Building a Graphics User Interface using PyQt

Audio Programming and Signal Processing 4

Sarah Clark

The aim of this project was to write a code to respond to MIDI signals produced by a MIDI keyboard, and to create a GUI using Qt that was able to interact with these signals. The MIDI signals would be produced by a virtual MIDI keyboard that was downloaded, and connected to a Python program in order to synthesise the correct keyboard note. A graphical user interface was then designed to allow users to make alterations to the sound. For example, here it was decided to have a volume slider on the GUI to change the volume of the notes, and to have the option between sine waves and square waves. This project was mainly created in Python, using Qt for Python, 'PyQt'.

1 Oscillator Class Written in C++

The first part of this project was to create an oscillator to synthesis the MIDI keyboard notes. A C++ class, 'Oscillate', was written, containing two functions. The first was to generate a sine wave recursively, given a wave frequency and a sampling frequency. The second function was to generate a square wave, given a wave frequency, sampling frequency and duty cycle. Using SWIG, the C++ code was wrapped for use in Python. The code files are shown below:

1.1 Oscillator.h

```
#include <stdio.h>
#include <math.h>

class Oscillate {
   public:
      void wavesine (double *arr1, std::size_t fs1, float f1);
      void wavesquare (double *arr2, std::size_t fs2, float f2, float
duty);
};
```

1.2 Oscillator.cxx

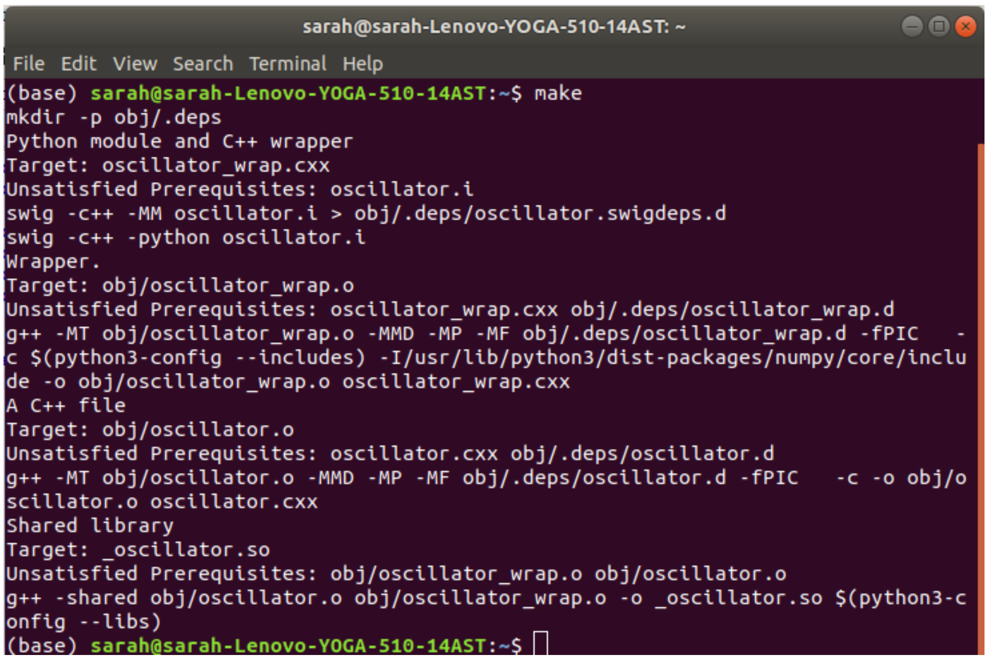
```
#include <stdio.h>
#include <math.h>
#include <iostream>
```

```
#include "oscillator.h"
#define PI 3.14159265

//Function to generate sine wave
void Oscillate::wavesine (double *arr1, std::size_t fs1, float f1)
{
   int i = 0;
   //Set input recursive variable values
   float one = 1;
   float two = 0;
   //Find radial frequency value
   float wT = (2*PI*f1)/fs1;
   for (int i = 0; i < fs1; i++)
   {
      float acc1 = one * (2*cos(wT));
      float acc2 = two * -1;
      arr1[i] = (acc1 + acc2);
      //Update recursive variables
      two = one;
      one = arr1[i];
   }
}

//Function to generate square wave
void Oscillate::wavesquare (double *arr2, std::size_t fs2, float
f2, float duty)
{
   int i = 0;
   int j = 0;
   float c = float(fs2);
   //Calulate length of one wave cycle
   float len = c/f2;
   //Calulate length of 'up' section of wave based on duty cycle value
   float dutylen = duty*len;
   for (int i = 0; i < fs2; i++)
   {
      //For each cycle length, set duty cycle percentage of wave
      //As 1 and the remainder as -1
      if (j<len)
      {
         if (j < dutylen)
         {
            arr2[i] = 1;
         }
         if (j> dutylen)
      {
         arr2[i]= -1;
      }
      j++;
   }
   //Reset for next wave cycle
   if (j>len)
   {
   j = 0;
      }
   }
}
```

1.3 Oscillator.i

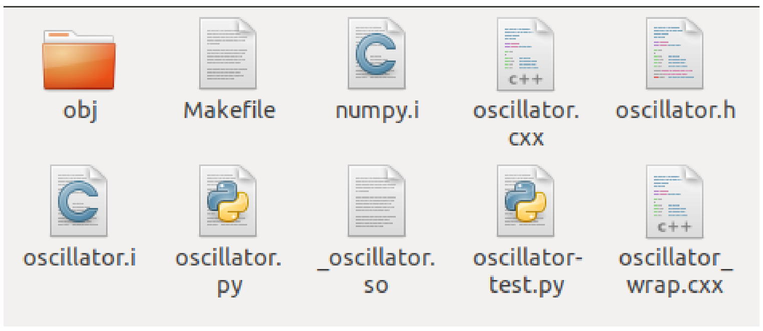
```
/* Name our python module */
%module oscillator

%include <std_string.i>
%include <math.i>

%{
   #define SWIG_FILE_WITH_INIT
   #include "oscillator.h"
%}

%include <numpy.i>
%init %{
import_array();
%}

%apply (double* ARGOUT_ARRAY1,int DIM1) {(double *arr1, std::size_t
fs1)};
%apply (double* ARGOUT_ARRAY1,int DIM1) {(double *arr2, std::size_t
fs2)};

%include "oscillator.h"
```

These files were built using ’make’.


sarah@sarah-Lenovo-YOGA-510-14AST: ~
File Edit View Search Terminal Help

```
(base) sarah@sarah-Lenovo-YOGA-510-14AST:~$ make
mkdir -p obj/.deps
Python module and C++ wrapper
Target: oscillator_wrap.cxx
Unsatisfied Prerequisites: oscillator.i
swig -c++ -MM oscillator.i > obj/.deps/oscillator.swigdeps.d
swig -c++ -python oscillator.i
Wrapper.
Target: obj/oscillator_wrap.o
Unsatisfied Prerequisites: oscillator_wrap.cxx obj/.deps/oscillator_wrap.d
g++ -MT obj/oscillator_wrap.o -MMD -MP -MF obj/.deps/oscillator_wrap.d -fPIC   -
c $(python3-config --includes) -I/usr/lib/python3/dist-packages/numpy/core/inclu
de -o obj/oscillator_wrap.o oscillator_wrap.cxx
A C++ file
Target: obj/oscillator.o
Unsatisfied Prerequisites: oscillator.cxx obj/.deps/oscillator.d
g++ -MT obj/oscillator.o -MMD -MP -MF obj/.deps/oscillator.d -fPIC   -c -o obj/o
scillator.o oscillator.cxx
Shared library
Target: _oscillator.so
Unsatisfied Prerequisites: obj/oscillator_wrap.o obj/oscillator.o
g++ -shared obj/oscillator.o obj/oscillator_wrap.o -o _oscillator.so $(python3-c
onfig --libs)
(base) sarah@sarah-Lenovo-YOGA-510-14AST:~$ 
```

(a) Using ’make’ to build C++ Ocillator class



(b) C++ Oscillator class source and built files

Figure 1

2 MIDI

In order to know when a note on the keyboard was being played, and which note, MIDI signals needed to be received by the program.

MIDI is a protocol which allows electronic instruments and other digital musical tools to communicate with each other. It is an acronym for **M**usical **I**nstrument **D**igital **I**nterface[1]. It was developed in the 1980s in order to allow increased compatability between digital music hardware by different manufacturers.

Importantly, MIDI doesn't transmit an audio signal, it only sends information. Therefore, a MIDI instrument needs an onboard sound source, i.e. a synthesier or a sampler, to make a sound. MIDI messages are digital data transmissions that tell the music equiptment what to do. In this project the MIDI message type that was transmitted was 'Note_on/off', which indicates whether the note is being depressed or released. This message has the following attributes associated with it:

Channel Number: The MIDI channel off the note

Note Number: The MIDI note number. For every musical note, a MIDI note number has been assigned, e.g. middle C is assigned the number 60.

Velocity: How fast the note is struck or released

Time: The length of the note

2.1 Receiving MIDI Signals in Python

Mido is a library for working with MIDI messages and ports.[2] It can be used to open input and output ports in the Python program to send and receive MIDI signals.

Before using 'mido' a backend, i.e. an API for using MIDI, needs to be selected and installed. When mido is downloaded, it includes several backend options, the default being 'Rtmidi'. When initialising 'mido' in a project , if the backend is not declared, this default option will be used, which is what was done in this project.

In order to receive MIDI signals from the virtual keyboard, a port for signals needed to be opened using. Using 'mido', this could be done by:

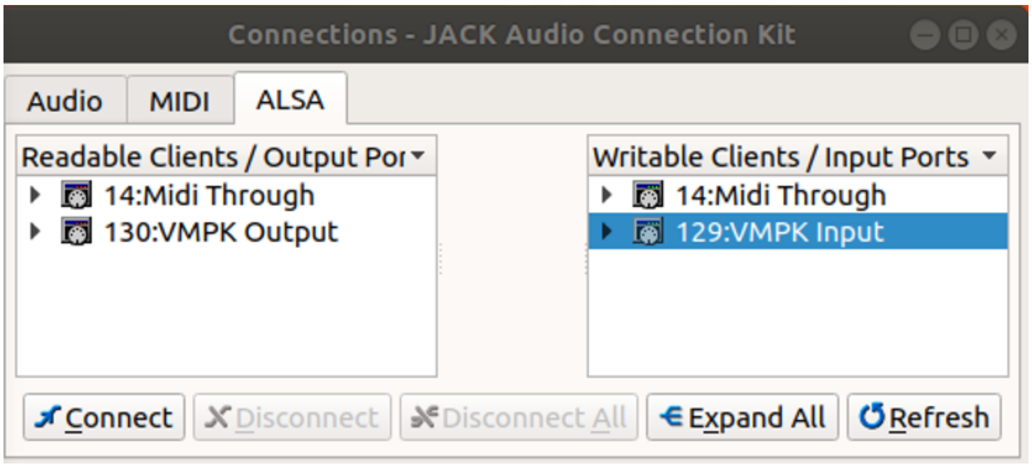
```
port = mido.open_input('port name',virtual=True)
```

Messages can then be received on the port in a continous loop using:

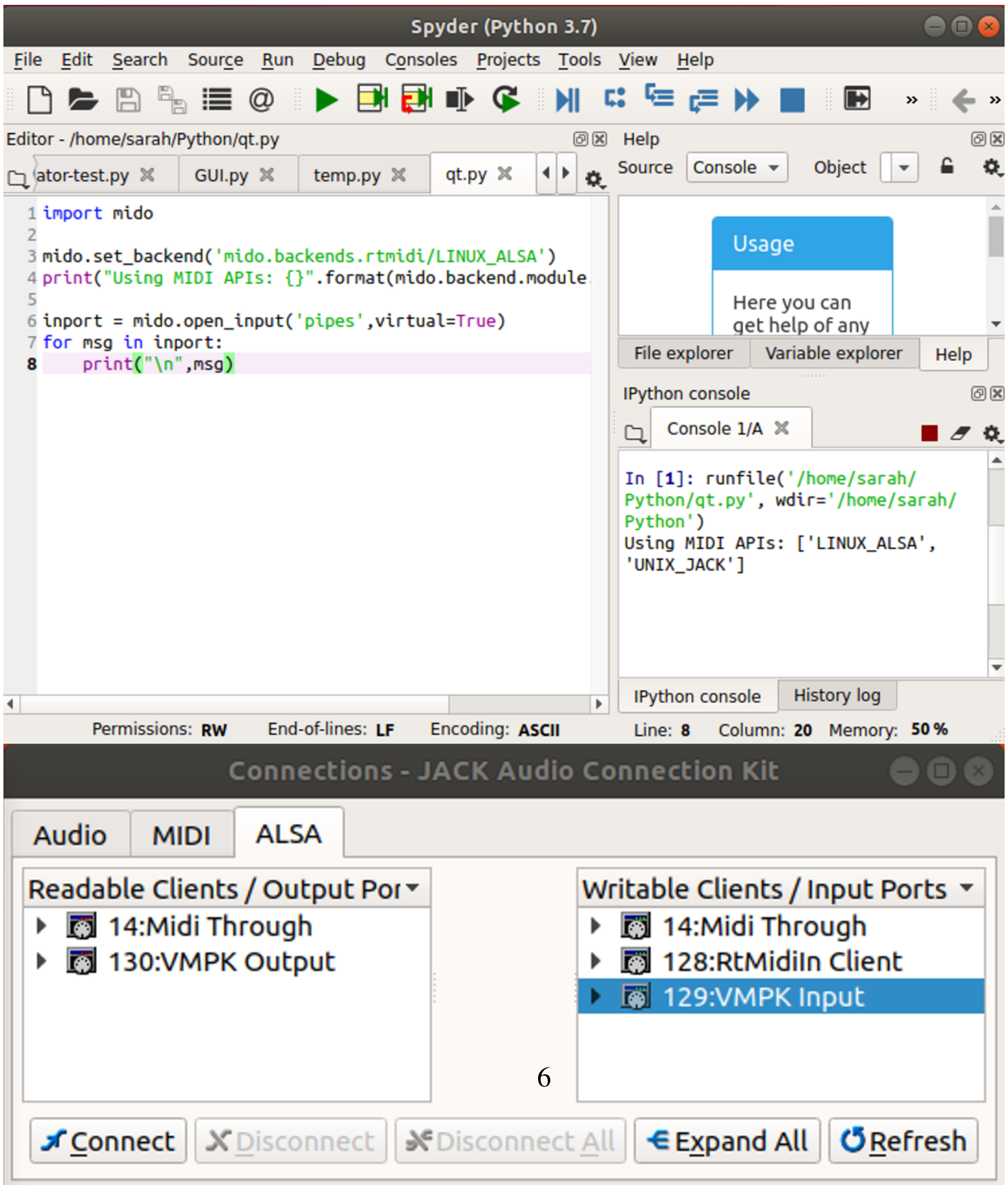
```
for msg in port:
```

An example of opening a port this was can be seen in Figure 2 below. To observe input and output MIDI ports available, the program QJackctl was used, where MIDI ports could be seen in the 'Connections/ALSA' tab. When the input port was opened in Python, this could then be connected to the output port of the virtual keyboard so that MIDI signals from this could be recieved.

With the ports now connected, when a note on the keyboard was pressed, the message was received in the Python progam. This can be seen in Figure 3.



(a) Before running the code, there is no midi input port to code


Spyder (Python 3.7)
File Edit Search Source Run Debug Consoles Projects Tools View Help
Editor - /home/sarah/Python/qt.py
ator-test.py
GUI.py
temp.py
qt.py
Help
Source Console Object
Usage
Here you can get help of any
File explorer Variable explorer Help
IPython console
Console 1/A
IPython console History log
Permissions: RW End-of-lines: LF Encoding: ASCII Line: 8 Column: 20 Memory: 50 %

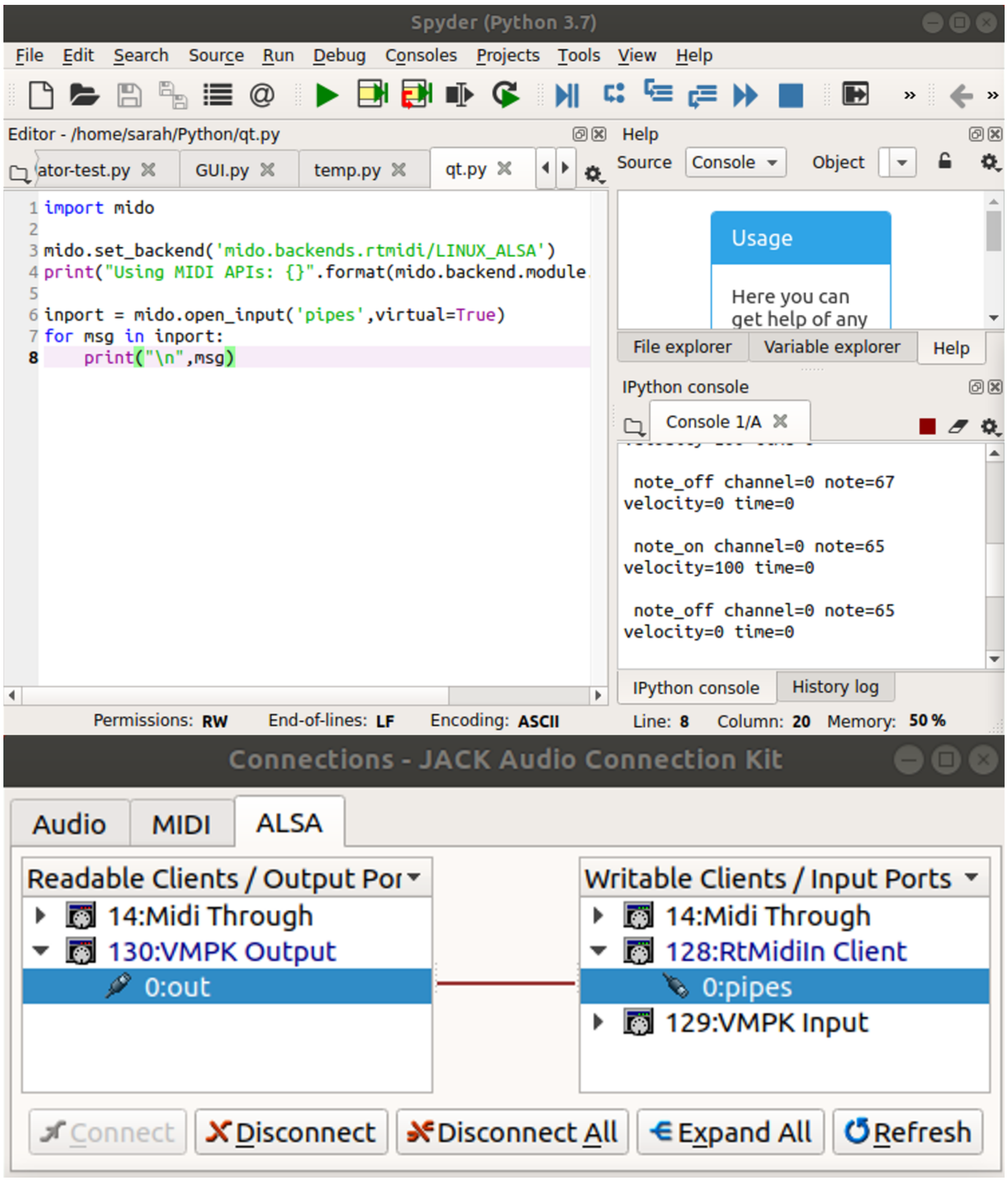
```
import mido

mido.set_backend('mido.backends.rtmidi/LINUX_ALSA')
print("Using MIDI APIs: {}".format(mido.backend.module

inport = mido.open_input('pipes',virtual=True)
for msg in inport:
    print("\n",msg)
```

```
In [1]: runfile('/home/sarah/
Python/qt.py', wdir='/home/sarah/
Python')
Using MIDI APIs: ['LINUX_ALSA',
'UNIX_JACK']
```

Connections - JACK Audio Connection Kit
Audio
MIDI
ALSA
Readable Clients / Output Por
14:Midi Through
130:VMPK Output
Writable Clients / Input Ports
14:Midi Through
128:RtMidiIn Client
129:VMPK Input
Connect
Disconnect
Disconnect All
Expand All
Refresh

(b) Open a port, which appears on the QJackctl ALSA connections tab

Figure 2: Opening a MIDI port in Python


Spyder (Python 3.7)
File Edit Search Source Run Debug Consoles Projects Tools View Help
Editor - /home/sarah/Python/qt.py
ator-test.py
GUI.py
temp.py
qt.py
Help
Source Console Object
Usage
Here you can get help of any
File explorer Variable explorer Help
IPython console
Console 1/A
IPython console History log
Permissions: RW End-of-lines: LF Encoding: ASCII Line: 8 Column: 20 Memory: 50 %
Connections - JACK Audio Connection Kit
Audio MIDI ALSA
Readable Clients / Output Por
14:Midi Through
130:VMPK Output
0:out
Writable Clients / Input Ports
14:Midi Through
128:RtMidiIn Client
0:pipes
129:VMPK Input
Connect Disconnect Disconnect All Expand All Refresh

```
import mido

mido.set_backend('mido.backends.rtmidi/LINUX_ALSA')
print("Using MIDI APIs: {}".format(mido.backend.module

inport = mido.open_input('pipes',virtual=True)
for msg in inport:
    print("\n",msg)
```

```
 note_off channel=0 note=67
velocity=0 time=0

 note_on channel=0 note=65
velocity=100 time=0

 note_off channel=0 note=65
velocity=0 time=0
```

Figure 3: Connect output MIDI signals from virtual keyboard to code input port. Recieve MIDI signals in code when a note is pressed

2.2 Frequency of MIDI signals

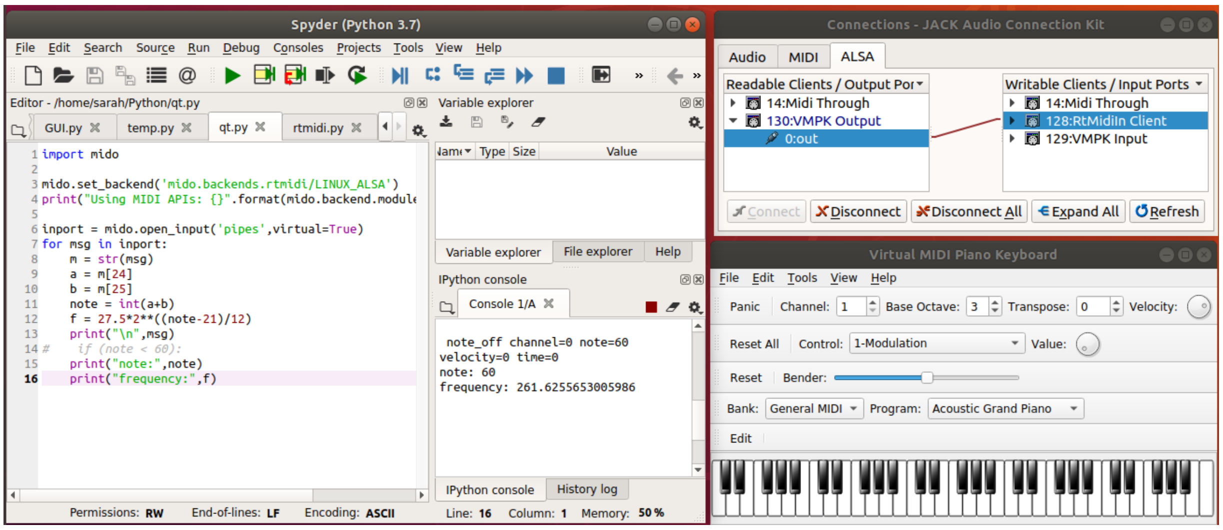
MIDI signals were received in the way described in section 2.1, by creating a mido port and connecting input and outputs. However, for the purposes on this project the only information that was required from the MIDI signal was the note number everytime a ’note_on’ occurred. This is because the timing of the note was not important as we wanted to play a short sample of the note pressed every time, and also the note 'velocity' would be determined instead by the volume slider (and also both of these values were always sent as zero from the MIDI keyboard).

Therefore, it was important to be able to select the useful information from the MIDI signal. This was done by converting the MIDI signal to a string and selecting the elements which detemined the MIDI note number so that frequency could be determined. Every time a key was pressed, two MIDI signals were sent, 'note_on' and 'note_off', but we only wanted the sound to play once, so the element of the string that determined whether this was a 'note_on' or 'note_off' signal was also found. This can be seen in the full code at the end of the report.

If the note was found to be a 'note_on', the next step was to determine the frequency of the MIDI note that had been sent. A formula to convert between MIDI note and frequency was found to be: [3]

$$frequency = 27.5x2^{(}\frac{note-21}{12}) \tag{1}$$

Therefore, the receieved MIDI note was converted in this way to find a note frequency value. Having found the frequency needed, this value could then be used in cunjunction with the C++ 'Oscillate' class to create a wave array.


```python
import mido

mido.set_backend('mido.backends.rtmidi/LINUX_ALSA')
print("Using MIDI APIs: {}".format(mido.backend.module

inport = mido.open_input('pipes',virtual=True)
for msg in inport:
    m = str(msg)
    a = m[24]
    b = m[25]
    note = int(a+b)
    f = 27.5*2**((note-21)/12)
    print("\n",msg)
#     if (note < 60):
    print("note:",note)
    print("frequency:",f)
```

```
 note_off channel=0 note=60
velocity=0 time=0
note: 60
frequency: 261.6255653005986
```

Figure 4: Finding the correct frequency of the note pressed from the transmitted MIDI signal

3 Graphical User Interface

The final part of the project was to create a graphical user interface (GUI) to be able to interact with the received MIDI signals. The GUI was made using a PyQt, an open source toolkit which can be used to develop GUIs and multiplatform applications. In this project we use the modules:

QtCore: A base library, containing classes used by other modules such as threading, I/O facilities and other non GUI functionality.

QtWidgets: Contains classes for GUI applications built up of widgets, i.e. applications or interfaces that enable users to perform a function.

3.1 QObjects and QWidgets

The QObject class is found inside the QCore library and is one of the main building blocks in Qt. Most of the Qt classes inherit from this class as seen in Figure 1. Some of the main properties of the QObject class include the parenting system, and slots and signals.

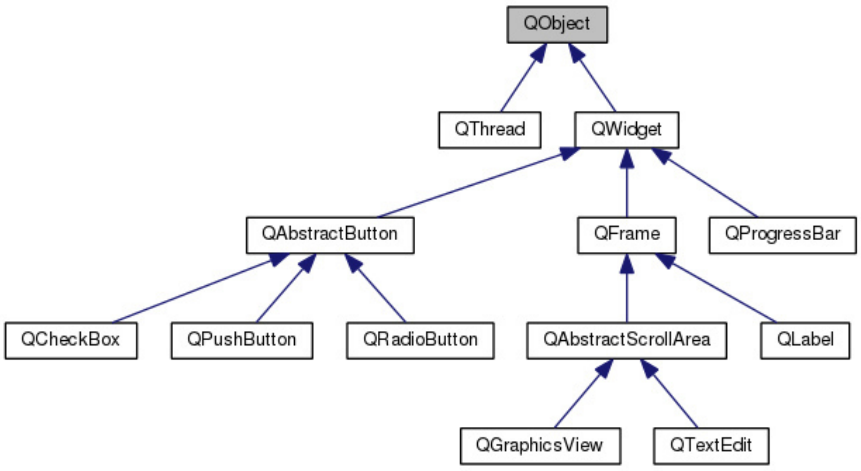


Figure 5: Qt Class Hierachy [5]

In Qt, a GUI can be build using widgets, created using the QWidget class which inherits from QObject. Widgets respond to events that are typically caused by user interaction, e.g. mouse clicks.

3.1.1 Parenting System

Any object that inherits from QObject, e.g. QWidget, can have a parent and children. QObjects are organised in object trees, i.e. when a QObject is created with another object as the 'parent', the object will automatically become a 'child' of that object.[6] Child widgets in a QWidget automatically appear inside the parent widget.

When something changes in the parent, e.g. it is resized, the children are notified to update themselves as well. Furthermore, when a parent object is destroyed, all of its children are also destroyed which can be useful for memory management. All QObjects have *findChild()* and *findChildren()* methods that can be used to search for children of a given object.

3.1.2 Signals and Slots

An important function of the QObject class is the use of signals and slots.

Signal: A signal is a message that can be sent by the QObject, usually to inform of a status change.

Slot: A slot is a function that is used to accept and respond to a signal.

If an event takes place, e.g. a button is pressed, the QObject can emit a signal. If the signal is connected to a slot, when the signal arrives the function contained in the slot will be called. It is important to note, however, that the signal does not execute any action on it's own. Unless it is connected to a slot, emitting a signal will have no affect on the rest of the program.[7]

Signals and slots are important as they can be used for communication between objects. This can be seen in Figure 2 below.

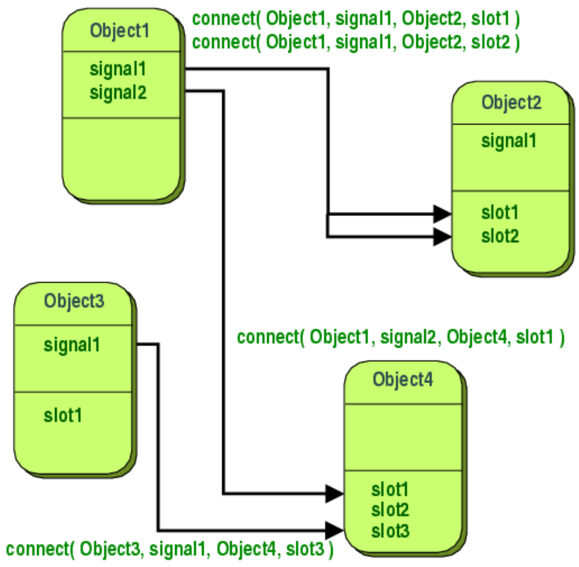

Object1
signal1
signal2
connect(Object1, signal1, Object2, slot1)
connect(Object1, signal1, Object2, slot2)
Object2
signal1
slot1
slot2
Object3
signal1
slot1
connect(Object1, signal2, Object4, slot1)
Object4
slot1
slot2
slot3
connect(Object3, signal1, Object4, slot3)

Figure 6: Slots and Signals between Objects [8]

Important features of signals and slots: [9]

- A signal may be connected to many slots.
- A slot may be connected to many signals.
- A signal may also be connected to another signal. (This is signal relaying, e.g. the second signal is sent if the first signal is sent.)
- Signals can send information as arguments.
- Connections may be direct (ie. synchronous) or queued (ie. asynchronous).
- Connections may be made across threads.
- Signals may be disconnected.

3.2 GUI Appearance

For this project, using an GUI, the idea was to be able to change the volume of the note played, and also whether this wave was a sine wave or square wave. This was built using widgets from the QWidget class. Interactable widgets have associated signals that can be connected to slots to produce functionality. The widgets used in the GUI for this project were: [10]

QPushbutton: This widget creates a push button. The button can be pressed to emit the signal *'clicked()'*, which can then be connected to perform an action, potentially by connecting the click signal to a slot. For this project three push buttons were created to change wave type, and also to provide an application 'exit' option.

QSlider: A horizontal or vertical slider widget can be created to control a variable. This widget has several signals:

valueChanged(): This signal is emmitted when the slider value is changed.

sliderPressed(): This signal is emmitted when the user starts to drag the slider.

sliderMoved(): This signal is emmitted when the user drags ths slider.

sliderReleased(): This signal is emmitted when the user released the slider

For this project a slider was created to control volume.

QLabel: This is used for displaying text or an image. It does not provide any user interaction functionality. For this project a label was used above the slider to make the purpose of it clear, by labelling it 'volume'.

To control the layout of the widgets, the QVBoxLayout class can be used to line up widgets horizontally and vertically. Two classes that inherit from this, QVBoxLayout and QHBoxLayout can also be used to create smaller layout designs which can then be collaborated using QBoxLayout. Just to add a bit of interest, the wave push buttons were made red.

The created GUI can be seen below in Figure 8.

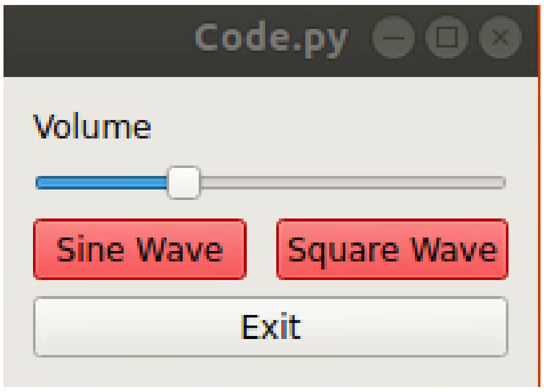


Figure 7: Graphical user interface to change volume and wave type

3.3 GUI Slots and Signals

To make the GUI work as desired, the signals emitted when the widgets were interacted with needed to be connected to PyQt slots. For this project, four slots were created.

sineClicked(): The sine wave push button's *'clicked()'* signal was connected to this slot. When pressed, a class variable 'wave' was set to zero (also the default value). This variable was later used to control which type of wave oscillation would be played, e.g. if the wave variable was equal to zero, a sine wave would be played.

squareClicked(): The square wave push button's *'clicked()'* signal was connected to this slot. When pressed, the class variable 'wave' was set instead to one, to indicate that a square wave should be played.

sliderClicked(): The slider's *'valueChanged()'* signal was connected to this slot. When the slider value was changed, a class variable 'v' would be updated to the current slider integer value. This variable was later used to control the volume of the notes played.

noteplay(): When a MIDI signal was received, this slot function was called. This is described in more detail in section 4.2.2. In this function, the values of the class variables 'wave' and 'v' were determined. Depending on the wave type selected, a sine wave or a square wave would be called from the C++ 'Oscillate' class using the frequency that had been sent in the 'newNoteFrequency' signal, and the sampling frequency value which had been saved as a class variable.

Once a wave frequency array had been produced, the volume was set a ratio calculated by ear and by multplying this against the slider value 'v'. In this way the volume could be altered. Finally, the wave array could be played using the 'sounddevice' Python module which was found to play arrays as audio files.[4]

Widgets were connected to slots using the Python code:

```
'widget name'.signal.connect('slot name')
```

4 Running the Program

In order for the program to function properly all of the individual aspects of the project needed to be brought together to run at the same time. The program needed to monitor the opened 'mido' port for MIDI signals, whilst also displaying the GUI and responding to signals produced there.

4.1 QApplication

The QApplication class is needed to run the GUI. It is another class that inherits from QObject, and it handles widget specific initialisation and finalisation. The QApplication object must be created before any other user interface objects and the must only be one QApplication object per GUI application. The QApplication object is important as it runs the event loop that waits for

user input in GUI applications, such as mouse clicks. To launch the event loop, *exec()* is called, and to end the event loop *exit* is called.
The remaining widget, the ’exit’ pushbutton, was connected to the exit signal of QApplicattion object, app, with: `self.exb.clicked.connect(app.exit).`
This meant that when pressed, the application would exit and the program would end.

4.2 Threading

Threads are used to run functions within a single process concurrently so that program processing runs faster. This can be implemented by time slicing processing or by making use of multiple cores. Generally in a program, there is a ’main thread’ and secondary threads, commonly referred to as ’worker threads’. This allows time critical main threads to remain responsive by offloading processing to worker threads.[11]

4.2.1 QThread

QThread is the Qt class that can be used to manage threads within a Qt program. In Qt the GUI program must run as the ’main thread’ and all other objects that need to interact with the GUI thread need to be made into worker threads. This can be achieved by creating a QThread object and moving the worker QObject into this thread with *’QObjectmoveToThread()’* in PyQt. The worker QObject would then run in a seperate thread from the GUI main thread, but signals and slots can be set up to allow the objects to communicate with each other. [12]

4.2.2 Connecting MIDI Signals to the GUI Thread

Therefore, whilst the main thread of the program would need to be the GUI thread as discussed earlier, a seperate ’worker’ thread was needed for the MIDI signals. This is because we want to be continuously listening out for keyboard notes, otherwise they may be missed. If the listening function was part of the main thread then the program would only run the constant loop listening for messages and the GUI would not be displayed or vice versa.

A QObject ’MidiPortReader’ was created, with a ’listener’ function that would continuously wait for messages on the opened input MIDI port. This QObject was then initialised within the main GUI thread and moved into a QThread so that it could run alongside the GUI event loop. To communicate between threads, the PyQt signal ’newNoteFrequency’ was created in the ’MidiPortReader’ QObject, emitting the frequency of every new note received, and connected to the ’noteplay’ slot in the main GUI thread to play the note.

5 Final Code

The main challenge encountered whilst creating this program was working out how to get everything to work together. Each individual section was tested individually, but full code functionality relied on setting up the signals and slots between code sections. It took a while to realise that the MIDI QThread should be stated within the QWidget initialisation function but once that was discovered everything started to work.

The final program worked as expected. Once the MIDI port had been opened and the output keyboard MIDI signals were connected to the input 'mido' port, signals were continously recieved on the 'MIDIPortReader' thread. The frequency of received notes were emitted, causing sound to play. The GUI was successful in modifying volume and wave type, and the two threads ran continously alongside each other. If there was more time to work on this project it may be worth refining the volume levels, as these can be subjective and also somewhat frequency dependant, i.e. humans hear different frequencies at different volumes. Nonetheless, the final working code was satisfying to play around with. An example video of it working in included in the ZIP file of code.

There are a lot of future additions that could be made to this code, such as more waveform options, or maybe even an attempt to simulate different instruments. If we were to take it further, it may be fun to try and create a whole MIDI syntheiser from scratch, e.g. the virtual keyboard or other instrument.

The full code written for this project is included overleaf.

FULL CODE

```python
from PyQt5.QtCore import Qt, pyqtSlot, QThread, pyqtSignal, QObject
from PyQt5.QtWidgets import QApplication, QWidget, \
    QSlider, QPushButton, QLabel, QVBoxLayout, QHBoxLayout
import sys
import mido
import sounddevice as sd
import oscillator as o

#Create QObject to listen for MIDI messages
class MidiPortReader(QObject):
    #Create signal for when a MIDI note_on message arrives
    newNoteFrequency = pyqtSignal(float)
    #Initialise Object
    def __init__(self):
        QObject.__init__(self)
        #Open MIDI input port to recieve MIDI messages
        self.port = mido.open_input('pipes',virtual=True)

    #Create function to recieve MIDI messages, calculate note
    #Frequency, and to emit the new frequency as a PyQt5 signal
    def listener(self):
        for msg in self.port:
            m = str(msg)
            #Select the relevant parts of the recieved MIDI message
            #Here this is the note number
            a = m[23]
            b = m[24]
            #Select the part of the MIDI message to distinguish if
            #this is a 'note_on' or 'note_off'
            z = m[6]
            #If the message is 'note_on', emit the note frequency
            if (z=="n"):
                note = int(a+b)
                f = 27.5*2**((note-21)/12)
                self.newNoteFrequency.emit(f)



#Create QWidget to produce GUI (main thread)
class Control(QWidget):

    #Set-up signals and variables
    fs = 44100
    v = 0
    wave = 0
    s = o.Oscillate()

    #Initialise object
    def __init__(self, parent = None):
        super().__init__(parent)

        #Create worker(MidiPortReader) thread inside QWidget
        self.obj = MidiPortReader()
        self.thread = QThread()
        #Connect worker's signal to GUI slot
        self.obj.newNoteFrequency.connect(self.noteplay)
        #Move the worker object to the thread object
        self.obj.moveToThread(self.thread)
        #Start thread
        self.thread.start()
        #Start GUI
        self.create_UI()

    #Create GUI
    def create_UI(self):
        #Create slider/buttons
        self.Slider1 = QSlider(Qt.Horizontal)
        self.label = QLabel("Volume")
        self.sib = QPushButton(self.tr('Sine Wave'))
        self.sqb = QPushButton(self.tr('Square Wave'))
        self.exb = QPushButton(self.tr('Exit'))

        #Set button colours
        self.sib.setStyleSheet("background-color: red")
        self.sqb.setStyleSheet("background-color: red")

        #Build horizontal layout
        hLayout = QHBoxLayout()
        hLayout.addWidget(self.sib)
        hLayout.addStretch(1)
        hLayout.addWidget(self.sqb)

        #Build vertical layout
        vLayout = QVBoxLayout(self)
        vLayout.addWidget(self.label)
        vLayout.addWidget(self.Slider1)
        vLayout.addLayout(hLayout)
        vLayout.addWidget(self.exb)

        #Connect buttons to slots
        self.Slider1.valueChanged[int].connect(self.sliderClicked)
        self.showb.clicked.connect(self.sliderClicked)
        #If exit button pressed, exit application
        self.exb.clicked.connect(app.exit)
        self.sib.clicked.connect(self.sineClicked)
        self.sqb.clicked.connect(self.squareClicked)
        self.obj.newNoteFrequency.connect(self.noteplay)

    #Create slots

    #If sine wave is selected, set 'wave' variable to '0'
    @pyqtSlot()
    def sineClicked(self):
        self.wave = 0

    #If square wave is selected, set 'wave' variable to '1'
    @pyqtSlot()
    def squareClicked(self):
        self.wave = 1

    #If slider is moved, set 'v' variable to new slider value
    @pyqtSlot()
    def sliderClicked(self):
        self.v = self.Slider1.value()
       #Do not accept slider value of zero as volume of zero
       #means zero output
       if (self.v==0):
            self.v = 1

    #Create function to play note using C++ oscillator class
    @pyqtSlot(float)
    #Slot receieves frequency value from listener thread
    def noteplay(self,f):
       #print("frequency:",f)
       #print("volume:",self.v)
       #print("wave:",self.wave)
       #If 'wave' variable is set to 0, use sine wave oscillator
       if (self.wave==0):
           #print("sine")
            b = self.s.wavesine(self.fs,f)
            #adjust note volume based on 'v' variable value
            note = (1/1000000*f)*b*self.v
            #play array to generate sound
            sd.play(note, self.fs)
       #If 'wave' variable is set to 1, use square wave oscillator
        if (self.wave==1):
           #print("square")
            b = self.s.wavesquare(self.fs,f,0.7)
            #adjust note volume based on 'v' variable value
            note = 0.001*b*self.v
            #play array to generate sound
            sd.play(note, self.fs)



app = QApplication(sys.argv)
window = Control()
window.show()
sys.exit(app.exec_())
```

References

[1] Michael Hahn. (2020). *What Is MIDI? How To Use the Most Powerful Tool in Music*. LANDR Blog. Available online: https://blog.landr.com/what-is-midi/. (Accessed May 2020)

[2] Ole Martin Bjørndalen (2018). *Mido - MIDI Objects for Python*. Mido Documentation. Available online: https://mido.readthedocs.io/en/latest/index.html

[3] William Zeitler. (2009). *Frequency / Midi Note Table*. The Glass Armonica. Available online: https://glassarmonica.com/science/frequency_midi.php

[4] Matthias Geier (2020). *Usage*. Python Sound-Device Documentation. Available online: https://python-sounddev
ice.readthedocs.io/en/0.3.15/usage.html#playback

[5] (2020). *Qt for Beginners*. Qt Wiki. Available online: https://wiki.qt.io/Qt_for_Beginners

[6] The Qt Company (2018). *QObject*. Qt Documentation. Available online: https://doc.qt.io/qt-5/qobject.html#details

[7] (2020). *PyQt5 signals and slots*. Pythonspot. Available online: https://pythonspot.com/pyqt5-signals-and-slots/

[8] The Qt Company (2018). *Signals and Slots*. Qt Documentation. Available online: https://doc.qt.io/qt-5/signalsandslots.html

[9] Riverbank Computing Limited (2020). *Support for Signals and Slots*. PyQt v5.14.1 Reference Guide. Available online: https://www.riverbankcomputing.com/static/Docs/PyQt5/signa
ls_slots.html#the-pyqt-pyobject-signal-argument-type

[10] The Qt Company (2018). *QWidget*. Qt Documentation. Available online: https://doc.qt.io/qt-5/qwidget.html

[11] The Qt Company (2019). *Threading Basics*. Qt Documentation Archives. Available online: https://doc.qt.io/archives/qt-4.8/thread-basics.html

[12] The Qt Company (2018). *QThread*. Qt Documentation. Available online: https://doc.qt.io/qtforpython-5.12/PySide2/QtCore
/QThread.html#more