação de reverbs em Csound era feita através de opcodes **combs** em paralelo e filtros *all-pass*, que adicionavam um atraso ao sinal.Vamos ver o opcode **comb**, que dá vários atrasos a uma onda inicial diminuindo sua amplitude. Ele proporciona o efeito de um *delay* colocado em *loopback***,** causando a realimentação do sistema e proporcionando o efeito de *reverb* . A sintaxe de **comb** que usaremos é:

```
ares comb asig, krvt, ilpt
```

O novo parâmetro de inicialização que não havia em **reverb** é **ilpt**, que é o tempo de loopback ou atraso entre um ciclo e outro dado pelo reverb, i.e., o tempo que o sinal leva para começar a ser reverberado. Os parâmetros de performance são:

. asig é o sinal de entrada.

. krvt ou *reverb time* é o tempo em que o sinal se mantém ecoando enquanto gradativament desaparece, e diz indiretamente quantas vezes o som sofrerá loopback até desaparecer completamente.

O segundo opcode da família de **comb**, que usaremos em nosso segundo instrumento, é o **vcomb**, criado por William “Pete” Moss, que tem como característica um tempo de atraso no *loopback* variável, e cuja sintaxe é:

```
ares vcomb asig, krvt, xlpt, imaxlpt [, iskip] [, insmps]
```

Os parâmetros novos aqui são **xlpt** e **imaxlpt.** Este último, **imaxlpt**, é de inicialização e nos diz qual o máximo atraso que o sinal pode assumir durante a execução, i.e., o tempo máximo de *loopback*.

Este parâmetro é necessário exatamente porque o parâmetro de performance anterior **xlpt** é variável, dizendo a cada instante qual o tempo de atraso que o sinal deve ter, e o fato dele ser variável é o que diferencia **vcomb** de **comb**.

Veremos agora o código equivalente de **reverb** usando **comb** e **vcomb**, especificando vários valores para **ilpt** diferentes para **comb,** que será implementado no **instr 98.**

No caso de **vcomb,** que será implementado no **instr 99**, usaremos três parâmetros adicionais para o *reverb* de loopback variável*:* **p5** será o valor inicial de loopback, **p6** será o valor final, e **p7** será o loopback máximo.

```
<CsoundSynthesizer>

<CsOptions>

-o combs.wav

</CsOptions>

<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

gamix init 0

instr 1
  kamp = 10000
  kcps = cpspch(p4)
  ifn = 1
  a1 oscil kamp, kcps, ifn
  out a1
  gamix = gamix + a1
endin

instr 2
  kamp = 10000
  kcps = 1
  ifn = 2
  ibas = 1
  a1 loscil kamp, kcps, ifn, ibas
  out a1
  gamix = gamix + a1
endin

instr 98
  krvt = p4
  ilpt = p5
  a99 comb gamix, krvt, ilpt
  out a99
  gamix = 0
endin

instr 99
  krvt = p4
  klpt line p5, p3, p6
  imaxlpt = p7
  a99 vcomb gamix, krvt, klpt, imaxlpt
  out a99
  gamix = 0
endin

</CsInstruments>

<CsScore>

f 1 0 16384 10 1
```

```
f 2 0 524288 1 "female.aif" 0 4 0

; reverbs usando comb
i 1 0 .1 8.05
i 1 1 .1 8.02
i 1 2 .1 8.05
i 1 3 .1 8.04
i 1 4 .1 8.05
i 1 5 .1 8.07
i 1 6 .1 8.05
i 1 7 .1 8.07
i 1 8 .1 8.04
i 1 9 .1 8.09
i 98 0  10  1   0.1
s

i 2  0 6
i 98 0 10  1  0.1

i 2  10 6
i 98 10 10 3  0.3

i 2  20 6
i 98 20 10 10 1
s

; reverbs usando vcomb
i 1 0 .1 8.05
i 1 1 .1 8.02
i 1 2 .1 8.05
i 1 3 .1 8.04
i 1 4 .1 8.05
i 1 5 .1 8.07
i 1 6 .1 8.05
i 1 7 .1 8.07
i 1 8 .1 8.04
i 1 9 .1 8.09
i 99 0 10  1  0.05 0.5 0.5
s

i 2  0 6
i 99 0 10  1  0.1 0.3 0.3

i 2  10 6
i 99 10 10 3  0.1 0.6 0.6

i 2  20 6
i 99 20 10 10 0.1 1.2 1.2
s

e

</CsScore>

</CsoundSynthesizer>
```

Fig. 5: *combs.csd*

Distorção

Descoberto por acidente na década de 50, um dos efeitos mais usados é a distorção, ou *overdrive,* obtido originalmente quando se coloca amplificadores de tubo em seu volume máximo. Ao contrário de manter o som original aumentando apenas sua amplitude, os amplificadores causavam uma alteração não-linear no som, causando clipping no sinal e criando novos harmônicos.

No Csound temos o opcode **distort1,** criado por Hans Mikelson, que simula a distorção através de *wave shaping,* ou alteração da forma de onda original. Apesar de não simular exatamente o overdrive, ele é o primeiro passo do processo em que obteremos uma distorção semelhante a dos amplificadores de tubo. A sintaxe de **distort1** é:

```
ares distort1 asig, kpregain, kpostgain, kshape1, kshape2
```

Todos os parâmetros são de tempo de performance. **Asig** é o sinal de entrada, **kpregain** e **kpostgain** são os valores de pré-amplificação e ganho de saída, e **kshape1** e **kshape2** são os valores que determinarão a quantidade de clipping do sinal, respectivamente na parte positiva e negativa. O valor 0 resulta em um clipping drástico do sinal, e para valores positivos pequenos há menos clipping e uma inclinação maior na crista da onda.

A distorção obtida com **distort1** ainda não é suficiente para alcançarmos o efeito de um verdadeiro tube amp em volume máximo. Para isso devemos fazer com que a variação positiva da onda sofra um atraso maior do que o restante, e podemos obter esse delay variável através de **deltap**.

Já vimos anteriormente que pode ser obtido o delay de um sinal usando o par **delayr** e **delayw**. Entretanto entre esses dois opcodes podemos variar o atraso que será usado na saída usando o opcode **deltap.** A sintaxe de **deltap** é simplesmente:

```
ares deltap kdlt
```

Onde **kdlt** é um sinal de controle que diz o tempo de atraso que será submetido o sinal de saída a cada instante.

Quando especificamos um tempo de delay em **delayr,** podemos imaginar que estabelecemos uma linha de condução do sinal mais lenta que seu tempo normal. Imagine um sinal viajando por um fio e de repente entrando numa linha de condução mais lenta, de modo que ao sair do outro lado dessa linha tenha sofrido um atraso especificado em **delayr**. Podemos entender o funcionamento de **deltap** como uma segunda linha desviando o sinal no meio da linha de atraso, fazendo com que ele chegue mais rápido à saída. O que **deltap** especifica é em que ponto do atraso esse sinal será desviado ao longo da duração do sinal, fazendo com que ele chegue à saída mais rápido ou mais tarde, variando com diferentes atrasos o sinal de entrada.

A especificação do tempo **kdlt** em **deltap** pode variar desde um ciclo de controle apenas (*k-pass)* até o tempo de atraso máximo especificado por **delayr**. É importante lembrar que quando declaramos **deltap**, o sinal ainda não é modificado. Apenas quando escrevemos o sinal com **delayw** é que ele é submetido às variações de atraso especificadas por **deltap.**

O exemplo a seguir de distorção nos dá um efeito extremamente semelhante à distorção de um tube amp, e foi implementado por Hans Mikelson, e é exposto aqui com leves alterações sintáticas.

No score abaixo usaremos a função **GEN05**, que constrói ondas a partir de segmentos de curvas exponenciais. Sua sintaxe é:

f # time size 5 a n1 b n2 c ...

A sintaxe é idêntica a de **GEN07**, ou seja, **a, b, c** são os valores que a onda deve assumir após **n1, n2,** etc pontos. A única

diferença é que os valores da onda **a, b, c,** (...) não podem ser zero, por ser uma função baseada na exponencial.

```
<CsoundSynthesizer>

<CsOptions>

-o distort1.wav

</CsOptions>

<CsInstruments>

; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

gadist init 0

instr 1
  iamp = 1000
  ifqc = cpspch(p4)
  asig pluck iamp, ifqc, ifqc, 0, 1
  gadist = gadist + asig
endin

instr 98
  ipregain   init p4
  ipostgain  init p5
  ishape1 init 0
  ishape2 init 0

  aout distort1 gadist, ipregain, ipostgain, ishape1, ishape2

  out aout

  gadist = 0
endin

instr 99
  ipregain  init p4
  ipostgain init p5
  ishape1  init      0
  ishape2  init      0
  asign   init       0

  adist distort1 gadist, ipregain, ipostgain, ishape1, ishape2

  aold    =       asign          ; Salva o último sinal
  asign   = gadist/30000         ; Normaliza o sinal atual

  atemp   delayr  .1

  ; Atraso baseado na amplitude e declividade do sinal.
  ; (Quanta menor a amplitude com sinal,
  ; e maior a diferença entre a amplitude atual e a amplitude anterior,
  ; maior será o atraso.)
  ; Você pode imaginar que ele “esticaria” a parte negativa,
  ; acentuando os picos da parte positiva
  ; e aumentando o efeito de distorção

  aout    deltapi (2-asign)/1500 + (asign-aold)/300
  delayw  adist
  out aout
  gadist = 0
```

```
endin

</CsInstruments>

<CsScore>

f 5 0   8192    8   -.8 336 -.78 800 -.7 5920 .7 800 .78 336 .8

i 1  0.0  1.6  6.04
i 1  0.2  1.4  6.09
i 1  0.4  1.2  7.02
i 1  0.6  1.0  7.07
i 1  0.8  0.8  7.11
i 1  1.0  0.6  8.04
i 98 0    1.6  2       1
s

i 1  0.0  1.6  6.04
i 1  0.2  1.4  6.09
i 1  0.4  1.2  7.02
i 1  0.6  1.0  7.07
i 1  0.8  0.8  7.11
i 1  1.0  0.6  8.04
i 99 0    1.6  4       1
s

i 1  0.0  0.2  7.00
i 1  0.1  0.2  7.02
i 1  0.2  0.2  7.04
i 1  0.3  0.2  7.05
i 1  0.4  0.2  7.07
i 1  0.5  0.2  7.09
i 1  0.6  0.2  7.11
i 1  0.7  0.2  8.00
i 1  0.8  0.2  8.02
i 1  0.9  0.2  8.04
i 1  1.0  0.2  8.05
i 1  1.1  0.2  8.07
i 1  1.2  0.2  8.11
i 1  1.3  0.2  9.00
i 1  1.4  0.2  9.02
i 1  1.5  0.2  9.04
i 99 0    1.6  4       1
s

e

</CsScore>

</CsoundSynthesizer>
```

Fig. 6: *distort1.csd*

Em *distort1.csd* usamos a mesma estrutura das orchestras de reverb, com instrumentos separados, um para a geração do som e outro para o efeito em si. No **instr 98** isso não seria necessário a princípio, pois usamos a distorção simplesmente com o opcode **distort1**, mas preferimos essa estrutura por ser mais coerente e legível na implementação dos dois efeitos de distorção, já que o segundo efeito **instr 99** usa delay e por isso precisa dessa separação. Além disso, essa estrutura simula melhor um efeito de pedal ou *stomp box***,** podendo ser ligado ou desligado a qualquer momento, independente do que está sendo tocado.

Capítulo 6

Combinando efeitos

Normalmente queremos usar mais de um efeito ao mesmo tempo, e desejamos estabelecer em qual ordem eles serão aplicados ao sinal, assim como escolheríamos a ordem em que ligaríamos vários pedais em uma guitarra. A maneira mais eficiente e organizada de se fazer isso em Csound é utilizando os opcodes da família **Zak**, criados por Robin Whittle, que simula um patch panel com entradas e saídas de sinal, onde podemos ligar cabos da saída de um efeito e conectá-lo com a entrada de outro efeito, em qualquer ordem.

Os opcodes Zak estabelecem canais onde podem ser escritos sinais que depois serão lidos por outro instrumento. Os três opcodes essenciais são **zakinit, zaw** e **zar.**

Zakinit é usado para a inicialização, e diz quantos sinais de a-rate e k-rate serão usados entre os instrumentos. A sintaxe de **zakinit** é:

```
zakinit isizea, isizek
```

Em **isizea** dizemos quantos sinais a-rate serão armazenados para depois serem lidos, e em **isizek** dizemos quantos sinais k-rate serão armazenados. Com essas informações é alocada a memória que será usada para escrita e leitura entre os instrumentos.

Para executar **zakinit** apenas uma vez, você deve colocá-lo antes dos instrumentos, após o cabeçalho da orchestra.

O par de opcodes **zar** e **zaw** escrevem e lêem de um determinado canal, tipicamente em instrumentos diferentes. A sintaxe de **zaw** é:

zaw asig, kndx

Onde **asig** é o sinal de entrada e **kndx** é número do canal em que ele será escrito.

A sintaxe de **zar** é:

ares **zar** kndx

Em **ares** é lido o sinal que foi escrito anteriormente no canal **kndx** por **zaw.**

Vamos agora colocar em uma mesma orchestra vários efeitos vistos no capítulo anterior, e vamos controlar quais serão utilizados e em que ordem, apenas alterando os canais de entrada e saída em cada instrumento, como se estivéssemos ligando cabos entre pedais de efeito.

Iremos alterar os instrumentos que utilizamos no capítulo anterior de modo que haja apenas um gerador de sinal (em nossa analogia, a guitarra), e os instrumentos subsequentes apenas lerão e modificarão o sinal anterior.

Para o efeito de flanger por exemplo, tiraremos dele o opcode **pluck** que gerava o sinal inicial, e substituiremos por um opcode **zar** que lerá o sinal que foi direcionado para seu canal de entrada, e no final do instrumento teremos um opcode **zaw** para escrever o sinal modificado para um próximo instrumento na linha de efeitos.

O instrumento gerador **instr 1** usa o opcode **soundin,** criado por Barry Vercoe, na linha:

```
asig soundin "female.aif"
```

que simplesmente armazena o arquivo *"female.aif"* na variável **asig**.

Abaixo temos o código completo. A partir de agora omitiremos o arquivo de saída em **CsOptions,** e o áudio irá direto para os alto-falantes.

```
<CsoundSynthesizer>

<CsOptions>
</CsOptions>

<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

zakinit 4, 1

instr 1
  iinch = p4
  ioutch = p5

  asig soundin "female.aif"
  zaw asig, ioutch
endin

instr 2
  iinch = p4
  ioutch = p5
  krvt = p6 ;1

  asig zar iinch
  arev reverb asig, krvt
  zaw arev, ioutch
endin

instr 3
  iinch = p4
  ioutch = p5

  asig zar iinch
  zacl
  out asig
endin


</CsInstruments>

<CsScore>

i1 0 10 1 1
i2 0 10 1 2 1
i3 0 10 2 3

</CsScore>

</CsoundSynthesizer>
```

Fig. 1: *zak.csd*

Note que controlamos os canais de entrada e saída através do score, assim **instr 2** lê do canal 1 e escreve no canal 2, **instr 2** lê do canal 2 e escreve no 3, que finalmente lê do 3 e escreve na saída. Alterando o quarto e o quinto parâmetro do score, que aqui especificam o canal de entrada e saída, podemos colocar quaisquer efeitos em qualquer ordem.

Em cada instrumento temos as variáveis **iinch** e **ioutch,** que são atribuídas com **p4** e **p5** para os canais de entrada e saída. Subsequentemente é lido o sinal de entrada usando o opcode **zar**, processado o som, e escrito no canal de saída usando o opcode **zaw**.

Agora que já aprendemos como funciona os opcodes Zak nesse exemplo mais simples, vamos incrementar nossa linha de efeitos num exemplo um pouco mais elaborado, assim **instr 1** será o gerador do sinal lendo o arquivo "*female.aif*"**, instr 95** será o efeito de distorção, o **instr 96** será o flanger, **instr 97** será o reverb, e o **instr 98** será o delay. Finalmente **instr 99** será apenas o instrumento que escreve o som processado na saída.

Além dos parâmetros **iinch** e **ioutch** definidos usando **p4** e **p5**, todos os instrumentos terão o parâmetro **igain,** que será definido com o parâmetro **p6** do score. Isso é necessário porque quando encadeamos muitos efeitos, é natural que o sinal final esteja fora de escala, e precisamos por isso atenuá-lo à medida que ele passa pelos efeitos.

```
<CsoundSynthesizer>

<CsOptions>
</CsOptions>

<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

zakinit 5, 1

instr 1
  iinch = p4
  ioutch = p5
  igain = p6

  asig soundin "female.aif"
  zaw asig*igain, ioutch
endin

instr 95
  iinch = p4
  ioutch = p5
  igain = p6

  ipregain = p7
  ipostgain = p8
  ishape1 = 0
  ishape2 = 0

  asign init 0

  asig zar iinch
  adist distort1 asig, ipregain, ipostgain, ishape1, ishape2

  aold = asign         ; Salva o último sinal
  asign = asig/30000   ; Normaliza o novo sinal
```

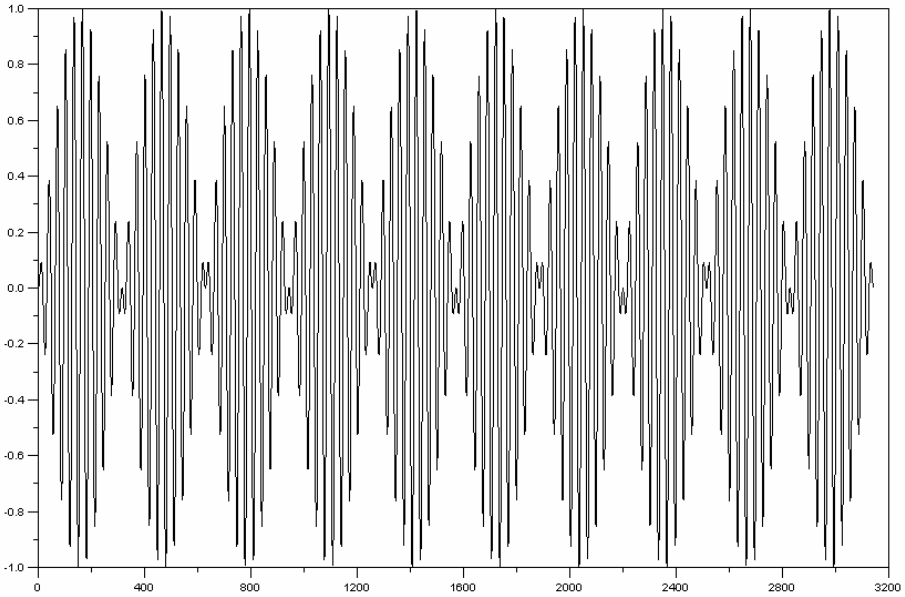
```
  atemp delayr  .1                  ; Faz um atraso baseado na declividade
  aout deltapi (2-asign)/1500 + (asign-aold)/300
  delayw  adist

  zaw aout*igain, ioutch
endin

instr 96
  iinch = p4
  ioutch = p5
  igain = p6

  initdel = p7 ;0.01
  ienddel = p8 ;0.07
  kfeedback = p9 ;0.7

  asig zar iinch
  adel line initdel, p3, ienddel
  aflang flanger asig, adel, kfeedback
  zaw aflang*igain, ioutch
endin

instr 97
  iinch = p4
  ioutch = p5
  igain = p6

  krvt = p7 ;1

  asig zar iinch
  arev reverb asig, krvt
  zaw arev*igain, ioutch
endin

instr 98
  iinch = p4
  ioutch = p5
  igain = p6

  krvt = p7
  ilpt = p8

  asig zar iinch
  aout comb asig, krvt, ilpt
  zaw (asig+aout)*igain, ioutch
endin

instr 99
  iinch = p4
  ioutch = p5
  igain = p6

  asig zar iinch
  zacl
  out asig*igain
endin

</CsInstruments>

<CsScore>


;entrada:    iinch  ioutch igain
i1  0 10     1      1      0.5

;distorção:  iinch  ioutch igain  pregain      postgain
i95 0 10     1      2      0.5    2            1
```

```
;flanger:    iinch  ioutch igain  initdel      ienddel      ifeedback
i96 0 10     2      3      0.5    0.01         0.07         0.8

;reverb:     iinch  ioutch igain  krvt
i97 0 10     3      4      0.5    5

;delay:      iinch  ioutch igain  krvt   ilpt
i98 0 20     4      5      0.5    90     5

;saída:      iinch  ioutch igain
i99 0 20     5      1      0.5

</CsScore>

</CsoundSynthesizer>
```

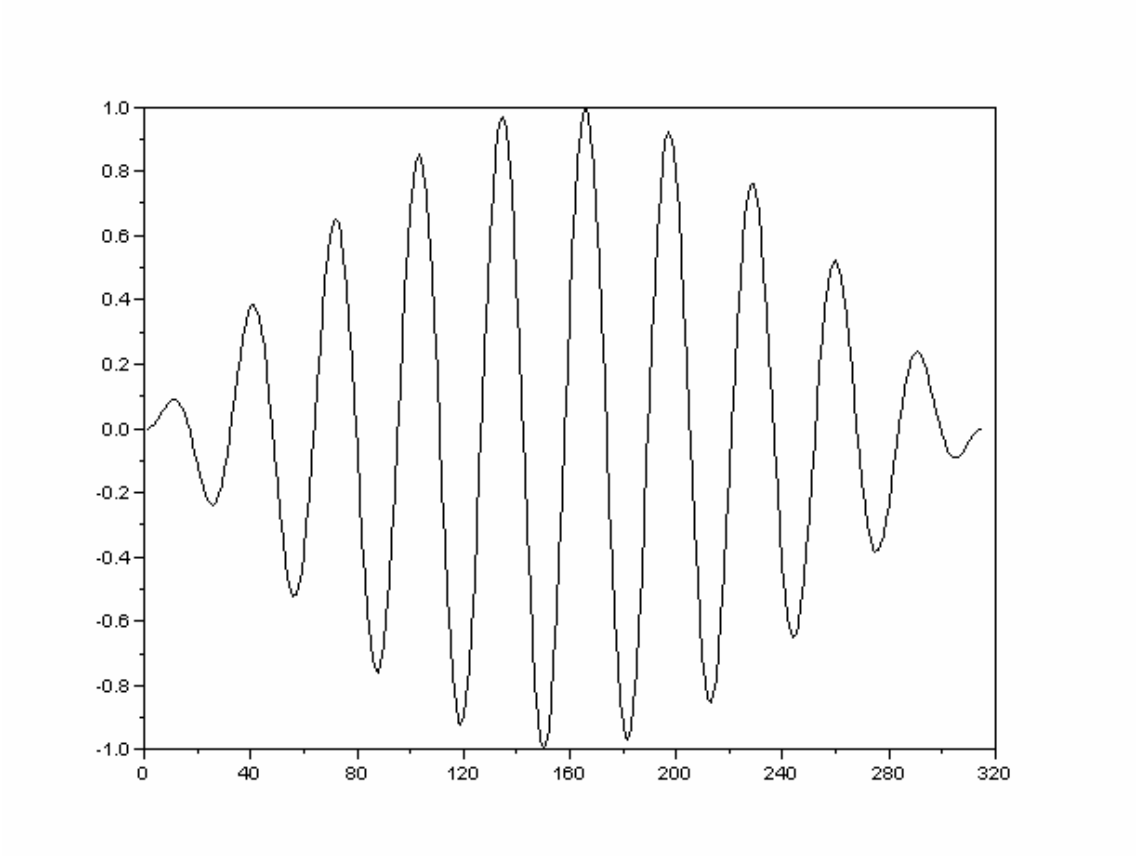
Fig.2: *zak2.csd*

Capítulo 7
Síntese Granular

O método de síntese granular no CSound é uma técnica que oferece resultados surpreendentes com baixo custo computacional. No tipo de síntese granular implementado no CSound e no qual estamos interessados, a idéia básica é gerar, a partir de um som inicial, vários “grãos” desse som encadeados. Normalmente esse sample inicial dura apenas alguns segundos, como a gravação de uma nota no piano ou uma senóide gerada no computador. O sample então é recortado em várias partes e cada uma delas é modulada por uma função de *envelope* gerando os grãos. O encadeamento desses grãos de som gera o efeito sonoro característico da síntese granular. A figura abaixo mostra uma onda típica de vários grãos encadeados:



Nesse caso mais simples o sample e a forma do envelope fôram uma senóide, sem haver intervalos de silêncio entre os grãos. A onda de um único grão seria:



Esse caso é útil para dar uma idéia intuitiva e visual do que se trata a síntese granular, mas ela vai muito além disso. No CSound os samples podem ser qualquer arquivo de áudio; o tamanho médio de cada grão pode ser definido, assim como o espaçamento entre eles; a curva de ataque e decaimento pode ser determinada separadamente, e muitos outros parâmetros. Tipicamente, várias vozes são utilizadas para gerar o arquivo de saída, o que caracteriza a síntese granular como um método de síntese aditiva.

Atualmente existem quatro opcodes de síntese granular no CSound: **granule**, criado por Allan Lee, **grain**, de Paris Smaragdis, e **grain2** e **grain3**, de Istvan Varga. Eles podem ser agrupados como os de fácil utilização, como **grain** e **grain2**, e os de maior controle, como **granule** e **grain3**. Vamos começar pelo **grain**, passando depois por **grain2, granule** e **grain3**, e ao final você terá a escolha de qual se adequa mais ao seu tipo de som desejado e o controle necessário.

Grain

Grain foi o primeiro opcode granular a ser implementado no CSound e é bastante intuitivo, apesar da quantidade de parâmetros fazê-lo um pouco mais complexo que a maioria dos opcodes de Csound. A sintaxe de **grain** é:

```
ares grain xamp, xpitch, xdens, kampoff, kpitchoff, kgdur, igfn, iwfn,
           imgdur [, igrnd]
```

Os parâmetros definidos em tempo de inicialização são:

. *igfn* determina que f-table será usada como sample.
. *iwfn* diz qual f-table será usada como *envelope* de amplitude para os grãos.

. *imgdur* é a duração máxima em segundos de um grão.

. *igrand* é um parâmtro opcional, com default 0. Para valores diferentes de zero, ele faz com que grain não leia mais aleatoriamente partes do sample, mas comece sempre a ler do início. Normalmente é interessante mantê-lo no default (aleatoriedade de leitura), pois isso é um dos fatores que diferencia o som de **grain** dos outros opcodes de síntese granular.

E os parâmetros para tempo de performance são:

. *xamp* é a amplitude média dos grãos. Quando a quantidade de grãos simultâneos aumenta é necessário diminuir esse valor, pois as ondas sobrepostas a aumentarão involuntariamente.

. *xpitch* é a freqüência inicial dos grãos. A princípio a freqüência é escolhida pelo usuário independente da freqüência original do sample, mas para usar a freqüência original, se o sample possuir exatamente um comprimento de onda, é possível obtê-la através da fórmula:

Frequência original = Taxa de amostragem / Tamanho da f-table

De fato, para se chegar a essa fórmula basta lembrar que:
*Tamanho da f-table em pontos = Taxa de amostragem * Período da amostra em segundos.* Isso nos leva a: *Tamanho da f-table = Taxa de amostragem / Frequência da amostra*

Na prática, se a taxa de amostragem fôr de 44100, e usando a função *ftlen* para obter o tamanho da amostra em pontos, conseguimos a frequência original do sample com a linha de código:

```
xpitch = 44100/ftlen(igfn)
```

. *xdens* é a densidade de grãos, diz quantos grãos por segundo a onda de saída deverá ter.

No caso típico em que **xdens** é constante, a saída será parecida com a gerada pelo opcode *fof* , sendo uma saída de síntese granular síncrona. No caso em que ela é aleatória a saída será mais parecida com síntese granular assíncrona, e o efeito sonoro é como se houvesse um ruído adicional.

. *kampoff* ou *amplitude-offset* é o deslocamento máximo que as amplitudes dos grãos poderão ter em relação à amplitude original do parâmetro xamp. Assim a amplitude de saída variará entre os valores de **xamp** e **xamp+kampoff**.

. *kpitchoff* ou *pitch-offset* é similar a **kampoff**, será o deslocamento máximo em relação à freqüência original, com a freqüência de saída dos grãos variando entre **xpitch** e **xpitch+kpitchoff**..

. *kgdur* é a duração dos grãos em segundos, limitada superiormente pelo parâmetro **imgdur**. Se em algum momento **kgdur** ultrapassar **imgdur**, ela é truncada.

No exemplo a seguir veremos a utilização de **grain** em quatro configurações diferentes, uma para cada um dos nossos quatro instrumentos. Poderíamos ter feito um único instrumento e parametrizado na seção de score, mas preferimos colocar as alterações dentro dos instrumentos, por ser mais claro e organizado.

```
<CsOptions>

-o grain.wav

</CsOptions>

<CsInstruments>

sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

; variações leves na frequência e na amplitude dos grãos
instr 1
    ifn = 1
    ienv = 2
    icps = cpspch(p4)
    iamp = ampdb(60)
    idens  = 1600
    iaoff  = ampdb(10)
    ipoff  = 10
    igdur  = .1
    imaxgdur =  .5

    ar  grain iamp, icps, idens, iaoff, ipoff, igdur, ifn, ienv, imaxgdur

    out ar
endin

; variação de até uma oitava na frequência dos grãos
instr 2
    ifn = 1
    ienv = 2
    icps = cpspch(p4)
    iamp = ampdb(50)
    idens = 1600
    iaoff = ampdb(10)
    ipoff = icps
    igdur = .1
    imaxgdur =  .5

    ar  grain iamp, icps, idens, iaoff, ipoff, igdur, ifn, ienv, imaxgdur

    out ar
endin

instr 3
    ifn = 1
    ienv = 2
    icps = cpspch(p4)
    kamp line ampdb(50), p3, ampdb(70)
    kdens linseg 1600, p3, 10
    iaoff  = ampdb(10)
    kpoff line icps, p3, icps*7
    igdur = .1
    imaxgdur =  .5

    ar  grain kamp, icps, kdens, iaoff, kpoff, igdur, ifn, ienv, imaxgdur

    out ar
endin

instr 4
    ifn = 1
    ienv = 2
    icps = cpspch(p4)
```

```
    iamp = ampdb(70)
    idens = 10
    iaoff  = ampdb(10)
    ipoff = icps*7
    igdur = .1
    imaxgdur =  .5

    ar  grain iamp, icps, idens, iaoff, ipoff, igdur, ifn, ienv, imaxgdur

    out ar
endin


</CsInstruments>

<CsScore>

; Tabela #1: uma simples onda de seno usando GEN10.
f 1 0 16384 10 1

; Tabela #2: função de envelope usando GEN7.
f 2 0 16384 7 0 4096 5000 4096 2500 4096 2500 4096 0

i 1 0  4 8.09
i 2 5  4 8.09
i 3 10 9 8.09
i 4 20 5 8.09
e

</CsScore>

</CsoundSynthesizer>
```

Fig. 1: *grain.orc*: quatro instrumentos diferentes para quatro configurações de grain.

No primeiro instrumento temos grain com uma densidade **kdens** de 1600 grãos de som por segundo, e uma leve variação **ipoff** na frequência **icps** e outra leve variação **iaoff** na amplitude. Trata-se de uma síntese granular com densidade regular e amplitude e frequência quase constantes. Nesse caso podemos perceber a frequência fundamental da nota, que aqui é A4.

No segundo instrumento temos uma variação de frequência entre a frequência fundamental **icps** e sua oitava acima **icps** + **ipoff**. Aqui os outros parâmetros permanecem constantes com uma leve variação **iaoff** na amplitude **iamp**. Já não podemos perceber uma frequência fundamental, dada as variações bruscas na frequência, talvez assemelhando-se um pouco ao som do vento.

Antes de falarmos do terceiro instrumento, falaremos do quarto por se tratar de um caso particular do terceiro. Temos aqui um uso bastante não-ortodoxo do opcode **grain**. Definimos sua densidade **idens** para apenas 10 grãos por segundo, cada um com uma duração média **igdur** de um décimo de segundo, e a frequência variando entre **icps** e **icps+ipoff**, três oitavas acima da frequência fundamental. Pela baixa densidade e duração dos grãos, podemos perceber cada grão individualmente, com suas frequências se alterando aleatoriamente, produzindo um som, dependendo da sua imaginação, semelhante à uma peça aleatória ou à um computador de seriados antigos de TV.

O terceiro instrumento é uma transição gradual do segundo para o quarto intrumento, com a amplitude **kamp,** a densidade **kdens** e o deslocamento de frequência **kpoff** variando entre os valores dos dois instrumentos através de segmentos **line** e **linseg**.

O score toca os quatro instrumentos com duração de quatro segundos, exceção feita para o terceiro instrumento, com uma duração maior.

Grain2

Grain2 foi criado para se dar uma alternativa mais simples a **grain3**, que veremos depois.

A sintaxe de **grain2** é dada por:

```
ares grain2 kcps, kfmd, kgdur, iovrlp, kfn, iwfn [, irpow] [, iseed]
[, imode]
```

Os parâmetros de tempo de inicialização são:

. iovrlp é o número de grãos que podem se sobrepor.

. iwfn é o índice em que está a f-table de envelope.

. irpow é um parâmetro opcional, com valor default 0. Ele determina a distribuição das frequências pseudo-aleatórias. Para valores de 0, 1 e -1 essa distribuição é uniforme.

Para mais detalhes sobre as fórmulas que governam essa distribuição, por favor consulte o *Csound Reference Manual*.

. iseed é um valor opcional, com default 0. Ele determina qual será a semente positiva a ser usada para se começar a geração de números randômicos, podendo ser um valor de até 32 bits com sinal. Se valores negativos ou zero são usados, a semente será o tempo atual, que é o default.

. imode também é opcional com default 0, e é a soma dos valores:

* 8 para interpolar os envelopes, fazendo uma passagem suave entre um e outro
* 4 para não interpolar a forma de onda dos grãos, sem fazer uma passagem suave
* 2 para modificar a frequência dos grãos sempre que **kcps** é modificado, em vez de mudá-la apenas a partir do próximo grão
* 1 para não fazer a inicialização.

Os parâmetros de tempo de performance são:

. kcps é a frequência fundamental dos grãos.

. kfmd é o deslocamento aleatório máximo em Hertz da frequência, para mais ou para menos.

. kgdur é a duração dos grãos em segundos a cada instante. Internamente, sso é feito pelo opcode mudando-se a velocidade que ele lê o sample, o que muda inclusive a duração dos grãos já começados.

. kfn é o índice da f-table contendo o sample. Note que ele pode ser definido em tempo de performance como um parâmetro k-rate, ao contrário do parâmetro **ifn** estático de **grain**.

Apesar da quantidade de parâmetros de **grain2**, a maioria é opcional. Portanto uma versão simplificada da sintaxe de **grain2** seria apenas:

```
ares grain2 kcps, kfmd, kgdur, iovrlp, kfn, iwfn
```

Vamos fazer agora a versão para **grain2** dos instrumentos usados com **grain** do exemplo anterior. Teremos novamente quatro instrumentos, para que póssamos fazer uma comparação direta entre **grain2** e **grain**.

```
<CsoundSynthesizer>

<CsOptions>

-o grain2.wav

</CsOptions>

<CsInstruments>

sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

giamp = 1000

; Variação de frequência pequena
instr 1
    kcps = cpspch(p4)
    kfmd  = 10 ; variação é bipolar
    kgdur  = .1
    iovrlp = 160 ; 1600 por segundo
    kfn = 1
    iwfn = 2 ; note que não há imaxgdur nem kamp, e iovrlp é i ao invés de k
como kdens
    irpow = 0
    iseed = 0
    imode = 10

    ares grain2 kcps, kfmd, kgdur, iovrlp, kfn, iwfn, irpow, iseed, imode

    out ares*giamp

endin

; variação de frequência média
instr 2
    kcps = cpspch(p4)
    kfmd  = kcps ; agora a variação de frequência é maior
    kgdur  = .1
    iovrlp = 160
    kfn = 1
    iwfn = 2
    irpow = 0
    iseed = 0
    imode = 10

    ares grain2 kcps, kfmd, kgdur, iovrlp, kfn, iwfn, irpow, iseed, imode

    out ares*giamp
endin

; variação de frequência começando pequena e indo a grande
instr 3
    icps = cpspch(p4)
    kfmd line icps, p3, icps * 7
    kgdur  = .1
    iovrlp = 160 ; nao posso fazer linseg 160, p3, 1
    kfn = 1
    iwfn = 2
    irpow = 0
    iseed = 0
```

```
    imode = 10

    ares grain2 icps, kfmd, kgdur, iovrlp, kfn, iwfn, irpow, iseed, imode

    out ares*giamp
endin

; variação de frequência grande, baixa densidade de grãos
instr 4
    kcps = cpspch(p4)
    kfmd = kcps * 7
    kgdur  = .1
    iovrlp = 1 ; 10 grãos por segundo
    kfn = 1
    iwfn = 2
    irpow = 0
    iseed = 0
    imode = 10

    ares grain2 kcps, kfmd, kgdur, iovrlp, kfn, iwfn, irpow, iseed, imode

    out ares*giamp
endin

</CsInstruments>

<CsScore>

; Tabela #1: uma simples onda de seno usando GEN10.
f 1 0 16384 10 1

; Tabela #2: função de envelope usando GEN7.
f 2 0 16384 7 0 4096 5000 4096 2500 4096 2500 4096 0

; Variação de frequência constante e pequena
i 1 0  4 8.09

; variação de frequência constante e média
i 2 5  4 8.09

; variação de frequência indo de média para grande
i 3 10 9 8.09

; variação de frequência grande, baixa densidade de grãos
i 4 20 4 8.09

e

</CsScore>

</CsoundSynthesizer>
```

Fig.2: *grain2.csd*, a versão para grain2 dos quatro instrumentos do exemplo anterior.

Para que possamos comparar os dois opcodes **grain e grain2,** é necessário fazer um parêntese. Em Csound podemos classificar as ondas em dois tipos: unipolar e bipolar. Ondas unipolares assumem apenas valores positivos, e são usadas normalmente para envelopes. Ondas bipolares, mais comuns, assumem valores positivos e negativos, como o caso da senóide.

Em **grain** tinhamos o parâmetro **kpitchoff**, que era a variação da frequência **xpitch**. Essa variação era unipolar, ou seja, kpitchoff era sempre positivo e a frequência variava apenas entre os valores **xpitch** e **xpitch+kpitchoff.** Entretanto em **grain2** a variação de frequência **kfmd** é bipolar, assume valores positivos e negativos.

Além disso, em **grain2** não especificamos diretamente a densidade de grãos por segundo, como em **grain** com o parâmetro **kdens**, mas temos o parâmetro semelhante **iovrlp**, que é a quantidade de grãos que podem se sobrepor a qualquer momento. Então se antes tínhamos uma densidade de grãos **kdens** de 1600 grãos por segundo, podemos alcançar essa mesma densidade através da fórmula **kdens = iovrlp/kgdur.**

Ou seja, apesar de não termos o parâmetro **kdens**, sabemos que a quantidade de grãos simultâneos **iovrlp** dividida pela duração de cada grão **kgdur** nos dá a densidade de grãos por segundo **kdens.** Por exemplo, se geramos dois grãos simultâneos, mas esses dois grãos durarem 10 segundos até podermos gerar outros dois grãos, teremos uma densidade de *2 / 10 = 0.2* grãos por segundo.

No nosso primeiro instrumento **instr 1** definimos a duração dos grãos **kgdur** como um décimo de segundo, e definimos que haveria 160 grãos sobrepostos a cada momento, ou seja, **iovrlp** é de 160 grãos à cada décimo de segundo, o que nos dá uma densidade de 1600 grãos por segundo.

Entretanto existe uma diferença adicional que não se pode emular: **iovrlp** é do tipo **i-rate**, é definida apenas na inicialização, enquanto **kdens** é **k-rate**, atualizada pela taxa de controle. Essa será a diferença de parametrização em nosso terceiro instrumento, pois no exemplo de **grain** variávamos a densidade gradualmente, e aqui a densidade será fixa. Afora esse aspecto, podemos fazer uma comparação direta entre os sons produzidos por **grain** e **grain2**.

Outras diferenças que não afetarão nossos instrumentos é que **grain2** não possui os parâmetros **imaxgdur** nem **kamp.** De fato não precisamos aqui de **imaxgdur,** mas faremos uma leve alteração para usarmos um fator de amplitude em **grain2.** Basta pegarmos o sinal de saída **ares** e multiplicarmos pelo fator de amplitude **giamp**, como na linha:

```
    out ares*giamp
```

Para alcançar qualidade máxima na síntese de **grain2,** definimos **imode** como 10, isto é, interpolamos o sample e alteramos dinamicamente a frequência dos grãos a partir de **kcps** e **kfmd**.

Agora podemos renderizar **grain2.wav** e notar as diferenças de som entre os dois opcodes.

Granule

Apesar de **granule** ser mais complexo que **grain**, a maioria de seus parâmetros adicionais são de inicialização, não sendo necessário manipulá-los durante a performance. A sintaxe de **granule** é:

```
ares granule xamp, ivoice, iratio, imode, ithd, ifn, ipshift, igskip,
  igskip_os, ilength, kgap, igap_os, kgsize, igsize_os, iatt, idec
  [, iseed] [, ipitch1] [, ipitch2] [, ipitch3] [, ipitch4] [, ifnenv]
```

Os parâmetros de **granule** são:

. xamp a amplitude.

. ivoice é a quantidade de vozes simultâneas que **granule** gera. É a versão de **kdens** em **grain** e **iovrlp** em **grain2**.

. iratio é a velocidade de leitura relativa à taxa de amostragem **sr** da onda armazenada na **f-table**.

Uma **iratio** de 0.5 seria metade da velocidade, e esticaria um sample de 1 segundo para 2 segundos, ou poderíamos usar uma **iratio** de 10, que comprimiria esse sample para um décimo de segundo, e etc.

. imode define a direção de leitura do sample. +1 faz com que a leitura seja normal para frente, -1 para trás e com 0 a leitura é definida aleatoriamente para cada grão.

. ithd é o valor de *threshold*, ou limite mínimo de amplitude para o sample. Para pontos no sample que tiverem amplitude abaixo desse valor, eles serão descartados, podendo assim pular silêncios e filtrar ruídos.

. ifn é o índice da f-table que será usada como sample.

. ipshift é o número de frequências diferentes que serão usados nas vozes de saída. Se **ipchift** é definida como 0, a frequência de cada grão variará aleatoriamente para uma oitava acima ou abaixo da frequência fundamental. **Ipshift** também pode assumir os valores 1, 2, 3 e 4, sendo esse o número de frequências diferentes para os grãos de saída.

Os valores exatos que cada uma das frequências é definida, é dito nos parâmetros opcionais **ipitch1, ipitch2, ipitch3, ipitch4.** Para um valor de 0.5 por exemplo, essa frequência será uma oitava abaixo da fundamental, para 2 uma oitava acima e etc., i.e., os parâmetros **ipitch** multiplicam a frenquência fundamental.

. igskip é o tempo em segundos a partir do qual se começa a leitura do sample para gerar o grão

. igskip_os ou *skip offset* é o deslocamento aleatório máximo, para mais ou para menos, do tempo inicial **igskip** de leitura do sample.

. ilenght é o tempo de leitura da tabela em segundos, começando em **igskip+igskip_os**. Se **ilength** terminar antes do fim da f-table, apenas essa parte da tabela será usada para gerar o grão.

. kgap o intervalo médio de segundos entre os grãos, sem as alterações dadas por **igap_os**.

. igap_os é o deslocamento aleatório do intervalo de tempo entre os grãos, para mais ou para menos, em termos de porcentagem do intervalo padrão definido em **kgap.** Para **kgap** igual a 1 segundo e **igap_os** 10, haverá um deslocamento aleatório nesse tempo de até 10%.

. kgsize é a duração média dos grãos, sem as alterações de **igsize_os.**

. igsize_os é a variação em porcentagem do tamanho dos grãos definido em **kgsize,** semelhante à ação de **igap_os** em **kgap**.

. iatt é a percentagem do tempo de duração do grão que será usado para o envelope de ataque.

. idec é a percentagem da duração do grão destinada para o decaimento.

. iseed é opcional, com default 0.5. É o valor inicial para o gerador de números randômicos.

. ipitch1, ipitch2, ipitch3, ipitch4 são parâmetros opcionais com default 1. Para mais detalhes veja **ipshift** acima.

. ifnenv é opcional, com default 0 É a f-table usada como envelope dos grãos, para índices diferentes de 0. Normalmente não é necessário, pois **iatt** e **idec** já dão um envelope.

Como visto acima **granule** tem várias diferenças de **grain** e **grain2**. O que antes eram versões para densidade de grãos, aqui é o número de vozes independentes,

podendo ter até quatro frequências fundamentais diferentes. Dessa maneira, podemos gerar tríades e tetracordes a partir de um único **granule**.

Mas **granule** tem uma desvantagem, já que em nenhum parâmetro é dada a frequência fundamental dos grãos, e a frequência é exatamente a da onda contida na f-table usada para gerar os grãos. Assim, se na f-table estiver contida uma senóide de 440 Hz, essa será a frequência fundamental dos grãos. Em nosso exemplo usaremos como sample uma f-table que lê o arquivo **seno.wav,** que é a senóide gerada pelo primeiro instrumento desse livro.

```
<CsoundSynthesizer>

<CsOptions>

-o granule.wav

</CsOptions>

<CsInstruments>

sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

instr 1

kamp = 1000
ivoice = 100
iratio = 1
imode = 0
ithd = 0
ifn = 1
ipshift = 2
igskip = 0
igskip_os = 0.005
ilength = 2
kgap = 0.01
igap_os = 50
kgsize = 0.05
igsize_os = 50
iatt = 50
idec = 50
iseed = 0.5
ipitch1 = cpspch(p4)/440
ipitch2 = cpspch(p5)/440

a1 granule kamp, ivoice, iratio, imode, ithd, ifn, ipshift, igskip, \
         igskip_os, ilength, kgap, igap_os, kgsize, igsize_os, iatt, \
         idec, iseed, ipitch1, ipitch2

out a1
endin


</CsInstruments>

<CsScore>

; Tabela #1: leitura de uma senóide de 440 Hz usando GEN01.
f 1 0 262144 1 "seno.wav" 0 0 0

; repetir a seção 4 vezes
r 4

i 1 0 1 8.05 8.09
```

```
i 1 1 1 8.02 8.09
i 1 2 1 8.05 8.09
i 1 3 1 8.04 8.07
i 1 4 1 8.05 8.09
i 1 5 1 8.07 8.11
i 1 6 1 8.05 9.02
i 1 7 1 8.07 8.11
i 1 8 1 8.04 9.04
i 1 9 1 8.09 9.00
i 1 10 1 7.11 8.11

; fim da seção
s

i 1 0 1 8.05 8.09

e

</CsScore>

</CsoundSynthesizer>
```

Fig. 3: *granule.csd*, acordes gerados a partir de um único opcode **granule**.

Apenas recapitulando. Em nosso instrumento na Fig.3 usamos o opcode **granule** como segue:

- 100 vozes simultâneas em **ivoice**
- Leitura sem modificar a frequência original em **iratio**
- Direção de leitura aleatória em **imode**
- Sem limite mínimo de amplitude **ithd** para descartar pontos
- Duas frequências **ipshift** diferentes nas vozes
- Começar a ler do ínicio do sample em **igskip...**
- **...**com uma variação no tempo de leitura inicial de 0.005 segundos em **igskip_os...**
- **...**e a partir daí ler 2 segundos do sample, como especificado em **ilenght.**
- Um intervalo entre grãos padrão de 0.01 segundos em **kgap**...
- ...e variações nesse intervalo de 50% em **igap_os.**
- Tamanho padrão de grãos **kgsize** de 0.05 segundos...
- ...e variações de tamanho **igsize_os** de 50%.
- 50% da duração dos grãos para o ataque em **iatt...**
- **...**e **idec** também de 50% para o decaimento.
- O valor opcional da semente de números randômicos **iseed** é o default de 0.5.
- Finalmente definimos os valores das frequências **ipitch1** e **ipitch2** para formar os acordes definidos no **score** nos parâmetros **p4** e **p5** que estão no formato *octave-point-pitch-class.* Como a frequência final será **ipitch1****frequência da f-table,* para obtermos a frequência desejada de **cpspch(p4),** temos que definir **ipitch1** como **cpspch(p4)** / *frequência da f-table,* e fazer o mesmo com **ipitch2,** como ficou na nossa orchestra:

```
ipitch1 = cpspch(p4)/440
ipitch2 = cpspch(p5)/440
```

Em nosso **score** usamos duas novas declarações, **r** e **s**. A primeira declaração é da forma **r p1** e repete o trecho a seguir **p1** vezes até encontrar uma outra declaração **r, s** ou **e.** A declaração **s** marca o fim da seção, e se ela estiver sendo repetida retorna ao **r** anterior. Nesse **score** repetimos quatro vezes o trecho entre **r** e **s,** notando que quando a seção termina, os tempos de início das notas seguintes são reinicializados para zero.

Assim nosso **score** toca aproximadamente o trecho abaixo, um belo tema a duas vozes:

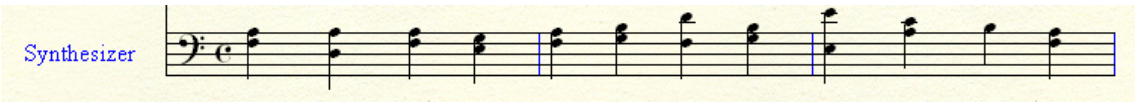


Fig. 4: a partitura aproximada do **score** de **granule.**

Grain3

Em muitos aspectos **grain3** é semelhante a **grain2**, basicamente trata-se de uma extensão, com parâmetros para variação de fase e de densidade que não haviam antes. Alguns desses parâmetros já são familiares para o leitor:

```
ares grain3 kcps, kphs, kfmd, kpmd, kgdur, kdens, imaxovr, kfn, iwfn,\
            kfrpow, kprpow [, iseed] [, imode]
```

Parâmetros de inicialização são:

. imaxovr é a quantidade máxima de grãos sobrepostos a qualquer momento. Esse número poderia ser obtido através de **kgdur*kdens**, mas via de regra esse cálculo superestima a quantidade de grãos simultâneos, tornando o processamento mais lento.

. iwfn é o índice da f-table contendo a onda de envelope.

. iseed é a semente de geração de números randômicos, opcional com default 0.

. imode também é opcional com default 0, é a soma dos valores abaixo, e a partir do valor 8 é idêntica a **grain2**:

* 64 para sincronizar a fase inicial dos grãos com **kcps**
* 32 para começar todos os grãos em valores inteiros de pontos no sample
* 16 não renderiza grãos com tempo de início negativo
* 8 para interpolar os envelopes
* 4 para não interpolar a forma de onda dos grãos
* 2 para modificar a frequência dos grãos sempre que **kcps** é modificado
* 1 para não fazer a inicialização.

Os parâmetros de performance são:

. kcps é a frequência fundamental dos grãos

. kphs é a fase padrão em que os grãos devem começar. **Kphs** é um valor entre 0 e 1 que diz em qual fração do comprimento da tabela começará a ser feita a leitura.

. kfmd é a variação aleatória em Hertz que a frequência pode ser alterada, e é bipolar (para mais ou para menos).

. kpmd é a variação aleatória bipolar em que a fase pode ser alterada.

. kgdur é a duração dos grãos em segundos.

. kdens é a densidade de grãos por segundo.

. kfrpow é o índice que controla a distribuição das frequências aleatórias, favor consultar **irpow** em **grain2.**

. kprpow controla a distribuição de fases aleatórias, semelhante a **irpow** em **grain2**.

. kfn é o índice da f-table da forma de onda dos grãos.

Em nosso exemplo usaremos todas as possibilidade de variações em **grain3**, com a exceção apenas de variar a forma de onda **kfn**. Usamos quatro instrumentos como em **grain** e **grain2**.

```
<CsoundSynthesizer>

<CsOptions>

-o grain3.wav

</CsOptions>

<CsInstruments>

sr = 44100
kr = 4410
ksmps = 1
nchnls = 1

; variação de fase
instr 1
    kamp = 1000
    kcps = cpspch(p4)
    kphs line 0, p3, 1
    kfmd = 10
    kpmd = 0
    kgdur = .1
    kdens = 160
    imaxovr = 160
    kfn = 1
    iwfn = 2
    kfrpow = 0
    kprpow = 0
    iseed = 0
    imode = 10

    ares  grain3 kcps, kphs, kfmd, kpmd, kgdur, kdens, imaxovr, kfn, \
                  iwfn, kfrpow, kprpow, iseed, imode

    out ares*kamp

endin

; densidade aumentando
instr 2
    kamp line 1000, p3, 5000
    kcps = cpspch(p4)
    kphs = 0.5
    kfmd = kcps  ; agora a variação de frequência é maior
    kpmd = 0.5
    kgdur line 1, p3, .1
    kdens linseg 1, p3/5, 10, p3/5, 100, p3*3/5, 1600
    imaxovr = 1600
    kfn = 1
    iwfn = 2
    kfrpow = 0
    kprpow = 0
    iseed = 0
    imode = 10

    ares  grain3 kcps, kphs, kfmd, kpmd, kgdur, kdens, imaxovr, kfn, \
                  iwfn, kfrpow, kprpow, iseed, imode
```

```
    out ares*kamp
endin

; frequência aumentando
instr 3
    kamp line 1000, p3, 5000
    icps = cpspch(p4)
    kphs = 0.5
    kfmd line icps, p3, icps * 7
    kpmd = 0.5
    kgdur line .1, p3, 1
    kdens linseg 1600, p3/2, 10, p3/2, 1
    imaxovr = 1600
    kfn = 1
    iwfn = 2
    kfrpow = 0
    kprpow = 0
    iseed = 0
    imode = 10

    ares  grain3 icps, kphs, kfmd, kpmd, kgdur, kdens, imaxovr, kfn, \
                  iwfn, kfrpow, kprpow, iseed, imode

    out ares*kamp
endin

; variação de frequência grande, baixa densidade de grãos
instr 4
    kamp line 1000, p3, 5000
    kcps = cpspch(p4)
    kphs = 0.5
    kfmd = kcps * 7
    kpmd = 0.5
    kgdur line .1, p3, 1
    kdens line 10, p3, 1
    imaxovr = 10
    kfn = 1
    iwfn = 2
    kfrpow = 0
    kprpow = 0
    iseed = 0
    imode = 10

    ares  grain3 kcps, kphs, kfmd, kpmd, kgdur, kdens, imaxovr, kfn, \
                  iwfn, kfrpow, kprpow, iseed, imode

    out ares*kamp
endin

</CsInstruments>

<CsScore>

; Tabela #1: uma simples onda de seno usando GEN10.
f 1 0 16384 10 1

; Tabela #2: função de envelope usando GEN7.
f 2 0 16384 7 0 4096 5000 4096 2500 4096 2500 4096 0

; variação de fase
i 1 0  9 8.09

; densidade aumentando
i 2 10 9 8.09

; variação de frequência e duração aumentando
i 3 20 9 8.09
```

```
; variação de frequência grande, baixa densidade de grãos
i 4 30 9 8.09

e

</CsScore>

</CsoundSynthesizer>
```

Fig. 5: *grain3.csd*, variações de densidade, frequência e durações

No primeiro instrumento variamos apenas a faze **kphs,** no segundo mantemos a variação de frequência **kfmd** constante, mas começamos com a duração dos grãos **kgdur** em um segundo e ao longo da nota muda para um décimo de segundo. A variação de densidade **kdens** foi feita em etapas com uma declaração **linseg:** .

```
kdens linseg 1, p3/5, 10, p3/5, 100, p3*3/5, 1600
```

No terceiro instrumento variamos simultaneamente a variação de frequência **kfmd,** a duração **kgdur** e a densidade **kdens,** com **kfmd** indo de uma oitava acima e abaixo, e aumentando até três oitavas com **kcps***7. A duração e a densidade fazem o caminho inverso do segundo instrumento, com **kgdur** indo de um décimo para um segundo e **kdens** indo de 1600 para 1 grão por segundo.

No quarto instrumento temos a nova versão do que foi usado em **grain2**, dessa vez além da variação de frequência ser grande e a densidade baixa, os grãos vão ficando com duração maior, dando uma sensação de retardo no andamento do tempo.

Fof

A técnica de síntese do opcode **fof** pode ser considerada de síntese granular, entretanto isso só ocorre com baixas frequências fundamentais. **Fof** gera um trem de pulsos como os opcodes anteriores, e a frequência fundamental aqui é na verdade a quantidade de pulsos gerados por segundo, isto é, ela se comporta como densidade dos grãos. Quando essa frequência aumenta ela deixa de ser a densidade e passa a ser realmente a fundamental de um outro tipo. Em outras palavras, quando temos frequências abaixo de 30 Hz, o opcode **fof** se comporta como de síntese granular, quando ela aumenta, temos a síntese timbral.

Essa dualidade do opcode **fof** o torna mais conveniente para esse segundo tipo de síntese, e em comparação com os opcodes anteriores ele possui vários parâmetros para controlar detalhadamente cada grão, mudando assim o timbre final.

A sintaxe de **fof** é**:**

```
ares fof xamp, xfund, xform, koct, kband, kris, kdur, kdec, iolaps,
         ifna, ifnb, itotdur [, iphs] [, ifmode] [, iskip]
```

Os parâmetros de inicialização são:

. iolaps é a quantidade máxima de grãos sobrepostos em qualquer momento. Pode ser estimada efetuando-se **xfund*kdur,** assim como fazíamos **xdens*kgdur** em **grain**.

. ifna é o índice da f-table que contém a forma de onda do grão

. ifnb é o índice que contém a forma de onda do envelope de ataque, e é feita sua reflexão para usar como envelope de decaimento.

. itotdur é a duração total do trem de pulsos, geralmente é o valor de **p3**.

Os três parâmetros de inicialização restantes são opcionais:

. iphs é a fase em que se inicia **ifna** e deve estar entre 0 e 1, tem default 0.

. ifmode nos diz, para valores positivos, se a frequência de um grão é variável durante a duração de **kdur,** seguindo a variação de frequência **xform,** e para o valor 0 (default) permanece com a frequência **xfund**.

Para que a frequêncie do grão varie continuamente, **ifmode** deve ser 1.

. iskip é por default 0, se fôr não-zero a inicialização é pulada.

Os parâmetros de tempo de performance são:

. xamp é a amplitude.

. xfund é a frequência de grãos por segundo. Para valores abaixo de 30 Hz se comporta como a densidade de grãos. Para valores acima, se torna a frequência fundamental da nota.

. xform é a frequência formante, ou seja, o harmônico mais alto que o grão deve ter.

. koct é o parâmetro de oitavação, normalmente 0. Para um número inteiro **x**, temos que a frequência fundamental é deslocada **x** oitavas para baixo. Para números fracionados, temos as frequências intermediárias entre as oitavas. O parâmetro **koct** realiza esse deslocamento atenuando grãos alternados para cada valor inteiro.

. kband normalmente é 0, para valores maiores ele acelera o decaimento natural dos grãos, antes da aplicação do envelope. Quanto maior **kband,** maior e mais rápida a atenuação.

. kris é o tempo de ataque do envelope em segundos.

. kdur é a duração dos grãos.

. kdec é o tempo de decaimento.

O opcode **fof** tem algumas semelhanças com os opcodes de síntese granular que vimos anteriormente. Neles havia uma variação assíncrona na emissão dos grãos, com duração e frequência variável, agora não possuímos mais esse controle, gerando um trem de pulsos síncronos. Por outro lado, temos mais controle do espectro de frequências de cada grão, especificando a fundamental e a frequência formante. Essa propriedade será aplicada para gerar síntese imitativa usando **fof.**

No código que veremos usamos o opcode **table,** que acessa diretamente um ponto em uma f-table através da sintaxe:

```
ires table indx, ifn
```

em que simplesmente armazenamos em **ires** o valor da posição **indx** da f-table **ifn.**

O opcode **fof** é muito usado para fazer síntese imitativa da voz humana. Isso é feito através de uma tabela, desenvolvida pra esse propósito, de cinco frequências formantes, cada uma com sua amplitude e banda. Isto é, cada vogal “a”, “e”, “i”, “o” ou “u” emitida, cada uma, tem cinco frequências formantes, com suas cinco amplitudes e cinco bandas.

Contruiremos as tabelas de formantes, amplitude e banda no score usando a **GEN02,** em que o valor de cada ponto é um parâmetro explicitado em sua declaração. Então na linha:

```
; baixo "a"
```

```
f40 0 16 -2 600 1040 2250 2450 2750 0  -7  -9  -9 -20 60  70 110 120 130
```

teremos criado uma tabela não-normalizada (-2) com 15 elementos, cada posição contendo o valor definido nos parâmetros.

Os valores "`600 1040 2250 2450 2750`" são as cinco frequências formantes necessárias pra gerar a vogal "a".

Em seguida "`0  -7  -9  -9 -20`" são as cinco amplitudes respectivas, e por fim "`60  70 110 120 130`" as cinco bandas.

Esses valores serão usados nas cinco instâncias de **fof** que precisaremos pra gerar, em nosso exemplo, a vogal "a":

```
  a1 fof iamp1, kfund, iform1, 0, iband1, iris, idur, idec, iolaps, ifna,\
            ifnb, itotdur
  a2 fof iamp2, kfund, iform2, 0, iband2, iris, idur, idec, iolaps, ifna,\
            ifnb, itotdur
  a3 fof iamp3,  kfund, iform3, 0, iband3, iris, idur, idec, iolaps, ifna,\
            ifnb, itotdur
  a4 fof iamp4, kfund, iform4, 0, iband4, iris, idur, idec, iolaps, ifna,\
            ifnb, itotdur
  a5 fof iamp4, kfund, iform5, 0, iband5, iris, idur, idec, iolaps, ifna,\
            ifnb, itotdur
  out 10000*(a1+a2+a3+a4+a5)
```

No código abaixo, vamos escolher a vogal pra cada nota, através do parâmetro **p5,** que variará entre 40 e 44, respectivos pra cada vogal.

```
<CsoundSynthesizer>

<CsOptions>

-o fof.wav

</CsOptions>

<CsInstruments>

sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

instr  1

  ; frequência formante, amplitude e banda para cada vogal
  iform1 table    0, p5
  iform2 table    1, p5
  iform3 table    2, p5
  iform4 table    3, p5
  iform5 table    4, p5

  iamp1  table    5,p5
  iamp2  table    6,p5
  iamp3  table    7,p5
  iamp4  table    8,p5
  iamp5  table    9,p5

  iband1 table    10,p5
  iband2 table    11,p5
  iband3 table    12,p5
  iband4 table    13,p5
  iband5 table    14,p5
```

```
  iamp1 =        ampdb(iamp1)
  iamp2 =        ampdb(iamp2)
  iamp3 =        ampdb(iamp3)
  iamp4 =        ampdb(iamp4)
  iamp5 =        ampdb(iamp5)

  ; vibrato para dar mais realismo ao vocal
  kvib lfo 10, 5, 1
  kfund = cpspch(p4)+kvib

  ; parâmetros comuns às cinco componentes da vogal
  ioct = 0
  iris = 0.003
  idur = 0.02
  idec = 0.007
  iolaps = 50
  ifna = 1
  ifnb = 3
  itotdur = p3

  ; finalmente a geração da vogal através de suas cinco componentes
  a1 fof iamp1, kfund, iform1, 0, iband1, iris, idur, idec, iolaps, ifna,\
            ifnb, itotdur
  a2 fof iamp2, kfund, iform2, 0, iband2, iris, idur, idec, iolaps, ifna,\
            ifnb, itotdur
  a3 fof iamp3,  kfund, iform3, 0, iband3, iris, idur, idec, iolaps, ifna,\
            ifnb, itotdur
  a4 fof iamp4, kfund, iform4, 0, iband4, iris, idur, idec, iolaps, ifna,\
            ifnb, itotdur
  a5 fof iamp4, kfund, iform5, 0, iband5, iris, idur, idec, iolaps, ifna,\
            ifnb, itotdur

  out 10000*(a1+a2+a3+a4+a5)

endin

</CsInstruments>

<CsScore>

f1 0 8192 10 1

; envelope de ataque e decaimento
f3 0 8193 7 0 8192 1000

; baixo "a"
f40 0 16 -2 600 1040 2250 2450 2750 0  -7  -9  -9 -20 60  70 110 120 130

; baixo "e"
f41 0 16 -2 400 1620 2400 2800 3100 0 -12  -9 -12 -18 40  80 100 120 120

; baixo "i"
f42 0 16 -2 250 1750 2600 3050 3340 0 -30 -16 -22 -28 60  90 100 120 120

; baixo "o"
f43 0 16 -2 400  750 2400 2600 2900 0 -11 -21 -20 -40 40  80 100 120 120

; baixo "u"
f44 0 16 -2 350  600 2400 2675 2950 0 -20 -32 -28 -36 40  80 100 120 120

i1    0     4     7.04   40
i1    4     3.5   7.05   41
i1    7.5   0.5   7.05   40
i1    8     3     7.07   42
i1    11    1     7.02   42
i1    12    4     7.04   40

e
```

```
</CsScore>

</CsoundSynthesizer>
```

Fig.6: *fof.csd*

Capítulo 8

Síntese Imitativa

Ao implementar um programa no Csound, não há limitação para algoritmos mais complexos simularem sintetizadores comerciais ou instrumentos eletro-acústicos. Nesse capítulo, o primeiro som característico de sintetizadores a ser emulado será o efeito de cordas, disponível em vários teclados comerciais. Usaremos o opcode especializado **oscbnk**, que contém embutido um número ilimitado de osciladores controlados por *lfo*'s e um equalizador paramétrico.

Cordas sintetizadas usando *oscbnk*

O opcode **oscbnk** foi criado por Istvan Varga especialmente para emular cordas sintetizadas, e para isso, nele é declarado um grande número de osciladores com frequências, amplitudes e fases variando levemente entre si, e tendo um equalizador paramétrico na saída para melhorar a qualidade do som, isso tudo em um único opcode. A sintaxe de **oscbnk** é extensa e podemos declará-la como:

```
ares oscbnk  kcps, kamd, kfmd, kpmd, iovrlap, iseed, kl1minf, \
               kl1maxf,kl2minf, kl2maxf, ilfomode, keqminf, keqmaxf, \
               keqminl, keqmaxl, keqminq, keqmaxq, ieqmode, kfn \
               [, il1fn] [, il2fn] [, ieqffn] [, ieqlfn] [, ieqqfn]\
               [, itabl] [, ioutfn]
```

Vamos primeiro entender como funciona os parâmetros mais importantes, e no caminho iremos entender como funciona **oscbnk**. Iremos então descrevê-los na ordem que mais facilita o entendimento.

. iovrlap nos diz quantos osciladores estarão sendo executados simultaneamente para a produção do som.

. kcps é a frequência fundamental dos osciladores.

. kfn é a forma de onda dos osciladores.

. kamd, kfmd, kpmd são respectivamente as variações máximas na amplitude, frequência e fase dos osciladores.

. ilfomode é o modo como os osciladores serão controlados. Cada oscilador tem dois **LFOs** (osciladores de baixa frequência) controlando sua frequência, fase, amplitude e equalização. De que maneira esse controle é feito é definido através do parâmetro **ilfomode,** que é a soma dos seguintes valores:

*128 para o LFO1 controlar a frequência
* 64 para ele controlar a amplitude
* 32 ele controla a fase
* 16 ele controla a equalização
* 8 para o LFO2 controlar a frequência
* 4 para ele controlar a amplitude
* 2 ele controla a fase
* 1 ele controla a equalização.

Muitos dos aspectos da saída de **oscbnk** variam aleatoriamente, mas dentro de uma faixa definida. Se definirmos em **ilfomode** que o primeiro LFO controlará a amplitude, a frequência e a fase dos osciladores, devemos definir os valores máximos dessa variação em **kamd, kfmd** e **kpmd**. O LFO, nesse sentido, age como um vibrato de baixa frequência alterando a amplitude, frequência e fase. Como essa variação é aleatória pra cada oscilador, cada oscilador terá uma variação diferente, mas sempre dentro dessa faixa. Se queremos que até num mesmo oscilador dois desses aspectos variem separadamente, devemos associar a um aspecto o LFO1 e ao outro o LFO2, através de **ilfomode**.

. iseed é a semente do gerador de números randômicos. Para o valor 0, ele tomará o tempo corrente.

.**kl1minf, kl1maxf, kl2minf, kl2maxf** são as frequências mínimas e máximas que o LFO1 e o LFO2 assumirão, variando dentro desses valores. Não confundir com **kfmd,** que é a variação da frequência do sinal. Aqui há a variação do tempo em que um novo ciclo de mudanças nos parâmetros ocorrerá, e é muitíssimo mais lento que as altas frequências de **kcps+-kfmd**.

. keqminf, keqmaxf, keqminl, keqmaxl, keqminq, keqmaxq são os mínimos e máximos para frequência, nível e "Q" do equalizador. Novamente os assuntos relacionados a equalização serão vistos em detalhe a seguir.

Os parâmetros a seguir são opcionais:

. il1fn, il2fn são as formas de onda do primeiro e do segundo LFO.

. ieqffn é a forma de onda da frequência do equalizador, onde a frequência será redimensionada para variar entre **keqminf** e **keqmaxf**, sendo então lida pelos LFOs**.**

. ieqlfn é a forma de onda para o nível do equalizador, que é então redimensionadas para valores entre **keqminl** e **keqmaxl**.

. ieqqfn é a forma de onda para o "Q" do equalizador (cujo significado será explicado logo abaixo após **ieqmode),** e essa forma de onda é então redimensionada para valores entre **keqminq** e **keqmaxq**.

. ieqmode nos diz qual será o modelo de equalização usado pelos equalizadores, seguindo o código:

* -1 desabilita o equalizador
* 0 a equalização é no modelo *peak*
* 1 para *low shelf*
* 2 para *high shelf*
* Os valores 3, 4, 5 repete a mesma sequência mas para valores interpolados.

Se faz necessário agora explicar o que é modelo de equalização *peak*, *low shelf* e *high* shelf. O padrão de equalização do equalizador paramétrico é atribuir a todas as frequências uma amplitude de 0 dB.

No modelo *peak*, todas as frequências tem uma resposta de 0 dB com exceção de um pico central na frequência que está sendo lida em **ieqffn** com a amplitude que está sendo lida em **ieqlfn**. Vale lembrar que a frequência **ieqffn** está devidamente desnormalizada para valores entre **keqminf** e **keqmaxf** e a amplitude **ieqlfn** para valores entre **keqminl** e **keqmaxl**.

A declividade desse pico é mapeada pelo Q em **ieqqfn,** que é lida pelo LFO e novamente desnormalizada para valores entre **keqminq** e **keqmaxq.** Quanto maior o valor de Q maior a declividade do pico central de frequência.

No modelo *low shelf*, novamente a resposta padrão do equalizador é de 0 dB para todas as frequências, mas abaixo da frequência lida naquele momento em **ieqffn** as frequências sofrem uma modificação de amplitude dada naquele momento por **ieqlfn,** e a declividade em **ieqffn** novamente é proporcional ao valor de Q em **ieqqfn**.

O modelo *high shelf* é similar ao *low shelf,* com exceção que agora modificamos o nível de amplitude de frequências *acima* de **ieqffn.**

. itabl é uma tabela opcional, para inicialização dos osciladores

. **iotfn** é uma tabela opcional, para escrita de valores de saída

Ambas não serão usadas

Nós veremos dois programas. O segundo é onde nós veremos realmente como programar **oscbnk.**

O primeiro programa, logo abaixo, é apenas para ser executado, sem a necessidade de ser lido, e servirá para entendermos visualmente os conceitos de frequência central, nível, Q e modelo de equalização. Para manipular graficamente esses parâmetros em tempo de execução, usaremos instruções gráficas de FLTK, que só serão vistas no próximo capítulo, e por isso mesmo não nos ateremos a elas agora. Execute-o e altere os knobs para ver como o espectro do equalizador se modifica:

```
<CsoundSynthesizer>
<CsOptions>
--displays -+rtaudio=null -o dac -b 1024 -B 2048
</CsOptions>
<CsInstruments>
sr      =  48000
ksmps   =  32
nchnls  =  1
0dbfs   =  1

        FLpanel "pareq", 360, 120

ih1             FLvalue " ", 50, 22, 20, 90
gkfco, ih1v     FLknob "Freq", 500, 10000, -1, 1, ih1, 60, 15, 10
                FLsetVal_i 5000, ih1v
ih2             FLvalue " ", 50, 22, 120, 90
gklvl, ih2v     FLknob "Level", 0.01, 3, -1, 1, ih2, 60, 115, 10
                FLsetVal_i 3, ih2v
ih3             FLvalue " ", 50, 22, 220, 90
gkQ, ih3v       FLknob "Q", 0.1, 3, -1, 1, ih3, 60, 215, 10
                FLsetVal_i 3, ih3v
ih4             FLvalue " ", 40, 22, 310, 90
gkmode, ih4v    FLroller "Mode", 0, 2, 1, 0, 2, ih4, 20, 60, 320, 10
                FLsetVal_i 2, ih4v

        FLpanelEnd

        FLrun

        opcode pulse, a, ki

        setksmps 1
kprd, iphs      xin
kcnt    init iphs
aout    =  0
kcnt    =  kcnt + 1
        if (kcnt < kprd) kgoto cont1
kcnt    =  0
aout    =  1
cont1:
        xout aout
```

```
        endop

        instr 1

gkmode_old      init -1
a1              pulse 4096, 2048
kc              port gkfco, 0.04
kv              port gklvl, 0.04
kq              port gkQ, 0.04
modeChange:
a2              pareq a1, kc, kv, kq, round(i(gkmode))
                if (gkmode == gkmode_old) kgoto noChange
                reinit modeChange
                rireturn
noChange:
gkmode_old      =  gkmode
                dispfft a2, 4096.01 / sr, 4096, 1

        endin

</CsInstruments>
<CsScore>
i 1 0 3600
e
</CsScore>
</CsoundSynthesizer>
```

Fig1.: *equalizer.csd*

A seguir temos um exemplo de como usar **oscbnk** para gerar sons semelhantes a cordas sintetizadas. Nele temos o trecho de código:

```
kcps = cpspch(p4)
kamd = 0.0
kfmd = 0.02 * kcps
kpmd = 0.0
iovrlap = 80
iseed = 200
kl1minf = 0.1
kl1maxf = 0.2
kl2minf = 0
kl2maxf = 0
ilfomode = 144
keqminf = 0.0
keqmaxf = 6000
keqminl = 0.0
keqmaxl = 0.0
keqminq = 1
keqmaxq = 1
ieqmode = 2
kfn = 1
il1fn = 3
il2fn = 0
ieqffn = 5
ieqlfn = 5
ieqqfn = 5
ares oscbnk  kcps, kamd, kfmd, kpmd, iovrlap, iseed, kl1minf, kl1maxf, \
                kl2minf, kl2maxf, ilfomode, keqminf, keqmaxf, keqminl, \
                keqmaxl, keqminq, keqmaxq, ieqmode, kfn, il1fn, il2fn, \
                ieqffn, ieqlfn, ieqqfn
```

A frequência da nota **kcps** é obtida a partir de **p4**, e a quantidade de osciladores simultâneos **iovrlap** é de 80 osciladores.

O controle dos LFOs é dado por **ilfomode** com o valor de 144, i.e., o LFO1 controla a frequência e a equalização dos osciladores, e nenhum LFO controla a amplitude e a fase, que tem variação 0 em **kamd** e **kpmd**.

A variação máxima de frequência **kfmd** dada pelo LFO1 é de 0.02***kcps,** 2% da frequência total.

A frequência do LFO1 em si varia entre **kl1minf** e **kl2maxf,** que é entre 0.1 e 0.2 Hertz. Como o LFO2 não controla nenhum aspecto dos osciladores, suas frequências mínimas e máximas são zero.

O método de equalização **ieqmode** é 2, *high shelf*, ou seja, as frequências baixas ficam inalteradas em 0 dB e as frequências altas são alteradas pelos parâmetros do equalizador.

As frequências mínima **keqminf** e máxima **keqmaxf** do equalizador são 0 e 6000 Hertz. Os níveis **keqminl** e **keqmaxl** são ambos zero e portanto atenuarão até anularem todas as frequências acima da máxima, e a declividade mínima **keqminq** e máxima **keqmaxq** é de 1. Com esses parâmetros o equalizador se torna quase um filtro passa-baixa, mantendo frequências abaixo da central audíveis em 0 dB, e acima tornando-as inaudíveis. O programa completo está a seguir:

```
<CsoundSynthesizer>

<CsOptions>

-o oscbnk.wav

</CsOptions>

<CsInstruments>

sr      =  48000
kr      =  750
ksmps   =  64
nchnls  =  1

ga01    init 0
ga02    init 0

i_      ftgen 1, 0, 16384, 7, 1, 16384, -1
i_      ftgen 3, 0, 4096, 7, 0, 512, 0.25, 512, 1, 512, 0.25, 512,  \
            0, 512, -0.25, 512, -1, 512, -0.25, 512, 0
i_      ftgen 5, 0, 1024, 5, 1, 512, 32, 512, 1

instr 1

  p3 = p3 + 0.4

  kcps = cpspch(p4)
  kamd = 0.0
  kfmd = 0.02 * kcps
  kpmd = 0.0
  iovrlap = 80
  iseed = 200
  kl1minf = 0.1
  kl1maxf = 0.2
  kl2minf = 0
  kl2maxf = 0
  ilfomode = 144
  keqminf = 0.0
  keqmaxf = 6000
  keqminl = 0.0
  keqmaxl = 0.0
  keqminq = 1
  keqmaxq = 1
```

```
  ieqmode = 2
  kfn = 1
  il1fn = 3
  il2fn = 0
  ieqffn = 5
  ieqlfn = 5
  ieqqfn = 5
  ares oscbnk  kcps, kamd, kfmd, kpmd, iovrlap, iseed, kl1minf, kl1maxf,
kl2minf, kl2maxf, ilfomode, \
                keqminf, keqmaxf, keqminl, keqmaxl, keqminq, keqmaxq,
ieqmode, \
                kfn, il1fn, il2fn, ieqffn, ieqlfn, ieqqfn
  out ares*1000
endin

</CsInstruments>

<CsScore>

t 0 60

i 1 0 4 6.05
i 1 0 4 8.00
i 1 0 4 8.05
i 1 0 4 8.09

e

</CsScore>

</CsoundSynthesizer>
```

Fig.2: *oscbnk.csd*

A f-table 1 é uma onda de dente de serra e é a forma dos osciladores de **oscbnk**, a f-table 3 é a onda do LFO1 e se assemelha a uma senóide, e a f-table 5 é a onda mapeada para as frequências centrais do equalizador.

O sintetizador Moog

Os sintetizadores analógicos criados por Robert Moog revolucionaram o mercado da década de 70, e se tornou referência para os sintetizadores atuais.

Existem atualmente três opcodes que emulam partes do sintetizador Mini-Moog, são eles **moog, moogvcf** e **moogladder**, sendo que os dois últimos emulam o cerne do Mini-Moog, que é o filtro ladder.

Veremos mais especificamente o **moogladder**, criado por Victor Lazzarini. A sintaxe de **moogladder** que usaremos é:

```
asig moogladder ain, kcf, kres
```

Temos em **ain** o sinal de entrada, em **kcf** a frequência de corte, e em **kres** o índice de ressonância. O filtro ladder do Mini-Moog é um tipo de filtro passa-baixa ressonante, isto é, acima de uma determinada frequência de corte o sinal é atenuado. Entretanto ao invés de apenas agir como um filtro passa-baixa comum, a ressonância causa também o efeito de *corner peaking*, isto é, as frequências próximas da frequência de corte são extremamente acentuadas, como um pico para em seguida as sequências superiores serem drasticamente atenuadas. Isso faz com que se acentue o espectro mais agudo das frequências imediatamente próximas a frequência de corte, e acima dela as frequências chegem próximo de se anular, como uma escalada íngreme e logo depois um vale.

Em nosso exemplo iremos variar a frequência de corte **kcf** de **moogladder**, e você poderá apreciar o efeito que ela causa. Usaremos dois instrumentos, no primeiro a onda de entrada é uma onda dente-de-serra e a frequência de corte aumenta ao longo do tempo, no segundo a entrada é uma senóide e a frequência de corte começa alta para ir diminuindo.

Apesar de variarmos completamente os sinais de entrada, a ação de **moogladder** é drástica no sentido que diminui as diferenças das ondas que entram. Se usássemos um sample de vocal a saída ainda seria semelhante à que obtemos. Na verdade, o que influi principalmente é quais frequências de corte usaremos.

No código abaixo usaremos como gerador de sinal do primeiro instrumento, o oscilador **vco**, visto no início do Capítulo 5.

```
<CsoundSynthesizer>

<CsOptions>

-o moogladder.wav

</CsOptions>

<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

instr 1
  iamp = 30000
  icps = cpspch(p4)
  isaw = 1

  asig vco iamp, icps, isaw

  kcf linseg icps, p3/2, icps*5, p3/2, icps

  kres init 1

  a1 moogladder asig, kcf, kres

  out a1
endin

instr 2
  iamp = 30000
  icps = cpspch(p4)
  ifn = 1

  kenv linseg 0, p3*.2, 1, p3*.8, 1

  asig oscil iamp, icps, ifn

  kcf line icps*10, p3, icps

  kres init 1

  a1 moogladder asig, kcf, kres

  out a1*kenv
endin

</CsInstruments>

<CsScore>
```

```
f1 0 8192 10 1

i1 0   20 8.04
i2 10  10 8.00
i2 10  10 7.09

e

</CsScore>

</CsoundSynthesizer>
```

Fig.3: *moogladder.csd*

Capítulo 9
Controle Gráfico

Além da maneira tradicional de se controlar os instrumentos através do score, podemos controlá-los graficamente e em tempo real através de uma interface gráfica, usando o **FLTK** (Fast Light Tool Kit) implementado por Gabriel Maldonado.

Existem três classes de componentes gráficos no Csound:

* os *containers*, como painéis e grupos
* os *valuators*, como sliders, knobs e outros controles gráficos
* e os componentes que não se enquadram nos dois anteriores, como botões e caixas de texto para monitoração.

Os containers são caixas visuais onde se pode organizar componentes dentro dele, e os valuators são os controles em si, assim podemos construir uma interface com controles organizados por painéis.

O *container* FLpanel

Veremos apenas os componentes gráficos mais usados, já que os outros são definidos de maneira semelhante. Como container usaremos o principal deles, o **FLpanel**, que define um painel e é o único capaz de criar uma janela a partir de sua declaração. Podemos encadear painéis dentro de painéis de uma maneira hierarquizada, através de sua sintaxe:

```
FLpanel "label", iwidth, iheight [, ix] [, iy] [, iborder]
```

Em **label** diremos o título que o painel terá. Essa informação só é útil se este **Flpanel** fôr o criador da janela do programa, e neste caso **label** será o título da janela.

Em **iwidth, iheight, ix** e **iy** diremos o valor em pixels da largura, altura e coordenadas x,y do painel.

Em **iborder** teremos o tipo de borda, variando entre 0 e 7. No próximo exemplo mostraremos todos os tipos de borda vendo-as na janela e com o valor de **iborder** comentado.

Abaixo da declaração de **Flpanel** virão as declarações dos outros componentes que estarão dentro do painel, e para terminar a seção desse painel o faremos através de **FlpanelEnd.** A forma que essas declarações assumirá será então a seguinte:

```
FLpanel "label", iwidth, iheight, ix, iy, iborder
        <declarações dos componentes que ficarão dentro do painel>
FlpanelEnd
```

Os *valuators* Flknob, Flslider e Flroller

Os controles que usaremos dentro do painel serão **Flknob, Flslider** e **Flroller.** A sintaxe deles é praticamente idêntica, respectivamente:

```
kout, ihandle FLknob "label", imin, imax, iexp, itype, idisp, iwidth,\
                      ix, iy
```

```
kout, ihandle FLslider "label",imin,imax, iexp, itype, idisp, iwidth,\
                       iheight, ix, iy

kout, ihandle FLroller "label",imin,imax, istep, iexp, itype, idisp, \
                       iwidth, iheight, ix, iy
```

Primeiro, os parâmetros comuns a todos eles.

Em **imin** e **imax** determinamos os valores mínimo e máximo que o controle assumirá.

Em **iexp** podemos definir essa variação como linear, para **iexp =** 0, ou exponencial, para **iexp** = -1.

Em **itype** definimos o tipo de visual que o controle assumirá, e veremos todos eles comentados e visualizados no segundo exemplo adiante.

Em **idisp** dizemos qual o controle do tipo **Flvalue** que estará encarregado de mostrar quais valores, entre **imin** e **imax,** estão sendo assumidos pelo controle.

E em **iwidth, iheight, ix** e **iy** dizemos as dimensões do controle e suas coordenadas x e y.

Os valores de x e y do controle não são absolutos, mas referentes as coordenadas dentro do container em que eles estão. Assim coordenadas x e y sendo (0,0) dirá apenas que eles estão no canto esquerdo superior do painel, mesmo que este painel esteja no fim da janela do programa.

Os valores de saída dos controles são **kout**, que contém o valor do controle em cada momento, e **ihandle,** que é uma referência fixa ao controle, de modo que ele pode ser acessado e modificado por outros comandos mais tarde.

As únicas diferenças entre **Flknob, Flslider** e **Flroller** é que **Flknob** precisa de um parâmetro a menos, não precisando do parâmetro **iheight,** já que suas dimensões são determinadas unicamente pelo quadrado com **iwidth** pixels de altura e largura, e **Flroller** precisa de um parâmetro a mais, **istep,** que diz em quanto o valor do controle é modificado a cada clique do mouse.

Exibindo os valores na tela

Os valores dos controles em cada momento são armazenados em **kout,** e podem ser exibidos visualmente através de controles do tipo **Flvalue**, cuja sintaxe é:

```
idisp FLvalue "label", idwidth, idheight, idx, idy
```

Os controles são ligados ao componente **Flvalue** através de seu parâmetro **idisp.** Note que o mesmo parâmetro de saída **idisp** em **Flvalue** é o mesmo parâmetro de entrada **idisp** em **Flknob,** por exemplo.

Podemos definir um valor inicial para cada controle através de **FlsetVal_i,** que deve ser chamada de dentro de um instrumento e cuja sintaxe é:

```
FLsetVal_i kvalue, ihandle
```

Onde definimos aqui o valor **kvalue** para o controle referenciado por **ihandle**. Um código simples contendo **Flpanel, Flknob, Flvalue** e **FlsetVal_i** seria:

```
<CsoundSynthesizer>

<CsOptions>
</CsOptions>
```

```
<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

;;;;;;;;;;;;;;;;;;;;;;;;;
;
; FLpanel
;
;;;;;;;;;;;;;;;;;;;;;;;;;
iwidth = 110
iheight = 110
ix = 0
iy = 0
iborder = 1
FLpanel "", iwidth, iheight, ix, iy, iborder
;;;;;;;;;;;;;;;;;;;;;;;;;
; FLvalue
;;;;;;;;;;;;;;;;;;;;;;;;;
  iwidth = 90
  iheight = 30
  ix = 10
  iy = 70
  idisp FLvalue "", iwidth, iheight, ix, iy
;;;;;;;;;;;;;;;;;;;;;;;;;
; FLknob
;;;;;;;;;;;;;;;;;;;;;;;;;
  imin = 0
  imax = 1
  iexp = 0
  itype = 1
  iwidth = 40
  ix = 10
  iy = 10
  gkout, gihandle FLknob "Knob1", imin, imax, iexp, itype, idisp, \
                    iwidth, ix, iy
FLpanelEnd

FLrun


;;;;;;;;;;;;;;;;;;;;;;;;;
;
; FLsetVal_i
;
;;;;;;;;;;;;;;;;;;;;;;;;;
instr 1
  FLsetVal_i 0.5, gihandle
endin

</CsInstruments>

<CsScore>

f0 100
i1 0 1

</CsScore>

</CsoundSynthesizer>
```

Fig. 1: *fltk.csd*

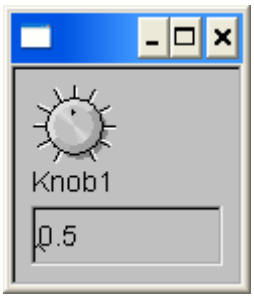


Fig.2: tela do programa *fltk.csd*

Depois das declarações dos componentes gráficos, chamamos **Flrun,** que executa todos os componentes e mostra a interface gráfica.

Diferenças visuais em *itype* e *iborder*

Em nosso próximo exemplo poderemos ver as diferenças visuais para cada valor de **itype** declarado em **Flknob, Flslider** e **Flroller.**

Em **Flknob,** o valor do parâmetro **itype** pode ser:

*** itype** = 1, para ter uma aparência 3-D
*** itype** = 2, para se ter um knob estilo fatia ou pie-like
*** itype**=3 para se ter um knob estilo relógio
*** itype**=4 nos dá um knob plano.

Fig.3: os valores de **itype** em **Flknob**

Em **Flslider**, o valor do parâmetro **itype** pode ser:

*** itype** = 1, para se ter um slider horizontal de preenchimento
* **itype**=2 para um slider vertical de preenchimento
* **itype**=3 nos dá um slider horizontal de rolagem
* **itype=4** um slider vertical de rolagem
* **itype**=5 um slider horizontal estilo mesa de som
* **itype**=6 um slider vertical de mesa de som.
Os tipos 7 e 8 estão documentados mas ainda não fôram implementados.

Fig.4: os valores de **itype** em **Flslider**

Em **Flroller,** o valor do parâmetro **itype** pode ser:

*** itype**=1 para rolagem horizontal

*** itype=**2 para rolagem vertical.

Fig.5: os valores de **itype** em **Flroller**

Em **Flpanel** temos o parâmetro **iborder** que nos dá o estilo de borda do painel, sendo:

*** iborder=**0 para nenhuma borda
*** iborder=**1 para o efeito de painel rebaixado
*** iborder=**2 para um painel levantado
*** iborder=**3 para uma borda gravada
*** iborder=**4 para uma borda levantada
*** iborder=**5 uma borda apenas de linha
*** iborder=**6 para o efeito de painel rebaixado com borda fina
*** iborder=**7 para um painel levantado de borda fina.


Fig.6: os valores de **iborder** em **Flpanel**

A seguir temos o programa completo que gerou todos os tipos de gráficos nesses componentes. Não se assuste com a quantidade de linhas do programa, de fato a maior parte é de especificação de coordenadas para cada componente. Com o tempo você notará que o mais trabalhoso em interfaces gráficas no Csound é especificar essas coordenadas de modo que os controles se alinhem bem na tela.

No programa os parâmetros com prefixo **ip** são referentes a parâmetros dos painéis, prefixos **id** são referentes a parâmetros dos componentes **Flvalue**, e os sem prefixo são referentes aos *valuators*. Logo depois do programa temos a imagem gerada por ele na tela.

```
<CsoundSynthesizer>

<CsOptions>
</CsOptions>

<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

ipwidth = 90
ipheight = 95
idwidth = 60
idheight = 30
idx = 10
idy = 60
```

```
imin = 0
imax = 1
iexp = 0 ; saída é linear, para -1 seria exponencial
iborder = 1 ; down box border
iwidth = 40
iheight = 40
ix = 20
iy = 5

FLpanel "Controle Gráfico", 700, 400
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Knobs
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

ipx = 5
ipy = 5
FLpanel "", ipwidth, ipheight, ipx, ipy, iborder
  idisp1 FLvalue "", idwidth, idheight, idx, idy
  itype = 1 ; um knob 3-D
  gk1, gi1 FLknob "itype=1", imin, imax, iexp, itype, idisp1, iwidth, ix, iy
FLpanelEnd

ipx = 100
FLpanel "", ipwidth, ipheight, ipx, ipy, iborder
  idisp2 FLvalue "", idwidth, idheight, idx, idy
  itype = 2 ; um knob pie-like
  gk2, gi2 FLknob "itype=2", imin, imax, iexp, itype, idisp2, iwidth, ix, iy
FLpanelEnd

ipx = 200
FLpanel "", ipwidth, ipheight, ipx, ipy, iborder
  idisp3 FLvalue "", idwidth, idheight, idx, idy
  itype = 3 ; um knob clock-like
  gk3, gi3 FLknob "itype=3", imin, imax, iexp, itype, idisp3, iwidth, ix, iy
FLpanelEnd

ipx = 300
FLpanel "", ipwidth, ipheight, ipx, ipy, iborder
  idisp4 FLvalue "", idwidth, idheight, idx, idy
  itype = 4 ; um knob flat
  gk4, gi4 FLknob "itype=4", imin, imax, iexp, itype, idisp4, iwidth, ix, iy
FLpanelEnd

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Sliders
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

iwidth = 40
iheight = 20
ipx = 5
ipy = 100
FLpanel "", ipwidth, ipheight, ipx, ipy, iborder
  idisp5 FLvalue "", idwidth, idheight, idx, idy
  itype = 1 ; horizontal fill slider
  gk5, gi5 FLslider "itype=1", imin, imax, iexp, itype, idisp5, iwidth,
iheight, ix, iy
FLpanelEnd

iwidth = 20
iheight = 40
ipx = 100
FLpanel "", ipwidth, ipheight, ipx, ipy, iborder
  idisp6 FLvalue "", idwidth, idheight, idx, idy
  itype = 2 ; vertical fill slider
```

```
  gk6, gi6 FLslider "itype=2", imin, imax, iexp, itype, idisp6, iwidth,
iheight, ix, iy
FLpanelEnd

iwidth = 40
iheight = 20
ipx = 200
FLpanel "", ipwidth, ipheight, ipx, ipy, iborder
  idisp7 FLvalue "", idwidth, idheight, idx, idy
  itype = 3 ; horizontal engraved slider
  gk7, gi7 FLslider "itype=3", imin, imax, iexp, itype, idisp7, iwidth,
iheight, ix, iy
FLpanelEnd

iwidth = 20
iheight = 40
ipx = 300
FLpanel "", ipwidth, ipheight, ipx, ipy, iborder
  idisp8 FLvalue "", idwidth, idheight, idx, idy
  itype = 4 ; vertical engraved slider
  gk8, gi8 FLslider "itype=4", imin, imax, iexp, itype, idisp8, iwidth,
iheight, ix, iy
FLpanelEnd

iwidth = 40
iheight = 20
ipx = 400
FLpanel "", ipwidth, ipheight, ipx, ipy, iborder
  idisp9 FLvalue "", idwidth, idheight, idx, idy
  itype = 5 ; horizontal nice slider
  gk9, gi9 FLslider "itype=5", imin, imax, iexp, itype, idisp9, iwidth,
iheight, ix, iy
FLpanelEnd

iwidth = 20
iheight = 40
ipx = 500
FLpanel "", ipwidth, ipheight, ipx, ipy, iborder
  idisp10 FLvalue "", idwidth, idheight, idx, idy
  itype = 6 ; vertical nice slider
  gk10, gi10 FLslider "itype=6", imin, imax, iexp, itype, idisp10, iwidth,
iheight, ix, iy
FLpanelEnd

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Rollers
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

istep = 0.1
iwidth = 40
iheight = 20
ipx = 5
ipy = 200
FLpanel "", ipwidth, ipheight, ipx, ipy, iborder
  idisp11 FLvalue "", idwidth, idheight, idx, idy
  itype = 1 ; horizontal fill slider
  gk11, gi11 FLroller "itype=1", imin, imax, istep, iexp, itype, idisp11,
iwidth, iheight, ix, iy
FLpanelEnd

iwidth = 20
iheight = 40
ipx = 100
FLpanel "", ipwidth, ipheight, ipx, ipy, iborder
  idisp12 FLvalue "", idwidth, idheight, idx, idy
  itype = 2 ; vertical fill slider
```

```
  gk12, gi12 FLroller "itype=2", imin, imax, istep, iexp, itype, idisp12,
iwidth, iheight, ix, iy
FLpanelEnd

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Panels
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

ipy = 300
ipx = 5
iborder = 1 ; down box border
FLpanel "", ipwidth, ipheight, ipx, ipy, iborder
FLpanelEnd

ipx = 100
iborder = 2 ; up box border
FLpanel "", ipwidth, ipheight, ipx, ipy, iborder
FLpanelEnd

ipx = 200
iborder = 3 ; engraved border
FLpanel "", ipwidth, ipheight, ipx, ipy, iborder
FLpanelEnd

ipx = 300
iborder = 4 ; embossed border
FLpanel "", ipwidth, ipheight, ipx, ipy, iborder
FLpanelEnd

ipx = 400
iborder = 5 ; black line border
FLpanel "", ipwidth, ipheight, ipx, ipy, iborder
FLpanelEnd

ipx = 500
iborder = 6 ; thin down border
FLpanel "", ipwidth, ipheight, ipx, ipy, iborder
FLpanelEnd

ipx = 600
iborder = 7 ; thin up border
FLpanel "", ipwidth, ipheight, ipx, ipy, iborder
FLpanelEnd

FLpanelEnd

FLrun

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Inicialização dos controles
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
instr 1
  igain init 0.5
  FLsetVal_i igain, gi1
  FLsetVal_i igain, gi2
  FLsetVal_i igain, gi3
  FLsetVal_i igain, gi4
  FLsetVal_i igain, gi5
  FLsetVal_i igain, gi6
  FLsetVal_i igain, gi7
  FLsetVal_i igain, gi8
  FLsetVal_i igain, gi9
  FLsetVal_i igain, gi10
  FLsetVal_i igain, gi11
```

```
  FLsetVal_i igain, gi12
endin

</CsInstruments>

<CsScore>

i1  0 100

</CsScore>

</CsoundSynthesizer>
```

Fig.7: *fltk2.csd*

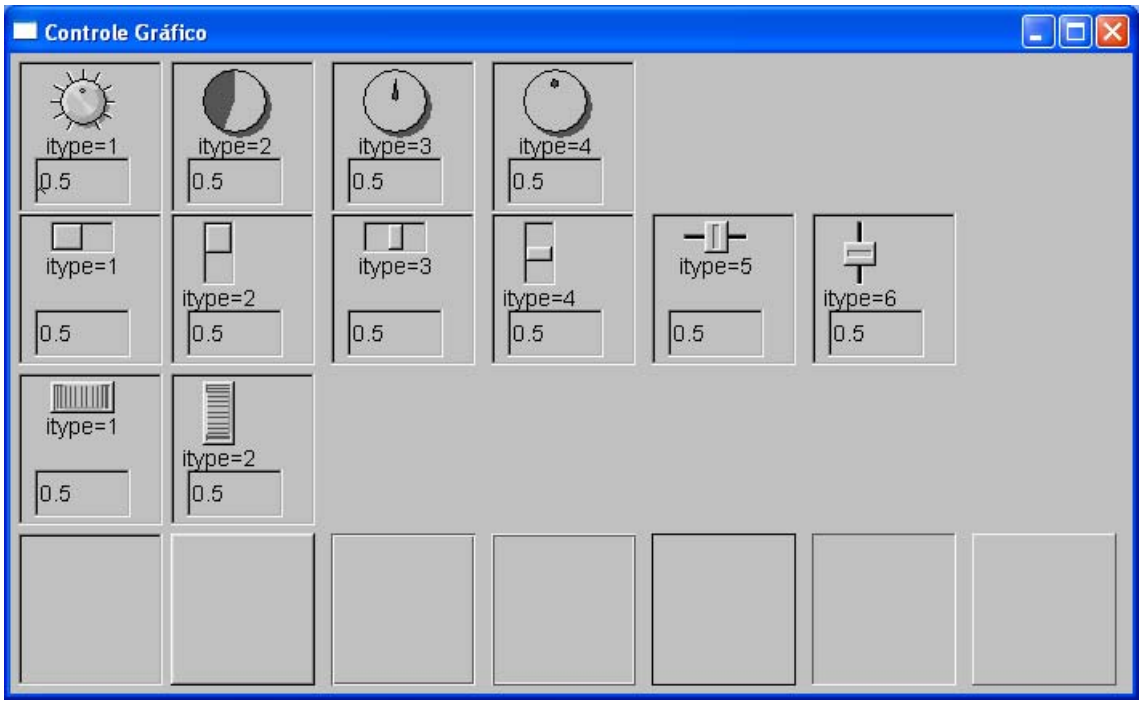


Fig.8: a tela do programa *fltk2.csd*

Controlando vários efeitos usando FLTK

No arquivo *zak2.csd* do Capítulo 6 implementamos uma cadeia de efeitos com parâmetros do score. Queremos agora controlá-los graficamente por controles **Flslider.**

Basicamente teremos que acrescentar no início da seção orchestra os vários controles FLTK e em cada instrumento definir o valor inicial do controle e associar o controle a um parâmetro do instrumento. Para o intrumento de distorção, por exemplo, teremos no cabeçalho os componentes que o controlarão, associando duas variáveis **gkpregain** e **gkpostgain** aos valores de dois controles **Flslider**.

```
FLpanel "Distorção", iwidth, iheight, ix, iy, iborder
  iwidth = 30
  iheight = 30
  ix = 20
  iy = 240
  idisp1 FLvalue "", iwidth, iheight, ix, iy

  imin = 1
```

```
  imax = 2
  iexp = 0 ; saída é linear, para -1 seria exponencial
  itype = 6
  iwidth = 30
  iheight = 200
  ix = 20
  iy = 20
  gkpregain, gipregain FLslider "Pre Gain", imin, imax, iexp, itype,\
                                    idisp1, iwidth, iheight, ix, iy

  iwidth = 30
  iheight = 30
  ix = 100
  iy = 240
  idisp2 FLvalue "", iwidth, iheight, ix, iy

  imin = 1
  imax = 2
  iexp = 0 ; saída é linear, para -1 seria exponencial
  itype = 6
  iwidth = 30
  iheight = 200
  ix = 100
  iy = 20
  gkpostgain, gipostgain FLslider "Post Gain", imin, imax, iexp,\
                                itype, idisp2, iwidth, iheight, ix, iy
FLpanelEnd
```

Esse trecho de código cria a interface gráfica para o efeito de distorção. Em seguida, dentro do instrumento, definimos os valores iniciais dos dois controles **Flslider** e atribuímos os valores assumidos pelos dois controles para duas variáveis locais do instrumento, como vemos a seguir:

```
instr 95
  iinch init p4
  ioutch init p5

  FLsetVal_i 2, gipregain
  kpregain = gkpregain

  FLsetVal_i 1, gipostgain
  kpostgain = gkpostgain

  ishape1 init 0
  ishape2 init 0

  asign init 0

  asig zar iinch
  adist distort1 asig, kpregain, kpostgain, ishape1, ishape2

  aold = asign         ; Salva o último sinal
  asign = asig/30000   ; Normaliza o novo sinal

; Faz um atraso baseado na declividade
  atemp delayr  .1
```

```
  aout deltapi (2-asign)/1500 + (asign-aold)/300
  delayw  adist

  zaw aout*gibalance, ioutch
endin
```

O código que veio no final do instrumento é exatamente o mesmo implementado antes no Capítulo 6, tendo apenas seu início adaptado tomar valores da interface gráfica.

Criaremos a mesma cadeia de efeitos daquele capítulo, fazendo a ligação entre eles através da família **zak** de opcodes, encadeando sucessivamente os efeitos de distorção, flanger, reverb e ganho de saída, e controlando seus parâmetros de performance através de controles **Flslider**.

Os parâmetros de inicialização serão fixados como constantes dentro do intrumento, e os de performance associados aos valores dos controles. Poderemos ver essa diferença na implementação do controle flanger no trecho abaixo, onde **iinitdel** e **ienddel** são fixados e **kfeedback** é associado ao valor do controle gráfico:

```
instr 96
  iinch init p4
  ioutch init p5

  iinitdel = 0.01
  ienddel = 0.07

  FLsetVal_i 0.7, gifeedback
  kfeedback = gkfeedback

  asig zar iinch
  adel line iinitdel, p3, ienddel
  aout flanger asig, adel, kfeedback
  zaw aout*gibalance, ioutch
endin
```

A seguir temos o programa completo, que na verdade é uma adaptação do programa *zak2.csd*, agora com componentes gráficos para todos os parâmetros de performance:

```
<CsoundSynthesizer>

<CsOptions>
</CsOptions>

<CsInstruments>

sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

iwidth = 500
iheight = 300
FLpanel "Controle Gráfico", iwidth, iheight

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;
; Controle da Distorção
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
iwidth = 180
iheight = 280
ix = 10
iy = 10
iborder = 1 ; down box border
FLpanel "Distorção", iwidth, iheight, ix, iy, iborder
  iwidth = 30
  iheight = 30
  ix = 20
  iy = 240
  idisp1 FLvalue "", iwidth, iheight, ix, iy

  imin = 1
  imax = 2
  iexp = 0 ; saída é linear, para -1 seria exponencial
  itype = 6
  iwidth = 30
  iheight = 200
  ix = 20
  iy = 20
  gkpregain, gipregain FLslider "Pre Gain", imin, imax, iexp, itype, idisp1,
iwidth, iheight, ix, iy

  iwidth = 30
  iheight = 30
  ix = 100
  iy = 240
  idisp2 FLvalue "", iwidth, iheight, ix, iy

  imin = 1
  imax = 2
  iexp = 0 ; saída é linear, para -1 seria exponencial
  itype = 6
  iwidth = 30
  iheight = 200
  ix = 100
  iy = 20
  gkpostgain, gipostgain FLslider "Post Gain", imin, imax, iexp, itype,
idisp2, iwidth, iheight, ix, iy
FLpanelEnd


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Controle do Flanger
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
iwidth = 90
iheight = 280
ix = 200
iy = 10
iborder = 1 ; down box border
FLpanel "Flanger", iwidth, iheight, ix, iy, iborder
  iwidth = 30
  iheight = 30
  ix = 20
  iy = 240
  idisp3 FLvalue "", iwidth, iheight, ix, iy

  imin = 0.5
  imax = 0.8
  iexp = 0 ; saída é linear, para -1 seria exponencial
  itype = 6
  iwidth = 30
```

```
  iheight = 200
  ix = 20
  iy = 20
  gkfeedback, gifeedback FLslider "Feedback", imin, imax, iexp, itype, idisp3,
iwidth, iheight, ix, iy
FLpanelEnd

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Controle do Reverb
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
iwidth = 90
iheight = 280
ix = 300
iy = 10
iborder = 1 ; down box border
FLpanel "Reverb", iwidth, iheight, ix, iy, iborder
  iwidth = 30
  iheight = 30
  ix = 20
  iy = 240
  idisp4 FLvalue "", iwidth, iheight, ix, iy

  imin = 0
  imax = 6
  iexp = 0 ; saída é linear, para -1 seria exponencial
  itype = 6
  iwidth = 30
  iheight = 200
  ix = 20
  iy = 20
  gkrvt, girvt FLslider "Reverb", imin, imax, iexp, itype, idisp4, iwidth,
iheight, ix, iy
FLpanelEnd

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Controle do Ganho da Saída
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
iwidth = 90
iheight = 280
ix = 400
iy = 10
iborder = 1 ; down box border
FLpanel "Gain", iwidth, iheight, ix, iy, iborder
  iwidth = 30
  iheight = 30
  ix = 20
  iy = 240
  idisp5 FLvalue "", iwidth, iheight, ix, iy

  imin = 0
  imax = 1
  iexp = 0 ; saída é linear, para -1 seria exponencial
  itype = 6
  iwidth = 30
  iheight = 200
  ix = 20
  iy = 20
  gkgain, gigain FLslider "Gain", imin, imax, iexp, itype, idisp5, iwidth,
iheight, ix, iy
FLpanelEnd

FLrun

zakinit 5, 1
```

```
gibalance = 0.4

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Entrada
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

instr 1
  iinch init p4
  ioutch init p5

  asig soundin "female.aif"
  zaw asig*gibalance, ioutch
endin

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Distorção
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
instr 96
  iinch init p4
  ioutch init p5

  FLsetVal_i 2, gipregain
  kpregain = gkpregain

  FLsetVal_i 1, gipostgain
  kpostgain = gkpostgain

  ishape1 init 0
  ishape2 init 0

  asign init 0

  asig zar iinch
  adist distort1 asig, kpregain, kpostgain, ishape1, ishape2

  aold = asign         ; Salva o último sinal
  asign = asig/30000   ; Normaliza o novo sinal

  atemp delayr  .1                  ; Faz um atraso baseado na declividade
  aout deltapi (2-asign)/1500 + (asign-aold)/300
  delayw  adist

  zaw aout*gibalance, ioutch
endin

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Flanger
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
instr 97
  iinch init p4
  ioutch init p5

  iinitdel = 0.01
  ienddel = 0.07

  FLsetVal_i 0.7, gifeedback
  kfeedback = gkfeedback

  asig zar iinch
  adel line iinitdel, p3, ienddel
  aout flanger asig, adel, kfeedback
```

```
  zaw aout*gibalance, ioutch
endin

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Reverb
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
instr 98
  iinch init p4
  ioutch init p5

  FLsetVal_i 1, girvt
  krvt = gkrvt

  asig zar iinch
  aout reverb asig, krvt
  zaw aout*gibalance, ioutch
endin

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Saída
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
instr 99
  iinch init p4
  ioutch init p5

  FLsetVal_i 0.8, gigain
  kgain = gkgain

  asig zar iinch
  zacl
  out asig*kgain
endin

</CsInstruments>

<CsScore>

; Entrada
i1  0 6 1 1
i1  6 6 1 1
i1  12 6 1 1
i1  18 6 1 1
i1  24 6 1 1

; Distorção
i96 0 30 1 2

; Flanger
i97 0 30 2 3

; Reverb
i98 0 30 3 4

; Saída
i99 0 30 4 1

</CsScore>

</CsoundSynthesizer>
```

Fig.9: *fltk3.csd*

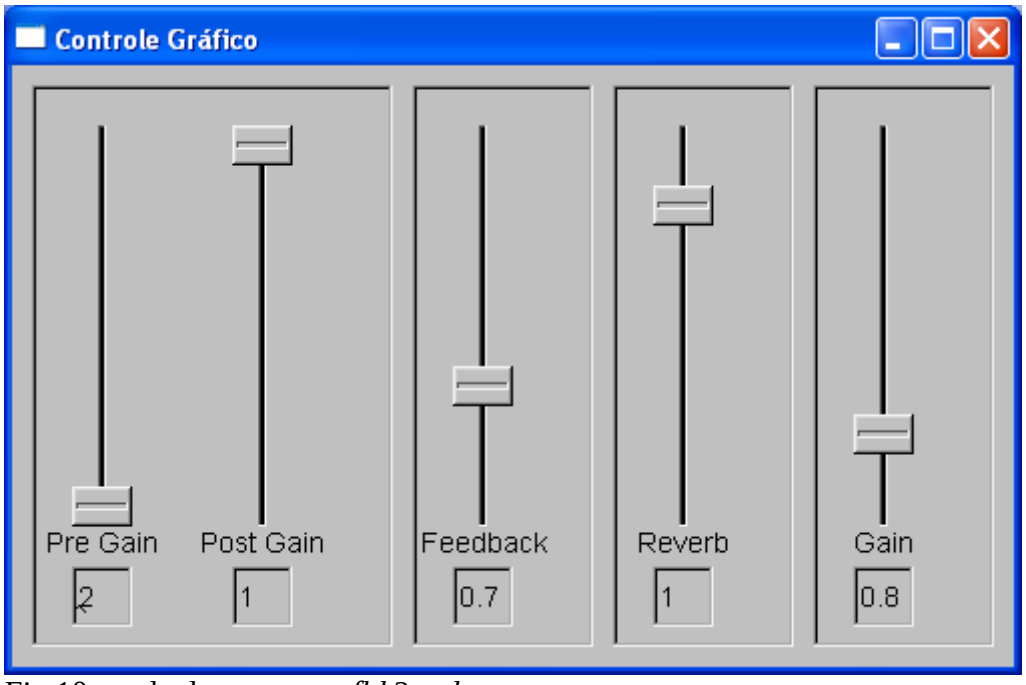


Fig.10: a tela do programa *fltk3.csd*

Capítulo 10
MIDI e Execução em Tempo Real

Existe uma gama de opcodes que lida com eventos **MIDI** no Csound, entretanto existe alguns que são usados na maioria das vezes. Tipicamente, queremos que uma nota tocada pelo controlador MIDI externo seja enviada para um instrumento do Csound, para que seja processada e emitida em tempo real.

O método do Csound para lidar com eventos MIDI é simples e eficiente. Cada canal MIDI é associado a um instrumento da seção orchestra, e sempre que um evento ocorre neste canal, o instrumento é executado pelo tempo que a nota MIDI durar, com acesso a todas as informações dessa nota. Quando a nota pára de ser tocada pelo instrumento real, o instrumento do Csound deixa de ser executado.

Faremos um programa que a partir de um teclado, toque uma senóide na frequência da tecla pressionada. A primeira coisa a fazer é associar um canal MIDI usado pelo teclado a um instrumento da seção orchestra do CSound.

Isso pode ser feito através do opcode **massign**, ou *MIDI assign*, e a sintaxe que usaremos é:

```
massign ichnl, insnum
```

que associa um canal **ichnl** ao instrumento número **isnum.** A declaração de **massign** deve estar sempre no cabeçalho, antes da declaração do primeiro instrumento. Para associar o canal 1 ao instrumento número 1, teremos no programa:

```
massign 1, 1
```

Sempre que uma nota fôr tocada no canal 1 pelo controlador MIDI, nosso instrumento 1 será executado pelo tempo que essa nota durar.

Dentro do instrumento 1, precisamos obter qual nota está sendo tocada, e em que velocidade ela foi tocada. Para obter a frequência da nota tocada, usaremos o opcode **cpsmidi,** cuja sintaxe é:

```
icps cpsmidi
```

onde a nota que está sendo tocada pelo controlador externo, será armazenada em Hertz em **icps**. Em alguns casos, poderíamos não querer essa informação direto em valores de frequência, mas apenas o número MIDI da nota, e para isso usaríamos:

```
ival notnum
```

em que o número da nota MIDI, que vai de 0 a 127, seria armazenado em **ival**. Poderíamos usar esse número para dizer qual a f-table usaríamos naquela nota:

```
a1 oscil kamp, icps, ival
```

e cada nota teria um timbre diferente.

E para obter com que velocidade, ou força, a nota está sendo tocada pelo controlador externo, podemos usar o opcode **iveloc,** e sua sintaxe é:

```
ival veloc [ilow] [, ihigh]
```

Se omitirmos **ilow** e **ihigh**, **iveloc** armazenará em **ival** o valor MIDI da dinâmica, que varia de 0 a 127.

Se escrevermos os valores para **ilow** e **ihigh**, eles determinarão a faixa em que **veloc** escalonará a dinâmica. Se quisermos obter um valor pra amplitude entre 0 e 30000, escreveríamos:

```
iamp veloc 0, 30000
```

Existe ainda um último opcode que será útil em nosso instrumento. Quando o controlador pára de tocar a nota, o instrumento é bruscamente interrompido. Às vezes queremos que o instrumento tenha um tempo de *release***,** tal que depois que a nota não está sendo mais tocada pelo controlador, ela sofra um decaimento suave durante um tempo extra.

Esse tempo extra de release é dado ao instrumento através do opcode de envelope **linsegr:**

```
kres linsegr ia, idur1, ib [, idur2] [, ic] [...], irel, iz
```

Com exceção dos dois últimos parâmetros **irel** e **iz**, o restante é idêntico a **linseg.** O opcode traçará um segmento de reta entre os valores **ia** e **ib** com duração de **idur1** segundos, e assim sucessivamente para os segmentos seguintes.

O parâmetro imediatamente anterior a **irel**, que será o último valor que **kres** assumirá, será mantido durante todo o restante da nota, até que o controlador MIDI pare de tocá-la. Quando o controlador pára, então o instrumento recebe mais **irel** segundos, para decair até o valor de **iz**.

Vamos adaptar o primeiro exemplo desse livro, *seno.csd*, para ser tocado por um controlador MIDI em tempo real.

Uma complicação que surgiria nesse momento seria conectar fisicamente o controlador MIDI ao computador, isso se você possuir o equipamente necessário. Para que esse exemplo seja geral no sentido de funcionar em qualquer máquina, usaremos um controlador MIDI virtual, que é o *Virtual MIDI Keyboard*, criado por Steven Yi.

Ele exibe uma interface gráfica com um teclado de piano na tela, onde selecionamos canais MIDI e bancos, que pode ser tocada com o mouse, ou até diretamente pelo teclado do PC, utilizando no teclado *qwerty* as fileiras de teclas Z-M, S-J, Q-P e 2-0, seguindo o desenho de teclas brancas e pretas do piano.

O Virtual MIDI Keyboard é ativado na linha de comando acrescentando-se o flag:

```
-+rtmidi=virtual
```

Nosso código então ficaria:

```
<CsoundSynthesizer>

<CsOptions>


-+rtmidi=virtual

</CsOptions>
```

```
<CsInstruments>
; Inicializa as variáveis globais.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; associa o canal midi 1 ao instrumento 1
massign 1,1

; o instrumento 1 será executado enquanto o evento midi durar no canal 1
instr 1

  ; Velocidade da nota midi
  iamp iveloc 0, 30000

  ; Frequência da nota midi
  icps cpsmidi

  ; Número da f-table com a senóide
  ifn = 1

  ; envelope com ataque de 0.5 segundos,
  ; e que sustentará a amplitude até que a nota midi termine,
  ; havendo então um tempo extra de 1 segundo para que o instrumento
  ; silencie suavemente.
  kamp linsegr 0, 0.5, iamp, 1, 0

  ; Toca a nota midi com uma senóide
  a1 oscil kamp, icps, ifn
  out a1

endin

</CsInstruments>

<CsScore>

; Tabela #1: uma simples onda de seno usando GEN10.
f 1 0 16384 10 1

; Toca o instrumento #1 por 2 segundos, começando em 0 segundos
f 0 120
e

</CsScore>

</CsoundSynthesizer>
```

Fig.3: *midi.csd*, a versão MIDI de *seno.csd*.

Na seção score usamos uma f-table vazia, com índice 0, para ser carregada após 120 segundos. Ela é necessária apenas para fazer com que possamos tocar por 120 segundos, enviando eventos MIDI para o instrumento, e então o programa termine.

Como no restante desse livro, tocamos apenas na superfície do suporte MIDI, esperando que sirva para que o usuário aprofunde-se nos meandros da programação nas áreas descritas neste livro, e em outras que estarão por vir.