Polphonic Markov Model

In this section you will develop the Markov modelling plugin further so that it can model polyphonic MIDI sequences. The basic principle here is that polyphonic states are just another type of state. Your job is to figure out a way to pass polyphonic states to the Markov model, and to deal with some practicalities relating to humans playing chords on keyboards.

Preparing polyphonic states

So far you have been passing single note numbers to the Markov model like this:

```
markovModel.putEvent(
       std::to_string(message.getNoteNumber())
      );
```

The MarkovManager putEvent function expects to receive a string and the code above simply converts a MIDI note number into a string. So if you want to send it a chord, you just need to figure out how to encode several numbers into a string. Then when the numbers come back out again from 'getEvent', you need code to convert back to several note numbers from your encoded string. Add the following functions to the private section of the PluginProcessor.h file:

```
std::string notesToMarkovState (const std::vector<int>& notesVec);
std::vector<int> markovStateToNotes (const std::string& notesStr);
```

These helper functions will allow you to convert between vectors of notes and strings. Here are implementations:

```
std::string MidiMarkovProcessor::notesToMarkovState(
               const std::vector<int>& notesVec)
{
std::string state{""};
for (const int& note : notesVec){
  state += std::to_string(note) + "-";
}
return state;
}

std::vector<int> MidiMarkovProcessor::markovStateToNotes(
              const std::string& notesStr)
{
  std::vector<int> notes{};
  if (notesStr == "0") return notes;
  for (const std::string& note :
           MarkovChain::tokenise(notesStr, '-')){
    notes.push_back(std::stoi(note));
  }
```

```
  return notes;
}
```

In essence, you create the state for a vector of notes by concatenating them with hyphens. Then to get back to the vector from the state, tokenise on the hyphen. the code uses appropriate const and & syntax to make it clear the sent data is not manipulated and to ensure it is not unnecessarily copied.

Deciding when it is polyphonic: how long between notes?

Now you can pass polyphonic states to the Markov model, you need to work out when your plugin is receiving polyphonic MIDI in its MIDI buffer. You might think - well I will simply check the timestamps of the notes, and if they are the same, it is polyphonic. If they are different, I will treat the notes as individual and pass them separately to the model. Unfortunately, the limitations of human beings might get in the way of that plan. When people play a chord on a MIDI keyboard, the notes do not tend to start at exactly the same time, especially when you measure the start time in samples. They might not even arrive in the same call to processBlock. For example, my RME sound card will reliably run at a sample rate of 96kHz and a buffer size of 16 samples. If my DAW is using those settings, processBlock will only see 16 samples at a time, which is about 0.0002 seconds or 0.2ms. I wrote some test code to see how far apart in time the MIDI notes are when I play a chord, which you can put at the top of your processBlock function:

```
for (const auto metadata : midiMessages){
    auto message = metadata.getMessage();
    if (message.isNoteOn()){
        std::cout << "note "
                  << message.getNoteNumber()
                  << " ts: "
                  << message.getTimeStamp()
                  << std::endl;
    }
}
```

Running on a machine with sample rate 96kHz and buffer size 256 I see the following output when I play a chord:

```
note 76 ts: 147
note 72 ts: 113
note 74 ts: 112
```

The time stamps are not the same - they are up to 30-40 samples or about 0.3ms apart at worst, and they potentially arrive in different calls to processBlock, so you need a way to remember notes across calls to processBlock. Another test is to play a fast monophonic sequence and to check the intervals between notes there.

Using the ChordDetector class

I have implemented a class called ChordDetector which implements some functionality to aid with chord detection. Essentially, you tell it the threshold interval which decides when notes are single or a chord. As discussed above, around 50ms of time works for me, but you may play faster. Then you send notes to the ChordDetector and if the notes are closely enough in time, it accumulates them into a vector which you can retrieve when you want to. You can add a ChordDetector object to your PluginProcessor.h's private fields, and an elapsed time variable to count total samples between notes:

```
ChordDetector chordDetect;
unsigned long elapsedSamples;
```

Initialise them both in your initialiser list. Then set the interval when prepareToPlay is called, since that is when you can calculate the number of samples you want to allow for the chord threshold interval:

```
double maxIntervalInSamples = sampleRate * 0.05; // 50ms
chordDetect = ChordDetector((unsigned long) maxIntervalInSamples);
```

Next add some code to update elapsed time every time processblock is called - at the end of processBlock:

```
elapsedSamples += buffer.getNumSamples();
```

Now you have elapsedSamples counting up the number of samples. Then to detect chords and send them to the Markov model appropriately formatted, replace your MIDI processing loop in processBlock with this:

```
juce::MidiBuffer generatedMessages{};
for (const auto metadata : midiMessages){
    auto message = metadata.getMessage();
    if (message.isNoteOn()){
        chordDetect.addNote(
            message.getNoteNumber(),
            // add the offset within this buffer
            elapsedSamples + message.getTimeStamp()
        );
        if (chordDetect.hasChord()){
            std::string notes =
                MidiMarkovProcessor::notesToMarkovState(
                    chordDetect.getChord()
                );
            markovModel.putEvent(notes);
        }
    }
}
```

In that code, you are calling addNote to tell the chord detector about the note,

querying if it is ready to give you a chord with hasChord, and finally retrieving the chord with getChord. Then you use notesToMarkovState to convert the vector of notes into a suitable string for the Markov model. You might get a single note back from the chord detector if there is not chord - that is fine.

Generating polyphonic output

Now you have fed polyphonic data to your Markov model you can query the model and use the markovStateToNotes function to return the string states back into vectors of note numbers. We can trigger this when the chord detector detects a chord, in processBlock:

```
if (chordDetect.hasChord()){
    std::string notes = markovModel.getEvent();
    for (const int& note : markovStateToNotes(notes)){
        juce::MidiMessage nOn = juce::MidiMessage::noteOn(1, note, 0.5f);
        generatedMessages.addEvent(nOn, 0);
        // remember to note off later
        noteOffTimes[note] = elapsedSamples + getSampleRate();
    }
}
```

You should already have some code in your processBlock function similar to code [code:note-offs-later]. This code sends the note off messages for any pending note offs. You should use AudioPluginHost's MIDI Logger to verify that the note ons and note offs are being sent at the correct times.

So now you have a polyphonic Markov model. Experiment with the plugin in a host environment. See if you can generate some interesting patterns. An interesting way to train the model is to pass it some data from pre-existing MIDI files. There are many MIDI files containing famous pieces of music, jazz solos and such on the internet. If you have a DAW that can import MIDI files, you can load a MIDI file in, play it into your plugin, and have the plugin improvise along as it learns the contents of the file. The big thing you will probably want to change at this point is the timing. At the moment the plugin plays notes when it receives notes. It has no understanding of timing. I am going to into timing in the next section, after one more functional addition: play only mode.

Simplifying the processBlock function

This is a good time to modularise the processBlock function. As you have progressed through the implementation of this musical agent, processBlock has become increasingly spaghetti-like and difficult to debug. Here you are going to break processBlock into a set of separate functions. My approach to programming is that it is fine to break a long block of code into separate functions, even if you only call the functions from one place in the code. Having multiple functions means each has a distinct purpose. Figure 1 presents a breakdown of what

processBlock is doing at the moment.

It is up to you to decide how to break that flow for processBlock down into functions. But I propose the following scheme. Add the following function signatures to the private section of the PluginProcessor:

```
void addUIMessageToBuffer(juce::MidiBuffer& midiMessages);
void learnNotes();
void playNotesFromModel(juce::MidiBuffer& midiMessages);
void addNoteOffs(juce::MidiBuffer& midiMessages);
```

I will leave it to you to work out what you should put into each of those functions, but here is my processBlock function after re-organising it:

```
void MidiMarkovProcessor::processBlock (
        juce::AudioBuffer<float>& buffer,
        juce::MidiBuffer& midiMessages)
{

addUIMessageToBuffer(midiMessages);
juce::MidiBuffer generatedMessages{};
for (const auto metadata : midiMessages){
    auto message = metadata.getMessage();
    if (message.isNoteOn()){
        chordDetect.addNote(message.getNoteNumber(),
                            elapsedSamples + message.getTimeStamp());

        learnNotes();
        playNotesFromModel(generatedMessages);
    }
}
addNoteOffs(generatedMessages);
midiMessages.clear();
midiMessages.addEvents(generatedMessages,
                       generatedMessages.getFirstEventTime(),
                       -1, 0);
elapsedSamples += buffer.getNumSamples();
```

Make sure the plugin is operating correctly after you make these changes.

Play-only mode

Once your model has learned some musical patterns, you might want to be able to just generate from the model without it learning anything else. This functionality should actually be quite straightforward to implement. You just need to be able to switch on or off the bit of code in processBlock that calls putEvent on the markov model. That is the part where the model learns.

You can achieve that by adding a 'learning on-off' toggle to your user interface

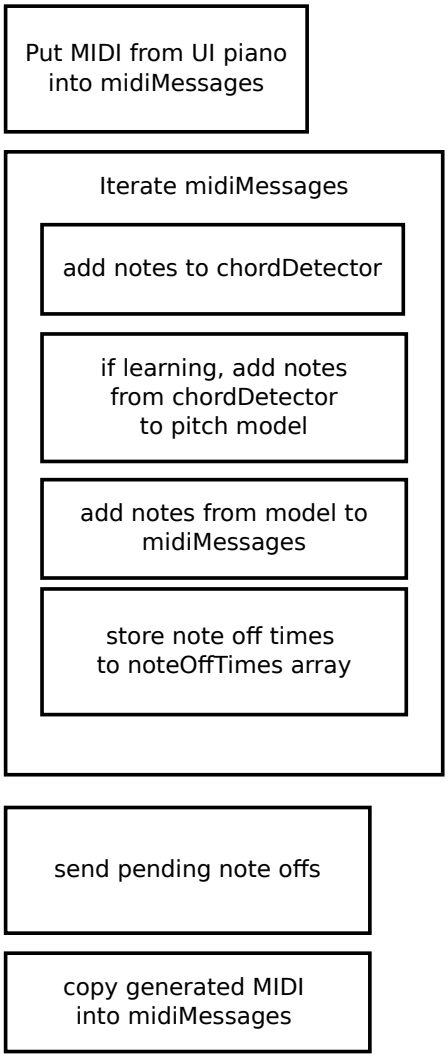

Figure 1: The processBlock function in the polymarkov system. It is ready for modularisation!

and adding functions to your PluginProcessor to switch on or off calls to that putEvent function. I will leave you to work that one out yourself.

Progress check: polyphonic Markov model

At this point you should have a polyphonic Markov model MIDI effect that you can load into a plugin host and train on MIDI input. The plugin will then generate polyphonic MIDI back out from its learned model.