Virtual Music Teacher

Project documentation

—

INSA de Lyon, Département Informatique

–

Universität Karlsruhe, Fakultät für Informatik

(Institut für Logik, Komplexität und Deduktionssysteme)

–

IRCAM - Centre Pompidou

Gilles Mathieu
Engineer Student
Computer Science Department, INSA de Lyon
sotww@yahoo.ie

March - August 2003

Abstract

Computers offer a great applications panel, and an important feature 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 realise such one which would allow computer aided music learning. This software does not give musical theory base, which can be found in any music book, but build 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 explain them.

Typical errors are for example wrong notes, rhythm errors, tempo errors, and interpretation errors. Applications could be everykind of tonal music (instrumental or vocal).

That implies good definitions of errors and diagnostic rules, but that also means an investigation on interface problems: way to acquire and to model played music, output definition, etc.

This report describes the project base 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.

Contents

Chapter 1

Project base

1.1 Motivation - Project Goal

1.1.1 Music, teaching and computer science

Computer science is useful in many domains : information systems, communication, security... and art. Many artists have used computers to create or perform their own creations. In those cases, they become powerful creation tools.

Using computers for music is for me one of their most interesting features, since I studied both music and computer science.

Another great application of computers is the way we can use them for teaching. As they are able to provide many education tools (interactive learning, mathematics, science...), we can imagine that in the next years a lot of things will be taught and learned with computers.

Then, realise a **computer-aided music learning system** seems to be a good idea...

1.1.2 Why a "Virtual Music Teacher" ?

Since more and more families have a personal computer at home, why couldn't they use it to learn music ?

The first time I told a musician about my intentions to realise a *virtual music teacher*, he seemed to be a bit shocked and answered me : "*Why ? Do you want the musicians to become robots ?*". It is then important to say that our goal is not to replace a real music teacher: music will always need feelings, that are so far not easy to express with a computer. *Virtual Music Teacher* means a **system which would help musicians to train at home, by detecting and explaining their errors**.

So far, there are **no existing systems** in this area. We can find a lot of teaching softwares, error detection systems and music software, but none allows to realise error detection on a music piece. I hope this project will help for future work in the domain.

1.2 Background - Existing systems

1.2.1 Score Following

An important part of the project consists on comparing a performance with a score: The matching of differences between them will lead us to errors detection. Such systems have been implemented for other purpose, mainly for real-time and interactive music. Known as score-following systems, their goal is to synchronise live instrumental music with computer activated samples, what is in fact very close to our goal: to activate samples, we need to know exactly when and how the expected note was played. These data could be compared to the expected ones in order to detect potential errors.

Score following for jMax - IRCAM, centre Pompidou (Paris, France)
jMax is a visual programming environment for interactive real-time music and multimedia, developped at IRCAM. For our purpose, the most important feature of jMax among all the different packages which have been so far implemented is the Score Following Package. Based on Hidden Markov Models, this package is being implemented for 2 years in the real-time system team at IRCAM. The score is modeled as a markov chain, with 2 states for each note or chord: One state for the right note (normal state, or n-state), one state for anything but the right note (ghost state, or g-state), and transitions between states, with their own probabilities, as described in [OD01], and shown on figure 1.1 (see [Rab89] for more informations about Hidden Marko

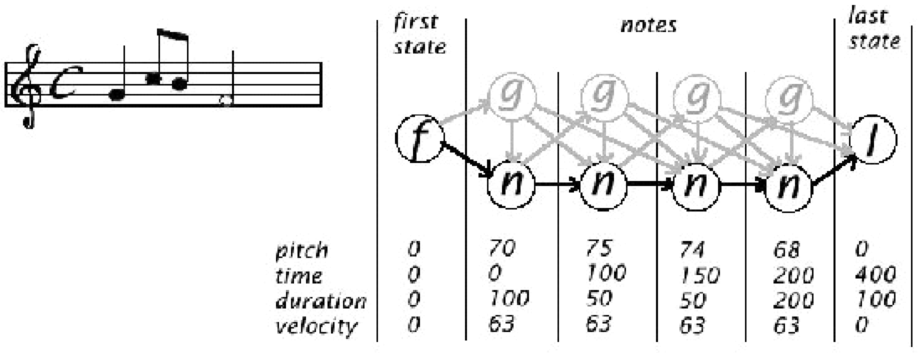


Figure 1.1: A score and its model

This states succession is very interesting for our purpose: if we know which state has been reached, we have the possibility to know where an error occured... Furthermore, it is possible to extend this model: type of used datas, other parameters, etc.

Two objects have been implemented for score following: **suivimidi** and **suiviaudio**, adapted w.r.t. to MIDI and audio input. For both, played flow is considered as the observation and matching is done with a classical decoding using Vitterby algorithm (see [ALB99]). Improvements have been made by using temporal parameters in HMM ([Mou01]), and by creating a new model for polyphonic music ([Mat02]).

It would be a good solution to use jMax and its Score Following Pqckage for the project. The best way to do it would be to implement a *Virtual Music Teacher package* in jMax. The main object of this package would have as input the output of the Score Follower.

1.2.2 Speech Recognition

As we can consider that music is a kind of language, the language recognition approach could be used for music recognition:
- get data (speech signal / musical signal)
- extract reliable features (dynamic, spectrum, ceptrum, ...)
- model (diagrams, trees, neural networks...)
- compare (expected text in a dictionnary / expected sequence in a score)

Furthermore, as a speaker adaptation is used in speech recognition systems, we can have an instrument adaptation in music recognition system, in order to improve the acuracy of the matched performance.

The speech recognition approach has been already used for the HMM score Following described above.

Chapter 2

Definitions

2.1 User study

In order to define exactely some points and to see what music teachers think of the project, an enquiry has be led (see complete enquiry in appendix A). 19 teachers (music school teachers, independent music teachers and university teachers) have answered. This section presents the results.

- 18 of them think that the system could be useful, but only 12 would like to use it with their students.
- 5 of them want an immediate signalisation when an error occurs, 3 of them want a report at the end, 10 want both.
- 3 of them think this system will be usefull for beginners and medium level musicians, 4 for medium level, 6 for medium and high level, 5 for all levels.
- All of them think the performance must be compared with a "fixed and perfect" score for beginners and low level musicians, and with a flexible score for high level musicians.
- In te report should appear : error location and a short explaination, eventualy statistics.
- Note errors should be detected first, then rhythm errors, and eventualy dynamic errors.

Definitions given in this chapter come from this study.

2.2 Input/Output definition

Our system must get data from the played music and output its diagnostic. Which kind of data will be handled ? We have to know exactly which informations we'll need to perform matching between the score and the performance. What will the output look like ? "diagnostic" is quite a fuzzy word...

2.2.1 Input Data

Input flow is played music: it means it can be mainly of two different kinds: Audio or MIDI. We'll describe those data representations and discuss pros and cons for each of them.

Audio Data

Audio data present a great advantage: as it is a very rich "format", we can have all the informations we need to evaluate music. It is by the way very general, i.e. everykind of music source could be used as input signal through an audio acquisition: piano, flute, guitar, but also voice or percussions.

The number of useful parameters can be very high, and considerabely improve results: Harmonic spectrum, enveloppe spectrum, energy, zero-crossing...
Since most of these parameters are instrument-depending, We can significantly increase system acuracy by building a different model for each kind of instrument.

On the other hand, audio data require space and time: real-time calculations on audio data means an optimal use of computer's ressources.
Another problem would be the possible lack of precision: for example, detecting the start time of a note can not be done without an error rate. This could be mainly due to signal perturbations (noise, low level signal...), especialy on weak attacked notes.

MIDI Data

Unlike audio, MIDI data allow few, but very precise parameters: pitch, duration, velocity, time... Of course, in real-time played music, duration is only known a posteriori. But the NOTE-ON/NOTE-OFF mechanism allows us to detect very simply note's start and note's end.
Another great advantage can be found in polyphonic music: we have, with MIDI, all the informations, while polyphonic audio data are more difficult to use.

The greatest problem with MIDI is the limited number of instruments that can product it. Except Keyboard instruments (MIDI-Piano, Organ, MIDI-accordeon...), the produced MIDI is not very clean. There are mainly two ways to produce MIDI for non-keyboard instruments:

Pitch trackers:
The audio signal is acquired, analysed, and MIDI parameters are extracted and outputed. Well working on simple signal, but often produce errors with harmonics on attack, or with special

musical features like vibrato, tremolo or tongue-flat for wind instruments.

Instrument "midification":
Captors are placed on the instrument to capture the mechanic features of the note without having to analyse it: for example MIDI-captors on guitar strings, mechanic captors on flute keys... The problem is always a lack of precision: for the guitar, note's beginning is easy to determine, but note's end is threshold depending. For the flute, attacks are detected by a microphone near the embouchure, and pitch with keys positions: the problem is we are not able to detect octaviations, since keys positions are the same for octaved notes, for example A4(440Hz) and A5(880Hz).
Furthermore, such instruments are rare, expensive, and very often heavy and hard to play because of all the "added mechanisms".

Our Choice

Choice criteria could be summarised as below:

CRITERION	**AUDIO**	**MIDI**
Space requirement and treatment duration	high	**low**
Complete data	**very good**	average
Precise data	average	**very good**
Noise and interference tolerance	low	**high**
Coding treatment functions	quite difficult	**quite easy**
Could be applied to everykind of instrument	**yes**	no
Allow system to be adapted to the source instrument	**yes**	no
Could be applied to polyphonic music	with difficulty	**yes**

A choice is difficult because of the complementarity of both data types. Moreover, the Score Following system described in document "Background - Existing Systems" gives two kinds of Score Followers: one with Audio Input, the other with MIDI input.

We can't make a choice without knowing which kind of music will be played. If we have for example a Chopin's piece, i.e. a polyphonic piano piece, we will obviously choose MIDI as input, whereas if we have a study for flute, i.e. a simple monophonic piece, audio will be more efficient.

Another (and the last) choice criterion will be the time we will have to realise the system. Since input data type does not affect other functions too much, we will focus on only one type, as the first realisation step, and then consider the second input type as a program extention. MIDI data allow to test the system without having to store a big amount of data. Moreover, we can easily produce a MIDI-score and a test MIDI-performance with artificial errors, which is more difficult with audio.

Our first goal will be to use MIDI data as input for the system. Detection algorithms will be implemented and tested first with a MIDI input, and only then extended to audio.

2.2.2 Output Data

Main goal of the project is to indicate where errors have been done and the kind of these errors. We have to find an output which would allow to reach this goal. There are two ways to indicate errors: immediately when they are done, or at the end of the performance with a report of all played errors.

Real-time Signalisation

As soon as an error has been done, we may want to have a feedback: "you have made an error!". This feedback could be a sound, a visual signal on the screen, or anything else that could indicate to the player that what he played just before was wrong. This cannot be done for everykind of errors: dynamic errors, for example, need a large "window" to be detected, and a real-time indication like "you have played too loud" would not be relevant.

For didactic purpose, it would be a good idea to indicate errors immediately: the players know what he just played, and doesn't have to remember "what did I do meas.37 ?". But the related problem would be the following one: it's sometimes difficult for a beginner to stop and then continue to play from a given point in the score. Moreover, a real-time signalisation could definitely disturb the player, and make him do much more errors because his attention has been distracted.

General Report

When the performance is over, all detected errors can be indicated. The output could be a text file, a screen text, an interactive dialogue... and it can contain all the informations we need about the played errors: when and where the error was done, its kind, the comparision between what was played and what should have been played, etc.

Our Choice

Since Music teachers disagree on the best way to indicate errors (half of them for each possibility), the best solution would be to implement both of them, and give the choice to the user: enable/disable real-time signalisation, and give the possibility to have the report or not.

2.3 Evaluation Criteria

We have to define which kind of errors will be handled by the system, and discuss about the program's tolerance.

2.3.1 Error types

Note errors

Wrong Notes
The note played in the performance is not the expected one: [A, C#, F] instead of [A, C#, E]. The error criterion is the pitch of the note.

Skip Notes
A note is missed by the performer: [A, C#, E] instead of [A, C#, F, E]. Error criteria are the pitch and the time of the note.

Extra Notes
A note is added by the performer: [A, C#, G, F, E] instead of [A, C#, F, E]. Error criteria are the pitch and the time of the note.

Tempo and Rhythm errors

General tempo errors
A part of the piece is played on a different tempo as the one expected. This feature should be chosen by user: the performer may want to train on a part while playing it slowly.

Duration errors
Played note duration is note the one expected: [100, 50, 100] instead of [100, 100, 100]. Error criterion is relative release time ([release time] - [start time]).

Start time errors
A note is played earlier or later than expected [0ms, 100ms, 200ms] instead of [0ms, 200ms, 400ms]. NOTE: we must not detect a start time error just after a duration error, since they are often related: this would lead us to detect 2 errors instead of only one.

Rest errors
A rest that should be made is missed, or an unexpected rest is added. Rest errors are linked to start time errors and duration errors. We can group them into 4 categories: note too long, note too short, added rest, skip rest.

Lack of regularity
Regular repeated notes are played with different times: [0, 50, 90, 150, 210, 250] instead of [0, 50, 100, 150, 200, 250]. Such errors are a kind of start time errors, but their musical signification is important enough to class them into a particular category.

Dynamics and phrasing errors

General dynamics errors
A part of the piece is played louder or softer than expected: mp instead of mf. Error criterion is note velocity.

Accentuation errors
Same thing as dynamics errors, but mainly for single notes. For example, a "should-be-accentuated note" is played with the same dynamic as the others.

Staccato/legato errors
A staccato (resp. *Legato*) phrase is played legato (resp. *Staccato*). Could be classified as duration errors, but have enough musical signification to be one distinct category.

A posteriori classification

Error classification could not (and should not) be made in real-time. Let's take the note errors as example:
If a "wrong note" is detected, before determining which kind of note error we have, we must wait to see what happens next. If next played note is the one expected after the one detected as wrong, then we have great probability to have a wrong note error. If the note detected as wrong is the one expected just after, we can consider it as a skip note. And if the note played after the wrong one is the one which was expected, then it can be handled as an extra note.

This implies we must have a quite large working window to see clearly what happens arround the note which is actually played.

2.3.2 Tolerence - Threshold values

Note pitch (Note errors)

For a MIDI input, we don't have many difficulties to define what is a right note and what is a wrong one: The pitch of the played note is not exactely the same as the expected one, then we have a wrong note.

It's a little bit more difficult with an audio input: We must in this case define a *frequency threshold* δf and consider we have the right note if its frequency is between $(f - \delta f)$ and $(f + \delta f)$

Note and Rest duration (Rhythm errors)

Things cannot be here defined in a "binary way": We must introduce a threshold δ and consider that the played duration d is right if we have:

(expected duration - δ) $<$ d $<$ (expected duration + δ)

Threshold value should depend on note's own duration, and not exceed 50% of this duration. The precise definition of this value will depend on what the player (or the teacher) wants: it can not be the same value for a beginner and for an experienced musician, and from a musical piece to another. Let's take an example:

A piece of J.S. Bach, with a lot of repeated and regular notes, has to be played with a strong exactitude. The δ value should not be too high. On the other hand, in a piece of Chopin where player's own "feeling" affects the general rhythm, we should not consider that would be an interpretation as an error. The threshold value should be here higher.

A good compromise would be to give the possibility to the user to define the value corresponding to the musical piece. This threshold value would become a parameter of the score, and directly correspond to the musical style of the piece.

Dynamics - Note velocity

General dynamics errors
This becomes here much more subjective. We have two ways to detect these errors: First, we can have the same idea as for note duration: compare the played note's velocity to the expected one, and validate it with a threshold:

$(expected\ velocity - \delta) < v < (expected\ velocity + \delta)$

The second way would be to define a "velocity scale" which would give, for each velocity value, the corresponding dynamic:

Between 0 and 10 : *ppp*
Between 10 and 20 :*pp*
Between 20 and 40 :*p*
Between 40 and 60 :*mp*
Between 60 and 80 :*mf*
Between 80 and 100 :*f*
Between 100 and 120 :*ff*
Between 120 and 127 :*fff*

The great advantage of this solution is an independant definition of dynamics. If rhythm is a fuzzy concept and can be different from a piece to another, a mezzo-forte will always be a mezzo-forte, and this notion is totally independant of the musical style of the piece.

Accentuation errors
We must here consider note velocity in comparison with both previous and next note. If the note should be accentuated, then we can say we have no error if:

$(v > Velo(previous\ note) + \delta\ )\ AND\ (\ v > Velo(next\ note) + \delta\ )$

For accentuation problems, we just need relative velocity: we don't care about absolute values, since the wanted effect of an accentuated note is to make contrast.

2.4 Score Definition

2.4.1 Needed data

We have defined which kind of errors will be detected. We must now define all data we need to reach this goal.

Note errors

Essential datum is the **pitch** of the note. We will use the MIDI definition of the pitch: an integer value between 0 and 127, based on C3 = pitch 60.
But we also need the **position** of the note to determine which kind of note error we have (wrong note, skip note or extra note. See "evaluation criteria" for more details). Position will be defined as a three values structure: *[measure number, beat number, beat fraction].*

Rhythm and Tempo errors

We need a **tempo parameter**, which could be different from a part of the piece to another (precise tempo change, accelerando...). We will take tempo values between 0 and 256, which will correspond to the number of beats pro minute.
Note position and **note duration** are also needed for start time errors, duration errors and lack of regularity errors. Duration will be the "relative" duration of the note, i.e. the note value (1, 1/2, 1/4, etc...) linked to the current tempo.

Dynamics and phrasing errors

We need for each note its velocity to find accentuation errors. We will use the MIDI definition of the **velocity**: an integer value between 0 and 127.

For general dynamics errors, a **global dynamic parameter** would be better (ex: mf from meas.21 to meas.27, f from meas.27 to meas.28, etc.). Values will be taken between:
[ppp, pp, p, mp, mf, f, ff, fff].

For the *staccato/legato* errors, we need a **special parameter** which would allow to indicate in the score how the part shall be played. Value of this parameter will be chosen between three values: [staccato, normal/undefined, legato].

Thresholds and user defined parameters

We said in the "evaluation criteria definition" that thresholds could be used to determine which tolerance level would be used for each piece. Since these threshold values are depending on the piece, they can be linked to the score. We have for the moment 2 parameters: **duration tolerance parameter** and **accentuation parameter**.
As we have said for the tempo parameter, those threshold values can change from a part of the piece to another (we may want to indicate in the score a "recitative" part, where the duration tolerance would be very high, for example for 2 or 3 measure, and then come back to a low tolerance).
The duration tolerance will be defined as a number between 0 and 100, corresponding to the

tolerance percentage. The accentuation parameter will be a binary parameter: 1 if accentuated, 0 if not.

2.4.2 jMax Sequence object

The jMax Score Following object uses a jMax sequence object as reference score. This object is a derivation of a MIDI file: mainly used data are, for each note: pitch, duration, time and velocity.

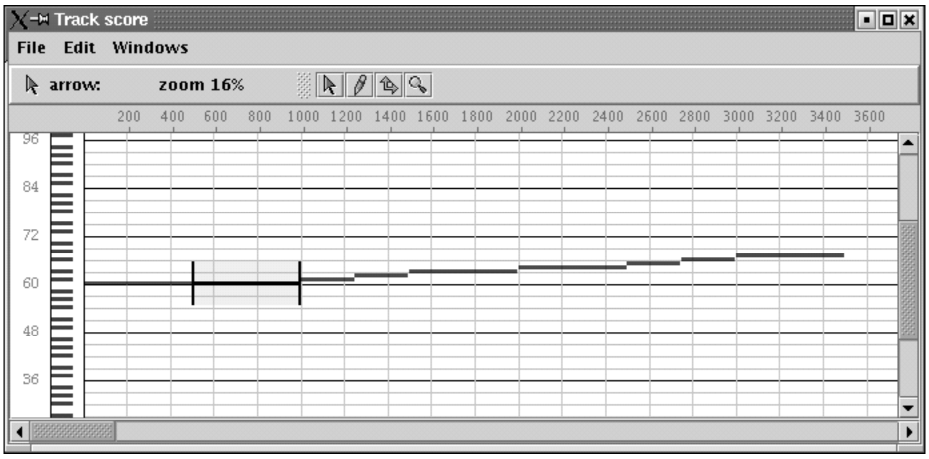


Figure 2.1: A jMax Sequence Object

If we want to use the Score Following Package without having to make changes in it, a part of our score shall be a jMax sequence. It is possible to build a jMax sequence from a MIDI file. But we need other data: the jMax score will not be enough to store all the information we want. We will have to join another file to it in order to have the complete score we need.

In order to avoid having to fill 2 files for the same score, the idea would be to make the jMax sequence object from this joint text file

2.4.3 Score text file

In this file, we will have to store: note pitch, note value, note position, note accentuation, tempo, time signature, dynamic, legato parameter and duration tolerance.

Note fields

Value, position, and accentuation are linked to each note. One part of our file will be the list of all notes with those parameters, like:

```
pitch   value   position   accent
60       1/4    1:1:00      0
62       1/8    1:2:00      0
64       1/8    1:2:48      1
```

```
57        1/4    1:3:00      0
```

Parameters - controlers

As in a MIDI file, we can set parameter values with "controlers": one controler type for each parameter to be set. For example:

```
Type         value     position
Time sig.    3:4       1:1:00
Tempo        120       1:1:00
Dur. Tol     20        1:1:00
Dynamic      mf        1:1:00
Legato       1         1:1:00
Dynamic      f         3:2:00
Dynamic      mf        4:1:00
...
```

File structure

We can fill our file as a sequence: notes and controlers are placed in the order they shall appear in the score.
The problem we will have here is that the score won't be very simple to fill for unexperienced users. In the future, it would be a good idea to create a tool to fill it. But let's forget it for the moment.

For more details about precise file syntax, see *chapter 2*.

2.4.4 Data overview

Used data

Some data are only used by the score follower and the intern model of the program, and some are only to give informations to the user:

DATUM	TYPE	USED BY
Note pitch	scaled(0-127)	ScoreFollower + Model
Note duration	time in ms	ScoreFollower + Model
Note value	predefined	User
Note start time	time in ms	ScoreFollower
Note position	predefined	User
Note velocity	scaled(0-127)	ScoreFollower + Model
Note accentuation	predefined	User + Model
Tempo	scaled	Model
Time signature	predefined	Model
Dynamic	predefined	Model
Legato	predefined	Model
Duration Tolerance	scaled(0-100)	Model

Data values: more details

Note value:
defined as a fraction of a 4/4 measure.

Note position:
3 values structure: [measure number, beat number, beat fraction]. The beat number is linked to time signature. Beat fraction is an integer between 0 and 95 (48 is the center between 2 beats. 96 values is the best solution to have a precise value for 4-multiple and 3-multiple note values.

Tempo:
numerical value corresponding to the number of beats per minute.

Time signature:
2 values, corresponding to the number of beats in the measure, and the note value that fills a beat (musical definition of time signature).

Dynamic:
integer value between 0 and 7, with the following correspondance: 0 for *ppp*, 1 for *pp*, ... 6 for *ff*, 7 for *fff*.

Legato:
integer value: 0 for *staccato*, 1 for normal/undefined, 2 for *legato*.

Duration tolerance:
numerical value between 0 and 100: 50 will mean we will detect no error if note is played between (t - note duration * 0.5) and (t + note duration * 0.5).

2.5 General program structure

Here is the general structure of the system.

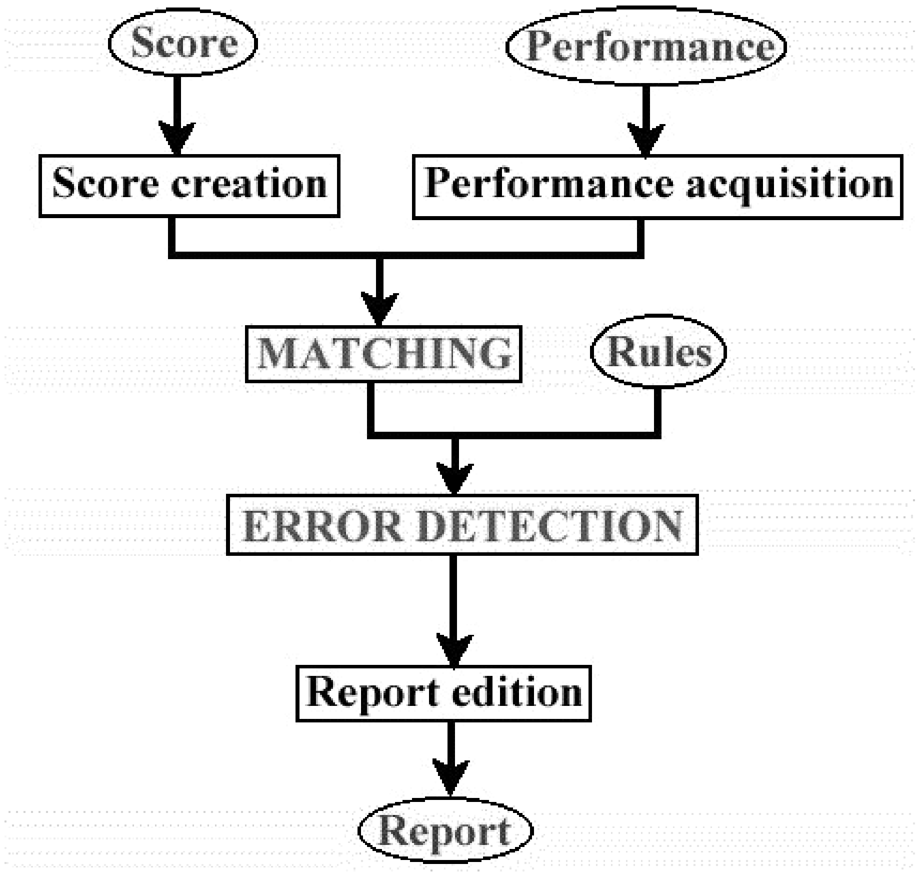


Figure 2.2: General program structure

Score creation and performance acquisition will be described in chapter 3, matching and error detection in chapter 4, and error signalisation in chapter 5.

Chapter 3

Score and performance acquisition - model creation

We have to find a way to acquire and creat a model for score and performance, in order to compare them. This chapter describes how this acquisition and modelisation has been realised.

3.1 Score acquisition

3.1.1 Score text file syntax

The file used as input to build the score is a text file (.txt format). in this file are stored all the events and their properties, with the following syntax:

```
Event_type [properties]
Event_type [properties]
...
END_OF_FILE
```

There are 6 different event types:
0 : note
1 : tempo
2 : time signature
3 : legato parameter
4 : duration tolerance
5 : dynamic
For each type, properties are described as shown:

```
note:      pitch, note_value(2 numbers), accentuation, measure, beat, fraction
tempo:     value,    0,         measure, beat, fraction
time_sign: up_value, low_value, measure, beat, fraction
legato:    value,    0,         measure, beat, fraction
dur_tol:   value,    0,         measure, beat, fraction
dynamic:   value,    0,         measure, beat, fraction
```

For a note, the two number of note_value are resp. the upper and the lower value of the duration fraction. Pitch, measure number, beat number and fraction values are like described in chapter 1. Rests are also coded: they are represented by notes with pitch 128.

example: for a C#5, 1/8 note, not accentuated, at measure 2, beat 3, fract 48, we will write:

```
0 61 1 8 0 2 3 48
```

END_OF_FILE is coded by a -1. Separators between values are blank spaces.

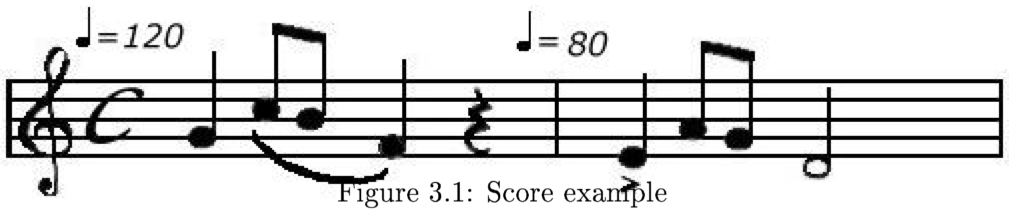


Figure 3.1: Score example

For example, the score presented here will finaly be coded like:

```
2  4    4          1 1 0
1  120  0          1 1 0
3  0    0          1 1 0
4  10   0          1 1 0
5  5    0          1 1 0

0  67   1 4  0     1 1 0
3  1     0         1 2 0
0  72   1 8  0     1 2 0
0  71   1 8  0     1 2 48
0  65   1 4  0     1 3 0
3  0     0         1 4 0
0  128  1 4  0     1 4 0

1  80   0         2 1 0

0  64   1 4  1     2 1 0
0  69   1 8  0     2 2 0
0  67   1 8  0     2 2 48
0  62   1 2  0     2 3 0
0  67   1 4  0     3 1 0

-1 -1 -1 -1 -1 -1 -1 -1
```

Of course it would take a while to code a long and complex score with this syntax (imagine how the “Toccata and fugue in Dm” from J.S.Bach would look like...). But building a tool which would allow to create the text file easier is another subject...

3.1.2 Data read from text file

When requested by sending the message “makescore” to the detector, those data are read by the function detector_create_score_from_file(), which is defined as a standard jmax inlet method (for more details about jMax and jMax objects, see [FD99], [Dec01] and the jMax documentation). Data are read on a classical way, using standard input/output functions:

```
BEGIN
   Open file
   get first event type

    while ( not END_OF_FILE and enough space)

        switch (event type)

             case TYPE_NOTE:
                 get (pitch)
                 get (duration upper value)
                 get (duration lower value)
                 get (accent)
                 get (measure number)
                 get (beat number)
                 get (fraction)
                 calculate duration in ms
                 calculate start time in ms
                 add event to jMax track
                  break;

             case TEMPO_CONTROLER
                  LEGATO_PARAM_CONTROLER
                  DURATION_TOL_CONTROLER
                  DYNAMICS_CONTROLER:
                 calculate time in ms
                 get(controler value)
                  break;

             case TIME_SIGNATURE_CONTROLER:
                 calculate time in ms
                 get(upper value)
                 get(lower value)
                  break;

        //end of switch
   //end of while

   close file
END
```

3.2 Data structures

3.2.1 Used structures

Special structures have been created for the Virtual Music Teacher package and the detector object to store data from the score. They are:

- Structure for notes:

```
typedef struct _scorenote_ /*------------------------------------- */
{
  int *pitch;          /* pitch of the note, between 0 and 127  */
  int *value_n;        /* upper number of the duration fraction */
  int *value_d;        /* lower number of the duration fraction */
  int *accent;         /* 1 if the note should be accentuated   */
  measure_t *position; /* musical time of current note          */

} scorenote_t;
```

- Structure for controlers:

```
typedef struct _scorecontroler_ /*-------------------------------- */
{
  int *type;           /* controler type: tempo, time signature, dynamics...*/
  int *value;          /* value of the controler */
  int *value2;         /* only for controlers that need 2 values */
  measure_t *position; /* musical time of current controler */

} scorecontroler_t;
```

- In both, the data type measure_t has the following definition:

```
typedef struct _measure_t_ /*------------------------------------- */
{
  int measure_number; /* begins with 1 and not 0 */
  int beat_number;    /* according to the time signature */
  int beat_fraction;  /* value between 0 and 95 */

} measure_t;
```

The main structure contains a pointer to a scorenote_t array and to a scorecontroler_t array, in which are stored all the event got from the score text file.

3.2.2 Numeric values calculation

The main problem of score acquisition is that we have to convert “musical time” into “official time”, i.e.:
- A position in [Measure, Beat, Fraction] into a start time in ms;
- A duration in x/y into a duration in ms.

These convertions are related to tempo, and are calculated as following:

$$duration = \left(\frac{60000}{tempo} * \frac{n}{d} * td\right)$$

With:
n = note duration upper value
d = note duration lower value
td = time signature lower value

Calculation of start time is a bit more complex because of the possible tempo changes. So we have to calculate elapsed time between two tempo changes and record it as offset.

$$time = \left(\frac{60000}{tempo} * \left((m1 - m2)tn + (b1 - b2) + \frac{f1 - f2}{96}\right) + offset\right)$$

With:
$m1$ = measure number of note position
$m2$ = measure number of last tempo change
$b1$ = beat number of note position
$b2$ = beat number of last tempo change
$f1$ = beat fraction number of note position
$f2$ = beat fraction number of last tempo change
tn = time signature upper value
$offset$ = time elapsed from the beginning to last tempo change
($offset$ is calculated by the same way)

3.3 jMax sequence object creation

All the data we need to fill our jMax sequence object have been read from file or calculated. The creation algorithm is the following:

```
BEGIN
    for each note
         if(pitch == 128)
             event_type = rest
         else
             event_type = note
             set note pitch
         endif
        set event duration
        set event velocity
        set event index
        create new object
        fill object with event
        add event to track
    end for
END
```

Our score is now complete: We have filled our internal model and the jMax sequence will be used by the score follower as its reference score.

3.4 MIDI performance acquisition

3.4.1 Data acquisition

We have defined an inlet for the detector object which is dedicated to input played flow: We recieve a list with *(pitch, velocity)* for notes-on, and *(pitch, 0)* for notes-off. The associated function detector_get_played has just to get those values, and to check if received event is a note-on or a note-off by checking velocity value.

Time of incoming event is calculated by the function, refering to initial time when first note was played.

3.4.2 Circular buffer storage

To have a quite large working window without having to create a big sized array, we store the data in a "circular buffer": When the fixed-size buffer is full, we continue to full it from the beginning.

Stored data are played pitch, played velocity, played time.

Chapter 4

Error detection and classification

We have now our score and our performance. We need now to detect errors, i.e. find where errors occured, and to classify them, i.e. find which kind of error it was. A function has been implemented to detect errors (see description of this function in section 4.1). this function calls other subfunctions to classify detected errors. These subfunctions are described in 4.2, 4.3 and 4.4.

4.1 Error checking function

When the detector is activated, we enter the function *mididetector_check* every 5ms[1]. This function does nothing unless we have got:
- a played note as input
- a state as output from the score follower

" Played note" are not only notes-on: with played note, we understand a note-on or a note-off, because we have to consider that a note-off (end of a note) can eventualy be a "rest-on", i.e. the beginning of an expected or unexpected rest.

A matching is also done:
- when we have got a note-on which has been recognised,
- when we have got a note-on for which the follower outputs a g-state (not recognised),
- when we have got a rest, recognised or not.

This function detect errors, but doesn't do any classification. When an error is detected, a flag is updated, and corresponding detection functions (which will check resp. note errors, rhythm errors and dynamic errors) will be called later (when we have enough information to classify that error).

For each check, we also have to update time-offset: we must "erase" the difference between time of played note and time of followed note, to avoid recording many times the same error.

Let's now see used algorithms for each kind of detection, and main algorithm.

[1]Actually, it is no more necessary to enter this function every 5ms. It was done to reset values that are now set in other functions. This alarm mechanism has to disappear in next versions.

4.2 Note errors detection

4.2.1 Indicators

The simpliest way for us to know if a note error occured is to wait until we get a g-state from the score follower. This would mean that somewhere in the score, the state corresponding to the expected note (n-state) has not been reached, i.e. we have something wrong in our alignment.

But knowing that we have a problem is not enough to say which kind of problem we have. To do this, we need to see what happens next: our way to know which kind of note error we had is to look at recognised notes indexes. By checking pitches and indexes of notes following the one who was wrong, we will be able to say what was played: state succession will tell us what happened (see [OD01] and figure 4.1).

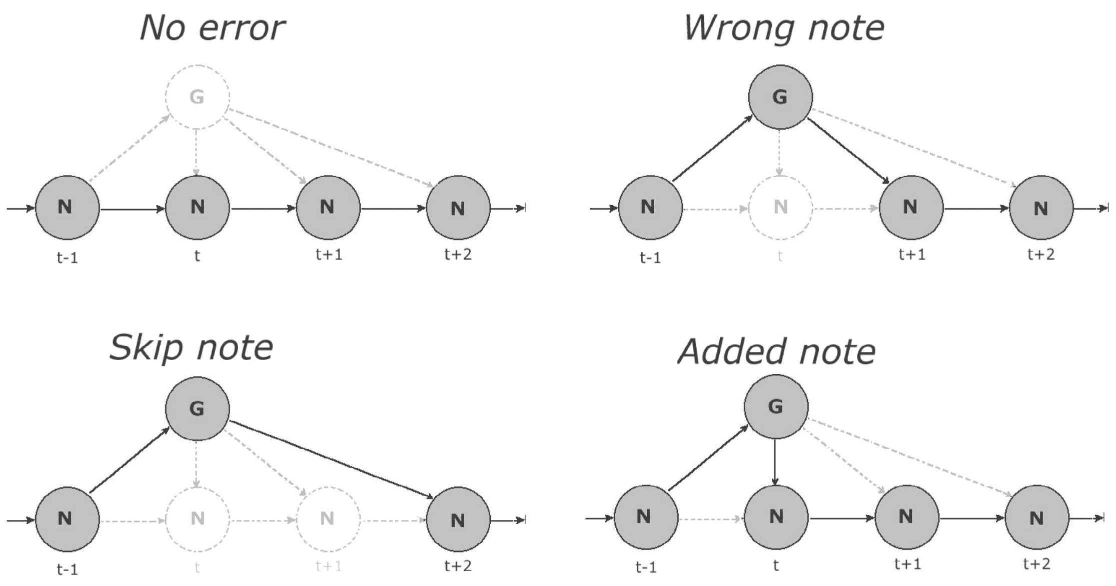


Figure 4.1: *Differences between alignments for each type of note error*

4.2.2 Algorithm

When the main checking function has detected a note error, next state reception will call the sub-function *mididetector_ checknoteerror* which implements the following algorithm:

```
BEGIN
    for (all detected errors since last check)
        index_offset = index(lastFollowedNote) - index(previousFollowedNote)
         if (index_offset == (number of detected errors since last check) + 1)
              if (pitch(lastPlayedNote) == pitch(nextExpectedNote)
                  Error = type SKIP NOTE
              else
                  Error = type WRONG NOTE
```

```
                endif
            else
                Error = type ADDED NOTE
            endif
        end for
    END
```

4.2.3 Possible problems

Using this way to detect note errors means we consider that the follower only outputs g-states when we have an error. We may have problems in some particulary cases: for example, overlaps between notes in the performance may lead to an alignment with g-state. The detector will try to classify an error that was not played...

4.3 Rhythm errors detection

4.3.1 Indicators

Except for rest problems, rhythm errors are not directly shown like note errors: Since the score follower is made to follow the performer without caring of rhythm differences between score and performance, the alignment is the same.

Rests are treated like note errors: For a skip rest, the score follower will output a g-rest, and a g-state for an added rest. If we have a g-state with a note-on, error is a note error ; if we have a g-state with a note-off, error is an added rest.

For other rhythm errors, we have to check played time and time of corresponding note in the score, with an updated offset.

4.3.2 Algorithm

Skip rests errors are detected in the main checking function:

```
(...)
    if (we have a g-rest)
        Error = type SKIP REST
(...)
```

Other errors are classified in the sub-function *mididetector_ checkrhythmerror*, which will be called if needed on next state reception (as is it for note error checking sub-function):

```
BEGIN
    if (previous state was a g-state)
        Error = type ADDED REST
    else
         if (new_time_offset - previous_time_offset > 0)
              Error = type NOTE TOO LONG
         else
              Error = type NOTE TOO SHORT
         endif
    endif
END
```

4.4 Dynamic errors detection

4.4.1 Indicators

There is only one indicator for dynamic errors: velocity of played note. We have to compare it with velocity:
- of corresponding note in the score in order to detect general dynamic errors,
- of previous and next note if the played note should be accentuated.

We have another problem in the case of general dynamic errors: we have to consider that we have only one error even if we have 10 notes which are played too loud (and not 10 errors). The idea is to store the first note n_1 which is too loud, and then consider we have an error when we get a good velocity again for the note n_2. Error will be considered as a general error between note n_1 and note $n_2 - 1$

4.4.2 Algorithm

The Dynamic error detection is implemented in the general checking function:

```
BEGIN
    if (previous note should have been accentuated)
        Error_next = played_velocity < next_velocity + accent_threshold
        Error_previous = played_velocity < previous_velocity + accent_threshold
        if (first note)
            error = error_next
        else if (last note)
            error = error_previous
        else
            error = error_next && error_previous
        endif
        if (error)
            error = Type ACCENT
    else
        if (played_velocity > expected_velocity + threshold)
            Error = type TOO LOUD
        else if(played_velocity < expected_velocity - threshold)
            Error = type TOO SOFT
        endif
    endif
END
```

4.5 Main algorithm overview

Here is the simplified version of implemented algorithm in function *mididetector_ check*:

```
BEGIN
    if (check is enabled && we had a played note to check)
         if ( we got a g-state from the follower && a note-on has been played)
             enable note error detection for next state
             output(``note error'')
             increment number of note errors
         else /* pitch is ok, now check other errors */
              if (we got a g-rest) /* skip rest error */
                  store error
                  Output (``Skip rest error'')
                  increment number of rhythm errors
              else /* check time for other possible rhythm errors */
                   if (played time - followed time - offset > tolerance)
                       enable rhythm error checking for next state
                       output(``rhythm error'')
                       increment number of rhythm errors
                   else /* time is ok, check now velocity */
                       check dynamic errors
                   endif
              endif
         endif
        new time offset calculation
    endif
   Reset flags
   Re-arm alarm
END
```

Chapter 5

Error Signalisation

As we said in chapter 1, we chose to have two ways to indicate detected errors : online and afterwards. Online signalisation and afterwards signalisation are described in sections 5.1 and 5.2.

5.1 Real time Online signalisation

As defined in chapter 1, user can choose to have this online signalisation or not. Errors are indicated in two ways: a "bang" is produced by the detector object, and a message is printed to the jMax console.

5.1.1 Immediate "bang"

Since we have three main error types (note, rhythm and dynamic), we will have three outputs to indicate them. These outputs are defined as *outlets* for the jMax object: that means we can get those outputs in the jMax environment and do what we want with it: alarm sound, visual signal, counters, etc.

Outputs are made as soon as the error has been detected, i.e. before everykind of classification. If we get a bang for a note error, We only know that an error has been made, we don't know which kind of note error (wrong, skip, added) it was.

5.1.2 Console printing with short explaination

At the same time, the object prints a message to the jMax console. A second message is printed after error classification, to give some clues about the error.

For a rythm error, we will have for example, appearing just after the error:

```
####### RHYTHM ERROR #######
```

And then, when the detector has been able to classify error and give numerical values:

```
Note (or rest) number 1 was too short of 275.000000 ms !
(55 percent shorter than normal duration)
```

5.2 Report edition

If requested, an error report is printed to an output file text when we stop the detector. File name and path are set by user.

5.2.1 Storing errors

Each error must be stored somewhere if we want to print a report at the end of the play. For each one, we need to know the error type, the note where the error has been done, and the "value" of the error (played pitch for a wrong note, time offset for a rhythm error...).
Errors that refer to more than one note like general dynamic errors need to refer to 2 indexes: note where the error begins, and note where the error stops.

The following structure has been implemented to store performed errors:

```
typedef struct _error_ /*--------------------------------- */
{
  int *type;     /* error type: note, rhythm, dynamics */
  int *note;     /* index of note where the error occured */
  double *value; /* played pitch for note errors, offset for rhythm errors... */
  int *end;      /* only used for errors between 2 notes */

} error_t;
```

The main structure vmt_t contains an array of error_t. This array is filled on each new detected (and classified) error, and when the report is printed, all the data are got from this array.

5.2.2 Retrieve Musical data from numeric data

Since all the errors are stored with their numeric data, we have to convert it into musical data to give a "friendly output". It would be nonsense to say to the user:

```
at time 1459,235674 you played a pitch 64 instead of a pitch 68.
at time 2546,984123 you played note number 26 too long of 567,235674ms
```

First thing to do is to get the note name and the octave value from the pitch. It can be easily made with a little tool function. Then, we have to convert absolute time into musical time. It is just the inverted operation of the one we made to calculate numeric data from the score (see Chapter 2, section 2.2).

5.2.3 Output file

Output file has two parts: It begins with a description of detection parameters and then give a list of all performed errors with an explaination for each one.

Typically, the report file will look like:

```
----------------- 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 PERFORMANCE
(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.

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

Chapter 6

First results

Tests have been made on mididetector from package VMT version 1.0, with different pieces, in different situations. 10 type of errors were used (played or simulated) to do the tests. They are : *wrong note, skip note, added note, note too long, note too short, skip rest, added rest, missed accent, play too loud, play too soft.*

Results are given as following:
- a = number of detected errors / number of played errors (in %),
- b = number of well classified errors / number of played errors (in %),
- c = number of good notes detected as errors / number of played good notes (in %).

6.1 Small sequences with computer generated performances

First tests have been done with a small sequence (8 notes and a rest). Performances with errors have been computer generated, and tests have been done by grouping error types.

6.1.1 Note errors

In testfiles where we only have note errors, results are:
a = **93.5%** of played errors have been detected.
b = **74.2%** of played errors have been detected and well classified.
c = **2.5%** of good notes have been recognised as errors.

Detection is not a problem, but classification remains uncertain, especially in case of consecutive errors. In our case, we have some good results with isolated errors, but if we have more than 3 consecutive wrong notes we obtain a bad classification.

This problem is linked to the size of the used piece: making more than 3 errors on a 8 notes piece does not really allow to have a good alignment...

6.1.2 Rhythm errors

Tests made with only rhythm errors give:
a = **96.2%** of played errors have been detected.
b = **96.2%** of played errors have been detected and well classified.
c = **5.7%** of good notes have been recognised as errors.

Detection and classification are quite good. High value of *c* can also be explained by the size factor: when too many errors are made on a small piece, it becomes hard to find where good notes are...

6.1.3 Dynamic errors

With only dynamic errors, we have:
a = **93.3%** of played errors have been detected.
b = **93.3%** of played errors have been detected and well classified.
c = **3.3%** of good notes have been recognised as errors.

Results are quite good too. Problems come when we have combinated errors: general dynamic errors / accentuation problems. An accentuated note in a too loud played part may be detected as an error while it was right.

6.1.4 Overview

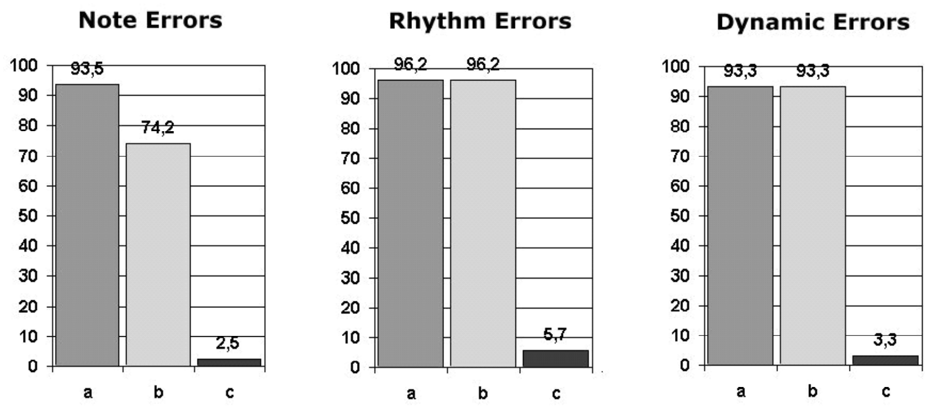


Figure 6.1: Results for small sequences with artificial errors, grouped by error type

6.2 Real scores

Two pieces have been chosen to test the detector:*"Polvtsian Dance n.17" from "Prince Igor"(A. Borodin)*, and *"czardas"(V. Monti)*.

For each one, a short simplified extract has been coded:

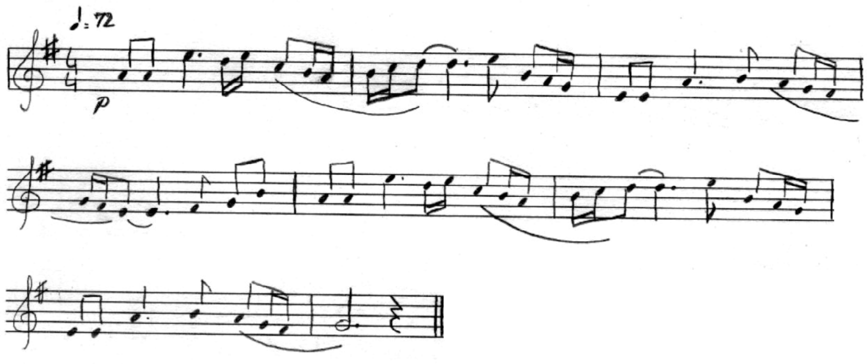


Figure 6.2: A. BORODIN, "Polvtsian Dance n.17" from "Prince Igor" - Extract

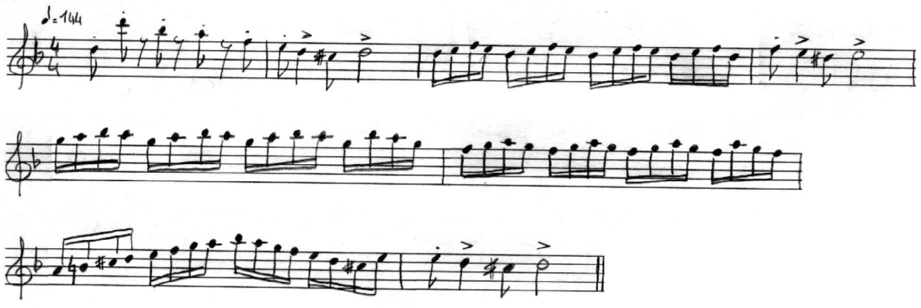


Figure 6.3: V. MONTI, "Czardas" - Extract

Those two pieces are short but representative. "Prince Igor" is slow, expressive and quite, not very difficult to play but containing a lot of feelings to express, while "Czardas" is a fast and furious extract that presents great difficulties for a medium level musician, and which requires precision and regularity.

These pieces have been computer generated for first tests, then played on a midi keyboard and recorded as midi files to have results with real played input.

6.2.1 With computer generated performances

Tests on Borodin's Polvtsian dance give the following results:
a = **76.7%** of played errors have been detected.
b = **76.7%** of played errors have been detected and well classified.
c = **2.8%** of good notes have been recognised as errors.

Here is what we obtain with Monti's Czardas:
a = **92.2%** of played errors have been detected.

b = **78.1%** of played errors have been detected and well classified.
c = **4.3%** of good notes have been recognised as errors.

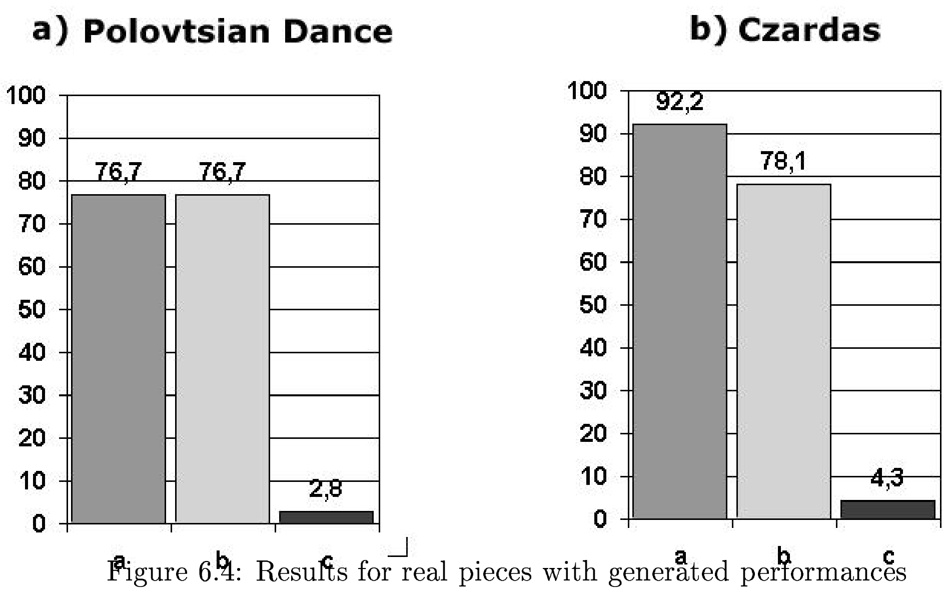


Figure 6.4: Results for real pieces with generated performances

Results for these "long pieces" are not very different from first results. That is a good point: size of pieces don't seem to have any influence on detection rates. It is anyway a bit amazing to see that error detection is more performant on a fast and complex piece than on a slow one. This can be explained by testing conditions: more note errors have been tested for the Czardas, which give high detection rate (a) and lower classification rate (b).

6.2.2 With played and recorded performances

With Borodin, we have:
a = **84.2%** of played errors have been detected.
b = **73.7%** of played errors have been detected and well classified.
c = **11.5%** of good notes have been recognised as errors.

With Czardas, we obtain:
a = **70.3%** of played errors have been detected.
b = **48.6%** of played errors have been detected and well classified.
c = **9.5%** of good notes have been recognised as errors.

There are two important things to notice. First (it's not a surprise), it is more difficult to do a correct classification when we have a real interpretation: Some parts of the performance are not really "clean" and can lead to a bad decoding of the alignment. It is also logical to observe that it is even more difficult if the piece is complex and fast.

Second point is the high value of *c*. For the same reasons, when the performance is not clean (with overlaps between notes and short added rests) we may detect errors where a human hear would not do. Especially, it happens when the follower outputs *g-states* when we don't have any error.

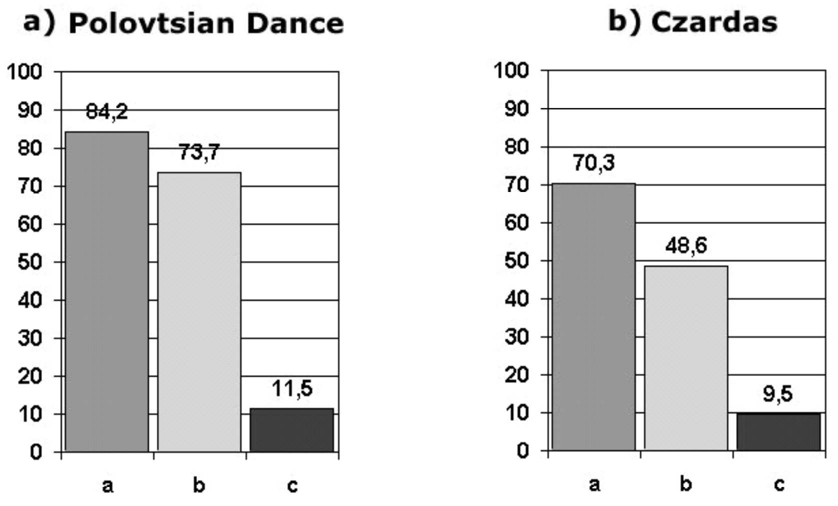


Figure 6.5: Results for real pieces with keyboard played performances

6.3 Results overview

results can be presented as follow:

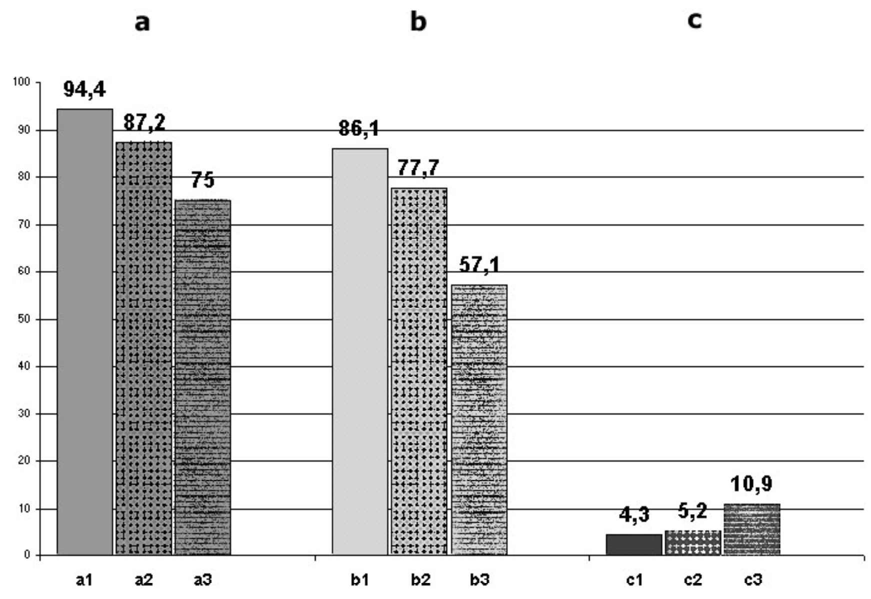


Figure 6.6: Detection rates obtained for different test conditions

Short pieces with generated performances	**Complex pieces with generated performances**	**Complex pieces with played performances**
a1 = 94.4%	a2 = 87.2%	a3 = 75.0%
b1 = 86.1%	b2 = 77.7%	b3 = 57.1%
c1 = 4.3%	c2 = 5.2%	c3 = 10.9%

Chapter 7

Adaptation to audio input

Main part of the project was done based on a MIDI input with object mididetector, as it was our first choice to work with MIDI.

In order to extend the Virtual Music Teacher, an other object, audiodetector, is still being implemented to allow a detection on audio input. This chapter describes main changes on objects and algorithms implied by this new object.

7.1 Performance acquisition

7.1.1 Get spectrum via Fast Fourier Transformation

Real audio signal cannot be represented with simple lists, like it was with MIDI input and note-on / note-off.

The idea will be to use the Fast Fourier Transformation (FFT) in order to have the spectrum of our signal with a determined sample rate.

jMax provides a FFT object, which will allow us to get directly real and imaginary part of the FFT as input for our system. Input flow will no longer be the played signal but the spectrum of this signal.

7.1.2 Structure changes

So far, MIDI parameters in structure *mididetector_t* have been replaced in structure *audiodetector_t* by members representing input parameters of the FFT. The structure does not contain circular buffer for played input anymore.

Note that current version of audiodetector is for the moment nothing but an untested skeleton. Some other members will probably change while going on programing.

7.2 Changes in model

We have now to compare a real signal to a model that cannot stay as it was for MIDI. The idea will be to represent each note of the score with an harmonic filter instead of a pitch, as it is done for audio score following and described in [OS01].

7.2.1 New structure *audioparam_t*

If we want to do a matching on audio signal, we have to store some audio parameters for each note of the score. We need to add members to the *scorenote_t* structure to store these values. This has been realised by creating a new structure *audioparam_t*, and including in *scorenote_t* a new member which is an array of *audioparam_t*:

```
typedef struct _audioparam_ /*------------- */
{
  double energy;
  int harmo_filters[NUM_FILTERS][2];

} audioparam_t;
```

energy will be a ratio calculated from the velocity value, and *harmo_filters* will contain the harmonic filters values, and calculated from pitch value.

7.2.2 Harmonic Filters calculation

We need two things to have audio parameters representing pitch: the frequency of each harmonic, and a bandwidth to compute a filter arround them.

harmonic frequencies f_i are calculated as follow:

$$f_i = (i+1)\left(f(a).e^{\alpha(pitch-p(a))}\right)$$

where:
$f(a)$ = 440Hz (A4 standard frequency)
$p(a)$ = 69 (jMax pitch value for A4)
α = 0,057762265

We choose to have a bandwidth of an half-tone arround each harmonic frequency. That means we will have:

$$harmo\_filters[i][0] = f_i - (f_i.\delta_{halftone})$$
$$harmo\_filters[i][1] = f_i + (f_i.\delta_{halftone})$$

With: $\delta_{halftone} = 0.0594631$

The problem is that this model is instrument depending, and for each harmonic band will we have to determine a factor in order to have an harmonic filter that corresponds to the instrument which is playing.

7.3 Changes in detection algorithm

First problem we have to face is how to handle output differences between suivimidi and suiviaudio: In mididetector detection algorithm, we use the current state of the follower to detect errors in alignment. We can't do the same thing for the moment with audiodetector, because suiviaudio doesn't output this current state.

We have so 2 possibilities:
- find another way to detect errors,
- change outputs of object suiviaudio.

First solution would be a good way to find improvements for the program (by combinating 2 algorithms for example), but is also time demanding. Second solution would be the most logical (it is not very difficult to add an output to object suiviaudio), but requires to be done by jMax and Score Following Developpers.

Second problem will come with all parameters we will have to determine before we are able to detect errors: for example, start time of a note will have to be checked out by looking at energy and spectrum parameters, i.e. with many calculations.

Chapter 8

Conclusion and outlook

8.1 Summary - Realised work

Our goal was to realise a computer-aided music learing system, called "Virtual Music Teacher", which would allow to detect errors in a played music piece and give explainations on these errors.

All error kinds have been described (2.3.1), input, output and other data to use have been defined (2.2 and 2.4). The system is based on score following using Hidden Markov Models (1.2).

The *Virtual Music Teacher* has been implemented as a package for **jMax**, which is a real-time programming environment for music and multimedia produced at **IRCAM - Centre Pompidou**, Paris.

Implementation work realised between march and august 2003 has led to a 1.0 version of the VMT package (**VMT-v1.0**) which has following features:
- Error detection can be made on a **monophonic piece**,
- **MIDI data** are used as input for the performance,
- Reference score is acquired with a text file, following a defined syntax,
- Detected errors are **note errors** (wrong, skip and added note), **rhythm errors** (note too long or too short, skip or added rests), and **dynamic errors** (note too loud or too soft, accentuation),
- An **online signalisation** is possible when an error occurs,
- It is possible to **edit a report** at the end of the play. This report contains : playing conditions, number of played errors, error details (location, kind, explaination).

VMT-v1.0 has been tested and gave its first results (ch. 6). Work to do and future work are described in next section.

8.2 Work to do - Future work

8.2.1 Untreated cases

Defined error not handled yet

All kinds of note errors, rhythm errors and dynamic errors have been handled, but some phrasing errors are still missing: version 1.0 of package VMT does not handle *legato/staccato* errors.

We have already introduced the *legato* parameter in the score text file syntax and the internal model; that means we are able to know when a *legato* or a *staccato* is expected. The problem is how to detect such an error, that can be explicitely in conflict with rhythm error detection (note too long, note too short).

Handle such phrasing errors will mean find another way to detect rhythm errors, or at least find a way to preserve real-time signalisation for errors that are during more than one note and that can be in conflict with other error types.

First and last notes problem

Some of the handled errors need to know the context in order to be detected or classified. We have a problem if an error occurs on the last note: mechanisms for error detection do not have enough visibility to determine what has been done.

That happens for exemple on wrong note problems: if a g-state is got for the last note, we don't know if this note was wrong or not played. We have also a problem is this last note is too short or too long, since nothing happens after it...

Sometime the problem may appear if an error occurs on the first note. Alignment is different in that case since first state to be reached is a g-state.

Multiple and consecutive errors

Making more than three consecutive wrong notes often leads to a bad detection or classification. This is mainly due to alignment's hesitation to find the best way in the model. To avoid this, we would have to extend our detection mechanism in order to have a greater working window, and only consider the last alignment as the right one.

8.2.2 Adaptation to polyphonic music

Package version 1.0 does not handle polyphony. Work in this direction would be a great improvement, especially for MIDI detection, since most of MIDI instruments are keyboards or pianos which are polyphonical instruments.

Implied changes in performance acquisition and model

There is nothing to change in the way the score is coded in the text file: to represent a chord, we will only have to code all the corresponding notes with the same position.

Main changes will appear in intern model. When creating the model from text file, we will have to collect all the notes arround the current one that belong to a chord. This means that our dynamic array won't be a note-array anymore, but a chord-array. There are new structures to define, and old structures to change.

Other problems will come when polyphonic events are no chords, but long overlaps. Idea will be to make a "score parsing" as it was done for the score follower (see [Mat02] for more details).

Implied changes in algorithms

Error detection will become much more complicated. First, in case of note errors, a g-state output will not be a detection criterium anymore: if only a note in the chord is wrong, then we have a note error, but the follower has a great probability to output a n-state anyway.

In case of rhythm errors, it will become harder to find which event is to store or not, especially in case of overlap problems between notes.

If we want to handle polyphony, the model has to be changed and the algorithms must be completely rewritten.

8.2.3 New features - improvements

Other kind of errors to detect - grouping errors

Some errors can be classified in many ways, and by grouping simple errors it would be possible to give a more precise explaination on what has been played. For example, rhythm errors on fast and regular notes are now handled as many consecutive "notes too short" and "notes too long". But it makes more sense to classify them as only one error, which will be defined as a "lack of regularity".

In that way, it would be a good idea too make a second classification after the first one, as a kind of supervising mechanism which would be looking for a logical link between played errors.

It would be great, in the future, to think about a 3-phased diagnostic system: detection - classification - grouping.

Improvement for "Two-hands played" instruments

At the beginning of the project, an inquiery has been realised in order to know what music teachers would expect from the project (see appendix A). Most of their answers have been used to define bases of the VMT, but some of them have been let as ideas for future development. They especially concern two-hands played instrument like keyboard or piano.

One of these ideas is the following: add to the VMT a sequencing function, which would allow to record one part of the score (for example the left hand accompaniment), and then replay it while the performer plays the other part(right hand melody).

Of course, this improvement could only be realised if we decide to handle polyphony.

Other possible features

Another suggested idea is to allow the performer to compare what he has played with what he should have played. This means adding a record and replay function for all the parts of the performance where errors have been done.

It would be also a good idea, but such an improvement does not have much to do with error detection, and could be realised in another research project.

Other interesting features would be to add more parameters to the score, like *crescendos*, *accelerandos* or recitatives. To do this we would need to find a way to code evolutive parameters. There are still a lot of things to define and a lot of things to do...

Appendix A

Enquiry for music teachers

In order to define exactely some points and to see what music teachers think about this project, the following questions have been asked to music school teachers, independant music teachers, and university music teachers.

1 - Do you you think such a tool would be useful ?

2 - This program would allow to detect played errors. Which kind of errors should be detected first ?
- Note errors (wrong note, skip note, extra added note...)
- Tempo and rhythm errors (note and rest duration, regularity...)
- Dynamic and phrasing errors (general dynamics, accentuation, legato/staccato...)

3 - Should the detected errors be immediately pointed out, or just be indicated when the play is finished ?

4 - How can we indicate errors in real-time ?
- With a sound (like a "bip")
- With a visual signal on the screen
- other:

5 - At the end of the performance, a report could be delivered by the program. Which information would you like to have in this report ?
- "numerical" data (measure number, beat...)
- Error "diagnostic": a short explaination of the error (ex: you have played a A, you should have played a A#)
- The possibility to hear one's error and to compare it with what should have been played.
- Error statistics (errors number, repeated errors...)
- Other data:

6 - **For which kind of musicians would be this program more useful (beginners, medium level, high level...) ?**

7 - **Would you like the performance to be compared to the "real, fixed and perfect" score, or with a more flexible model, like the piece played by an experienced musician ?**

8 - **If such a program would exist, would you like to work with it (for exemple by asking your students to give you their reports to see their progression) ?**

9 - **Other suggests ?**

Appendix B

Virtual Music Teacher: mididetector user guide

B.1 Generalities about jMax and jMax objects

jMax is a graphical programming environment: The user can put and connect graphical objects in windows called "patches". with the mouse, the user connects standard objects, or objets of other librairies like synthesis, filters, effects, score following...

While running, jMax looks like:
- a console where are printed logs, messages and other outputs,
- one or more patches with the inserted and connected objects.

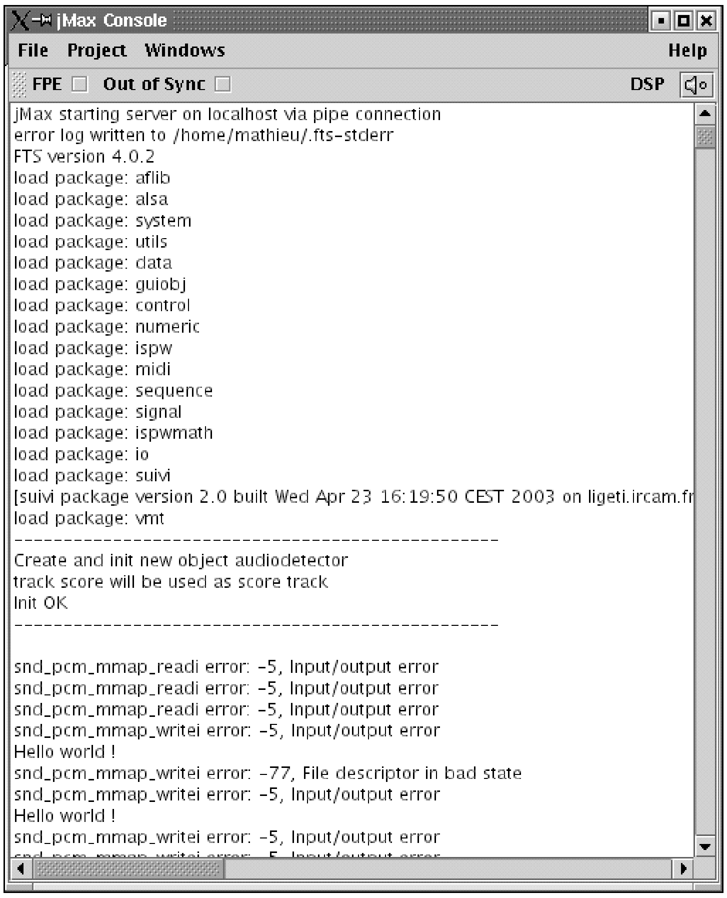
Each object can have inputs (inlets) and outputs (outlets). some objects outlets can be connected to other objects inlets, so that we can graphically chain functions by using several and different objects.

Example: a classical "hello world" program made with jMax will look like:
A patch in which will be inserted a “message box” object containing the sentence “Hello World”, and a “print” object. message box’s outlet will be connected to print’s inlet. By a mouse-click on the message box, we will make the message “Hello World” printed on the console (see fig B.2).

Most useful objects are:
- display objects, which allow to print or make a graphical display of what they get as input
- sender objects: message boxes, bangs, sliders, etc.
- data storage objects: vectors, matrices, lists, etc.
- sequencing objects: sequences handling, MIDI,
- audio objects, which allow to read and extract data from sound files or direct stream,
- much more, like video objects, mathematical functions, logical functions, etc.

Objects can be combined and connected without limits by using many patchers, editors...

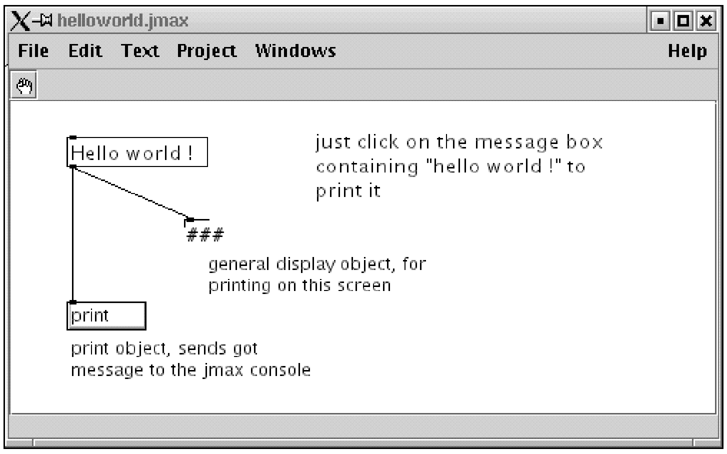
For more details about jMax, see [FD99] an [Dec01], or see IRCAM free software webpage on *http://freesoftware.ircam.fr*



```
jMax starting server on localhost via pipe connection
error log written to /home/mathieu/.fts-stderr
FTS version 4.0.2
load package: aflib
load package: alsa
load package: system
load package: utils
load package: data
load package: guiobj
load package: control
load package: numeric
load package: ispw
load package: midi
load package: sequence
load package: signal
load package: ispwmath
load package: io
load package: suivi
[suivi package version 2.0 built Wed Apr 23 16:19:50 CEST 2003 on ligeti.ircam.fr
load package: vmt
--------------------------------------------------
Create and init new object audiodetector
track score will be used as score track
Init OK
--------------------------------------------------

snd_pcm_mmap_readi error: -5, Input/output error
snd_pcm_mmap_readi error: -5, Input/output error
snd_pcm_mmap_readi error: -5, Input/output error
snd_pcm_mmap_writei error: -5, Input/output error
Hello world !
snd_pcm_mmap_writei error: -77, File descriptor in bad state
snd_pcm_mmap_writei error: -5, Input/output error
Hello world !
snd_pcm_mmap_writei error: -5, Input/output error
```

Figure B.1: The jMax console



just click on the message box containing "hello world !" to print it

general display object, for printing on this screen

print object, sends got message to the jmax console

Figure B.2: The "Hello World" patch

B.2 Score creation

You will need to code your score as a text file in order to use it with mididetector. This score will contain:
- time signature definition,
- tempo definition,
- user defined parameters for the detection,
- notes of the score.

Each event (note or controler) is defined by its type and its value, as shown below:
note:
type = 0, values = pitch, note_value(2 numbers), accentuation, measure, beat, fraction
tempo controler:
type = 1, values = tempo value, 0, measure, beat, fraction
time signature controler:
type = 2, values = upper value, lower value, measure, beat, fraction
legato parameter:
type = 3, values = value, 0, measure, beat, fraction
duration tolerance:
type = 4, values = value, 0, measure, beat, fraction
dynamic controler:
type = 5, values = value, 0, measure, beat, fraction

For a note, the two number of note_value are resp. the upper and the lower value of the duration fraction. Pitch of a note is an integer between 0 and 127 (center value is C5 = 60). Rests are also coded: they are represented by notes with pitch 128.

example: for a C#5, 1/8 note, not accentuated, at measure 2, beat 3, fract 48, we will write:

```
0 61 1 8 0 2 3 48
```

Tempo value is an integer representing number of beats per minute. Legato parameter has three values: 0 (staccato), 1 (undefined), 2 (legato). Duration tolerance is between 0 and 100%. Dynamic controler value is an integer between 0 and 7, corresponding to *[ppp, pp, p, mp, mf, f, ff, fff]*.

"-1" at the end indicate end of score.

B.3 Parameter initialisation

B.3.1 Objects to insert and to connect

Objects of package VMT work with objects of package suivi (score Following Package). Mididetector must be connected to suivimidi in a jMax Patch, as shown on figure B.3.

"Played_input" is a list with [note pitch, note velocity]. You get it from the midi input device, or from the "play" object if you just want to test the detector with a recorded sequence.

For more details on suivimidi, see the suivi package help patches.

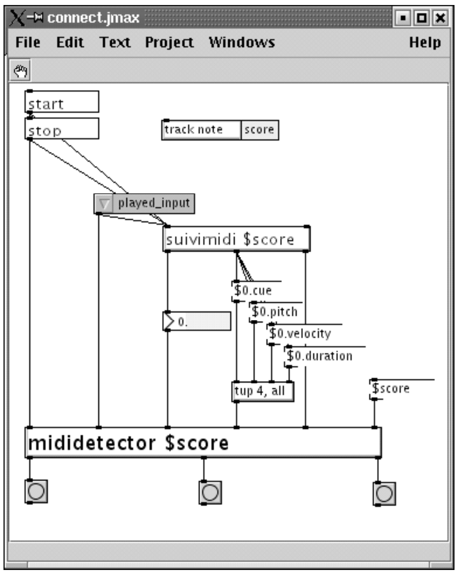


Figure B.3: How to connect mididetector with suivimidi

You also have to insert a jMax sequence in your patch, which will be used as score by object suivimidi.

B.3.2 Set detection parameters

Detection parameters that can be set are:

- tempo (message "settempo" followed by the value),
- rhythm Tolerance (message "setrhytol" followed by the value),
- type of errors to detect (message "seterrortype" followed by the corresponding number),
- kind of error signalisation (message "setsignalisation" followed by the corresponding nb).

See mididetector help patch presented on figure B.4 for more details about how to set these values.

You can also enable or disable debug printing on the console by sending the message "setverbose" followed by 1 (enable) or 0 (disable).

B.3.3 Load your score

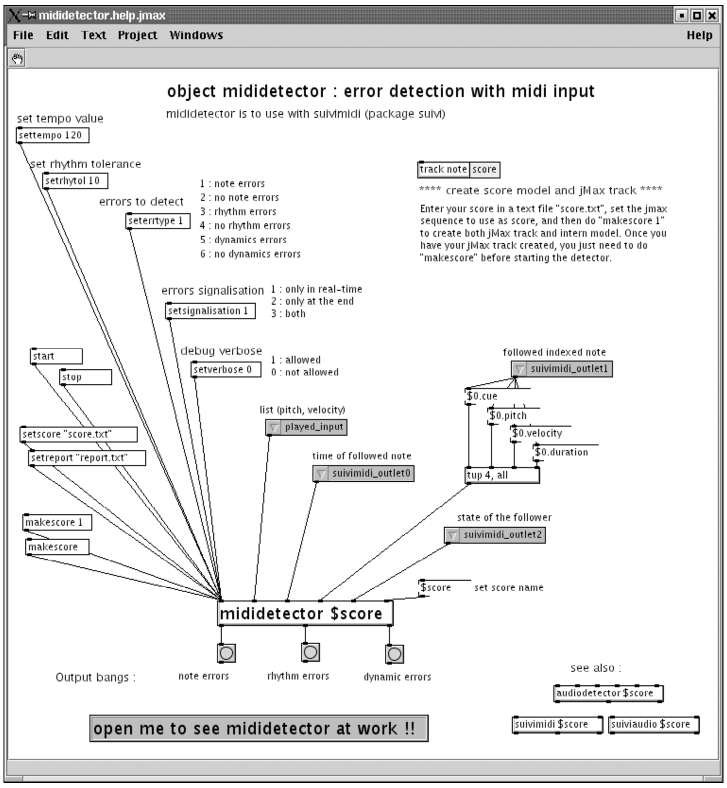
If you named your score “foo.txt” and stored it in your home directory, you can set score name in jMax patch by sending the message [setscore “/home/myhome/foo.txt”] to the mididetector object. Then you have to send message [makescore 1] in order to create both jMax sequence and intern model. If you already have created the jMax sequence, you just have to create intern model by sending message [makescore]

B.4 Make it work !

Create, connect and set parameters of your objects in your jMax patch (mididetector, suivimidi, sequence object). then load your score as described above.

If you chose to have a report at the end of the play, you can also set report name by sending the message “setreport” followed by the name between double-cotes.

Then start suivimidi and mididetector by sending message “start”.



object mididetector : error detection with midi input

mididetector is to use with suivimidi (package suivi)

1 : note errors
2 : no note errors
3 : rhythm errors
4 : no rhythm errors
5 : dynamics errors
6 : no dynamics errors

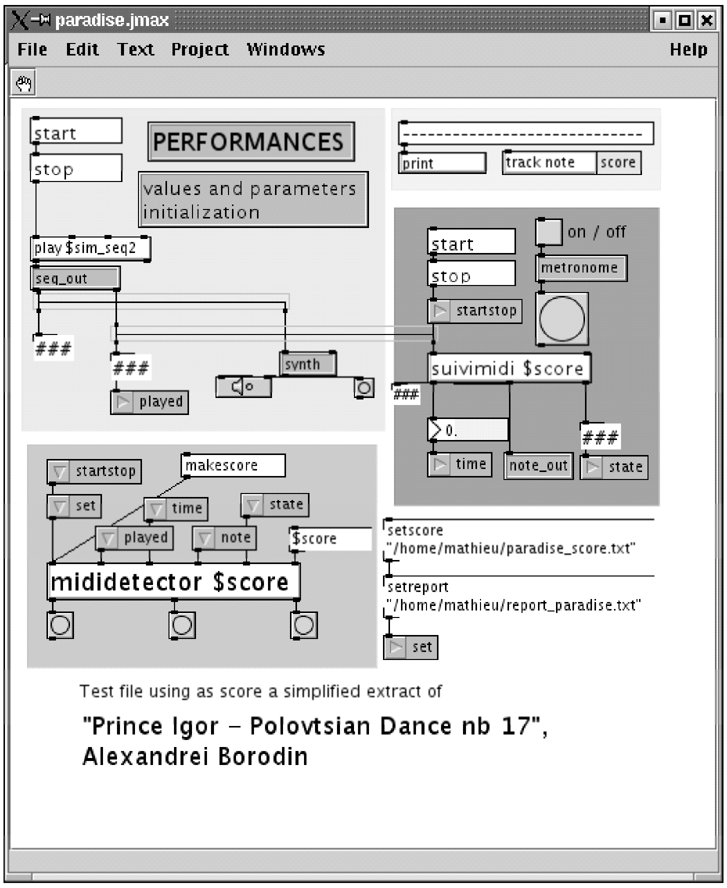
1 : only in real-time
2 : only at the end
3 : both

1 : allowed
0 : not allowed

**** create score model and jMax track ****

Enter your score in a text file "score.txt", set the jmax sequence to use as score, and then do "makescore 1" to create both jMax track and intern model. Once you have your jMax track created, you just need to do "makescore" before starting the detector.

Figure B.4: Help patch of object mididetector



Test file using as score a simplified extract of

"Prince Igor – Polovtsian Dance nb 17",
Alexandrei Borodin

Figure B.5: Object mididetector in test situation

Bibliography

[ALB99] P. Cano A. Loscos and J. Bonada. Score-Performance Matching using HMMs. In *Proceedings of the ICMC*, pages 441–444, 1999.

[Dec01] F. Dechelle. jMax : un environnement pour la réalisation d'applications musicales sur Linux. In *Journées d'informatique musicale*, Bordeaux, France, 2001.

[FD99] 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.

[Mat02] G. Mathieu. *Suivi de partitions musicales : rapport d'activité.* IRCAM, Paris, 2002.

[Mou01] V. Mouillet. *Prise en compte des informations temporelles dans les Modèles de Markov Cachés.* IRCAM, Paris, 2001.

[OD01] N. Orio and F. Déchelle. Score Following Using Spectral Analysis and Hidden Markov Models. In *Proceedings of the ICMC*, Havana, Cuba, 2001.

[OS01] N. Orio and D. Schwarz. Alignment of Monophonic and Polypophonic Music to a Score. In *Proceedings of the ICMC*, Havana, Cuba, 2001.

[Rab89] L.R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–285, 1989.