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

# Â«Â Le traitement du signal pour les musiciensÂ Â»

## Vesa Norilo

Résumé

Ce texte porte sur l?utilité pour des étudiants en musique, en composition et en ingénierie du son de s?intéresser aux techniques de traitement du signal. Nous exposons les questions et les idées concernant les principaux obstacles à cet apprentissage et présentons des outils logiciels appropriés pour y remédier. Les caractéristiques des langages de programmation, telles que la gestion de la complexité et de la simplicité, les capacités d'abstraction, la visualisation et le développement itératif sont examinées du point de vue de l'enseignant et de l'apprenant. En outre, nous proposons des synergies entre apprentissage, programmation créative et codage en direct.

# Introduction

Alors que le traitement du signal est un élément essentiel dans les musiques numériques, les obstacles à franchir pour y accéder demeurent importants. Même les manuels élémentaires de traitement du signal supposent d?être familier avec les nombres complexes, les analyses et, en général, d?avoir eu une formation d?ingénieur ou de mathématicien. En outre, pour construire des instruments vraiment utiles avec ces processeurs de signaux, le travail nécessite des compétences supplémentaires en programmation informatique. Il n?est pas étonnant que de nombreux musiciens trouvent le pas trop difficile à franchir.

Même si la créativité repose à la fois sur la musique et l?ingénierie, les différentes étapes du travail traditionnel sont en fait très différentes dans ces traditions respectives. Un design, des spécifications et une mise en ?uvre soignés sont les fondements d?une ingénierie solide, alors que le processus musical est souvent exploratoire, intuitif et itératif.

L?idée de la programmation informatique en tant que processus qui se réalise fondamentalement en temps différé est à la base de la forme et des concepts des systèmes informatiques. Pour les systèmes les plus anciens, il y avait une limite technologique : soumettre des cartes perforées le soir et, le matin, récupérer le résultat. Bien que les systèmes informatiques aient évolué de façon spectaculaire au cours des dernières décennies, cette idée définit toujours la façon dont de nombreux systèmes sont organisés.

Bien que ces étapes de travail, de la préparation minutieuse, du traitement et de l?évaluation des résultats soient adaptées à la nature traditionnelle de l?ingénierie, elles sont quasiment à l?opposé de la façon dont les musiciens travaillent. Quiconque travaille sur l?interprétation connaît à coup sûr l?importance fondamentale de l?interaction en temps réel, tant dans la pratique de studio que sur scène. Pour intéresser, inspirer les musiciens et leur donner des moyens, le traitement du signal et la programmation devraient être conçus comme des processus exploratoires et interactifs.

Ce chapitre décrit mon opinion en ce qui concerne la conception et la mise en ?uvre de langages de programmation pour les musiciens, qui puissent remplir à la fois une fonction pédagogique et une

fonction artistique.

# 1. Contexte

Tout comme la prospective a sa place dans la pratique musicale, l?exploration et la spontanéité sont des notions qui ne sont pas étrangères aux bons ingénieurs. L?histoire des systèmes informatiques est riche en exemples de systèmes et d?environnements de programmation interactifs et expérimentaux.

## 1.1. Histoire des débuts de l?informatique interactive

Pour les premiers ordinateurs, la réalité physique de l?unité centrale de calcul a déterminé le langage avec lequel ils ont été programmés. Plus tard, des jeux d?instructions ont été adoptés pour permettre à un programme écrit en code machine de s?exécuter sur plusieurs puces différentes mais compatibles.

Dans les années 1950, le langage de programmation Fortran a été développé chez IBM comme l?indique John Backus en 1978. Celui-ci déclare que l?objectif initial était de « concevoir un langage qui permettrait aux ingénieurs et aux scientifiques d?écrire eux-mêmes des programmes » pour l?ordinateur IBM 704 ? une étape marquante qui peut être considérée comme une préfiguration à ce que les langages pour la musique tentent actuellement.

Fortran a été conçu autour d?un compilateur qui pouvait produire un code machine efficace (Backus, 1978). Par contraste, Lisp était un langage de programmation pionnier qui était moins concerné par l?efficacité ou la réalité physique des machines, en se concentrant sur un ensemble sémantique minimum permettant de décrire des programmes informatiques arbitraires. John McCarthy (1960) l?avait envisagé, mais n?avait pas cru qu?une mise en ?uvre serait possible, jusqu?à ce que Steve Russell ne le prouve (Graham, 2004).

Ce qui rend Lisp particulièrement intéressant pour l?informatique interactive est la boucle lecture-évaluation-sortie (REPL pour *read-eval-print-loop*). REPL est un mode d?interaction entre le programmeur et l?ordinateur, où l?ordinateur Lit une expression, l?Evalue, Imprime les résultats et retourne au début de la Boucle. Peter Deutch a implémenté un REPL pour Lisp dès 1964 (Deutch & Berkeley, 1964).

Dans l?intervalle, Ivan Sutherland a conçu un système de programmation graphique appelé Sketchpad, ce qui était une idée révolutionnaire : le programmeur pouvait construire des programmes graphiques avec un stylo lumineux, utilisé directement sur un écran vidéo (Sutherland, 1964).

Ce type d?innovation était typique des débuts des sciences informatiques. Au cours des décennies suivantes, ce domaine a muri et s?est renforcé autour d?un ensemble de principes universellement accepté ? le courant dominant des langages de programmation, stimulé par les besoins économiques des grandes entreprises.

La combinaison des systèmes Unix et du langage C a eu un impact important sur l?ensemble de l?industrie. Les langages compilés sont devenus la norme. Pour résoudre le problème de l?accroissement permanente des code-sources, des stratégies supplémentaires telles que la programmation objet ont été adoptées. Les grands environnements de développement intégré et les débogueurs sont devenus des nécessités pour permettre des développements à grande échelle.

Fait intéressant, une REPL Lisp possède la plupart des caractéristiques d?une grande IDE contemporaine : elle peut être utilisée pour analyser et modifier le programme en cours d?exécution, ainsi que pour inspecter son état.

## 1.2. Regard sur les tendances actuelles

Aujourd?hui, de nouveaux langages de programmation apparaissent à une vitesse sans précédent. Une raison importante à cela est probablement l?apparition d?une infrastructure open-source qui offre la possibilité à une petite équipe ou même un seul individu de créer un nouveau langage. LLVM (Lattner & Adve, 2004) est un outil de génération de code sophistiqué qui peut faire abstraction des détails de l?architecture de la machine, du système d?exploitation et du mode de compilation, standard ou à la volée. Les machines virtuelles telles que la machine virtuelle Java, Common Language Runtime et, plus récemment, les compilateurs JavaScript sophistiqués présents dans les navigateurs Web sont des cibles de compilation relativement accessibles.

De plus, l?interaction et les REPL connaissent un nouvel essor. Les discussions de Bret Victor sur l?informatique et la programmation ont influé sur la réintroduction et la vulgarisation d?une histoire alternative de l?informatique, incluant notamment les idées pédagogiques de Seymour Papert (1980) et de Chris Hancock (2003). Les environnements de développement contemporains tels que la Light Table de Chris Granger et Swift, conçus par Chris Lattner chez Apple, mettent l?accent sur le développement exploratoire, itératif et progressif, citant Victor comme source d?inspiration.

# 1.3. Aperçu des langages pour la musique

Au cours des dernières décennies du xx$^{e}$ siècle, C et C++ sont devenus de fait les langages standard pour le traitement du signal. Les nouveaux langages ont tardé à rattraper leur retard dans le domaine du traitement du signal, en grande partie en raison de nombreuses nouvelles abstractions ayant eu un impact négatif sur les performances du traitement numérique à large bande passante. Tout comme les scientifiques et les ingénieurs ont bataillé avec le code machine dans l?ère pré-Fortran, les musiciens trouvent souvent que C et C++ sont des langages intimidants.

Pour surmonter cet obstacle, des ensembles de logiciels académiques et commerciaux ont été élaborés pour permettre aux musiciens de programmer. Les premiers comprennent des langages tels que CSound (Boulanger, 2000), Pure Data (Puckette, 1996), SuperCollider (McCartney, 2002), Nyquist (Dannenberg, 1997) et Faust (Orlarey *et al.*, 2009). Les suivants comprennent des dispositifs comme Max et Reaktor. Tous ces éléments visent à simplifier le processus de construction de processeurs de signaux afin que ces derniers soient accessibles à des musiciens.

Beaucoup de gens sont attirés par les systèmes de programmation visuelle qui offrent une métaphore de « patch » (Puckette, 1988) : le programmeur dispose d?un ensemble de n?uds ou de boîtes avec lesquels il peut produire un graphique de signal ou un patch. Les n?uds représentent des unités fonctionnelles de base ou des unités génératrices, telles que des oscillateurs, des filtres, des primitives de contrôle ou arithmétiques. Les graphiques décrivent le flux du signal passant d?une unité à l?autre.

## 1.3.1. Max

Max, langage développé par Cycling?74, est un excellent exemple d?une telle métaphore, et probablement le langage de programmation musicale ayant eu le plus de succès, puisqu?il a réussi à attirer de nombreux musiciens vers la programmation. Max correspond à un idéal de faible abstraction : il existe une relation individuelle entre les unités fonctionnelles du programme et le graphique affiché à l?écran. Cet élément concret, ainsi que les excellents matériaux d?apprentissage et de documentation qui sont généralement proposés, ont été les clés du succès de Max.

Cependant, l?ensemble des progrès dans les sciences de l?informatique repose sur l?abstraction. Je postule que le manque d?abstraction finit généralement par limiter le programmeur et par réduire les avantages de travailler sur un ordinateur. Si l?abstraction rend la programmation plus difficile, c?est que nous devons utiliser de mauvaises abstractions.

Diverses anecdotes relatives à la frustration d?utilisateurs expérimentés et de programmeurs expérimentés avec Max, ainsi que les activités de recherche autour des langages de programmation musicale suggèrent que le problème n?est pas encore résolu de manière satisfaisante.

Quelques systèmes de programmation musicale bien connus introduisent diverses abstractions qui permettent de surmonter certaines des limites des graphes construits avec des générateurs. La possibilité de changer les échelles des abstractions est encore plus importante lorsque l?on considère l?enseignement : maîtriser une abstraction de plus en plus sophistiquée est à la fois le défi et la récompense de l?apprentissage de la programmation.

### 1.3.2. SuperCollider

SuperCollider (McCartney, 2002) est construit comme un programme client/serveur, où le serveur héberge un graphe de flux de données efficace construit à partir de générateurs. Les générateurs sont écrits en C/C++. Ce qui caractérise SuperCollider est le langage utilisé pour générer et contrôler le graphe DSP. Il est dérivé de SmallTalk, un puissant langage généraliste orienté objet. SuperCollider offre une solution élégante à la programmation musicale au niveau des partitions et des instruments, mais il ne permet pas d?afficher le niveau des unités génératrices, le noyau du traitement du signal, à ses utilisateurs.

### 1.3.3 Nyquist

Nyquist (Dannenberg, 1997), tout comme SuperCollider, aborde le problème de la description des instruments et des partitions, à partir d?unités génératrices, de façon très efficace, mais avec un paradigme différent : il est basé sur la composition des signaux dans un style déclaratif et fonctionnel. Nyquist est particulièrement adapté à décrire les échéances et les séquences. Les primitives de traitement du signal qu?il utilise sont similaires à celles de SuperCollider, écrites en C, et sont également opaques pour les utilisateurs de cet environnement.

### 1.3.4. Faust

Contrairement aux langages mentionnés précédemment, Faust (Orlarey *et al.*, 2009) est dédié au problème de la programmation de processeurs de signaux directement au niveau des unités génératrices. Faust est un langage par bloc-diagramme qui obéit au paradigme fonctionnel, permettant au programmeur de composer des fonctions arbitrairement complexes à partir de primitives fondamentales. Faust est conçu comme un compilateur traditionnel, générant un binaire exécutable à partir d?un fichier source textuel.

## 2. Programmer pour l?apprentissage et l?exploration

Idéalement, un langage pour la musique devrait être aussi abordable que Max, aussi efficace sur le plan informatique que Faust, et aussi efficace que Nyquist ou SuperCollider pour faire varier les niveaux d?abstraction. Ces souhaits créent des conflits évidents : Max échappe à l?abstraction pour offrir à ses patchs graphiques une impression rassurante de matérialité. Il n?est pas immédiatement évident de voir comment l?accroissement de l?abstraction et la programmation visuelle pourraient s?accorder. De plus, une abstraction plus élevée dans la programmation est traditionnellement associée à des coûts excessifs d?exécution et à une efficacité de calcul plus faible.

Plusieurs auteurs considèrent que le modèle basé sur des unités génératrices est une source de problème (voir Brandt 2002 ; Nishino 2016). De nombreuses frustrations proviennent en effet du fait que l?utilisateur est limité à composer des unités relativement monolithiques fournies par l?environnement de programmation, sans moyen de dupliquer ou de modifier leur comportement à la base. Ceci est également un inconvénient pour l?apprenant : le code de traitement du signal réel ne peut pas être étudié ni manipulé. La situation est différente avec des langages de programmation de systèmes généraux tels que C++, qui peuvent fonctionner à un très bas niveau, reflétant les fonctionnalités de base du CPU, mais sont également adaptés pour décrire des abstractions arbitrairement compliquées.

## 2.1. Des arguments pour la programmation graphique

Les langages graphiques sont souvent réputés pour être facilement accessibles. Selon Mikael Laurson (1996) :

*Lorsqu?on affiche graphiquement à l?écran du matériel musical, on peut combiner idéalement les avantages de l?affichage des partitions traditionnelles avec les nouvelles possibilités dynamiques de l?ordinateur.*

Brad Myers (1986) indique que le cerveau humain est capable de comprendre des données multidimensionnelles, comme la topologie d?un diagramme de flux de signaux. Les langages textuels sont généralement organisés comme des séquences linéaires, même si celles-ci décrivent des abstractions multidimensionnelles.

En outre, le graphique de flux de signaux basé sur les patchs a une ressemblance skeuomorphique (1) avec le monde physique, où des câblages interconnectent les équipements audio. Cela donne une connaissance instantanée à la plupart des technologues en musique.

### 2.1.1. Syntaxe graphique et flux de signaux

Les graphes de flux de signaux sont courants dans la programmation musicale ? même les langages de musique non graphiques utilisent souvent des sémantiques de graphes. Ce qui est intéressant, c?est que presque tous les langages formels peuvent être décrits comme des *arbres syntaxiques abstraits* ou AST. Un tel langage peut être transformé en langage graphique en représentant l?arborescence de cette syntaxe abstraite sous une forme graphique, au lieu de l?analyser à partir d?une représentation textuelle. La question est alors de savoir si l?arborescence d?une syntaxe abstraite d?un langage particulier est utile en tant qu?interface de programmation.

Si on considère les langages impératifs et les paradigmes fonctionnels : une des caractéristiques des langages fonctionnels est que l?arbre syntaxique abstrait coïncide avec le flux de données ? les données progressent des feuilles de l?arbre syntaxique vers sa tige.

En revanche, la sémantique impérative, en particulier avec les affectations, introduit des flux de données qui ne sont pas évidents avec les AST. Les données peuvent circuler entre les n?uds enfants dans les AST. Les fonctions pures, d?autre part, produisent des données uniquement via des valeurs qui sont retournées. Cela rend les AST fonctionnels particulièrement adaptés pour la programmation graphique : le graphe du signal est dirigé et il est facile de suivre graphiquement les flux de données.

Cependant, un programme fonctionnel peut décrire le flux de données de manière indirecte. Si on considère Faust : la syntaxe décrit la composition des fonctions plutôt que le flux de signaux. Par essence, les données qui circulent dans les AST sont composées par des fonctions. Le compilateur appliquera implicitement la fonction composite résultante en tant que processeur audio. En raison de cette indirection, le flux de signal audio ne coïncide pas avec les AST Faust et une visualisation directe des AST serait donc probablement une interface de programmation confuse.

## 2.2. Des arguments pour une programmation interactive

Les évolutions rapides et les capacités du temps réel permettent une approche exploratoire, spontanée et créative de la programmation, qui me semble être particulièrement importante pour la programmation musicale. De même, je crois que les fonctionnalités émergentes sont également en forte synergie avec l?apprentissage lui-même. David Griffith attribue la description adéquate de cette relation à Julian Rohrhuber dans son rapport du séminaire Dagstuhl pour « une collaboration et un apprentissage par codage direct » (2013) : « Vous pouvez utiliser le son pour entendre ce que vous faites ? le son est l?exécution temporelle du code ».

Dans le reste de cette section, je présente un certain nombre d?interactions entre le programmeur et son

programme qui paraissent bénéfique pour la créativité exploratoire spontanée et pour l?enseignement.

### 2.2.1. Analyse

Analyser est le fait de diviser un problème selon ses parties constitutives pour les examiner séparément. Une telle approche est essentielle pour résoudre des problèmes de programmation difficiles et l?environnement de programmation devrait être capable de proposer une méthode de travail selon laquelle des sous-composants simples sont développés et ensuite combinés en composants toujours plus performants.

### 2.2.2. Conversation

Une fois que les parties constitutives d?un programme sont construites et remplissent chacune un rôle unique, le programmeur doit pouvoir converser avec eux : poser des questions et recevoir des réponses. Cela favorise la compréhension de la façon dont les composantes du programme répondent à différents stimuli.

### 2.2.3. Compréhension

La compréhension de la façon dont les algorithmes et les programmes fonctionnent peut résulter d?une observation multimodale du programme, de son exécution et de son état. L?instrumentation visuelle et auditive de plusieurs composants à la fois peut aider à comprendre intuitivement comment le comportement évolue dans des systèmes complexes.

### 2.2.4. Découverte

L?environnement de programmation doit permettre au programmeur de parcourir le programme d?une manière non linéaire, de suivre les symboles jusqu?à leurs définitions et leur documentation, directement à partir du code source. Cela permet au programmeur à la fois de comprendre comment utiliser un composant et d?apprendre comment celui-ci a été créé.

## 3. La méthodologie de programmation de Kronos

Kronos fait l?objet de ma thèse de doctorat. C?est un langage conçu pour les musiciens qui souhaitent faire du traitement du signal. Dans cette section, je vais examiner ses concepts en réponse aux problèmes soulevés dans les sections précédentes, ainsi que mon expérience dans l?enseignement du traitement du signal à des étudiants en musique.

Il y a certains concepts qui, d?après mon expérience, sont responsables de la plupart des problèmes chez les apprenants :

- les affectations mémoire, les pointeurs et les états ;
- les annotations des types ;
- la planification *(scheduling)*.

Kronos vise à minimiser, sinon à supprimer, toutes ces préoccupations pour le programmeur. Faust

fournit un exemple de la façon dont les processeurs de signaux peuvent être décrits de manière compacte et efficace sans recourir à la gestion bas niveau de la mémoire, ni des pointeurs, ni des états. Faust accomplit cela en adoptant un système de typage (intentionnellement) trivial et en appliquant un graphique de signal statique.

Au lieu de variables de données explicites, l?état de Faust est modélisé par une boucle avec un retard d?une unité ou par des chemins de signal retardés. En contrepartie, nous avons un nouveau problème pour la compréhension des programmes cycliques. Heureusement, ils ne sont pas complètement étrangers aux technologues de musique qui connaissent les effets de retard et les boucles audio.

Kronos prend appui sur plusieurs points issus des technologies Faust en raison des objectifs de conception suivants :

- une aptitude à la programmation visuelle ;
- le polymorphisme ? le modèle de programmation générique ;
- l?interaction avec les flux d?événements (MIDI, OSC).

# 3.1. Programmation graphique

Comme nous l?avons indiqué précédemment dans la section « des arguments pour la programmation graphique », en adoptant une relation directe entre le flux de signal et l?arbre de syntaxe abstraite, l?AST devient une surface utile et adaptée pour la programmation. Bien que Kronos puisse modéliser la composition de fonctions ou même l?algèbre de type bloc-diagramme comme dans Faust, par défaut, le système est plutôt conçu pour propager des valeurs à l?échantillon près.

# 3.2. Polymorphisme et Généricités

Un problème commun dans le traitement du signal est le problème combinatoire résultant du fait que des opérations similaires sont souhaitées pour une variété de configurations, telles que la résolution d?amplitude ou le nombre de canaux en sortie. En adoptant un système F? (Barendregt, 1991) de lambda calcul, Kronos permet l?utilisation de fonctions polymorphes ? fonctions qui modifient leur comportement en fonction du type de données de leurs paramètres d?entrée. Des résultats similaires pourraient, en théorie, être réalisés avec l?extension de réécriture de termes de Faust (Gräf, 2010), mais le polymorphisme est omniprésent dans Kronos et en constitue une pierre angulaire.

En gardant à l?esprit l?objectif de toucher des musiciens plutôt que des informaticiens, il nous faut faire attention en approfondissant la théorie des types, les mathématiques et l?informatique. J?ai conçu le système de type Kronos pour qu?il soit puissant mais surtout transparent. Il est inspiré du métacompilateur de templates C++ (C++). En C++, les modèles sont des classes et des fonctions polymorphes qui peuvent être *monomorphisées* à la demande ? spécialisées pour un certain ensemble de types sources. Cela permet au code source d?être largement exempt d?annotations de type, alors que le compilateur peut encore utiliser pleinement le typage statique et tous les avantages liés aux performances. Il existe un corollaire intéressant pour les langages typés de manière dynamique : les fonctions génériques peuvent être très flexibles, mais elles ne disposent pas de garde-fous contre les erreurs de type, ce qu?offre l?utilisation du typage statique.

Comme C++, Kronos emploie une dérivation de type « vers l?avant », au lieu des schémas d?inférence de type plus avancés tels que ceux de Hindley-Milner (Heeren *et al.*, 2002). La dérivation de type et la monomorphisation sont effectivement un passage informatique calculé par type dynamique sur l?ensemble du programme pendant lequel les types statiques sont établis. L?avantage de ce schéma est que les annotations manuelles de type ne sont jamais requises, mais peuvent être utilisées si un contrôle fin par rapport au comportement polymorphe est requis ? comme le choix d?un algorithme efficace avec des hypothèses basées sur les types de données d?entrée.

L?inconvénient majeur de la dérivation de type est lié à la manière dont il doit être exécuté sur

l?intégralité du programme utilisateur et de toutes ses dépendances, et comment le nombre de monomorphisations et la taille du code qui en résulte peuvent croître rapidement dans des programmes plus importants. Ceci est généralement gérable, car les noyaux de traitement du signal ont tendance à être de taille limitée, mais dans les systèmes plus vastes, une certaine compréhension de la manière dont le système de type se comporte peut devenir essentielle. Pour une discussion plus approfondie du système de type et de ses divers compromis, je renvoie le lecteur aux publications antérieures (Norilo, 2013, 2016).

## 3.3. Flux d?événements

Kronos modélise tous les signaux en tant que paires temps-valeur, qui définissent une fonction évaluée de façon continue avec des étapes discrètes. Le modèle de signal devient significatif lorsqu?un traitement multi-rate est utilisé, combinant des signaux qui se mettent à jour à différents moments. Dans la pratique, cela pourrait signifier des signaux complètement asynchrones, comme les événements de contrôle OSC qui influent sur le traitement audio, ou des subdivisions synchrones mais différentes d?une horloge principale, telles que les signaux à vitesses audio et ceux dédiés au contrôle. Le compilateur Kronos génère un état implicite pour le comportement en échantillonnage et maintien (*sample&hold*) lorsque différentes horloges de signal sont combinées aux entrées d?une fonction. Un rééchantillonnage et une interpolation plus sophistiqués peuvent être mis en ?uvre sur cette base.

Le compilateur Kronos est conçu pour générer des codes basés sur ces sémantiques sans aucune surcharge de calcul. Pour cette raison, le système peut envoyer des échantillons audio individuels en tant que paires de temps-valeur et encore générer un code machine qui ressemble à celui généré à partir du code C avec un traitement audio classique avec des tampons mémoire. Le bénéfice de ce schéma est que différents types de signaux, tels que MIDI et OSC, ainsi que des taux de contrôle arbitraires basés sur l?horloge audio peuvent être combinés librement.

## 3.4. Outillage

Jusqu?à présent, j?ai décrit la conception des langages et comment cela peut aider à surmonter certains problèmes que présentent les systèmes précédents de traitement du signal. Cependant, une grande partie du défi décrit précédemment dans la section « Programmation pour l?apprentissage et la découverte » se trouve dans l?outillage construit autour du langage.

### 3.4.1. REPL

Alors que la plupart des nouveaux langages de programmation qui apparaissent ont des REPL interactifs, tels que ceux initiés par Lisp, l?utilisation d?un REPL pour le traitement du signal est encore relativement rare. Une exception notable est xtlang de Andrew Sorensen (2010), qui fait une utilisation importante de la programmation dynamique et des REPL.

Le développement orienté REPL peut permettre de nombreuses techniques exploratoires tout en restant technologiquement remarquablement simple. Cependant, un REPL fonctionnel nécessite un langage de programmation conçu autour du développement incrémental et de la programmation conversationnelle. Dans Kronos, le défi d?un REPL provient de la dérivation des types : un changement local mineur peut avoir des ramifications globales dues à la propagation du type. Pour cette raison, le REPL de Kronos traque les fonctions et leurs dépendances.

### 3.4.2. Un REPL graphique

Le comportement polymorphe de Kronos peut être étendu à « l?impression » dans la lecture-eval-print-loop. En surchargeant l?opération par type, une REPL visuelle peut être créée qui affiche des textes riches et des images en plus du texte. Actuellement, un prototype REPL existe sur le Web pour Kronos qui peut afficher des informations graphiques codées en HTML. Étant donné que toutes

les données de Kronos se composent de paires de valeurs temporelles, ces informations visuelles peuvent être animées et réactives : imaginez le graphe donnant la réponse fréquentielle d?un filtre qui réponde à ses contrôles.

### 3.4.3. Des patchs interactifs

À plus long terme, mon objectif est de combiner l?élégance et la simplicité d?un REPL avec la familiarité d?une interface de patchs. Une telle formulation pourrait combiner les avantages de la programmation graphique et multidimensionnelle avec le modèle conversationnel d?une REPL traditionnelle. Une fois que la zone d?esquisse du programmeur devient un canevas plutôt qu?une entrée par des lignes de codes, d?autres interactions deviennent possibles : faire suivre les fonctions par leurs définitions, affichage de la documentation et des exemples en ligne lorsque le programmeur se concentre sur certaines sections du patch ou autorisations de plusieurs visualisations simultanées de l?état du programme en cours d?exécution.

# Conclusion

Dans ce texte j?ai présenté mes réflexions au sujet de la conception de langages de programmation pour les musiciens, ainsi que des considérations techniques et philosophiques autour du projet de Kronos. L?un des points clés concerne la combinaison d?une méthode de programmation graphique conviviale pour les débutants avec un langage qui peut se développer jusqu?à des abstractions de niveaux élevés. Une programmation fonctionnelle par flux de données et une correspondance directe entre la syntaxe de la langue et le graphe de signal est une solution possible.

Pour obtenir un langage suffisamment performant avec un haut degré d?abstraction, Kronos utilise les principes de la méta-programmation, les systèmes de typage et la monomorphisation. La combinaison d?un tel langage et d?un compilateur avec un REPL fournit une méthode de développement conversationnel exploratoire pour le traitement du signal, qui est radicalement différente de la méthodologie conventionnelle basée sur C.

En tant que projet de recherche, Kronos est constitué d?un dispositif technologique qui vise à remplir les conditions requises pour constituer un environnement idéal pour l?apprentissage de la programmation et du traitement du signal ; il reste encore beaucoup à faire pour évaluer l?expérience des utilisateurs en matière d?interaction avec cet environnement, en particulier à travers l?utilisation de surfaces de programmation graphiques, et des améliorations restent à développer.

---

1. Un skeuomorphisme est un ornement dans un modèle dérivé qui ressemble à des structures qui étaient nécessaires dans le modèle original.

---

**Pour citer ce document:**
Vesa Norilo, Â« Le traitement du signal pour les musiciens 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 14/06/2018
URL: http://revues.mshparisnord.org/rfim/index.php?id=534