MIDI Keyboard Synthesizer

Wells L. Santo

***Abstract*—Musical Instrument Digital Interface (MIDI) devices have long been a bedrock in digital music production, often essential to the use of digital audio workstation (DAW) software such as Pro Tools, Logic Pro, Ableton Live, and FL Studio. This project explores MIDI audio synthesis by means of (1) detecting events from an Alesis Q49 MIDI keyboard using the Python package PyGame, and (2) producing audio output corresponding to those events by using the Python package PyAudio.**

I. Introduction

A valuable and integral component of digital music production is the ability to simulate musical instruments. Instead of being limited to the equipment that a musician has in her immediate possession, digital audio workstations (DAWs) provide the functionality to select from a wide variety of simulated instruments to use during the music production process, from acoustic guitars to horned instruments to grand pianos, all with a high quality of authenticity to their original sound.

In order to use these software instruments, however, a musician needs to utilize specific hardware equipment to play and record different notes of the instrument they wish to simulate. This equipment often comes in the form of a MIDI controller, which is a device that uses the MIDI protocol to send event information from hardware to the musician's computer, so that DAW software can pick up and utilize that information to produce audio output.

In this project, a MIDI keyboard (a device arranged to operate like a musical keyboard) is used in conjunction with Python scripts in order to produce real-time audio. The aim of this project is two-fold: (1) to detect events triggered by playing the MIDI keyboard, and (2) use those events to produce real-time audio that corresponds to the pitches played on the MIDI keyboard.

Furthermore, the completion of this project was achieved using an incremental programming process, where smaller scripts were written sequentially, in order to break up the project into separate subproblems, with each script building off the functionality of the previous script. These scripts can be found in the included `Incremental Scripts` folder within this directory, and an overview of these individual scripts is described in the included `README.md` Markdown file. This project report will attempt to discuss the functionality of the final product as a whole, rather than detail the specifics of the incremental programming process, which is described in detail in `README.md`.

II. Hardware and Software Dependencies

In order to complete this project, one MIDI controller and two non-native Python packages were used. For the MIDI controller, an Alesis Q49 MIDI Keyboard was used, though the final script in this project can be modified to work with any other MIDI controller, with slight modification. For the Python packages, PyGame was used in order to pick up MIDI events from the Alesis Q49, and PyAudio was used in order to produce audio output from the computer's speakers. For any of the project scripts to work, both of these packages (in addition to the Python language) *must* be installed. Though this report will lightly touch on some of the functionality of these packages, it is recommended for anyone using these scripts to look at the full online documentation for both PyAudio and PyGame for a deeper understanding of the packages.

W. Santo is a student within the Computer Science and Engineering Department at the Tandon School of Engineering within New York University, 6 Metrotech Center, Brooklyn, NY 11201.

This paper was written as part of DSP Lab I (EE4163) offered by Professor Ivan Selesnick in Fall 2015, which fulfills the undergraduate Computer Engineering Senior Design Project (DP1) requirement.

III. Detecting MIDI

The first component of this project was to use PyGame in order to detect keypresses on the Alesis Q49 in terms of MIDI events. After the PyGame package has been imported, the MIDI functionality of PyGame must be initialized using the command `midi.init()`. This tells the script to expect to send and/or receive MIDI information to/from an external MIDI controller.

For this project, we only expect to receive information from a single input device, so we must open an input stream that reads from the particular MIDI device that we want. In my case, my Alesis Q49 was set as the default input MIDI device, so the below code was used to open the input stream:

```
INPUTNO = midi.get_default_input_id()
input = midi.Input(INPUTNO)
```

To continuously listen to MIDI events, an infinite while loop is used, which will run for the entire duration of the script. Within this infinite loop, we will poll for events, so that when an event from the MIDI keyboard is detected (that is, when we press a key or let go of a key), we will trigger some corresponding functionality. This can be achieved with the code below, which also captures the MIDI events that were triggered (since multiple keys can be pressed at once) and stores them in the list `eventslist`.

```
while True:
  if input.poll():
    eventslist = input.read(1000)
```

Once the list of MIDI events has been captured, we must iterate through the list of events and make use of the relevant pieces of information to trigger audio output. For this project, we will need to (1) distinguish keydown/keyup events, (2) identify which key was pressed/released, and if it is a keypress, (3) identify the intensity with which the key was pressed. To accomplish this, some empirical testing was done (using the `0_miditest.py` script) in order to figure out what values in the MIDI event list correspond to relevant attributes of the MIDI keyboard. This information can be found in `AlesisQ49MIDIvalues.txt`, which includes the finding that the "event type" value for a keydown event is 144 and the "event type" value for a keyup event is 128. The following code picks out the relevant information from `eventslist` and prints out information about keydown events, when they are detected:

```
KEYDOWN   = 144
KEYUP     = 128

for e in eventslist:
  event      = e[0]
  eventType  = event[0]
  eventID    = event[1]
  eventValue = event[2]

  if eventType == KEYDOWN:
    print 'Keydown on key', eventID, 'with
    intensity', eventValue
```

Notice that this code will keep track of what key was actually pressed on the keyboard (`eventID`) as well as the intensity with which that key was pressed (`eventValue`). The variable `eventType` keeps track of whether a keydown or keyup event has occured, and we set the macro `KEYDOWN` to 144, to correspond to the code for the keydown event, which is specific to the Alesis Q49. For different MIDI controllers, this value may vary, and must either be empirically tested or looked up in a reference manual. With keydown events captured, all that remains is actually outputting audio when a keydown is detected.

IV. PyAudio

In order to generate audio, we will use PyAudio in conjunction with a second-order difference equation in order to compute a pure-tone signal and output it to the speakers. This code is essentially the same as that which was provided in the course, with the modification that we will keep track of a list of different coefficients that correspond to different frequencies for the second-order difference equation. Based on which key is pressed on the MIDI keyboard, we will use a different set of coefficients to trigger audio of a specific pitch.

The process of computing the output of the second-order difference equation, using the `struct` package to convert that output value to hexadecimal, and sending it to the PyAudio stream is the same as what was done in class. In the following section, some explanation will be given for the code that generates a list of frequencies to be used.

V. Musical Pitches

Since we want a faithful reproduction of the pitches that we expect to hear when playing the keyboard, we will need to get the specific frequencies that correspond to each pitch. Luckily, the current standard in music is for the note A4 to correspond to the frequency of 440 Hz, where 'A' is the name of the pitch (in a 12-pitch system) and '4' is the octave that the pitch is played in.

I will not digress too much into music theory here, other than mention that different pitches on the keyboard correspond to specific frequencies, and all consecutive pitches on the keyboard are separated by the same factor in frequency. The central piece of information that we will make use of is the fact that frequency is doubled across one octave. That is, since A4 is 440 Hz, the note A5 is 880 Hz, and the note A3 is 220 Hz. Since the separation of 12 notes is equal to a factor of 2 in frequency, the separation of consecutive notes can be calculated as $log_{12}(2) \approx 1.059463$. Using this information, we can create a list of frequencies that increase by a factor of 1.059463 from some lowest frequency starting pitch. For my project, I use 16.35 Hz as the starting frequency, which is the frequency that corresponds to the note C0, which is the lowest note playable on a standard grand piano. I use Python's list comprehension to create the list of frequencies, as seen below:

```
f = [16.35 * 1.059463 ** i for i in range
    (0, 120)]
```

Since my Alesis Q49 can play up to 120 notes, my list is created to accomodate at least 120 different frequencies. Using these frequencies, I will also have three other lists of length 120 to keep track of the different possible coefficients that can be used with the second-order difference equation, so that audio tones of different frequencies can be created on the fly. When a keypress is detected from the Alesis Q49, the `eventID` variable is used to figure out what frequency needs to be played. Therefore, we will use `eventID` to select the index of the correct coefficients to use with the second-order difference equation to generate audio output.

VI. Multiple PyAudio Streams

Up to this point, we have discussed how keypresses on the Alesis Q49 keyboard can be detected, and how different frequencies that correspond to the different pitches on the keyboard can be played. However, the code that we have used in the course so far has the limitation of only being able to play one pitch at a time. For real MIDI synthesizers, we should be able to simulate the ability to play multiple notes on the keyboard simultaenously, so it was important in this project to find a way to play more than just one pitch at a time. Getting around this limitation was potentially the most difficult part of this project, since there were potentially a few different ways to go about it.

One possible solution was to use multiple second-order difference equations to represent the distinct output signals of potentially different frequencies, and then sum the values of the two signals. This would invoke the concept of superposition, since two signals can be directly added in the time domain and still retain the frequency content of both, together. Although this may be a more mathematically pure solution, it has some limitations in implementation, especially when considering that we can only pack a finite amount of values to the output stream. If we use this approach, clipping may occur too easily. Trying to scale the values dynamically would be somewhat mathematically involved as well, and when considering the fact that many notes can be played at once, this may become too complicated of a solution to implement nicely. For my project, I decided to explore an alternate approach, which takes advantage of the features already provided by PyAudio.

My solution was as follows: In order to play multiple pitches, we will open multiple PyAudio streams, and simply play one pitch per stream. This, of course, assumed that PyAudio would be able to support the use of multiple streams and that the output of each stream would be played simultaneously. By using the `5_doublepitchtest.py` script, I was indeed able to confirm that multiple streams could be opened to play two different pitches at the same time.

A second question that then arose was how to utilize multiple streams to keep track of the notes presently being played. One potential approach is to just open a single stream per note on the keyboard, but since my Alesis Q49 supports up to 49 keys played at once, keeping open 49 streams to be continuously updated might be too computationally heavy to deal with. Instead, in order to keep the number of streams relatively small, we will store the streams in a circular queue, so that the stream that was used earliest in time (and thus the most likely to not have a pitch still playing) will

be the one that we use next for output. This way, we are likely to only utilize a stream that has no audio playing (or where a pitch has already played out completely). In order to accomplish this, we will once again use Python's list comprehensions to create a list of PyAudio streams.

```
# Choose how many notes you want to be able
    to play at once
NOSTREAMS = 10

# Put the streams into a circular buffer
# Use list comprehension to make the
    streams directly in this list
stream_buffer = [p.open(format = p.
    get_format_from_width(sampleWidth),
    channels = numChannels,
    frames_per_buffer = blockSize,
    rate = samplingRate,
    input = False,
    output = True)
  for i in range(NOSTREAMS)]
```

In the above code, 10 streams are created, and we will utilize each stream one at a time, as new keydown events are detected. It is important to note that we need to continuously update *every* stream throughout the execution of our program, even when keypresses are not detected, so that all of the old pitches that have been played are able to sound out completely. In order to accomplish this, we will need to keep track of which pitch is used per stream and compute the output of a second-order difference equation at every instance, per stream. That is, along with the circular buffer of streams, we will need to keep a circular buffer of frequencies as well. To use a circular buffer implementation, we will also need to keep track of one index value, which represents the next available stream and frequency to utilize on keypress. The code is shown below:

```
pitch = [0 for i in range(NOSTREAMS)]
accesskey = 0

...

# When a keypress event is triggered:
accesskey = (accesskey + 1) % NOSTREAMS
```

Finally, at each instant, we need to make sure to compute the output of the difference equation *for each stream*, pack that information, and send it to each stream. This is seen in the following code snippet:

```
# Update the value of the difference
    equation
for n in range(blockSize):

  # Update output for all streams
  for i in range(NOSTREAMS):
    y[i][n] = b0[pitch[i]] * x[i][n] -
    a1[pitch[i]] * y[i][n-1] -
    a2 * y[i][n-2]

# Output the value of all streams
    concurrently
for i in range(NOSTREAMS):
  y[i] = np.clip(y[i], -2**15+1, 2**15-1)
  data = struct.pack('h'*blockSize, *y[i])
  stream_buffer[i].write(data, blockSize)
```

It is important to note that although we are writing to each stream sequentially in the code, PyAudio will automatically collect all of the streams together and send them to the audio card to be processed concurrently. This way, the sound of all of the streams will be played simultaneously, which is what we desire. In total, we will now be able to output multiple pitches of audio at the same time, by using a circular buffer of PyAudio streams, each being updated with output from a second-order difference equation at every instance of the program's execution.

VII. Conclusion

In conclusion, this project was successfully able to detect (multiple) keypresses on my Alesis Q49 MIDI controller and generate correctly pitched audio in real-time. This was accomplished by using the packages PyGame and PyAudio, which have the built-in functionality to interface with MIDI devices and then pack time-domain values of an output audio signal to play tones from a computer's speaker, respectively. Overall, much was learned in this project, especially in figuring out how to play multiple tones at once, but much can still be done to extend the project. For instance, the intensity of the keypress and the duration of the keypress could be used to influence the duration of the tone (aka the decay time) generated from the second-order difference equation. Higher order difference equations may also be used to produce better sounding audio, potentially closer in authenticity to that of an actual keyboard. In any case, this project was quite a success, and the scripts that were used can be found online on GitHub at https://github.com/Devking/PythonMIDISynth.

VIII. Appendix: The MIDI Keyboard Synthesizer Script

```
# Get a range of the frequencies for 120 keys, starting with the pitch C0
# This mathematically works because there is a separation of 1.059463 per
# pitch, in frequency. You may not be able to hear pitches until you get to
# around C4, which is the 48th key.
f = [16.35 * 1.059463 ** i for i in range(0, 120)]

# These macros come from the MIDI event values that my Alesis Q49 uses for
# detecting KEYUP and KEYDOWN events.
KEYDOWN   = 144
KEYUP     = 128

# Choose how many notes you want to be able to play at once
NOSTREAMS = 10

# Import all the necessary packages
from pygame import midi
import pyaudio
import struct
import numpy as np
from math import sin, cos, pi


################################
# Initialize sound parameters #
################################

# This is based on the second-order difference equation code that we have used
# in the class, written by Professor Ivan Selesnick.

blockSize    = 32
sampleWidth  = 2
numChannels  = 1
samplingRate = 16000

Ta = 0.8
r = 0.01 ** (1.0 / (Ta * samplingRate))

# Calculate coefficients based on frequencies
om = [2.0 * pi * float(f1) / samplingRate for f1 in f]
a1 = [-2*r*cos(om1) for om1 in om]
a2 = r**2
b0 = [sin(om1) for om1 in om]

# Open the audio output streams
p = pyaudio.PyAudio()

# Put the streams into a circular buffer
# Use list comprehension to make the streams directly in this list
stream_buffer = [p.open(format            = p.get_format_from_width(sampleWidth),
                        channels          = numChannels,
                        frames_per_buffer = blockSize,
                        rate              = samplingRate,
                        input             = False,
                        output            = True)
                 for i in range(NOSTREAMS)]

# Circular buffer of arrays
y = [np.zeros(blockSize) for i in range(NOSTREAMS)]
x = [np.zeros(blockSize) for i in range(NOSTREAMS)]

# Circular buffer of ints
pitch = [0 for i in range(NOSTREAMS)]
accesskey = 0

```

```
65
66 ########################################
67 # Initialize input detection for MIDI #
68 ########################################
69
70 midi.init()
71 INPUTNO = midi.get_default_input_id()
72
73 print '******************'
74 print '** Ready to play **'
75 print '******************'
76
77 while True:
78
79     # For ALL streams, set current input to 0, since nothing is being played
80     # at the current moment (until a key is pressed)
81     for n in range(NOSTREAMS):
82         x[n][0] = 0.0
83
84     if input.poll():
85         eventslist = input.read(1000)
86
87         for e in eventslist:
88             event      = e[0]
89             eventType  = event[0]
90             eventID    = event[1]
91             eventValue = event[2]
92
93             if eventType == KEYDOWN:
94                 print 'Keydown on key', eventID, 'with intensity', eventValue
95
96                 # Trigger an impulse due to a keypress
97                 # Notice that we are triggering the impulse in the stream that
98                 # 'accesskey' refers to -- and then we will update 'accesskey'
99                 # to utilize our circular array of streams properly.
100                x[accesskey][0] = 15000 * (eventValue / 130.0)
101                pitch[accesskey] = eventID % 60;
102                accesskey = (accesskey + 1) % NOSTREAMS
103
104            elif eventType == KEYUP:
105                print 'Keyup on key', eventID
106
107    # Update the value of the difference equation
108    for n in range(blockSize):
109
110        # Update output for all streams
111        for i in range(NOSTREAMS):
112            y[i][n] = b0[pitch[i]] * x[i][n] - a1[pitch[i]] * y[i][n-1] - a2 * y[i][n-2]
113
114    # Output the value of all streams (this all happens at once!)
115    # PyAudio will allow this to play the output CONCURRENTLY
116    # (That is, sound from all streams will play at the same time, not one after the other!)
117    for i in range(NOSTREAMS):
118        y[i] = np.clip(y[i], -2**15+1, 2**15-1)
119        data = struct.pack('h' * blockSize, *y[i])
120        stream_buffer[i].write(data, blockSize)
121
122 # Close up all of the streams properly
123 for i in range(NOSTREAMS):
124     stream_buffer[i].stop_stream()
125     stream_buffer[i].close()
126
127 p.terminate()
128 midi.quit()
```