Apprendre et comprendre l'improvisation musicale

Mémoire

Alexandre Casanova–Franger Gauthier Lamarque
Paul Simorre Lucas Vivas

5 avril 2018

Table des matières

1 Introduction

Ce projet a été réalisé par quatre étudiants de première année de Master Informatique de l'Université de Bordeaux (parcours Génie Logiciel), dans le cadre de l'UE Projet de Programmation.

1.1 Un outil pour assister l'improvisation musicale

Le projet a été proposé par Myriam Desainte-Catherine, chercheur au Studio de Création et de Recherche en Informatique et Musiques Expérimentales (SCRIME) et enseignante à l'ENSEIRB-MATMECA, et Yacine Amarouchene, physicien du Laboratoire d'Onde et Matière d'Aquitaine (LOMA). Il a été pensé comme un outil d'aide à "l'improvisation musicale", un dispositif permettant d'aider un groupe de musiciens à improviser mais permettant aussi de mieux comprendre la nature de l'improvisation en musique.

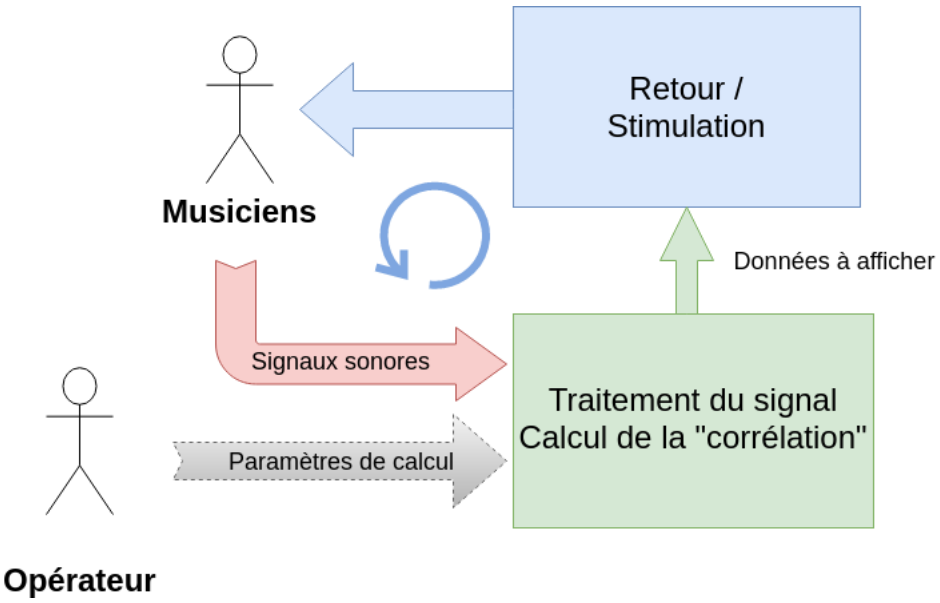

FIGURE 1 – Principe général du projet

Le schéma ci-dessus résume le principe général du projet. Les musiciens jouent librement, et leur musique une fois transformée en signal discret, pourra alors être traitée pour que se dégage en sortie un résultat que l'on appelle la "corrélation" (voir la section suivante). Ensuite, il s'agit d'établir un retour (visuel, audio) pour que les musiciens puissent être stimulés. L'objectif de cet outil est de reproduire la boucle décrite sur le schéma.

1.2 Improvisation et corrélation

Il est essentiel de comprendre ces deux notions afin de bien situer l'intérêt du projet.

Lors d'une improvisation en groupe, des musiciens jouent sans règle établie ; ils cherchent alors à fournir à leur auditoire une production cohérente, ou du moins au sein de laquelle chaque musicien joue un rôle dans l'harmonie du morceau en s'accordant avec ses pairs. La relation qui unit le jeu de deux musiciens et évalue la qualité de leur accord porte un nom : c'est la corrélation.

Au sens premier et basique du terme, la corrélation est le rapport réciproque entre deux éléments. En statistiques, on parle souvent de corrélation pour mesurer l'intensité de la liaison existant entre deux variables.

En musique, il n'existe pas une mais un nombre indéfini de "fonctions de corrélation" potentiellement existantes, dont la complexité et les paramètres varient. Pour établir ce "rapport réciproque" entre deux signaux sonores, on pourrait comparer leurs tempos, leurs amplitudes, leurs volumes sonores... "Comprendre" la corrélation et l'improvisation, dans le cadre de ce projet, c'est aussi trouver, inventer la fonction de corrélation qui répond le mieux aux besoins d'un groupe d'improvisateurs. Une bonne fonction de corrélation, dans ce cadre spécifique, est une fonction qui fait en sorte qu'un groupe de musiciens joue un morceau plus satisfaisant lorsque ses membres sont davantage corrélés ("découvrir" la fonction de corrélation relève ici du rôle de l'utilisateur ou du physicien, le logiciel que nous implémentons n'étant qu'un assistant dans cette démarche).

Les traductions graphiques de signaux sonores par transformée de Fourier peuvent jouer le rôle des courbes dont on mesure la corrélation. Tout au long du déroulement de notre projet, nous travaillons uniquement sur des fonctions de corrélation portant sur la comparaison des signaux graphiques obtenus à partir des pistes sonores.

1.3 L'existant

Ce projet a été initié par les clients précédemment cités il y a plus d'un an. Nous sommes le troisième groupe à travailler sur ce projet et développons donc sur la base des travaux réalisés successivement par nos prédécesseurs.

1.3.1 Premiers travaux réalisés sur le projet

Les premiers travaux portant sur ce projet ont été réalisés par un groupe de six étudiants de l'ENSEIRB-MATMECA. Cette équipe est la première à réaliser un programme informatique analysant et comparant des pistes mono-instrumentales pour évaluer leurs corrélations. Ce programme, prenant en entrées des pistes sonores, retourne une matrice graphique prenant les mêmes pistes en abscisses et en ordonnées et retournant pour chaque case une couleur permettant d'évaluer la corrélation existant entre les deux pistes correspondantes grâce à un code couleur précis.

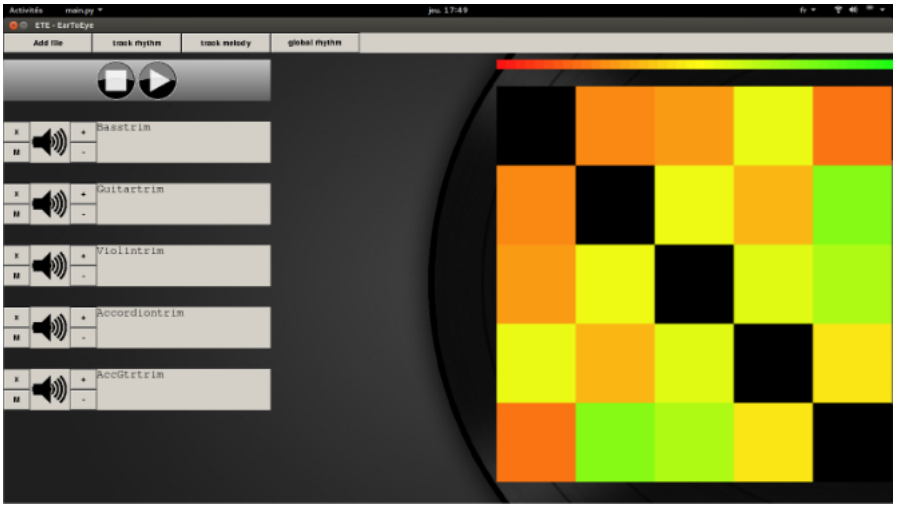

FIGURE 2 – Capture d'écran du projet mené par les étudiants de l'ENSEIRB

Ci-dessus, la matrice obtenue à l'issue d'un test du programme. Plus la couleur d'une case est proche du vert, plus la corrélation entre les deux pistes correspondant au point d'abscisse et au point d'ordonnée est élevée. À l'inverse, une case dont la couleur est proche du rouge indique que les deux pistes évaluées sont décorrélées. La corrélation d'une piste mono-instrumentale avec elle-même, qui donne toujours un résultat maximal, n'est pas calculée, ce qui se traduit sur le résultat graphique ci-dessus par une diagonale de cases noires. La matrice de corrélation est donc systématiquement symétrique.

1.3.2 Le système embarqué BELA

Avant de confier la suite du projet à d'autres étudiants, nos clients ont choisi de changer de support et ont opté pour l'utilisation d'un système embarqué particulier : BELA.[?]

Ce choix s'explique par la mobilité du support, mais également par le fait que BELA est basé sur un système appelé Xenomai Linux, un framework qui ajoute au noyau Linux des éléments permettant de traiter des opérations en temps réel.[?] Cela permet au système de faire du traitement audio en temps réel avec un temps de latence extrêmement faible (100 microsecondes).

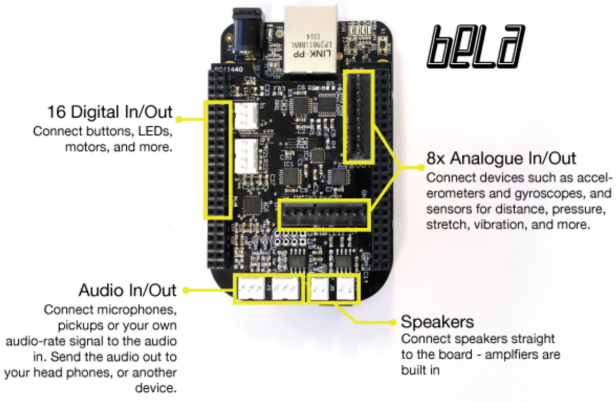

FIGURE 3 – Capture de la documentation proposée par le site officiel de Bela

L'aperçu du dispositif externe de BELA présenté ci-après témoigne notamment de la présence d'entrées analogiques pour connecter des instruments électriques ou enregistreurs sonores. On comprend alors que le rôle de BELA au sein de ce projet est l'utilisation d'un programme en temps réel par un groupe d'improvisateurs, similaire à celui implémenté par les étudiants de l'ENSEIRB, dont les instruments seraient connectés à cet outil. Ce dernier dispose déjà de fonctions de traitement du son codées en C++, mais revêt surtout un intérêt pour les développeurs, qui peuvent librement modifier et enrichir son programme.

Le dispositif externe BELA comprend huit entrées analogiques et deux entrées audio. Son architecture lui permet d'interpréter parallèlement jusqu'à seize fichiers audio numériques supplémentaires.

Le framework de BELA permet l'implémentation de programmes divers par son utilisateur à partir de deux fichiers sources codés en C++ : le "main" exécutant le programme principal et le "render" qui met à jour l'analyse des signaux reçus en entrée.

1.3.3 Un logiciel d'aide à l'improvisation musicale

En juin 2017, Jérémy Lixandre, membre des six étudiants ayant amorcé le projet, poursuit seul ces travaux en s'aidant cette fois de BELA, au cours d'un stage de deux mois au laboratoire du SCRIME de Bordeaux. Il reprend le concept de "matrice de corrélation" mais doit reprendre l'implémentation du logiciel à zéro, puisqu'il abandonne le langage Python utilisé lors des premiers travaux pour passer à la programmation C++ exigée par le code de BELA.

De plus, il relègue les travaux de recherche sur la corrélation et ses formules, des tâches relevant de la physique et des mathématiques, au second plan pour mener un projet essentiellement logiciel. En utilisant le code de BELA, il recrée un outil permettant d'analyser la corrélation de signaux sonores et d'afficher la matrice de corrélation présentée plus haut. Seulement cette fois, les signaux sonores traités proviennent de BELA et peuvent donc être produits par des instruments branchés directement sur le système. Ce nouveau programme permet donc à des musiciens d'avoir un retour visuel en temps réel de leur improvisation, et peut même comparer des pistes mono-instrumentales jouées en temps réel avec des fichiers sonores numériques déjà existant.

1.3.4 Détails sur le programme existant

Le programme développé par Jérémy Lixandre s'intitule `VisualImpro`. Il se veut "générique", a été implémenté de sorte à permettre à des développeurs de l'améliorer et de le modifier facilement, et à des utilisateurs renseignés de modifier certains paramètres et configurations liés aux calculs de la matrice. Son architecture logicielle constitue le socle de notre travail, c'est elle que nous allons devoir ré-organiser et enrichir, afin notamment d'ajouter de nouvelles fonctionnalités au programme.

Le code du programme doit notamment contenir trois fichiers .cpp dont les noms sont `Prepoc*.cpp`, `Coeff*.cpp` et `Color*.cpp` et qui contiennent respectivement la fonction de "pré-traitement" ou traitement du signal en amont, la fonction de calcul du coefficient de corrélation et la fonction associant un coefficient à une couleur. Le programme est construit de sorte à ce que tout fichier respectant ce nommage puisse être ajouté au programme afin de permettre à un utilisateur de choisir la fonction de pré-traitement/calcul de coefficient de corrélation/traduction en couleur de son choix.

- La fonction `Preproc` prend en entrée une matrice de vecteurs représentant les signaux d'entrée. Elle retourne une nouvelle matrice de vecteurs, qui pourra présenter des échantillons de plus petite taille que la matrice d'entrée par exemple, dans un souci d'optimisation des performances du programme.
- La fonction `Coeff` prend en entrée deux vecteurs (appartenant à la matrice de sortie précédemment abordée) et retourne une valeur comprise entre 0 et 1 et correspondant à la corrélation établie entre les deux signaux traités. On peut imaginer une infinité potentielle de calculs pour donner lieu à une corrélation dans le cadre de ce programme ; il s'agit de calculs relativement arbitraires qui seront décrits plus en détail dans la suite de ce mémoire.
- La fonction `Color` prend le coefficient précédemment obtenu pour entrée et retourne un triplet RGB ; il s'agit d'un objet C++ défini par l'une des classes du programme.

Ces trois fichiers sont répertoriés dans un dossier `process`. Un fichier de configuration permet d'écrire quel fichier choisir pour chacune des trois fonctions.

Le seul fichier du programme VisualImpro imposé par le framework BELA se nomme `render.cpp`. Il contient lui-même trois fonctions :

— La fonction `setup()` initialise et prépare les ressources de traitement du son.
— La fonction `render()` s'appelle de manière régulière et répétée tout au long du processus audio. Elle a pour arguments des buffers contenant les échantillons à traiter.
— La fonction `cleanup()` est appelée à la fin du processus pour libérer les ressources allouées et mettre fin à certaines tâches.

Une structure `AuxiliaryTask` est mise à la disposition du programmeur et est destinée à exécuter de façon parallèle du code spécifié en amont et considéré comme trop coûteux (dans le programme, il s'agit du traitement des données), afin de soulager l'éxécution de `render.cpp`.

Le fichier `main.cpp` lance le programme. Le fichier de configuration se nomme `config.cfg`, placé dans le dossier principal `VisualImpro`. L'utilisateur peut y entrer les noms des fonctions de *pre-processing*/calcul de coefficient/calcul de triplet RGB de son choix. D'autres configurations purement relatives au traitement de l'audio peuvent être modifiées dans le fichier render.cpp.

1.3.5 Motivation, intérêt et avenir du programme VisualImpro

Le programme a été développé dans le but de fournir une rétrospective visuelle en temps réel à des musiciens jouant en même temps et censée évaluer leur improvisation. La fonction de corrélation, le critère de cette évaluation, doit être modifiable selon les objectifs de l'utilisateur, car aucune science exacte ne saurait vraiment évaluer la qualité de l'harmonie existant entre les jeux de deux musiciens. Plutôt que de leur indiquer la qualité de leur improvisation comme elle pourrait le laisser penser, la matrice graphique doit informer les musiciens sur la nature même de l'improvisation. En observant la matrice tout en jouant pour une configuration du logiciel donné, les musiciens pourraient non seulement établir des liens entre l'évaluation de la corrélation entre leurs jeux respectifs et le son qu'ils produisent, mais également comprendre le fonctionnement du logiciel lui-même, et en s'adaptant progressivement pour améliorer ces indices de corrélation, déterminer si la configuration choisie leur convient et produit un résultat agréable à l'oreille dans leur façon de jouer.

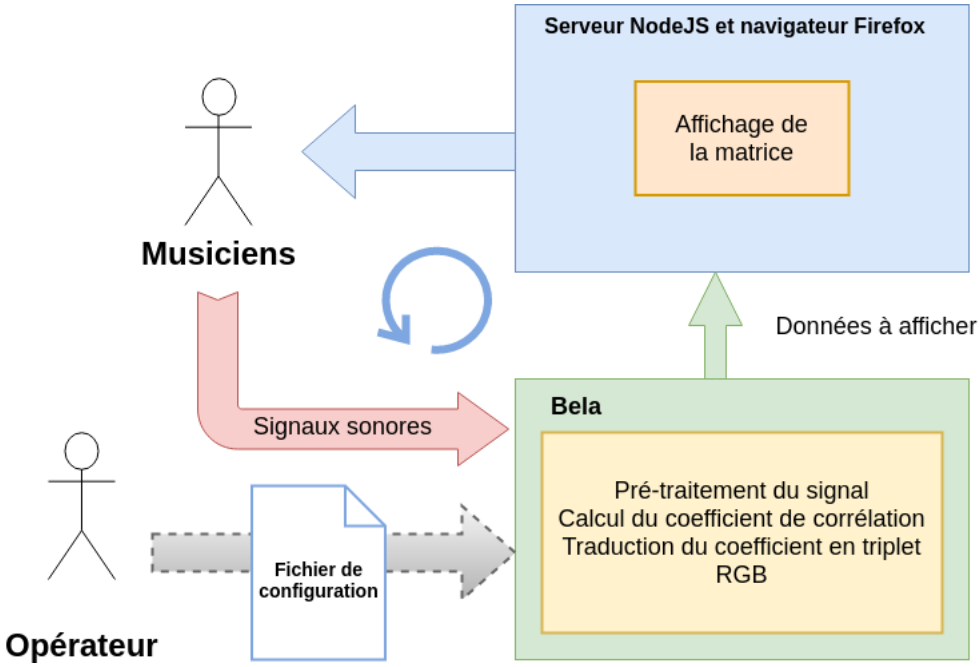

FIGURE 4 – Schéma global du dispositif de VisualImpro

Ci-dessus, un schéma global du fonctionnement du dispositif. Les musiciens jouent un morceau traité par Bela qui, via l'intermédiaire d'une machine, affiche sur le navigateur web Mozilla Firefox la matrice graphique de corrélations censée assister le jeu des musiciens.

Différentes pistes ont été proposées par Jérémy dans son rapport pour améliorer le logiciel produit : implémenter un retour sonore indiquant aux musiciens la qualité des corrélations en fonction du coefficient calculé à la place du retour visuel, ajouter des effets audio aux pistes sonores en amont du calcul de corrélation...

2 Analyse des besoins

2.1 Besoins fonctionnels

L'objectif principal de notre projet est l'implémentation d'un retour sonore sur la sortie audio dont dispose le système embarqué. Grâce à la technologie de Bela, on devrait être en mesure de remplacer ou compléter le retour visuel par une évaluation audio du jeu des musiciens.

Ce qui motive ce besoin particulier est l'intérêt technologique que représente l'utilisation de Bela pour produire un résultat sonore dépendant de nos traitements de calcul mais également de fournir une assistance aux musiciens supposée plus efficace qu'un simple affichage visuel. En effet, lorsqu'un groupe tente d'improviser ensemble, il ne doit pas être évident pour ses membres de se focaliser sur un code couleur ou un affichage visuel dynamique pour adapter et réviser leurs jeux. Ce sur quoi se basent usuellement des musiciens tentant de s'accorder les uns avec les autres sans partition ou chef d'orchestre, c'est bien évidemment le son, celui que produisent les autres. Avec l'implémentation de cette sortie sonore, qui serait une version modifiée en temps réel du morceau joué en entrée du système, on tente de dénaturer cette logique et d'apporter aux musiciens une nouvelle façon d'improviser qui, selon les configurations du logiciel choisies, pourra se révéler plus efficace. Ce nouvel outil mérite quelques éclaircissements quant à son fonctionnement et son intérêt global.

Le retour sonore dont il est question est une version modifiée du morceau joué en temps réel. Ce morceau est composé du même nombre de pistes instrumentales que celui interprété en entrée du système embarqué, mais chacune de ces pistes est préalablement modifiée en terme de niveau sonore en fonction du coefficient de corrélation. Une interface de configuration dans le logiciel permettrait alors à l'utilisateur de définir une "consigne", une loi décrivant quelles pistes doivent être augmentées en niveau sonore par rapport aux autres dans le retour et selon quels critères. L'exemple de configuration qui nous a semblé le plus judicieux est le suivant : les paires de musiciens les plus corrélées entre elles seront plus augmentées en niveau sonore dans le morceau de retour. Nous avons imaginé d'autres configurations possibles : augmenter en niveau sonore les pistes étant les plus corrélées avec une "piste de référence" ayant pour rôle de "mener" l'improvisation, augmenter en niveau sonore les paires de pistes instrumentales les moins corrélées, augmenter en niveau sonore les pistes dont les sommes des coefficients de corrélation avec toutes les autres sont les plus élevées...

L'intérêt du logiciel a alors légèrement changé, et peut paraître plus difficile à comprendre. Ce retour sonore, qui pourra être combiné ou non avec l'affichage visuel de la matrice, a le rôle nouveau de "provoquer" les musiciens. On les désoriente volontairement, en leur faisant entendre via des casques auditifs un morceau qui n'est pas celui qu'ils sont en train de jouer, mais une version différente, où la musique de chacun est soit augmentée soit diminuée par rapport à celles des autres en terme de niveau sonore. Cela aura pour

conséquence de motiver les musiciens lésés par ce nouveau mixage à redoubler d'efforts pour adapter leurs jeux, pour modifier positivement les couleurs de leurs lignes/colonnes dans la matrice graphique, afin que le logiciel "approuve" leur performance et rehausse leur musique dans le mix de sortie.

Comme vous pouvez le constater ci-dessus, le schéma global de fonctionnement de l'outil n'a pas tellement changé. Cependant, le changement qu'on apporte a une conséquence indéniable sur le jeu des musiciens, puisqu'il les désoriente en trompant leur audition et les force à se concentrer davantage sur la matrice pour comprendre les rouages du logiciel... et à force de plusieurs utilisations de celui-ci sous diverses configurations, pour comprendre les rouages de la notion même d'improvisation.

Un autre besoin fonctionnel indispensable est l'implémentation d'une interface permettant à l'utilisateur de sélectionner ses configurations ; non seulement les fichiers de *pre-processing*/calcul de coefficient/calcul de triplet RGB, mais également la "consigne" déterminant quelles pistes doivent être augmentées ou diminuées dans le retour sonore.

2.2 Besoins non fonctionnels

Au cours de son travail, Jérémy Lixandre a pris soin de rendre génériques les trois fonctions de *pre-processing*, de calcul de coefficient de corrélation et de calcul du triplet RGB. Afin de coller avec l'architecture de Bela et avec ses travaux, nous devrons tâcher de rendre notre fonction de mixage du retour audio générique également. De plus, il faudra veiller à ce que son calcul n'entraîne pas de latence, et que le retour parvienne aux musiciens sans décalage temporel par rapport à leur jeu.

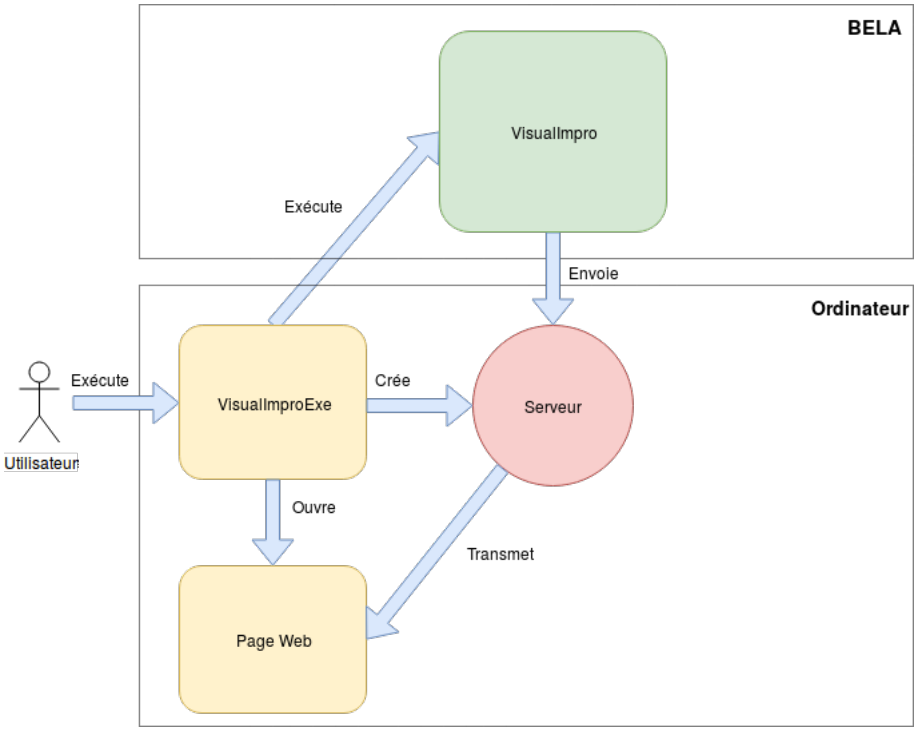

FIGURE 5 – architecture de base

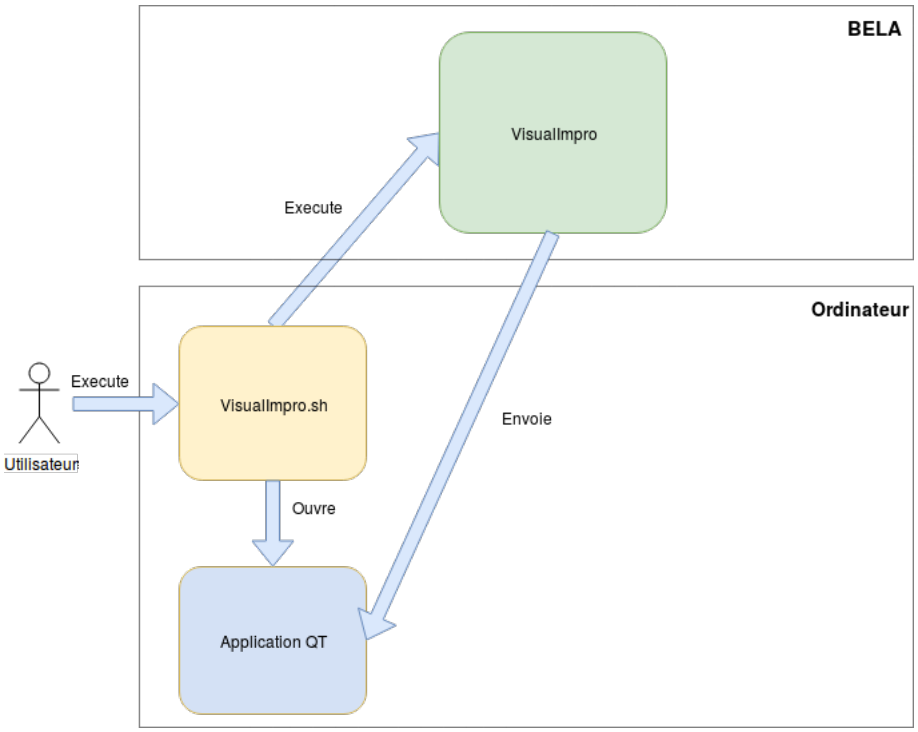

FIGURE 6 – Nouvelle architecture

Le refactoring du code est également nécessaire au projet. En effet, certains choix effectués par notre prédécesseur sur celui-ci nous paraissent discutables. Notamment, le code est peu commenté, il est parfois impossible de distinguer le code pré-existant dans Bela de celui écrit par le programmeur, et le choix des outils destinés à l'affichage de la matrice nous paraissent discutables. En effet, Jérémy Lixandre ne semble pas justifier l'emploi de NodeJS et d'un navigateur web pour l'affichage de la matrice graphique. Nous choisissons alors de modifier ce processus d'affichage : la matrice sera affichée via une simple interface graphique du framework Qt, ce qui nous permet de rassembler l'ensemble du projet sous le langage C++.

Le travail à effectuer sur le code existant se découpe alors en deux parties : le refactoring du code d'une part, et l'implémentation du retour sonore d'autre part.

3 Description détaillée du nouveau logiciel

Dans cette partie, nous allons explorer la mécanique interne du programme VisualImpro, tout d'abord de façon générale, puis nous allons entrer en détail dans les composantes principales de ce programme, afin que l'on puisse avoir une vision d'ensemble claire et précise des fonctionnalités.

3.1 Le point d'entrée du programme

Lorsque le programme est lancé via l'éxécutable, il se passe alors plusieurs choses :

— Les paramètres sont chargés sur BELA à travers un fichier de configuration. Ce fichier contient les différentes entrées (audio, analogiques ou fichiers `.wav`) que Bela va traiter ainsi que les différents traitements qui seront appliqués à ces entrées.

— Si des fichiers `.wav` sont indiqués dans le fichier de configuration, alors ceux-ci seront aussi chargés sur BELA.

Ensuite, le programme VisualImpro de BELA débute. La première tâche effectuée par la fonction `main` est de récupérer le fichier de configuration et de stocker les divers paramètres écrits à l'intérieur de celui-ci. Puis à l'aide des différents paramètres, la deuxième tâche de la fonction `main` va être d'initialiser les éléments correspondants. Enfin, la dernière tâche de la fonction `main` va être de donner la main à la fonction principale de traitement audio à travers l'appel `Bela_startAudio`. Avec cet appel, on passe à la fonction principale qui se situe dans le fichier `render.cpp` et qui porte le nom de `render`.

3.2 La boucle de traitement principale

Avant de passer directement à la fonction `render`, le programme va passer par la fonction `setup` qui va initialiser toutes les structures nécessaires au bon déroulement de la fonction `render`, ainsi qu'un autre élément central du programme, les tâches auxiliaires. Ces tâches auxiliaires, qui font partie des éléments fournis par le framework Bela, permettent au programmeur de créer des segments de code qui seront exécutés en parallèle, ce qui permet d'alléger le travail effectué dans la fonction `render`, et donc d'éviter de créer des latences qui pourraient perturber le retour en temps réel. C'est pour cela que l'on initialise des tâches auxiliaires dans la fonction `setup` en spécifiant le code qui sera éxécuté dans chacune d'entre elles. Parmi les 3 différentes tâches présentes dans le code, seules 2 sont intéressantes : la tâche qui aura pour but de remplir les tampons, et celle qui traitera un tampon rempli. La troisième consiste à appliquer des effets sur un tampon, mais ce n'est pas un aspect du projet qui sera traité.

Par la suite, la fonction principale `render` va être appelée en chaîne, jusqu'à ce qu'un signal lui indique qu'elle peut s'arrêter. Dans cette fonction, il va se passer plusieurs choses :

— Si les différents tampons ne sont pas tous remplis, alors on déclenche la tâche auxiliaire de remplissage,

— Une fois que les tampons sont remplis, on déclenche la tâche auxiliaire de traitement,
— Enfin, on récupère les signaux après traitement et on les envoie sur la sortie audio de Bela.

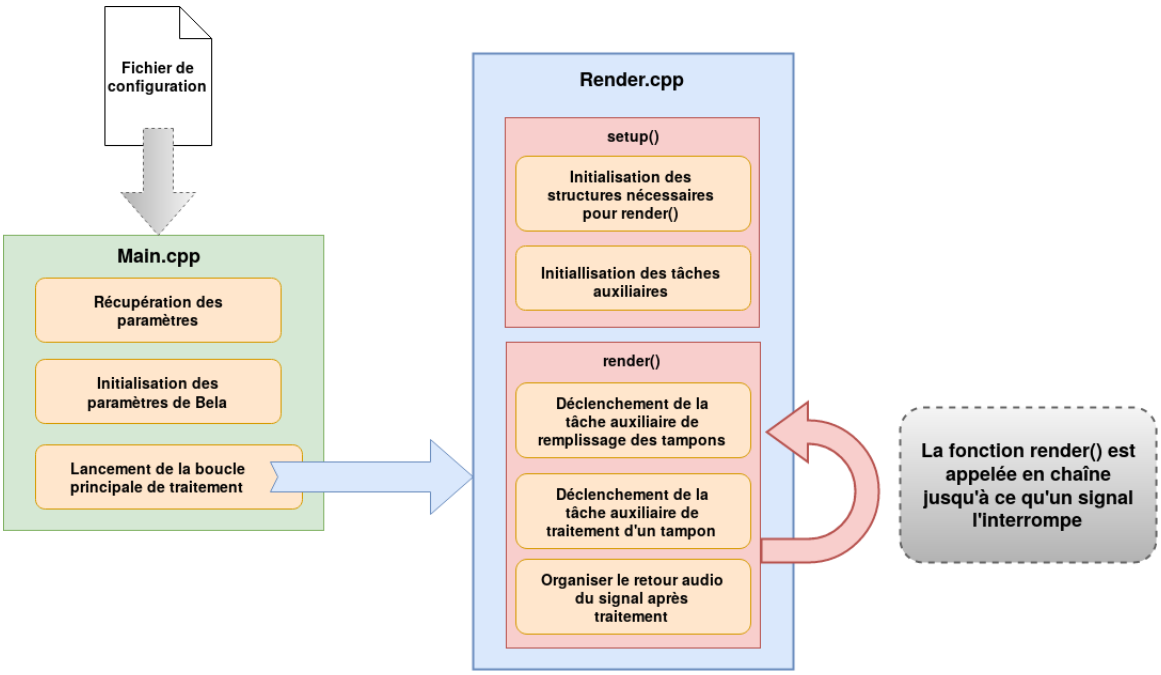

FIGURE 7 – Fonctionnement des fonctions principales de l'outil

3.3 La tâche auxiliaire de traitement

Dans ce mémoire, on ne traitera que la tâche auxiliaire concernant le traitement, car la tâche auxiliaire de remplissage de tampon est réalisée par le code de la classe `SampleStream` qui est une classe qui est fournie sur le site de Bela [**?**]. Son contenu n'est pas voué à être modifié, mais il reste tout de même essentiel à la mécanique générale du programme.

La tâche auxiliaire de traitement du contenu audio est liée à une classe importante de notre outil, la classe `ProcessMultiCorrel`. C'est cette classe qui va prendre un fragment des entrées, stocké dans un tampon, et qui va appliquer différents traitements sur ce tampon. Le traitement va se dérouler en 4 étapes exécutées séquentiellement.

3.3.1 Le pré-traitement

La fonction de pré-traitement va servir à dégager une caractéristique du signal. Parmi les fonctions de pré-traitement dsiponibles, il y a la récupération de l'énergie du signal et

le calcul de l'enveloppe du signal. Le graphique suivant représente l'enveloppe supérieure (en rouge) et l'enveloppe inférieure (en jaune) d'un signal (en bleu). [?]

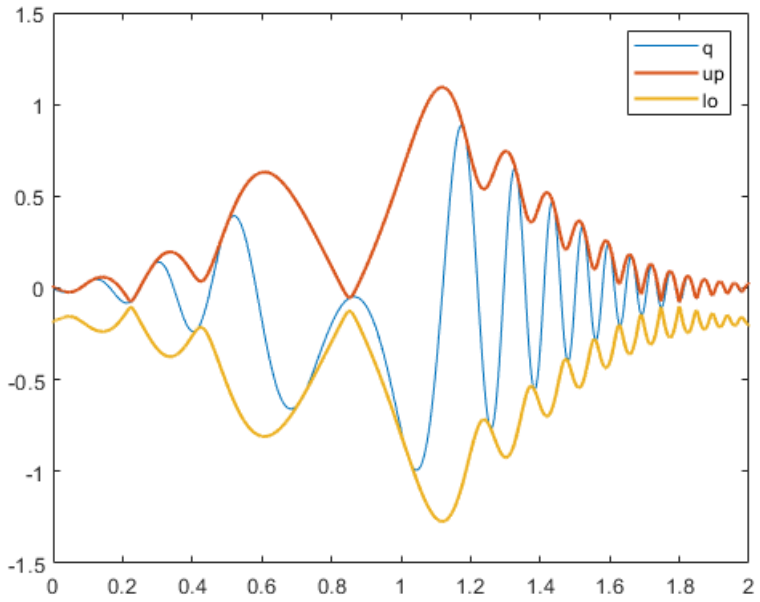

FIGURE 8 – Représentation de l'enveloppe d'un signal

Le principal bénéfice du calcul de l'enveloppe est d'obtenir des valeurs de signaux plus stables.

En ce qui concerne l'énergie du signal, le principe est le suivant : on va découper le tampon en blocs, et pour chaque bloc, on va calculer la moyenne des valeurs du tampon associées au bloc au carré. Cela a pour but d'éliminer les valeurs négatives du signal et de réduire la taille de l'échantillon à traiter, ce qui va réduire les temps de calcul, donc les latences.

3.3.2 Le calcul du coefficient de corrélation

C'est à cette étape que l'on va calculer le coefficient de corrélation. Les coefficients sont calculés pour chaque paire d'entrées, donc pour N entrées nous obtenons une matrice de coefficients $N \times N$ symétrique où la diagonale n'est significative. Comme indiqué dans l'introduction de ce mémoire, la vision que l'on a d'une bonne improvisation est totalement subjective. De ce fait, les fonctions calculant un coefficient de corrélation n'ont pas l'obligation d'être rigoureuses (physiquement et mathématiquement parlant). Pour illustrer notre propos, nous avons mis à disposition une fonction retournant des valeurs

aléatoires (toujours comprises entre 0 et 1 dans un soucis de généricité), ce qui donnera un retour visuel et audio tout aussi aléatoire.

3.3.3 La modification du volume

Après avoir obtenu un coefficient de corrélation (et quelque soit la manière), on va modifier les volumes des entrées en fonction des coefficients obtenus. À ce jour, il y a deux fonctions de mixage des volumes disponibles : une qui va augmenter le volume proportionnellement par rapport à la moyenne des coefficients de corrélation d'une piste par rapport aux autres, et sa fonction de mixage complémentaire.

3.3.4 La traduction en triplet RGB

Pour obtenir un retour graphique pertinent, la matrice de coefficients de corrélation va être associée à une couleur dans une échelle indiquée en paramètre. Les échelles de couleur disponibles à ce jour sont l'échelle noire/blanche et l'échelle rouge/vert. Les coefficients vont alors être traduits en triplets RGB, afin qu'ils puissent être affichés simplement par la suite.

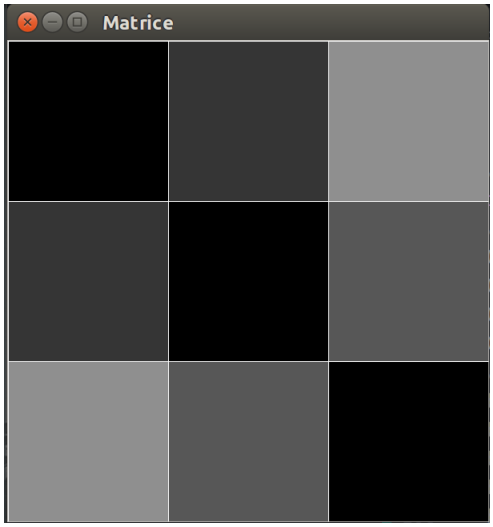

FIGURE 9 – Échelle de couleur Noir/Blanc

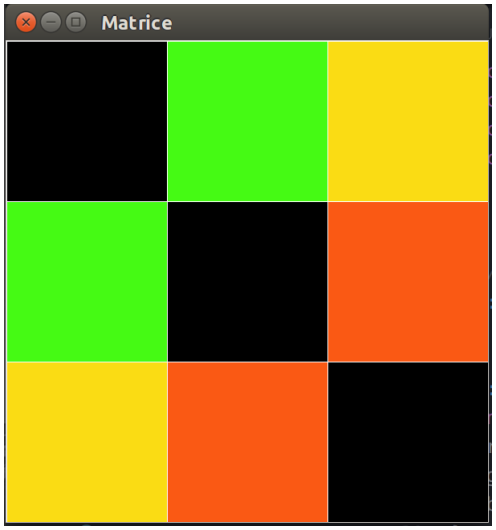

FIGURE 10 – Échelle de couleur Rouge/Vert

Enfin, après avoir réalisé ces 4 étapes, l'ultime travail de la tâche auxiliaire est d'envoyer la matrice de triplets RGB via une connexion TCP vers l'entité s'occupant de l'affichage (serveur NodeJS ou application QT).

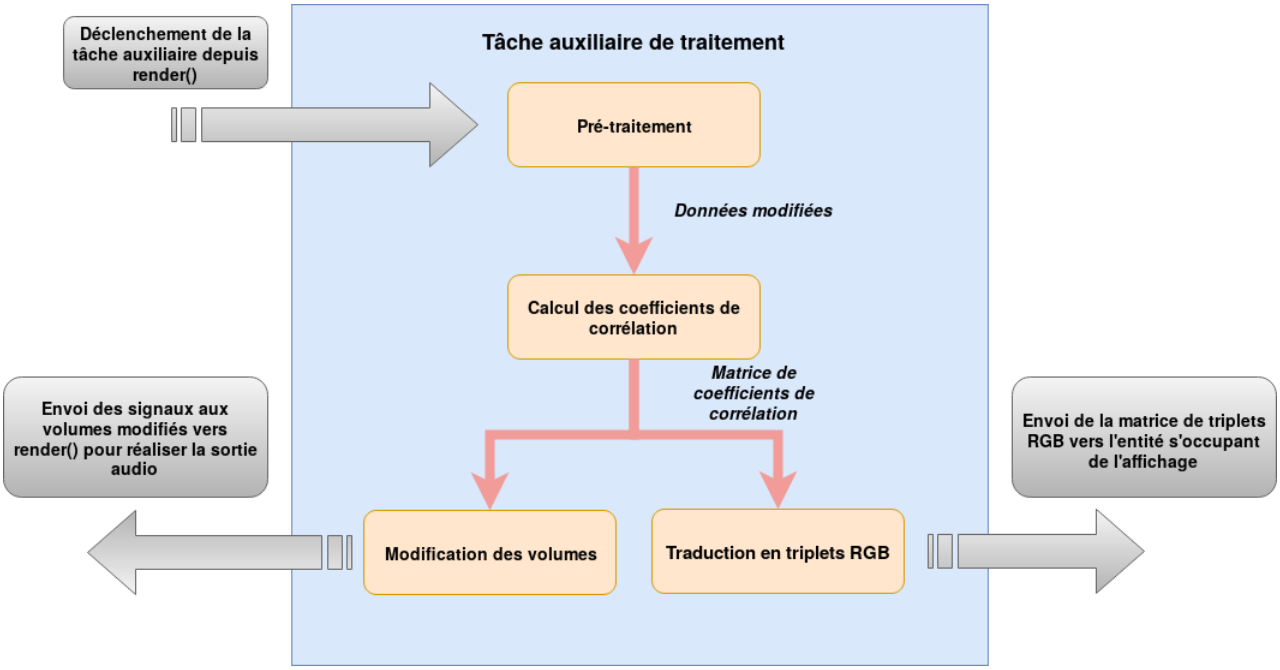

FIGURE 11 – Déroulement de la tâche auxiliaire de traitement

4 Ajout de fonctionnalités

4.1 L'implémentation du retour sonore

Dans le programme de Jérémy Lixandre, la classe `Parser` a pour utilité de récupérer les informations du fichier de configuration `config.cfg`, notamment toutes les fonctions de *processing*. Ce fichier a donc été modifié afin de prendre en compte notre nouvelle fonction de *processing*, la fonction de mixage, qui vient s'ajouter aux trois existantes. Tout comme ces fonctions, elle est générique et doit pouvoir être sélectionnée par l'utilisateur depuis un menu de configuration.

Un fichier dédié à la fonction de mixage a un nom commençant par `Mix`. Il est répertorié dans le dossier `Mix` du dossier `Process`. La fonction de mixage doit prendre une matrice en paramètre ; il s'agit de la matrice retournée par la fonction de calcul du coefficient de corrélation. Elle retourne un vecteur de coefficients : à un instant donné, chaque piste dispose désormais d'un seul coefficient qui doit déterminer la façon dont elle va être diminuée en volume sonore dans le retour audio.

Nous avons implémenté plusieurs fonctions de mixage. La fonction `vector<float> MixMaxCorrelated`, par exemple, renvoie un coefficient égal à la moyenne des coefficients de corrélation de la piste avec toutes les autres pistes. Les instruments les moins corrélés recevront ici un malus sur l'amplitude de leur signal. Tandis que la fonction de traduction du coefficient de corrélation en triplet RGB est dédiée uniquement au retour visuel, celle de mixage est dédiée uniquement au retour sonore.

```
vector<float> MixMaxCorrelated(const Matrix<float>& correlMatrix
    ){

  int row = correlMatrix.getSize();
  int col = correlMatrix.getRow(0).size();

  // initialize the result vector with zeros
  vector<float> meanCorrelations(row, 0.0f);

  // fill the vector with the mean correlation of
  // each instrument with others
  for (int i = 0; i < row; i++) {
    for (int j = 0; j < col; j++) {
      if (i != j)
        meanCorrelations[i] += correlMatrix.getCase(i, j);
    }
    meanCorrelations[i] /= (float)row-1;
  }
  return meanCorrelations;
```

```
}
```

Ci-dessus, l'exemple de la fonction de mixage précédemment cité

Dans la fonction `void parseProcessFunc` appelée dans le fichier `main.cpp`, nous avons dû ajouter l'analyse de la fonction de mixage sur le modèle des analyses des trois autres fonctions de *processing*.

L'architecture de Bela a été conçue pour permettre la synthèse d'un retour sonore dans le corps de la fonction `void render` du fichier éponyme. Grâce au code implémenté par notre prédécesseur au sein de cette fonction, Bela peut synthétiser un retour sonore modifié par notre fonction de mixage.

```
for(unsigned int i=0; i<context->audioOutChannels; i++){
    if(gSampleFactor == STANDARD_SAMPLE_RATE){
        audioWrite(context, 2 * n, i, out);
        audioWrite(context, 2 * n + 1, i, out);
    } else {
        audioWrite(context, n, i, out);
    }
}
```

Ci-dessus, l'implémentation du retour sonore dans le corps de la fonction principale de `render.cpp`. Via la fonction `audioWrite`, la variable `out` récupère un à un les signaux de chaque type de piste sonore (audio/analogique/digitale) multipliés par la moyenne de leurs coefficients de corrélation par rapport aux autres signaux.

Dans le fichier `render.cpp`, la fonction `void processBuffer()` est utilisée comme tâche auxiliaire de la boucle de traitement principale. Nous avons déclaré un vecteur `gMeanCorrel` dans `render.cpp`; initialisé avec une valeur de 1 pour tous les indices, il prend la valeur que retourne la fonction de mixage à l'intérieur du code de la fonction `void ProcessMultiCorrel::process` que nous avons modifiée et qui est appelée dans `processBuffer` de `render.cpp`.

Dans l'implémentation de Jérémy Lixandre, le traitement des tâches auxiliaires dans `render.cpp` se fait à sens unique (exécution de tâches auxiliaires sans retour de valeur dans la boucle de traitement principale). Afin de récupérer le résultat du traitement de la fonction de mixage, nous avons passé la référence du vecteur contenant les moyennes de coefficients de corrélation, `gMeanCorrel`, en paramètre de la fonction `ProcessMultiCorrel::process` lors de son appel dans la fonction principale de `render.cpp`. On accède ainsi à la case mémoire de la variable pour permettre à la fonction de modifier son contenu et ainsi d'affecter les valeurs de retour de la fonction de mixage au vecteur ; c'est ainsi que la modification des volumes devient possible.

```
void ProcessMultiCorrel::process(const Matrix<float>& buffer,
                                 vector<float>& meanCorrelations
                                    ,
                                 Connection conn){
  Matrix<float> copy = buffer;

  // Processing functions
  copy = _preprocess(buffer);
  Matrix<float> correlMatrix = calcul_correl(copy);
  process_volume(correlMatrix, meanCorrelations);
  Matrix<RGB> mat = color_matrix(correlMatrix);

  // Send data
  string str = mat.toString();
  conn.send(str);
}
```

L'appel de la fonction `process` *ci-dessus exécute une série de fonctions de traitement de manière séquentielle.*

```
  void ProcessMultiCorrel::process_volume(const Matrix<float>&
                                            correlMatrix,
                                          vector<float>&
                                            meanCorrelations){
    meanCorrelations = this->_mixLevel(correlMatrix);
  }
```

Ci-dessus, la fonction appelée dans le corps de la fonction `process`*. Nous l'avons implémentée de sorte à altérer en son sein la valeur du vecteur des moyennes de corrélation passé par référence en paramètre.*

4.2 L'interface de configuration utilisateur

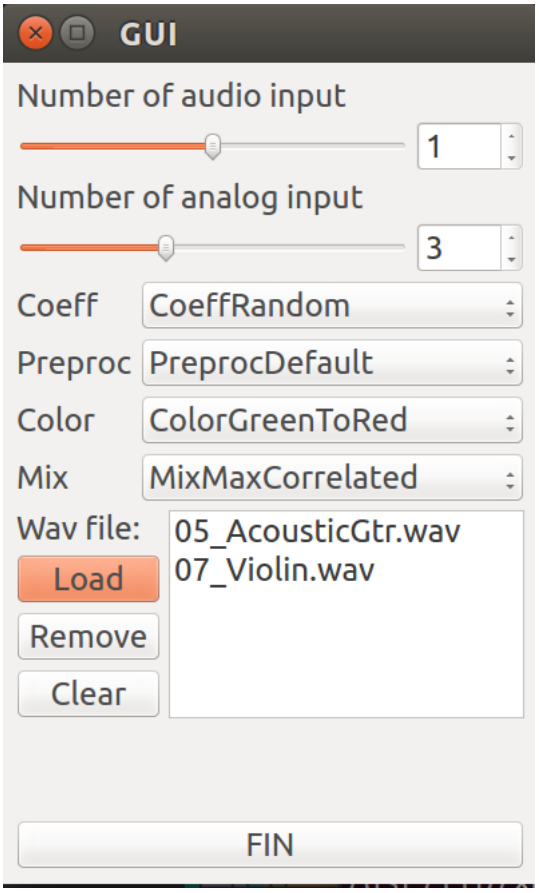

FIGURE 12 – Fenêtre de configuration

4.2.1 Présentation de l'architecture

Afin d'implémenter l'interface de configuration utilisateur précédemment abordée nous avons ajouté à l'architecture du programme un dossier `GUI` (*Graphic User Interface*) à la racine du programme. Le sous-répertoire contenant les fichiers relatifs à l'implémentation de l'interface de configuration se nomme `settingWindow`.

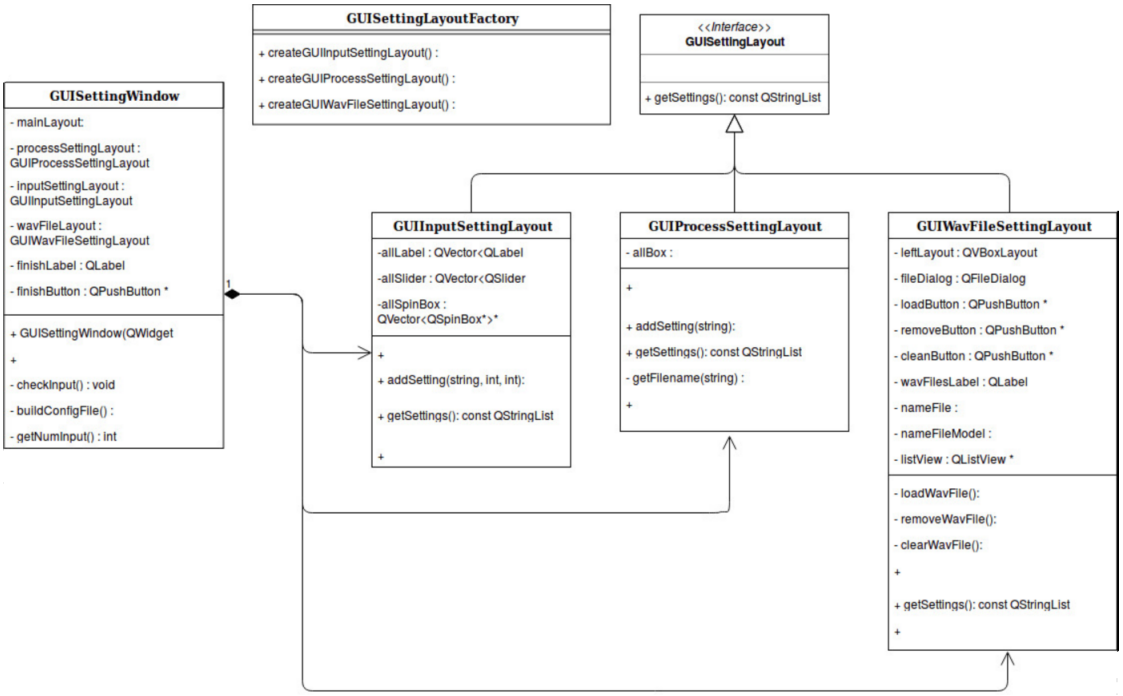

FIGURE 13 – Schéma UML de la fenêtre de configuration

La classe principale de cette interface de configuration est la classe `settingWindow`. Elle constitue une fenêtre vide destinée à afficher les données suivantes via les fichiers du répertoire `layout` :

— Paramètres des fonctions de traitement.
— Paramètres du nombre d'entrées audio et analogiques.
— Choix des fichiers audio *.wav*.

Afin de pouvoir créer ces parties, on utilise le *pattern* de conception *factory*, ou "fabrique". Une seule fabrique, implémentée dans l'architecture du sous-répertoire `layout` à partir de la classe `GUISettingLayoutFactory` permet de réunir l'ensemble des données exposées ci-dessus dans la fenêtre graphique.

Chacune des autres classes du répertoire `layout` représente un paramètre disposable dans la fenêtre principale de configuration.

— `GUIInputSettingLayout.cpp` permet d'afficher le menu de sélection du nombre d'entrées sonores en entrée
— `GUIProcessSettingLayout.cpp` permet d'afficher le menu de sélection des fonctions de traitement que l'on souhaite employer
— `GUIWavFileSettingLayout.cpp` permet d'afficher le menu de sélection des fichiers numériques audio à ajouter au programme

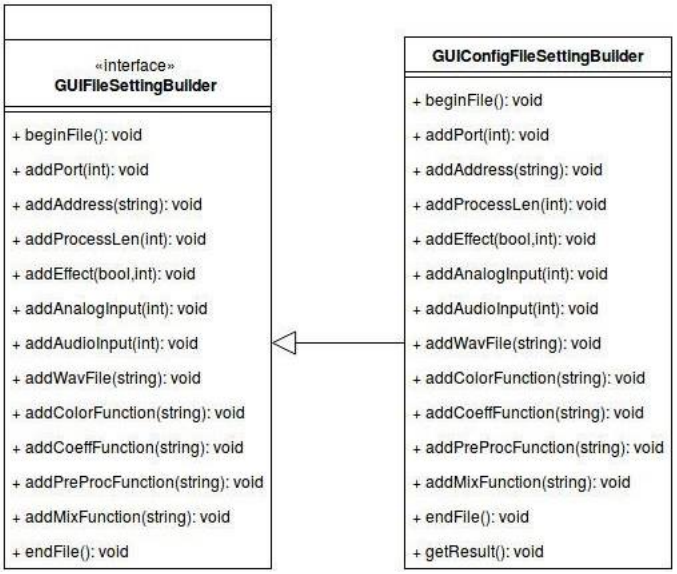

FIGURE 14 – Diagramme UML du *builder*

Le répertoire `fileSetting` contient un autre *pattern* de conception, un *builder* ou "monteur" générant un fichier de sortie au format particulier. L'implémentation du *builder* a été pensée de sorte à ce qu'à l'avenir, un futur programmeur puisse modifier ce format de retour afin que le programme puisse se passer du fichier de configuration. Pour l'instant, le fichier de retour est un fichier de type `.cfg` transmis à BELA.

4.2.2 Détails de l'implémentation

La classe GUISettingWindow contient tous les *layouts* et un bouton "fin" quand tous les paramètres sont rassemblés. Elle garantit que le nombre de pistes sonores en entrée (*inputs*) est supérieur ou égal à 2 ; dans le cas contraire, on ne peut pas calculer de corrélation.

Les classes du dossier layout héritent toutes de la classe `GUISettingLayout` qui, grâce à la fonction abstraite `virtual const QStringList getSettings`, peut retourner tous les paramètres sélectionnés dans la fenêtre graphique.

Parmi celles-ci, **la classe GUIInputSettingLayout** implémente un menu déroulant et une boîte dans laquelle l'utilisateur peut sélectionner un nombre compris entre un et maximum initialisé à la création de la fenêtre. **La classe GUIWavFileSettingLayout** stocke le chemin absolu des fichers `.wav` sélectionnés par l'utilisateur. Enfin, **la classe GUIProcessSettingLayout** va chercher, dans un dossier portant le nom donné en paramètre, les fichiers identifiés comme correspondant au nom en question, afin de permettre le choix de la fonction de traitement.

4.3 Autres ajouts mineurs sur le programme

Nous avons implémenté une nouvelle fonction de corrélation dédiée aux tests : appelée `CoeffRandom`, elle établit un coefficient de corrélation de manière aléatoire. La généricité de l'implémentation existante a rendu la tâche triviale, il nous a suffi d'écrire un fichier d'en-tête `CoeffRandom.hpp` et un fichier source `CoeffRandom.cpp` dans le dossier `Coeff` contenant les fonctions de calcul du coefficient de corrélation (dossier `process`). Cette fonction renvoie un flottant compris entre 0 et 1, comme les autres fonctions de corrélation. L'accomplissement de cette tâche et la trivialité qu'il représente témoigne de la généricité des fonctions de corrélation, et des fonctions de traitement en général, qui possèdent désormais toutes plusieurs versions alternatives.

5 Refactoring et révision de l'interface graphique

5.1 Révisions générales sur le code

Nous avons rapidement établi que le refactoring du code existant faisait partie des besoins de notre projet. Dans cette partie, nous décrirons l'ensemble des travaux réalisés dans ce cadre.

5.1.1 Conventions de programmation et assistance au programmeur

L'un des premiers reproches faits par l'équipe pédagogique à l'architecture héritée de Jérémy Lixandre est son manque de clarté ; le code livré était trop peu commenté pour permettre à un programmeur de comprendre ne serait-ce que le rôle de chaque classe sans rentrer en détail dans l'intégralité de leur code. Pour cette raison, nous avons abondemment commenté le code en rajoutant également de la documentation doxygen. Pour tous les fichiers sources, nous trouvons dans l'en-tête de chaque début de fichier `.cpp` le rôle et le fonctionnement de ceux-là.

Des commentaires ont également été ajoutés massivement au niveau des fonctions principales du programme. En effet, la compréhension du programme existant a considérablement ralenti le démarrage de notre implémentation sur ce projet, et a nécessité plusieurs recherches sur le web et la consultation conjointe de divers documents de sources différentes. Les deux fichiers ayant été les plus commentés de la sorte sont `main.cpp` et `render.cpp`.

La syntaxe du code a été remodelée de manière générale pour répondre à des conventions autant que pour faciliter sa compréhension : les "include" en début de fichiers ont été classés selon le type de fichiers inclus et par ordre alphabétique, la langue des commentaires a été uniformisée, les indentations, choix de placement de crochets/accolades etc. ont été uniformisés également pour une meilleure cohérence de la syntaxe, les variables ont été renommées pour être plus claires et éviter de heurter les conventions qui nous ont été enseignées.

5.1.2 Modification de l'architecture logicielle

D'autres approches du code réalisées par Jérémy Lixandre nous semblaient maladroites. Afin de rendre une architecture répondant au mieux aux conventions de programmation vues en cours et qui soit la plus compréhensible et claire possible aux yeux d'un potentiel futur programmeur, nous avons apporté plusieurs révisions liées à l'organisation des fichiers.

— Le code de `main.cpp` a été entièrement révisé. En effet, il contenait une unique fonction `main(int argc, char *argv[])` qui nous paraissait bien trop dense. Afin de faciliter la compréhension du lecteur et de mieux correspondre aux conventions, nous avons effectué un découpage fonctionnel de cette fonction en plusieurs fonctions. La fonction `main` appelle une fonction `launch(int argc, char *argv[])`

qui elle-même appelle d'autres fonctions (lesquelles, parfois, en appellent elles-mêmes de nouvelles).

```
static void launch(int argc, char *argv[]){
  ChSettings gChSettings;
  Parser config;
  void *handle;
  ProcessMultiCorrel *p;
  config = initParser(argc, argv);
  handle = initHandler();
  p = initProcessMultiCorrel(handle);
  setupSettings(gChSettings, config, p, handle);
  initAndRun(gChSettings, config, argv);
  stopAndCleanupAudio();
  freeAndClose(gChSettings, config, p, handle);
}
```

Ci-dessus, le code de la fonction launch

— Nous avons de même effectué un découpage fonctionnel sur la fonction `bool setup` du fichier `render.cpp`. Ce fichier, qui présentait tout comme `main.cpp` une organisation assez condensée, ne pouvait pas cependant être remanié autant en profondeur dans son intégralité que le code précédemment abordé. En effet, la fonction principale du fichier, `void render`, ne pouvait être découpée de manière maintenable. En effet, elle exécute une boucle de traitement audio, en créant des taches auxiliaires exécutées par d'autres threads pour garantir un traitement rapide (qui doit être au minimum plus rapide que la vitesse de lecture des pistes), et il n'était pas possible de la découper sans créer de conflits sur les variables enregistrant le nombre de tours de boucle ou les variables contenant les signaux audio par exemple.

```
bool setup(BelaContext *context, void *userData) {
  gUserSet = *((ChSettings *)userData);
  initUserSet(gUserSet);
  initBuffers();
  initSampleStreams(gUserSet);
  printInfo();
  return initAuxiliaryTasks();
}
```

Ci-dessus, le code de la fonction setup

— À l'intérieur du dossier `process`, les différents fichiers de traitement ont été classés par fonctions en quatre dossiers `Coeff`, `Color`, `Preproc` et `Mix`. À l'intérieur du dossier `test`, les fichiers que nous avons implémenté pour tester les différentes fonctions du programme ont été classés de la même manière (Jérémy Lixandre n'avait pour sa part pas implémenté de fichiers de test). Le fichier TestMain contenu

à la racine du dossier test se chargera de récupérer le registre de l'ensemble des tests contenus dans les différents dossiers.
— D'une manière générale, les fichiers ont été triés pour ordonner l'architecture proposée par Jérémy qui était un peu confuse ; des fichiers de code n'ayant rien à voir entre eux se côtoyaient au sein d'un même dossier, et l'architecture présentait une hiérarchie n'étant pas toujours en rapport avec celle que présente le logiciel. Pour ces raisons, nous avons partagé les fichiers en plusieurs sous-dossiers, notamment les fichiers sources et les fichiers d'en-tête au sein du dossier `VisualImpro`, et nous avons modifié les fichiers Makefile en conséquence. Le fichier de configuration `config.cfg` a notamment été déplacé dans un nouveau dossier `bin`.
— Dans le code livré par Jérémy Lixandre, un dossier `VisualImproExe` regroupait un fichier de configurations, les fichiers relatifs à NodeJS, un fichier html lié à l'affichage de la matrice graphique via le navigateur Mozilla Firefox et les fichiers sources C++ permettant les manipulations nécessaires au lancement du programme VisualImpro. Nous avons implémenté un script bash voué à remplacer la majeure partie de cette architecture, le fichier `VisualImpro.sh` placé dans le dossier `bin`.

Dans la version précédente de l'exécutable (`VisualImproExe`), les tâches réalisées par le code C++ étaient les suivantes :
— Copier les fichiers `.wav` indiqués dans le fichier de configuration dans Bela,
— Créer une copie du fichier de configuration, et y modifier les chemins des fichiers `.wav` pour que ceux-ci correspondent au système de fichiers de Bela,
— Copier la copie du fichier de configuration dans Bela,
— Lancement du serveur NodeJS avec le fichier `server.js`,
— Ouverture d'un onglet (ou d'une nouvelle fenêtre) du navigateur Firefox,
— Connexion à Bela (en ssh) et lancement du programme VisualImpro,
— Si le programme reçoit un signal d'interruption (Ctrl-C), alors certaines tâches sont réalisées :
  — Arrêt du programme VisualImpro,
  — Arrêt du serveur NodeJS,
  — Suppression des fichiers `.wav` sur Bela,
  — Rapatriement des logs vers le dossier `VisualImproExe`,
  — Suppression de la copie modifiée du fichier de configuration,
  — Arrêt de l'éxécutable.

Durant l'audit, le fait de réaliser des manipulations de fichiers de configuration, ainsi que des transferts de fichiers ou des connexions vers Bela à l'aide de fichiers C++ a été pointé du doigt. En effet, cette façon de faire reposait sur des classes C++ annexes qui effectuaient les manipulations de chaînes de caractères dans les fichiers de configuration, et pour ce qui concerne le transfert de fichiers et la connexion vers Bela, cela été réalisé à l'aide d'appels à la fonction C `int system(const char *command)`.

Réaliser toutes ces opérations à l'aide d'un script bash paraissait beaucoup plus pertinent. Cela permet de se passer des différents fichiers sources et d'en-tête C++, et de plus, il existe des commandes bash permettant de réaliser ces manipulations de chaînes de caractères de façon beaucoup moins lourde et fastidieuse.

En plus de reproduire le comportement de l'ancienne version de l'exécutable, notre script bash `VisualImpro.sh` permet de choisir entre les deux versions graphiques maintenant disponibles. Il suffit de préciser à l'aide d'une option (`qt` ou `firefox`) quel affichage on souhaite obtenir.

Parmi les autres aspects de l'existant que nous avons tenu à réviser, la classe `utilities` posait problème dans le sens où elle contenait un trop grand nombre de fonctions n'ayant rien à voir entre elles. Nous avons tâché de réduire son contenu au maximum, et actuellement, elle ne contient plus que deux fonctions utilisées par les fichiers `main.cpp` et `render.cpp`, et son rôle est véritablement celui d'une classe utilitaire pour les fichiers principaux du programme. Les fonctions de traitement que contenaient cette classe et nécessitaient son inclusion dans tous les fichiers du répertoire `process` ont été déplacées et réparties dans le code pour éviter ces inclusions systématiques. Nous avons également crée une classe `RGB` en remplacement de la structure `RGB` que contenait `utilities`. Elle regroupe également les fonctions refactorées de l'ancienne classe `triplet`.

Face au grand nombre de traitements de matrices dans le code, nous avons choisi d'implémenter une classe `Matrix.cpp` afin d'éviter d'utiliser systématiquement des vecteurs de vecteurs de flottants, procédé lourd en écriture et exigeant l'écriture de fonctions de traitement de vecteurs de vecteurs. La nouvelle classe `Matrix` rassemble toutes les opérations sur les matrices nécessaires au programme, en plus de divers constructeurs pour créer des matrices de différents types selon différents paramètres. Classe *template*, `Matrix` permet de créer des matrices d'entiers, de flottants ou bien de triplets RGB.

L'implémentation de Jérémy Lixandre comportait une classe `ProcessMulti` dont héritaient les classes `ProcessMultiCorrel` et `ProcessMultiWriteWav`, mais l'héritage n'avait pas lieu d'être puisque la classe `ProcessMulti` en question définissait une fonction abstraite `process`, redéfinie dans ses deux classes filles et prenant en paramètre une matrice de flottants (représentant les signaux d'entrée) et une connexion. Or, la classe `ProcessMultiWriteWav` sert à créer des fichiers `.wav` et ne recquiert aucune connexion. Nous avons donc cassé l'héritage entre ces classes et supprimé la classe `ProcessMulti`.

Nous avons ajouté des fichiers d'en-tête pour tous les fichiers source contenant les fonctions de traitement, afin de pouvoir utiliser ces fonctions, dans les fichiers de test notamment, en incluant les fichiers nécessaires. Les fonctions de traitement étaient auparavant copiées dans la classe `utilities`, mais lors du refactoring de cette dernière, nous avons réalisé que cette solution n'était pas très conventionnelle.

FIGURE 15 – Arborescence des fichiers

Voici ci-dessus l'arborescence de nos fichiers.

5.1.3 Optimisation du code

Afin de garantir un traitement rapide des informations par BELA et de produire un retour sonore sans risque de latence, nous avons dû optimiser certaines parties du code. Dans son rapport, Jérémy Lixandre avait précisé que le traitement de BELA occupait une part trop importante du CPU à partir de quinze fichiers `.wav` traités simultanément ; ce constat nous a encouragés à optimiser le programme pour limiter le coût de son exécution.

L'implémentation de l'existant présentait notamment de nombreuses copies des matrices passées en paramètres. Ces matrices sont des éléments particulièrement lourds,

contenant un nombre de données égal au produit du nombre de pistes sonores en entrée par 32768 (taille des buffers traités). Nous avons choisi de remplacer ces copies successives de matrices par le placement en paramètres de références constantes de ces matrices.

```
void ProcessMultiCorrel::process(const Matrix<float>& buffer,
                                 vector<float>& meanCorrelations
                                     ,
                                 Connection conn){
  Matrix<float> copy = buffer;

  // Processing functions
  copy = _preprocess(buffer);
  Matrix<float> correlMatrix = calcul_correl(copy);
  process_volume(correlMatrix, meanCorrelations);
  Matrix<RGB> mat = color_matrix(correlMatrix);

  // Send data
  string str = mat.toString();
  conn.send(str);
}
```

Ci-dessus, la fonction principale de `ProcessMultiCorrel.cpp`*. Ce code témoigne de multiples opérations de refactoring et d'optimisations du code ; en effet, le nombre d'appels de fonctions au sein de la fonction process témoigne du découpage fonctionnel de la fonction d'origine, laquelle était autrement plus dense. De plus, on peut observer dans les paramètres qu'un vecteur des moyennes de coefficients de corrélation a été ajoutée pour permettre l'implémentation du mixage du retour sonore. Enfin, la matrice de flottants "buffer" a été changée en sa référence constante, afin d'éviter d'avoir à en faire une copie dans le corps de la fonction.*

Au sein de la nouvelle classe `RGB`, de nombreux calculs notamment ceux permettant la conversion d'entiers en valeurs hexadécimales ont été simplifiés.

5.2 Les changements apportés à l'interface graphique

Précédemment, nous parlions de la nécessité de modifier l'affichage de l'interface graphique pour diverses raisons. Nos clients n'exprimaient pas de préférence quant à la méthode employée pour afficher la matrice imaginée par nos prédécesseurs de l'ENSEIRB, mais nous trouvions peu judicieux et coûteux pour le programme de passer par NodeJS et Mozilla Firefox pour la produire. Un affichage graphique implémenté en C++ grâce au framework Qt nous paraissait plus logique, judicieux et conforme au reste de l'architecture de notre programme.

Cependant, nous pouvons reconnaître l'avantage que peut avoir l'affichage de la matrice sur un navigateur répandu comme Mozilla Firefox. En effet, on peut imaginer qu'à l'avenir, le programme embarqué sur le système Bela pourra se passer d'un ordinateur pour fonctionner et afficher la matrice sur l'écran d'un smartphone par exemple en passant par un navigateur web. Pour cette raison, nous avons décidé d'offrir à l'utilisateur la possibilité de choisir entre l'affichage Qt et l'affichage sur navigateur au moment de lancer l'exécution du programme.

5.2.1 L'implémentation de la méthode d'affichage sous Qt

À l'intérieur du dossier `GUI` répertoriant les classes dédiées aux affichages graphiques, le dossier `mainWindow` contient l'architecture propre à l'affichage de la matrice graphique via Qt. Elle contient une classe, `GUIWindow`, et deux répertoires : `layout` dédié à l'affichage de la matrice, et `server` contenant le serveur établissant la communication avec BELA.

L'architecture `mainWindow` a pour unique rôle d'afficher la matrice graphique de corrélations et de la mettre à jour en temps réel. Elle réagit aux mises à jour transmises par BELA via `render.cpp` à l'aide de d'une connection TCP et hérite de la classe Qt `QMainWindow` ce qui lui permet notamment d'afficher des menus. À l'avenir, ces menus pourront permettre à l'utilisateur de modifier des configurations du programme en temps réel si un successeur implémente cette fonctionnalité.

FIGURE 16 – Aperçu de la matrice de corrélation affichée avec Qt

La classe `GUIImageMatrix` du répertoire `layout` contient un objet de type `QPixMap` dont les couleurs sont mises à jour pour chaque rectangle le composant.

La classe `GUITCPServer` du répertoire `server` constitue un serveur TCP permettant la communication avec BELA. BELA se connecte a ce serveur et crée une socket ; les paquets transmis par BELA et contenant une matrice codée sous forme d'un objet de type `string` sont retraduits en matrice RGB. Le framework Qt permet la communication entre le *layout* et le *server* via l'émission de signaux depuis le serveur, transmettant les données de la matrice à *layout*.

Comme cette méthode d'affichage contient uniquement un serveur et un affichage de matrice via une image, le programme se retrouve extrêmement allégé par rapport au programme de notre prédécesseur qui utilisait Mozilla Firefox ainsi que NodeJS.

6 Tests du logiciel

6.1 Tests unitaires

Les tests, absents de la version du logiciel VisualImpro livrée par Jérémy Lixandre, ont été entièrement implémentés par nos soins. Nous les avons regroupés dans un dossier `Tests` qui comprend un fichier source `TestMain.cpp`, lequel affiche les résultats de tous les autres fichiers de test sur la sortie standard, et six répertoires contenant ces fichiers de test. Il contient également un `Makefile` et un répertoire `Sine-samples` contenant des signaux sinusoïdaux à tester.

6.1.1 Tests de classes

Le répertoire `TestClasses` vérifie le bon fonctionnement de certaines classes.

Le fichier `TestMatrix.cpp` est implémenté dans le but de démontrer le correct fonctionnement de la classe codant l'objet matrice, de la classe `Matrix.cpp`. Il teste des opérations portant sur des matrices, sur des cases, des lignes et des colonnes spécifiques de matrices.

Le fichier `TestMultiCorrel.cpp` teste des opérations de base sur l'objet permettant de traiter le retour sonore, `ProcessMultiCorrel.cpp`, telles que des opérations d'accès et d'écriture basiques *getter/setter* aux données membres de la classe.

6.1.2 Tests de la fonction de traduction du coefficient de corrélation en triplets RGB

Le répertoire `TestColor` contient deux fichiers de test testant chacun le bon fonctionnement des deux fonctions de traduction du coefficient de corrélation en triplet RGB (la première établit un code couleur allant du rouge au vert, la seconde un code couleur en niveaux de gris allant du noir au blanc). Ces tests fonctionnent en vérifiant que des flottants ayant des valeurs extrêmes et une valeur intermédiaire retournent bien les couleurs attendues, par exemple : une couleur rouge pour un coefficient de 0 avec la première fonction.

6.1.3 Tests de la fonction de pré-traitement

Le répertoire `TestPreproc` contient trois fichiers de test, un pour chacune des trois fonctions de pré-traitement existantes. L'implémentation de `TestPreprocDefault.cpp`, `TestPreprocEnergy.cpp` et `TestPreprocStrenghtEnergy.cpp` permet de comparer le résultat des fonctions de pré-traitement correspondantes, prenant des matrices simples en paramètres, avec les résultats attendus.

6.1.4 Tests de la fonction de calcul de coefficient de corrélation

Le répertoire `TestCoeff` contient deux fichiers testant chacun l'une de nos deux fonctions de calcul de coefficient de corrélation. Le premier, `TestCoeffScalar.cpp`, vérifie que la fonction testée renvoie bien la bonne valeur en comparant son résultat pour deux vecteurs arbitraires passés en paramètre avec le résutat du calcul brut du produit scalaire de ces deux mêmes vecteurs, qui doivent être égaux. Le second fichier de test unitaire, `TestCoeffRandom`, teste que pour mille itérations, le résultat de la fonction testée est toujours compris dans l'ensemble de valeur attendu (entre 0 et 1).

6.1.5 Tests de la fonction de mixage

Les fichiers contenus dans le répertoire `TestMix` s'assurent pour chaque fonction de mixage existante que le résultat obtenu pour une matrice passée en paramètre de la fonction est égal à celui attendu. Par exemple, la fonction de `MixNeutral.cpp`, qui simule l'absence de fonction de mixage, doit renvoyer 1 comme coefficient de modification du volume sonore quelque soit la matrice d'entrée ; on s'en assure dans le fichier de test `TestMixNeutral.cpp`.

6.1.6 Test de l'interface graphique utilisateur

Les fichiers contenus dans le répertoire `TestGUI` assurent le fonctionnement de l'interface graphique. Le seul test implémenté et le test du builder qui est la seule classe qui construit le fichier description de configuration. En effet, hormis en utilisant une capture d'écran de l'affichage voulu afin de la comparer à celle qui est affichée, il est difficile de tester une interface graphique.

6.2 Test globaux

Pour tester l'outil dans son intégralité, i lfallait des échantillons pour lesquels nous connaissions les résultats à l'avance. Pour cela, nous avons généré des signaux sinusoïdaux et carrés, car ce sont les signaux les plus simples. Nous en avons généré plusieurs, et à chaque fois, nous avons décalé les phases pour pouvoir réaliser plusieurs tests. De plus, nous avons réalisé nos tests avec deux focntions de pré-traitement différentes : `PreprocDefault` et `PreprocStrengthEnergy`. La première ne touche pas au signal, la seconde transforme la composante négative en positive.

6.2.1 Avec PreprocDefault

Les figures suivantes doivent être interprétées de la façon suivante : plus la couleur est verte, plus les signaux sont proches, plus on tend vers le rouge, plus les signaux sont différents.

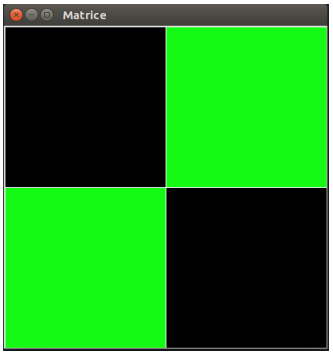

FIGURE 17 – Test avec deux sinus identiques

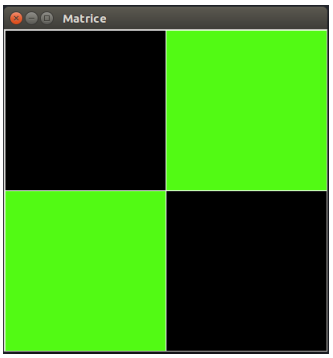

FIGURE 18 – Test avec deux sinus décalés de 30 degrés

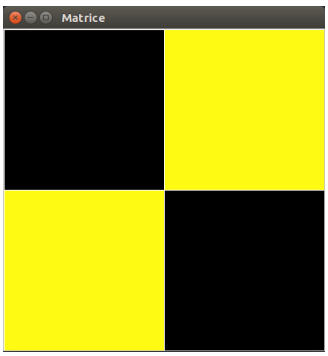

FIGURE 19 – Test avec deux sinus décalés de 60 degrés

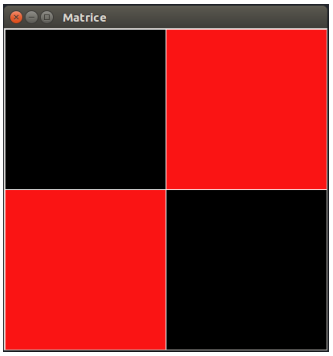

FIGURE 20 – Test avec deux sinus décalés de 90 degrés

À 90 degrés, les sinus sont inversées, d'où le fait que la corrélation soit nulle.

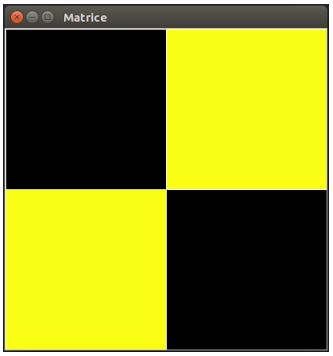

FIGURE 21 – Test avec deux sinus décalés de 120 degrés

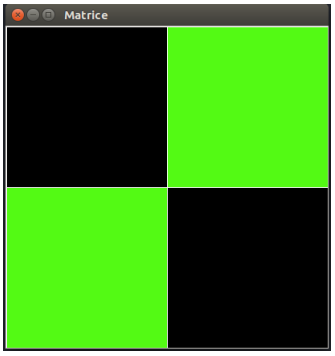

FIGURE 22 – Test avec deux sinus décalés de 150 degrés

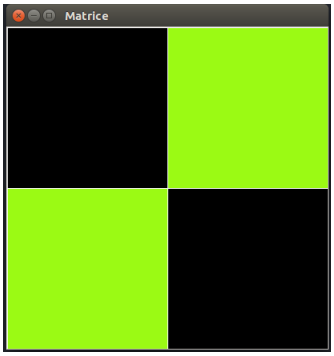

FIGURE 23 – Test avec deux sinus décalés de 180 degrés

6.2.2 Avec PreprocStrenghtEnergy

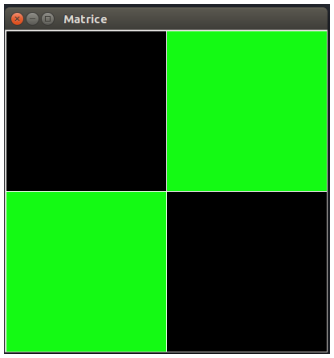

FIGURE 24 – Test avec deux sinus identiques

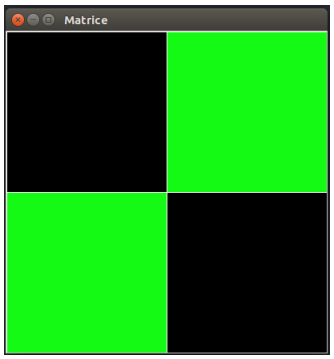

FIGURE 25 – Test avec deux sinus décalés de 90 degrés

Avec PreprocStrenghtEnergy, comme il n'y a plus de valeurs négatives, la sinus décalée de 90 degrés est identique à la sinus normale.

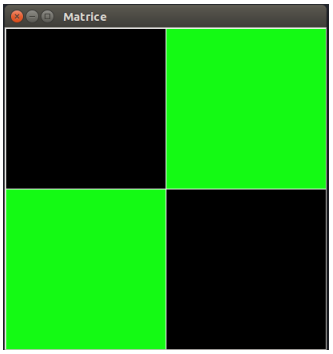

FIGURE 26 – Test avec deux sinus décalés de 180 degrés

7 Conclusion

7.1 Éléments manquants du projet

Parmi les demandes de nos clients, il nous était demandé de fournir une nouvelle fonction de calcul de corrélation, basée sur l'auto-corrélation temporelle, qui pourrait retourner non plus un coefficient de corrélation, mais pour chaque paire d'entrées, un décalage avec lequel la paire est la plus corrélée. Seulement cette fonction de calcul a suscité chez nous quelques interrogations :

— Comment représenter visuellement un décalage temporel ?
— Est-ce que la complexité d'une telle fonction ne créerait pas trop de latences ?
— Si l'échelle temporelle devait être plus grande, faudrait-il utiliser une file contenant les tampons sur lesquels il faut lancer le calcul ?

De ce fait, nous avons revu la priorité de cette demande à la baisse, et finalement, nous n'avons pas eu le temps de mener cet objectif à bien.

De plus, l'outil VisualImpro avait pour objectif de pouvoir ajouter des effets aux entrées. Dès le départ, la priorité de ce besoin était basse, nous avions prévu de ne pas s'attarder sur cet aspect là.

Nous avions l'intention de tester Bela avec des musiciens (en "live"), mais nos tests de l'outil se sont limités à l'utilisation de fichiers `.wav` triviaux, afin de prédire les retours et tester la fiabilité de notre programme.

7.2 Évolution de l'outil

Dans une future itération de ce projet, on pourrait imaginer un changement des paramètres en pleine exécution au lieu de seulement les fixer avant. De plus, tous les éléments cités précédemment pourront être implémentés.

Références

[ENV18] Signal envelope - matlab envelope - mathworks france. `https://fr.mathworks.com/help/signal/ref/envelope.html?requestedDomain=true`, 2018.

[Lab16] The Augmented Instruments Laboratory. Bela. `http://bela.io`, 2016.

[XEN18] Xenomai linux. `https://xenomai.org/`, 2018.