Modelling time

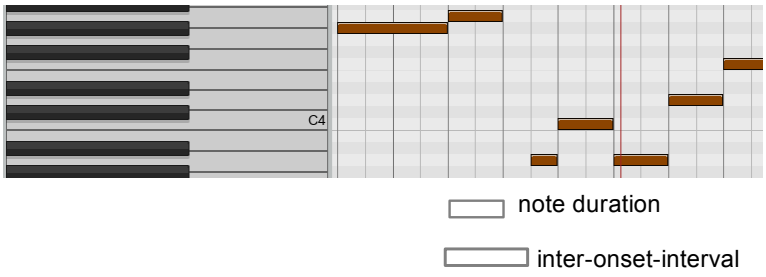

Figure 1: Note duration is the length the note plays for. Inter-onset-interval is the time that elapses between the start of consecutive notes.

How did you get on the polyphonic Markov model? If you did not quite get it working correctly, you can start this section with the code in [code:markov-poly-plugin].

Now we need to teach the musical agent about timing. There are two aspects of timing: how long to play the notes for and how long to wait between notes. I will refer to these two as note duration and inter-onset-interval (IOI).

Modelling note duration

To measure note duration, you can use a similar technique to the one used to prepare note-offs in in the pitch based Markov modellers. That technique involved storing data pertaining to the different notes in a 127 element array. For note duration, you can store the elapsed time when the note-on message arrives, then when the note off message arrives, you can compute the duration of the note. So in PluginProcessor.h, private fields:

```
unsigned long noteOnTimes[127];
```

Intialise all values to zero in the constructor, as you did for the noteOffTimes array, in fact, just use the same loop:

```
for (auto i=0;i<127;++i){
    noteOffTimes[i] = 0;
    noteOnTimes[i] = 0;
}
```

Then when you receive a note on in processBlock, log elapsed time into the array for that note:

```
...
for (const auto metadata : midiMessages){
```

```
    auto message = metadata.getMessage();
    if (message.isNoteOn()){
        noteOnTimes[message.getNoteNumber()] = elapsedSamples;
...
```

So now you know when any note started. The next step is to detect when the note ends and compute its length. Going back to the note iterator, add a new block to deal with note-offs:

```
...
for (const auto metadata : midiMessages){
    auto message = metadata.getMessage();
    if (message.isNoteOff()){
        unsigned long noteLength =
        elapsedSamples - noteOnTimes[message.getNoteNumber()];
        std::cout << "n: "
                  << message.getNoteNumber()
                  << "lasted "
                  << noteLength << std::endl;
    }
...
```

Try it! I see output like this when I play notes on the plugin's built in piano keyboard:

```
n: 108 lasted 4096
n: 69 lasted 4096
n: 76 lasted 25088
n: 81 lasted 13312
n: 91 lasted 14848
```

Now you need a Markov model to model these note durations. You can simply add another MarkovManager object to your PluginProcessor.h private fields:

```
MarkovManager noteDurationModel;
```

Then when a note off is received, pass the note duration to the model, in processBlock:

```
if (message.isNoteOff()){
    unsigned long noteLength =
    elapsedSamples - noteOnTimes[message.getNoteNumber()];
    noteDurationModel.putEvent(std::to_string(noteLength));
}
```

Remember to reset the new model as well in the resetModel function on your PluginProcessor.

That was quite straightforward. You may have observed that the code does not worry about modelling different notes with different lengths. Any note's duration just goes into the model the same way. You might want to tidy the

note-off processing code into a function, as you did previously in the polyphonic Markov example.

Generating note durations from the model

Now you have a model of the note durations, how can you use it to dictate the duration of the notes played by the model? The answer is to use the model to decide how far in the future the note offs will occur, instead of hard coding the value. Remember the code we used to generate note-off times into the future in the polyphonic Markov example?

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

The last line dictates the length of the note, which is hard coded to one second. Instead, try this line, which queries the note duration model for a length:

```
unsigned int duration = std::stoi(noteDurationModel.getEvent(true));
noteOffTimes[note] = elapsedSamples + duration;
```

You should probably not call getEvent for every note in the case where you are playing more than one note - if you are playing a chord, you want each note to have the same duration, and you also do not want to over-query the model. Over-querying the model will cause the duration patterns to not represent the actual learned pattern. So - just call getEvent once, then use the same duration for each note. Experiment with the plugin - try sending it series of long and short notes to see if it is modelling note duration correctly. As usual, use the AudioPluginHost MIDI Logger to check raw MIDI output.

Challenge Can you work out a way to quantise the note durations? If you quantise them you will get a richer (higher order) model as there will be more instances of each duration. Of course, increasing quantisation also makes the timing more 'robotic' and you might not want robotic timing. You can actually go much further with timing. For example, you might want to implement sample accurate timing. In that case, you need to look at the timestamp of the MIDI messages within the buffer, and work on a more accurate 'elapsedSamples' variable.

Model the inter-onset-interval

The second aspect of time mentioned at the start of this section was inter-onset-interval. Measuring the elapsed time between note on events should not be too

difficult. In the PluginProcess.h private fields, add a new model and a new variable:

```
MarkovManager iOIModel;
unsigned long lastNoteOnTime;
```

Initialise lastNoteOnTime to zero in the constructor/ initialiser list.

Now in processBlock:

```
if (message.isNoteOn()){
    ...
    unsigned long interOnsetInt = elapsedSamples - lastNoteOnTime;
    lastNoteOnTime = elapsedSamples;
    iOIModel.putEvent(std::to_string(interOnsetInt));
    ...
}
```

As before, you can tidy that into a function to clear up your processBlock function. I call that function learnIOI in my code. You might want to add some sort of limit for the maximum IOI time. When the IOI model is used to choose how long to wait between notes, very long waits might make the model seem non-responsive. How about a limit of 2 seconds, and a minimum length of 0.05 seconds?

```
unsigned long interOnsetInt = elapsedSamples - lastNoteOnTime;
 lastNoteOnTime = elapsedSamples;
if (interOnsetInt < getSampleRate() * 2 &&
    interOnsetInt > getSampleRate() * 0.05){
    iOIModel.putEvent(std::to_string(interOnsetInt));
}
```

Generate with the inter-onset-intervals

You are making great progress - you have a model of the inter-onset-intervals, but how can you use it to specify the wait between notes? At the moment, the model simply plays a note whenever it receives a note. This is mot very musically interesting. Instead, we want the model to pull an IOI value from the IOI model and to wait that long before it plays the next note. You are going to need two functions and a variable to make this work. Add these to your PluginProcessor.h:

```
unsigned long modelPlayNoteTime;
// return true if time to play a note
bool isTimeToPlayNote();
// call after playing a note
void updateTimeForNextPlay();
```

Set modelPlayNoteTime to zero in the constructor. Here is a step-by-step procedure for what those functions and variables need to do:

1. Call isTimeToPlayNote
2. in isTimeToPlayNote, return true if elapsedTime > modelPlayNoteTime and modelPlayNoteTime > 0, otherwise return false
3. if isTimeToPlayNote returns true, call playNotesFromModel which will generate either a single note or a chord to play
4. if isTimeToPlayNote was true, also call updateTimeForNextPlay
5. in updateTimeForNextPlay, query the IOI model for the next note duration, add it to elapsedSamples and store it to modelPlayNoteTime

Over to you - can you implement the functions and carry out that procedure in processBlock?

Challenge Can you add a load and save feature to the plugin? Or even better, automatically load and save the model by implementing the getStateInformation and setStateInformation functions which are part of the plugin standard. If you implement those functions such that they save and restore the model to and from the sent data, the plugin will start with a trained model when you re-open a project in your DAW or other plugin host. You can call getModelAsString and setupModelFromString on MarkovManager to convert the models into strings. Then you can concatenate the three models into a single string and convert it to the juce::MemoryBlock format needed.

Progress check

At this point you should have a fully working Markov model-based musical agent plugin. It can learn in realtime from a live MIDI feed and it can generate its output at the same time as learning. At this point you can experiment with the musical agent.