Giorgio Zucco

Laboratorio di SuperCollider

Giancarlo Zedde

Giorgio Zucco, *Laboratorio di SuperCollider*

©2021 Giancarlo Zedde
Giancarlo Zedde
Via Duchessa Iolanda 12, 10138 Torino
www. zedde. com

ISBN 9788899778248
GZ00196

I diritti di traduzione, memorizzazione elettronica, di riproduzione e di adattamento totale e parziale, con qualsiasi mezzo, sono riservati in tutti i Paesi.

Sommario

Capitolo 1

1.1 Introduzione 10
1.3 Architettura del linguaggio 13
1.4 Fondamenti di informatica 15
1.5 Help di SuperCollider 18

Capitolo 2

Fondamenti del linguaggio 21

1.1 Hello world 22
1.2 Matematica 25
1.3 Variabili 27
1.4 Funzioni 30
1.5 Array 33
1.6 Condizioni, cicli 38

Capitolo 3

I mattoni di SuperCollider 41

1.1 Primi esempi sonori 42
1.2 Audio rate, control rate 45
1.3 Basic waveforms 46
1.4 Tools di analisi e controllo 48
1.5 mul, add 52
1.6 Mono e stereo 54
1.7 Polifonia, multichannel expansion 58
1.8 Crossfade 60
1.9 Mouse controller 61
2 Basic Live coding 62
2.1. Recording 64
2.2 I metodi delle unità generatrici 66

Capitolo 4

Tecniche di modulazione 71

1.1 Lfo 72
1.2 Tremolo 74
1.3 Ring modulation 75
1.4 AbstractFunction 77

Capitolo 5

Inviluppi e interpolazioni 79

5.1 Line, glissandi, DoneAction 80
5.2 Punti di un inviluppo 83
5.3 Adsr 84

Capitolo 6

Cicli e randomness 89

6.1 Loop in Sc 90
6.2 Rumore in Sc 95

Capitolo 7

Strutture di controllo, trig 105

7.1 Trigger 106
7.3 Step sequencer 111
7.4 SynthDef 113
7.5 Auto Gui 117
7.5 Midi In 121

Capitolo 8

Sintesi additiva 123

8.1 Da Csound a SuperCollider 126
8.2 Array, campi armonici 129
8.3 Splay 131
8.4 Buzz 133
8.5 Csound gen10 in SuperCollider 136
8.6 Wavetable Synthesis 138
Breve glossario di sintesi del suono 139

Capitolo 9

Sintesi sottrattiva 143

9.1 Rumore bianco e colorato 144
9.2 Panoramica dei filtri in Sc 146
9.3 Filtri risonanti 150
9.4 Formant Filter 152
9.5 Analog modeling 154

Capitolo 10

Sintesi FM 163

10.1 Portante e modulante 164
10.2 Da Csound a SuperCollider 170

10.3 Modulazione di fase 171

Capitolo 11

Sintesi granulare 177

11.1 Primi approcci 178
11.2 Pattern per la generazione di grani 183
11.3 Granular sampling 190

Capitolo 12

Sintesi per modelli fisici 193

12.1 Impulsi, risonatori 194
12.2 Simulazione di corda pizzicata 199
12.3 Estensioni in Sc 203
12.4 Stk in SuperCollider 206

Capitolo 13

Introduzione ai Dsp 209

13.1 Linee di ritardo 210
13.2 Chorus, flanger 218
13.3 Bit reduction 220
13.4 Reverb 221
13.5 Distorsione, Waveshape 226

Capitolo 14

Composizione in SuperCollider 229

14.1 Csound score in SuperCollider 230
14.2 Scale musicali 242
14.3 Pattern 245

Capitolo 15

15.1 Programmare una classe 254
15.2 Un approccio al live coding 256
15.3 Introduzione alle GUI 259

Indice alfabetico di tutte le classi, aggiornato alla versione 3.11.2 269
Bibliografia 287
SuperCollider nel Web 287

Capitolo 1

1.1 Introduzione
1.2 SuperCollider Ide
1.3 Architettura del linguaggio
1.4 Breve glossario informatico
1.5 Help di SuperCollider

1.1 Introduzione

Introdurre e definire con estrema chiarezza la definizione di *computer music* è qualcosa di estremamente complesso, specie in relazione alla data di pubblicazione di questo libro. Sentiamo spesso parlare di categorie come musica elettronica, musica elettroacustica, acusmatica, composizione algoritmica, ecc. in relazione a qualsiasi intervento musicale in cui il suono elettronico è protagonista.

Proprio da questa realtà nasce uno dei problemi principali: definire delle categorie escludendole dal contesto storico e stilistico. Se tentassimo un parallelismo con la terminologia legata al mondo della didattica, ci troveremmo sicuramente nella stessa confusione o nella stessa semplificazione, associando, per esempio, la videoscrittura musicale (nulla di più lontano dal significato intriseco di suono elettronico) alla definizione di informatica musicale, accostamento tutt'altro che sbagliato ma incompleto.

In merito alle problematiche che possono sorgere affacciandosi al contesto storico della musica elettronica, prenderemo in esame due importanti compositori elettronici come Morton Subotnick[1] e James Dashow. Escludiamo per un attimo questioni relative alle tecniche di produzione sonora utilizzate da questi due musicisti (analogico contro digitale) e concentriamoci sull'ascolto spassionato di qualche loro composizione. Qualche considerazione dopo un primo ascolto comparativo? Elementi in comune? Sicuramente molti, iniziando dalla scelta di utilizzare unicamente suoni di sintesi. Troviamo poi numerose analogie timbriche, ritmiche, alta densità di eventi in contrasto con episodi rarefatti, utilizzo di forme d'onda elementari, organizzazione delle altezze spesso atonale o tendente all'alea, assenza assoluta di timbri imitativi o manipolazione di materiale concreto pre registrato e molto altro.

In sintesi, a un primo ascolto superficiale e senza gli strumenti necessari di conoscenza per intuire le tecnologie impiegate, potremmo dire di essere di fronte a un genere musicale comune ai due autori. Stiamo quindi parlando di un genere musicale? Sicuramente la definizione musica elettronica soddisfa in parte la questione; possiamo parlare di *computer music*? Nel caso dei lavori di Subotnick la definizione è errata, in quanto il musicista americano costruisce i suoi timbri utilizzando sintetizzatori analogici (come i moduli *Buchla*). E nel caso di Dashow? Ecco un esempio perfetto di *computer music*, in quanto egli utilizza unicamente i linguaggi di programmazione per la sintesi del suono, in particolare antenati di *Csound* e *SuperCollider* come *Music5*, *Music11* e *Music360*.

Si può pertanto liquidare la comparazione relegando il tutto a questioni di tecnologie impiegate? Non solo, stiamo trascurando anche una questione di metodo compositivo e azzarderemo a dire filosofico. Un sintetizzatore commerciale (*hardware* o *software*) offre al compositore una serie di timbri efficaci e pronti all'utilizzo, su cui il compositore può crearne di nuovi modificando i parametri offerti dalla macchina stessa. Cosa acca-

[1] Un disco di riferimento potrebbe essere il celebre *Silver Apples Of The Moon* (1967).

de invece quando la generazione sonora è affidata alla pagina bianca di un linguaggio di programmazione? In questo caso il compositore si sentirà come una sorta di liutaio virtuale. Un linguaggio di programmazione non ha i limiti comuni di un sintetizzatore commerciale, al contrario, offre una miriade di elementi basilari con cui poter costruire il proprio personalissimo sintetizzatore o processore effetti.

Eccoci finalmente arrivati alla definizione corretta di *computer music*, ossia quella fase tecnico/artistica in cui una composizione o un processo musicale elettronico viene generato dal codice intepretato o compilato di un linguaggio di programmazione. Pertanto, ci troviamo in una categoria o approccio compositivo possibile unicamente nel dominio digitale. La storia della *computer music* parte da lontano, dagli anni '50 in cui Max Mathews realizzò il primo linguaggio di programmazione in grado di generare suoni elettronici, nasceva la generazione dei linguaggi *Music N* che porteranno negli anni '80 al celebre *Csound*[2].

SuperCollider si pone in una doppia prospettiva: da una parte eredita l'utilizzo delle unità generatrici, si tratta di moduli precostituiti in grado di realizzare qualsiasi tipo di calcolo matematico, effettuare operazioni sui segnali, generare forme d'onda elementari o complesse, costruire filtri, modulatori e qualsiasi altra struttura elementare dedicata alla produzione e manipolazione del suono. Dall'altra parte, troviamo il sogno e l'ambizione del suo ideatore James McCartney, che nel 1996 sviluppa la prima versione di *SuperCollider* ma bisognerà attendere la terza *release* del linguaggio per arrivare alle innovazioni, architettura e versatilità che oggi ritroviamo nel linguaggio.

Questo affascinante e potente linguaggio non nasce (come spesso accade) in contesti accademici di ricerca, bensì dall'esigenza personalissima di un abile programmatore e appassionato di sintesi del suono, il quale non voleva rassegnarsi a utilizzare una sintassi di tipo procedurale (simile al linguaggio *Assembly*) come quella di *Csound*. L'idea geniale di McCartney è stata probabilmente quella di integrare elementi della moderna programmazione ad oggetti all'interno di un linguaggio di sintesi.

Immaginate la flessibilità di definire liste dati (array), cicli, condizionali, funzioni, classi, all'interno di un pensiero elettroacustico, tutto questo risulta impossibile nei vecchi linguaggi *Music N* essendo la loro architettura basata sulla definizione di una lista di istruzioni con un preciso ordine da seguire. Significa che in *SuperCollider* possiamo definire una macro struttura ed affidare l'interpretazione di questi dati alla macchina, si tratta quindi di un approccio particolarmente efficace nella creazione di processi generativi, composizione algoritmica e molto altro.

Mettere oggi le mani su un linguaggio così potente richiede certamente dei sacrifici poiché il suo apprendimento necessita di tempo. Chiaramente dovremo dimenticare la facilità di un sintetizzatore commerciale in cui il confronto nasce dalla lettura del manuale. *SuperCollider* è un vero e proprio strumento musicale, richiede curiosità, competenze di varia area (seppur non decisive ma consigliate), conoscenze base delle meto-

[2] *Csound*, Barry Vercoe 1985.

dologie informatiche generali, cenni di acustica musicale e naturalmente una costanza nella pratica.

Per i musicisti, i ricercatori, i programmatori provenienti da studi seri sui linguaggi di sintesi come *Csound* e *Max*, questa potrebbe essere l'occasione di applicare le proprie conoscenze in un sistema aperto, moderno ed in continua evoluzione. Per i nuovi utenti ed appassionati di musica elettronica… questo è il momento di fare un salto verso l'ignoto, è il momento di capire davvero come stanno le cose!

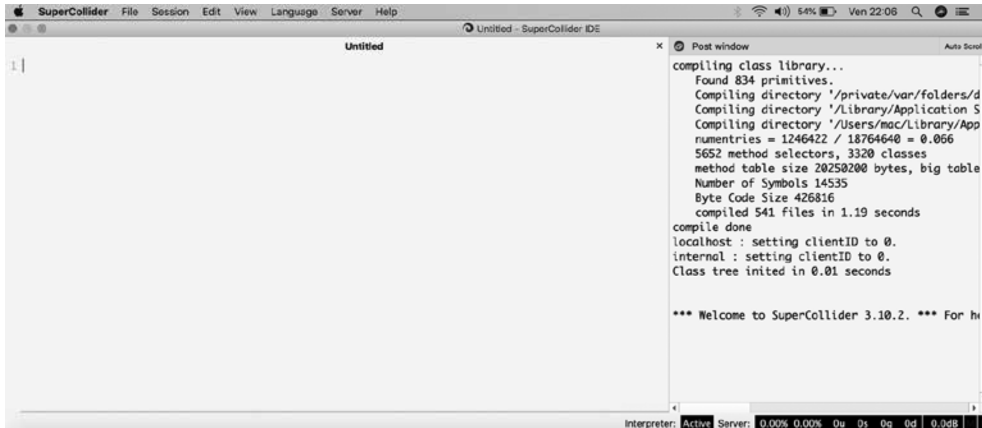
Torino, 2 febbraio 2021

Giorgio Zucco

1.2 SuperCollider Ide

Il *download* del pacchetto di *SuperCollider* comprende il tipico ambiente di sviluppo integrato, che include l'*engine* in *real time* di *SuperCollider*, un editor di testo, strumenti di *debugging*, una sintassi colorata e l'*help* del linguaggio con numerosi tutorial ed esempi pratici.

SuperCollider è un linguaggio multipiattaforma, pertanto i sorgenti (essendo *open source*) e le versioni compilate sono disponibili per ambiente *Windows*, *Osx*, *Linux*.



Prestiamo attenzione alla finestra chiamata *Post window* (per gli appassionati di *Max/Msp*, si tratta del corrispettivo di *Max console*), in questa finestra possiamo monitorare numerosi aspetti legati al processo di sintesi, debugging, informazioni sulla frequenza di campionamento, sul driver del proprio convertitore analogico/digitale e molto altro.

1.3 Architettura del linguaggio

SuperCollider è un linguaggio basato sullo *Smalltalk*, un linguaggio sviluppato verso la metà degli anni '70, nato prevalentemente con intenzioni didattiche e che all'epoca poteva vantare uno dei primi prototipi di interfaccia grafica.

Smalltalk è un linguaggio molto importante per la storia della moderna programmazione, in quanto fu uno dei primi sistemi a introdurre la programmazione a oggetti, in un certo senso è stato l'ispiratore di linguaggi moderni e contemporanei come *C++* e *Java*.

Concretamente, in cosa consiste la compilazione di un frammento di codice in linguaggio *Sc*? *SuperCollider* è un linguaggio interpretato, significa che il codice viene letto ed eseguito al volo, in tempo reale, esattamente come accade in linguaggi come *PHP*. Per prima cosa dobbiamo distinguere le due componenti in cui *SuperCollider* è diviso, *sclang* (SuperCollider-linguaggio) ed *scsynth* (SuperCollider-sintetizzatore).

La pagina bianca del nostro *Ide* rappresenta lo spazio chiamato *client* o interprete delle istruzioni in cui definiamo i nostri algoritmi. Il dualismo *client/server* avviene proprio nella comunicazione tra la scrittura di codice (*sclang*) e l'invio del codice al server (*scsynth*) per mezzo di un protocollo moderno chiamato *Osc* (*Open sound control*).

Semplificando all'estremo, il server di *SuperCollider* altro non è che il motore audio in attesa di ricevere istruzioni dal client (*sclang*); quindi il server si occupa di gestire ed *eseguire* le linee di codice, per esempio l'ascolto in tempo reale di una sinusoide con un inviluppo di quattro secondi? La lettura di un sample al contrario? Il processamento di un segnale microfonico? Un calcolo matematico?

SuperCollider crea automaticamente un server all'avvio (per mezzo di una specifica istruzione o tramite un tool contenuto nell'Ide), un possibile messaggio nella *Post window* potrebbe essere il seguente:

```
*** Welcome to SuperCollider 3.10.2. *** For help press Cmd-D.


Booting server 'localhost' on address 127.0.0.1:57110.
```

Il termine *localhost* è relativo al server locale creato sulla propria macchina, questa precisazione viene fatta considerando che *SuperCollider* permette la creazione di server diversi, e la comunicazione tra client e server può anche avvenire tra computer diversi.

Simuliamo una tipica performance in *SuperCollider*: Dall'icona del software avviamo prima di tutto il nostro ambiente integrato, nello spazio bianco scriviamo la seguente istruzione:

```
s.boot
```

Questo frammento di codice prepara l'avvio del server (s/server). Osservate il pannello in basso nella *Post window*, noterete l'indicazione in colore verde, cioè attiva. Alla voce *interprete*, notiamo invece che il server è spento.

A questo punto avviciniamo il cursore del mouse sopra un punto casuale del codice appena scritto, per mezzo della seguente combinazione di tasti:

Mac: command/return

se non abbiamo commesso errori, il server si è appena avviato mostrando il colore verde (interprete).

A questo punto *scsynth* è pronto a ricevere istruzioni più corpose (che dovremo sempre inviare al server per mezzo della stessa combinazione di tasti). Gli outputs possibili sono tre:

- un messaggio testuale nella *Post window*, ad esempio il risultato di un calcolo matematico, oppure il monitoraggio di dati numerici provenienti da un processo sonoro.
- Audio sui nostri speaker attivi (mono, stereo, quadrifonia, ecc.)

- Interfaccia grafica esterna all'Ide, perché *SuperCollider* permette anche lo sviluppo di *Gui* (graphical user interface) con cui controllare l'algoritmo sonoro programmato (esattamente come una patch in *Max/Msp*).

La programmazione di un algoritmo di sintesi o di trattamento del segnale potrebbe essere schematizzato in due strutture primarie:

- funzione: immaginiamo un blocco di codice avviato che realizzi un particolare timbro o effetto sonoro, in attesa di un comando in grado di terminare il processo.
- controllo esterno: immaginiamo un blocco di codice che descriva un algoritmo di sintesi, caricato sul server ed in attesa di istruzioni musicali sotto forma di partitura, un approccio non troppo lontano dalla coppia *orchestra/score* nel linguaggio *Csound*.

1.4 Fondamenti di informatica

Avvicinarsi al mondo dei linguaggi di sintesi implica inevitabilmente una conoscenza basilare della terminologia legata alla programmazione informatica generale. Concetti e termini come classi, programmazione a oggetti, cicli, funzioni, array, rappresentano alcuni dei blocchi principali della programmazione in *SuperCollider*.

Da un punto di vista strettamente pratico, lo sviluppo di un algoritmo di sintesi mediante un linguaggio specifico per l'audio, non è dissimile dal modus operandi di uno sviluppatore web, aziendale, amministrativo o scientifico.

Iniziamo dalla fase finale che vogliamo raggiungere, ossia realizzare un suono in *SuperCollider*, sicuramente qualche lettore potrà avere avuto esperienze con un linguaggio antenato come *Csound*, ricordate il processo di compilazione?

```
codice sorgente - compilatore - scrittura di un file in formato wav o aiff.
```

Csound è forse il linguaggio di sintesi piu vicino a linguaggi tradizionali come *C*, *Assembly*, ecc; esattamente come nel linguaggio *C* (da cui ovviamente *Csound* deriva), un codice viene analizzato per verificare possibili errori di sintassi, e in seguito, uno strumento chiamato compilatore realizza un file in formato eseguibile (in *Csound* sarà naturalmente un file di tipo audio).

Questa modalità che definiremo in *differita*[3] è assai lontana dalla scrittura di materiale sonoro in linguaggi come *SuperCollider* o *MaxMsp*. Quindi come avviene il processo di scrittura su disco in un linguaggio nato espressamente per il real time? Nato per realizzare un processo sonoro di cui non conosciamo ancora il risultato finale, né la durata del processo stesso? Semplice, avremo bisogno di avviare la nostra performance sonora ed utilizzare un tool di registrazione contenuto all'interno del linguaggio; se la nostra scheda audio sarà settata, per esempio, alla frequenza di campionamento di 96000 hz, *SuperCollider* registrerà un file a 96000 hz e 32 bit, naturalmente sarà possibile specifi-

[3] Nelle recenti versioni di *Csound* è naturalmente possibile lavorare anche in real time, il termine "differita" è legato all'approccio storico con *Csound*, quando le Cpu dei calcolatori non erano sufficientemente potenti per inviare al *Dac* il risultato audio in tempo reale.

per risolvere il problema iniziale, prende il nome di *programma*.

Sintassi: ogni linguaggio di programmazione ha una sua sintassi specifica, oltre alla logica da seguire nella descrizione di un certo algoritmo, logica legata alla struttura e forma, una delle principali difficoltà riguarda gli errori di *grammatica*, essendo i computer totalmente privi di intelligenza autonoma !?!?! un semplice errore di battitura come una lettera sbagliata, l'avere dimenticato una parentesi oppure il non avere dichiarato il nome di una variabile, genera immediatamente messaggi di errore che compromettono l'avvio della performance.

Nel caso specifico del linguaggio *SuperCollider*, le istruzioni di definizione algoritmica utilizzano qualsiasi tipo di parentesi e vengono riconosciute le lettere maiuscole da quelle minuscole, un esempio tipico del linguaggio:

```
SynthDef(\test, {
 arg freq = 440;
 var sig;
 sig = SinOsc.ar(freq) !2;
 sig = sig * XLine.kr(0.2,0.01,0.3,doneAction:2);
 Out.ar(0, sig);
}).add;
```

boot, **quit**: ogni performance in *SuperCollider* necessita di un server attivo, l'avvio e la chiusura si ottiene mediante istruzioni o il tool specifico dell'*Ide*.

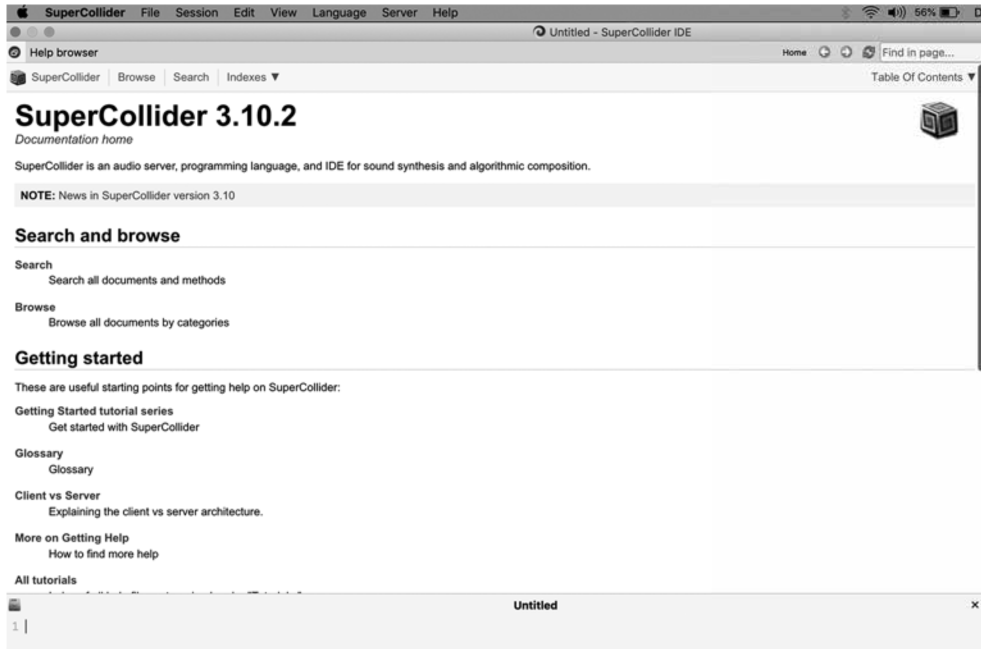
buffer: esattamente come nel linguaggio *MaxMsp*, il buffer rappresenta uno spazio di memoria in cui collocare un sample audio oppure una tabella contenente le parziali con cui sintetizzare una certa forma d'onda.

Array: chiamato anche vettore o matrice, rappresenta un insieme di dati dello stesso tipo, ogni valore della lista viene chiamato *cella* e si comporta come una normale variabile. Immaginiamo un tipico esempio di sintesi additiva in cui avremo bisogno di una lista di dati numerici da associare al parametro frequenza, un *array* viene delimitato da parentesi quadre:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

1.5 Help di SuperCollider

Si tratta di uno strumento indispensabile non solo per chi muove i primi passi nel linguaggio, alla voce Help aprire *Show Help Browser*.



In questa sezione troviamo ampio materiale didattico e di documentazione, inclusi numerosi tutorial storici di immediata efficacia, diamo un'occhiata alla voce *Browse/ ugen*.

Il termine *ugen*, unità generatrice, corrisponde al concetto di *opcode* in *Csound* o di *object* in *MaxMsp*, si tratta di un incapsulamento di codice in grado di realizzare una determinata funzione, in pratica le *ugen* rappresentano i moduli di *SuperCollider* dedicati alla sintesi, campionamento, trattamento del segnale, funzioni matematiche, spazializzazione.

Qualche esempio di unità generatrici in *SuperCollider*:

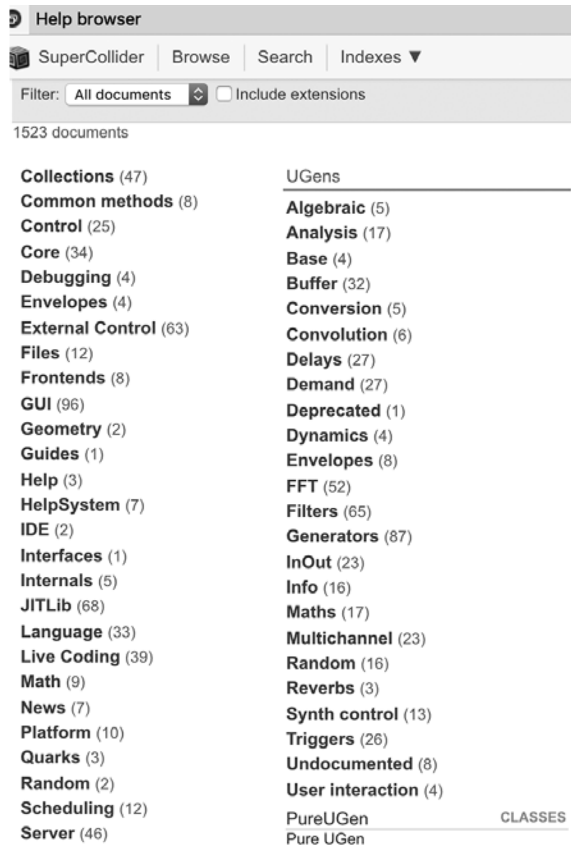
SinOsc: utilizzato come generatore di onde sinusoidali o come segnale di controllo, ad esempio per realizzare effetti di *autopan*, tremolo, vibrato.

Pan2: gestisce la posizione di una sorgente sonora all'interno del panorama stereo.

WhiteNoise: generatore di rumore bianco.

Saw: generatore di onda a dente di sega.

Blip: simile all'*opcode Buzz* di *Csound*, genera impulsi con la possibilità di definire il numero di armoniche (tutte con la stessa ampiezza), segnale molto adatto per la sintesi sottrattiva.



RLPF: filtro passa basso risonante.
Splay: distribuisce un'array di canali intorno al panorama stereo.
Mix: distribuisce un'array di canali intorno a un singolo canale.

Notate la lettera maiuscola iniziale? tutte la parole chiave di *SuperCollider* che iniziano con la lettera maiuscola vengono riconosciute dall'editor di sviluppo e colorate, significa che sono delle classi. *SuperCollider* è un linguaggio moderno basato sulla programmazione a oggetti, una classe non è nient'altro che un contenitore di variabili, funzioni con lo scopo di realizzare una struttura specifica nella risoluzione di un problema e riutilizzabile. Potremmo definire una classe anche come un contenitore di oggetti.

Abbiamo fatto cenno alla programmazione ad oggetti, che cosa sono gli oggetti? Un oggetto rappresenta un insieme di dati e metodi, ogni oggetto può svolgere delle operazioni, cioè può avere un comportamento.

Oltre alle *ugen* proprietarie del linguaggio, nel corso degli anni sono stati sviluppati numerosi altri moduli[4] che espandono *SuperCollider* in maniera molto corposa, queste unità esterne prendono il nome di:

Sc3 plugins
Quarks

Una pratica funzione per accedere alla pagina di *help* relativa ad un modulo, è quella di scrivere la parola da cercare nell'editor, selezionarla ed in seguito premere la combinazione di tasti command/D, si aprirà l'*help* di *SuperCollider* direttamente alla pagina dedicata.

[4] SuperCollider extension, sono pacchetti aggiuntivi da installare, per il download fare riferimento al sito ufficiale. Precisiamo che il 90% di questo libro utilizzerà esclusivamente le *ugen* fondamentali del linguaggio.

Capitolo 2

Fondamenti del linguaggio

1.1 Hello world
1.2 Matematica
1.3 Variabili
1.4 Funzioni
1.5 Array
1.6 Condizioni, cicli

1.1 Hello world

Qualsiasi manuale dedicato alla programmazione informatica non può che cominciare dal classico *Hello world*, ossia un programma elementare in grado di visualizzare un messaggio nella finestra di console.

Per valutare ed eseguire un esempio in *SuperCollider* abbiamo bisogno di posizionare il cursore del mouse vicino sul codice, ed in seguito usare le seguenti combinazioni di tasti:

esegui: command enter (mac)
esegui: ctrl return (windows)
ferma key . (mac)
ferma key alt . (windows)

Apriamo il nostro ambiente di sviluppo e vediamo quanto sarà semplice digitare il nostro messaggio per ottenere…un messaggio di errore !?!?!?

```
Hello world
```

Valutiamo il codice con la combinazione di tasti *command/enter* e come per magia apparirà nella Post window il seguente messaggio:

```
ERROR: syntax error, unexpected NAME, expecting $end

in interpreted text
line 1 char 11:
Hello world
```

Cerchiamo di analizzare a cosa sia dovuto il messaggio di errore, partiamo proprio dal tipo di messaggio che abbiamo scritto nell'editor, possiamo dire di aver definito un tipo di dato chiamato *stringa di testo*? purtroppo no, *SuperCollider* ha processato un file di testo contenente due parole di cui non riesce a capire se si tratta di variabili, dati numerici o stringhe di testo.

Per definire un messaggio testuale da stampare nella *Post window* avremo bisogno di includere il nostro messaggio all'interno di virgolette, nel seguente modo:

```
“Hello world”;
```

Funziona? ecco il nostro output:

```
-> Hello world
```

Come possiamo notare, non riceviamo piu un messaggio di errore, tuttavia non abbiamo ancora raggiunto lo scopo di stampare sulla console il nostro messaggio di saluto, la sintassi esatta sarà:

```
“Hello world”.post;
```

oppure:

```
“Hello world”.postln
```

Possiamo utilizzare il metodo post oppure postln, le uniche differenze riguardano la formattazione, il metodo postln va a capo al termine della lettura. Proviamo a valutare i seguenti esempi (selezionate le quattro righe e premete la combinazione di tasti, in seguito vedremo metodo più pratici per gestire l'esecuzione di blocchi di codice ampio):

```
“Hello world”.post;
“Hello world”.post;
“Hello world”.post;
“Hello world”.post;
```

oppure:

```
“Hello world”.postln;
“Hello world”.postln;
“Hello world”.postln;
“Hello world”.postln;
```

Adesso scriviamo e valutiamo i seguenti esempi, procedendo una riga alla volta:

```
1;
10;
10 20 30;
cane;
gatto;
```

Prima osservazione possibile? Un singolo elemento numerico viene stampato sulla finestra di console senza generare nessun tipo di errore, cosa accade con la riga contenente i tre valori 10, 20 30? *SuperCollider* non è in grado di riconoscere il tipo di messaggio, una possibile soluzione potrebbe essere:

```
[10, 20, 30];
```

Come possiamo comportarci con i messaggi di errore durante la valutazione delle parole singole? Senza addentrarci troppo nella questione, per il momento, proviamo con la seguente soluzione:

```
~cane;
~gatto;
```

Come vedremo in seguito, il simbolo matematico della ~ (tilde) viene utilizzato in *SuperCollider* per definire variabili associate a qualsiasi tipo di parola.

L'esempio seguente indica la modalità con cui scrivere dei commenti durante la programmazione dei nostri suoni, si tratta del doppio // che già troviamo nei linguaggi *C*, *C++* e *Java*.

```
// questo è un commento
```

Possiamo anche commentare su molte linee in maniera più efficace:

```
/*
questo è un commento
distribuito
su varie linee
*/
```

Esiste una modalità per valutare numerosi messaggi con una singola azione? Naturalmente sì, il metodo adatto è quello di includere i nostri messaggi all'interno di una coppia di *parentesi tonde*, separando un messaggio dall'altro per mezzo di un *punto e virgola*.

Provate a selezionare con il cursore del mouse un punto qualsiasi del seguente esempio:

```
(
1.postln;
2.postln;
“cane”.postln;
“gatto”.postln;
)
```

Osserviamo gli esempi successivi in cui il risultato è identico per tutte le tre forme:

```
“Quel pomeriggio di un giorno da…”.postln;
```

Ora lo stesso esempio ma con differenti spazi tra una parola e l'altra:

```
“Quel pomeriggio di un giorno da…”.postln
```

oppure:

```
“
quel
pomeriggio
di
un
giorno
da
“
.
postln
```

1.2 Matematica

In questa breve sezione inizieremo ad esplorare alcune delle principali funzioni matematiche in *SuperCollider*, in questi esempi vedremo come effettuare calcoli di qualsiasi tipo senza la dichiarazione di variabili o senza dovere specificare nessun tipo di dato (intero, virgola mobile).

Scrivere e valutare le seguenti linee di codice, una alla volta:

```
1 + 1;
20 / 4;
(2 + 3) + (4 * 4);
(1 + 1).postln;
```

Come in tutti i moderni linguaggi di programmazione, troviamo in *SuperCollider* numerose funzioni matematiche sottoforma di parole chiave, ecco una breve panoramica di quelle principali:

round: questo metodo restituisce il valore intero più vicino, esempio:

```
14.94875.round; // restituisce il risultato 15.0
14.14875.round; // restituisce il risultato 14.0
```

sqrt: calcola la radice quadrata di un numero, esempio:

```
128374.sqrt;
```

rand: questa funzione restituisce un numero casuale compreso tra 0 ed il numero indicato, eliminando l'ultimo valore, proviamo a valutare varie volte il seguente esempio:

```
10.rand; // otteniamo valori casuali compresi tra 0 e 9
```

altri metodi per la generazione di numeri casuali sono i seguenti: *rand2* (restituisce anche valori negativi), *linrand*, ecc.

Altre funzioni matematiche:

```
100.log; // logaritmo naturale di un numero
345.abs; // valore assoluto
[9, 10, 20, 2, 3, 234].sort; // ordina una lista numerica partendo dal
valore basso
100.reciprocal; // reciproco di un numero, ossia la frazione 1/numero
100.ceil; // arrotonda un numero per eccesso
100.floor; // arrotonda un numero per difetto
1 * pi; // pi greco
```

Un esempio che integra vari metodi:

```
(((10 + 20).rand)/100.log).ceil;
```

L'utilizzo di parentesi tonde influenza il risultato, a causa dell'ordine in cui vengono eseguite le operazioni:

```
10 + 20 * 2; // risultato = 60
(10 + 20) * 2; // risultato = 60
10 + (20 * 2); // risultato = 50
(1+2+3)*(1+2+3); // risultato = 36
1+2+3*1+2+3; // risultato = 11
(1 + 2) + 3 * 1 + (2 + 3); // risultato = 11
```

Osserviamo un modello costruito mediante la dichiarazione di variabili (di cui ci occuperemo in maniera più approfondita nei paragrafi successivi), selezionare, con il cursore del mouse, un punto qualsiasi nel seguente esempio delimitato da parentesi tonde e valutare:

```
(
var num1, num2, calcolo ; //dichiaro tre variabili (num1, num2, calcolo)
num1 = 100; // assegno un valore alla variabile num1
num2 = num1.rand; assegno un metodo alla variabile num2
calcolo =  um1*num2 //moltiplico il risultato delle due variabili
)
```

Qualche esempio con le espressioni booleane, un valore booleano restituisce solo due tipi di output: *vero* o *falso* (*true*, *false*). Si tratta di una comparazione tra dati (o blocchi di codice ampio) in cui verificare una delle due condizioni, esempio:

```
(5) == (20 / 4) // true
(20 / 4) == (20 / 3) // false
(20 / 4) == (20 / 3) // false
(20 / 4) < (20 / 3) // true
```

ecco il risultato:

```
ERROR: Variable 'sine_1' not defined.
  in interpreted text
  line 1 char 6:
  sine_1.value;
```

SuperCollider non è in grado di trovare la dichiarazione della variabile sine_1, proprio perché è contenuta all'interno di un blocco di codice in cui è stata dichiarata come variabile locale (*var*), il valore di questa variabile sarà accessibile solo all'interno di questo codice.

Osserviamo il comportamento del seguente esempio:

```
(
sine_1 = 100;
sine_2 = 300;
sine_3 = 500;
additiva =  sine_1 + sine_2 + sine_3;
)
```

otteniamo:

```
ERROR: Variable 'additiva' not defined.
```

In questo caso *SuperCollider* non riconosce la parola identificativa usata per dichiarare ogni variabile, è un problema che non riguarda il tipo di variabile (locale o globale), ma la scelta di utilizzare delle parole al posto di singole lettere. Risolviamo il problema facendo precedere ogni parola dal simbolo della tilde ~:

```
(
~sine_1 = 100;
~sine_2 = 300;
~sine_3 = 500;
~additiva =  ~sine_1 + ~sine_2 + ~sine_3;
)
```

Ora funziona? considerando che il blocco di codice non specifica se la variabile è locale o globale, possiamo accedere al contenuto di una qualsiasi delle variabili dichiarate come nel seguente esempio:

```
~additiva.value.postln;
```

I seguenti esempi rappresentano tre tipi diversi di sintassi per dichiarare le stesse variabili:

```
(var a , b , c;
a = 100;
b = 200;
c = 300;)
```

oppure:

```
(var a = 100;
var b = 200;
var c = 300;
)
```

oppure:

```
(var a = 100, b = 200, c = 300;)
```

con accesso al contenuto delle variabili:

```
(
var a = 100, b = 200, c = 300;
(a*b-c).postln
)
```

In questi esempi abbiamo visto come accedere al contenuto di una variabile senza modificarne il contenuto, vediamo adesso come assegnare un valore diverso all'interno di un programma, per esempio provando a trasferire il contenuto di una variabile in un'altra.

```
(
var a = 100;
var b = 200;
var c = 300;
b = (a + b).rand;
c = b;
)
```

Potremmo anche considerare un'assegnazione multipla a una singola variabile, immaginiamo una variabile chiamata *synth*, che nel corso del programma subisce alcune modifiche timbriche con l'aggiunta di un *Lfo* sulla frequenza, una modulazione d'ampiezza oppure l'indirizzamento ad uno spazializzatore, esempio:

```
(
var sig;
sig = 200;
sig = 300;
sig = (sig + 100) - sig;
sig = sig.reciprocal;
)
```

Negli esempi dei capitoli successivi, ci occuperemo anche di un altro tipo di variabile globale indicata come *arg*, esempio:

```
arg a=100, b=200;
```

che diventerà decisiva per il controllo di un synth da parte di una struttura di controllo esterna alla funzione, il tipico caso di un sequencer (*pattern*) che gestisce gli eventi musicali e ritmici di un synth dichiarato e caricato sul server in attesa di istruzioni.

A proposito di variabili globali, possiamo anche definire variabili per mezzo del simbolo ~, questo approccio sarà particolarmente utile nel *live coding*, esempio:

```
~data_1 = 440;
~data_2 = 880;
~somma = (~data_1 * ~data_2);
~somma.value
```

Per aggiornare il contenuto delle variabili ~data_1 e ~data_2, sarà necessario valutare ogni frammento di codice, inclusa la variabile che realizza l'operazione matematica. Possiamo anche includere l'intero blocco di codice all'interno di parentesi tonde per una piu facile valutazione/esecuzione del codice:

```
(
~data_1 = 440;
~data_2 = 880;
~somma = (~data_1 * ~data_2);
~somma.value
)
```

1.4 Funzioni

Definire e costruire un programma dal punto di vista informatico, implica la definizione di sottoprogrammi o routine, ossia di singole istruzioni che una volta raggruppate possono risolvere un determinato tipo di problema.

Per definizione, una *funzione* viene identificata con un nome univoco (esattamente come la dichiarazione di una variabile) e si occupa di svolgere un certo compito. Ogni *funzione* deve essere provvista di ingressi e uscite; sarà possibile accedere in qualsiasi momento alle varie funzioni che definiscono un programma, senza doverle dichiarare ogni volta.

Schematizzando in maniera elementare, potremmo semplificare l'intero processo informatico in un grande blocco di codice definito programma, il quale a sua volta sarà costituito da varie funzioni (ognuna specifica per un determinato compito), una funzione conterrà a sua volta delle variabili: programma - funzioni - variabili.

Esattamente come nel linguaggio *C* (e non solo), anche in *SuperCollider* definiamo una funzione come un contenitore delimitato da parentesi graffe.

Il prossimo esempio realizza un calcolo matematico partendo da matrici, anche se non è ancora il momento di addentrarci nella definizione di *array* (a cui dedicheremo molta attenzione essendo uno degli aspetti di maggior interesse in questo linguaggio).

```
x = {
 arg a, b, c;
 a = [1,2,3,4,5];
 b = [10,20,30,40,50].scramble;
 c = (a*b);
}
```

la lista associata alla variabile *b* utilizza il metodo *scramble*, *SuperCollider* ricostruirà la lista permutando in modo casuale il contenuto. Esattamente in cosa consiste la moltiplicazione di due liste?

Proviamo ad interrogare la funzione varie volte nel seguente modo:

```
x.value.postln;
```

ed ecco alcune delle possibili liste ottenute e visualizzate nella *Post window*:

```
-> [ 50, 20, 90, 80, 200 ]
-> [ 20, 100, 120, 40, 150 ]
-> [ 30, 20, 60, 200, 200 ]
-> [ 30, 80, 150, 40, 100 ]
```

otteniamo questo risultato perché un elemento della lista *a* è stato moltiplicato per il corrispondente (significa che occupa la stessa posizione nella lista) elemento della lista *b*, quest'ultima restituisce un ordine diverso ad ogni valutazione (scramble).

1.5 Array

Una delle caratteristiche piû importanti di *SuperCollider* è la possibilità di utlizzare tipi di variabili chiamate *array*, esattamente come nei moderni linguaggi di programmazione come *C++* e *Java*.

Si tratta di una sorta di contenitore, una lista di dati appartenenti ad una particolare area, immaginiamo di definire un contenitore chiamato scuola, i singoli componenti alunni rappresenteranno il contenuto di questo array; si tratta quindi di una raccolta di variabili dello stesso tipo.

Esempi dal punto di vista musicale? l'*array scala maggiore* conterrà sette note, oppure le frequenze che definiscono il campo armonico del timbro di una campana, ecc.

Prendiamo in considerazione la costruzione di alcuni semplici *array* costituiti da numeri e lettere:

```
x = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100];
x.value;
x.postln;
y = ["il", "mattino", "ha", "l'oro", "in", "bocca"]
y.value;
y.postln;
```

selezionate e valutate controllando il risultato nella *Post window*.

Una volta definito il nostro *array*, possiamo analizzarne il contenuto ed effettuare operazioni di manipolazione sul contenuto, *SuperCollider* offre numerosi metodi assai utili, vediamone alcuni nel dettaglio:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10].choose
```

in questo esempio verrà prelevato, in maniera casuale, un valore dalla lista, provate a valutare diverse volte la riga di codice.

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10].reverse
```

Il metodo *reverse* restituisce il contenuto della lista per moto retrogrado, quindi: 10, 9, 8, 7, 6, 5, 4, 3, 2, 1

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10].mirror
```

Il metodo *mirror* restituisce l'immagine speculare della lista, immaginiamo una lista del tipo [a,b,c] che sottoposta a questo metodo diventerà [a,b,c,b,a].

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10].size
```

Questo metodo è utile per visualizzare il numero degli elementi contenuti nella lista,

in questo caso il risultato sarà il numero 10.

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10].scramble
```

Il metodo *scramble* restituisce una permutazione casuale degli elementi contenuti nella lista, uno dei possibili output sarà quindi: [9, 3, 2, 5, 10, 4, 8, 7, 1, 6].

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10].rand
```

Metodo simile a *scramble* ma la lista ottenuta sarà sempre priva dell'ultimo elemento.

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10].rand2
```

simile al metodo *length* del linguaggio Lisp.

Un altro metodo simile è *rand2*, in grado di permutare l'ordine degli elementi alternando tra valori positivi e negativi.

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10].rotate
```

Quest'ultimo esempio è in grado di modificare il punto di partenza della lista, è anche possibile specificare il punto esatto con la seguente istruzione:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10].rotate(2)
```

in questo caso la lista verrà riscritta a partire dal penultimo elemento, nella Post window:
`-> [ 9, 10, 1, 2, 3, 4, 5, 6, 7, 8 ]`

Naturalmente possiamo svolgere sull'array anche delle operazioni più complesse concatenando vari metodi:

```
[1, 2, 30, 99, 5].reverse.rand.mirror
```

Naturalmente le possibilità offerte sono immense e conviene fare riferimento alla documentazione ufficiale, come ultimo esempio prenderemo ancora in considerazione il metodo del riscalamento, simile all'oggetto *scale* in Max, con cui limitare una lista dati attraverso un determinato range.

```
Array.series(size, min, max).normalize(min, max)
```

Consideriamo adesso un particolare *array,* in cui un elemento sia a sua volta una lista, si tratta di un tipo di *array* molto utilizzato in *SuperCollider*, ad esempio per definire le componenti di una struttura accordale o per costruire una completa progressione di accordi, esempio:

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

oppure:

```
[[60,65,68].midicps, [64,74,66].midicps, [60,63,67].midicps] // 3 accordi
```

Osserviamo il metodo *midicps*, utile per convertire il pitch (da 0 a 127) in frequenza, corrisponde all'oggetto *mtof* in *Max*.

Negli esempi fin qui analizzati, ci siamo occupati della creazione di liste con elementi definiti dall'utente, immaginiamo di affidare al linguaggio la generazione di una lista, partendo dalla descrizione di un metodo o di una funzione.

SuperCollider contiene una moltitudine di classi in grado di generare array molto complessi, incluse le matrici multidimensionali. Prima di addentrarci nello studio di queste classi, iniziamo con due semplici esempi di iterazione estremamente efficaci, entrambi i metodi generano il medesimo output:

```
~freq = 440 !4 //primo metodo
~freq = 440 dup:4 //secondo metodo
```

eseguiamo una riga alla volta ed osserviamo il risultato:

```
-> [ 440, 440, 440, 440 ]
```

Analizziamo nel dettaglio il breve frammento, abbiamo definito una variabile *~freq* attribuendole il valore numerico 440, i due metodi generano una lista di *n* elementi (in questo caso 4) iterando il valore iniziale, vedremo nei capitoli successivi come utilizzare questi metodi per problematiche sonore fondamentali come la selezione del numero di canali audio in uscita.

Proviamo a modificare l'esempio precedente in modo da generare una matrice bidimensionale, esattamente in cosa consiste una matrice a piu dimensioni? immaginiamo una struttura formata da righe e colonne, una struttura in cui l'array è formato da altri array, ricordate l'esempio per definire delle strutture armoniche musicali?

```
~freq = 440 dup:[2,4] //array bidimensionale
```

eseguiamo il codice ed otteniamo:

```
-> [ [ 440, 440, 440, 440 ], [ 440, 440, 440, 440 ] ]
```

Le seguenti linee di codice rappresentano il metodo più semplice per generare liste numeriche attraverso un determinato range di valori. Provate a scrivere due numeri separati da un doppio punto:

(1..10) // genera la lista [1,2,3,4,5,6,7,8,9,10]

stesso metodo utilizzando delle variabili:

```
(
~min = 1; //valore iniziale
~max = 4; //valore massimo
~list = (~min..~max);
~list.mirror.rand dup:2
)
```

alcuni possibili valori ottenuti:

```
-> [ [ 0, 0, 2, 2, 0, 0, 0 ], [ 0, 0, 2, 2, 0, 0, 0 ] ]
-> [ [ 0, 1, 1, 3, 2, 0, 0 ], [ 0, 1, 1, 3, 2, 0, 0 ] ]
-> [ [ 0, 1, 2, 2, 0, 1, 0 ], [ 0, 1, 2, 2, 0, 1, 0 ] ]
-> [ [ 0, 1, 2, 1, 1, 0, 0 ], [ 0, 1, 2, 1, 1, 0, 0 ] ]
-> [ [ 0, 0, 2, 1, 0, 0, 0 ], [ 0, 0, 2, 1, 0, 0, 0 ] ]
-> [ [ 0, 1, 2, 3, 1, 1, 0 ], [ 0, 1, 2, 3, 1, 1, 0 ] ]
```

Gli ultimi esempi sono solo una parte infinitesimale della generazione automatica di *array* che può fare *SuperCollider*, vedremo infatti come sarà possibile generare complesse liste attraverso metodi stocastici, geometrici, espressioni e formule matematiche. Il metodo dei due punti potrebbe essere scritto mediante la seguente classe di *array*:

```
Array.series(10,1,1)
```

che corrisponde al metodo:

```
(1..10)
```

Valutiamo i due esempi ed otteniamo la medesima lista [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] , diamo ora uno sguardo alla sintassi della classe *Array.series*:

```
Array.series (size, start, step)
```

size: numero degli elementi di cui è composto l'array.
start: valore numerico iniziale della lista.
step: valore di incremento numerico tra un elemento e l'altro della lista

Oltre alla consultazione dell'help di Sc, possiamo sfruttare una pratica funzione dell'ambiente di sviluppo per conoscere i nomi delle varie classi che generano *array*, proviamo a scrivere la parola *Array* seguita da un punto, apparirà un menù a tendina contenente una lista molto ampia di parole chiave.

La nostra esplorazione sul mondo degli *array* continua proprio con le classi appena suggerite dal nostro *ide*, iniziamo dalla classe *Array.fill*, la cui sintassi è:

```
Array.fill(size,function)
```

Proviamo ad inserire una piccola funzione al suo interno, esempio completo:

```
(
~size = 8; //definisco il numero di elementi
Array.fill(~size,
{arg freq; //dichiaro una funzione
freq + 440 //il numero 440 indica il punto di partenza
})
)
```

otteniamo nella *Post window*:

```
-> [ 440, 441, 442, 443, 444, 445, 446, 447 ]
```

e se volessimo ottenere una lista numerica con un incremento pari a 2? Proviamo a modificare l'esempio precedente nel modo seguente:

```
(~size = 8;
Array.fill(~size,
{
arg freq, init, incremento;
init = 100;
incremento = 2;
(freq * incremento) + init
}))
```

Provate a modificare l'esempio modificando il fattore di incremento, magari indicando un valore decimale, oppure provate ad inserire alcuni metodi visti in precedenza come *mirror*, *rotate*, *rand*; un metodo elegante potrebbe essere quello di assegnare un nome all'intero blocco di codice (x = (~size..), e in seguito interrogare la funzione con l'istruzione:

```
x.value.mirror.rand
```

Esaminiamo ora ulteriori classi contenute in *SuperCollider*, il cui utilizzo si rivelerà, tra le varie cose, assai utile nella sintesi additiva, come vedremo nei capitoli successivi.

Array.rand(numero di elementi, valore minimo, valore massimo), un possibile risultato:

```
-> [ 77, 58, 83, 64, 74, 41, 49, 46, 74, 20 ]
```

Array.rand2(numero di elementi, valore), un possibile risultato:

```
-> [ 1, 10, 1, 8, 4, 4, -7, 6, -10, -7 ]
```

Array.geom(numero di elementi, valore iniziale, fattore di moltiplicazione), un possibile risultato:

```
-> [ 10, 20, 40, 80, 160, 320, 640, 1280, 2560, 5120 ]
```

Array.series(numero di elementi, valore iniziale, incremento), un possibile risultato:

```
-> [ 10, 11, 12, 13, 14, 15, 16, 17, 18, 19 ]
```

Array.interpolation(numero di elementi, valore iniziale, valore finale), un possibile risultato:

```
-> [ 10.0, 8.9, 7.8, 6.7, 5.6, 4.5, 3.4, 2.3, 1.2, 0.1 ]
```

Array.fib(numero di elementi, elemento a, elemento b), un possibile risultato:

```
-> [ 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 ]
```

1.6 Condizioni, cicli

Come in tutti i moderni linguaggi di programmazione, anche in *SuperCollider* troviamo le strutture di controllo o di ciclo.

Le strutture di controllo permettono l'iterazione di un determinato processo purché vengano rispettate alcune condizioni. Immaginiamo per esempio di affidare all'utente la scelta di ascoltare un tipo di suono piuttosto che un altro, tale scelta sarà vincolata dal verificarsi della seguente condizione: x<10.

Una delle condizioni maggiormente usate nella programmazione è il costrutto *if*, un semplice esempio restituisce il risultato numerico 100 oppure 200 in base alla grandezza della variabile *a*:

```
(
var a = 6, z;
z = if (a<5, { 100 },{ 200 });
z.postln;
)
```

Se la variabile *a* è minore del valore 5, allora la variabile *z* restituirà il risultato 100, altrimenti verrà valutata la seconda funzione contenente il valore 200. Proviamo ad applicare questo semplice concetto per controllare il numero di elementi di una lista, esempio:

```
(
var a = 6, z;
z = if (a<5, { 5 },{ 10 });
Array.series(z, 0, 10); //z corrisponde al parametro size
)
```

Il prossimo esempio descrive uno dei cicli maggiormente utilizzati in *SuperCollider*, il costrutto *for*, vediamo un esempio esempio con cui costruire un contatore, ossia una

struttura numerica legata ad un range definito in cui si passa da un valore al successivo per mezzo di un determinato incremento.

```
~min = 1; //valore minimo
~max = 10; //valore massimo
for (~min, ~max, //tre parametri (min, max, funzione)
 {
  arg i;
  i.postln;  //stampa la lista da 1 a 10
 });
```

valutiamo l'esempio ottenendo la lista:

```
1
2
3
4
ecc.
```

Il prossimo esempio realizza un processo iterativo di una determinata funzione, molto utile per processi generativi musicali in cui è anche possibile definire l'intervallo di tempo tra una iterazione e la successiva. Vedremo nei capitoli successivi (ad esempio quello sulla sintesi additiva) come ascoltare, in successione, le parziali di una frequenza fondamentale; la sintassi della routine con *do* è la seguente:

```
{10.do( // 10 indica il numero delle ripetizioni
 { //inizio dello spazio dedicato alla funzione
// scrivere del codice
 1.wait;  // intervallo di tempo tra un'istanza e la successiva
 } //chiusura della funzione
)}.fork; // mette in sequenza le varie istanze
```

Proviamo a inserire qualche frammento di codice nello spazio dedicato alla funzione, per esempio una semplice stringa di testo come *"hello".postln*, valutiamo il codice ed osserviamo l'output nella *Post window*, l'istruzione *1.wait* stamperà il messaggio *hello* in sequenza ad un intervallo di tempo della durata di un secondo.

Possiamo anche creare accelerazioni e ritardi sull'intervallo di tempo, per esempio associando un *array* al metodo *wait*, il seguente esempio sceglie in modalità casuale un valore dalla lista modificando quindi l'andamento ritmico delle varie istanze in successione.

```
[0.1, 0.2, 0.3, 0.4, 1, 1.2, 0.05, 2].choose.wait
```

oppure:

```
100.rand.wait
```

Elementi del linguaggio incontrati in questo capitolo

post
postln
round
log
abs
reciprocal
ceil
floor
midicps

arg

var

Array.series

Array.fill
Array.rand
Array.geom
Array.interpolation
Array.fib

choose
reverse
mirror
size
scramble
rand
rand2
rotate