Numéros / n° 6 - Techniques et méthodes innovantes pour l'enseignement de la musique et du traitement de signal

Â«Â Faust dans une salle de classe : démonstration, live, coding (programmation en direct) et implémentations de référenceÂ Â»

Julius Smith

Résumé

Ce texte présente mon utilisation du langage Faust pour l'enseignement. J?utilise Faust pour des démonstrations de traitement du signal audio. Je trouve que le moyen le plus efficace d?apprendre la programmation est de « coder en direct » en classe avec l'aide des étudiants. Un langage de haut niveau comme Faust est excellent pour cela, en raison de la quantité réduite et pourtant très lisible du code nécessaire pour avoir des résultats efficaces. J?utilise les « salles de classe inversées » avec des cours enregistrés à l?avance, ce qui laisse plus de temps pour des activités interactives en classe. Une troisième utilisation de Faust est la présentation d?implémentations de référence avec les différentes bibliothèques de Faust. Ces ressources peuvent faire gagner du temps aux étudiants lorsqu'ils doivent mettre en ?uvre quelque chose qu'ils connaissent mais qu'ils n'ont pas encore codé. Le fait que Faust compile en C++ et soit exportable immédiatement vers une grande variété de formats de plugin standard le rend très utile comme point de départ pour de nombreux contextes de développement.

1. Trois utilisations de Faust dans les cours

Je souhaite montrer comment j?utilise Faust dans mes enseignements au CCRMA. Je l?utilise essentiellement de trois façons. Les étudiants de mes cours sont généralement à un niveau Master (quelques étudiants de premier cycle, quelques candidats au doctorat, mais surtout des étudiants de Master). Ils ont déjà quelques compétences en informatique et sont censés avoir acquis des bases solides en mathématiques dans le secondaire. Les enseignements concernent le traitement du signal appliqué au son et à la musique. Mes cours se déroulent plus ou moins en classe inversée, dans la mesure où mes conférences sont toutes préenregistrées et distribuées comme des devoirs à la maison, et donc en classe nous pouvons faire ce que nous estimons être le plus important. Faust participe de plusieurs façons à mes activités de classe inversée.

Trois utilisations de Faust dans les cours :

dans mes enseignements sur le traitement de signal au niveau Master, j?utilise Faust pour :

- des démonstrations en temps réel sur le traitement du signal audio ;
- la programmation en direct d?algorithmes de traitement du signal ;
- des implémentations de référence pour des utilisations ultérieures.

Ma première utilisation concerne des *démonstrations* en temps réel sur le traitement du signal audio. L?idée est de proposer un sujet, en fournissant une application Faust ou un plugin de démonstration, ce qui permet aux étudiants d?écouter le son traité ou synthétisé et de me voir le contrôler en temps réel.

Le deuxième type d?utilisation de Faust en classe est la programmation en direct. La plupart des programmations que nous réalisons en direct sont faites dans MATLAB, mais Faust est un langage très agréable pour la programmation en direct car il produit une application ou un plugin qui peut fonctionner directement. La syntaxe Faust se situe à un niveau élevé et elle est concise. Il est possible d?écrire de manière très lisible quelque chose qui ressemble beaucoup à du MATLAB, que les étudiants connaissent déjà.

Ma troisième façon d?utiliser Faust consiste à donner aux étudiants une bibliothèque d?implémentations de référence. Faust est un langage dans lequel on peut mettre en ?uvre de manière compacte les algorithmes standard ? les algorithmes les plus connus pour diverses situations ? et les mettre dans une bibliothèque à laquelle les utilisateurs peuvent facilement accéder lorsqu?ils travaillent sur des projets de classe, ou qu?ils pourront continuer à utiliser après l?obtention de leur diplôme, lorsqu?ils seront dans une situation professionnelle. Très souvent, les tâches à réaliser sont programmées en C++, et ils peuvent facilement compiler de Faust vers C ++, puis copier et coller le résultat là où ils en ont besoin. Faust est un langage utile à l?apprentissage des bonnes pratiques de la programmation dans le domaine du traitement du signal audio.

2. Introduction à Faust

Faust (Functional Audio Stream) :

- un langage de programmation fonctionnel haut niveau pour le traitement du signal et la spécification d?interfaces graphiques pour les utilisateurs (GUI) ;
- compile vers C++, LLVM, JavaScript?
- génère immédiatement des plugins et des applications autonomes utiles grâce à des fichiers d?architectures encapsulant des API spécifiques à chaque plateforme.

Faust est un acronyme pour Functionnel AUdio STream. Cela signifie que c?est un langage de programmation fonctionnel qui considère des blocs-diagrammes audio comme des objets de classe supérieure, des blocs-diagrammes opérant sur des flux audio.

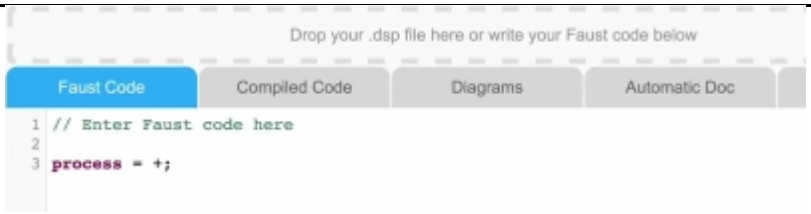
On peut également spécifier une interface graphique (GUI) avec des curseurs et d?autres dispositifs qui sont très utiles. C?est formidable de pouvoir ainsi produire un programme complet qui soit utilisable immédiatement quand il est compilé et exécuté dans un environnement donné, et il existe de plus de très nombreux environnements pour lesquels il peut être compilé.

Un programme peut être compilé vers C++, mais aussi pour LLVM, JavaScript et même simplement en C. On dispose donc d?un très bel ensemble de langages en *backend* disponibles.

Pour accéder aux différents environnements selon les plateformes, il existe une notion de « fichiers d?architecture », des fichiers qui encapsulent l?API (Application Programming Interface) spécifique à la plateforme. À la base, l?architecture est un exemple du programme que vous voulez produire, avec des crochets qui indiquent où se produit l?initialisation du bloc-diagramme, où est réalisé le traitement du signal et où sont placés les éléments de l?interface graphique. La disponibilité de tous ces fichiers d?architecture élargit considérablement la portabilité immédiate de l?ensemble.

Voyons cela en détail. J?aime présenter Faust en utilisant son compilateur en ligne.

Figure . Le compilateur Faust en ligne

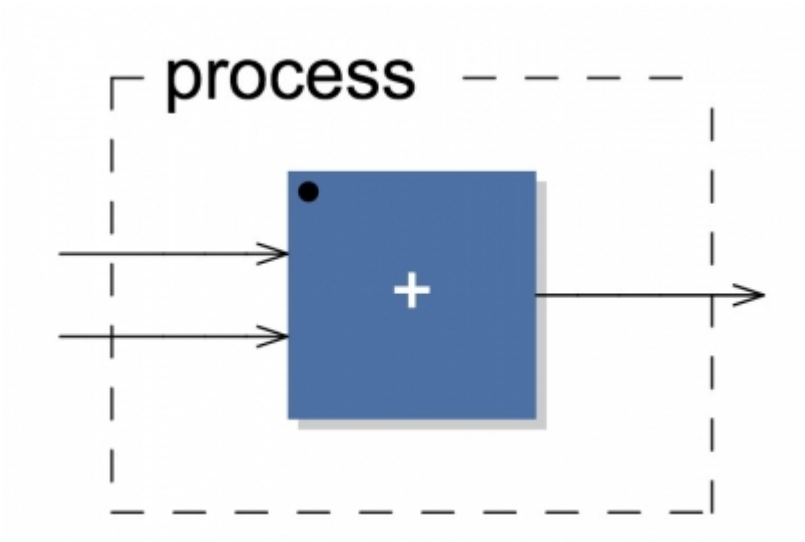


Source : http://faust.grame.fr/onlinecompiler/

La figure 1 donne un aperçu de ce à quoi ressemble le compilateur quand on y accède. Nous avons un programme très simple, une fonction appelée « process » qui est définie par « + » (« plus »). C?est une définition de la fonction. Dans Faust, « process » est une fonction spéciale qui doit toujours être présente. C?est la fonction principale de haut niveau, qui représente donc le bloc-diagramme principal qui sera ensuite défini typiquement grâce à d?autres fonctions. Ces fonctions sont définies ailleurs, elles peuvent l?être avant ou après, l?ordre des instructions peut être arbitraire, et lorsque l?ensemble se développe, on obtient un diagramme. Ce bloc-diagramme spécifie la manière dont les entrées sont traitées pour produire la sortie.

Dans Faust, « + » est un bloc-diagramme prédéfini qui signifie « ajouter deux signaux d?entrée pour produire un signal de sortie ». L?une des grandes fonctionnalités de Faust est la capacité de générer un bloc-diagramme à partir d?un langage de haut niveau. Voici ce que nous obtenons pour le schéma du bloc « plus ».

Figure . Le bloc-diagramme « + »



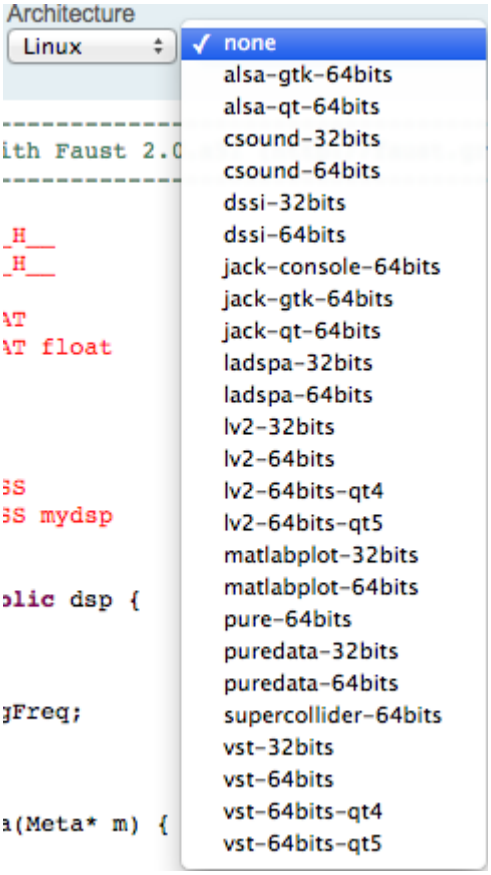
Ainsi, le processus est « + », avec deux entrées et une sortie. Lorsque je travaille avec Faust, j?ai simplement à regarder ce schéma. En d?autres termes, (1) j?écris le code, (2) je m?assure que la syntaxe est correcte, c?est-à-dire qu?elle compile et, pendant qu?elle compile, (3) je regarde le diagramme et si ce dernier me semble correct, cela doit fonctionner, tout simplement. Donc, en d?autres termes, le débogage consiste le plus souvent à vérifier le diagramme.

Ensuite, une chose que je fais en classe est de montrer le code compilé. C?est un exemple de ce qui est produit si vous avez spécifié un langage de sortie et un type de fichier d?architecture particuliers :

Figure . Les langages et les architectures disponibles avec le compilateur en ligne

Il vaut mieux commencer par « none » à ce stade. Ensuite je demande le langage C++ (mais je peux également avoir plus de choix), et je spécifie la plateforme Linux. Je peux aussi choisir Windows, OSX, Raspberry, Android, Ros et Web. Sous Linux, j?ai à nouveau plusieurs possibilités :

Figure . Architectures disponibles pour la plateforme Linux



Il existe de nombreux types de programmes autonomes et de plugins. Ceux que j?utilise le plus souvent sont les programmes autonomes pour Jack. Je préfère QT, mais on peut également utiliser Gimp ToolKit. Il existe d?autres alternatives de haut niveau sur les systèmes Windows. Pour les plugins, j?aime utiliser LV2, grâce aux travaux d?Albert Graëf [(1)]. On a également la possibilité de générer des modules pour MATLAB. Vous pouvez contrôler les signaux et aller directement vers pure (toujours grâce à Albert [(2)]). On peut également produire des plugins pour PureData, Supercollider, Chuck et Csound. C?est donc un ensemble de possibilités très riche.

Si vous avez un fichier d?architecture supportant la norme OSC, vous pouvez le rajouter, avec la case à cocher OSC, puis vous pouvez également générer un exécutable dans l?architecture courante choisie.

Examinons maintenant le code suivant. C?est le code C++ généré à partir du programme succinct

« process = + » :

```
14 class mydsp : public dsp {
15
16   private:
17
18     int fSamplingFreq;
19
20   public:
21
22     void static metadata(Meta* m) {
23     }
24
```

Au début, nous trouvons quelques définitions. Nous avons une classe, c?est la classe dsp, puis les variables dont nous avons besoin : la fréquence d?échantillonnage que nous utilisons. La rubrique « metadata » concerne les différentes étiquettes, etc.

```
25     virtual int getNumInputs() {
26         return 2;
27
28     }
29     virtual int getNumOutputs() {
30         return 1;
31
32     }
```

Vous pouvez demander le nombre d?entrées et de sorties que vous voulez, donc respectivement deux et une dans cet exemple.

```
33     virtual int getInputRate(int channel) {
34         int rate;
35         switch (channel) {
36             case 0: {
37                 rate = 1;
38                 break;
39             }
40             case 1: {
41                 rate = 1;
42                 break;
43             }
```

Vous pouvez indiquer le taux d?échantillonnage et ainsi de suite.

```
83
84     virtual void buildUserInterface(UI* interfa
85         interface->openVerticalBox("0x00");
86         interface->closeBox();
87
88     }
```

Vous pouvez voir où l?interface graphique est définie, mais dans ce cas, il n?y en a pas.

```
74      virtual void instanceInit(int samplingFreq) {
75          fSamplingFreq = samplingFreq;
76
77      }
78
79      virtual void init(int samplingFreq) {
80          classInit(samplingFreq);
81          instanceInit(samplingFreq);
82      }
```

Ensuite, ce sont des méthodes d?initialisation.

```
115     virtual void compute(int count, FAUSTFLOAT** inputs, FAUSTFLOAT** outputs) {
116         FAUSTFLOAT* input0 = inputs[0];
117         FAUSTFLOAT* input1 = inputs[1];
118         FAUSTFLOAT* output0 = outputs[0];
119         for (int i = 0; (i < count); i = (i + 1)) {
120             output0[i] = FAUSTFLOAT((float(input0[i]) + float(input1[i])));
121
122         }
123
124     }
```

La fonction « compute » peut être assimilée au c?ur d?un processus unique en action. Les arguments de la fonction « compute » sont la taille du buffer, généralement 64 échantillons, c?est la taille du buffer audio typiquement, ainsi que les signaux d?entrée et de sortie. Notre exemple comporte un code placé dans une boucle. C?est une boucle sur tous les échantillons pour simplement ajouter les deux nombres afin de produire la sortie.

Il est très agréable d?avoir une idée sur la façon dont nous spécifions un code de bas niveau à partir d?une formulation de haut niveau aussi élégante.

3. Démonstrations de Faust en classe

Pour introduire un nouveau sujet, il est utile de commencer par une démonstration faite directement en classe, afin de présenter le cadre et justifier le sujet.

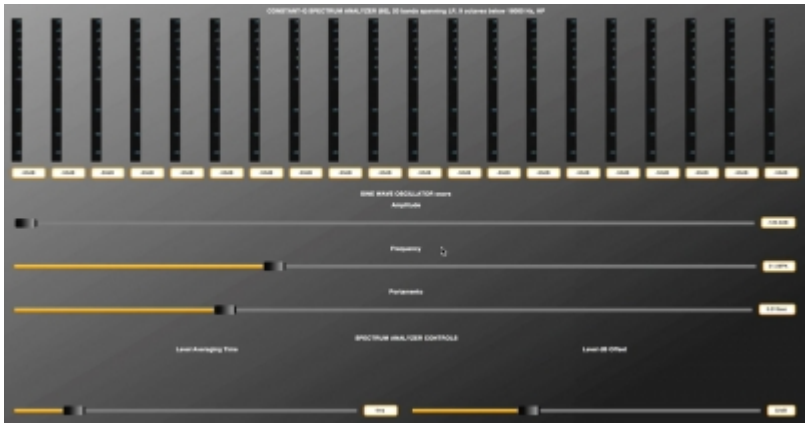
Les démonstrations de traitement de signal audio sont généralement :

- des applications Qt autonomes générées directement par Faust ;
- utilisées lors de la première séance d?une classe et lors du démarrage d?un nouveau cours.

Les démonstrations illustrent des utilisations pratiques et ajoutent de la motivation.

J?utilise des démonstrations en classe pour présenter un sujet, établir le cadre et motiver les élèves. Je parle généralement de sujets que les élèves connaissent déjà, comme des analyses spectrales de base, etc. Nous étudions des applications Faust en fonctionnement, discutons des blocs-diagrammes et regardons éventuellement un aperçu du code. En général, j?ouvre la première séance de cours avec ce type de démonstrations, ce qui permet d?aborder l?ensemble de ce que nous avons à apprendre. Les démonstrations sont également intéressantes lors du démarrage d?un nouveau cours. Elles permettent d?aborder d?emblée des utilisations pratiques et ajoutent une motivation à la classe. Les démonstrations que j?utilise le plus souvent sont des applications QT autonomes, générées directement à partir de Faust en utilisant un fichier d?architecture Qt, qu?il s?agisse d?un support JACK (Linux) ou Core Audio (Mac).

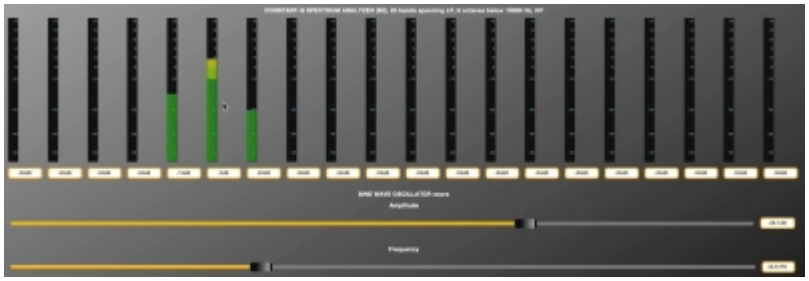
Figure . Un analyseur de spectre à Q constant en Faust



Voici une démonstration de la banque de filtres de la bibliothèque *filter.lib* utilisée comme analyseur de spectre. Nous avons un analyseur de spectre à Q constant fabriqué à l?aide d?une banque de filtres. Nous disposons de vingt bandes, passant par un filtre passe-bas, par neuf filtres passe-bande par octaves, inférieurs à 16 kilohertz, et finalement par un filtre passe-haut. Nous disposons de bargraphes (vumètres) configurés en dB qui indiquent le niveau du signal qui les traverse. Ensuite, en bas, nous trouvons le contrôle de la moyenne temporelle. Les analyseurs de spectre effectuent généralement une moyenne sur le spectre de puissance pour le rendre plus stable. En réduisant la durée pour le calcul de la moyenne de niveau, nous permettons à l?affichage du niveau spectral de changer très rapidement. Il s?agit là uniquement du filtrage en fonction du temps de la puissance de sortie de chaque bande individuelle issue de la banque de filtres. À droite, on trouve un décalage arbitraire de l?affichage du niveau. Ceci est un programme entièrement construit à partir de l?exemple de programmation Faust nommé « spectral_level.dsp ».

Nous produisons une onde sinusoïdale et pouvons en contrôler l?amplitude. Elle arrive dans l?entrée de la banque de filtres. Ensuite, nous pouvons modifier sa fréquence en temps réel. On peut voir, même s?il ne s?agit que d?un son pur, que celui-ci excite également les bandes des filtres adjacents, car les canaux de ces filtres ne sont pas idéaux. Nous obtenons toujours plusieurs bandes d?énergie dans le spectre. Il est facile d?augmenter *l?ordre* des filtres utilisés dans la banque de filtres afin de réduire ce « débordement » vers les bandes adjacentes.

Figure . Le spectre d?une onde sinusoïdale



Cet analyseur de spectre, basique, fabriqué à l?aide d?une banque de fil à Q constant, est conforme à la nature de l?audition. La résolution en fréquence de l?audition est à Q constant en première approximation sur la plus grande partie du spectre. Des échelles de fréquence auditives plus précises existent comme les Mel, les Bark et Equivalent Rectangular Bandwidth (ERB).

Le contrôle du portamento peut être facilement illustré. En augmentant le portamento, on obtient une transition de fréquence plus lente. Il est donc très rapide et facile de se faire une idée de la manière dont fonctionne le portamento. Je n?insiste donc pas beaucoup là-dessus. Voilà qui illustre « comment faire un analyseur spectral à l?aide d?une banque de filtres ». Ceci fait partie d?un cours sur les filtres et il s?agit donc d?un bon moyen pour mettre en place un affichage et discuter de la façon dont on peut faire cet affichage avec des filtres.

```
// Spectrum analyzer

declare name      "spectrum_level";

ol = library("oscillator.lib"); // /l/fad/oscillator.lib
fl = library("filter.lib");

BandsPerOctave = 2;
process = ol.oscrs_demo : fl.mth_octave_spectral_level_demo(BandsPerOctave) <: _,_;
```

```
// Spectrum analyser (3)

declare name "spectrum_level";

import ("stdfaust.lib");

declare BandsPerOctave = 2;

process = dm.oscre_demo : dm.mth_octave_spectral_level_demo(BandsPerOctave);
```

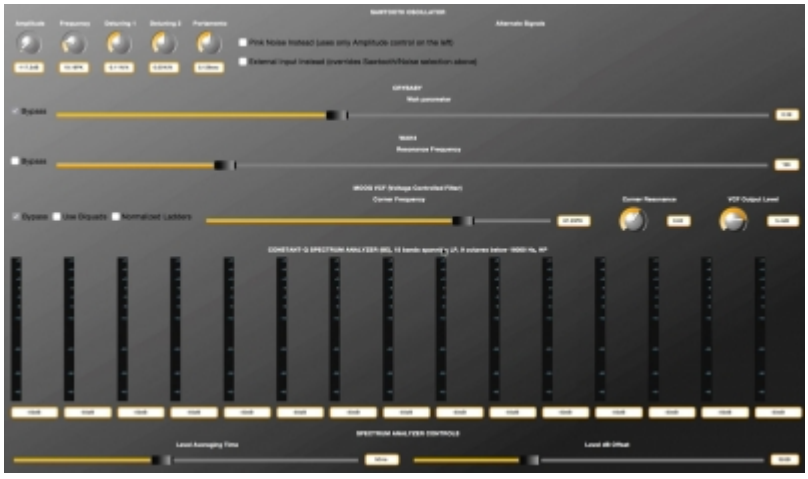
Voici le code du programme utilisé pour la démonstration du même type d?oscillateur entrant dans une banque de filtres. C?est très court car cela utilise des démos déjà définies dans les bibliothèques d?oscillateurs et de filtres distribuées avec Faust. Ces démos sont utilisées dans les exemples de Faust afin de conserver un code très condensé.

La fonction « process » est définie avec l?oscillateur « oscrs » (« osc » et « rs ») de la bibliothèque oscillator, avec « r » qui signifie une rotation de phase et « s » une sortie sinusoïdale (il existe également une fonction cosinus). Il existe une démo qui délivre une petite interface pour contrôler l?amplitude et la fréquence. Nous la branchons vers le « spectral_level_demo » de la bibliothèque des filtres.

Le spectral_level_demo utilise le « mth_octave_spectral_demo » avec un argument « BandsPerOctave » que nous avons réglé sur 2. Typiquement, on utilise une banque de filtre par tiers d?octave, mais cela prendrait trop de place pour tenir sur mon écran, je l?ai donc simplement remplacée par une banque de filtres par demi-octaves.

Le programme est aussi simple que ça. Bien sûr, lorsqu?on entre dans le code, il y a quelques lignes en plus, mais pas trop. Faust est un langage de haut niveau très compact.

Figure . Un synthétiseur simple avec trois types de filtres différents



La distribution Faust comprend un exemple qui s?appelle « vcf_wah_pedals.dsp » et qui montre l?utilisation de filtres variables agissant sur un oscillateur en dents de scie et dans lequel nous avons trois

filtres différents. Tout en bas, nous avons à nouveau la démo *spectral_level*, comme dans l?exemple précédent. Nous avons le même niveau de calcul du temps et le contrôle du décalage de niveau. Par contre, au lieu d?une onde sinusoïdale, nous introduisons ici une onde en dents de scie afin d?obtenir un spectre plus riche. L?oscillateur en dents de scie dispose d?un contrôle d?amplitude ainsi que d?un contrôle de la fréquence. Il existe un contrôle du portamento, comme précédemment. L?onde en dents de scie donne un spectre plus large ici, qui décroit de 6 dB par octave. Nous pouvons désactiver les filtres et écouter la différence. Une autre chose qui se fait couramment avec ce type d?oscillateurs « analogiques virtuels » est de désaccorder plusieurs oscillateurs à des fréquences très proches. Nous avons trois oscillateurs, un juste sur la bonne fréquence et les deux autres qui sont légèrement désaccordés. On peut obtenir de cette façon les sons classiques des oscillateurs vintage.

Discutons désormais des filtres. L?onde en dents de scie possède un spectre riche et nous pouvons élargir ou resserrer la bande passante en utilisant des filtres variables. Observons maintenant le « paramètre Wah ». Il s?agit de la pédale wah qu?utilisent de nombreux guitaristes, qui est en fait une numérisation de la pédale Crybaby wah. C?est un ingrédient de base classique dans le bagage de tout guitariste, qu?on observe souvent dans les performances à la guitare électrique.

Passons ensuite au filtre suivant. Si on désactive la crybaby (bypass) et qu?on active la « wah4 », on monte le volume, puis on observe.

Il s?agit d?un filtre variable d?ordre 4, qui nous permet d?écouter la différence entre des filtres de deuxième, troisième et quatrième ordre. Le Crybaby est un filtre d?ordre 2. Ce n?est qu?un résonateur unique d?ordre 2. Un autre cas de filtre d?ordre 4 très célèbre est le Moog VCF, qui est celui que les joueurs de synthétiseur utilisent le plus. Il s?agit d?un circuit Moog *(ladder)* d?ordre 4, un circuit classique, mais nous pouvons le mettre en ?uvre de la même manière que le Wah d?ordre 4 avec deux sections biquadratiques (« biquad »). Il y a trois implémentations ici pour permettre des comparaisons.

Cette démonstration ressemble beaucoup à un synthétiseur analogique dans lequel vous avez un oscillateur en dents de scie et un filtre à tension contrôlée (VCF) qui filtre l?onde en dents de scie pour constituer une variante classique de « synthèse soustractive ». On trouvait cela couramment dans les dispositifs analogiques, contrôlé par tensions (d?où le nom de « filtre contrôlés par tension »), et c?est d?ailleurs Robert Moog qui a développé le premier circuit classique, encore largement utilisé aujourd?hui.

Je peux également contourner tous ces filtres et écouter le signal d?origine. Nous pouvons implémenter l?un de ces filtres en utilisant une ou deux biquads. Un biquad peut être implémenté à l?aide d?une structure de filtre *ladder*, qui présente assez peu d?artefacts lorsque vous le modifiez rapidement. J?ai démontré cela lors de la Linux Audio Conference 2012 (Smith, 2012), donc je ne le détaillerai pas ici, mais c?est une caractéristique très importante des filtres *ladder* normalisés que de permettre de déplacer une fréquence instantanément sans générer d?autres « pop » que ceux qu?ils produisent naturellement, du fait que l?énergie est normalisée dans ces filtres, et donc que les coefficients de l?énergie ne sont pas couplés avec l?énergie du signal. On peut moduler arbitrairement ces filtres même à des vitesses audio sans obtenir d?artefacts en raison de ce qu?on appelle « l?amplification paramétrique ».

Une autre bonne chose pour l?enseignement est une illustration de la pente de résonance *(Corner Resonance)*. Pour cela, nous augmentons la pente de la résonance jusqu?à ce que nous obtenions un sifflement.

Nous pouvons voir ce qui se passe dans l?affichage spectral. Lorsque nous augmentons la pente de la résonance, nous constatons davantage de résonance de bordure, à droite du spectre. Lorsque la résonance est plus faible, nous voyons une forme en passe-bas dans l?affichage du spectre. Le spectre reste un passe-bas lorsque je déplace la fréquence de VCF. Il n?y a pas de relation entre DC et le fondamental, donc cela peut aider à accorder le fondamental. Nous pouvons produire un gros son classique avec une dent de scie et un VCF analogiques virtuels.

Figure . Le spectre d?un oscillateur en dents de scie filtré par une émulation de filtre MOOG VCF

```
vcf_wah_pedals.dsp — Edited
ol = library("oscillator.lib");
fl = library("filter.lib");
el = library("effect.lib");

process =
 vgroup("[1]", ol.sawtooth_demo) :
 vgroup("[2]", el.crybaby_demo) :
 vgroup("[3]", el.wah4_demo) :
 vgroup("[4]", el.moog_vcf_demo) :
 vgroup("[5]", fl.spectral_level_demo) <:
  _,_;
```

Ci-dessus, vous trouvez le code de la démo précédente. L?onde en dents de scie entre dans le passe-bas Crybaby d?ordre 4, le passe-bas d?ordre 4, le Moog VCF, un autre passe-bas d?ordre 4 et enfin, dans la démo servant à l?affichage spectral. Ceux-ci sont tous en série, comme le montre l?opérateur « : » dans Faust. La sortie est distribuée en stéréo, et l?ordonnancement de l?affichage est déterminé par les nombres entre crochets (« [1] »).

Les fonctions ci-dessus se trouvent toutes dans les bibliothèques Faust, et les illustrations associées sont contenues dans la distribution Faust dans l?exemple appelé « vcf_wah pedals.dsp ».

Voici comment j?ai compilé ces démos à partir de l?exemple Faust :

```
>/usr/local/bin/faust2caqt vcf_wah_pedals.dsp
./vcf_wah_pedals.app;
>open vcf_wah_pedals.app/
```

J?ai utilisé le script « faust2caqt » (Faust to Core Audio QT) qui vient avec la distribution Faust, en lui donnant l?exemple source « vcf_wah_pedals.dsp », et cela a créé une application que je peux ouvrir en utilisant la commande Apple « open ». C?est ainsi que nous avons fait les choses dans les démos précédentes.

Maintenant, je veux faire quelque chose de différent : utiliser un script Faust vers JACK QT (« faust2jaqt ») et le connecter à la démo « ZitaRev1 ». La raison pour laquelle j?utilise JACK consiste à tester ZitaRev1 avec une application autonome séparée appelée « reverb_tester.dsp ».

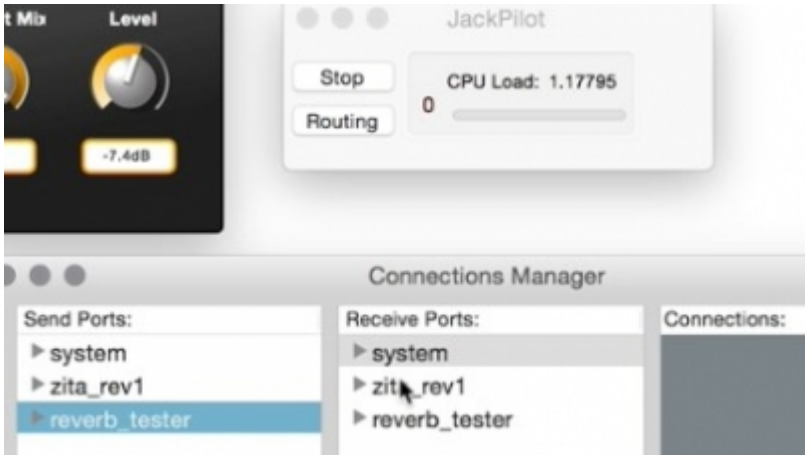
```
>/usr/local/bin/faust2jaqt zita_rev1.dsp
```

Figure . Utilisation de deux applications connectées ensemble via JACK

Nous avons un testeur de réverbération stéréo qui émet une impulsion dans les canaux du milieu, de gauche ou de droite. On peut également écouter le son produit par des microphones externes. On peut également alimenter le système avec un bruit rose.

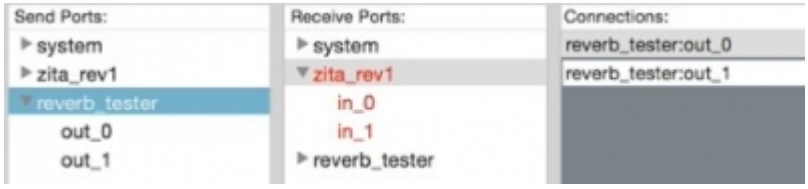
Ce que nous utilisons ici est la réverbération Zita Rev1 (Zita_Rev1) de Fons Adriaensen, portée dans Faust et présente dans la bibliothèque de Faust « effect.lib ». Elle a un délai en entrée, et des contrôles de bande passante du filtre qui sont étiquetés « low-frequency » et « transition-frequency ». On peut régler le temps de décroissance (le temps d?atténuation à -60 dB) ainsi que trois canaux : « LowRT », « MidRT » et un canal spécial haute fréquence, « HF Damping », qui contrôle l?amortissement. Il y a également une égalisation, le dosage « Dry/Wet Mix », et un contrôle du niveau global. C?est une très belle réverbération, très riche.

Figure . Les connexions Jack



Voici comment nous les connectons : nous avons besoin de JackPilot ou d?un autre gestionnaire JACK, afin de vérifier que le serveur fonctionne. Le bouton de routage ouvre un panneau dans lequel on peut se connecter. Dans cet exemple, j?ai le testeur de réverbération et une Zita_Rev1. Je peux sélectionner reverb_tester et double-cliquer sur zita_rev1 pour effectuer les connexions. Il connecte automatiquement n?importe quel nombre de canaux lorsque vous faites cela.

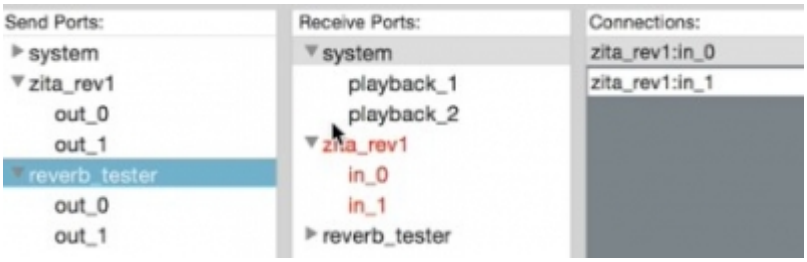
Figure . Les connexions Jack



Nous sommes désormais prêts à l?essayer en lui donnant une impulsion et en écoutant la réponse impulsionnelle.

Un exemple classique de la classe inversée est le fait pour moi, lorsque quelque chose ne fonctionne pas, de faire appel à la classe pour obtenir de l?aide. Qu?est-ce qui ne va pas ? Pourquoi ne reçois-je aucun son ? Pourquoi ne reçois-je aucun son avec cet exemple ? Cela réveille les étudiants et les motive. Ils participent plus et m?accordent davantage d?attention. Ils proposent des idées et très rapidement quelqu?un dit quelque chose comme : « vous n?avez pas branché la sortie ». Dans cet exemple, j?ai connecté le Reverb Tester à la Zita_Rev1, mais je n?ai jamais connecté Zita_Rev1 aux sorties du système. Je peux le faire en double-cliquant et la Zita_Rev1 va sur les sorties 1 et 2.

Figure . Les connexions Jack



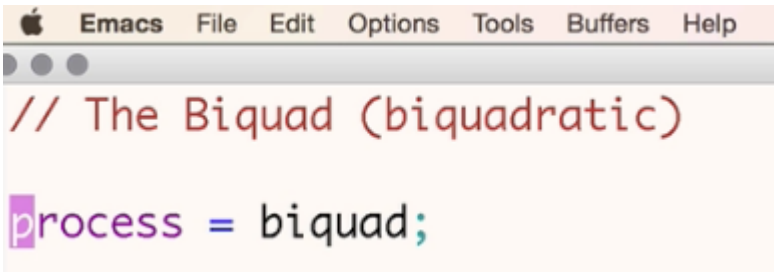
Maintenant, nous devrions avoir du son et entendre une très belle réverbération.

Ce sont là quelques exemples d?applications autonomes utilisées en classe.

4. La programmation Faust en direct en classe

Je pense que la programmation en direct est actuellement mon activité favorite pour les salles de classe inversées. Cela fait vraiment participer les étudiants et cela les réveille, ce qui maximise leur engagement.

Prenons un exemple de codage en direct dans Faust pour produire un biquad (« fonction de transfert biquadratique »). Son équation comporte un numérateur de second ordre et un dénominateur de second ordre. L?un des avantages de la programmation dans Faust, c?est que vous pouvez mettre les déclarations dans n?importe quel ordre. Nous pouvons commencer par dire « process = biquad », puis définir biquad, et ainsi de suite :



Pour le calcul de la réponse impulsionnelle, nous pouvons créer une impulsion en prenant la constante 1 et en soustrayant cette constante 1 retardée d?un échantillon. Cela nous donnera une bonne impulsion pour le biquad :

```
process = 1-1' : biquad;
```

Maintenant, nous pouvons définir ce qu?est le biquad. Pour plus de clarté, nous pouvons définir une « partie FIR » suivie d?une « partie IIR ». Nous pouvons les intervertir, c?est-à-dire brancher la partie

FIR dans la partie IIR et inversement, cela fournit la même réponse impulsionnelle, et un exemple du fait que les systèmes linéaires invariables dans le temps sont commutatifs quand ils sont en série.

```
biquad = firPart : iirPart;
```

Maintenant, nous pouvons définir les coefficients :

```
firPart(x) = b0 * x + b1 * x' + b2 * x'';
```

```
iirPart = + ~( _<: *(-a1),  (mem:*(-a2)) :>_ );
```

Maintenant, définissons les coefficients des filtres :

```
// Two zeros at z = -1:
b0 = 1;
b1 = 2;
b2 = 1;
```

```
// Two poles:
a1 = -2*R*cos(th);
a2 = R*R;


R = exp(-PI*B*T);
```

```
import("music.lib");
```

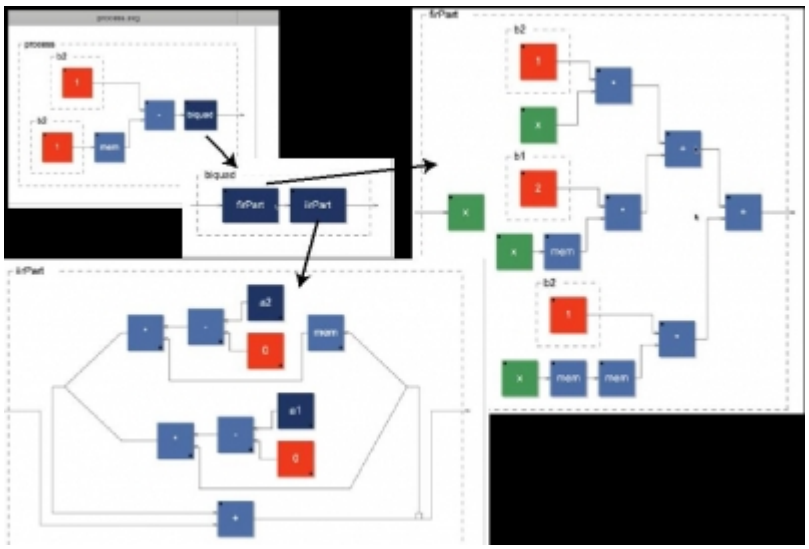
```
R = exp(-PI*B*T);
th = 2*PI*f*T;
T = 1/fs;
fs = 44100;
B = 100;
f = fs/4;
```

```
// The Biquad (biquadratic)
import("music.lib");
process = 1-1' : biquad;
biquad = firPart : iirPart;
firPart(x) = b0 * x + b1 * x' + b2 * x'';
iirPart = + ~( _<: *(-a1),  (mem:*(-a2)) :>_ );
// Two zeros at z = -1:
b0 = 1;
b1 = 2;
b2 = 1;
// Two poles:
a1 = -2*R*cos(th);
a2 = R*R;
R = exp(-PI*B*T);
th = 2*PI*f*T;
T = 1/fs;
fs = 44100;
B = 100;
f = fs/4;
```

```
-:---  b.dsp          Bot L20    (FAUST mode)
                        // post processing
                        fRec0[2] = fRec0[1]; fRec0[1] = fRec0[0];
                        iVec1[2] = iVec1[1]; iVec1[1] = iVec1[0];
                        iVec0[1] = iVec0[0];
                }
        }
};

>which f2ff
f2ff:    aliased to faust2firefox
>f2ff b.dsp
>
-:**-  *shell*        Bot L74    (Shell:run)
```

Figure . Le schéma bloc-diagram d?un filtre biquad



Cela semble correct ! Nous regardons ce diagramme et pensons : « ça va fonctionner ».

```
faustout = [ ...
 1; ...
 2; ...
 0.0141466; ...
 -1.97171; ...
 -0.0139464; ...
 1.94381; ...
 0.0137491; ...
 -1.91632; ...
 -0.0135546; ...
 1.88921; ...
 0.0133629; ...
 -1.86248; ...
 -0.0131738; ...
 1.83613; ...
 0.0129875; ...
 -1.81016; ...
];

plot(faustout);
title('Plot generated by ./b made using ''faust -a matlabplot.cpp ...''');
xlabel('Time (samples)');
ylabel('Amplitude');
>f2ff b.dsp
>f2o b.dsp
```

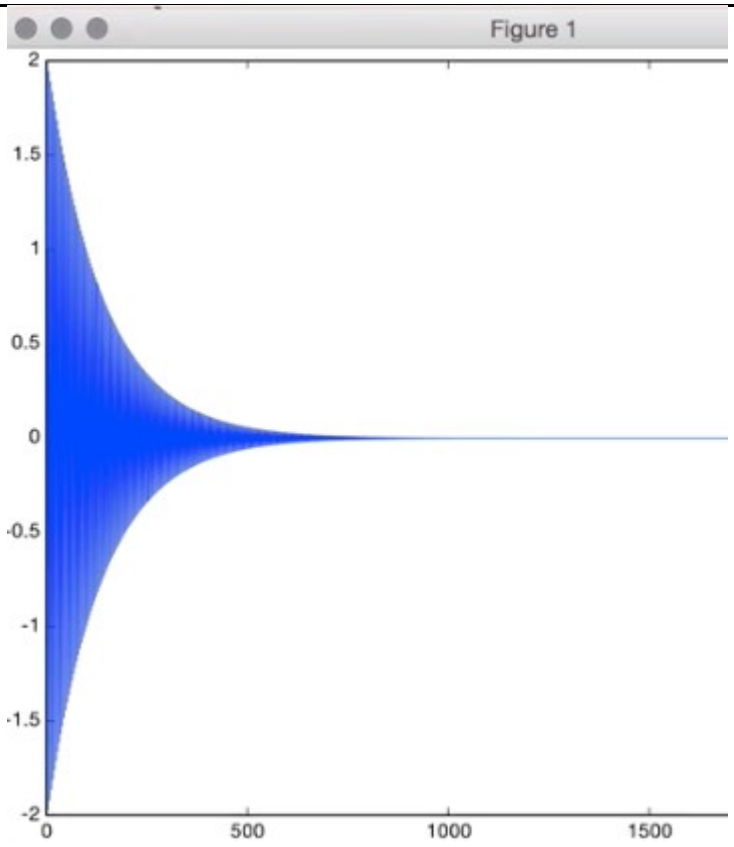
Voyons notre ligne de commande affichée avec Emacs. Maintenant, je vais exécuter faust2octave sur « b.dsp » (f2o b.dsp), parce que je veux montrer une manière élégante d?observer ces choses qui sont des filtres. On peut les observer dans le domaine fréquentiel. Je peux connecter le signal de sortie dans Octave en tapant « plot (faustout) » :

```
octave:2> plot(faustout)
```

S?il y a plusieurs signaux de sortie, ils seront tous superposés.

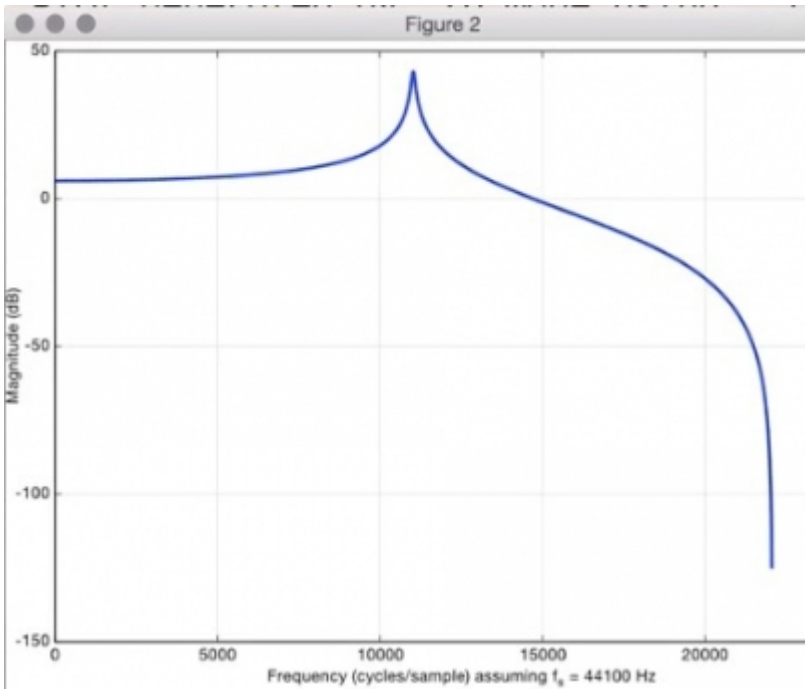
Voici donc notre réponse impulsionnelle, sonnant comme il se doit.

Figure . Enveloppe temporelle de la réponse impulsionnelle



J?obtiens un petit script Octave qui calcule la FFT et fournit une représentation dans le domaine fréquentiel. Il donne une échelle log-log.

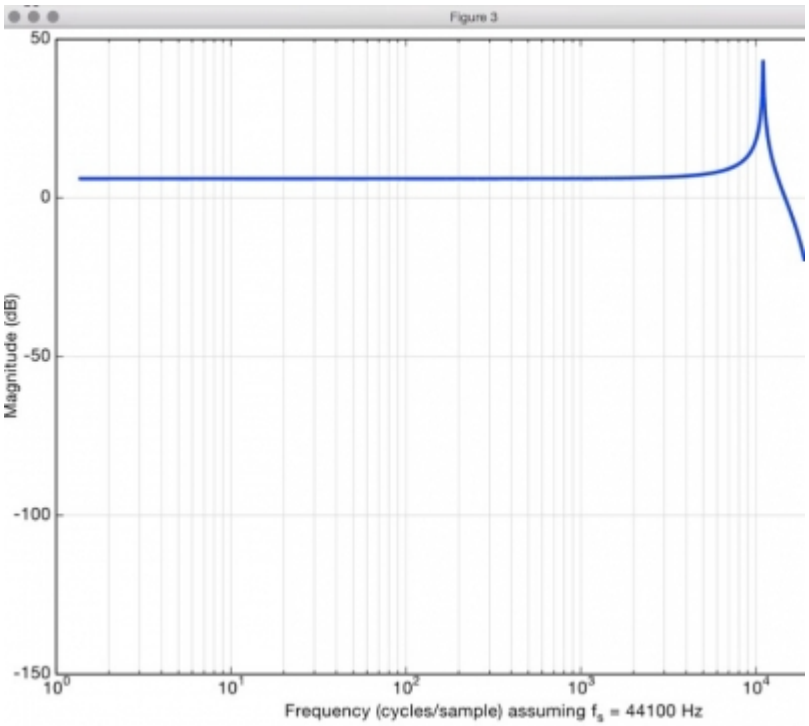
Figure . Enveloppe spectrale de la réponse impulsionnelle (échelle de fréquence linéaire)



On peut également obtenir une autre représentation, avec une échelle de fréquence linéaire. Nous voyons dans la figure 15 que nous avons un filtre passe-bas avec un pic (pseudo-résonance). La bande passante va de CC à 10 kHz, et la largeur de bande de la résonance n?est que de 100 Hz. Nous avons nos zéros

près de la moitié du taux d?échantillonnage, donc c?est qualitativement ce que nous attendions. Sur une échelle en log-log (Figure 16), plus orientée audio, nous voyons un passe-bande large, la résonance avec un pic, puis une extinction rapide.

Figure . Enveloppe spectrale de la réponse impulsionnelle (échelle de fréquence logarithmique)



Donc, il s?agit d?un exemple en codage direct en Faust du dispositif de traitement appelé le biquad, configuré comme un filtre passe-bas. Je peux le reconfigurer rapidement comme un filtre passe-haut en déplaçant les zéros de la moitié de la fréquence d?échantillonnage à 0 Hz :

```
b0 = 1;
b1 = -2;
b2 = 1;
```

high-pass

```
b0 = 1;
b1 = 0;
b2 = 1;
```

band-pass

Nous pouvons le configurer comme un passe-bande en mettant un zéro à la moitié de la fréquence d?échantillonnage et l?autre à 0 Hz, ce qui donne les modes classiques passe-bas, passe-bande et passe-haut que l?on voit souvent parmi les filtres de second ordre, en particulier les filtres à état variable. Voilà donc un exemple de codage en direct.

5. Implémentations Faust de référence

Implémentations de référence FAUST

Les bibliothèques Faust suivantes ont accumulé des implémentations de référence pour le traitement du signal audio :

- osc.lib = oscillateurs analogiques et analogiques virtuels
- filter.lib = filtres numériques de différents types

- effect.lib = effets audio numériques

Voir le document [Linux Audio Conference 2012] pour un aperçu

- Les implémentations de référence sont utiles aux étudiants les plus avancés dans leurs projets logiciels optionnels.
- Ils sont également utiles quand ces étudiants sortent dans le monde et doivent mettre en ?uvre quelque chose qu?ils connaissent mais n?ont pas encore codés.
- Le fait que Faust compile sur C++ et soit portable immédiatement sur une grande variété de formats de plugins standard les rendent très utile comme point de départ dans de nombreux contextes de développement.

Pour finir, nous discuterons davantage du troisième usage en classe, *les implémentations de référence*.

Nous avons déjà vu des exemples de la bibliothèque « oscillator », de la bibliothèque « filter » et de la bibliothèque « effects », et je vous renvoie aux publications de la Linux Audio Conference (Smith, 2008, 2012) pour en savoir plus. Vous pouvez télécharger les transparents et la vidéo (4).

Il est agréable de disposer d?implémentations de référence gratuites dans n?importe quel domaine qui permettent de travailler rapidement et facilement. Nous devrions tous contribuer à de tels efforts. Le fait que Faust compile vers C++, qui est actuellement le langage numéro un dans le monde, en fait un outil largement utilisable. Avec ses nombreux fichiers d?architecture, Faust importe facilement vers une grande variété d?environnements hôtes, et divers types d?applications autonomes et de plugins. Faust est également un langage de haut niveau bien conçu : il est très lisible, agréable à écrire, et peut être condensé, resserré et élégant, et il génère un code bien optimisé, très concurrentiel par rapport à du C++ codé à la main. Je l?ai utilisé dans le développement d?applications iOS, et je n?ai généralement pas pu le battre avec un code écrit à la main. J?ai essayé d?écrire diverses choses en C++, mais je reviens à Faust afin d?avoir un logiciel plus propre et plus fiable, dont l?efficacité est vraiment très bonne. Il continue de s?imposer pour moi comme le principal langage pour le traitement du signal.

Conclusion

L?expérience à ce jour indique que :

? les démonstrations en classe fonctionnent très bien pour éclaircir ce qu?il faut étudier ;

? les exemples de programmation en direct fonctionnent bien comme des représentations alternatives et incitent à apprendre Faust ;

? les démonstrations en temps réel et la programmation en direct permettent de bien profiter du temps de cours supplémentaire offert par les classes inversées ;

? les implémentations de référence sont utiles pour un développement pratique et permettent aux étudiants plus avancés de démarrer leurs projets de logiciels, à la fois à l?école et après l?obtention du diplôme ;

Pour conclure et pour résumer, je peux dire qu?à ce jour mon expérience montre que Faust est un excellent outil pour des démonstrations en classe, le codage en temps réel et des implémentations de référence. Nous avons plus de temps pour ces activités dans le cas de séances en classes inversées où les conférences ont été enregistrées à l?avance. Lorsque le cours est écrit et disponible en ligne, cela libère du temps en classe, qui nous pouvons utiliser pour faire tout le reste. Les démonstrations en classe sont utiles pour préparer les débats, et c?est quelque chose de plaisant. Le codage en direct est une excellente expérience en classe qui s?adresse particulièrement bien aux étudiants. Si le fait de participer est ce qui

permet principalement de mesurer la réussite de la classe, alors le codage en temps réel est un outil important qui devrait y être ajouté. Enfin, il existe des implémentations de référence qui sont utiles pour des développements pratiques, tant dans le cadre de projets réalisés en classe qu?à l?issue de cette formation, en essayant de faire des choses dans le monde extérieur. C?est agréable d?avoir beaucoup d?utilitaires de base écrits avec lesquels on peut commencer à aller de l?avant.

Je vais terminer en présentant une démo d?une application sur laquelle je travaille.

Figure . L?interface de GeoShred © Wizdom Music & MoForte Inc.

Voici GeoShred. C?est une application iOS pour iPad qui est basée sur un modèle physique d?une guitare. On peut complétement configurer cette guitare.

Figure . L?interface guitare de GeoShred

Il s?agit d?une guitare implémentée dans Faust. Elle comporte des contrôles sur les cordes, des contrôles sur le corps de l?instrument et des contrôles pour l?interprétation. On peut contrôler les caractéristiques du chevalet, qui dépendent de la nature de l?approche de la modélisation physique, toutes écrites dans Faust. En plus de la guitare, il existe plusieurs effets, également écrits dans Faust : des unités de distorsion (certaines numérisées à partir de schémas classiques), un égaliseur de quatre octaves, une unité d?écho, une ligne à retard, un phasing, un chorus, une réverbération, etc. Tout cela fonctionne en temps réel, la plupart même sur un iPad 2.

Voici une performance de quelqu?un qui arrive vraiment à en jouer :

https://youtu.be/Imm88pWlWTQ (lien vérifié le 30/05/2018).

https://youtu.be/CDLA8B8BkbQ (lien vérifié le 30/05/2018).

Et ici, j?interprète une mélodie de Camel : https://www.youtube.com/watch?v=kTyrT2wRj74 (lien vérifié le 30/05/2018).

1. Cf. https://bitbucket.org/agraef/faust-lv2 (lien vérifié le 30 mai 2018).

2. Cf. https://github.com/agraef/pure-lang/wiki/GettingStarted (lien vérifié le 30 mai 2018)

3. Tous les exemples de Faust présentés ici sont valables pour Faust version 0.9.85 ? dernière version

avant la nouvelle organisation des bibliothèques (septembre 2016) ? ainsi que pour beaucoup de versions antérieures.

4. http://lac.linuxaudio.org/2012/files (lien vérifié le 30 mai 2018).

Pour citer ce document:
Julius Smith, Â« Faust dans une salle de classe : démonstration, live, coding (programmation en direct) et implémentations de référence Â», *RFIM* [En ligne], Numéros, n° 6 - Techniques et méthodes innovantes pour l'enseignement de la musique et du traitement de signal, Mis Ã jour le 10/07/2018
URL: http://revues.mshparisnord.org/rfim/index.php?id=550