Numéros / n° 4 - automne 2014

« Des programmes Faust dans Csound »

Laurent Pottier

1. Introduction

Csound (Boulanger, 2000) est un langage de type Music-N qui a été publié la première fois en 1986 par portage en langage C du programme Music 11, développé à l'origine pour les ordinateurs DEC PDP-11 au MIT dans les années 1970. Depuis sa sortie, Csound a subi un certain nombre de modifications, et en 2006, Csound 5, un système complètement remodelé, 5, a été lancé. Ce système offrait un certain nombre de nouvelles possibilités, car il a été conçu en tant que bibliothèque pouvant être incorporé dans différents environnements. Il offrait notamment la possibilité d'intégrer comme *opcodes* des *plugins*, étendant le langage sans nécessiter de recompiler l'ensemble du code du programme. En 2013, une nouvelle version du système a été réalisée, et une nouvelle version, majeure, Csound 6 (Cabrera *et al.*, 2013), a été lancée, avec des améliorations et des ajouts substantiels. C'est le système actuel, sur lequel porte cet article.

Faust (Orlarey, 2009) est un langage fonctionnel conçu pour traduire des blocs diagramme de traitement de signal en code source en C++ ou en Javascript, ou en code binaire LLVM. Il permet de construire des *plugins* et de les traduire en code C++ qui peuvent ensuite être compilés en bibliothèques dynamiques qui peuvent être chargées dans Csound comme de nouveaux générateurs unitaires (*opcodes*). Cela est réalisé par le compilateur Faust, qui peut être exécuté à partir d'une ligne de commande, de l'application (IDE) Faustworks ou en ligne, via une interface basée sur le Web.

Par ailleurs, une nouvelle version du système, Faust 2, a récemment été développée dans laquelle le compilateur Faust est maintenant disponible en tant que bibliothèque (libfaust) qui peut être intégrée dans un autre programme. libfaust peut fournir les fonctionnalités de Faust de façon dynamique, de sorte que les programmes Faust peuvent être compilés à la volée en code binaire LLVM qui peut être exécuté directement. Cela donne la possibilité de court-circuiter le processus de développement, de sorte qu'un *opcode* de type *plugin* n'est plus nécessaire comme intermédiaire entre le programme Faust original et le générateur unitaire fonctionnant dans Csound.

2. Utilisation de la bibliothèque Faust

Il y a trois étapes fondamentales dans la compilation dynamique de Faust et dans son exécution :

1. Compilation du code Faust en une *factory* DSP.
2. Instanciation d'un objet DSP à partir de la *factory* DSP.
3. Exécution du processus DSP.

La première étape utilise une des deux fonctions suivantes :

```
llvm_dsp_factory*
```

createDSPFactoryFromFile(const std::string& filename, int argc,

const char *argv[],

const std::string& target,

std::string& error_msg, int opt_level = 3);

.

llvm_dsp_factory*

createDSPFactoryFromString(const std::string& name_app, const

std::string& dsp_content,

int argc, const char *argv[], const

std::string& target,

std::string& error_msg, int opt_level = 3);

Celles-ci traduisent le code Faust sous une forme de code binaire qui sera stocké en mémoire, prêt à être exécuté par un programme. Pour cela, il faut instancier un objet DSP à partir de la *factory* :

```
llvm_dsp* createDSPInstance(llvm_dsp_factory* factory);
```

La class DSP a un certain nombre de méthodes publiques qui peuvent être utilisées pour la manipuler :

```
class llvm_dsp :public dsp {

    public:
     virtual int getNumInputs();
     virtual int getNumOutputs();
     virtual void init(int samplingFreq);
     virtual void buildUserInterface(UI* inter);
     virtual void compute(int count,
          FAUSTFLOAT**input,FAUSTFLOAT**output);
};
```

Enfin, une fois l'objet DSP créé, on peut l'exécuter en invoquant sa méthode `compute()`. Cela va prendre un bloc d'échantillons en entrée et produire un bloc d'échantillons en sortie. Pour contrôler la synthèse/traitement, on peut également définir des contrôles via un objet de l'interface utilisateur que l'on aura spécialement dessiné pour ça. Celui-ci peut être ajouté aux fonctionnalités de l'objet DSP par l'intermédiaire de la méthode `buildUserInterface()`. Dans le cas de Csound, c'est utile pour pouvoir transmettre des données de commande de Csound vers le programme Faust.

3. Les *opcodes*

Les *opcodes* Faust pour Csound (voir *The Csound Reference Manual*) sont construits en utilisant les fonctionnalités ci-dessus. Afin de faciliter leur utilisation, le processus est divisé en deux étapes : la

compilation et l'exécution.

Dans Csound, l'initialisation et l'exécution DSP sont séparées en deux phases distinctes. Au cours de la première, tous les générateurs unitaires qui ont des tâches d'initialisation à exécuter sont lancés (une fois), et ensuite on passe à la phase suivante, qui est une boucle qui appelle les fonctions qui exécutent chaque *opcode* de traitement du signal. Certains *opcodes* n'ont pas ces fonctions, car ils ne sont pas des générateurs ou des transformateurs de signaux (ou les deux). Ces *opcodes* ne fonctionnent que lors de l'initialisation.

L'étape de compilation dans Faust est évidemment une histoire de temps initial. Elle ne concerne pas de tâches d'exécution (j'entends par là le traitement du signal). Il est logique d'en faire un *opcode* fonctionnant uniquement au temps initial (init-time). En le séparant d'un *opcode* DSP d'exécution, cela permet ainsi de créer autant d'objets qu'on veut à partir d'une même architecture.

L'aspect exécution de l'intégration de Faust, est subdivisée en deux éléments : le traitement du signal et son contrôle, et correspond donc à deux *opcodes* différents. La partie traitement du signal consiste uniquement à récupérer une entrée audio (si elle existe), de faire quelque chose et de le transmettre à la sortie. Pour réaliser le contrôle des paramètres, on utilise un autre *opcode* qui peut régler un élément d'interface utilisateur donné défini dans la partie DSP.

Enfin, il y a également un *opcode* qui a été conçu pour des instances DSP uniques, qui intègre une phase de compilation au temps initial et un traitement du signal au temps de l'exécution. Cet *opcode* peut être utilisé pour les effets « *on-off* » qui ne sont pas conçus pour être exécutés dans plusieurs instances.

3. 1. Faustcompile

L'*opcode* `faustcompile` appelle le compilateur temps réel pour produire un processus DSP instanciable à partir d'un programme Faust. Il va compiler un programme Faust à partir d'une chaîne, contrôlée par divers arguments. Les chaînes multi-lignes sont acceptées, en utilisant {{}} pour encadrer la chaîne.

3. 1. 1. Syntaxe

ihandle faustcompile Scode, Sargs

`Scode`?une chaîne (entre guillemets ou encadrée par {{ }}) contenant un programme Faust

`Sargs`?une chaîne (entre guillemets ou encadrée par {{ }}) contenant les arguments du compilateur Faust

Exemple :

ihandle faustcompile "process=+;", "-vec -lv 1"}

3. 2. Faustaudio

L'*opcode* `faustaudio` instantie et exécute un programme Faust compilé. Il fonctionne avec un programme compilé avec `faustcompile`.

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

`ifac`? un lien vers un programme Faust compilé, produit par faustcompile

`ihandle`? un lien vers une instance DSP Faust qui peut être utilisé pour accéder à ses contrôleurs via faustctl

`ain1,...`? signaux d'entrée

`a1,...`? signaux de sortie

Exemple :

```
ifac faustcompile "process=+;", "-vec -lv 1"

idsp,a1 faustaudio ifac,ain1,ain2
```

3. 3. Faustctl

L'*opcode* `faustctl` permet de régler les commandes de l'interface utilisateur dans une instance DSP Faust. Il va ajouter un contrôleur donné au programme Faust en cours.

3. 3. 1. Syntaxe

`faustctl idsp,Scontrol,kval`

`Scontrol` ? une chaîne contenant le nom du contrôleur

`dsp` ? un lien vers une instance DSP Faust

`kval` ? valeur qui va être affectée au contrôleur

Exemple :

```
idsp,a1 faustgen {{

gain =hslider("vol",1,0,1,0.01);

process=(_ * gain);

}}, ain1

faustctl idsp,"vol", 0.5
```

3. 4. Faustgen

L'*opcode* `faustgen` compile, instancie et exécute un programme Faust compilé. Il va invoquer le

compilateur temps réel à l'initialisation (i-time) et instancier le programme DSP. Pendant la production du son (perf-time), il va exécuter le code Faust compilé.

3. 4. 1. Syntaxe

```
ihandle,a1[,a2,...]faustgen SCode[,ain1,...]
```

`Scode` ? une chaîne contenant un programme Faust

`ihandle` ? un lien vers l'instance DSP Faust, qui peut être utilisé pour accéder à ses contrôles avec faustctrl

`ain1,...` ? signaux en entrée
`a1,...` ? signaux en sortie

Exemple :

```
idsp,a1 faustgen "process=+;",ain1,ain2
```

4. Applications

Un certain nombre d'applications peuvent être envisagées pour Csound (et ses interfaces) avec les nouveaux *opcodes* Faust :

- Synthèse : Csound peut fonctionner comme un synthétiseur logiciel autonome, contrôlé par des messages MIDI ou par des commandes OSC. Il peut être utilisé par séquenceur logiciel MIDI comme un synthétiseur système, et les programmes Faust peuvent augmenter ces possibilités. L'interface CLI classique, ou tout autre interface peuvent être utilisés pour cela.
- *Plugins* de type « instruments » : l'environnement Cabbage peut être utilisé pour créer des *plugins* VSTi à partir de codes Csound. Ceux-ci peuvent ensuite être chargés dans n'importe quel logiciel hôte VST (comme des séquenceurs, des éditeurs de partitions etc.). Les *plugins* peuvent être créés à partir du code de Csound standard, sans avoir besoin d'extensions particulières.
- Processeur audio : Csound peut être utilisé en tant que processeur audio temps réel ou temps différé. Pour le fonctionnement en temps réel, Cabbage et CsLadspa peuvent être utilisés pour créer des *plugins* qui peuvent être chargés dans des logiciels hôtes VST ou LADSPA. La puissance de Faust peut être utilisée pour enrichir Csound avec des algorithmes de traitement personnalisés performants.
- Logiciels sur mesure: Csound peut également être utilisé pour créer des applications logicielles sur mesure, à la fois pour des systèmes d'exploitation de bureau ou mobiles (Android, iOS). Elles peuvent programmées à différents niveaux, du niveau basique, à haut niveau, par le biais de Cabbage et csoundo, à un niveau intermédiaire, en utilisant Python, Clojure ou Java, jusqu'au plus bas niveau avec les langages C et C++.

5. Conclusions

Dans ce texte, nous avons montré qu'il était possible d'intégrer des programmes Faust directement dans un orchestre de Csound. Nous avons présenté les détails des technologies et de ses mécanismes, et nous avons indiqué comment la bibliothèque Faust peut être utilisée grâce à un ensemble d'*opcodes* fournis dans une bibliothèque particulière de Csound. Chaque *opcode* a été présenté séparément et sa syntaxe a été présentée. Pour conclure, un certain nombre d'applications de ces technologies ont été suggérées.

Pour citer ce document:
Laurent Pottier, « Des programmes Faust dans Csound », *RFIM* [En ligne], Numéros, n° 4 - automne 2014, Mis à jour le 14/10/2014
URL: http://revues.mshparisnord.org/rfim/index.php?id=338