Markov Based Improviser

Lelio Casale (10582124)

July 22, 2024

1 Introduction

1.1 Presentation of the project

The project consists in an audio plugin that is a **Markov Model based improviser**. The **Markov Model** allows to generate sequences of notes based on an input MIDI data, both with real-time played notes and with loaded MIDI files. This allows to create notes that resemble the style of the input notes.

On top of this, many other features were implemented, such as a function to save and load trained models, a key detection tool and many others. The plugin output was also trained with different musical pieces, to show its musical abilities and to see which styles are reproduced better by the improviser. The plugin is fully implemented in *C++* using the *Juce* library and built using *CMake*.

1.2 Structure

The main features implemented are:

- A **Markov Model** that is the base of the audio plugin. It can learn in real time input MIDI sequences or the training can be done using a pre-recorded MIDI file. The trained Markov Model is capable of **generating notes sequences** that resemble the style of the training ones.
- **Two toggles**: one can be used to give to the user the possibility to **stop the training process**, such that the plugin doesn't learn MIDI sequences when the toggle is set to off, while the other is used to **start the notes generation process** based on the learned sequences when it is set to on.
- A **reset function** that completely cleans the Markov Model such that the Markov Model has no memory about the older input sequences.
- Two functions to **load and save learned models** to keep track of any musically interesting model and to eventually restart the plugin directly from the saved model without needing to re-train it.
- A **key detector** that displays the key and mode based on the MIDI data. This feature works both for real-time learned models and for models loaded from a previously saved file. This information is then used by the randomness selector to choose which random notes can be played and which not.
- A **randomness selector** which is used to give some variation to the Markov Model output: the user can select a randomness value in percentage which represents the number of notes in percentage that need to be randomized.

2 Implementation

2.1 Markov Model

The base feature of this project is a **variable order Markov Model** that learns MIDI input sequences. A first-order Markov Model is a **stochastic** model describing a sequence of possible events in which

the probability of each event depends only on the state of the previous event. Generalizing, an n-order Markov model describes the possible next event based on the states of the n previous events.

The Markov Model used for this project is a variable order model: this means that the next state doesn't depend only on a fixed number of previous states, but it chooses the maximum order available. For example, if the Markov Model has never seen in input the sequence of notes C-D-E, it will discard the older note and will work only on the probabilities based on the state sequence D-E. This process is done iteratively until there is at least one possible state available. A Markov Model could be used to characterize sequences of MIDI note numbers as shown in the previous example, but it can model also other musical parameters.

In the context of the project, four models are used:

- A model for the **MIDI note numbers**, which represent the notes on the piano keyboard. Their values range from 0 to 127, with the value 60 representing the C4 note.
- A model for the **inter-onset intervals**, which are the times elapsed between the start of a note and the start of the next note.
- A model for the **notes duration**., that means how much time elapses between a *noteOn* message and the corresponding *noteOff*.
- A model for the **note velocities**, which represent how hard a note is played on the keyboard.

The four learned parameters are stored into their respective models using four different functions in the *PluginProcessor* class: *analysePitches*, *analyseIoI*, *analyseDuration* and *analyseVelocity*. The developed plugin can work also with **chord sequences**. If more than one note is played within an interval of 50ms, the program recognizes it as a chord and treats it as a whole. In this way, chords can be thought as a single note and also sequences of chords, or notes alternated with chords, can be generated.

2.2 Graphical User Interface

The **graphical user interface (GUI)** consists of several components:

- A piano MIDI **virtual keyboard** that allows the user to play MIDI notes to be fed to the Markov Model. The input notes are then processed and their parameters are eventually added to their respective models in the *processBlock* function that is implemented in the *PluginProcessor* class.
- Two *juce::ToggleButton* which switch on and off the learning and generating processes.
- A *juce::TextButton* that is used to reset the Markov Model and clean it from the previously learned data.
- Two *juce::TextButtons* responsible of the possibility of saving a model and importing a previously saved one.
- A *juce::Label* that displays the detected key.
- A *juce::Slider* used to choose the desired amount of randomness.

All these components, thanks to the *PluginEditor* class, are connected to the functions that implement their respective features and placed using the *resized* function.

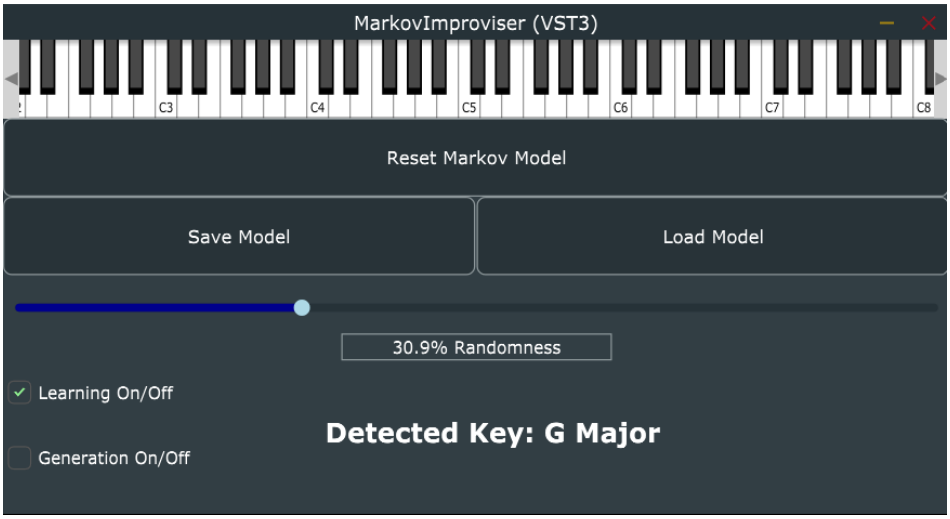

Figure 1: GUI of the plugin

2.3 Generate and Learn Toggles

The two toggles are both connected to two different boolean variables. The *learnOn* variable connected to the *onOffButton* controls the part of the code that is responible of **learning the note parameters**. When it is set to false, the program skips the four functions named in section 2.1 used for the learning process. This variable decides also if the input notes must be considered when computing the most probable detected key.

Similarly, the *canGenerateNotes* boolean variable is linked to the *genButton* toggle. When set to false, the program doesn't execute the part of the code responsible for the **note generation process** contained in the *processBlock* function.

In this way, the user has available a learn-only and a generate-only Markov Model, which are useful to separate the two main processes of the plugin.

2.4 Reset Function

The purpose of the **"Reset Markov Model" button** is to clean the plugin from any learned data, such that the user can start another learning process. This feature is implemented through the *resetMarkovModel* function in the AudioProcessor class, and works both for real-time learned models and for file-imported models.

2.5 Key Detection

An additional feature that helps in musical performance is the **detected key**. Taken as an input one MIDI note from the input stream, the plugin calculates a **score for each key**, which represents how much the input note is related to that key. This score depends on three factors: if the note is the fundamental note of a key (+1), if the note belongs to the first, third or fifth of a key (+1) and if the note belongs to the scale of a key (+1).

For example the C note will get score 3 for the C major key (it is the fundamental, first and belongs to the C major scale) while it will get score 1 for the G major scale (because it belongs to the scale but it's not a first, third or fifth of G major). These scores are stored in the *keyProbs* array which contains the values for all the 24 possible keys (12 major and 12 minor). When each input note is processed, all its key scores are added to the previous array; after the processing of n notes, we can know which key best represents those notes looking at the index which has the highest score in the array. This array is saved with the data of the models when the user presses the "Save Model" button, and overwritten when the load model feature is used: in this way we can know which is **the most probable key** also for loaded models.

2.6 Load and Save Model

These two buttons are useful in a particular way when we get a musically interesting model after a training process. With the **"Save Model" button** the user can save all the information regarding the learned model. The information about the pitches model, the inter-onset intervals model, the durations model, the velocities model and the key detected for the model that is being saved, are converted into a string and written into a *.txt* file.

The **"Load Model" function** takes as an input one of these .txt files, parses it, converts all the information about the 4 models into their respective models (using the *setupModelFromString* function) and sets correctly the *keyProbs* array containing information about the detected key using the data in that file.

2.7 Randomness Selector

To achieve a result that differs more from the original training notes, a **randomness function** was added to the plugin. This function takes as an input **a value in percentage**, which is then converted to a number from 0 to 1 and stored in the float *randomness* variable, that represents the number of notes that need to be randomized. When the *generateNotesFromModel* function is called, which, based on the instantaneous model, computes the next generated note, a number of the notes generated from the model, depending on the randomness value, is modified.

Half of the times the original note is converted into one of the six notes nearest to the original one **belonging to the scale** of the detected key, For example, if the model outputs a C4 note and the detected key is C major, it can be converted into one of these 6 notes: D4, E4, F4 (three higher) and B4, A4, G3 (three lower). This type of variation is particularly good when working with monophonic melodies.

The other half of the times the original note is modified into a note that **belongs to the third, fifth or octave** with respect to the detected key. Also in this case the possible choices for the final note are six. For example, if the note resulting from the model is a D4 and the detected key is C major, the program can choose one of these six notes: E4, G4, C5 (the three nearest third, fifth and octave that are higher than the original note) or C4, G3, E3 (the three nearest third, fifth and octave that are lower than the original note). This second type of variation works well with chords, in which using intervals like second or seventh could result in a bad hearing output.

3 User Evaluation

The goal of the user evaluation was to answer to this question: **how much are musically interesting the outputs of the improvising plugin?** To answer to this question in the best way possible the plugin was trained by a **professional piano player** who played four different musical sheets of different periods and styles. The performance was possible thanks to a four-octave MIDI keyboard, that obviously has limitations in the keyboard extension and in the possibility to play it with the desired velocity.

The four training inputs played were:

1. **Sonatina No. 1 in C major, Op.36** by Muzio Clementi, dated 1797. This piece was composed for didactic purposes and for this reason is very schematic and follows regular patterns. The main problem encountered when hearing to the output of the model trained with this piece is the fact that the musical phrase **ends in a bad way** or is composed by too many notes with respect to the original style. To mitigate this problem, a possibility could be to **introduce an "end-of-phrase"** landmark after a played note when the keyboard is not played for a certain amount of times (e.g. 5 seconds) and restart the generation from zero when the end of phrase is encountered.

2. **Invention No. 8 in F major, BWV 779** by Johann Sebastian Bach, composed in 1723. This piece was thought to be played on the **harpsichord**, so the note velocity was not taken into account in the composing process. In fact, we can hear better results in terms of velocity but the bigger problem with this piece is that we notice **too much repeated notes**. This problem can be resolved modifying the model to get less times a one-order choose. The problem of the bad ending phrase is less relevant with this training set.

3. **Musical moment No. 4, Op. 16** by Sergej Vasil'evič Rachmaninov, dated 1896. This time there was an implementative problem: in fact, the data regarding this musical piece were too much and in the string conversion, when loading the model related, an out of bounds exception was thrown and not the entire model was loaded. This can be solved **lowering the maximum order possible** of the Markov chain (which at the time of the collecting of the data was 100) or playing only a part of the musical piece. Unfortunately, the day of the training we did not hear the improvising result of the model fed with this musical piece, but this result is anyway important to know where to improve the development of the plugin.

4. **These Foolish Things**, written in the early 90s, which is an important jazz masterpiece. The result achieved is probably the better hearing one, but there is an important problem with this musical piece. In fact, the notes generation process sometimes ends in a loop. This could be avoided adding a loop detection function, that restarts the generation process if the same note or chord is played for too many consecutive times.

In all the heard output, the randomness value selected has a significant impact on the overall sound. With low values of randomness (10%-15%) the variations are good hearing and the original style is preserved, while with very high values the piece loses its characteristic style and the final output begin to be confused.

4 Conclusions and Future Improvements

In conclusion, this plugin works both for real-time generated and for imported models, allowing the user to save any interesting model achieved. The randomness selector gives some variations to the output, generating a modified version of the input style.

Anyway, as discussed during the user evaluation in section 3, there are many possible changes to generate better hearing sequences.

Possible ideas for future improvements are:

- For each generated note, randomize the order used to generate that note. In that way, the user will hear less frequently long sequences of notes that resemble exactly the same pattern of the input notes.

- Implement a way to give to the user the possibility of choosing the maximum order possible of the Markov Model, to avoid problems when loading models trained with a big number of notes.

- Implement two different randomness sliders for chords and notes, because while melodies can vary much without losing their musical meaning, chords are more related to the harmonic rules, which are stricter. A random change in a chord could affect in a dramatic way the overall harmony of the musical piece. Another idea could be using two different functions depending on the fact that the output of the model is a chord or a note. As discussed in section 2.7, harmony and melody follow different musical rules that need to be respected.

- Divide the two hands of the player on the keyboard and treat the sequences generated by the two hands separately, eventually using some kind of sensors, to achieve a polyphonic output with more possible variations.

- Add an "end-of-phrase" and a loop detection function as result of section 3.

- Make the GUI better looking with the help of tools and libraries external to *Juce*.