Integrate Markov modelling into a MIDI effect

In this section you are going to find out how to integrate Markov modelling into a JUCE MIDI processing plugin.

The concept is that the plugin receives MIDI at one end and produces MIDI at the other end. In the middle, it learns a model of the incoming MIDI in realtime and uses the learned model to generate the MIDI that comes out of the output.

Modelling pitch

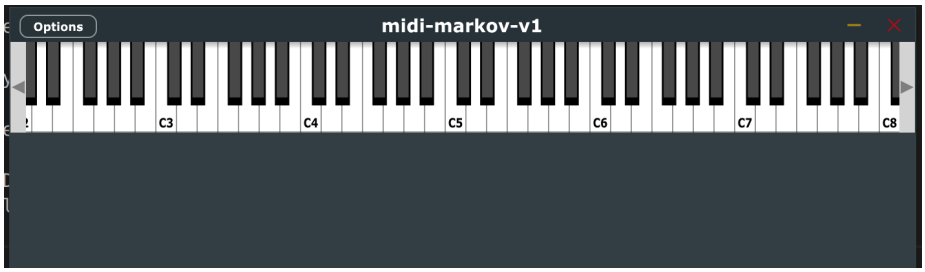

Figure 1: The user interface for the basic JUCE MIDI processing plugin.

You can start by modelling pitch sequences. The starter project already has a basic JUCE plugin architecture in place including a handy on-screen piano keyboard. Figure 1 illustrates the user interface for the starter project. Let's see if you can hook up the on-screen keyboard to a Markov model. The best place for the MarkovManager object is the PluginProcessor class, since the PluginProcessor class ultimately deals with the incoming MIDI. The incoming MIDI can come from the on-screen keyboard or from the host that is hosting the plugin. So in PluginProcesser.h:

```
// at the top
#include "MarkovManager.h"
// ...
// in te private fields of the MidiMarkovProcessor class:
MarkovManager markovModel;
```

In PluginProcessor.cpp, initialise the markovModel variable in the initialiser list on the constructor:

```
...
    ,  markovModel{}
    {
        // the body of the constructor here
    }
```

Now in the MidiMarkovProcessor::processBlock function, you should see a block of code like this:

```
if (midiToProcess.getNumEvents() > 0){
    midiMessages.addEvents(midiToProcess, midiToProcess.getFirstEventTime(), midiToProcess.g
    midiToProcess.clear();
}
```

Remember that this code takes any MIDI that has come in from the on-screen keyboard via the MidiMarkovProcessor::addMidi function and adds it to the main midiMessages buffer which is sent to the processBlock function by the plugin host. After that block of code, iterate over the midiMessages and add all note on messages to the markov model:

```
for (const auto metadata : midiMessages){
    auto message = metadata.getMessage();
    if (message.isNoteOn()){
        markovModel.putEvent(std::to_string(message.getNoteNumber()));
        DBG("Markov model: " << markovModel.getModelAsString());
    }
}
```

This code pulls any MIDI note on messages out of the buffer and adds them to the Markov model. Then it prints out the model. Try running this and pressing the keys on the on-screen piano keyboard. You can see your model rapidly growing.

Generating MIDI notes from the model

Now you have a model you can use it to generate notes and send them to the output. Start by just printing some notes out. Here you encounter your first algorithm design question - when should the algorithm generate a note? The simplest option is to generate a note when it receives a note. Here is some pseudocode for an algorithm. The objective is to process all notes in midiMessages, then for each note on we find, to add a new note to that buffer.

```
midiMessages is a MIDIbuffer received from the host
add notes collected from on-screen piano keyboard to midiMessages
create a new MIDIBuffer: generatedMessages
iterate over midiMessages
   if note on
     add the note to the model with putEvent
     call getEvent on the model and create a new midi note on with the note from the model
     add the new midi note note message to generatedMessages

add generatedMessages to midiMessages so the next plugin in the chain can receive them
```

Here is some code that implements the new parts of that algorithm (some of it is already in our processBlock function):

```
juce::MidiBuffer generatedMessages{};
for (const auto metadata : midiMessages){
```

```
    auto message = metadata.getMessage();
    if (message.isNoteOn()){
        markovModel.putEvent(std::to_string(message.getNoteNumber()));
        int note = std::stoi(markovModel.getEvent());
        juce::MidiMessage nOn = juce::MidiMessage::noteOn(1, note, 0.5f);
        // (const MidiMessage& m, int sampleNumber
        generatedMessages.addEvent(nOn, message.getTimeStamp());
    }
}
// optionally wipe out the original messsages
midiMessages.clear();
midiMessages.addEvents(generatedMessages, generatedMessages.getFirstEventTime(), -1, 0);
```

If you want to play those notes on a synthesizer, you will need note off messages as well. Change the code so it also adds note off messages to the midiMessages buffer. You can set the timing to say, 0.5 seconds after the note on messages. Timestamps in this context are measured in samples, so try this after adding the note on message:

```
juce::MidiMessage nOff = juce::MidiMessage::noteOn(1, note, 0.5f);
generatedMessages.addEvent(nOff, message.getTimeStamp() + getSampleRate()/2);
```