ANNO 2 // N. 2 // MAGGIO 2024

auditoriuM

SUPPLEMENTO

DIGITAL SIGNAL PROCESSING, ACUSTICA E COMPOSIZIONE

ISSN 2974-9360

WWW.CONSERVATORIOCOSENZA.IT

auditoriuM

Rivista del Conservatorio di Musica “Stanislao Giacomantonio” di Cosenza
Supplemento II, 2024
ISSN 2974-9360
Registrazione presso il Tribunale di Cosenza
Registro Generale n° 2973/2023 e del Registro Stampa N° 2/23

INDICE

Luca Bimbi

Una introduzione pratica a Faust

Cosa è Faust

Faust (Functional Audio Stream) è un linguaggio di programmazione ad alto livello diretto al Digital Signal Processing, soprattutto nel dominio temporale. In particolare, Faust è capace di generare output per una serie cospicua di linguaggi di programmazione: C, C++, WebAssembly, Java, eccetera. Il codice generato per effetto della compilazione è ad alta efficienza. Una installazione di Faust contiene anche una serie di script chiamati faust2 (da leggersi faust to…), che consentono di compilare il codice Faust per una serie di destinazioni ("architetture") che includono, fra le varie possibili, plugins AudioUnit, VST, External Max, external PureData e file SVG rappresentante il diagramma a blocchi del processamento.

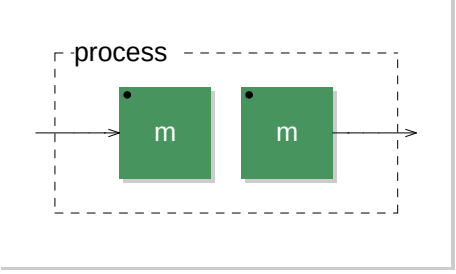
Cosa occorre per eseguire gli esempi

Questa introduzione, tranne che per l'ultimo paragrafo, non richiede necessariamente una installazione di Faust sul proprio computer. Sarà sufficiente ricorrere a un browser web ed utilizzare l'IDE online all'URL https://faustide.grame.fr. Gli esempi dovranno essere digitati nell'editor dell'IDE ed eseguiti per mezzo del bottone Run. Sarà possibile interrompere l'esecuzione chiudendo il tab DSP. Si suggerisce l'utilizzo di Google Chrome. L'ultimo paragrafo richiede una installazione di Csound e Faust nel sistema, reperibili ai siti web https://csound.com/download.html e https://faust.grame.fr/downloads/.

Programmazione funzionale e le basi della sintassi

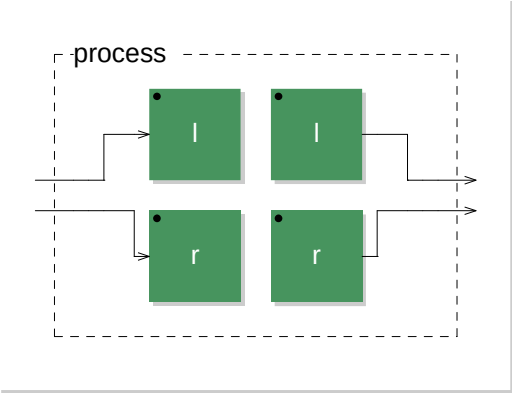
Faust è un linguaggio di programmazione **funzionale**, si regge cioè su tale specifico paradigma di programmazione, analogo al concetto di funzione matematica. Ogni programma Faust deve obbligatoriamente includere una funzione chiamata **process** (l'analogo di *main* in C e C++). Ogni funzione, in Faust, descrive un *diagramma a blocchi* ed è terminata da un punto e virgola. Ciascun diagramma a blocchi contiene un determinato numero di segnali in ingresso ed uscita. Faust accetta, come il C ed il C++, commenti nella forma `//` o `/* ... */`. Il più semplice programma Faust è il seguente:

```
process(m) = m;
```



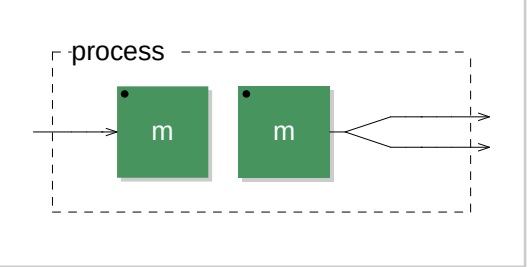
Si avrà una singola connessione fra un solo input ad un solo output (connessione monofonica). Nel caso della connessione stereofonica si potrà ricorrere all'operatore virgola `,` che indica connessione parallela, e definire quanto segue:

```
process(l,r) = l,r;
```



Vediamo come prendere un segnale monofonico e dividerlo su due uscite, mediante l'operatore split `<:`

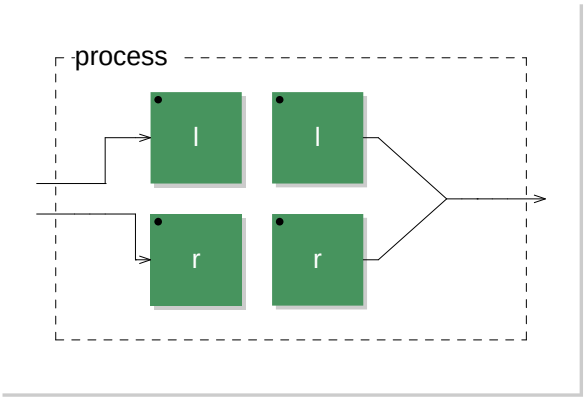
```
process(m) = m<:_,_;
```



Dove il tratto inferiore `_` chiamato **wire** rappresenta un segnale. In questo caso, il segnale `m` viene diviso in due segnali paralleli.

Il processo inverso fa ricorso all'operatore merge `:>`

```
process(l,r) = l,r:>_;
```



Se volessimo ricorrere alle etichette, dovremmo definirle precedentemente. Ad esempio:

```
l = _;
r = _;
process(m) = m<:l,r;
```

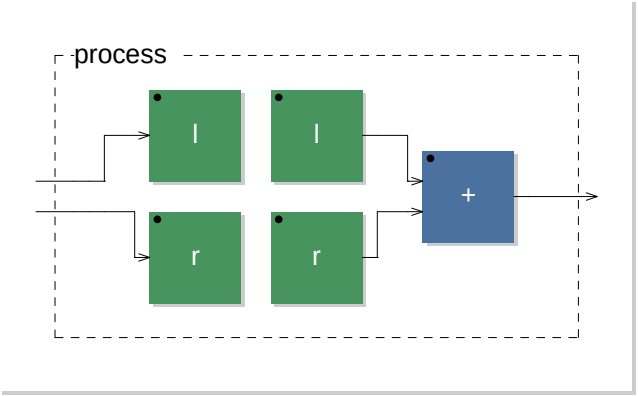
oppure:

```
m = _;
process(l,r)= l,r:>m;
```

La connessione in serie si può attuare mediante l'operatore due punti `:`

Vediamo come prendere due segnali e sommarli su un'unica uscita mediante l'addizione esplicita:

```
process(l,r) = l,r:+;
```



Dal momento che l'operatore + sottintende la presenza di due segnali, l'istruzione può essere riscritta come segue:

```
process = +;
```

Faust accetta i quattro operatori matematici `+,-,*,/`, oltre che l'operatore modulo `%`. Faust mette altresì a disposizione gli operatori di comparazione `<,>,<=,>=,!=,==.` Sono altresì previsti i bit operators `<<` e `>>`.

Possiamo rivedere gli esempi precedenti utilizzando il wire:

```
process = _;
```

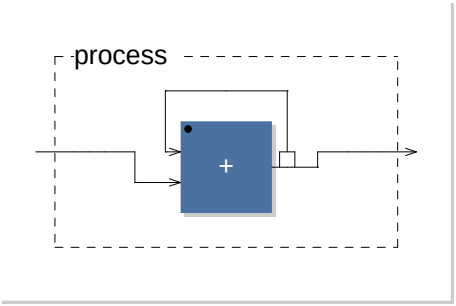
```
process = _,_;
```

```
process = _<:_,_;
```

```
process = _,_:>_;
```

Fra gli operatori possibili indichiamo anche l'operatore feedback tilde `~`. Tale operatore sottintende che un ingresso viene utilizzato proprio per il ritorno del segnale. Ad esempio:

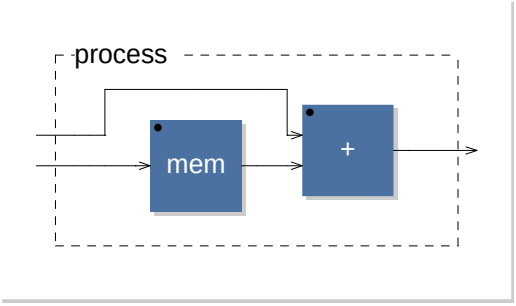
```
process = _+_~_;
```



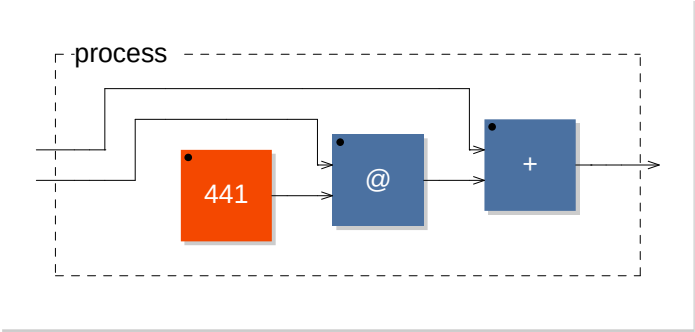
Determina la somma fra il segnale in ingresso diretto e quello determinato dal feedback. L'operatore `~` sottintende sempre il delay di un sample, rappresentato nel diagramma a blocchi dal quadrato che precede la freccia del feedback verso il primo ingresso.

Il delay di un sample su un segnale può essere indicato mediante l'operatore apice `'`. Un delay di quantità di samples diverse, mediante l'operatore a commerciale `@`. Si vedano i due seguenti esempi:

```
process = _+_';
```



```
process = _+_@(441);
```



Rumore bianco, spazializzazione e filtraggio

Cominciamo col generare rumore bianco con un livello di -18 dbFS. Ricorriamo a due funzioni presenti nella libreria standard di Faust, che importiamo con specifico statement. `ba.db2linear` prende come argomento un valore numerico in dbFS e restituisce un valore di livello lineare ricompreso fra 0 ed 1. `no.noise`, invece, genera rumore bianco. Moltiplichiamo per `level` per ottenere l'attenuazione richiesta.

```
import("stdfaust.lib");
level = ba.db2linear(-18); // -18 dbFS
process = no.noise*(level) : _; // applica un livello di uscita pari a level
```

Possiamo scrivere la medesima cosa utilizzando etichette l e r per i due canali:

```
import("stdfaust.lib");
level = ba.db2linear(-18);
l = _;
r = _;
process = no.noise*(level) <: l, r;
```

Questo ci consente, ad esempio, di specificare in modo chiaro livelli diversi sui due canali:

```
import("stdfaust.lib");
l = _*(ba.db2linear(-12)); // livelli diversi sui due canali
r = _*(ba.db2linear(-18));
process = no.noise <: l,r;
```

La libreria standard include una serie di funzioni legate al panning ed alla spazializzazione. `sp.panner` effettua un panning lineare; se come argomento ha valore 0, il pan è tutto a sinistra, se è 1, tutto a destra. 0.5 dà il pan centrale.

```
import("stdfaust.lib");
level = ba.db2linear(-18);
l = 0;
r = 1;
process = no.noise*(level) : sp.panner(l): _, _; // panning lineare
```

Inseriamo adesso uno slider orizzontale, mediante la funzione `hslider`, la cui sintassi è:

```
nome = hslider("etichetta",valoredefault,minimo,massimo,step)
```

nome è nome della funzione, "`etichetta`" è una stringa che apparirà sulla GUI in corrispondenza d ello slider orizzontale, `valoredefault` è il valore che lo slider assumerà in partenza fra `minimo` e `massimo` e la variazione è definita dallo `step`.

```
import("stdfaust.lib");
level = ba.db2linear(-18);
pan = hslider("Pan",0,0,1,0.1); // slider orizzontale
process = no.noise*(level) : sp.panner(pan): _, _;
```

Sostituiamo quindi il panning lineare col panning a potenza costante, dato dalla funzione `sp.constantPowerPan`.

```
import("stdfaust.lib");
level = ba.db2linear(-18);
pan = hslider("Pan",0,0,1,0.1);
process = no.noise*(level) : sp.constantPowerPan(pan): _, _; // pot. costante
```

Vediamo infine un esempio di utilizzo di spazializzazione che tiene in considerazione due parametri: la distanza e la rotazione. `sp.spat` accetta come parametri il numero di speaker, un valore normalizzato per la rotazione ed uno normalizzato per la distanza. `si.bus` pone il numero di canali specificato in parallelo.

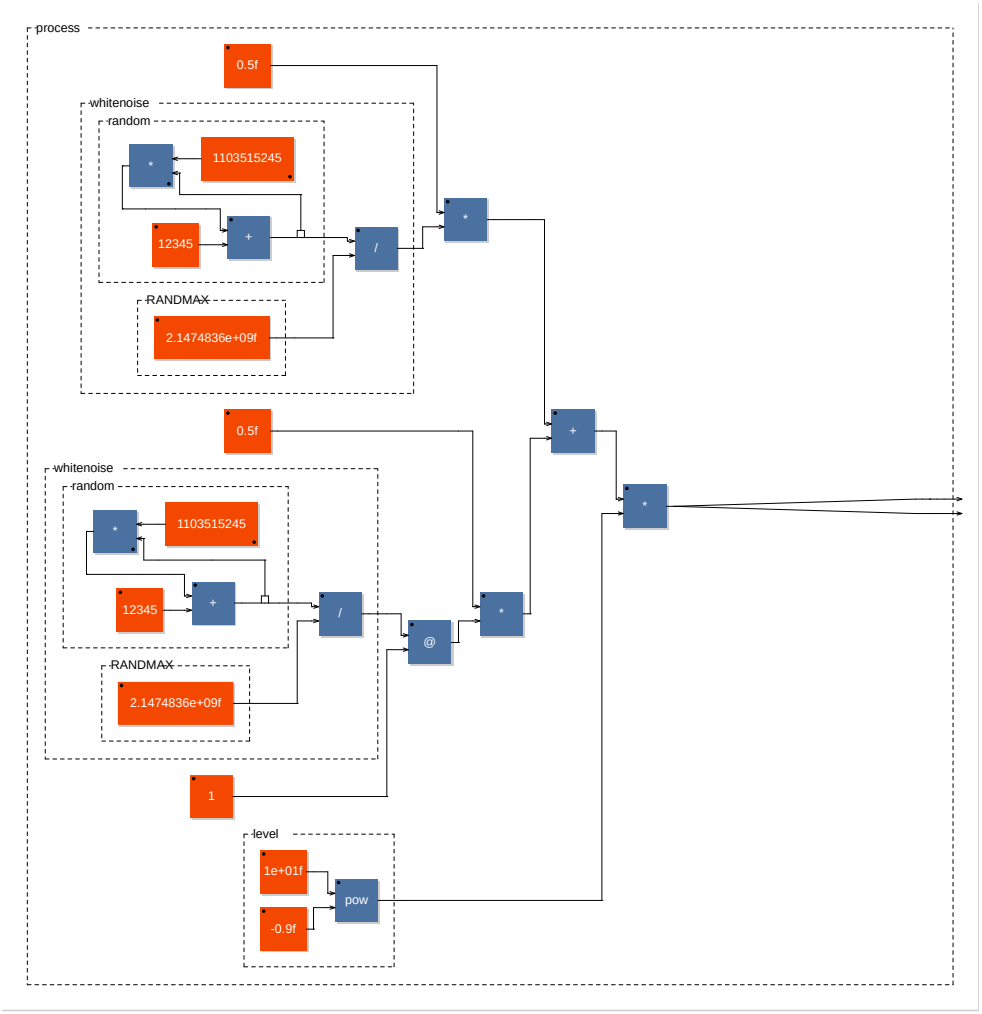
```
import("stdfaust.lib");
level = ba.db2linear(-18);
rot = hslider("Rotazione",0,0,1,0.01);
dis = hslider("Distanza", 1,0,1,0.01);
nspeaker = 2;
// esempio di utilizzo di spat, data distanza rotazione e numero di speaker
process = no.noise*(level) : sp.spat(nspeaker, rot, dist): si.bus(nspeaker);
```

Passiamo adesso ad una delle tematiche centrali del DSP: il filtraggio di segnale. Considerando l'equazione alle differenze di un filtro FIR (Finite Impulse Response) passa basso a media mobile del primo ordine:

$$y(n) = 0.5 * x(n) + 0.5 * x(n-1)$$

Considerando come segnale in ingresso del rumore bianco, possiamo scrivere in codice quanto segue:

```
import("stdfaust.lib");
level = ba.db2linear(-18);
whitenoise = no.noise;
process = (0.5*whitenoise+0.5*whitenoise@(1))*(level)<:_,_;
// whitenoise@(1) equivale a whitenoise'
```



Cambiando il segno nell'equazione alle differenze, si ottiene un filtro passa alto del primo ordine:

$$y(n) = 0.5 * x(n) - 0.5 * x(n-1)$$

```
import("stdfaust.lib");
level = ba.db2linear(-18);
whitenoise = no.noise;
process = (0.5*whitenoise-0.5*whitenoise@(1))*(level)<:_,_;
```

Con altra equazione alle differenze, considerando gli ultimi due samples in ingresso:

$$y(n) = 0.5 * x(n) + 0.5 * x(n-2)$$

Otteniamo il filtro FIR rigetta banda del secondo ordine. Il ritardo di due samples è indicato dall'operatore `@(2)`, ma sarebbe utilizzabile anche il doppio apice (`wnoise''`).

```
import("stdfaust.lib");
level = ba.db2linear(-18);
wnoise = no.noise;
process = (0.5*wnoise+0.5*wnoise@(2))*(level)<:_,_;
```

Cambiando il segno, si ottiene un filtro banda passante FIR del secondo ordine:

$$y(n) = 0.5 * x(n) - 0.5 * x(n-2)$$

```
import("stdfaust.lib");
level = ba.db2linear(-18);
wnoise = no.noise;
process = (-0.5*wnoise+0.5*wnoise@(2))*(level)<:_,_;
```

Passando invece alle tipologie di filtri IIR, cioè ricorsivi, otteniamo per l'equazione alle differenze per il filtro passa basso:

$$y(n) = ax(n) - by(n-1)$$

con a e b pari entrambi a 0.5.

```
import("stdfaust.lib");
level = ba.db2linear(-18);
whitenoise = no.noise;
filter = _ + *(0.5)~*(0.5);
process = (whitenoise : filter)*(level) <: _,_;
```

Se vogliamo creare un filtro passa basso IIR a frequenza variabile che sia l'analogo dell'opcode `tone` in Csound, dobbiamo calcolare i coefficienti come segue:

$$c = 2 - \cos(2\pi fc/sr)$$

$$b = \sqrt{c^2 - 1} - c$$

$$a = 1 + b$$

E otteniamo, in codice:

```
import("stdfaust.lib");
level = ba.db2linear(-18);
frequenza = hslider("Frequenza", 1000, 0, 10000, 10);
c = 2-cos((2*ma.PI*frequenza)/ma.SR);
b = sqrt(c^2-1)-c;
a = 1+b;
// y(n)=ax(n)-by(n-1)
whitenoise = no.noise;
filter = _+*(a)~*(-b);
process = (whitenoise : filter)*(level) <: _,_;
```

`ma.PI` restituisce il valore di pi Greco in precisione doppia. Eliminando il segno meno dal coefficiente `b` in filter, si ottiene un filtro passa alto, analogo dell'opcode Csound `atone`.

Le librerie filters `(fi)` e vaeffects `(ve)` includono numerose tipologie di filtri. Si riportano due esempi diversi di utilizzo.

Il primo esempio vede un semplice lowpass risonante:

```
import("stdfaust.lib");
level = ba.db2linear(-18);
frequenza = hslider("Frequenza", 1000, 0, 10000, 10);
q = hslider("Q", 0.5, 0.5, 10, 0.01);
gain = hslider("Gain", 0.7, 0, 1, 0.01);
process = no.noise : fi.resonlp(frequenza, q, gain) <: _*(level),_*(level);
```

Il secondo, vede invece l'uso di un filtro lowpass modellato sulla tipologia Oberheim. Il valore di frequenza deve essere normalizzato e ricompreso fra i valori zero e uno.

```
import("stdfaust.lib");
level = ba.db2linear(-18);
frequenza = hslider("Frequenza", 0.3, 0, 1, 0.01);
q = hslider("Q", 0.5, 0.5, 10, 0.01);
process = no.noise : ve.oberheimLPF(frequenza, q) <: _*(level),_*(level);
```

Incremento e ramp generator; sinusoide. Cenni agli iteratori ed esempi d'uso

Uno degli strumenti più utili nell'ambito dell'informatica musicale è il *ramp generator*, ossia un dispositivo che mandi in output in modo continuo e regolare da zero a uno. Possiamo fare ricorso agli operatori incontrati ed alla funzione `ma.frac` che restituisce la parte frazionaria di un numero, per prevenire errori. Nel contatore dobbiamo considerare l'incremento necessario che dipende dalla frequenza desiderata divisa per la frequenza di campionamento. `ma.SR` ci consente proprio di ottenere la frequenza di campionamento in uso, quindi:

```
import("stdfaust.lib");
frequenza = hslider("Frequenza", 440, 100, 4500, 10);
incremento = frequenza/ma.SR;
process = (+(incremento) ~ ma.frac);
```

E questo restituisce un'onda a dente di sega unipolare non band-limited.

Possiamo utilizzare questo ramp generator per ottenere una sinusoide in tempo reale, ricorrendo ovviamente alla funzione seno (`sin`).

```
import("stdfaust.lib");
level = ba.db2linear(-18);
frequenza = hslider("Frequenza", 440, 100, 4500, 10);
incremento = frequenza/ma.SR;
phasor(f) = (+(incremento) ~ ma.frac);
osc(f)= sin(2*ma.PI*phasor(f));
process = osc(frequenza)*(level) <: _,_;
```

Faust mette a disposizione del programmatore degli iteratori (i corrispettivi di `for` in diversi linguaggi di programmazione).

Gli iteratori in Faust sono:

1. `par` per la duplicazione di una espressione in parallelo;

2. `seq` per la duplicazione di una espressione in serie;

3. `sum` per la duplicazione di una espressione come somma;

4. `prod` per la duplicazione di una espressione come prodotto.

Per tutti gli iteratori:

1. il primo argomento è il nome di una variabile che contiene l'iterazione corrente, e parte dal valore zero;

2. il secondo argomento è il numero di iterazioni come valore intero;

3. il terzo argomento è l'espressione da duplicare.

Vediamo quindi un semplice esempio di sintesi additiva mediante l'iteratore sum. Si avrà una somma di oscillatori sinusoidali `os.oscsin`.

```
import("stdfaust.lib");
frequenza = hslider("Frequenza",440,100,880,10);
level = ba.db2linear(-12);
nparziali = 24;
addsum = sum(indice, nparziali, os.oscsin(frequenza*(indice+1)))/(nparziali);
process = addsum<:_,_;
```

Si potrebbe ottenere il medesimo risultato mediante l'iteratore par, assieme all'operatore merge `:>`

```
import("stdfaust.lib");
frequenza = hslider("Frequenza",440,100,880,10);
level = ba.db2linear(-12);
nparziali = 24;
addpar = par(indice, nparziali, os.oscsin(frequenza*(indice+1))):>/(nparziali);
process = addpar*level<:_,_;
```

Un esempio interessante di utilizzo dell'iteratore `par` può essere individuato nella realizzazione di un filtro formante. Ci avvaliamo di un primitivo del linguaggio, `waveform` la cui sintassi è:

```
etichetta = waveform{a, b, c, ...};
```

che ci consente di rappresentare un table di dati. Tale table è leggibile mediante un altro primitivo del linguaggio, `rdtable.` La sintassi di `rdtable` è:

```
table, indice : rdtable;
```

Passiamo quindi come argomenti la table, che nel nostro caso è l'array generato con `waveform,` e l'indice del valore da leggere. L'operazione di lettura è definita in una funzione *ad hoc*, `read(array, index)`.

Utilizziamo tre table: una per le frequenze formanti, una per la metà della larghezza di banda ed una per l'ampiezza delle stesse. Inseriremo nell'iteratore il filtro Butterworth dalla libreria, `fi.bandpass` la cui sintassi è:

```
fi.bandpass(N, fl, fu)
```

Ossia prende come argomenti un valore che rappresenta la metà dell'ordine del bandpass, la frequenza inferiore e la frequenza superiore.

Usiamo inoltre `si.smoo` per fare lo smoothing nel cambio di frequenza dello slider, evitando fastidiosi click. Il segnale in ingresso è costituito da un'onda a dente di sega generata mediante `os.sawtooth.`

```
import("stdfaust.lib");
level = ba.db2linear(-12);
frmnts = waveform{400,800,2600,2800,3000}; // valori in hz
hbw = waveform{36, 40, 50, 66, 68};
amp = waveform{0,-10, -12,-12,-26}; // valori in dBFS
frequenza = hslider("Frequenza", 220, 110, 440, 10) : si.smoo;
saw = os.sawtooth(frequenza);
read(array, index) = array, index : rdtable;
vwl = par(i, 5, saw : fi.bandpass(1, (read(frmnts,i)-read(hbw,i)),
      (read(frmnts,i)+read(hbw, i)))*(ba.db2linear((amp, i: rdtable)))) :> _;

process = vwl*level<: _, _;
```

Cenni per l'utilizzo elementare di Faust all'interno di Csound

Csound, come linguaggio a dominio specifico per le applicazioni sonore e musicali, può fungere da valido motore audio per il codice Faust. I tre opcode fondamentali per l'utilizzo di codice Faust all'interno di Csound sono `faustcompile`, `faustaudio` e `faustctl`. Gli opcode fanno parte del plugin repository di Csound, accessibile al sito: https://github.com/csound/plugins. Richiedono la presenza nel sistema di *libfaust*. `faustcompile` invoca il compilatore just-in-time per produrre un processo DSP instanziabile partendo da un programma Faust.

La sintassi essenziale di `faustcompile` è:

`ihandle faustcompile Scode, Sargs`

Ossia restituisce un handle a init-time ed accetta in input una stringa per il codice Faust ricompresa fra `""` o `{{}}`, ed una stringa per gli argomenti del compilatore Faust.

L'opcode `faustaudio` ha invece la seguente sintassi:

`ihandle, a1[,a2...] faustaudio ifac[,ain1,...]`

`ifac` è l'handle prodotto da `faustcompile`, `ihandle` è un handle verso l'istanza di Faust DSP. `a1` e successivi sono i segnali di output e `ain1` e successivi i segnali di input.

`faustctl` regola un dato controllo nel codice Faust. La sintassi è:

`faustctl idsp,Scontrol, kval`

dove `idsp` è l'handle restituito da `faustaudio`, `Scontrol` una stringa che contiene il nome del controllo da regolare, `kval` il valore a control time che regolerà il controllo nel codice Faust.

Vediamo un semplice esempio applicativo dove riprendiamo il precedente esempio col filtro Oberheim. Mediante delle variabili a control-time gestiamo gli slider previsti nel codice Faust per il controllo della frequenza e del Q per il Low Pass Filter Oberheim. Il segnale in ingresso viene fornito a Faust da `vco2` che genera un'onda a dente di sega ad una frequenza di 440 hz.

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 64
0dbfs = 1
nchnls = 2

instr 1
  kfreq linseg 0, p3/2, 1, p3/2, 0
  kq linseg 0.5, p3/2, 10, p3/2, 0.5
  kenv linen 0dbfs, 0.1, p3, 0.1
  ain vco2 kenv, 440
  ihandle faustcompile {{
    import("stdfaust.lib");
    level = ba.db2linear(-18);
    frequenza = hslider("Frequenza", 0.3, 0, 1, 0.01);
    q = hslider("Q", 0.5, 0.5, 10, 0.01);
    process = _ : ve.oberheimLPF(frequenza, q) <: _*(level),_*(level);
}}, ""
  idsp, al, ar faustaudio ihandle, ain
  faustctl idsp, "Frequenza", kfreq
  faustctl idsp, "Q", kq
  out al, ar
endin
</CsInstruments>
<CsScore>
i 1 0 5
</CsScore>
</CsoundSynthesizer>
```

Otteniamo quindi lo sweep a control time del filtro con controllo sulla frequenza e sul Q per il tempo di esecuzione.

Marco Giommoni

Composizione musicale come strategia operativa

La necessità di ridiscutere i metodi e le procedure attraverso i quali si giunse alla definizione delle strutture della Nuova Musica a partire dalla seconda metà degli anni ’50 ha stimolato la nascita e lo sviluppo di una nuova e peculiare area di ricerca: la «composizione parametrica» (o «algoritmica»), ove si affronta il problema della formalizzazione del linguaggio musicale attraverso lo studio e l’applicazione alla composizione della musica di specifiche procedure traducibili in altrettanti algoritmi.

Oggi, più o meno tutti gli autori di lavori indirizzati ai futuri sviluppi della teoria della musica concordano sul fatto che questo settore possa costituire lo strumento più potente, estensibile ed aggiornabile per l’allargamento degli orizzonti del linguaggio musicale stesso, nonché possa lasciar intravvedere una prospettiva più organica e meno ambigua nell’approccio alle forme espressive della musica, del tempo presente.

Ciò ovviamente premette una radicale ridiscussione delle tradizionali definizioni degli apparati teorici attraverso i quali si sono sempre legittimate le tradizionali prassi compositive, primo fra questi una ridefinizione del concetto stesso di «teoria della musica», una volta accolta come «teoria del suo linguaggio»

Come ci ricorda Hugues Dufourt, la teoria della musica trova la sua condizione necessaria nel reciproco adeguamento di strutture formalizzate, modelli operativi, e connessioni di simboli.

Oggi una teoria della musica deve potersi legittimare anche come teoria della generazione di nuovi oggetti musicali.

In tale contesto una formalizzazione può essere assimilata, nelle linee generali, all’elaborazione in termini razionali di schemi teorici astratti, ed in questo senso essa caratterizza tutte le prassi compositive. In questi termini una formalizzazione non potrà che consistere di un insieme di regole di organizzazione musicale che si traducono un modello operativo suscettibile di applicazione a partire da una riflessione teorica iniziale, e per questo dipendente da alcune condizioni e criteri preliminari, e diretto al conseguimento di uno specifico obbiettivo. Tale prassi procede dunque dalla riflessione e mira ad un obbiettivo: dopo la concezione del sistema segue necessariamente la sua messa in opera.

Su tale premessa possiamo definire una strategia operativa per la scrittura della musica come una procedura formalizzata che si articola in quattro distinti momenti:

(1) *Ideazione*. Determinazione di uno o più schemi concettuali come riflessione, ipotesi, insieme di congetture sulla natura, sulle qualità e sulle possibili relazioni del materiale musicale, in previsione

del suo costituirsi in una struttura effettiva. Possiamo definire questo momento "livello dell'invenzione".

(2) *Progettazione*. Elaborazione di un modello con il quale rendere esplicito e praticabile lo schema concettuale idealizzato. Questo modello contemplerà tutti gli eventuali i dispositivi generativi e trasformativi ai quali si affida il compito di produrre ed elaborare il materiale musicale e i principi normativi che ne regolano l'aggregazione, sia che ciò avvenga attraverso la formulazione ex novo di regole, limiti, scelte di metodo ed altro, sia mediante la collazione e la riformulazione di principi già a suo tempo definiti. Il modello, sebbene ancora momento di rappresentazione astratta, teorica, deve essere tuttavia strutturato secondo precisi criteri che ne consentano una trasformazione operativa, in forma, cioè di passi successivi gerarchicamente organizzati secondo una logica sequenziale, congruente e produttiva, in grado di fornire un risultato effettivo, concreto e coerente con i presupposti che stanno alla base del modello stesso. Possiamo definire questo momento "livello del modello".

(3) Programmazione. Traduzione del modello in uno schema attuativo, in un programma operativo; passaggio che implica l'individuazione di codici interpretativi e di adeguati strumenti di rappresentazione, nonché la definizione dei principi e delle norme che ne stabiliscono i criteri di applicazione. Se ciò avviene in termini di descrizione formale, come processo o catena di processi che possono essere descritti in termini espliciti e rigorosi, quest'indagine si può indirizzare alla ricerca di specifiche funzioni traducibili in procedure informatiche le quali, agendo su elementi musicali di base secondo l'organizzazione e la successione logica loro conferita dal modello, sono effettivamente e concretamente in grado di produrre la struttura cercata. Possiamo definire questo momento "livello del programma".

(4) *Applicazione*. Traduzione dello schema, in forma eseguibile (una partitura, i dati di un *input*, ecc.) che attraverso la mediazione di un interprete o di uno strumento informatico sia in grado di restituire l'oggetto musicale cercato. Possiamo definire questo momento "livello della scrittura".

Il compositore austriaco Karlheinz Essl fa rilevare che, nonostante una siffatta teoria descriva un iter produttivo che muove dall'astratto al concreto, da generiche congetture sul materiale all'oggetto musicale definito, la sua articolazione nelle quattro fasi non deve considerarsi una successione irreversibile e irrevocabile di passi rigidamente compartimentati, ma ogni passaggio implica dei correttivi e dei ripensamenti che danno al compositore la facoltà di riconsiderare quanto elaborato in precedenza, e di converso le ipotesi di partenza, per quanto generiche ed astratte, devono sempre far implicito riferimento a precise regole attuative, a specifici meccanismi generativi e normativi: in altre parole, questa teoria della scrittura obbedisce ai criteri fondamentali del controllo cibernetico, ottenendo un'omeòstasi (autoregolazione) del sistema attraverso l'applicazione di dispositivi di anticipazione e retroazione (*feedforward* e *feedback*)

Dunque, alle convenzionali definizioni di "abbozzo" o "schizzo", generiche ed ambigue, subentra il concetto assai meglio definito e preciso di "strategia compositiva" (*composition design*) che Richard Morris indica così: «in generale le strategie compositive possono modellare tutto, dai passaggi più semplici ai brani più complessi. Possono essere altresì considerate tentativi di suggerire o

implementare una sintassi musicale sia in assenza che in presenza di una grammatica convenzionale».[1]

Per chiarire in che cosa in sostanza consista la «strategia compositiva» Morris s'interroga su «come un compositore costruisce le strategie che egli immagina»,[2] ed allora egli vede la necessità di introdurre il concetto di «spazio compositivo» (*compositional space*) come «un insieme di oggetti musicali collegati e/o connessi in almeno un modo specifico».[3] Ma, cosa assai più importante, gli spazi compositivi sono interpretati in modo non temporale, cioè essi risultano "atemporali". In altri termini questi "spazi compositivi" sono «strutture generiche fuori dal tempo dalle quali possono essere desunte strategie compositive, meglio specificate e collocate nel tempo».[4] Ma un siffatto "spazio compositivo" non avrebbe senso se non dotato di alcuni peculiari caratteri che lo rendono effettivamente e concretamente capace di costituire una vera e propria cornice interpretativa entro la quale il compositore può attivare le strategie operative di volta in volta individuate. Questi caratteri si riassumono in: (1) la disponibilità di un linguaggio simbolico interpretabile in termini musicali; (2) la possibilità di operare su di esso tramite procedure classificabili (per qualità e/o modalità operative); (3) l'interattività; (4) l'estensibilità (sia del linguaggio che delle procedure).

[1] Robert Morris, *Compositional Spaces and Other Territories*, in «Perspectives of New Music», Vol. 33, n. 1/2, 1995, p. 329, (trad. di Marco Giommoni).
[2] *Ivi*, p. 330, (traduzione in italiano di Marco Giommoni).
[3] *Ivi*, p. 336, (traduzione in italiano di Marco Giommoni).
[4] *Ivi*, p. 330, (traduzione in italiano di Marco Giommoni).

Luca Bimbi

Il deciBel acustico e la direttività sonora

Watt acustico, Intensità, Pressione

Individuiamo i concetti di Watt acustico, Intensità acustica e Pressione Sonora.

Il Watt è un valore che esprime una quantità di lavoro svolta nell'unità di tempo, ed è indipendente da altri fattori, quali ad esempio la distanza.

Prendiamo in esame la legge di Ohm, che mette in relazione la resistenza (R) espressa in Ohm, la corrente (I) espressa in Coulomb e tensione (V) espressa in Volt:

$$R = \frac{V}{I}$$

La potenza (P) espressa in Watt (W), sarà data da:

$$P = V \times I$$

Il Watt acustico fa riferimento ad un valore di *potenza acustica*.

L'intensità acustica considera tale potenza acustica in relazione ad una *superficie*.

Nel caso di diffusione del suono nello spazio da *sorgente puntiforme omnidirezionale in campo libero*, tale superficie sarà quella di una *sfera*.

L'intensità sarà quindi data da:

$$I = \frac{W}{4\pi r^2}$$

Prendendo in esame le analogie acustiche con la legge di Ohm, riscontriamo che la corrente ha come analogo la velocità particellare, la tensione ha come analogo la pressione, la resistenza ha come analogo l'impedenza e la potenza ha come analogo l'intensità.

L'impedenza (Z) caratteristica dell'aria ρc è data dal rapporto della Pressione con la velocità particellare:

$$Z = \frac{P}{v} = \rho c$$

Dove ρ rappresenta la densità dell'aria, e c la velocità del suono. Entrambi dipendono dalla temperatura del medio. L'impedenza caratteristica è misurata in RAYLS.

Il livello di Pressione sonora in valore efficace, espresso in Pascal (Pa), è quindi uguale a:

$$P_{rms} = \sqrt{I \times \rho c}$$

L_w L_i, L_p

Una volta chiariti i concetti di base, possiamo calcolare corrispondenti valori di livello per il Watt acustico, l'Intensità acustica e la Pressione Sonora

Per il livello di potenza (L_w) e di intensità ((L_i), il riferimento è il *picoWatt*, ossia 10^{-12} W.

$$L_w = \frac{W}{10^{-12}}$$

$$L_i = \frac{W/m^2}{10^{-12} W/m^2}$$

Per il livello di pressione sonora (L_p), la soglia di udibilità, utilizzata come riferimento, è di $20\ \mu Pa$ ossia 10^{-5} Pa.

$$L_p = \frac{Pa}{2 \times 10^{-5} Pa}$$

La legge dell'inverso del quadrato

Spieghiamo adesso una relazione importante in molti ambiti della fisica, compresa la fisica acustica, chiamata legge dell'inverso del quadrato.

Immaginiamo di avere una sorgente acustica che emette una potenza di 1 W acustico.

Il livello corrispondente in deciBel sarà:

$$L_w = 10\log_{10}\left(\frac{1W}{10^{-12}W}\right) = 120\ dB$$

Consideriamo adesso di avere una sfera di $1\ m^2$, ciò comporta che il raggio *r* della sfera sia di circa 0.282 m

L'intensità I sarà quindi:

$$I = \frac{1\ W}{4\pi \times 0.282^2} \simeq 1\ W/m^2$$

ed il Livello di intensità in deciBel:

$$L_i = 10log_{10}\left(\frac{1\ W/m^2}{10^{-12} W/m^2}\right) = 120\ dB$$

Calcoliamo adesso la Pressione sonora, considerando una impedenza caratteristica dell'aria corrispondente ad un valore di 400 RAYLS, e successivamente, il livello di pressione sonora in deciBel:

$$P_{rms} = \sqrt{\frac{1W}{m^2} \times 400\ RAYLS} = \ 20\ Pa$$

$$L_p = \ 20\log_{10}\left(\frac{20\ Pa}{2 \times 10^{-5} Pa}\right) = \ 120\ dB$$

Supponiamo adesso di raddoppiare la distanza dalla sorgente puntiforme, e quindi di avere un raggio *r* pari a 0.564 m. L'intensità sonora sarà uguale a:

$$I \ = \frac{1\ W}{4\pi \times 0.564^2} \simeq \ \frac{1}{4}\ W/m^2$$

Al raddoppiare della distanza, l'intensità sonora si riduce ad un quarto di quella originaria. Convertito in deciBel:

$$L_i = 10log_{10}\left(\frac{1/4\ W/m^2}{10^{-12} W/m^2}\right) \ \simeq \ 114\ dB$$

Si ha quindi una perdita di intensità sonora pari a 6 dB.

Ricalcolando la pressione sonora e il corrispondente livello in deciBel:

$$P_{rms} = \sqrt{\frac{1}{4}\frac{W}{m^2} \times 400\ RAYLS} = \ 10\ Pa$$

$$L_p = \ 20\log_{10}\left(\frac{10\ Pa}{2 \times 10^{-5} Pa}\right) \simeq \ 114\ dB$$

Si ha, quindi, anche per il livello di pressione sonora una perdita di 6 dB, che corrispondono però non ad un quarto della pressione, ma alla metà.

Un livello di pressione sonora ad una certa distanza è, in pratica, calcolabile con la seguente formula:

$$L_p = \ 20\log_{10}\left(\frac{D1}{D0}\right)$$

Dove D0 rappresenta una distanza di riferimento, e D1 la distanza a cui si vuole calcolare la perdita di livello di pressione sonora.

Ciò si rivela particolarmente utile per valutare, data la sensibilità di un altoparlante considerando una emissione sferiforme, di quando decadrà il livello di pressione sonora ad una data distanza dal diffusore stesso.

Nel caso di array lineari di speaker, si dovrà considerare invece il caso della approssimazione di emissione cilindrica, per cui data la superficie laterale del cilindro uguale a $S_L = \ 2\pi \times r$, si avrà una perdita di intensità al raddoppio della distanza pari a 3 dB.

La direttività sonora

Immaginiamo adesso che la nostra emissione da sferica, ad esempio per una costrizione data da superfici, sia semisferica.

Seguendo la linea dell'esempio precedente, avremo per un raggio r pari a 0.282 m una Intensità di 0.5 W/m^2.

Il fattore di direttività (Q) di una emissione sferica si considera pari ad 1.

Immaginiamo quindi, considerando l'angolo solido in steradianti, di avere un pubblico di uditori situato a 0.282 m dal diffusore, coprenti una superficie di 0.5 m^2

L'angolo solido in steradianti (sr) è dato, matematicamente da:

$$sr = \frac{A}{r^2}$$

E al contempo, dato che l'angolo solido della sfera è pari a 4π, otteniamo:

$$sr = \frac{4\pi}{Q}$$

Possiamo, unendo le due formule, ricavare quindi il fattore di direttività:

$$Q = \frac{4\pi \times r^2}{A} = \frac{4\pi \times 0.282^2}{0.5} = 2$$

Una emissione emisferica avrà quindi Q = 2; una emissione di quarto di sfera Q = 4; una emissione di ottavo di sfera Q = 8, e così via.

L'indice di direttività, in deciBel, sarà uguale a:

$$I_D = 10\log_{10} Q$$

Nel nostro esempio, sarà quindi uguale a 3 dB.

CONSERVATORIO
DI MUSICA
COSENZA