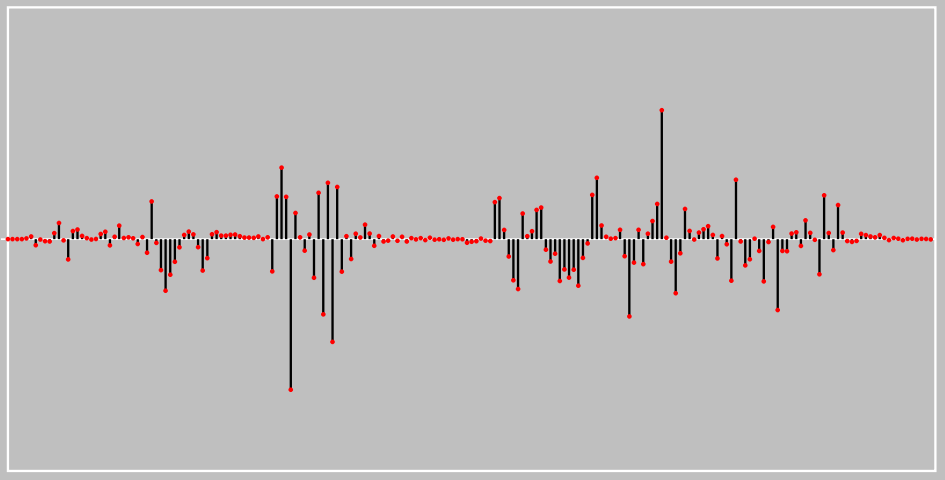
APOGEO education

Andrea Valle

Introduzione a SuperCollider



MAGGIOLI EDITORE

Introduzione a SuperCollider

Andrea Valle

Originariamente sviluppato da James McCartney e ora diventato progetto open source, SuperCollider è un pacchetto software per la sintesi e il controllo dell'audio in tempo reale. Attualmente rappresenta lo stato dell'arte nell'ambito della programmazione audio: non c'è altro software disponibile che sia insieme così potente, efficiente, flessibile. Eppure, c'è indubbiamente un motivo per cui spesso SC è guardato con sospetto o con timore: SC non è esattamente "intuitivo". In primo luogo, l'interfaccia che offre all'utente è di tipo testuale. Propriamente, SuperCollider si scrive: quanto basta per spaventare l'utente usuale che nell'uso del calcolatore è abituato a vedere più che a leggere. In secondo luogo, come tutti i software dedicati all'audio avanzato, SuperCollider richiede competenze relative alla sintesi del segnale audio e alla composizione musicale (nel senso più lato, aperto e sperimentale del termine), ma vi aggiunge competenze più strettamente informatiche, ad esempio, oltre alla teoria dei linguaggi di programmazione, il riferimento al concetto di architettura client/server. Tuttavia, quelli che appaiono come ostacoli sono in realtà il punto di forza di SC: la sua versatilità per le applicazioni software avanzate, la generalità in termini di relazioni con altri aspetti di modellizzazione informatica, l'espressività in termini di rappresentazioni simboliche agevolmente ottenibili.
Questo manuale si propone di fornire una breve panoramica ed insieme un'introduzione a SuperCollider in italiano e di introdurre alcuni aspetti fondamentali relativi a quell'ambito che viene usualmente indicato, con una definizione a volte imprecisa e a volte impropria, computer music.

Andrea Valle è ricercatore di cinema, fotografia e televisione all'Università di Torino, ed è attivo come musicista e compositore. Tra gli altri suoi lavori, con V. Lombardo ha scritto *Audio e multimedia* (4 ed., Milano, Maggioli 2014).

Introduzione a SuperCollider
Andrea Valle
Apogeo Education
Maggioli editore
2015
ISBN: 978-88-916-1068-3

Riferimento BibTeX:

```
@book{ValleIntroduzioneSC2015,
       Author = {Andrea Valle},
       Publisher = {Apogeo Education - Maggioli Editore},
       Title = {Introduzione a SuperCollider},
       Address = {Milano}
       Year = {2015}}
```


47822 Santarcangelo di Romagna (RN) • Via del Carpino, 8
Tel. 0541/628111 • Fax 0541/622595
www.maggioli.it/servizioclienti
e-mail: clienti.editore@maggioli.it

Typeset with ConTEXt and SuperCollider by Andrea Valle

Indice

1 SuperCollider grado 0

1.1 Ciò di cui si parla: SuperCollider

SuperCollider (SC) è un pacchetto software per la sintesi e il controllo dell'audio in tempo reale. Attualmente rappresenta lo stato dell'arte nell'ambito della programmazione audio: non c'è altro software disponibile che sia insieme così potente, efficiente, flessibile. Eppure, c'è indubbiamente un motivo per cui spesso SC è guardato con sospetto o con timore: SC non è esattamente "intuitivo". In primo luogo, l'interfaccia che offre all'utente è di tipo testuale. Propriamente, SuperCollider si scrive: quanto basta per spaventare l'utente usuale che nell'uso del calcolatore è abituato a *vedere* più che a *leggere*. In secondo luogo, come tutti i software dedicati all'audio avanzato, SuperCollider richiede competenze relative alla sintesi del segnale audio e alla composizione musicale (nel senso più lato, aperto e sperimentale del termine), ma vi aggiunge competenze più strettamente informatiche, ad esempio, oltre alla teoria dei linguaggi di programmazione, il riferimento al concetto di architettura client/server. Tuttavia, quelli che appaiono come ostacoli sono in realtà il punto di forza di SC: la sua versatilità per le applicazioni software avanzate, la generalità in termini di relazioni con altri aspetti di modellizzazione informatica, l'espressività in termini di rappresentazioni simboliche agevolmente ottenibili.

SuperCollider è stato originariamente sviluppato da James McCartney a partire dal 1996 su piattaforma Macintosh. In particolare, la versione 2 era fortemente integrata con il sistema operativo Mac OS9. SuperCollider 3 (che è insieme molto simile e molto diverso da SC 2) è stato sviluppato inizialmente per

il sistema operativo Mac OSX ed è ora un software open source, a cui contribuisce una consistente comunità di programmatori a partire dal lavoro di James McCartney. La comunità di sviluppatori ha così effettuato il porting anche per le piattaforme Windows e Linux. Gli ultimi anni hanno visto grandi cambiamenti nel software, non nel senso del funzionamento di base (che è sostanzialmente immutato, per quanto costantemente migliorato), quanto piuttosto nella interfaccia con l'utente. Le prime versioni di SC per sistemi diversi da Mac OSX differivano principalmente per difetto, nel senso che alcune funzionalità presenti in quest'ultima non erano state portate nelle altre due. In particolare, le versioni Linux e OSX si sono progressivamente allineate in termini di funzionalità, ma differivano comunque in termini di interfaccia utente, mente la versione Windows è stata a lungo una sorella minore, utilizzabile ma non con la fluidità complessiva delle altre due. Questa situazione è stata radicalmente cambiata dall'introduzione, nella versione 3.6 del software (quella di riferimento per questo manuale), del nuovo ambiente di sviluppo integrato (IDE), che si è aggiunto al nuovo sistema di help files già introdotto nella versione 3.5. Quest'ultimo è ora del tutto integrato nell'IDE, e ormai SC si presenta e funziona allo stesso modo su tutte e tre le piattaforme, offrendo perciò un'esperienza d'uso omogenea per tutti gli utenti[1].

Cosa si può fare con SuperCollider? La comunità SC è decisamente variegata, e SC viene impiegato con successo per la costruzione di sistemi per l'improvvisazione live, incluse interfacce grafiche; per la realizzazione di musica dance (nel senso più vago del termine); per la composizione elettroacustica; per la spazializzazione del suono; per l'interfacciamento audio nel contesto del physical computing; per la sonificazione in progetti scientifici; per l'integrazione audio in contesti multimediali che includano più applicazioni in rete; per la definizione di sistemi distribuiti per la performance (laptop ensemble). Ci sono poi alcuni progetti che in qualche misura sono possibili solo con SC o che almeno sono stati resi possibili a partire dal paradigma affermatosi con SC. Ad esempio, il cosiddetto "live coding" è una pratica di improvvisazione in cui l'esecuzione strumentale –per usare un'espressione curiosa nel contesto ma non erronea– consiste nello scrivere codice dal vivo per controllare i processi di generazione e elaborazione del suono[2]. Il live coding, ormai possibile anche in altri ambienti software, nasce proprio con SuperCollider (in qualche modo, è connaturato

[1] Come si vedrà, in realtà ci sono molte possibilità di personalizzazione dell'uso di SC, che dipendono appunto dalla sua natura linguistica.

[2] Il sito di riferimento è

```
http://toplap.org
```

alla sua natura). Un altro progetto "esoterico" in SuperCollider è SC-Tweet, un approccio in cui il musicista è vincolato a produrre codice della lunghezza massima di 140 caratteri (la dimensione dei tweet "legali"): microprogrammi che, una volta esegui, possono produrre risultati di complessità sorprendente[3].

Dunque, ci sono molti buoni motivi per imparare a usare SuperCollider.

1.2 SC a volo d'uccello

Ripartendo dall'inizio, SuperCollider è dunque un pacchetto software per la sintesi e il controllo dell'audio in tempo reale. La definizione di "pacchetto software" tuttavia si rivela piuttosto vaga. Per arrivare ad una definizione più precisa ed esauriente, è meglio partire dalla seguente definizione di SC (che appariva sulla homepage del sito ufficiale precedente su Sourceforge):

> "SuperCollider is an environment and programming language for real time audio synthesis and algorithmic composition. It provides an interpreted object-oriented language which functions as a network client to a state of the art, realtime sound synthesis server"
> Più analiticamente:

1. *an environment*: SC è un'applicazione che prevede più componenti separate. Di qui l'utilizzo del termine "ambiente".

2. *and*: SC è anche *un'altra cosa* del tutto diversa.

3. *a programming language*: SC è infatti anche un linguaggio di programmazione. Come si dice nel seguito della definizione, appartiene alla famiglia dei linguaggi "orientati agli oggetti" (se ne discuterà ampiamente più avanti). Il codice del linguaggio SC, per essere operativo ("per fare qualcosa"), deve essere interpretato da un interprete. Un interprete è un programma che

[3] La prima collezione di SC-Tweet è l'album *sc140* in collaborazione con la rivista *The Wire*

`http://www.thewire.co.uk/audio/tracks/supercollider-140.1?SELECT\%20*\%20\%20FROM\%20request_redirect`

"capisce" il linguaggio e agisce di conseguenza. SC è *anche* l'interprete del linguaggio SC.

4. *for realtime sound synthesis*: SC è ottimizzato per la sintesi del segnale audio in tempo reale. Questo lo rende ideale per un utilizzo strumentale (performance live) così come per la realizzazione di installazioni/eventi. È senz'altro possibile utilizzare SC non in tempo reale per generare materiale audio, ma in un certo senso è meno immediato che non utilizzarlo in tempo reale.

5. *and algorithmic composition*: uno dei punti di forza di SC sta nel permettere due approcci complementari e opposti alla sintesi audio. Da un lato, permette di svolgere operazioni di basso livello sul segnale audio. Dall'altro, permette al compositore di esprimersi ad alto livello, cioè non in termini di campioni audio, ma di strutture che rappresentino oggetti per la composizione musicale (ad esempio: scale, pattern ritmici, etc.). In questo senso, si rivela ideale per la composizione algoritmica, ovvero per un approccio alla composizione musicale basato sull'utilizzo di procedure formalizzate. In SC questo tipo di operazioni può essere svolto interattivamente ed in tempo reale.

6. *[the] language [...] functions as a network client to a [...] server*: l'applicazione che interpreta il linguaggio SC è *anche* un cliente che comunica, attraverso una rete, con un server, un fornitore di servizi.

7. *a state of the art*: attualmente SC rappresenta lo stato dell'arte nell'ambito della programmazione audio: non c'è altro software disponibile che sia insieme così potente, efficiente, flessibile (e ormai anche portabile).

8. *sound synthesis server*: SC è un fornitore di servizi, in particolare di servizi audio. La locuzione può sembrare misteriosa. Si traduce così: SuperCollider genera audio in tempo reale su richiesta. In questo senso, SC fornisce audio su richiesta: chi richiede audio a SC è un suo cliente (client).

Riassumendo: quando si parla di SC si possono indicare (generando una certa confusione) sei cose diverse. Come schematizzato in Figura **1.1**, queste cose sono:

1. un server ($\rightarrow$ un fornitore di servizi) audio
2. un linguaggio di programmazione per l'audio
3. l'interprete ($\rightarrow$ il programma interprete) per il linguaggio

4. l'interprete in quanto cliente del server audio
5. il programma (→ l'applicazione complessiva) che comprende tutte le componenti 1-4
6. e che include l'IDE, cioè l'editor di scrittura del codice, il quale recepisce l'input dell'utente e lo dispaccia opportunamente all'interprete.

L'applicazione SC prevede tre parti: la prima è il server audio (denominato *scsynth*); la seconda è l'interprete per il linguaggio (denominato *sclang*) che, oltre a interpretare il linguaggio SuperCollider, svolge il ruolo di client rispetto a scsynth; la terza è l'IDE. Le tre applicazioni comunicano tra di loro all'interno dell'applicazione complessiva[4].

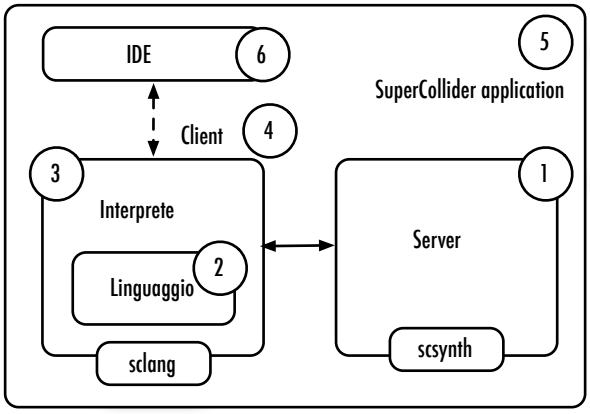


Fig. 1.1 Struttura di SC.

Può sembrare complicato. In effetti lo è.

Installare SC vuol dire perciò installare un'applicazione complessiva che comprende, se si esclude l'IDE che è integrato nell'applicazione, due programmi potenzialmente del tutto autonomi: un server audio e un interprete del linguaggio/client del primo. Si vedrà in seguito meglio che cosa indicano i termini: per ora si tenga a mente che esistono due programmi distinti, e che quando si installa SC si ottengono due programmi al costo di 1 (il costo si calcola così:

[4] Senza addentrarsi per ora nel dettaglio, la comunicazione IDE/sclang e quella sclang/scsynth avvengono secondo modalità diverse: di qui la differenza tra corpo solido e tratteggiato nelle due frecce.

$2 \times 0 = 0$. Come recita un madrigale di Cipriano de Rore, "mia benigna fortuna").

1.3 Installazione e uso

Il sito ufficiale di SC è

`http://supercollider.github.io/`

Gestito dalla comunità, include sostanzialmente tutte le informazioni necessarie per utilizzare SC, oltre a collezionare risorse didattiche (la sezione "learning") e a offrire molti esempi d'uso (si veda in particolare la sezione "video" del sito). Rispetto alla comunità, sebbene esistano ovviamente varie risorse di tipo "social" (forum, gruppi facebook etc), vale la pena menzionare la storica mailing list degli utilizzatori, attraverso la quale è possibile confrontarsi rapidamente con i migliori programmatori di SC:

`http://www.beast.bham.ac.uk/research/sc_mailing_lists.shtml`

È una mailing list ad alto traffico, che assicura tipicamente risposte immediate, indipendentemente dal livello delle domande poste. Gli archivi della lista sono poi una risorsa preziosa, in cui molto spesso è già disponibile una risposta ai propri quesiti.

Infine, il sito di James McCartney ha un valore eminentemente storico:

`http://www.audiosynth.com/`

SuperCollider può perciò essere scaricato da Github, in particolare all'indirizzo:

`http://supercollider.github.io/download.html`

Nel caso si utilizzino i file binari (OSX e Windows), l'installazione in sé è un processo banale, del tutto analogo a quanto avviene per le usuali applicazioni, e su cui perciò non vale la pena spendere tempo. L'installazione su Linux prevede anch'essa modalità tipiche del pinguino, variabili in funzione della distribuzione prescelta.

Le estensioni di SuperCollider, cioè l'insieme delle componenti non incluse nella distribuzione, sono definite *quarks*, e la loro installazione è possibile dall'interno del programma stesso.

Alla sua esecuzione, l'applicazione SC si può presentare come in Figura **1.2**. Quella riportata non è la disposizione usuale dei campi grafici, e dimostra già una possibilità di personalizzazione dell'IDE: in ogni caso gli elementi sono

gli stessi. Sul lato destro è presente un campo testuale (vuoto) deputato alla scrittura del codice. Sul lato sinistro si possono individuare due componenti: la finestra di stampa (Post window) attraverso la quale l'interprete restituisce all'utente i risultati del codice valutato (in questo caso, informazioni relativi all'inizializzazione del programma stesso); la finestra del sistema di aiuti (Help browser, qui nascosta dalla post window), che è riservata alla visualizzazione degli help files e alla loro navigazione. Nella barra in basso, in particolare sul lato destro, sono visualizzate informazioni rispetto a sclang (Interpreter, attivo, in verde) e a scsynth (Server, inattivo, informazioni in bianco).

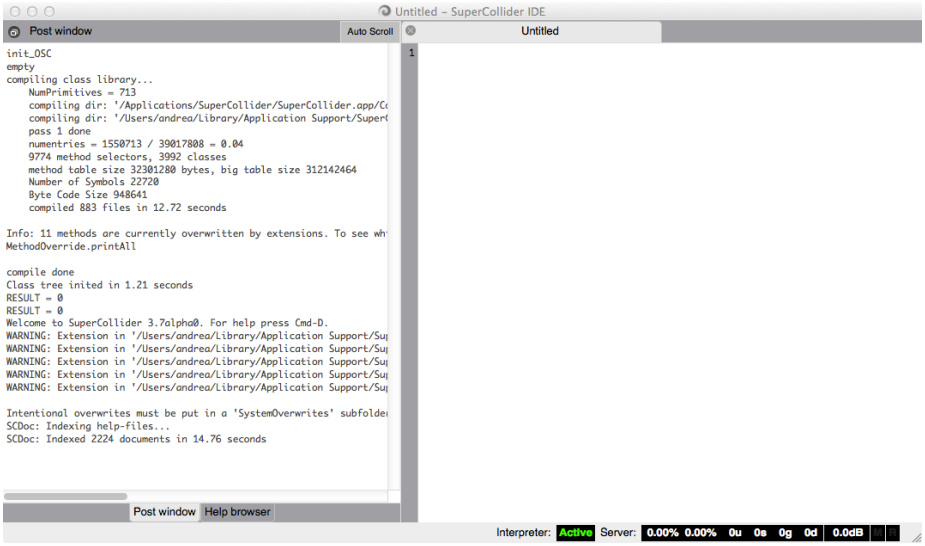


Fig. 1.2 IDE di SC.

Il sistema dei menu dell'IDE non è particolarmente complesso. Seguono alcune indicazioni di massima o indicazioni in relazione alle peculiarità di SC, molte delle quali verrano riprese in seguito. Al lettore l'esplorazione nei dettagli dell'interfaccia.

- *File*: dedicato all'apertura e salvataggio dei documenti di SC. Questi ultimi sono file in formato testuale ASCII e possono essere di tre tipi, in funzione dell'estensione:

  — `scd` è dedicata ai documenti (Super Collider Document);
  — `sc` è dedicata ai file per le definizioni delle classi di SC (si veda dopo);

  - `schelp` è riservata per la scrittura dei file di help, che richiedono una sintassi specifica per essere visualizzati nell'Help browser;
    la voce *Open startup file* permette di modificare un file di configurazione che viene eseguito ogni volta che si apre il programma SC.

- *Session*: permette di gestire in maniera unitaria gruppi di file che appartengono allo stesso progetto ;

- *Edit*: contiene un insieme di voci per l'usuale editing dei testi. L'ultima sezione del menu include aspetti propri della scrittura del codice. Al di là dell'editing, di particolare utilità nell'interazione con l'utente è "Select Region";

- *View*: permette di gestire gli aspetti visivi dell'IDE (ad esempio la disposizione reciproca di Post window, Help Browser e finestra di scrittura. La Figura **1.2** mostra una disposizione personalizzata) ;

- *Language*: è il menu che si riferisce ad aspetti tipici esclusivamente di SC. È diviso in cinque blocchi:

  - il primo concerne la gestione dell'interprete. Ad esempio, sclang (che è un programma autonomo) può essere chiuso (`Quit`) o riavviato (`Reboot`);
  - il secondo concerne la gestione del server audio (anche scsynth è un programma autonomo);
  - il terzo permette di attivare alcune interfacce grafiche per la visualizzazione di informazioni sul server audio ;
  - il quarto è dedicato alla valutazione del codice, cioè ad indicare all'interprete quale codice prendere in considerazione per l'interpretazione (e per le azioni che ne conseguono) ;
  - il quinto infine include un insieme di comandi per esplorare interattivamente il codice sorgente. SC è infatti scritto per larga parte in SC, e questo permette una forte "introspezione", cioè la possibilità per SC di esplorare la sua stessa struttura;

- *Help*: consente di accedere interattivamente ai file di help collegati con i relativi elementi di SC.

Riassumendo, come si usa SC? Scrivendo codice, selezionando una riga o un blocco di righe, chiedendo all'interprete di valutare quanto selezionato. Navigando nei file di documentazione. Quindi, ripetendo interattivamente le azioni precedenti durante una sessione di lavoro che è a tutti gli effetti una sessione

di scrittura. Ma anche valutando un'unica volta un programma scritto in precedenza che può innescare un processo sonoro autonomo di durata indefinita o costruire interfacce grafiche che saranno utilizzate nella performance. Come si vede, sono possibili stili diversi di lavoro: l'unica azione che non può essere omessa almeno una volta è la valutazione di un blocco di codice che rappresenti un programma in SC.

1.4 Obiettivi, riferimenti, convenzioni tipografiche

L'obiettivo del testo che segue è allora duplice:

1. fornire una breve panoramica ed insieme un'introduzione a SuperCollider in italiano, visto che allo stato attuale non ne esistono altre;

2. introdurre alcuni aspetti fondamentali relativi a quell'ambito che viene usualmente indicato, con una definizione a volte imprecisa e a volte impropria, *computer music*.

L'ipotesi di partenza (e indipendentemente dai risultati) è quella di far interagire i due ambiti, attraverso una reciproca specificazione, e di mostrare come SuperCollider possa essere lo strumento ideale per farlo. Ma, poiché SuperCollider richiede a tutti gli effetti di programmare, questa introduzione è anche una introduzione alla programmazione, per quanto molto obliquamente, attraverso cioè il caso specifico dell'audio e della musica e il paradigma linguistico interpretato a oggetti di SuperCollider.
Il materiale presentato è, in qualche misura, "originale". La parte relativa alla sintesi riprende alcune parti di *Audio e multimedia*[5], testo a cui questo manuale farà più volte riferimento poiché ne condivide il taglio di introduzione avanzata ma non troppo tecnica. Trattandosi di materiale introduttivo, è chiaro che il testo che segue si affida saldamente al suo intertesto costituito dai molti testi analoghi, più complessi e più completi, che lo precedono[6]. Questo vale a

[5] Lombardo, V. e Valle, A., *Audio e multimedia*, Milano, Apogeo 2014 (4$^{ed.}$), d'ora in poi AeM.

[6] Non è questo il luogo per una bibliografia sulla computer music, ma si vedano in proposito i riferimenti in AeM, cap. 5.

maggior ragione per la parte dedicata a SuperCollider. Il manuale non è una traduzione di scritti già esistenti, né tantomeno un sostituto o un alternativa per il testo di riferimento, *The SuperCollider Book*[7]. Quest'ultimo, prendendo a modello il suo analogo precedente relativo al linguaggio Csound (*The Csound Book*), è costituito da un insieme di capitoli che includono tutorials, approfondimenti e descrizioni di applicazioni e progetti, ma non è pensato come una guida unitaria, che è lo scopo invece della presente trattazione. E tuttavia è chiaro che il materiale presentato è fittamente intessuto di prestiti provenienti dagli help file, dai tutorial di James McCartney, Mark Polishook, Scott Wilson[8], dal testo di David Cottle[9], dalle osservazioni preziosi fornite dalla SuperCollider mailing list[10]. La lettura di queste fonti non è in nessun modo resa superflua dal testo seguente, il quale ha invece semplicemente un ruolo propedeutico rispetto alle stesse, perché quantomeno evita al lettore italiano la difficoltà supplementare della lingua straniera.

L'ordinamento delle materie nel caso di SC è sempre molto difficile, appunto perché in qualche misura tutto (programmazione, architettura, audio) andrebbe introdotto insieme. Dalla prima edizione (2008) di questo manuale ho pensato più volte a modificare la sequenza dei capitoli in modo da arrivare immediatamente alla sintesi del suono, che è tipicamente il motivo di interesse dell'utente normale di SC. Tuttavia, se nell'esperienza dell'insegnamento questo è senz'altro preferibile, in una discussione testuale si rischia di lasciare troppi puntini di sospensione e di generare confusione nel momento in cui gli aspetti linguistici del codice utilizzato non sono stati discussi in precedenza. L'ordinamento segue perciò volutamente un percorso "arido", che arriva alla programmazione audio soltanto dopo aver discusso il linguaggio SC e il suo uso in quanto tale.

Dal punto di visto tipografico, il manuale prevede tre distinzioni:

---

[7] S. Wilson, D. Cottle, N. Collins, *The SuperCollider Book*, Cambridge, MIT press 2011. Una risorsa recente è M. Koutsomichalis, *Visualisation in SuperCollider* , che per quanto dedicato alla visualizzazione in SuperCollider offre una discussione molto sofisticata ma passo passo sia di SuperCollider in generale che di applicazioni potenzialmente multimediali.

[8] I tutorial sono inclusi nel sistema di help files di SC.

[9] D. Cottle, *Computer Music with examples in SuperCollider 3*, `http://www.mat.ucsb.edu/275/CottleSC3.pdf`

[10] Va altresì segnalata la sezione "Docs" sul sito Github che raccoglie molte risorse didattiche relative a SC.

1. testo: in carattere nero normale, senza particolarità, esattamente come quanto scritto qui ;

2. codice: è scritto in carattere `monospaced`, utilizza uno schema di colorazione della sintassi, e le righe sono numerate. Al di sotto di ogni esempio è presente un marcatore interattivo. Esso permette di accedere al file sorgente dell'esempio *che è incluso nel pdf*, e di aprirlo direttamente con l'applicazione SuperCollider. Al lettore interessato a questa funzione è consigliato l'uso di *Adobe Acrobat Reader* (che è gratuito e multipiattaforma), perché supporta questa caratteristica avanzata del formato PDF[11]. Alla prima apertura *Acrobat Reader* richiede di impostare il suo comportamento nei confronti dell'allegato.

```
1 // ad esempio
2 "a caso".postln ;
```

3. post-window (finestra di stampa): è scritto in nero; con carattere `monospaced`, e riporta una parte di sessione con l'interprete SuperCollider. È riquadrato in arancio e le righe sono numerate[12].

```
1 ad esempio
```

[11] Ad esempio, il visualizzatore di sistema su OSX, *Anteprima*, non mostra neppure l'icona del marcatore.

[12] In alcuni rari casi, lo stesso formato viene utilizzato per riportare frammenti testuali non in linguaggio SC. Nel caso, il contesto disambiguerà l'uso. Si noti che anche in questo caso è comunque presente il marcatore che permette di aprire il file in SC.

2 Programmazione in SC

2.1 Linguaggi di programmazione

Un calcolatore è un dispositivo in grado di manipolare entità simboliche rappresentabili in forma binaria. In altri termini, l'informazione in un calcolatore è rappresentata da sequenze di quantità che possono avere solo due valori possibili, 0/1, accesso/spento, e così via. La fondazione teorica della scienza dei calcolatori viene tipicamente ascritta a Alan Turing, che per primo propose una formalizzazione astratta di una macchina pensata come dispositivo di lettura/scrittura su una memoria. Salvo rivoluzioni per ora molto discusse ma senza risultati empirici, tutti i computer su cui l'umanità lavora implementano questa macchina astratta di Turing in una architettura tecnologica che si chiama, dal suo ideatore, di Von Neumann. Costruito il calcolatore, il problema sta nel controllarlo, cioè nel definire le operazioni di lettura/scrittura che questa macchina dovrà eseguire. Una simile operazione può essere svolta a contatto stretto con la macchina, specificando cioè direttamente come essa dovrà gestire la manipolazione delle celle di memoria. Questa operazione di codifica delle operazioni da svolgere può essere svolta in modi diversi. Si consideri a mo' d'esempio il caso della rappresentazione binaria, che è il modo in cui alla fin fine tutta l'informazione è descritta in un calcolatore. La descrizione di un carattere alfabetico –che si può scrivere in un editor di testo quale quello dell'IDE di SC– richiede 7 bit (è la cosiddetta codifica ASCII). In altri termini, ogni carattere è associato a una sequenza di sette 0/1. Ad esempio, nella codifica ASCII il codice binario di 7 bit `1100001` rappresenta il carattere a. È abbastanza intuitivo

che descrivere esplicitamente l'informazione a questo livello (bit per bit) comporta uno sforzo tale per cui l'entusiasmo nell'uso del calcolatore rapidamente svanisce. Invece, uno degli aspetti affascinanti della teoria della programmazione sta nel fatto che certe rappresentazioni possono essere descritte ad un livello più alto in forma unitaria. Si noterà ad esempio che `a` è una rappresentazione molto più compatta e intuitiva di `1100001`. Analogamente, la sequenza di caratteri `SuperCollider` è dunque rappresentata da questa sequenza binaria:

```
    01010011011101010111000001100101011100100100000110110
111101101100011011000110100101100100011001010111001 0
```

La sequenza alfabetica `SuperCollider` è dunque più "astratta" e di più "alto livello" di quella binaria. Se si assume che quest'ultima rappresenti il livello più basso, cioè l'unico con cui si può effettivamente "parlare" alla macchina, nel momento in cui si vuole controllare la macchina attraverso la sequenza alfabetica diventa necessario tradurla in sequenza binaria. Chi svolge questo lavoro? Un programma "traduttore". Ad esempio, la rappresentazione binaria precedente è ottenibile in SuperCollider attraverso il programma seguente, che è perciò un traduttore del programma `SuperCollider` nel programma binario:

```
1 "SuperCollider".ascii.collect{|i| i.asBinaryDigits}
2 .flat.asString.replace(", ", "")
```

Le conseguenze della discussione minimale precedente sono due.

La prima è che nel momento in cui c'è un linguaggio di programmazione, c'è sempre bisogno di un programma traduttore, che tipicamente prende il nome di "compilatore" o di "interprete".

Un compilatore produce in uscita un eseguibile, cioè un programma che può essere eseguito. Ad esempio, l'*applicazione* SuperCollider, come praticamente tutti i programmi che risiedono sui computer, risulta dalla compilazione di codice scritto nel linguaggio C++. In altri termini, il codice C++ viene compilato e produce in uscita l'eseguibile (cioè il programma SC che l'utente lancia). Gli sviluppatori (non gli utenti!) di SuperCollider scrivono codice C++ che serve per costruire il programma complessivo, che include le funzionalità di base dell'audio, l'interfacciamento con il sistema operativo, l'utilizzo di librerie già pronte per la gestione di GUI, e così via. Quindi compilano questi codici, detti

sorgenti[1], attraverso un compilatore che è specifico per ognuna della piattaforme e architetture supportate (rispettivamente ad esempio OSX, Linux, Windows, e Intel, PowerPc, e così via), e ottengono l'eseguibile (quello rappresentato in Figura **1.1**). Essendo un progetto open source, questi codici sorgenti sono disponibili a tutti, e ognuno può eventualmente compilarli sul proprio computer (a patto di avere un compilatore installato).

Un interprete invece non genera una rappresentazione intermedia che sarà poi eseguita in seguito, ma traduce ed esegue direttamente le istruzioni specificate. Poiché il *linguaggio* SuperCollider è interpretato, il codice SC dell'esempio precedente può essere valutato dentro l'IDE (che lo passa all'interprete) e viene immediatamente eseguito. Per essere eseguito in SuperCollider, il codice deve essere selezionato, quindi deve essere lanciato il comando "Evaluate Selection or Line" dal menu *Language* (intuibilmente, è opportuno familiarizzarsi da subito con le combinazioni di tasti per la scelta rapida). La conseguenza dell'interpretazione è che sulla post window verrà stampata la sequenza binaria di cui sopra.

SuperCollider, in quanto linguaggio, è dunque un linguaggio interpretato, e l'utente si rapporta con l'interprete. L'interpretazione avviene immediatamente (il codice viene eseguito il più in fretta possibile) e l'utente continua ad avere l'interprete disponibile: è dunque un processo interattivo, in cui l'utente si rapporta propriamente con un *ambiente (*environment*) di programmazione*. Se si seleziona di nuovo il codice (nel caso di valutazione di singole linee è sufficiente posizionare il cursore nella linea che interessa), lo si può valutare nuovamente, e nuovamente la post window recherà traccia dell'avvenuta interazione. Va dunque tenuta presenta una possibile fonte di confusione. Da un lato, il codice selezionato viene eseguito in sequenza (l'interprete legge la selezione "riga per riga", più correttamente "espressione per espressione"), dall'altro il documento di testo su cui si lavora interattivamente non è una pagina che si legga necessariamente dall'alto al basso, ma è una lavagna su cui si può scrivere dove si vuole, in alto come in basso, poiché l'ordine di esecuzione del codice è dettato dall'ordine di valutazione. Ciò che rimane scritto è in questo senso una memoria dell'interazione attraverso la scrittura.

Tornando alla questione del rapporto tra linguaggi di programmazione, la seconda conseguenza che emerge nella relazione tra rappresentazione binaria e rappresentazione alfabetica sta nel fatto che i livelli tra cui si stabilisce una relazione, ad esempio di astrazione, non sono necessariamente soltanto due. In

[1] I sorgenti includono anche codice SC, che è accessibile mentre si usa SC (si veda dopo).

altri termini, si potrebbe pensare ad un altro linguaggio in cui tutta la stringa "SuperCollider" è rappresentata da un unico carattere, ad esempio `@`. Questo linguaggio sarebbe ad un livello di astrazione superiore, perché la sua compilazione/interpretazione produrrebbe in uscita una rappresentazione (`SuperCollider`) che a sua volta dovrebbe essere interpretata/compilata. Senza addentrarci, il concetto è ci possono essere diversi livelli di astrazione rispetto alla macchina fisica, secondo una stratificazione la cui utilità è di solito esattamente quella di far dimenticare all'utente finale la macchina stessa (che pure c'è) fornendogli invece costrutti linguistici specifici, utili ad esempio in certi ambiti di utilizzo o rispetto a certi paradigmi concettuali. La distinzione tra livello alto e basso nei linguaggi è di solito relativa alla vicinanza/lontananza dall'utente. I linguaggi ad alto livello sono più vicini a forme concettuali "umane" (e tipicamente meno efficienti), quelli a basso livello agli aspetti tecnologici del computer (e tipicamente più efficienti). SuperCollider è un linguaggio di alto livello, che però fa affidamento per l'audio (un ambito computazionalmente intensivo) ad una componente dedicata di basso livello e molto efficiente, il server audio. In questo modo l'applicazione SC complessiva riesce a tenere insieme alto livello nel controllo ed efficienza nella computazione audio. Infine, nella costruzione dei linguaggi di programmazione si sono progressivamente affermati più modelli di riferimento, veri e propri paradigmi concettuali: in particolare si distingue tra paradigmi imperativo, funzionale, logico e a oggetti.

SuperCollider è allora un linguaggio di programmazione dotato di una buona generalità, ma che è pensato per descrivere concetti propri dell'ambito dell'audio digitale e della musica. È un linguaggio di alto livello, che permette cioè di rappresentare quegli stessi concetti in maniera concettualmente elegante e indipendentemente dalla sua implementazione. È un linguaggio che segue il paradigma della programmazione ad oggetti. Quest'ultima è al centro della prossima sezione.

2.2 Minima objectalia

Nella programmazione orientata agli oggetti (*Object-oriented programming*, OOP) si assume che l'utente, per programmare il comportamento di un calcolatore, manipoli entità dotate di proprietà e di capacità. Il termine, volutamente generico, per indicare queste entità è "oggetti", mentre tipicamente le proprietà

sono pensate come "attributi" degli oggetti stessi e le capacità come "metodi" che gli oggetti possono adottare per compiere delle operazioni.

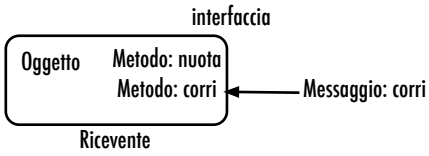


Fig. 2.1 Struttura e comunicazione nell'OOP.

Nel paradigma object-oriented il mondo si presenta allora al programmatore come un insieme di oggetti che si prestano, sotto determinate condizioni, ad essere manipolati. In particolare, per manipolare un oggetto –per chiedergli di fare qualcosa– è necessario inviargli un "messaggio". Un oggetto, per poter rispondere al messaggio, deve conoscere un metodo. In sostanza, può rispondere ad una richiesta (messaggio) attraverso una competenza (metodo). In termini di comunicazione, l'oggetto che riceve il messaggio è il "ricevente" di quel messaggio e vi può rispondere se implementa un metodo corrispondente. Riassumendo:

- *oggetto* e *metodo* concernono la *definizione* dell'oggetto *dall'interno*
- *messaggio* e *ricevente* concernono la *comunicazione* con l'oggetto *dall'esterno*

L'insieme dei messaggi a cui un oggetto può rispondere prende il nome di "interfaccia": ed è un'interfaccia in senso proprio, perché è ciò che l'oggetto rende disponibile all'utente per l'interazione, dove l'utente può essere anche un altro oggetto. La situazione è schematizzata in Figura **2.1**. Nella maggior parte dei linguaggi ad oggetti, la sintassi tipica per inviare un messaggio ad un oggetto utilizza il punto (`.`) e prende la forma `oggetto.messaggio`. La relazione tra ricevente e messaggio non va pensata come una descrizione alla terza persona ("l'oggetto fa una certa cosa"), ma piuttosto come una coppia vocativo/imperativo: "oggetto, fai qualcosa!". Ad esempio, `conio.imprimi`, o anche: `ippopotamo.nuota`. Insomma, detto figurativamente, il programmatore è una sorta di apprendista stregone che si trova a comandare un insieme di oggetti eterogenei[2].

[2] Non a caso, il paradigma OOP è considerato anche come una specificazione del

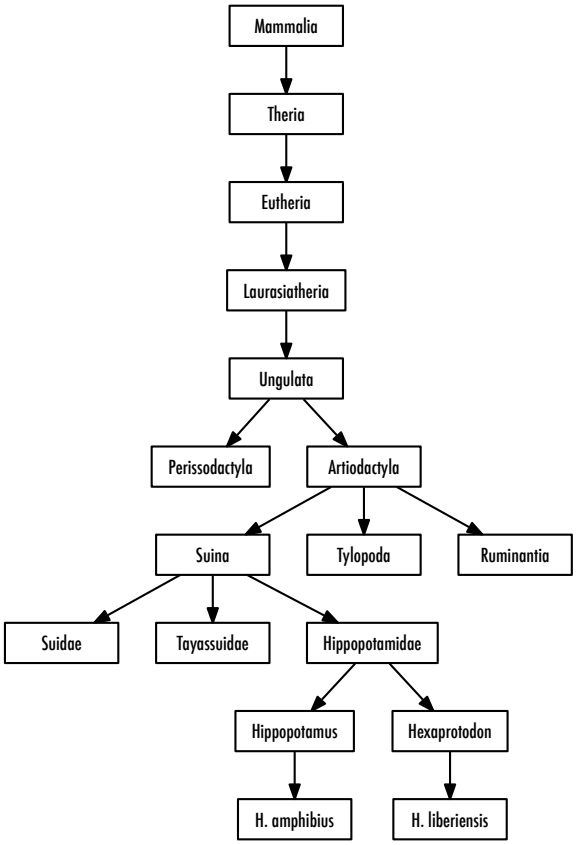


Fig. 2.2 Tassonomia dell'ippopotamo.

Per poter essere riconosciuti dal linguaggio, gli oggetti devono appartenere ad un insieme finito di tipi: un oggetto è del tipo A, l'altro del tipo B e così via. I tipi vengono chiamati "classi" in OOP. Un oggetto è dunque una particolare "istanza" di una classe: la classe può essere pensata come il tipo astratto, ma anche come lo stampo da cui si fabbricano le istanze. Da un unico conio (la classe) si stampa un un numero indefinito di monete uguali (gli oggetti istanziati). È

---

paradigma imperativo.

nella classe che si definiscono i metodi di cui tutti gli oggetti di quel tipo saranno dotati. Una classe descrive anche il modo in cui creare un oggetto a partire dalla classe stessa.

Le classi sono organizzate gerarchicamente: ogni classe può derivare da un'altra classe e ogni classe può avere delle classi derivate. Questo principio prende il nome di "ereditarietà". Ad esempio, un conio è una ulteriore specificazione di un più generico "stampo": lo stampo è la superclasse del conio, e il conio è una sottoclasse dello stampo. Un sigillo (per la ceralacca) è un altro stampo, ma di un tipo completamente diverso dal conio: il sigillo è una sottoclasse dello stampo, da cui eredita alcuni aspetti che condivide con il conio (la capacità di impressione), ma da cui si differenza per altri (prevede un'impugnatura manuale, mentre il conio viene battuto a martello).

L'ereditarietà va pensata in termini genetici: i caratteri del padre sono presenti (come patrimonio genetico, appunto) nei figli, ma, a scanso di equivoci, si noti che qui ereditarietà è intesa in termini sistematici, non evolutivi. La relazione di ereditarietà prende infatti a modello le tassonomie naturalistiche. Ad esempio, il grafo di Figura **2.2** illustra la posizione tassonomica dell'ippopotamo. L'ippopotamo appartiene al subordine dei Suina (ad esempio, intuibilmente, i maiali), con i quali condivide alcuni tratti, che differenziano entrambi dai Ruminantia (ad esempio, le mucche), pur essendo Ruminantia e Suina entrambi Artiodactyla (e distinguendosi entrambi dai Perissodactyla, ad esempio i cavalli). Se le classi sono astratte (ad esempio, la species dell'ippopotamo), gli oggetti (gli ippopotami in carne ed ossa da classificare) sono concreti.

La ragione alla base del paradigma a oggetti risiede nell'applicazione di un principio di incapsulamento. Ogni classe (e ogni oggetto che ne deriva) definisce in maniera chiara e circoscritta le sue caratteristiche in termini di dati (attributi) e di processi (metodi). Se è un ippopotamo, non può volare.

La ragione alla base del principio di ereditarietà sta in un principio di economia computazionale e cognitiva. Sarebbe inutile definire come proprietà di ogni mammifero (ippopotamo, uomo, lama, iena, etc.) l'allattamento dei pargoli. È invece utile definirla in alto nell'albero tassonomico così che essa venga automaticamente ereditata da tutti i nodi inferiori.

2.3 Oggetti in SC

Il linguaggio SuperCollider è un linguaggio orientato agli oggetti. Lo è per di più in termini molto "puri", poiché ha come suo modello storico, e come parente tipologico assai prossimo, il linguaggio Smalltalk[3]. In Smalltalk, come in SC, letteralmente ogni entità possibile è un oggetto. Questa radicalità può essere spiazzante inizialmente, ma è un punto di forza poiché garantisce che tutte (proprio tutte) le entità potranno essere controllate dall'utente secondo un unico principio: tutte avranno attributi e metodi, e dunque a tutte sarà possibile inviare dei messaggi poiché presenteranno all'utente una certa interfaccia.

Un esempio iniziale particolarmente rilevante è quello delle strutture dati: SC possiede una grande ricchezza di classi che rappresentano strutture dati, cioè entità che funzionano da contenitori di altri oggetti, ognuna dotati di particolari capacità e specializzata per certi tipi di oggetti. Ad esempio un "array", una struttura dati fondamentale in informatica, è un contenitore ordinato di oggetti. Si scriva `Array`. SC sa che `Array` è una classe perché la prima lettera è maiuscola: tutto ciò che inizia con la maiuscola per SC indica una classe. Se si valuta il codice, SC restituisce (per ora si intenda: stampa sullo schermo) la classe stessa. Informazioni su `Array` sono disponibili attraverso l'Help File, selezionando il testo e scegliendo dal menu *Help* la voce `Look Up Documentation for Cursor`. Se si richiama l'Help File, si nota come immediatamente vengano fornite indicazioni sulle relazioni tassonomiche che `Array` intrattiene con le classi dei nodi padre e figli. Ad esempio, si vede immediatamente come erediti da `ArrayedCollection`, che eredita da `SequenceableCollection`, e così a seguire. L'Help File fornisce alcune indicazioni sui metodi disponibili per gli oggetti di tipo Array. Il codice:

```
1 z = Array.new;
```

costruisce un nuovo array vuoto attraverso il messaggio `new` inviato alla classe `Array`[4]. Finora, l'assunto è stato che un messaggio viene inviato ad un'istanza particolare, non ad una classe. Ma prima di poter inviare un messaggio ad un oggetto, è necessario che questo sia stato costruito. Al messaggio

[3] In realtà, SC include anche aspetti propri di altri linguaggi, in primo luogo C e Python.

[4] Si noti come il codice sia colorato in funzione della sintassi: alle classi come `Array` è assegnato (arbitrariamente) l'azzurro.

`new` rispondono allora tutte le classi in SC, restituendo una loro istanza. Il metodo `new` è il "costruttore" della classe: il metodo cioè che istanzia un oggetto a partire dalla classe. Oltre a `new` in una classe possono esserci molti metodi costruttori, che restituiscono un oggetto secondo modalità specifiche: sono tutto metodi di classe perché, inviati alla classe, restituiscono un oggetto. Nel caso di `Array` uno di essi è il messaggio `newClear`, che prevede anche una parte tra parentesi tonde:

```
1 z = Array.newClear(12) ;
```

Nell'esempio le parentesi contengono una lista di "argomenti" (uno solo, in questo caso), che specificano ulteriormente il messaggio `newClear`, ovvero "oggetto, fai qualcosa (così)!"[5]. In particolare `newClear(12)` prevede un argomento (`12`) che indica che l'array dovrà contenere al massimo 12 posti. È possibile indicare esplicitamente il nome dell'argomento: "oggetto, fai qualcosa (nel modo: così)! ". Ogni argomento ha cioè un nome specifico, la sua *keyword*: nel caso di `newClear` è `indexedSize`, che indica il numero di posti contenuti nel nuovo array. Il codice:

```
1 z = Array.newClear(indexedSize:12) ;
```

è identico al precedente ma esplicita la keyword dell'argomento.

Infine, `z =` indica che l'array verrà assegnato alla variabile `z`. Si noti che la lettera utilizzata è minuscola: se si scrivesse `Z` , SC interpreterebbe `Z` come una classe (inesistente) e solleverebbe un errore. Adesso `z` rappresenta un array vuoto di capienza 12: è un'istanza della classe `Array`. Si può chiedere a `z` di comunicare la classe a cui appartiene invocando il metodo `class`:

```
1 z.class
```

[5] Tornando a prima, `conio.imprimi(forte)` o `ippopotamo.nuota(veloce)`.

Il metodo `class` restituisce la classe di `z`: la valutazione del codice stampa sulla post window `Array`. Traducendo in italiano, la frase equivalente potrebbe essere: "z, dichiara la tua classe!". Quando si usano degli array molto spesso si è insoddisfatti dei metodi elencati nell'Help File: sembra che manchino molti metodi intuitivamente utili. È molto difficile che sia veramente così: molto spesso il metodo cercato c'è, ma è definito nella superclasse ed ereditato dalle classi figli. A partire da `Array` si può navigare la struttura degli Help File risalendo a `ArrayedCollection`, `SequenceableCollection`, `Collection`: sono tutte superclassi (di tipo sempre più astratto) che definiscono metodi che le sottoclassi possono ereditare. Se si prosegue si arriva a `Object`. Come recita l'Help File:

> "Object is the root class of all other classes. All objects are indirect instances of class Object."

In altri termini, tutti le classi in SC ereditano da `Object`, e dunque tutti gli oggetti "indirettamente" sono istanze di Object. Un esempio di ereditarietà è così il metodo `class` che è stato chiamato su z nell'esempio precedente: è definito a livello di `Object` ed ereditato, lungo l'albero delle relazioni di ereditarietà, da `Array`, così che un'istanza di quest'ultima classe (`z`) vi possa rispondere.

Al di là della navigazione nella struttura degli Help Files, SC mette a disposizione dell'utente molti metodi per ispezionare la struttura interna del codice: è la capacità che tipicamente viene definita "introspezione". Ad esempio i metodi `dumpClassSubtree` e `dumpSubclassList` stampano su schermo rispettivamente una rappresentazione gerarchica delle sottoclassi della classe su cui è invocato il metodo e una lista in ordine alfabetico. Le due rappresentazioni sono equivalenti. Nella prima sono più chiari i rapporti di parentela tra le classi attraverso la struttura ad albero, nella seconda è invece possibile percorrere -per ognuna delle sottoclassi della classe - la struttura dell'albero lungo i rami ascendenti fino a `Object`. Si prenda la classe `Collection`, una classe molto generale di cui `Array` è una sottoclasse, e si inviino i messaggi `dumpClassSubtree` e `dumpSubclassList`, ovvero:

```
1 Collection.dumpClassSubtree ;
2 Collection.dumpSubclassList ;
```

Quanto segue è quanto appare sulla Post Window nei due casi.

```
Collection
[
  Array2D
  Range
  Interval
  MultiLevelIdentityDictionary
  [
    LibraryBase
      [ Archive Library ]
  ]
  Set
  [
    Dictionary
    [
      IdentityDictionary
      [
        Environment
          [ Event ]
      ]
    ]
    IdentitySet
  ]
  Bag
    [ IdentityBag ]
  Pair
  TwoWayIdentityDictionary
    [ ObjectTable ]
  SequenceableCollection
  [
    Order
    LinkedList
    List
      [ SortedList ]
    ArrayedCollection
    [
      RawArray
      [
        DoubleArray
        FloatArray
          [ Wavetable Signal ]

[...]

]
Collection
```

```
Archive : LibraryBase : MultiLevelIdentityDictionary : Collection : Object
Array : ArrayedCollection : SequenceableCollection : Collection : Object
Array2D : Collection : Object
ArrayedCollection : SequenceableCollection : Collection : Object
Bag : Collection : Object
Collection : Object
Dictionary : Set : Collection : Object
DoubleArray : RawArray : ArrayedCollection : SequenceableCollection :
    Collection : Object
Environment : IdentityDictionary : Dictionary : Set : Collection : Object
Event : Environment : IdentityDictionary : Dictionary : Set : Collection :
    Object
FloatArray : Raw

[...]

36 classes listed.
Collection
```

Con `Collection.dumpClassSubtree` si vede la posizione di `Array` in relazione ai suoi vicini. È sullo stesso livello di `RawArray`, entrambi sono sottoclassi di `ArrayedCollection`. Quest'ultima classe appartiene alla famiglia delle `SequenceableCollection`. Il metodo `Collection.dumpSubclassList` lista le classi in ordine alfabetico: è agevole trovare `Array` per poi seguire i rami dell'albero (lungo la stessa riga) fino a `Object`.

La Figura **2.3** è una visualizzazione tramite un grafo ad albero di una parte della struttura delle classi di `Collection`, ottenuta elaborando automaticamente l'output di `Collection.dumpSubclassList`. L'esempio è tratto dall'Help File *Internal-Snooping*, che è dedicato all'introspezione in SC[6]. L'ultima riga che SC stampa è in entrambi i caso l'oggetto `Collection` (39), che è ciò che i metodi ritornano. Il perché verrà discusso a breve. Se si sostituisce a `Collection` la classe `Object`, si ottiene tutta la struttura delle classi di SC. La Figura **2.4** riporta una rappresentazione radiale della struttura di tutte delle classi di SC, ottenuta elaborando il risultato di `Object.dumpSubclassList`[7]. Si noti come un punto di

[6] Anche l'Help File di `Class` è particolarmente interessante al proposito.

[7] Propriamente si tratta della struttura delle classi installate sulla macchina di scrive, che include alcune classi aggiuntive.

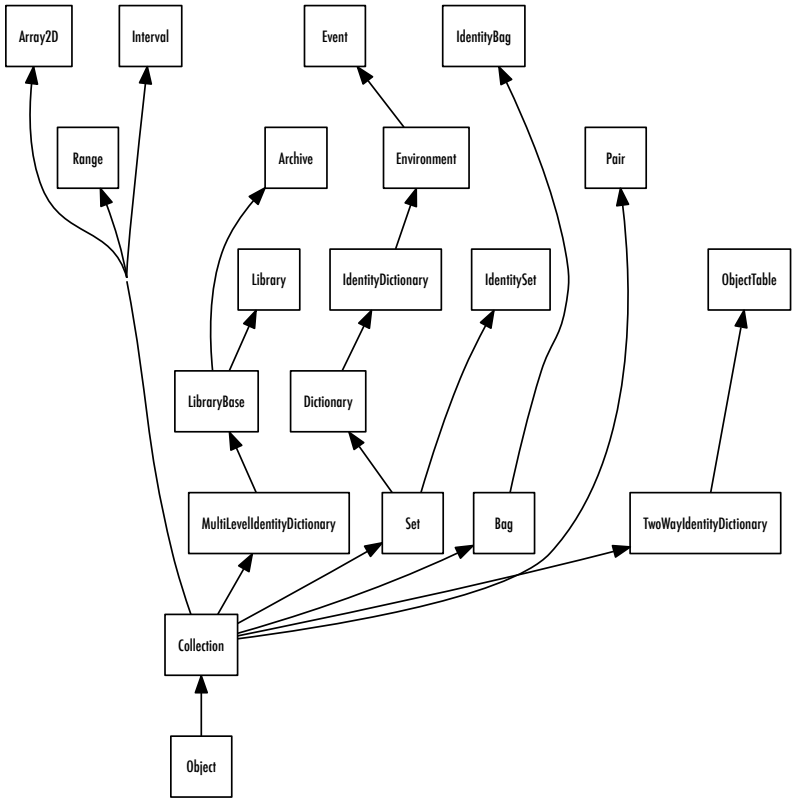


Fig. 2.3 Grafo ad albero di alcune delle sottoclassi di `Collection`, a partire da `Object`.

addensamento sia rappresentato da `UGen`, la superclasse diretta di tutte le classi che generano segnali in SC. È una classe comprensibilmente molto numerosa.

2.4 Metodi e messaggi

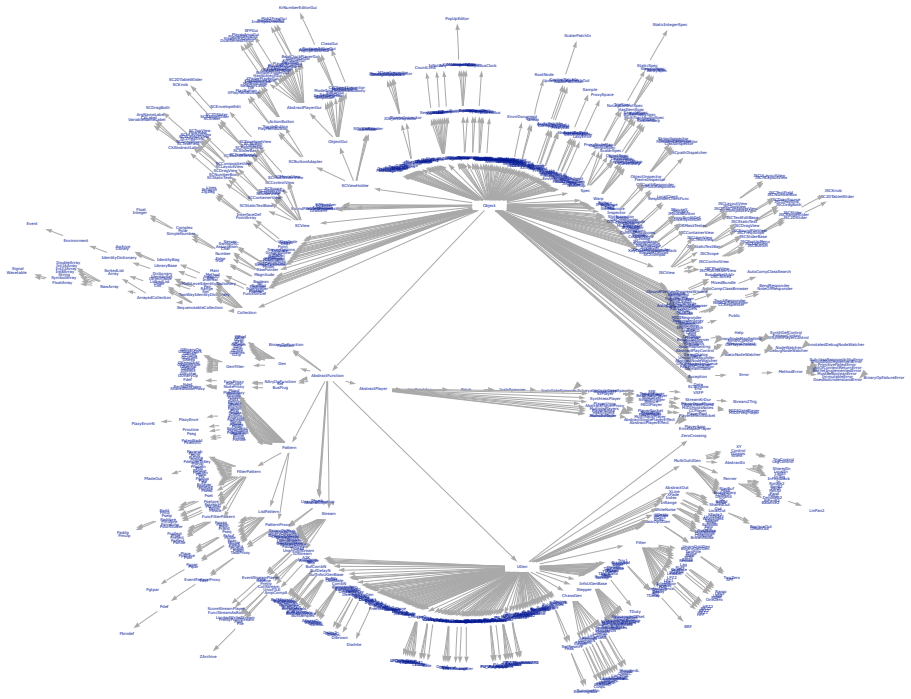


Fig. 2.4 Grafo radiale della struttura delle classi di SC, a partire da `Object`.

Il linguaggio SC è scritto per la maggior parte nello stesso SC (con l'eccezione di un nucleo di primitive scritte in linguaggio C per ragioni di efficienza): il codice stesso del programma è trasparente al programmatore, nel senso che è scritto in un linguaggio comprensibile per chi parla SC. Questo ovviamente è molto diverso da capire esattamente cosa viene detto *attraverso* il linguaggio: ma in ogni caso sbirciando i sorgenti si possono recuperare molte informazioni interessanti. È questo fatto che permette a SC un grande potere di introspezione. Dopo aver selezionato `Array` attraverso il menu *Language* è possibile accedere alla definizione della classe Array attraverso `Look Up Implementations for Cursor`.

```
Array[slot] : ArrayedCollection {

    *with { arg ... args;
        // return an array of the arguments given
        // cool! the interpreter does it for me..
        ^args
    }
    reverse {
        _ArrayReverse
        ^this.primitiveFailed
    }
    scramble {
        _ArrayScramble
        ^this.primitiveFailed
    }
    mirror {
        _ArrayMirror
        ^this.primitiveFailed
    }
    mirror1 {
        _ArrayMirror1
        ^this.primitiveFailed
    }
// etc
    sputter { arg probability=0.25, maxlen = 100;
        var i=0;
        var list = Array.new;
        var size = this.size;
        probability = 1.0 - probability;
        while { (i < size) and: { list.size < maxlen }}{
            list = list.add(this[i]);
            if (probability.coin) { i = i + 1; }
        };
        ^list
    }

// etc
}
```

Senza addentrarsi nel merito si noti come la prima riga (1) definisca la classe `Array` come una sottoclasse di `ArrayedCollection`(`Array[slot] : ArrayedCollection`). Fa seguito (da 3 in poi) la lista dei metodi implementati per la classe (`width, reverse, scramble`, ognuno chiuso tra una coppia di graffe).

Un modo agevole per ottenere una lista dei metodi implementati consiste nello sfruttare il potere di introspezione di SC[8]. SC fornisce molti metodi per conoscere informazioni relativi al suo stato interno. I metodi `dumpInterface`, `dumpFullInterface`, `dumpMethodList` visualizzano sulla post window informazioni sui metodi implementati per l'interfaccia di una classe. In particolare:

- `dumpInterface`: stampa tutti i metodi definiti per la classe;
- `dumpFullInterface`: come prima, ma include anche i metodi ereditati dalle superclassi della classe;
- `dumpMethodList`: come il precedente, ma con i metodi in ordine alfabetico e l'indicazione della classe da cui sono ereditati.

Ad esempio quanto segue riporta il risultato della valutazione di `Array.dumpInterface`. Le liste fornite dagli altri due metodi proposti sono decisamente più lunghe.

[8] Si tratta di una abbreviazione di comodo della scrittura esplicita. Il linguaggio SuperCollider prevede molte di queste abbreviazioni (dette syntax sugar, "zucchero sintattico") che permettono un guadagno di espressività, ma introducono potenzialmente una certa confusione nel neofita.

```
Array.dumpInterface
   reverse (  )
   scramble (  )
   mirror (  )
   mirror1 (  )
   mirror2 (  )
   stutter ( n )
   rotate ( n )
   pyramid ( patternType )
   pyramidg ( patternType )
   sputter ( probability, maxlen )
   lace ( length )
   permute ( nthPermutation )
   allTuples ( maxTuples )
   wrapExtend ( length )
   foldExtend ( length )
   clipExtend ( length )
   slide ( windowLength, stepSize )
   containsSeqColl (  )
   flop (  )
   multiChannelExpand (  )
   envirPairs (  )
   shift ( n )
   source (  )
   asUGenInput (  )
   isValidUGenInput (  )
   numChannels (  )
   poll ( interval, label )
   envAt ( time )
   atIdentityHash ( argKey )
   atIdentityHashInPairs ( argKey )
   asSpec (  )
   fork ( join, clock, quant, stackSize )
   madd ( mul, add )
   asRawOSC (  )
   printOn ( stream )
   storeOn ( stream )
   prUnarchive ( slotArray )
   jscope ( name, bufsize, zoom )
   scope ( name, bufsize, zoom )
Array
```

Attraverso la stessa procedura discussa per accedere alla definizione delle classi (*Language* →`Look Up Implementations for Cursor`), è possibile a partire da un metodo risalire alle classi che lo implementano.

L'esempio seguente, che potrebbe essere una sessione con l'interprete e si legge dall'alto in basso, valutando riga per riga, permette di avanzare nella discussione sui metodi.

```
z = [1,2,3,4] ;
z.reverse ;
z ;
z = z.reverse ;
z ;
z.mirror ;
z ;
z.reverse.mirror.mirror ;
```

Quanto risulta sulla post window in conseguenza della valutazione riga per riga è:

```
[ 1, 2, 3, 4 ]
[ 4, 3, 2, 1 ]
[ 1, 2, 3, 4 ]
[ 4, 3, 2, 1 ]
[ 4, 3, 2, 1 ]
[ 4, 3, 2, 1, 2, 3, 4 ]
[ 4, 3, 2, 1 ]
[ 1, 2, 3, 4, 3, 2, 1, 2, 3, 4, 3, 2, 1 ]
```

Intanto, un modo più agevole per creare un array consiste semplicemente nello scrivere l'array in questione tra parentesi quadre (secondo una notazione molto diffusa). Ad esempio, `z = [1,2,3,4]` assegna a `z` l'array `[1,2,3,4]` (1). Nella sessione con l'interprete dell'esempio viene valutato il codice `z = [1,2,3,4]` (codice, 1): SC restituisce l'array `[ 1, 2, 3, 4 ]` (post window, 2) e lo assegna a `z`. L'ultimo oggetto restituito viene stampato da SC sullo schermo come risultato del processo di interpretazione.

Come si è visto nella definizione di classe, uno dei metodi che la classe `Array` prevede è `reverse`: intuibilmente, il metodo prende l'array e ne inverte l'ordine

degli elementi. Nel momento in cui si passa il messaggio `reverse` a `z` che ne diventa il *ricevente* (3), `z` cerca il metodo `reverse` tra quelli che sono definiti nella sua classe, e si comporta conseguentemente. Nel caso in questione, come l'operazione di inversione venga svolta da SC non è interessante in questa sede (ne è molto interessante di per sé): in ogni caso, se si vede la definizione del metodo (nella classe Array, supra, 8-11), si nota come il metodo chiami una riga misteriosa, `_ArrayReverse` (supra, 9): la presenza del segno _ indica che l'operazione di inversione è allora realizzata da una primitiva di SC, scritta in linguaggio C e non in SC. Nella stessa classe, al contrario il metodo `sputter` (classe Array, righe 25-35) è scritto completamente in SC. I metodi restituiscono delle entità come risultato delle operazioni svolte: queste entità sono oggetti a tutti gli effetti. Ad esempio, `z.reverse` restituisce un nuovo array, invertito rispetto a `z`. Nella riga 2 del codice, `z.reverse` chiede a `z` di svolgere le operazioni previste da `reverse`: il risultato è `[ 4, 3, 2, 1 ]`, che non è assegnato ad alcuna variabile (post, 2). Se infatti si chiama `z` (codice, 3), si ottiene `[ 1, 2, 3, 4 ]` (post, 3). Per assegnare a `z` il risultato della computazione effettuata da `reverse` è necessario riassegnare a `z` il valore calcolato dal metodo attraverso `z = z.reverse` (codice, 4). Chiamando `z` (codice, 5) si ottiene il suo valore: questa volta è l'array `z` invertito (post, 5). Il metodo `mirror` genera invece un nuovo array a partire da quello a cui il messaggio è passato, simmetrico rispetto al centro (palindromo): `z.mirror` (codice, 6) restituisce `[ 4, 3, 2, 1, 2, 3, 4 ]` (post, 6), senza assegnarlo a `z`. L'ultima riga di codice (8) mette in luce un aspetto importante di SC: il cosiddetto concatenamento dei messaggi ("message chaining"). Sul risultato di `z.reverse` viene calcolato `mirror`, su questo secondo risultato viene calcolato `mirror` di nuovo (post, 8). Questi sono i passaggi a partire da `z = [ 4, 3, 2, 1 ]` (valore iniziale più tre messaggi):

$$[4,3,2,1] \rightarrow [1,2,3,4] \rightarrow [1,2,3,4,3,2,1] \rightarrow [1,2,3,4,3,2,1,2,3,4,3,2,1]$$

Sebbene permetta di scrivere codice in forma estremamente economica, il concatenamento dei messaggi va usato con cautela perché rischia di rendere di difficile lettura il codice. Il prossimo esempio riporta due espressioni che sfruttano il concatenamento.

```
1 [1,2,3,7,5].reverse.reverse ;
2 [1,2,3,7,5].class.superclass.subclasses[2].newClear(3) ;
```

Il primo caso non presenta alcuna peculiarità rispetto a quanto già visto, ma è inserito per contrasto rispetto al secondo, i cui effetti sono molto diversi. A partire da `[1,2,3,7,5]` il primo messaggio `class` restituisce la classe; su questa si può chiamare il metodo `superclass` che restituisce una classe; da quest'ultima attraverso `subclasses` si ottiene un array che contiene classi (un array generico è tipicamente agnostico rispetto a ciò che contiene). La notazione `[2]` non è ancora stata introdotta: il numero `2` rappresenta un indice (una posizione) nell'array. In sostanza, il metodo `[n]` permette di accedere ad un elemento $n-1$ dell'array. Così, `[ class Polynomial, class RawArray, class Array ][2]` restituirà `class Array`. Su quest'elemento (la classe `Array`) è allora possibile chiamare un metodo costruttore che generi un nuovo array vuoto (`nil`) ma di tre posti `[nil, nil, nil]`.

Una rappresentazione schematica di quanto avviene nei due casi è in Figura **2.5**.

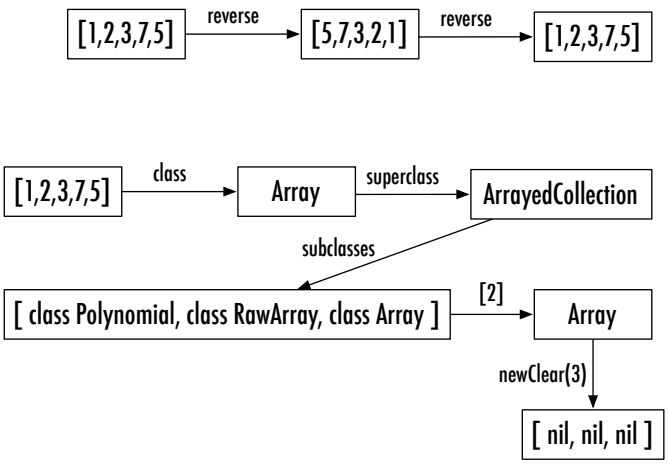


Fig. 2.5 Due esempi di concatenamento.

2.5 I metodi di tipo `post` e `dump`

Si è detto che tutti i metodi restituiscono un oggetto. A scanso di equivoci va ricordato il comportamento dei metodi che permettono di ottenere informazioni attraverso la post window. Esempi già visti sono `dumpClassSubtree` e

`dumpSubclassList`, e `dumpInterface`, `dumpFullInterface`, `dumpMethodList`. Il metodo più usato per ottenere generiche informazioni su quanto sta avvenendo è `postln`, che stampa una stringa di informazione relativa all'oggetto su cui è chiamato (e va a capo). Ad esempio si consideri il codice `Array.postln`. Una volta valutata, l'espressione produce sulla post window:

```
Array
Array
```

Quando viene chiamato il metodo `postln` su `Array`, SC esegue il codice il quale prevede di stampare informazioni su `Array`: in questo caso, essendo `Array` una classe, viene semplicemente stampato il nome della classe, `Array` appunto (1). Infine SC stampa sempre sullo schermo un'informazione sull'ultimo oggetto su cui è stato invocato un metodo: in sostanza SC chiama sull'ultimo oggetto proprio `postln`. E infatti si ottiene di nuovo `Array` (2). Se in questo caso l'utilità di `postln` è virtualmente nulla, si consideri invece il caso seguente:

```
z = [ 4, 3, 2, 1 ] ;
z.postln.reverse.postln.mirror.postln.mirror
z.postln.reverse.postln.mirror.postln.mirror.postln
```

La valutazione delle tre espressioni produce sulla post window i tre blocchi a stampa seguenti:

```
1  [ 4, 3, 2, 1 ]

3  [ 4, 3, 2, 1 ]
4  [ 1, 2, 3, 4 ]
5  [ 1, 2, 3, 4, 3, 2, 1 ]
6  [ 1, 2, 3, 4, 3, 2, 1, 2, 3, 4, 3, 2, 1 ]

8  [ 4, 3, 2, 1 ]
9  [ 1, 2, 3, 4 ]
10 [ 1, 2, 3, 4, 3, 2, 1 ]
11 [ 1, 2, 3, 4, 3, 2, 1, 2, 3, 4, 3, 2, 1 ]
12 [ 1, 2, 3, 4, 3, 2, 1, 2, 3, 4, 3, 2, 1 ]
```

A `z` è assegnato l'array `[ 4, 3, 2, 1 ]`. Viene quindi chiamato il concatenamento di metodi `reverse.mirror.mirror`, senonché dopo ogni metodo è concatenato anche un messaggio `postln`. In sostanza in questo caso `postln` permette di vedere sullo schermo il risultato intermedio restituito da ognuno dei metodi invocati. Si noti come sia inutile (nella circostanza) concatenare di nuovo `postln` dopo l'ultimo `mirror` (come avviene nel codice a riga 3), visto che per definizione SC stampa il risultato dell'ultima computazione (ciò che `mirror` restituisce). Infatti, le righe 11 e 12 riportano lo stesso risultato.

Ci si potrebbe aspettare che, poiché `postln` serve a stampare sullo schermo, ciò che questo metodo restituisce sia un oggetto di tipo stringa, un insieme di caratteri. Ma fortunatamente non è così. Infatti, `postln`:

- *stampa* una stringa sullo schermo
- *restituisce* l'oggetto su cui è chiamato il metodo

Le due cose sono completamente diverse. La prima è un comportamento che è indipendente dalla computazione, la seconda invece riguarda il flusso della computazione, perché l'oggetto restituito è disponibile per ulteriori elaborazioni. Da quest'ultimo punto di vista (che è ciò che conta per il concatenamento dei messaggi), `postln` è totalmente *trasparente* rispetto all'oggetto su cui è chiamato. Questo comportamento è assolutamente cruciale nella fase di correzione degli errori del codice ("debug"), perché permette di concatenare messaggi di stampa per verificare i comportamenti dei metodi invocati, ma senza per questo interferire con il processo. Se infatti il metodo `postln` restituisse una stringa,

allora in `z.postln.reverse` il messaggio `reverse` sarebbe ricevuto da un oggetto di tipo stringa e non da un oggetto array, come nell'esempio seguente, in cui reverse è chiamato su una stringa:

```
z = [ 4, 3, 2, 1 ] ;
z.postln.reverse.postln.mirror.postln.mirror
z.postln.reverse.postln.mirror.postln.mirror.postln
```

Il risultato sarebbe allora:

```
] 4 ,3 ,2 ,1 [
```

In altri termini, l'inversione avviene sui caratteri che compongono la stringa (si veda dopo per una spiegazione dettagliata). Questo tipo di comportamento vale tipicamente per tutti i metodi di stampa e di introspezione. I metodi di stampa a schermo molti: ad esempio varianti di `postln` sono `post`, `postc`, `postcln`. Per l'introspezione, si osservino gli esempi di `Collection.dumpClassSubtree`, `Collection.dumpSubclassList`, `Array.dumpInterface`. In tutti e tre i casi l'ultima riga stampa l'oggetto che il metodo restituisce: si noti come venga restituita la classe, secondo quanto stampato nell'ultima riga delle rispettiva schermate (`Collection`, `Collection`, `Array`).

2.6 Numeri

L'interprete di SC può essere utilizzato come calcolatore. Ad esempio, si consideri questa sessione interattiva:

```
2.3*2 ;
4/3 ;
4**3 ;
4+2*3 ;
```

A cui l'interprete sulla post window risponde con:

```
4.6
1.3333333333333
64
18
```

Due cose sono di rilievo. La prima, molto (molto) importante, concerne l'ordine di esecuzione. Se si osserva `4+2*3` si nota come, a differenza della convenzione matematica standard, non ci sia gerarchia tra gli operatori: ad esempio, la moltiplicazione non viene valutata prima dell'addizione. Nella riga prima viene valutato `4+2`, quindi `*3` che viene riferito al risultato dell'operazione precedente ($4 + 2 = 6 \times 3 = 18$). L'ordine di esecuzione può essere specificato con l'uso delle parentesi, così: `4+(2*3)`. Il secondo aspetto che dovrebbe stupire il lettore è che la sintassi utilizzata contraddice il presupposto per cui in SC ogni cosa è un oggetto dotato di un'interfaccia, per cui ogni operazione dovrebbe seguire il modello generale `oggetto.metodo`. Qui in effetti SC fa un'eccezione, almeno per le quattro operazioni, che possono essere scritte in modo più intuitivo nella consueta forma funzionale. Si noti che si tratta soltanto di una convenzione di notazione. Ciò non toglie che i numeri (interi, a virgola mobile, etc.) siano oggetti a tutti gli effetti. Se si chiama il metodo `class` su un intero, ad esempio `5` (1), si ottiene la classe a cui appartiene in quanto istanza: `Integer`. Si può allora chiamare su `Integer` il metodo `superclasses` che restituisce un array (attenzione, questa volta il metodo *restituisce un array*) contenente tutte le superclassi fino a `Object`. Valutato riga per riga, il codice:

```
5.class ;
Integer.superclasses ;
```

restituisce:

```
Integer
[ class SimpleNumber, class Number, class Magnitude, class Object ]
```

Intuibilmente, a parte `Object`, che è superclasse di tutto, `Magnitude` è la classe che più in generale si occupa di grandezze (tra cui i numeri). Con `Magnitude.allSubclasses` si ottiene allora:

```
[ class Association, class Number, class Char, class Polar, class Complex,
class SimpleNumber, class Float, class Integer ]
```

Con `Magnitude.dumpClassSubtree` si accede ad una rappresentazione ad albero delle sottoclassi di `Magnitude`: tutte le classi che si occupano di grandezze: `Integer` –i numeri interi– è vicino a `Float` –i numeri a virgola mobile–, poiché sono due sottoclassi di `SimpleNumber`. Quest'ultima classe fa parte del più vasto insieme delle sottoclassi di `Number`, i numeri in generale, compresi quelli polari e quelli complessi (`Polar`, `Complex`, che qui non interessano).

```
Magnitude.dumpClassSubtree
Magnitude
[
  Association
  Number
  [
    Polar
    Complex
    SimpleNumber
      [ Float Integer ]
  ]
  Char
]
Magnitude
```

In quanto oggetto (cioè, propriamente, in quanto istanza indiretta di `Object`), è possibile inviare ai numeri, il `3` ad esempio, il messaggio `postln`, il quale stampa il numero e restituisce il numero stesso. Dunque il codice:

```
3.postln ;
3.postln * 4.postln ;
```

restituisce sulla post window:

```
3
3
3
4
12
```

Dove (1) e (2) costituiscono l'output della riga di codice (1), e (3), (4), (5) della riga di codice (2). Si ricordi quanto osservato sulla chiamata da parte dell'interprete di `postln` sull'ultima espressione del codice valutato e sulla trasparenza del metodo.

Per numerose operazioni matematiche è disponibile una doppia notazione, funzionale e ad oggetti[9].

```
1  sqrt(2) ;
2  2.sqrt ;

4  4**2 ;
5  4.pow(2) ;
6  4.squared ;

8  4**3 ;
9  4.pow(3) ;
10 4.cubed ;
```

Ad esempio `sqrt(2)` (1) chiede di eseguire la radice quadrata di 2: in notazione ad oggetti si tratta di invocare su `2` il metodo `sqrt` (2), che restituisce il risultato dell'operazione radice quadrata applicata all'oggetto su cui è chiamato. Analogamente, l'elevamento a potenza può essere scritto funzionalmente come `4**2` (4), oppure come `4.pow(2)` (5): si chiama il metodo `pow` con argomento 2 sull'oggetto `4`. Ovvero, tradotto in lingua naturale: "oggetto 4, èlevati a potenza con esponente 2". Ancora, una terza opzione (6) consiste nell'utilizzare un metodo dedicato per l'elevante al quadrato, `squared`. Lo stesso vale per l'elevamento al cubo (8-10).

Dopo questa rapida introduzione alla programmazione ad oggetti, diventa opportuno affrontare nel dettaglio la sintassi di SC, così da scrivere un programma vero e proprio.

[9] Intuibilmente, sono altri casi di "syntax sugar".

3 Sintassi: elementi fondamentali

Come in ogni linguaggio, per parlare in SuperCollider è necessario seguire un insieme di regole. Come in tutti i linguaggi di programmazione, queste regole sono del tutto inflessibili e inderogabili. In SuperCollider, un enunciato o è sintatticamente corretto o è incomprensibile all'interprete, che lo segnalerà all'utente. Questo aspetto non è esattamente amichevole per chi non è abituato ai linguaggi di programmazione, e si trova così costretto ad una precisione di scrittura decisamente poco "analogica". Tuttavia, ci sono due aspetti positivi nello scrivere codice, interrelati tra loro. Il primo è lo sforzo analitico necessario, che allena ad un'analisi precisa del problema che si vuole risolvere in modo da poterlo formalizzare linguisticamente. Il secondo concerne una forma di consapevolezza specifica: salvo "bachi" nel linguaggio (rarissimi sebbene possibili), il mantra del programmatore è: "Se qualcosa non ti funziona, è colpa tua".

3.1 Parentesi tonde

Negli esempi di codice SC si troveranno spesso le parentesi tonde, `()`, utilizzate come delimitatori. Le parentesi tonde non sono esplicitamente previste a tal fine nella sintassi di SC. Tuttavia è una convenzione stabilizzata (basta vedere gli help files) che la loro presenza indichi un pezzo di codice che deve essere valutato tutto insieme (ovvero, selezione di tutte le linee e valutazione). Va ricordato che quando si scrive o si apre un documento, la valutazione del codice è in carico all'utente. Nell'IDE il doppio-click dopo una parentesi tonda di apertura permette così la selezione di tutto il blocco di codice fino alla

successiva parentesi tonda di chiusura: le parentesi agevolano perciò di molto l'interazione dell'utente con l'interprete. Le parentesi tonde sono anche un modo per organizzare un codice più complesso in blocchi di funzionamento separato: ad esempio un blocco che deve essere valutato in fase di inizializzazione, un altro che determina una certa interazione con un processo in corso (audio o GUI), e così via.

3.2 Espressioni

Un'espressione in SC è un enunciato finito e autonomo del linguaggio: una frase conclusa, si potrebbe dire. Le espressioni in SC sono delimitate dal ;. Ogni blocco di codice chiuso da un ; è dunque un'espressione di SC. Quando si valuta il codice SC nell'interprete, se il codice è composto da una riga sola è possibile omettere il ;. L'interprete infatti considererà allora tutto il blocco come una singola espressione (cosa che in effetti è).

```
1 a = [1,2,3]
2 a = [1,2,3] ;
```

Valutate riga per riga, le due espressioni precedenti sono equivalenti. In generale, meglio prendere l'abitudine di mettere il ; anche quando si sperimenta interattivamente con l'interprete riga per riga. Quando si valutano più linee di codice la presenza del ; è l'unica informazione disponibile all'interprete per sapere dove finisce un'espressione e ne inizia un'altra. In sostanza, quando si richiede all'interprete di valutare il codice, questi inizierà a passare in scansione carattere per carattere il testo, e determinerà la chiusura di un'espressione in funzione del ;. Se invece si valuta il codice tutto insieme, come potrebbe essere descritto dall'esempio seguente (si noti anche la delimitazione attraverso delle parentesi):

```
(
a = [1,2,3]
a = [1,2,3] ;
)
```

al di là dell'intenzione dell'utente di scrivere due espressioni, di fatto ce n'è una sola per l'interprete: essendo insensata, l'interprete segnalerà un errore e bloccherà l'esecuzione.

Nell'esempio seguente, le due espressioni sono uguali poiché l'a capo non è rilevante per SC (questo permette di utilizzare l'a capo per migliorare la leggibilità del codice, o di peggiorarla ovviamente, come succede nell'esempio).

```
(
a = [1,2,3]
)


(
a = [    1,
         2,
         3    ]
)
```

Di nuovo, si noti come in questo caso l'assenza del punto in virgola anche nella seconda versione multilinea non crei problemi. In assenza del ; SC considera un'espressione tutto ciò che è selezionato: poiché il codice selezionato è effettivamente un'unica espressione "sensata", l'interprete non segnala errori.

L'ordine delle espressioni è l'ordine di esecuzione delle stesse. In altri termini, l'interprete passa in scansione in codice, e quando trova un terminatore di espressione (; o la fine della selezione) esegue l'espressione; quindi reinizia la

scansione da lì in avanti fino a quando non ha ottenuto una nuova espressione da eseguire.

3.3 Commenti

Un "commento" è una parte di codice che l'interprete non tiene in considerazione. Quando l'interprete arriva ad un segnalatore di commento salta fino al segnalatore di fine commento e di lì riprende il normale processo esecutivo. I commenti sono perciò indicazioni meta-testuali che si rivelano molto utili per rendere il codice leggibile, fino alla cosiddetta "autodocumentazione del codice", in cui il codice esibisce in varia forma anche indicazioni sul suo uso. In SC i commenti sono di due tipi:

a. `//` indica un commento che occupa una riga o la parte terminale di essa. Quando l'interprete incontra `//` salta alla riga successiva;
b. la coppia `/* ...*/` delimita un commento multilinea: tutto ciò che è incluso tra i due elementi `/*` e `*/`, anche se occupa più linee, è ignorato dall'interprete.

L'esempio seguente dimostra l'uso dei commenti, in modo abbondante.

```
/*
%%%%%% DOCUMENTAZIONE ABBONDANTE %%%%%%

Ci sono 3 modi per elevare alla 3
   - n**3
   - n.pow(3)
   - n.squared

%%%%%% FINE DOCUMENTAZIONE ABBONDANTE %
*/

// primo modo
2**3 // fa 8
```

3.4 Stringhe

Una sequenza di caratteri delimitata da doppi apici è una "stringa". Le stringhe possono occupare più linee (anche l'a capo in fatti è propriamente un carattere). Si noterà che una stringa è una sequenza ordinata di elementi proprio come un array. Infatti, la classe `String` è una sottoclasse di `RawArray`: le stringhe sono cioè sequenze di oggetti a cui è possibile accedere. La valutazione riga per riga del codice seguente:

```
t = "stringa"
t[0]
t[1]
t.size
```

produce sulla post window:

```
stringa
s
t
7
```

Così, `t[0]` chiede il primo elemento dell'array `"stringa"`, ovvero `s`, e così via.

È nella classe `String` che vengono definiti i metodi `post` e affini. Quando si invia ad un oggetto il messaggio di `post`, SC tipicamente chiede all'oggetto una sua rappresentazione in stringa, e sulla stringa richiama il metodo `post`. Ad esempio il metodo di concatenazione di stringhe `++` vale anche se gli oggetti concatenati alla prima stringa non sono stringhe: `++` chiede internamente a tutti

gli oggetti una loro rappresentazione come stringa: `"grazie"++ 1000` equivale cioè a `"grazie"++ (1000.asString)`, e restituisce la stringa `"grazie1000"`.

3.5 Variabili

Una variabile è un segnaposto. Tutte le volte che si memorizza un dato lo si assegna ad una variabile. Infatti, se il dato è nella memoria, per potervi accedere, è necessario conoscere il suo indirizzo, la sua "etichetta" (come in un grande magazzino in cui si va a cercare un oggetto in base alla sua collocazione). Se il dato è memorizzato ma inaccessibile (come nel caso di un oggetto sperso in un magazzino), allora non si può usare ed è soltanto uno spreco di spazio. La teoria delle variabili è un ambito molto complesso nella scienza della computazione. Ad esempio, una aspetto importante può concernere la tipizzazione delle variabili. Nei linguaggi "tipizzati" (ad esempio C), l'utente dichiara che userà quella etichetta (la variabile) per contenere solo ed esclusivamente un certo tipo di oggetto (ad esempio, un numero intero), e la variabile non potrà essere utilizzata per oggetti diversi (ad esempio, una stringa). In questo caso, prima di usare una variabile se ne dichiara l'esistenza e se ne specifica il tipo. I linguaggi non tipizzati non richiedono all'utente di specificare il tipo, che viene inferito in vario modo (ad esempio, in funzione dell'assegnazione del valore alla variabile). Alcuni linguaggi (ad esempio Python) non richiedono neppure la dichiarazione della variabile, che viene semplicemente usata. È l'interprete che inferisce che quella stringa è una variabile. La tipizzazione impone vincoli d'uso sulle variabili e maggiore scrittura del codice, ma assicura una chiara organizzazione dei dati. In assenza di tipizzazione, si lavora in maniera più rapida e snella, ma potenzialmente si può andare incontro a situazioni complicate, come quando si cambia il tipo di una variabile "in corsa" senza accorgersene.

In SC è necessario dichiarare le variabili che si intendono utilizzare, ma non il tipo. I nomi di variabili devono iniziare con un carattere alfabetico minuscolo e possono contenere caratteri alfanumerici (caratteri maiuscoli, numeri). La dichiarazione delle variabili richiede la parola riservata `var` (che dunque non può essere usata come nome di variabile). È possibile assegnare un valore ad una variabile mentre la si dichiara.

```
(
var prima, seconda;
var terza = 3;
var quarta;
)
```

Se si valuta l'esempio precedente si nota che l'interprete restituisce `nil`. Nel valutare una serie di espressioni, l'interprete restituisce sempre il valore dell'ultima: in questo caso quello della variabile `quarta`, a cui non è stato ancora assegnato alcun valore, come indicato da `nil`. Tra l'altro, `nil` è un'altra parola riservata che non può essere usata come nome di variabile. La dichiarazione delle variabili può anche occupare più righe, purché iniziali rispetto al codice che le usa, e consecutive. Altrimenti detto, il blocco di dichiarazione delle variabili deve necessariamente essere posto in apertura al programma in cui le stesse sono usate. In SC esistono due tipi di variabili, quelle *locali* e quelle *d'ambiente* ("environment variables"). Come discusso, un linguaggio interpretato offre interazione all'utente attraverso un ambiente. In SC, le variabili d'ambiente ("environment variable") sono variabili che valgono in tutto l'ambiente. Praticamente, valgono per tutta la sessione di lavoro con l'interprete. Un caso particolare è dato dalle lettere `a-z` che sono immediatamente riservate dall'interprete per le *variabili d'ambiente*. Si possono utilizzare (ad esempio in fase di testing) senza bisogno di dichiarazione. Finora gli esempi che hanno fatto uso di variabili hanno sempre fatto riferimento a variabili di questo tipo, ad esempio `a` o `z`, con cui era possibile interagire nell'ambiente. Per capire le differenze si consideri l'esempio seguente:

```
a = [1,2,3] ;
array = [1,2,3] ;
(
var array ;
array = [1,2,3]
)
```

```
[ 1, 2, 3 ]
ERROR: Variable 'array' not defined.
  in file 'selected text'
  line 1 char 17:

  array = [1,2,3] ;

----------------------------------
nil
[ 1, 2, 3 ]
```

Quando si valuta la riga (1), `a` è un nome valido e legale per una variabile ambientale che non deve essere dichiarata. L'interprete vi assegna allora l'array `[1,2,3]` (post, 1). Quando si valuta (2) l'interprete solleva un errore (post, 2-9), poiché riconoscendo un'assegnazione di valore a una variabile, rileva che la variabile locale in questione `array` non è stata dichiarata (`· ERROR: Variable 'array' not defined.`). Il problema si risolve dichiarando la variabile (codice, 3-6; post, 10). Si noti l'uso delle parentesi tonde ad indicare che le riga di codice vanno valutate tutte insieme. L'esistenza della variabile vale soltanto per il momento in cui viene valutato il codice. In altre parole, se si esegue ora l'espressione `array.postln`, si potrebbe assumere che ormai la variabile `array` sia dichiarata e dunque legale. Ma si ottiene di nuovo errore. Le variabili locali sono dunque utilizzabili esclusivamente all'interno di quei blocchi di codice che le dichiarano. Durante una sessione interattiva può essere desiderabile mantenere l'esistenza di variabili per poterle riutillizare in un secondo momento. Le variabili d'ambiente servono esattamente per questo. Come si è visto, i caratteri alfabetici sono per convenzione interna assegnabili senza dichiarazione. In più, ogni variabile il cui primo carattere sia ~ è una variabile d'ambiente. Una volta dichiarata in questo modo (senza essere preceduta da `var`):

```
~array = [1,2,3] ;
```

la variabile `~array` è persistente per tutta la sessione[1]. Sopratutto il neofita alla programmazione potrebbe chiedersi perché non usare esclusivamente variabili ambientali. La risposta è che esse servono per poter lavorare interattivamente con SC, per "conversare" –per così dire– con SC, ma non per scrivere codice in forma strutturata, ad esempio progetti di sistemi per live music. Infatti, in questi contesti le variabili ambientali sono molto pericolose per la programmazione, semplicemente perché sono sempre accessibili, e dunque scarsamente controllabili.

Riassumendo:

- un nome di variabile è alfanumerico e inizia sempre con un carattere alfabetico minuscolo.
- le variabili sono locali o ambientali;
- quelle locali vanno dichiarate in testa al programma e valgono solo nella sua interpretazione
- quelle ambientali valgono per tutta la sessione di lavoro dell'interprete
- i nomi delle variabili ambientali sono preceduto da ~
- sono variabili ambientali anche i singoli caratteri alfabetici (non preceduti da ~)

3.6 Simboli

Un simbolo è un nome che rappresenta qualcosa in modo unico. Può essere pensato come un identificativo assoluto. È cioè un nome che rappresenta univocamente un oggetto, un nome proprio. Si scrive tra apici semplici, o, nel caso la sequenza di caratteri non preveda spazi al suo interno, preceduto da un \. L'esempio seguente va eseguito riga per riga, con il relativo risultato sulla post window.

[1] Si discuterà più avanti il problema dello scoping, cioè dell'ambito di validità delle variabili.

```
1 a = \symbol  ;
2 b = 'sono un simbolo' ;
3 a.class ;
4 [a,b].class.post.name.post.class ;
5 \Symbol .post.class.post.name.post.class ;
```

```
1 symbol
2 sono un simbolo
3 Symbol
4 ArrayArraySymbol
5 SymbolSymbolSymbolSymbol
```

Le righe (1) e (2) assegnano alle variabili `a` e `b` due simboli (1-2). Quindi (3) chiede di stampare per verifica la classe di `a`: `Symbol`. Infine (4) si comporta in maniera un po' più esoterica. Intanto, usa `post`, che stampa ma non va a capo: dunque ogni stampa semplicemente segue la precedente. I due simboli sono inseriti in un array, classe restituita da `class`: di qui il primo `Array`. Viene chiesto quale il nome della classe (`name`). Infatti, una classe è un'entità, e ha un nome: intuitivamente la classe `Array` ha nome `Array`. Ma cos'è questo nome? È un simbolo che rappresenta la classe. La classe del *nome* `Array` (che nomina la *classe* `Array`) è `Symbol`. Figurativamente: il lettore concorderà di essere un entità concreta, la cui classe è `Homo Sapiens`. L'etichetta `Homo Sapiens` (il nome della classe) è un oggetto che appartiene a sua volta alla classe `Simbolo tassonomico`. L'ultimo esempio (5) segue lo stesso modello di (4) ma ingarbuglia il tutto con un po' di omonimie. La situazione per entrambi i casi è raffigurata in Figura **3.1**, la cui comprensione il lettore può assumere come esercizio.

Si noti la differenza con le stringhe. Una stringa è una sequenza di caratteri. Ad esempio, qui `a` e `b` sono due stringhe.

```
a = "symbol" ; b = "symbol" ;
```

Le due stringhe sono equivalenti (detto approssimativamente: "hanno lo stesso contenuto").

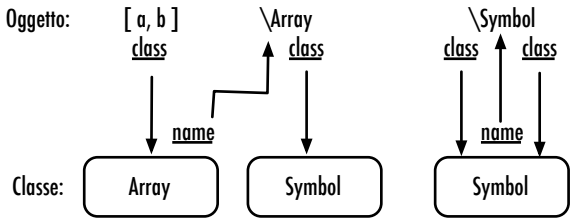


Fig. 3.1 Relazioni tra oggetti e classi rispetto a `Symbol`.

```
a == b        // == chiede: sono equivalenti?
// risposta sulla post window: true, vero
```

ma non sono lo stesso oggetto (due copie dello stesso libro restano due libri diversi) .

```
a === b     // === chiede invece: sono lo stesso oggetto?
// risposta sulla post window: false, falso
```

Invece, la relazione di identità è vera nel caso dei simboli:

```
a = \symbol ; b = 'symbol'    // scritture equivalenti
a == b; // post: true
a === b    // lo stesso oggetto? post: true
```

Qui, `a` e `b` sono due etichette per un unico oggetto.

Questi ultimi esempi permettono tra l'altro di introdurre altre due parole riservate, `true` e `false`, dedicate appunto a rappresentate i valori di verità. Le parole riservate (come anche `var` e altre che si vedranno) non possono ovviamente essere utilizzare come nomi di variabili, anche per struttura potrebbe

essere legali sintatticamente (sono alfanumeriche e iniziano per lettera minuscola).

3.7 Errori

Se si riconsidera l'esempio precedente in cui si utilizzava una variabile senza averla dichiarata:

```
ERROR: Variable 'array' not defined.
  in file 'selected text'
  line 1 char 17:

  array = [1,2,3] ;

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

si può vedere come l'interprete SC segnali gli errori. SC è molto fiscale: è un interprete decisamente poco caritatevole. Questo richiede una particolare attenzione ai principianti, che rischiano di impiegare un tempo interessante prima di riuscire a costruire un'espressione corretta. In più, la segnalazione degli errori è piuttosto laconica in SC: se nel caso precedente è decisamente chiara, in altri può esserlo meno. In particolare può essere poco agevole individuare dove si trova l'errore. Di solito la parte di codice segnalata da SC mentre riporta l'errore è il punto immediatamente successivo a dove si è verificato l'errore. Nel caso in esame, ciò che manca è una dichiarazione di variabile *prima* di `array = [1,2,3]`.

3.8 Funzioni

Le funzioni sono uno degli elementi meno intuitivi da comprendere per chi non arrivi da un background informatico. Si consideri la definizione fornita dall'help file:

> "A Function is an expression which defines operations to be performed when it is sent the 'value' message."

La definizione è precisa ed esaustiva. Una funzione è:

1. un'espressione
2. che definisce operazioni
3. che vengono effettuate *soltanto* nel momento in cui la funzione riceve il messaggio `value`. Una funzione è perciò un oggetto: implementa un metodo `value` con cui risponde al messaggio `value` (si provi `Function.dumpInterface`).

Una funzione può essere pensata come un oggetto (fisico) capace di un fare certe cose. Ad esempio, un frullatore. Nel momento in cui la si dichiara si dice a SC di *costruire l'oggetto, non di farlo funzionare*. A quel punto l'oggetto c'è: si tratta poi di metterlo in funzione *quando serve* attraverso il messaggio `value`.

Le definizioni delle funzioni sono racchiuse tra parentesi graffe `{}`. Le graffe rappresentano una sorta di involucro trasparente che racchiude l'oggetto, il cui contenuto è un insieme di espressioni che verranno valutate a chiamata. Il concetto di funzione è fondamentale nella programmazione strutturata ("organizzata") perché permette di applicare un principio di *incapsulamento*. Sequenze di espressioni che si intendono usare più volte possono allora essere definite una sola volta e associate a una variabile.

```
1 f = { 5 } ;
2 f.value ;
```

La funzione `f` è un oggetto che butta fuori a richiesta il valore 5. La definizione stocca l'oggetto funzione (1) il cui comportamento viene attivato a richiesta attraverso il messaggio `value` (2). Un uso più interessante delle funzioni, dimostrato dall'esempio seguente, prevede l'uso di "argomenti": gli argomenti possono essere pensati come gli input dell'oggetto. Gli argomenti vengono definiti attraverso la parola riservata `arg` cui fanno seguito i nomi degli argomenti separati da una `,` e delimitati da un `;`. Ad esempio la funzione `g` è definita come `{ arg input; input*2 }` (8): `g` accetta un argomento e restituisce il risultato dell'operazione sull'argomento. In particolare `g` restituisce il doppio del valore

`input` che gli viene dato in entrata. La funzione g è come un frullatore: si mette l'uovo `input` in entrata e il frullatore restituisce il risultato di `frullatore.sbatti(uovo)` in uscita[2].

```
1 g = { arg input; input*2 } ;
2 g.value(2) ;
3 g.value(134) ;
```

```
1 A Function
2 4
3 268
```

Infine la funzione `h = { arg a, b; (a.pow(2)+b.pow(2)).sqrt }` del prossimo esempio è un oggetto-modulo di calcolo che implementa il teorema di Pitagora: accetta in entrata i due cateti a e b e restituisce l'ipotenusa c, secondo la relazione $c = \sqrt{a^2 + b^2}$. Si noti tra l'altro come la radice quadrata sia un messaggio inviato all'intero risultante dal calcolo della parentesi.

```
1 h = { arg a, b; (a.pow(2)+b.pow(2)).sqrt } ;
2 c = h.value(4,3) ; // -> 5
```

In una seconda versione (qui di seguito) la definizione non cambia in sostanza ma permette di definire ulteriori aspetti.

[2] La somiglianza con la sintassi oggetto.metodo non è casuale.

```
1  (
2  h = { // calcola l'ipotenusa a partire dai cateti
3       arg cat1, cat2 ;
4       var hypo ;
5       hypo = (cat1.pow(2)+cat2.pow(2)).sqrt ;
6       "hypo: "++hypo } ;
7  )

9  h.value(4,3) ;

11 h.value(4,3).class ;
```

```
1 hypo: 5
2 String
```

Si notino alcuni passaggi:

- i commenti funzionano come al solito all'interno delle funzioni (2);
- gli argomenti vanno specificati per primi (3) ;
- i nomi degli argomenti seguono i criteri definiti per le variabili (3);
- a seguito degli argomenti è possibile aggiungere una dichiarazione di variabili (4). Nel corpo della funzione, specie se complessa, può essere utile avere a disposizione dei nomi di variabili. In questo caso, `hypo` è un nome significativo che permette di rendere più leggibile l'ultima riga, in cui vi si fa riferimento (`"hypo: "++hypo`). Per le variabili valgono le osservazioni già riportate;
- una funzione restituisce un unico valore (sia esso un numero, una stringa, un oggetto, un array, etc): il valore dell'*ultima espressione* definita nel corpo della funzione. L'output dell'ultima espressione è allora l'output della funzione. In particolare, in questa seconda versione, la funzione `h` restituisce una stringa composta da `"hypo: "` a cui è concatenato attraverso `++` il contenuto della variabile `hypo`. Ciò che la funzione restituisce in questo caso è dunque una stringa. La cosa appare chiara se si valutano le righe (9) e (10), il cui output sulla post window è riportato qui di seguito.

```
hypo: 5
String
```

Quest'ultimo punto ha conseguenze di rilievo. Se si ridefinisce `h` -nella sua prima versione- secondo quanto proposto dall'esempio seguente, se ne altera radicalmente il funzionamento.

```
h = { arg a, b; (a.pow(2)+b.pow(2)).sqrt ; a } ;
```

L'aggiunta dell'espressione `a` in coda alla definizione fa sì che la funzone `h` calcoli certamente l'ipotenusa ma restituisca in uscita `a` (cioè il primo argomento in entrata (1)).

In definitiva una funzione ha tre parti, tutte e tre opzionali, ma in ordine vincolante:

1. una dichiarazione degli argomenti (input)
2. una dichiarazione delle variabili (funzionamento interno)
3. un insieme di espressioni (funzionamento interno e output)

Una funzione che ha solo la dichiarazione degli argomenti è un oggetto che accetta entità in entrata, ma non fa nulla. Nell'esempio seguente, la funzione `i` accetta `a` in entrata, ma alla dichiarazione degli argomenti non fanno seguito espressioni: la funzione non fa nulla e restituisce `nil`. Al suo caso limite, è possibile anche una funzione che non ha nessuno dei tre componenti: nell'esempio, la funzione `l` restituisce sempre e solo `nil`.

```
i = {arg a;} ;
l = {} ;
```

La situazione può essere schematizzata come in figura **3.2**, dove le funzioni sono rappresentate come moduli, che possono essere dotati di entrate ed uscite. Il testo nell'ultima riga rappresenta il codice SC relativo a ogni diagramma.

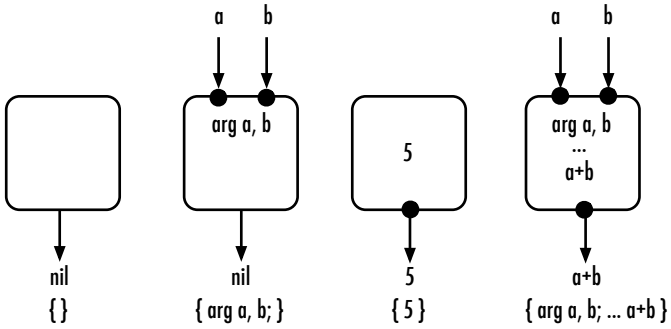


Fig. 3.2 Funzioni.

L'introduzione delle funzioni permette di affrontare il problema dell'ambito di visibilità delle variabili ("scope"). Ogni variabile "vale", è riconosciuta, è associata a quel valore, è manipolabile, all'interno di una certa regione di testo.

Nell'esempio seguente seguente, `func.value` restituisce `8` perché la variabile `val`, essendo dichiarata fuori della funzione `func`, vale anche al suo interno.

```
(
var val = 4 ;
var func = { val*2 } ;
func.value ;
)
```

Qui di seguito invece `func` restituisce sempre `8`, ma ciò dipende dalla dichiarazione di `val` all'interno di `func`.

```
(
var val = 4 ;
var func = { arg val = 4 ; val*2 } ;
func.value ;
)
```

Tant'è che l'esempio seguente solleva un errore perché a `val` (dichiarata dentro `func`) non è assegnato valore, dunque l'operazione richiesta è `nil*2` che è illegale.

```
(
var val = 4 ;
var func = { arg val ; val*2 } ;
func.value ;
)
```

In sostanza, la regola generale è che l'ambito di visibilità delle variabili procede dall'esterno all'interno. Una variabile è visibile fintanto che lo stesso nome non è dichiarato più internamente.

3.9 Classi, messaggi/metodi e keyword

Si è già visto come in SC le classi siano indicate da una sequenza di caratteri che inizia con una maiuscola. Se si valutano le due righe del codice seguente (si noti la sintassi colorata):

```
superCollider ;
SuperCollider ;
```

si ottiene infatti nei due casi:

```
ERROR: Variable 'superCollider' not defined.
  in file 'selected text'
  line 1 char 13:

  superCollider ;

----------------------------------
nil
ERROR: Class not defined.
  in file 'selected text'
  line 1 char 13:

  SuperCollider ;

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

Come già ricordato, un messaggio viene inviato a una classe a ad un oggetto con il `.`: rispettivamente per il tramite delle sintassi `Classe.metodo` e `oggetto.metodo`. I metodi possono in fondo essere pensati come funzioni definite per una classe o un oggetto: quando si invoca un metodo tramite un messaggio è come se si mandasse un messaggio `value` a una funzione. In più, anche i metodi possono avere argomenti che costituiscono i parametri di entrata. SC tipicamente prevede opportuni valori di default nel caso in cui il metodo richieda degli argomenti, così che in molti casi non è necessario specificare gli argomenti. L'uso delle keyword è utile perché consente di selezionare di quale argomento si vuole specificare il valore, lasciando agli altri i valori predefiniti. Laddove non si usino le keyword, l'unico criterio disponibile per SC per poter attribuire un valore ad un argomento è l'ordine in cui il valore si presenta nella lista degli argomenti. Ad esempio il metodo `plot` per gli oggetti `ArrayedCollection` prevede gli argomenti

```
plot(name, bounds, discrete, numChannels, minval, maxval, parent, labels)
```

Il metodo crea una finestra e vi disegna sotto forma di spezzata un oggetto di tipo `ArrayCollection`. L'argomento `name` definisce il titolo della finestra. Così, la finestra creata da `[1,2,3,1,2,3].plot("test")` ha come titolo "test". Il metodo consente anche di definire il numero dei canali `numChannels`. Se il numero è superiore a 1, `plot` assume che i primi n campioni siano i campioni

numero 1 dei canali $1...n$. Ad esempio, se i canali sono due, allora i primi due campioni sono i campioni numero 1 dei canali 1 e 2, e così via: `plot` disegna una finestra per canale[3]. Se si volesse specificare che `numChannels` deve essere pari a 2, senza keyword sarebbe necessario specificare anche tutti gli argomenti precedenti. Ad esempio:

```
1 [1,4,3,2].plot("test", Rect(200 , 140, 705, 410), false, 2) ;
```

Assai più agevolmente è possibile scrivere:

```
1 [1,4,3,2].plot(numChannels:2)  ;
```

Infine, l'uso delle keyword è in generale leggermente più costoso da un punto di vista computazionale ma rende il codice molto più leggibile.

3.10 Un esempio grafico

Un esempio di codice relativo alla creazione di un semplice elemento grafico permette di introdurre i fondamenti della sintassi di SC. Il codice presentato si propone di creare un elemento grafico a rotativo, la cui rotazione controlli in forma parametrica il colore dello sfondo.

SuperCollider è dotato di una ricca tavolozza di elementi grafici che permetto di costruire interfacce grafiche (GUI) sofisticate. Dunque, se è vero che l'interfaccia utente di SC è testuale in fase di implementazione dei propri progetti, non lo è necessariamente in fase di utilizzo del software, ad esempio nel

[3] L'utilità del metodo sta nel fatto che i segnali audio multicananle, rappresentabili attraverso un array, sono memorizzati in questa forma, "interallacciata". Se il segnale è stereo, `plot` con `numChannels:2` disegna la forma d'onda dei segnali sui due canali.

caso di performance live. Ovviamente, un'interfaccia grafica in SuperCollider va programmata. E il caso delle GUI è un classico della programmazione a oggetti, perché a ben vedere un elemento GUI si presta in maniera semplice ad essere definito come oggetto (è una entità dai chiari confini, con proprietà e comportamenti). La gestione GUI in SuperCollider avviene attraverso una libreria molto diffusa, Qt, che viene compilata insieme all'IDE. Molto semplicemente, è subito disponibile per essere utilizzata. Si noterà però che quando si fa uso di un GUI si utilizza non l'applicazione SuperCollider (IDE), ma invece sclang, a cui fanno riferimento (anche in termini di focus) le GUI. In termini di meccanismo di comunicazione/rappresentazione, gli oggetti GUI in Qt sono rappresentati in SC da classi, ad esempio `Window` è la classe SC che rappresenta un oggetto finestra in Qt. Se si ricorda che si tratta di un linguaggio, è come dire che `Finestra` è la rappresentazione concettuale e linguistica (un segno dotato di caratteristiche di espressione e di contenuto) di una categoria di oggetti del mondo (un tipo di serramento). Passando al programma:

```
(
/* accoppiamento di view e controller */

var window, knob, screen ; // dichiarazione delle variabili usate

// una finestra contenitore
window = Window.new("A knob", Rect.new(300,300, 150, 100)) ;
window.background = Color.black ;

// una manopola nella finestra, range: [0,1]
knob = Knob.new(window, Rect(50, 25, 50, 50)) ;
knob.value = 0.5 ;

// azione associata a knob
knob.action_({ arg me;
    var red, blue, green ;
    red = me.value ;
    green = red*0.5 ;
    blue = 0.25+(red*0.75) ;
    ["red, green, blue", red, green, blue].postln  ;
    window.background = Color(red, green, blue);
});

// non dimenticarmi
window.front ;
)
```

- **1**: il blocco di codice è racchiuso tra parentesi tonde (1 e 26);
- **3**: un commento multilinea è utilizzato a mo' di titolo (2) e sono presenti altri commenti che forniscono alcune informazioni sulle diverse parti del codice (ad esempio, 6). Commentare il codice è ovviamente opzionale ma è notoriamente buona prassi: permette di fornire informazioni generali sulla struttura e eventualmente dettagli sull'implementazione;
- **4**: a parte i commenti, il codice inizia con la dichiarazione delle tre variabili utilizzate. Sono appunto in testa al blocco di codice valutato;
- **7-8**: la prima cosa da fare è creare una finestra contenitore, cioè un oggetto di riferimento per tutti gli altri elementi grafici che verranno creati in seguito. È un approccio tipico nei sistemi di creazione di GUI. Alla variabile `window` viene assegnato un oggetto `Window`, generato attraverso il metodo costruttore `new`. Al metodo `new` vengono passati due argomenti (nuova finestra, con questi attributi): una stringa che indica il titolo visualizzato dalla finestra (`"A knob"` e un oggetto di tipo `Rect`. In altri termini, la dimensione della finestra, invece di essere descritta un insieme di parametri, è descritta da un oggetto rettangolo, cioè da un'istanza della classe `Rect`, dotato di suoi attributi, cioè , cioè `left, top width, height`. La posizione del rettangolo è data dall'angolo in alto a sinistra (argomenti `left` e `top`), mentre `width` e `height` specificano le dimensioni. Il rettangolo stabilisce allora che la finestra sarà di 150x100 pixel, il cui angolo superiore sinistro è nel pixel (300,300). Si noti che `Rect.new` restituisce un oggetto (`new` è un costruttore) ma *senza* assegnarlo a variabile. Infatti, da un lato non serve che l'oggetto abbia una identificabilità al di fuori della GUI finestra, dall'altro in realtà rimane accessibile a future manipolazioni, poiché viene memorizzato attraverso una proprietà della finestra, `bounds`. Se si valuta `window.bounds` si può notare come venga restituito proprio quel oggetto `Rect`. Attraverso lo stesso sistema è possibile modificare la stessa proprietà, ad esempio con questo codice: `w.bounds = Rect(100, 100, 1000, 600)`, che assegna a `window` un nuovo oggetto `Rect` (con dimensioni e posizione diverse). A parte il rettangolo dei confini (per così dire), tra gli attributi della finestra c'è il colore dello sfondo, accessibile attraverso l'invocazione di `background`. Il valore di `background` viene specificato un colore attraverso un oggetto `Color`. Anche i colori sono oggetti in SC e la classe `Color` prevede alcuni metodi predefiniti, che permettono di avere a disposizione i nomi più comuni dei colori: ad esempio il nero con

`Color.black`. In quest'ultimo caso, si noti che `black` è un costruttore che restituisce una istanza con certe particolarità;

- **10-12**: la costruzione di un elemento grafico a manopola segue un procedimento analogo a quanto avvenuto per la finestra-contenitore. Alla variabile `knob` viene assegnato un oggetto `Knob` (11). Il costruttore sembra uguale a quello di `Window`: senonché questa volta è necessario specificare a quale finestra-contenitore vada riferito l'oggetto `Knob`: la finestra-contenitore è `window`, e il rettangolo che segue prende come punto di riferimento non lo schermo, ma la finestra `window`. Dunque un rettangolo 50x50, la cui origine è nel pixel (30,30) *della finestra* `window`. Si noti anche che il gestore geometrico di Knob (sul modello di `Window`) è ottenuto con questo costrutto: `Rect(50, 25, 50, 50)`. Ad una classe segue una coppia di parentesi con argomenti. Dov'è finito il metodo costruttore? È un esempio di syntax sugar. Se dopo una classe si apre una parentesi con i valori degli argomenti, allora l'interprete sottintende che l'utente abbia omesso `new`, che è cioè il metodo predefinito. Quando SC vede una classe seguita da una coppia di parentesi contenenti dati assume che si sia invocato Class.new(argomenti). Il punto di partenza della manopola è `0.5` (12). Per default l'escursione di valori di un oggetto `Knob` varia tra 0.0 e 1.0 (escursione *normalizzata*): dunque attraverso l'attributo `knob.value = 0.5` si indica la metà. La ragione dell'escursione normalizzata sta nel fatto che in fase di costruzione non si può sapere a cosa servirà il rotativo (controllerà la frequenza di un segnale? 20-20.000 Hz; o una nota MIDI? 0-127). Si noti che la proprietà è impostata attraverso l'uso del = che è l'operatore di assegnazione. Nel caso di attributi di un oggetto è anche possibile un'altra sintassi. Infatti, il = è sinonimo del metodo *setter* rappresentato dal simbolo _, che esplicitamente assegna alla proprietà in questione il valore desiderato. In altri termini le due sintassi successive sono equivalenti.

```
1 knob.value = 0.5 ;
2 knob.value_(0.5) ;
```

La sintassi setter è perciò del tipo `oggetto.proprietà_(valore)`.
La parte successiva del codice definisce l'interazione con l'utente.

- **15-22**: un rotativo è evidentemente un "controller", un oggetto usato per controllare qualche altra cosa. All'oggetto `knob` è perciò possibile associare

un azione: è previsto per definizione che l'azione venga portata a termine tutte le volte che l'utente cambia il valore di `knob`, cioè muove la manopola. Una funzione rappresenta opportunamente questo tipo di situazione, poiché come si è visto è un oggetto che definisce un comportamento richiamabile di volta in volta e parametrizzato da un argomento. Il metodo `knob.action` chiede di attribuire a `knob` l'azione seguente, descritta attraverso una funzione: la funzione è tutto quanto è compreso tra parentesi graffe, 15-22. Quanto avviene è che, dietro le quinte, quando si muove la manopola alla funzione viene spedito un messaggio `value`. Il messaggio `value` chiede di calcolare la funzione per il valore della manopola, che è l'input della funzione: dietro le quinte cioè viene inviato alla funzione il messaggio `value(knob.value)`. In altre parole, la funzione risponde alla domanda "cosa bisogna fare quando la manopola `knob` si muove". Nella funzione l'input è descritto dall'argomento `me` (15): l'argomento, il cui nome è del tutto arbitrario (e scelto da chi scrive) serve per poter rappresentare in forma riflessa l'oggetto stesso all'interno della funzione. A cosa può servire questo riferimento interno? Ad esempio, a dire ad un oggetto grafico di modificare se stesso come risultato della propria azione.

- **16-21**: nell'esempio però, il comportamento previsto richiede di cambiare il colore di sfondo di `window`. Vengono dichiarate tre variabili (`red, green, blue`) (16). Esse identificano i tre componenti RGB del colore dello sfondo di `window`, che SC definisce nell'intervallo $[0, 1]$. A `red` viene assegnato il valore in entrata di `me` (17). A `green` e `blue` due valori proporzionali (ma in modo diverso) a `me`, in modo da definire un cambiamento continuo in funzione del valore di `me` nelle tre componenti cromatiche. Si tratta appunto di un'operazione di *mapping*: a un dominio di valori ($[0, 1]$) se ne fanno corrispondere altri tre, uno per ogni componente ($[[0, 1], [0, 0.5], [0.25, 1]]$). Quindi si dice di stampare su schermo un array composto da una stringa e dei tre valori (20): in questo caso SC stampa gli elementi dell'array sulla stessa riga opportunamente formattati. La stampa su schermo permette di capire come vengono calcolati i valori (ed è utile senz'altro in fase di debugging). Infine, all'attributo `background` di `window` viene assegnato un oggetto `Color`, a cui sono passate le tre componenti. Il costruttore di `Color` accetta cioè le tre componenti RGB in escursione [0,1] come definizione del colore da generare. È una delle molte abbreviazioni possibili in SC. Dunque, `Color(red, green, blue)` è equivalente in tutto e per tutto a `Color.new(red, green, blue)`.

Infine una considerazione importante.

- **25**: tutti i sistemi GUI distinguono tra creazione e visualizzazione. Un conto è creare gli oggetti GUI, un conto è dire che debbano essere visibili: questa distinzione permette di fare apparire/sparire elementi GUI sullo schermo senza necessariamente costruire e distruggere nuovi oggetti. Il metodo `front` rende `window` e gli elementi che da essa dipendono visibili: in caso d'omissione tutto funzionerebbe uguale, ma nulla sarebbe visualizzato sullo schermo.

3.11 Controlli di flusso

In SC il flusso della computazione segue l'ordine di lettura delle espressioni. I controlli di flusso sono quei costrutti sintattici che possono modificare quest'ordine di computazione. Ad esempio, un ciclo `for` ripete le istruzioni annidate al suo interno per un certo numero di volte, e quindi procede sequenzialmente da lì in avanti, mentre un condizionale `if` valuta una condizione rispetto alla quale il flusso di informazioni si biforca (se è vero / se è falso). I controlli di flusso sono illustrati nell'help file "Control structures", da cui sono tratti (con una piccola modifica) i tre esempi (rispettivamente `if`, `while` e `for`).

```
(
var a = 1, z;
z = if (a < 5, { 100 },{ 200 });
z.postln;
)



(
i = 0;
while ( { i < 5 }, { i = i + 1; [i, "boing"].postln });
)


for (3, 7, { arg i; i.postln });

forBy (0, 8, 2, { arg i; i.postln });
```

Nel primo caso è illustrato l'uso di `if`. La sintassi è:

```
if ( condizione da valutare, { funzione se è vero } , { funzione se è falso })
```

In altre parole la valutazione della condizione dà origine a una biforcazione a seconda che il risultato sia `true` oppure `false`. Passando all'esempio, la variabile a (che è dichiarata, dunque, si noti, è locale) vale 1. La condizione è `a < 5`. Se la condizione è vera, viene eseguita la funzione `{ 100 }`, che restituisce `100`, se è falsa viene eseguita la funzione `{ 200 }`, che restituisce `200`. Poiché la condizione è vera, viene restituito il valore 100, che viene assegnato a z: z vale 100.

Come è noto, la traduzione in italiano di `while` (in computer science) è "finché":

```
while ({ condizione è vera }, { funzione da eseguire } )
```

Nell'esempio , `i` vale 0. Finché `i` è inferiore a 5, viene chiamata la funzione seguente. La funzione incrementa `i` (altrimenti non si uscirebbe mai dal ciclo) ed esegue una stampa di un array che contiene `i` e la stringa `"boing"`.

Infine, il caso del ciclo `for`, che itera una funzione.

```
for (partenza, arrivo, { funzione } )
```

Nell'esempio La funzione viene ripetuta cinque volte $(3, ...7)$. Il valore viene passato alla funzione come suo argomento in modo che sia accessibile: la funzione infatti stampa `i` per ogni chiamata $(3, ...7)$. Si noti che il fatto che l'argomento si chiami `i` è del tutto arbitrario. Le due espressioni dell'esempio seguente dimostrano che è la posizione dell'argomento a specificarne la semantica (contatore), non il nome (arbitrario):

```
1 for (3, 7, { arg i; i.postln });
2 for (3, 7, { arg index; index.postln });
```

L'istruzione `ForBy` richiede un terzo parametro che specifica il passo:

```
forBy (partenza, arrivo, passo, { funzione } )
```

L'esempio è una variazione del precedente che stampa l'escursione $[0, 8]$ ogni 2. Esistono altre strutture di controllo. Qui vale la pena di introdurre `do`, che itera sugli elementi di una collezione. Si può scrivere così:

```
do ( collezione, funzione )
```

ma molto più tipicamente la si scrive come un metodo definito sulla collezione. Ovvero:

```
collezione.do({ funzione })
```

l'esempio è tratto dall'help file "Control-structures", con alcune piccole modifiche.

```
[ 101, 33, "abc", Array ].do({ arg item, i; [i, item].postln; });

5.do({ arg item; ("item"+item.asString).postln });

"you".do({ arg item; item.postln });
```

Se si valuta la prima riga si ottiene nella post window:

```
[ 0, 101]
[ 1, 33 ]
[ 2, abc ]
[ 3, class Array ]
[ 1, 2, abc, class Array ]
```

A scanso d'equivoci, l'ultima riga semplicemente restituisce l'array di partenza. Alla funzione vengono passati l'elemento su cui sta effettuando l'iterazione (`item`) e un contatore (`i`). Meglio ribadire: i nomi `item` e `i` sono totalmente arbitrari. È il loro posto che ne specifica la semantica. Ovvero:

```
[ 101, 33, "abc", Array ].do({ arg moby, dick; [dick, moby].postln; });
[ 0, 101 ]
[ 1, 33 ]
[ 2, abc ]
[ 3, class Array ]
[ 101, 33, abc, class Array ]
```

La funzione stampa un array che contiene il contatore `i` (colonna di sinistra delle prime quattro righe) e l'elemento `item` (colonna di destra). Il metodo `do` è definito anche sugli interi (n `volte.valuta la funzione`). Il funzionamento è illustrato nel secondo esempio. Se lo si esegue si ottiene:

```
item 0
item 1
item 2
item 3
item 4
5
```

L'ultima riga è l'intero su cui è chiamato il metodo. La funzione stampa una stringa costituita dal concatenamento di `"item"` e della rappresentazione sotto forma di stringa restituita dal metodo `asString` chiamato sul numero intero `item` ($0, \ldots 4$). Poiché la maggior parte dei cicli `for` iterano a partire da 0 e con passo 1, molto spesso si trovano scritti in SC attraverso `do`. La sintassi di do (oggetto.metodo) è più OOP. Infine, l'ultimo esempio dimostra semplicemente che ogni stringa è una collezione i cui elementi sono i singoli caratteri alfanumerici che compongo la stringa.

3.12 Ancora un esempio GUI

L'esempio seguente presenta il codice di creazione di una GUI per la scelta del colore, Simple Color Selector. Il controllo di elementi GUI è particolarmente

interessante per dimostrare alcuni aspetti della sintassi, e la visualizzazione del processo è di aiuto alla comprensione.

La GUI progettata si compone di tre cursori. Ogni cursore controlla uno dei tre parametri che definiscono il colore (il quarto, qui non considerato, è la trasparenza, `alpha`). Il colore viene visualizzato come sfondo della finestra. Inoltre, il valore ottenuto per ogni componente è visualizzato in forma numerica a lato del cursore relativo. La GUI permette di scegliere tra due tipici spazi colore. Come si è visto, il colore viene definito in un calcolatore attraverso il modello RGB. Tuttavia, questo modello è poco intuitivo rispetto alla percezione del colore: viene così anche utilizzato (tra gli altri) un modello che descrive il colore nei termini di tinta ("hue", circolarmente dal rosso al giallo al verde al blu al viola, al rosso), saturazione ("saturation", che esprimi la quantità di tinta, dall'assenza al massimo), brillantezza ("brightness" o "value", che indica la luminosità, dal bianco al nero). Il modello è perciò definito "HSB" o "HSV". Una codifica HSV può essere convertita con una formula in RGB (l'unico formato realmente implementato su un calcolatore). Così, in SC `Color` prevede un costruttore `hsv` che permette di definire un colore attraverso appunto tinta, saturazione, valore (sempre in escursione normalizzata $[0, 1]$). Nella GUI un pulsante permette di scegliere tra RGB e HSV. Chiaramente, esistono più modi di organizzare la GUI, e di nuovo più modi per implementarla in SC.

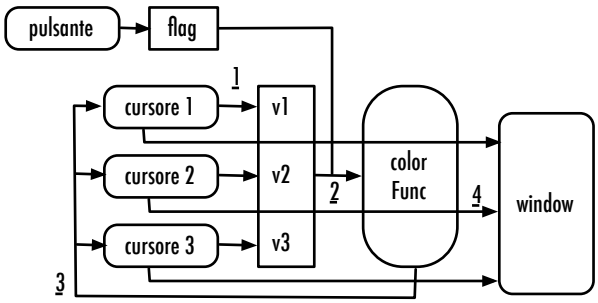


Fig. 3.3 Struttura di Simple Color Selector.

Un diagramma del progetto è rappresentato in Figura **3.3**. In Figura si riconoscono due dati: un flag, cioè una variabile che può avere valori discreti (qui due: nel codice, `rgb` o `hsv`, e un array di tre valori (in Figura, v1-v3) per la specificazione del colore. Un pulsante permette di scegliere quale dei due valori del flag è attivo (e quindi lo spazio colore selezionato). I tre cursori svolgono ognuno quattro azioni (numerate e sottolineate in Figura). Ognuno seleziona un valore

relativo nell'array (1), quindi chiama la funzione colore a cui passa l'array e il flag (2), la funzione restituisce un colore (3), e si procede al cambiamento dello sfondo (4). La funzione ha un ruolo importante perché è il nucleo computazione del tutto, che va mantenuto separato dall'interfaccia utente per garantire incapsulamento. Dunque, la funzione non farà riferimento a variabili esterne, ma riceverà le stesse come argomenti. Analogamente, non sarà la funzione a modificare lo sfondo, ma semplicemente restituirà il colore risultante.

Il programma risultante è il seguente:

```
(
/*
Simple Color Selector
RGB-HSV
*/

var window = Window("Color Selector", Rect(100, 100, 300, 270) ).front ;
var guiArr, step = 50 ;
var flag = \rgb , colorFunc ;
var colorArr = [0,0,0] ;

colorFunc = { arg flag, cls;
    var color, v1, v2, v3 ;
    # v1, v2, v3 = cls ;
    if(flag == \rgb ){
        color = Color(v1, v2, v3)
    }{
        color = Color.hsv(v1.min(0.999), v2, v3)
    } ;
    color ;
} ;

Button(window, Rect(10, 200, 100, 50))
.states_([["RGB", Color.white, Color.red], ["HSV", Color.white, Color.black]])
.action_({ arg me; if (me.value == 0) {flag = \rgb } {flag = \hsv } });

guiArr = Array.fill(3, { arg i ;
    [
        Slider(window, Rect(10, (step+10*i+10), 100, step)),
        StaticText(window, Rect(120, (step+10*i+10), 120, step))
    ]
}) ;

guiArr.do{|item, index|
    item[0].action_{|me|
        item[1].string_(me.value) ;
        colorArr[index] = me.value ;
        window.background_(colorFunc.value(flag, colorArr));
}} ;
)
```

- **7-10**: dichiarazione delle variabili. Si noti che `window` è immediatamente associata alla creazione della finestra che viene visualizzata. Altre variabili vengono inizializzate con valori "sensati";
- **12-21**: il blocco è dedicato alla definizione della funzione `colorFunc`. La funzione prende come argomenti in ingresso `flag` e `cls`. Il primo è il flag, il secondo l'array dei tre valori colore. La riga 14 introduce un'altra abbreviazione utile: `# v1, v2, v3 = cls` equivale a `v1 = cls[0]; v1 = cls[1]; v3 = cls[2]`. Il blocco condizionale (15) opera in relazione alla verifica del flag. Se `flag` è `rgb` (letteralmente, se `flag == rgb` è vero), allora a `color` viene assegnato un colore costruito secondo il modello RGB (default). Altrimenti (i casi sono solo due, RGB o HSV, dunque non ci si può sbagliare), gli stessi valori definiscono un colore costruito in riferimento a HSV. In quest'ultimo caso, si noti che per definizione (si veda l'help file di `Color.hsv`) l'argomento `hue` può valere al massimo 0.999. Se il primo cursore viene mosso a 1, allora ci sarà un problema. La soluzione è il metodo `min` definito sui numeri, che restituisce il minore tra il numero su cui è chiamato e il valore di soglia. Dunque, per tutti i valori minori di 0.999, restituirà gli stessi, mentre restituirà 0.999 se il numero è maggiore. Si noti che l'ultima espressione della funzione è semplicemente una invocazione di `color`, che verrà restituito in uscita. Infine, da notare la sintassi "zuccherina" nella definizione del condizionale, che elimina un po' di parentesi;
- **23-25**: viene costruito un pulsante. Si noti che non essendo più modificato, non viene associato a variabile (l'oggetto semplicemente funziona). In SC un pulsante si costruisce come le altre GUI. In più è dotato di una proprietà `states` che definisce attraverso un insieme di array ogni stato del pulsante (possono essere costruiti pulsanti a n stati). I parametri sono agevolmente inferibili dalla GUI in funzione. L'argomento `action` associa un'azione al pulsante. Il valore di un pulsante è l'indice dello stato: cioè, il pulsante vale 0 nel primo stato, 1 nel secondo, e così via. Diventa quindi possibile ragionare condizionalmente in funzione dello stato in cui si trova il pulsante stesso. Qui i valori sono solo due (si tratta di definire il valore del flag, attraverso un simbolo), e un `if` è sufficiente, che determina in funzione dello stato il valore del flag;
- **27-32**: si tratta ora di costruire i cursori e le etichette che ne visualizzano il valore. L'approccio presentato è "sovradimensionato" per il caso, ma è utile in generale. Invece di costruire uno per uno i 6 elementi necessari, viene utilizzato un approccio procedurale. Il costruttore `fill` su `Array` costruisce un array di n posti, per ognuno dei quali calcola la funzione passata per argomento. La funzione ha un argomento che rappresenta il contatore (nel caso,

`i`). Qui i posti sono tre, e la funzione restituisce per ognuno un array fatto di un cursore e di un'etichetta. Ogni elemento dell'array `guiArr` sarà perciò a sua volta un array di due elementi. La costruzione del cursore `Slider` e dell'etichetta `StaticText` è del tutto analoga a quanto visto per `Knob`. Si noti che la posizione degli elementi grafici dipende da un parametro `step` comune per tutti e modificato dal contatore. L'idea è "fai tre volte un cursore e un'etichetta, uguali ogni volta e ogni volta scendendo di un po'". Quando `i = 0`, allora il cursore in ascissa è nel punto 10, quando `i = 1` è nel punto 70, e così via. Un simile approccio si rivela estremamente utile nei casi in cui: gli elementi non siano tre ma molti di più; il numero degli elementi possa non essere conosciuto in anticipo ma possa dipendere da qualche variabile. Il costo della programmazione della GUI è allora compensato dalla grande flessibilità ottenuta;

- **34-39**: il blocco definisce l'azione di ogni cursore. L'azione è assegnato iterando sull'array che contiene gli elementi. Ogni elemento è accessibile attraverso `item` e la sua posizione attraverso `index` (si ricordi che i nomi degli argomenti sono arbitrari, ma la loro posizione è vincolante). Ora, ogni elemento dell'array `guiArr` è un array composto da cursore e etichetta. Dunque, `item[0]` restituirà il cursore, e `item[1]` l'etichetta relativa. Ecco che allora l'azione associata ad ogni cursore (35, la funzione associata a ogni movimento del cursore) consisterà nell'aggiornare il valore dell'etichetta associata (attraverso il suo attributo `string`) (36); nell'aggiornare l'array `colorArr` nella posizione `index` col valore del cursore (37); nel modificare lo sfondo di `window` con il risultato della chiamata alla funzione `colorFunc` a cui sono passati `flag` e `colorArr` (38). Si noti che questo passaggio include i passi 2-4 di Figura **3.3** (in cui non sono peraltro raffigurate le etichette).

Quanto visto nel capitolo permette di iniziare a muoversi agevolmente nel linguaggio SC. C'è molto di più, chiaramente. Ma con i riferimenti generali è possibile esplorare il linguaggio stesso attraverso il sistema di help file interattivi e gli esempi che questi ultimi provvedono. Vale la pena chiudere ricordando o introducendo alcune abbreviazioni (un po' di "zucchero sintattico") molto usate e utili, ma che possono indurre il lettore in confusione:

```
// omissione di new in Object
a = Qualcosa.new(argomento) ;
a = Qualcosa(argomento) ;

// omissione di value in Function
funzione.value(valore) ;
funzione.(valore) ;

// assegnazione multipla a Array
# a,b,c = array ;
a = array[0]; b = array[1]; c = array[2] ;

// meno parentesi
if (a, {fai}, {fai altro}) ;
if (a) {fai} {fai altro} ;

Qualcosa.metodoConFunzione_({}) ;
Qualcosa.metodoConFunzione_{} ;

// argomento abbreviato attraverso |
{ arg argomento, unAltroArgomento ; } ;
{ |argomento, unAltroArgomento| } ;
```

Per una discussione si veda il l'help file "Syntax Shortcuts".

4 Sintesi, I: fondamenti di elaborazione del segnale

SuperCollider è indubbiamente specializzato nella sintesi del suono, e tipicamente in tempo reale, attraverso scsynth, il server audio. Obiettivo di questo capitolo non è però introdurre il server audio e le sue funzioni. La discussione seguente si propone invece di fornire una introduzione rapidissima ad alcuni aspetti della rappresentazione e della sintesi del segnale, ma proseguendo nell'interazione esclusivamente con il *linguaggio* SuperCollider. Il lettore che sia già minimamente esperto o sia impaziente di vedere come funziona la sintesi attraverso il server può tranquillamente saltare al capitolo successivo. La discussione sui fondamenti della sintesi permette comunque di approfondire nella pratica alcuni aspetti linguistici già discussi. In ogni caso, per una discussione introduttiva ma dettagliata sul suono e sulla sua rappresentazione si rimanda a AeM, soprattutto capitoli 1, 2, 3.

4.1 Poche centinaia di parole d'acustica

Un suono è una variazione continua della pressione atmosferica percepibile dall'orecchio umano. In quanto vibrazione, dipende dall'eccitazione di corpi del mondo fisico (una chitarra suonata con un plettro, una tenda agitata dal vento, un tavolo battuto con le nocche). Un suono è dunque una serie di compressioni e rarefazioni delle molecole d'aria intorno all'ascoltatore: ciò che si

propaga è appunto questa oscillazione (come in un sistema di biglie che si urtino), non le molecole che oscillano invece intorno ad una posizione d'equilibrio. Un segnale è invece una rappresentazione di un andamento temporale, ad esempio appunto un suono. In particolare, un segnale audio, poiché rappresenta una sequenza di compressioni/rarefazioni della pressione atmosferica, assume la forma di una oscillazione tra valori positivi e valori negativi. Se quest'oscillazione è regolare nel tempo, il segnale è periodico, altrimenti è aperiodico: la maggior parte dei suoni si colloca in un punto tra i due estremi, è cioè più o meno periodica/aperiodica. Il segnale periodico più elementare è la sinusoide, corrispondente fisicamente più o meno al suono di un diapason. La Figura **4.1** permette di riassumere quanto detto. Si supponga di colpire il diapason (come fanno i cantanti per ottenere una nota di riferimento). A seguito dell'energia fornita, il diapason emetterà un suono, che risulta dall'oscillazione delle lamelle. In altri termini, l'oscillazione delle lamelle intorno alla posizione d'equilibrio verra trasmesso alle molecole intorno che oscilleranno "analogicamente" di conseguenza. Dunque, la "forma" dell'oscillazione delle lamelle è sostanzialmente la stessa "forma" dell'oscillazione delle molecole d'aria, che l'ascoltatore percepisce come suono. Si supponga allora di collegare un pennino alla lamella del diapason e di avere un nastro che scorre a velocità uniforme (in effetti, il modello è proprio quello del sismografo): il dispositivo ottenuto registra l'escursione della lamella ed il tracciato che ne consegue è, nel caso del diapason, una sinusoide.

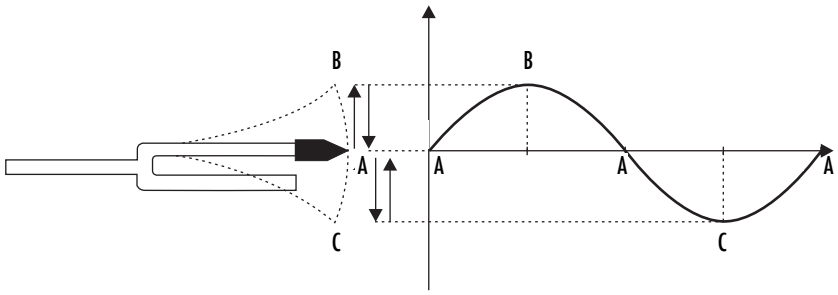


Fig. 4.1 Vibrazione di un diapason e sinusoide (da AeM).

Se si osserva la sinusoide si notano le dimensioni fondamentali del segnale:

- dopo che A è stato raggiunto per la seconda volta, idealmente il tracciato si ripete. Il segnale è dunque *periodico* perché si ripete dopo un certo *periodo* di

tempo, a cui ci si riferisce con T. La *frequenza, f* di un segnale è il numero delle ripetizioni del ciclo nell'unità di tempo. Intuibilmente, $f = \frac{1}{T}$ e viceversa. Le frequenze udibili sono (approssimativamente) comprese tra i 16 e i 25.000 Hz (*Herz*), dove l'unità indica il numero dei cicli al secondo, per cui vi ci si riferisce anche con *cps*);

- l'*ampiezza* di un segnale è appunto l'ampiezza dell'oscillazione, nell'esempio l'escursione massima della lamella. Si noti che il segnale audio è *bipolare*, cioè ha un massimo positivo e uno negativo (il segno è arbitrario ma indica le direzioni opposte dell'oscillazione) ed è (usualmente) simmetrico rispetto allo 0 che rappresenta il punto di inerzia (in cui la lamella è a riposo). La misurazione dell'ampiezza può avvenire in vari modi. Molto spesso (così in SC e in molti software) le unità di misura sono due. Da un lato, un'escursione normalizzata $[-1, 1]$, che astrae dal valore fisico, dall'altro una rappresentazione in *deciBel, dB* che rappresenta una unità di misura fisica ma in qualche modo vicina alla percezione;
- si supponga di avere due diapason identici e di colpirli uno dopo l'altro con un certo intervallo di tempo. Intuibilmente, svolgeranno lo stesso tracciato, ma si troveranno rispettivamente uno in anticipo e l'altro in svantaggio. Avranno cioè una differenza di *fase*, ϕ. Poiché nei segnali periodici si ha ripetizione del ciclo, la fase si misura in *gradi* (come su una circonferenza) o in *radianti*, cioè in frazioni di π, dove 2π rappresenta la circonferenza;
- il movimento di oscillazione del diapason segue un certo tracciato, il segnale mostra cioè una certa *forma d'onda*, la sinusoide. Altre sorgenti acustiche (un oboe, ad esempio) delineerebbero tracciati diversi.

Quanto visto sopra concerne la cosiddetta rappresentazione nel dominio del tempo di un segnale. Il segnale è infatti rappresentato come un fenomeno temporale. Tuttavia, una rappresentazione alternativa è possibile, quella nel dominio della frequenza, in cui infatti il segnale è rappresentato in funzione della frequenza. In particolare, il teorema di Fourier stabilisce che ogni segnale periodico può essere decomposto in una somma di sinusoidi di diversa ampiezza: come se un numero (teoricamente infinito) di sinusoidi di diverso volume suonassero tutte insieme. La sinusoide è matematicamente la forma più semplice di curva periodica (per questo motivo con i termini "segnale semplice" e "tono puro" si fa spesso riferimento a sinusoidi). Il risultato di questa decomposizione è uno *spettro*, che può essere visto come il modo in cui l'energia si distribuisce tra le varie componenti sinusoidali in cui può essere decomposto il segnale in ingresso. Uno spettro non ha tempo, è come una immagine istantanea della composizione interna del suono. Dunque, se si rappresenta un segnale non nel

tempo ma in frequenza si ottiene il suo spettro: a partire dal teorema di Fourier, si può osservare come lo spettro di ogni segnale complesso (non sinusoidale) sia costituito di molte componenti di frequenza diversa. In un segnale periodico queste componenti (dette "armoniche") sono multipli interi della frequenza fondamentale (che ne è il massimo comun divisore). Segnali di questo tipo sono ad esempio l'onda a dente di segna, l'onda quadra, l'onda triangolare, ed in generale la fase stazionaria di tutti i segnali ad altezza musicale riconoscibile ("intonabile"). In un segnale aperiodico le componenti possono essere distribuite in frequenza in maniera arbitraria. Quando si parla (molto vagamente) di "rumore" spesso (ma non sempre) si indicano segnali aperiodici.

4.2 Analogico vs. digitale

Un segnale digitale è una rappresentazione numerica di un segnale analogico ed è doppiamente discreto: rappresenta cioè variazioni d'ampiezza discrete (*quantizzazione*) in istanti discreti di tempo (*frequenza*, o meglio tasso, di campionamento). La digitalizzazione di un segnale analogico è rappresentata in Figura **4.2**. Un segnale continuo (ad esempio, una variazione di tensione elettrica in uscita da un generatore analogico che produca una sinusoide a 440 Hz), (a), viene campionata (b): nel campionamento, un convertitore analogico/digitale (analog-to-digital converter, ADC) "pesca" a instanti regolari definiti da un "orologio" (clock) il valore di tensione. In sostanza, impone una griglia "verticale" di riferimento sul segnale analogico. Il risultato (c) è un segnale impulsivo, cioè costituito di impulsi, tra i quali propriamente non c'è altra informazione. Un meccanismo analogo avviene rispetto all'ampiezza del segnale (d). Qui la griglia definisce un insieme discreto (finito) di valori a cui può essere agganciato il valore d'ampiezza rilevato nel segnale analogico (e). Il risultato è un segnale impulsivo le cui ampiezze sono riferite ad una scala discreta (f): un segnale digitale.

La digitalizzazione può essere cioè pensata come una griglia che viene sovrapposta ad un segnale analogico, e infatti un segnale digitale è spesso rappresentato nei software tipicamente come un curva continua per ragioni di comodità. Ancora, alcune volte è rappresentato come una spezzata: in realtà, matematicamente esiste una sola funzione continua che passa per tutti punti, e dunque, una volta riconvertito in analogico (attraverso un convertitore, digital-to-analog

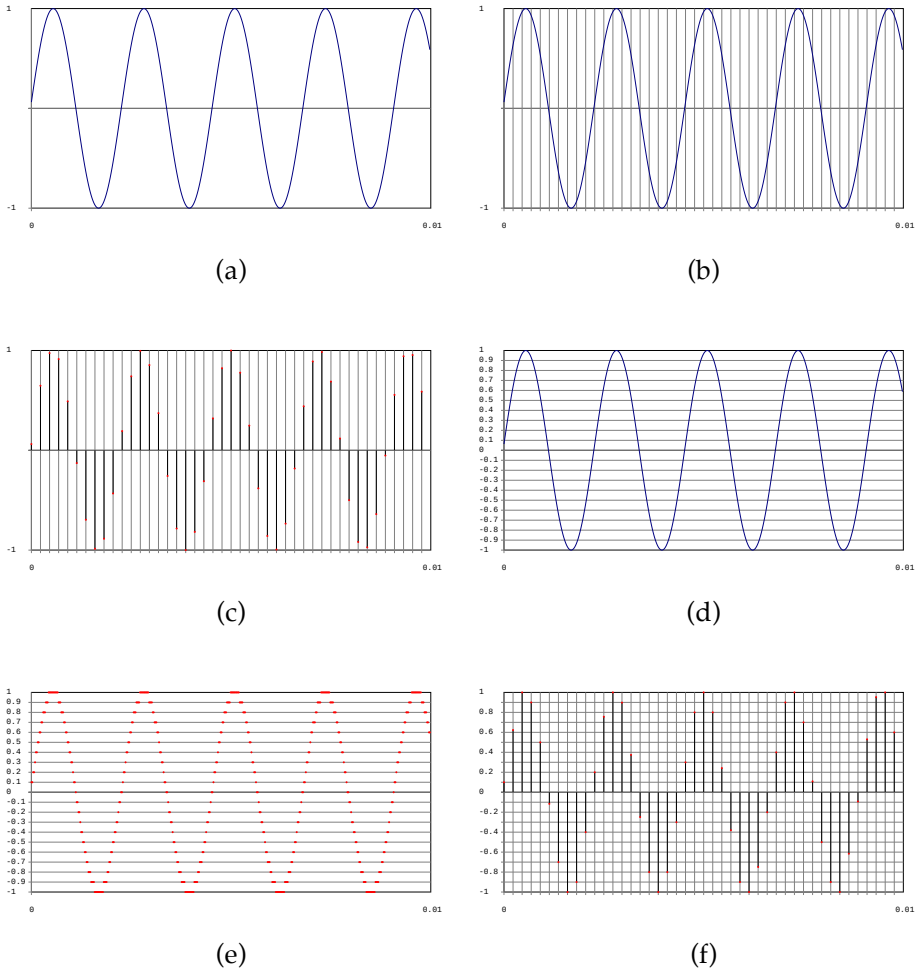


Fig. 4.2 Segnale digitale: da analogico a digitale.

converter, DAC), la curva avrà esattamente la stessa forma di quella in ingresso. Ovviamente il processo produce delle approssimazioni. La quantizzazione definisce la gamma dinamica massima del segnale (a qualità CD, 16 bit, 96 dB) mentre il campionamento definisce la massima frequenza rappresentabile nel segnale digitale. Rispetto a quest'ultimo punto, come dimostrato dal teorema di Nyquist, la frequenza di campionamento stabilisce infatti che le frequenze rappresentabili in digitali siano tutte quelle entro la metà della frequenza di

campionamento: a qualità CD (44.100), dunque si potranno rappresentare frequenze fino a 22.050 Hz (una buona approssimazione rispetto alle frequenze udibili).

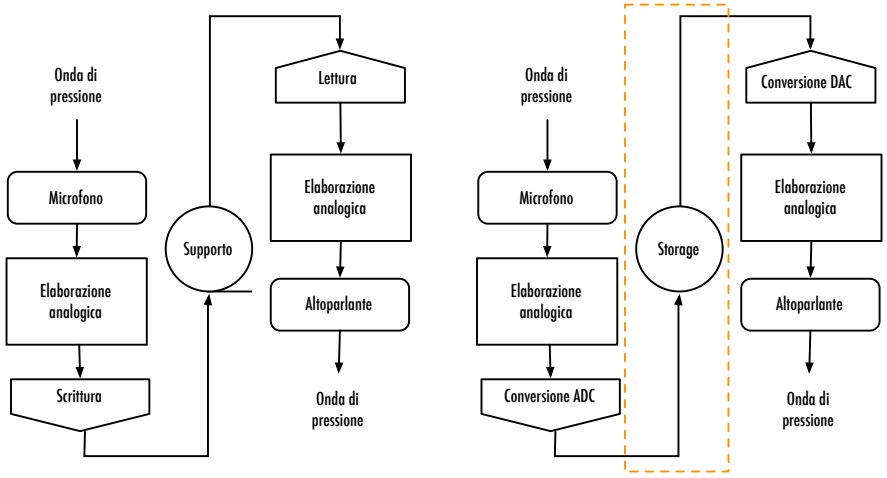


Fig. 4.3 Catena dell'audio: analogico e digitale

Nel caso dell'audio digitale, il segnale, appunto "digitalizzato", è disponibile sotto forma di informazione numerica, e dunque può essere rielaborato attraverso un calcolatore: si possono cioè effettuare calcoli a partire dai numeri che rappresentano il segnale. I passi principali della registrazione digitale sono i seguenti (Figura **4.3**):

1. **conversione anologico-digitale**: il segnale analogico viene filtrato e convertito dal dominio analogico (in quanto variazione continua della tensione elettrica, ad esempio prodotta da un microfono) nel formato numerico attraverso l'ADC;

2. **elaborazione**: il segnale digitalizzato, che ha assunto la forma di una sequenza di numeri, può essere modificato;

3. **conversione digitale-analogica**: per essere ascoltabile, la sequenza numerica che compone il segnale viene riconvertita in segnale analogico attraverso il DAC: effettuato il filtraggio, si origina di nuovo una variazione continua della tensione elettrica che può, ad esempio, mettere in azione un altoparlante.

Nella discussione precedente, il processo di digitalizzazione è stato descritto rispetto ad un segnale analogico preesistente, e d'altra parte l'analogico è sempre l'orizzonte del digitale, per il semplice fatto che, per essere ascoltato, un segnale digitale deve essere necessariamente riconvertito in un segnale continuo, elettrico in particolare, così da poter essere utilizzato, attraverso un insieme di passaggi, per mettere in vibrazione un altoparlante (anche solo un auricolare). Tuttavia, un segnale digitale non deve necessariamente partire dall'analogico. L'assunto di partenza della computer music è che il calcolatore possa essere impiegato per sintetizzare direttamente il suono. Il cuore della sintesi digitale sta nell'escludere il passaggio 1, generando direttamente la sequenza numerica che poi verrà convertita in segnale analogico. Questo non esclude affatto la possibilità di lavorare con campionamenti provenienti da fonti "esterne", ma sottolinea piuttosto come l'aspetto fondamentale risieda nei metodi e nelle procedure di calcolo che presiedono alla sintesi. Se dunque è sempre possibile per il compositore "digitale" lavorare sulla componente analogica (ad esempio registrando ed elaborando analogicamente il segnale), la componente più caratterizzante della sua prassi sta nello sfruttare la natura numerica (e dunque "calcolabile") del segnale digitale (Figura **4.4**).

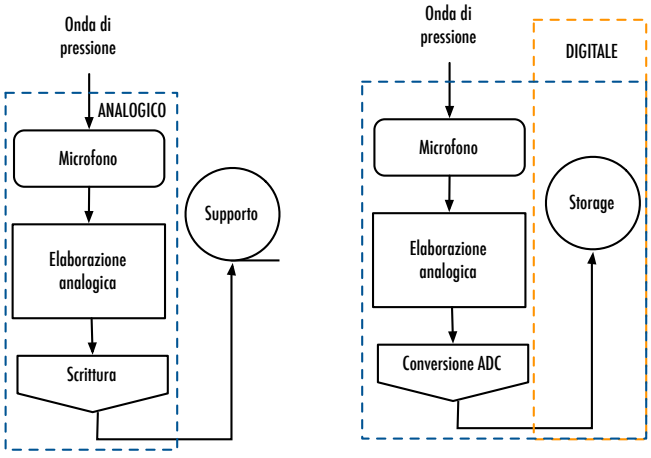


Fig. 4.4 Composizione analogica e digitale: ambiti.

Un segnale è descrivibile, nel caso discreto, attraverso una funzione matematica

$$y = f[x]$$

la funzione indica che per ogni istante discreto di tempo x il segnale ha il valore d'ampiezza y. Un segnale digitale è una sequenza di caselle $x_0, x_1, x_2, ...$ a cui corrispondono valori d'ampiezza $y_0, y_1, y_2, ...$

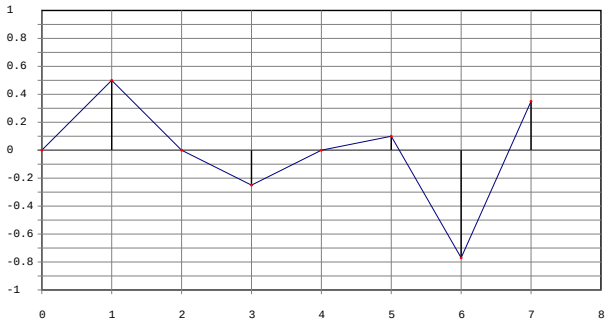


Fig. 4.5 L'array `[0, 0.5, 0, -0.25, 0, 0.1, -0.77, 0.35]`.

La struttura dati che rappresenta un segnale è allora tipicamente un array, una sequenza di celle di memoria consecutive e omogenee, contenenti cioè lo stesso tipo di dati (il tipo numerico prescelto per il segnale in questione). Così ad esempio un array come

```
[0, 0.5, 0, -0.25, 0, 0.1, -0.77, 0.35]
```

descrive un segnale composto da 8 campioni (Figura **4.5**), dove l'indice (il numero d'ordine che etichetta progressivamente ognuno degli otto valori) rappresenta x inteso come istante di tempo, mentre il dato numerico associato rappresenta y, inteso come il valore d'ampiezza del segnale nell'istante x:

$$x = 0 \rightarrow y = 0$$
$$x = 1 \rightarrow y = 0.5$$
$$...$$
$$x = 7 \rightarrow y = 0.35$$

SuperCollider permette agevolmente di visualizzare strutture dati attraverso il metodo `plot`. Ad esempio, la classe `Array` risponde al messaggio `plot` generando una finestra e disegnandovi la curva spezzata ottenuta congiungendo i valori contenuti nell'array.

```
[0, 0.5, 0, -0.25, 0, 0.1, -0.77, 0.35].plot ;
[0, 0.5, 0, -0.25, 0, 0.1, -0.77, 0.35].plot
    ("un array", minval:-1, maxval:1, discrete:true) ;
```

La prima riga utilizza i valori predefiniti, la seconda riga fornisce un esempio di uso di alcune opzioni possibili. Il metodo `plot` è implementato non solo nella classe `Array` ma in molte altre, ed è estremamente utile per capire il comportamento dei segnali su cui si sta lavorando.

4.3 Algoritmi di sintesi

Un algoritmo per la sintesi del suono è una procedura formalizzata che ha come scopo la generazione della rappresentazione numerica di un segnale audio.

Il linguaggio SC (sclang) permette di sperimentare algoritmi di sintesi del segnale in tempo differito senza scomodare –per ora– il server audio (scsynth). Non è certo un modo usuale di utilizzare SuperCollider, e tra l'altro sclang è pensato come un linguaggio di alto livello ("lontano dalla macchina e vicino al programmatore"), intrinsecamente non ottimizzato per operazioni numericamente massive (quello che in gergo si chiama "number crunching"). Tuttavia, a livello didattico, sia sul lato della sintesi dei segnali che su quello dell'approfondimento linguistico, può essere utile ragionare su alcuni semplici algoritmi di sintesi.

Poiché il segnale previsto per i CD audio è campionato a 44.100 Hz (attualmente lo standard audio più diffuso), se si vuole generare un segnale mono della durata di 1 secondo a qualità CD, è necessario costruire un array di 44.100 posti: il processo di sintesi del segnale consiste allora nel definire ed implementare un algoritmo che permetta di "riempire" ognuno di questi posti (letteralmente numerati) con un valore. Così, se si vuole generare un segnale sinusoidale puro, il metodo più semplice consiste nel calcolare l'ampiezza y per ogni campione x del segnale in accordo con la funzione del seno e nell'associare il

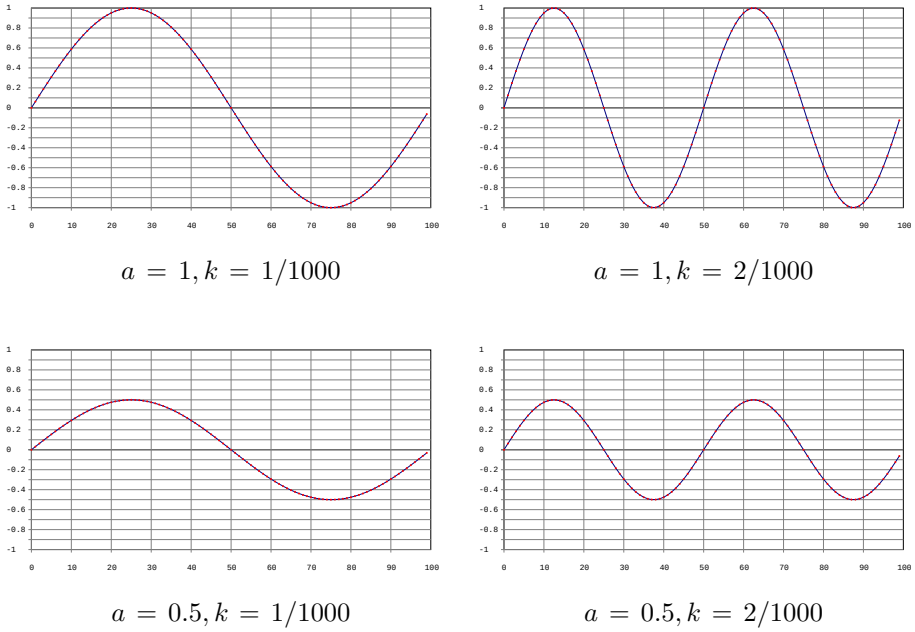
valore y all'indice x dell'array A, che rappresenta S. Una funzione periodica si definisce come segue:

$$y = f(2\pi \times x)$$

Un segnale sinusoidale è descritto dalla formula:

$$y = a \times sin(2\pi \times k \times x)$$

L'effetto dei parametri a e k è rappresentato in Figura **4.6**, dove è disegnato (in forma continua) un segnale (discreto) composto da 100 campioni.



$a = 1, k = 1/1000$ $a = 1, k = 2/1000$

$a = 0.5, k = 1/1000$ $a = 0.5, k = 2/1000$

Fig. 4.6 Sinusoide e variazione dei parametri a e k.

L'algoritmo di sintesi per un segnale sinusoidale, scritto in pseudo-codice (ovvero in un linguaggio inesistente ma che permette di illustrare in forma astratta la programmazione), è dunque il seguente:

```
Per ogni x in A:
      y = a*sin(k*x)
      A[x] = y
```

dove la riga 1 del ciclo calcola il valore y in funzione di due parametri a e k che controllano l'ampiezza e la frequenza della sinusoide, mentre la seconda riga assegna all'indice x di A il valore y. SC permette agevolmente di implementare un simile algoritmo. Ad esempio, il primo segnale di Figura **4.6** è stato ottenuto con il codice seguente:

```
1 var sig, amp = 1, freq = 1, val ;
2 sig = Array.newClear(100) ;
3 sig.size.do({ arg x ;
4     val = amp*sin(2pi*freq*(x/sig.size)) ;
5     sig[x]= val ;
6 }) ;
7 sig.plot(minval:-1, maxval:1, discrete:true) ;
```

Nel codice, la riga 1 definisce le variabili che conterranno rispettivamente il segnale, l'ampiezza, la frequenza e il valore incrementale dei campioni. Viene quindi creato un array di 100 elementi. Le righe 3-6 sono occupate da un ciclo: `sig.size` restituisce la dimensione dell'array (100). Per `sig.size` (100) volte viene valutata la funzione nel ciclo `do`: poiché `x` rappresenta l'incremento lungo l'array (che rappresenta il tempo, ovvero $0, 1, 2...98, 99$), è in funzione di `x` che viene calcolato il valore della funzione ($f[x]$). Il valore della frequenza del segnale desiderato indica il numero di cicli (2π) al secondo. Nel caso, `freq = 1` indica che un ciclo della sinusoide (che va da 0 a 1) verrà "distribuito" su 100 punti. Di qui, il significato di `x/sig.size`. Se la frequenza desiderata fosse 440 Hz ($\rightarrow$ cicli al secondo, `freq = 440`) allora ci dovrebbero essere $440 \times 2\pi$ cicli ogni secondo (`2*pi*freq`). Questo valore deve essere distribuito su tutti i posti dell'array (`x/sig.size`). Ottenuto il valore `val`, questo sarà compreso (per definizione trigonometrica) nell'escursione $[-1, 1]$ e dunque può essere scalato per `amp`. La riga 5 assegna al posto `x` di `sig` il valore `val`. Il segnale viene quindi disegnato nell'escursione d'ampiezza $[-1, 1]$ in forma discreta (7).

Una classe utile per calcolare segnali secondo quanto visto finora è `Signal`, che è una sottoclasse di `ArrayedCollection` (la superclasse più generale degli oggetti array-like) specializzata per la generazione di segnali. La classe `Signal` è specializzata per contenere array di grandi dimensioni e omogenei per tipo di dato (come tipico per i segnali audio). L'esempio seguente è un'ovvia riscrittura del precedente con `Signal` e un segnale di 44100 campioni, pari a un secondo di audio a tasso di campionamento standard CD.

```
var sig, amp = 1, freq = 440, val ;
sig = Signal.newClear(44100) ;
sig.size.do({ arg x ;
    val = amp*sin(2pi*freq*(x/sig.size)) ;
    sig[x]= val ;
}) ;
```

Il segnale ottenuto può essere salvato su hard disk in formato audio così da essere eseguito: `Signal` può così essere utilizzato per generare materiali audio utilizzabili in seguito, attraverso (e sempre dal lato client) la classe `SoundFile`.

```
(
var sig, amp = 1, freq = 440, val ;
var soundFile ;

sig = Signal.newClear(44100) ;
sig.size.do({ arg x ;
    val = amp*sin(2pi*freq*(x/sig.size)) ;
    sig[x]= val ;
}) ;

soundFile = SoundFile.new ;
soundFile.headerFormat_("AIFF").sampleFormat_("int16").numChannels_(1) ;
soundFile.openWrite("/Users/andrea/Desktop/signal.aiff") ;
soundFile.writeData(sig) ;
soundFile.close ;
)
```

Nell'esempio, `SoundFile` crea un file audio (11), di cui sono specificabili le proprietà (12): il tipo (`"AIFF"`), la quantizzazione (16 bit, `"int16"`), il numero di canali (mono, `1`)[1]. È importante specificare la quantizzazione perché SC internamente (e per default) lavora a 32 bit, in formato float: un formato utile per la precisione interna ma piuttosto scomodo come formato di rilascio finale. Dopo aver create l'oggetto di tipo file è necessario specificare il percorso del file

[1] Si noti il concatenamento dei messaggi: ognuno dei metodi restituisce infatti l'oggetto stesso.

richiesto (13). A questo punto si possono scrivere sul file i dati contenuti nell'array `sig` (14), che è stato generato esattamente come nell'esempio precedente. Ad operazioni concluse, il file deve essere chiuso (15), altrimenti non sarà leggibile. Il file creato può così essere ascoltato.

Per evitare tutte le volte di scrivere su file i dati generati, è possibile utilizzare il metodo `play` che offre la possibilità di ascoltare il contenuto dell'oggetto `Signal` (come ciò avvenga nel dettaglio lo si vedrà poi).

```
1 var sig, amp = 1, freq = 441, val ;
2 sig = Signal.newClear(44100) ;
3 sig.size.do({ arg x ;
4     val = amp*sin(2pi*freq*(x/sig.size)) ;
5     sig[x]= val ;
6 }) ;
7 sig.play(true) ;
```

Ci si può chiedere a quale frequenza: fino ad ora la frequenza è stata infatti specificata soltanto in termini di frequenza relativa tra le componenti. Poiché con `play` si passa al tempo reale[2], vanno considerate altre variabili che verranno discusse in seguito. SC per default genera un segnale con un tasso di campionamento (*sample rate, sr*) pari a 44.100 campioni al secondo. Il contenuto dell'array, dopo essere stato messo in una locazione di memoria temporanea (un "buffer") viene letto perciò alla frequenza di 44.100 campioni al secondo (un segnale di 44.100 campioni viene "consumato" in un secondo). In altri termini, SC preleva un valore dal buffer ogni $1/44.100$ secondi. Con il metodo `play(true)` (il valore di default) l'esecuzione è in loop: una volta arrivato alla fine, SC riprende da capo. Dunque, se $size$ è la dimensione in punti dell'array, il periodo del segnale ("quanto dura" in secondi) è $size/sr$, e la frequenza è il suo inverso: $1/size/sr = sr/size$. Se $size = 1000$, allora $f = 44.100/1000 = 44.1Hz$. Viceversa, se si intende ottenere un segnale la cui fondamentale sia f, la dimensione dell'array che contenga un singolo ciclo deve essere $size = sr/f$. È un calcolo soltanto approssimativo perché $size$ deve essere necessariamente intero. Se la dimensione dell'array è 44.100 (come in molti esempi nel capitolo, ma non in tutti) allora il segnale viene letto una volta al secondo. Poiché l'array contiene un numero $freq$ di cicli, `freq` indica effettivamente la frequenza del segnale.

[2] Perciò è necessario effettuare il boot del server audio, premendo la finestra "Server" e selezionando "Boot Server" nel menu. Tutto risulterà chiaro più avanti.

Nel seguito, non verrà esplicitamente usato `play`: al lettore la possibilità di utilizzare il metodo e il compito di adeguare gli esempi.

Tornando ora alle questioni relative alla sintesi, come si è detto, un segnale periodico è una somma di sinusoidi. Una implementazione "rudimentale" potrebbe essere la seguente:

```
1  var sig, sig1, sig2, sig3 ;
2  var amp = 1, freq = 1, val ;
3  sig = Signal.newClear(44100) ;
4  sig1 = Signal.newClear(44100) ;
5  sig2 = Signal.newClear(44100) ;
6  sig3 = Signal.newClear(44100) ;

8  sig1.size.do({ arg x ;
9      val = amp*sin(2pi*freq*(x/sig.size)) ;
10     sig1[x]= val ;
11 }) ;
12 sig2.size.do({ arg x ;
13     val = amp*sin(2pi*freq*2*(x/sig.size)) ;
14     sig2[x]= val ;
15 }) ;
16 sig3.size.do({ arg x ;
17     val = amp*sin(2pi*freq*3*(x/sig.size)) ;
18     sig3[x]= val ;
19 }) ;
20 sig = (sig1+sig2+sig3)/3 ;
21 sig.plot ;
```

Nell'esempio si vogliono calcolare la fondamentale e le prime due armoniche. Allo scopo si generano quattro oggetti `Signal` della stessa dimensione (3-6). Quindi si calcolano i quattro segnali, ripetendo un codice strutturalmente identico che varia solo per la presenza di un moltiplicatore di `freq` (8-19). Infine, `sig` è utilizzato per contenere la somma degli array (implementata negli array come somma degli elementi per le posizioni successive). L'obiettivo è appunto una "somma" di sinusoidi. I valori nell'array `sig` vengono quindi cautelativamente divisi per 3. Infatti, se il segnale deve rimanere nell'escursione $[-1, +1]$, allora lo scenario peggiore possibile è quello dei tre picchi (positivi o negativi) in fase, la cui somma sarebbe appunto 3. Dividendo per 3, allora l'ampiezza massima possibile sarà -1 o 1. Il segnale può essere scritto su file audio come

nell'esempio precedente. Si noti che per chiarezza grafica, se li si vuole disegnare sullo schermo è conveniente ridurre molto il numero dei punti e impostare $freq = 1$.

In programmazione, la ripetizioni di blocchi di codice sostanzialmente identici è sempre sospetta. Infatti, indica che l'iterazione non è in carico al programma ma al programmatore. Inoltre, la ripetizione tipicamente porta errori. Ancora, il programma non è modulare perché è pensato non per il caso generale di una somma di n sinusoidi, ma per il caso specifico di 3 sinusoidi. Infine, il codice diventa inutilmente prolisso e difficile da leggere.

L'algoritmo seguente permette in maniera più elegante di calcolare un segnale periodico che includa un numero n di armoniche, definito nella variabile `harm`.

```
1  var sig, amp = 1, freq = 440, val ;
2  var sample, harm = 3 ;
3  sig = Signal.newClear(44100) ;
4  sig.size.do({ arg x ;
5      sample = x/sig.size ;
6      val = 0 ;
7      harm.do{ arg i ;
8          harm = i+1 ;
9          val = val + (amp*sin(2pi*freq*(i+1)*sample) );
10     } ;
11     sig[x]= val/harm ;
12 }) ;
```

Il ciclo (4) calcola il valore di ogni campione e usa `sample` per tenere in memoria la posizione. Quindi per ogni campione reinizializza `val` a 0. A questo punto per ogni campione effettua un numero n di calcoli, cioè semplicemente calcola il valore della funzione seno per la frequenza fondamentale e le prime $n - 1$ armoniche, come fossero n segnali diversi. A ogni calcolo, aggiunge il valore ottenuto a quello calcolato per lo stesso campione dalle altre funzioni (la "somma" di sinusoidi). Infine, `val` viene memorizzato nell'elemento di `sig` su cui si sta operando, dopo averlo diviso per il numero delle armoniche (si ricordi l'argomento di cautela precedente). Rispetto all'esempio precedente, il codice definisce univocamente la funzione richiesta e la parametrizza come desiderato (si provi a variare `harm` aumentando le armoniche), è più semplice da correggere, è generale, è più compatto. Vale la pena introdurre un altro punto, anche se non avrà conseguenze pratiche. L'esempio iterativo è un esempio di approccio

non in tempo reale. Prima si calcolano i tre segnali, quindi si sommano (mixano, si potrebbe dire). In tempo reale, come si vedrà, il segnale viene generato continuamente, dunque non è pensabile effettuare una sintesi di un segnale la cui durata è a rigore indeterminata ("qualcosa sta continuando a suonare"). Il secondo esempio è invece potenzialmente implementabile in tempo reale: infatti, l'algoritmo calcola il valore finale di un campione alla volta. Nel caso discusso, questo viene scritto in una posizione incrementale dell'array `sig`, ma potrebbe essere invece inviato alla scheda audio per la conversione.

A partire dall'ultimo esempio diventa agevole possibile generare altri segnali periodici già ricordati. Così, per definizione, un onda a dente di sega è un segnale periodico che ha teoricamente infinite armoniche di frequenza $f \times n$, dove f è la frequenza fondamentale e $n = 2, 3, 4, ...$, e di ampiezza rispettivamente pari a $1/2, 3, 4...$ (ovvero ognuna inversamente proporzionale al numero della armonica relativa). L'esempio seguente introduce una piccola modifica nell'algoritmo precedente.

```
var sig, amp = 1, freq = 440, val ;
var ampl ; // ampl vale per ogni componente
var sample, harm = 10 ;
sig = Signal.newClear(44100) ;
sig.size.do({ arg x ;
    sample = x/sig.size ;
    val = 0 ;
    harm.do{ arg i ;
        harm = i+1 ;
        ampl = amp/harm ;
        val = val + (ampl*sin(2pi*freq*(i+1)*sample) );
    } ;
    sig[x]= val/harm ;
}) ;
```

L'unica differenza è l'introduzione della variabile `ampl`, per ogni campione, per ogni armonica (10) viene calcolata l'ampiezza relativa dividendo l'ampiezza di riferimento `amp` per il numero d'armonica. Se si prova a incrementare `harm` si nota come l'onda a dente di sega venga progressivamente approssimata con maggiore esattezza.

Un'onda quadra può essere generata nello stesso modo dell'onda a dente sega, ma aggiungendo soltanto le armoniche dispari ($n = 1, 3, 5...$). In altri termini, un'onda quadra è un'onda a dente di sega in cui le armoniche pari hanno ampiezza nulla. Il codice è riportato nell'esempio seguente.

```
var sig, amp = 1, freq = 440, val ;
var ampl ; // ampl vale per ogni componente
var sample, harm = 20 ;
sig = Signal.newClear(44100) ;
sig.size.do({ arg x ;
    sample = x/sig.size ;
    val = 0 ;
    harm.do{ arg i ;
        harm = i+1 ;
        if(harm.odd){ // e' dispari?
            ampl = amp/harm ;
            val = val + (ampl*sin(2pi*freq*(i+1)*sample) );
        }
    } ;
    sig[x]= val/harm ;
}) ;
```

Come si vede, il valore di `ampl` questa volta è sottoposto ad una valutazione condizionale (10), in cui `harm.odd` restituisce vero o falso a seconda se la variabile (il numero di armonica) è dispari o pari. Se dispari, il valore viene calcolato come nel caso dell'onda a dente di sega, se è pari, semplicemente `val` non è calcolato per la componente che non viene incrementata.

Alcune variazioni del numero di armoniche (3,5,40) per i tre casi discussi (sinusoidi a ampiezza costante, onda a dente di sega e onda quadra) sono raffigurate in Figura **4.7**. In particolare, nell'onda a dente di sega e in quella quadra il contributo delle armoniche è visibile nel numero delle "gobbe" che il segnale presenta.

4.4 Metodi di `Signal`

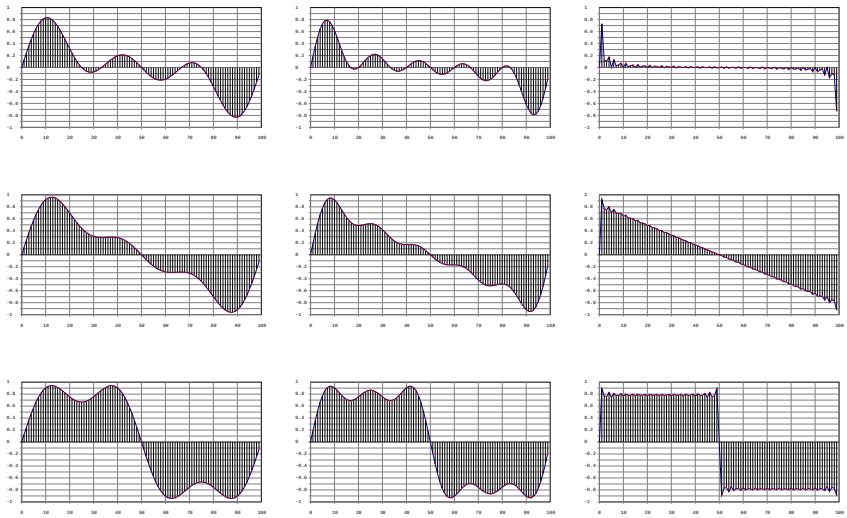


Fig. 4.7 Da sinistra, somma di $3, 5, 40$ sinusoidi: dall'alto, ampiezza costante, onda a dente di sega, onda quadra.

La classe `Signal` prevede molte possibilità di elaborazione, che non sono stati discussi perché era più utile una implementazione "diretta". `Signal` è infatti una classe che serve per generare segnali, tipicamente non perché questi siano scritti su file, quanto per utilizzarli in alcuni metodi per la sintesi del suono (sintesi *wavetable*, se ne parlerà più avanti). Ad esempio, il metodo `sineFill` è espressamente dedicato alla generazione di segnali per somma di sinusoidi. I suoi argomenti sono

1. la dimensione
2. un array che specifica una serie di ampiezze
3. un array che specifica una serie di fasi

Nell'esempio minimale seguente, una sinusoide di 100 punti è generata specificando un solo elemento (dunque, la fondamentale) nell'array delle ampiezze. Si noti anche il metodo `scale` definito su `Signal` ma non su altre sottoclassi di `ArrayedCollection`, che moltiplica gli elementi dell'array per un fattore specificato.

```
var sig, amp = 0.75, freq = 440 ;
sig = Signal.sineFill(100, [1]).scale(amp) ;
sig.plot(minval:-1, maxval:1) ;
```

Come si diceva, ampiezze e fasi sono riferite alle armoniche del segnale sinusoidale. Ad esempio, un array d'ampiezze `[0.4, 0.5, 0, 0.1]` indica che verranno calcolate le prime 4 armoniche, dove f_2 avrà ampiezza `0.4`, f_3 `0.5` e così via. Si noti che per eliminare una componente armonica è sufficiente specificare un valore d'ampiezza 0 (è il caso di f_4). Il file di help propone il codice:

```
Signal.sineFill(1000, 1.0/[1,2,3,4,5,6]) ;
```

Il codice genera un array di 1000 punti e lo riempie con una sinusoide e con le sue prime 5 armoniche. La sintassi `1.0/[1,2,3,4,5,6]` è interessante. Se la si valuta, la post window restituisce:

```
1.0/[1,2,3,4,5,6]
[ 1, 0.5, 0.33333333333333, 0.25, 0.2, 0.16666666666667 ]
```

Cioè: un numero diviso un array restituisce un array in cui ogni ogni elemento è pari al numero diviso all'elemento di partenza. In altre parole è come scrivere `[1.0/1, 1.0/2, 1.0/3, 1.0/4, 1.0/5, 1.0/6]`. L'array contiene dunque una serie di 6 ampiezze inversamente proporzionali al numero di armonica. Come ormai intuibile, il segnale risultante approssima un'onda a dente di sega[3]. Nell'esempio seguente l'approssimazione è decisamente migliore. Il metodo `series`, definito per `Array`, crea un array ed ha come argomenti `size`, `start` e `step`: l'array è riempito da una serie di `size` interi successivi che, iniziando da `start`, proseguono incrementando di `step`. Dunque, l'array contiene i valori $1, 2, 3, ...1000$. Il segnale `sig` genera una sinusoide e i suoi primi 999 armonici

[3] Qui non si considera la fase, ma il discorso è analogo.

superiori con valore inversamente proporzionale al numero d'armonica. Si noti la grande compattezza della rappresentazione linguistica.

```
1 var sig, arr ;
2 arr = Array.series(size: 1000, start: 1, step: 1) ;
3 sig = Signal.sineFill(1024, 1.0/arr) ;
```

Un'onda quadra è un'onda a dente sega, senza armoniche pari. Il codice è riportato nell'esempio seguente.

```
1 var sig, arr, arr1, arr2 ;
2 arr1 = Array.series(size: 500, start: 1, step: 2) ;
3 arr1 = 1.0/arr1 ;
4 arr2 = Array.fill(500, {0}) ;
5 arr = [arr1, arr2].flop.flat ;
6 // arr = [arr1, arr2].lace(1000) ;

8 sig = Signal.sineFill(1024, arr) ;
9 sig.plot ;
```

Le ampiezze delle armoniche pari devono essere pari a 0, quelle dispari inversamente proporzionali al loro numero d'ordine. L'array `arr1` è l'array delle ampiezze delle armoniche dispari. Si noti che `step: 2`, e che `arr1` è già opportunamente scalato (3). L'array `arr2` (4) è creato con il metodo `fill` che riempie un array della dimensione voluta (qui `500`) valutando per ogni posto la funzione. Siccome è necessario un array costituito da zeri, la funzione deve semplicemente restituire `0`. La riga 5 crea il nuovo array `arr`, ed è più interessante, poiché fa uso dei metodi `flop` e `flat`. Si veda l'esempio dalla post window seguente:

```
1  a = Array.fill(10, 0) ;

3  [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]

5  b = Array.fill(10, 1) ;

7  [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ]

9  c = [a,b] ;

11 [ [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
12 [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ] ]

14 c = c.flop ;

16 [ [ 0, 1 ], [ 0, 1 ], [ 0, 1 ], [ 0, 1 ], [ 0, 1 ], [ 0, 1 ], [ 0, 1 ],
17 [ 0, 1 ], [ 0, 1 ], [ 0, 1 ] ]

19 c = c.flat ;

21 [ 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1 ]
```

Dopo aver creato due array di 10 posti `a` e `b` (1, 7), viene create un nuovo array `c` che contiene i due array `a` e `b` come elementi (9, come si vede in 11 e 12). Il metodo `flop` (14) "interallaccia" coppie di elementi dai due array (si veda 16). Il metodo `flat` (18) "appiattisce" un array "eliminando tutte le parentesi": si perde la struttura in sotto-array degli elementi. Rispetto alla riga 9 il risultato è un'alternanza di elementi da uno e dall'altro array (da `a` e da `b`). Nell'esempio relativo all'onda quadra il risultato della riga 5 è un'alternanza di elementi di `arr1` e di zeri (provenienti da `arr2`). Come accade molto spesso, l'operazione è in realtà praticabile in SC più semplicemente attraverso un metodo dedicato, `lace` (6, commentata): `lace(1000)` restituisce un array di dimensione `1000` pescando alternativamente da `arr1` e `arr2`.

4.5 Altri segnali e altri algoritmi

Altri segnali periodici possono essere generati in forma trigonometrica/procedurale, attraverso una somma di sinusoidi calibrata in ampiezza. È il caso ad esempio dell'onda triangolare. Per quanto descritto in forma discreta, il metodo ha una sua fondatezza di derivazione continua. Tuttavia nel dominio discreto sono anche possibili metodi diversi, definiti tipicamente come "non standard". Ad esempio, si può assumere un approccio di tipo geometrico. Il periodo di un'onda triangolare può essere pensato come costituito di quattro segmenti: il primo nell'intervallo $[0.0, 1.0]$, il secondo in quello $[1.0, 0.0]$, il terzo in quello $[0.0, -1.0]$ e il quarto in quello $[-1.0, 0]$.

```
1  // Onda triangolare per segmenti

3  var first, second, third, fourth, total ;
4  var size = 50 , step;

6  step = 1.0/size ;
7  first = Signal.series(size, 0, step) ;
8  second = (first+step).reverse ;
9  third = second-1 ;
10 fourth = first-1 ;
11 total = (first++second++third++fourth) ;

13 total.plot;
```

Nell'esempio la variabile `size` rappresenta la dimensione degli array che contengono i quattro segmenti, mentre `step` è l'incremento del valore di ampiezza. Il primo segmento è allora un array di tipo `Signal` riempito da un numero $step$ di valori con incremento $step$: contiene valori da 0 a $1 - step$ (7). Il secondo segmento segue il percorso contrario: il metodo `reverse` restituisce un array leggendo dall'ultimo al primo elemento l'array sui cui è chiamato. Prima, viene aggiunto uno step ad ognuno degli elementi dell'array: `second` contiene valori da 1 a $0 + step$. I segmenti successivi sono ottenuti generando due array `third` e `fourth` che sottraggono 1 rispettivamente a `second` e `first`, cioè li "traslano in basso" (9, 10). Infine l'array `total` è ottenuto concatenando i quattro segmenti. Si noti che le operazioni di addizione (come le altre operazioni sugli array) restituiscono un array in cui ognuno degli elementi risulta dall'applicazione dell'operazione sul rispettivo elemento dell'array di partenza. Ovvero:

```
[1, 2, 3, 4]*2
[ 2, 4, 6, 8 ]
```

Il metodo di sintesi geometrico permette in effetti di ottenere risultati migliori di una somma di sinusoidi che può soltanto ad approssimare l'onda triangolare. Sommando sinusoidi sarebbero necessarie infinite componenti per generare un'onda triangolare: gli spigoli risultanti sono sempre un po' "smussati".

Rispetto all'esempio è possibile un'osservazione analoga a quella precedente relativa a tempo reale / differito. Un approccio a segmenti deve prima calcolare i segmenti per intero e poi connetterli: non è dunque una tecnica ottimale per il tempo reale. Un approccio alternativo è presentato nell'esempio seguente:

```
// Onda triangolare per serie e cambio segno

var size = 100, freq = 2 ;
var step, val, start = 0, sign = 1 ;
var sig = Signal.newClear(size) ;
step = 1.0/size*(freq*4);
val = 0 ;
sig.size.do{ arg x;
    if((val >= 1) || (val<=1.neg)) {
        sign = sign.neg
    } ;
    val = val+(step*sign);
    sig[x] = val ;
} ;

sig.plot ;
```

L'idea è in questo caso nasce dall'osservazione che la forma d'onda è una retta la cui pendenza viene invertita quando un limite viene raggiunto, ovvero i due picchi simmetrici per valore, positivo e negativo ($[-1, +1]$). La variabile `step` (6) determina la pendenza della retta. Quattro sono i segmenti in gioco, e `freq` li moltiplica per il numero delle loro ripetizioni (in questo caso, 2). Se si considera un array di dimensione fissa e si immagina un'onda triangolare e

un'altra di frequenza doppia, diventa chiaro che la pendenza della seconda è il doppio (sale/scende a "doppia velocità") della prima (Figura **4.8**).

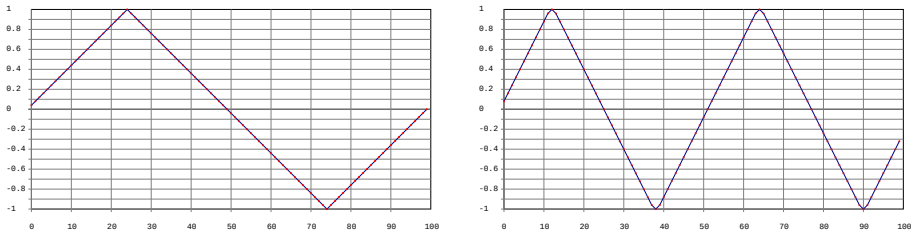


Fig. 4.8 Onde triangolari con frequenza f e $2f$

Definito perciò il significato di `step`, si inizializza `val`, il valore del primo campione (7). Il ciclo itera sui campioni di `sig` (8-14). Verifica che `val` non sia maggiore del picco positivo o minore del negativo, e in quel caso aggiunge a `val` un incremento `step` (12). L'incremento però è moltiplicato per `sign`.

```
1 0.216.sign
2 1

4 -0.216.sign
5 -1
```

Inizialmente `sign` vale 1, dunque non sposta il valore dell'incremento. Se però un picco è superato, allora `sign` è invertito con `sign.neg` (9-11). In altri termini, l'onda cresce fino a 1 e da lì decresce (incremento negativo) fino a -1.

Tornando alla sintesi attraverso componenti sinusoidali, l'esempio già visto di somma di sinusoidi dimostra la rilevanza della fase. Infatti, le 50 componenti in fase crescono tutte insieme a inizio ciclo producendo il picco iniziale per poi "neutralizzarsi" fino al picco negativo di fase π[4]. In Figura **4.9**, in alto, sono riportate le fasi delle componenti (sempre 0) e il segnale risultante.

[4] Si noti che pi e 2pi sono parole riservate in SC.

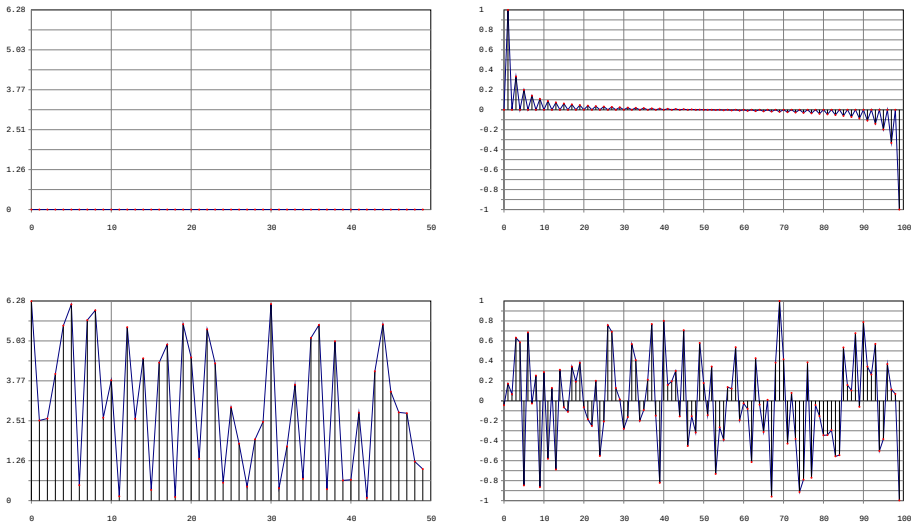


Fig. 4.9 Somma di sinusoidi: in alto da sinistra, fase 0 e segnale; in basso da sinistra, fase casuale e segnale.

Nell'esempio seguente la variabile `harmonics` contiene il numero delle armoniche. L'array `phases` è invece creato attraverso il metodo `fill` (3) che prevede di specificare una dimensione (qui `harmonics`) e una funzione che viene applicata per il calcolo di ognuno degli elementi. Qui la funzione è `{2pi.rand}`, che specifica di scegliere un valore casuale tra $[0, 2\pi]$, cioè una fase casuale nell'intero ciclo. Le ampiezze delle componenti sono tutte pari a 1 (4) e stoccate in `amplitudes`. Infine, `sig` è generato come somma di sinusoidi armoniche con la stessa ampiezza ma fase casuale.

```
var sig, harmonics = 50 ;
//var phases = Array.fill(harmonics, {0});
var phases = Array.fill(harmonics, {2pi.rand});
var amplitudes = Array.fill(harmonics, {1}) ;
sig = Signal.sineFill(1024, amplitudes, phases) ;
sig.plot ;
```

In questo caso, il valore d'ampiezza di ognuna delle componente è pari a 1 ma generato dalla funzione `{1.0.rand}` ed è un valore pseudo-casuale compreso tra $[0.0, 1.0]$ (tra assenza e massimo valore normalizzato). Se si esegue il

codice più volte si noterà che il suono cambia, poiché l'apporto delle componenti dipende dalla funzione del metodo `fill`. Se si incrementa o decrementa il valore di `partials` il segnale rispettivamente si arricchisce o si impoverisce di componenti elevate. La riga 7 permette di visualizzare l'array delle ampiezze come una spezzata che unisce i valori discreti che lo compongono: con `plot` si possono specificare i valori d'escursione massimi e minimi ($[0, 1]$). Si tratta, come si vedrà, di una forma di sintesi additiva. In Figura **4.9**, in basso, sono riportate le fasi delle componenti (casuali nell'escursione $[0, 2\pi]$) e il segnale risultante. Si noti che il contenuto armonico è uguale ma la forma d'onda è radicalmente diversa. Se si esegue il codice più volte si noterà che la forma d'onda cambia, poiché l'apporto delle componenti dipende dalla funzione stocastica definita nel metodo `fill` per le fasi.

L'introduzione dei numeri pseudo-casuali permette di avvicinare anche i segnali non periodici. Un rumore bianco è un segnale il cui comportamento non è predicibile se non in termini statistici. Questo comportamento si traduce in una distribuzione uniforme dell'energia su tutto lo spettro del segnale. Un rumore bianco può essere descritto come una variazione totalmente aperiodica nel tempo: in altre parole, il valore di ogni campione è del tutto indipendente da quelli precedenti e da quelli successivi[5]. Dunque, il valore di un campione x è indipendente dal valore del campione precedente $x - 1$: x può avere qualsiasi valore, sempre, evidentemente, all'interno dello spazio finito di rappresentazione dell'ampiezza. Intuibilmente, l'algoritmo di generazione è molto semplice. In pseudo-codice:

```
Per ogni x in A:
       y = a*rand(-1, 1)
       A[x] = y
```

Nell'esempio seguente, il metodo `fill` (6) valuta per ogni campione `x` una funzione che restituisce un valore casuale all'interno dell'escursione normalizzata $[-1, 1]$ (`rrand(-1.0,1.0)`). Il codice specifica una frequenza di campionamento (`sr`) e una durata in secondi (`dur`). Il risultato (un secondo di rumore bianco) viene scalato per `amp`, e dunque, dato un valore di 0.75 per `amp`, si otterrà un'oscillazione (pseudo-)casuale nell'escursione $[-0.75, 0.75]$. Si noti come la funzione (nella definizione di partenza $f[x]$) in realtà sia calcolata indipendente da x: essendo totalmente aperiodica, il suo comportamento non dipende dal tempo.

[5] Si dice perciò che ha autocorrelazione $= 0$.

```
var sig, amp = 0.75, dur = 1, sr = 44100 ;
sig = Signal.fill(dur*sr, { amp*rrand(-1.0,1.0) }) ;
sig.plot(minval:-1, maxval:1) ;
```

I software rappresentano tipicamente il segnale audio attraverso una curva continua che connette i valori d'ampiezza. Nel caso del rumore, la rappresentazione continua è decisamente meno icastica di una rappresentazione discreta (Figura **4.10**), in cui chiaramente emerge l'idea di una sorta di nuvola di punti distribuiti pseudo-casualmente.

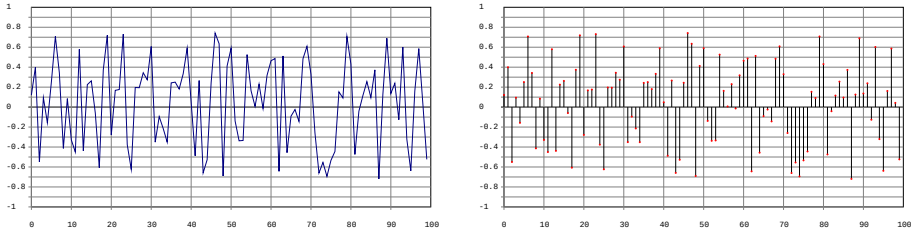


Fig. 4.10 Rumore bianco: curva continua e dispersione dei valori.

È un'ovvietà ma è bene ricordarsi che un segnale digitale è appunto semplicemente una sequenza di valori, sulla quale è possibile svolgere operazioni matematiche. Tutta l'elaborazione digitale del segnale nasce da quest'assunto. I prossimi sono due esempi tratti da Puckette[6]. Una funzione periodica, ad esempio al solito la funzione seno, oscilla periodicamente tra $[-1, 1]$. Calcolando il suo valore assoluto semplicemente si ribalta la parte negativa sul positivo: se si osserva la curva si nota come i due emicicli siano identici: ne consegue un segnale con frequenza doppia di quello di partenza. In generale l'applicazione della funzione del valore assoluto è una tecnica rapida per far saltare d'ottava un segnale. Poiché la curva occupa soltanto valori positivi (tra $[0, 1]$ in caso di valori normalizzati), è possibile traslarla in modo da evitare un offset rispetto allo zero (il segnale da unipolare ritorna bipolare): decrementando di 0.5 l'escursione del segnale diventa $[-0.5, 0.5]$ ed è simmetrica tra positivo e negativo (Figura

[6] M. Puckette, *The Theory and Technique of Electronic Music*, accessibile on line: `http://msp.ucsd.edu/techniques.htm`.

4.11). Nell'esempio seguente, il metodo `abs` invocato su un oggetto `Signal` restituisce un altro oggetto `Signal` in cui ogni elemento è il risultato del valore assoluto dell'elemento del segnale di partenza (è come chiamare `abs` per ogni elemento). Lo stesso vale ovviamente per la sottrazione e per la moltiplicazione: si applicano a tutti gli elementi dell'array. La sinusoide viene dunque moltiplicata per amp e traslata verso il negativo di $\frac{amp}{2}$, qui implementato con `amp*0.5`[7]. La situazione è rappresentata in Figura **4.11**.

```
1 var sig, amp = 0.75 ;
2 sig = (Signal.sineFill(100, [1]).abs)*amp-(amp*0.5) ;
3 sig.plot(minval:-1, maxval: 1) ;
```

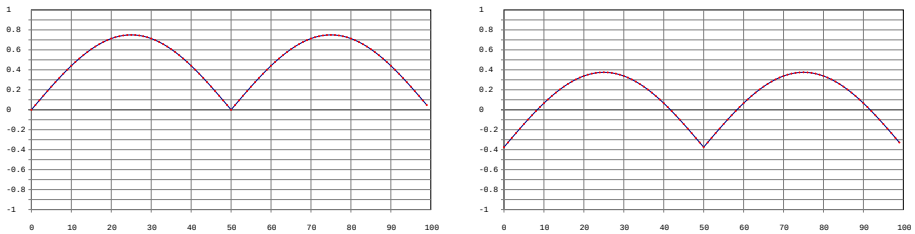


Fig. 4.11 Valore assoluto della funzione seno e traslazione ($A = 0.75$).

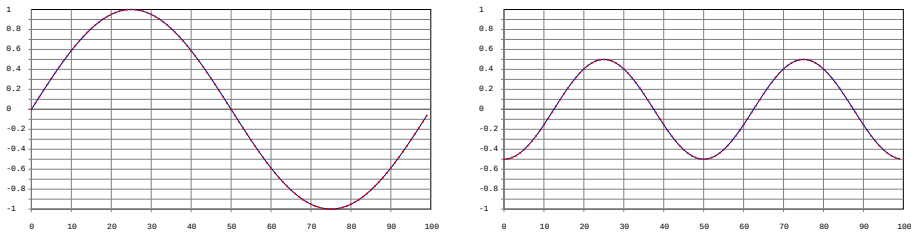


Fig. 4.12 $y = sin(2\pi \times x)^2, y = sin(2\pi \times x)^2 - 0.5$

[7] È una tipica prassi informatica. La moltiplicazione costa, in termini computazionali, molto meno della divisione.

Una caratteristica dell'applicazione del valore assoluto è la generazione di un segnale asimmetrico rispetto all'ampiezza, e la presenza di uno spigolo molto acuto (il punto di ribaltamento) che introduce molte componenti armoniche superiori. L'utilizzo dell'elevamento al quadrato evita questa caratteristica ottenendo un risultato analogo: infatti il quadrato dei valori negativi del segnale del segnale è positivo, e si ottiene un ribaltamento della curva sostanzialmente analogo a quanto avviene con il valore assoluto (Figura **4.12**).

```
var sig, amp = 0.75 ;
sig = (Signal.sineFill(100, [1]).squared)*amp-(amp*0.5) ;
sig.plot(minval:-1, maxval: 1) ;
```

Come ulteriore esempio di elaborazione del segnale si può considerare l'operazione di "clippping": in caso di clipping tutti i valori superiori ad una certa soglia s (o inferiori al negativo della stessa) vengono riportati a $\pm s$. Il clipping prende anche il nome di "distorsione digitale" perché è quanto avviene quando il segnale digitalizzato ha un'ampiezza superiore a quella rappresentabile dalla quantizzazione e viene perciò "tagliato" agli estremi. Il clipping è una sorta di limiter (limita infatti l'ampiezza del segnale) radicale e può essere usato come "effetto" di tipo distorcente. In SC è definito il metodo `clip2(t)` che "taglia" un valore fuori dell'escursione $[-t, t]$ a $\pm t$. Si consideri l'esempio seguente, in cui $t = 3$.

```
1.clip2(3)
1

-1.clip2(3)
-1

4.clip2(3)
3

-4.clip2(3)
-3
```

Anche se `clip2` è definito per `Signal`, come esercizio può essere interessante implementare un modulo "clipper".

```
1  var sig, sig2, sig3, clipFunc ;
2  sig = Signal.sineFill(100, [1]) ;
3
4  clipFunc = { arg signal, threshold = 0.5 ;
5      var clipSig = Signal.newClear(signal.size) ;
6      signal.do({    arg item, index;
7          var val ;
8          val = if (item.abs < threshold, { item.abs },
9              { threshold}) ;
10         val = val*item.sign ;
11         clipSig.put(index, val) ;
12     }) ;
13     clipSig ;
14 } ;
15
16 sig2 = clipFunc.value( sig ) ;
17 sig3 = clipFunc.value( sig, threshold: 0.75 ) ;
18
19 [sig, sig2, sig3].flop.flat.plot
20      (minval:-1, maxval: 1, numChannels:3) ;
```

Si è detto che una funzione esegue un comportamento quando riceve il messaggio `value` in funzione degli argomenti in entrata. Nell'esempio è appunto definita una funzione che implementa il clipping, `clipFunc`: gli argomenti in entrata prevedono un segnale (un oggetto `Signal`) e un valore di soglia (`threshold`). Ciò che `clipFunc` restituisce è un altro oggetto `Signal`. La riga 5 dichiara la variabile a cui viene assegnato il segnale e gli assegna subito un oggetto `Signal` di dimensione pari a quella del segnale in entrata, riempito di 0. L'idea è quella di valutare ognuno degli elementi (che rappresentano campioni audio) di `signal`: si tratta cioè di ciclare su `signal` (6). Nel ciclo la variabile `val` rappresenta il valore che dovrà essere scritto nell'array `clipSig`. Il valore da assegnare a `val` dipende da una condizione. Se il campione valutato è all'interno dell'escursione $[-threshold, threshold]$ allora non ci sarà clipping: `val` ha lo stesso valore del valore di `signal` in entrata. Se invece il valore in entrata cade all'esterno dell'intervallo, allora in uscita si avrà il valore della soglia stessa. Il condizionale è contenuto nelle righe 8-9. La valutazione avviene sul valore assoluto di `item` (`item.abs`). Avviene cioè nell'intervallo $[0, 1]$. In uscita si ha lo stesso valore assoluto di `item` oppure `threshold`. Se il valore in entrata è negativo, verrebbe così restituito un valore positivo. Il segno del valore in entrata

viene però recuperato con la riga 10 in cui il valore `val` viene moltiplicato per il segno di `item`: infatti, `item.sign` restituisce ± 1.

Se `item = -0.4` si ha:

- `item.abs = 0.4`
- è inferiore a `threshold = 0.5`? Sì, allora `val = item.abs = 0.4`
- `item.sign = -1` (`item = -0.4`: è negativo)
- `val = 0.4 * -1 = -0.4`

Seguono due esempi, uno che sfrutta il valore predefinito di `threshold` (0.5), l'altro in cui `threshold` vale 0.75. Il clipping tende a "squadrare" la forma d'onda ed in effetti il segnale risultante tende ad assomigliare –anche all'ascolto– ad un'onda quadra. Si notino anche le righe 19-20. Il metodo `plot` è definito per un array, ma prevede l'argomento `numChannels`, che tratta la sequenza di valori come se fosse un insieme interallacciato di valori di più canali audio. Quindi, con `numChannels = 3` vengono disegnate tre curve, prendendo in successione tre campioni come fossero valori della prima, della seconda, della terza. Si tratta di costruire un array opportuno perché possano essere disegnati i tre array di partenza. Con il metodo `flop` un array di array viene riconfigurato così: nel nostro caso, si passa da tre array di cento valori a cento array di tre valori. Il metodo `flat` appiattisce l'array che a questo punto può essere disegnato specificando che i canali sono tre. Si inseriscano gli opportuni messaggi `postln` per capire analiticamente la situazione. Il risultato è in Figura **4.13**.

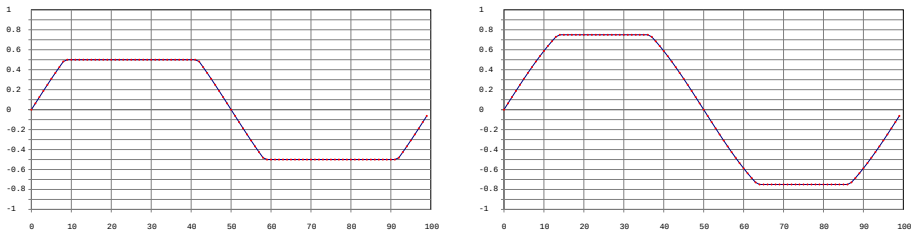


Fig. 4.13 Clipping di una sinusoide, `threshold = 0.5`, `threshold = 0.75`.

Esistono altri modi ben più eleganti di implementare `clipFunc`. L'esempio successivo utilizza il metodo `collect`, che, insieme ad altri metodi analoghi, è ereditato dalla classe `Collection`. Questi metodi, chiamati per un oggetto di tipo collezione, restituiscono una nuova collezione applicando in modo diverso per ogni metodo, una funzione per ogni elemento. Se l'approccio precedente era tipico del paradigma imperativo, questi metodi invece hanno una vocazione

funzionale (e sono molto eleganti, cioè chiari e sintetici). Nell'esempio seguente, `collect` applica ad ogni elemento la funzione definita al suo interno.

```
clipFunc = { arg signal, threshold = 0.5 ;
    signal.collect({ arg item;
        var val, sign ;
        sign = item.sign ;
        val = if (item.abs < threshold, { item.abs },
            { threshold}) ;
        val*sign ;
    }) ;
} ;
```

Una funzione restituisce il valore dell'ultima espressione che la compone. Qui c'è un'unica espressione, che restituisce un `Signal` con l'applicazione della funzione (analoga all'implementazione precedente) ad ogni elemento. Si noti che interessava mantenere la modularità di `clipFunc` ma si potrebbe direttamente chiamare `collect` qui definito sull'oggetto `Signal` di partenza.

L'approccio modulare dell'esempio precedente permette di riscrivere i casi relativi al valore assoluto e al quadrato. In particolare si potrebbero esplicitamente definire due *funzioni di trasferimento*:

$$W_{abs} : f(x) = x^2$$

$$W_{square} : f(x) = |x|$$

Queste funzioni si comportano come veri moduli di elaborazione del segnale in entrata. Una versione modulare della funzione valore assoluto, scritta in maniera molto compatta (si noti l'uso di `|`) è la seguente:

```
absFunc = {|sig| sig.collect{|i| i.abs} - (sig.peak*0.5) } ;
```

Nel codice l'unica cosa di rilievo è l'eliminazione automatica dell'offset. Il metodo `peak` restituisce l'elemento di `Signal` che ha il valore assoluto più alto (il massimo d'ampiezza del segnale). Assumendo che il segnale sia simmetrico rispetto allo 0, ogni nuovo valore risultante dall'applicazione della funzione

viene traslato di $peak/2$. Se il segnale è ad esempio compreso in $[-0.7, 0.7]$ allora $peak = 0.7$: il segnale trasformato sarà compreso in $[0.0, 0.7]$ e viene traslato di $0.7/2 = 0.35$, così che la sua escursione in ampiezza risulti simmetricamente intorno allo 0 in $[-0.35, 0.35]$.

4.6 Ancora sull'elaborazione di segnali

La natura numerica del segnale permette di definire operazioni analoghe alle precedenti anche su materiale pre-esistente. La classe `SoundFile` permette non soltanto di scrivere sui file ma anche di accedere a file audio disponibili sull'hard-disk. Il codice seguente riempie l'oggetto `Signal` `sig` con il contenuto del file audio `sFile` attraverso il metodo `readData` di `sFile`. Il file è uno degli esempi che vengono forniti con SC e vi si accede attraverso `Platform.resourceDir`, una classe che permette di gestire percorsi di file indipendentemente dalla piattaforma su cui si ta operando (OSX, Windows, Linux). Si noti come la dimensione di `sig` sia stata fatta dipendere dal numero di campioni di `sFile` attraverso l'attributo `numframes`.

```
1 var sFile, sig ;
2 sFile = SoundFile.new;
3 sFile.openRead(Platform.resourceDir +/+ "sounds/a11wlk01.wav");
4 sig = Signal.newClear(sFile.numFrames) ;
5 sFile.readData(sig) ;
6 sFile.close;
7 sig.plot ;
8 sig.plot ;
```

L'operazione seguente sfrutta la natura di sequenza numerica del segnale audio per implementare una sorta di granulazione (si veda dopo, nel caso). In sostanza il segnale audio importato viene suddiviso in un numero `numChunks` di pezzi, ognuno dei quali comprende un numero `step` di campioni (13, qui 500, 12). Quindi i pezzi vengono mischiati a caso e rimontati. Nell'implementazione `indices` è la sequenza degli indici dei pezzi del segnale ed è una progressione a partire da 0 (17). Questa progressione lineare $(1, 2, 3, 4...)$ viene mescolata attraverso il metodo `scramble` (così da diventare ad esempio $9, 1, 3, 7...$). Quindi,

attraverso il ciclo su ognuno degli indici, viene recuperato il pezzo corrispondente nel segnale originale e concatenato in sequenza. Il metodo `copyRange` copia da un array in quello su cui è invocato un insieme di elementi specificato dagli argomenti che rappresentano gli indici di partenza e fine. È probabile che lo `step` non sia un divisore intero del segnale in ingresso. La parte che avanza (`tail`) viene concatenata alla fine del nuovo segnale `newSig`.

```
var sFile, sig, newSig ;
var numChunks, step, rest, indices ;
var block, tail ;

sFile = SoundFile.new;
sFile.openRead
   (Platform.resourceDir +/+ "sounds/a11wlk01-44_1.aiff");
sig = Signal.newClear(sFile.numFrames) ;
sFile.readData(sig) ;
sFile.close;

step = 500 ; // provare: 10, 200, 1000
numChunks = (sig.size/step).asInteger ;
// la coda...
tail = (sig.size-(step*numChunks)) ;

indices = Array.series(numChunks).scramble;

newSig = Signal.new;
indices.do({arg item;
    block = sig.copyRange(item*step, (item+1)*step-1) ;
    newSig = newSig.addAll(block) ;

}) ;

tail = sig.copyRange(sig.size-tail, sig.size) ;
newSig = newSig.addAll(tail) ;
```

Questa forma di granulazione del segnale permette di introdurre infine una forma di sintesi per permutazione. Attraverso lo "scrambling" del segnale si produce un nuovo segnale che mantiene più o meno "in memoria" il segnale originale. Minore è lo `step` più grande è la ricombinazione del segnale. Se un'operazione simile viene svolta su un segnale sinusoidale si nota ancora più chiaramente la proporzione tra diminuzione dello `step` e incremento della rumorosità. Nell'esempio seguente viene implementata una forma di sintesi per

permutazione. Il processo è del tutto analogo a quanto avviene nell'esempio precedente, con una differenza. Qui la sinusoide è creata concatenando molte copie (100, cioè `44100/period`) del segnale `sig` (9-11). Il segnale viene sempre suddiviso in blocchi di durata `step`. Al posto dell'operazione di `scramble` viene invece implementata una permutazione di questo tipo:

$$[\ 0, 1, 2, 3, 4, 5\ ] \rightarrow [\ 1, 0, 3, 2, 5, 4\ ]$$

Sostanzialmente, coppie di blocchi vengono permutate. La permutazione degli indici è ottenuta in due passaggi. Prima si creano una serie pari e una dispari con passo pari a 2 e a partire da valori diversi (1 e 0) (18-19). Quindi, si interallacciano via `flop` e si ottiene l'array unico finale con `flat` (20).

Essendo la permutazione del tutto periodica si ottiene un segnale dallo spettro molto ricco ma che presenta una periodicità che dipende sia dalla frequenza della sinusoide in entrata sia dalla dimensione di `step`.

```
/* Distorsione per permutazione su una sinusoide */

var sig, sig2, newSig ;
var numChunks, step, rest, indices ;
var block, tail ;
var period = 441;

// creazione della sinusoide
sig = Signal.new ;
sig2 = Signal.sineFill(period, [1]) ;
(44100/period).do(sig = sig.addAll(sig2)) ;

step = 50 ; // provare altri passi
numChunks = (sig.size/step).asInteger ;
tail = (sig.size-(step*numChunks)) ;

// creazione della sequenza di indici
a = Array.series((numChunks/2).asInteger, 1,2) ;
b = Array.series((numChunks/2).asInteger, 0,2) ;
indices = [a,b].flop.flat ;

newSig = Signal.new;
indices.do({ arg item;
    block = sig.copyRange(item*step, (item+1)*step-1) ;
    newSig = newSig.addAll(block) ;
}) ;

tail = sig.copyRange(sig.size-tail, sig.size) ;
newSig = newSig.addAll(tail) ;
```

La periodicità diventa evidente, oltre all'orecchio, se si osserva la sinusoide distorta per permutazione (Figura **4.14**).

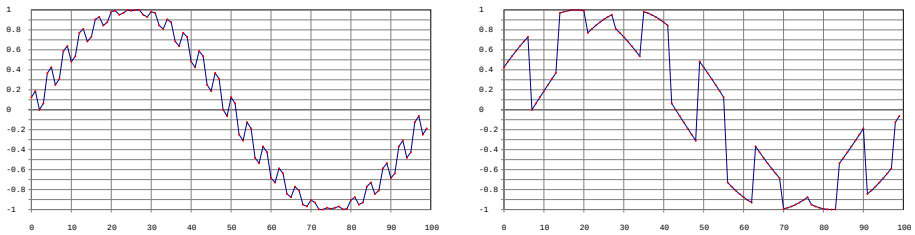


Fig. 4.14 Sinusoide di 100 punti e scrambling con passo $= 3$ e $= 7$

4.7 Segnali di controllo

Un segnale sinusoidale, come tutti i segnali perfettamente periodici, manca totalmente delle caratteristiche di dinamicità usualmente ascritte ad un suono "naturale" o, meglio, "acusticamente interessante": è un suono, come direbbe Pierre Schaeffer, senza "forma" temporale, "omogeneo". Suoni di questo tipo, peraltro, arredano il paesaggio sonoro della modernità meccanica ed elettrica sotto forma di *humming*, *buzzing* -e così via- prodotti dai ventilatori, dall'impedenza elettrica, dai motori. A parte questi casi, tipicamente la "forma" temporale di un suono prende la forma di un profilo dinamico, di una variazione della dinamica del suono che è descritta acusticamente sotto forma di una curva di inviluppo, le cui fasi prendono usualmente il nome di attacco/decadimento/sostegno/estinzione: da cui l'acronimo ADSR (*attack, decay, sustain, release*). Il modo più semplice di rappresentare un simile inviluppo consiste nell'utilizzare una spezzata (Figura **4.15**).

A guardare l'inviluppo si osserva agevolmente come si tratti di altra curva, e cioè propriamente di un altro segnale, che si distingue dai segnali finora considerati per due caratteristiche importanti:

1. non è periodico (si compone di un unico ciclo). Dunque, il periodo del segnale d'inviluppo è pari alla durata del segnale audio: se il segnale dura 2 secondi (ovvero il periodo dell'inviluppo) allora l'inviluppo ha una frequenza $1/2 = 0.5Hz$. Si noti come la frequenza non rientri nel dominio udibile;

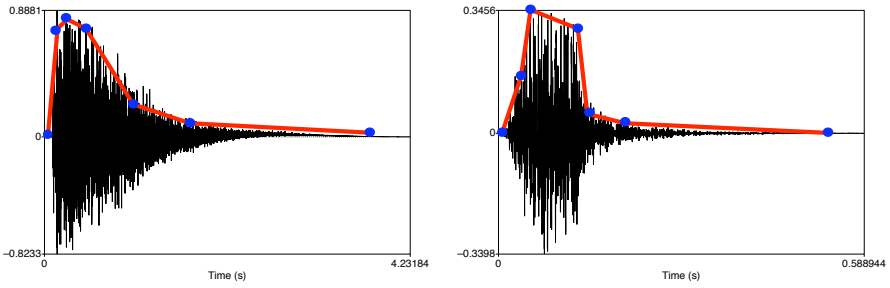


Fig. 4.15 Descrizione del profilo dinamico attraverso una spezzata (inviluppo).

2. è unipolare: assumendo che il segnale audio sia nell'escursione normalizzata $[-1, 1]$ (bipolarità), l'inviluppo è compreso in $[0, 1]$ (unipolarità)[8].

Passando dall'analisi del segnale alla sua sintesi, si tratterà perciò di riprodurre le proprietà dell'inviluppo (la sua "forma") e di applicare questa forma al segnale audio. L'inviluppo è un tipico *segnale di controllo*: un segnale che modifica -controlla- un segnale audio.

Essendo un segnale, per rappresentare un inviluppo si può utilizzare un array. Ad esempio, un tipico inviluppo ADSR potrebbe essere descritto dall'array di dieci punti di Figura **4.16**.

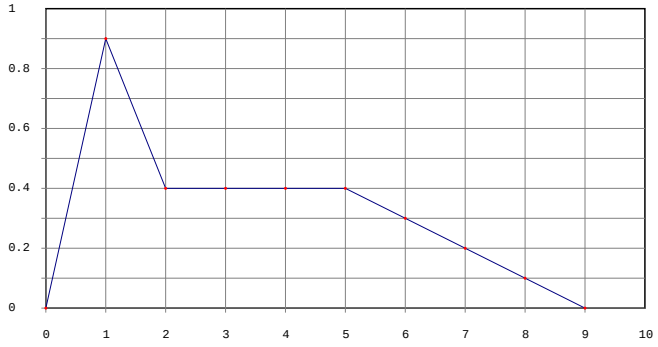


Fig. 4.16 [0, 0.9, 0.4, 0.4, 0.4, 0.4, 0.3, 0.2, 0.1, 0].

[8] Si ricordi il caso della funzione valor assoluto che rende unipolare un segnale bipolare.

Detto interattivamente:

```
1 [0, 0.9, 0.4, 0.4, 0.4, 0.4, 0.3, 0.2, 0.1, 0].plot
```

Per incrementare o decrementare l'ampiezza di un segnale si può (lo si è visto abbondantemente) moltiplicare il segnale per una costante: ogni campione viene moltiplicato per la costante. Ad esempio, se la costante $Amp = 0.5$, l'ampiezza del segnale viene ridotta della metà. Si potrebbe pensare ad una simile costante nei termini di un segnale: come un array, di dimensione pari a quella del segnale scalato, che contenga sempre lo stesso valore. Ogni valore dell'array da scalare viene moltiplicato per il rispettivo valore dell'array Amp (sempre lo stesso). L'idea smette di essere una complicazione inutile nel momento in cui l'array Amp non contiene più sempre lo stesso valore, ma contiene invece valori variabili che rappresentano appunto un inviluppo d'ampiezza. Dunque, ogni campione del segnale audio (ogni elemento dell'array) viene moltiplicato per un valore incrementale del segnale d'inviluppo ottenuto. La situazione è rappresentata in Figura **4.17**.

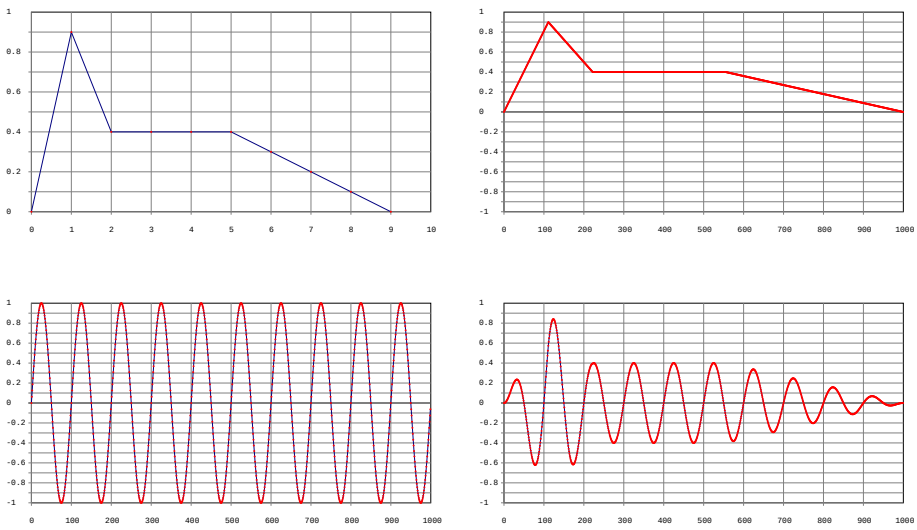


Fig. 4.17 Inviluppo, segnale audio risultante, segnale audio, segnale audio inviluppato.

Dovendo descrivere lo sviluppo del segnale audio in tutta la sua durata, la dimensione dell'array d'inviluppo deve essere la stessa del segnale audio. Emerge qui un punto fondamentale, su cui si ritornerà: tipicamente è sufficiente un numero molto minore di punti per rappresentare un inviluppo rispetto ad un segnale audio, ovvero -assumendo che il segnale audio duri un solo secondo a qualità CD- il rapporto tra i due è pari a $10/44.100$. Questo è in effetti il tratto pertinente che identifica un segnale di controllo.

Passando all'implementazione in SC (qui di seguito), per intanto si può generare una sinusoide di durata pari ad un secondo attraverso il metodo `waveFill`. La frequenza prescelta è grave perché così sono meglio visibili i cicli del segnale. L'array di partenza che si vuole implementare è composto da dieci segmenti: poiché questi devono essere ripartiti sulla durata del segnale audio, ognuno dei segmenti comprenderà $44100/10$ campioni. Questo valore è assegnato alla variabile `step`. Si tratta ora di generare gli array che costituiscono i quattro segmenti ADSR (gli array `att, dec, sus, rel`, 8-11), per poi concatenarli in un array complessivo `env`. Poiché devono essere moltiplicati per un oggetto `Signal` allora devono anch'essi essere oggetti `Signal`. Il metodo `series(size, start, step)` crea una progressione lineare di `size` valori a partire da `start` con incremento `step`. L'attacco parte da 0 e arriva a 0.9, occupa un segmento, cioè 4410 campioni. Il valore 0.9 deve cioè essere raggiunto in 4410 punti: l'incremento di ognuno sarà perciò di $0.9/4410$. Dunque, `0.9/step`. All'ultimo punto si avrà un valore pari a $0.9/step \times step = 0.9$. Nel caso di `dec` si tratta di scendere a 0.5 nella durata di un altro segmento. In questo caso la progressione parte da 0.9 e l'incremento è negativo: bisogna scendere di 0.5 in uno step, dunque l'incremento è `-0.5/step`. Nel terzo caso (`sus`) il valore è costante a 0.4 per una durata di 4 passi, dunque si può pensare ad un incremento 0. Il caso di `rel` è analogo a quello di `dec`. Il nuovo segnale di inviluppo `env` (13) è ottenuto concatenando i quattro segmenti ed è impiegato come moltiplicatore di `sig` per ottenere il segnale inviluppato `envSig` (14). La riga 16 disegna segnale, inviluppo e segnae inviluppato.

```
var sig, freq = 440, size = 44100, step ;
var env, att, dec, sus, rel, envSig ;

step = 44100/10 ;
sig = Signal.newClear(size) ;
sig.waveFill({ arg x, i; sin(x) }, 0, 2pi*50) ;

att = Signal.series(step, 0, 0.9/step) ;
dec = Signal.series(step, 0.9, -0.5/step) ;
sus = Signal.series(step*4, 0.4, 0) ;
rel = Signal.series(step*4, 0.4, -0.4/( step*4)) ;

env = att++dec++sus++rel ;
envSig = sig * env ;

[sig, env, envSig].flop.flat.plot(minval:-1, maxval: 1, numChannels: 3) ;
```

Il metodo precedente è molto laborioso, e del tutto teorico in SuperCollider. Quest'ultimo prevede infatti una classe `Env` specializzata nel costruire inviluppi. `Env` assume che un inviluppo sia una spezzata che connette valori d'ampiezza nel tempo e fornisce diverse modalità di interpolazione per i valori intermedi. Si considerino i due array seguenti `v` e `d`:

```
v  = [0,   1,     0.3,     0.8,     0] ;
d  = [   2,    3,       1,       4   ] ;
```

Una coppia simile è tipicamente utilizzata per specificare un inviluppo:

- `v`: indica i punti che compongono la spezzata (i picchi e le valli);
- `d`: indica la durata di ogni segmento che connette due punti.

Dunque, l'array delle durate contiene sempre un valore di durata in meno di quello delle ampiezze. Infatti, `t[0]` ($= 2$) indica che per andare da `v[0]` ($= 0$) a `v[1]` ($= 1$) sono necessari 2 unità di tempo (in SC: secondi, ma in realtà si tratta come si vedrà di unità astratte). Attraverso i due array `v`, `d` è così possibile descrivere un profilo (`v`) temporale (`d`). Nell'esempio, resta tuttavia da specificare cosa succede per ogni campione compreso nei due secondi che intercorrono

tra 0 e 1. Il modo in cui i campioni sono calcolati dipende dalla modalità din interpolazione. Nell'esempio seguente `e1, e2, e3` sono oggetti `Env` specificati dalla stessa coppia di array `v, d`, ma con differenti modalità di interpolazione (lineare, discreta, esponenziale). L'ultimo esempio, `e4`, dimostra un'ulteriore possibilità: un array di modalità di interpolazione che vengono applicate per ogni segmento (l'ultima modalità è `'sine'`).

```
1  var v, d, e1, e2, e3, e4 ;

3  v  = [0,   1,     0.3,     0.8,     0] ;
4  d  = [   2,    3,       1,       4   ] ;

6  e1 = Env.new(v, d,'linear') ;
7  e2 = Env.new(v, d,'step') ;

9  v  = [0.0001,   1,     0.3,     0.8,     0] ;
10 e3 = Env.new(v, d,'exponential').asSignal ;
11 e4 = Env.new(v, d,[\linear , \step ,\exponential , \sine ]) ;

13 [e1, e2, e3, e4].collect{|i| i.asSignal}.flop.flat.plot(numChannels:4) ;
```

Si noti che nel caso di un inviluppo esponenziale il valore di partenza non può essere pari a 0: il primo valore di `v` viene quindi ridefinito con un valore prossimo allo zero. Il significato dei parametri è illustrato nella Figura **4.18** che commenta quanto disegnato dal metodo `plot`. La classe `Env` eredita direttamente da `Object` e dunque non è un oggetto di tipo array. Tipicamente viene utilizzata come specificazione di inviluppo per il tempo reale (come si vedrà). Quando però un oggetto `Env` riceve il messaggio `asSignal`, esso restituisce un oggetto `Signal` che contiene un inviluppo campionato nel numero di punti che compongono il nuovo array. Il metodo è utilizzato (13) per ottenere array dagli inviluppi in modo da utilizzare la tecnica di plotting già discussa.

Attraverso `asSignal` la classe `Env` permette di utilizzare anche in tempo differito una specificazione per gli inviluppi decisamente più comoda. Inoltre, la classe prevede alcuni costruttori che restituiscono inviluppi particolarmente utili. Due esempi:

- `triangle` richiede due argomenti: il primo indica la durata, il secondo il valore di picco di un inviluppo triangolare (il picco cade cioè a metà della durata).

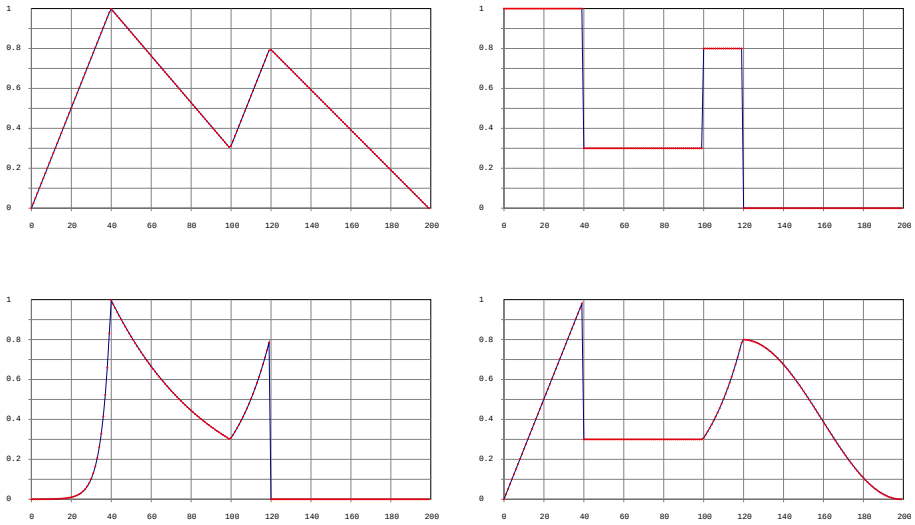


Fig. 4.18 `Env`: tipi di interpolazione.

- `perc`: permette di definire un inviluppo percussivo (attacco + rilascio). Gli argomenti sono tempo d'attacco, tempo di rilascio, valore di picco e valore di curvatura.

```
var sig, freq = 440, size = 1000 ;
var envT, envP ;

sig = Signal.newClear(size) ;
sig.waveFill({ arg x, i; sin(x) }, 0, 2pi*50) ;

envT = Env.triangle(1,1).asSignal(size);
envP = Env.perc(0.05, 1, 1, -4).asSignal(size) ;

[sig, envT, sig*envT, envP, sig*envP].flop.flat
   .plot(minval:-1, maxval: 1, numChannels: 5) ;
```

Inviluppi di tipo `triangle` e `perc`, e le loro applicazioni a una sinusoide sono rappresentati in Figura **4.19**.

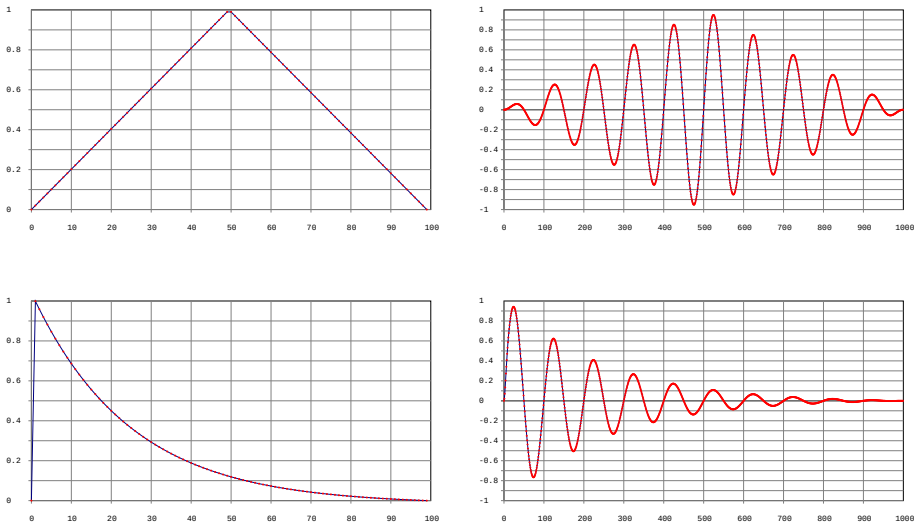


Fig. 4.19 Inviluppi con `Env`, di tipo `triangle` (alto) e `perc` (basso).

L'applicazione di un segnale di inviluppo è ovviamente possibile anche per un segnale audio di provenienza concreta. Nell'esempio seguente il segnale `sig` è ottenuto importando il contento di `sFile`, e viene normalizzato in modo che il suo picco sia pari a 1 (6). A `sig` viene applicato un segnale di inviluppo `env`: `env` è ottenuto attraverso due array riempiti di numeri pseudo-casuali, `v` e `d`. Il primo oscilla nell'intervallo $[0.0, 1.0]$ (è un segnale unipolare). Per evitare offset nell'ampiezza, il primo e l'ultimo valore dell'array vengono posti a 0, ed aggiunti in testa e in coda (11). L'array `d` è composto di un numero di elementi pari a quelli di `v` -1, secondo quanto richiesto dalla sintassi di `Env`. Gli intervalli di durata variano nell'intervallo $[0.0, 4.0]$ (12).

```
var env, v, d, breakPoints = 10 ;
var sig, sFile ;

sFile = SoundFile.new;
sFile.openRead(Platform.resourceDir+/+"sounds/a11wlk01-44_1.aiff");
sig = Signal.newClear(sFile.numFrames).normalize ;
sFile.readData(sig) ;
sFile.close;

v = Array.fill(breakPoints-2, { arg i ; 1.0.rand }) ;
v = v.add(0) ; v = [0].addAll(v) ;
d = Array.fill(breakPoints-1, { arg i; 4.0.rand }) ;

env = Env(v, d, 'lin').asSignal(sig.size) ;

[sig, env, sig*env].flop.flat.plot(minval:-1, maxval: 1, numChannels: 3) ;
```

Due inviluppi pseudo-casuali generati dal codice sono disegnati in Figura **4.20**: ad ogni valutazione del codice l'inviluppo assume infatti una forma diversa, a parte per i due estremi pari a 0.

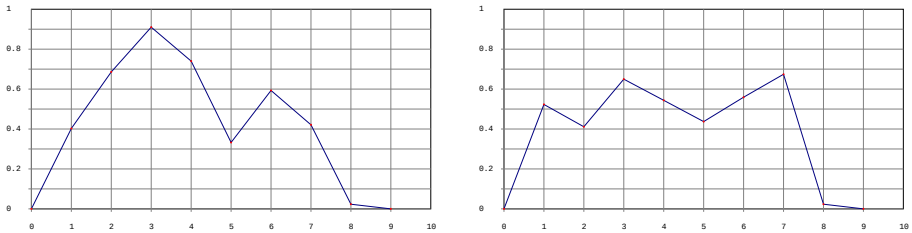


Fig. 4.20 Inviluppi pseudo-casuali.

4.8 Conclusioni

L'obiettivo di quanto visto finora era di introdurre il concetto di segnale digitale, attraverso alcune operazioni che si rendono tipicamente possibili grazie

alla sua natura numerica. Si è poi avuto modo di osservare come sia possibile modificare un segnale attraverso un altro segnale. In particolare, un segnale di controllo è un segnale che richiede una risoluzione temporale molto minore del segnale e la cui frequenza di situa al di sotto delle frequenze udibili ("sub-audio range"). Un segnale di controllo tipicamente modifica un segnale audio. Ma il capitolo è stato anche l'occasione per approfondire alcuni aspetti linguistici di SuperCollider. È a questo punto opportuno riprendere ed espandere gli aspetti affrontati attraverso il *modus operandi* più tipico di SC, il tempo reale.

5 L'architettura e il server

Finora, di SuperCollider si è discusso di tutto, a parte di quello per cui è massimamente famoso, la sintesi e l'elaborazione del segnale audio. È perciò tempo di affrontare la questione, ma come al solito è necessaria una certa pazienza per mettere insieme tutti i pezzi necessari.

5.1 Client vs. server

Come si è avuto modo di osservare, scaricando il programma SC ci si porta a casa due componenti, di principio autonomi, un server e un client. Il primo viene chiamato *scsynth*, il secondo *sclang*. Quest'ultimo è l'interprete del linguaggio SuperCollider che è stato discusso nei precedenti capitoli. Ma funziona anche come client del server audio. Riassumendo, il server audio è un fornitore di servizi audio, cioè è capace di generare e gestire in vario modo processi di sintesi ed elaborazione del segnale: è un fornitore perché questi servizi devono essere richiesti. E sono richiesto da clienti. Questo tipo di organizzazione software è definita "architettura client/server". In Figura **5.1** è descritta una generica architettura di rete: più client comunicano con un server scambiando messaggi.

Il programma SC sfrutta perciò un'architettura client/server, separando due funzioni, una di richiesta e l'altra di fornitura servizi. In SC il client e il server comunicano attraverso una rete (per la precisione, secondo il protocollo di rete UDP) attraverso messaggi scritti in uno specifico protocollo di comunicazione (un "codice" conosciuto a entrambi), piuttosto usato nell'ambito delle

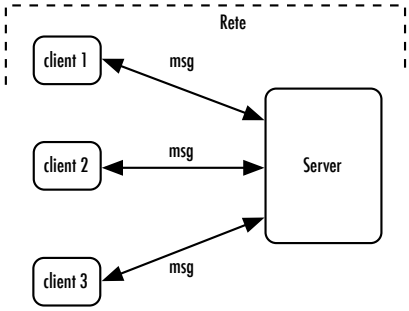


Fig. 5.1 Architettura client/server generica.

applicazioni multimediali (ad esempio, è implementato in Max/MS, PD, EyesWeb, Processing, etc.), che si chiama Open Sound Control (OSC)[1]. Il protocollo OSC, di per sé, non specifica una semantica, cioè non definisce il significato di un insieme chiuso di messaggi possibili, ma definisce invece una sintassi per costruire messaggi che si adeguino al protocollo. In questo modo, applicazioni ed utenti possono definire i propri messaggi, con una semantica a scelta. Nel caso di scsynth, quest'ultimo definisce un insieme di messaggi OSC che permettono di accedere a tutte le operazioni che scsynth rende possibile.

A scanso di equivoci, la rete di cui si parla è definita a livello astratto. Ciò vuol dire che client e server possono essere in esecuzione sulla stessa macchina. È ciò che avviene quando si manda in esecuzione l'*applicazione* SC: senza alcuna impostazione aggiuntiva, l'interprete è già configurato per comunicare in rete come client del server. Repetita iuvant. Aprendo SC si mandano in esecuzione due programmi, scsynth e sclang. Il server è un motore per la sintesi audio, di basso livello, potente, efficiente, e non molto intelligente (non ha capacità di programmazione). L'interprete sclang ha due funzioni:

1. è il client: in altre parole, è l'interfaccia che permette all'utente di scrivere e spedire messaggi OSC al server. Per scrivere una lettera al server, è necessario avere un foglio di carta e un postino che la consegni: sclang fa entrambe le cose.

[1] Un tutorial dettagliato su OSC è presente in AeM, cap. 7. Storicamente, James McCartney, il primo autore di SC, ha anche contribuito alla definizione del protocollo stesso.

2. è l'interprete del linguaggio: i messaggi OSC sono piuttosto macchinosi da scrivere, e condividono con il server la prospettiva di basso livello. Come abbondantemente visto, il linguaggio sclang è invece un linguaggio di alto livello. Il codice sclang, quando si rivolge al server, viene allora tradotto in messaggi OSC dall'interprete e questi vengono così inviati al server. La poesia che l'utente scrive in linguaggio sclang viene parafrasata in prosa OSC dall'interprete sclang per essere inviata al (prosaico) server.

La situazione è schematizzata in Figura **5.2**. La comunicazione tra lato client e lato server avviene attraverso messaggi OSC che il client spedisce al server. L'interprete sclang spedisce messaggi OSC in due modi:

1. **direttamente**. In altre parole, sclang-interprete è un buon posto per l'utente da dove parlare al server al livello di quest'ultimo (da dove spedire messaggi OSC);

2. **indirettamente**. Il codice del linguaggio SuperCollider (ad un livello più astratto) a disposizione dell'utente viene tradotto dall'interprete automaticamente in messaggi OSC (a livello server) per il server (è il cosiddetto *language wrapping*).

Riassumendo, a partire dall'applicazione sclang-interprete si può scrivere in poesia affidandosi alla traduzione prosastica ad opera dello stesso interprete (che è traduttore e postino) o direttamente in prosa OSC (l'interprete fa solo il postino). Ci si potrebbe chiedere perché complicarsi la vita con una simile architettura. I vantaggi sono i seguenti:

- **stabilità**: se il client sperimenta un crash, il server continua a funzionare (ovvero: l'audio non si ferma, ed è un fatto importante per un concerto/installazione/performance) e viceversa.
- **modularità**: un conto è la sintesi, un conto il controllo. Separare le due funzioni consente ad esempio di controllare scsynth anche da applicazioni che non siano sclang: l'importante è che sappiano spedire i giusti messaggi al

server. Il server è democratico (tutti possono ottenere servizi audio) e burocratico allo stesso tempo (l'importante è rispettare il protocollo OSC). Come si vede in Figura **5.2**, altre applicazioni posso inviare messaggi al server[2].

- **controllo remoto**: la rete di cui si parla può essere sia interna al calcolatore, sia esterna. Quando ci si occupa esclusivamente di sintesi audio, tipicamente le due componenti lavorano sullo stesso calcolatore attraverso un'indirizzo locale. Ma client e server potrebbero benissimo trovarsi via rete e, con qualche *caveat*, pure ai due estremi opposti del globo, comunicando via internet.

Gli svantaggi principali di una simile architettura sono due:

1. la circolazione dei messaggi introduce un piccolo ritardo (che può essere di rilievo però vista la sensibilità temporale dell'audio);
2. in caso di alta densità temporale dei messaggi sulla rete, quest'ultima può essere sovraccaricata, e la gestione dei messaggi può indurre un ritardo.

Va altresì notato che è decisamente raro incorrere in simili problemi.

Dalla discussione precedente risulta infatti chiaro che sclang è soltanto uno dei possibili client di scsynth. Tuttavia, sclang, essendo pensato esplicitamente per lavorare con scsynth, ne è in qualche modo il client privilegiato, poiché fornisce in maniera integrata all'utente un linguaggio di alto livello (altri livelli più astratti) e lo traduce per lui in messaggi OSC[3].

Come si è detto, il server può essere controllato (soltanto) attraverso messaggi OSC: espone una interfaccia di comandi che gli possono essere inviati. Tuttavia, il linguaggio SuperCollider definisce delle classi che rappresentano tutti gli oggetti del server con i relativi comandi OSC. Dunque, tutto quanto visto in precedenza può essere perfettamente sfruttato nell'interazione via programmazione con il server. Quindi, è opportuno sapere che c'è circolazione di

[2] Nel corso degli anni infatti sono stati scritti client per il server scsynth anche in altri linguaggi, ad esempio Clojure, Java, Scala. Programmatori esperti di altri linguaggi hanno così accesso ai servizi audio di scsynth senza dover necessariamente conoscere il linguaggio SC.

[3] In realtà, l'elemento per cui sclang è privilegiato è la formattazione delle SynthDef in modo che queste possano essere inviate al server, un'operazione che è possibile reimplementare in altri linguaggi ma che è oggettivamente macchinosa. Sulle SynthDef si tornerà abbondantemente dopo.

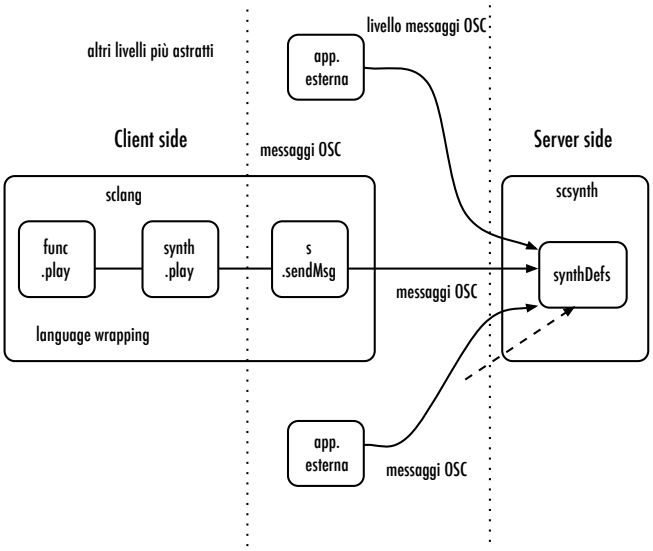


Fig. 5.2 Architettura client/server di SC.

messaggi OSC tra server e client. Ma salvo casi particolari, si può tranquillamente far finta di nulla.

5.2 Ontologia del server audio come impianto di sintesi

Dunque scsynth è un motore di sintesi audio programmabile e controllabile in tempo reale. Non è agevole di primo acchito riuscire a tener presente le relazioni tra tutti gli elementi pertinenti per il server audio scsynth. Infatti, ci sono aspetti puramente informatici (l'architettura client/server), altri tipici dell'audio digitale (campionamento, quantizzazione etc), altri tipici della computer music (il concetto di "strumento"), altri della musica elettronica tout court (molte tecniche di sintesi sono implementazioni digitali di metodi analogici, come pure l'idea di "patching"), altri ancora derivano dall'audio analogico ma sono invece propri della tecnica elettroacustica (il concetto di bus nei mixer, presente anche in digitale nelle Digital Audio Workstation). Conviene perciò

introdurre una quadro metaforico –un po' bislacco a dire il vero ma auspicabilmente utile– e pensare al server come ad un impianto chimico per la sintesi di liquidi. A partire da questa discussione può essere discussa preliminarmente l'ontologia del server, cioè l'insieme delle parti che lo compongono. Ognuno di esse verrà poi affrontata analiticamente. Nella discussione seguente si prenda in considerazione la figura **5.3**.

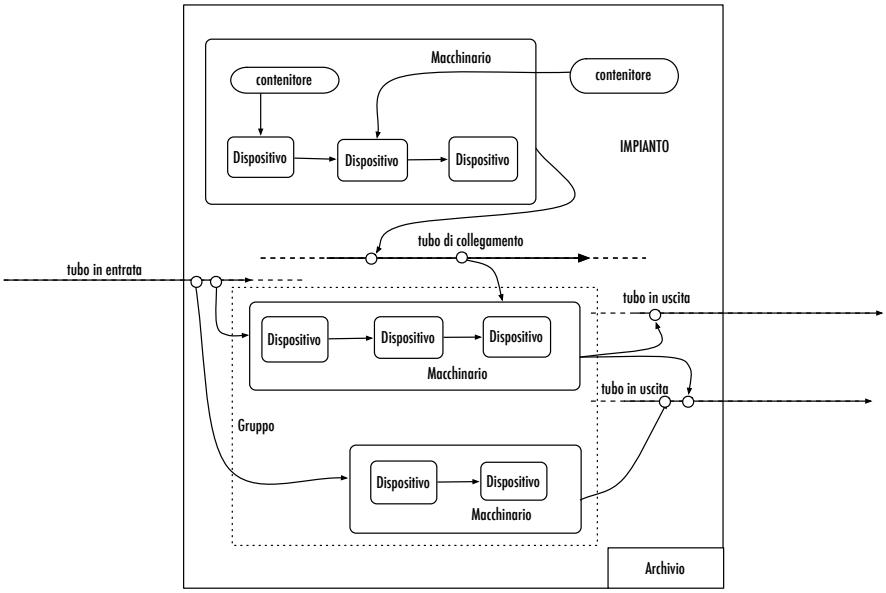


Fig. 5.3 Il server audio come impianto chimico per generazione di liquidi di sintesi.

1. L'intero processo di produzione è gestito non da un macchinario ma da un impianto;
2. Per sintetizzare un liquido è necessaria un macchinario complesso;
3. Un macchinario è costituito di dispositivi specializzati in cui i liquidi subiscono trasformazioni. I dispositivi sono collegati attraverso tubi interni;
4. Un macchinario deve essere progettato predisponendo le relazioni tra dispositivi componenti. A partire da un progetto, può essere costruito un numero indefinito di macchinari identici;
5. Una volta costruito, il macchinario non può essere modificato nella sua struttura interna;
6. Ma un addetto può controllarne il comportamento dall'esterno attraverso leve e comandi, così come monitorarne il funzionamento;

7. Un impianto può comprendere più macchinari che lavorano in parallelo;
8. Gruppi macchinari autonomi possono essere coordinati;
9. Quando l'impianto è in funzione i liquidi scorrono lungo i tubi a velocità costante, senza mai potersi fermarsi;
10. I liquidi possono scorrere nei tubi a due velocità differenti (ma sempre costanti), in particolare a velocità di controllo e a velocità di sintesi;
11. I liquidi possono però essere stoccati in quantità limitate dentro appositi contenitori, da cui è possibile attingere quando serve. Questi liquidi di per sé non scorrono, ma, attraverso dispositivi specializzati, possono essere riversati in un liquido in scorrimento;
12. Tipicamente (anche se non necessariamente) un macchinario prevede un dispositivo munito di un tubo che permette di far uscire il liquido all'esterno. Altre volte può avere anche dispositivo con un tubo in entrata da cui ricevere un liquido che proviene da altri macchinari;
13. La circolazione dei liquidi tra l'impianto e l'esterno (l'acqua dall'acquedotto in entrata, il prodotto sintetizzato in uscita) oppure tra i diversi macchinari nell'impianto avviene attraverso tubi speciali. I primi sono tubi di entrata/uscita, i secondi di collegamento. Attraverso questi ultimi, i liquidi possono così circolare nell'impianto e sono a disposizione degli altri macchinari. Questi tubi permettono perciò di connettere diversi macchinari;
14. I tubi di collegamento disperdono il loro liquido (che non è uscito attraverso i tubi di uscita dell'impianto) negli scarichi dell'impianto. I liquidi non inquinano e la loro dispersione non è rilevante;

È possibile a questo punto riconsiderare i punti precedenti, sostituendo opportunamente i nomi di Figura **5.3** con quelli di Figura **5.4**.

Osservazione 1
L'intero processo di produzione è gestito non da un macchinario ma un impianto

È utile pensare al server non tanto come ad una macchina, quanto piuttosto come ad un vero e proprio impianto di sintesi audio, ad un ambiente complesso e organizzato internamente. Il server deve essere messo in moto. Si noti che quando si apre l'IDE, l'interprete è già attivo (in basso a sinistra, è in verde), ma il server no. Per far partire il server, si può utilizzare la GUI, cioè premere sulla finestra e selezionare `boot` oppure scrivere `s.boot`, poiché la variabile ambientale `s` è associata per default dall'interprete (per comodità mnemonica) al server audio.

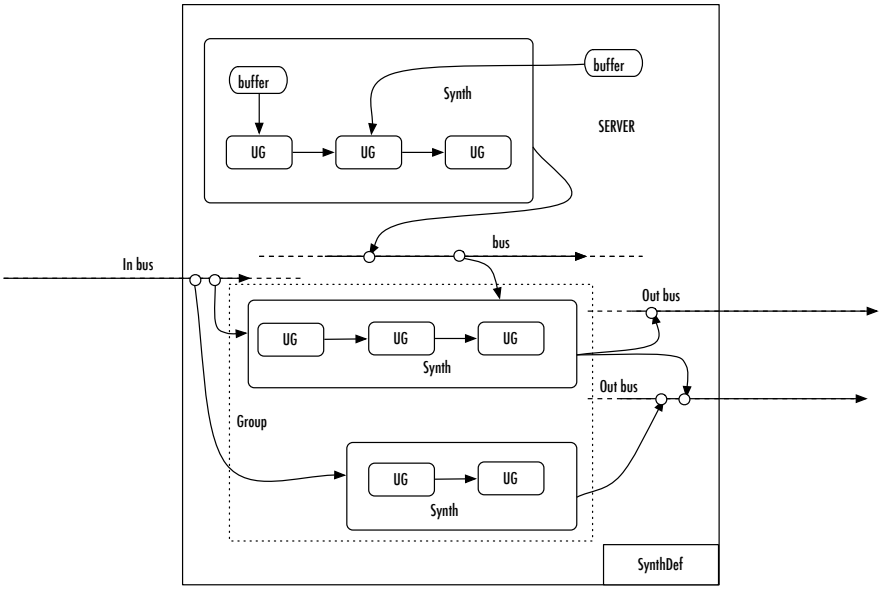


Fig. 5.4 Componenti del server audio.

Osservazione 2
Per sintetizzare un liquido è necessaria un macchinario complesso

Per sintetizzare un segnale audio è necessario un macchinario software che in SC prende il nome di `Synth`: un "synth" è appunto un sintetizzatore audio. Qui sintetizzatore va inteso letteralmente con un riferimento strumentale, ai sintetizzatori analogici. Un synth è uno strumento che suona.

Osservazione 3
Un macchinario è costituito di dispositivi specializzati in cui i liquidi subiscono trasformazioni. I dispositivi sono collegati attraverso tubi interni

Per generare segnale audio un synth richiede di specificare quali algoritmi di elaborazione/sintesi del segnale devono essere utilizzati. In SC, seguendo la tradizione della famiglia di linguaggi *Music N*, gli algoritmi di elaborazione/sintesi sono implementati in "UGen" (→ *U*nit *Gen*erator): una UGen è semplicemente un dispositivo software che elabora o sintetizza segnale audio. Ad esempio `SinOsc` è una UGen che genera segnali sinusoidali. Le UGen costituiscono i componenti di base di un synth, sono oggetti atomici al di sotto dei quali

non si può andare in SC[4]. Un synth è appunto un sintetizzatore, un dispositivo di sintesi costruito con componenti UGen. Una UGen ha il suo equivalente analogico in un modulo hardware elettronico di sintesi che può essere collegato via cavo ad altri dispositivi hardware.

Osservazione 4
Un macchinario deve essere progettato predisponendo le relazioni tra dispositivi componenti. A partire da un progetto, può essere costruito un numero indefinito di macchinari identici

Il client di SC chiede perciò al server di costruire e di far funzionare uno (o più) synth per lui. Per soddisfare la richiesta il server deve sapere quali pezzi (UGen) utilizzare e in quali relazioni combinarli (patching in uno "UGen-graph", un grafo di UGen). Poiché è probabile che all'utente possano servire più volte gli stessi dispositivi, SC prevede un passaggio supplementare. L'utente prima specifica al server una definizione di un synth (una "synthDef"), una sorta di progetto dettagliato di come deve essere fatto il synth desiderato, e quindi chiede al server di costruire un synth seguendo quel progetto. Vista dal punto di vista dell'hardware analogico, una synthDef è come lo schema elettrico sulla base del quale si costruisce un sintetizzatore. Una synthDef associa un nome n a una configurazione di UGen u, in modo che si possano creare synth di tipo n che generano segnali attraverso le relazioni tra UGen previste da u. Una volta create, le synthDef possono anche essere memorizzate su hard disk e restare perciò sempre disponibili all'utente. L'utente può in altre parole crearsi una libreria di synthDef (intese come progetti o come stampi da cui creare synth) e, quando è opportuno, chiedere al server di creare un synth a partire dalla synthDef.

In sostanza per usare SC come motore di sintesi è necessario compiere almeno due passi:

1. definire una synthDef (definire il progetto del sintetizzatore)
2. istanziare un synth a partire da una synthDef (costruire il sintetizzatore)

Osservazione 5
Una volta costruito, il macchinario non può essere modificato nella sua struttura interna

[4] Il loro equivalente in Csound è l'opcode, così come lo sono le unità di generazione che corrispondono a un blocco nei linguaggi grafici come Max/MSP o Pure Data.

Una synthDef è un diagramma, uno schema: è un oggetto statico. Una volta spedita al server, è immutabile. Se prevede due entrate, quelle avrà. D'altra parte, è sempre possibile spedire al server una nuova synthDef che prevede le modifiche desiderate, così come sovrascrivere una synthDef esistente.

Osservazione 6
Ma un addetto può controllarne il comportamento dall'esterno attraverso leve e comandi, così come monitorarne il funzionamento

Il progetto di un sintetizzatore è descritto in una synthDef attraverso lo UGen-Graph. Quest'ultimo può prevedere argomenti in entrata: questi argomenti sono appunto parametri per il calcolo, svolto all'interno dello UGen-graph, del valore (l'ampiezza del segnale, si ricordi che un segnale è una sequenza di numeri) che esso restituisce in uscita.

Osservazione 7
Un impianto può comprendere più macchinari che lavorano in parallelo

Ogni macchinario dell'esempio idraulico rappresenta un synth, ovvero, musicalmente parlando, uno strumento -o in fondo musicalmente anche una voce. Sul server può risiedere un numero molto grande di synth, di default 1024, ma il valore può essere incrementato dall'utente (risorse computazionali permettendo). Tutti i synth possono operare in parallelo.

Osservazione 8
Gruppi macchinari autonomi possono essere coordinati

I synth possono lavorare in parallelo, ma è anche possibile controllarli in gruppo. Cioè, è possibile inviare comunicazioni coordinate a gruppi di synth (diversi tra loro) se essi fanno parte dello stesso "group".

Osservazione 9
Quando l'impianto è in funzione i liquidi scorrono lungo i tubi a velocità costante, senza mai potersi fermarsi

Si è detto che il liquido rappresenta il segnale audio. La scelta del liquido dipende dal fatto che, poiché a questo punto non si sta parlando solo di segnali, ma di segnali in tempo reale, i segnali sono certo sequenze di valori di ampiezza secondo quanto visto finora, ma in più con il vincolo che tali campioni devono essere inesorabilmente calcolati ad un tasso uniforme nel tempo

(tipicamente, ma non necessariamente, nel caso di segnali audio, 44.100 volte in un secondo, il tasso di campionamento del CD). Ad ogni istante di tempo, un nuovo campione deve essere calcolato nella sequenza: ogni synth effettua tutti i calcoli previsti da tutte le UGen che lo compongono e restituisce un valore. In altre parole, ad ogni istante deve essere attraversato tutto lo UGen-Graph, indipendentemente dalla sua complessità. Attenzione: se si considera l'esempio idraulico, ciò significa che se una goccia entra da un tubo in entrata nell'istante x l'istante dopo ($x + 1$) deve essere già attraversato tutto l'impianto, ed essere in uscita. Ovvero: dentro ogni macchinario, lo scorrimento del liquido lungo i tubi che connettono i dispositivi è letteralmente istantaneo.

Osservazione 10
I dispositivi possono lavorare a due velocità differenti (ma sempre costanti), in particolare a velocità di controllo e a velocità di sintesi

Dunque ad ogni istante una nuova goccia deve uscire da un macchinario dopo aver percorso tutto il complesso dei dispositivi. I dispositivi non necessariamente però aggiornano il loro comportamento ad ogni istante. In alcuni casi, possono modificare la loro azione soltanto una volta ogni n istanti. Un segnale di controllo è tipicamente un segnale che cambia meno nel tempo di un segnale audio e che quindi può essere calcolato ad una risoluzione temporale più bassa. Ad esempio, si prenda il caso di un segnale che rappresenta un vibrato, un oscillazione della frequenza. Quest'oscillazione varierà, a mo' d'esempio, al massimo 10 volte al secondo: è un segnale sub-audio. Se si pensa alla curva che rappresenta l'oscillazione, allora si può descriverne la frequenza come $f = 10$ Hz. Il Teorema di Nyquist stabilisce che per rappresentare 10 è necessario un tasso di campionamento di $10 \times 2 = 20$ Hz. Dunque un tasso di 44.100 è uno spreco di risorse computazionali. Se si assume di lavorare a tasso audio (ad esempio, 44.100 Hz), si potrebbe allora calcolare un valore del segnale del vibrato per il quale moltiplicare il segnale audio, mantenerlo costante per n campioni audio, per poi ricalcolarlo al campione $n+1$, di nuovo mantenendolo costante per altri n, e così via. Un segnale simile è un segnale calcolato non a tasso audio (*audio rate*), ma a tasso di controllo (*control rate*). Come si vedrà, le UGen generano segnali nel momento in cui ricevono il messaggio `.ar`, o `.kr`: rispettivamente il segnale risultante sarà aggiornato a tasso audio (*audio rate*) o a tasso di controllo (*[k]ontrol rate*). Si noti che si sta parlando di tasso di aggiornamento, e non di numero di campioni. SC genera un segnale audio in tempo reale per forza a tasso audio: ma alcuni segnali che intervengono nella sintesi sono aggiornati ad un tasso più basso di controllo.

Osservazione 11
I liquidi possono però essere stoccati in quantità limitate dentro appositi contenitori, da cui è possibile attingere quando serve. Questi liquidi di per sé non scorrono, ma, attraverso dispositivi specializzati, possono essere riversati in un liquido in scorrimento

Un buffer è una memoria temporanea che permette di conservare dei dati audio richiesti da certi algoritmi di sintesi. Ad esempio, si consideri la lettura di un file audio per il playback. Esistono UGen specializzate in questa operazione. Il contenuto del file audio deve allora essere letto dall'hard disk per essere conservato in un blocco di memoria temporanea (della RAM). In SC un buffer è appunto un simile blocco che il server alloca sulla RAM a richiesta. Il segnale audio contenuto nel buffer di per sé è statico (è una sequenza di dati): e tuttavia vi può essere una UGen che legge il contenuto del buffer in tempo reale e lo invia alla scheda audio. Se si osserva la Figura **5.4** si nota come esistano due tipi di buffer. Alcuni possono essere interni ai synth, allocati direttamente quando un synth è creato e inaccessibili all'esterno. Altri invece sono residenti al di fuori dei synth e accessibili a più synth.

Osservazione 12
Tipicamente (anche se non necessariamente) un macchinario prevede un dispositivo munito di un tubo che permette di far uscire il liquido all'esterno. Altre volte può avere anche dispositivo con un tubo in entrata da cui ricevere un liquido che proviene da altri macchinari

Il segnale numerico sintetizzato deve essere inviato alla scheda audio in modo tale che quest'ultima lo converta in segnale elettrico e lo invii agli altoparlanti. Per questo compito esistono UGen specializzate, che prevedono entrate ma non uscite: infatti il segnale che vi transita non è più disponibile per la ulteriore elaborazione in SC, ma viene inviato alla scheda audio. Tipico esempio è la UGen `Out`. Evidentemente UGen di questo tipo occupano l'ultimo posto nello UGen-Graph che rappresenta un synth. Se si omette una UGen di uscita il segnale viene calcolato secondo quanto previsto dalle altre UGen nello UGen-Graph ma non inviato alla scheda audio (fatica mentale e computazionale sprecata, e classico errore del neofita). Si supponga poi di collegare un dispositivo di entrata alla scheda audio, ad esempio un microfono. Una UGen specializzata può rendere disponibile al synth tale segnale, in modo che possa essere elaborato (ad esempio subire una distorsione). A tal proposito SC prevede la UGen `SoundIn`.

Osservazione 13
La circolazione dei liquidi tra l'impianto e l'esterno (l'acqua dall'acquedotto in entrata, il prodotto sintetizzato in uscita) oppure tra i diversi macchinari nell'impianto avviene attraverso tubi speciali. I primi sono tubi di entrata/uscita, i secondi di collegamento. Attraverso questi ultimi, i liquidi possono così circolare nell'impianto e sono a disposizione degli altri macchinari. Questi tubi permettono perciò di connettere diversi macchinari

Si è già osservato come il server preveda una comunicazione con la scheda audio, in entrata ("dal microfono") e in uscita ("agli altoparlanti"). Questi canali di comunicazione prendono il nome di "bus", secondo un termine che deriva dalla tecnologia dei mixer[5]. In effetti, rispetto a "canale" (che pure è il termine audio più vicino) il termine "tubo" può essere meno fuorviante oltre che mnemonicamente efficace. Il sistema dei bus non deve essere pensato come un sistema di tubi che connettono staticamente i macchinari, ma come un sistema di tubi disponibili a cui i macchinari si raccordano. Ad esempio, tutti i synth che intendono elaborare un segnale che provenga dall'esterno possono raccordarsi al bus che è deputato alla lettura dell'entrata della scheda audio (al tubo che immette un liquido dall'esterno). Tutti i synth, che inviano i segnali in uscita alla scheda audio, si raccordano ai bus che gestiscono la comunicazione con quest'ultima: poiché un segnale digitale è una sequenza numerica, i segnali sui bus in uscita semplicemente si sommano. I bus finora citati sono specializzati per segnali audio (*audio busses*), ma esistono anche bus specificamente dedicati ai segnali di controllo *control busses*. I bus sono indicati attraverso un numero progressivo, un indice a partire da 0. Per i bus di controllo, la numerazione è progressiva e non ci sono aspetti particolari da tener presenti. Per i bus audio, è invece necessario ricordare che essi gestiscono la comunicazione con la scheda audio. I primi indici sono dedicati ai bus "esterni", di comunicazione I/O con la scheda audio. Ad essi seguono quelli ad uso "interno"[6] A cosa servono i bus interni? A connettere diversi synth. Ad esempio, un synth instrada il segnale in uscita sul bus 4 da dove altri synth possono prenderlo. Ovvero, idraulicamente un macchinario si raccorda ad un tubo in un punto e vi immette del liquido:

[5] A scanso d'equivoci, il termine non sta per "bus" come mezzo di trasporto. Ed infatti pensare ad un bus audio come a un autobus è fuorviante.

[6] La distinzione interno/esterno è d'uso, ma non c'è differenza di progettazione o implementazione di principio tra i due "tipi".

più avanti lungo il tubo un secondo macchinario può raccordarvisi e prelevare il liquido immesso.

Osservazione 14
I tubi di collegamento disperdono il loro liquido (che non è uscito attraverso i tubi di uscita dell'impianto) negli scarichi dell'impianto. I liquidi non inquinano e la loro dispersione non è rilevante

Una volta immessi su un bus, i segnali sono disponibili. Se non si scrive su un bus connesso alla scheda audio semplicemente non si ha un risultato percepibile. In altre parole, inviare un segnale su un bus non richiede di sapere cosa altri potranno fare di quel segnale, e neppure se mai qualche altro synth lo utilizzerà. Che cosa succede al segnale sul bus è irrilevante. Questo assicura una comunicazione possibile tra synth, ma senza che questa diventi obbligatoria o prevista in anticipo.

Riassumendo, il server prevede:

- **SynthDef**: progetti di strumenti, disponibili per la costruzione;
- **UGen**: unità atomiche di elaborazione del segnale, utilizzabili nella progettazione degli strumenti;
- **Synth**: strumenti veri e propri, costruibili e controllabili in tempo reale;
- **Group**: gruppi di synth, coordinabili;
- **Buffer**: blocchi di memoria per contenere dati utili;
- **Bus**: canali di comunicazione verso l'esterno e verso l'interno;

Che cosa si fa quando si usa il server? Si inviano comandi che permettono in tempo reale di definire, costruire, usare, distruggere, connettere strumenti che comunicano tra loro e che possono usare dati disponibili sul server. Si tratta ora di riprendere questi aspetti con il codice sotto le dita.

5.3 Il server

La prima cosa da fare per comunicare con il server è farlo funzionare. Il server è rappresentato nel linguaggio SC dalla classe `Server` che incapsula tutte le funzionalità dello stesso. È possibile avere più istanze del server che possono

essere controllate autonomamente (ad esempio, si pensi a una rete di computer, su ognuno dei quali funziona un server, tutti controllabili da un client). Per convenzione e comodità, un'istanza di `Server`, direttamente accessibile dall'interprete, è assegnata alla variabile globale s[7]. Non è dunque opportuno usualmente impiegare s per altri usi. L'esempio seguente riporta un insieme di messaggi che possono essere inviati al server per controllarne il funzionamento. Le righe 14-15 esemplificano l'assegnazione del server predefinito a un'altra variabile (`~myServer`). Nell'IDE la finestra in basso a sinistra riporta alcune informazioni di monitoraggio del server.

```
1  s // SC risponde con "localhost"
2  Server.default ; // lo stesso, assegnato a s

4  // Controllo minimale
5  s.boot ; // avviare il server
6  s.quit ; // fermarlo
7  s.reboot ; // riavviare il server

9  // Informazioni sui segnali generati
10 s.scope ; // visualizzare i segnali nel dominio del tempo
11 s.freqscope ; // visualizzare i segnali nel dominio della frequenza

13 Server.killAll ; // in caso di processi zombie

15 ~myServer = Server.default ; // altra variabile
16 ~myServer.boot ; // come prima
```

All'avvio del server sulla post window appaiono indicazioni analoghe alle seguenti.

- `booting 57110`: il server viene avviato per ricevere sulla porta (arbitraria e per default) 57110;
- `localhost`: è appunto il server di default;

[7] Esistono due tipi di server: un server "local" e uno "internal". Questo secondo tipo condivide parte della memoria con l'interprete e funziona per forza sulla stessa macchina dell'interprete. A parere di chi scrive, questa struttura è in contrasto con l'assunto di partenza dell'architettura client/server. Era utile in alcune situazioni nelle versioni precedenti. Sostanzialmente adesso il server internal è del tutto vestigiale, e non verrà preso in considerazione.

- righe 4-7: qui scsynth si interfaccia con il sistema operativo e si fa restituire una lista dei dispositivi audio accessibili, che possono includere configurazioni diverse a seconda del sistema operativo, dei dispositivi esterni etc.;
- righe 9-15: a questo punto viene selezionato il dispositivo effettivamente in uso, nell'esempio la scheda audio built-in. In entrambi i casi si tratta di due bus in ingresso (il microfono è stereo) e due in uscita (i due canali degli altoparlanti);
- riga 17: specifica il tasso di campionamento;
- riga 18: il server è pronto;
- righe 19-20: è stata ricevuta una notifica di ritorno dal server, dunque c'è comunicazione sulla rete di messaggi OSC. L'ultima riga non è qui rilevante ma indica che tutto è andato a buon fine.

```
1
2  booting 57110
3  localhost
4  Found 0 LADSPA plugins
5  Number of Devices: 2
6     0 : "Built-in Input"
7     1 : "Built-in Output"
8
9  "Built-in Input" Input Device
10    Streams: 1
11       0  channels 2
12
13 "Built-in Output" Output Device
14    Streams: 1
15       0  channels 2
16
17 SC_AudioDriver: sample rate = 44100.000000, driver's block size = 512
18 SuperCollider 3 server ready.
19 Receiving notification messages from server localhost
20 Shared memory server interface initialized
21
```

Avviato il server, è ora possibile procedere a sfruttarne le potenzialità. Rispetto al quadro metaforico dell'impianto, si tratta di far arrivare corrente all'impianto, che parte completamente vuoto. Sta all'utente costruire e far funzionare i macchinari. Un metodo utile a livello didattico (olte che in situazioni di risoluzione degli errori) è `s.dumpOSC`: con `dumpOSC(1)` tutti i messaggi OSC inviati al

server vengono stampati sulla post window. In questo modo, diventa evidente il flusso, altrimenti nascosto, delle comunicazioni OSC dal client al server. La stampa è disabilitata con `dumpOSC(0)`.

5.4 SynthDef

In primo luogo, è necessario costruire lo strumento che generi il segnale. Un esempio minimale di synthDef è quello riportato di seguito.

```
SynthDef.new("bee",
    { Out.ar(0, SinOsc.ar(567))}
).add ;
```

- `SynthDef`: è la classe a lato linguaggio che rappresenta un oggetto synthDef nel server;
- `.new( ... )`: `new` è il metodo costruttore, che costruisce effettivamente la synthDef (restituisce l'oggetto synthDef). Il metodo `new` prevede un certo numero di argomenti. Qui ne vengono specificati due, per gli altri è opportuno lasciare quelli predefiniti.
- `"bee"`: il primo argomento è una stringa che rappresenta il nome della synthDef: il nome verrà associato allo UGen-graph. I synth generati a partire da questa synthDef saranno dei "bee", cioè dei synth del tipo "bee". Qui "bee" è una stringa, ma potrebbe anche essere un simbolo (`\bee`). Si noti che la synthDef non è assegnata a una variabile. Infatti, l'unica utilità di una synthDef è quella di essere spedita al server;
- `{Out.ar(0, SinOsc.ar}`: lo UGen-graph è racchiuso tra parentesi graffe. Tutto ciò che è tra graffe in SC è una funzione. Dunque, lo UGen-graph è descritto da una funzione. Lo UGen-graph è costituito da due UGen, `Out` e `SinOsc`: questo fatto è reso esplicito dal messaggio `.ar` che le due UGen ricevono. In generale ciò che risponde al messaggio `.ar` o `.kr` è una UGen. E perché una UGen generi in tempo reale un segnale deve ricevere i messaggi `.ar` o `.kr`. Perché una funzione per descrivere le relazioni tra UGen? Si ricordi che

una funzione è un oggetto che restituisce un valore ogni qualvolta glielo si chiede. Un synth è allora descritto da una funzione perché ad ogni istante che passa gli viene chiesto di restituire un valore di ampiezza, il valore del campione audio. In altri termini, ad ogni istante (a tasso di campionamento di default, 44.100 volte al secondo) viene calcolato il valore della funzione descritta dalla UGen-Graph.

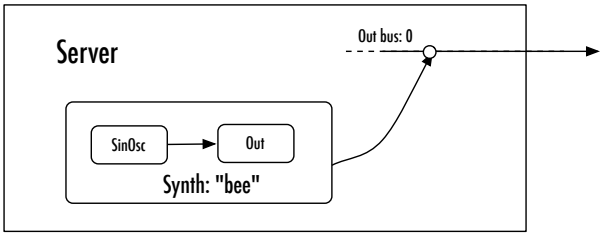


Fig. 5.5 Schema di una synthDef minimale.

In Figura **5.5** è rappresentato, secondo le convenzioni precedenti, un synth costruito a partire dalla synthDef "bee". Come si vede il segnale risultante viene "raccordato" (inviato) sul bus 0. In maniera più consueta lo UGen-Graph può essere descritto in forma di diagramma di flusso dal grafo di Figura **5.6**. Un aspetto interessante della Figura è che è stata generata in automatico a partire dalla SynthDef: non è perciò uno schema ma una vera e propria visualizzazione di informazioni.[8]. `Out` è la UGen che si occupa di inviare alla scheda audio il segnale generato: senza `Out` non c'è comunicazione con la scheda audio, quindi non c'è suono udibile. `Out` prevede due argomenti. Il primo è l'indice del bus su cui inviare il segnale in uscita, il secondo è il segnale stesso. `Out` riceve un segnale e lo spedisce al bus audio `0` che rappresenta il primo canale in uscita disponibile (è un bus esterno). Il segnale da spedire gli viene fornito da `SinOsc`: è un caso di patching, di innesto di una UGen (`SinOsc` in un'altra (`Out`, si veda dopo). `SinOsc` genera una segnale sinusoidale: laddove non si specifichino la frequenza e la fase, queste varranno rispettivamente 440 (Hz) e 0.0 (radianti).

[8] La figura è stata ottenuta utilizzando una versione modificata delle estensioni `dot` di Rohann Drape.

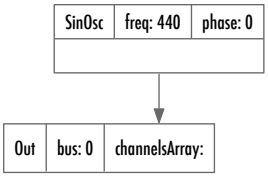


Fig. 5.6 Rappresentazione dello UGen-graph.

- `add`: la synthDef, di per sé, non serve a nulla se non è caricata sul server. Il metodo `add` definito per la synthDef dice alla synthDef di "spedirsi" al server `s`. La synthDef viene compilata in un formato specifico e spedita al server. Attenzione: tutto ciò che sta in una synthDef è costituito da specifiche istruzioni per la sintesi del segnale, niente altro. Con il metodo `add` la synthDef diventerà residente (cioè, disponibile) sul server fino a che questo sarà in funzione.

Oltre a `add`, vi sono molti metodi definiti per la classe `SynthDef`, che permettono ad esempio di scrivere la definizione su file (si tratta del metodo `writeDefFile`): le synthDef così memorizzate verrano caricate ad ogni accensione del server, e saranno perciò subito disponibili. Anche se di primo acchito può sembrare logico scrivere librerie di synthDef su file in modo che vengano caricate quando si avvia il server, in realtà è prassi diffusa caricarle ogni volta. Il metodo `add` è perciò di gran lunga il metodo usuale quando si lavora con le synthDef.

Tornando all'esempio, ora il server ha pronto il progetto "bee" per poter creare dei synth di tipo "bee". Che cosa è stato fatto? Una synthDef è stata definita (attraverso l'associazione di un nome e uno UGen-graph), compilata nel formato binario che il server accetta e inviata a quest'ultimo, che la tiene disponibile in memoria. Se si attiva la stampa su post window dei messaggi (`dumpOSC(1)`), all'invio della synthDef si ottiene:

```
1 [ "/d_recv", DATA[106], 0 ]
```

Una volta costruito, il synth `"bee"` può essere pensato come una sorta di modulo hardware senza controlli esterni, una scatola nera che semplicemente

può essere accesa o spenta. Vale la pena rivedere le considerazioni sulle funzioni espresse nella discussione sulla sintassi: una funzione può comunicare all'esterno attraverso i suoi argomenti. L'esempio seguente propone un caso già più complesso:

```
1  SynthDef.new(\pulseSine , { arg out = 0, amp = 0.25, kfreq = 5 ;
2      Out.ar(
3          bus:out,
4          channelsArray: SinOsc.ar(
5              freq: kfreq*50,
6              mul: LFPulse.kr(
7                  freq: kfreq,
8                  width: 0.25
9              )
10         )
11         *amp);
12 }).add;
```

Si noti come in questo caso lo UGen-Graph associato al nome `\pulseSine` (questa volta espresso come simbolo) preveda alcuni argomenti in entrata che ne permettono il controllo in tempo reale. Essi sono `out, amp, kfreq`. Ogni synth di tipo `"pulseSine"` metterà a disposizione dell'utente i tre controlli equivalenti in entrata. Se si pensa a un modulo hardware equivalente, questa volta, il synth relativo presenterà tre controlli all'esterno (per dire: tre cursori). È buona prassi prevedere quando possibile valori predefiniti per gli argomenti delle synthDef, in modo che un synth possa essere creato con il minimo sforzo e con parametri "sensati". La riga 12 chiude lo UGen-graph e spedisce la synthDef.

5.5 UGen e UGen-Graph

Le UGen sono unità atomiche di elaborazione del segnale. Le UGen possono avere più entrate, ma hanno sempre soltanto un'uscita. Una UGen può ricevere in entrata un'altra UGen: questo processo si chiama *patching*, e può essere tradotto (non letteralmente ma *ad sensum*) con "innesto", così come *to patch*

ha un buon equivalente in "innestare". Un insieme di UGen innestate tra di loro formano uno UGen-graph, un grafo di UGen, una struttura che rende conto delle relazioni tra UGen. Poiché le UGen generano segnali, lo UGen-graph descrive il flusso dei segnali che dalle diverse sorgenti si "raccolgono" nel segnale. Altra metafora: lo UGen-graph è la cartina geografica di un fiume che raccoglie contributi di diversi affluenti per poi terminare in mare.

Nella definizione della synthDef `"pulseSine"`, il programma utilizza la formattazione e una scrittura inusualmente prolissa per rendere più chiara possibile l'organizzazione del patching. Nell'esempio, le righe 2-11 descrivono il grafo delle UGen con un esempio di patching tra (`Out, SinOsc, LFPulse`). Si osservi come quest'ultima (un generatore di onde quadre) aggiorni i propri valori ricalcolando il valore in uscita a tasso di controllo, secondo quanto previsto dal messaggio `kr` inviato a `LFPulse`.

È opportuno ora soffermarsi di più su una UGen, in particolare su `SinOsc`. Per sapere come si comporta ci si può evidentemente rivolgere all'help file relativo. Un'altra opzione, utile in fase di studio almeno, consiste nell'accedere alla definizione nel codice sorgente.

```
1  SinOsc : UGen {
2      *ar {
3          arg freq=440.0, phase=0.0, mul=1.0, add=0.0;
4          ^this.multiNew('audio', freq, phase).madd(mul, add)
5      }
6      *kr {
7          arg freq=440.0, phase=0.0, mul=1.0, add=0.0;
8          ^this.multiNew('control', freq, phase).madd(mul, add)
9      }
10 }
```

Si noti come `SinOsc` erediti da `UGen`, la superclasse generica di tutte le UGen. In più, definisce soltanto due metodi di classe, `ar` e `kr`. Lasciando perdere l'ultima riga di ogni metodo, si nota come i metodi prevedano un certo numero di argomenti a cui può essere passato un valore "da fuori". Ancora, tutti gli

argomenti tipicamente sono provvisti di un valore predefinito. Nell'esempio seguente le tre righe sono equivalenti[9]

```
1 SinOsc.ar;
2 SinOsc.ar(440.0, 0.0, 1.0, 0.0);
3 SinOsc.ar(freq: 440.0, phase: 0.0, mul: 1.0, add: 0.0);
```

Il primo infatti, in assenza di indicazioni, utilizza i valori predefiniti per il metodo `ar`. Il secondo specifica per ogni argomento un valore. Il terzo infine fa uso di keywords, che come si sa permettono un ordine libero degli argomenti nella scrittura. Gli ultimi due argomenti sono `mul` e `add`, e sono condivisi dalla maggior parte delle UGen: `mul` è un moltiplicatore del segnale mentre `add` è un incremento (positivo o negativo) che viene sommato al segnale. Il segnale generato da una UGen tipicamente è normalizzato e la sua ampiezza oscilla nell'escursione $[-1, 1]$ (altre volte, in caso di UGen dedicate a segnali unipolari, è compreso in $[0, 1]$). L'argomento `mul` definisce un moltiplicatore che opera sull'ampiezza così definita, mentra `add` è un incremento che si aggiunge allo stesso segnale. I valori predefiniti per `mul` e `add` sono 1 e 0: il segnale è moltiplicato per 1 e sommato a 0 ed è perciò immutato rispetto all'escursione predefinita. A scanso di equivoci, "moltiplicare" e "aggiungere" significa che ogni campione del segnale è moltiplicato e sommato per i valori specificati nei due argomenti. Invece in questo esempio:

```
1 SinOsc.ar(220, mul:0.5, add:0) ;
2 SinOsc.ar(220, mul:0.5, add:0.5) ;
```

alla riga 1 il segnale generato attraverso il metodo `ar` risulta moltiplicato per 0.5 (e sommato a 0, ma è irrilevante): la sua ampiezza sarà compresa in $[-1.0, 1.0] \times 0.5 = [-0.5, 0.5]$; mentre alla riga 2 il segnale viene sommato a 0.5: la sua ampiezza sarà compresa in $[-1.0, 1.0] \times 0.5 + 0.5 = [-0.5, 0.5] + 0.5 =$

[9] Ovviamente è inutile valutarle nell'interprete, non succede nulla, perché descrivono oggetti che hanno senso una volta inseriti in una synthDef e inviati al server.

$[0.0, 1.0]$. Sarà cioè unipolare, asimmetrico rispetto allo 0. L'assegnazione `mul` del valore costante 0.5 indica che ogni nuovo campione verrà moltiplicato per 0.5. Si potrebbe pensare allora che `mul` sia un segnale, ma costante. A tal proposito si può prendere in considerazione la UGen `Line`. Come dice l'help file, si tratta di un generatore di "linee": una linea qui è un segnale che "generates a line from the start value to the end value". I primi tre argomenti di `Line` sono `start, end, dur`: `Line` genera una sequenza di valori che vanno da `start` a `dur` in `dur` secondi. Nel codice seguente

```
1 SinOsc.ar(220)*Line.kr(0.5,0.5, 10) ;
```

`Line` genera per 10 secondi una sequenza di valori pari a 0.5 (cioè una progressione da 0.5 a 0.5). Il segnale in uscita dall'oscillatore `SinOsc` viene moltiplicato per l'uscita di `Line`. Ad ogni istante di tempo il campione calcolato dalla prima UGen viene moltiplicato per il campione calcolato dalla seconda (che ha sempre valore 0.5). Si noti che il segnale risultante è uguale a quello precedente. L'esempio intende dimostrare come si può pensare ad una costante (un valore) nei termini di un segnale (una sequenza di valori): è chiaro che l'aspetto interessante nell'uso di `Line` sta invece proprio nel fatto che i valori che la UGen genera non sono costanti ma variano invece secondo una progressione lineare tra un due estremi.

L'esempio seguente produce un crescendo dal niente, proprio perché `start` $= 0$:

```
1 SinOsc.ar(220)*Line.ar(0.0,1.0, 10) ;
```

Se dunque una costante può essere pensata come un segnale, allora è possibile pensare ad ogni valore di un argomento in una UGen come ad un segnale costante. E pensare perciò al caso in cui invece di una UGen che generi costanti ci sia una UGen che generi (come usuale) sequenze di valori diversi. Il patching è appunto l'innesto di una UGen in un argomento di un'altra o il calcolo di un segnale a partire dal contributo offerto da più UGen in una qualche relazione reciproca (qui di moltiplicazione). L'esempio permette di capire come gli argomenti possano essere descritti non da costanti ma da variabili, cioè da altri

segnali. In altre parole, i segnali possono modificare qualsiasi aspetto controllabile di altri segnali. Si ricordi che il tasso di campionamento stabilisce a che frequenza il grafo debba essere percorso. In altre parole, a ogni istante $\frac{1}{sr}$ tutte le UGen ricalcolano il loro valore che viene utilizzato dove richiesto.

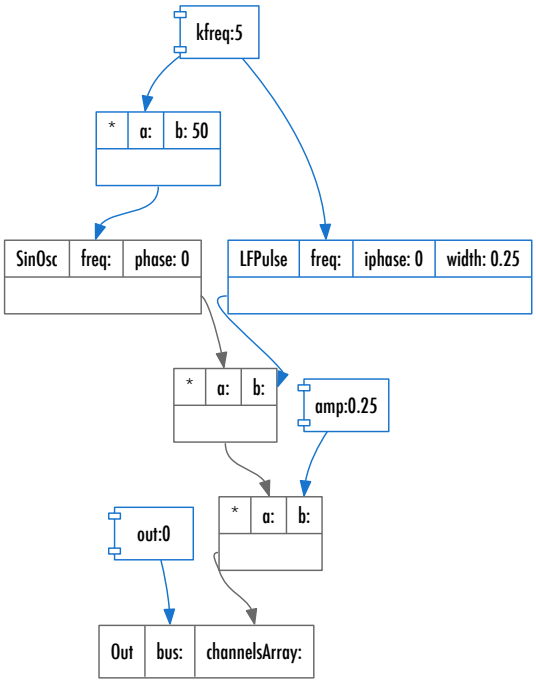


Fig. 5.7 Rappresentazione dello UGen-graph.

Nella figura **5.7** è rappresentato il diagramma di flusso della synthDef `"pulseSineGraph"`. Gli elementi grigi descrivono il flusso a tasso audio, quelli blu il flusso di controllo, i riquadrati di tipo connettore i valori numerici. `LFPulse` lavora a tasso di controllo, mentre `kfreq, amp, out` vengono modificate a tasso di evento (ogni qualvolta un utente modifica i parametri). I blocchi "* a b" indicano blocchi di moltiplicazione. I valori di argomenti nelle UGen non specificati sono indicati attraverso i valori predefiniti.

Il segnale moltiplicatore è prodotto da `LFPulse`, un generatore di onde quadre, con frequenza `kfreq` (quindi collegata alla frequenza della sinusoide). Il segnale in uscita da `LFPulse` è unipolare, cioè compreso nell'intervallo $[0, 1]$.

Utilizzando un segnale simile come moltiplicatore di un altro segnale, si ha che, quando l'ampiezza è 0.0, il segnale risultante ha ampiezza 0.0 (silenzio), quando l'ampiezza è 0.5, in uscita si ha il segnale di partenza scalato per 0.5. In sostanza, si produce una intermittenza. Si noti che nell'esempio la frequenza della sinusoide è correlata alla frequenza di intermittenza (più è acuta la frequenza, più frequentemente è intermittente).

Riassumendo: le UGen sono unità atomiche di elaborazione, descritte dal lato del linguaggio SuperCollider come classi che rispondono ai metodi `ar` e `kr`, i quali ricevono argomenti che ne costituiscono i parametri. Durante la fase di sintesi in tempo reale, le UGen generano in uscita segnali, cioè sequenze di valori a tasso audio. I valori di questi argomenti possono essere altri segnali (patching). Nell'esempio della synthDef `pulseSine` la scrittura è stata piuttosto prolissa, in particolare nell'uso delle keywords nelle UGen. I due esempi seguenti sono del tutto identici alla versione precedente. Il primo è una forma di scrittura compatta che è tipico delle synthDef ma che è di solito abbastanza complesso da decifrare per il neofita (con un po' di esercizio ci si fa l'occhio). Il secondo è una versione invece piuttosto prolissa ma molto chiara. Si ricordi che lo UGen-graph è descritto linguisticamente da una funzione, e dunque per esso valgono le considerazioni sulle funzioni. Nell'esempio, vengono usate due variabili (`pulser`, `sine`) a cui vengono associati i due segnali, per rendere più chiaro il flusso dell'informazione.

```
// compatta
SynthDef.new(\pulseSine , { arg out = 0, amp = 0.25, kfreq = 5 ;
    Out.ar(out, SinOsc.ar(
            kfreq*50, mul: LFPulse.kr(kfreq, width: 0.25)
        )*amp);
}).add;

// espansa
SynthDef.new(\pulseSine , { arg out = 0, amp = 0.25, kfreq = 5 ;
    var pulser, sine;
    pulser = LFPulse.kr(freq: kfreq, width: 0.25) ;
    sine =  SinOsc.ar(freq: kfreq*50, mul: pulser) ;
    sine = sine*amp;
    Out.ar(bus:out, channelsArray: sine);
}).add;
```

Le UGen sono unità atomiche di elaborazione del segnale. Il loro numero è molto grande. Per verificarlo si può valutare `UGen.subclasses.size`: nella installazione di chi scrive sono 305. Una classificazione approssimativa permette di distinguere:

- **Generazione**: sintesi del segnale, in forma deterministica o casuale (oscillatori, generatori di rumore) ;
- **Elaborazione**: trasformazione di un segnale in ingresso (filtri, ritardi, riverberi) ;
- **Spazializzazione**: diffusione su più canali di un segnale in ingresso ;
- **Analisi**: analisi di un segnale per l'estrazione di parametri;
- **Conversione**: conversione di segnali da audio a controllo e viceversa;
- **Buffer**: lettura/scrittura di buffer;
- **Inviluppi**: generazione/lettura di inviluppi;
- **Trigger**: segnali di innesco di vario tipo;
- **Input/Output**: accesso alla scheda audio in ingresso o in uscita;
- **Info**: permettono di accedere a informazioni audio a lato server;
- **Interazione con l'utente**: permettono di recuperare a lato server l'interazione (ad esempio via mouse) con l'utente;

SC fornisce una classificazione interna delle UGen, la seguente, in cui sono listate le UGen relative (sempre rispetto all'installazione presa in considerazione):

```
Algebraic (5)
Analysis (74)
Analysis:Synthesis (11)
Base (4)
Buffer (42)
Conversion (5)
Convolution (6)
Delays (34)
Demand (28)
Deprecated (1)
Dynamics (4)
Envelopes (8)
FFT (85)
Filters (107)
Generators (156)
GranularSynthesis (32)
InOut (20)
Info (15)
InfoUGens (1)
Maths (17)
Multichannel (126)
PhysicalModels (2)
Random (19)
Reverbs (3)
Synth control (13)
Triggers (31)
Unclassified (4)
Undocumented (218)
User interaction (4)
```

5.6 Synth e Group

Una volta scritta la synthDef e speditala al server, quest'ultimo semplicemente la archivia: essa è un progetto disponibile per costruire al volo un synth, uno strumento per produrre suono in tempo reale. Il synth può essere controllato a sua volta in tempo reale interattivamente dall'utente. Un tipico programma minimale in SC è il seguente:

```
1  // 1. avviare il server
2  s.boot ;
3  // tracciare i messaggi OSC
4  s.dumpOSC;

6  // 2. inviare la synthDef
7  (
8  SynthDef.new(\pulseSine , { arg out = 0, amp = 0.25, kfreq = 5 ;
9      Out.ar(
10         bus:out,
11         channelsArray: SinOsc.ar(
12             freq: kfreq*50,
13             mul: LFPulse.kr(
14                 freq: kfreq,
15                 width: 0.25
16             )
17         )
18         *amp);
19 }).add;
20 )

22 // 3. creare un synth
23 x = Synth(\pulseSine ) ;
```

Il programma è costituito di tre blocchi. In primo luogo (8-19), si avvia il server. E si attende che questo abbia risposto. Quindi si valuta il blocco contenente la synthDef. E si attende che il server abbia risposto. Infine, si crea un synth (23). I tre blocchi rappresentano perciò tre momenti di interazione asincrona tra client e server. In altri termini, il client di principio non sa quanto tempo ci

metterà il server a rispondere alla sue richieste. Ovviamente, empiricamente il tempo sarà ridotto al massimo nell'ordine dei millisecondi, ma ciò non sposta il punto: si tratta di una interazione asincrona. Se si valutasse tutto il codice in colpo solo, l'interprete eseguirebbe tutte le espressioni una di fila all'altra il più velocemente possibile. Ma allora il server starebbe ancora facendo l'avvio quando arriverebbe la definizione della synthDef, che non sarebbe perciò ricevuta. A questo punto, alla richiesta di un synth si otterrebbe:

```
*** ERROR: SynthDef pulseSine not found
FAILURE IN SERVER /s_new SynthDef not found
```

Un messaggio di errore che appunto indica come la synthDef richiesta per costruire un synth non è disponibile sul server.

Tornando all'esempio di programma, come intuibile, c'è una classe `Synth` che incapsula tutti i comandi relativi a un oggetto synth sul server. Il metodo costruttore `new`, omissibile al solito e qui omesso, prevede come argomento una stringa che indica la synthDef da cui il synth viene fabbricato (`"pulseSine"`). Nel momento in cui viene creato attraverso `new`, il synth viene attivato (= suona). Questo comportamento è tipicamente utile, perché in realtà un synth può essere pensato come uno strumento (un sintetizzatore) ma anche come un evento sonoro (se si preferisce, una "nota"): a pensarlo così, diventa ovvio che la costruzione del synth equivale alla generazione di un evento sonoro. Nel momento in cui si crea il synth, nella finestra dell'IDE dedicata al server il valore dei campi "u" (UGen attive) e "s" (synth attivi) varia di conseguenza.

Prima di entrare nel dettaglio del controllo del synth, molto generalmente, come si ferma il suono? La sequenza fondamentale ("salvavita") che arresta ogni processo di generazione (anche iterativa) è nell'IDE "CTRL/APPLE + ." (il tasto di controllo dipende dalla piattaforma), che equivale a "Stop" nel menu Language.

Se si considera nuovamente il programma precedente e si include `dumpOsc`, si ottiene la seguente post window:

```
1 a SynthDef
2 [ "/d_recv", DATA[343], 0 ]
3 Synth('pulseSine' : 1000)
4 [ 9, "pulseSine", 1000, 0, 1, 0 ]
5 [ "/g_freeAll", 0 ]
6 [ "/clearSched", ]
7 [ "/g_new", 1, 0, 0 ]
```

Senza entrare troppo nei dettagli, le righe 1 e 2 indicano la ricezione della synthDef (in forma di risposta standard dell'interprete e di messaggio OSC), quindi (3-4) la creazione del synth, infine (5-7) la sequenza dei messaggi OSC inviata a seguito di "CTRL/APPLE + .". Per quanto riguarda il synth, una osservazione: il numero `1000` è un identificativo che il server assegna progressivamente ai synth, a partire da 1000 (i numeri precedenti sono riservati per potenziale uso interno). Se si crea un altro synth (senza chiamare Stop), si noterà come questo avrà identificativo `1001`. In ogni caso, usualmente, si fa riferimento a un synth assegnando un oggetto SC di classe `Synth` a una variabile e non attraverso l'ID. È un esempio di utilità delle classi sul lato linguaggio: permettono all'utente di non occuparsi di questi dettagli. Invece, 5-7 indicano che a seguito del comando di Stop al server è stato richiesto di eliminare tutti i synth. Qui `g` fa riferimento ad un group che è sempre presente (si veda dopo).

Il prossimo esempio, abbondantemente commentato, assume che la synthDef `pulseSine` sia disponibile sul server. La riga 4 dimostra come si può istanziare un synth impostando subito i valori di alcuni argomenti della UGen-graph function (che altrimenti riceverebbero il valore predefinito dalla synthDef): si utilizza un array che alterna nome dell'argomento e valore. Le righe 5-6 indicano come si può controllare l'esecuzione/pausa di un synth attraverso il metodo `run`. Le righe 8-9 sono due esempi di controllo degli argomenti definiti nella UGen-graph function attraverso il metodo `set` che può ricevere liste argomento/valore per impostare il parametro desiderato. La riga 11 indica come eliminare (tecnicamente, "deallocare") un synth attraverso `free`. Infine, le righe 14-15 dimostrano che in alcuni casi è opportuno creare un synth senza che questo suoni con `newPaused`, in modo da mandarlo in esecuzione in un secondo momento (riga 15).

```
// Argomenti e controllo del synth

// controllo interattivo, valutare riga per riga (= performance)
x = Synth(\pulseSine , [\kfreq , 14, \amp , 0.7]) ;
x.run(false) ; // = premere il pulsante pausa sul synth
x.run(true) ; // = x.run, true e' il valore default

x.set(\kfreq , 15, \amp , 0.125) ; // controllo degli argomenti
x.set(\kfreq , 20) ; // controllo degli argomenti, uno solo

x.free ; // deallocazione: eliminazione del synth

// da capo ma iniziando con un synth in pausa
x = Synth.newPaused(\pulseSine , [\kfreq , 5, \amp , 0.5]) ;
x.run ;
```

Il codice precedente è una sessione interattiva con l'interprete in cui si controlla in tempo reale il server. È un esempio di live coding, per quanto minimale: di programmazione live. Di fatto, con SC si fa in qualche modo sempre live coding.

I synth possono anche essere coordinati in un gruppo: un gruppo (group) è semplicemente una lista di synth a cui è possibili inviare lo stesso messaggio. Group e synth hanno sostanzialmente la stessa interfaccia, cioè i metodi di base dei synth (che abbiamo già discusso) valgono anche per i group. Si consideri l'esempio seguente:

```
SynthDef(\sine , {arg  freq = 100; Out.ar(0, SinOsc.ar(freq))}).add ;
SynthDef(\pulse , {arg freq = 100; Out.ar(1, Pulse.ar(freq))}).add ;

s.scope ;
g = Group.new ;
x = Synth(\sine , [\freq , 200], target:g) ;
y = Synth(\pulse , [\freq , 1000], target:g) ;

g.set(\freq , 400) ;
x.set(\freq , 1000) ;
g.free ;
```

Le righe 1-2 definiscono due synthDef minimali basate rispettivamente su un oscillatore sinusoidale e uno a onda quadra. Entrambe espongono all'esterno l'argomento `freq` che intuibilmente controlla la frequenza nei due oscillatori. I due oscillatori instradano il segnale rispettivamente verso il canale sinistro e quello destro attraverso l'argomento di `Out`, che specifica il bus (`0` vs. `1`). Attraverso `s.scope` è più chiaramente osservabile cosa succede (4). Viene quindi creato un gruppo attraverso la classe `Group`. Segue la creazione di due synth (righe 6-7), del tutto usuale se si eccettua la specificazione dell'argomento `target`, che specifica l'eventuale assegnazione del synth a un gruppo, qui il gruppo `g`. A questo punto diventa possibile controllare l'intero gruppo. La riga 9 invia (si noti la sintassi identica a quella per synth[10]) a tutti i synth del gruppo `g` il messaggio che imposta `freq` $= 400$. Ad esso, rispondono tutti i synth che lo prevedono (entrambi nel caso in questione). L'appartenenza a un gruppo non impedisce un controllo dei singoli synth (10). Si noti che è possibile deallocare in un unico messaggio l'intero gruppo, ovvero tutti i synth che ne fanno parte (11). I gruppi sono utili per avere forte modularità (i synth) ma allo stesso tempo coordinamento: si pensi a scalare il volume di molti synth insieme. Tra l'altro, un gruppo può a sua volta contenere altri gruppi. Quando si crea un synth senza specificare il gruppo, questo viene aggregato ad un gruppo di default.

5.7 Un theremin

Un theremin è uno strumento elettronico inventato dal russo Léon Theremin in cui due antenne, senza essere toccate ma in funzione della prossimità delle mani dell'esecutore, controllano l'ampiezza e la frequenza di un oscillatore. Si consideri allora il seguente esempio (commentato per ribadire un'altra volta alcuni punti sulle synthDef):

[10] Infatti, sia `Synth` che `Group` sono sottoclassi di `Node`, e tra l'altro l'help file di quest'ultimo contiene informazione rilevante per entrambi.

```
SynthDef.new(\theremin ,  // nome simbolico
    // segue il grafo delle ugen
    {
    // c'e' sempre Out per uscire
    Out.ar(
        0, // indice del bus
        // segue segnale generato dall'oscillatore digitale
        SinOsc.ar( // la UGen oscillatore
            freq: MouseX.kr(200, 5000, 1), // argomento frequenza
            mul: MouseY.kr(0,1))) // moltiplicatore dell'ampiezza
}).add ;
```

La struttura è molto semplice: un generatore di sinusoidi è controllato in ampiezza e in frequenza da due UGen peculiari, `MouseX` e `MouseY`. Si tratta di due "pseudo-UGen", infatti intercettano il mouse (che è a lato client, evidentemente) per generare segnali di controllo. Sono molto utili per verificare rapidamente cosa succede variando i valori di certi argomenti. Le due UGen, riferite alla posizione del mouse sui due assi, permettono di mappare l'escursione sui due assi tra `minval` e `maxval`, e definiscono anche la curva che connetterà i due estremi (`warp`). Nella frequenza (`MouseX`) l'escursione è $[200, 5000]$, poiché si fa riferimento agli Hz, nell'ampiezza (`MouseY`) invece è $[0, 1]$ (ampiezza normalizzata). La curva è di tipo lineare (`0`) per l'ampiezza e esponenziale per le frequenze (`1`), in modo da simulare la percezione delle altezze[11]. In questo modo, attraverso il mouse, si simula il funzionamento di base di un theremin.

Il grafo di UGen è in Figura **5.8**. Il grafo permette di discutere un punto. Come si è detto, le UGen condividono gli argomenti `mul` e `add`. Internamente, in realtà esiste una UGen specializzata, `MulAdd`, che opera molto efficientemente. La UGen non è utilizzata esplicitamente dall'utente, ma c'è, e il grafo, essendo generato ispezionando la struttura della synthDef, la rileva e la disegna (il blocco `*`).

[11] L'argomento `lag`, di solito non impostato, è un ritardo che evita una eccessiva sensibilità temporale del controllo via mouse.

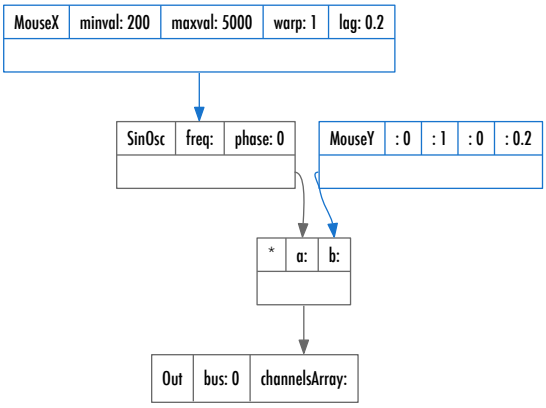


Fig. 5.8 UGen-graph di un theremin.

Se si vuole vedere cosa succede in termini di forma d'onda e spettro si usino `s.scope` e `s.freqscope`.

5.8 Un esempio di sintesi e controllo in tempo reale

A questo punto, vale la pena discutere un esempio che unisca un controllo di tipo grafico alla sintesi. È un primo esempio minimale di programmazione che mette in relazione strutturata client e server.

```
Server.local.boot ;    // come s.boot

(
// audio
SynthDef.new("square", { arg out = 0, freq = 400, amp = 0.75, width = 0.5;
    Out.ar(out, Pulse.ar(freq, width: width, mul: amp));
}).add;
)

(
// variabili
var aSynth, window, knob1, knob2, button;

aSynth = Synth.new(\square ); // il synth

// GUI: creazione
window = Window.new("Knob", Rect(300,300,240,100));
window.front;

knob1 = Knob.new(window, Rect(30, 30, 50, 50));
knob1.valueAction_(0.25);

knob2 = Knob.new(window, Rect(90, 30, 50, 50));
knob2.valueAction_(0.3);

button = Button.new(window, Rect(150, 30, 50, 50)) ;
button.states = [    // array di stati
            [ "stop", Color.black ], ["start", Color.red]] ;

// GUI: controllo dell'audio
knob1.action_({arg me; aSynth.set(\amp , me.value) });
knob2.action_({arg me; aSynth.set(\freq , me.value.linlin(0,1, 200, 2000)) });
button.action = ({ arg me;
    var val = me.value.postln;
    if (val == 1) { aSynth.run(false) } { aSynth.run }
});

window.onClose_({aSynth.free})
)
```

La synthDef è un semplice involucro per la UGen `Pulse` che genera onde quadre, e permette di controllarne i parametri: `width` è il duty cycle, cioè il rapporto tra segnale positivo e negativo (nell'escursione $[0, 1]$, dove il default è 0.5

che indica simmetria). Il blocco 10-38 è invece deputato a istanziare il synth e al suo controllo via GUI. La riga 14 crea un synth e lo associa alla variabile `aSynth`. Sulla creazione di elementi GUI non c'è molto da dire. Si tratta di costruire due rotativi all'interno di una finestra, secondo una tecnica già vista. Più interessanti invece le azioni collegate alla variazione dei rotativi e al pulsante. La riga 31 definisce la connessione tra il controller GUI `knob1` e sintesi audio. L'azione è associata alla manopola attraverso il metodo `action`. Ogni volta che cambia il valore del rotativo viene chiamato sull'oggetto `aSynth` (il synth) il metodo `set` che assegna al parametro specificato (`amp`) un valore, qui `v.value`. Poiché un rotativo `Knob` genera valori nell'intervallo $[0, 1]$, cioè nell'escursione d'ampiezza del segnale audio in forma normalizzata, l'escursione è già quella utile per un controllo dell'ampiezza (ne sarà il "volume"). Analogamente, la riga 32 assegna all'argomento `freq` il valore del rotativo `knob2`. Poiché sono in gioco le frequenze, l'escursione di default $[0, 1]$ non è opportuna. Il metodo `linlin`, definito sulle grandezze, permette agevolmente di mappare linearmente una escursione di partenza(i suoi primi due argomenti) su una di uscita (definita dai secondi due argomenti), qui $[200, 2000]$. Analogamente, si potrebbe sfruttare il metodo `linexp` che ha la stessa sintassi ma effettua una trasformazione esponenziale. La situazione è rappresentata in Figura **5.9**.

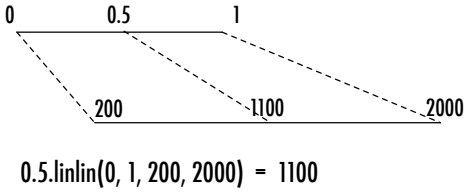


Fig. 5.9 Interpolazione lineare con `linlin`.

Anche a `button` viene assegnata un'azione secondo una sintassi analoga a `Knob`: `me.value` permette di accedere allo stato del pulsante (ovvero l'indice dello stato nell'array degli stati `button.states`). Ad ogni pressione del pulsante viene richiamata l'azione. L'azione valuta lo stato del pulsante attraverso il costrutto condizionale (35). Se il valore `val` del pulsante (che viene stampato) è `1`, viene chiamato il metodo `run(false)` su `aSynth`, che mette in pausa il synth. Nel caso negativo (e dunque se `val` è `0`) il synth viene attivato (`aSynth.run`). Quando si esegue il codice, il synth è attivo e lo stato è `0`. Alla prima pressione lo stato diventa `1`, la GUI viene aggiornata e viene eseguito il ramo condizionale che contiene `aSynth.run(false)`.

Le righe 21 e 24 impostano il valore dei due rotativi con il metodo `valueAction` che non solo modifica l'elemento grafico ma valuta l'azione relativa al valore, in questo modo mantenendolo in relazione con il controllo. Infine, la riga 38 associa attraverso il metodo `onClose` alla finestra contenitore un'azione che viene eseguita alla sua chiusura: qui la deallocazione del synth `aSynth`.

5.9 Espressività del linguaggio: algoritmi

Il linguaggio SuperCollider non ha con buona probabilità per il musicista il pregio di essere un'interfaccia intuitiva, ma ha d'altro canto quello di essere molto espressivo. Si consideri ad esempio il programma seguente:

```
1  (
2  SynthDef(\sine , {|freq = 440, amp = 1, gain = 1|
3      Out.ar(0, SinOsc.ar(freq, mul:amp*gain))}).add ;
4  )

6  (
7  var base = 20, keys = 88 ;
8  var  synths, group;
9  var window, step = 15, off = 20, len = 80 ;
10 group = Group.new;
11 synths = Array.fill(keys, {|i|
12     Synth(\sine , [\freq , (base+i).midicps, \amp , 0], group)});
13 window = Window.new("sliderPiano", Rect(10, 10, keys*step, off+len+off+10))
14    .front ;
15 keys.do{|i|
16     StaticText(window, Rect(i*step, 0, step, step))
17         .string_((base+i).midinote[0..1]);
18     Slider(window, Rect(i*step, off, step, len))
19         .action_{|me|synths[i].set(\amp , me.value/keys)}
20 };
21 Slider(window, Rect(0,step+len+10, step*keys, off))
22    .action_{|me| group.set(\gain , me.value.linlin(0, 1, 1, 9.dbamp))};
23 window.onClose_{group.free} ;
24 )
```

Il codice crea una "tastiera" di cursori, ognuno associato ad una nota di pianoforte che controlla un generatore di sinusoidi le cui altezze sono relative alla nota indicata in alto. Il cursore orizzontale controlla un guadagno, che incrementa il volume di tutti i generatori. Tre sono le considerazioni da fare:

- due righe (11-12) producono 88 synth, in realtà è un'unica espressione;
- analogamente, le righe 15-20 generano, di nuovo in un'unica espressione, 88 cursori relativi ai synth;
- l'uso estensivo delle variabili permette di parametrizzare il tutto. Ad esempio, si provi `base` $= 20$ e `keys` $= 12$. Si otterrà l'ottava del do centrale

Il codice sfrutta costrutti abbondantemente discussi e assume che il server sia avviato. La synthDef è molto semplice. L'unica questione di rilievo è un moltiplicare del moltiplicatore `amp`. Può essere pensato come un guadagno, di qui il nome `gain`. La Figura **5.10** illustra la struttura della synthDef come appare a SC.

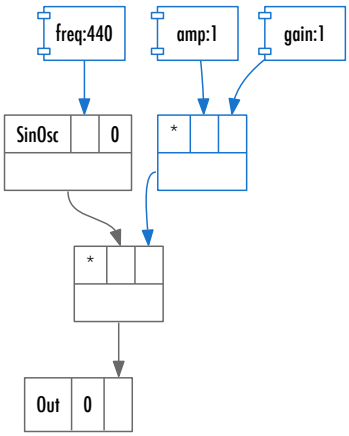


Fig. 5.10 Stuttura della synthDef `sine`.

Le variabili `base` e `keys` indicano rispettivamente la nota più grave e il numero delle note. Qui le altezze sono espresse seguendo il protocollo midi che assegna un intero progressivo ad ogni nota (secondo il modello della tastiera del pianoforte) a partire da 60 che (arbitrariamente) rappresenta il do centrale. Invece i parametri `step, off, len` sono relativi ai parametri grafici, in particolare `step`

è il passo su cui è costruita tutta la grafica. Dopo aver definito un gruppo `group`, viene creato l'array `synths` deputato a contenere i synth nel numero indicato da `keys`. La funzione prevista da `fill` permette di sfruttare il contatore `i`. I synths avranno una frequenza progressiva con incremento di 1 a partire da `base`. Partono tutti con ampiezza $= 0$, e sono dunque tutti attivi (stanno generando segnale, ma con ampiezza nulla). I synths sono aggregati nel gruppo `group` (12). La larghezza della finestra di contenimento generale è calcolata in funzione del numero di synth (per `step`) (13). A questo punto viene creato un numero `keys` di elementi grafici a coppie (cursori, slider, e scritte, staticText). Si noti come la posizione degli oggetti faccia riferimento al contatore `i` e a `step`. La classe `StaticText` permette di definire una stringa che verrà visualizzata con il metodo `string`. Qui il testo da scrivere è il numero della nota, che viene recuperato attraverso `midinote`, di cui si prendono i primi due caratteri (`[0..1]`), evitando l'indicazione di ottava per compattezza (17). L'azione collegata al cursore (19) invece assegna al synth recuperato in `synths` all'indice relativo `i` un'ampiezza recuperata dal valore del cursore (19). Per sicurezza, l'ampiezza è divisa per il numero delle note, per evitare possibili clip (secondo un metodo già descritto). Il cursore (21-22) sfrutta il gruppo, e invia a tutti i synth del gruppo un valore che viene utilizzato come moltiplicare di `amp` (19). Qui il guadagno è espresso in deciBel (`9.dbamp`) e riconvertito in ampiezza. Si noti che è necessario raccogliere i synth in un array (`synths`) perché dopo i cursori vi faranno riferimento. Al contrario, gli elementi grafici successivi vengono semplicemente stampati sullo schermo e non più modificati, dunque è inutile che essi vengano immessi in un array. Lo stesso vale per il cursore che controlla il gain. Infine, un ultimo elemento di controllo è associato alla finestra. Quando questa si chiude, tutto il gruppo viene deallocato (23).

Si noti la compattezza e la modificabilità del codice che produce un vero e proprio banco di oscillatori controllabili via GUI.

5.10 Espressività del linguaggio: abbreviazioni

Quando c'è suono dal server ci sono per forza due elementi:

- una synthDef che specifica una organizzazione di UGen;
- un synth costruito a partire da essa.

Tertium non datur. Eppure, soprattutto se si esplorano gli help file relativi alle UGen, ad esempio quello di `SinOsc`, si trovano espressioni di questo tipo:

```
1 { SinOsc.ar(200, 0, 0.5) }.play;
```

Se si valuta l'espressione, una sinusoide suona. Eppure non c'è synthDef, non c'è synth, c'è solo una funzione (che non contiene la UGen `Out`) su cui viene invocato il metodo `play`. Quello che succede è che il l'interprete, come conseguenza del messaggio `play`

- crea una synthDef
- definisce un nome arbitrario e lo associa al grafo
- immette nel grafo un blocco di tipo `Out.ar(0, func)` che indirizza sul bus `0` il segnale
- associa nome a grafo e spedisce la synthDef al server
- attende che la synthDef sia compilata
- crea un synth da essa.

Se si valuta il codice sulla post window si ottiene una risposta analoga alla seguente:

```
1 Synth('temp__0' : 1000)
```

Si vede che la synthDef ha come nome `'temp__0'`. Se si disegna il grafo della synthDef si ottiene la Figura **5.11**. L'unico fatto nuovo sta nel fatto che qui `Out` (in giallo) non è a tasso audio né di controllo ma a tasso `ir`, cioè "initial rate". Indica che non può essere modificato (ed infatti è disegnato come un dato statico) ed è computazionalmente più efficiente. D'altronde l'utente non ha richiesto il controllo (non ha specificato `Out`).

Il costrutto `{...}.play` è di grande utilità per almeno due motivi. In primo luogo, permette molto rapidamente di sperimentare con le UGen, saltando il percorso più strutturato ma macchinoso in fase di prototipazione di definizione della synthDef e costruzione del synth. In seconda battuta, permette molto agevolmente di avere un synth. Infatti, linguisticamente restituisce un riferimento al synth creato. Nell'esempio seguente:

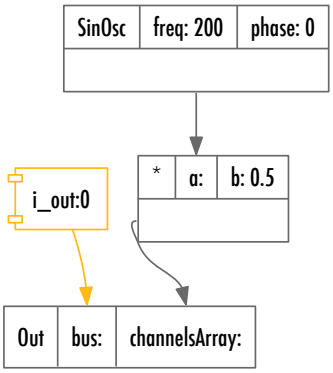


Fig. 5.11 Grafo della synthDef temporanea.

```
x = {|fr = 1000| SinOsc.ar(fr, 0, 0.5) }.play ;
x.set(\fr , 400) ;
x.free ;
```

La variabile ambientale `x` è associata al synth restituito dalla funzione. Il synth a questo punto può essere controllato come al solito. In Figura **5.12** il grafo delle UGen relativo, in cui si vede l'argomento accessibile.

Si noti che anche il bus di uscita può essere comunque controllato. Nell'esempio seguente (che introduce la UGen generatore di onde a dente si sega, `Saw`), l'argomento `out` specifica il valore del bus di `Out` specificato esplicitamente nella funzione, e può essere controllato dal synth.

```
x = {|fr = 1000, out = 0| Out.ar(out, Saw.ar(fr, 0.5)) }.play ;
x.set(\out , 1) ; // ora sul canale destro, bus 1
x.free ;
```

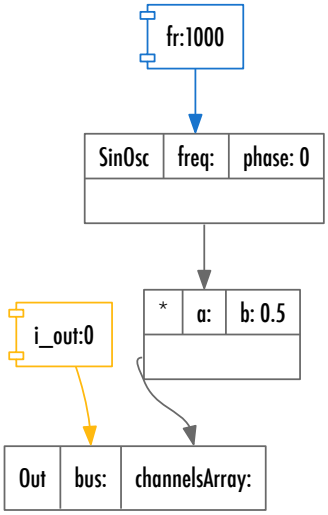


Fig. 5.12 Grafo della synthDef temporanea, con argomento.

Il grafo è in Figura **5.13**, dove si vede che `i_out` è creato comunque, ma non utilizzato.

Il metodo `play` è stato introdotto solo ora, perché con tutto quello che richiede dietro le scene, sebbene assai utile, rischia in realtà di indurre in confusione chi si avvicina inizialmente a SuperCollider. Ma sarà molto utile di qui in poi.

5.11 Conclusioni

Una volta introdotta, la struttura del server (se si tiene presente Figura **5.4**) non è poi così complessa. Ci sono alcuni elementi (sei per la precisione) e un protocollo di comunicazione per controllarli attraverso l'invio di messaggi. Ciò che deve essere molto chiaro è questo punto: *se c'è suono in tempo reale, allora c'è un synth sul server*. Distinguere chiaramente ciò che sta sul server da ciò che sta

sul client è il presupposto fondamentale per operare efficacemente con SuperCollider.

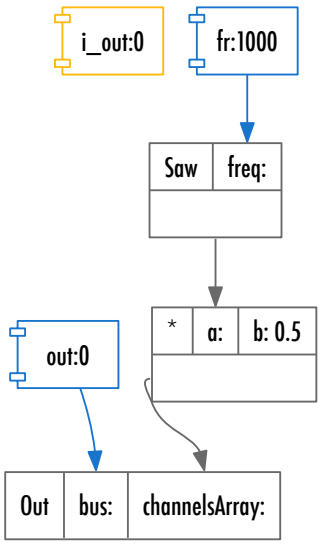


Fig. 5.13 Grafo della synthDef temporanea, con `Out`.

6 Controllo

La discussione sull'ontologia del server non è ancora terminata, poiché all'appello mancano bus e buffer. Ancora, vale la pena riprendere le considerazioni sugli inviluppi, già introdotte in precedenza. Insieme ad altre, si tratta di risorse che il server mette a disposizione per i processi di sintesi ed elaborazione del segnale, e che sono perciò qui rubricate (con larga approssimazione) sotto la categoria del controllo.

6.1 Inviluppi

Si è già discusso degli inviluppi d'ampiezza a proposito del calcolo di segnali in tempo differito. In particolare si era avuto modo di notare come la classe `Env` fosse specializzata nella generazione di "segnali" d'inviluppo. In realtà un oggetto di tipo `Env` non è un segnale, ma una sorta di stampo del profilo temporale dilatabile (o comprimibile) a piacere. Tant'è che per ottenere un segnale in tempo differito era stato necessario inviare il messaggio `asSignal`. In tempo reale, un oggetto `Env` è una "forma" temporale a disposizione di una UGen specializzata che, leggendo il contenuto di `Env`, genera un segnale: la UGen `EnvGen` (intuibilmente: "envelope generator") è appunto specializzata in questo compito. In effetti, l'argomento `times` del costruttore di `Env`, che richiede di specificare un array di durate, deve essere inteso non come una specificazione assoluta, quanto piuttosto proporzionale. Le durate specificate indicano cioè la proporzione tra parti dell'inviluppo (si ricordi che le durate sono qui riferite agli intervalli tra valori d'ampiezza), non necessariamente una indicazione cronometrica.

Si considerino gli argomenti nel sorgente SC del metodo `*ar` di `EnvGen` (che è uguale, tasso a parte, a quella di `*kr`):

```
*ar { arg envelope, gate = 1.0, levelScale = 1.0, levelBias = 0.0,
    timeScale = 1.0, doneAction = 0;
```

Come specificato ulteriormente dall'help file relativo, l'argomento `envelope` è un'istanza di Env. Gli argomenti `levelScale, levelBias, timeScale` sono tutti operatori che trasformano l'inviluppo passato come primo argomento. Si prenda come esempio l'inviluppo

```
e = Env.new(
    levels:[0.0, 1.0, 0.5, 0.5, 0.0],
    times: [0.05, 0.1, 0.5, 0.35]
).plot ;
```

- `levelScale`: moltiplica i valori d'ampiezza dell'inviluppo. Il valore predefinito è 1.0, che lascia i valori di `levels` in e immutati
- `levelBias`: viene sommato ai valori d'ampiezza. Se questi sono compresi in entrata nell'escursione $[0.0, 1.0]$, in uscita essi saranno compresi in $[0 + levelBias, 1.0 + levelBias]$. Anche qui il valore di default (0.0) non modifica i valori di e
- `timeScale`: moltiplica i valori di durata. La durata del nuovo inviluppo sarà perciò $timeScale \times times$. Il totale degli intervalli in e è pari a 1, dunque la durata dell'inviluppo generato da `EnvGen` sarà pari a $1 \times timeScale$. Come in precedenza, il valore predefinito (1.0) non cambia i valori dell'inviluppo passato come primo argomento.

Come si vede `EnvGen` può "stirare" la forma dell'inviluppo in lungo e in largo. In effetti, conviene utilizzare come escursione sia per l'ampiezza che per la durata di un inviluppo `Env` l'intervallo normalizzato $[0.0, 1.0]$. Si potrà poi agire sugli argomenti di `EnvGen`. Nell'esempio seguente viene modificato un esempio

tratto dall'help file. Come si vede, viene utilizzato il metodo `play` sulla funzione. Qui l'inviluppo è il tipico inviluppo percussivo senza sostegno, accessibile inviando a `Env` il messaggio (costruttore) `perc`:

```
// moltiplicazione esplicita tra segnali
{ EnvGen.kr(Env.perc, 1.0, doneAction: 0)
    * SinOsc.ar(mul: 0.5) }.play ;

// effetti di timeScale
{ EnvGen.kr(Env.perc,  1.0, timeScale: 10,  doneAction: 0)
    * SinOsc.ar(mul: 0.5) }.play ;

// moltiplicazione usando mul
{ SinOsc.ar(mul: 0.1
    * EnvGen.kr(Env.perc,  1.0, timeScale: 10,  doneAction: 0) ) }.play ;
```

Si confrontino i segnali risultanti dai primi due esempi: si noterà l'effetto di `timeScale`, che dilata gli intervalli temporali di un fattore 10. I primi due esempi illustrano un'implementazione tipica dell'inviluppo: i due segnali vengono moltiplicati tra di loro ed, essendo uno unipolare e l'altro bipolare, il risultato sarà un segnale bipolare (la sinusoide inviluppata, come si vede sostituendo a `play` il metodo `plot`). L'altra implementazione tipica in SC (che produce lo stesso risultato) è illustrata nel terzo esempio in cui il segnale di inviluppo generato da `EnvGen` è il valore dell'argomento `mul` della UGen interessata dall'inviluppo.

Gli altri due argomenti di `EnvGen` sono `gate` e `doneAction`. Entrambi rispondono ad un problema, quello di mettere in relazione un segnale di durata indefinita (ad esempio, quello che risulta da `SinOsc`) con un segnale che ha una durata specifica (e che rappresenta il concetto di "evento" sonoro), il segnale di inviluppo.

- `gate`: l'argomento `gate` specifica un *trigger*. Un trigger è un segnale di attivazione, è "qualcosa che fa partire qualcosa". Per la precisione, un trigger funziona come una fotocellula: ogniqualvolta registra un passaggio, invia un segnale. Tipicamente in SC il passaggio che fa scattare il trigger è il passaggio dallo stato di partenza ad uno stato con valore > 0. In sostanza, `EnvGen` genera al tasso prescelto un segnale d'ampiezza pari a 0 finché non riceve il trigger. A quel punto legge l'inviluppo. Nei tre esempi precedenti il valore di `gate` era 1.0 (che è anche quello di default): essendo superiore a 0 fa scattare il trigger. L'attivazione del trigger dice alla UGen `EnvGen` di leggere

l'inviluppo. Dunque, eseguendo il codice si ascolta il segnale inviluppato. Il primo esempio del codice seguente prevede come valore di `gate` 0, e il trigger infatti non scatta. Se si sostituisce ad una costante un segnale il trigger viene attivato tutte le volte che viene sorpassato il valore di soglia 0. Si noti che il passaggio è direzionato, da $<= 0$ a > 0, il contrario non vale. Nel secondo esempio è una sinusoide con frequenza 4 Hz che controlla il trigger `gate`. Tutte le volte che supera l'asse delle ascisse (entrando nel semipiano positivo sopra lo 0) il trigger viene attivato. Ciò avviene 4 volte al secondo. Infine, nell'ultimo esempio `gate` è un segnale generato da `MouseX`. L'asse orizzontale dello schermo viene ripartito nell'intervallo $[-1, 1]$. Lo 0 è perciò a metà dello schermo. Tutte le volte che lo si supera il trigger viene attivato. Si noti che spostando il mouse intorno allo zero ma da sinistra a destra non succede nulla. L'argomento `gate` risponde al problema di implementare un inviluppo a a chiamata: un esempio tipico è quello di una tastiera in cui alla pressione del tasto deve corrispondere, unitamente alla sintesi del segnale, anche l'innesco dell'inviluppo associato.

```
// gate = 0
{ EnvGen.kr(Env.perc, 0.0, doneAction: 0)
    * SinOsc.ar(mul: 0.5) }.play ;

// controllando gate
// con un segnale
{ EnvGen.kr(Env.perc, SinOsc.kr(4), doneAction: 0)
    * SinOsc.ar(mul: 0.5) }.play ;
// con il mouse
{ EnvGen.kr(Env.perc, MouseX.kr(-1,1), doneAction: 0)
    * SinOsc.ar(mul: 0.5) }.play ;
```

- `doneAction`: un suono percussivo quale quello generato inviluppando una sinusoide, ad esempio con `Env.perc`, genera il tipico inviluppo percussivo senza sostegno. Ma che succede quando l'inviluppo è terminato? Che fine fa il synth? Senza deallocazione esplicita, resta attivo: `EnvGen` continua a generare un segnale d'ampiezza pari all'ultimo valore calcolato, e un'altra UGen responsabile della sinusoide, `SinOsc` un altro segnale. Negli esempi precedenti, il valore finale degli inviluppi tipicamente era $= 0$, e dunque l'evento "terminava" da un punto di vista acustico, ma non dal punto di

vista tecnico. La situazione è chiara nella riga 1 dell'esempio successivo, in cui l'inviluppo è di fatto una rampa che cresce in 2 secondi fino a valore $= 1$, e lì rimane. Anche se il valore dell'inviluppo fosse $= 0$, spetta comunque all'utente la rimozione del synth, nei casi precedenti un volta che questi ha generato il segnale percussivo richiesto. Una sequenza di 20 suoni percussivi, senza deallocazione, allocherebbe un numero analogo di synth, con intuibile spreco di RAM e di CPU. L'argomento `doneAction` permette di evitare il lavoro esplicito di deallocazione, poiché esso viene preso in carico direttamente da scsynth. L'argomento `doneAction` permette di specificare che cosa il server debba fare del synth in questione una volta terminata la lettura dell'inviluppo. I valori possibili sono attualmente (ben) 14. Ad esempio con `doneAction = 0` scsynth non fa nulla e il synth resta allocato e funzionante. Il valore più utilizzato è `doneAction = 2`, che dealloca il synth, che è dimostrato nella synthDef dell'esempio. In sostanza, quando l'inviluppo termina, il synth non c'è più e non c'è da preoccuparsi della sua esistenza. Quando si valuta la riga 12, si osservi cosa succede nella finestra del server al numero dei synth. Ovviamente, una volta deallocato dall'inviluppo, il synth non è più disponibile. Se si volesse generare un'altra sinusoide percussiva sarebbe necessario costruire un altro synth dalla stessa synthDef.

```
1  { EnvGen.kr(Env([0,1], [2], doneAction: 0)) * SinOsc.ar(mul: 0.1) }.play ;

3  (
4  SynthDef.new("sinePerc",
5      { Out.ar(0,
6          EnvGen.kr(Env.perc, 1.0, doneAction: 2)
7          *
8          SinOsc.ar(mul: 0.1))
9  }).add ;
10 )

12 Synth.new("sinePerc") ;
```

Si consideri l'esempio seguente in cui il mouse funziona come un trigger. Prima di valutare l'esempio il mouse deve essere nel lato sinistro dello schermo. Se `doneAction = 0` (1) il synth è residente e ad ogni passaggio del mouse può essere attivato grazie al trigger. Se `doneAction = 2` (riga 3) dopo la prima

attivazione il synth è deallocato e il comportamento del mouse non può inviare alcun messaggio. Nell'interfaccia grafica del server è visibile il numero dei synth attivi: si nota allora l'incremento del numero quando è creato il synth (implicitamente) e il diverso comportamento (persistenza/decremento) in funzione di `doneAction`.

```
1 {EnvGen.kr(Env.perc, MouseX.kr(-1,1),doneAction:0) * SinOsc.ar(mul:0.1)}.play;

3 {EnvGen.kr(Env.perc, MouseX.kr(-1,1),doneAction:2) * SinOsc.ar(mul:0.1)}.play;
```

I due valori tipi dell'argomento `doneAction`, cioè 1 e 2, di fatto rappresentano due accezioni diverse del synth. Da un lato, infatti, un synth è, come esplicitato dal nome, uno strumento, un sintetizzatore. Dunque, una volta creato, è inteso rimanere disponibile per poter essere controllato, esattamente come avviene per un sintetizzatore a tastiera, in cui lo strumento certamente non è pensato per dissolversi dopo che si è premuto un tasto (dunque, `doneAction:0`). Ma se invece si pensa al concetto di evento sonoro, orientato verso il suono prodotto piuttosto che verso lo strumento che lo produce, allora è sensato pensare che sotto certe condizioni si producano eventi sonori che ovviamente hanno una certa durata, e che perciò finiscono (`doneAction:2`). La deallocazione automatica permette perciò di trasformare uno strumento in un evento.

L'esempio seguente crea una sorta di mini-strumento con interfaccia grafica a tastiera di un'ottava, ed è composto da due espressioni. La prima crea la finestra (1), la seconda è un ciclo che fa tutto il resto. Per 12 volte (una per nota) il ciclo:

1. definisce la nota sommando il contatore alla codifica midi del do centrale (60);
2. crea un pulsante posizionandolo in progressione da sinistra a destra nella finestra (4);
3. ne definisce l'unico stato: il testo è il nome della nota (6), in bianco (7) su sfondo che progressivamente (in funzione del contatore) copre la ruota delle tinte (8);
4. associa alla pressione del pulsante l'esecuzione di un synth, che genera un'onda quadra con inviluppo percussivo, la cui frequenza è relativa all'altezza

della nota. Il synth non è assegnato a variabile, perché semplicemente produce l'evento sonoro richiesto e si elimina (`doneAction:2`). È chiaro che qui l'equazione è synth = nota.

```
w = Window("miniHarp", Rect(100, 100, 12*50, 50)).front ;
12.do{|i|
    var note = 60+i;
    Button(w, Rect(i*50, 0, 50, 50))
    .states_([[
        note.midinote[0..1],
        Color.white,
        Color.hsv(i/12, 0.7, 0.4)]])
    .action_{
        {Pulse.ar(note.midicps)*EnvGen.kr(Env.perc, doneAction:2)}.play
    }

}
```

6.2 Generalizzazione degli inviluppi

Sebbene l'uso degli inviluppi sia tipico per l'ampiezza, va comunque ricordato che `EnvGen` è di fatto un lettore di dati in un formato speciale, quello definito dalla classe `Env`. Un oggetto `Env` è in effetti del tutto analogo ad una cosiddetta tabella in cui sono tabulati punti che descrivono un profilo (break point). `EnvGen` è allora un'unità di lettura di tabelle speciali (di tipo `Env`) il cui contenuto può essere utilizzato per controllare parametri di diverso tipo. Si consideri il codice seguente:

```
1 {
2     var levels, times, env ;
3     levels = Array.fill(50, { arg x ; (x%7)+(x%3)}).normalize ;
4     times = Array.fill(49, 1).normalizeSum ;
5     env = Env.new(levels, times) ;

7     Pulse.ar(EnvGen.kr(env, 1, 100, 200, 20, 2))
8 }.play
```

Sono allora possibili alcune considerazioni sulla synthDef.

Intanto, l'inviluppo. Le righe 2-5 definiscono un inviluppo `env`, costituito da 50 punti d'ampiezza (3) intervallati da 49 durate (4). In particolare `levels` è un array che contiene una spezzata data da due cicli di iterazioni, poiché usa il modulo (`%`) 7 e 3. La progressività è data dal moltiplicatore `*x`, dove `x` è l'indice dell'elemento calcolato. L'utilizzo del modulo è interessante per produrre agevolmente curve complesse ma non casuali. L'array risultante è in Figura **4.15**.

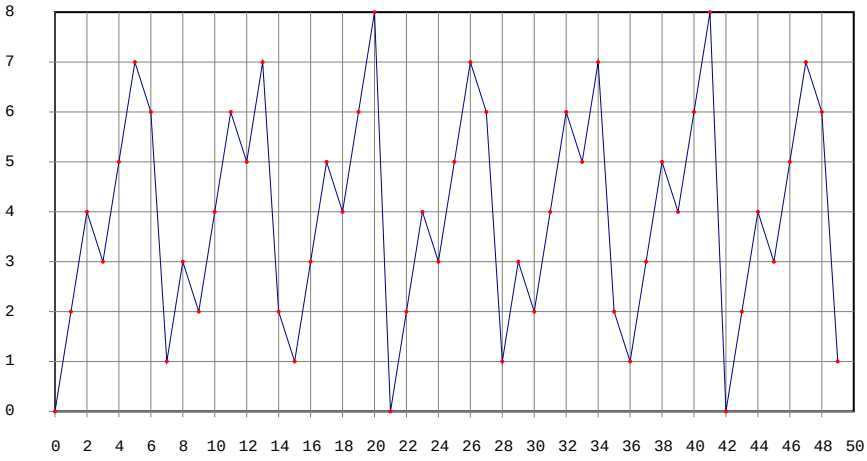


Fig. 6.1 Array costruito attraverso l'operatore modulo.

Essendo la periodicità di entrambi pari rispettivamente a 7 e 3, i due contatori andranno in fase nel punto del loro minimo comune multiplo 21. Il massimo è dato da $6 + 2 = 8$ (entambi al massimo) e il minimo ovviamente da $0 + 0 = 0$

(si osservino i punti a modulo 21). Il metodo `normalize` scala i valori dell'array nell'intervallo $[0.0, 1.0]$. I punti d'ampiezza calcolati per `levels` sono pensati come equispaziati. È per questo che la definizione di `times` produce semplicemente un array riempito dal valore 1. Il metodo `normalizeSum` restituisce un array `array/array.sum`, dove a sua volta il metodo `sum` restituisce la somma degli elementi dell'array in questione. L'esempio seguente dimostra interattivamente l'uso di `normalize, sum, normalizeSum`.

```
[1,2,3,4,5].sum // somma
15

[1,2,3,4,5].normalize // max e min tra 0 e 1
[ 0, 0.25, 0.5, 0.75, 1 ]

[1,2,3,4,5].normalizeSum // somma elementi = 1
[ 0.066666666666667, 0.13333333333333, 0.2, 0.26666666666667,
    0.33333333333333 ]

[1,2,3,4,5].normalizeSum.sum // somma normalizzata sommata = 1
1
```

I due array `levels` e `times` sono allora nella forma normalizzata preferibile. L'idea alla base della synthDef è quella di utilizzare l'inviluppo per controllare la frequenza di `Pulse`. Dunque, `levelScale` è pari a 100 mentre `levelBias` è 200: la curva descritta si muoverà nell'intervallo $[200, 300]$ Hz, per un totale di 20 secondi (`timeScale`), quindi il synth si deallocherà (`doneAction:2`). La curva è una rampa, e intuibilmente produce un glissando. Nel prossimo esempio, viene invece utilizzata un'altra UGen, `Latch`.

```
{
    var levels, times, env ;
    levels = Array.fill(50, { arg x ; (x%7)+(x%3)}).normalize ;
    times = Array.fill(49, 1).normalizeSum ;
    env = Env.new(levels, times) ;

    Pulse.ar(Latch.kr(EnvGen.kr(env, 1, 100, 200, 20, 2), Impulse.kr(6)))
}.play
```

`Latch` implementa un algoritmo classico, il cosiddetto *Sample & Hold*. Tutte le volte che riceve un segnale di trigger, `Latch` campiona il segnale che ha in ingresso, e genera in uscita quel valore finché un nuovo trigger non innesca una nuova pesca dal segnale in ingresso. Nella synthDef il trigger è `Impulse`, un generatore di singoli campioni, con frequenza $= 6$: ad ogni periodo $T = \frac{1}{6}$, `Impulse` genera un singolo impulso di ampiezza `mul`. Si ricordi che la forma del segnale di triggering non è importante se non per quando si attraversa lo zero dal negativo al positivo. Dunque la curva `env` di 20 secondi è campionata ogni $\frac{1}{6} = 0.16666666666667$ secondi. Ne risulta un effetto "notale". Si potrebbe espandere ulteriormente quest'effetto. Nell'esempio seguente, `levelScale` è pari a 24 e `levelBias` a 60.

```
{
    var levels, times, env ;
    levels = Array.fill(50, { arg x ; (x%7)+(x%3)}).normalize ;
    times = Array.fill(49, 1).normalizeSum ;
    env = Env.new(levels, times) ;

    Pulse.ar(Latch.kr(EnvGen.kr(env, 1, 24, 60, 20, 2).poll.midicps.poll,
        Impulse.kr(6)))
}.play
```

Dunque, l'intervallo in cui si muove l'inviluppo in ampiezza è $[60, 84]$. L'idea è di ragionare non in frequenza ma in altezza, come se ci si trovasse sulle due ottave a partire dal do centrale del piano (codificato in MIDI come 60), per poi convertirlo attraverso `midicps`. La cosa rilevante in quest'ultimo punto è che `midicps` è un operatore di conversione: in generale, su tutti i segnali in uscita dalle UGen è possibile applicare operatori matematici, ad esempio, `squared` o `abs`, per tornare a casi menzionati. Osservando il codice, si nota anche che è stato inserito due volte il metodo `poll`. Detto in maniera molto approssimativa, quest'ultimo è una sorta di analogo a lato server di `postln`. L'architettura client/server di SC pone un problema di rilievo che spesso risulta nascosto. Il processo di sintesi è controllato dal client, ma è realizzato dal server. Il cliente dice cosa fare al fornitore di servizi il quale svolge la sua azione ottemperando alle richieste. Ma come può sapere il client cosa succede dentro il server? Come si fa a sapere se non si sono commessi errori di implementazione? Il solo feedback audio non è sufficiente (il fatto che il risultato sia interessante indipendentemente dalla correttezza dell'implementazione è senz'altro positivo, ma la

serendipità non aiuta il debugging ...). Il feedback visivo (attraverso i metodi `scope` e `plot` ad esempio) non è analitico, cioè non comunica direttamente cosa succede in termini di campioni audio calcolati. Il metodo `poll`, definito sulle UGen, dice al server di inviare indietro al client il valore di un campione audio, ad un tasso impostabile come argomento del metodo, e di stamparlo sulla post window. Questo permette di monitorare cosa succede a livello campione nel server in uscita da ogni UGen. In sostanza, il metodo può essere concatenato dopo i metodi `ar` e `kr`. Ad esempio:

```
{SinOsc.ar(Line.ar(50, 10000, 10).poll).poll}.play ;
```

produce

```
Synth("temp__1198652111" : 1001)
UGen(Line): 50.0226
UGen(SinOsc): 0.00712373
UGen(Line): 149.523
UGen(SinOsc): -0.142406
UGen(Line): 249.023
UGen(SinOsc): -0.570459
UGen(Line): 348.523
UGen(SinOsc): -0.982863
UGen(Line): 448.023
UGen(SinOsc): -0.616042
UGen(Line): 547.523
UGen(SinOsc): 0.676455
```

Per ogni UGen, `poll` stampa sullo schermo il valore del segnale in uscita. Si noti come SinOsc oscilli in $[-1, 1]$ mentre Line inizi la progressione lineare da 50 a 10000.

Tornando all'ultima synthDef, la sequenza `poll.round.poll.midicps.poll` stampa prima il valore in uscita da EnvGen, quindi il suo arrotondamento, infine la sua conversione in Hz. Il metodo `poll` (che deriva da una UGen, `Poll`) effettua una operazione interessante, che verrà discussa in seguito ma che vale la pena sottolineare: invia messaggi *dal* server *al* client, mentre finora la comunicazione è avvenuta unidirezionalmente nel verso opposto. Se si legge l'output

sullo schermo si nota come le note midi siano espresse con numeri con la virgola. In altri termini, la notazione midi è un modo in cui può essere gestita una altezza, ma non necessariamente quest'ultima deve essere temperata (una "nota"). Ponendosi come esercizio proprio quest'ultimo obiettivo, si può utilizzare un operatore di arrotondamento, `round`, come nell'esempio seguente.

```
{
    var levels, times, env ;
    levels = Array.fill(50, { arg x ; (x%7)+(x%3)}).normalize ;
    times = Array.fill(49, 1).normalizeSum ;
    env = Env.new(levels, times) ;

    Pulse.ar(Latch.kr(EnvGen.kr(env, 1, 24, 60, 20, 2)
        .poll.round.poll.midicps.poll,
        Impulse.kr(6)))
}.play
```

In questo modo, un segnale continuo diventa doppiamente discreto: da un lato, si producono 6 note al secondo attraverso `Latch`, dall'altro il segnale in altezza viene riportato ad una griglia di interi ($[60, 84]$). Di fatto, è una sorta di nuova digitalizzazione del segnale. L'operatore `round` permette ovviamente di definire l'unità di arrotondamento: nell'esempio, se fosse $= 0.5$, si arrotonderebbe al quarto di tono.

La Figura **6.2** mostra un frammento dello UGen graph della synthDef, poiché la struttura di `Env` è molto grande (sono 50 posti). Si noti la presenza della UGen `Poll` (che nella synthDef "c'è ma non si vede"), il suo trigger interno (una UGen `Impulse` con `freq` $= 10$) e gli operatori `Round` e `MIDICPS`.

Il grafo mette in luce un punto. Gli operatori matematici disponibili sulle UGen non sono altro che UGen speciali, per la precisione di tipo `BinaryOpUGen` e `UnaryOpUGen`. L'utente non se ne deve preoccupare, ma l'osservazione serve per rimarcare come il server conosce soltanto UGen all'interno della definizione di una synthDef. La situazione è descritta in maniera minimale dai due esempi sottostanti:

```
{SinOsc.ar(200).round(0.5)}.play ;
{SinOsc.ar(200).abs-0.5}.play ;
```

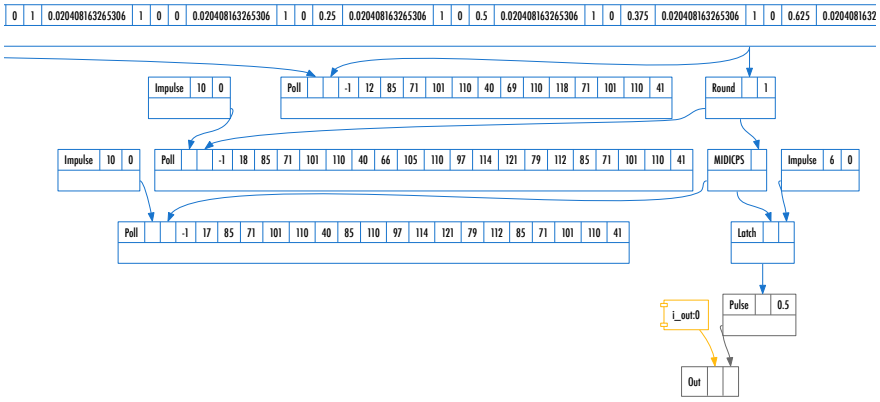


Fig. 6.2 UGen graph della synthDef.

I cui grafi ed i segnali sono invece in Figura **6.3**.

6.3 Sinusoidi & sinusoidi

L'ipotesi alla base dell'utilizzo di un inviluppo d'ampiezza sta nel controllo di un segnale da parte di un altro segnale in modo da ottenere un risultato più "complesso", "naturale", "interessante" etc. Un segnale di inviluppo è un segnale unipolare, ma è evidentemente possibile utilizzare segnali bipolari. In particolare la sinusoide non è soltanto la forma d'onda che produce lo spettro più semplice ma è anche la forma tipica che assume una variazione regolare intorno ad un punto di equilibrio. Dunque, una sinusoide descrive opportunamente un fenomeno di oscillazione intorno ad un valore medio. Si considerino due casi:

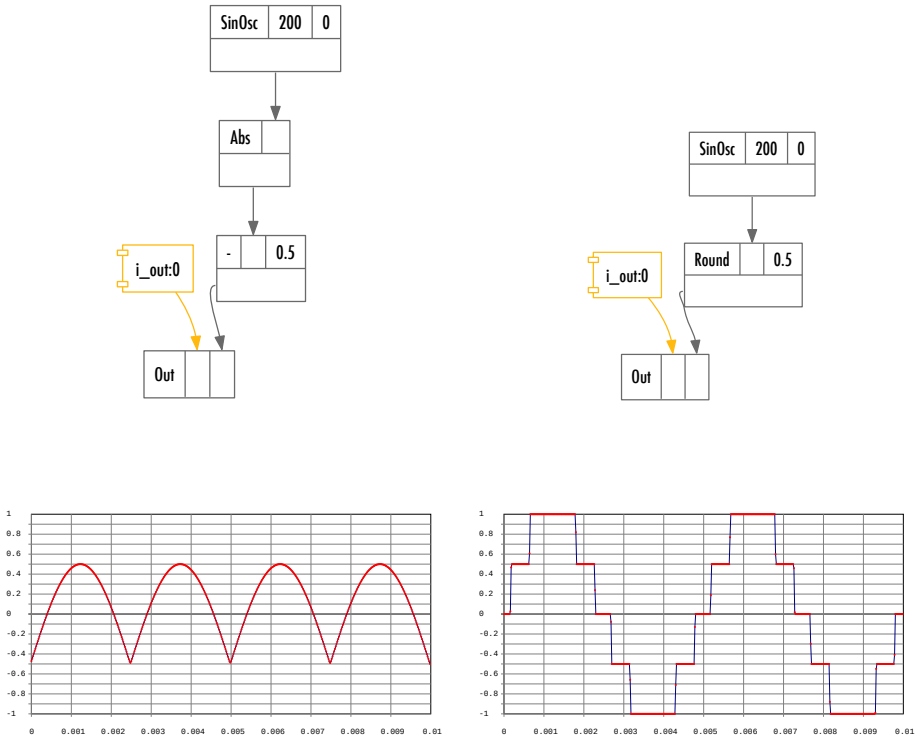


Fig. 6.3 OpUgen: grafi e segnali.

```
1  // tremolo minimale
2  { SinOsc.ar(mul: 0.5+SinOsc.kr(5, mul: 0.1)) }.play ;
3  // vibrato minimale
4  { SinOsc.ar(freq: 440+SinOsc.kr(5, mul: 5)) }.play ;

6  // con MouseX/Y
7  // tremolo
8  { SinOsc.ar(mul: 0.5 + SinOsc.kr(
9              freq: MouseX.kr(0, 10),
10             mul: MouseY.kr(0.0, 0.5))) }.play ;
11 // vibrato
12 { SinOsc.ar(freq: 440 + SinOsc.kr(
13             freq: MouseX.kr(0, 10),
14             mul: MouseY.kr(0, 10))) }.play ;
```

- **tremolo**: in musica un tremolo è una variazione periodica della dinamica, ovvero dell'ampiezza come dimensione percettiva. L'implementazione di un tremolo è evidentemente semplice. È sufficiente sommare all'ampiezza del primo oscillatore il segnale prodotto da un oscillatore di controllo: l'incremento varierà periodicamente con frequenza pari a quella dell'oscillatore di controllo da 0 fino al massimo (l'ampiezza dello stesso oscillatore), discenderà a 0 e al massimo negativo per ritornare infine a 0. Nell'esempio l'argomento `mul` contiene una costante (0.5) a cui viene sommato il segnale in uscita da un oscillatore che varia 5 volte al secondo nell'escursione $[-0.1, 0.1]$. Dunque, con la stessa frequenza l'ampiezza dell'oscillatore portante (audio) varierà nell'intervallo $[0.4, 0.6]$. Il contributo dell'oscillatore di controllo è allora una variazione periodica dell'ampiezza del segnale controllato, variazione il cui periodo è specificato dalla frequenza dell'oscillatore di controllo. Il suono sintetico acquista così una caratteristica tipica dell'esecuzione strumentale (per esempio, dei fiati). Nell'esempio alle righe 8-10 viene utilizzato il mouse per controllare i due parametri del tremolo;

- **vibrato**: se si applica il ragionamento svolto per il tremolo questa volta non all'ampiezza ma alla frequenza, si ottiene un vibrato. Un oscillatore di controllo controlla un incremento (minimo) della frequenza dell'oscillatore principale. Supponendo che f_1, amp_1, f_2, amp_2 siano frequenza e ampiezza rispettivamente dell'oscillatore audio e di quello di controllo, la frequenza

dell'oscillatore audio (f_1, finora costante) dopo l'incremento varia periodicamente (secondo la frequenza dell'oscillatore f_2 di controllo) tra $f_1 - amp_2$ e $f_1 + amp_2$. Si ricordi che l'uscita del segnale di controllo è infatti sempre una variazione d'ampiezza $\pm amp_2$: questa variazione si somma in questo caso alla frequenza dell'oscillatore controllato f_1. Nell'esempio (5), la variazione di frequenza è data dalla somma alla costante 440 del segnale di un oscillatore che 5 volte al secondo oscilla tra $[-5, 5]$ (si veda `mul`): dunque per 5 volte al secondo la frequenza audio varierà tra $[435, 445]$. Analogamente a quanto visto per il tremolo, è fornito anche un esempio in cui frequenza e ampiezza del vibrato sono controllabili via mouse. Il risultato musicale di una simile variazione periodica dell'altezza di una nota viene definito vibrato. Il periodo del vibrato dipende dal periodo dell'oscillatore di controllo: si pensi a un violinista che sposta di poco ma continuamente il suo dito attorno alla posizione della nota prescelta. Ancora: nella tipica tecnica del canto operistico si mettono in atto simultaneamente tremolo e vibrato.

Un esempio riassuntivo è il seguente:

```
1  SynthDef("tremVibr",
2          { arg     freq = 440, mul = 0.15,
3                  tremoloFreq = 5 , tremoloMulPercent = 5,
4                  vibratoFreq = 10, vibratoMulPercent = 5  ;
5          var tremoloMul = mul*tremoloMulPercent*0.01 ;
6          var vibratoMul = freq*vibratoMulPercent*0.01 ;
7          var tremolo = SinOsc.kr(tremoloFreq, 0, tremoloMul) ;
8          var vibrato = SinOsc.kr(vibratoFreq, 0, vibratoMul) ;
9          var sinOsc = SinOsc.ar(freq+vibrato, 0, mul+tremolo) ;
10         Out.ar(0, sinOsc) ;

12 }).add ;
```

La synthDef `"tremVibr"` prevede argomenti per il controllo dell'oscillatore audio e di tremolo e vibrato (si noti come tutti abbiano valori predefiniti). Sia del tremolo che del vibrato è possibile controllare la frequenza e l'incidenza. Le prima è in entrambi i casi descritta in termini assoluti, cioè attraverso i cicli al secondo (insomma, in Hz). Come si vede alle righe 8 e 9, `tremolo` e `vibrato` sono due segnali in uscita da due oscillatori sinusoidali che hanno appunto come `freq` rispettivamente `tremoloFreq` e `vibratoFreq`. Più interessante la descrizione dell'incidenza dei due parametri (ovvero di quanto variano il segnale

audio). In entrambi i casi l'incidenza è relativa allo stato del segnale da controllare. in altre parole, l'incidenza è proporzionale ed è descritta percentualmente. Si consideri la riga 5: `tremoloMul` è calcolato assumendo che `tremoloMulPercent`, il parametro controllabile dal synth, corrisponda ad una percentuale dell'ampiezza del segnale. Se `mul = 0.5` e `tremoloMulPercent = 10` allora `tremoloMul` sarà pari al 10 di `mul`, cioè a 0.05. Dunque, il segnale assegnato a `tremolo` sarà compreso nell'escursione $[-0.05, 0.05]$ e l'ampiezza del segnale audio assegnato alla variabile `sinOsc` oscillerà nell'intorno $[0.45, 0.55]$. Si ricordi che `sinOsc` è appunto una variabile, da non confondere con la UGen `SinOsc`): attraverso la variabile `sinOsc` il segnale viene passato come argomento ad `Out`. Un discorso analogo al calcolo del tremolo vale per il vibrato con il calcolo di `vibratoMul`. Nell'esempio, viene quindi creato un synth `aSynth` in cui viene annullato in fase di creazione il contributo di tremolo e vibrato assegnando un valore pari a 0 ai due argomenti `tremoloMulPercent` e `vibratoMulPercent` (non c'è incidenza di tremolo e vibrato poiché i due segnali di controllo hanno ampiezza nulla).

A partire da una simile synthDef è possibile costruire una interfaccia grafica per controllare gli argomenti di un synth. L'idea di partenza è avere per ogni parametro un cursore e di visualizzarne il nome e il valore. Dato il numero esiguo di parametri si potrebbe semplicemente costruire l'interfaccia "artigianalmente", definendo cioè singolarmente tutti i componenti GUI necessari. Vale la pena però sperimentare un altro approccio, più automatizzato. Le classi di elementi necessarie sono due:

1. `Slider`: permette di costruire un cursore (in inglese, uno *slider*): la sintassi del costruttore è piuttosto ovvia, giacché richiede una finestra di riferimento (qui `window`[1]) e un rettangolo in cui viene disegnato il cursore.
2. `StaticText`: è un campo di testo per la visualizzazione, che però non prevede riconoscimento dell'input (scrive testo sullo schermo ma non serve per immetterlo). La sintassi del costruttore è quella consueta, e definisce un riquadro ove verrà stampato sullo schermo il testo. In più viene chiamato sull'istanza così ottenuta il metodo `string_` che definisce il testo da stampare.

Si tratta di definire un array che contenga i nomi degli argomenti, e a partire da questo di

[1] Si noti come window sia resa più alta del richiesto aggiungendovi 30×2 pixel in altezza.

1. generare gli oggetti grafici
2. definire le azioni associate a questi ultimi

L'unica difficoltà sta nel fatto che nel connettere valori in uscita dal cursore e argomenti del synth è necessario tenere in conto di un mapping disomogeneo. Ogni cursore ha un'estensione compresa in $[0, 1]$. La frequenza `freq` dell'oscillatore audio può avere ad esempio un'escursione compresa in $[50, 10000]$, mentre la sua ampiezza `mul` è invece compresa in $[0, 1]$, le frequenze di tremolo e vibrato sono tipicamente comprese in un registro sub-audio, $[0, 15]$, mentre le due incidenze devono essere espresse percentualmente, dunque in $[0, 100]$. È perciò necessario un lavoro di scalatura dei valori espressi da ogni cursore in quanto relativo ad un preciso parametro. Non è opportuno passare i valori non scalati al synth includendo dentro la synthDef le operazioni necessarie allo scaling: in questo modo si verrebbe infatti a definire una dipendenza della synthDef dall'interfaccia grafica. Nel lavoro con le interfacce grafiche è invece bene assumere che il modello e i dati siano indipendenti dall'interfaccia di controllo/display. In questo modo, si può buttare via l'interfaccia senza dover modificare il modello (qui, la synthDef). D'altra parte, nella definizione di un algoritmo di sintesi quale previsto dalla synthDef è del tutto fuori luogo prevedere elementi che invece concernono la GUI. La soluzione proposta dell'esempio prevede l'uso di un `IdentityDictionary`, ovvero di una struttura dati che associa in maniera univoca ad una chiave un valore, secondo il modello del dizionario che prevede per ogni lemma (la chiave) una definizione (il valore): `controlDict` associa ad ogni stringa-argomento un array che ne definisce l'escursione (secondo il modello `[minVal, maxVal]`). La parte di codice compresa tra 14 e 24 definisce tre array che contengono un numero pari alla dimensione di `controlDict` rispettivamente di blocchi di testo per la visualizzazione del nome del parametro, cursori e blocchi di testo deputati alla visualizzazione del valore degli argomenti. Quindi viene istanziato un synth. Infine si tratta di definire la funzionalità della GUI attraverso un ciclo su ognuno degli elementi contenuti in `controlDict`. Nel ciclo, `value` indica l'elemento di destra (l'escursione), ed a partire da questa si ottiene la chiave `name` attraverso l'invocazione del metodo `controlDict.findKeyForValue(value)`. Ad esempio, se `value` è `[50, 1000]`, la chiave associata a `name` sarà `"freq"`. Quindi viene calcolata un'escursione `range` tra $[0, 1]$ come differenza tra gli estremi (nell'esempio, $1000 - 50 = 950$) e il minimo viene considerato come scarto di partenza (`offset`). L'espressione `labelArr[index].string_(name)` assegna all'elemento di testo `index` in `labelArr` la stringa `name` che vi risulta visibile (nell'esempio, `"freq"`). Infine, `slidArr[index].action = ...` assegna all'elemento `index` di `slidArr` un'azione attraverso

il consueto meccanismo di una funzione che viene valutata ogni qualvolta si registra una variazione nello stato del cursore (un movimento della leva grafica). Nella funzione l'argomento (qui: `theSlid`) è come di consueto l'istanza dell'elemento grafico su cui è definita la funzione. L'azione realizza quattro comportamenti, ognuno descritto da un'espressione alle righe 32-35.

1. dichiarando la variabile `paramValue` se ne calcola il valore. Nell'esempio precedentemente relativo a `"freq"`, poiché `theSlid.value` è sempre compreso in $[0, 1]$ ed essendo $range = 50$, l'escursione varia appunto tra $[0, 950] + 50 = [50, 1000]$;
2. la seconda azione consiste nello stampare sullo schermo il nome del parametro ed il suo valore;
3. si tratta di controllare il synth impostando per l'argomento `name` il valore `paramValue` (ovvero: `aSynth.set("freq", 123.4567890);`
4. infine, il valore `paramValue` viene scritto nell'elemento di testo `index` in `valueArr`.

```
var aSynth =  Synth.new("tremVibr");
var controlDict = IdentityDictionary[
    "freq"                ->    [50, 1000],
    "mul"                 ->    [0,1],
    "tremoloFreq"         ->    [0, 15],
    "tremoloMulPercent"   ->    [0, 50],
    "vibratoFreq"         ->    [0, 15],
    "vibratoMulPercent"   ->    [0, 50]
];

var window = Window.new("Vibrato + tremolo",
    Rect(30,30, 900, controlDict.size+2*30)) ;

var labelArr = Array.fill(controlDict.size, { arg index ;
    StaticText( window, Rect( 20, index+1*30, 200, 30 )).string_( 0 ) ;
}) ;

var slidArr = Array.fill(controlDict.size,
    { |index| Slider( window, Rect( 240, index+1*30, 340, 30 )) }) ;

var valueArr = Array.fill(controlDict.size,
    { |index| StaticText( window, Rect( 600, index+1*30, 200, 30 ))
        .string_( 0 ) ;
}) ;

controlDict.do({ arg value, index ;
    var name = controlDict.findKeyForValue(value) ;
    var range = value[1]-value[0] ;
    var offset = value[0] ;
    labelArr[index].string_(name) ;
    slidArr[index].action = { arg theSlid ;
        var paramValue = theSlid.value*range + offset ;
        [name, paramValue].postln ;
        aSynth.set(name, paramValue) ;
        valueArr[index].string_(paramValue.trunc(0.001) ) ;
    }
}) ;

window.front ;
```

6.4 Segnali pseudo-casuali

La discussione precedente in relazione al controllo di un oscillatore da parte di un altro oscillatore può essere intuibilmente estesa alla più generale prospettiva del controllo di una UGen da parte di un'altra UGen. I segnali di controllo precedenti erano sinusoidi: in particolare la forma d'onda sinusoidale era appunto il profilo temporale della variazione indotta nel segnale. Un profilo di variazione può evidentemente assumere forme diverse. Negli esempi seguenti le prime funzioni contengono un segnale di controllo che verrà impiegato per controllare il vibrato di un oscillatore (con frequenza sub-audio e a tasso di controllo) nelle seconde. Attraverso il metodo `scope` è possibile visualizzare il profilo dei primi segnali di controllo (per comodità di visualizzazione la frequenza è stata modificata a 1000 Hz). In generale le UGen che iniziano con `LF` sono specializzate nella generazioni di segnali di controllo.

```
1  { SinOsc.ar(1000) }.scope ;
2  { SinOsc.ar(freq: 440+SinOsc.kr(2, mul: 50), mul: 0.5) }.play ;

4  { LFSaw.ar(1000) }.scope ;
5  { SinOsc.ar(freq: 440 + LFSaw.kr(2, mul: 50), mul: 0.5) }.play ;

7  { LFNoise0.ar(1000) }.scope ;
8  { SinOsc.ar(freq: 440 + LFNoise0.kr(2, mul: 50),mul: 0.5) }.play ;

10 { LFNoise1.ar(1000) }.scope ;
11 { SinOsc.ar(freq: 440 + LFNoise1.kr(2, mul: 50), mul: 0.5) }.play ;
```

Si nota immediatamente la differenza tra i due segnali periodici, la sinusoide e l'onda a dente di quadra, proprio in relazione al profilo udibile della variazione. Le due altre UGen impiegate sono specializzate nella generazione di segnali pseudo-casuali. In particolare `LFNoise0` generata ad una certa frequenza valori d'ampiezza che tiene per la durata di ogni periodo (cioè, fino quando non calcola un un nuovo valore): se si osserva la finestra attivata quando `scope`

è invocato sulla UGen si nota l'andamento a gradoni. Come `LFNoise0`, `LFNoise1` genera segnali pseudo-casali alla frequenza impostata, ma interpola tra gli estremi successivi. In altre parole, non salta da un valore al successivo, ma vi arriva gradualmente. L'esempio seguente è minimale ed il comportamento dei due generatori ne risulta chiaramente illustrato.

```
{SinOsc.ar(LFNoise1.ar(10, mul:200, add: 400))}.play ;
{SinOsc.ar(LFNoise0.ar(10, mul:200, add: 400))}.play ;
```

In entrambi i casi la frequenza dell'oscillatore è controllata da un generatore di numeri casuali che aggiorna il suo stato 10 volte al secondo. In considerazione degli argomenti `mul` e `add` il segnale di controllo della frequenza varia casualmente allora nell'escursione $[-1, 1] \times 200 + 400 = [200, 600]$. Con `LFNoise0` si ha un *gradatum* (di tipo scalare), con `LFNoise1` un *continuum* (un glissando). Si potrebbe pensare a `LFNoise0` come ad una versione di `LFNoise1` *sampled and held*: si prende un campione di un segnale che interpola tra valori pseudo-casuali quale quello generato da `LFNoise1` ad un ogni intervallo di tempo prestabilito e lo si tiene fino al prelevamento successivo. Come si è visto, un simile comportamento è realizzato da `Latch`. L'esempio seguente è una variazione di un esempio dall'help file di `Latch`. In entrambi i casi il segnale in uscita varia la sua frequenza ad un tasso di 9 volte al secondo scegliendo casualmente nell'intervallo $[100, 900]$.

```
{ SinOsc.ar(LFNoise0.kr(9, 400, 500), 4, 0.2)}.play ;

// lo stesso, ma meno efficiente
{ SinOsc.ar(Latch.ar(LFNoise1.ar, Impulse.ar(9)) * 400 + 500, 4, 0.2) }.play;
```

Per la sua natura discreta `LFNoise0` può essere utilizzato come componente di un sequencer, cioè come generatore di sequenze discrete di valori. Due variazioni sul tema sono presentate nell'esempio seguente.

```
SynthDef(\chromaticImproviser , { arg freq = 10 ;
    Out.ar(0, SinOsc.ar(
        freq:     LFNoise0.kr(freq, mul:15, add: 60).round.midicps,
        mul:
        EnvGen.kr(Env.perc(0.05), gate: Impulse.kr(freq), doneAction:2)
    )
}).play ;

SynthDef(\modalImproviser , { arg freq = 10;
    var scale = [0, 0, 0, 0, 3, 3, 4, 5, 5, 6, 6, 7, 7, 7, 10]+60 ;
    var mode = scale.addAll(scale+12).midicps ;
    var range = (mode.size*0.5).asInteger ;
    Out.ar(0, SinOsc.ar(
        freq:     Select.kr(LFNoise0.kr(freq,
            mul: range,
            add: range).round, mode),
        mul:
        EnvGen.kr(Env.perc(0.05), gate: Impulse.kr(freq), doneAction:2)
    )
}).play ;
```

I due "improvvisatori" generano sequenze casuali di altezze ad un tasso predefinito di 10 note al secondo. In entrambi i casi il generatore audio è un oscillatore sinusoidale con un inviluppo percussivo. Le due synthDef esemplificano anche la possibilità (non molto usata) di inviare il messaggio `play` direttamente alla synthDef, con invio d quest'ultima al server e generazione automatica di un synth.

La prima synthDef, vagamente *hard bop*, genera una sequenze di altezze cromatiche comprese tra in midi tra $[45, 75]$. Nel midi sono numerate linearmente le altezze, come se si trattasse di assegnare un indice ai tasti (bianchi e neri) di un pianoforte. In notazione midi il *do centrale* è 60, dunque le altezze variano tra il quindicesimo tasto che lo precede, corrispondente al *la* al di sotto dell'ottava inferiore, e il quindicesimo tasto seguente, il *mi bemolle* al di sopra dell'ottava superiore. Infatti la frequenza dell'oscillatore audio è data dalla sequenza di valori in uscita da `LFNoise0`, moltiplicati per 15 e sommati a 60 (secondo quanto previsto da `mul, add`). L'esempio riprende molti aspetti già discussi.

La seconda synthDef gestisce le altezze in maniera diversa. In primo luogo `scale` definisce una scala che comprende le altezze indicate nell'array. Si noti che all'array viene aggiunto `60`, dunque le altezze (in notazione midi) saranno:

```
[0, 0, 0, 0, 3, 3, 4, 5, 5, 6, 6, 7, 7, 7, 10]+60
[ 60, 60, 60, 60, 63, 63, 64, 65, 65, 66, 66, 67, 67, 67, 70 ]
```

ovvero *do, mi bemolle, mi, fa, fa diesis, sol, sib*, alcune delle quali ripetute. Il modo (cioè la sequenza effettiva) su cui il generatore improvvisa è definito come `scale` a cui viene aggiunta una copia di se stesso trasposta di un'ottava. Si ricordi che in midi il salto di ottava, pari ad un raddoppio della frequenza, equivale ad un incremento di 12 (ovvero: 12 tasti sul pianoforte). L'array `mode` viene quindi convertito in Hz attraverso `midicps`. La scelta della frequenza da generare è determinata dalla UGen `Select`, i cui metodi `ar, kr` prevedono due argomenti: `which` e `array`. In sostanza, dato un `array`, `Select` ne restituisce l'elemento `which`. A scanso di equivoci, si ricordi che, essendo una UGen, `Select` restituisce un segnale, cioè una sequenza, a tasso audio o di controllo, composta dall'elemento `which` di `array`. L'idea è quella di far selezionare casualmente a `Select` una frequenza dall'array `mode`. Per fare ciò viene utilizzato `LFNoise0`. Quest'ultimo, a frequenza `freq`, genera un numero compreso in $[-r, +r] + r = [0, r \times 2]$, dove la variabile r (qui `range`) è definita come la metà intera della dimensione di `mode`. Quindi, se `mode.size = 26`, allora `range = 13`, e `LFNoise` oscillerà in $[0, 26]$. Essendo i valori in uscita da `LFNoise0` gli indici dell'array da cui `Select` legge, devono essere interi: di qui l'invocazione del metodo `round`. Si è notato che `mode` contiene più volte gli stessi valori: è un modo semplice (un po' primitivo, ma efficace...) per fare in modo che certe altezze vengano scelte con frequenze fino a quattro volte superiori ad altre. La scala enfatizza allora il primo, e poi il terzo minore, il quarto, il quarto aumentato e il quinto grado, ed è per questo (assai) vagamente blueseggiante. Infine, un'ultima annotazione sull'inviluppo d'ampiezza. Il segnale in uscita dall'oscillatore è continuo e deve essere inviluppato ogni volta che si produce un cambiamento di altezza, ovvero alla stessa frequenza di `LFNoise0`, in modo tale da sincronizzare altezze e ampiezza in un evento notale: per questo l'argomento `gate` riceve come valore il segnale in uscita da `Impulse`, ovvero una sequenza di impulsi che attivano l'inviluppo alla stessa frequenza `freq` di `LFNoise0`.

Nell'esempio precedente l'utilizzo di `EnvGen` richiede di specificare per l'argomento `gate` un trigger che operi alla stessa frequenza di `LFNoise0`: a tal fine è stata impiegata la UGen `Impulse`. Un segnale pseudo-casuale spesso utilizzato come trigger è quello generato da `Dust`: quest'ultima produce una sequenza di impulsi, nell'escursione $[0, 1]$, che è distribuita irregolarmente (stocasticamente)

nel tempo, alla densità media al secondo definita dal primo argomento `density`. Gli argomenti e relativi valori predefiniti dei suoi metodi `*ar` e `*kr` sono allora `density = 0.0, mul = 1.0, add = 0.0` . Se negli esempi precedenti si sostituisce a `Impulse` la UGen `Dust`, lasciando inalterato il resto del codice, si ottiene che gli inviluppi verrano generati in un numero medio pari a `freq` al secondo, ma non in una serie cronometricamente regolare. Nell'esempio seguente le ascisse del mouse controllano la densità, che varierà nell'escursione $[1, 500]$.

```
{ Dust.ar(MouseX.kr(1, 500)) }.play ;
```

L'utilizzo di `Dust` come trigger è piuttosto diffuso in SC. L'esempio seguente è tratto dall'help file di `TRand`, un altra UGen interessante: `TRand` genera un valore pseudo-casuale compreso nell'escursione descritta dai due primi argomenti `lo = 0.0` e `hi = 1.0`. La generazione avviene ogni qualvolta il trigger `trig` cambia stato, passando da negativo a positivo. Nell'esempio seguente la prima parte prevede come trigger `Impulse`: ad ogni nuovo impulso un `TRand` genera un nuovo valore utile alla frequenza di `SinOsc`. Si noti che è un altro modo per ottenere l'effetto *sample & hold* implementato con `Latch`. Se si sostituisce `Impulse` con `Dust`, si ottiene una sequenza di impulsi (e dunque di inviluppi) in una quantità media pari a `freq`.

```
// deterministico
{
    var trig = Impulse.kr(9);
    SinOsc.ar(
        TRand.kr(100, 900, trig)
    ) * 0.1
}.play;

// stocastico
{
    var trig = Dust.kr(9);
    SinOsc.ar(
        TRand.kr(100, 900, trig)
    ) * 0.1
}.play;
```

Degli elementi presenti sul server mancano ancora all'appello: i bus e buffers. È ora di introdurre i primi. Il concetto di bus deriva dall'audio analogico, e, come specificato dalla Figura **5.4**, può essere utilmente pensato nei termini di una sorta di tubo in cui scorre il segnale. Ogni bus riceve un indice (un identificativo numerico), attraverso il quale vi si può fare riferimento. Una prima distinzione è quella tra bus dedicati ai segnali di controllo e bus dedicati ai segnali audio. I primi sono necessariamente "interni" al server (il tasso di controllo è appunto un tasso di calcolo interno al server). I secondi hanno invece la funzione di veicolare segnali audio e per questo motivo è dunque possibile distinguere tra "condutture" che portano i segnali audio d/al di fuori del server (pubbliche) e conduttore audio per uso interno (private). Riguardo al primo caso, alcuni bus servono intuibilmente per connettere il server con la scheda audio, leggendovi i segnali in ingresso (ad esempio da un microfono) e scrivendovi i segnali in uscita (sui canali in uscita della scheda audio, ad esempio per poter arrivare agli altoparlanti). Poiché anche i bus audio hanno un loro indice associato, scsynth non fa differenza tra pubblico e privato: tutti i bus sono trattati allo stesso modo. Tuttavia, esiste una convenzione importanti. I primi bus audio $(0, ..., n)$ sono riservati alle uscite della scheda audio, i secondi alle entrate $(n + 1, ..., m)$ (i tubi di entrata/uscita), seguono i bus ad uso interno $(m + 1, ...)$ (i tubi di collegamento). Dunque, la distinzione tra pubblico e privato dipende dalla scheda audio utilizzata da SC. La configurazione standard con uscita stereo e ingresso microfono stereo prevede i bus $0, 1$ per i due canali stereo (i due altoparlanti) e i bus $2, 3$ per il microfono. Dal 4 in avanti, i bus possono essere utilizzati internamente.

Nell'esempio seguente una synthDef molto semplice permette di controllare attraverso l'argomento `out` il bus su cui viene scritto il segnale in uscita. La UGen `Out` è appunto quella dedicata a scrivere sui bus, e i suoi argomenti sono l'indice del bus e il segnale da scrivere. La riga 2 permette di monitorare visivamente i primi 4 bus (indici $0, 1, 2, 3$). Come si vede, il bus 0 è quello dedicato al canale sinistro nella classica configurazione stereo, mentre la riga 4 sposta il segnale sul canale destro (con indice 1). Quindi, le righe 5 e 6 muovono il segnale sui bus 2 e 3: se la scheda audio è stereo (come tutte quelle integrate nei computer) allora il segnale c'è (si vede nella finestra stethoscope) ma non si sente. Infine, il segnale viene instradato nuovamente sul canale sinistro.

```
1 SynthDef(\chan , {arg out; Out.ar(out, LFPulse.ar(5)*SinOsc.ar(400))}).add ;
2 s.scope(4) ; // monitor per i bus
3 x = Synth(\chan ) ;
4 x.set(\out , 1) ; // a destra
5 x.set(\out , 2) ; // c'e' ma e' privato
6 x.set(\out , 3) ; // idem
7 x.set(\out , 0) ; // di nuovo a sinistra
```

È perfettamente possibile avere più out (anche a tasso diverso) all'interno della stessa synthDef. Il codice seguente scrive due diversi segnali in due bus adiacenti. Quindi, variando il valore di `out`, si "traslano" entrambi.

```
1 SynthDef(\chan , { arg out = 0;
2     Out.ar(out, LFPulse.ar(5)*SinOsc.ar(400)) ;
3     Out.ar(out+1, LFPulse.ar(7)*SinOsc.ar(600)) ;
4 }).add ;
5 s.scope(4) ; // monitor per i bus
6 x = Synth(\chan ) ;
7 x.set(\out , 1) ; // verso destra
8 x.set(\out , 0) ; // da capo
```

La gestione di sistemi multicanale dunque è agevole: in una scheda audio che gestisca 8 uscite, ad esse saranno dedicati i bus $0…7$. Con `Out` è così possibile controllare il routing dei segnali. Come si vede, è estremamente semplice gestire la multicanalità in SuperCollider, ma in realtà lo è ancora di più, come si vedrà più avanti.

Se si utilizza la scheda audio integrata al calcolatore e si visualizzano più di due bus, è possibile notare che in realtà almeno il bus 2 (il terzo) non è silenzioso, ma presenta invece un segnale. Infatti, i bus i cui indici sono successivi alle uscite sono dedicati alle entrate. La situazione è schematizzata in Figura **6.4** in relazione a diverse possibili configurazioni hardware.

Nell'esempio seguente, con la riga 1 la finestra stethoscope visualizza i bus $0…3$, dedicati alle uscite e agli ingressi in caso di microfono stereo (la situazione comune alla scheda audio integrata ai computer).

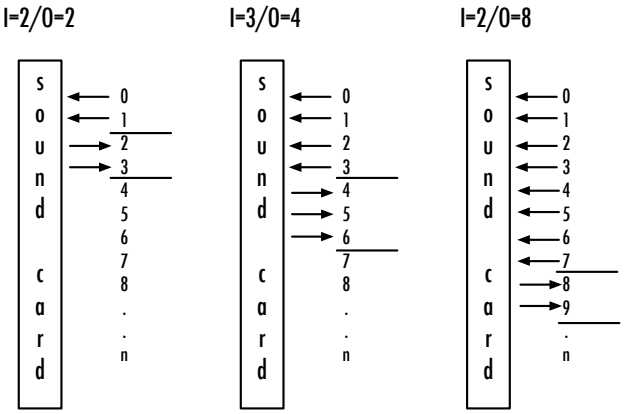


Fig. 6.4 Bus privati e pubblici in diverse configurazioni hardware.

```
1 s.scope(4,0) ;
2 {Out.ar(0, In.ar(2))}.play ;
```

La riga 2 invece:

1. introduce la UGen `In`, che è l'analogo di `Out` per la lettura da bus (invece della scrittura);
2. `In` legge dal bus 2, che è uno degli ingressi della scheda audio. Il segnale è ora disponibile a scsynth;
3. il segnale viene indirizzato attraverso `Out` al bus 0, cioè quello dedicato al canale sinistro della scheda audio;

(Attenzione al feedback microfono/altoparlanti). La situazione è schematizzata in Figura **6.5**.

Si supponga ora di connettere una scheda audio che abbia ad esempio 4 canali in uscita, come la seconda della Figura **6.4**. In questo caso l'indice 2 rappresenta un bus di uscita. Questa è una situazione che può generare una potenziale confusione: lo stesso codice infatti avrebbe una semantica diversa in funzione delle schede audio in uso. La UGen dedicata alla lettura dagli ingressi della scheda audio è `SoundIn` che fortunatamente risolve questo problema in automatico, poiché definisce come suo primo argomento il bus da cui leggere, ma a

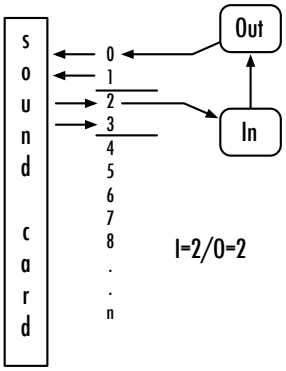


Fig. 6.5 Instradamento tra bus di ingresso e di uscita.

partire dal primo disponibile come ingresso. Nel prossimo esempio sul bus di uscita 0 (per l'altoparlante sinistro in un computer) viene scritto il segnale letto sul primo bus in ingresso (tipicamente, il microfono):

```
1 SynthDef(\in , {Out.ar(0, SoundIn.ar(0))}).add ;
2 x = Synth(\in ) ;
```

Si noti che l'indice 0 in `SoundIn` in realtà rappresenta su un computer standard (con configurazione di uscite stereo) il bus 2. Ma su una scheda con 10 uscite, il primo ingresso avrebbe l'indice 10. La specifica in termini di indice di ingresso e non di indice assoluto è molto più comoda e permette di scrivere codice indipendente rispetto al dispositivo hardware.

Quanto osservato prima vale per i bus pubblici che si interfacciano con la scheda audio. I bus privati invece permettono un routing dei segnali interno al server. Il loro uso si rivela particolarmente utile per costruire catene di elaborazione sul modello classicamente analogico, in cui ad esempio siano previsti synth per svolgere determinate operazioni, che possono essere connessi/disconnessi e che interessino più synth. Un esempio classico riguarda le linee di ritardo, ad esempio i riverberi. In forma minimale, un'unità di riverberazione è un elemento che genera in uscita un'insieme di copie del segnale in ingresso, temporalmente ritardate in forma variabile. La sua utilità sta appunto nel simulare (in vario grado) quanto avviene nella diffusione del suono in uno spazio. Una sorgente acustica puntiforme può essere pensata come un insieme di

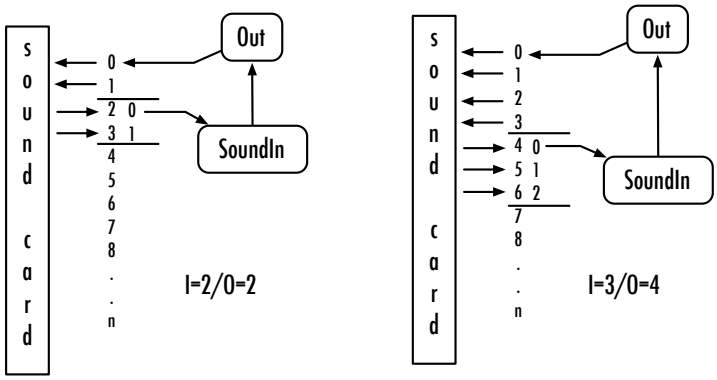


Fig. 6.6 Semantica di `SoundIn`.

"raggi" che si diffondono da un punto nello spazio, e vengono riflesse dalle superfici dello spazio secondo il modello della pallina che rimbalza sui bordi di un biliardo. Se l'ascoltatore è in un altro punto dello spazio, in un certo momento riceverà più raggi che però avranno un certo ritardo tra loro, poiché pur a velocità omogenea avranno però compiuto percorsi di lunghezza variabile. Il riverbero, che caratterizza tutti gli spazi acustici in vario modo, si simula prendendo il segnale in ingresso e ritardandolo più volte con tempi variabili. Un riverbero può essere implementato in molti modi, ma in ogni caso SuperCollider prevede alcune UGen dedicate, ad esempio `FreeVerb`, che implementa uno schema molto usato nell'ambito open source. Ad esempio, il codice seguente prevede due synth. Il primo è breve impulso di 50 millisecondi ottenuto con `Line`, un tipo di inviluppo a rampa che qui genera una linea "dritta" a valore costante 1, per poi eliminare il synth. Il secondo vi applica un riverbero.

```
1 {Pulse.ar(100)*Line.kr(1,1, 0.05, doneAction:2)}.play ;
2 {FreeVerb.ar(Pulse.ar(100)*Line.kr(1,1, 0.05, doneAction:2))}.play ;
```

Differenza: praticamente nessuna. Il motivo è che un suono riverberato dura di più di uno non riverberato, proprio perché il riverbero aggiunge copie ritardate. Ma nell'esempio, `Line` (che è un inviluppo), una volta terminata la sua durata, dealloca il synth a causa di `doneAction:2`. Dunque, il riverbero c'è ma

non ha il tempo di essere applicato come dovrebbe. L'esempio seguente sfrutta una logica più modulare e strutturata.

```
1  (
2  SynthDef(\blip , {arg out = 0;
3      Out.ar(out, Pulse.ar(100)*Line.kr(1,1, 0.05, doneAction:2))
4  }).add ;
5
6  SynthDef(\rev , {arg in, out = 0;
7      Out.ar(out, FreeVerb.ar(In.ar(in)))
8  }).add ;
9  )
10
11 Synth(\blip , [\out , 0]) ;
12
13 //vs
14 (
15 ~revBus = Bus.audio(s,1) ;
16 ~rev = Synth(\rev ) ;
17 ~rev.set(\in , ~revBus) ;
18 Synth(\blip , [\out , ~revBus]) ;
19 )
20
21 x = {Out.ar(~revBus, Dust.ar(3))}.play ;
22 x.free ;
```

Il primo blocco (1-9) definisce due synthDef. La prima è semplicemente un generatore di brevi impulsi, esattamente come visto sopra. La seconda invece è un modulo di riverberazione minimale che permette di definire ingresso e uscita del segnale. Per quanto semplice, per chiarezza la Figura **6.6** ne riporta il grafo di UGen.

Il synth di riga 11 genera direttamente l'evento impulsivo, senza riverbero. Invece, il blocco 14-19 sfrutta il modulo di riverbero. In primo luogo, un bus viene riservato sul server attraverso la classe `Bus` che invia gli opportuni messaggi al server e permette di controllare il bus in questioni. Si noti che non serve specificare l'indice, perché linguisticamente basta fare riferimento alla variabile: l'oggetto `Bus` tiene internamente traccia dell'indice (se necessario, accessibile con `~revBus.index`). Il metodo `audio` è un costruttore che restituisce un oggetto `Bus` che rappresenta in questo caso un bus allocato sul server `s` e di 1 canale (mono), come specificato dai valori dei due argomenti. Un bus può essere infatti

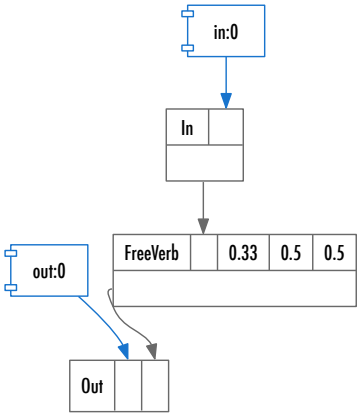


Fig. 6.7 Un modulo di riverbero minimale.

anche multicanale: in realtà, il server conosce solo bus singoli, e un bus multicanale è semplicemente un'astrazione linguistica che gestisce in forma unica (e comoda) un insieme di bus adiacenti (si ricordi la discussione precedente). Un altro esempio di potenza espressiva data dall'immissione di uno strato linguistico rispetto al protocollo OSC di controllo del server. Viene quindi creato il synth `~rev` (16) che legge in ingresso dal bus `~revBus` (17), quindi un synth `"blip"` viene instradato su `~revBus`: il segnale entra allora nel modulo di riverbero il quale scrive la versione elaborata sul bus 0, cioè sul canale sinistro. L'unità di riverbero è persistente, anche se l'evento causato dal synth `"blip"` è terminato, e il riverbero funziona come previsto. Se si valuta di nuovo la riga 18, il risultato è lo stesso, il modulo di riverbero essendo sempre attivo. Ancora, si può mandare sullo stesso bus `~revBus` una sequenza di impulsi attraverso un synth che sfrutta la UGen `Dust` (21) per apprezzare l'effetto riverbero. Anche in questo caso, la deallocazione della sorgente (22) non influisce sul modulo effetto.

L'utilizzo dei bus richiede di fare attenzione ad un punto importante dell'organizzazione del server. Su quest'ultimo tutti i nodi (gruppi e synth) sono organizzati in una sequenza che determina l'ordine di esecuzione: in altri termini, per quanto rapidamente possa avvenire il calcolo, l'ordine di esecuzione specifica quali synth vadano calcolati prima degli altri. Ad ogni periodo di campionamento $T = \frac{1}{sr}$ tutto i grafi di tutti i synth vengono comunque attraversati

dalla computazione per calcolare il valore effettivo del segnale per quell'istante. Ma, poiché la computazione è sequenziale (e per ora continua a esserlo) è anche necessario specificare quale synth debba essere calcolato prima di un altro. Finché non si utilizzano catene di synth si può vivere benissimo in SC senza sapere nulla dell'ordine di esecuzione. L'esempio seguente dimostra il punto:

```
(
SynthDef(\pulse , {arg out = 0, freq = 4;
    Out.ar(out, Pulse.ar(freq, 0.05))}).add ;
)

// 1.
~revBus = Bus.audio(s) ;
~src = Synth(\pulse , [\out , ~revBus]) ;
~rev = Synth(\rev ) ;
~rev.set(\in , ~revBus) ;

// 2.
~revBus = Bus.audio(s) ;
~rev = Synth(\rev ) ;
~src = Synth(\pulse , [\out , ~revBus]) ;
~rev.set(\in , ~revBus) ;

// 3.
~revBus = Bus.audio(s) ;
~src = Synth(\pulse , [\out , ~revBus]) ;
~rev = Synth(\rev , addAction:\addToTail ) ;
~rev.set(\in , ~revBus) ;


// 4.
~revBus = Bus.audio(s) ;
~src = Synth(\pulse , [\out , ~revBus]) ;
~rev = Synth.after(~src, \rev , addAction:\addToTail ) ;
~rev.set(\in , ~revBus) ;
```

Nel caso 1 il synth `~src` è allocato prima di quello `~rev`. E non si sente nulla. Infatti, l'azione predefinita quando si alloca un synth consiste nel metterlo in testa all'ordine di esecuzione, e cioè come primo elemento della sequenza. La cosa non è esattamente intuitiva, perché ci si aspetterebbe che ciò che viene dopo in termini di allocazione sia anche successivo. Se si vuole vedere cosa

succede sul server in termini di synth e gruppi si possono utilizzare le seguenti due espressioni:

```
1 s.queryAllNodes ;
2 s.plotTree ;
```

I due metodi definiti su `Server` permettono rispettivamente di stampare sulla post window e di costruire una GUI che mostri synth, gruppi presenti sul server, e il loro ordine. Se i synth precedenti sono ancora attivi si otterrà una stampa analoga:

```
1 localhost
2 NODE TREE Group 0
3    1 group
4       1001 rev
5       1000 dust
```

La GUI è ancora più evidente. In sostanza, c'è un gruppo (che viene creato per default) e ad esso sono associati i due synth. Come si vede l'ordine di esecuzione è `~rev`→ `~pulse`. Dunque il riverbero viene eseguito *prima* che venga calcolato il segnale impulsivo. Quindi, il synth `~rev` scrive il valore 0 su `Out`, non avendo segnali in ingresso, e soltanto dopo viene calcolato il synth `~pulse`. La situazione è rappresentata in Figura **6.8**.

Nel codice del blocco 2 dell'esempio precedente, il giusto ordine di esecuzione è ottenuto creando prima il modulo di riverberazione e dopo la sorgente impulsiva, che perciò è messa *prima* nella sequenza. Il blocco 3 ottiene un risultato analogo attraverso l'indicazione esplicita dell'argomento (finora mai considerato) `addAction` nella creazione di un synth. Quest'ultimo permette di controllare diversi aspetti dell'ordine di esecuzione: ad esempio, in 3 il riverbero viene creato dopo ma aggiunto esplicitamente in coda alla sequenza attraverso il valore `addToTail`), l'azione di default essendo invece `addToHead`. Infine, l'esempio 4 dimostra l'uso del costruttore `after` che permette di specificare come primo argomento il synth dopo il quale va posizionato il synth creato (il metodo ha un ovvio analogo simmetrico `before`).

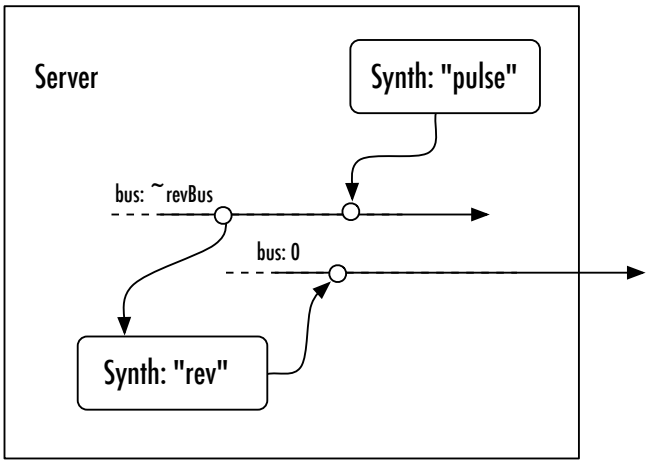


Fig. 6.8 Ordine di esecuzione erroneo.

Come si è accennato, i bus possono essere anche dedicati ai segnali di controllo: la loro utilità sta appunto nel condividere in forma agevole informazione ta più synth. Nell'esempio seguente, le pseudo-UGen `MouseX` e `MouseY` sono isolate e possono essere utilizzate per gestire due synth dello stesso tipo (`~tri1` e `~tri2`), che contengono un oscillatore per onde triangolari (tra l'altro come si vede le UGen `LF`, pur dedicate ai segnali di controllo, possono essere usate anche a tasso audio). In questo caso, un'interfaccia di controllo è allora separata dalla generazione. Come si nota, un punto di rilievo che differenzia bus audio e bus di controllo è per i bus di controllo l'ordine di esecuzione non è rilevante. Dunque, i due synth `"tri1"` e `"tri2"` leggono in ingresso dai bus su cui scrive `"mouse"`. Il blocco 18-23 dimostra invece il metodo `get` definito sui bus, che prevede una funzione come argomento, il cui argomento a sua volta rappresenta il valore del bus[2]. La funzione allora converte in midi il valore sul bus di controllo, lo arrotonda e lo stampa (20). Quindi chiama di nuovo il metodo `get` ma su `~yBus` e replica la stessa logica. Come risultato, i valori dei due bus controllati dal mouse sono stampati sullo schermo a ogni valutazione. Le ultime righe

[2] Si tratta di una funzione perché il valore del bus è sul server: e quindi un messaggio OSC deve essere inviato a quest'ultimo dal client che replicherà al client con un altro messaggio contenente il valore. È un'operazione asincrona: quando arriva la risposta del server, solo allora viene valutata la funzione.

dimostrano il metodo simmetrico `set`. Un nuovo bus di controllo `~vBus` è allocato e instradato a un nuovo synth `~tri3`, che scrive sul bus audio pubblico 1 (il canale destro, come visibile invocando `scope` su `s`, 27). Nella riga 26 invece l'azione collegata a `~xBus.get` consiste nel prendere il valore `v` di `~xBus`, raddoppiarlo (un'ottava sopra) e scriverlo nel bus `~vBus`, da cui verrà instradato in `~tri3`.

```
1  (
2  SynthDef(\tri , {arg out = 0, inFreq = 0, amp = 1;
3      Out.ar(out, LFTri.ar(In.kr(inFreq), mul:amp))}).add ;

5  SynthDef(\mouse , {arg out1, out2 ;
6      Out.kr(out1, MouseX.kr(36, 96).round.midicps);
7      Out.kr(out2, MouseY.kr(36, 96).round.midicps);
8  }).add ;
9  )

11 ~xBus = Bus.control(s) ;
12 ~yBus = Bus.control(s) ;
13 ~mouse = Synth(\mouse , [\out1 , ~xBus, \out2 , ~yBus]) ;
14 ~tri1 = Synth(\tri , [\amp , 0.5]) ;
15 ~tri2 = Synth(\tri , [\amp , 0.5]) ;
16 ~tri1.set(\inFreq , ~xBus) ;
17 ~tri2.set(\inFreq , ~yBus) ;
18 (
19 ~xBus.get{|v|
20     v.cpsmidi.round.postln;
21     ~yBus.get{|v| v.cpsmidi.round.postln};
22 } ;
23 )
24 ~vBus = Bus.control(s) ;
25 ~tri3 = Synth(\tri , [\inFreq , ~vBus, \out , 1])  ;
26 ~xBus.get{|v| ~vBus.set(v*2); }
27 s.scope ;
```

Infine, si potrebbe anche osservare che l'uso di un bus di controllo per l'immissione di segnali di controllo è senz'altro comodo, ma costringe a introdurre una UGen `In` (a tasso di controllo). In altri termini, la synthDef assume necessariamente di usare un bus di controllo e non è più generica. Un'altra possibilità di rilievo è quella di usare il metodo `map`.

```
(
SynthDef(\tri , {arg out = 0, freq = 440;
    Out.ar(out, LFTri.ar(freq))}).add ;

SynthDef(\mouse , {arg out ;
    Out.kr(out, MouseX.kr(20, 5000, 1));
}).add ;

SynthDef(\sine , {arg out, freq = 10 ;
    Out.kr(out, SinOsc.kr(freq, mul: 200, add:500));
}).add ;
)

~kBus = Bus.control(s) ;
~mouse = Synth(\mouse , [\out , ~kBus]) ;
~tri1 = Synth(\tri ) ;
~tri1.map(\freq , ~kBus) ; // dal mouse
~mouse.run(false) ; // ultimo valore mouse
~mouse.run ; // di nuovo
~sine = Synth(\sine , [\out , ~kBus]) ; // da sine, sovrascritto
~sine.run(false) ;
~tri1.set(\freq , 100) ; // fisso
~tri1.map(\freq , ~kBus) ; // dal mouse
~sine.run ; // da sine
```

Nell'esempio precedente ci sono tre synthDef, una per la sintesi audio (`"tri"`), le altre due per il controllo. Dopo aver allocato un bus di controllo (14), viene allocato un synth di controllo che intercetta il mouse (15). Quindi (dopo, ma è irrilevante qui) viene creato un synth audio (16). Infine (17), con il metodo `map` viene associato all'argomento `\freq` il bus `~kBus`. Ecco che il segnale dal mouse controlla la frequenza. Se quest'ultimo è messo in pausa (18), l'ultimo valore generato resta nel bus. Quindi il synth per il mouse riparte (19) ma un altro segnale di controllo viene instradato sullo stesso bus, e i suoi valori sovrascrivono quelli dal mouse. Quando quest'ultimo viene messo in pausa (21), allora il segnale dal mouse non è più riscritto e torna ad essere disponibile sul bus. Se si imposta un valore per l'argomento sul synth, la connessione con il bus del primo viene rimossa (22). Quindi, di nuovo riassociazione col mouse (23) e se il synth oscillatore di controllo riparte, si ha riscrittura dei valori del bus da

parte di quest'ultimo (24). Con il metodo `map` dunque non è necessario prevedere una UGen `In`, e la synthDef resta "pulita", cioè pronta ad un uso generale indipendente dall'associazione con bus.

Va notato che il comportamento dei bus è diverso tra controllo e audio nel momento in cui più segnali scrivono su uno stesso bus. Come si è visto, un segnale su un bus di controllo riscrive un eventuale segnale di controllo precedente. Invece, nel caso di bus audio, i segnali si sommano. È quanto avviene normalmente instradando più segnali su `Out.ar(0, ...)`, ed è logico per i segnali audio che si mixano su un bus. L'esempio seguente rende comunque chiaro il punto.

```
1  (
2  SynthDef(\tri , {arg out = 0;
3      Out.ar(out, LFTri.ar)}).add ;

5  SynthDef(\sin , {arg out ;
6      Out.ar(out, SinOsc.ar(1000));
7  }).add ;

9  SynthDef(\mix , {arg out = 0, in ;
10     Out.ar(out, In.ar(in));
11 }).add ;
12 )

14 b = Bus.audio ;
15 x = Synth(\tri , [\out , b]) ;
16 y = Synth(\sin , [\out , b]) ;
17 b.scope ; // monitoraggio
18 z = Synth.tail(nil, \mix , [\in , b]) ;
```

Qui una synthDef è dedicata al mix (9-11), e semplicemente inoltra verso un'uscita pubblica il contenuto di un bus privato in ingresso. Il synth mixer `z` è allocato per ultimo ma attraverso il metodo `tail` che lo pone al fondo del gruppo di default. I due synth `x` e `y` scrivono sul bus da cui legge `z` e come si

sente i segnali sono sommati sul bus b. È possibile monitorare un bus (sia audio che di controllo) attraverso il metodo scope (17).

6.5 Costruzione procedurale delle synthDef

Che cos'è una synthDef? È un progetto di un synth, lo schema elettrico di un sintetizzatore. Una synthDef viene scritta nel linguaggio SuperCollider (su sclang), e quando viene spedita al server scsynth, viene compilata in un formato (binario) che quest'ultimo può comprendere. Infatti, scsynth non sa nulla del linguaggio SuperCollider. Se ci fosse un interprete opportuno, la si potrebbe scrivere in italiano: sarebbe l'interprete che dovrebbe occuparsi di convertirla nel formato opportuno. Se dunque una synthDef è una descrizione linguistica, allora può sfruttare la potenza espressiva del linguaggio per specificare la struttura dello strumento in maniera molto compatta. Si prenda in considerazione la synthDef seguente:

```
SynthDef(\chan , { arg out = 0 ;
    8.do({|i|
        Out.ar(out+i, LFPulse.ar(i+5)*SinOsc.ar(i+1*200))
    }) ;
}).add ;
```

In essa, il grafo delle UGen è descritto facendo riferimento ad un ciclo. Per 8 volte si crea un segnale, dato da una sinusoide "pulsata", i cui parametri dipendono dal contatore, e che viene instradato su un bus che dipende dal contatore stesso e da out (secondo il meccanismo dei bus adiacenti già sperimentato). Il codice seguente permette di ascoltare i primi due segnali (in caso di scheda audio stereo) e di vedere i primi 8 bus audio.

```
1 s.scope(8) ; // monitor per i primi 8 bus
2 x = Synth(\chan ) ;
3 x.set(\out , 1) ; // spostati da destra
4 x.set(\out , 0) ; // da capo
```

La descrizione linguistica è molto compatta e specifica un insieme di relazioni che non appaiono esplicitamente del grafo delle UGen. Analogamente, si consideri la semplice parametrizzazione della synthDef precedente nell'esempio che segue:

```
1 SynthDef(\chan , { arg out = 0, freq = 200, kFreq = 5 ;
2     8.do({|i|
3         Out.ar(out+i, LFPulse.ar(i+kFreq)*SinOsc.ar(i+1*freq))
4     }) ;
5 }).add ;
```

I grafi delle due synthDef sono riportati in Figura **6.9**. Si ricordi che le figure di questo tipo sono ottenute attraverso l'analisi automatica della struttura della synthDef, dunque sono esattamente la struttura dati che risulta dalla descrizione. La potenza espressiva dell'uso di un linguaggio è autoevidente. Una scrittura compatta permette di descrivere un grafo molto esteso attraverso la specificazione delle relazioni tra le sue UGen.

Se si ascolta su un normale dispositivo stereo (quale quello di un pc) il risultato della valutazione del codice precedente, saranno udibili soltanto le due prime sinusoidi, instradate sui bus 0 e 1. A tal proposito SC mette a disposizione dell'utente una UGen specializzata nella riduzione multicanale, ovvero nel missaggio di più segnali in un unico segnale: `Mix`. Quest'ultima semplicemente prevede come argomento un array di segnali che restituisce mixati in un unico segnale. La versione seguente modifica l'esempio in discussione fornendo ad `Out` un unico segnale, grazie all'intervento di `Mix` (riga 14). Si noti che quest'ultima UGen semplicemente somma l'output dei segnali che compongono l'array che riceve come argomento. In realtà, dal punto di vista tecnico, `Mix` non è una UGen, e infatti il metodo che si usa non è `ar` ma semplicemente `new`[3]. L'esempio

che segue riscrive la synthDef precedente utilizzando un array in cui vengono stoccati i segnali, per poi mixarli in `mix` (e scalare l'ampiezza complessiva, data dalla somma lineare dei segnali, per il numero di segnali).

```
SynthDef(\chan , { arg out = 0, freq = 200, kFreq = 5 ;
    var n = 8;
    var arr = Array.newClear(n);
    n.do({|i|
        arr[i] = LFPulse.ar(i+kFreq)*SinOsc.ar(i+1*freq)
    }) ;

    Out.ar(out, Mix(arr)/n) ;
}).add ;

s.scope ; // 1 canale
Synth(\chan ) ;
```

La descrizione può essere resa ancora più snella nell'esempio seguente, utilizzando il metodo `fill` di `Array`:

```
SynthDef(\chan , { arg out = 0, freq = 200, kFreq = 5 ;
    var n = 8;
    var arr = Array.fill(n, {|i| LFPulse.ar(i+kFreq)*SinOsc.ar(i+1*freq)
    }) ;
    Out.ar(out, Mix(arr)/n) ;
}).add ;
```

Infine, `Mix` prevede a sua volta un metodo costruttore `fill` che abbrevia ulteriormente la scrittura.

[3] Quest fatto può essere in effetti una fonte di confusione. In ogni caso, è anche possibile utilizzare i metodi `ar` e `kr` che sono sostanzialmente sinonimi di `new`.

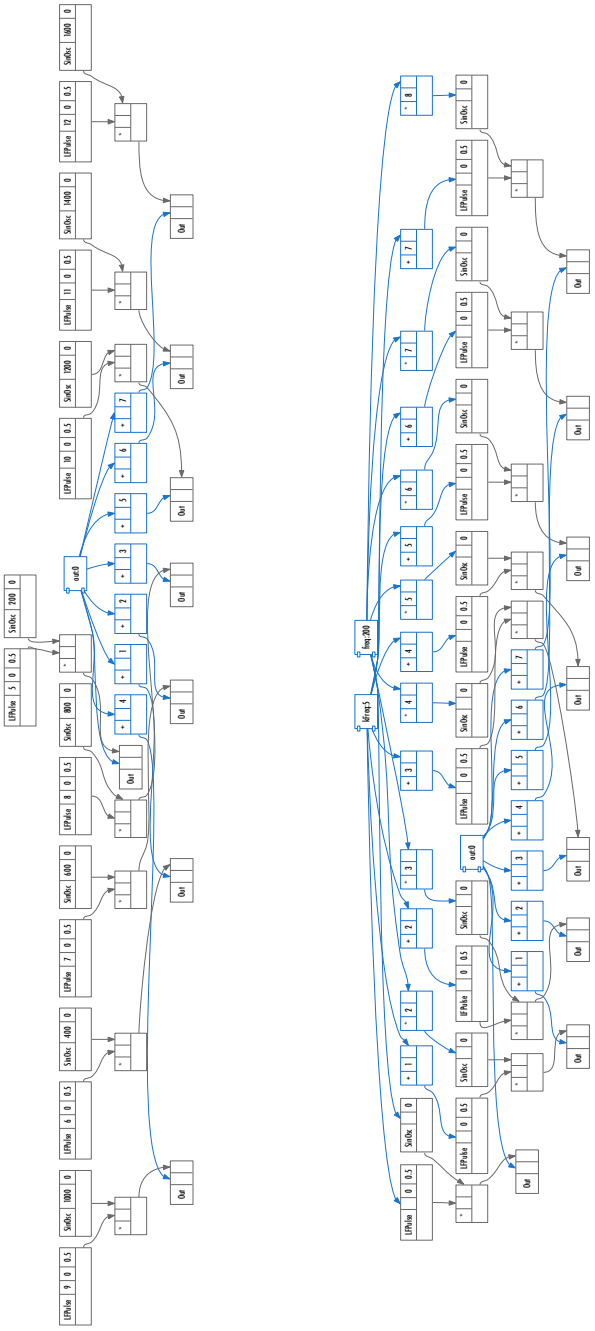


Fig. 6.9 Due grafi delle UGen in un caso di ciclo.

```
SynthDef(\chan , { arg out = 0, freq = 200, kFreq = 5 ;
    var n = 8;
    var mix = Mix.fill(n, {|i| LFPulse.ar(i+kFreq)*SinOsc.ar(i+1*freq)}) ;
    Out.ar(out, mix/n) ;
}).add ;
```

Il grafo delle UGen della synthDef precedente è riportato (parzialmente) in Figura **6.10**[4].

[4] `Mix` non c'è, in realtà internamente (e solo internamente) si usano insiemi di UGen `Sum4`, ottimizzate per mixare blocchi di 4 segnali.

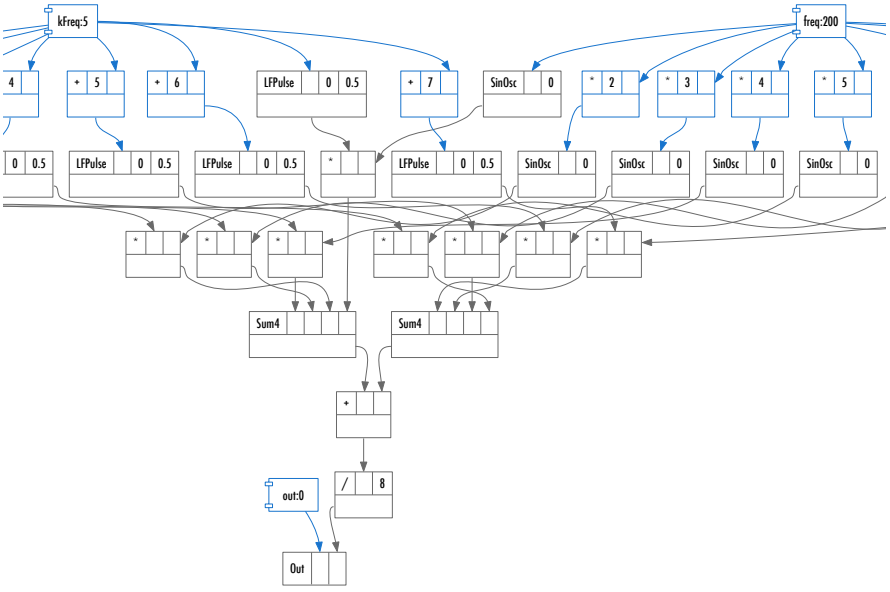


Fig. 6.10 Grafo della synthDef "chan".

6.6 Espansione multicanale

Le considerazioni precedenti su espressività del linguaggio e definizione dei bus possono essere concluse con una discussione del meccanismo della "espansione multicanale". Si consideri il prossimo esempio:

```
SynthDef(\pan , {arg out;
    Out.ar(out,
        Pan2.ar(SinOsc.ar, MouseX.kr(-1,1))
    )
}).add ;

s.scope ;
Synth(\pan ) ;
```

Come si vede nella finestra stethoscope, l'ascissa del mouse controlla la posizione sui due canali, sinistro e destro, del segnale sinusoidale. Gli argomenti della UGen `Pan2` –responsabile dell'effetto– sono `in, position, level`: il primo è il segnale che deve essere posizionato sul fronte stereo; il secondo è la posizione, modulabile, nell'escursione $[-1, 1]$ (da sinistra a destra); il terzo semplicemente un moltiplicatore per l'ampiezza. L'effetto di `Pan2` è dunque quello di posizionare un segnale sul fronte stereo. In realtà, ciò che avviene è che la UGen riceve un segnale mono in ingresso e fornisce in output una coppia di segnali risultanti da una varia distribuzione dell'ampiezza del segnale. Un segnale stereo, come quello in uscita da `Pan2`, è in effetti una coppia di segnali che devono essere eseguiti in parallelo, tipicamente su due (gruppi di) altoparlanti. Gli esempi precedenti hanno dimostrato come i bus pubblici possano (e debbano) essere utilizzati per inviare segnali alla scheda audio, dove nella convenzione stereo 0 rappresenta il canale sinistro e 1 quello destro. Tuttavia, nella synthDef non è stato necessario specificare nella sintassi di Out una coppia di bus, bensì soltanto l'indice `0`. Nella fattispecie, SC distribuisce automaticamente i due segnali sul necessario numero di bus contigui (in questo caso 0 e 1). Quest'operazione prende il nome di *multichannel expansion*. La Figura **6.11** riporta il grafo delle UGen. La UGen `Out` riceve come secondo `numChannels` un array di segnali (si rilegga l'help file). Finora, la UGen `Out` è stata utilizzata inviando un singolo segnale, ma con `Pan2` l'argomento è effettivamente un array, come indicato dalle due posizioni occupate.

Dal grafo risulta chiaro che `Pan2` genera in uscita due segnali racchiusi in un array, come è altresì chiaro dal prossimo esempio, in cui l'accesso ai singoli segnali avviene tramite sintassi tipica degli array:

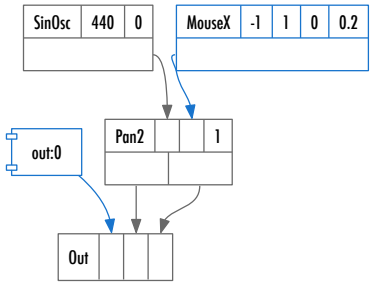


Fig. 6.11 Grafo della synthDef "pan2".

```
s.scope ;
{SinOsc.ar([60.midicps, 69.midicps])}.play ;
{SinOsc.ar([60.midicps, 69.midicps][0])}.play ;
{SinOsc.ar([60.midicps, 69.midicps][1])}.play ;
```

Nell'esempio seguente si hanno tre versioni di una syntDef.

```
// 1. routing esplicito
SynthDef( "multi1", {
    Out.ar(0, SinOsc.ar(300)) ;
    Out.ar(1, SinOsc.ar(650)) ;
}).add ;

// 2. channelsArray in Out
SynthDef( "multi2", {
    Out.ar(0,
        [SinOsc.ar(300), SinOsc.ar(650)]
    )
}).add ;

// 3. array in argomento
SynthDef( "multi1", {
    Out.ar(0,  SinOsc.ar([300, 650]))
}).add ;
```

Nel primo caso si esplicita l'instradamento del segnale sui due bus 0 e 1. Nel secondo caso, l'argomento `numChannels` di `Out` riceve un array di due segnali, e dunque ne invia uno a 0 e il successivo a 1. Il terzo esempio mostra un uso che tecnicamente è identico ma che è ancora più espressivo perché non si rivolge direttamente al controllo delle uscite. Qui l'unico array presente è nella gestione della frequenza di `SinOsc`. Dunque, in generale, SC opera una espansione multicanale tutte le volte che un argomento di una UGen è costituito da un array. L'espansione multicanale avviene soltanto ed esclusivamente quando agli argomenti di una UGen viene passato un oggetto di tipo `Array`, non per le sue superclassi né per le sue sottoclassi. Le tre synthDef sono "sinonimi" nel senso che danno origine esattamente alla stessa struttura, riportata per completezza in Figura **6.12**.

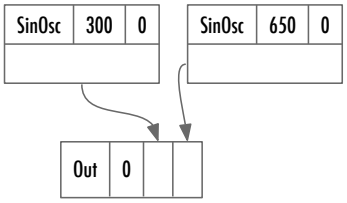


Fig. 6.12 Grafo delle synthDef con espansione multicanale.

Il prossimo esempio dimostra ulteriormente l'interesse dell'espansione multicanale per il controllo di processi complessi.

```
1  (
2  SynthDef(\sum , { arg out = 2 ;
3      Out.ar(out, Array.fill(16, {
4          SinOsc.ar(
5              freq:Rand(48, 84).midicps,
6              mul: LFNoise1.kr(Rand(2, 0.1))
7          )
8      })
9      )
10 }).add ;


13 SynthDef(\mix , {arg in;
14     Out.ar(0, Mix(In.ar(in, 16)/16))
15 }).add ;
16 )

18 b = Bus.audio(s, 16) ; b.scope ;
19 x = Synth(\sum , [\out , b]) ;
20 Synth.after(x, \mix , [\in , b]) ;
```

La synthDef "sum" invia in uscita un array di 16 canali. In caso di sistema audio a 16 canali, è così possibile gestire direttamente un segnale per ognuno. Ogni segnale è una sinusoide la cui frequenza è scelta casualmente tra le altezze temperate di 3 ottave a partire dal do in chiave di basso (48 in notazione midi). L'ampiezza è gestita da una rampa (LFNoise1) che si muove nell'escursione $[0, 1]$ ad una frequenza casuale tra 2 e $0,1$ (periodo pari a 10 secondi). La UGen in uso è Rand che genera un numero casuale solo in fase di creazione del synth. La synthDef "mix" è funzionalmente un mixer per un bus di 16 canali. Si noti che in ingresso In può ricevere un bus multicanale. Quindi, viene riservato un bus a 16 canali, cioè 16 bus contigui, che viene visualizzato (18, si noti che scope è definito anche direttamente per gli oggetti della classe Bus). Il synth x quindi genera 16 segnali sul bus multicanale b (come visibile nella GUI), e questo bus viene instradato al synth mixer, che lo mixa in mono (come udibile).

Nell'esempio seguente viene generato un array di 16 segnali:

```
1  SynthDef("multi16mixPan", { arg bFreq = 100 ; // base freq
2              var left, right ;
3              var sigArr = Array.fill(16,
4                  { arg ind ;
5                  var index = ind+1 ;
6                      Pan2.ar(
7                          in:    SinOsc.ar(
8                              freq: bFreq*index+(LFNoise1.kr(
9                                      freq: index,
10                                     mul: 0.5,
11                                     add: 0.5)*bFreq*index*0.02) ,
12                             mul: 1/16 ;
13                             ),
14                         pos: LFNoise1.kr(freq: index)
15                         )
16                  }) ;
17         sigArr.postln ;
18         sigArr = sigArr.flop.postln ;
19         left = Mix.new(sigArr[0]) ;
20         right = Mix.new(sigArr[1]) ;
21         Out.ar(0, [left, right])
22         }
23 ).add ;

25 a = Synth(\multi16mixPan ) ;

27 c = Bus.control ;
28 a.map(\bFreq , c) ;

30 x = {Out.kr(c, LFPulse.kr(
31     MouseX.kr(1,20), mul:MouseY.kr(1, 100), add:250).poll)}.play ;
32 x.free ;
33 x = {Out.kr(c, Saw.kr(MouseX.kr(0.1, 20, 1), 50, 50))}.play ;
```

Al lettore come esercizio i dettagli del controllo della sinusoide (7-13). Il punto di rilievo è che l'array `sigArr` è composto di 16 segnali stereo, poiché ognuno è generato attraverso `Pan2`. Per ogni sinusoide, `pos` varia casualmente in $[-1, 1]$ con una frequenza pari a `index` (riga 14). Dunque, più elevato è l'indice, più elevata è la frequenza dell'oscillatore, più rapidamente varia la distribuzione spaziale. Si noti che è stata utilizzata la UGen `LFNoise1`, che interpola tra valori successivi, per evitare salti di posizione, simulando invece un progressivo movimento tra i valori assunti dalla distribuzione.

Se si valuta la synthDef sulla post window si ottiene una situazione simile alla seguente:

```
[ [ an OutputProxy, an OutputProxy ], [ an OutputProxy, an OutputProxy ],
    [ an OutputProxy, an OutputProxy ], [ an OutputProxy, an OutputProxy ],
    [ an OutputProxy, an OutputProxy ], [ an OutputProxy, an OutputProxy ],
    [ an OutputProxy, an OutputProxy ], [ an OutputProxy, an OutputProxy ],
    ...etc...

[ [ an OutputProxy, an OutputProxy, an OutputProxy, an OutputProxy,
        an OutputProxy, an OutputProxy, an OutputProxy, an OutputProxy,
        an OutputProxy, an OutputProxy, an OutputProxy, an OutputProxy,
        an OutputProxy, an OutputProxy, an OutputProxy, an OutputProxy ],
    [ an OutputProxy, ...etc...
```

La denominazione `OutputProxy` è il modo con cui SC chiama l'output di alcune UGen che hanno uscite multiple (come `Pan2`). A parte questo aspetto, qui irrilevante, è interessante osservare che ogni elemento dell'array è a sua volta un array composto di due elementi, uno per canale. La prima stampa è data dalla riga 17, e rappresenta 16 array di 2 segnali (è lo stereo). Attraverso `flop` si ottengono allora 2 array da 16 segnali (secondo blocco postato). la disposizione degli elementi in `sigArr` è perciò

```
[[sig0sx, sig0dx], [sig1sx, sig1dx], [sig3sx, sig3dx]
                    ..., [sig15sx, sig15sx]]
```

Come già visto, chiamando il metodo `flop` si ottiene un nuovo array con questa struttura

```
[[sig0sx, sig1sx, sig2, sx, sig3sx, ..., sig15sx],
 [sig0dx, sig1dx, sig2, dx, sig3dx, ..., sig15dx]]
```

che è composta di due array, uno per canale, contenenti in contributi di ognuno dei segnali sinusoidali a quel canale. È allora possibile mixare ognuno dei due array ottenendo due singoli segnali, per il canale sinistro e per quello destro (righe 19 e 20), assegnati alle variabili `left, right`. Inviando a Out l'array `[left, right]` si produce di nuovo un'espansione multicanale, per cui il

primo (`left`) verrà inviato, attraverso il bus `0`, all'altoparlante sinistro, il secondo (`right`), attraverso il bus `1`, all'altoparlante destro. La riga 25 crea un synth, mentre quelle successive sperimentano alcuni segnali di controllo assegnati su `bFreq`. Anche in questo, al lettore come esercizio la comprensione analitica del relativo grafo delle UGen.

Infine, la Figura **6.13** riproduce il grafo delle UGen della synthDef `"multi16mixPanGraph"`.

6.7 Conclusioni

Sotto l'ombrello del controllo sono stati rubricati molti aspetti diversi. Tuttavia, alla fine di questo capitolo, molti degli elementi del funzionamento di SC sono stati affrontati. Il prossimo capitolo sarà in qualche modo dedicato di nuovo al controllo, nel senso però dell'organizzazione temporale degli eventi.

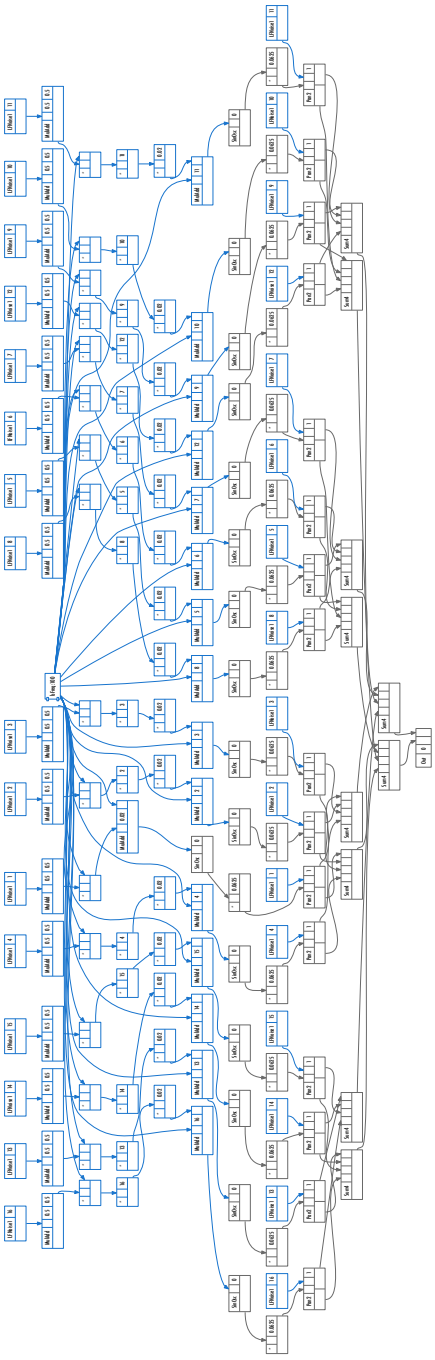


Fig. 6.13 Grafo delle synthDef "multi16mixPan".

7 Suono organizzato: scheduling

Nel corso dei capitoli precedenti, sono stati introdotti molti aspetti relativi a SuperCollider. A partire dalle UGen messe a disposizione da SC e dal loro concatenamento (patching) è già possibile sperimentare con la generazione del suono e con il suo controllo interattivo in tempo reale. Prima di introdurre rapidamente l'implementazione in SC delle tecniche più conosciute di sintesi del segnale audio, è allora opportuno discutere alcuni aspetti relativi alla gestione del tempo in SuperCollider. Quest'ultimo infatti eccelle rispetto ad altri ambienti di programmazione proprio perché riesce ad integrare senza soluzione di continuità sintesi del segnale e composizione algoritmica. Il controllo dell'informazione e dei suoi processi, attraverso il linguaggio SC, e la sintesi del segnale, attraverso la gestione delle UGen nelle synthDef, sono due momenti fondamentali che devono essere integrati: mutuando un'espressione di Edgar Varèse, si tratta di arrivare al "suono organizzato". La locuzione è interessante perché unisce il lavoro sulla materia sonora alla sua organizzazione temporale: ne consegue una definizione in fondo molto generale di musica e composizione, come un insieme di eventi sonori. In termini molto generali, lo scheduling è appunto l'assegnazione di risorse per la realizzazione di un evento in un certo momento[1]. Come usuale, in SC esistono potenzialmente modi diversi di realizzare lo scheduling.

7.1 A lato server, 1: attraverso le UGen

[1] Ad esempio, `http://en.wikipedia.org/wiki/Scheduling`. Un termine italiano possibile è "programmazione", nel senso di gestione nel tempo delle risorse, ma il termine italiano è intuibilmente molto meno specifico.

Una opzione che deriva dai sintetizzatori analogici è quella di gestire il sequencing (la messa in sequenza) degli eventi attraverso segnali. L'esempio seguente, pur nella sua semplicità, dimostra un aspetto che si è già discusso: un inviluppo applicato ad un segnale continuo lo può trasformare in un insieme di eventi discreti.

```
1  (
2  SynthDef.new("pulseEvent", {|seqFreq = 4, sawFreq = 0.125, sinFreq = 0.14|
3      Out.ar(0,
4          Pulse.kr(seqFreq, width:0.1, mul:0.5, add:0.5)
5          *
6          Formant.ar(fundfreq:(
7              LFSaw.kr(sawFreq, mul:6, add:60)
8              +
9              SinOsc.ar(sinFreq, mul:3)
10         ).round.midicps),
11     )
12 }).add ;
13 )

15 x = Synth.new("pulseEvent", [\seqFreq , 4]) ;
16 x.set(\seqFreq , 5) ;
17 x.set(\sawFreq , 1/4) ;
18 x.set(\sinFreq , 1/2) ;
```

In questo caso, l'inviluppo è dato da un segnale a forma di onda quadra unipolare ($[0, 1]$), con un duty cycle del 10% (`0.1`). Il segnale "finestra" un segnale generato dalla UGen `Formant`, che produce un segnale armonico complesso a partire da una frequenza fondamentale e da un altro insieme di frequenze specificabili (sul modello della voce umana). La frequenza di quest'ultimo è data da una curva di controllo complessa che risulta dall'interazione di una onda a dente di sega (dunque una rampa che si ripete) e da una sinusoide (che la incrementa/decrementa). Attraverso `round` la curva, che intende rappresentare altezze espresse in notazione MIDI, è "scalettata" e convertita in frequenza. Si noti che la sequenza d'altezza dipende dalle interrelazioni tra le tre frequenze `seqFreq, sawFreq, sinFreq`, come esperibile alle righe 16-18.

Il prossimo esempio (che il lettore è invitato a implementare anche in altro modo) sfrutta la UGen `InRange`. Quest'ultima restituisce un segnale il cui valore dipende dall'inclusione o meno del valore del campione in entrata nell'intervallo specificato (qui $[0, 1]$). Se il valore è incluso, `InRange` restituisce il valore

stesso, altrimenti restituisce 0. Il metod `sign` restituisce 1 se il segnale è positivo, 0 se è pari a 0. Dunque, il segnale in uscita sarà composto da 1 o 0 in funzione rispettivamente del superamento o meno della soglia di 0.35. Un simile segnale d'inviluppo "buca" il segnale audio tutte le volte che vale 0, di fatto trasformando un segnale continuo in un insieme di eventi.

```
(
SynthDef.new("pulseEventThresh", {
    |seqFreq = 4,     sawFreq = 0.125,     sinFreq = 0.14|
    Out.ar(0,
        InRange.kr(LFNoise0.kr(15), 0, 1).sign
        *
        Formant.ar(fundfreq:(
            LFSaw.kr(sawFreq, mul:6, add:60)
            +
            SinOsc.ar(sinFreq, mul:3)
        ).round.midicps),
    )
}).add ;
)

x = Synth.new("pulseEventThresh") ;
```

Il codice seguente dimostra cosa succede in 3 secondi.

```
{InRange.ar(LFNoise2.ar(15), 0, 1).sign}.plot(3)
```

Se si valuta più volte la riga 15, si ha (ormai è chiaro) istanziazione multipla di più synth. Questo è un punto generale in relazione allo scheduling: il parallelismo (voci, strati, come li voglia chiarire) è gestito da SC semplicemente istanziando più synth, esattamente come quando si valuta il codice interattivamente. Ad esempio, il codice che segue sfrutta una variante della synthDef precedente per istanziare 100 synth in parallello:

```
(
SynthDef.new("schedEnv", {
    Out.ar(0,
    Pan2.ar(        // panner
        InRange.ar(LFNoise2.ar(10), 0.35, 1).sign
        *
         Formant.ar(LFNoise0.kr(1, mul: 60, add: 30).midicps, mul:0.15),
    LFNoise1.kr(3, mul:1) ) // random panning
    )
 }).send(s) ;
)

(
100.do({
    Synth.new("schedEnv") ;
})
)
```

Come si vede, una (1) espressione linguistica (`100.do`) produce 100 (cento) synth. E ne può produrre di più, risorse computazionali permettendo. Dal punto di vista audio, poiché il moltiplicatore balza da 0 a 1 e viceversa, ne risulta un forte rumore impulsivo di fondo (che alla lontana ricorda una sorta di effetto da vecchio vinile).

Il prossimo esempio –per quanto semplice– è un esempio di sound design procedurale in SC. Usualmente, il sound designer (per il cinema e gli audiovisivi, ma lo scenario dei videogame impone prospettive più innovative) lavora in tempo differito attraverso il montaggio di materiali audio in una DAW (Digital Audio Workstation). Questo tipo di lavoro nasce in relazioni ai "fixed media" che producono un oggetto/testo chiuso (basta pensare al film). Ma altre situazioni produttive impongono invece condizioni diverse: ad esempio, un'installazione interattiva richieda la produzione di audio a richiesta, controllabile interattivamente e senza una durata predefinita. Il codice seguente si occupa di modellare un suono specifico (e certo non complesso), ma importante nel paesaggio sonoro della modernità: il segnale del telefono, ed in particolare in Italia. In assenza di una specifica delle sue proprietà (che pure ci dovrà essere, essendo uno standard), si può procedere empiricamente. Se si analizzando una registrazione, si osserva come, intanto, il segnale sia una sinusoide di frequenza pari a 422 Hz (tra la e la ♭ calante, si valuti `68.midicps`= 415.30469757995 e `69.midicps`= 440). Il segnale dell'occupato è il più semplice: si tratta di una

sequenza regolare di impulsi e silenzi a 200 ms di distanza. Il sequencing può allora essere ottenuto con un'onda quadra unipolare che fa come al solito da "gate": $[0, 1]$. Poiché ogni semiperiodo deve essere pari a 200 ms, allora il periodo sarà $200 \times 2 = 400$ ms. L'implementazione usa allora semplicemente `LFPulse` che inviluppa `SinOsc` (7). Il caso del segnale di linea libera è temporalmente un po' più complesso. Se si analizza il segnale si può notare un pattern più musicale, che è stato notato attraverso un inviluppo e assumendo che l'unità temporale di base valga 1. Empiricamente (a orecchio), tuttavia i segmenti temporali non sembravano esattamente proporzionali, e ne sono risultati alcuni aggiustamenti (si vedano le durate 2.5 e 1.5) (15): se si considera il fattore `timescale` allora l'inviluppo dura 1.2 secondi (cioè: `env.sum * 1/5`). Il periodo di ripetizione è invece 2 secondi (`cycleDur`), e determina la frequenza di triggering (`1/cycleDur`).

```
// Segnale telefonico (italiano)

(
// occupato: pulsazioni a intervalli di 200 msec
{
    var freq = 422, dur = 0.2;
    LFPulse.kr(1/( dur*2))*SinOsc.ar(freq)
}.play ;
)

(
// libero
{
    var freq = 422, cycleDur = 2 ;
    var env = Env([0,1,1,0,0,1,1,0, 0], [0,1,0, 1, 0, 2.5, 0, 1.5]);
    EnvGen.kr(env, 1,
        gate: Impulse.kr(1/cycleDur),
        timeScale:1/5, doneAction:0)
    * SinOsc.ar(freq)
}.play
)
```

Un'applicazione possibile, oltre all'esercizio di modellazione in quanto tale che aiuta capire l'organizzazione dei materiali sonori, è quella di una installazione interattiva in cui una certa condizione determini il libero o l'occupato.

La UGen `Select.kr(which, array)` implementa a livello server una funzione tipica del sequencing: a tasso di controllo, ogni volta che calcola un nuovo valore selezione un elemento `which` dell'array `array`.

```
(
SynthDef(\select , {
    var array ;
    array = Array.fill(64, {|i| (i%4) + (i%7) + (i%11) + 60}) ;
    array = array.add(90).postln.midicps;
    Out.ar(0,
        Saw.ar(
            Select.kr(
                LFSaw.kr(1/6).linlin(-1.0,1.0, 0, array.size),
                array
            ),
            0.2
    ));
}).add;
)

Synth(\select ) ;
```

Il codice precedente presenta due aspetti di rilievo. In primo luogo, l'array `array` è costruito utilizzando un modulo variabile sul contatore `i`. Ne consegue la sequenza di altezze in Figura **7.1**. L'ultimo elemento (90) è stato aggiunto per fornire un indice acustico della fine del ciclo.

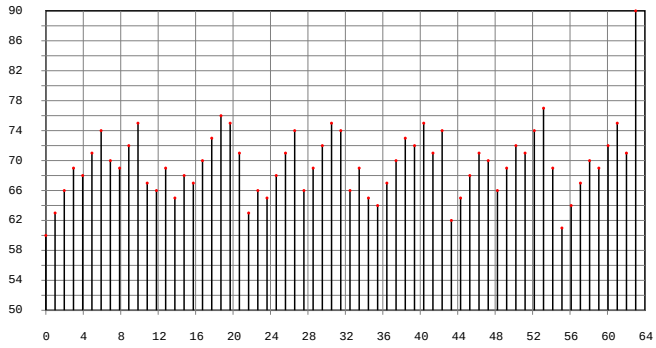


Fig. 7.1 Sequenza costruita con più applicazioni dell'operatore modulo.

`Select` controlla la frequenza di `Saw`. In uscita, cicla perciò su `array` in funzione di un segnale che fornisce gli indici. Qui è `LFSaw`, che è infatti una rampa (un segnale che cresce linearmente tra un minimo e un massimo): `LFSaw` viene scalato attraverso l'operatore `linlin` in modo da percorrere tutti gli indici dell'array (`0, array.size`). La sua periodicità indica in quanto tempo la rampa occorre, e cioè 6 secondi: la frequenza è infatti $\frac{1}{6}$, espressa in modo da rendere subito evidente il periodo $T = 6$[2]. L'utilizzo di segnali simili (*ramp*, "a rampa") è tipico nei synth analogici[3], in cui spesso lo scheduling è gestito attraverso segnali continui opportunamente generati.

7.2 A lato server, 2: le UGen Demand

Una UGen come `Sequencer` introduce nella generazione del segnale (in una synthDef) aspetti di controllo tipicamente di livello superiore. In questa direzione, un approccio allo scheduling molto peculiare in SC è implementato nelle UGen di tipo Demand ("a richiesta"), o "demand rate", secondo una locuzione –come si vedrà– significativa. La UGen `Demand.ar(trig, reset, [..ugens..])` opera in relazione ad un trigger (`trig`). Ogni qualvolta un segnale di trigger è ricevuto[4], la UGen richiede un valore ad ognuna delle altre UGen incluse nell'array `[..ugens..]`. Queste UGen devono essere di tipo speciale, ovvero di tipo `Demand`: sono infatti delle UGen che generano un valore (ed uno solo) a richiesta. Nell'esempio seguente, il segnale di trigger per `Demand` è generato dalla UGen `Impulse`. Ogni impulso prevede una transizione tra minimo (0) e massimo (qui 1), e dunque un trigger. Inutile discutere della sintesi (riga 7), salvo notare che la frequenza di `Pulse` è gestita da `freq`. La riga 6 assegna un valore a `freq` attraverso una UGen `Demand`. Ogni qualvolta un trigger è ricevuto, `Demand` chiede il prossimo valore nell'array demand `a`. Quest'ultimo è riempito da `Dseq` (riga

[2] Un'altra UGen che permette di generare eventi discreti in maniera analoga è `Stepper`.

[3] L'implementazione più usuale prende il nome di "phasor": si veda la UGen omonima in SC e il prossimo capitolo.

[4] Si ricordi che si ha un evento di triggering ogni qualvolta si verifica una transizione da 0 a un valore positivo.

4), che produce una sequenza di valori quale quella prevista nell'array fornito come primo argomento (`[1,3,2,7,8]`), ripetuta –in questo caso– per 3 volte. Ascoltando il risultato, ci si accorge come la sequenza `a` sia appunto costituita dalla ripetizione per 3 volte di un segmento: quando il segmento è concluso, `Demand` restituisce l'ultimo valore della sequenza. A latere, a livello si sintesi si noti come il segnale audio implementi una sorta di effetto chorus, poiché risulta dalla somma di 10 onde quadre ma con un aggiunta casuale di valori nell'intervallo $[0, 5]$ Hz, una sorta di intonazione variabile (ma non tale da introdurre dissonanza) tipica di più strumenti che suonano all'unisono.

```
(
{
    var a, freq, trig;
    a = Dseq([1, 3, 2, 7, 8]+60, 3);
    trig = Impulse.kr(4);
    freq = Demand.kr(trig, 0, a.midicps);
    Mix.fill(10, { Pulse.ar(freq+5.rand)}) * 0.1

}.play;
)
```

Come si vede, in sostanza le UGen di tipo Demand sono generatori di valori a richiesta, e si differenziano tra loro per le sequenze che possono produrre. In realtà, queste sequenze sono codificate in una forma interessante, sono cioè più propriamente dei "pattern". Non è la sequenza in quanto tale a essere descritta, ma una regola per la sua produzione. Infatti, nell'esempio precedente la sequenza finale generata sarebbe:

```
[ 61, 63, 62, 67, 68 , 61, 63, 62, 67, 68, 61, 63, 62, 67, 68 ]
```

Un pattern è perciò una forma descritta proceduralmente. Ad esempio, come si è visto `Dseq` genera sequenze di valori costruite iterando n volte un array, dove n può essere pari a infinite volte, un valore che SC rappresenta attraverso la parola riservata `inf`. Una UGen come `Drand` descrive un pattern diverso. Essa riceve due argomenti: il primo è un array di valori, il secondo (`repeats`)

un numero che rappresenta il numero di valori pescati a caso nell'array fornito. Nell'esempio, l'array è lo stesso del caso precedente, mentre `freq` è una sequenza di durata `inf` di valori prelevati pseudo-causalmente sullo stesso array `a`. Inoltre la frequenza di triggering è stata incrementata da 4 a 10.

```
(
{
    var a, freq, trig;
    a = Drand([1, 3, 2, 7, 8]+60, inf);
    trig = Impulse.kr(10);
    freq = Demand.kr(trig, 0, a.midicps);
    Mix.fill(10, { Pulse.ar(freq+5.rand)}) * 0.1

}.play;
)
```

Ne consegue una sorta di improvvisazione su un insieme finito di altezze (un modo, si potrebbe dire). La potenza espressiva delle UGen di tipo Demand sta nella possibilità di innesto ricorsivo. L'esempio seguente è del tutto analogo al primo caso qui discusso, senonché uno degli elementi dell'array su cui opera `Dseq` è `Drand`.

```
(
x = {|trigFr = 1|
    var freq, trig, reset, seq;
    trig = Impulse.kr(trigFr);
    seq = Dseq(
        [42, 45, 49, 50,
        Dxrand([78, 81, 85, 86], LFNoise0.kr(4).unipolar*4)
        ], inf).midicps;
    freq = Demand.kr(trig, 0, seq);
    Pulse.ar(freq + [0,0.7] + LFPulse.kr(trigFr, 0, 0.1,freq*2))* 0.5;
}.play;
)

x.set(\trigFr , 10) ;
```

Ciò che avviene è che `seq` è una sequenza che ripete infinite volte (`inf`) un pattern costituito dai numeri $42, 45, 49, 50$, e da un quinto elemento definito da `Dxrand`: analogamente a `Drand`, quest'ultima UGen pesca a caso nell'array fornito (`[78, 81, 85, 86]`), ma la sequenza in uscita non prevede ripetizioni dello stesso elemento. Quando `Demand` arriva all'elemento dell'array `Dseq` costituito da `Dxrand` lo esegue, cioè genera un certo numero di elementi pari a `repeats` a partire dalla definizione di quest'ultimo. La dimensione della sequenza in uscita (cioè il valore dell'argomento `repeats`) è controllata da `LFNoise0`: in sostanza, varia pseudo-casualmente nell'escursione $[0, 4]$. Nel primo caso, il contributo di `Dxrand` è nullo, nel secondo consiste in tutti e quattro i valori, in ordine pseudo-casuale ma senza ripetizioni adiacenti. Dal punto di vista audio, la presenza dell'array `[0,0.7]` nell'argomento `freq` produce espansione multicanale: il canale destro sarà "detuned" di 0.7 Hz. Inoltre, `LFPulse` produce una sorta di acciaccatura iniziale all'ottava sopra, che è collegata con il triggering dell'evento demand rate attraverso `trigFr`.

```
(
SynthDef("randMelody",
    { arg base = 40, trigFreq = 10;
    var freq, trig, reset, seq;
    var structure = base+[0, 2, 3, 5, 6, 8] ;
    trig = Impulse.kr(trigFreq);
    seq = Dseq(
        structure.add(
        Dxrand(structure+12, LFNoise0.kr(6).unipolar*6))
        , inf).midicps;
    freq = Demand.kr(trig, 0, seq);
    Out.ar(0,
        Mix.fill(5, {Saw.ar(freq +0.1.rand + [0,0.7])* 0.1}));
}).add;
)

x = Synth.new("randMelody") ;
x.free ;

(
15.do({ arg i ;
    Synth.new("randMelody",
        [\base , 20+(i*[3, 5, 7].choose), \trigFreq , 7+(i/10) ])
})
)
```

Nell'esempio precedente una synthDef costruita in maniera analoga al caso precedente prevede come argomenti `base` e `trigFreq`: il primo rappresenta la frequenza di base in notazione MIDI, il secondo la frequenza di triggering. Qui `Dxrand` aggiunge alla sequenza delle altezze la stessa sequenza ma un'ottava sopra, con il solito meccanismo della selezione casuale delle altezze e della lunghezza (9). Anche a livello di sintesi, si tratta di una variazione di aspetti già discussi (13), come si può ascoltare nell'istanziazione di un synth (17). Il ciclo successivo sovrappone 15 voci: in ognuna la frequenza di base è incrementata di `i` per un valore a scelta tra `[3, 5, 7]` (musicalmente, terza minore, quarta e quinta). In più, ad ogni iterazione (per ogni synth) la frequenza di triggering (`7`) incrementa di una quantità pari a `i/10`. Quest'ultimo aspetto permette di realizzare un progressivo *dephasing*: i tempi degli strati sono cioè lievemente differenti, e il sincronismo iniziale si disperde progressivamente.

L'idea alla base delle UGen che si basano sul meccanismo Demand è quella di fornire la possibilità di annidare all'interno delle synthDef aspetti tipicamente di più alto livello. Una synthDef diventa un sequencer a tutti gli effetti. Per di più, la possibilità dell'incassamento di una UGen nell'altra è estremamente potente. Ma in realtà forse è concettualmente più lineare separare due aspetti, che lavorano tipicamente a tassi di aggiornamento diverso: la sintesi (a tasso audio) e lo scheduling (a tasso di evento). Non a caso, le UGen Demand sono strettamente relate con i cosiddetti "Pattern", strutture per il controllo compositivo di alto livello sul lato del linguaggio. Infatti, le UGen Demand costituiscono una sorta di versione sul lato server dei "Pattern", che verranno discussi più avanti.

7.3 A lato linguaggio: Orologi e routine

In effetti, il modo più usuale (e forse più sensato) per effettuare lo scheduling degli eventi consiste nel gestirlo dal lato del linguaggio. Per definire l'esecuzione di eventi nel tempo, SC prevede una classe astratta `Clock` da cui discendono tre classi utilizzabili, `SystemClock`, `TempoClock` e `AppClock`. Dunque, un evento può essere schedulato così:

```
1 (
2 "aspettando tre secondi".postln ;
3 SystemClock.sched(3.0, { "fatto".postln });
4 )
```

Il metodo `sched` definito sulla classe `SystemClock` prevede come argomenti un intervallo di tempo e una funzione che verrà valutata dopo che questo ultimo sarà passato. Se si valuta il codice, si ottiene:

```
1 aspettando tre secondi
2 SystemClock
3 fatto
```

L'interprete esegue immediatamente la riga 2 (e stampa sulla post window), quindi passa alla riga successiva e la valuta, terminando l'interpretazione: infatti, subito stampa `SystemClock` sullo schermo, cioè l'oggetto che viene restituito dall'ultima espressione. Durante l'esecuzione, lo scheduling è avviato, dopo 3 secondi la funzione viene valutata, e `fatto` risulta sullo schermo. I tre tipi di clock si differenziano per priorità e per versatilità. `SystemClock` è l'orologio a priorità più alta e utilizza un concetto di tempo assoluto (non musicale), cronografico. `TempoClock` ha funzionalità analoga a `SystemClock` ma in più permette di gestire un concetto di tempo musicale. In `TempoClock` il valore predefinito dell'unità temporale assume che 1 beat abbia una durata di 1 secondo (tempo = 60 bpm). In questo caso, il calcolo del tempo coincide con quello cronografico. Ma l'attributo tempo può essere modificato a piacere secondo appunto il modello del tempo musicale. Nell'esempio seguente, ci sono tre orologi in azione, oltre a `SystemClock` due istanze di `TempoClock` con tempi diversi. Si noti che il tempo non è gestito come numero di beat al minuto (bpm) ma come beat al secondo: dunque, 240 bpm equivalgono a $240/60$ "bps". Le righe 7-9 gestiscono lo scheduling e nella valutazione risulta evidente che il valore temporale di 4.0 dipende dal beat definito.

```
(
"via".postln ;
// due TempoClock: t, u
t = TempoClock(240/60) ;
u = TempoClock(120/60) ;
// scheduling
d = thisThread.seconds;
SystemClock.sched(4.0, { "fatto con il cronometro".postln; });
t.sched(4.0, { "fatto col bmp 240".postln });
u.sched(4.0, { "fatto col bpm 120".postln });
)
```

Infine, `AppClock` è un orologio a bassa priorità che deve essere utilizzato per lo scheduling di eventi che riguardino le GUI. Infatti, esse hanno in SC una priorità più bassa dell'audio. Questo vuol dire che il calcolo dell'audio precede sempre come priorità le GUI, le quali vengono aggiornate solo se ci sono risorse computazionali disponibili. Le GUI perciò devono essere schedulate via `AppClock`.

L'uso delle sottoclassi di `Clock` è alla base dello scheduling in SC, ma tipicamente non in forma diretta: piuttosto, sono altre strutture dati che possono essere schedulate in relazione ad un clock. La struttura di controllo fondamentale in proposito in SC è la routine. Di per sé, le routine sono strutture dati che estendono il modus operandi delle funzioni, e il loro uso non è limitato allo scheduling: lo scheduling è soltanto una delle applicazioni possibili delle routine, anche se la più tipica. L'esempio seguente mostra una routine minimale. Come si vede, la routine riceve come argomento una funzione. L'unico messaggio non conosciuto è `wait`, ricevuto da `2`, che è un numero a virgola mobile. Intuibilmente, `wait` concerne la gestione del tempo, cioè lo scheduling. La routine `r` definisce cioè una programmazione che deve essere eseguita, in cui il messaggio `wait` ricevuto da un numero indica un tempo di attesa (pari al ricevente del messaggio) prima di proseguire nell'esecuzione della espressione successiva. Le espressioni sono tre (4-6): due richieste di stampa inframmezzate appunto dalla chiamata di `wait`.

```
(
// Minimal routine
r = Routine.new({
    "aspetto 2 secondi".postln ;
    2.wait;
    "fatto".postln ;
}) ;
)

SystemClock.play(r) ;
```

La gestione dell'interpretazione della funzione nel tempo è affidata ad un orologio, nella circostanza `SystemClock`: all'orologio è chiesto di eseguire (`play`) la routine (`r`, 10). L'orologio interpreta gli oggetti che ricevono il messaggio `wait` come quantità temporali in cui sospendere l'esecuzione della sequenza di espressioni nella funzione. Quando questa riprende, le espressioni successive sono valutate il più in fretta possibile (cioè, ad un tempo definito dai cicli del processore, che empiricamente si può assumere come nullo).

Si potrebbe obiettare che l'esempio non è qualitativamente molto diverso da quello precedente in cui si utilizza direttamente il metodo `sched` su `SystemClock`. Ma c'è una differenza radicale: con una routine si può implementare una sequenza complessa a piacere di espressioni, che vengono interpretate nel tempo. Nel prossimo esempio, la funzione contiene un ciclo che per 10 volte stampa `"rallentando"`, quindi assegna a `time` un valore pari a `i*0.1`, cioè determina lo scheduling proceduralmente in funzione del contatore. Nella circostanza, la routine `r` attende un tempo `time` che cresce progressivamente. Realizzato il ciclo, `r` prescrive di attendere 1 secondo e di scrivere `"fi"`, poi un altro secondo e di scrivere `"ne"`.

```
(
// Qualche espressione in piu'
r = Routine.new(
 {     var time ;
    10.do ({ arg i ;
        "rallentando".postln ;
        time = (i*0.1).postln ;
        time.wait ;
    }) ;
    1.wait ;
    "fi".postln ;
    1.wait ;
    "ne".postln ;
 }
) ;
)

SystemClock.play(r) ;
r.reset ;
SystemClock.play(r) ;
```

Si noti che la routine "ricorda" il suo stato interno: se si valuta nuovamente `SystemClock.play(r)` (18) si ottiene di ritorno sulla post window l'oggetto `SystemClock`, poiché la routine è ormai terminata. Per riportare lo stato interno della routine alla condizione iniziale è necessario inviarle il messaggio `reset` (19). A questo punto è possibile eseguire da capo la routine (20).

Come usuale, il linguaggio SC permette costrutti ancora più sintetici, che sfruttano il cosiddetto "polimorfismo", cioè il fatto che certi metodi abbiano lo stesso nome ma semantica diversa su oggetti diversi, come si può vedere nell'esempio seguente.

```
1  (
2  // polimorfismo, 1
3  r = Routine({
4      10.do{|i| i.postln; 1.wait}
5  })
6  )
7  r.play ;

9  // polimorfismo, 2
10 { 10.do{|i| i.postln; 1.wait} }.fork ;
```

Dopo la definizione della routine `r` (3-5), questa viene eseguita chiamando su di essa il metodo `play`, il quale chiamerà un orologio (qui quello di default, cioè `TempoClock` ed eseguirà il processo già discusso. Ancora il metodo `fork` è definito direttamente su una funzione (10), che viene allora avviluppata in una routine e la esegue sull'orologio di default (sempre rispetto alle funzioni, si pensi analogamente all'insieme di operazioni "dietro la scena" nel caso metodo `play`). A margine, un elemento di controllo interattivo utile va subito specificato: una routine si interrompe chiamando STOP dal menu Language (ovvero con la solita combinazione di tasti per interrompere l'audio).

Da questi semplici esempi dovrebbe emergere chiaramente come una routine costituisca la base per la gestione di processi temporali anche di grande complessità. Semplificando, quando si vogliono eseguire espressioni secondo una certa progressione temporale è sufficiente avvilupparle in una routine ed inframmezzare opportunamente espressioni del tipo `n.wait`. Nella sua forma più semplice, questa sequenza di espressioni può essere pensata come una lista da eseguire in sequenza, esattamente come avviene in un file MIDI o in un file di testo "score" di Csound. Ma ovviamente l'aspetto interessante in SC è l'utilizzo di controlli di flusso nella funzione, e la possibilità di interazione con i processi attraverso l'uso di variabili, aspetti che sono la cifra del linguaggio di programmazione.

7.4 Orologi

Il prossimo esempio discute la generazione di una GUI che funziona da semplice cronometro, e permette di introdurre alcuni elementi in più nella discussione sullo scheduling. Una volta eseguito il codice, si apre una piccola finestra che visualizza l'avanzamento del tempo a partire da 0: il cronometro parte e viene interrotto nel momento in cui la finestra viene chiusa.

```
1  (
2  var w, x = 10, y = 120, title = "Tempus fugit"  ; // GUI var
3  var clockField ;
4  var r, startTime = thisThread.seconds ; // scheduling

6  w = Window.new(title, Rect(x, y, 200, 60)) ;
7  clockField = StaticText.new(w, Rect(5,5, 190, 30))
8      .align_(\center )
9      .stringColor_(Color(1.0, 0.0, 0.0))
10     .background_(Color(0,0,0))
11     .font_(Font(Font.defaultMonoFace, 24));
12 r = Routine.new({
13     loop({
14         clockField.string_((thisThread.seconds-startTime)
15             .asInteger.asTimeString) ;
16         1.wait }) // l'orologio si aggiorna ogni secondo
17     }).play(AppClock) ;
18 w.front ;
19 w.onClose_({ r.stop }) ;
20 )
```

Le prime righe dichiarano le variabili. Come si vede, fondamentalmente il codice prevede due tipi di elementi: elementi GUI (righe 2, 3) e elementi che si occupano di gestire l'aggiornamento dei valori temporali. In particolare, la riga 4 assegna a `startTime` il valore di `thisThread.seconds`: `thisThread` è un oggetto particolare, una pseudo-variabile ambientale che contiene una istanza della classe `Thread`, la quale tiene il conto di di quanto tempo è passato dall'inizio della sessione dell'interprete. Se si valuta un paio di volte l'espressione `thisThread.seconds` si noterà come il valore restituito dal metodo `seconds` incrementi di conseguenza (e corrisponda ai secondi passati da quando si è avviata l'applicazione). Dunque, `startTime` contiene un valore che rappresenta il tempo in cui il codice viene valutato (il momento 0 del cronometro). Inutile dilungarsi analiticamente sulla costruzione dell'interfaccia grafica che non ha nulla di particolare. Si noti soltanto lo stile tipico di programmazione nelle GUI che

prevede il concatenamento dei messaggi (righe 8-11) per impostare le diverse proprietà grafiche. Ancora, il tipo di carattere è definito da un oggetto `Font` e prende come nome del tipo quello restituito da `Font.defaultMonoFace`, cioè quello predefinito a spaziatura uniforme ("mono") dal sistema operativo su cui si sta lavorando. Le righe 12-17 definiscono invece la routine che si occupa ogni secondo di calcolare il nuovo valore del cronometro e di aggiornare il campo dell'elemento GUI. La funzione contenuta nella routine contiene un ciclo infinito. Il ciclo viene definito da `loop`: è un sinonimo di `inf.do`, quindi di un ciclo infinito che esegue due espressioni: la prima aggiorna il campo dell'ora impostando la proprietà `string` di `clockField` (righe 13-14), la seconda richiede di attendere un secondo prima di ripetere il ciclo (`1.wait`, riga 16). Il nuovo valore del tempo da visualizzare nella GUI viene calcolato in tre passi. In primo luogo viene chiesto all'interprete il valore attuale del tempo passato attraverso una chiamata di `thisThread.seconds`: ad esso viene sottratto il tempo di partenza `startTime` per ottenere la differenza. Quindi il risultato viene convertito in numeri interi (qui non interessano i valori decimali dei secondi): dunque nel numero intero di secondi passati da quando il cronometro è esistito. Infine, il metodo `asTimeString`, definito per la classe `SimpleNumber`, restituisce una stringa nella forma "ore:minuti:secondi". Ad esempio:

```
20345.asTimeString
05:39:05:000
```

Il metodo `play` riceve come argomento un oggetto di tipo `Clock`, ma non è `SystemClock`: infatti, quest'ultimo non può essere utilizzato nel caso di GUI. Al suo posto deve essere utilizzato `AppClock`. Infine, la riga 19 definisce una proprietà della finestra `w`: `onClose` prevede come valore una funzione che viene eseguita nel momento in cui `w` è chiusa. La funzione contiene `r.stop`: la routine `r` viene ciò arrestata alla chiusura della finestra.

Il codice seguente è una variante della definizione della routine `r` che dimostra le ultime considerazioni. Come si diceva, la soluzione è obbligata perché dipende dalla gestione delle priorità: nel caso di scheduling, i messaggi al server audio hanno la priorità su altre funzionalità, GUI inclusa. Questo non permetterebbe di gestire da una stessa routine il controllo di un synth e di un elemento GUI: una simile operazione è possibile utilizzando all'interno della routine il metodo `defer` implementato dalle funzioni. In sostanza, nella routine

le espressioni che concernono la GUI vengono raccolte all'interno di una funzione (racchiuse tra parentesi graffe) a cui viene inviato il messaggio `defer` che permette di differire il risultato della loro valutazione nel momento in cui vi siano risorse computazionali disponibili (ovvero: senza sottrarne alla computazione audio). Dunque nell'esempio l'aggiornamento della GUI è racchiuso in una funzione a cui è inviato `defer` (riga 3-6): è allora possibile utilizzare `SystemClock` (8).

```
1 r = Routine.new({
2     loop({
3         {
4             clockField.string_((thisThread.seconds-startTime)
5                 .asInteger.asTimeString.postln) ;
6         }.defer ; // componenti GUI devono essere "differite"
7         1.wait })
8 }).play(SystemClock) ;
```

7.5 Sintetizzatori vs. eventi

L'approccio precedente permette a tutta evidenza di essere esteso all'audio. Se si eccettua il problema della priorità, non c'è infatti niente di peculiare alle interfacce grafiche nella gestione dello scheduling discussa precedente. La synthDef seguente genera un segnale dato dalla somma di 5 onde a dente di sega la cui frequenza è armonica. In più, è presente una componente di "detune" gestita da una sinusoide la cui frequenza dipende dall'armonica (attraverso `i`). Il segnale è inviluppato da una rampa (6) che cresce da 0 a 1 per $\frac{9}{10}$ del suo sviluppo e ridiscende a 0 nell'ultimo $\frac{1}{10}$. Un punto di rilievo è dato dal trigger, che in questo caso non è generato da una UGen ma innescato da fuori, attraverso un argomento, `t_trig`. Una volta che il trigger ha valore ≥ 0, perché possa scattare di nuovo, dovrebbe essere riportato (con l'invio di un altro messaggio) ad un valore ≤ 0. Il che è piuttosto scomodo. Se però il nome di un argomento che indica un trigger inizia con `t_`, allora per convenzione di SC è possibile semplicemente inviare un singolo messaggio, senza quello di "reset".

```
(
SynthDef(\saw , { arg freq = 440, detune = 0.1, dur = 0.1, t_trig = 1;
    Out.ar(0,
        Pan2.ar(
            Mix.fill(5, {|i|
                EnvGen.kr(Env([0,1,0], [dur*0.9,dur*0.1]), t_trig)
                *
                Saw.ar(
                    freq:freq*(i+1)+
                    (SinOsc.kr((i+1)*0.2, 0, detune, detune*2)),
                    mul:1/5)})
    ))
}).add
)

(
x = Synth(\saw ) ;
~mul = 1 ;
~base = 60 ;
Routine.new({
    var dur ;
    inf.do {|i|
        dur = [1,2,3,4,2,2].choose*~mul ;
        x.set(
            \t _trig, 1,
            \freq , ([0,2,3,4,5,7, 9,10,12].choose+~base).midicps,
            \detune , rrand(1.0, 3.0),
            \dur , dur*0.95
        ) ;
        dur.wait ;
    };
    }
).play(SystemClock)
)
// controllare la routine interattivamente
~mul = 1/16; ~base = 72 ;
~mul = 1/32; ~base = 48 ;
```

Una volta costruito il synth `x`, lo scheduling è gestito attraverso una routine infinita (`inf.do`, 20), che ad ogni iterazione imposta (`x.set`, 22) i valori degli argomenti del synth. La routine sfrutta abbondantemente valori pseudo-casuali: ad esempio, pescando tra un insieme di durate (21) e altezze (24) e impostando il detune nell'escursione $[1.0, 3.0]$ (25). Due variabili ambientali sono utilizzate

nella routine per gestire la velocità degli eventi e l'altezza di base, rispettivamente a `~mul` e `~base`. In questo modo, diventa possibile gestire lo scheduling interattivamente (si valutino 34 e 35).

Nell'esempio precedente l'idea fondamentale è quella di costruire un synth (uno strumento) e controllarlo attraverso una routine. La synthDef seguente permette di introdurre un secondo approccio. Essa prevede una semplice sinusoide a cui è aggiunto un vibrato ed un inviluppo d'ampiezza. I parametri di entrambi sono controllabili dall'esterno: `a, b, c` rappresentano punti dell'inviluppo, `vibrato` e `vibratoFreq` i due parametri del vibrato.

```
(
SynthDef("sineMe1",{ arg out = 0, freq = 440, dur = 1.0, mul = 0.5, pan = 0,
    a, b, c,
    vibrato, vibratoFreq;

    var env;
    env = Env.new([0,a,b,c,0], [dur*0.05,dur*0.3,dur*0.15,dur*0.5], 'welch');
    Out.ar(out,
        Pan2.ar(
            SinOsc.ar(
                freq: freq+SinOsc.kr(mul:vibrato, freq: vibratoFreq),
                mul:mul
            ) * EnvGen.kr(env, doneAction:2)
    ), pan)
}).add;
)
```

L'inviluppo d'ampiezza è utilizzato da una UGen `EnvGen`, il cui argomento `doneAction` riceve un valore 2. Ciò significa che, una volta concluso l'inviluppo, il synth viene deallocato sul server. Questo implica che il synth non esiste più: esso si comporta perciò non tanto come uno strumento ma come un evento. Si osservi cosa avviene nello routine:

```
(
var r = Routine.new({
    inf.do({ arg i ;
        var env, dur = 0.5, freq, end, mul, pan, vibrato, vibratoFreq ;
        a = 1.0.rand ;
        b = 0.7.rand ;
        c = 0.5.rand ;
        pan = 2.0.rand-1 ;
        // 13 altezze su un modo un frammento modale non ottavizzante
        freq = ([0,2,3,5,6,8,10,12,13,15,16,18].choose+70).midicps ;
        dur = rrand(0.015, 0.5) ;
        mul = rrand(0.05, 0.8) ;
        vibrato = (dur-0.015)*100 ;
        vibratoFreq = dur*10 ;
        Synth.new(\sineMe1 , [
            \vibrato , vibrato,
            \vibratoFreq , vibratoFreq,
            \a , a,
            \b , b,
            \c , c,
            \freq , freq,
            \dur , dur,
            \mul , mul,
            \env , env]
            ) ;
        end = 0.15.rand;
        (dur+end).wait;
    });
});

SystemClock.play(r);
)
```

Ad ogni iterazione del ciclo viene istanziato un synth che si dealloca nel momento in cui l'inviluppo è concluso. In questo secondo esempio, l'oggetto synth non è trattato come uno strumento, cioè come un dispositivo persistente (una tromba, un basso, un fagotto) il cui comportamento va controllato in funzione del presentarsi di nuovi eventi. Piuttosto, qui il synth diventa un evento sonoro, l'equivalente di una nota musicale. Il metodo `doneAction:2` dimostra qui uno dei suoi uso più tipici. Si tratta di un esempio semplice di procedura generativa che può durare un tempo indefinito, come potrebbe avvenire in un installazione. Nuovamente, la synthDef fa largo uso di valori pseudo-casuali,

che offrono alla sinusoide una qualità tipicamente "fischiata". L'unico aspetto di rilievo è il controllo della frequenza (riga 10). L'array definisce una sequenza di altezze che descrivono un modo non ottavizzante di 13 altezze, ne seleziona una stocasticamente, e vi aggiunge 70 (l'altezza base).

Il prossimo esempio riprende il ragionamento sui segnali del paesaggio sonoro conosciuto già visto nel caso del telefono. Il segnale dell'ora esatta emesso dalla Radio RAI è un segnale di "sincronizzazione e disseminazione" che prende il nome di "Segnale orario RAI Codificato (SRC)", generato dall'Istituto Nazionale di Ricerca Metrologica (INRIM). Come indicato dall'Istituto:

> "Il segnale orario generato dall'Istituto e diffuso dalla RAI (SRC), è costituito da un codice di data suddiviso in due segmenti di informazione, generati in corrispondenza dei secondi 52 e 53, e da sei impulsi acustici sincroni con i secondi 54, 55, 56, 57, 58, e 00. I sei impulsi acustici sono formati da 100 cicli sinusoidali di una nota a 1000Hz. La durata di ciascun impulso è di 100 millisecondi[5]."

La Figura **7.2** è lo schema di codifica ufficiale dell'SRC fornita dall'INRIM. Il segnale è formato da due blocchi di dati binari codificati con segmenti di 30 ms in cui una sinusoide a 2000 Hz rappresenta lo 0 e una sinusoide a 2500 Hz l'1. A un primo blocco di 32 bit (totale: $32 \times 30 = 960$ ms) al secondo 52, fa seguito al secondo 53 un secondo blocco di 16 bit (totale: $16 \times 30 = 480$ ms). Quindi 5 emissioni di sinusoidi di 100 ms (secondi $54 - 58$), silenzio al secondo 59, e l'ultima emissione al secondo 60. Come si vede, i 48 bit rappresentano un insieme di informazioni relative all'ora, e l'alternanza (pr quanto irregolare) tra le due frequenze a 2 e 2.5 kHz che ne consegue produce acusticamente il tipico suono trillato d'apertura.

L'esempio seguente reimplementa il segnale SRC in SuperCollider. Le righe 5 e 6 generano due sequenze casuali di 32 e 16 bit. Sarebbe ovviamente possibile generare le sequenze corrette, e in questo modo l'implementazione successiva non cambierebbe. I due cicli (9-13 e 16-20) generano i due blocchi di 960 e 480 ms. Qui si è scelto un uso dei synth come eventi. Segue (22) la generazione di una sequenza di impulsi a 1000 Hz, in questo caso, per differenziare gli approcci, il primo blocco di 5 impulsi è ottenuto inviluppando una sinusoide con un segnale di tipo `LFPulse`, opportunamente parametrato. L'ultimo impulso è invece un synth-evento di 100 ms. In tutti i casi, la UGen `Line` è usata sostanzialmente come "deallocatore" temporizzato del synth.[6].

[5] `http://www.inrim.it/res/tf/src_i.shtml`

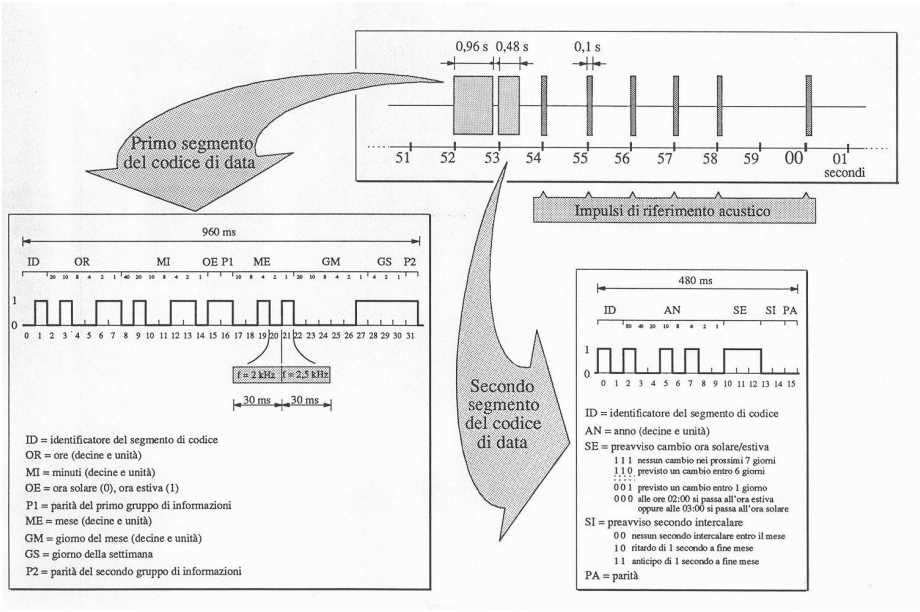


Fig. 7.2 "Schema temporale delle emissioni dei vari elementi che costituiscono il SRC".

[6] Dal punto di vista acustico, sia questo segnale che quello telefonico precedente suonano un po' troppo puliti e brillanti. Questo perché di solito sono ascoltati a partire da una ricezione analogica, per di più di frequente a bassa qualità. Se si vuole ottenere questo effetto, si può applicare ad esempio un filtro, secondo quanto si vedrà nel prossimo capitolo.

```
(
// Segnale orario SRC
{
    var amp = 0.125 ;
    var firstBit = Array.fill(32, {[0,1].choose}) ;
    var secondBit = Array.fill(16, {[0,1].choose}) ;
    var freq ;
    // primi 32 bit
    firstBit.do{|i|
        freq = [2000, 2500][i] ;
        {SinOsc.ar(freq, mul:amp)*Line.kr(1,1,0.30, doneAction:2)}.play ;
        0.03.wait
    } ;
    0.04.wait ;
    // secondi 16 bit
    secondBit.do{|i|
        freq = [2000, 2500][i] ;
        {SinOsc.ar(freq, mul:amp)*Line.kr(1,1,0.30, doneAction:2)}.play ;
        0.03.wait
    } ;
    0.52.wait;
    // 5 impulsi a 1000 Hz
    {
        SinOsc.ar(1000, mul:amp)
        *
        LFPulse.ar(1, width:0.1)*Line.kr(1,1,5, doneAction:2)
    }.play ;
    6.wait;
    // ultimo
    {SinOsc.ar(1000, mul:amp)*Line.ar(1, 1, 0.1, doneAction:2)}.play ;
}.fork ;
)
```

Nell'esempio si noti che non c'è menzione della routine. La routine c'è ma non si vede. Infatti, il metodo `fork` definito sulle funzioni (vero perno concettuale e operativo di SC) assume che la funzione su cui è chiamato sia intesa come una funzione da schedulare (cioè come quelle che si specificano per le routine). Dunque, `fork` inviluppa la funzione su una routine e la esegue con TempoClock.

Le routine (e in generale i processi) possono sempre avvenire in parallelo. Nel prossimo esempio, vi sono 20 routine in esecuzione contemporanea. Le altezze sono generate da un array (`pitches`) che incrementa con un passo di 4:

essendo altezze, si ha una sequenza di terze maggiori. L'altezza di partenza è molto grave (20, meno di 26 Hz). Ogni routine genera ripetutamente un evento (un'onda quadra inviluppata percussivamente). L'intervallo di generazione è dato dalla serie geometrica `times`. Altezza, intervallo di ripetizione e posizione sul fronte stereo sono vincolate al contatore (è il cosiddetto *parameter linking*), in modo che le frequenze gravi si ripetano più frequentemente e stiano a sinistra, e quelle acute si ripetano meno frequentemente e stiano a destra. L'effetto è una sorta di sgranamento di un arpeggio che produce un *dephasing* progressivo, per poi tornare in fase alla durata minima data dal comune multiplo delle durate in gioco. Il ritardo negli attacchi consegue al fatto che prima le routine attendono (un tempo variabile), e poi istanziano un synth.

```
(
var num = 20 ;
var pitches = Array.series(num, 0, 4)+20 ;
var times =  Array.geom(num, 1, 1.01) ;
num.do{|i|
    { inf.do{
        times[i].wait;
        {
            Pan2.ar(
                in:    Pulse.ar(pitches[i].midicps)
                *
                EnvGen.kr(Env.perc, 1, 0.2, doneAction:2),
                pos: i.linlin(0, num.size-1, -1, 1)
            )
        }.play
    }}.fork
} ;
s.scope ;
)
```

7.6 Interludio grafico: disegni e animazioni

SuperCollider offre grandi possibilità grafiche, non solo rispetto all'insieme degli oggetti che implementano direttamente l'interazione con l'utente (pulsanti, cursori, rotativi, e così via), ma anche rispetto alla grafica generativa. Il prossimo esempio illustra alcuni aspetti di base.

```
1  w = Window("tester", Rect(10, 10, 500, 500))
2  .background_(Color.white).front ;
3  w.drawFunc = {
4      Pen.strokeColor = Color.rand(0.1, 0.9) ;
5      Pen.moveTo(0@250) ;
6      Pen.lineTo(250@250) ;
7      Pen.lineTo(500@500) ;
8      Pen.stroke ;
9  } ;

11 w.refresh ;
```

Come si vede, una finestra (ma lo stesso vale anche per una classe grafica che rappresenta appositamente l'idea di "canvas", `UserView`) definisce tra i suoi metodi `drawFunc`, cioè letteralmente una funzione di disegno (3-9). Questa metodo viene valutato quando la finestra viene costruita (se definito in fase di inizializzazione), ma può essere richiamato a richiesta attraverso il metodo `refresh` (11). Se si valuta nuovamente la riga 11 si nota come il colore della riga cambi, in funzione del colore scelto casualmente (riga 4). Poiché la scelta casuale è interna alla funzione, ogni valutazione della stessa forzata da `refresh` produce infatti un nuovo colore. All'interno di `drawFunc`, e soltanto in quella sede, è possibile utilizzare la classe `Pen`, la quale non ha metodi di istanza ma solo metodi di classe: molto semplicemente, non si creano oggetti `Pen` ma si usano direttamente i metodi della classe. `Pen` implementa una logica che è tipica di molti linguaggi di programmazione della grafica (in primis PostScript, e poi anche Processing o NodeBox), e che deriva direttamente dal controllo dei plotter, le stampanti dotate di braccio meccanico. Quest'ultimo fatto non è accessorio per capire la logica di funzionamento del tutto. Sostanzialmente, `Pen` è come un pennino a cui si dice: muoviti in quel punto, disegna una riga fino a quel punto, cambia colore dell'inchiostro, riempi una superficie, disegna una linea curva, disegna un ovale, e così via. Nell'esempio, prima viene definito il colore del tratto ("stroke", `strokeColor`, 4), quindi il pennino viene spostato nel punto $(0, 250)$. Due cose in proposito: la sintassi `0@250` è un'abbreviazione (zucchero

sintattico) per un'istanza della classe `Point`, cioè `Point(0,250)`, che intuibilmente rappresenta il concetto di punto nel piano; `move` non indica il tracciamento di una riga, ma lo spostamento del pennino (si pensi al braccio del plotter). Quindi vengono tracciate due righe attraverso il metodo `lineTo`: si noti che il processo è incrementale, cioè il pennino si muove verso un punto, dove rimane fino a nuova indicazione. Fino a questo momento in realtà si ha una descrizione di ciò che il plotter `Pen` deve fare ma non la sua effettiva realizzazione, che si ottiene soltanto quando si chiama `stroke` ("traccia"). A quel punto le istruzioni precedenti vengono effettivamente prese in carico dal processo di disegno. Dunque, `Pen` definisce una macchina a stati per il disegno, molto efficiente dal punto di vista computazionale.

Il prossimo esempio dimostra le possibilità di interazione con `Pen`.

```
1  // dimensione del cerchio
2  ~dim = 50 ; ~dim2 = 25 ;
3  // parametri del colore
4  ~hue = 0; ~sat = 0.7; ~val = 0.7;

6  w = Window("tester", Rect(10, 10, 500, 500))
7  .background_(Color.white).front ;
8  w.drawFunc = {
9      var oo = 250-(~dim*0.5) ; // per centrare il cerchio
10     Pen.addOval(Rect(oo, oo, ~dim, ~dim)) ; // il cerchio
11     Pen.color_(Color.hsv(~hue, ~sat, ~val)) ; // colore
12     Pen.fill ; // e riempimento
13     oo = 250-(~dim2*0.5) ; // per centrare il cerchio
14     Pen.addOval(Rect(oo, oo, ~dim2, ~dim2)) ; // il cerchio
15     Pen.color_(Color.white) ; // colore
16     Pen.fill ; // e riempimento
17 } ;

19 ~hue = 0.2; w.refresh ;
20 ~dim = 200; w.refresh ;
21 ~dim2 = 10; w.refresh ;
```

La funzione assegnata a `drawFunc` disegna un cerchio attraverso il metodo `addOval` il quale descrive un ovale a partire dal rettangolo in cui questo deve essere incluso: se questo è un quadrato, allora si ottiene un cerchio (10), il cui lato è dato dalla variabile ambientale `~dim`. Il cerchio viene centrato nella finestra attraverso la riga 9 (il quadrato è definito infatti dal vertice in alto a sinistra,

e dunque deve essere traslato opportunamente). Quindi, il cerchio è colorato: prima si definisce il colore del pennino (11) riferendosi a tre variabili ambientali definite in precedenza (4), poi si chiede al pennino di riempire la superficie con il metodo `fill` (si noti: non `stroke`, 12). Nelle righe successive il processo è ripetuto aggiungendo un secondo cerchio di dimensione `~dim2` e colorato del colore dello sfondo (bianco) che produce l'effetto "bucato" sovrapponendosi al precedente. Si ricordi che infatti `Pen` è incrementale: se `~dim2` è maggiore di `~dim` allora non si vedrà nulla. Le righe 19-21 dimostrano l'interazione possibile attraverso il riferimento alle variabili ambientali. Se ne varia il valore e si forza l'esecuzione di `drawFunc` attraverso `refresh`. `Pen` offre moltissime possibilità, la cui esplorazione è al di fuori degli obiettivi di questo testo e che è lasciata al lettore, a partire dall'help file. Inoltre, poiché nelle GUI è possibile ad esempio tracciare la posizione del mouse, diventa possibile interagire visivamente con la grafica disegnata. In questo capitolo è particolarmente interessante il controllo temporale (ovvero, l'animazione) della grafica. Il codice seguente, che assume che il precedente sia stato valutato, è ormai di agevole comprensione:

```
(
{inf.do{|i|
    k = i*3 ;
    ~hue = k%500/500;
    ~dim = k%500;
    ~dim2 = ~dim*0.25 ;
    w.refresh ;
    0.1.wait ;
}}.fork(AppClock)
)
```

Una routine infinita utilizza il contatore `i` per calcolare un secondo contatore che incrementa più velocemente, `k`. Quindi, la variabile associata alla tinta, `~hue`, viene messa in relazione al modulo di 500, e divisa per 500 (4). La ratio è che al massimo potrà avere valore 1, che è il valore massimo consentito per l'argomento `hue` in `Color.hsv`. Analogamente, su `~dim` viene applicato lo stesso modulo (5), in modo che non superi la dimensione della finestra (un fatto del tutto possibile, ma qui non desiderato). Quindi, `~dim2` è calcolata come $\frac{1}{4}$

(`0.25`, 6) di `~dim`. La finestra è aggiornata e si attendono 100 millisecondi per ripetere l'operazione. Infine, la routine è schedulata (per forza, salvo `defer`) con `AppClock`.

7.7 Routine vs. Task

Le routine possono essere riportate nella condizione iniziale attraverso il messaggio `reset` e interrotte attraverso il messaggio `stop`. Tuttavia, nel momento in cui ricevono il messaggio `stop`, per poter essere di nuovo attivate devono prima ricevere il messaggio `reset` che le riporta nella condizione di partenza. Questo aspetto costituisce una limitazione potenzialmente importante all'uso musicale. La classe `Task` implementa questo comportamento: è un processo "che può essere messo in pausa" (*pauseable process*). Dal punto di vista implementativo, la definizione di un task assomiglia molto a quella di una routine, come si può vedere nell'esempio seguente:

```
(
t = Task({
        inf.do({|i| i.post; " passi verso il nulla".postln; 1.wait})
    }) ;
)

// inizio
t.play ;

// pausa: lo stato interno e' conservato
t.pause ;

// riparti dall'ultimo stato
t.resume ;

// reimposta a 0
t.reset ;

// ferma: lo stesso di pause
t.stop ;

// play ha lo stesso effetto di resume
t.play ;
```

Come si vede, i metodi `stop`/`pause`, e `play`/`resume` si comportano esattamente nello stesso modo. Dal punto di vista interno, una routine è in certe condizioni leggermente più precisa in caso di arresto/riavvio. Ma comunque, in generale, se un processo non deve essere controllato (perché non termina mai o perché termina autonomamente) conviene usare una routine, se invece deve essere fermato e riavviato, allora conviene usare un task. L'esempio seguente sfrutta i task per "mettere in pausa" dei processi. L'occasione della synthDef `"bink"` permette di discutere anche alcuni aspetti di trattamento del segnale.

```
(
SynthDef("bink", { arg freq = 440, pan = 1;
    var sig, del;
    // sorgente
    sig = Pulse.ar(freq
        *Line.kr(1,
            LFNoise1.kr(0.1)
            .linlin(-1,1, -0.5, 0).midiratio, 0.1),
        width:0.1
    ) ;
    // coda di ritardi
    del = Mix.fill(20, {|i|
        DelayL.ar(sig,
            delaytime: LFNoise1.kr(0.1)
            .linlin(-1,1, 0.01, 0.1)
        )
    }) ;
    // mix, invillupoo, spazializzazione
    Out.ar(0,
        Pan2.ar(
            (sig+del)*EnvGen.kr(Env.perc, doneAction:2),
            LFNoise1.kr(pan), 0.1
    ))
}).add;
)
s.scope ;
x = Synth(\bink ) ;
x = Synth(\bink , [\freq , 60.midicps]) ;
```

La synthDef è logicamente distinta in 3 blocchi. Nel primo un suono sorgente è definito come un'onda quadra "stretta", cioè con duty cycle ridotto (`width:0.1`), la cui frequenza di base `freq` è moltiplicata per un segnale a rampa che va da 1 (quindi, nessuna modifica) a un valore pseudo-casuale tra $[-0.5, 0]$, che esprime una "scordatura" in semitoni. Infatti, `midiratio` converte un valore in semitoni in un moltiplicatore per la frequenza: `0.midiratio` indica allora 0 semitoni e risulta in 1, `12.midiratio` indica 12 semitoni e risulta in 2^7. L'uso di `LFNoise1` assicura che la transizione tra valori generati sia continua. In sostanza,

[7] Per approssimazioni del formato float se si valuta il codice si ottiene, ma è del tutto normale, 1.9999999999945.

il risultato è un glissando a scendere dalla frequenza `freq`, nell'escursione massima di un quarto di tono (mezzo semitono). Il secondo blocco (da 11) invece utilizza la UGen `Delay` per produrre copie ritardate del segnale attraverso l'argomento `delaytime` che stabilisce di quanto il segnale debba essere ritardato. Il ritardato è di nuovo gestito pseudo-casualmente da `LFNoise1` che oscilla tra 10 e 100 ms. Il ritardo è però inserito all'interno di `Mix.fill`, che dunque produce 20 copie con ritardi pseudo-casuali e le miscela. Nel terzo blocco (da 19), la copia non ritardata è unita (di fatto, è un mix) con quella ritardata e inviluppata da un inviluppo percussivo (è un synth di tipo "nota"), e il tutto è distribuito sul fronte stereo con un posizionamento pseudo-casuale. Dal punto di vista sonoro, si ottiene una sorta di suono di corda metallica pizzicata, dove il metallico è dato dallo spettro ricco dell'onda quadra, il pizzicato dall'inviluppo percussivo, e l'effetto di simulazione (vagamente) acustica dalla presenza dei ritardi, che in qualche modo simulano ciò che avviene nella cassa di risonanza, che si comporta come una sorta di piccola camera di eco. Il codice seguente definisce invece un insieme di processi che sfruttano la synthDef.

```
(
var arr, window ;
arr = Array.fill(8, { arg i ;
    Task({
        var pitch ;
        inf.do({
            pitch = (i*[3, 5, 7].choose+40)%80+20 ;
            Synth(\bink , [\freq , pitch.midicps,
                \pan , pitch/50 ]) ;
            ((9-i)/8).wait ;
        })
    })
}) ;

arr.do({ |t| t.play}) ; // play all

// GUI per i task
window = Window("control", Rect(100, 100, 8*50, 50)).front ;
8.do {|i|
    Button(window, Rect(i*50, 0, 50, 50))
    .states_([[i.asString, Color.red],[i.asString, Color.black]])
    .action_{|me| if(me.value == 1) {arr[i].pause}{arr[i].play}}

}
)
```

Alla variabile `arr` è assegnato un array che contiene 8 task. Ognuno di essi permette di generare una voce in cui le altezze dipendono dal contatore (in sostanza le voci tendono a differenziarsi per registro). Il task sceglie un valore tra $[3, 5, 7]$, lo moltiplica per l'indice (il registro della voce, per così dire). Il punto di partenza è la nota 40, e la progressione viene bloccata al valore 80. La sequenza delle altezze che risultano in ogni strato è complessa perché dipende dalle relazioni tra il contatore `i`, l'array `[3,5,7]` e l'operatore modulo 20. Ogni strato ha un suo tasso di sviluppo autonomo: il più grave si muove più lentamente, il più acuto più velocemente, nell'intervallo di $[0.25, 1.125]$ secondi (rispettivamente con `i = 7` e `i = 0`). Si noti che si tratta di una quantizzazione al sedicesimo con tempo pari a 60. La progressione procede inesorabilmente (15) in parallelo. Il blocco seguente (17-24) permette di esplorare interattivamente gli strati e di variarne la fase relativa. Una finestra contiene 8 pulsanti a due stati. Molto

semplicemente, premendoli si dis/attiva il task relativo. Diventa così possibile ascoltare i singoli strati e variarne la sincronizzazione.

7.8 Pattern

È abbastanza chiaro che con routine e task si ottiene una grande duttilità nel controllo del tempo. SuperCollider mette a disposizione anche alcuni tipi di strutture dati che sebbene non necessariamente relativi allo scheduling, pure trovano la loro applicazione più tipica nella gestione di eventi nel tempo, i Pattern. Si tratta di un tipo di organizzazione delle informazioni che in realtà è stato già discusso nel contesto delle UGen Demand. In effetti, queste ultime implementano a lato server i Pattern a lato client, che in realtà sono molto più utilizzati.

Si può riprendere esattamente quanto già discusso in precedenza. Un pattern è la forma descritta proceduralmente di una sequenza di dati. Non è la sequenza in quanto tale a essere descritta, ma una regola per la sua produzione. Ad esempio, `Pseq` descrive sequenze che sono ottenute ripetendo una sequenza di partenza: così, `Pseq([1,2,3,4], 3)` descrive una sequenza ottenuta ripetendo tre volte i quattro valori contenuti in `[1,2,3,4]`. Il codice seguente:

```
1 p = Pseq([1,2,3,4], 3).asStream ;
2 13.do{ "il prossimo...? -> ".post; p.next.postln } ;
```

una volta valutato, stampa sulla post window:

```
il prossimo...? -> 1
il prossimo...? -> 2
il prossimo...? -> 3
il prossimo...? -> 4
il prossimo...? -> 1
il prossimo...? -> 2
il prossimo...? -> 3
il prossimo...? -> 4
il prossimo...? -> 1
il prossimo...? -> 2
il prossimo...? -> 3
il prossimo...? -> 4
il prossimo...? -> nil
13
```

Il codice precedente associa alla variabile `p` un Pattern `Pseq`, ma non solo: attraverso il messaggio `asStream` produce la sequenza effettiva. I altri termini, `Pseq` è una forma che, per poter essere utilizzata, deve prima essere convertita in un flusso (`stream`) di dati. Si noti che propriamente il messaggio `asStream` restituisce una routine. Infatti, è sulla routine che si invoca il metodo `next`, per 13 volte, così da accedere al valore successivo: quando (alla tredicesima volta) la sequenza è finita, `next` restituisce `nil`.

La prossima synthDef ottiene il segnale `sig` sommando 10 sinusoidi la cui frequenza varia di un massimo del 2.5% (3): ne risulta una sorta di effetto chorus, leggermente in vibrato. Una versione del segnale inviluppata percussivamente viene riverberata (4). Per lasciare che il riverbero si sviluppi in tutta la sua lunghezza, viene utilizzata la UGen `DetectSilence` (5). Questa UGen analizza il segnale in ingresso e quando l'ampiezza di quest'ultimo scende al di sotto di una soglia, esegue `doneAction`. La UGen restituisce un segnale binario, 0 o 1 in funzione del superamento della soglia, che può essere usato come trigger. Nel caso in questione, questa opzione non serve, ma semplicemente interessa che, verificato che il segnale si sia estinto, venga deallocato in automatico attraverso `doneAction:2`. È un modo per avere un riverbero completo senza utilizzare un bus. Il segnale `sig` è spazializzato come al solito su due canali (6).

```
SynthDef(\sinPerc , { |freq = 440, pos = 0, level = 0.125, detune = 0.025|
    var sig =
    Mix.fill(10, {SinOsc.ar(freq+Rand(0, freq*detune))}) ;
    sig = FreeVerb.ar(sig* EnvGen.kr(Env.perc)) ;
    DetectSilence.ar(sig, -96.dbamp, doneAction:2) ;
    Out.ar(0, Pan2.ar(sig, pos, level))
}).add ;
```

Il prossimo esempio include tre processi che sfruttano il pattern `Pseq`. Nel primo, una sequenza di 10 altezze è codificata intervallarmente (in semitoni) da una base ($= 0$). La routine (4-9) genera un insieme di note ogni $\frac{1}{4}$ di secondo, a partire da `p`, aggiungendo 70 e convertendo in frequenze. La seconda (12-22) utilizza la stessa sequenza, ma ne definisce un'altra per le durate, espresse in unità (15) e scalate in seguito (19). Poiché la sequenza delle altezze ha dimensione diversa da quella delle durate (rispettivamente, 10 contro 9), si verifica una situazione classica, quella del sistema di sfasatura tra *talea* e *color*, che rientreranno in fase dopo $10 \times 9 = 90$ eventi. Il terzo caso è molto più complesso a vedersi, ma semplicemente estende il principio della asimmetria *talea*/*color* ad altre dimensioni. In particolare, viene definito un pattern che controlla la densità, `~density`: un evento può allora essere composto da 1 fino a 4 synth paralleli (da una nota a un tetracordo). Ogni nota è allora armonizzata in funzione del pattern `~interval` che definisce un intervallo a partire dall'altezza di partenza. Infine, a questa determinazione dell'altezza va aggiunto il valore risultante da `~octave`, che sposta d'ottava il valore. Si noti anche che l'ampiezza è scalata in funzione della densità, così che accordi di diversa densità abbiamo la stessa ampiezza.

```
(
// 1. una sequenza
p = Pseq([0,2,4,3,10,13,12,6,5,7], inf).asStream ;
{
    inf.do{
        Synth(\sinPerc , [\freq , (p.next+70).midicps]) ;
        0.25.wait ;
    }
}.fork
)

(
// 2. color vs. talea (10 contra 9)
p = Pseq([0,2,4,3,10,13,12,6,5,7], inf).asStream ;
q = Pseq([1,1,2,1,4,1,3,3,1], inf).asStream ;
{
    inf.do{
        Synth(\sinPerc , [\freq , (p.next+70).midicps]) ;
        (q.next*0.125).wait ;
    }
}.fork
)

(
// 3. color vs. talea (10 contra 9)
// vs. chord dimension vs. interval vs. octave...
p = Pseq([0,2,4,3,10,13,12,6,5,7], inf).asStream ;
q = Pseq([1,1,2,1,4,1,3,3,1], inf).asStream ;
~density =  Pseq([1,2,4,1,3], inf).asStream ;
~interval =  Pseq([3,4,7,6], inf).asStream ;
~octave =  Pseq([-1,0,0,1], inf).asStream ;
{
    inf.do{
        var den = ~density.next ;
        den.do{
            var delta = ~interval.next ;
            var oct = ~octave.next ;
            Synth(\sinPerc ,
                [
                    \freq , (p.next+70+delta+(12*oct)).midicps,
                    \level , 0.1/den
        ]) } ;
        (q.next*0.125).wait ;
    }
}.fork
)
```

Una struttura canonica è invece alla base del prossimo programma, che definisce una sequenza di altezze, di durate e di posizioni sul fronte stereo (2-3). Quindi costruisce 3 array di 4 routine, secondo quanto previsto da `Pseq().asStream`. Dunque, ogni array contiene i dati di controllo complessivi per una sequenza musicale. Il ciclo successivo fa partire in parallelo 4 routine. Ognuna di esse fa riferimento attraverso il contatore `i` al relativo elemento nelle routine. In più, un moltiplicatore per le altezze e le durate è calcolato a partire da `i`, ed in questo modo si genera un canone a quattro voci su ottave diverse.

```
(
var mel = [0,3,5,6,7,9,10], rhy = [1,1,3,2,1,2] ;
var pan = Array.series(5, -1, 2/4) ;
var arrPseq = Array.fill(4, { Pseq(mel, inf).asStream } ) ;
var durPseq = Array.fill(4, { Pseq(rhy, inf).asStream } ) ;
var panPseq = Array.fill(4, { Pseq(pan, inf).asStream } ) ;

4.do{|i|
    { inf.do{
        var freqSeq = arrPseq[i] ;
        var freq = (12*i+freqSeq.next+48).midicps ;
        var durSeq = durPseq[i] ;
        var dur = durSeq.next*0.125*(i+1) ;
        var pan = panPseq[i].next ;
        Synth(\sinPerc , [\freq , freq, \pos , pan, \level , 0.07]) ;
        dur.wait ;
    }
    }.fork
}
)
```

Ancora due esempi di composizione algoritmica. Il prossimo utilizza tre synthDef:

```
SynthDef(\sinPerc , {
    |out = 0, freq = 440, pos = 0, level = 0.125, detune = 0.025|
    var sig =
    Mix.fill(10, {SinOsc.ar(freq+Rand(0, freq*detune))}) ;
    sig = FreeVerb.ar(sig* EnvGen.kr(Env.perc)) ;
    DetectSilence.ar(sig, -96.dbamp, doneAction:2) ;
    Out.ar(out, Pan2.ar(sig, pos, level))
}).add ;

SynthDef(\impPerc , {
    |out = 0, freq = 440, pos = 0, level = 0.125, detune = 0.025|
    var sig =
    Mix.fill(10, {Impulse.ar(freq+Rand(0, freq*detune))}) ;
    sig = FreeVerb.ar(sig* EnvGen.kr(Env.perc)) ;
    DetectSilence.ar(sig, -96.dbamp, doneAction:2) ;
    Out.ar(out, Pan2.ar(sig, pos, level))
}).add ;

SynthDef(\pulsePerc , {
    |out = 0, freq = 440, pos = 0, level = 0.125, detune = 0.025|
    var sig =
    Mix.fill(10, {Pulse.ar(freq+Rand(0, freq*detune), width:0.1)}) ;
    sig = FreeVerb.ar(sig* EnvGen.kr(Env.perc)) ;
    DetectSilence.ar(sig, -96.dbamp, doneAction:2) ;
    Out.ar(out, Pan2.ar(sig, pos, level))
}).add ;
```

che vengono utilizzate nel seguente processo:

```
(
~bs = Bus.audio(s, 2) ;

a = Pseq([1,3,4,1,1], 2) ;
b = Pseq([1,1,1]/3, 1) ;
c = Pseq([1,1,2]/2, 2) ;
d = Pseq([-1], 1);
e = Prand([5,0], 3);

f = Pseq([0,0, 1,0,0,1,2, 2,0,1,2,0,1,2], inf).asStream ;

p = Pxrand([a,b,c, d,e], inf).asStream ;

{
    inf.do{
        var which, id ;
        n = p.next ;
        if (n == -1) {
            [20,40].do{|i|
                Synth(\sinPerc ,
                    [\freq , i.midicps, \detune , 0.0125, \level , 0.2,
                        \out , ~bs]);
            } ;
            n = 1/8 ;
        } {
            id = f.next ;
            which = [\sinPerc , \impPerc ,\pulsePerc ][id] ;
            Synth(which,
                [\freq , (n*3+70).midicps,
                    \detune , 0.05, \out , ~bs, \pos , id-1*0.75]);
            Synth(which, [\freq , (n*3+46).midicps,
                \detune , 0.025, \out , ~bs, \pos , id-1*0.75]);
        } ;
        (n*0.25*60/84).wait
    }
}.fork ;

x = {|vol = 0.5| Out.ar(0, In.ar(~bs, 2)*vol)}.play(addAction:\addToTail ) ;
s.scope ;
x.set(\vol , 0.15)
)
```

Al lettore i dettagli. Vale la pena sottolineare tre aspetti. In primo luogo, la presenza di un condizionale all'interno della routine che discrimina in base

al valore di `p.next`. Se questo è pari a -1 allora il primo blocco viene eseguito che impone anche il valore di `n`, sovrascrivendo quello inizialmente assegnato attraverso `p.next`. In secondo luogo, si noti nel secondo blocco che la synthDef viene scelta anch'essa in funzione di un pattern (`f`, per la precisione). Infine, per controllare l'ampiezza del segnale complessivo in maniera efficace ed efficiente tutti i segnali sono scritti sul bus `~bs` da cui legge il synth `x` il quale ha l'unica funzione di scalare il segnale per poi mandarlo in uscita.

Infine, l'esempio seguente sfrutta la synthDef `sinPerc` per produrre un insieme di arpeggi che ricordano vagamente un piano elettrico o una celesta. Il primo blocco definisce le strutture dati di controllo. In particolare, l'array `~arr` contiene un insieme di pattern che intendono gestire l'armonizzazione (cioè le altezze che compongono l'arpeggio). L'array `p` contiene invece un'altezza di base che verrà sommata a 60 (21). L'array `q` invece contiene i valori che definiscono la rapidità dell'arpeggio, mentre il tempo che separa un arpeggio da un altro è gestito da `timeBase` (31). Si noti che quest'ultimo è correlato al parametro `oct` che definisce l'ottava di trasposizione dell'altezza di base attraverso il condizionale `case` (17-20). Arpeggi gravi suonano più lentamente di quelli acuti.

```
(
a = Pseq([3,4,4,3,3], inf) ;
b = Pseq([6,7,7], inf) ;
c = Pseq([9,10, 9, 9], inf) ;
d = Pseq([11,12,13,12,14], inf) ;
e = Pseq([5, 6,5]+12, inf) ;
f = Pseq([8,9,9,8,9]+12, inf) ;

~arr = [a,b,c,d,e,f].collect{|i| i.asStream} ;
p = Pseq([0,3,3,2,2, 3,0, 7, 6, 11,11, 10, 9, 12], inf).asStream ;
q = Prand([ 1/64, 1/32, 1/16], inf).asStream ;
r = Pseq(
    Array.fill(12, {0})++
    Array.fill(3, {-12})++
    Array.fill(8, {7}),
    inf).asStream ;
)
(
var base, harm, timeBase, oct ;
{
    inf.do{
        oct = r.next ;
        case
        {oct == 7} { timeBase = 1/16}
        {oct == -12} { timeBase = 2/3}
        {oct == 0} { timeBase = 1/4} ;
        base = p.next+60+oct ;
        Synth(\sinPerc ,
            [\freq , base.midicps, \detune , 0.01,\level , -30.dbamp]) ;
        6.do{|i|
            harm = base+~arr[i].next ;
            Synth(\sinPerc ,
                [\freq , harm.midicps, \detune , 0.01,
                    \level ,-30.dbamp, \pos , i.linlin(0,1.0, -0.5,0.5)]) ;
        q.next.wait;
        } ;
        timeBase.wait ;

}}.fork ;
)
```

I pattern definiti in SuperCollider sono moltissimi, non certo soltanto `Pseq` e `Pxrand`: ad esempio, alcuni servono anche per gestire l'audio e non solo il controllo degli eventi, mentre altri ancora permettono di filtrare il contenuto di

flussi di dati generati da altri pattern (i *filter pattern*). Da un certo punto di vista, l'insieme dei pattern costituisce una sorta di sottolinguaggio di SC dedicato specificamente alla rappresentazione di flussi di dati.

7.9 Eventi e Pattern di eventi

I pattern sono strutture dati che codificano in forma compatta sequenze di dati. Come si è visto negli esempi precedenti, possono essere utilizzati all'interno di processi di scheduling usuali, che utilizzino routine o task. Essi costituiscono però anche una componente fondamentale di una logica di sequencing molto peculiare, che si basa sul concetto di "evento". Ad esempio, quest'espressione misteriosa "suona":

```
1 ().play ;
```

La parentesi tonda è un'abbreviazione per create un'istanza della classe `Event`, e dunque l'espressione è un sinonimo di

```
1 Event.new.play ;
```

Un "evento" in SC è semplicemente un'associazione tra variabili ambientali e valori, che può rispondere al metodo `play`. Tecnicamente, è di fatto un dizionario che associa nomi a valori. La classe `Event` predefinisce molti di questi nomi così come funzioni che a questi nomi sono associate, così da specificarne la semantica. Se si valuta la riga precedente accadono infatti due cose, si sente un suono e si vede sulla post window apparire quanto segue:

```
( 'instrument': default, 'msgFunc': a Function, 'amp': 0.1, 'sustain': 0.8,
'server': localhost, 'isPlaying': true, 'freq': 261.6255653006,
'hasGate': true, 'detunedFreq': 261.6255653006, 'id': [ 1002 ] )
```

Si riconoscono nomi che concernono tipicamente la sintesi, associati a valori. SC definisce cioè un evento predefinito in cui un synth costruito dalla synthDef `\default` (variabile 'instrument')[8] suona a 261.6255653006 Hz (`'freq'`) con ampiezza 0.1 (`'amp'`), e altri parametri ancora. Questo evento viene realizzato quando si esegue `play`. La situazione diventa chiara osservando il prossimo esempio:

```
(\instrument : \sinPerc , \midinote : 65).play ;
```

Qui lo strumento è invece `\sinPerc` (la synthDef degli esempi precedenti) mentre la frequenza non è definita esplicitamente con un riferimento a `'freq'` ma a `'midinote'`: quest'ultima variabile è associata a una funzione che permette automaticamente a sua volta di definire il valore di `'freq'` (si noti la post window). Due considerazioni sono importanti:

1. `Event` predefinisce molte variabili, che ad esempio permettono di gestire in maniera molto semplice sistemi di altezze (scale di vario tipo, non necessariamente temperate) senza occuparsi del calcolo delle frequenze. Lo stesso vale per altri parametri. A tal proposito si può consultare l'help file relativo;
2. per utilizzare i valori delle variabili in relazione ad una synthDef, è necessario che questa synthDef li preveda come argomenti. Ad esempio, un evento è pensato sul modello della nota, dunque è necessario impiegare `doneAction:2`, altrimenti il synth generato non sarà deallocabile. Ancora, nomi di argomenti necessari (se utilizzati) sono `freq, amp, gate`.

[8] La synthDef `\default` è predefinita in SC e caricata sul server quando questo parte.

Nel prossimo esempio, una synthDef prevede `freq` ma non include `doneAction:2`. Il modello di altezze funziona (`freq` viene correttamente calcolato a partire dalla variabile `midinote`), senonché l'evento non finisce (perché ogni synth continua a essere presente).

```
SynthDef(\cont , {arg freq = 440; Out.ar(0, SinOsc.ar(freq))}).add ;

(\instrument : \cont , \midinote : 80).play ;
(\instrument : \cont , \midinote : 60).play ;
```

Il prossimo esempio avvicina il concetto di evento a quello di pattern.

```
(
SynthDef(\sinPerc , { |freq = 440, pos = 0, amp = 1, detune = 0.025|
    var sig =
    Mix.fill(10, {SinOsc.ar(freq+Rand(0, freq*detune))*0.1}) ;
    sig = FreeVerb.ar(sig* EnvGen.kr(Env.perc)) ;
    DetectSilence.ar(sig, -96.dbamp, doneAction:2) ;
    Out.ar(0, Pan2.ar(sig, pos, amp))
}).add ;
)

p = Pseq([0,2,5,7,9], inf).asStream ;

{
    inf.do{
        (\instrument :\sinPerc , \amp :0.5, \ctranspose :p.next).play ;
        0.25.wait ;
    }
}.fork ;
```

In primo luogo, la synthDef `'sinPerc'` è riscritta in maniera da adeguarsi al modello previsto da `Event`. L'ampiezza è normalizzata in modo che il mix al massimo risulti in un valore di picco $= 1$ (4), e invece dell'argomento `level` si introduce `amp`. Quindi, `p` è un pattern che rappresenta una melodia pentatonica (11). La routine successiva istanzia ogni 250 ms (16) un nuovo evento che utilizza `sinPerc`, ne definisce l'argomento `amp` e passa il valore successivo di `p` alla

variabile `\ctranspose`, ancora un altro modello per la definizione delle altezze, che aggiunge il valore a quello di una nota midi di base (di default $= 60$) e lo converte in frequenza.

È del tutto lecito utilizzare in questo modo il concetto di evento, ma l'uso più comune (ed in qualche modo la ratio alla base del concetto stesso) è legato all'uso di "event pattern". I pattern discussi finora sono usualmente descritti come pattern "di valori" o "basati su lista" (*value/list pattern*). Una ulteriore possibilità è data dai pattern di eventi. Un pattern di eventi, tipicamente `Pbind`, mette in relazione eventi con pattern di dati, e può essere eseguito. Ne consegue una notazione estremamente compatta ed elegante per la specifica di flussi di eventi sonori. Nell'esempio minimale che segue, prima si definisce un pattern di eventi `p` (come nell'esempio precedente); quindi un pattern di eventi che associa alla variabile `\ctranspose` il pattern `p` (3) e alle altre variabili i valori descritti nell'esempio precedente; infine si esegue `play` che restituisce un oggetto di tipo `EventStreamPlayer`, cioè letteralmente un esecutore di flussi di eventi, una sorta di lettore che genera il suono a partire dalle specifiche. Quest'ultimo, `f` (5), può essere controllato in tempo reale interattivamente (6-9). Si noti che è `EventStreamPlayer` che si occupa di generare stream a partire dai pattern.

```
1 p = Pseq([0,2,5,7,9], inf) ;

3 e = Pbind(\ctranspose ,p,\instrument ,\sinPerc ,\amp ,0.5,\dur ,0.25) ;

5 f = e.play ;
6 f.pause ;
7 f.play ;
8 f.mute ;
9 f.unmute ;
```

Il prossimo esempio introduce due pattern `p` e `d` per altezze e durate, secondo il modello *talea/color* (1-3). Una singola sequenza può essere ascoltata con il blocco 2 (1-5). Il blocco 3 costruisce una polifonia di 4 voci a diverse ottave e in cui l'altezza è proporzionale al moltiplicatore della dinamica (17-18). Si noti che la riga 21 riassegna ad `a` (che prima conteneva le istanze di `Pbind`)) gli `EvenStreamPlayer` relativi. Se il blocco 3 è ancora in esecuzione, si può valutare il blocco 4 che fa partire ogni secondo routine che silenziano e poi ripristinano gli strati. Infine il blocco 5 ferma la routine `r` e mette in pausa tutti gli `EventStreamPlayer`.

```
// 1. pattern
p = Pseq([0,2,3, 5, 6, 0,2,3, 5, 7, 8,0,7], inf);
d = Pseq([1,1,2,1,1,2,3,1,1,2], inf);

// 2. primo test, talea vs. color
(
Pbind(
    \instrument , \sinPerc , \amp , 0.25,
    \ctranspose , p, \dur , d*0.125).play ;
)

// 3. canone
(
a = Array.fill(4, {|i|
    Pbind(
        \instrument , \sinPerc , \amp , 0.25,
        \root , i*12-24,
        \ctranspose , p, \dur , d*0.125*(i+1))
}) ;

a = a.collect{|i| i.play} ; // a contiene ora i player
)

// 4. un processo che mette in pausa selettivamente
(
r = {
    inf.do{|i|
        {b = a[i%4].mute ;
            2.wait ;
            b.unmute ;}.fork;
        1.wait ;
    }
}.fork
)

// 5. tutti in pausa
r.stop; a = a.do{|i| i.pause} ;
```

Può non essere sempre intuitivo interagire con gli event pattern, perché in qualche modo le variabili ambientali che vengono organizzate negli eventi richiedono sempre di operare con una logica conforme a quella dei pattern. L'esempio seguente dimostra a tal fine l'uso di `Pfunc`, un pattern che ad ogni

chiamata restituisce il risultato di una funzione. In questo frangente, quest'ultima ha come scopo semplicemente di accedere al valore `i` dell'array `r`, operazione che permette di definire per ogni event pattern una diversa trasposizione. La riga 16 definisce allora una sequenza `[ -12, -5, 2, 9, 16 ]` mente la 17 ripristina la sequenza di partenza `[ 0, 0, 0, 0, 0 ]`, in cui le quattro voci suonano un canone all'unisono.

```
1  Pbind(\ctranspose , p, \stretch , 1/2, \dur , d, \root , 1).play ;

3  ~seq = [0,2,5,7,9] ;
4  p = Pseq(~seq, inf) ;
5  d = Pseq([1,1,2,1,1,3,1,4,1,1], inf) ;
6  r = Array.series(~seq.size, 0, 0) ;

8  (
9  8.do{|i|
10     Pbind(\ctranspose , p+Pfunc({r[i]}), \stretch , 1/( i+1), \dur , d,
11     ).play ;
12 } ;
13 )

15 // controllo
16 r = Array.series(~seq.size,-12,7) ;
17 r = Array.series(~seq.size,0,0) ;
```

L'ultimo esempio mette intende mettere in luce le potenzialità espressive che derivano dall'incassamento dei pattern. La synthDef è una versione leggermente modificata di una versione utilizzata in precedenza. In particolare, è stato aggiunto l'argomento `amp` e il panning non è più gestito da una UGen pseudo-random.

```
SynthDef("bink", { arg freq = 440, pan = 0, amp = 0.1;
    var sig, del;
    // sorgente
    sig = Pulse.ar(freq
        *Line.kr(1,
            LFNoise1.kr(0.1)
            .linlin(-1,1, -0.5, 0).midiratio, 0.1),
        width:0.1
    ) ;
    // coda di ritardi
    del = Mix.fill(20, {|i|
        DelayL.ar(sig,
            delaytime: LFNoise1.kr(0.1)
            .linlin(-1,1, 0.01, 0.1)
        )
    }) ;
    // mix, invillupoo, spazializzazione
    Out.ar(0,
        Pan2.ar(
            (sig+del)*EnvGen.kr(Env.perc, doneAction:2),
            pan, amp
    ))
}).add;
```

Il primo blocco (1-24) organizza le durate. L'uso estensivo e disordinato di variabili ambientali è stato lasciato come testimonianza "etnografica" del lavoro incrementale e interattivo di composizione. L'idea di base è di avere una prima sezione in due varianti, a cui segue una sezione di tipo "fill" che seleziona tra un insieme abbastanza esteso di possibilità ritmiche. Come si vede, le durate usano i numeri interi, di più semplice gestione. I due pattern `a` e `b` durano entrambi 24 unità, il primo alterna le durate 7 e 5. Le sezioni "fill" (7, 15) durano invece sempre 16 e organizzano pattern ritmicamente più densi (si notino le durate). Le righe 17 e 18 definiscono due pattern `Prand` che scelgono a caso una tra le frasi (`h`) e uno tra i fill (`i`). Quindi, il pattern `i` è invece una sequenza infinita di `h` e `i`, cioè di una frase e un fill. Il totale, un po' "sbilenco", è di 40 unità. Il basso è allora costituito da tre "battute" da 40 unità (`j`) e da una specifica sequenza di altezze `l`. Quest'ultimo pattern dimostra anche l'uso speciale di `\r` per indicare una pausa. I due `Pbind` utilizzano `\stretch` come indicazione di tempo (definita per via del tutto empirica). Il basso `~e2` procede in maniera ordinaria. La voce superiore `~e1` utilizza `Pkey`, un pattern speciale che permette

di accedere al valore di un'altra variabile all'interno di `Pbind`. È in questo modo che vengono definite le note midi suonate: attraverso una interpolazione lineare (un mapping) dalle durate alle altezze (per cui durate maggiori risultano in altezze più gravi e viceversa).

```
(
// organizzazione delle durate
// frase
a = Pseq([7,5], 2) ;
b = Pseq([8], 3) ;
// fills
c = Pseq([4], 4) ;
d = Pseq([16/3], 3) ;
e = Pseq([1,2,3,4,6], 1) ;
f = Pseq([8/3, 16/3], 2) ;
g = Pseq([6,5,4,1], 1) ;
y = Pseq([5,5,1,5], 1) ;
z = Pseq([4,3,2,5,2], 1) ;
w = Pseq([30/5, 25/5, 15/5, 10/5], 1) ;
u = Pseq([30/5, 25/5, 15/5, 10/5].reverse, 1) ;
// primo incassamento
h = Prand([a, b],1) ;
i = Prand([c,d,e, f,g, y,z,w,u], 1) ;
// secondo
k = Pseq([h, i], inf) ;
// basso
j = Pseq([10, 20, 10, 40, 40/3, 40/3, 40/3], inf) ;
l = Pseq([
    -24, -24, -12, -24, \r , -26, -25,
    -24, -14, -12, -24, -13, \r , -25,
], inf) ;
)

(
// due voci in parallelo
~e1 = Pbind(\instrument , \bink , \stretch , 60/32/60, \dur , k.next,
    \ctranspose , Pkey(\dur ).linlin(0, 12, 12, 0.0), \pan , rrand(-0.5, 0.5)
).play ;

~e2 = Pbind(\instrument , \bink , \ctranspose , l.next, \amp , 0.1,
    \stretch , 60/32/60, \dur , j.next).play
)
// stop
[~e1, ~e2].do{|i| i.pause} ;
```

7.10 Conclusioni

Il tempo è ovviamente un aspetto fondamentale nella gestione del suono e il capitolo ha allora tentato di fornire alcune prime indicazioni sulle molte possibilità (concettuali e operative) che SuperCollider mette a disposizione. Questa ricchezza espressiva rende SC adatto a molte situazioni diverse, che beneficiano di modalità specifiche di concettualizzazione. Tuttavia, quanto discusso è soltanto la parte emersa di un iceberg molto profondo che il lettore è invitato a esplorare attraverso gli help files. L'interazione costante tra programmazione e risultato sonoro che SC offre rende il compito estremamente stimolante.

8 Sintesi, II: introduzione alle tecniche di base in tempo reale

Finora la sintesi del segnale audio è stata introdotta di soppiatto, in relazione all'acustica e ai segnali di controllo. Il prossimo capitolo intende invece fornire una introduzione di base alle tecniche di sintesi del segnale audio, nella prospettiva della loro implementazione in SuperCollider. Il capitolo seguirà da vicino la discussione proposta in *Audio e multimedia*, a cui si rimanda il lettore interessato. Nel testo si farà ampio uso delle notazioni abbreviate del tipo `func.play` (o `scope`, `freqscope`, `plot`), che si rivelano particolarmente compatte proprio nel caso di sperimentazione con i grafi di UGen per l'elaborazione e sintesi del segnale, e che tra l'altro sono abbondantemente in uso negli help files per le UGen. Le synthDef introdotte non saranno particolarmente complesse, proprio perché la prospettiva che interessa riguarda le tecniche in se stesse, non la loro implementazione ottimale.

8.1 Oscillatori e tabelle

Un algoritmo per la sintesi del suono è una procedura formalizzata che ha come scopo la generazione della rappresentazione numerica di un segnale audio. Introducendo gli algoritmi di sintesi del segnale non in tempo reale, più volte si è discusso della generazione del segnale a partire dal calcolo di una

funzione, tipicamente $sin(x)$. Sebbene concettualmente elegante, il computo diretto della funzione si rivela assai poco efficiente dal punto di vista computazionale: costringe infatti l'elaboratore a calcolare il valore della funzione per un numero di volte al secondo pari al tasso di campionamento prescelto (cioè 44.100 nel caso di qualità CD, il tasso di campionamento predefinito in SC). Diventa immediatamente chiaro come la scelta di specifiche strategie algoritmiche sia perciò un elemento chiave negli approcci alla sintesi del segnale. Così, rispetto al problema precedente, è possibile impiegare un altro metodo, che ha una lunga tradizione in computer music: si tratta cioè di costruire un *oscillatore digitale*. L'oscillatore digitale è un algoritmo fondamentale nella computer music, poiché non è soltanto impiegato per generare direttamente un segnale ma è anche un'unità componente di molti altri generatori di suono. Tornando al problema della generazione di una sinusoide, e ragionando diversamente, si può osservare come un suono sinusoidale (ma in generale la parte stazionaria di ogni suono periodico) sia caratterizzato da una grande prevedibilità: esso si ripete uguale ad ogni periodo. Dato un periodo del segnale, si può allora pensare di prelevarne il valore in n punti equidistanti. Questi valori campionati possono essere immessi in una tabella di n punti: alla lista dei punti corrisponde un'altra lista con i valori della funzione. Una tabella di questo tipo prende il nome di *wavetable* (tabella d'onda o forma d'onda tabulata). Ad esempio, si consideri il seguente codice, che usa un array per calcolare una sinusoide campionata in 16 punti di frequenza pari a 2π:

```
1 // un array di 16 punti, freq = 1 in 2pi
2 ~sig = Array.fill(16, {|i| sin(i/16*2pi)});

4 // stampare la tabella
5 ~sig.do{|i,j| ("["++j++"] = "++i).postln } ;
```

La tabella è la seguente:

```
[0] = 0
[1] = 0.38268343236509
[2] = 0.70710678118655
[3] = 0.92387953251129
[4] = 1
[5] = 0.92387953251129
[6] = 0.70710678118655
[7] = 0.38268343236509
[8] = 1.2246467991474e-16
[9] = -0.38268343236509
[10] = -0.70710678118655
[11] = -0.92387953251129
[12] = -1
[13] = -0.92387953251129
[14] = -0.70710678118655
[15] = -0.38268343236509
```

Il segnale ottenuto invece è disegnato in Figura **8.1**.

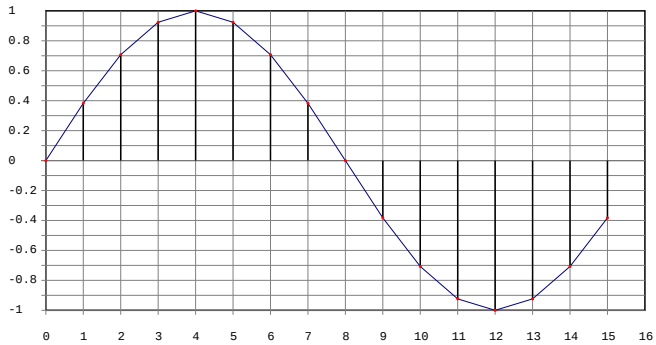


Fig. 8.1 Forma d'onda campionata

Se si assume che una simile struttura dati rappresenti una tabella, l'algoritmo dell'oscillatore digitale prevede fondamentalmente il semplice svolgimento di due operazioni:

1. leggi i valori della tabella dall'area di memoria in cui questa è conservata;

2. arrivato all'ultimo indirizzo (il 15 nell'esempio), riparti dal primo (0). Quest'ultima operazione si definisce *wrappping around*.

Il metodo di sintesi descritto (*Table Look-Up Synthesis*) è estremamente efficiente: la lettura di un dato dalla memoria è infatti un processo assai più veloce che non il calcolo del valore di una funzione. È agevole osservare come una simile operazione di stoccaggio di una "forma" e suo successivo reperimento sia tra l'altro lo stesso approccio preso in considerazione nel momento in cui si è discusso di una generalizzazione degli inviluppi. La tabella costituisce infatti un modello (statico e caricato in fase di inizializzazione) della forma d'onda: sta all'utente decidere ampiezza e frequenza del segnale sintetizzato dall'oscillatore. Per quanto concerne l'ampiezza, l'operazione è ormai ovvia. Ad esempio, se il segnale tabulato ha ampiezza compresa in $[-1, 1]$, moltiplicando nell'oscillatore tutti i valori della tabella per 0.5, il segnale in uscita avrà un'ampiezza dimezzata, che oscillerà nell'escursione $[-0.5, +0.5]$.

```
1 // un array di 16 punti, freq = 1 in 2pi, amp = 0.5
2 ~sig = Array.fill(16, {|i| sin(i/16*2pi)*0.5});
```

Come ormai noto, moltiplicare un segnale per un numero k produce come risultato un segnale in cui ognuno dei valori risulta moltiplicato per k. Quindi, ognuno dei valori del segnale in entrata (la forma d'onda tabulata) viene scalato per 0.5 (Figura **8.2**).

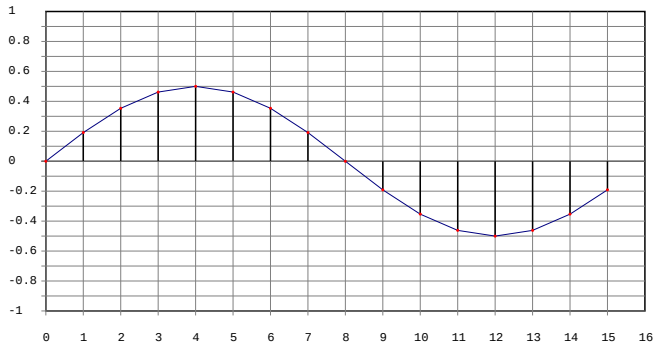


Fig. 8.2 Scalatura dell'ampiezza

Passando all'implementazione in SC, si noti che, se il problema fosse la generazione di una sinusoide, in SC sarebbe sufficiente utilizzare la UGen specializzata `SinOsc` che implementa l'algoritmo di table look-up, ma non richiede di specificare una tabella poiché la prevede internamente (di 8192 punti)[1]. Tuttavia l'utilizzo di tabelle non si limita evidentemente alla tabulazione di una sinusoide, ma è molto più generale, e per questo vale perciò la pena di soffermacisivi. Una delle UGen che si occupano esplicitamente di leggere da tabelle è `Osc`, il cui primo argomento è una tabella, la cui dimensione deve essere un multiplo di 2 (per ragioni di ottimizzazione), e il secondo la frequenza alla quale l'oscillatore legge la tabella. Se la tabella rappresenta il periodo di un segnale, la frequenza indicherà la frequenza del segnale risultante. La tabella può essere fornita a `Osc` attraverso un buffer. Come si è visto discutendo il server, un buffer è una locazione di memoria temporanea, allocata dal server nella RAM, in cui possono essere memorizzati dati: ogni buffer (come ogni bus) è indirizzato attraverso un numero intero univoco attraverso il quale vi si può far riferimento. SC implementa una classe `Buffer` che si occupa della gestione dei buffer. In altre parole, con `Buffer` si dice al server di riservare un certo spazio per certi dati. La classe `Buffer` prevede alcuni metodi per allocare buffer, per leggervi dentro un file audio, ma anche per riempire direttamente un buffer con specifici tipi di segnali, di larga utilità.

```
(
var buf = Buffer.alloc(s, 2.pow(10)) ;
// 1 componente di ampiezza normalizzata
buf.sine1([1]) ;
{Osc.ar(buf, 440)}.play ;
)

(
var buf = Buffer.alloc(s, 2.pow(10)) ;
// 20 componenti di ampiezza casuale
buf.sine1( Array.fill(20, { 1.0.rand }) ) ;
{Osc.ar(buf, 440, mul: 0.6)}.play
)
```

[1] `FSinOsc` invece implementa un approccio diverso al problema della generazione di una sinusoide, in modo molto efficiente (di qui la `F` di `Fast`) seppur con qualche limitazione d'uso.

Il codice precedente contiene due esempi. Nel primo alla variabile `buf` viene assegnato un oggetto `Buffer` ottenuto chiamando sulla classe il costruttore `alloc`. Il metodo dice al server `s`[2] di allocare un buffer di una certa dimensione. La dimensione è di per sé arbitraria: tuttavia nell'implementazione degli oscillatori (e non solo in SC) viene tipicamente richiesto per questioni di ottimizzazione che la dimensione della tabella sia una potenza di 2 (`2.pow(10)`). Quindi viene inviato a `buf` il messaggio `sine1` che riempie il buffer con una somma di sinusoidi armonicamente relate. La sintassi è analoga al metodo `waveFill` della classe `Array`: l'array contiene le ampiezze delle armoniche. Nel primo esempio c'è solo una sinusoide (la fondamentale) con ampiezza unitaria. `Osc` richiede come primo argomento il buffer da leggere (la tabella, appunto). La tabella memorizzata nel buffer viene quindi letta con una frequenza pari a 440 volte al secondo. Nel secondo esempio, l'array di ampiezze è generato attraverso il metodo `fill` invocato su `Array`, che restituisce un array di 20 numeri pseudo-casuali compresi in $[0.0, 1.0]$: essi rappresentano le ampiezze stocastiche delle prime 20 armoniche.

È altresì possibile fare in modo che `Osc` legga da una tabella che contiene un segnale audio residente sul disco rigido. Nell'esempio seguente viene dapprima generato un segnale attraverso la classe `Signal`, a cui viene inviato il messaggio `sineFill`, che genera la consueta somma di sinusoidi: si noti che la dimensione dell'array (primo argomento di `sineFill`) è immediatamente stabilita in una potenza di 2, ovvero $2^{16} = 65536$. Inoltre, per poter essere letto da `Osc`, il segnale deve essere convertito in un formato apposito, quale quello previsto dalla classe `Wavetable`: di qui la conversione `sig = sig.asWavetable`. La synthDef seguente semplicemente ingloba in una UGen `Out` una UGen `Osc`, prevedendo come argomento `buf`. Quindi la synthDef viene spedita al server.

Infine, dopo aver memorizzato il segnale sotto forma di file audio (10-14), è possibile caricarlo in un buffer: la riga 28 utilizza il metodo `read` che alloca un buffer sul server `s` e vi legge il file indicato dal percorso. Il buffer viene assegnato alla variabile `buf`. La dimensione del buffer (quanta memoria occupa) è inferita direttamente da SC a partire dalla dimensione del file. Così, il nuovo `Synth` che viene generato (31) utilizza il buffer `buf`.

[2] Si ricordi che alla variabile globale `s` è riservato per default il server audio.

```
(
var sig ;
var soundFile ;
//--> generazione di un segnale
sig = Signal.sineFill(2.pow(16), [1]) ; // 65536
sig = sig.asWavetable ; // obbligatorio!
//--> scrittura del file audio
soundFile = SoundFile.new ;
soundFile.headerFormat_("AIFF").sampleFormat_("int16").numChannels_(1) ;
soundFile.openWrite("/Users/andrea/musica/signalTest.aiff") ;
soundFile.writeData(sig) ;
soundFile.close ;
)

(
//--> synthDef per la lettura
SynthDef("tableOsc",{ arg buf, freq = 440, amp = 0.4 ;
        Out.ar(0,
            Osc.ar(buf, freq, mul: amp))
    }).add ;
)

(
var freq = 440 ;
var buf, aSynth;

//--> allocazione di un buffer e immediato riempimento dal file
buf = Buffer.read(s, "/Users/andrea/musica/signalTest.aiff") ;

//--> lettura dal buffer
aSynth = Synth.new("tableOsc", ["buf", buf]) ;
)
```

La caratteristica precipua dell'oscillatore è soltanto quella di generare campioni sulla base della lettura della tabella: la forma d'onda tabulata non dev'essere necessariamente sinusoidale. È possible infatti tabulare qualsiasi forma, come pure importare una forma d'onda da un qualsiasi segnale preesistente.

Nell'esempio seguente la tabella è riempita con valori pseudo-casuali: in particolare la riga 11 sceglie per ogni elemento dell'array (di tipo `Signal`) un valore pseudo-casuale tra $[0.0, 2.0] - 1 = [-1.0, 1.0]$: il risultato viene quindi

convertito in una `Wavetable`. Il resto del codice assume la synthDef precedente (`tableOsc`) e riprende quanto già visto nel caso della generazione di una sinusoide. Una variabile di rilievo è `exp` (10), l'esponente di `2` che determina la dimensione della tabella (riga 11). Un oscillatore che non si arresti dopo una lettura della tabella (nel qual caso il concetto di frequenza di lettura non avrebbe molto senso) produce per definizione un segnale periodico: infatti reitera la lettura della tabella e conseguentemente il segnale si ripete uguale a se stesso. Più la tabella è grande, maggiore è l'informazione sulla sua "forma" (il dettaglio temporale) e più il segnale è rumoroso. Viceversa, tabelle di dimensioni ridotte producono segnali il cui profilo temporale è più semplice (meno valori pseudo-casuali), e dunque maggiormente "intonati". Si provi ad esempio a variare `exp` nell'escursione $[1, 20]$, e a visualizzare il contenuto del buffer così ottenuto con `plot`. Le righe 23 e 24 permettono altresì di variare la frequenza di lettura della tabella, che a periodi più lunghi rivela maggiormente il suo profilo rumoroso.

```
1  /*
2      Oscillatore periodico-aperiodico:
3      legge da una tabella di valori pseudo-casuali
4  */
5
6  (
7  var sig, exp ;
8  var soundFile;
9  // generazione della tabella
10 exp = 6 ;        // provare nell'escursione [1, 20]
11 sig = Signal.fill(2.pow(exp), {2.0.rand-1}).asWavetable ;
12 sig.plot ; // visualizzazione della tabella
13 //--> scrittura del file
14 soundFile = SoundFile.new ;
15 soundFile.headerFormat_("AIFF").sampleFormat_("int16").numChannels_(1) ;
16 soundFile.openWrite("/Users/andrea/musica/signalTest.aiff") ;
17 soundFile.writeData(sig) ;
18 soundFile.close ;
19 )
20
21 ~buf = Buffer.read(s, "/Users/andrea/musica/signalTest.aiff") ;
22 ~aSynth = Synth.new(\tableOsc , [\buf , ~buf, "amp", 0.1]) ;
23 ~aSynth.set(\freq , 10) ;
24 ~aSynth.set(\freq , 1) ;
```

Una modulazione della frequenza dell'oscillatore indica allora una variazione della velocità di lettura della tabella. La synthDef seguente prevede una variazione pseudo-casuale della frequenza con `LFNoise0`, secondo una modalità già abbondantemente descritta.

```
// modulando la freq
SynthDef(\tableOsc ,{ arg buf = 0, freq = 440, amp = 0.4 ;
    Out.ar(0, Osc.ar(buf,
        LFNoise0.ar(10, mul: 400, add: 400), mul: amp))
}).add ;
```

8.1.1 Sintesi per campionamento

Il metodo di sintesi concettualmente più semplice è il campionamento. Fondamentalmente, con l'espressione "prelevare un campione"[3] si indica l'ottenimento di un segnale di breve durata o attraverso la registrazione diretta, oppure attraverso un'operazione di *editing* da un file audio. In ogni caso, nella sintesi per campionamento l'idea centrale è quella del playback di materiale sonoro preesistente.

Nel campionamento semplice si ripropone l'idea incontrata discutendo dell'oscillatore: la lettura di una tabella. Se però in quel caso la forma d'onda, oltre a essere estremamente semplice (ma sarebbe stato possibile costruire una forma qualsiasi), era tipicamente ridotta a un unico periodo, nel campionamento invece la forma d'onda tabulata rappresenta un suono complesso dotato di inviluppo, la cui durata dipende soltanto dalle limitazioni imposte dall'hardware. Come intuibile, l'origine del campione (per registrazione diretta o per estrazione da un file preesistente) non fa alcuna differenza. Una volta messo in memoria, il campione può essere richiamato ogni qualvolta l'utente lo desideri attraverso un dispositivo di controllo. Nonostante la semplicità concettuale, è evidente la

[3] Si noti che qui la semantica di campione è diversa da quella considerata finora, in riferimento cioè al tasso di campionamento.

potenza di un simile approccio, poiché si può avere agevolmente a disposizione una ricca tavolozza di suoni pre-registrati: ad esempio un intero set di percussioni, così come molte altre librerie di campioni[4].
Sebbene possa essere senz'altro possibile utilizzare un oscillatore come `Osc` per leggere un file audio preventivamente caricato in un buffer, la UGen specializzata nella lettura di un buffer è `PlayBuf`. Essa non incorre nel limite di avere una tabella la cui dimensione sia multiplo di 2, ma può invece leggere un buffer di dimensioni qualsiasi.

```
(
SynthDef(\playBuf , { arg buf, loop = 0 ;
    Out.ar(0, PlayBuf.ar(1, buf, loop: loop) )
}).add ;
)
(
var buf, aSynth ;
buf = Buffer.read(s, Platform.resourceDir +/+ "sounds/a11wlk01.wav") ;
aSynth = Synth(\playBuf , [\buf , buf, "loop", -1]) ;
)
```

Nell'esempio precedente la synthDef "avvolge" semplicemente `PlayBuf` di cui dimostra tre degli argomenti disponibili, i primi due e l'ultimo.

- Il primo argomento indica il numero dei canali (tipicamente, mono o stereo): se il suo valore è 1, allora il buffer è mono. Si noti che indicazioni discordanti tra numero indicato e numero reale possono portare ad un mancato funzionamento ma senza che l'interprete lo segnali;
- il secondo specifica il buffer, qui controllabile come argomento `buf` della synthDef ;
- l'argomento `loop` indica la modalità di lettura e può avere due valori: 0 indica un'unica lettura del buffer, 1 una lettura ciclica.

[4] Non è questo il luogo per una sitografia, ma vale la pena di segnalare una risorsa di grande interesse, *The Freesound Project*, `http://freesound.iua.upf.edu/` che mette a disposizione con licenza Creative Commons decine di migliaia di file audio, tutti indicizzati, scaricabili o ascoltabili *on-line*.

Nel seguito viene allocato un buffer `buf` che legge dal disco fisso. L'uso di `Platform.resourceDir` permette di accedere all'installazione locale di SC indipendentemente dal sistema operativo in uso, mentre il file `a11wlk01-44_1.aiff` è in dotazione con l'applicazione SC. Si noti che il file è stato campionato ad un tasso di campionamento pari a 44.100 Hz (come dice anche il nome). SC lo legge alla sua frequenza di campionamento, predefinita a 44.100 Hz: essendo le due uguali, non c'è problema. Se invece fossero diverse sarebbe necessario tenerne conto.

8.1.2 Resampling e interpolazione

La semplice lettura del buffer, secondo quanto descritto per `PlayBuf`, riproduce il segnale tale e quale vi è stato memorizzato. Ripensando al caso di `Osc` si può tuttavia notare come fosse specificabile una frequenza: questa frequenza indicava il numero di letture al secondo della tabella contenuta nel buffer da parte dell'oscillatore. Se la tabella fosse di dimensione fissa, si potrebbe leggere più o meno velocemente la tabella, variando opportunamente il tasso di campionamento di scsynth. Ma poiché un buffer ha una dimensione fissa e il tasso di campionamento del server non è modificabile, per poter variare la frequenza del segnale uscente è necessario lavorare sulla tabella in un altro modo. Questo lavoro è comune anche a `PlayBuf` e permette così di "intonare" anche un suono campionato in una tabella. Il procedimento più usato a tale scopo è usualmente definito *resampling*, che equivale, come indica il nome, a un vero e proprio ricampionamento. Supponendo una semplice lettura della tabella si ha che il periodo T (ovvero quanto dura un ciclo della forma d'onda tabulata) è pari a $T = 16/44100 = 0.00036281179138322$ secondi. Ciò indica che la frequenza del segnale risultante è $f = 1/T = 2756.25$ Hz. Se invece si legge un campione ogni due, ovvero il passo di lettura è 2, allora il periodo del nuovo segnale è $T = 16/2 \rightarrow 8/44100 = 0.00018140589569161$ secondi, e dunque la frequenza è $\rightarrow f = 1/T = 5512.5$ Hz. ovvero il doppio della frequenza originale. Rispetto alla tabella, la lettura può procedere "saltando" un valore ogni due. In Figura **8.3** (a) è rappresentato un segnale sinusoidale di la cui frequenza è pari a $\frac{8}{T}$, con $T = 512$ punti. Leggendo un punto ogni due, si ottiene il segnale di Figura **8.3** (b-c): in (b) si può osservare come ogni ciclo del segnale sottocampionato sia rappresentato da metà del numero di campioni rispetto all'originale mentre in (c) si osserva come nella stessa durata il numero di picchi e gole nel segnale

risultante raddoppi, mentre il segnale sottocampionato ha frequenza pari a $\frac{16}{512}$ e durata pari a metà del segnale originale, come si vede in Figura **8.3** (c).

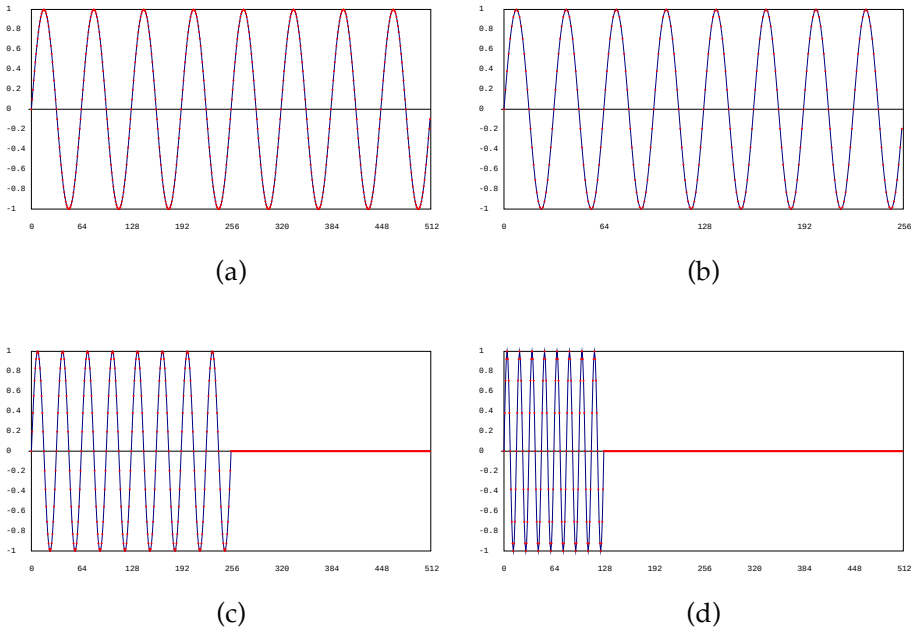


Fig. 8.3 Sinusoide con frequenza f e periodo T (a), segnale sottocampionato leggendo un campione ogni due, $2f$ e $\frac{T}{2}$ (b-c), segnale sottocampionato con $f = 4$ e $\frac{T}{4}$ (d).

Se il passo di lettura è 2 allora la frequenza raddoppia e si ha una all'ottava superiore, mentre il rapporto $4 : 1$ indica una trasposizione pari a quattro ottave. L'operazione fin qui descritta si chiama *downsampling*. L'*upsampling* è il procedimento simmetrico attraverso il quale è possibile diminuire la frequenza del campione originario: a ogni punto della tabella vengono fatti corrispondere non uno, ma due campioni (nel senso di punti di campionamento) nel segnale derivato. In sostanza, è come se tra un campione e l'altro della tabella originale venisse messo un nuovo campione. Rispetto alla tabella originale, ci sarà tra ogni coppia di punti un punto intermedio in più. La frequenza risultante è la metà di quella originale, il che equivale musicalmente a una trasposizione al grave di un'ottava. In questa situazione però i valori dei campioni aggiunti non sono presenti nella tabella originale e devono essere inferiti in qualche modo,

intuibilmente a partire dai valori contigui. Tipicamente ciò avviene per interpolazione tra i due valori originali che precedono e seguono ogni campione aggiunto. Esistono diversi metodi per l'interpolazione, il più semplice e meno oneroso computazionalmente dei quali è l'interpolazione lineare, che geometricamente equivale a tracciare una retta tra i due punti e a cercare sulla stessa il punto intermedio necessario. Ad esempio, se il primo valore è 0 e il secondo 1, il valore interpolato per un punto mancante tra i due sarà 0.5. Altre forme di interpolazione, che qui non si discutono, si basano sullo stesso principio geometrico di determinare una certa curva, non necessariamente una retta, tra due punti e determinare il valore di una ordinata per una ascissa intermedia nota.

L'utilizzo dell'interpolazione permette di determinare il valore di campioni che avrebbero un indice frazionario: cioè, di poter prendere non solo un campione ogni due, tre etc, ma anche ogni 0.9, 1.234 etc. Ciò significa poter trasporre il campione originale non solo all'ottava, ma a tutte le altre frequenze corrispondenti alle altre note di una tastiera, per i quali è necessario considerare anche rapporti numerici non interi tra il numero dei punti iniziali e e quello finale. Ad esempio, il rapporto tra due frequenze a distanza di semitono (sulla tastiera, tra due tasti consecutivi: ad esempio, tra *do* e *do*♯) è $\sqrt[12]{2} \approx 1.06$. In SC si può vedere calcolando il rapporto tra le frequenze (si ricordi che in notazione midi 60 indica il *do* e l'unità rappresenta il semitono):

```
61.midicps / 60.midicps // -> 1.0594630943593
```

Per ottenere una trasposizione di semitono sarebbe dunque necessario prelevare un punto ogni ≈ 1.06 punti. Questa operazione è possibile calcolando per interpolazione il nuovo valore. Infine si ricordi che, poiché dal punto di vista timbrico raddoppiare la frequenza è ben diverso da trasporre strumentalmente all'ottava, se si vuole fare della sintesi imitativa (cioè che cerchi di simulare uno strumento) è opportuno campionare tutte le diverse regioni timbriche (i registri) dello strumento prescelto.

In un oscillatore come `Osc` l'idea è quella di fornire un periodo di un segnale e di poter controllare la frequenza del segnale risultante. La frequenza viene espressa in termini assoluti, poiché indica il numero di volte in cui la tabella viene letta al secondo, e dunque il numero di cicli per secondo. `PlayBuf` è invece un lettore di buffer, impiegato tipicamente quando il buffer contiene non un ciclo ma un segnale complesso, di cui infatti non è necessario conoscere la dimensione: la frequenza di lettura viene allora più utilmente espressa in termini di

proporzione tra i periodi. Il rapporto viene espresso in termini della relazione $\frac{T_1}{T_2}$, dove T_1 è il numero di campioni contenuti nel buffer e T_2 quello del segnale risultante. La stessa relazione può essere espressa in termini di frequenza con $\frac{f_2}{f_1}$, dove f_1 indica la frequenza del segnale di partenza e f_2 quella del segnale risultante. `PlayBuf` permette di specificare un simile tasso attraverso l'argomento `rate`: se $rate = 1$ allora il buffer viene letto al tasso di campionamento del server, se $rate = 0.5$ allora $T_2 = 2$, cioè il buffer viene "stirato" su una durata doppia, e di conseguenza il segnale ha una frequenza f_2 pari alla metà dell'originale, ovvero scende di un'ottava. Se `rate` è negativo, allora il buffer viene letto in senso inverso, dall'ultimo campione al primo, secondo il rapporto espresso dal valore assoluto dell'argomento. Nell'esempio seguente l'argomento `rate` è gestito da un segnale che risulta dall'uscita di una UGen `Line`. Quest'ultima definisce un segnale che decresce linearmente in $[1, -2]$. Questa progressione determina una lettura del buffer inizialmente senza modificazioni rispetto all'originale ($rate = 1$), a cui segue un progressivo decremento che si traduce in un abbassamento della frequenza (e, si noti, in un incremento della durata) fino a $rate = 0$. Quindi il valore cresce ma con segno inverso: così fa la frequenza, ma il buffer viene letto all'inverso (dalla fine all'inizio). Il valore finale del segnale generato da `Line` è -2, che equivale ad una "velocità di lettura" pari al doppio di quella normale e al contrario, raggiunta la quale (in 100 secondi) il synth viene deallocato, stante il valore 2 di `doneAction`.

```
(
SynthDef(\playBuf2 , { arg buf = 0;
    Out.ar(0, PlayBuf.ar(1, buf,
        rate: Line.kr(1, -2, 100, doneAction:2), loop: 1))
}).add ;
)

(
var buf, aSynth ;
buf = Buffer.read(s, Platform.resourceDir +/+ "sounds/a11wlk01.wav") ;
aSynth = Synth(\playBuf2 , [\buf , buf]) ;
)
```

L'esempio seguente dimostra la potenza espressiva di SC (oltre alla sua efficienza) e riprende molti aspetti di un esempio già visto (che non si ridiscuteranno). L'array `source` contiene 100 segnali stereo che risultano dalla spazializzazione attraverso `Pan2` di `PlayBuf`. La variabile `level` è un controllo sull'ampiezza determinato empiricamente. In ognuna delle UGen `PlayBuf` l'argomento `rate` è controllato da una UGen `LFNoise0`, così che l'escursione possibile dello stesso è $[-2.0, 2.0]$, ovvero fino ad un'ottava sopra ed anche al contrario. I 100 `Out` leggono lo stesso buffer, ognuno variando pseudo-casualmente la velocità di lettura. I segnali stereo sono quindi raggruppati per canale (attraverso `flop`, riga 13), i canali (ognuno un array di 100 segnali) sono mixati (14-15) e inviati come array di due elementi a `Out` (16).

```
1  (
2  SynthDef(\playBuf3 , { arg buf = 0;
3      var left, right ;
4      var num = 100 ;
5      var level = 10/num ;
6      var source = Array.fill(num, { arg i ;
7          Pan2.ar(
8              in: PlayBuf.ar(1, buf, rate:
9                  LFNoise0.kr(1+i, mul: 2), loop: 1),
10             pos: LFNoise0.kr(1+i),
11             level: level) ;
12     }) ;
13     source = source.flop ;
14     left = Mix.new(source[0]) ;
15     right = Mix.new(source[1]) ;
16     Out.ar(0, [left, right])
17 }).add ;
18 )

20 (
21 var buf, aSynth ;
22 buf = Buffer.read(s, Platform.resourceDir +/+ "sounds/a11wlk01.wav") ;
23 aSynth = Synth(\playBuf3 , [\buf , buf]) ;
24 )
```

8.2 Generazione diretta

I metodi di sintesi presentati qui di seguito non fanno uso di materiale preesistente come nel campionamento, ma si occupano di generare il segnale direttamente attraverso il calcolo del valore delle formule matematiche che lo descrivono. Sebbene questo valga anche per altre tecniche, l'enfasi è posta in questo caso sulla generazione *ex-novo* del segnale, e non sulla sua successiva modificazione.

8.2.1 Sintesi a forma d'onda fissa

Il metodo più semplice per generare un segnale acustico è quello di calcolarne la forma d'onda attraverso una funzione matematica periodica (*fixed waveform*), secondo quanto esemplificato in precedenza attraverso il calcolo del segnale sinusoidale. Anche se è possibile calcolare il valore d'ampiezza per ogni campione, è spesso usato il metodo *Table Look-Up*, già descritto trattando dell'oscillatore digitale: una forma d'onda viene tabulata per poi essere letta attraverso una scansione ciclica. È possibile generare agevolmente non solo onde sinusoidali, ma di qualsiasi forma: quadra, triangolare, a dente di sega. Si noti che in questi ultimi casi non necessariamente un oscillatore digitale è la soluzione ottimale. Ad esempio, un'onda quadra o rettangolare, in cui l'ampiezza assume un unico valore assoluto, può però essere generata molto più semplicemente stabilendo con quale frequenza, o più generalmente con quale andamento temporale, alternare il segno dello stesso valore. Oltre a `SinOsc`, SuperCollider prevede una vasta gamma di generatori di segnali periodici, di funzione e uso intuibili. Ad esempio, `Pulse` genera onde quadre a *duty cycle* variabile, il termine indicando il rapporto tra parte positiva e negativa del ciclo della forma d'onda, mentre `Saw` genera onde a dente di sega, ancora `Impulse` genera un segnale composto di singoli campioni a determinata frequenza. Molti generatori prevedono anche una versione specializzata per i segnali di

controllo (ad esempio, `LFPulse`, `LFSaw`). Un caso interessante è quello dei generatori di rumore. Un rumore è un segnale il cui comportamento non è predicibile se non in termini statistici. Il rumore perciò è un segnale i cui valori d'ampiezza variano pseudo-casualmente secondo certi criteri. La generazione di un rumore richiede appunto da parte del calcolatore una sequenza di numeri pseudo-casuali: "pseudo" perché un computer ha sempre un comportamento deterministico. La legge che presiede alla generazione della sequenza perciò simula un comportamento casuale, ed è di non facile definizione. Dal punto di vista acustico, rumore è una parola-ombrello che comprende fenomeni molto diversi, di grande rilevanza sonora: ad esempio la componente di rumore è fondamentale nei transitori d'attacco, che spesso definiscono la riconoscibilità di un suono. Nel rumore bianco i valori variano casualmente all'interno dell'intera gamma di variazione definita: il risultato è uno spettro piatto, in cui l'energia è distribuita uniformemente su tutta la gamma. Sono però possibili molte altre distribuzioni statistiche, che definiscono rumori colorati: nei quali cioè alcune aree spettrali risultano preponderanti. Ad esempio, il rumore marrone prevede un'attenuazione dell'ampiezza di 6 dB per ottava. I due spettri, ottenuti con le UGen `WhiteNoise` e `BrownNoise`, sono in Figura **8.4** (a) e (b). Un altro generatore interessante è `Dust`, che distribuisce singoli campioni nel tempo secondo una densità media, cioè in forma approdai. In Figura **8.4** (c) viene rappresentato lo spettro di un segnale in cui la densità $= 100$, confrontato con quello di un generatore `Impulse`, che produce singoli campioni, ma periodicamente (dunque con frequenza $= 100$). Il codice è il seguente:

```
// rumore bianco vs. rosa
{WhiteNoise.ar(0.5)}.play ;
{BrownNoise.ar(0.5)}.play ;
// Dust vs. Impulse
{Dust.ar(100)}.play ;
{Impulse.ar(100)}.play ;
```

L'output di un generatore di numeri pseudo-casuali può essere impiegato come sorgente ricca da sottoporre alla elaborazione (si veda infra, sintesi additiva) o, secondo quanto già visto, come segnale di controllo, ad esempio per introdurre una percentuale di variazione non deterministica nell'ampiezza o nella frequenza di un oscillatore.

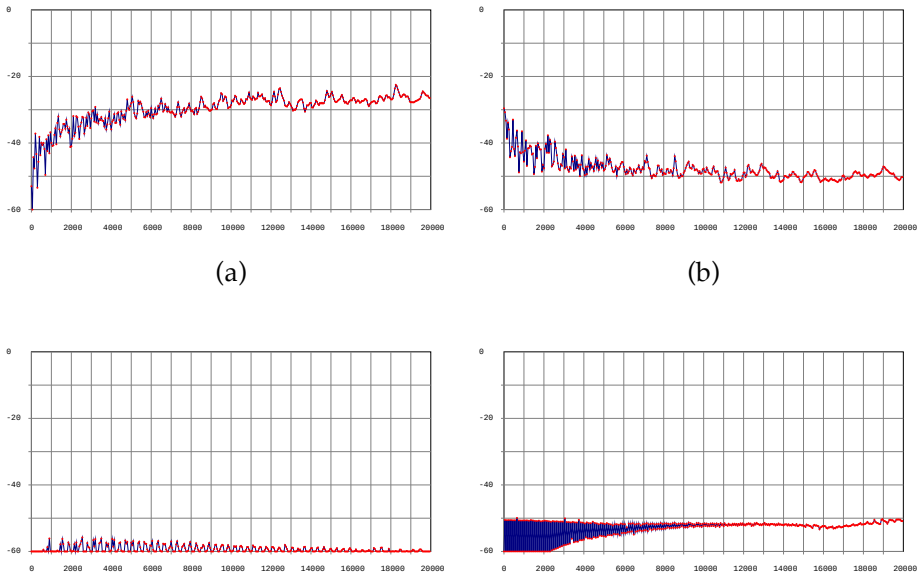


Fig. 8.4 Spettri: `WhiteNoise` (a), `BrownNoise` (b), `Dust` (c) e `Impulse` (d), con densità / frequenza $= 100$.

8.2.2 Modulazione

Si ha modulazione quando un aspetto di un oscillatore (ampiezza, frequenza, fase) varia continuamente in relazione a un altro segnale. Il segnale che viene trasformato si definisce "portante" (*carrier*), quello che controlla la trasformazione "modulante" (*modulator*). Un oscillatore è definito da un'ampiezza e da una frequenza fisse: nella modulazione si sostituisce uno dei due parametri con l'output di un altro oscillatore. Tremolo e vibrato costituiscono di fatto due esempi di modulazione, rispettivamente dell'ampiezza e della frequenza: due segnali di controllo (modulanti) che modificano periodicamente i segnali originali (portanti). Le implementazioni più semplici delle modulazioni di ampiezza e frequenza sono perciò analoghe a quelle del vibrato e del tremolo. Nei casi di tremolo e vibrato però la modulazione avviene al di sotto della soglia di udibilità (*subaudio range*), il segnale modulante ha cioè una frequenza inferiore

ai 20 Hz, e la modulazione viene percepita non come trasformazione timbrica, ma espressiva. Se invece la frequenza della modulante cade all'interno del campo uditivo, il risultato della modulazione è un cambiamento spettrale che si manifesta percettivamente come cambiamento qualitativo del timbro.

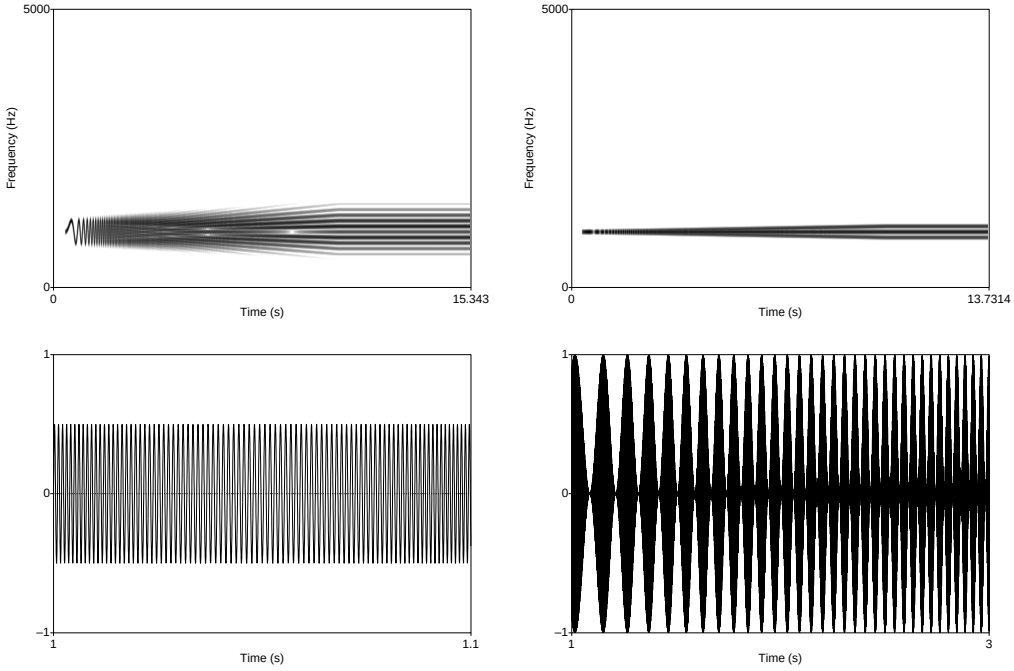


Fig. 8.5 Incremento della frequenza di vibrato (destra) e di tremolo (sinistra), sonogramma (alto) e forma d'onda (basso): dall'espressività al timbro.

In generale, il risultato di una modulazione è un nuovo segnale le cui caratteristiche dipendono dalla combinazione dei due parametri fondamentali: frequenza e ampiezza di portante e modulante. La modulazione è una tecnica di larghissimo impiego perché è estremamente economica in termini computazionali: contrariamente a quanto avviene per la sintesi additiva, che richiede la definizione di moltissimi parametri di controllo, attraverso la modulazione, anche impiegando soltanto due oscillatori come portante e modulante, è possibile

creare spettri di grande complessità. Proseguendo nell'analogia con tremolo e vibrato, la modulazione può avvenire nell'ampiezza e nella frequenza.

8.2.3 Modulazione ad anello e d'ampiezza

Quando il segnale modulante (M) controlla l'ampiezza della portante (C), si possono avere due tipi di modulazione d'ampiezza, che dipendono dalle caratteristiche della modulante. Si ricordi che un segnale bipolare si estende tra un massimo positivo e uno negativo, mentre un segnale unipolare è invece compreso completamente tra valori positivi. Il prossimo esempio mostra due sinusoidi a 440 Hz, bipolare e unipolare. Si noti l'uso del metodo di convenienza `unipolar`.

```
// bipolare
{SinOsc.ar}.plot(minval:-1, maxval:1) ;
// unipolare
{SinOsc.ar(mul: 0.5, add:0.5)}.plot(minval:-1, maxval:1) ;
{SinOsc.ar.unipolar}.plot(minval:-1, maxval:1) ;
```

Un segnale audio è tipicamente bipolare (in ampiezza normalizzata oscilla nell'intervallo $[-1.0, 1.0]$). Come si vede, per trasformare un segnale bipolare in uno unipolare è sufficiente moltiplicarlo e aggiungervi un offset (nell'esempio: $[-1.0, 1.0] \rightarrow [-0.5, 0.5] \rightarrow [0.0, 1.0]$). Se il segnale modulante è bipolare si ha una modulazione ad anello (*Ring Modulation*, RM), se è unipolare si ha quella che viene definita per sineddoche modulazione d'ampiezza (*Amplitude Modulation*, AM). Supponendo che portante e modulante siano due sinusoidi di frequenza rispettivamente C e M, il segnale risultante da una modulazione ad anello ha uno spettro complesso formato da due frequenze, pari a $C - M$ e $C + M$, chiamate bande laterali (*side-bands*). Se $C = 440Hz$ e $M = 110Hz$, il risultato della modulazione ad anello è un segnale il cui spettro è dotato di due componenti armoniche, 330 e $550Hz$. Se invece la modulante è unipolare, e si ha una modulazione d'ampiezza, lo spettro del segnale risultante conserva anche la componente frequenziale della portante, oltre a somma e differenza di portante e modulante. Mantenendo tutti i parametri identici, ma realizzando

una AM attraverso l'impiego di un segnale unipolare, si avranno perciò $C-M$, C, $C+M$: tre componenti pari a 330, 440, $550Hz$. Nel caso in cui la differenza risulti di segno negativo si ha semplicemente un'inversione di fase: il che equivale a dire che la frequenza in questione comunque si "ribalta" al positivo: ad esempio, una frequenza risultante di $-200Hz$ sarà presente come frequenza di $200Hz$. L'esempio seguente mostra due possibili implementazioni, del tutto identiche nei risultati, ma utili come esempio di patching. Nelle prime implementazioni (3, 7) la modulante controlla l'argomento `mul` della portante. Nelle seconde, si ha moltiplicazione diretta dei segnali. L'argomento `mul` definisce il valore per cui ogni campione deve essere moltiplicato: il moltiplicatore è fornito dal valore del segnale modulante. La moltiplicazione di segnali fa la stessa cosa: ogni nuovo campione della portante viene moltiplicato per un nuovo campione della modulante.

```
1 // meglio selezionare la scala logaritmica nella visualizzazione
2 // RM: 2 componenti
3 { SinOsc.ar(440, mul: SinOsc.ar(110))}.freqscope ;
4 { SinOsc.ar(440)*SinOsc.ar(110) }.freqscope ; // lo stesso

6 // AM: 3 componenti
7 { SinOsc.ar(440, mul: SinOsc.ar(110, mul: 0.5, add:0.5))}.freqscope ;
8 { SinOsc.ar(440)*SinOsc.ar(110, mul:0.5, add:0.5) }.freqscope ; // lo stesso
```

8.2.4 Modulazione ad anello come tecnica di elaborazione

La sintesi per modulazione d'ampiezza (AM) nasce come tecnica per costruire spettri più complessi a partire da sinusoidi. La modulazione ad anello ha invece un'origine e un lungo passato analogico[5] non tanto come tecnica di

[5] Lo stesso nome deriva dalla configurazione "ad anello" dei diodi usata per approssimare la moltiplicazione nelle implementazioni in tecnologia analogica.

generazione *ab nihilo* ma come tecnica di elaborazione di segnali complessi[6]. In primo luogo si può pensare di modulare un segnale complesso (ad esempio di origine strumentale) con un segnale sinusoidale. Un segnale complesso avrà uno spettro composto di un insieme di componenti "portanti" C_n:

$$C = C_1, C_2, C_3..., C_n$$

Data una sinusoide M, una modulazione ad anello $C \times M$ risulta allora nella somma e differenza tra ogni componente C e M. Se il segnale modulante è unipolare, anche le componenti C saranno presenti nello spettro:

$$\begin{gathered} C_1 - M, (C_1), C_1 + M; \\ C_2 - M, (C_2), C_2 + M, \\ \ldots \\ C_n - M, (C_n), C_n + M \end{gathered}$$

Se M è ridotto, allora $C_n - M$ e $C_n + M$ saranno vicini a C_n: ad esempio, con uno spettro inarmonico $Cn = 100, 140, 350, 470, etc$ e $M = 10$, si avrà $90, 110, 130, 150, 340, 360, 460, 480, etc - M, etc + M$. Sostanzialmente l'inviluppo spettrale non cambia, ma raddoppia di densità. Quando invece M è elevato, allora si realizza una espansione spettrale. Nell'esempio precedente, se $M = 2000$ allora lo spettro risultante sarà $1900, 2100, 1860, 2140, 1650, 2350, 1730, 2470, ... - M, ... + M$. La situazione è rappresentata in Figura **8.6** (tratta da Puckette *cit.*).

Nell'esempio seguente un file audio è modulato da una sinusoide la cui frequenza incrementa esponenzialmente tra 1 e 10000 Hz nell'arco di 30 secondi. Si noti all'inizio l'effetto di "raddoppio" intorno alle frequenze dello spettro e la progressiva espansione spettrale, che porta, nel finale, all'emergenza della simmetria spettrale intorno alle frequenze della sinusoide.

[6] Alcuni riferimenti e figure da Miller Puckette, *Theory and Techniques of Electronic Music*,
`http://www-crca.ucsd.edu/~msp/techniques.htm`

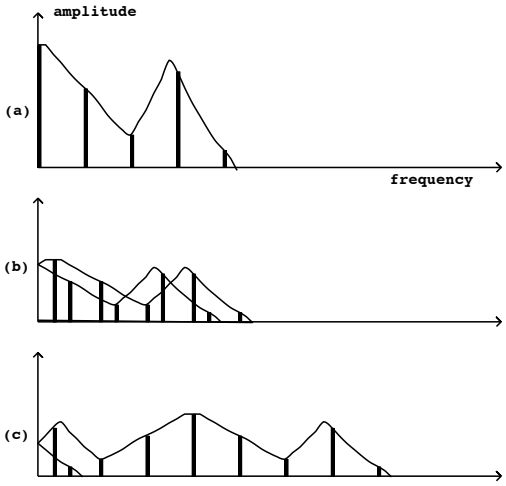


Fig. 8.6 Influenza di M sull'inviluppo spettrale: segnale originale, addensamento ed espansione (da Puckette *cit.*).

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

(
{ PlayBuf.ar(1, b, loop: 1)
    *
    SinOsc.ar(XLine.kr(1, 10000, 30))
}.freqscope
)
```

Lo stesso approccio può essere esteso ai casi in cui M è complesso: è quanto avveniva tipicamente nell'uso analogico dell'RM, in cui spesso due segnali strumentali venivano modulati tra loro[7]. Nel caso allora in cui C e M siano complessi, si ha addizione/sottrazione reciproca di tutte le componenti, così che, se C e M hanno rispettivamente i e k componenti spettrali, $C \times M$ avrà $i \times k$

[7] Ad esempio, in alcuni importanti lavori di K.H.Stockhausen (dal vivo): *Kontakte*, *Hymnen*, *Mikrophonie*, *Prozession*.

componenti, un risultato finale evidentemente di grande complessità. L'esempio seguente utilizza lo stesso file audio predefinito in SC. Il primo segnale modulante è una copia del medesimo file, la cui velocità di lettura è modificata attraverso un generatore di segnale esponenziale da 0.1 a 10.0 (si può ascoltare valutando il primo blocco). Come si vede, il segnale risultante dalla RM (righe 8-14) presenta un inviluppo spettrale molto complesso, in cui pure resta traccia percettiva dei due segnali di partenza.

```
1  ( // il segnale modulante
2  { PlayBuf.ar(1, b,
3      rate: XLine.kr(0.1, 10.0, 30, doneAction:0),
4      loop: 1) }.play ;
5  )

7  (
8  {
9      PlayBuf.ar(1, b, loop: 1)
10     *
11     PlayBuf.ar(1, b,
12     rate: XLine.kr(0.1, 10.0, 30, doneAction:0),
13     loop: 1)
14     *5
15 }.freqscope
16 )
```

Nel prossimo esempio, lo stesso segnale dal file audio è moltiplicato per un onda quadra la cui frequenza incrementa in 30 secondi da 0.5 (quindi $T = 2$) a $100Hz$. Si noti che finché la frequenza è nel registro subaudio ($freq <\approx 20$), il segnale modulante opera propriamente come un inviluppo d'ampiezza (percussivo).

```
1  {     PlayBuf.ar(1, b, loop: 1)
2      *
3      Pulse.ar(XLine.kr(0.5, 100, 30, doneAction:2))
4  }.freqscope
```

Un altro esempio interessante di applicazione della RM si ha quando il segnale portante è un segnale complesso armonico con frequenza fondamentale C, mentre la modulante è una sinusoide con frequenza $M = \frac{C}{2}$. Lo spettro risultante è armonico con frequenza $M = \frac{C}{2}$, e comprende soltanto armoniche dispari. In sostanza, si tratta dello spettro del segnale portante abbassato di un'ottava. Ad esempio, se $C_{1-n} = 100, 200, 300, ...n$ e $M = 50$, allora il segnale modulato avrà queste componenti:

$$100 \pm 50, 200 \pm 50, 300 \pm 50$$
$$=$$
$$50, 150, 150, 250, 250, 350, 350$$

Si tratta delle armoniche dispari (raddoppiate) di $\frac{C}{2}$. I rapporti Tra frequenze risultanti sono infatti $1, 3, 5, 7, ...$ Per recuperare le armoniche pari (altrimenti il segnale è più "vuoto" rispetto all'originale), è sufficiente sommare il segnale originale.
Si ottiene in questo modo una implementazione digitale di un tipico effetto analogico dell'ambito pop/rock, l'*octaver*.

```
//RM octaver
(
SynthDef.new(\RmOctaver , {
    var in, an, freq ;
    in = SoundIn.ar(0) ;     // audio dalla scheda
    an = Pitch.kr(in).poll ; // segnale d'analisi
    freq = an[0] ;    // frequenza dell'altezza
    Out.ar(0, SinOsc.ar(freq: freq*0.5)*in+in);
    // RM freq/2 + source
}).add ;
)

Synth.new(\RmOctaver ) ;
```

La synthDef `RmOctaver` riceve in input il segnale dall'ingresso della scheda audio (il microfono, tipicamente), attraverso la UGen `SoundIn`. Nella UGen, il primo argomento indica il bus richiesto: si ricordi che per convenienza l'indice `0` rappresenta sempre il primo ingresso. La UGen `Pitch` invece è una UGen di analisi, che estrae un segnale di controllo che rappresenta la frequenza fondamentale (*pitch* in inglese) del segnale analizzato. `Pitch` fornisce in uscita un

array che comprende due segnali. Il primo è costituito dai valori di frequenza rilevati, il secondo da un segnale che può avere valore binario $[0, 1]$, ad indicare rispettivamente assenza/presenza di frequenza fondamentale rilevabile. La riga 6 chiama sul segnale in uscita da `Pitch` il metodo `poll` che campiona il segnale sul server e ne stampa il valore sullo schermo. Come si nota, in uscita per ogni campionamento vengono stampati due segnali[8]. La frequenza $\frac{f}{2}$ viene utilizzata per controllare un oscillatore che moltiplica il segnale `in`, infine `in` viene sommato al segnale risultante.
In modo analogo è poi possibile utilizzare una modulante con $M = n \times C$. In questo caso si ha un incremento delle componenti superiori dello spettro del segnale portante. Se $M = n \times C$ si ha un "ribattimento" delle armoniche. Si consideri: $C = 100, 200, 300, n \times 100$ e $M = 200(n \times 2)$. Lo spettro risultante sarà: $100 - 200 = 100, 100 + 200 = 300, 200 - 200 = 0, 200 + 200 = 400, ...$

8.2.5 Modulazione di frequenza

Nella modulazione di frequenza (FM) è la frequenza della portante che viene modificata dalla modulante. Si considerino due oscillatori che producano due sinusoidi di frequenza C e M. Alla frequenza C dell'oscillatore portante viene sommato l'output dell'oscillatore modulante. In questo modo la frequenza C subisce una serie di variazioni che la fanno deviare verso l'acuto (quando l'output di M è positivo, C viene incrementata) e verso il grave (inversamente, quando l'output di M è negativo, C decrementa). Con il termine frequenza di deviazione di picco (*peak frequency deviation*) – o semplicemente deviazione (D) – si indica l'escursione massima, misurata in Hz, subita dalla frequenza della portante, che dipende dall'ampiezza della modulante.

La differenza fondamentale tra FM da una parte e RM e AM dall'altra sta nel fatto che con la FM non si producono solo 2 o 3 bande laterali, ma una serie teoricamente infinita data da tutte le frequenze $C \pm n \times M$. Se $C = 220$ e $M = 110$,

[8] Si noti che le UGen di analisi tipicamente implementano solo il metodo *kr. Infatti, per analizzare un segnale non si può prendere in considerazione un campione, ma solo ciò che avviene in una certa durata, dunque un blocco di campioni. Infatti: che senso avrebbe cercare di stimare la frequenza fondamentale di un campione, cioè di un singolo valore?

si ha con $n = 1$: 330 e 110; con $n = 2$: 440 e 0; con $n = 3$: 550 e 110 (–110 invertito di fase), e così via. Valgono a proposito le considerazioni sugli spettri già svolte per la AM. Con la FM si apre allora la possibilità di creare spettri anche molto complessi attraverso una tecnica di grande semplicità computazionale. Proprio per questo motivo la FM è diventata la prima tecnica di sintesi digitale di grandissimo successo commerciale, grazie a una serie, fortunatissima negli anni '80, di sintetizzatori della Yamaha (soprattutto il celebre DX7). La FM è stata usata dall'inizio '900 nelle telecomunicazioni (come noto, è di uso comune la locuzione "radio in modulazione di frequenza"): sul finire degli anni '60, all'università di Stanford, John Chowning sperimenta inizialmente con rapidissimi vibrato, implementa quindi una versione digitale, e ne esamina a fondo le possibilità (1973).*Turenas* (1972) è il primo brano ad usare estensivamente la FM, anche se il primo tentativo in assoluto è *Sabelithe* del 1966, completato nel 1988. Chowning ottiene con due oscillatori risultati timbrici pari all'uso di 50 oscillatori in sintesi additiva. Quindi, la Yamaha compra il brevetto (ancora adesso quello più redditizio per Stanford...) e applica la tecnica sui sintetizzatori.

Dunque, la FM si caratterizza per la ricchezza timbrica. In realtà, delle infinite bande teoricamente presenti, poche sono quelle significative: per determinarne il numero approssimativo è utile il cosiddetto indice di modulazione I. La relazione che lega deviazione, frequenza modulante e indice di modulazione è la seguente: $I = \frac{D}{M}$. In questo senso, D rappresenta la "profondità" della modulazione. Il valore dato da $I + 1$ viene considerato un buon indicatore del numero approssimativo di bande laterali "significative". L'utilità della formula sta in questo: se si considerano D e M costanti (poiché dipendono da modulante e portante), l'indice fornisce una misura della ricchezza in bande laterali dello spettro. Dalla relazione precedente discende che l'ampiezza della modulante è data dalla relazione: $D = I \times M$. Se $I = 0$, non c'è deviazione, poiché l'incremento fornito dalla modulante alla portante (la deviazione) è nullo, e il segnale portante non viene modulato. L'incremento dell'indice di modulazione indica un incremento della ricchezza dello spettro.

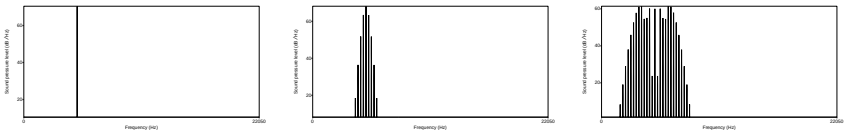


Fig. 8.7 $I = 1, 3, 7$.

Riassumendo: nella FM, la natura dello spettro generato (cioè, la posizione delle bande laterali) è determinata dalla relazione tra la portante e la modulante,

mentre la ricchezza dello spettro (cioè, il numero di bande laterali) è proporzionale all'ampiezza del segnale modulante.

I prossimi due esempi sono più "visivi" che sonori. Entrambi assumono di visualizzare lo spettro del segnale in modalità lineare, in modo da rendere più chiaro il gioco di espansione simmetrica dello spettro per somma/sottrazione della frequenza della modulante rispetto alla portante che fa da perno (riga 1). Nel primo esempio, la portante di 5.000 Hz viene progressivamente modulata con una frequenza da 10 a 1.000 Hz.

```
1 s.freqscope ; // impostare visualizzazione lineare

3 (
4 { SinOsc.ar( 5000     // portante C
5     + SinOsc.ar(XLine.kr(10, 1000, 60, doneAction:2), mul: 1000),
6   mul: 0.5
7     )}.play
8 )
```

Il prossimo esempio invece intende dimostrare l'importanza dell'ampiezza della modulante, che crescendo sottrae energia alla portante per distribuirla sulle bande laterali. Si noti che l'inviluppo spettrale segue un andamento simmetrico intorno alla portante ma non è sempre a forma di lancia, poiché è invece definito teoricamente da un gruppo di funzioni che si chiamano "funzioni di Bessel".

```
1 (
2 { SinOsc.ar(
3     10000     // portante C
4     + SinOsc.ar(500, mul: XLine.kr(1, 20000, 60, doneAction:2)),
5   mul: 0.5
6     )}.play
7 )
```

Infine, ancora un esempio minimale, basato soltanto su due oscillatori, in cui però il mouse permette di esplorare interattivamente gli effetti della modulazione e apprezzarne la complessità dei risultanti nonostante la semplicità dell'implementazione.

```
(
{ SinOsc.ar(
    2000     // portante C
    + SinOsc.ar(    // modulante M
        freq: MouseX.kr(0, 1200), // freq M
        mul: MouseY.kr(0, 20000)    // amp M
            ),
  mul: 0.5
    )}.freqscope
)
```

8.2.6 C:M ratio

Oltre all'ampiezza della modulante, l'altro fattore che va tenuto in considerazione nello studio delle caratteristiche spettrali del segnale risultante da una modulazione è il rapporto tra le frequenze di portante e modulante. Tale fattore è di particolare rilievo per la modulazione di frequenza o nella modulazione ad anello di segnali complessi: in entrambi casi, a differenza di quanto avviene nella AM o nella RM in cui C e M siano sinusoidali, il segnale in uscita risulta costituito da numerose componenti, e non soltanto da due (RM) o tre (AM). Il rapporto tra le frequenze dei due segnali viene usualmente definito come *C:M ratio* (di qui in poi: *cmr*). Poiché le frequenze componenti lo spettro del segnale comprendono somma e differenza di C e M, se *cmr* è un intero lo spettro sarà armonico, comprendendo due multipli del massimo comun divisore tra C e M. Ad esempio, se $C = 5000$ Hz e $M = 2500$ Hz ($cmr = 2:1$), si avranno (in AM) 2500, 5000 e 7500 Hz: uno spettro armonico formato da fondamentale (2500) e dalle prime due armoniche (i multipli 5000 e 7500 Hz). Due casi interessanti in RM e in FM si hanno quando $cmr = 1:2$ e $cmr = 1:1$. Nel primo caso sono presenti soltanto le armoniche dispari: con $C = 1000$ e $M = 2000$, si ha in RM $(-)1000, 3000$, a cui si aggiungono in FM: $(-)3000, 5000, (-)5000, 7000$, ecc. Nel secondo caso, tutte le armoniche sono presenti nel segnale modulato: con $C = M = 1000$, si ha in RM $0, 2000$, e in FM $0, (-)1000, 3000, (-)2000, 4000$ e così via. Se la frazione ha denominatore 1 (come nell'esempio precedente),

allora le frequenze ottenute sono multipli della modulante, che diventa fondamentale del segnale generato. Se il denominatore è maggiore di 1, è il massimo comun divisore tra C e M a diventare la fondamentale del segnale risultante dalla modulazione: con $C = 3000$ e $M = 2000$ ($cmr = 3 : 2$) la nuova fondamentale è il massimo comun divisore, ovvero 1000 Hz, lo spettro essendo composto da 1000 Hz ($3000 - 2000$), cui si aggiungono (in AM) 3000 Hz (C), e 5000 Hz ($3000 + 2000$). A differenza di questo caso, la fondamentale può anche essere apparente. Ad esempio, se $C = 5000$ e $M = 2000$, ($cmr = 5 : 2$) la nuova fondamentale è sempre 1000 Hz (il MCD tra 5000 e 2000), ma lo spettro è composto da 3000 Hz ($5000 - 2000$), 5000 (in AM), e 7000 ($5000 + 2000$). La fondamentale di 1000 Hz è apparente perché può essere ricostruita dall'orecchio, pur non essendo presente fisicamente nel segnale, come la fondamentale di cui sono presenti le armoniche III, V e VII. La fondamentale viene ricostruita dall'orecchio se cade nel campo di udibilità e se sono presenti un numero sufficiente di armoniche.

Quest'ultima considerazione vale per la modulazione di frequenza, poiché in RM e AM semplici (dove C e M sono segnali sinusoidali) le bande laterali sono rispettivamente sempre solo 2 e 3. Nell'esempio precedente, sempre con $C = 5000$ Hz ma con $M = 2000$ Hz, si avranno (in AM) frequenze risultanti pari a $3000, 5000, 7000$ Hz: terzo, quinto, settimo armonico di 1000 Hz. La *cmr* è perciò un'indicatore della "armonicità" dello spettro: più è semplice la frazione (cioè minore è il prodotto $C \times M$), più vicini sono gli armonici risultanti. Se la *cmr* è quasi intera (ad esempio, $2.001 : 1$) si ottiene un suono che non solo viene ancora percepito come armonico, ma che risulta invece meno artificiale proprio perché simula le inarmonicità presenti negli strumenti acustici.

In generale, si ha che valori di *cmr* pari a $N : 1, 1 : N$ riproducono lo stesso spettro. Il numero dei parziali può essere calcolato a partire dai componenti della *cmr*. Ad esempio, se *cmr* è $2 : 3$, si ha allora $|2 \pm 3 \times n| = 1, 2, 4, 5, 8, 11$.... Se $C > 1$, allora ci sono bande laterali inarmoniche (o "fondamentale mancante"). Ad esempio, se $cmr = 2 : 5$, lo spettro risultante sarà $2, 3, 7, 8, 12, 13, ...$: come si vede, manca la fondamentale (la componente 1) e molte armoniche. In più, lo spettro si allontana all'acuto. Si consideri $cmr = 5 : 7$: si produce uno spettro distintamente inarmonico: $2, 5, 9, 12, 16, 19, 23, 26, ...$

La synthDef seguente permette di controllare una modulazione di frequenza utilizzando, oltre la frequenza di base `freq`, ovvero C, i parametri derivati dalla *cmr*, ovvero `c, m, a`. I primi due indicano numeratore e denominatore della *cmr*, il terzo l'ampiezza della modulante.

```
(
SynthDef(\cm , { arg f = 440, c = 1, m = 1, a = 100, amp = 0.5 ;
    Out.ar(0,
    SinOsc.ar(
    f     // freq base C
    + SinOsc.ar(    // modulante M
        freq: f * m / c, // freq M, calcolata dalla cmr
        mul: a    // ampiezza M
            ),
  mul: amp) // ampiezza C
    )
}).add ;
)
```

L'interfaccia grafica del prossimo esempio permette di controllare i parametri `f, c, m, a` attraverso cursori e campi d'inserimento numerico. Si noti che il codice produce solo valori interi di C e M. Nonostante vega utilizzata una sintassi procedurale (attraverso `Array.fill`), il codice è piuttosto lungo (come tipico con le interfacce grafiche), ma ciò che fa è semplicemente associare ad ogni elemento grafico l'azione di controllo sul synth (`cmsynth.set`) e aggiornare l'elemento correlato (ovvero: cambiando il valore del cursore, cambia anche quello del campo numerico, e viceversa). Il codice in un blocco unito (ed eseguibile direttamente) è accessibile qui:

```
var cmsynth = Synth("cm") ;
var freq = 2000 ; // f C: 0-2000 Hz
var num = 30 ; // intervallo per c:m
var w = Window("C:M player", Rect(100, 100, 220, 420)).front ;
var sl = Array.fill(4, {|i| Slider(w, Rect(i*50+10, 10, 50, 350))}) ;
var nb = Array.fill(4, {|i| NumberBox(w, Rect(i*50+10, 360, 40, 20))}) ;
["freq C", "C", "M", "amp M"].do{|i,j|
    StaticText(w, Rect(j*50+10, 390, 40, 20)).string_(i).align_(\center )
} ;

sl[0].action = { arg sl ; // freq base
    var val = sl.value*freq ;
    cmsynth.set("f", val) ; nb[0].value = val ;
    } ;
nb[0].action = { arg nb ;
    var val = nb.value ; // 0-1000 Hz
    cmsynth.set("f", val) ;    sl[0].value = val/freq ;
    } ;

sl[1].action = { arg sl ; // numeratore C
    var val = (sl.value*(num-1)).asInteger+1 ;
    cmsynth.set("c", val) ;    nb[1].value = val ;
    } ;
nb[1].action = { arg nb ;
    var val = nb.value.asInteger ;
    cmsynth.set("c", val) ; sl[1].value = val/num ;
    } ;

sl[2].action = { arg sl ; // denominatore M
    var val = (sl.value*(num-1)).asInteger+1 ;
    cmsynth.set("m", val) ;    nb[2].value = val ;
    } ;
nb[2].action = { arg nb ;
    var val = nb.value.asInteger ;
    cmsynth.set("m", val) ;    sl[2].value = val/num ;
    } ;

sl[3].action = { arg sl ; // ampiezza M
    var val = sl.value*10000 ;
    cmsynth.set("a", val) ;    nb[3].value = val ;
    } ;
nb[3].action = { arg nb ;
    var val = nb.value ;
    cmsynth.set("a", val) ;    sl[3].value = val/10000 ;
    } ;
```

8.2.7 Waveshaping

La tecnica *waveshaping*, detta anche distorsione non lineare, in qualche misura mette insieme aspetti della sintesi wavetable e della modulazione. Se si moltiplica un segnale per una costante k la forma d'onda è immutata: poiché si incrementano/decrementano proporzionalmente il valore di tutti i campioni, il risultato sarà un incremento/decremento dell'ampiezza globale del segnale. Il waveshaping è invece un'operazione non lineare, proprio perché si prefigge di modificare ("distorcere") la forma d'onda di un segnale in entrata (nel caso più tipico una semplice sinusoide, ma non necessariamente): tale distorsione ha come correlato una conseguente alterazione dello spettro del segnale in uscita: sia il clipping, sia la distorsione impiegata nella musica pop-rock, possono essere considerati in questi termini funzioni distorcenti a tutti gli effetti, poiché producono in uscita (tipicamente) una "quadratura" del segnale in entrata. La distorsione in un circuito analogico si ottiene esattamente nel momento in cui le componenti elettroniche non restituiscono un'ampiezza proporzionale rispetto all'ingresso, ma in qualche misura amplificano e schiacciano il segnale, di fatto squadrandolo.

Nel waveshaping la distorsione viene operata attraverso una tabella che associa alla serie dei valori d'entrata un'altra serie di valori.

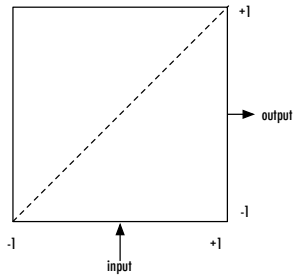


Fig. 8.8 Waveshaping: tabella della funzione di trasferimento.

Una tabella per il waveshaping associa un valore in input del segnale a un valore in output, come in Figura **8.8**, e rappresenta infatti la "funzione di trasferimento". Se la funzione è una retta a 45°, a ogni valore in input corrisponde esattamente lo stesso valore in output, mentre ogni scostamento dalla stessa produce una distorsione in output del valore in input. Un'ipotesi di implementazione è riportata nel prossimo esempio.

```
// waveshaping
t = FloatArray.fill(512, { |i| i.linlin(0.0, 512.0, -1.0, 1.0) });
t.plot ;
b = Buffer.sendCollection(Server.local, t)

{
    var sig = SinOsc.ar(100) ;
    Out.ar(0, BufRd.ar(1, bufnum: b,
        phase: sig.linlin(-1.0, 1.0, 0, BufFrames.ir(b)-1)
    ))
}.scope
```

L'array `t` contiene 512 valori, scalati nell'intervallo $[-1, 1]$. In altri termini, è una rampa, come si vede dal diagramma. È pensato cioè come implementazione di una funzione di trasferimento neutra. La classe `Buffer` è dotata di alcuni metodi che permettono di calcolare valori per un buffer sul lato del linguaggio (client) e caricarli sul buffer in questione, a lato server, ad esempio `sendCollection`, attraverso il quale il buffer `b` viene allocato e riempito con `t`. La synthDef successiva assume come sorgente una sinusoide a 100 Hz. La UGen `BufRd` genera in uscita un segnale il cui valore corrisponde al buffer assegnato (qui `b`) nella posizione definita da `phase`. Quest'ultimo argomento dipende dal valore del segnale in ingresso. In altri termini, il valore della sinusoide diventa l'indice della tabella da cui leggere. La tabella è di (soli) 512 punti, e dunque deve essere indicizzata opportunamente: quindi il valore della sinusoide (compreso in $[-1, 1]$) viene riportato agli indici della tabella, che iniziano da 0 e finiscono a 511, valore qui ottenuto sottraendo 1 alla dimensione del buffer ottenuta attraverso l'uso della UGen di utilità `BufFrames`. Si noti il tasso `ir`, che indica "instrument rate", e viene cioè aggiornato solo quando il synth è allocato. I prossimi esempi sfruttano, all'interno della stessa logica, le possibilità offerte dalla costruzione funzionale dell'array, e dimostrano la sensibilità alla dinamica resa possibile dal waveshaping. La prima tabella comprime i valori al di sotto dell'ampiezza

0.5 e espande quelli al di sopra. Poiché il valore supera 1 si ha clipping (desiderato). Il metodo `normalize` permette mantenere la "forma" prevista all'interno dell'escursione normalizzata utile per il segnale di ingresso. Il synth permette di valutare il diverso comportamento attraverso il controllo dell'ampiezza tramite mouse.

```
// 1
(
t = FloatArray.fill(512, { |i|
    v = i.linlin(0.0, 512.0, -1.0, 1.0) ;
    if (abs(v) < 0.5){v*0.5} { v*1.25}
}).normalize(-1.0, 1.0);
t.plot ;
b = Buffer.sendCollection(Server.local, t)
)

(
{
    var sig = SinOsc.ar(100, mul:MouseX.kr(0,1)) ;
    Out.ar(0, BufRd.ar(1, bufnum: b,
        phase: sig.linlin(-1.0, 1.0, 0, BufFrames.ir(b)-1)
    ))
}.scope
)
```

Il secondo esempio utilizza una tabella molto complessa (costruita del tutto empiricamente) che produce uno spettro ricco, di nuovo sensibile alla dinamica. Si noti che la tabella produce a basse ampiezze un segnale asimmetrico rispetto allo zero, ovvero un "DC offset" (dove DC indica "direct current", perché questo fenomeno è tipicamente associato in analogico ad una qualche corrente continua che interferisce). Per ovviare al problema si può usare la UGen `LeakDC`, che lo rimuove.

```
// 2
(
t = FloatArray.fill(512, { |i|
    v = i.linlin(0.0, 512.0, -1.0, 1.0) ;
    v.round(0.125+(v*1.4*(i%4)))
}).normalize(-1.0, 1.0);
t.plot ;
b = Buffer.sendCollection(Server.local, t)
)

(
{
    var sig = SinOsc.ar(100, mul:MouseX.kr(0,1)) ;
    Out.ar(0, LeakDC.ar(
        BufRd.ar(1,
            bufnum: b,
            phase: sig.linlin(-1.0, 1.0, 0, BufFrames.ir(b)-1))
    ))
}.scope
)
```

La Figura riporta due spettri per i casi in cui l'ampiezza di `SinOsc` sia 0.0001 e 1.0. I due segnali sono stati normalizzati per enfatizzare il diverso sviluppo spettrale.

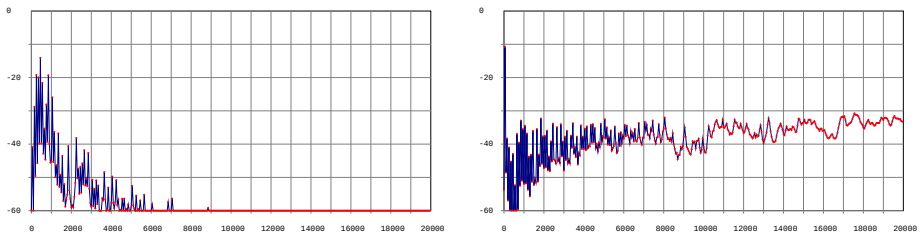


Fig. 8.9 Spettri per ampiezza di `SinOsc` $= 0.0001$ e $= 1.0$.

In realtà, il modo standard in cui è implementato il waveshaping in SC utilizza una UGen apposita, `Shaper`. Il prossimo esempio ne dimostra l'uso. Viene utilizzata la classe `Signal` per costruire la tabella, implementando il primo caso dell'esempio precedente. Si noti che `Signal` deve avere una dimensione pari a potenza di $2 + 1$ (513). Quindi viene allocato un buffer di dimensione doppia

(1024), perché deve contenere i dati in un formato speciale ("wavetable"), ottenuto attraverso `asWavetableNoWrap` (si tratta di aspetti che dipendono esclusivamente dall'implementazione). I parametri di `Shaper` sono di agevole lettura: il buffer e il segnale in ingresso.

```
// dimensione potenza di 2 + 1
t = Signal.fill(513, { |i|
    var v = i.linlin(0.0, 512.0, -1.0, 1.0);
    if (abs(v) < 0.5){v*0.5} { v*1.25}
}).normalize(-1.0,1.0);
t.plot ;

// buffer doppio
b = Buffer.alloc(s, 1024, 1);
b.sendCollection(t.asWavetableNoWrap);

{ Shaper.ar(b, SinOsc.ar(440, 0, MouseX.kr(0,1)))*0.75 }.scope ;
```

L'utilità di `Shaper` sta nell'implementare agevolmente l'utilizzo di un insieme di funzioni di trasferimento, dette polinomi di Chebyshev, che permetto un calcolo preciso del contenuto armonico in uscita dall'applicazione della funzione ad una sinusoide. Infatti, il waveshaping produce risultati molto ricchi ma difficilmente controllabili dal punto di vista teorico. La tecnica si avvicina in questo modo alle modulazioni. Nell'esempio seguente un buffer viene riempito con gli opportuni polinomi che permettono di ottenere l'insieme delle prime 20 armoniche, in questo caso con ampiezza pseudo-casuale.

```
b = Buffer.alloc(s, 1024, 1);
b.cheby(Array.fill(20, {1.0.rand}));

{ Shaper.ar(b, SinOsc.ar(440, 0, 0.4)) }.scope;
```

In conclusione, le tecniche per modulazione hanno dalla loro parte la capacità di creare spettri di grande complessità unitamente ad un'estrema economicità in termini computazionali: si pensi alla complessità spettrale ottenuta

ad esempio in FM a partire da due sinusoidi. D'altra parte, non è ovvia la relazione tra controllo e risultato. In più, le tecniche per modulazione scontano una inapplicabilità analitica: è estremamente difficile l'estrazione funzionale dei parametri controllo dall'analisi di materiale sonoro preesistente.

8.3 Modellazione spettrale

Le tecniche di modellazione spettrale hanno come obiettivo esplicito quello di ottenere in uscita un certo spettro che può essere specificato in forma analitica, a partire cioè dalle singole componenti. Nei metodi che appartengono a questa categoria l'interesse precipuo per il dominio della frequenza pone in qualche misura in secondo piano il dominio del tempo.

8.3.1 Sintesi additiva

Il teorema di Fourier stabilisce che ogni segnale periodico per quanto complesso può essere rappresentato come la somma di semplici segnali sinusoidali. L'idea alla base della sintesi additiva è quello di procedere in senso inverso, ottenendo un segnale complesso a partire dalla somma di semplici segnali sinusoidali, ognuno dei quali dotato di un suo inviluppo d'ampiezza.

La Figura **8.10** illustra un banco di oscillatori che lavorano in parallelo: gli n segnali risultanti vengono sommati e risultano in un segnale complessivo spettralmente più ricco. Nella sintesi additiva classica gli oscillatori sono "intonati" a partire da una frequenza fondamentale: le loro frequenze ($f2...fn$) sono cioè multipli interi della frequenza fondamentale ($f1$).

Si tratta di uno dei metodi di sintesi in uso da più antica data proprio perché le due operazioni (sintesi di segnali sinusoidali e loro somma) sono di implementazione molto semplice. Se nella teoria dei segnali si parla di "somma di segnali", più tipicamente in ambito audio ci si riferisce alla stessa cosa in termini di missaggio (mixing, miscelazione). Come si è visto, la UGen specializzata nel missaggio in SC è `Mix`, che riceve in entrata un array si segnali, li somma e

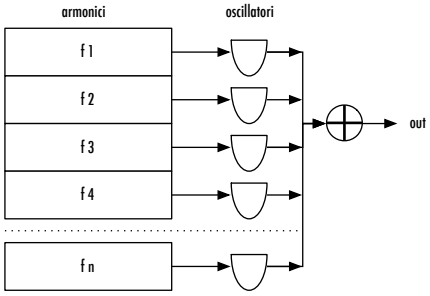


Fig. 8.10 Banco di oscillatori.

restituisce un nuovo segnale (mono). Il missaggio in digitale è propriamente e banalmente una somma.

Passando alla sintesi, l'esempio seguente –per quanto minimale– è istruttivo rispetto alla potenza espressiva di SC. Come si vede, la riga 3 introduce un oggetto `Mix`: esso deve riceve come argomento un array di segnali che sommerà restituendo un segnale unico. A `Mix` viene passato un array che contiene 20 segnali (generati da `SinOsc`). Si noti che `i` è il contatore e viene utilizzato come moltiplicatore della frequenza degli oscillatori, che saranno perciò multipli di $200Hz$. Poiché `i` è inizializzata a 0 il primo oscillatore avrebbe una frequenza nulla (= assenza di segnale), e dunque è necessario aggiungere 1 a `i`. Nel missaggio, la somma dei segnali è lineare: ciò vuol dire che n segnali normalizzati nell'intervallo $[-1.0, 1.0]$ verranno missati in un segnale risultante la cui ampiezza sarà compresa tra $n \times [-1.0, 1.0] = [-n, n]$: intuibilmente, ciò risulta in un abbondante clipping. Nell'esempio, il problema è risolto attraverso `mul:1/20`: ogni segnale ha al massimo ampiezza pari a $1/20$, e dunque, pure nella circostanza in cui tutti i segnali siano simultaneamente al massimo dell'ampiezza, il segnale mixato non supererà i massimi previsti.

```
1 { Mix.new    // mix
2     ( Array.fill(20, { arg i ; // 20 armoniche
3         SinOsc.ar(200*(i+1), mul: 1/20)}))
4 }.scope ;
```

Vista l'attitudine algoritmica di SC, il linguaggio mette a disposizione un metodo `fill` definito direttamente sulla classe `Mix`. Il codice seguente è esattamente identico nei risultati al precedente.

```
{
// lo stesso
Mix.fill(20, { arg i ;
    SinOsc.ar(200*(i+1), mul: 1/20)})
}.scope ;
```

Negli approcci precedenti ogni sinusoide inizia il suo ciclo nello stesso momento. Questo determina un picco visibile nella forma d'onda, che si traduce in una sorto di impulso. In generale, la fase è irrilevante dal punto di vista acustico nella parte stazionaria del segnale, mente assume importanza critica nei transitori e nelle discontinuità del segnale. Poiché quanto qui interessa è soltanto la relazione tra le componenti (in termini stazionari), è meglio evitare allineamenti di fase. Nell'esempio qui di seguito ogni oscillatore riceve una fase iniziale casuale attraverso `2pi.rand`.

```
// evitando la sincronizzazione delle fasi
{ Mix.fill(20, { arg i ;
    SinOsc.ar(200*(i+1), 2pi.rand, mul: 1/20)})
}.scope ;
```

Variando l'ampiezza delle armoniche componenti, si modifica evidentemente il peso che queste hanno nello spettro. Qui di seguito, un segnale stereo che risulta dalla somma di 40 sinusoidi a partire da 50 Hz viene modulata nell'ampiezza in modo differente sul canale destro e su quello sinistro: nel primo caso l'ampiezza dei componenti varia casualmente con una frequenza di 1 volta al secondo, nel secondo caso segue una oscillazione sinusoidale con frequenza casuale nell'intervallo $[0.3, 0.5]$ Hz. Gli argomenti `mul` e `add` rendono il segnale unipolare (compreso nell'intervallo $[0.0, 1.0]$). L'ampiezza è divisa per 20: secondo quanto vista prima, dovrebbe essere divisa per 40, ma, a causa delle variazioni delle ampiezze dei segnali, una divisione (empirica) per 20 è sufficiente ad evitare il clipping.

```
// movimento spettrale in stereo
{ Mix.fill(40, { arg i ;
    var right = LFNoise1.kr(1, 1/20)  ;
    var left = SinOsc.kr(rrand(0.3, 0.5), 2pi.rand, mul: 0.5, add: 0.5) ;
    SinOsc.ar(50*(i+1), [2pi.rand, 2pi.rand], mul: [left/20, right]) })
}.scope ;
```

L'esempio seguente scala l'ampiezza di ogni componente in proporzione inversa al numero di armonica: più il numero è grande, minore è l'ampiezza, secondo un comportamento fisicamente comune negli strumenti musicali. La variabile `arr` contiene prima una sequenza di interi da 1 a 20, il cui ordine viene invertito (attraverso il metodo `reverse`), il cui valore viene normalizzato in modo che la somma di tutti i componenti sia pari a 1. In altre parole, sommando tutti gli elementi dell'array `arr` si ottiene 1. Questo vuol dire che se si usano gli elementi dell'array come moltiplicatori delle ampiezze non si incorre in fenomeni di clipping. La frequenza dell'oscillatore è espressa in notazione midi (60 = do centrale), che viene quindi convertita in cicli al secondo (`midicps`).

```
{
var arr = Array.series(20, 1).reverse.normalizeSum ;
Mix.new    // mix
    ( Array.fill(20, { arg i ; // 20 partials
        SinOsc.ar(60.midicps*(i+1), 2pi.rand, mul: arr[i])}))
}.scope ;
```

Attraverso la sintesi additiva possono essere generate proceduralmente molte forme d'onda di tipo armonico. Ad esempio, l'onda quadra può essere descritta come una somma infinita di armoniche dispari la cui ampiezza è proporzionale all'inverso del numero d'armonica, secondo la relazione $f_1 \times 1, f_3 \times 1/3, f_5 \times 1/5, f_7 \times 1/7...$

La Figura **8.11** mostra la progressione delle armoniche secondo la legge dell'onda quadra: l'aumento del numero di armoniche produce una progressiva approssimazione dell'onda quadra. Teoricamente, l'onda quadra si raggiunge all'infinito, e come intuibile in generale se si vuole produrre un'onda quadra meglio sfruttare altri approcci e generatori dedicati. Come si nota, la

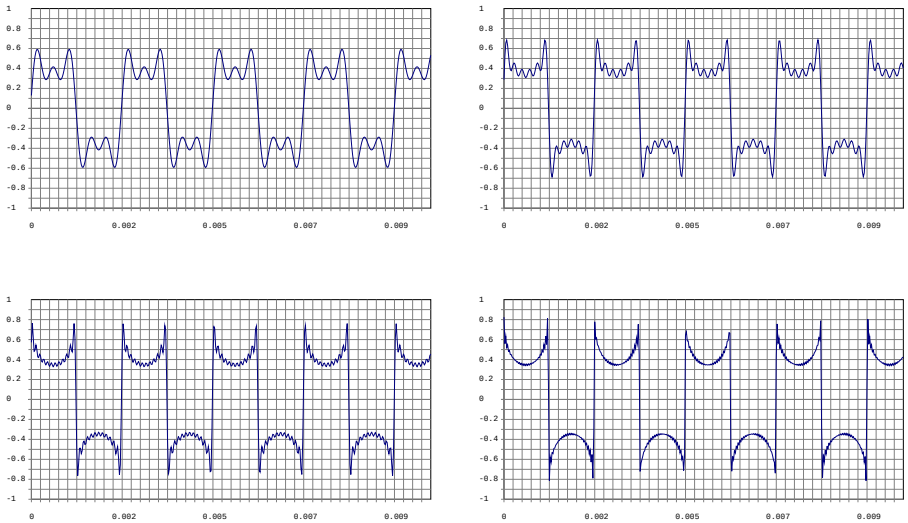


Fig. 8.11 Onda quadra: prime 3, 6, 12, 24 armoniche.

ripidità degli spigoli nella forma d'onda è proporzionale al numero dei componenti. Dunque, nell'osservazione di una forma d'onda l'arrotondamento indica un basso numero di componenti e, all'opposto, la presenza di transizioni ripide un numero alto. Un ragionamento del tutto analogo può essere fatto per l'onda a dente di sega, che è costruibile in maniera analoga all'onda quadra, con la differenza che tutti gli armonici, e non solo quelli dispari, vanno considerati. Il codice relativo ai due casi è il seguente:

```
~numArm = 10 ; ~baseFreq = 69.midicps ;

// onda quadra
{Mix.fill(~numArm, {|i|
    SinOsc.ar(~baseFreq*(i*2+1), mul:1/( i+1))
})}.play ;

// onda a dente di sega
{Mix.fill(~numArm, {|i|
    SinOsc.ar(~baseFreq*(i+1), mul:1/( i+1))
})}.play ;
```

La discussione sulla sintesi additiva ha introdotto non a caso la classe `Mix`, perché in effetti la sintesi additiva può essere generalizzata nei termini di un micro-missaggio/montaggio in cui un insieme arbitrario di componenti viene sommato per produrre un nuovo segnale. Si parla in questo caso di "somma di parziali", intendendo come parziale una componente di frequenza arbitraria in uno spettro (e che perciò non deve essere relata armonicamente alla fondamentale, e che può includere anche segnali "rumorosi"). Restando nell'ambito dei componenti sinusoidali, si può descrivere l'armonicità nei termini di un rapporto tra le frequenze delle componenti (*ratio*, in latino e in inglese, si pensi alla discussione precedente sulla *cmr*). In caso di armonicità, si ha *integer ratio* tra fondamentale e armoniche (il rapporto è espresso da un numero intero), nel caso di un parziale si ha *non-integer ratio*, poiché $\frac{parziale}{f_1}$ non è intero. Nell'esempio seguente l'unica novità è la riga 10, che introduce un incremento casuale (e proporzionale al numero di armonica) nelle frequenze delle armoniche. Dunque lo spettro sarà inarmonico. Tra l'altro, una componente ridotta di inarmonicità è tipica di molti strumenti acustici ed è di solito auspicabile se si intende ottenere una "memoria acustica" nello strumento di sintesi (si parla allora di *quasi-integer ratio*). Si noti che ogni volta che il codice viene valutato, nuovi valori vengono generati con conseguente cambiamento dello spettro.

```
1  // invilluppi variabili con quasi-integer ratio
2  {
3  Mix.new( Array.fill(50,
4      { arg k ;
5      var incr = 1 ; // quasi-intero. Provare a aumentarlo: 2, 5, ...
6      var env ;
7      i = k+1 ;
8      env = LFNoise1.ar(LFNoise0.ar(10, add:1.75, mul:0.75), add:0.5, mul:0.5) ;
9      SinOsc.ar(50*i
10         +(i*incr).rand,
11         mul: 0.02/i.asFloat.rand)*env })
12 )}.scope
```

Uno spettro di parziali può essere ottenuto per addizione di componenti sinusoidali, in numero di 80 nell'esempio qui di seguito.

```
// Uno spettro generico di parziali
{
var num = 80 ;
Mix.new( Array.fill(num, { SinOsc.ar(20 + 10000.0.rand, 2pi.rand, 1/num) }) );
}.scope
```

È possibile sommare sinusoidi in un intervallo ridotto, intorno ai 500 Hz:

```
// Modulazione intorno a 500 Hz
{
Mix.new( Array.fill(20, {
        SinOsc.ar(500 +
        LFNoise1.ar(
                LFNoise1.ar(1, add:1.5, mul:1.5),
                add:500, mul: 500.0.rand), 0, 0.05) }) );
}.scope ;
```

Infine, in relazione alla generalizzazione della tecnica in termini di montaggio/missaggio, nell'esempio seguente quattro segnali ottenuti con tecniche di sintesi diverse (e diverse sono infatti le UGen coinvolte: `SinOsc, Blip, HPF, Dust, Formant`) sono correlate attraverso la stessa frequenza (`f`): l'ampiezza di ognuno di essi è controllata da un generatore pseudo-casuale (contenute in `arr`). Il segnale è poi distribuito sul fronte stereo attraverso `Pan2`. Si noti che la selezione delle altezze avviene attraverso generatori pseudo-casuali con frequenza pseudo-casuale. Ne risulta una microcomposizione algoritmica.

```
// Additiva in senso lato:
// 4 UGens intonate intorno a una freq
// mix e pan pseudo-casuale
{
    var arr = Array.fill(4, {LFNoise1.ar(1, add:0.15, mul:0.15)}) ;

    f = LFNoise0.ar(LFNoise0.ar(
        SinOsc.kr(0.25, 0, 0.75, 1).unipolar.round(0.0625),
        add:0.95, mul: 0.95),
        add: 48, mul:12).round.midicps; // 24 semitoni, 36-60 MIDI
    Pan2.ar(
        Mix.new([
            SinOsc.ar(f, mul:arr[0]),
            Blip.ar(f, mul:arr[1]),
            RLPF.ar(Dust.ar(f*0.2), f, mul:arr[2]),
            Formant.ar(f,mul:arr[3]),
    ])
        , LFNoise1.ar(0.2, mul:1)
) }.scope;
```

La sintesi additiva si rivela particolarmente adatta per i suoni ad altezza determinata, in cui è sufficiente la definizione di un numero limitato di armoniche particolarmente evidenti. In effetti, per costruire uno spigolo (come quello di un'onda quadra, o di un onda triangolare o a dente di sega) sono necessarie teoricamente infinite componenti sinusoidali. A tal proposito, SC prevede specifici generatori, ad esempio `Pulse` genera onde impulsive (tra cui l'onda quadra) e `Saw` onde a dente di sega.[9]. Una somma di sinusoidi, per quanto numerose esse possano essere, produce un segnale aperiodico, ma non "rumoroso", ed è il risultato tipico è un effetto di "liquidità".

[9] Ancora, `Blip.ar(freq, numharm, mul, add)` genera un numero `numharm` controllabile di componenti armoniche di un fondamentale `freq`, dotate tutte della stessa ampiezza.

```
// Riempire lo spettro con sinusoidi: frazioni di semitono
(
x = {|base = 30| // 30 MIDI = rombo
    var comp = 350; // numero dei componenti
    var res = 0.1; // risoluzione in semitoni
    Mix.new( Array.fill(comp,
        { arg i;
            SinOsc.ar(freq: (base+(i*res)).midicps,
                phase: 2pi.rand, mul: 4.0/comp )}) // fase random
)}.scope ;
)
// cambiando l'altezza base
x.set(\base , 50)
```

L'esempio crea 350 sinusoidi in parallelo (è un bel numero...): il numero degli stessi è controllato da `comp`. La strategia di riempimento gestisce le frequenze attraverso la notazione midi. Questo assicura la possibilità di equispaziare in altezza (percepita) le componenti (e non in frequenza). La variabile `res` controlla la frazione di semitono che incrementa in funzione del contatore `i` a partire dalla altezza di base, `base`. Si noti come storicamente il principale problema della sintesi additiva sia stata la complessità di calcolo richiesta. Infatti un segnale complesso richiede la definizione di almeno 20 armoniche componenti, di ognuna delle quali è necessario descrivere l'inviluppo in forma di spezzata. Come si vede, il problema computazionale è ormai di scarso rilievo, anche perché SC è particolarmente efficiente. La sintesi di suoni percussivi, nello spettro dei quali si evidenzia una preponderante componente di rumore (cioè di parziali non legate alla serie armonica di Fourier), richiederebbe (al fine di ottenere un risultato discreto) un numero di componenti elevatissimo. In realtà, è necessario un altro approccio, quale ad esempio quello della sintesi sottrattiva.

8.3.2 Sintesi sottrattiva

Nella sintesi sottrattiva il segnale in input è generalmente un segnale complesso, dotato di uno spettro molto ricco. Questo segnale subisce un processo di filtraggio in cui si attenuano le frequenze indesiderate, enfatizzando soltanto

alcune regioni (più o meno estese) dello spettro. Più in generale, un filtro è un dispositivo che altera lo spettro di un segnale in entrata. Un filtro agisce cioè enfatizzando o attenuando determinate frequenze del segnale: una modifica dello spettro determina a livello percettivo un cambiamento nel timbro del suono. I parametri fondamentali di un filtro sono: il tipo, la frequenza di taglio/centrale, l'ordine. Si riconoscono usualmente quattro tipi di filtri: passa-basso (*low-pass*), passa-alto (*high-pass*), passa-banda (*band-pass*) e elimina-banda (*band-reject*, o *notch*). I quattro tipi sono schematizzati in Figura **8.12**.

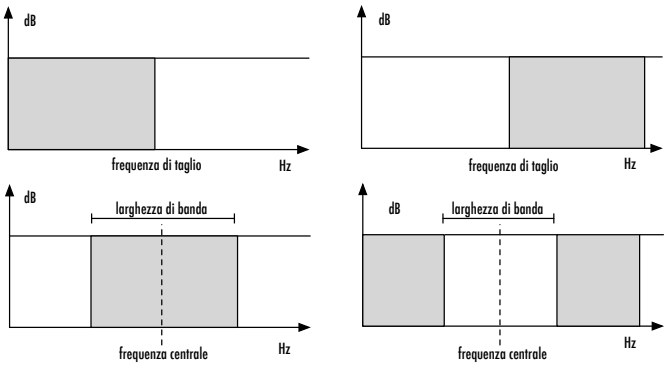


Fig. 8.12 Tipi di filtro: passa-basso, passa-alto, passa-banda, elimina-banda. L'area grigia rappresenta l'intervallo di frequenze che il filtro lascia passare.

In un filtro passa-basso o passa-alto ideali, data una frequenza di taglio, tutte le frequenze rispettivamente superiori o inferiori a questa dovrebbero essere attenuate a 0. Allo stesso modo, in un filtro passa-banda o elimina-banda ideali, data una banda di frequenze, tutte le frequenze rispettivamente esterne o interne alla banda dovrebbero essere attenuate a 0. La frequenza di taglio è perciò quella frequenza a partire dalla quale viene effettuato il filtraggio. Nei filtri passa- o elimina-banda si definiscono sia la larghezza di banda (*bandwidth*) che la frequenza centrale (*cut frequency*): data una regione dello spettro, la prima ne misura la larghezza, la seconda la frequenza al centro. Ad esempio, in un filtro che passa tutte le frequenze nell'intervallo $[100, 110]$ Hz, la larghezza di banda è 10 Hz, la frequenza centrale è 105 Hz. Quelli rappresentati in figura sono comportamenti di filtri ideali. La differenza tra mondo ideale e mondo reale è evidente se si valuta:

```
{ LPF.ar(WhiteNoise.ar, freq: 1000) }.freqscope ;
```

La UGen `LPF` è un filtro passa-basso (acronimo di "Low Pass Filter"), e `freq` indica la frequenza di taglio. Poiché la sorgente è un rumore bianco (si noti il patching), il risultato visibile dovrebbe assomigliare alla Figura **8.12**, passa-basso. In realtà, l'attenuazione è sempre progressiva, e proprio la ripidità della curva è un indicatore della bontà del filtro (intuibilmente, più l'inviluppo spettrale è ripido dopo la frequenza di taglio, meglio è). Poiché filtri che rispondano ai requisiti dell'idealità non esistono, si considera come frequenza di taglio quella a cui il filtro attenua di 3 dB il livello d'ampiezza massimo. Se perciò il passaggio tra la regione inalterata e quella attenuata dal filtro è graduale, un ultimo parametro diventa rilevante: la pendenza della curva. Quest'ultima, misurata in dB per ottava, definisce l'ordine del filtro. Ad esempio, un filtro del I ordine presenta una attenuazione di 6 dB per ottava, uno del II di 12 dB, del III di 18 dB e così via. La UGen `LPF` implementa un filtro passa-basso del II ordine. Per incrementare la ripidità della curva, tipicamente un segnale può subire ricorsivamente più filtraggi, come nell'esempio seguente:

```
{ LPF.ar(
    in: LPF.ar(in:WhiteNoise.ar, freq: 1000),
    freq: 1000)
}.freqscope ;
```

Gli spettri risultanti dai due esempi sono rappresentati in Figura **8.13**.

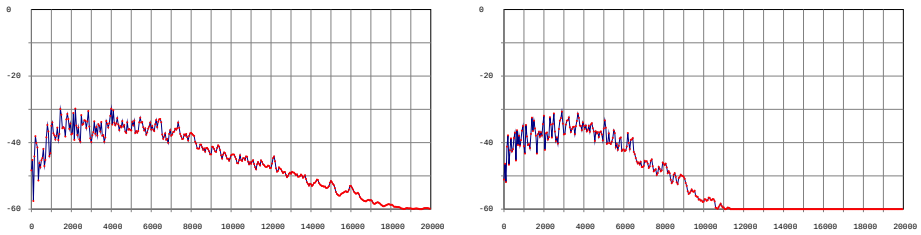


Fig. 8.13 Filtro `LPF`, spettri: singolo filtro e cascata di due.

Se si pensa al caso discusso del rumore bianco, si nota come una parte dell'energia (quella relativa alle frequenze "tagliate") sia attenuata (idealmente a zero). Questa diminuzione riduzione dell'energia complessiva di un segnale risulta, nel dominio del tempo, in una modifica dell'ampiezza del segnale, con una conseguente diminuzione del "volume" del suono risultante. Tipicamente, più il filtraggio è consistente, maggiore è il decremento dell'energia. Ragion per cui spesso è necessario incrementare in uscita l'ampiezza del segnale, compensando l'energia persa (un processo tipicamente detto *balancing*).
Un altro parametro di rilievo in un filtro passa-banda è il cosiddetto "Q". Il Q intuitivamente, rappresenta il grado di risonanza di un filtro. Più precisamente:

$$Q = \frac{f_{centrale}}{larghezzaBanda}$$

Q è cioè il rapporto tra la frequenza centrale e la larghezza di banda. Esprimere il funzionamento del filtro attraverso Q permette di tenere in conto il problema della percezione della frequenza. Mantenere Q costante lungo tutto lo spettro vuol dire infatti adeguare la larghezza di banda all'altezza percepita (all'intervallo musicale). Ad esempio: se

$$f_{centrale} = 105$$
$$f_{acuta} = 110$$
$$f_{grave} = 100$$

allora

$$Q = \frac{105}{110-100} = 10,5$$

Se si mantiene Q costante, e si incremente la frequenza centrale del nostro filtro a 10.500 Hz, si ottengono come estremi 11.000 Hz e 10.000 Hz. La larghezza di banda è passata da 10 Hz a 1000 Hz, conformemente con la percezione dell'altezza. Dunque, $Q \propto risonanza$, perché se Q è elevato, la banda è (percettivamente) stretta e diventa progressivamente percepibile un'altezza precisa. Il parametro Q indica allora la "selettività" del filtro, la sua risonanza. La sintesi sottrattiva è computazionalmente piuttosto efficiente ma allo stesso tempo consente di rappresentare sistemi acustici complessi. Se la cassa armonica di una chitarra si comporta come un risonatore (vale a dire, opera un filtraggio selettivo attenuando certe frequenze ed enfatizzandone altre), in molti strumenti acustici una fonte di eccitazione subisce più filtraggi successivi. Se si pensa a un flauto, la sorgente è il soffio, cioè un rumore bianco, mentre il tubo costituisce il filtro (principale). Un caso tipico è quello della voce umana, nella quale l'eccitazione glottidale passa attraverso il filtraggio successivo di tutti i componenti

dell'apparato fonatorio, le cui cavità possono perciò essere rappresentate come una serie di filtri che trasformano progressivamente il segnale sorgente. Proprio per questo la sintesi sottrattiva è il metodo alla base delle tecniche standard di simulazione artificiale della voce, dal metodo VOSIM alla cosiddetta codifica a predizione lineare (*Linear Predictive Coding*[10]). Nella circostanza, usuale nella sintesi sottrattiva, in cui si impieghino più filtri, il filtraggio può avvenire in parallelo o in serie: nel primo caso più filtri operano sullo stesso segnale simultaneamente (analogamente a quanto avveniva per il banco di oscillatori); nel secondo caso, i filtri sono invece collegati in cascata, e il segnale in entrata passa attraverso più filtri successivi. Seppur in termini non del tutto esatti, la sintesi sottrattiva può essere pensata come un procedimento simmetrico a quella additiva: se in quest'ultima si parte generalmente da forme d'onda semplici (sinusoidi) per ottenere segnali complessi, nella sintesi sottrattiva si parte invece da un segnale particolarmente ricco per arrivare a un segnale spettralmente meno denso. Come sorgenti possono essere utilizzati tutti i generatori disponibili, a parte evidentemente le sinusoidi. In generale, sono di uso tipico segnali spettralmente densi. Ad esempio, il rumore bianco, ma anche quello colorato, tendono a distribuire energia lungo tutto lo spettro e ben si prestano ad essere punti di partenza per una tecnica sottrattiva. In Figura **8.14** la larghezza di banda di quattro filtri (con frequenze centrali 400, 800, 1200, 1600 Hz) che operano in parallelo su un rumore bianco viene diminuita da 1000, a 10, fino a 1 Hz, trasformando il rumore in un suono armonico.

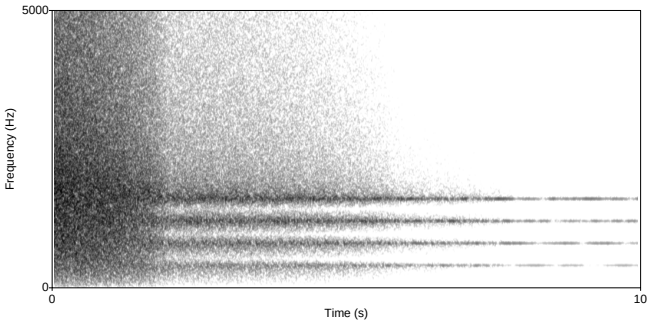


Fig. 8.14 Filtraggio: da rumore bianco a spettro armonico

[10] Da questo punto di vista queste tecniche possono anche essere inserite nella categoria della modellazione della sorgente.

Per questioni di efficienza computazionale, le UGen specializzate nel filtraggio prevedono come argomento in SC non Q ma il suo reciproco, indicato come `rq`. Intuibilmente, varrà allora la relazione per cui minore è `rq` più stretto è il filtraggio (maggiore la risonanza). L'esempio seguente implementa la stessa situazione di filtraggio armonico in SC: `a` è un array che contiene 10 valori da utilizzare come frequenze centrali per il filtro passa-banda `BPF`. Poiché l'array viene passato come argomento a `BPF` ne consegue una espansione multicanale: ne risultano 10 segnali. Di essi, soltanto due sono udibili (nel caso la scheda audio sia stereo), quelli che risultano dal filtraggio a 100 e 200 Hz. Attraverso `scope` si possono osservare tutti i segnali e notare come il rumore bianco in entrata faccia risuonare ogni filtro intorno alla sua frequenza centrale. L'esempio successivo è uguale, ma invia i segnali ad un oggetto `Mix`: il segnale complessivo, che risulta dalla somma dei dieci filtraggi sul rumore bianco le cui frequenze sono armoniche di 100 Hz è udibile, e visibile come forma d'onda e spettro (`scope` e `freqscope`).

```
// Filtraggio che risulta in uno spettro armonico
(
var sound = {
    var rq, i, f, w;
    i = 10; rq = 0.01; f = 100;     // rq = reciproco di Q -> bw/cutoff
    w = WhiteNoise.ar;             // sorgente
    a = Array.series(i, f, f);
    // a = [ 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000 ]
    m = BPF.ar(w, a, rq, i*0.5);
} ;
sound.scope(10) ;    // see 10 channels of audio, listen to first
)

(
var sound = {
    var rq, i, f, w;
    i = 10; rq = 0.01;f = 100;        // rq = reciproco di Q -> bw/cutoff
    w = WhiteNoise.ar;                // sorgente
    a = Array.series(i, f, f);
    // a = [ 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000 ]
    n = BPF.ar(w, a, rq, i*0.5);
    m = Mix.ar(n);                // mixDown
} ;
sound.scope ;                // mixdown: forma d'onda
sound.freqscope ;                // mixdown: spettro
)
```

Un aspetto particolarmente interessante della sintesi sottrattiva sta nella somiglianza che presenta con il funzionamento di molti strumenti acustici, nei quali una fonte di eccitazione subisce più filtraggi successivi. Ad esempio, la cassa armonica di una chitarra si comporta come un risonatore: vale a dire, opera un filtraggio selettivo attenuando certe frequenze ed enfatizzandone altre. Un discorso analogo vale per la voce umana, nella quale l'eccitazione glottidale passa attraverso il filtraggio successivo di tutti i componenti del cavo orale: proprio per questo la sintesi sottrattiva è il metodo alla base della tecnica standard di simulazione artificiale della voce, la cosiddetta codifica a predizione lineare (*Linear Predictive Coding*). Nell'esempio successivo, è possibile confrontare alcune sorgenti e verificare il risultato del filtraggio già discusso selezionando quella che interessa. Il codice fa anche uso del routing attraverso i bus: il synth di filtraggio (il primo) legge dal bus `~bus`, sul quale scrivono i synth successivi (che, si ricordi, sono posizionati prima in termini di ordine di esecuzione).

```
// Sorgenti

~bus = Bus.audio(s, 1) ; // bus dove instradare

(
// un synth che filtra
~filt = {|in|
    var input = In.ar(in) ; // legge dal bus di ingresso
    var i = 10; q = 0.01; f = 100;
    a = Array.series(i, f, f);
    n = BPF.ar(input, a, q, i);
    m = Mix.ar(n)*0.2;
    [m, Silent.ar(1), input] // scrive su tre bus per visualizzare
}.scope ;
)

~filt.set(\in , ~bus) ; // legge da ~bus

// vari synth sorgente scrivono su ~bus
~source = { Out.ar(~bus, Pulse.ar(100, 0.1, mul: 0.1)) }.play
~source.free ; // e vengono deallocati
~source = { Out.ar(~bus, Dust2.ar(100, mul: 1) ) }.play
~source.free ;
~source = { Out.ar(~bus,LFNoise0.ar(100, 0.1, mul: 1) ) }.play
~source.free ;
~source = { Out.ar(~bus, WhiteNoise.ar(mul: 0.1)  ) }.play
~source.free ;
~source = { Out.ar(~bus, BrownNoise.ar(mul: 0.1)  ) }.play
~source.free ; ~filt.free ;
```

Si noti che la visualizzazione usa i primi tre bus, per far sentire il segnale filtrato e far vedere la sorgente sul canale 3 (assumendo in linea di massima che il lettore usi una scheda audio stereo, e che quindi solo i primi due bus siano pubblici). Il bus 1 (il secondo) è occupata da un segnale generato dalla UGen `Silent`, la quale semplicemente scrive un segnale di ampiezza 0.

8.3.3 Analisi e risintesi: Phase vocoder

Come si è visto, il momento fondamentale sia nella sintesi additiva che nella sintesi sottrattiva è il controllo dei parametri che determinano rispettivamente l'inviluppo delle componenti armoniche e le caratteristiche dei filtri impiegati. Da questo punto di vista, sintesi additiva e sottrattiva sono analoghe, ma simmetriche, tant'è che alcuni sistemi di sintesi sono stati implementati sia come banchi di oscillatori che come banchi di filtri. Nelle tecniche di sintesi basate su analisi e risintesi sono proprio i dati di controllo che vengono derivati dall'analisi di un segnale preesistente ad essere utilizzati nella fase di sintesi. Il processo viene usualmente scomposto in tre fasi:

1. creazione di una struttura dati contenente i dati ricavati dall'analisi;
2. modifica dei dati d'analisi;
3. risintesi a partire dai dati d'analisi modificati.

Se questa è l'architettura generale, molte sono comunque le possibilità di realizzazione. Il caso del Phase Vocoder (PV) è particolarmente interessante e permette di introdurne l'implementazione in SC. Nel PV, il segnale viene tipicamente analizzato attraverso una STFT ("Short Time Fourier Transform"). In un'analisi STFT il segnale viene suddiviso in *frame* (finestre, cioè segmenti di segnale), ognuno dei quali passa attraverso un banco di filtri passa-banda equispaziati in parallelo tra 0 e sr (la frequenza di campionamento). Il risultato complessivo dell'analisi di tutti i frame è, per ogni filtro, l'andamento di una componente sinusoidale di frequenza pari alla frequenza centrale del filtro stesso.
In sostanza:

1. si scompone ogni frame del segnale originale in un insieme di componenti di cui si determinano i valori di ampiezza e frequenza;
2. da queste componenti si costruiscono gli inviluppi relativi all'ampiezza di ogni frequenza per tutti i segnali sinusoidali: l'ampiezza, la fase e la frequenza istantanee di ogni componente sinusoidale sono calcolate interpolando i valori dei frame successivi.

Gli inviluppi superano perciò i limiti del singolo frame e possono essere impiegati per controllare un banco d'oscillatori che riproducono per sintesi additiva il segnale originale. Tipicamente, una implementazione particolarmente efficiente della STFT è la FFT (Fast Fourier Transform). Se il file d'analisi non viene modificato, la sintesi basata su FFT riproduce in maniera teoricamente identica il segnale originale, anche se in realtà si verifica nel processo una certa

perdita di dati. Il PV sfrutta l'aspetto più interessante dell'analisi FFT: la dissociazione tra tempo e frequenza. È così possibile modificare uno dei due parametri senza per questo alterare l'altro. Più in generale, è possibile modificare autonomamente i parametri di controllo di tutte le componenti. Un'altra possibilità di grande rilievo è l'estrazione degli inviluppi di soltanto alcune delle componenti.

Essendo il passo di analisi necessariamente antecedente alla sintesi (e costoso computazionalmente), le implementazioni classiche prevedono la scrittura su file dei dati d'analisi e la successiva importazione ed elaborazioni dei dati per la risintesi. L'utilizzo in tempo reale richiede nell'implementazione in SC di allocare un buffer in cui vengono scritti progressivamente i dati d'analisi mano a mano che vengono calcolati da un segnale sorgente. In particolare, la dimensione del buffer (che deve essere una potenza di 2 per questioni di efficienza computazionale) corrisponde alla dimensione della finestra d'analisi (il frame). In sostanza, ogni finestra prelevata sul segnale viene memorizzata nel buffer: ogni nuova finestra sostituisce la precedente. I dati memorizzati nel buffer sono il risultato della conversione dal dominio del tempo al dominio della frequenza effettuata sulla finestra (detto un po' approssimativamente, lo spettro istantaneo della finestra, che conta come un'unica unità di tempo). Questa operazione è svolta dalla UGen `FFT`, che effettua appunto una Fast Fourier Transform sulla finestra. I dati conservati nel buffer (appunto, una sorta di istantanea spettrale) possono essere a quel punto manipolati opportunamente attraverso un insieme di UGen estremamente potenti, il cui prefisso è `PV_` (Phase Vocoder). Tipicamente la manipolazione lavora sullo stesso buffer, sostituendo i dati precedenti. Il buffer contiene comunque ancora dati che rappresentano il segnale (la finestra prelevata) nel dominio della frequenza (l'istantanea spettrale). Per essere inviati in uscita, i dati devono perciò essere convertiti dal dominio della frequenza a quello del tempo (dallo spettro alla forma d'onda): l'operazione è svolta dalla UGen `IFFT`, ovvero "Inverse Fast Fourier Transform". In sostanza, l'intero processo prende la forma seguente:

$$\text{segnale in entrata} \rightarrow \text{FFT} \rightarrow \text{PV\_...} \rightarrow \text{IFFT} \rightarrow \text{segnale in uscita}$$

L'elemento `PV_` è in teoria opzionale, nel senso che i dati possono essere risintetizzati senza che siano stati rielaborati. Evidentemente una simile situazione non è particolarmente utile, ma permette di spiegare il processo:

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

(
SynthDef("noOperation", { arg soundBuf, out = 0 ;
    var in, chain ;
    var fftBuf = LocalBuf(2048, 1) ;
    in = PlayBuf.ar(1, soundBuf, loop:1) ;
    chain = FFT(fftBuf, in) ;     // time --> freq
    Out.ar(out,
        IFFT(chain) // freq --> time
    );
}).play(s,[\soundBuf , b]) ;
)
```

Nell'esempio, il file audio di default di SC viene caricato in un buffer `b` e letto dal synth, che converte il segnale dal dominio del tempo a quello della frequenza via `FFT`, per poi riconvertirlo senza modifiche via `IFFT`. il segnale `in`, che risulta dalla lettura del buffer via `PlayBuf`, viene trasformato nel dominio della frequenza dalla UGen `FFT` (si noti il nome tipico assegnato alla variabile, `chain`, ad indicare che si tratta di un concatenamento di finestre in successione). I dati d'analisi vengono memorizzati nel buffer `fftBuf`. Quest'ultimo viene creato da una UGen apposita, `LocalBuf`, utile quando un buffer è locale ad un synth, come in questo caso in cui esso copre una funzione di servizio interna[11]. Si noti la dimensione del buffer pari a 2048, cioè ad una potenza di 2 (2^{11}). Quindi in uscita a `Out` viene semplicemente inviato il segnale risultante dalla trasformazione al contrario (da frequenza a tempo) operata da `IFFT` su `chain`. In un mondo ideale, non ci sarebbe perdita di dati tra analisi e risintesi ed il segnale in uscita sarebbe una ricostruzione perfetta di quello in entrata: nel mondo reale si producono invece alcune trasformazioni del segnale indotte dalla catena di elaborazioni, tipicamente ridotte, ma potenzialmente importanti, soprattutto nel caso di segnali molto rumorosi.

L'insieme delle UGen `PV_` è molto esteso, e molto potente. Qui di seguito verranno soltanto presentate un paio di applicazioni.

[11] Si noti che `LocalBuf`, come anche `FFT` e `IFFT`, non sono invocati con i metodi `*ar` e `*kr` che pertengono infatti al dominio del tempo.

Si consideri l'esempio seguente: la synthDef realizza un "noise gate", cioè lascia passare il segnale al di sopra di una certa ampiezza selezionabile con il mouse.

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

3  SynthDef(\noiseGate , { arg soundBuf ;
4      var sig;
5      sig = PlayBuf.ar(1, soundBuf, loop:1);
6      sig = sig.abs.thresh(MouseX.kr(0,1)) * sig.sign;
7      Out.ar(0, sig);
8  }).play(s, [\soundBuf , b]) ;

10 // cosa succede?
11 s.scope(1) ;
```

Il codice di rilievo è quello di riga 6. La parte negativa del segnale è ribaltata in positivo attraverso `abs`, quindi `tresh` lascia inalterati i campioni la cui ampiezza è superiore alla soglia (gestita da `MouseX`), mentre restituisce il valore 0.0 se il campione è al di sotto della soglia (cioè: azzera il segnale al di sotto di una certa soglia). In questo modo, tutta la parte di segnale inferiore ad un certo valore di soglia viene eliminata. Tipicamente ciò permette di eliminare i rumori di fondo, che si presentano come segnali con ampiezza ridotta e costante. Tuttavia, il segnale, che ora è unipolare, suona più o meno all'ottava superiore a causa del ribaltamento (le semionde sono infatti ribaltate sul quadrante positivo). Il metodo `sign` restituisce -1 se il campione ha valore negativo e 1 se questo è positivo. Moltiplicando ogni campione per il segno del campione originale la parte di segnale negativa ribaltata ritorna ad avere segno negativo. Vale la pena di osservare come, secondo prassi usuale in SC, lo stesso risultato può essere ottenuto attraverso differenti implementazioni[12]. Ad esempio:

[12] Un suggerimento per la prima implementazione proviene da Nathaniel Virgo, la seconda è stata proposta da Stephan Wittwer, entrambi attraverso la SC mailing list.

```
SynthDef("noiseGate2", { arg soundBuf = 0;
    var pb, ir, mx;
    mx = MouseX.kr;
    pb = PlayBuf.ar(1, soundBuf, loop: 1);
    ir = InRange.ar(pb.abs, mx, 1);
    Out.ar(0,  pb * ir)
}).play(s, [\soundBufnum , b]);

// cosa succede?
s.scope(1) ;
```

La UGen `InRange.kr(in, lo, hi)` restituisce 1 se il campione ha valore incluso nell'intervallo $[lo, hi]$, 0 altrimenti. Nell'esempio, `lo` è gestito da `MouseX`, mentre `hi` è pari a 1. Dunque, il segnale in uscita `ir` è una sequenza di 0 e 1 in funzione del valore assoluto del campione: se il valore assoluto (`abs`) è superiore a `low` –qui assegnato alla variabile `mx`–, allora l'output è 1. Il segnale sarà necessariamente uguale o inferiore a 1, che è il massimo nella forma normalizzata. Il segnale originale `pb` viene moltiplicato per `ir`, che lo azzera se l'ampiezza è inferiore alla soglia `mx` (poiché lo moltiplica per 0) e lo lascia inalterato se è superiore alla soglia (poiché lo moltiplica per 1).

Si noterà che, seppur efficace, il noise gate produce importanti "buchi" nel segnale. L'esempio seguente sfrutta un altro approccio e realizza quello che si potrebbe definire un (noise) gate spettrale. La UGen `PV_MagAbove` opera sui dati d'analisi assegnati alla variabile `chain`: lascia invariati i valori d'ampiezza dei frame (tipicamente detti "bin"), la cui ampiezza è superiore ad un valore di soglia (qui controllato dal `MouseX`), mentre azzera quelli la cui ampiezza è inferiore. Come si nota, è possibile in questo modo eliminare completamente lo sfondo rumoroso le cui componenti spettrali si situano ad un ampiezza ridotta rispetto ad altri elementi (la voce) in primo piano. L'operazione è molto potente, ma non indolore, perché le stesse componenti sono rimosse da tutto lo spettro (anche dalla figure in primo piano, per così dire)[13].

[13] Si noti che la soglia è definita in `MouseX` in termini di "magnitudine". A onor del vero, nella documentazione di SC non sono chiarissimi il significato e l'escursione utile dell'unità di misura, e nell'esempio si procede empiricamente.

```
// Gate spettrale
b = Buffer.read(s, Platform.resourceDir +/+ "sounds/a11wlk01.wav") ;

(
// Controllo con il mouse della soglia spettrale
SynthDef("magAbove", { arg bufnum, soundBuf ;
    var in, chain;
    var fftBuf = LocalBuf(2048, 1) ;
    in = PlayBuf.ar(1, soundBuf, loop: 1);
    chain = FFT(fftBuf, in);
    chain = PV_MagAbove(chain, MouseX.kr(0, 40, 0));
    Out.ar(0, 0.5 * IFFT(chain));
}).play(s,[\soundBuf , b]);
)
```

Un'altra applicazione che ha un suo corrispettivo nel dominio della frequenza è il filtraggio. Intuibilmente, se il segnale finestrato è scomposto in frequenza, è allora possibile eliminare i bin relativi alle frequenze che si vogliono filtrare (ad esempio, tutti quelli sopra una certa soglia: filtro passa-basso, tutti quelli sotto: filtro passa-alto). Nell'esempio seguente il mouse controlla l'argomento `wipe` di `PV_BrickWall`. Se `wipe` è pari a 0, non c'è effetto, se è < 0 la UGen lavora come un filtro passa-basso, se > 0 come un passa-alto. L'escursione possibile è compresa nell'intervallo $[-1.0, 1.0]$ (il che richiede di determinare empiricamente il valore pari alla frequenza desiderata). Il filtro spettrale prende giustamente il nome di "brickwall" perché la ripidità in questo caso è verticale. La synthDef fa anche uso, prima di inviare il segnale a `Out`, della UGen `Normalizer` che analizza il segnale lo riscala in modo che il valore di picco sia pari al parametro `level` (qui $= 1$), operazione che prende appunto il nome di "normalizzazione". In questo modo si neutralizza l'effetto di perdita di energia associata al filtraggio.

```
b = Buffer.read(s, Platform.resourceDir +/+ "sounds/a11wlk01.wav") ;
s.freqscope ; // cosa succede?
(
//  Filtro FFT
SynthDef("brickWall", { arg soundBuf ;
    var in, chain;
    var fftBuf = LocalBuf(2048, 1) ;
    in = PlayBuf.ar(1, soundBuf, loop: 1);
    chain = FFT(fftBuf, in);
    chain = PV_BrickWall(chain, MouseX.kr(-1.0,1.0, 0));
    // -1.0 --> 0.0: LoPass ; 0.0 --> 1.0: HiPass
    Out.ar(0, Normalizer.ar(IFFT(chain), level:1));
}).play(s,[\soundBuf , b]);
)
```

Una delle applicazioni più tipiche del PV consiste nell'elaborazione della frequenza indipendentemente dal tempo. Tra le molte UGen disponibili, l'esempio seguente sfrutta `PV_BinShift`. `PV_BinShift(buffer, stretch, shift)` permette di traslare e di scalare tutti i bin, rispettivamente di un fattore `shift` e `stretch`. La traslazione (`shift`) corrisponde ad uno spostamento nello spettro: ad esempio, dato un spettro di tre componenti $[100, 340, 450]$, una traslazione $+30$ risulta in $[130, 370, 480]$. Nel codice seguente, l'argomento `stretch` riceve valore pari a 1 (nessuna trasformazione), mentre `shift` varia in funzione di `MouseX` nell'intervallo $[-128, 128]$. Osservando lo spettro e –in particolare– muovendo il mouse verso destra, si nota come l'intero spettro si sposti lungo l'asse delle frequenze.

```
b = Buffer.read(s, Platform.resourceDir +/+ "sounds/a11wlk01.wav") ;
// cosa succede?
s.freqscope ;
s.scope ;

(
SynthDef(\fftShift , {  arg soundBuf ;
    var in, chain;
    var fftBuf = LocalBuf(2048, 1) ;
    in = PlayBuf.ar(1, soundBuf, loop: 1);
    chain = FFT(fftBuf, in);
    chain = PV_BinShift(chain, 1, MouseX.kr(-128, 128) );
    Out.ar(0, 0.5 * IFFT(chain).dup);
}).play(s, [\soundBuf , b]);
)
```

In maniera analoga, l'argomento `stretch` permette di moltiplicare il valore dei bin: come si vede osservando la visualizzazione dello spettro nell'esempio seguente (in particolare muovendo progressivamente il mouse da sinistra a destra), dalla variazione di `scale` nell'intervallo $[0.25, 4]$ (con incremento esponenziale) consegue una progressiva espansione spettrale.

```
SynthDef("fftStretch", { arg soundBuf ;
    var in, chain;
    var fftBuf = LocalBuf(2048, 1) ;
    in = PlayBuf.ar(1, soundBuf, loop: 1);
    chain = FFT(fftBuf, in);
    chain = PV_BinShift(chain,MouseX.kr(0.25, 4, \exponential ) );
    Out.ar(0, 0.5 * IFFT(chain).dup);
}).play(s,[\soundBuf , b]);
```

Due note per concludere.

Il metodo `dup(n)`, definito su `Object`, restituisce un array che contiene n copie dell'oggetto stesso. Il valore di default di n è 2. Negli esempi precedenti, `dup` inviato a `IFFT` restituisce un array `[IFFT, IFFT]`, che dunque impone una espansione multicanale: essendo il valore predefinito di $n = 2$, il segnale in uscita è stereo.

Infine, si sarà notato come i valori di controllo per le UGen PV non siano espressi in Herz, ma alcune volte in forma normalizzata (è il caso di `PV_BrickWall`), altre in escursioni che dipendono dall'implementazione delle UGen stesse (si pensi a $[-128, 128]$ nel caso di `PV_BinShift`). Come si è accennato, la situazione, per cui i valori dipendono fondamentalmente dall'implementazione, non è chiarissima e deve tipicamente essere risolta in maniera empirica.

8.4 Modellazione della sorgente

All'origine delle tecniche di modellazione della sorgente (comunemente dette anche "sintesi per modelli fisici") vi è l'assunto che il segnale possa essere calcolato sulla base di equazioni che descrivano il comportamento fisico (acustico e meccanico) dei corpi che interagiscono nella sua produzione. Se nelle tecniche di modellazione spettrale –ad esempio in sintesi additiva– si poteva approssimare il *timbro* di un clarinetto attraverso una opportuna miscela di sinusoidi che approssimassero il tipico spettro dello strumento formato solo da armoniche dispari, nella sintesi per modelli fisici si tratta invece di realizzare un modello matematico del *funzionamento* di un clarinetto, nel quale le variabili fondamentali sulla base delle quali generare il segnale siano, ad esempio, il volume del tubo, il comportamento dell'ancia, la densità del legno.

Le tecniche di modulazione fisica tipicamente sono computazionalmente costose, ad esempio a causa del grande numero di calcoli necessari per risolvere le equazioni lineari di alcuni approcci. In SC tipicamente non si implementano attraverso composizione di UGen in una synthDef, ma attraverso UGen che le implementano internamente come primitive. SC non ha una dotazione particolarmente ricca di questi modelli nella distribuzione ufficiale, ma molti approcci sono disponibili come estensioni aggiuntive ad opera di altri sviluppatori. Ad esse si rimanda, così come ad altri testi più generali sulla sintesi per una introduzione teorica. I due esempi che seguono sono però utili come introduzione ad aspetti della sintesi per modelli fisici e a SC.

Un modello fisico rappresenta una formalizzazione di un dispositivo acustico. Ovviamente, la formalizzazione può avvenire a risoluzione diversa. Ad esempio, molti dispositivi acustici possono essere descritti come sistemi di filtri in varie combinazioni. Il grado di precisione in cui il comportamento dei filtri

è descritto determina l'accuratezza della modellazione. Da questo punto di vista, la sintesi sottrattiva (se ne è accennato in precedenza) è spesso tangente a quella per modelli fisici. Ad esempio, l'apparato fonatorio umano produce i vocoidi (le vocali) modulando la struttura di un tubo complesso, formato da più componenti (condotto oro-faringeo, cavo orale, cavo nasale), in modo da variare le sue caratteristiche filtranti applicate alla sorgente acustica: la vibrazione delle pliche vocali. Gli studi di fonetica acustica hanno dimostrato che una caratteristica fondamentale dell'organizzazione delle vocali è la presenza di formanti, cioè di bande spettrali in cui l'energia è concentrata. Lo spettro di ogni vocale comprende da 4 a 5 formanti: le prime due tipicamente servono per il riconoscimento linguistico, le altre per quello del parlante. Si noti che le formanti sono indipendenti dalla frequenza fondamentale: in questo modo, una "a" pronunciata da un bambino è riconosciuta appunto in quanto tale anche se la frequenza fondamentale è molto più acuta di quella della fonazione di un adulto. I fonetisti acustici hanno allora definito uno spazio bidimensionale i cui due assi rappresentano le prime due formanti. Su di esso, possono essere individuati punti (in realtà sono zone, veri e propri bersagli acustici dei parlanti) che rappresentano le diverse vocali. Ogni lingua definisce i propri punti utili in termini di numero e posizione. Ad esempio, in Figura **8.15** è riprodotto lo spazio vocalico dell'italiano.

Ora, la sintesi della voce è ovviamente un capitolo enorme della ricerca in ambito di elaborazione dei segnali digitali. Si potrebbe però proporre un modello semplificato, "cartonificato", del dispositivo di produzione vocalica a partire dal risultato, cioè dalla presenza delle due formanti. L'esempio successivo utilizza come segnale sorgente un'onda a dente di sega. Si tratta di una forma non distante da quella prodotto dalle pliche vocali (ecco una forma di modellizzazione). La frequenza è assegnata attraverso `fund`, il cui valore predefinito è 70 Hz, un valore tipico in un parlante uomo. Quindi il segnale sorgente `source` è filtrato attraverso due filtri passa-banda intorno alle due frequenze che rappresentano le formanti `f1` e `f2`. Per ridurre l'importanza della perdita di energia, i due segnali vengono normalizzati prima di essere mixati.

[14] Da Ferrero, Genre, Böe, Contini, *Nozioni di fonetica acustica*, Torino, Omega, 1978.

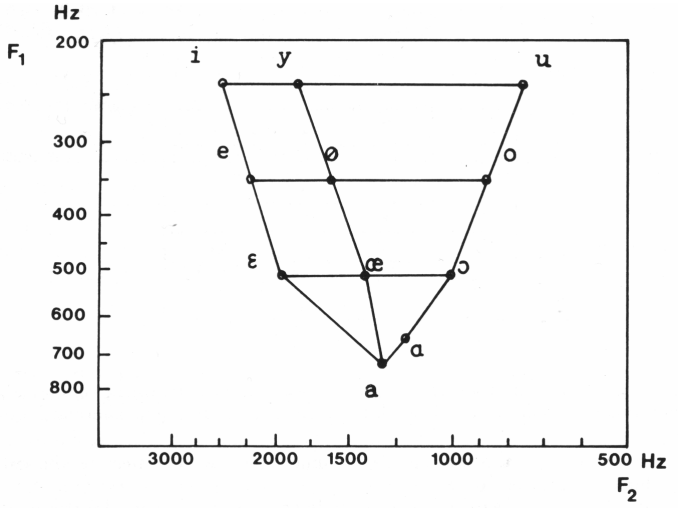


Fig. 8.15 Spazio vocalico per l'italiano[14].

```
// Generatore di spettri vocalici
SynthDef(\vocali , { arg f1, f2, fund = 70, amp = 0.25 ;
    var source = Saw.ar(fund); // sorgente
    var vowel =
      Normalizer.ar(BPF.ar(source, f1, 0.1))
      +
      Normalizer.ar(BPF.ar(source, f2, 0.1))
      * amp ; // volume generale
    Out.ar(0, vowel.dup)
}).add ;
```

Uno dei punti chiave della sintesi per modelli fisici è il controllo. In altri termini la modellazione della sorgente richiede una importante fase di definizione dei parametri del modello, in assenza dei quali anche il modello migliore non produce risultati utili. A partire dalla synthDef, il prossimo esempio costruisce una interfaccia grafica che definisce uno spazio di controllo per l'utente (cioè, lo spazio delle formanti f2-f1, rappresentate con una scala lineare in Hz).

```
(
~synth = Synth(\vocali ) ;

d = 600; e = 400;
w = Window("Spazio formantico", Rect(100, 100, d+20, e+20) ).front ;
Array.series(21, 2500, 100.neg).do{|i,j|
    StaticText(w, Rect(j*(d/21)+5, 10, 30,10 ))
    .font_(Font("Helvetica", 8))
    .string_(i.asString)
} ;
Array.series(14, 200, 50).do{|i,j|
    StaticText(w, Rect(d, j*(e/14)+20, 30, 10 ))
    .font_(Font("Helvetica", 8))
    .string_(i.asString)
} ;
u = UserView(w, Rect(0, 20, d, e)).background_(Color.white) ;

~vow = (
    // dati che definiscono punti "sensibili" in Hz
    \i :[2300, 300], \e : [2150, 440], \E : [1830, 580],
    \a : [1620, 780], \O : [900, 580], \o : [730, 440],
    \u : [780, 290],\y : [1750, 300],\oe : [1600, 440],
    \OE : [1400, 580]
) ;
f = {|v, f2, f1| StaticText(u,
    Rect(f2.linlin(500, 2500, d, 0),
        f1.linlin(200, 800, 0, e)-18, 40,40))
    .string_(v).font_(Font("Helvetica", 18))
} ;
~vow.keys.asArray.do{|key|
    var f2 = ~vow[key][0] ;
    var f1 = ~vow[key][1] ;
    f.value(key.asString, f2, f1)
} ;

w.acceptsMouseOver = true ;
u.mouseOverAction_({|v,x,y|
    ~synth.set(
        \f2 , x.linlin(0, d, 2500, 500).postln,
        \f1 , y.linlin(0, e, 200, 850).postln,
    )
    })
)
```

Il codice è volutamente un po' "sporco" per lasciare l'idea del lavoro di prototipazione incrementale tipico in SC. Ad esempio, ci sono un po' di variabili ambientali che potrebbero più opportunamente essere convertite in variabili locali (`d, e, f, w, u`). La riga 2 istanzia il synth `~synth` che si occuperà della sintesi. Alle variabili `d` e `e` sono assegnati i valori di larghezza e lunghezza della finestra `w`, la quale contiene una `UservView` (una veduta grafica controllabile dall'utente, 16). I due array (6-15) in realtà servono solo come metodi per generare le etichette degli assi. Le righe 36-24 definiscono le azioni collegate all'interazione con l'utente via mouse. Prima è necessario indicare che tutti i figli della finestra `w` accettano eventi di tipo mouse (36), quindi viene specificato cosa succede quando il mouse viene mosso sulla vista `u`: come si vede, il metodo `mouseOverAction` permette di accedere alla posizione (in pixel) del cursore con gli argomenti `x` e `y`. Questi sono utilizzati per impostare il valore di $f1$ e $f2$ attraverso l'opportuno scaling. La funzione `f` è pensata per generare etichette sullo spazio, a partire dall'etichetta e dalla posizione in Hz delle due formanti passate. In questo modo è possibile generare un insieme di etichetta (30-34) a partire da un insieme di dati (`~vow`).

Un'altra tecnica che tipicamente è associata alla modellizzazione della sorgente, sebbene non possa essere considerata a pieno titolo un metodo basato su un modello fisico, è l'algoritmo di Karplus-Strong. La tecnica può essere paragonata a quella della "waveguide" o guida d'onda, che simula il passaggio di un segnale sorgente in un dispositivo fisico rappresentato nei termini di una sequenza di filtri e di ritardi (causati dalle riflessioni delle onde sulle superfici interessate). Se il paradigma della guida d'onda si basa a tutti gli effetti su un modello fisico del risonatore in questione, l'algoritmo Karplus-Strong è molto più astratto ma include comunque l'idea di filtraggio e di ritardo. Di semplice implementazione, il metodo si rivela tuttavia efficace nel generare suoni ascrivibili o alla classe delle corde pizzicate o a quella delle percussioni. Come si vede in Figura **8.16**, un generatore di rumore riempie una tabella (come quelle viste per la sintesi wavetable) una sola volta per ogni evento (a ogni "nota"): a questo punto l'interruttore si sposta per ricevere i dati dal modificatore, disconnettendo il generatore. Si legge il valore del primo elemento della tabella, sulla base del quale si manda in output il primo campione, mentre l'elemento viene modificato e inserito retroattivamente all'inizio della tabella attraverso una linea di ritardo. In sostanza, la tabella viene continuamente riscritta per l'intera durata dell'evento: per questo si parla di "recirculating wavetable". Facendo un esempio, se la tabella è di 256 punti e il tasso di campionamento è 44.100, allora la tabella viene letta (e modificata) circa 172 volte in un secondo ($\frac{sr}{n_{tab}} = \frac{44.100}{256} = 172,265$).

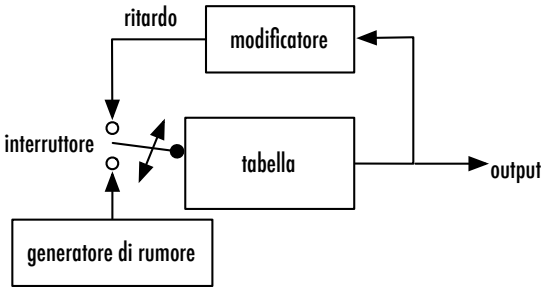


Fig. 8.16 Algoritmo di Karplus-Strong.

Poiché la tabella viene riscritta molte volte, la modificazione può essere molto rapida. Il cuore dell'operazione sta nel tipo di modificazione subita dal valore che viene poi riscritto nella tabella: realizzando un opportuno filtraggio, si può ottenere così il tipico suono di una corda pizzicata, che parte con un attacco intenso per decadere immediatamente in ampiezza e in ricchezza spettrale, così come un suono percussivo, che può variare da un rullante a un tom. Due sono le conseguenze della riscrittura circolare della tabella. In primo luogo, il segnale risultante non è necessariamente un rumore, come potrebbe far pensare l'immissione di una sequenza pseudo-casuale di valori nella tabella, ma può essere un suono "intonato": il segnale è infatti periodico poiché la tabella viene riletta con un determinato periodo, e i valori della tabella dopo qualche ciclo rimangono pressoché costanti (un comportamento analogo all'oscillatore digitale descritto in precedenza; nell'esempio numerico di prima, la frequenza risultante sarà allora di circa 172 Hz). In secondo luogo, è possibile simulare con grande semplicità l'inviluppo dinamico e spettrale di una corda pizzicata. La porzione dell'attacco corrisponde alla immissione della serie di numeri pseudo-casuali nella tabella (inizio dell'evento), che producono lo spettro diffuso tipico del rumore bianco. SC implementa l'algoritmo nella UGen `Pluck`.

```
{ Pluck.ar(
    in:WhiteNoise.ar(0.25),
    trig:Impulse.kr(1),
    delaytime:60.midicps.reciprocal,
    maxdelaytime:60.midicps.reciprocal,
    decaytime:MouseY.kr(1,20,1),
    coef:MouseX.kr(-0.75, 0.75))
}.play ;
```

Come si vede nell'esempio, i parametri includono la sorgente di eccitazione che riempie la tabella (qui un rumore bianco) e un trigger che indica appunto il momento di riempimento (qui generato ogni secondo da `Impulse`). La frequenza è gestita attraverso la specificazione del ritardo, che diventa il periodo della frequenza richiesta (infatti è specificato come l'inverso, `reciprocal`, della frequenza `60.midicps`). Il parametro `maxdelaytime` determina la dimensione del buffer interno, che dovrà essere almeno pari al periodo della frequenza ricercata (qui vi è allineato). I due parametri più interessanti sono il tempo di decadimento che costituisce un indicatore della risonanza (espresso come attenuazione di 60 dB in secondi) e `coeff`, che controlla il filtro interno applicato ai valori che vengono riscritti nella tabella. Sperimentando, si nota come si abbiano agevolmente effetti di corda (dalla corda libera a quella muta) fino ad effetti percussivi più o meno intonati. Un esercizio interessante può consistere nell'implementazione di una tecnica ispirata all'algoritmo KS.

```
(
// Karplus-Strong
SynthDef(\ks , {
    arg freq = 440, amp = 1, out = 0, thresh = -90, decrease = -0.25 ;
    var baseFreq = 48.midicps ; // freq di base, riferimento arbitrario
    var buf, index, sig, num = 2, scale = decrease.dbamp ;
    var samples = 44100/baseFreq ;
    var actualValue ;
    buf = LocalBuf(samples) ;
    // tabella random = un rumore bianco
    buf.set(Array.fill(buf.numFrames, { 2.0.rand-1 }));
    index = Phasor.ar(
        trig:Impulse.ar(buf.numFrames/SampleRate.ir),
        rate:freq/baseFreq,
        // -> tasso di lettura
        start:0, end:buf.numFrames, resetPos:0);
    actualValue = BufRd.ar(1, buf, index) ;
    // lettura circolare, fino a che livello scende sotto soglia
    Out.ar(out, actualValue*amp) ;
    DetectSilence.ar(actualValue, thresh.dbamp, 0.1, 2) ;
    // riscrittura circolare
    sig = Array.fill(num, {|i| BufRd.ar(1, buf, index-i)}).sum/num*scale;
    BufWr.ar(sig, buf, index) ;
}).add ;
)

Synth(\ks , [\freq , 50.midicps, \amp , 0.5, \out ,0 ]) ;
```

La synthDef utilizza una frequenza di riferimento arbitraria, definita come `baseFreq` (un do sotto il rigo) e assume che il tasso di campionamento sia 44.100: costruisce perciò un buffer locale `buf` di un numero di punti `samples` pari a $\frac{44.100}{baseFreq}$ (7-8). In altri termini, il buffer rappresenta un numero di punti che corrisponde (al tasso di campionamento scelto) al periodo della frequenza di base. Il buffer viene riempito di valori pseudo-casuali nell'escursione $[-1.0, 1.0]$, rappresentando perciò un rumore bianco. Il problema ora è leggere la tabella. La UGen `Phasor` produce un segnale utile per leggere una tabella: in particolare, è un segnale a rampa, cioè una progressione di valori che può essere utilizzata come indice di una tabella. Qui `Phasor` viene innescata da un `Impulse` il cui periodo è pari alla dimensione del buffer. In pratica, ogni volta che il buffer finisce, parte un trigger che lo re-innesca (e si riparte da 0, come specificato da `resetPos`). L'argomento `rate` indica a quale velocità il buffer verrà letto, e dipende

dal rapporto tra frequenza di base e frequenza `freq` desiderata. Se la `freq` è il doppio di `baseFreq` allora il buffer viene letto alla velocità doppia. Si ricordi che il tasso di `Phasor` è *ar, dunque, ad ogni nuovo campione, un nuovo indice verrà calcolato. L'indice `index` viene utilizzato (18) per leggere il campione relativo (assegnato a `actualValue`) dal buffer attraverso la UGen `BufRd`, che viene inviato in uscita e di cui si verifica il valore di ampiezza attraverso `DetectSilence`: quando questo scende sotto la soglia `thresh` il synth viene deallocato (`doneAction:2`, si tratta di un synth "notale"). Fin qui si ha solo lettura del buffer. Invece attraverso le due ultime righe il buffer viene modificato. Viene calcolato un valore `sig` dato dalla somma dei valori contenuti nel buffer tra `index-i` e `i`, diviso per il numero dei campioni e scalato per `scale`. Con $num = 2$, vengono considerati il campione attuale e il precedente. Si supponga 0.5 e 1 e `scale` pari a 0.97162795157711 (si ricordi che il valore in dB, predefinito a -0.25 è convertito in ampiezza lineare): allora $\frac{0.5+1.0}{2} \times 0.97162795157711 = 0.72872096368283$. Il nuovo valore viene scritto al posto del precedente (1), con un effetto di attenuazione. La synthDef è rappresentata in Figura **8.17**, dove è stata eliminata parte della rappresentazione del buffer per ragioni di visibilità.

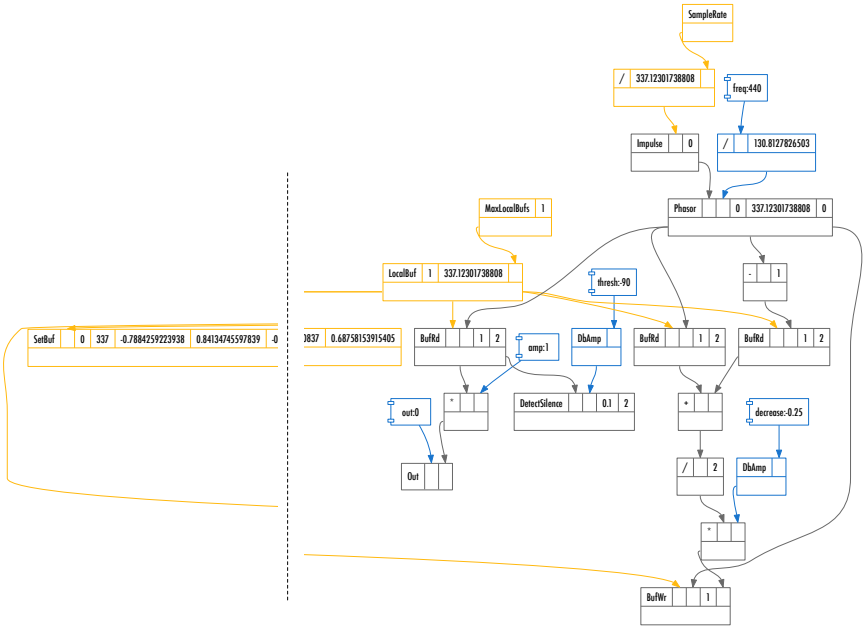


Fig. 8.17 Diagramma della synthDef `ks`.

Nell'esempio discusso, il modificatore consiste in un filtro passa-basso elementare detto "averaging filter": il valore del campione in output è ottenuto calcolando la media (*average*: di qui il nome) tra i valori del campione in input attuale e di quello precedente. Come si può intuire, la conseguenza è che durante il corso dell'evento la forma dell'onda tende progressivamente a smussarsi, causando un'attenuazione delle componenti di alta frequenza del segnale. Il risultato della modificazione circolare è allora un filtraggio passa-basso che produce nel segnale una progressiva diminuzione della ricchezza spettrale, unitamente al tipico inviluppo dinamico con attacco intenso e decadimento immediato delle corde pizzicate.

8.5 Metodi basati sul tempo

A questa categoria è possibile ascrivere un insieme di metodi che considerano la sintesi a partire dalla rappresentazione del segnale nel dominio del tempo. Si tratta di metodi tipicamente computazionali, sia perché spesso operano ad una risoluzione temporale molto elevata non gestibile in un contesto analogico, sia perché si basano direttamente sulla natura numerica del segnale come sequenza discreta di valori.

8.5.1 Sintesi granulare

La sintesi granulare muove dal presupposto che il suono possa essere pensato non solo in termini ondulatori ma anche in termini corpuscolari. È usuale il paragone con i procedimenti pittorici del "pointillisme": la somma di frammenti sonori di dimensione microscopica viene percepita come un suono continuo, come la somma di minuscoli punti di colore puro origina la percezione di tutti gli altri colori. Si tratta allora di definire grani sonori la cui durata vari tra 1 e 100 millisecondi, ognuno dei quali caratterizzato da un particolare inviluppo d'ampiezza, che influisce notevolmente sul risultato sonoro. Come intuibile, se per sintetizzare un secondo di suono sono necessari ad esempio 100 grani, non è pensabile la definizione deterministica delle proprietà di ognuno di essi.

È perciò necessario ricorrere a un controllo statistico, che ne definisca i valori medi per i diversi parametri, ad esempio durata, frequenza, forma d'onda, inviluppo: più complessivamente deve essere stabilita la densità media dei grani nel tempo. La necessità di un controllo d'alto livello rispetto al basso livello rappresentato dai grani richiede così la definizione di un modello di composizione: non a caso la sintesi granulare è una delle tecniche che ha maggiormente stimolato un approccio algoritmico alla composizione. Ad esempio si può organizzare la massa dei grani seguendo la legge di distribuzione dei gas perfetti (Xenakis), oppure attraverso una tipologia degli ammassi che deriva da quella meteorologica delle nuvole (Roads), o infine immaginare le sequenze di grani come dei flussi di particelle di cui determinare le proprietà globali tramite "maschere di tendenza" (Truax). In generale, va notato che per la sintesi granulare è richiesto al calcolatore un notevole lavoro computazionale per controllare i valori dei parametri relativi alla sintesi di ogni singolo grano. Esistono molti modi differenti di realizzare la sintesi granulare, che rispondono a obiettivi anche assai diversi tra loro: in effetti, seppur con qualche approssimazione, può essere descritta come "sintesi granulare" ogni tecnica di sintesi che si basi sull'utilizzo di segnali (quasi) impulsivi.

```
(
{
// sintesi granulare sincrona
var baseFreq = MouseX.kr(50, 120, 1).midicps ;
var disp = MouseY.kr  ;
var strata = 30 ;
var minDur = 0.05, maxDur = 0.1 ;

Mix.fill(strata,
    {
    // sorgente
    SinOsc.ar(
        freq: baseFreq +
            LFNoise0.kr(20)
            .linlin(-1.0,1.0, baseFreq*disp*1.neg, baseFreq*disp),
        mul: 1/strata)
    // inviluppo
    * LFPulse.kr(
            freq:
            LFNoise0.kr(20)
            .linlin(-1.0,1.0, minDur.reciprocal, maxDur.reciprocal))
})
}.freqscope
)
```

Nell'esempio precedente è implementato un approccio in sincrono alla sintesi granulare. In sostanza, nell'algoritmo di sintesi una sinusoide è inviluppata in ampiezza da un'onda quadra, generata da `LFPulse`: poiché `LFPulse` genera un segnale unipolare (la cui ampiezza è inclusa in $[0.0, 1.0]$), quando l'ampiezza dell'onda quadra è > 0, lascia passare il segnale, altrimenti lo riduce ad ampiezza 0. Il risultato è una "finestrazione" del segnale. La durata di ogni finestra dipende dal ciclo dell'onda quadra di `LFPulse`, ed è gestito da `minDur` e `maxDur`. Le durate specificano il periodo della finestra, dunque la frequenza di `LFPulse` sarà $\frac{1}{T}$. In questo caso, l'inviluppo del grano è appunto dato da un segmento lineare positivo (il ciclo positivo dell'onda quadrata). Come si vede, attraverso `Mix` viene mixato insieme un numero `strata` di segnali: in ognuno, `baseFreq` (controllata da `MouseX`) gestisce la frequenza di base a cui viene aggiunto un valore pseudo-casuale tra 0 e `disp` (controllata da `MouseY`), generato da `LFNoise0`, che rappresenta una percentuale normalizzata della frequenza di base (se $baseFreq = 100$ e $disp = 0.5$, allora la frequenza dell'oscillatore varierà

nell'intervallo $[100 - 50, 100 + 50]$). Si noti che ogni `SinOsc` ha associato un suo `LFNoise0`.

L'approccio precedente è implementabile in tempo reale, e parte dall'assunto di convertire, per così dire, segnali continui in segnali impulsivi a partire da una operazione di finestrazione. Si noti che la generazione di valori pseudo-casuali è necessariamente affidata a UGen (`LFNoise0`), che richiedono a loro volta un tasso di aggiornamento, qui impostato arbitrariamente a 20 Hz.

Un altro approccio possibile prevederebbe di pensare ad ogni singolo grano come ad un evento di taglia "compositiva". In quel caso, si potrebbe cioè trattare ogni grano come un evento sottoposto a processi di scheduling. Sebbene possibile, tipicamente si tratta di un approccio computazionalmente molto oneroso a causa della quantità di grani-eventi necessari.
Un uso tipico delle tecniche granulari sfrutta la granularità non tanto per la sintesi *ab nihilo* quanto piuttosto per l'elaborazione: tipicamente, l'operazione è chiamata "granulazione", e come intuibile consiste nel sottoporre a scomposizione in grani e successiva ricombinazione un segnale preesistente. In qualche misura, l'operazione è intrinsecamente in tempo differito, perché è necessario avere a disposizione una parte di segnale per poterla scomporre e ricomporre. Ciò vuol dire che, in tempo reale, è necessario quantomeno riempire un buffer (una locazione di memoria temporanea) per poterlo granulare.

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

SynthDef(\grainBuf , { arg sndbuf;
    Out.ar(0,
        GrainBuf.ar(2,
            trigger: Impulse.kr(MouseX.kr(10, 20)),
            dur: MouseY.kr(0.01, 0.1),
            sndbuf: sndbuf,
            rate: LFNoise1.kr.range(0.5, 2),
            pos: LFNoise2.kr(0.1).range(0.0, 1.0),
            interp:1,
            pan:LFNoise1.kr(3)
    ))
}).add ;

x = Synth(\grainBuf , [\sndbuf , b]) ;
```

L'esempio precedente è una semplificazione di quello proposto nel file di help della UGen `GrainBuf`. Nel buffer `b` viene memorizzato il consueto segnale d'esempio, sul quale `GrainBuf` opera. La sintassi di `GrainBuf` è già esplicitata attraverso le keywords nel codice. Il primo argomento specifica il numero dei canali, poiché la UGen può generare direttamente segnali multicanale. Il secondo argomento è il segnale di triggering che innesca un nuovo grano: nell'esempio è una sequenza di impulsi con frequenza tra 10 e 20 Hz. L'argomento `dur` determinata la durata di ogni grano (tra 0.01 e 0.1, cioè tra 10 e 100 ms), il successivo il buffer (nello specifico, sarà `b`). Gli argomenti `rate, pos, interp, pan, envbufnum` controllano rispettivamente il tasso di lettura del campione (come in `PlayBuf`), la posizione di lettura del buffer (normalizzata tra $[0.0, 1.0]$, il metodo di interpolazione per il pitch-shifting dei grani (qui, `1`, ad indicare assenza di interpolazione), la posizione sul fronte stereo (in caso di segnali stereo, come in `Pan2`), un buffer (opzionale) in cui è contenuto un inviluppo da utilizzare per inviluppare i grani. Alcuni argomenti sono controllati da generatori di tipo `LF`. Il metodo `range` permette di scalare un segnale all'interno dell'intervallo specificato dai due valori passati: è decisamente più intelligibile del lavoro (istruttivo) con `mul` e `add`[15]. La UGen `LFNoise2` produce un rumore a bassa frequenza con interpolazione quadratica tra un valore e il successivo, a differenza che nel caso di `LFNoise1` in cui l'interpolazione è lineare. Il risultato è un segnale molto "arrotondato".

8.5.2 Tecniche basate sulla generazione diretta della forma d'onda

Vale la pena chiudere la breve panoramica su alcune delle tecniche di sintesi più usate o storicamente più importanti introducendo una famiglia che mette in luce la natura strettamente numerica del segnale digitale. Alcuni approcci sviluppati a partire già dagli anni '60 (da Koenig, Xenakis, Brün) si caratterizzano per un approccio non modellistica. In altri termini, non prendono come obiettivo un modello acustico, ma esplorano direttamente la manipolazione del segnale come sequenza numerica. Si tratta di approcci che si potrebbero definire "costruttivisti". Ad esempio, si può pensare al segnale come una sequenza di

[15] Il metodo assume che il segnale sia nell'escursione normalizzata, quindi non funziona se gli argomenti `mul` e `add` vengono specificati.

segmenti di retta, ognuno composto da *n* campioni, che possono essere montati algoritmicamente in sequenza. Oppure si può pensare ad operazioni di inversione, permutazione, trasformazione su blocchi di campioni. Alla base di questi approcci vi è la volontà di esplorare la natura intrinsecamente numerica del segnale generato da un calcolatore, in qualche modo il suo "timbro" caratteristico. Spesso i segnali risultanti sono particolarmente ricchi di asperità, per così dire, sia in termini di componenti acute dello spettro, sia in termini di discontinuità temporali. Nel capitolo dedicato ai fondamenti della sintesi è stato discusso un esempio che opera in questa prospettiva: si tratta della "distorsione per permutazione", in cui blocchi di campioni vengono permutati, così che la sequenza di blocchi `[a,b,c,d]` risulti in `[b,a,d,c]`. L'implementazione in tempo reale di un simile esempio si scontra con un limite intrinseco di SC, e di praticamente tutti i software per la sintesi audio. Infatti, nel server audio non è possibile accedere al livello del singolo campione. Dunque, non è possibile una implementazione *diretta* a lato server degli algoritmi descritti in quel capitolo in riferimento ad operazioni a lato linguaggio sugli array, perché sul server si hanno a disposizione soltanto segnali (non singoli campioni) che risultano da UGen[16]. Una implementazione ispirata alla distorsione per permutazione è la seguente[17]:

[16] In realtà è possibile in qualche misura, ad esempio sfruttando le UGen `LocalIn` e `LocalOut` e impostando la dimensione del block size a 1. Ancora, un'altra opzione sarebbe utilizzare le UGen `DbufRd` e `DbufWr` che permettono di scrivere e leggere a certe condizioni un singolo campione. Si tratta di opzioni avanzate per questa trattazione.

[17] Una implementazione esaustiva ed efficiente si può fare ovviamente attraverso la programmazione di una UGen specifica, si veda Giacomo Valenti, Andrea Valle, Antonio Servetti, *Permutation synthesis*, Atti del XX CIM Colloquio di Informatica Musicale., Roma 2014.

```
(
b = Buffer.alloc(s, s.sampleRate * 1.0, 1); // buffer mono di 1 secondo

SynthDef(\sin , {|freq = 100, buf|
    RecordBuf.ar(SinOsc.ar(freq), buf, loop:1)}).add ;

SynthDef(\perm , { arg buf, permFreq = 10 ;
    var trig = Impulse.ar(permFreq) ;
    var startPos = LFSaw.ar(permFreq, iphase:1,mul:0.5, add:0.5); // 0 - 1
    var periodInSamples = permFreq.reciprocal*SampleRate.ir ;
    var sig =
    BufRd.ar(1, buf, Phasor.ar(trig, 1,
        startPos*periodInSamples,
        startPos*periodInSamples*2, startPos*periodInSamples)
    ) ;
    Out.ar(0, sig) ;
}).add ;
)

// scrittura su un buffer circolare
x = Synth(\sin , [\buf , b])
// dopo
y = Synth.after(x, \perm , [\buf , b]) ;

// controllo
y.set(\permFreq , 20) ;
x.set(\freq , 1000) ;
// cosa succede
s.scope ;
```

Inizialmente viene allocato un buffer `b`, che contiene 1 secondo di segnale al tasso di campionamento del server. Il buffer verrà utilizzato per registrare un segnale e renderlo disponibile per ulteriori elaborazioni. Per poter permutare due blocchi di campioni `[a,b]` in tempo reale è necessario registrare i due blocchi per poi invertirli e mandarli in uscita. Di qui l'utilità del buffer `b` che viene utilizzato dal synth `x` (riga 21). La synthDef relativa sfrutta `RecordBuf`, una UGen che prende come argomenti rispettivamente un segnale di ingresso (qui generato da `SinOsc`), un buffer su cui il segnale viene registrato (`b`) e l'argomento `loop`, che qui ha valore `1`, indicando una scrittura circolare sul buffer. In altri termini, il segnale viene continuamente registrato circolarmente sul buffer. Si noti che manca `Out`. Infatti, il segnale viene semplicemente registrato

su un buffer, non instradato su un bus. In altri termini, l'idea è di registrare il segnale sinusoidale per potervi effettuare della manipolazioni successive: la registrazione ovviamente introduce un ritardo pari alla lunghezza del buffer nel momento in cui si modificano i parametri di `SinOsc`, perché il segnale modificato è disponibile su `b` dopo la registrazione.

La synthDef seguente implementa un approccio permutativo. Invece che in termini di numero di campioni il controllo avviene in termini di frequenza. La UGen `LFSaw` genera una rampa che viene normalizzata unipolarmente in $[0, 1.0]$ attraverso gli argomenti `mul` e `add`. Inoltre, l'argomento `iphase`, che gestisce la fase iniziale del segnale vale `1`: nella UGen ciò indica che si parte da metà del ciclo. Dunque, se si considerano le ampiezze, da 0.5 a 1, quindi da 0.0 a 0.5. La frequenza è quella della permutazione desiderata, `permFreq`. Per comodità viene calcolato il periodo in campioni di quest'ultima, `periodInSamples`. A questo punto si tratta di leggere il buffer `buf` (che sarà `b` nel caso). Per leggere i valori si usa `Phasor`. La UGen ad ogni `trig` ricevuto avanza di 1 campione (secondo argomento) nella lettura a partire dal terzo argomento fino al quarto. Come si vede, questi due sono calcolati attraverso la moltiplicazione si `startPos` (che risulta da `LFSaw`) per il periodo di permutazione richiesto. Si legge da `startPos` fino ad un altro periodo. L'ultimo argomento di `Phasor` indica da dove ripartire una volta ricevuto il segnale `trig`.

La riga 21 scrive una sinusoide sul buffer, quindi si istanzia il synth permutatore che legge dal buffer. Infine si controllano i parametri della sinusoide e del processo di permutazione.

Il processo è riassunto in Figura **8.18**.

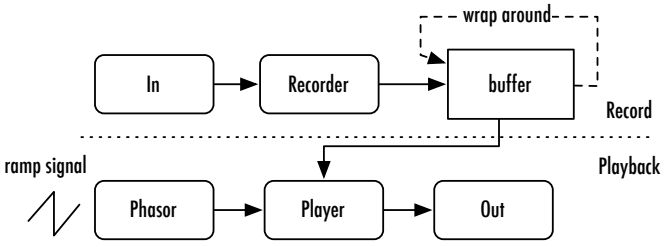


Fig. 8.18 Sintesi per permutazione.

Una elaborazione di questo tipo permette di costruire spettri complessi a partire da sinusoidi (anche se ovviamente altri segnali potrebbero esservi sottoposti)

senza riferimenti acustici, semplicemente operando sulla struttura temporale del segnale in ingresso.

8.6 Conclusioni

Il mondo della sintesi audio nel dominio digitale è sconfinato, e apre grandi possibilità al compositore/sound designer. Gli esempi discussi costituiscono uno sguardo parziale non solo al mondo della sintesi in sé ma anche a quello già disponibile in SuperCollider. In tutte le tecniche di sintesi in realtà il livello forse più importante è quello del controllo delle stesse. Dunque, quanto visto sulla sintesi dovrà essere incorporato in un sistema concettuale più ampio, che includa quanto visto in precedenza: il livello del controllo dei segnali e quello del controllo dell'organizzazione complessiva del materiale sonoro

9 Comunicazione

Quest'ultimo capitolo si propone di discutere rapidamente il problema della comunicazione in SuperCollider. Ovviamente una comunicazione si definisce tra attori: come si è detto fin dagli inizi, SC ne identifica immediatamente due, un client e un server, che finora hanno comunicato in una forma che non è mai stata discussa approfonditamente, perché opportunamente resa trasparente dallo strato linguistico del linguaggio SuperCollider. Tuttavia, non sono pochi i casi in cui in qualche misura è necessario invece definire esplicitamente una modalità di comunicazione tra oggetti diversi e SC. Dopo alcuni esempi di comunicazione tra client e server, il capitolo intende fornire alcune indicazioni preliminari rispetto alla comunicazione tra SC e l'esterno: per questi ultimi casi, gli esempi sono funzionanti ma non del tutto autonomi, perché spesso fanno riferimento a formati, servizi e applicazioni esterni a SC.

9.1 Dal server al client: uso dei bus di controllo

La comunicazione discussa finora riguardava il passaggio di messaggi tra client e server. In una espressione come `s.boot`, l'interprete è il client che converte la sintassi in un messaggio per il server che indica a quest'ultimo di avviarsi. Una situazione analoga vale per tutti quei costrutti che controllano aspetti della sintesi audio (a lato server): la comunicazione è monodirezionale, dal client al server. Si noti che il client demanda tutto il calcolo dell'audio al server, sulla cui attività propriamente non ha alcuna informazione. Tuttavia, è spesso assai utile avere informazioni sull'audio. Un caso banale ma dirimente è quello

del disegno dei segnali audio da parte della GUI. Se il segnale è calcolato a lato server, come può essere visualizzato dalle classi GUI che invece risiedono a lato client? La soluzione sta nella possibilità della comunicazione inversa, dal server al client. Un primo modo in cui questa può essere implementata e sfruttata in SC si basa sull'uso di bus di controllo. Nell'esempio seguente la synthDef è pensata come un synth che genera un segnale di controllo attraverso una generatore di rumore che interpola quadraticamente tra valori pseudo-casuali successivi, `LFNoise2`: ne risulta un segnale particolarmente arrotondato. Le righe 8-10 istanziano un synth `x` a partire dalle synthDef, che viene instradato sul bus di controllo `~bus`. Il synth `y` legge dal bus i valori generati e lì scritti da `x` e li ri-mappa sulla frequenza di una sinusoide. Tutta questa comunicazione, per quanto ovviamente descritta a lato client, avviene tra synth e bus, cioè a lato server. Il blocco successivo (12-31) invece recupera i valori del segnale di controllo per disegnarne una curva in una GUI. Il cuore dell'operazione è la routine `r` che ogni 20 ms (`0.05.wait`, 29) sfrutta il metodo `get` su `~bus`. Questo metodo chiede al server di restituire il valore che questi ha scritto sul bus, cioè il segnale generato da `x`. L'argomento di `get` rappresenta il valore del bus. Si noti che `get` riceve una funzione perché è asincrono: in altri termini, chiede un valore al server, ma non può sapere quando arriverà di preciso. Dunque, la funzione viene eseguita solo quando il server ha risposto alla chiamata. In questo caso, la variabile ambientale (condivisa) `v` memorizza il valore sul bus, il contatore viene incrementato, e a quel punto viene aggiornata la `UserView` `u`. Si noti il costrutto `defer` necessario perché la routine `r` è schedata su SystemClock (implicito). La `UserView` `u` ha associata una funzione `drawFunc` che disegna un cerchio in cui la posizione x dipende da un contatore modulo 500, cioè la larghezza della finestra `u`: quando la si supera, il contatore si azzera, e il prossimo cerco avrà ascissa zero sulla vista. Il valore di y dipende invece dal valore del segnale di controllo, `v`, che viene aggiornata attraverso ad ogni `~bus.get`. Attraverso `u.clearOnRefresh_(false)`, la finestra non viene ripulita, e dunque si assiste alla sovrapposizione dei tracciati una volta che `i` supera la dimensione 500.

```
(
SynthDef(\cntr , {arg in = 0, out = 0, dbGain = 0;
    Out.kr(out, LFNoise2.kr(0.5))
}).add ;
)

(
~bus = Bus.control(s,1) ;
x = Synth(\cntr , [\out , ~bus]) ;
y = {SinOsc.ar(In.kr(~bus).linlin(-1, 1, 48, 72).midicps, mul:0.2)}.play ;

i = 0;  v = 0.0  ;
w = Window.new("Plotter", Rect(10, 10, 500, 500)).front ;
u = UserView(w, Rect(0,0, 500, 500)).background_(Color.grey(0.75)) ;
u.drawFunc_{
    Pen.fillColor = Color.red(0.5);
    Pen.addOval(Rect((i%500), v.linlin(-1.0,1.0, 500,0), 2,2)) ;
    Pen.fill
} ;
u.clearOnRefresh_(false) ;

r = {
    inf.do{
        ~bus.get({ arg amp ;
            v = amp ;
            i = i+1 ;
            {u.refresh}.defer
        });
        0.05.wait ;
    }
}.fork
)
```

L'esempio seguente sfrutta un meccanismo analogo per visualizzare l'ampiezza del segnale. La UGen `Amplitude` restituisce un segnale di controllo che risulta dalla stima dell'ampiezza del segnale in ingresso ("amplitude follower"). Il segnale di controllo può essere utilizzato direttamente sul server attraverso un patching con altre UGen. Nell'esempio seguente, il synth `a` è un amplitude follower che legge un segnale audio in ingresso sul bus audio `~audio` e scrive il segnale ottenuto dall'analisi dello stesso sul bus di controllo `~ampBus`. Il blocco successivo definisce una finestra su cui, attraverso la funzione `drawFunc`, viene disegnato un cerchio nel centro in cui dimensione e colore dipendono da

quattro variabili ambientali (`~dim, ~hue, ~sat, ~val`, inizializzate nelle righe 10-12). Nelle righe 24-27 è invece definita una routine infinita che definisce il valore di queste ultime pescando dal bus `~ampBus` ad un tasso `~update` e definendo un semplice mapping tra dominio dell'ampiezza (lineare da `Amplitude`, ma qui convertito in dB) e dimensioni in pixel e parametri del colore. Si noti che teoricamente l'escursione in dB dell'ampiezza sarebbe $[-96, 0]$, ma $[-60, 0]$ è più adeguata all'input.

```
~ampBus = Bus.control(Server.local) ;
~audio = Bus.audio(Server.local) ;
a = {
    var sig = In.ar(~audio) ;
    var amp = Lag.kr(Amplitude.kr(sig)) ;
    Out.kr(~ampBus, amp) ;
}.play ;

~update = 0.1 ; // tasso di aggiornamento
~dim = 50 ; // dimensione del cerchio
// parametri del colore
~hue = 0; ~sat = 0.7; ~val = 0.7;
// finestra e funzione disegno
w = Window("tester", Rect(10, 10, 500, 500))
.background_(Color(1,1,1,1)).front ;
w.drawFunc = {
    var oo = 250-(~dim*0.5) ; // per centrare il cerchio
    Pen.addOval(Rect(oo, oo, ~dim, ~dim)) ; // il cerchio
    Pen.color_(Color.hsv(~hue, ~sat, ~val)) ; // colore
    Pen.fill ; // e riempimento
} ;

// una routine infinita per aggiornare la finestra
{
inf.do{
        // per infinite volte vai a vedere il contenuto di ~bus
        ~ampBus.get({ arg amp ; // rappresenta il valore nel ~bus
            // conversione in db
            ~dim = amp.ampdb.linlin(-60, 0, 10, 500) ;
            ~hue = amp.ampdb.linlin(-60, 0, 0.4, 0) ;
            ~sat = amp.ampdb.linlin(-60, 0, 0.8, 1) ;
            ~val = amp.ampdb.linlin(-60, 0, 0.8, 1) ;
            {w.refresh}.defer ; // aggiorna la finestra
        }) ;
        ~update.wait ;
    }
}.fork ;
```

A questo punto si può scrivere sul bus di controllo `~ampBus`. Il primo synth x effettua un test in cui il mouse sull'asse y controlla l'ampiezza di una sinusoide. Come si vede, spostandosi lunga la verticale, la dimensione e il colore del cerchio cambiano. Il synth viene deallocato (7), e la variabile viene assegnata a un nuovo synth, che legge da un buffer. Si noti che il tasso di aggiornamento

cresce (attraverso la diminuzione del suo periodo `~update`, 10), e il synth scrive su due canali audio, `0` e quello privato che viene instradato al synth di analisi a.

```
x = {
    var sig = SinOsc.ar(mul: MouseY.kr(0,1)) ;
    //Out.ar(0, sig) ; // non lo facciamo uscire
    Out.ar(~audio, sig) ; // ma solo scrivere il segnale sul bus
}.play ;

x.free ; // eliminiamo il synth, ma la routine continua

~buf = Buffer.read(Server.local, Platform.resourceDir
    +/+ "sounds/a11wlk01.wav") ;
~update = 0.025 ; // piu' preciso
x = {
    var sig = PlayBuf.ar(1, ~buf, loop:1) ;
    Out.ar([0, ~audio], sig) ; // sul primo canale e sul bus privato
}.play ;
```

Ovviamente i valori del segnali "pescati" da `~bus.get`, e così ormai disponibili a lato client (questo è il punto), non devono per forza essere utilizzati per una grafica.

9.2 Dal server al client: uso dei messaggi OSC

Si è detto finora che client e server comunicano via messaggi OSC. Ma finora, di questi stessi messaggi non se n'è visto neanche uno. Infatti, nell'approccio linguistico privilegiato i messaggi sono nascosto al di sotto di uno strato predisposto dal linguaggio SC. Il protocollo OSC, inizialmente sviluppato per le applicazioni audio (è l'acronimo di Open Sound Control) è ormai uno standard nella comunicazione multimediale tra applicativi di vario uso, dall'audio (SuperCollider, Max/MSP, PD) alla grafica (Processing) e all'elaborazione di immagino (Eyesweb, VVVV), dai linguaggi di programmazione (praticamente tutti, ad esempio Python, Java, Ruby) agli ambienti audio multitraccia avanzati (ad esempio, è supportato da Ardour). È un protocollo che non prevede

una semantica, ma una sintassi dei messaggi. In altri termini, stabilisce *come* i messaggi debbano essere costruiti per essere "ben formati", ma non *cosa* debbano dire. La semantica è definita dall'applicazione che li invia o riceve. Così, scsynth comunica via OSC, ma definisce una propria semantica: stabilisce cioè cosa vogliano dire certi messaggi in termini della sua ontologia di server audio. Non è questa la sede per un tutorial su OSC: si veda piuttosto *Audio e multimedia*, dove sono trattati in maniera esaustiva. Intanto, per avere un'idea della comunicazione client-server si può valutare:

```
1 OSCFunc.trace(true) ;
2 // ora basta
3 OSCFunc.trace(false) ;
```

La classe `OSCFunc` rappresenta il gestore generico della comunicazione OSC a lato client. In altri termini, con `OSCFunc` è possibile definire cosa succede in caso di ricezione da parte di sclang di messaggi OSC. Una funzionalità molto utile è quella definita dal metodo `trace`, che in funzione dell'argomento booleano stampa (o no) sullo schermo tutti i messaggi OSC ricevuti (e dunque potenzialmente interpretabili) da parte di sclang. Se dunque si valuta la riga 1 si ottiene sulla post window una situazione analoga alla seguente:

```
1  OSC Message Received:
2      time: 111609.33037712
3      address: a NetAddr(127.0.0.1, 57110)
4      recvPort: 57120
5      msg: [ /status.reply, 1, 0, 0, 2, 63, 0.031290706247091, 0.16726991534233,
6      44100, 44099.998039566 ]
7
8  OSC Message Received:
9      time: 111610.0269853
10     address: a NetAddr(127.0.0.1, 57110)
11     recvPort: 57120
12     msg: [ /status.reply, 1, 0, 0, 2, 63, 0.037191119045019, 0.12907001376152,
13     44100, 44100.000112071 ]
```

Altri messaggi analoghi seguiranno. Sono i messaggi che regolarmente scsynth invia a sclang per indicare che la comunicazione tra i due è attiva. Dunque, sclang avvisa che un messaggio è stato ricevuto, nel momento `time` da quanto l'interprete è attivo. Il messaggio è inviato dall'indirizzo `127.0.0.1`, `57110`, in cui il primo elemento indica una comunicazione locale dalla stessa macchina, e il secondo la porta dalla quale è inviato: si ricordi che quando scsynth effettua il boot, `57110` è la porta che gli viene assegnata e che viene stampata sullo schermo. Il messaggio è ricevuto sulla porta `57120`, cioè quella assegnata di default a sclang. Questo indica che se si vuole mandare un messaggio a sclang da un'altra applicazione, bisogna utilizzare la porta `57120`. Segue (5) il messaggio vero e proprio, che è caratterizzato da un nome (`/status.reply`, si noti il carattere `/` che apre tutti i messaggi OSC per definizione del protocollo) e da un insieme di elementi, il tutto impacchettato in un array. Si tratta in questo caso del messaggio di scsynth che dice a sclang di essere attivo.

Nel prossimo esempio, viene introdotta la UGen `SendReply`, che appartiene ad una famiglia finora mai incontrata, le UGen che si occupano di spedire messaggi OSC dal server. `SendReply` spedisce messaggi indietro al client ad ogni trigger, qui specificato dal primo argomento che riceve `Impulse`: nel caso, 10 volte al secondo. Il secondo argomento indica invece il nome del messaggio inviato, `/ping`. Il terzo infine, un valore, che qui viene "pescato" dal segnale generato da `WhiteNoise`. In altri termini, ogni decimo di secondo, un messaggio `/ping` che contiene l'ampiezza del rumore bianco generato viene spedito al client. Si noti che la synthDef non genera audio (o meglio, calcola un segnale audio per spedirne l'ampiezza, ma si potrebbe spedire anche un valore numerico costante). Se si tracciano i messaggi con `OSCFunc.trace` e si alloca il synth `x`, si vede cosa succede nella comunicazione. Si tratta ora di utilizzarla. Proprio `OSCFunc` è ciò che serve. Il prossimo esempio reimplementa quello già discusso in precedenza.

```
(
SynthDef(\impl , {arg in = 0, out = 0, dbGain = 0;
    var sig = LFNoise2.ar(0.5) ;
    Out.ar(0, SinOsc.ar(sig.linlin(-1, 1, 48, 72).midicps, mul:0.2)) ;
    SendReply.ar(Impulse.ar(10), '/amp', values: sig)
}).add ;
)

(
x = Synth(\impl ) ;

i = 0;  v = 0.0  ;
w = Window.new("", Rect(10, 10, 500, 500)).front ;
u = UserView(w, Rect(0,0, 500, 500)).background_(Color.grey(0.75)) ;
u.drawFunc_{
    Pen.fillColor = Color.red(0.5);
    Pen.addOval(Rect((i%500), v.linlin(-1.0,1.0, 0,500), 2,2)) ;
    Pen.fill
} ;
u.clearOnRefresh_(false) ;
o = OSCFunc({ |msg|
    v = msg[3] ;
    i = i+1 ;
    {u.refresh}.defer
     }, '/amp');
)
```

Questa volta il segnale `sig` viene utilizzato per controllare `SinOsc` che viene immediatamente instradato sul bus `0`. Quindi `SendReply` spedisce indietro al client un messaggio `/amp` con il valore del segnale `sig`. L'unica cosa nuova nel resto del codice è `OSCFunc`. Partendo dall'ultimo argomento, si vede come risponda soltanto ai messaggi `/amp`. Se arrivassero messaggi con un altro nome, verrebbero ignorati. Dunque, se un messaggio `/amp` è ricevuto, viene valutata la funzione, che ha come argomento il messaggio stesso (qui `msg`) che diventa quindi accessibile all'interno della stessa. Nel messaggio il quarto elemento è il valore di `sig`, che, assegnato a `v`, viene utilizzato da `drawFunc` per disegnare (secondo quanto visto nell'esempio precedente). Si noti che quando si esegue Cmd+. anche il "responder" OSC viene rimosso.

Nel prossimo esempio, la comunicazione avviene in due passaggi, dal server al client e da questo di rimando al server. La synthDef `player` semplicemente legge dal buffer (il solito `b`) e inviluppa il segnale con un inviluppo percussivo, che dealloca il synth. Più interessante la synthDef `listener`. Questa legge dal primo ingresso della scheda audio (su un computer, il microfono) e analizza il segnale attraverso `Onsets`, una UGen che effettua anche un'analisi FFT (come si vede) per rilevare un attacco nel segnale in ingresso. `Onsets` funziona da trigger per `SendReply`. In altri termini, ad ogni attacco, `Onsets` produce un trigger che innesca in `SendReply` l'invio di un messaggio `'/attack'` verso sclang. Il messaggio contiene il valore di `Loudness`, una UGen (a tasso di controllo) basata su FFT (e infatti semplicemente riceve `chain`) che effettua una stima della loudness, cioè del volume percepito, in una scala da 0 a 100. A lato client, `OSCFunc` per ogni messaggio `'/attack'` legge il valore ricevuto (cioè la loudness) e la mappa tra 1 e 2 (25). Questo valore diventa parametro di controllo per l'argomento `rate` di un synth `player` che viene generato (e la comunicazione ora andrà da sclang a scsynth). In sostanza, ad ogni attacco rilevato nell'ingresso microfonico una "nota" viene generata, la cui altezza varia in funzione del volume dell'ingresso.

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

SynthDef(\player , { arg buf, out = 0, rate = 1 ;
    Out.ar(out,
        FreeVerb.ar(in:PlayBuf.ar(1, buf, rate)
            *EnvGen.kr(Env.perc, doneAction:2)*2)
        )
}).add ;

SynthDef(\listener , {
    var sig = SoundIn.ar(0) ;
    var loc = LocalBuf(1024, 1) ;
    var chain = FFT(loc, sig);
    SendReply.kr(
        Onsets.kr(chain, 0.75, \rcomplex ),
        '/attack',
        Loudness.kr(chain));
}).add ;
)

(
OSCFunc({
    arg msg;
    var rate = msg[3].linlin(20, 40, 1, 2) ;
    Synth(\player ).set(\rate , rate, \buf , b) ;
}, '/attack');

Synth(\listener )

)
```

9.3 OSC da e verso altre applicazioni

L'oggetto `OSCFunc` è ovviamente la chiave di volta per la comunicazione OSC con sclang. Quest'ultimo riceve sulla porta 57120. Diventa dunque possibile inviare messaggi da altre applicazioni in rete a sclang. Ad esempio, si

supponga di avere uno smartphone e una rete locale. Ovviamente, è necessario conoscere alcuni dati della rete locale. Nell'esempio, il computer che ospita SC ha indirizzo `172.20.10.3` (un dato recuperabile attraverso le utilità di sistema). Una diffusa applicazione OSC per dispositivi mobili è TouchOSC, che permette di utilizzare l'interfaccia tattile di smartphone e tablet come controller esterno. Sarà dunque necessario impostare dentro TouchOSC l'indirizzo di rete del computer che ospita SuperCollider e la porta a cui inviare messaggi OSC (cioè, `172.20.10.3` e `57120`, dove riceve sclang). Se si utilizza `OSCFunc.trace` si ricavano dati utili. Ad esempio, l'indirizzo da cui vengono spediti i messaggi (nel caso in questione, `172.20.10.1`). Un altro dato interessante è il nome del messaggio, che in TouchOSC non è specificato altrove, ad esempio `'/1/fader1'` è associato a un cursore di una delle schermate GUI disponibili. Il codice seguente intercetta i messaggi in questione e ne stampa tutti i contenuti. Diventa ovvio modificare la funzione per svolgere altre azioni.

```
OSCFunc(
    { arg msg, time, addr, recvPort;
        [msg, time, addr, recvPort].postln;
}, '/1/fader1') ;
```

Ma la comunicazione via OSC non deve necessariamente essere monodirezionale, al contrario può essere bidirezionale, cioè da sclang verso un altro indirizzo: d'altronde è quanto avviene normalmente nella comunicazione tra sclang e scsynth. Il prossimo esempio presenta un nuovo oggetto, specializzato nella comunicazione via rete, `NetAddr`, che permette di specificare un indirizzo in termini di ip e di porta, e di inviare messaggi.

```
~phone = NetAddr.new("172.20.10.1", 9000);

~phone.sendMsg("/1/fader1", 0);
~phone.sendMsg("/1/fader1", 0.5);
```

Nell'esempio `~phone` rappresenta lo smartphone che spediva messaggi via OSC: alla variabile viene assegnato un oggetto `NetAddr` composto dall'indirizzo

(è quello già discusso) e un numero di porta a cui si intende inviare il messaggio: qui è ipoteticamente `9000`, ma l'identificativo dipenderà dalle impostazioni a lato TouchOSC sullo smartphone. A questo punto diventa possibile inviare messaggi OSC all'indirizzo `~phone`: il nome del messaggio è quello che rappresenta l'oggetto grafico cursore che nell'esempio precedente inviava messaggi quando veniva modificato. Ovviamente la semantica del messaggio dipende in questo caso da TouchOSC, il quale permette una comunicazione bidirezionale con lo stesso messaggio: se la posizione del cursore viene modificata, TouchOSC invia un messaggio con il nome e il valore; se TouchOSC invece riceve un messaggio con il nome e il valore allora modifica la posizione del cursore in funzione del valore. La situazione è schematizzata in Figura **9.1**

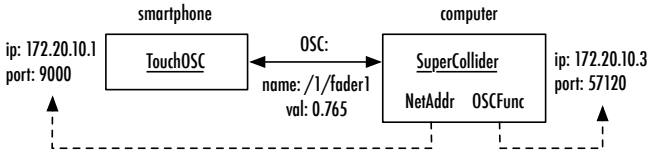


Fig. 9.1 Comunicazione via OSC tra computer e dispositivo mobile.

L'esempio via smartphone è un po' complicato perché la sua implementazione richiede un riferimento ad un oggetto esterno, con le complicazioni che ne conseguono, ad esempio TouchOSC. Tuttavia ha permesso di chiarire il sistema di relazioni via rete che si possono definire. Il prossimo esempio (dalla documentazione di `NetAddr`) funziona di sicuro perché sclang comunica con se stesso, ma diventa più comprensibile solo alla luce di quanto visto.

```
1 n = NetAddr("127.0.0.1", 57120); // 57120 is sclang default port
2 r = OSCFunc({ arg msg, time; [time, msg].postln }, '/good/news', n);

4 n.sendMsg("/good/news", "you", "not you");
5 n.sendMsg("/good/news", 1, 1.3, 77);
```

L'indirizzo locale è sempre `127.0.0.0` o `127.0.0.1`. La riga 1 specifica via `NetAddr` a chi sclang spedirà: ovvero alla stessa macchina, sulla porta `57120`. Quindi (2) viene definito un `OSCFunc`, che risponde su sclang ai messaggi `'/good/news'`.

Di seguito (4, 5) sclang invia all'indirizzo specificato da `NetAddr` due messaggi. Ma allora a sclang arriveranno due messaggi sulla porta `57120`, e dunque la funzione definita in `OSCFunc` verrà eseguita.

9.4 Il protocollo MIDI

La modalità più diffusa per comunicare tra dispositivi audio resta probabilmente ancora il protocollo MIDI, in uso dai primi anni '80. Il protocollo MIDI è infatti lo standard per la comunicazione tra hardware musicale ed è ovviamente implementato nei software dedicati. Come nel caso del protocollo OSC, non è questa la sede per una discussione approfondita, per la quale si rimanda a *Audio e multimedia*. SuperCollider prevede una implementazione del protocollo sostanzialmente completa, in ingresso come in uscita, anche se probabilmente in questo secondo caso meno esaustiva: l'uso più tipico in SC è infatti quello di permettere la comunicazione da dispositivi esterni (controller gestuali, più tipicamente) a SC. Dal punto di vista hardware, i dispositivi MIDI richiedevano inizialmente un connettore specifico, mentre ora tipicamente utilizzano la porta USB.

Per poter usare il MIDI è prima di tutto necessario attivare una componente software dedicata, `MIDIClient`, quindi accedere ai dispositivi disponibili.

```
1 MIDIClient.init ;
2 MIDIIn.connectAll ;
```

La post window tipicamente stampa qualcosa di analogo a quanto segue, che fa riferimento al dispositivo collegato.

```
MIDI Sources:
    MIDIEndPoint("USB X-Session", "Port 1")
    MIDIEndPoint("USB X-Session", "Port 2")
MIDI Destinations:
    MIDIEndPoint("USB X-Session", "Port 1")
MIDIClient
```

A questo punto, l'approccio di SuperCollider rispetto alla comunicazione in ingresso segue (fortunatamente) il modello già visto per OSC: l'omologo di `OSCFunc` a lato MIDI è `MIDIFunc`. Allo stesso modo, `MIDIFunc` è dotato di un metodo `trace`, che permette di tracciare la comunicazione ingresso (e chiaramente, anche l'assenza della stessa). Si tratta di una funzione utile perché, proprio come nel caso di OSC, spesso non sono note le caratteristiche del messaggio che un dispositivo MIDI esterno invierà a SC. Nell'esempio seguente si vede il risultato di `trace`. Tre messaggi vi sono riportati: si vede il tipo (`control, noteOn, noteOff`), il canale sul quale sono ricevuti (si tratta di una caratteristica del MIDI), il numero e il valore. Tipicamente (ma non sempre) il canale è `0` o omogeneo per il dispositivo, mentre il numero consente di discriminare il tipo di messaggio. Il valore in MIDI è per definizione del protocollo nell'intervallo $[0, 127]$, cioè 7 bit (128 valori, non una grande risoluzione in molti casi). Come si nota, i due ultimi messaggi hanno lo stesso numero (`43`) ma valore `127` e `0`. Si tratta infatti di un pulsante premuto sul controller che attiva un messaggio `noteOn` alla pressione e uno `noteOff` al rilascio. Qui i valori `0` e `127` rappresentano semplicemente l'equivalente di on/off. Il fatto che il numero sia lo stesso è appunto perché i due messaggi sono associati sul controller allo stesso oggetto fisico (è comunque una scelta, sensata, di costruzione del dispositivo). Il primo messaggio, di tipo `control`, invece è originato dalla rotazione di una manopola.

```
MIDI Message Received:
    type: control
    src: -2025843672
    chan: 0
    num: 26
    val: 88

MIDI Message Received:
    type: noteOn
    src: -2025843672
    chan: 0
    num: 43
    val: 127

MIDI Message Received:
    type: noteOff
    src: -2025843672
    chan: 0
    num: 43
    val: 0
```

Dunque, con `trace` si possono esplorare i messaggi in entrata, per poi utilizzarli con `MIDIFunc`. Quest'ultima classe è dotata di un insieme di metodi già predisposti in funzione dei tipi più usati tra i messaggi MIDI. Il protocollo distingue infatti tra, ad esempio, messaggi noteOn, noteOff, control, touch, bend, polytouch. Nell'esempio seguente sono illustrati due usi di `MIDIFunc.cc`. In entrambi i casi l'oggetto stampa il messaggio ricevuto. Nel primo, esso risponde a tutti i messaggi di tipo control, nel secondo solo a quelli che hanno numero `26` sul canale `0` (come nel caso del rotativo precedente).

```
MIDIFunc.cc({arg ...args; args.postln});
MIDIFunc.cc({arg ...args; args.postln}, 26, 0);
```

Il prossimo esempio discute la connessione di un controller (nel caso, si tratta dello stesso controller precedente, di cui si usa il rotativo associato al numero `26`) per la gestione di parametri audio. La synthDef `midiSynth` prevede due parametri, frequenza dell'oscillatore a onde quadre e un tasso `rate` che controlla

l'oscillazione tra i due canali audio destro e sinistro. I due parametri sono associati al rotativo del controller MIDI. Come si vede nella funzione associata a `MIDIFunc`, ad ogni cambio di posizione del controller, un messaggio viene inviato e ricevuto da `MIDIFunc`, che valuta la funzione, in cui si chiede l'aggiornamento dei due parametri e il loro passaggio al synth `x`.

```
(
SynthDef(\midiSynth , {|freq = 440, rate = 1|
    Out.ar(0,
        Pan2.ar(Pulse.ar(freq)*LFSaw.ar(rate).unipolar,
            LFPulse.kr(rate)
        )
    )
}).add ;
)

x = Synth(\midiSynth ) ;

MIDIFunc.cc({arg ...args;
    var v = args[0] ;
    x.set(\freq , ((v*0.5)+30).midicps,
        \rate , v.linlin(0,127, 1, 10))
    }, 26, 0) ;
```

La comunicazione MIDI in uscita sfrutta l'oggetto `MIDIOut`, di più basso livello. Il suo utilizzo non è particolarmente complesso, come si può vedere in questo esempio tratto dalla documentazione:

```
MIDIClient.init;

m = MIDIOut(0);
m.noteOn(16, 60, 60);
m.noteOn(16, 61, 60);
m.noteOff(16, 61, 60);
m.allNotesOff(16);
```

Come si nota, è necessario istanziare un oggetto `MIDIOut` a partire dai dispositivi MIDI accessibili: `0` indica infatti il primo dispositivo elencato attraverso

`MIDIClient.init`. A questo oggetto si inviano i messaggi utilizzando tipicamente l'interfaccia SC (cioè, i metodi `noteOn`, `noteOff` etc). In generale, nell'attuale implementazione in SuperCollider, la comunicazione MIDI in uscita è più legata ad un insieme di caratteristiche dell'hardware MIDI utilizzato e del sistema operativo di riferimento. Inoltre, `MIDIOut` non è considerato affidabilissimo in termini di sincronizzazione.

9.5 Lettura e scrittura: File

Una forma di comunicazione classica in informatica utilizza la lettura e scrittura da file. La classe `SoundFile`, specializzata per i file audio, è già stata discussa. Vale la pena invece introdurre brevemente una classe generica ma molto utile per scrivere/leggere file di testo e in formato binario, appunto `File`. Di seguito è riportata la serie oraria della situazione meteo rilevata a Torino Caselle e recuperata da una risorsa in rete. La prima colonna rappresenta l'ora della rilevazione, la terza e la quarta rispettivamente temperatura e umidità.

```
02:50    1014    9    87    VAR-2    Buona    Sereno    -
03:50    1014    8    93    VAR-2    Buona    Sereno    -
05:20    1014    8    81    -    Buona    Sereno    -
05:50    1014    7    81    W-3    Buona    Sereno    -
06:20    1014    8    75    WNW-3    Buona    Sereno    -
06:50    1015    8    87    VAR-2    Buona    Sereno    -
07:20    1015    12    66    -    Buona    Sereno    -
07:50    1015    12    71    VAR-2    Buona    Sereno    -
08:20    1015    13    66    -    Buona    Sereno    -
08:50    1015    14    62    SSE-3    Buona    Sereno    -
09:20    1015    14    62    ESE-3    Buona    Sereno    -
09:50    1015    15    62    -    Buona    Sereno    -
10:20    1014    15    58    E-2    Buona    Sereno    -
10:50    1015    16    55    ESE-3    Buona    Sereno    -
11:20    1014    16    59    S-4    Buona    Sereno    -
11:50    1014    17    51    SE-4    Buona    Sereno    -
12:20    1014    18    52    S-5    Buona    Sereno    -
12:50    1014    18    55    VAR-2    Buona    Sereno    -
13:20    1013    18    55    ESE-4    Buona    Sereno    -
13:50    1013    18    52    E-6    Buona    Sereno    -
14:20    1013    19    48    E-5    Buona    Sereno    -
14:50    1012    20    45    ENE-4    Buona    Poco nuvoloso    -
15:20    1012    19    48    E-5    Buona    Poco nuvoloso    -
15:50    1012    19    48    ENE-3    Buona    Poco nuvoloso    -
16:20    1012    19    45    ENE-3    Buona    Poco nuvoloso    -
16:50    1012    18    55    ENE-5    Buona    Poco nuvoloso    -
17:20    1012    17    55    SE-7    Buona    Poco nuvoloso    -
17:50    1012    15    62    SE-6    Buona    Poco nuvoloso    -
18:50    1013    14    67    S-4    Buona    Poco nuvoloso    -
19:20    1014    14    67    S-4    Buona    Nubi sparse    -
19:50    1013    14    67    SSW-4    Buona    Poco nuvoloso    -
20:20    1014    13    71    SSW-6    Buona    Poco nuvoloso    -
20:50    1014    13    76    SSW-5    Buona    Poco nuvoloso    -
21:20    1015    13    76    SSW-3    Buona    Poco nuvoloso    -
21:50    1013    12    81    WSW-4    Buona    Poco nuvoloso    -
22:20    1014    12    81    SW-3    Buona    Poco nuvoloso    -
22:50    1014    11    87    SSW-3    Buona    Poco nuvoloso    -
23:20    1015    11    87    -    Buona    Poco nuvoloso    -
23:50    1014    11    87    -    Buona    Nuvoloso    -
00:50    1014    11    87    NE-2    Buona    Nuvoloso    -
01:20    1014    11    87    -    Buona    Nuvoloso    -
01:50    1014    10    93    NE-3    Buona    Nubi sparse    -
```

La possibilità di leggere da file permette a SC di accedere agevolmente a varie informazioni che possono essere elaborate in modo da ricostruire una certa struttura dati. Si supponga perciò di aver salvato i dati precedenti in un file di testo. Nell'esempio successivo, l'oggetto `File f` accede al file specificato dal path `p` in sola lettura, come indicato da `"r"`. Il metodo `readAllString` restituisce tutto il contenuto del file come stringa, assegnato a `t` (4). Quindi il file `p` viene chiuso. Seguono quattro righe di elaborazione della stringa-contenuto per ricostruire una struttura dati, che ovviamente andrà conosciuta in precedenza. La riga 6 utilizza `split` per restituire da una stringa un array di sottostringhe tagliate quando viene individuato il carattere `$\n`, che indica il carattere "invisibile" di nuova linea[1]. In questo modo si ottiene un array di stringhe-linee. A questo punto attraverso `collect` ogni riga diviene un array di elementi, separati in questo caso dal carattere di tabulazione (`$\t`) (7). Righe e colonne vengono ora invertite, e dunque ci saranno array di valori omogenei per tipo (8). Infine, vengono scelte solo le colonne relative a temperatura e umidità, che sono convertite in interi (infatti, in precedenza erano comunque stringhe). Si noti che in questo modo siamo passati da caratteri ASCII in un file una struttura dati di numeri interi[2].

```
1 p = "/Users/andrea/SC/introSC/code/comunicazione/dati/tocas01042014" ;

3 f = File(p, "r") ; // aprire il file
4 t = f.readAllString ; // l'intero contenuto come stringa
5 f.close ; // chiudere il file di testo
6 t = t.split($\n ) ; // un array di righe
7 t = t.collect{|i| i.split($\t )} ; // ogni riga -> un array di elementi
8 t = t.flop ; // invertire righe e colonne
9 t = t[2..3].asInteger ; // solo temperatura e umidita'
```

Il tutto può essere più elegantemente incapsulato in una funzione, come `~meteo` nell'esempio seguente, che richiede un percorso come argomento e restituisce la struttura dati discussa:

[1] Il carattere `$` indica invece la classe `Char` in SC.

[2] Peraltro SC definisce altre classi che rendono agevole la lettura da file come quello discusso: ad esempio, `FileReader`, `TabFileReader`, `CSVFileReader`.

```
~meteo = {|path|
    var f = File(path, "r") ;
    var t = f.readAllString ; f.close ;
    t.split($\n ).collect{|i| i.split($\t )}.flop[2..3].asInteger ;
} ;
```

Infine, i dati possono essere rappresentati attraverso il suono, secondo una procedura che si chiama "sonificazione". Nella sonificazione, invece di rappresentare un insieme di dati graficamente, lo si rappresenta attraverso il suono. Nell'esempio seguente, la synthDef sfrutta un approccio additivo, in cui, oltre alla frequenza fondamentale, si controlla anche `fact`, che determina il tasso di smorzamento delle componenti armoniche, che si potrebbe definire brillantezza. Viene quindi calcolato `m`, come risulta dal computo di `~meteo` (9). La routine successiva legge in sequenza coppie temperatura/umidità e li associa a frequenza e brillantezza. In questo modo, si nota ad orecchio come vi sia un progressivo aumento della temperatura e diminuzione dell'umidità (gli eventi cresco in altezza e in brillantezza) cui segue un movimento opposto, secondo appunto quanto avviene nei dati giornalieri.

```
SynthDef(\meteo , { |freq = 440, fact = 0.1|
    var sig = Mix.fill(40, {|i|
        SinOsc.ar(freq*(i+1))
        *(i.neg*fact).dbamp
    })*EnvGen.kr(Env.perc, doneAction:2) ;
    Out.ar(0, FreeVerb.ar(sig))
}).add ;

m = ~meteo.(p) ;

{
    m.flop.do{|i|
        Synth(\meteo , [
            \freq , i[0].linlin(-10, 40, 48, 98).midicps,
            \fact , i[1].linexp(30, 90, 0.1,4)]) ;
        0.15.wait
        }
}.fork ;
```

Attraverso `File` è anche possibile scrivere, come si vede nel prossimo esempio, in cui un verso da una poesia di Toti Scialoja è scritto su file. Si noti che l'estensione del file è irrilevante (e qui assente). Nel secondo blocco il file scritto in precedenza è letto dal disco e il suo contenuto assegnato alla variabile, `~text`.

```
(
t = "Il coccodrillo artritico che scricchiola" ;

p = "/Users/andrea/coccodrillo" ;
f = File(p, "w") ; f.write(t); f.close ;
)

(
f = File(p, "r") ;
~text = f.readAllString ; f.close ;
)
```

L'intera poesia di Scialoja è la seguente:

Il coccodrillo artritico che scricchiola
arranca lungo il greto verso un croco
giallo cromo, lo fiuta, fa una lacrima
se il croco raggrinzisce a poco a poco.

Ecco allora un altro esempio di sonificazione, del dato alfabetico in questo caso. L'esempio assume che alla variabile `~text` sia assegnato tutto il testo precedente da Scialoja. La stringa (multilinea) viene letta e progressivamente scritta in una GUI testuale. Ogni evento-lettera è anche "suonato".

```
(
SynthDef(\reader ,
{ arg freq = 100, vol = 1;
    Out.ar(0,
    Pan2.ar(
        FreeVerb.ar(
            MoogFF.ar(
                Pulse.ar(freq, mul:1)
                *
                EnvGen.kr(Env.perc, timeScale:2, doneAction:2),
                freq*2,
        )
    ),
    LFNoise1.kr(1),
    vol
    )
)}).add
)

(
~time = 60/72/4 ;
~minus = 20 ;

w = Window.new("da Toti Scialoja, Il gatto bigotto (1974-1976)"  ,
    Rect(0,0, 500, 300)).front ;
d = TextView.new(w, w.view.bounds)
    .stringColor_(Color(1,1,1))
    .background_(Color(0, 0, 0))
.font_(Font.monospace(16)) ;

// la Routine esegue una scansione del testo al tasso ~time
Routine({
    var txt = "" ;
    ~text.do {arg letter, index ;
    var f = (letter.ascii-~minus).midicps ; // ascii
    txt = txt++letter ; // il testo incrementa
    d    .string_(txt) // e sostituisce il precedente
        .stringColor_(Color.hsv(0, 0, 1-(index/~text.size*0.8))) ;
        // tutto diventa progressivamente scuro
    Synth(\reader , [\freq , f.max(20)]) ;
    ~time.wait ;
    };
    1.wait;
    w.close // la finestra si chiude
}).play(AppClock) // AppClock per la GUI
)
```

Senza discutere nel dettaglio (lasciato al lettore), solo alcune considerazioni: `~time` è espresso in semicrome a m.m. 72; `~minus` è un fattore di trasposizione; l'altezza è ricavata convertendo ogni carattere del testo nel suo valore numerico in ASCII (34).

9.6 Pipe

Una delle applicazioni interessanti di `File` sta nell'utilizzare SuperCollider come "scripting" o "gluing language". Un linguaggio di scripting è un linguaggio che serve come controllo di alto livello per un altro linguaggio. L'utente utilizza il linguaggio di scripting, il quale internamente gestisce la comunicazione con un altro linguaggio. Con "gluing" si intende invece un uso di un linguaggio per "incollare" insieme in forma unitaria funzionalità espletate da altri linguaggi o programmi. Ad esempio, Postscript è un linguaggio di programmazione per la grafica. Un file Postscript contiene codice Postscript (cioè, semplicemente testo ASCII, come quello di SuperCollider), che viene interpretato e restituisce in uscita un file di immagine. Se si apre un file Postscript con l'IDE SuperCollider si vedrà il codice. Il frammento seguente è un esempio minimale di codice Postscript. Se lo si copia in un file di testo e lo si salva con estensione `ps`, si ottiene un file Postscript. Quando si apre il file, il suo contenuto viene interpretato (assumendo di avere un interprete Postscript, che di solito è presente nei sistemi operativi). L'interprete produrrà a partire dal codice il disegno di una linea.

```
%!
144 72 moveto
288 216 lineto
stroke
showpage
```

Tutti i blocchi di esempio che in questo manuale includono sia sintassi colorata che finestre di post window sono stati generati direttamente dal codice SuperCollider attraverso una classe apposita che analizza il codice SC e genera l'opportuno codice Postscript introducendo la colorazione del testo e dello

sfondo, la riquadratura, i numeri di linea. Ancora, tutte le figure che illustrano segnali sono state generate a partire da strutture dati in SuperCollider, analizzate per produrre codice Postscript. Infatti, se si considera il codice Postscript precedente come una stringa di testo, è evidente ormai come via `File` sia possibile scriverla su un file con estensione `ps`. Questa possibilità è largamente sfruttabile per tutte quelle parti in un programma che sono di tipo descrittivo, e può essere utilizzata ad esempio per generare codice in molti ambienti che prevedano interfacce testuali, come Processing, NodeBox, R, e così via. Al limite, si potrebbero generare da SC file contenenti codice SC.

Il prossimo esempio dimostra come scrivere un file da SuperCollider (scripting) per poi chiamare un programma che vi faccia riferimento (gluing).

```
1  ~psCode = " %! 144 72 moveto 288 216 lineto stroke showpage ";
2
3  ~path = "/Users/andrea/minimal.ps" ;
4
5  ~file = File(~path, "w") ;
6  ~file.write(~psCode);
7  ~file.close ;
8
9  ~pipe = Pipe("pstopdf %".format(~path),"w") ;
10 ~pipe.close ;
```

La variabile `~psCode` è una stringa che contiene il codice Postscript discusso in precedenza, mente `~path` rappresenta un percorso sul file system. Il blocco 11-13 apre un file su `~path`, scrive il codice e lo chiude. Le righe 15-16 invece utilizzano un nuovo oggetto, `Pipe`, che serve per intercettare il terminale, ovvero una modalità di interazione con il sistema operativo che non fa uso della GUI ma della cosiddetta shell (come avveniva in passato). Poiché l'interazione dell'utente attraverso la shell avviene scrivendo righe di comando, è possibile, invece che aprire un terminale e scrivere a mano la riga in questione, chiedere a SuperCollider di farlo. Lo fa `Pipe`, che apre un canale (di qui il nome) con la shell. Nell'esempio, `Pipe` scrive sulla shell (si noti il parametro `"w"`, che sta per "write") la stringa `pstopdf` a cui aggiunge il percorso `~path`. La stringa è ottenuta con il metodo `format` che semplicemente sostituisce a `%` il primo argomento passato. La stringa in questione, che potrebbe essere ovviamente scritta in un terminale, chiama il programma `pstopdf`, che converte un file Postscript in un

file PDF (assumendo che quel programma sia installato sul sistema operativo e accessibile alla shell, come piuttosto usuale nei sistemi Unix-like). La riga 16 chiude il canale aperto con `Pipe`, ed è obbligatoria. Se il processo è avvenuto con successo, allora sulla post window si otterrà `0`[3]. Per la semplice scrittura sulla shell, SuperCollider mette a disposizione il metodo `unixCmd` definito sulle stringhe. Il blocco 15-16 potrebbe essere allora sostituito con quello seguente:

```
1 "pstopdf %".format(~path).unixCmd ;
```

Il prossimo esempio dimostra un'altra applicazione di `Pipe`, a partire da un esempio della documentazione. Data una cartella, la funzione restituisce un array contenente i nomi dei file che questa contiene.

```
1  ~sampleList = { arg samplesPath ;
2      var p, l, labelList = [], fileName ;
3      p = Pipe.new("ls" + samplesPath, "r") ;
4      l = p.getLine ;
5      while({l.notNil}, {
6          l.postln;
7          if ( l.contains("."))
8              { labelList = labelList.add(l) } ;
9          l = p.getLine;
10     }) ;
11     p.close ;
12     labelList
13 } ;

15 ~sampleList.("/Users/andrea/")
```

Il comando Unix `ls` richiede di specificare un percorso di cartella del file system, e restituisce una lista di file e cartelle che questa contiene. la funzione `~sampleList` prende come come argomento un percorso (come si vede nell'uso a riga 15). Alla riga 3 `Pipe` chiama sulla shell il comando `ls` per il percorso. Si noti

[3] Si tratta di uno dei codici numerici che Unix restituisce dopo ogni processo, e che indicano il risultato del processo stesso. L'uso della shell Unix ovviamente esula del tutto da questa trattazione.

che il secondo argomento di `Pipe` questa volta è `"r"`: ciò vuol dire che interessa leggere ciò che viene restituisco dalla shell (in gergo Unix, il cosiddetto `stdout`), ovvero i dati relativi ai nomi di file e cartelle. Infatti, a `l` viene assegnata la stringa restituita dal metodo `getLine`, che restituisce riga per riga l'output di `ls` (4). Il ciclo `while` esaurisce allora l'output dello `stdout`, fintanto cioè che `getLine` non ha più nulla da restituire. Il condizionale verifica attraverso la presenza del punto se si tratta di file o cartella (7-10). È ovviamente un'euristica discutibile in termini generali (i file possono non avere estensione) ma ad esempio tipicamente funziona se si considerano i file audio. L'array ottenuto può essere utilizzato per caricare file audio, per generare GUI con i nomi dei file, etc.

L'uso di `File` e `Pipe` rende estremamente flessibile l'ambiente SuperCollider, perché lo integra dentro l'ecosistema molto vasto dei programmi Unix.

9.7 SerialPort

Infine, un'ultima possibilità di comunicazione, che estende ulteriormente i limiti di SuperCollider verso altri ambienti, è costituita dalla porta seriale. La porta USB (Universal Serial Bus) è a tutt'oggi lo standard industriale di comunicazione tra dispositivi digitali. Si tratta di una comunicazione di tipo seriale, che infatti, come si è visto, ben si presta a sostituire lo strato hardware per il protocollo MIDI, anch'esso basato su una logica di tipo seriale. Come discusso in precedenza a riguardo del MIDI, in quest'ultimo caso, il sistema operativo riconosce un dispositivo collegato fisicamente attraverso la porta USB come dispositivo MIDI, e dunque lo tratta di conseguenza. Tuttavia, la porta USB permette altresì di collegare altri dispositivi, il cui uso è costantemente in crescita, in particolare microcontroller e computer single board. Esempi di quest'ultima categoria sono Raspberry Pi e UDOO, mentre il microcontroller *par excellence* è Arduino (in tutte le sue versioni, anche se esistono anche altre opzioni). Arduino comunica attraverso la porta USB in fase di caricamento dei suoi programmi: questi ultimi vengono sviluppati sul calcolatore nell'ambiente di sviluppo dedicato, compilati e spediti sulla scheda via USB. Tuttavia, la comunicazione può avvenire anche interattivamente in tempo reale: in altri termini Arduino può scambiare dati, in ingresso e in uscita, con un software residente sul calcolatore, ad esempio SuperCollider. Esistono diversi modi in cui la comunicazione con Arduino può avvenire. Ad esempio si può nascondere la comunicazione

di più basso livello in modo da fornire all'utente una interfaccia più amichevole. Un approccio tipico prevede l'uso di librerie che vanno caricate su Arduino a cui fanno da sponda a lato SuperCollider classi che gestiscono la comunicazione con le stesse librerie. In altri termini, ciò che l'utente SC vede è analogo a quanto avviene nella comunicazione da sclang verso scysnth nell'approccio privilegiato in questo testo: un insieme di metodi per accedere alle funzionalità di Arduino.

Tuttavia, ciò che passa tra Arduino e calcolatore è un insieme di byte. È dunque possibile (istruttivo, utile, e a volte necessario) gestire direttamente questa comunicazione di più basso livello, attraverso la classe `SerialPort`. Attraverso

```
SerialPort.devices ;
```

si ottiene una lista dei dispositivi accessibili all'interfaccia seriale. Ad esempio:

```
[ /dev/tty.Bluetooth-Incoming-Port, /dev/tty.Bluetooth-Modem,
/dev/tty.usbmodem641 ]
```

La lista include tutti i dispositivi che fanno uso del bus (canale di comunicazione) seriale, inclusi ad esempio i servizi bluetooth. L'ultimo elemento è invece un microcontroller Arduino. Conosciuto l'identificativo del dispositivo (che è restituito come una stringa), diventa possibile accedervi e aprire la comunicazione.

```
~ard = SerialPort("/dev/tty.usbmodem641", 115200, crtscts:true) ;
```

L'oggetto `SerialPort` prevede un insieme di parametri che dipendono direttamente dalle specifiche hardware del bus. A parte il nome, nell'esempio vengono specificati il tasso di trasferimento dati e un parametro che concerne il cosiddetto "flow control" (`crtscts`). In quest'ultimo caso, il valore `true`

indica che se la quantità di dati da trasferire è superiore al tasso di trasferimento, gli stessi verranno messi un buffer e progressivamente scaricati quando ci saranno risorse disponibili. A questo punto, è possibile leggere e scrivere valori (byte) attraverso la seriale. La semantica della comunicazione ovviamente dipende dal programma caricato a lato Arduino. Si consideri però il prossimo esempio:

```
~port = 2; ~value = 255 ;
~ard.putAll([253, port, value, 255]) ;
```

Il codice assume che il programma a lato Arduino accetti blocchi di 4 byte. Il primo e l'ultimo sono byte di controllo: in altri termini, la presenza di due byte pari a 253 e 255 rispettivamente in testa e in coda in un blocco di quattro byte indica al programma Arduino che un messaggio è in arrivo. Gli altri due valori, rappresentati dalle variabili `~port` e `~value`, sono interpretati come valori che selezionano una porta di uscita su Arduino e un valore (espresso con una risoluzione a 8 bit, 256 valori). In altri termini, il programma a lato Arduino riceve quattro byte, se il primo e il quarto sono `253` e `255`, allora seleziona una porta (qui la `2`) e un valore (`255`). Arduino genererà sulla porta 2 (associata ai segnali di tipo PWM) un segnale elettrico in PWM pari al massimo possibile. L'interpretazione del messaggio inviato da SC attraverso la porta seriale dipende dal codice Arduino (che qui è stato descritto soltanto rispetto alla sua logica). Si noti che fortunatamente `SerialPort` converte i numeri interi in formato byte. In modo del tutto analogo ma simmetrico può avvenire la comunicazione da Arduino a SuperCollider (ad esempio nel caso in cui si invii il valore generato da un sensore). Nel caso, SC leggerà byte inviati e deciderà come interpretarli. Di per sé dunque la comunicazione è semplice, assumendo però di sapere scrivere/leggere attraverso il programma su Arduino i messaggi voluti.

Infine, la porta seriale deve essere chiusa quando si intende disconnettere il dispositivo. Nell'esempio di seguito la prima riga chiude la comunicazione con il dispositivo precedente, mentre la seconda chiude tutte le porte.

```
~ard.close ;
SerialPort.closeAll ;
```

9.8 Conclusioni

Le classi e gli oggetti discussi nel capitolo permettono di ampliare notevolmente l'ecosistema informazionale in cui si situa ed opera SuperCollider. Due sono gli aspetti di rilievo di questo ampliamento. Da un lato, molte funzionalità supplementari possono essere integrate in progetti multimediali complessi. Dall'altro, simmetricamente, l'approccio algoritmico di alto livello di SuperCollider può essere esteso ad altri ambienti hardware e software.