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.

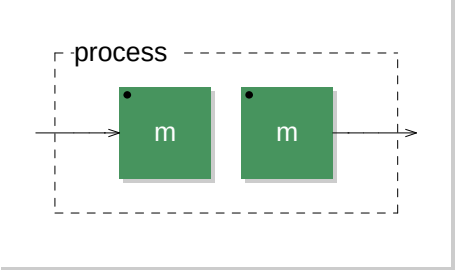
Cosa occorre per eseguire gli esempi

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

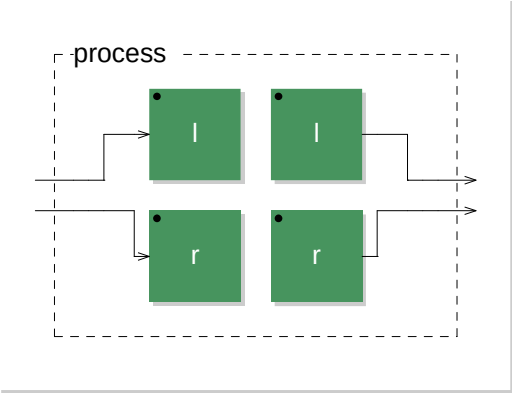
Faust è un linguaggio di programmazione **funzionale**, si regge cioè su tale specifico paradigma di programmazione, analogo al concetto di funzione matematica. Ogni programma Faust deve obbligatoriamente includere una funzione chiamata **process** (l'analogo di *main* in C e C++). Ogni funzione, in 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;
```



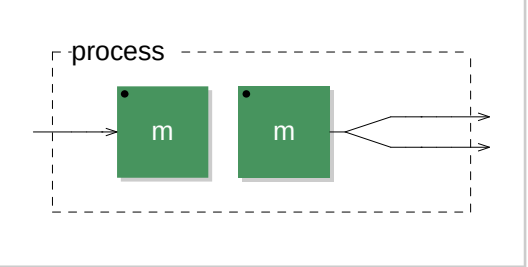
Si avrà una singola connessione fra un solo input ad un solo output (connessione monofonica). Nel caso della connessione stereofonica si potrà ricorrere all'operatore virgola `,` che indica connessione parallela, e definire quanto segue:

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



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

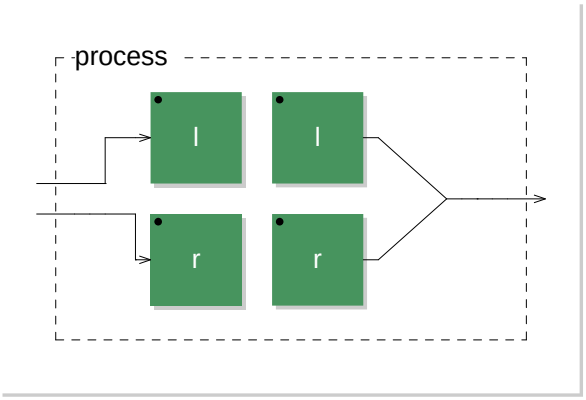
```
process(m) = m<:_,_;
```



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;
```

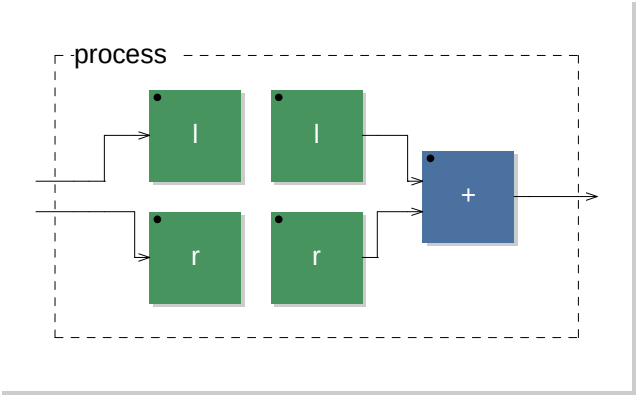
oppure:

```
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 = _;
```

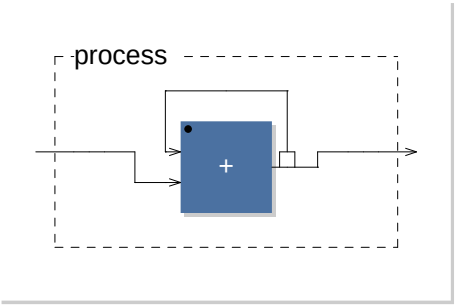
```
process = _,_;
```

```
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:

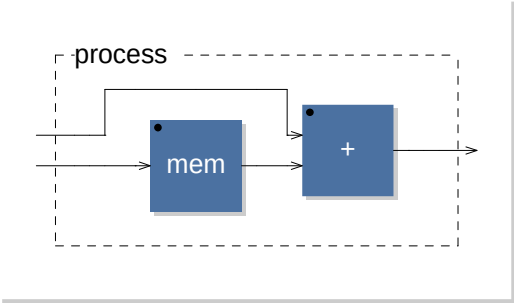
```
process = _+_~_;
```



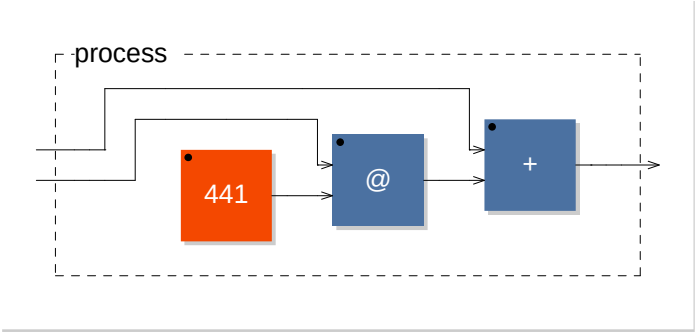
Determina la somma fra il segnale in ingresso diretto e quello determinato dal feedback. L'operatore `~` sottintende sempre il delay di un sample, rappresentato nel diagramma a blocchi dal quadrato che precede la freccia del feedback verso il primo ingresso.

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.

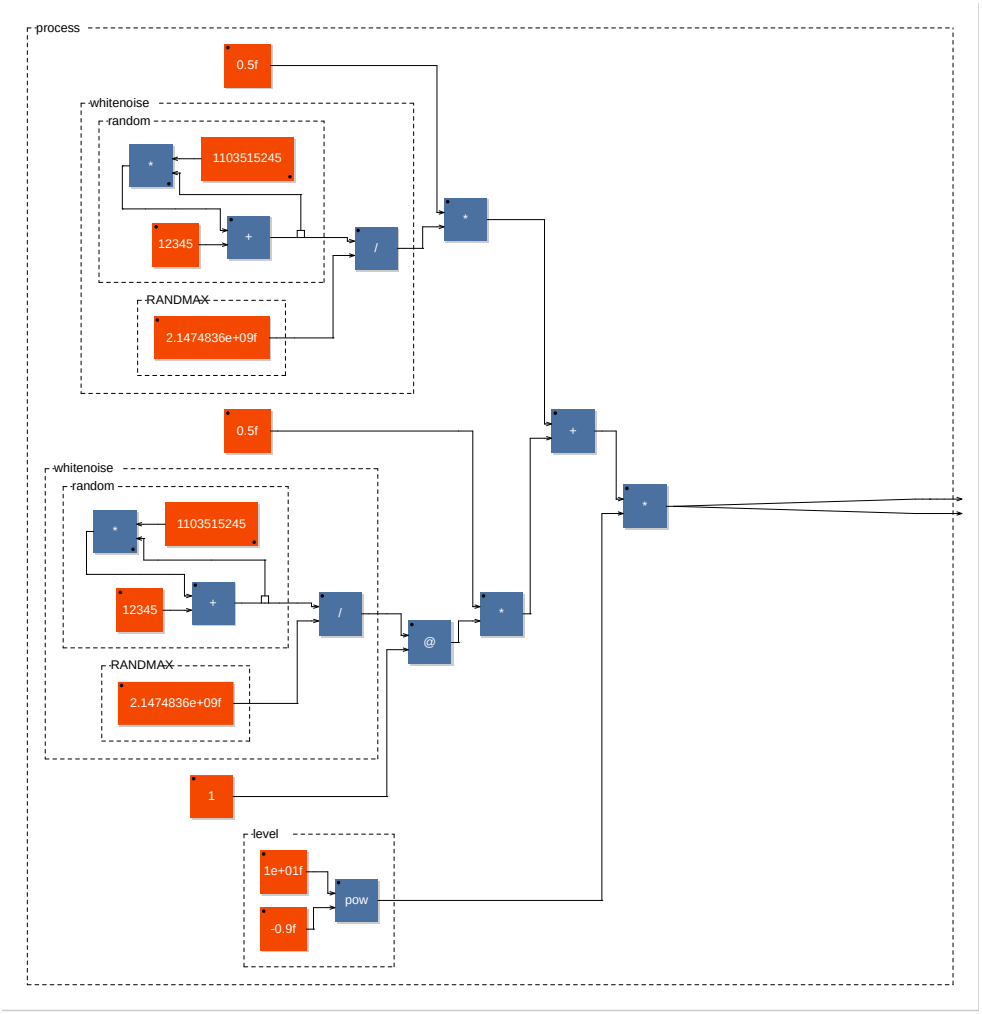
```
import("stdfaust.lib");
level = ba.db2linear(-18);
rot = hslider("Rotazione",0,0,1,0.01);
dis = hslider("Distanza", 1,0,1,0.01);
nspeaker = 2;
// esempio di utilizzo di spat, data distanza rotazione e numero di speaker
process = no.noise*(level) : sp.spat(nspeaker, rot, dist): si.bus(nspeaker);
```

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.