Projet de Fin d'Etudes - Rapport de Synthèse

Professeur de Musique Virtuel

–

INSA de Lyon, Département Informatique

Gilles Mathieu
sotww@yahoo.ie

mars - août 2003

Universität KARLSRUHE, Fakultät für Informatik
Institut für Logik, Komplexität und Deduktionssysteme

Encadrement :
Universität Karlsruhe : **Hagen Soltau** (soltau@ira.uka.de)
INSA Lyon : **Jean-Claude Commerçon** (jean-claude.commercon@if.insa-lyon.fr)

Résumé

L'un des enjeux les plus importants de l'informatique aujourd'hui est l'utilisation que l'on peut en faire dans les domaines de l'éducation, l'apprentissage et l'enseignement.

Le but du présent projet est de définir un moyen d'enseigner la musique au moyen de l'informatique, et de réaliser un logiciel permettant un "apprentissage musical assisté par ordinateur" : A partir d'une partition de référence, il s'agit de détecter les erreurs jouées et donner une explication sur ces erreurs (fausses notes, erreurs de rythme, de tempo, de nuances...).

Le programme devrait ainsi permettre aux élèves musiciens qui l'utilisent de répéter leurs morceaux chez eux, en suivant et corrigeant leurs erreurs.

Ceci implique tout d'abord de définir précisément toutes les notions relatives à la détection d'erreurs et aux règles de diagnostique, ainsi qu'un travail de recherche sur les données à utiliser et les problèmes d'interface : acquisition du flux d'entrée, des sorties, etc.

Le système réalisé, basé sur du suivi de partition utilisant des Modèles de Markov Cachés, permet dans l'etat actuel des travaux de réaliser un diagnostique de fautes sur un morceau monophonique au format MIDI.

One of the most important features of computers today is the way we can use them for education, learning and teaching. The goal of the project is to define how we can teach music with a software, and to realize such one which allows computer aided music learning. This software should not give musical theory basis, which can be found in any music book, but realize an "artistical diagnostic" on played music, as would do a real teacher : with a score as reference, detect errors on played music, and find ways to avoid them (typical errors are for example wrong notes, rhythm errors, tempo errors, and interpretation errors). Applications could be everykind of tonal music.

That implies good definitions of errors and diagnostic rules, but that means too a searching work on interface problems : way to acquire and to model played music, output definition, etc.

This report describes the project bases and gives a documentation of what has been done since the beginning. This is not a complete, finished documentation but rather a "state of the art" of the Virtual Music Teacher at the end of August 2003. Things may change in the future if the project is continued.

1 Introduction

Effectué dans le cadre d'un échange académique, Le projet "Virtual Music Teacher" a été lancé en mars 2003, et le présent document en donne l'état d'avancement à la fin du mois d'août 2003. Outre les universités d'accueil et d'origine, le projet a été réalisé en collaboration avec l'IRCAM - Centre Pompidou, qui garde l'exclusivité du code fourni.

Sont présentés dans ce rapport les démarches de recherche et de définitions relatives à la détection de fautes et aux questions de formats de données et d'interfaces, les solutions retenues pour la réalisation, ainsi que les premiers résultats obtenus.

2 Définition des erreurs

2.1 classification

Les erreurs de jeu peuvent être classées en trois grand groupes, avec pour chacun des sous-catégories :

erreurs de notes : fausse note, note sautée, note ajoutée,

erreurs de rythme : note trop longue ou trop courte, silence sauté, silence ajouté,

erreurs d'interprétation : passage joué trop fort ou trop doux, erreur d'accentuation (notes marquées), erreur de phrasés (notes liées ou non).

Il est très important de faire la distinction entre ces différentes erreurs de manière à obtenir un diagnostique le plus précis et le plus utile possible.

2.2 critères

Une erreur de jeu se caractérise par une différence entre la partition et le jeu. Pour les erreurs de notes, il s'agit de la **hauteur** de la note ainsi que de sa **position** (mesure, temps) ; pour les erreurs de rythme, le **temps de départ** ainsi que la **durée** ; pour les erreurs d'interprétation, les erreurs seront détectées sur la **vélocité** (niveau sonore) de la note, et également sur la **durée**.

2.3 tolérances

Pour certains critères de comparaison, il s'agit de définir une tolérance au delà de laquelle on pourra détecter une erreur : Il n'y a en effet aucun intérêt à détecter comme erreur une différence de quelques millisecondes sur une note qui dure 4 secondes. Ce facteur de tolérance s'exprimera en pourcentage de durée de note pour les erreurs de rythme, et en valeurs seuils pour les erreurs de vélocité.

3 Suivi de partitions

3.1 Principe et applications

Les techniques de suivi de partition sont nées du besoin de synchroniser un système informatique sur le jeu d'un instrumentiste ; afin que le dispositif jouant le rôle d'accompagnateur (diffusion de sons, spacialisations, effets...) soit toujours en phase avec l'interprète, il s'agit de repérer en temps-réel l'endroit précis où celui-ci se trouve dans la partition.

Les applications sont nombreuses : concerts de musique mixte [2], alignement de texte sur voix chantée [1], karaoké [7]...

Les systèmes de suivi font l'objet de nombreuses recherches, notamment ceux basés sur les Modèles de Markov Cachés : voir [1] et [10].

3.2 Fonctionnement

Les Modèles de Markov Cachés (HMMs), décrits par Rabiner dans [11], servent de base à la plupart des systèmes en cours de développement. L'idée est de réaliser l'alignement entre l'interprétation et la partition en utilisant un HMM, où la partition serait le modèle à reconstituer et l'interprétation les observations du système. Le principe est décrit de manière complète dans [12].

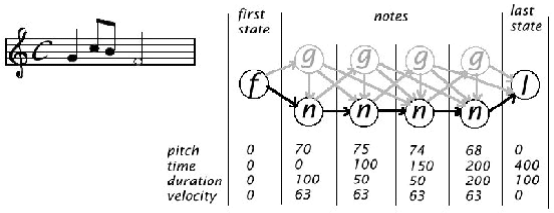

FIG. 1 – *Une partition et son modèle*

Une mise en pratique de ce principe est le système en cours de développement à l'IRCAM, décrit dans [9]. La modélisation de la partition est faite de la manière suivante : à chaque note est associé un état correspondant à la bonne note (n-state), ainsi qu'un état "fantôme" (g-state) correspondant à "tout sauf la bonne note" (voir fi-

gure 1). Les transitions entre états correspondent aux probabilités de passer d'une note à l'autre.

Le décodage des observations, utilisant l'algorithme de Vitterbi, permet de reconstituer la séquence jouée en calculant la probabilité maximale d'être dans tel ou tel état.

A un niveau inférieur, chaque état est en fait le regroupement de plusieurs sous-états en clusters, qui permettent une meilleure prise en compte du temps [8].

3.3 Utilité pour le présent projet

L'intérêt du suivi de partition pour un système de détection de fautes est évident : si l'on est capable de savoir où on est (donc ce qu'on a joué), on doit pouvoir être capable de savoir où les fautes ont été faites.

Ainsi, en comparant l'alignement correspondant à l'interprétation avec l'alignement idéal, il est possible de repérer et d'analyser les erreurs.

4 Détection d'erreurs

4.1 Modéle pour la partition

Nous utilisons deux modèles pour représenter la partition.

Le premier, qui sera utilisé pour le suivi de partition, se présente sous la forme d'un HMM, et correspond au modèle théorique décrit dans [9] et présenté figure 1.

Le deuxième, qui sert à la récupération de données pour la caractérisation des erreurs, est simplement un tableau dynamique d'objets dans lequel sont stockés tous les paramètres caractéristiques de chaque note (hauteur, durée, volume, accent, etc.).

Pour chacun de ces deux modèles, les données associées à chaque note modélisée dépendent du format choisi en entrée. La plus grande partie des travaux effectués jusqu'à présent utilsant le format standard MIDI, les paramètres stockés sont codés en correspondance avec ce format.

4.2 Représentation de la performance

La représentation de l'interprétation du morceau dépend aussi du format choisi pour l'acquisition. Avec une entrée de type MIDI, chaque note jouée sera représentée de la même façon qu'une note de la partition.

La différence est qu'il ne s'agit plus là d'un modèle statique, mais d'un modèle remis à jour à chaque réception de note.

4.3 Mécanismes de détection

La détection se divise en deux temps : la détection proprement dite, et la caractérisation. Le problème de la détection est qu'elle doit être effectuée au fur et à mesure de l'arrivée des informations, c'est à dire sans le recul nécessaire à la caractérisation.

Dans notre cas, la difficulté vient du fait que les erreurs ne sont pas des réponses fausses à des questions, mais des différences entre deux alignements qui devraient être identiques. Ainsi, la détection est possible en temps réel (dès que l'on remarque une différence entre les deux alignements), mais pas la caractérisation.

Certaines méthodes incrémentales ont été proposées pour la détection d'erreurs. Dans [6], la méthode mise en oeuvre est l'utilisation d'un arbre pour caractériser l'enchainement des réponses et détecter les erreurs.

Ici, l'utilisation d'un HMM ainsi que d'un tableau de données nous permet de repérer facilement les écarts.

4.3.1 erreurs de note

Un faute de note se caractérise par le passage dans un *g-state* (cf. figure 1) lors de la reconstruction du modèle. La détection est donc simple : un *g-state* = une faute.

En revanche, la caractérisation est plus ardue. Il faut en effet attendre d'avoir le recul nécessaire pour savoir ce qui s'est passé avant et après le passage dans cet état du système. En observant ensuite la succession des états atteints, il est possible de déterminer le type de l'erreur observée (voir figure 2).

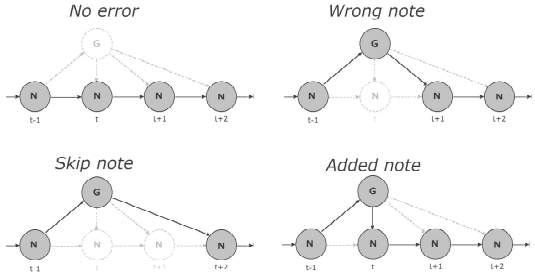

FIG. 2 – *Différences dans la succession des états atteints selon le type d'erreur de note*

4.3.2 erreurs de rythme

A part les erreurs de type "silence sauté" qu'il est possible de détecter de la même manière que les erreurs de note (un silence est modélisé comme une note, *i.e.* avec un *g-state* associé appelé *g-rest*), l'observation de l'alignement ne suffit pas à détecter des erreurs de rythme. Il faut pour cela utiliser le deuxième modèle, et comparer le temps d'arrivée de la note jouée avec le temps attendu dans la partition, en tenant bien sûr compte des problèmes de tempo et des décalages dus aux notes précédentes.

La caractérisation est par contre plus simple : la valeur obtenue en comparant les temps nous donne directement le type de l'erreur (note trop longue, trop courte,...)

4.3.3 erreurs d'interprétation

Le problème de ce type d'erreurs est d'une part qu'elles sont parfois ambiguës (est-ce une erreur ou une interprétation...), et d'autres part qu'elles sont souvent globales, *i.e.* étendues sur plusieurs notes. Il est donc difficile de faire de la détection en temps réel. C'est le cas des fautes de nuances, où il faut attendre d'avoir de nouveau la bonne vélocité pour pouvoir dire qu'il y avait une erreur.

Les fautes d'accentuation sont détéctées relativement aux notes qui les entourent, ce qui rend les erreurs combinées nuances/accentuation difficiles à identifier.

5 Réalisation

5.1 L'environnement jMax

Le système a été réalisé sous la forme d'une extension du programme existant **jMax** : il s'agit d'un environnement graphique de programmation pour la musique et le multimédia en temps réel.

Principalement utilisé en concert, jMax sert également au traitement du son en post-production, ainsi qu'en simulation ou en traitement du signal dans les domaines comme l'éducation et la recherche.

jMax est un logiciel développé par l'IRCAM Centre Pompidou.

5.1.1 Côté utilisateur

jMax se programme par assemblage de modules graphiques appelés "*patches*". A la souris, l'utilisateur connecte des modules standards ou ceux qu'il trouve dans les nombreuses bibliothèques applicatives : synthèse, filtres, effets, spatialisation, suivi de partition...

En fonctionnement, jMax se présente sous la forme suivante :
- une console, où s'affichent les messages à destination de l'utilisateur (logs, initialisations...),
- un ou plusieurs *patch* contenant les objets insérés et connectés entre eux par l'utilisateur.

Chaque objet comporte son nombre d'entrées (*inlets*) et de sorties (*outlets*), correspondant chacune à une fonction précise. Les *outlets* de certains objets sont connectables aux *inlets* des autres, ce qui permet de programmer graphiquement des enchainements de fonctions.

Parmi les objets les plus utiles, on trouve
- des objets d'affichage, permettant d'afficher sous forme graphique ou textuelle ce qu'ils recoivent en entrée,
- des objets émetteurs tels que *message boxes*, potentiomètres, *bangs*, etc.,
- des objets de stockage de données comme des vecteurs, matrices, listes, etc.,
- des objets sequenceurs qui permettent de manipuler des sequences de type MIDI ou autres,
- des objets audio permettant de lire des fichiers son,
- beaucoup d'autres, comme les objets vidéo, traitements mathématiques, etc.

Les objets peuvent être combinés et reliés à l'infini, en utilisant des *sous-patches*, en modifiants le contenu des objets au moyen d'éditeurs, etc.

5.1.2 côté programmeur

jMax est composé d'un moteur temps-réel écrit en C, et d'une interface utilisateur écrite en java.

La classe de base de tout objet programmé pour jMax est la FTS-class (FTS est le moteur temps-réel de jMax). Toutes les variables typées (à l'exception des *integer*, *float* et *symbol*) sont des objets, et font donc référence à une FTS-class.

Les objets programmés pour jMax sont regroupés en packages. En général, il y a une classe FTS par package. A l'intérieur de chaque package, les objets sont définis par des structures dont le premier membre est toujours l'objet FTS de référence, ou l'objet "générique" du package (qui est lui même une structure contenant comme premier membre l'objet FTS). Ceci permet aux fonctions de chaque objet de disposer de mécanismes d'appel assez simples, en créant une architecture du type *classes à héritage.*

les classes permettent d'implémenter des méthodes pour les messages ou les inlets d'un objet, et permettent de déclarer des outlets. Ces méthodes peuvent être appelées par :
- la réception d'un message envoyé directement à l'objet,
- la réception d'un message envoyé par l'*outlet* d'un autre objet dans le premier *inlet* de l'objet,
- la réception de données envoyées par l'*outlet* d'un autre objet dans n'importe quel *inlet* de l'objet,
Chaque méthode peut ensuite envoyer des données par un *outlet* de l'objet, ou retourner ces données à l'objet FTS principal.

On trouve donc plusieurs types de méthodes dans les codes sources de chaque objet :
- les méthodes système : construction, destruction, initialisation, qui sont appelés lorsqu'on insère l'objet dans un *patch* jMax,
- les méthodes d'*inlets*, qui sont invoquées sur réception d'un message ou de données au niveau de l'*inlet* spécifié,
- les méthodes internes, utilisées uniquement par les autres méthodes de l'objet.

Les méthodes d'*inlets* ont toutes le même prototype :

```
static void method(fts_object_t *o,
    int winlet, fts_symbol_t s,
    int ac,const fts_atom_t *at);
```

o est un pointeur sur l'objet de référence (et donc sur l'objet lui même, il suffit pour cela de faire un *cast*), winlet est le numéro d'*inlet*, et s, ac et at sont les variables permettant de récupérer les données reçues.

Pour plus d'informations et de détails sur jMax, voir [4] et [5], ou consulter la page des produits IRCAM :
http ://freesoftware.ircam.fr

5.2 Suivi de Partitions pour jMax

Les recherches théoriques sur le suivi de partitions présentées dans [10] et [9] ont mené à la réalisation d'un package pour jMax : le package *suivi*. Ce package regroupe plusieurs objets, dont deux objets de suivi de partitions : *suivimidi* et *suiviaudio*, suivant le type de données utilisées en entrée.

Ces objets utilisent comme partition de référence un objet jMax de type séquence, à partir duquel est construit le HMM. Les données relatives aux notes reconnues de la partition ainsi qu'aux états traversés sont envoyées en sortie de l'objet.

Ce système est encore en cours de développement à l'IRCAM. L'état actuel des travaux est présenté dans [3].

5.3 Objets implémentés : le package "Virtual Music Teacher" (VMT)

Le coeur de projet a été l'implémentation d'un nouveau package, afin de réaliser l'analyse de l'alignement trouvé et de caractériser les erreurs. Ce package contient deux objets, *mididetector* et *audiodetector*, dédiés respectivement à une entrée MIDI et une entrée audio. La plupart des recherches ont été menées sur l'objet *mididetector*.

5.3.1 Paramétrage - initialisation

La partition du morceau à jouer est saisie par l'utilisateur sous la forme d'un fichier texte (la syntaxe d'écriture ainsi qu'un exemple sont donnés en annexe B). Ceci permet de regrouper tous les événements de la partition : notes, controleurs de tempo, de nuances, etc.

A partir de ce fichier texte, l'objet construit son propre modèle interne sous forme de tableau dynamique, et génère une séquence jMax qui servira de base pour établir le HMM utilisé par le suivi de partition.

Un certain nombre de paramètres sont aussi réglables par l'utilisateur *via* des potentiomètres ou des boites-messages que l'on peut insérer dans le *patch* jMax : tolérances, choix des erreurs à détecter, choix de la partition en entrée, type de sortie souhaité...

5.3.2 Entrées

L'objet reçoit en entrée :
- le flux joué, en l'occurence une succession de listes de type [*hauteur, vélocité*] [1],

- les données relatives à la note reconnue dans la partition : le temps et les données MIDI [2],

- l'état du modèle correspondant, qui est lui aussi une sortie de l'objet *suivimidi.*

Ces sorties ne sont pas forcément simultanées, et en aucun cas synchronisées. Un travail de synchronisation est donc nécessaire à la récupération, de manière à toujours traiter les bonnes données.

5.3.3 Traitement

La détection d'erreurs est activée à chaque réception d'évènement.

L'algorithme de détection recherche en premier lieu les erreurs de notes (recherche d'un *g-state* dans l'alignement récupéré), puis les erreurs de rythme (recherche d'un éventuel *g-rest* et comparaison des paramètres temporels), et enfin les erreurs de dynamique (comparaison des vélocités et paramètres d'accentuation).

La détection de l'un de ces types d'erreurs entraine l'activation des algorithmes de classification correspondants, qui suivent les principes de classification décrits en 4.3 .

5.3.4 Sorties

L'objet *mididetector* propose deux types de sorties.

La première, en temps réel, se présente sous la forme d'un affichage sur la console jMax indiquant qu'une erreur a été detectée, ainsi qu'une sortie sous forme de *bang* dans le *patch* jMax (ce type de sortie est utile si l'on souhaite chainer l'objet avec d'autres objets, des compteurs par exemple).

La seconde est un rapport des erreurs jouées, édité et présenté à la fin de l'interprétation. On y trouve un résumé des conditions de la détection (valeur des paramètres de réglage, types d'erreurs à détecter...), une synthèse du nombre de fautes jouées, classées selon leur type, et enfin la liste exhaustive de toutes les erreurs classifiées avec l'explication correspondante (voir annexe C).

[1] C'est la manière la plus simple d'acquérir le signal joué, puisque ces deux paramètres sont disponibles dans le format standard MIDI. A noter que chaque liste ne correspond pas à une note, mais à un évènement de type début ou fin de note (Note-on/Note-off), une note-off étant caractérisée par une vélocité égale à zéro

[2] Ces données sont envoyées en sortie de l'objet de suivi de partition *suivimidi.* Ils sont récupérés en chainant les objets dans le *patch* jMax

6 Résultats

Les résultats obtenus lors des différentes phases de tests sont donnés selon trois coefficients, chiffrés de la manière suivante :

- a = pourcentage d'erreurs détectées par rapport aux erreurs effectuées,

- b = pourcentage d'erreurs correctement classifiées par rapport aux erreurs effectuées,

- c = pourcentage d'erreurs détectées en trop par rapport aux nombre de notes jouées justes.

6.1 Tests de base avec erreurs simulées

Ces tests utilisent la même partition de base, à savoir un morceau de 8 notes et 1 silence, et couvrent tous les cas possibles d'erreurs. Les performances utilisées ont été générée artificiellement, avec insertion d'erreurs. Les résultats sont présentés figure 3.

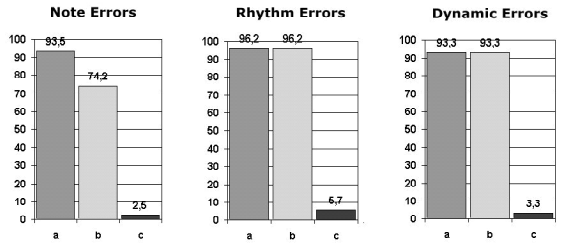

FIG. 3 – *Résultats sur tests de bases répartis selon le type d'erreurs*

6.2 Tests sur morceaux complexes

Deux morceaux ont été utilisés pour ces tests : un extrait de la *Czardas* de Monti, ainsi qu'une version de la *danse Polvtsienne N.17* de Borodine (voir les partitions présentées en annexe D). Le premier est un morceau rapide et difficile, le second lent et expressif. Ces deux extraits permettent de valider les résultats obtenus sur morceaux simples, en observant l'influence de la difficulté et de la longueur du morceau sur la détection de fautes.

6.2.1 Erreurs simulées

Les résultats ne sont pas très différents de ceux obtenus avec les tests de base. La durée et la difficulté du morceau ne semblent donc pas interférer dans les résultats, en tout cas avec des interprétations simulées.

6.2.2 Interprétations réelles

Les tests effectués à partir des deux morceaux précédents joués au clavier donnent de moins bons résultats : le taux de détection est de l'ordre de 75%, et le taux de détection + bonne classification est approximativement de 57%.

La première chose à constater est la différence liée à la difficulté du morceau. Plus celui-ci est rapide et complexe, plus l'alignement est incertain, et en tout cas imprécis, surtout si l'interprétation est également imprécise. En second lieu intervient la notion de "propreté" du jeu : la partition modélise des notes de durée exacte, sans recouvrement et sans silences entre elles. Un jeu réel, même parfait, ne réalisera jamais cette condition, ce qui rend une fois de plus l'alignement incertain. Ainsi, il est possible de recevoir un *g-state* en sortie du suiveur de partition même si aucune erreur n'a été effectuée.

6.3 Synthèse

Les résultats peuvent être présentés comme suit :
série 1 = pièces courtes, performances simulées,
série 2 = pièces complexes, performances simulées,
série 3 = pièces complexes, performances jouées.

a	b	c
a1 = 94,4%	b1 = 86,1%	c1 = 4,3%
a2 = 87,2%	b2 = 77,7%	c2 = 5,2%
a3 = 75,0%	b3 = 57,1%	c3 = 10,9%

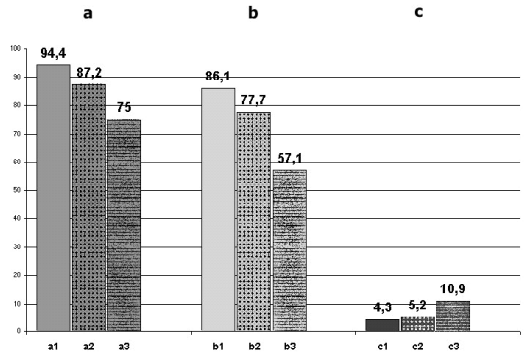

FIG. 4 – *Résultats présentés par séries de tests*

Ces résultats montrent bien le traditionnel décalage entre les situations idéales (erreurs simulées) et les situations réelles (interprétation des morceaux). Néanmoins, les premiers résultats permettent de valider le modèle théorique choisi et les algorithmes utilisés. De nombreuses améliorations sont possibles en rendant ces algorithmes plus robustes aux diffucultés du jeu réel.

7 Futur - Perspectives

7.1 Cas non traités

Un certain nombre d'erreurs ne sont pas encore traitées ou bien le sont mal. Une amélioration dans ce sens peut faire l'objet d'un développement futur. Certaines de ces erreurs non traitées sont les erreurs définies au début du projet et dont la détection et la classification n'ont pas été implémentées dans la version 1.0 du package VMT, comme par exemple les erreurs de *legato/staccato*. Il y a également les cas problématiques de détection que représentent les nombreuses erreurs consécutives, les erreurs en fin de morceau, les erreurs doubles sur une même note (erreur de note et erreur de rythme par exemple).

7.2 Flux d'entrée audio

La plus grande partie du travail a été effectueée en utilisant un flux d'entrée de type MIDI. Paralèlement, un autre objet (*audiodetector*) est en cours d'implémentation afin d'utiliser une entrée de type flux audio pour acquérir la performance.

L'idée est d'utiliser l'objet de suivi de partition *suiviaudio*, c'est à dire de travailler directement sur les paramètres audio du signal sans avoir à passer par une modélisation de type MIDI de la partition : ainsi pour chaque note, nous n'aurons plus de données telles que hauteur ou vélocité, mais le résultat d'un filtre harmonique et des ratios d'énergie. La comparaison est alors possible entre le spectre extrait du signal d'entrée par transformée de Fourier et les filtres de bandes harmoniques de chacune des notes de la partition. Ce type de comparaison est déjà utilisé pour le suivi de partition audio (voir [10]).

7.3 Améliorations

7.3.1 Regroupement d'erreurs

Certaines erreurs peuvent être classées de plusieurs façons, et par regroupement d'erreurs simples il est possible de donner une explication plus précise sur ces erreurs.

C'est le cas par exemple des erreurs de rythme répétées sur des notes rapides et régulières, qui

pourront être interprétées comme une seule erreur du type "manque de régularité".

Pour cela, il pourrait être possible de faire une seconde classification une fois la première classification terminée : il s'agirait en quelques sortes d'un mécanisme de supervision, qui chercherait un lien logique entre les erreurs jouées.

Il pourrait donc être intéressant d'envisager, dans le futur, un système de diagnostique d'erreurs en trois phases : détection - classification - regroupement.

7.3.2 polyphonie

Le système actuel ne permet pour l'instant de traiter qu'un morceau monophonique. L'application à la polyphonie est une direction à privilégier, surtout dans le cas du detecteur à entrée MIDI (la pupart des instruments MIDI sont des claviers, donc des instruments à jeu polyphonique).

La difficulté de l'adaptation tient surtout au fait que les indicateurs d'erreurs sont beaucoup plus ardus à reconnaitre. En effet, une seule fausse note dans un accord ne change pas forcément l'alignement dans la mesure où la majorité des notes sont justes. De plus, la correspondance n'est plus "un état = une note", mais "un état = un accord". Comment, dès lors, déterminer la note de l'accord qui est fausse ?

7.3.3 souhaits des utilisateurs

Lors de la mise en place du projet, un questionnaire à été diffusé auprès de professeurs de musiques et de potentiels utilisateurs. La plupart de leurs souhaits ont servi de bases au projet. Deux souhaits exprimés ont cependant été mis entre parenthèses et laissés comme idées pour un développement futur, car quelque peu éloignés du but principal du projet.

Il s'agit d'une part de la possibilité d'écouter son erreur en comparaison de ce qu'il aurait fallu jouer, et d'autre part d'ajouter une fonction séquenceur à l'objet pour permettre aux pianistes de jouer une main après l'autre, avec l'une des deux parties en accompagnement.

L'intérêt pédagogique de ces deux fonctions est indéniable, néanmoins leur réalisation s'éloigne du thème de la détection et classification d'erreurs qui faisait l'objet du présent projet. En l'occurrence, ces ajouts au programme sont suffisemment indépendants pour être réalisés à tout moment, par une autre personne ou un autre groupe de recherche par exemple.

8 Conclusion

Ce Projet de Fin d'Etudes ne devrait être qu'une partie du projet VMT lui même. Il reste évidemment beaucoup de choses à faire : une version 1.0 conçue et réalisée en six mois ne peut pas répondre de manière complète à tous les besoins spécifiés, surtout si l'on considère que ces besoins sont en constante évolution.

Le package jMax implémenté n'est pas encore prêt à servir dans les foyers, mais c'est un outil de recherche dont les premiers résultats encouragent à persévérer.

A Captures d'écran

Les figures suivantes présentent les deux types de fenêtres de jMax en fonctionnement : la console avec ses affichages pour l'utilisateur, ainsi qu'un *patch* (ici, l'un des *patches* utilisés pour les tests sur morceaux réels).

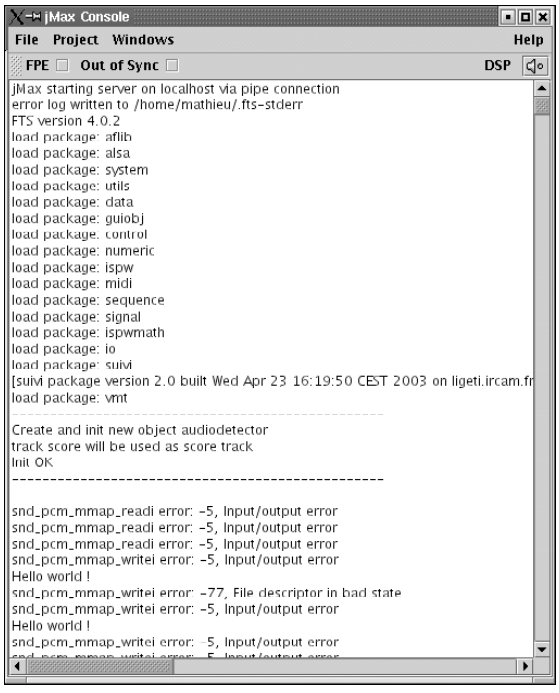

FIG. 5 – *La console jMax*

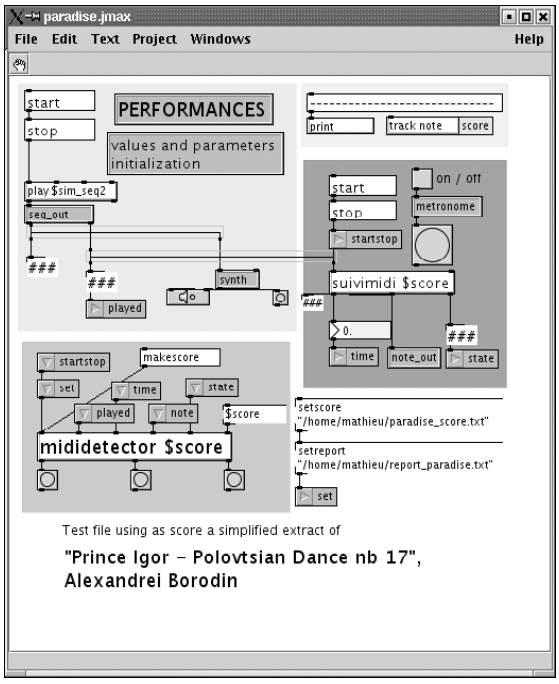

FIG. 6 – *patch utilisant les objets de suivi et de détection*

B Partition au format texte

Les événements de la partition sont codés dans le fichier texte de la manière suivante : le premier chiffre est le code de l'événement (0 pour une note, 1 pour un controleur de tempo, etc.). les valeurs suivantes dépendent du type d'événement.

Pour une note, il y aura le *pitch* MIDI, puis la valeur musicale de la note codée sur deux nombres (1 4 pour une noire, 1 2 pour une blanche, 1 8 pour une croche, etc.), puis le paramètre d'accentuation (0 ou 1), puis la position en [mesure, temps, fraction].
Exemple : pour coder un do#5, noire non accentuée au 3ème temps de la mesure 5, nous allons écrire :

```
0  64  1 4   0  5 3 0
```

Pour un controleur, il y aura sa valeur (sur un ou deux nombres selon le type du controleur), et sa position en [mesure, temps, fraction].
Exemple : nous codons ici en début de morceau une mesure à 3/4 ainsi qu'un tempo égal à 120 battements/seconde :

```
2  3    4        1 1 0
1  120  0        1 1 0
```

La fin de la partition est signalée par -1.

C Rapport d'erreurs

Ci-dessous, un exemple de rapport édité au format texte par l'objet mididetector.

```
-------- REPORT ON YOUR PLAY ----------
Detection of note errors allowed
Detection of rhythm errors allowed
Detection of dynamics errors allowed

Duration and rhythm tolerance used :
10.000000 percent

---------------------------------------

2 ERRORS HAVE BEEN DETECTED IN YOUR PLAY
(1 note error(s),
1 rhythm error(s),
0 dynamic error(s))

--------
- ERROR NUMBER 1 :
Note error : you played a wrong note
at measure 2, beat 1, fract 0 :
You played a C5 instead of the expected D#5

--------
- ERROR NUMBER 2 :
Rhythm error : note (or rest) too long
at measure 2, beat 3, fract 0 :
F5 Was too long of 1 beat and 0 fract.

---------------------------------------
```

D Partitions utilisées pour les tests

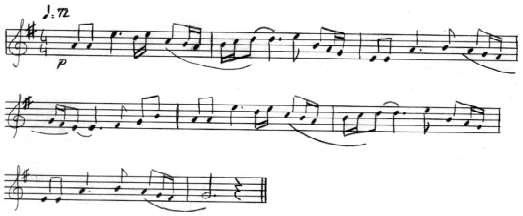

FIG. 7 – *A. BORODINE, "Danse Polvtsienne n.17", dans "Prince Igor" - Extrait*

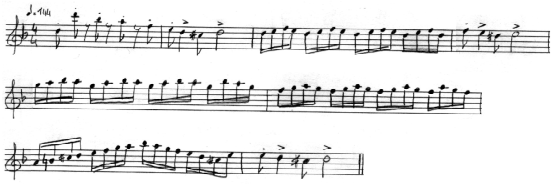

FIG. 8 – *V. MONTI, "Czardas" - Extrait*

Références

[1] P. Cano A. Loscos and J. Bonada. Score-Performance Matching using HMMs. In *Proceedings of the ICMC*, pages 441–444, 1999.

[2] C.Raphael. Music Plus One : A System for Expressive and Flexible Musical Accompaniment. In *Proceedings of the ICMC*, Havana, Cuba, 2001.

[3] N. Orio D. Schwarz. *Score Following : project report.* IRCAM, Paris, 2002.

[4] F. Dechelle. jMax : un environnement pour la réalisation d'applications musicales sur Linux. In *Journées d'informatique musicale*, Bordeaux, France, 2001.

[5] M. de Cecco E. Maggi J. Rovan N. Schnell F. Déchelle, R. Borghesi. jMax : An Environment for Real-Time Musical Applications. *Computer Music Journal*, 23(3) :50–58, 1999.

[6] G. Tisseau J. Duma M. Urtasun H. Giroire, F. Le Calvez. A mechanism for incremental error detection and its application to a pedagogical system. In *Congrès de Reconnaissance des Formes et Intelligence Artificielle*, pages 1063–1072, 2002.

[7] A. Loscos, P. Cano, J. Bonada, M. de Boer, and X. Serra. Voice Morphing System for Impersonating in Karaoke Applications. In *Proceedings of the ICMC*, 1999.

[8] V. Mouillet. *Prise en compte des informations temporelles dans les Modèles de Markov Cachés.* IRCAM, Paris, 2001.

[9] N. Orio and F. Déchelle. Score Following Using Spectral Analysis and Hidden Markov Models. In *Proceedings of the ICMC*, Havana, Cuba, 2001.

[10] N. Orio and D. Schwarz. Alignment of Monophonic and Polypophonic Music to a Score. In *Proceedings of the ICMC*, Havana, Cuba, 2001.

[11] L.R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2) :257–285, 1989.

[12] C. Raphael. Automatic Segmentation of Acoustic Musical Signals Using Hidden Markov Models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(4) :360–370, 1999.