CALCUL D'UNE EXPRESSION FAUST ÉQUIVALENTE À PARTIR D'UN GRAPHE D'APPLICATIONS

S. Denoux *Y. Orlarey* *S. Letz* *D. Fober*
Grame
Centre nationale de création musicale
Lyon - France
{sdenoux,orlarey,letz,fober}@grame.fr

RÉSUMÉ

Nous proposons une méthode permettant de traduire un graphe de programmes FAUST, en un programme FAUST équivalent. Le programme ainsi obtenu peut être compilé, et donc bénéficier de toutes les optimisations du compilateur FAUST, mais il peut également être exporté vers les différentes plateformes supportées par FAUST (VST, Max/MSP, SuperCollider, Csound, etc.). Nous décrivons l'algorithme qui parcourt le graphe et calcule l'expression FAUST équivalente ainsi que les principes de modularité de FAUST qui permettent de combiner les fichiers sources pour réaliser l'opération. De plus, nous présentons une implémentation de l'algorithme dans le cadre de l'application FaustPlayground.

Keywords

FAUST, Composition, Web, programmation DSP

1. INTRODUCTION

Nous présentons un algorithme permettant de passer d'un patch graphique, sous la forme d'un graphe de noeuds représentant chacun un programme FAUST, à un programme FAUST unique équivalent.

Cet algorithme a été développé pour l'application Web FaustPlayground [1] qui propose une forme simplifiée de programmation FAUST. FaustPlayground permet de développer une application audio en connectant graphiquement des modules de haut-niveau écrits en FAUST. L'utilisateur peut ensuite exporter sa réalisation vers toutes les plateformes supportées par le service de compilation en ligne [2]. Pour pouvoir réaliser cette exportation, le patch/graphe de l'utilisateur doit préalablement être transformé en un code source FAUST unique, obtenu en collectant les implémentations FAUST de chaque noeuds du graphe.

L'objectif du présent article est de détailler comment cette transformation est réalisée. Cela suppose en particulier deux choses. D'une part d'être capable de combiner automatiquement des codes sources FAUST. C'est une opération qui serait extrêmement difficile à faire avec du code C par exemple, mais qui est ici possible grâce aux possibilités de modularisation du langage (voir paragraphe 2). D'autre part de traiter correctement tous les cycles du graphe à partir d'un parcours de type deep-first-search [6].

La méthode présentée ici pourrait, moyennant quelques extensions, s'appliquer à d'autres contextes que FaustPlayground, par exemple pour passer d'un patch Max/MSP ou PD à un programme FAUST équivalent.

La suite de l'article est organisée comme suit. La section 2 présentera les principes de composition du langage. On verra comment on peut combiner, au niveau du code source, plusieurs programmes FAUST pour en faire un seul. Ensuite, la section 3 présentera l'algorithme de parcours du graphe, et la façon dont sont traités les cycles. Enfin nous décrirons rapidement l'application FaustPlayground comme un cas concret d'utilisation de cette technique.

[1] http://faust.grame.fr/faustplayground/
[2] http://faustservice.grame.fr/targets

2. COMBINER DES PROGRAMMES FAUST

FAUST (http://faust.grame.fr) est un langage de programmation fonctionnel[4] synchrone spécifiquement conçu pour décrire des algorithmes de synthèse et de traitement du signal.

Un programme FAUST dénote un processeur de signaux : une fonction, au sens mathématique du terme, qui prend des signaux en entrée et produit des signaux en sortie. Programmer en FAUST consiste à combiner des processeurs de signaux élémentaires grâce à une algèbre de composition.

Ainsi par exemple si A et B sont deux processeurs de signaux, $(A : B)$ représente le processeur de signaux obtenu en branchant les sorties de A sur les entrées correspondantes de B. Tandis que (A, B) représente la mise en parallèle de A et B.

L'une des caractéristiques de FAUST, contrairement aux langages musicaux traditionnels comme Puredata, Csound, Max-MSP, etc., est d'être entièrement compilé. On peut utiliser FAUST à la place de C pour écrire par exemple des plugins audio. Les techniques de compilation mises en oeuvre permettent de générer du code de qualité dont l'efficacité est généralement comparable à du code C écrit à la main. Le compilateur propose également des options de parallélisation automatique s'appuyant entre autre sur l'interface de programmation OpenMP [8].

Le système d'architecture de FAUST facilite le déploiement des programmes et permet de générer, à partir d'un même fichier source, du code pour les principales plateformes audio : VST, iOS, Puradata, Max/MSP, Csound, SuperCollider, etc.

Nous n'avons pas la place pour décrire ici le langage FAUST de manière exhaustive. Le lecteur peut se reporter à [1] et [3] pour cela. Nous allons nous concentrer sur les aspects liés à la modularité.

2.1. Algèbre de composition

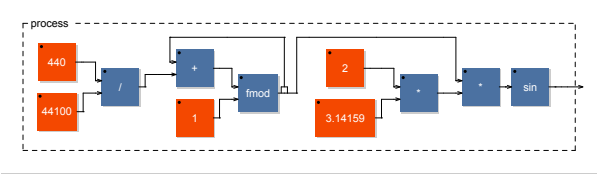
L'algèbre de composition de FAUST est basée sur cinq opérateurs qui peuvent être vus comme des extensions de l'opération $\circ$ de composition de fonctions en mathématiques. [3]

Programmer en FAUST c'est composer des processeurs de signaux pour en former de nouveaux grâce à cette algèbre. Il se trouve que tous ces opérateurs ont une traduction graphique, de sorte que l'on sait facilement produire l'équivalent, sous forme de Patch, d'un programme FAUST. Ainsi le programme FAUST figure 1a, qui implémente un sinus à 440 Hz, se représente graphiquement par le schéma figure 1b.

[3] $(f \circ g)$ en mathématiques se traduit par $(g : f)$ en Faust

```
process = 440/44100 : (+,1:fmod) ~ _
        : *(2*3.14159265359) : sin;
```

(a) Implémentation FAUST d'un LA 440 Hz



(b) Représentation graphique du LA 440 Hz

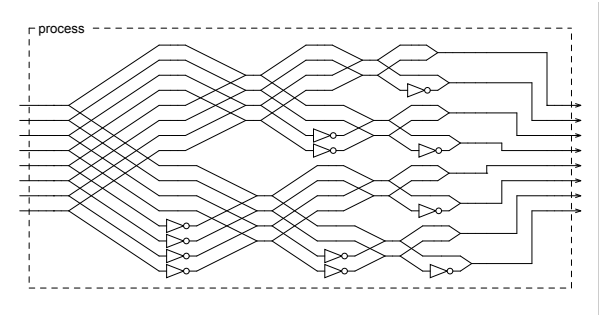
Figure 1: LA 440

```
H(1) = _;
H(n) = B(n) <: (B(n/2), B(n/2) :> H(n/2)),
       (B(n/2), I(n/2) :> H(n/2))
    with {
       B(1) = _;
       B(n) = _,B(n-1);

       I(1) = *(-1);
       I(n) = *(-1),I(n-1);
   };

process = H(8);
```

(a) FAUST Implémentation Faust d'une matrice de Hadamard $H(8)$



(b) Représentation graphique d'une matrice de Hadamard $H(8)$

Figure 2: Matrice d'Hadamard

`(A,B)`	composition parallèle
`(A:B)`	composition séquentielle
`(A<:B)`	composition split
`(A:>B)`	composition merge
`(A~B)`	composition récursive

Table 1: Les cinq opérateurs de composition de FAUST

Mais l'opération inverse, aller d'un patch à un programme FAUST équivalent, ce qui est l'objet de cet article, n'avait été jusqu'à présent que relativement peu explorée [5]. Bien entendu la situation n'est pas symétrique. Le langage textuel FAUST permet de générer algorithmiquement des patchs complexes comme on le voit avec l'exemple d'Hadamard (voir figures 2a et 2b). Il parait difficile d'inférer automatiquement, à partir du patch de la figure 2b, la construction récursive du programme FAUST figure 2a (même si visuellement pour nous cette structure est très apparente). En d'autres termes il est possible facilement d'aller d'une description textuelle en *intention* (i.e. algorithmique) à une description graphique en extension, mais pas l'inverse.

2.2. Principes de modularité

L'algèbre de FAUST permet une grande modularité des expressions. Il est par exemple facile de modifier une chaine de traitements telle que : `e:f:g:h` pour, par exemple, supprimer une étape. C'est plus compliqué avec une expression en notation traditionnelle telle que: $h(g(f(e(x))))$, ne serait-ce que parce qu'il faut intervenir à deux endroits de l'expression.

Mais qu'en est-il de la modularité des programmes FAUST eux-même ? Peut-on combiner des programmes entiers comme on le fait avec des expressions ?

Un programme FAUST est une liste de déclarations (principalement des définitions, dont la définition du mot clé `process` qui est l'équivalent de `main()` en C). On ne peut donc pas combiner deux programmes simplement en concaténant les fichiers sources, ne serait-ce parce que les redéfinitions sont interdites en FAUST. Mais comme nous allons le voir, FAUST propose des constructions qui permettent de composer des programmes. La technique de base consiste à transformer un programme complet en une simple expression FAUST.

2.2.1. *Component*

La construction `component("mon/beau/fichier.dsp")` permet de transformer un programme FAUST, repéré par son fichier source, en une expression correspondant à la définition de `process` dans le fichier en question. De plus toutes les définitions contenues dans le fichier sont isolées du reste. On peut ainsi composer séquentiellement deux programmes à partir de leurs fichiers sources de la manière suivante :

```
process = component("fichier1.dsp")
        : component("fichier2.dsp");
```

sans craindre d'interférences entre les définitions des deux fichiers.

2.2.2. *Environment*

La construction `environment{...}` permet de créer un dictionnaire de définitions, un *namespace* anonyme en quelque sorte, qui isole ces définitions du reste du programme. Pour comprendre son utilisation, supposons que l'on veuille reprendre l'exemple précédent, mais en codant tout en un seul fichier. On peut procéder comme suit :

```
process = environment{
        ...
        contenu de fichier1.dsp
        ...
    }.process : environment{
        ...
        contenu de fichier2.dsp
        ...
    }.process;
```

Dans un premier environnement on copie le contenu de fichier1, dans un deuxième environnement celui de fichier2. On accède

à une définition particulière d'un environnement, par exemple la définition de `process`, par la construction:

```
environment{...}.process
```

Il est souvent pratique de pouvoir donner un nom à un environnement. On utilise pour cela le mécanisme standard de définition comme dans l'exemple suivant:

```
e1 = environment{
    ...
    contenu de fichier1.dsp
    ...
};

e2 = environment{
    ...
    contenu de fichier2.dsp
    ...
};

process = e1.process : e2.process;
```

C'est cette approche qui va être utilisée pour combiner en un seul fichiers les codes sources de tous les modules du graphe.

2.2.3. *Stereoïsation*

L'algèbre de composition de FAUST impose un certain nombre de contraintes quand au nombre d'entrées et de sorties des expressions que l'on combine (voir table 2). Ainsi la composition séquentielle `(A:B)` impose que le nombre de sorties de `A` soit égale au nombre d'entrées de `B`.

`(A,B)`	$(\mathbb{S}^n \to \mathbb{S}^m) \to (\mathbb{S}^o \to \mathbb{S}^p) \to (\mathbb{S}^{n+o} \to \mathbb{S}^{m+p})$
`(A:B)`	$(\mathbb{S}^n \to \mathbb{S}^m) \to (\mathbb{S}^m \to \mathbb{S}^p) \to (\mathbb{S}^n \to \mathbb{S}^p)$
`(A<:B)`	$(\mathbb{S}^n \to \mathbb{S}^m) \to (\mathbb{S}^{k.m} \to \mathbb{S}^p) \to (\mathbb{S}^n \to \mathbb{S}^p)$
`(A:>B)`	$(\mathbb{S}^n \to \mathbb{S}^{k.m}) \to (\mathbb{S}^m \to \mathbb{S}^p) \to (\mathbb{S}^n \to \mathbb{S}^p)$
`(A~B)`	$(\mathbb{S}^{m+n} \to \mathbb{S}^{o+p}) \to (\mathbb{S}^o \to \mathbb{S}^m) \to (\mathbb{S}^n \to \mathbb{S}^{o+p})$

Table 2: Contraintes sur les opérations de composition

Dans le cadre de l'application qui nous intéresse ici, nous procédons à une stéréoïsation systématique des modules FAUST que nous combinons, de façon à toujours obtenir des applications légales [4].

Cette opération de stéréoïsation consiste à transformer une expressions Faust, avec un nombre quelconque d'entrées-sorties, en une expressions stéréo avec deux entrées et deux sorties. Elle s'écrit directement en FAUST sous la forme d'une série de règles de réécritures tenant compte du nombre d'entrées-sorties d'une expression.

```
stereoize(p) = S(inputs(p), outputs(p))
with {
    S(n,0) = !,! : 0,0;

    S(0,1) = !,! : p <: _,_;
    S(0,2) = !,! : p;
    S(0,n) = !,! : p,p :> _,_;

    S(1,1) = p,p;
    S(1,n) = p,p :> _,_;

    S(2,1) = p <: _,_;
    S(2,2) = p;

    S(n,m) = _,_ <: p,p :> _,_;
};
```

[4] Cette simplification est admissible dans notre cas car nous cherchons à proposer une forme de programmation Faust spécialement adaptée aux enfants.

Dans cette définition `inputs` et `outputs` sont deux primitives du langage qui donnent respectivement le nombre d'entrées et de sorties d'une expression et qui servent à choisir quelle règle appliquer. La primitive `!` permet de créer une entrée factice. La primitive `_` représente la fonction identité (en fait un simple fil). Les opérateurs `,`, `:`, `<:` et `:>` représentent des opérations binaires de composition d'expressions.

Ainsi si `p` a deux entrées et deux sorties, il n'y a rien à faire et `stereoize(p)` donne `p`. Si par contre `p` n'a qu'une entrée et qu'une sortie (c'est-à-dire si `p` est mono), alors il suffit de mettre `p` en parallèle avec lui même, et dans ce cas `stereoize(p)` donne `p,p`, et ainsi de suite.

3. L'ALGORITHME DE FAUST ÉQUIVALENT

Dans cette partie, nous présentons l'algorithme qui permet de passer d'un graphe d'applications à une unique expression FAUST équivalente ($G \to$ codeFaust). Un exemple de graphe est présenté Figure 3.

Pour réaliser cette opération, on définit les deux étapes de l'algorithme:

- la fonction $\mathcal{D}$, de "Dérecursivisation" qui parcourt le graphe et détecte les boucles,
- la fonction $\mathcal{F}$, de calcul du "FAUST Equivalent".

Ce modèle prend en compte 3 types de compositions: série, parallèle et la récursion.

On peut retrouver la plupart des règles de notation dans la référence [2].

3.1. Données

On considère un graphe orienté, G, pouvant contenir des boucles.

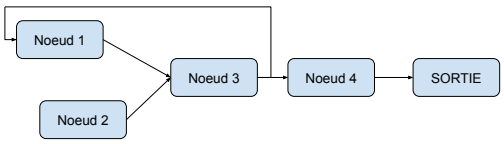


Figure 3: Exemple de graphe

On définit le graphe G = $\langle$ N, I $\rangle$ avec:
N : l'ensemble des noeuds composant le graphe
I : la fonction qui rend la liste des sources d'un noeud avec $I : N \to [N]$

D'autre part, on peut distinguer un noeud particulier : la Sortie Audio, d avec $d \in$ N

On dispose d'une fonction C, qui pour un noeud rend le code FAUST qui lui est associé. $C : N \to$ codeFAUST

3.2. Conditions

Tous les objets FAUST manipulés doivent former un graphe simple (c'est pourquoi dans notre cas tous les objets et toutes les connexions sont en stéréo). D'autre part, la sortie audio ne peut pas être l'entrée d'un autre noeud ($\forall\ e \in$ N $\ then\ \ d \notin I(e)$).

3.3. Fonctions et type de base

Pour décrire l'algorithme, nous définissons les types et fonctions de base suivantes :

- N : ensemble des noeuds
  avec les variables d, e, f $\in N$
- $[N]$: ensemble des listes de noeuds
  avec les variables O, V $\in [N]$ de la forme $V = [v_1, v_2, ..., v_k]$ où $v_i \in N$
- $[[N]]$: ensemble des listes de listes de noeuds
  avec les variables S $\in [[N]]$ de la forme $S = [s_1, s_2, ..., s_k]$ où $s_i \in [N]$
- N^* : ensemble des séquences de noeuds ordonnés
  avec les variables p $\in N^*$ de la forme $p = p_1p_2...p_k$ où $p_i \in N$
- $[N^*]$: ensemble des listes de séquences noeuds ordonnés
  avec les variables B $\in [N^*]$ de la forme $B = [b_1, b_2, ..., b_k]$ où $b_i \in N^*$
- ϵ : séquence vide
- [] : liste vide

Les fonctions de base sur ces types sont :

- Concaténer
  - un noeud, e et une séquence, p : *ep*
  - deux séquences p, q : *pq*
- Ajouter une séquence dans une liste
  - une séquence, p dans une liste B : *p.B*
- Décomposer
  - une séquence $p = f.p^{'}$
  - une liste $S = O.S^{'}$
- Les opérateurs d'appartenances ($\in$), d'intersection ($\cap$) et d'union ($\cup$) seront utilisés.
- Préfixer
  - $\mathcal{P}(p) = [p_1, p_1p_2, p_1p_2p_3, ..., p_1p_2...p_{n-1}]$
  - $\mathcal{P}(B) = U_{p_i \in B}\ \mathcal{P}(p_i)$

  Seuls les préfixes propres sont pris en compte, c'est-à-dire les préfixes de p différents de ϵ et de p.

3.4. Fonctionnement général

L'objectif est de créer une expression FAUST équivalente à partir d'un graphe, G : $G \rightarrow$ codeFaust

Pour réaliser cette opération, on définit 2 fonctions

- $\mathcal{D}$ ou "Dérecursivisation" parcourt le graphe grâce à un algorithme de "Deep-First-Search" et détecte les boucles. Elle rend une liste de boucles, c'est-à-dire des séquences de noeuds du type 'ABA'.

$$\mathcal{D} : N^* \rightarrow [[N]] \rightarrow [N] \rightarrow [N^*] \rightarrow [N^*]$$

- $\mathcal{F}$ ou "FAUST Equivalent" consiste à créer l'expression FAUST équivalente. Cette fonction s'appuie sur le résultat de $\mathcal{D}$ pour former ses règles de réécriture.

$$\mathcal{F} : N \rightarrow N^* \rightarrow codeFaust$$

3.5. Etape 1 - Dérecursivisation - $\mathcal{D}(p, S, V, B)$

La première étape de l'algorithme doit permettre de détecter les différentes boucles présentes dans le graphe analysé.

On introduit les éléments suivants
$\mathcal{D}(p, S, V, B)$

- p : N^* la séquence des nœuds qui ont été parcourus depuis le noeud de départ d jusqu'au noeud courant e,
- S : $[[N]]$: liste des sources à parcourir pour chaque nœud du parcours.
- V : $[N]$ la liste des nœuds visités. Un nœud est visité lorsque toutes ses sources ont été parcourues,
- B : $[N^*]$ la liste des boucles qui ont été trouvées dans le graphe.
  Une boucle est une séquence de nœuds, qui contient 1 même nœud au début et à la fin. Exemple 'ABA'

Le principe est simple. Au départ de la sortie audio, le graphe est parcouru nœud après nœud. Lorsque toutes les entrées d'un nœud ont été parcourues, un nœud est considéré comme visité, ce qui permet de ne pas revisiter une branche déjà parcourue. D'autre part, une boucle est détectée lorsque le parcours du nœud courant contient ce même nœud. Le parcours représente la séquence des nœud parcourus pour arriver au nœud courant depuis le nœud de départ.

Le nœud de départ de l'algorithme étant la sortie audio, d, on a l'initialisation suivante $\mathcal{D}(d, [I(d)], [], [])$, avec:

- d : N : le nœud de départ de l'algorithme. Il correspond à la sortie audio
- e : N : représente le nœud courant
- I : $N \rightarrow [N]$: la fonction qui pour un nœud rend la liste de ses sources.

$$\frac{e \notin V \qquad p = p^{'}ep^{''}}{\langle\ e.p \mid O.S \mid V \mid B\ \rangle \rightarrow \langle\ p \mid S \mid V \mid ep^{'}e.B\ \rangle} \tag{1}$$

e fait partie de son parcours, p, et forme donc une boucle
$\rightarrow$ on revient en arrière dans le parcours et la boucle ($ep^{'}e$) est repérée

$$\frac{e \notin V \qquad p \neq p^{'}ep^{''}}{\langle\ e.p \mid [f.O].S \mid V \mid B\ \rangle \rightarrow \langle\ f.e.p \mid I(f).O.S \mid V \mid B\ \rangle} \tag{2}$$

e n'a pas encore été visité, e ne fait pas partie de son parcours, p, et a des nœuds d'entrée, $f.O$, non parcourus
$\rightarrow$ on parcourt f, le premier nœud.

$$\frac{e \notin V \qquad p \neq p^{'}ep^{''}}{\langle\ e.p \mid [].S \mid V \mid B\ \rangle \rightarrow \langle\ p \mid S \mid e.V \mid B\ \rangle} \tag{3}$$

e n'est pas encore repéré comme visité et ses nœuds d'entrée ont tous été parcourus ($O = []$)
$\rightarrow$ on revient en arrière dans le parcours et e est repéré comme visité

$$\frac{e \in V}{\langle\, e.p \mid O.S \mid V \mid B \,\rangle \rightarrow \langle\, p \mid S \mid V \mid B \,\rangle} \tag{4}$$

e a déjà été visité
$\rightarrow$ on revient en arrière dans le parcours

$$\frac{e \in V}{\langle\, e \mid [[]] \mid V \mid B \,\rangle \rightarrow B} \tag{5}$$

tous les noeuds sont visités. C'est la condition de fin.
$\rightarrow$ on rend B

3.6. Etape 2 - Calcul de l'expression Faust équivalente - $\mathcal{F}(e, p)$

Une fois les boucles détectées dans le graphe, il est possible de recréer l'expression FAUST équivalente. De la même manière qu'à la première étape, le graphe est parcouru à partir du nœud particulier qu'est la sortie audio. Prenant en compte l'information de récursion, les différents opérateurs FAUST de composition sont utilisés pour composer les codes FAUST des différents nœuds: la mise en parallèle `,`, le merge `:>`, la récursion `~`.

On retrouve certains paramètres:
$\mathcal{F}(e, p)$

- $e : N$ le noeud courant,
- $p : N^*$ la séquence des noeuds qui ont été parcourus pour arriver à e depuis d.

De plus, certaines données sont nécessaires
Initialisation : $\mathcal{F}(d, \epsilon)$

- $d : N$ le noeud de départ de l'algorithme correspondant à la sortie audio,
- $B = \mathcal{D}(d, [I(d)], [], [])$ la liste des boucles détectées dans le graphe,
- $C : N \rightarrow$ `code` la fonction qui rend le code FAUST associé à un noeud,
- $I : N \rightarrow [N]$ la fonction qui pour un noeud rend la liste de ses sources.

$$\frac{I(e) = []}{\mathcal{F}(e, p) = C(e)} \tag{6}$$

e n'a pas de de noeuds connectés en entrée
$\rightarrow$ son code FAUST équivalent est donc le code FAUST de ce noeud.

$$\frac{e \notin \mathcal{P}(R) \qquad I(e) = [e_1, ..., e_k] \qquad \mathcal{F}(e_i, ep) = f_i}{\mathcal{F}(e, p) = (f_1, ..., f_k) \;\texttt{:>}\; C(e)} \tag{7}$$

e n'est pas un départ ni une arrivée de récursion
$\rightarrow$ son code FAUST équivalent est donc la mise en parallèle des codes FAUST équivalent de ses entrées mergés dans le code FAUST de ce noeud.

$$\frac{e \in \mathcal{P}(R) \quad R \cap \mathcal{P}(ep) = [] \quad I(e) = [e_1, ..., e_k] \quad \mathcal{F}(e_i, ep) = f_i}{\mathcal{F}(e, p) = ((f_1, f_2, ..., f_k) \;\texttt{:>}\; C(e)) \sim _} \tag{8}$$

e est un départ de récursion
$\rightarrow$ son code FAUST équivalent est donc la mise en parallèle des codes FAUST équivalent de ses entrées mergés dans le code FAUST de ce noeud avec une récursion.

$$\frac{e \in \mathcal{P}(R) \quad R \cap \mathcal{P}(ep) \neq []}{\mathcal{F}(e, p) = _} \tag{9}$$

e est une arrivée de récursion
$\rightarrow$ son code FAUST équivalent est donc _.

3.7. Exemple

Pour illustrer cet algorithme, un exemple est présenté en figure 3. La figure 4 présente les étapes de résolution de la dérecursivisation, $\mathcal{D}$.

p	S	V	B	Règle
S	[B,W]	[]	[]	1
BS	[A,J],[W]	[]	[]	1
ABS	[A,B,M],[J],[W]	[]	[]	1
AABS	[A,B,M],[B,M],[J],[W]	[]	[]	2
ABS	[B,M],[J],[W]	[]	[AA]	1
BABS	[A,J],[M],[J],[W]	[]	[AA]	2
ABS	[M],[J],[W]	[]	[BAB],[AA]	1
MABS	[K,M],[],[J],[W]	[]	[BAB],[AA]	1
KMABS	[],[M],[],[J],[W]	[]	[BAB],[AA]	3
MABS	[M],[],[J],[W]	[K]	[BAB],[AA]	1
MMABS	[K,M],[],[],[J],[W]	[K]	[BAB],[AA]	2
MABS	[],[],[J],[W]	[K]	[MM],[BAB],[AA]	3
ABS	[],[J],[W]	[M,K]	[MM],[BAB],[AA]	3
BS	[J],[W]	[A,M,K]	[MM],[BAB],[AA]	1
JBS	[],[],[W]	[A,M,K]	[MM],[BAB],[AA]	3
BS	[],[W]	[J,A,M,K]	[MM],[BAB],[AA]	3
S	[W]	[B,J,A,M,K]	[MM],[BAB],[AA]	1
WS	[A],[]	[B,J,A,M,K]	[MM],[BAB],[AA]	1
AWS	[M,B,A],[],[]	[B,J,A,M,K]	[MM],[BAB],[AA]	4
WS	[],[]	[B,J,A,M,K]	[MM],[BAB],[AA]	3
S	[[]]	[W,B,J,A,M,K]	[MM],[BAB],[AA]	5

Figure 4: Résolution de $\mathcal{D}$ pas à pas

Connaissant la liste des boucles grâce à l'algorithme de dérecursivisation, $\mathcal{D}$. On peut maintenant calculer l'expression FAUST équivalente grâce à l'algorithme, $\mathcal{F}$. Pour rappel, la fonction $\mathcal{C}$ rend le code FAUST d'un noeud.

$$\mathcal{F}(S, \varnothing) = (\mathcal{F}(B, S), \mathcal{F}(W, S)) \;\texttt{:>}\; \mathcal{C}(S)$$

$$= (((\mathcal{F}(J, BS), \mathcal{F}(A, BS)) \;\texttt{:>}\; \mathcal{C}(B) \sim _),$$
$$((\mathcal{F}(A, WS)) \;\texttt{:>}\; \mathcal{C}(W))) \;\texttt{:>}\; \mathcal{C}(S)$$

$$= (((\mathcal{C}(J), ((\mathcal{F}(A, ABS),$$
$$\mathcal{F}(B, ABS, \mathcal{F}(M, ABS) \;\texttt{:>}\; \mathcal{C}(A) \sim _)) \;\texttt{:>}\; \mathcal{C}(B) \sim _),$$
$$((((\mathcal{F}(A, AWS), \mathcal{F}(B, AWS,$$
$$\mathcal{F}(M, AWS) \;\texttt{:>}\; \mathcal{C}(A) \sim _)) \;\texttt{:>}\; \mathcal{C}(W))) \;\texttt{:>}\; \mathcal{C}(S)$$

$$= (((\mathcal{C}(J), ((_, _, ((\mathcal{F}(M, MABS,$$
$$\mathcal{F}(K, MABS) \;\texttt{:>}\; \mathcal{C}(M) \sim _)) \;\texttt{:>}\; \mathcal{C}(A) \sim _)$$
$$) \;\texttt{:>}\; \mathcal{C}(B) \sim _), ((((_, ((\mathcal{F}(J, BAWS,$$
$$\mathcal{F}(A, BAWS) \;\texttt{:>}\; \mathcal{C}(B) \sim _), ((\mathcal{F}(M, MAWS,$$
$$\mathcal{F}(K, MAWS) \;\texttt{:>}\; \mathcal{C}(M) \sim _)) \;\texttt{:>}\; \mathcal{C}(A) \sim _)$$
$$) \;\texttt{:>}\; \mathcal{C}(W))) \;\texttt{:>}\; \mathcal{C}(S)$$

$$= (((\mathcal{C}(J), ((_, _, ((_, \mathcal{C}(K)]) \;\texttt{:>}\; \mathcal{C}(M) \sim _)) \;\texttt{:>}\; \mathcal{C}(A) \sim _)$$
$$) \;\texttt{:>}\; \mathcal{C}(B) \sim _), ((((_, ((\mathcal{C}(J),$$
$$((\mathcal{F}(A, ABAWS), \mathcal{F}(B, ABAWS,$$
$$\mathcal{F}(M, ABAWS) \;\texttt{:>}\; \mathcal{C}(A) \sim _)) \;\texttt{:>}\; \mathcal{C}(B) \sim _),$$
$$((_, \mathcal{C}(K)) \;\texttt{:>}\; \mathcal{C}(M) \sim _)) \;\texttt{:>}\; \mathcal{C}(A) \sim _)$$
$$) \;\texttt{:>}\; \mathcal{C}(W))) \;\texttt{:>}\; \mathcal{C}(S)$$

$$= (((\mathcal{C}(J), ((_, _, ((_, \mathcal{C}(K)]) \;\texttt{:>}\; \mathcal{C}(M) \sim _)) \;\texttt{:>}\; \mathcal{C}(A) \sim _)$$
$$) \;\texttt{:>}\; \mathcal{C}(B) \sim _), ((((_, ((\mathcal{C}(J), ((_, _, ((($$
$$\mathcal{F}(M, MABAWS, F\mathcal{F}(K, MABAWS$$

$$
\begin{aligned}
&) :> \mathcal{C}(M) \sim _)) :> \mathcal{C}(A) \sim _)) :> \mathcal{C}(B) \sim _), ((_,\\
&\mathcal{C}(K)) :> \mathcal{C}(M) \sim _)) :> \mathcal{C}(A) \sim _)\\
&) :> \mathcal{C}(W))) :> \mathcal{C}(S)
\end{aligned}
$$

$$
\begin{aligned}
= ((&(\mathcal{C}(J), ((_,_, ((_, \mathcal{C}(K)]) :> \mathcal{C}(M) \sim _)\\
&) :> \mathcal{C}(A) \sim _)) :> \mathcal{C}(B) \sim _), ((((_, ((\mathcal{C}(J), ((_,_, ((_,\\
&\mathcal{C}(K)) :> \mathcal{C}(M) \sim _)) :> \mathcal{C}(A) \sim _)) :> \mathcal{C}(B) \sim _), ((_,\\
&\mathcal{C}(K)) :> \mathcal{C}(M) \sim _)) :> \mathcal{C}(A) \sim _)\\
&) :> \mathcal{C}(W))) :> \mathcal{C}(S)
\end{aligned}
$$

En simplifiant l'écriture pour plus de lisibilité, on a:

$X2 = (_, \mathcal{C}(K)) :> \mathcal{C}(M) \sim _;$
$X1 = \mathcal{C}(J), ((_,_, X2) :> \mathcal{C}(A) \sim _) :> \mathcal{C}(B) \sim _;$
$\text{process} = X1, (((_, X1, X2 :> \mathcal{C}(A) \sim _)) :> \mathcal{C}(W)) :> \mathcal{C}(S);$

4. UN EXEMPLE D'IMPLÉMENTATION : LE FAUSTPLAYGROUND

Le FaustPlayground est une plateforme Web qui permet de créer des graphes d'applications FAUST, voir figure 5.

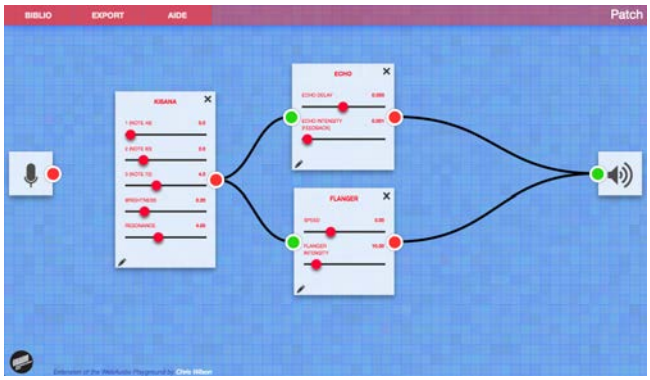


Figure 5: Exemple d'utilisation de Faust Playground

Il est ainsi possible de programmer des applications FAUST de manière simplifiée, à partir de modules de plus haut niveau. Le programme FAUST ainsi créé peut ensuite être exporté sous forme d'une application (d'une plateforme/architecture disponible dans le service de compilation FAUST). Avant d'être envoyé au serveur [5] , le code FAUST équivalent au graphe créé doit être calculé dans la page (voir figure 6).

4.1. Mise en oeuvre de l'algorithme en Javascript

La mise en œuvre suit la structure proposée en Section 3. Deux fonctions principales effectuent les étapes de l'algorithme:

- function createTree(Module, Parent) : construit un arbre représentant le graphe, en ayant mis à plat et marqué les branches récursives. Cette fonction est récursive et prend en entrée un Module (classe spécifique qui contient les informations du nœud : nom, code faust, entrées, etc).

- function getFaustEquivalent(Tree, PatchName) : utilise l'arbre pour calculer l'expression FAUST équivalente et donne le nom du patch au programme créé.

[5] Le serveur de compilation FAUST : faustservice.grame.fr, permet à une application (via une API RESTful) ou à un utilisateur (via la page web) d'avoir accès à un compilateur FAUST sans avoir à l'installer sur sa machine. Par de simples requêtes http, il peut récupérer la liste des plateformes supportées et envoyer son code pour obtenir une application compilée.

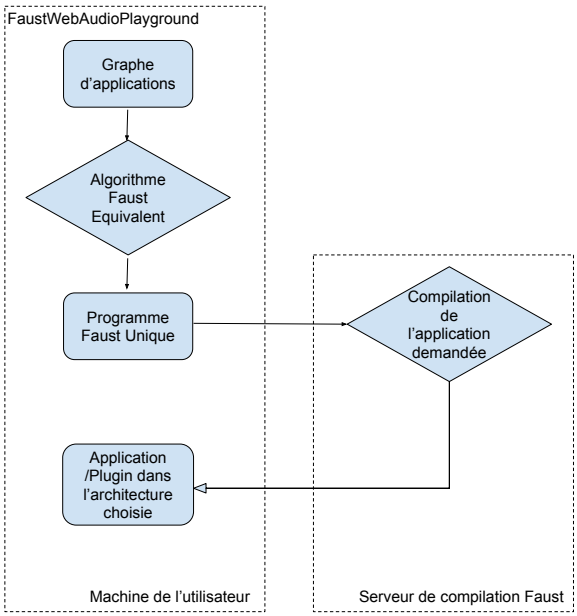


Figure 6: Création de l'application FAUST unique à partir d'un graphe d'applications

4.2. Gains de fonctionnalités

Cet algorithme ouvre de nouveaux horizons concernant la programmation FAUST. Dans le contexte de cette plateforme web, il est possible de récupérer et de tester des programmes FAUST publiés sur le web (puisque le compilateur FAUST permet de compiler des urls), de les combiner avec d'autres pour former son propre DSP. Cette composition à l'échelle 'macroscopique' est très simple à utiliser et permet visuellement de bien voir ce qu'il se passe, donnant une porte d'entrée à la programmation FAUST à de nouveaux utilisateurs.

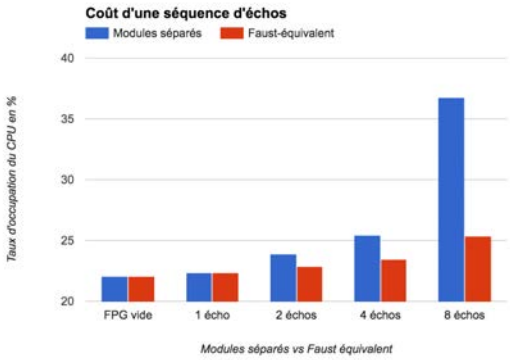


Figure 7: Performances comparées entre une séquences de 1 à 8 échos sous la forme de modules séparés et sous la forme d'un module FAUST équivalent

Lorsque l'on est satisfait de sa composition, la possibilité de recréer un unique DSP permet d'utiliser moins de ressources puisque le compilateur fait des optimisations et d'autre part, il est moins coûteux de faire tourner un seul DSP dans la WebAudio API qu'un graphe entier comme le montre le comparatif figure 7. Enfin, à partir d'une expression FAUST, il est possible d'exporter son programme pour une architecture et une plate-

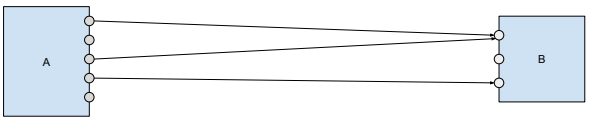
forme supportée par le compilateur FAUST. Une description plus approfondie de la plateforme et de ses performances est disponible dans [7].

5. CONCLUSION

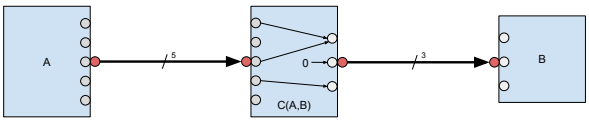
L'algorithme que nous avons présenté permet de passer d'un graphe, dont les noeuds sont des programmes FAUST, à un programme FAUST équivalent. Le programme ainsi obtenu peut être compilé, et donc bénéficier de toutes les optimisations du compilateur FAUST, mais il peut également être exporté vers les différentes plateformes supportées par FAUST (vst, max, supercollider, csound, etc.)

Dans le cadre de l'application FaustPlayground pour laquelle nous avons développé cette technique, nous nous sommes restreints à un graphe simple orienté: il ne peut y avoir au plus qu'une connexion entre deux nœuds. Pour cela nous avons contraints les nœuds à n'avoir qu'une entrée et une sortie stéréo (via une opération de stereoization)

Ce modèle est évidemment trop restrictif si l'on voulait convertir un multigraphe d'applications JACK, ou un patch Max ou Puredata en FAUST. En effet les nœuds d'un patch peuvent avoir de multiples ports d'entrées et de sorties avec des connexions arbitraires entre ces ports comme illustré sur la figure 8a:



(a) Connexions multiples entre deux noeuds d'un patch



(b) Connexion unique en utilisant un noeud intermédiaire

Figure 8: passage de connexions de type multigraphe à une connexion simple via un noeud intermédiaire

Néanmoins il est facile de ramener un patch à un graphe simple en introduisant des nœuds intermédiaires de routage comme illustré figure 8b. Le code du nœud intermédiaire peut être généré de manière automatique en FAUST en fonction des connexions entre A et B. Voici le code FAUST correspondant au noeud intermédiaire de la figure 8:

```
process = (_,!,_ :> _), 0, _, !;
```

Une telle extension est en cours d'étude. Elle permettra de traduire en FAUST, et donc de compiler, des patchs Max/MSP ou Puredata simples.

Remerciements
Le travail présenté ici a été développé dans le cadre du projet FEEVER [ANR-13-BS02-0008] soutenu par l'Agence Nationale pour la Recherche.

6. REFERENCES

[1] Y. Orlarey, D. Fober, and S. Letz, "Syntactical and semantical aspects of Faust", *Soft Computing*, 8(9), 2004, pp. 623–632.

[2] C. Gunter, "Semantics of Programming Languages", *The MIT Press*, 1992.

[3] Y. Orlarey, S. Letz and D. Fober, "FAUST: an Efficient Functional Approach to DSP Programming" *Editions DELATOUR FRANCE*, 2009

[4] J. Hughes, "Why Functional Programming Matters" *Editions D. Turner*, 1990

[5] Abdullah Onur Demir, "Mephisto: A source to source transpiler from pure data to Faust" *Middle East Technical University*, 2015

[6] Shimon Even, "Graph Algorithms", *Computer Science Press*, 1979

[7] S. Denoux, S. Letz, Y. Orlarey and D. Fober, "Composing a web of audio applications" *Web Audio Conference Proceedings*, 2015

[8] http://www.openmp.org