

Convert Spike Hound from Matlab to Python

A Design Project Report Presented to the School of Electrical and Computer
Engineering of Cornell University

in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering, Electrical and Computer Engineering

Submitted by

1. Chen, Ziqi (zc444)
2. Gao, Wangzekun (wg265)
3. Madhavaram, Kranthi Kumar (km853)

MEng Field Advisor: Bruce Land

Degree Date: May 2020

Abstract

Master of Engineering Program

School of Electrical and Computer Engineering

Cornell University

Design Project Report

Project Title: Convert Spike Hound from Matlab to Python

Authors: Chen Ziqi, Gao Wangzekun and Madhavaram Kranthi Kumar

Abstract: Spike Hound is a data acquisition, stimulus generation, instrument control and real time analysis package implemented in Matlab and designed for use in physiology research and education. The software allows for connection to a broad range of popular data acquisition hardware from high end, calibrated data acquisition boards to the sound card in the personal computer. This project is going to convert this comprehensive signal-processing package into a Python version. The first task is to complete a layout that is able to receive and print signals like oscilloscopes. The next task will work on supportive code, drivers and toolkits from different device manufacturers, making the interface compatible to a wide range of devices. The final task is going to write functions to implement signal visualization, real-time filter applications, audio output to a monitor, and data logging with associated meta-data, etc. Python is a language with strong generality which supports almost all kinds of platforms. This project will be able to run on different terminals including Mac, Windows and by extension can be used in microcontrollers like Raspberry Pi and Arduino. The major goal is to publish the code to enable low cost, electrophysiological data acquisition and analysis to students and researchers at institutions with fewer resources.

Executive Summary

First, we find a liable and robust library (tkinter) as a GUI for oscilloscope display on screen, then we work on the part of data reading in python code, which includes the interface from the NI data acquisition device. Second, we added a configurable trigger level for stabilized input signal waveforms. Finally, we use the build-in method of the interface from the NI data acquisition device to control and output a configurable digital signal waveform.

We anticipated to add more functions to this oscilloscope such as basic fourier transform and operations between two different input signals, however, because of the epidemic, we can only finish the part of displaying and sampling input analog signal with a precision of 0.1-1,000ms for time scale and 10-5,000mV for voltage scale. As for the output we can generate any digital signal that has the minimum precision of around 10ms, and this may vary from different platforms or PCs.

Finally, we use switch buttons to control the input and output by setting up three modes in this oscilloscope. First mode is for input data acquisition and second one is used in output data, then the third one is in favor of input and output simultaneously. The reason for these three modes is to save more energy firstly, and to gain more precision, because simultaneously output and input data will require multi-threading programming, which is very likely to invoke uncertain execution time. And uncertain execution time will influence the delay in our output. Therefore, using the third mode is not recommended until further update is achieved.

The original plan for this project is to convert the total Spike Hound software into a python version to save the authorization fee from Matlab, however, through the outbreak of this epidemic, we can no longer have access to the laboratory. And the project is ceased here, but we have managed to finish up to 80% of this project and the input and output of this python version software is thoroughly done. And further update would be adding more rudimental functions such as fourier transform for signal processing and data or waveform saving.

Index

Abstract	2
Executive Summary	3
Introduction	6
Methodology	8
Individual contribution	10
Design	10
Initial setup	10
NI converters interfacing	11
Trigger Input	13
Digital outputs	13
Feedback - Multithreading	14
Control buttons	14
Limitations	15
GUI Attempt on Mac OS	15
Code Errors	16
Pandemic	17
Results	18
Future scope	24
Trigger	24
Functions implementation	24
User Interface Optimize	24
Mac OS	25
Multi Platform Integration	25
Conclusion	27
References	28
Appendix	29
Code:	29
User Manual	46
Install package:	46
Interface:	46

Introduction

Spike Hound (formerly “g-PRIME”) is a data acquisition and real time analysis package designed for use in physiology labs where continuous multi-channel data is acquired and discrete events are analyzed. The software allows for connection to a broad range of popular data acquisition hardware from high end, calibrated data acquisition boards to the sound card in the personal computer. Spike Hound allows for signal visualization, real- time filter applications, audio output, and data logging with meta- data. Spike Hound replaces and exceeds the functionality of several classical bench-top tools including the oscilloscope and arbitrary function generator considering the specificity of physiology lab requirements. Spike Hound offers many powerful features for scientific visualization with a user friendly interface developed in a feedback process with neurophysiology researchers and students at the university level.

This software, however, has been written in MATLAB by Dr. Gus K Lott III in 2010 and so, unfortunately needs MATLAB licensing for any further development or customization. This limits the usage of the software by students studying at budget constrained universities and educational institutions. This brings the undertaking of this project, requested by the client Prof. Bruce Johnson from the Neurobiology department of Cornell University, to port the package to Python, an OSI approved open source programming language. This will help the package reach out to a larger target audience and ease the data acquisition and analysis in physiology labs across institutions with limited resources. More particularly, the software is expected to be used by Prof. Bruce Johnson for the courses he takes at Cornell University and hence demands to be made “foolproof” to avoid unexpected errors and crashes as much as possible.

Elementally, the package is set at zero cost with the only other requirements being the ADC/DAC converters. The aim is making the package work with inputs from both USB and Line-in available on a typical laptop/computer. This can be made possible by making use of the sound card on the computer for high speed data acquisition (which is typically around 44000 Hz). The multi input option increases the adaptability of using almost any ADC/DAC converters available on the market. Leaving the choice to the user to pick the converter to their liking and their requirement.

The primary job of the software is data acquisition and processing very much similar to an Oscilloscope. As such the goals of the project were set and followed in the following chronological order. Setting up of a skeletal code for data acquisition from the sound card onto a buffer, plotting of the data live on screen, increasing the number of inputs, testing with the ADC/DAC converters, introducing trigger channels to the data acquisition both external and internal, Digital output and adding features such as data logging, math functions, fourier transformations etc as time permits.

The demand of producing a fully functioning package is the predominant challenge of the project. However, as we go about the project there are several things to be noted and expected. The synchronous tasks of data acquisition and data plotting will pose as the first challenge to the project. Hereon, adding different processes and modes to the package that will be run concurrently will add to the complexity of the project. The choices of data structures for data buffers and such will have a huge interest in increasing the efficiency and decreasing the complexity of the working. Further, compatibility to different inputs demands for a versatile algorithm from the beginning. In brief, we expect the project to pose challenges on an exponentially increasing level as we move ahead at every stage.

Methodology

As discussed, the project has various stages as we progress. However, the project scope can be divided into three stages. First, would be deciding on the libraries and having a skeletal code running that can input from the audio jack of a system and display the signal. Second, would be adding the supportive code that can make the code work with different drivers and toolkits of major brands producing ADC/DAC converters. The final step would be adding functionalities to the package for filtering, data logging, audio output, etc. At every step, however, each team member will be working individually on different platforms of Windows and Mac OS to ensure the package's compatibility.

As discussed, the elemental requirement of the project is speed and platform compatibility. Picking the GUI would be essential considering Python offers GUIs over a broad range on the basis of aesthetics and speed. For these requirements, a simple GUI capable of working at a speed of 20 kHz is needed. The constant erasing and redrawing of the signal realisation on the screen will have to be matched at the rate of data acquisition from the hardware. The libraries are also expected to have simple functions for data acquisition in familiar data structures and to be able to distinguish between the left and right audio channels available on the sound card. Having the libraries figured out would then lead to building the skeletal code for acquiring a signal and displaying it on screen, similar to that of an oscilloscope.

The skeleton code for GUI we used here is from Mr. Onno Hoekstra's work on an audio input oscilloscope built with python. It uses tkinter as the GUI library and can change time and voltage division while running it. Original code divides the window we have into 10 blocks on each X and Y axis, and each block can have one hundred points at maximum to draw at each time we update the plot. The minimum time division we have is 0.1mS and maximum is 1S, while the voltage has a minimum of 10mV and maximum of 5V. The window has four main buttons to decrease and increase the time and amplitude division, and two buttons to run and stop the oscilloscope. The mechanism behind this is that we read the audio input from the buffer as a list with a fixed size of 1024 and convert the values to points on screen with respect to the time and amplitude division we choose at the moment. Obviously, the smaller time division we choose the less number of points we will draw on screen resulting in faster processing speed and closer to real-time input. Additionally, there is no trigger in this oscilloscope which displays much jittering and fluctuation.

Our main revision of this skeleton is to add a configurable trigger level, a second optional input as a hardware interface to read from the NI device, and an output digital signal using the

NI device. Hence, we can create another function similar to the original input processing and a decision making process before sending the data to plotting and such progress is based on whether the current point we have is passing through the trigger level.

Once the code is up and running, the next task would be working on having Serial interface and Line-in as inputs apart from the existing audio-in. With the client's requirements pointing towards the usage of National Instruments' ADC/DAC converters being prominent in the field, most programming missions will be working towards interfacing the package with the drivers and toolkit of the same. However, the inputs should not be limited by the external drivers. Initial research tells that different converters of the said company have different offerings in terms of sampling speed and such. Therefore, there will be a simple drop down menus to match the specifications of the hardware with the software and also possibly, an option for automatic detection of the hardware. Unfortunately, the latest version of NI interface software only supports the version of macOS at 10.10.v which is a bit obsolete at this time, and this is why we abandon the chance to develop macOS supported versions of this software.

Having a fully functional skeletal code that takes multiple inputs and displays the signal on screen as an oscilloscope would now need additional functionalities that make the data analysis easier. With the data structures known, python offers function insertions in the code that can be called on demand. Having multiple functions, including but not limited to data logging, real time filtering, trigger mode settings etc should be fairly simple steps and would have huge value addition to the package. Further functionalities can be added at the request of the client.

It is also important to note that a significant offering of the package will be platform independent. As individuals each team member will be working on individual platforms of Windows and MAC and as a team, there will be collaborating to keep all members on the same page and ensure there is no deviation from the outcomes on any of the platforms. Furthermore, the end of the project will only be marked when members come up with a carefully detailed user manual for the package. This is to ensure ease of understanding for those with limited knowledge of computers and software, and also to bring to pace the enthusiastic developers who could further improve and better the package with change of time.

Individual contribution

Ziqi Chen: coding and debugging digital output module, drafting result, summary, conclusion and methodology

Wangzekun Gao: model design, coding and debugging input module, setting up switch buttons for different modes, drafting the user manual and feed back in final report

Kranthi Kumar Madhavaram: coding and debugging digital output module, drafting limitation and design in final report

Design

Initial setup

Spikehound, although, has already been written and readily available in MATLAB language, it isn't so simple to translate it into Python. In general, it is tough to translate a code from language to another especially if the data structures and the libraries between them vary with huge differences. Hence, we ventured out to start the programming from scratch. With the minimum expertise shared between the three of us in Python, we tried to solve the problem one step at a time, starting with finding the best fit for a GUI library needed to run the software at the speed we desire.

Python Wiki, a website that contains the know-how of everything in the Python world gave us brief descriptions and differences of the available python GUI libraries. Further research into it and with help from the advisor and Prof. Joseph Skovira from the Electrical Dept. at Cornell, we have narrowed down to 'Tkinter', the standard Python interface to the Tk GUI toolkit from Scriptics. Both Tk and Tkinter are available on most Unix platforms, including Windows and Macintosh systems which satisfies the requirements for this project without much trouble. Further, the amount of resources available for Tkinter and the modules built on top of it are immense making it an easy choice for our development.

Building on top of the GUI library we have found we have started to learn to use it in making widgets and simple push buttons on screen until we came across a website by Onno. A hobbyist software developer, Onno, has done various projects on audio analysis in Python, including a basic framework for an Oscilloscope that uses Sound card and produces a real time graph of the data. This open source software has since then formed the basic skeletal framework for our project.

The software uses the Tkinter and pyaudio library for the major part where the soundcard is accessed by the pyaudio library and the data stored is presented on the screen using the Tkinter library. Although the software was very carefully written there have been some minor errors like wrong function calls in getting the code running. We then proceeded to verify those errors in our system to have a structural code running. Further to this we then proceeded

to make modifications on the code to have it running at our specifications, more particularly, setting the sample rate to 20 KHz, changing the screen size and ratios for a smoother interface, addition of drop menu to select between the audio inputs available etc. At this stage, we had a skeletal framework which could access data from the sound card via the built-in microphone and plot the data on screen at the rate of data acquisition. Once we had the software running we continued to work on increasing the inputs available.

NI converters interfacing

The next goal of our project is to make our code support as many inputs as possible. As mentioned in the former part, pyaudio library is a tool to control the devices, get the input waveform, and control the possible audio-related hardware to do the operation we want. We used pyaudio functions to get the available input devices and made a menu for the devices. Once we select a device, we open the data stream and then get then read in the input in mV. The first input we are able to support is the input from the sound card which reads from the microphone. The result is shown in Fig 1.

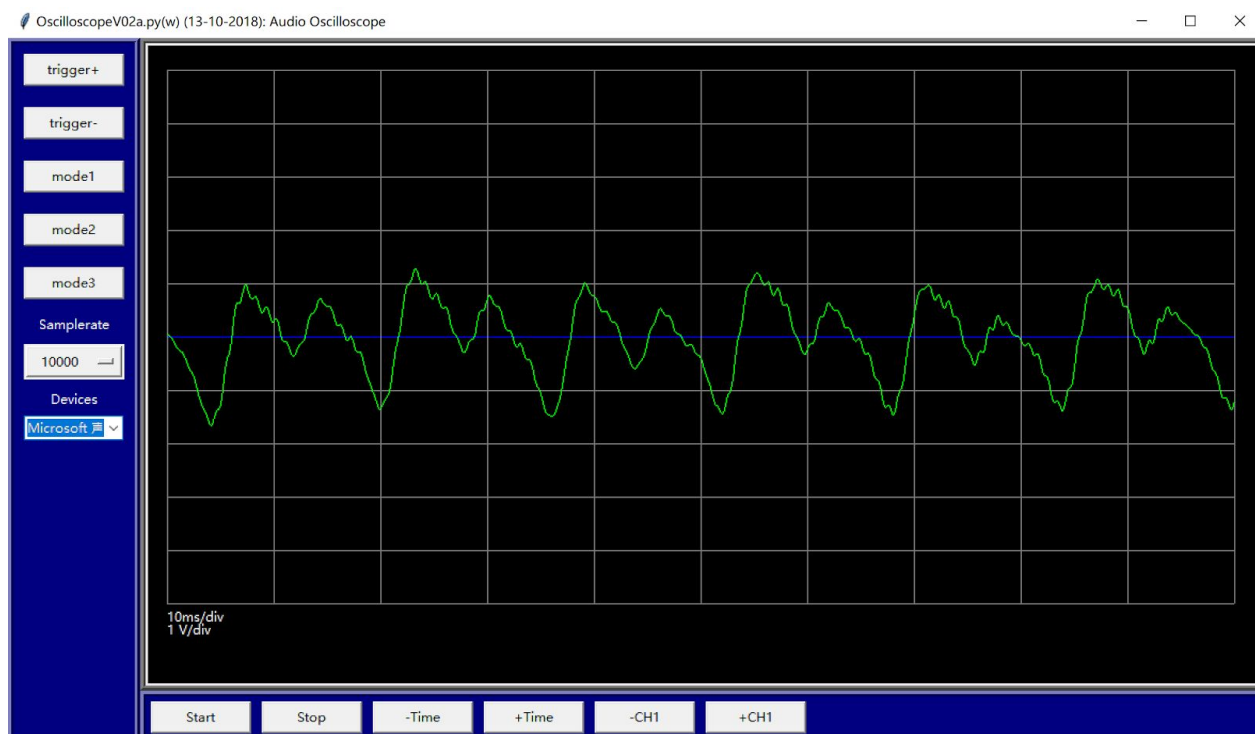


Fig 1. Input from Sound Card

The next step we worked on is to allow our code to support NI DAQ devices. Data acquisition (DAQ) is the process of measuring electrical or physical phenomena such as

voltage, current with a computer. The DAQ devices are able to take in raw analog data from sensors and send it through a bus (maybe USB, PCI or even wireless connection). USB is the most widespread bus, so we worked on allowing a NI DAQ device to be able to transmit data to a computer through USB and then, the code gets the data and plots the waveform. The working steps are similar to pyaudio. Select the name of the device channel, create the observation task and read in the data. We tested our work on a NI USB-6008. Note that here we drive the NI device by importing 'nidaqmx' library (because the pyaudio cannot drive and operate on peripheral devices, we need to import supporting libraries).

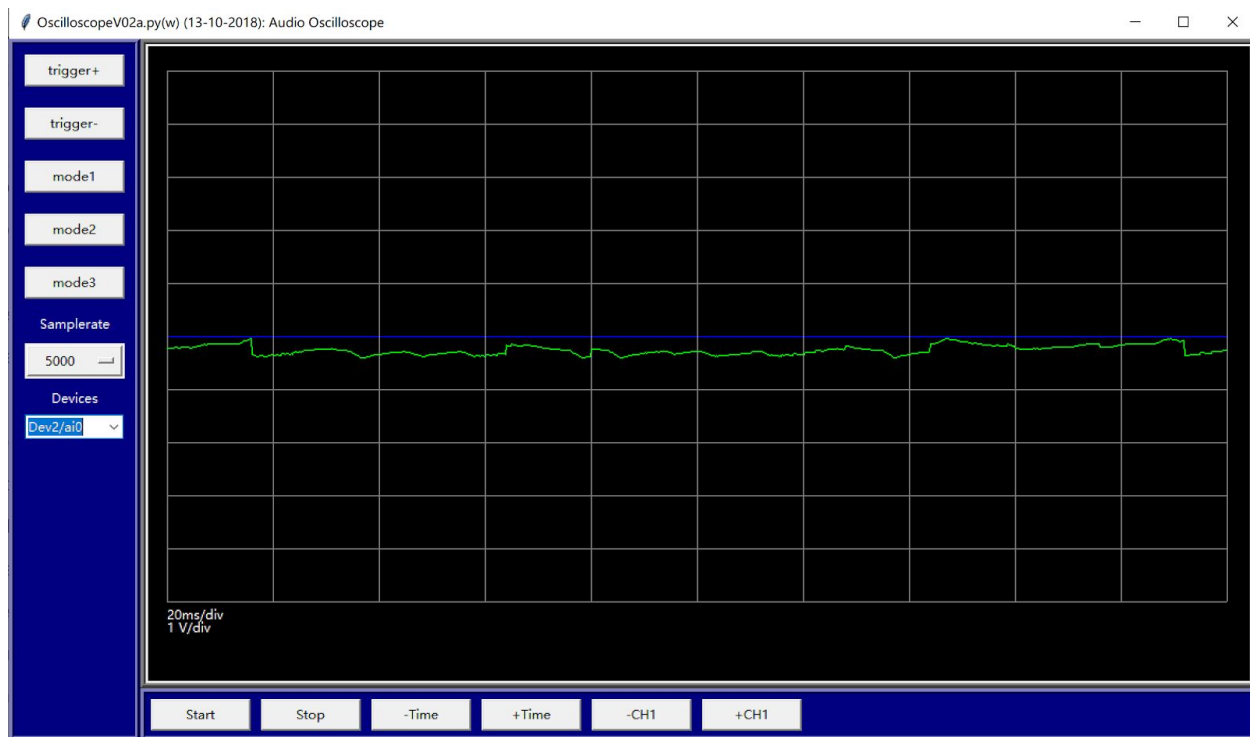


Fig 2. Input from NI USB-6008

Regular computer OS does not have the drivers for NI devices and the drivers cannot be installed automatically. To make our computer able to communicate with DAQ devices, we need to install NI Measurement and Automation Explorer(MAX). NIMAX provides access to a variety of NI devices. It includes all National Instrument drivers (NI-VISA, NI-DAQmx etc.) and the Ni System Configuration Full package (including the nidaqmx library needed for coding in python). This interface allows us to configure NI hardware and software, create and edit channels, tasks, interfaces.

Trigger Input

At this stage we have a software that samples data from both a microphone and a USB device at a rate of 20 KHz and concurrently plots the signal on screen. However, at this stage the screen refresh rate isn't constant or more particularly we don't have a stable signal (for a periodic wave). This is primarily because the data read is stored in a buffer limited to 1024 points, and the number of points shown at a given instant depended on the time division on screen which varied between 100-1000 points. However, the screen is refreshed at a constant rate and the values in the buffer were shown as it is without any trigger.

To tackle this we have introduced a trigger level to the input signal. Initially, we just tested with the ground level as the trigger. Here, we constantly read the input into a buffer of 1024 points at the rate of screen refresh and before the screen print we look for the index of the buffer where the data changes sign from negative to positive. This point essentially marks the signal crossing zero level. We then plot the data from this point index to the number of points the screen can show depending on the time scaling selected.

Further, once the zero trigger level satisfactorily worked, we introduced buttons on screen to increase and decrease the trigger level to the user's liking. This new trigger level essentially replaces the zero in the code i.e., we look for the data that crosses the new level selected from the buffer and print accordingly. However, at the start of the program the default trigger level is set to zero. Further additions to trigger level could be having an external trigger

Digital outputs

So far we have been discussing the input capabilities of the software which was essentially to acquire data and plot with a trigger. In this section we mention about adding output capabilities to the software. As per the client's requests we were asked to generate digital pulses through the NI device at a time period of upto 1 ms or less (considering the subjects he works with generates around 1-5 kHz signals). This output was necessary to create self timed triggers for data acquisition.

For this we particularly used the *task.write()* function of the 'nidaqmx' library. The specifications of the USB6001 instrument limits the Digital i/o pins to function only using a software timed signal for all sampling and reading data. However, the analog i/o pins have access to both software timed and hardware timed signals. Python, on the other hand, can only generate time signals of time period upto 200 microseconds without much possible errors.

Going below 200 means the rise time and fall time of the pulse is comparable to 200 microseconds and produces errors.

Bearing all the above conditions and requirements in mind we have decided to use digital output pins for generating the pulse train. We could generate a pulse signal with a time period of 400 microseconds and 50% duty cycle ratio without having any significant influence of the rise and fall time on the time period. This, however, doesn't meet the requirements of having a time pulse of 0.1 ms. However, considering the pulse output is used for triggering, having a single pulse of 200 microseconds wide at a time period of 1 second is a good enough signal. With this compromise, we could generate a 3 pulse train, 200 microseconds wide, at every 1 second time period.

Feedback - Multithreading

After having both the input and output functionalities of the software working, the next step was to run them simultaneously. This was required primarily so that the digital output generated can be used as a trigger channel to the input data acquisition.

Python itself has thread class. Therefore, we can easily create a thread and base on which mode the user chooses to generate output signal or not. We just need to specify the callback function for the thread to run. As stated above, we could generate a 3 pulse train, 200 microseconds wide, at every 1 second time period. However, in the end we could only simultaneously generate a signal and read it in the input before COVID-19 pandemic required us to be restrictively working to finish the project. Right now we don't have the control buttons to control the parameters like pulse width, the period, the pulse number of the signals.

And since we are using multithread, the performance of the output thread and input thread will affect each other. If we are reading a large time scale in the input thread, we will take longer time to sweep for the signal and therefore affect the performance of the input thread. Due to this, our whole feedback process can only work on a small time scale.

Control buttons

Although the critical parts of the project have been discussed so far, the program doesn't provide an "auto trigger" mode (as in an oscilloscope) yet, that could adjust the scaling to give the best possible view of the signal. Our project, however, offers manual controls to the users. Buttons for scaling the time division and the signal magnitude are offered (two each for increasing and decreasing) along with two more buttons for starting and stopping a currently reading signal. These buttons are regularly scanned throughout the algorithm to change the variables of time division and such accordingly. At the same time, there are drop down menus available to select the device you wish to use and also vary the sampling rate (default: 10000)

Limitations

GUI Attempt on Mac OS

We first tried tkinter, which is a commonly used library in python gui programming because of its brevity, maturity, extensibility and cross platform. Note that we finally need to make this oscilloscope to work on all platforms, which means we need to find a robust library that can make this program dispensable between different systems. The ultimate purpose of this software is for education, and that is when extensibility comes into consideration. The project needs this program to be easy to maintain so as to modify into future versions if needed.

Pyqt5 is the first library we used. However, after testing this library with different window size and style, we found that the statement in python code for pyqt5 of macOS system is quite different from windows system. The display of menu bar in macOS needs to be explicitly stated before we modify any features of it. For instance, we need to add this statement before every menu bar we create: `setNativeMenuBar(False)`. `NativeMenuBar` property specifies whether or not the menu bar should be used as a native menu bar on platforms that support it. If this property is true, the menu bar is used in the native menu bar and is not in the window of its parent, if false the menu bar remains in the window. And for macOS such property is configured as true, which means it will not be seen in the window of its parent, thus, it cannot be seen. We need to modify each part of the code for windows that changes the feature of menu bar to make sure we have a consistent GUI across the platforms. This is a naïve and hacky way.

Although we can release two different versions for this software: one for macOS and the other for windows, we want this problem to be simplified as to be more friendly to students who do not know much about computer programming. To save the trouble over this trifle we change the library of gui from pyqt5 to tkinter.

Python programs using Tkinter can be very brief, partly because of the power of Python, but also due to Tk. In particular, reasonable default values are defined for many options used in creating a widget, and packing it (i.e., placing and displaying). Tk provides widgets on Windows, Macs, and most Unix implementations with very little platform-specific dependence. Some newer GUI frameworks are achieving a degree of platform independence, but it will be some time before they match Tk's in this respect. Also, tkinter is first released in 1990, the core is well developed and stable. Many extensions of Tk exist, and more are being frequently distributed on the Web. Any extension is immediately accessible from Tkinter, if not through an extension to Tkinter, then at least through Tkinter's access to the Tcl language. There is some concern

with the speed of Tkinter because Dr. Johnson needs this oscilloscope to be real-time displaying of signal. Most calls to Tkinter are formatted as a Tcl command (a string) and interpreted by Tcl from where the actual Tk calls are made. This theoretical slowdown caused by the successive execution of two interpreted languages is rarely seen in practice. Fortunately, by some real-time experiments we found that most of the time of oscilloscope is spent on calling plotting functions, which means we can speed up processing by reducing the number of points each time we update the screen. 500 points for each update according to our experiments should be more than enough.

Code Errors

We use the code of an oscilloscope as a frame and modify it with a configurable drop-down bar that can choose different inputs we want and a variable time scope and amplification scope. However, we did not find the bug for the macOS system when we ran the exact code for audio input from a jack or microphone. So far, we believe the problem is the buffer value, when the audio input is triggered, the signal value will be stored in audio buffer temporally. And we set a sample rate such as 44100 Hz to sample the data from the buffer and plot the signal. But here comes an issue, if we sample the buffer too slow, the buffer will be flushed by the newly inputted data since the execution time of data input for the buffer is much faster than the sampling procedure because the program needs to plot on the screen. A certain amount of data loss is allowed, however, if the sampling rate is much slower than the data input rate, there will be a huge amount of data loss. And we cannot let this happen, which means we need to choose the sampling rate wisely according to different hardware input. On the other hand, we can change the buffer size in order to store more data at the same period of time.

The exact error we encountered is `oserror: [errno -9981] input overflowed`, which occurs whenever we call method `read.stream()`. The error is inflicted because the data input of `read.stream()` is overflowed. We did some online research about this problem and applied two viable solutions. The first one is to change the buffer size, which we have mentioned above, and it turned out to be futile. And the second one is to change the inner implementation of the `pyaudio` library. Since we do not need to plot all the data that we have captured in the buffer, it is trivial harm for us to lose some data because of the speed lapse. Therefore, we set the variable that indicates the overflow status to be true, in other words, we overlook the overflow of buffer intentionally. Unfortunately, this does not fix the problem either.

Pandemic

This project was taken up for the period between 2019-2020 and unfortunately, the world was hit by a pandemic caused by Coronavirus around the month of March and for our own personal safety and that of every other individual we know and don't know whom we could've been in contact, we had to quarantine ourselves at home. With limited access to instruments and lab, we had to end our project abruptly in the middle. However, we had achieved the majority of the goals that we had set for ourselves at the beginning of the project.

Results

As we have mentioned before, due to the epidemic, we cannot fulfill all the requirements of our client, yet we have achieved about 80% of the work we anticipated in the beginning, and here are the results.

Besides the audio input from the sound card we have mentioned, we also have managed to correctly input analog signal with configurable trigger level, time and amplitude scale. This is depicted in the figures below. We listed three pairs of variable input signal waveforms, as we can see the triangle waves, sin waves, and square waves are displayed correctly on screen with tolerable jitters and deviations compared to the hardware oscilloscope.

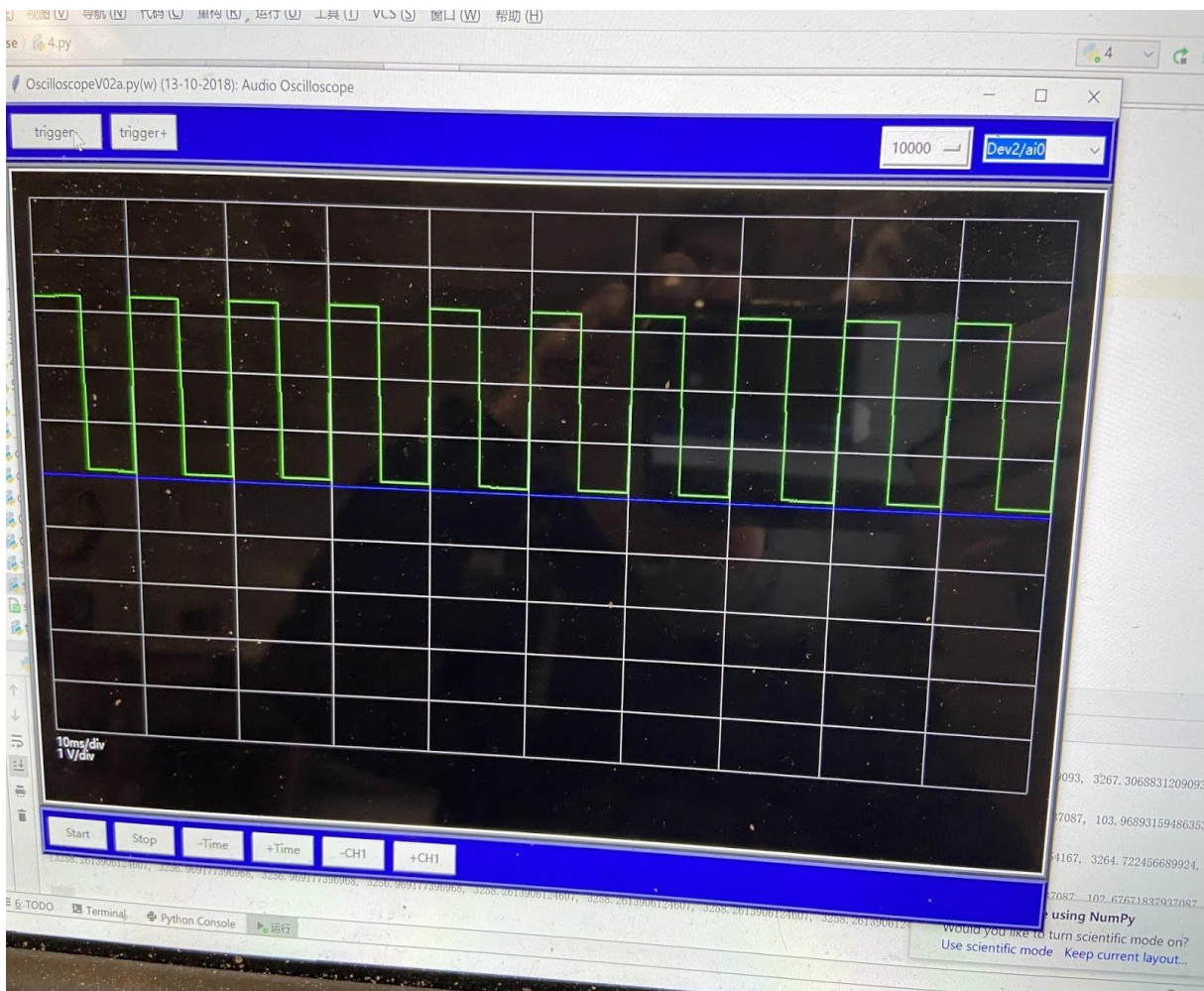


Fig 3 (a) square waves input on python oscilloscope

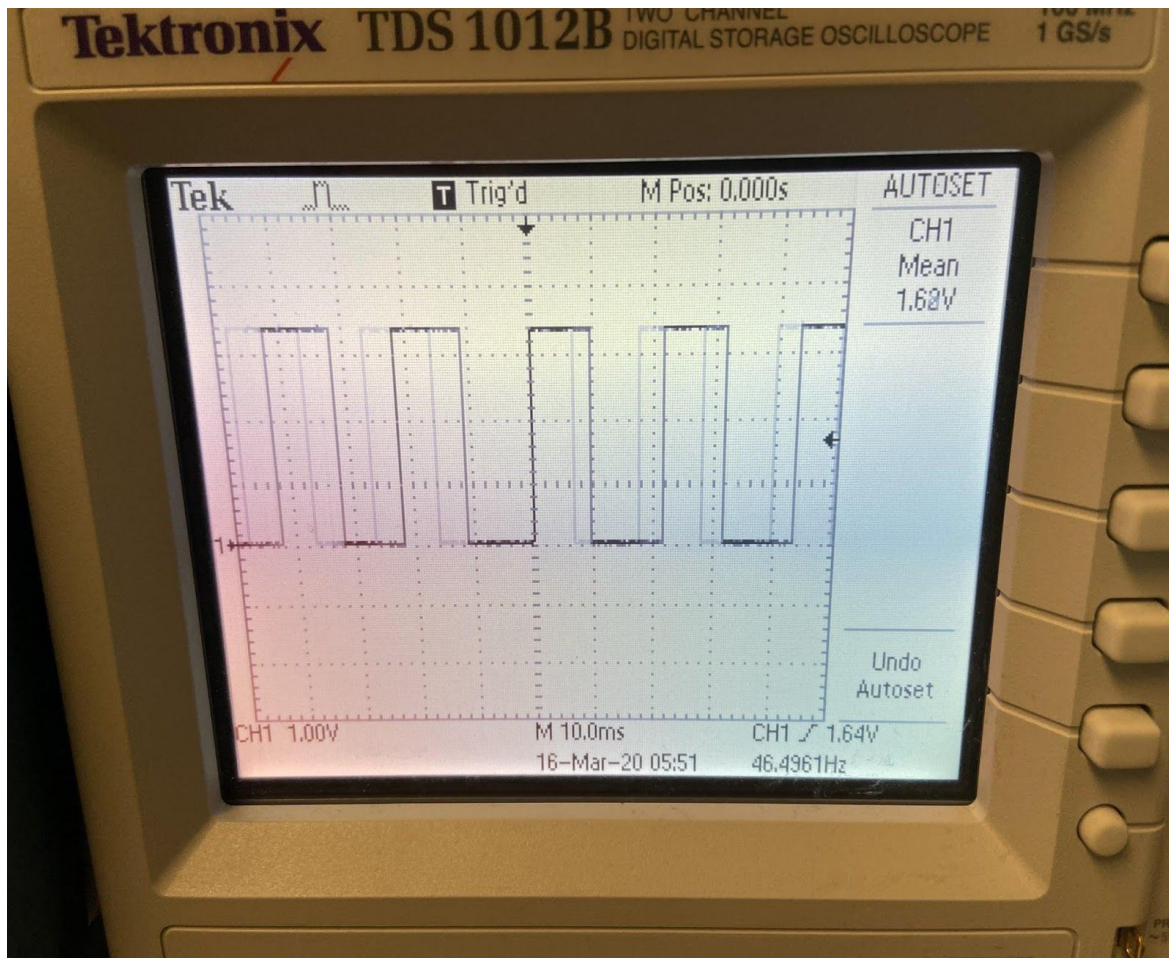


Fig 3 (b) square waves input on oscilloscope

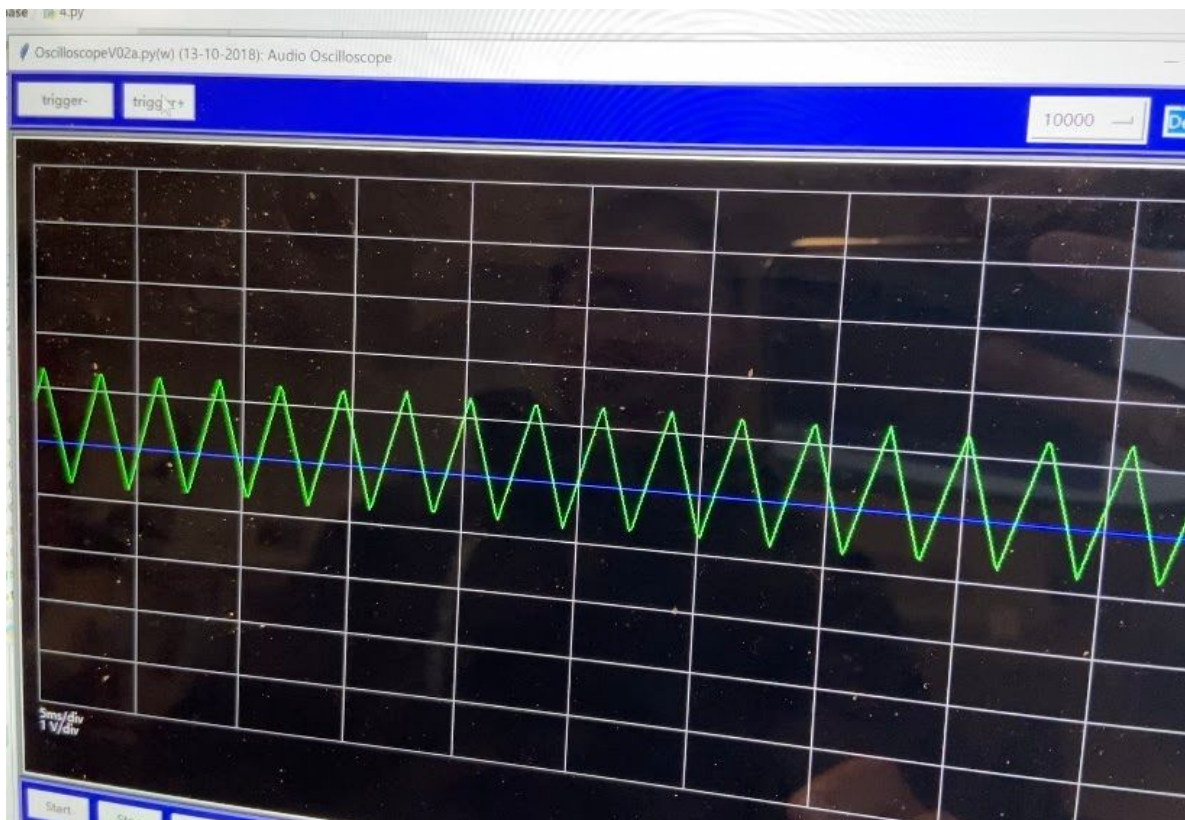


Fig 3 (a) triangle waves input on python oscilloscope

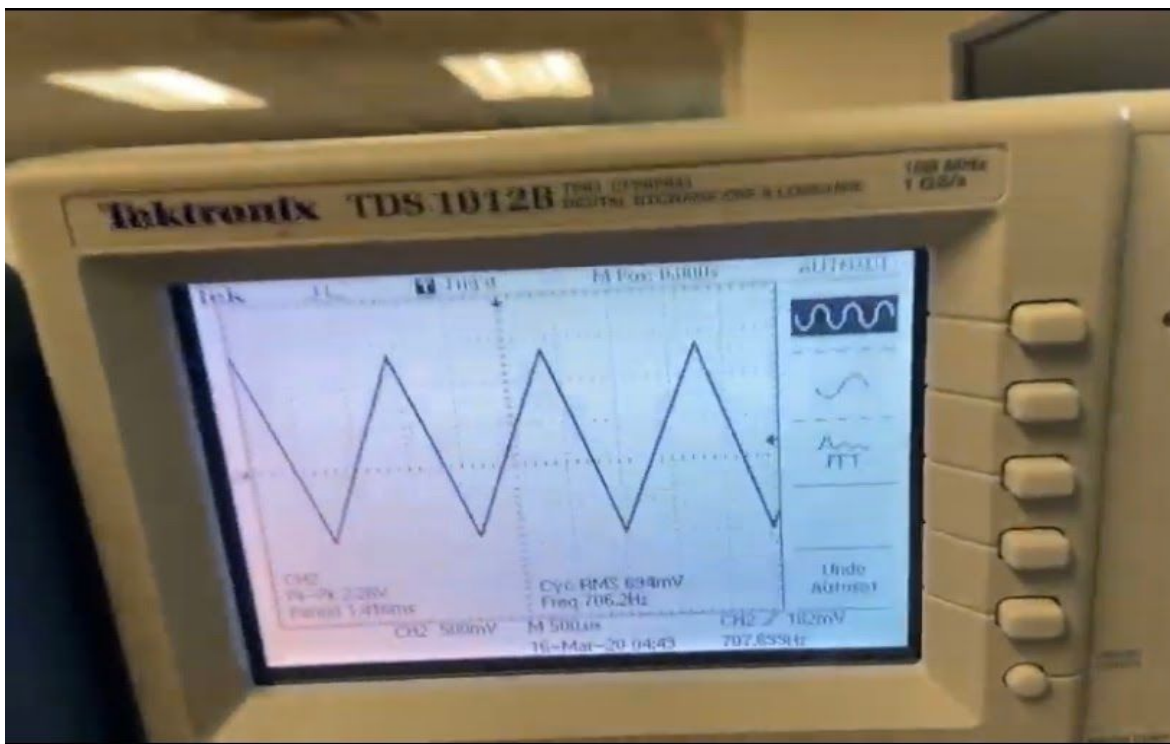


Fig 4 (b) triangle waves input on oscilloscope

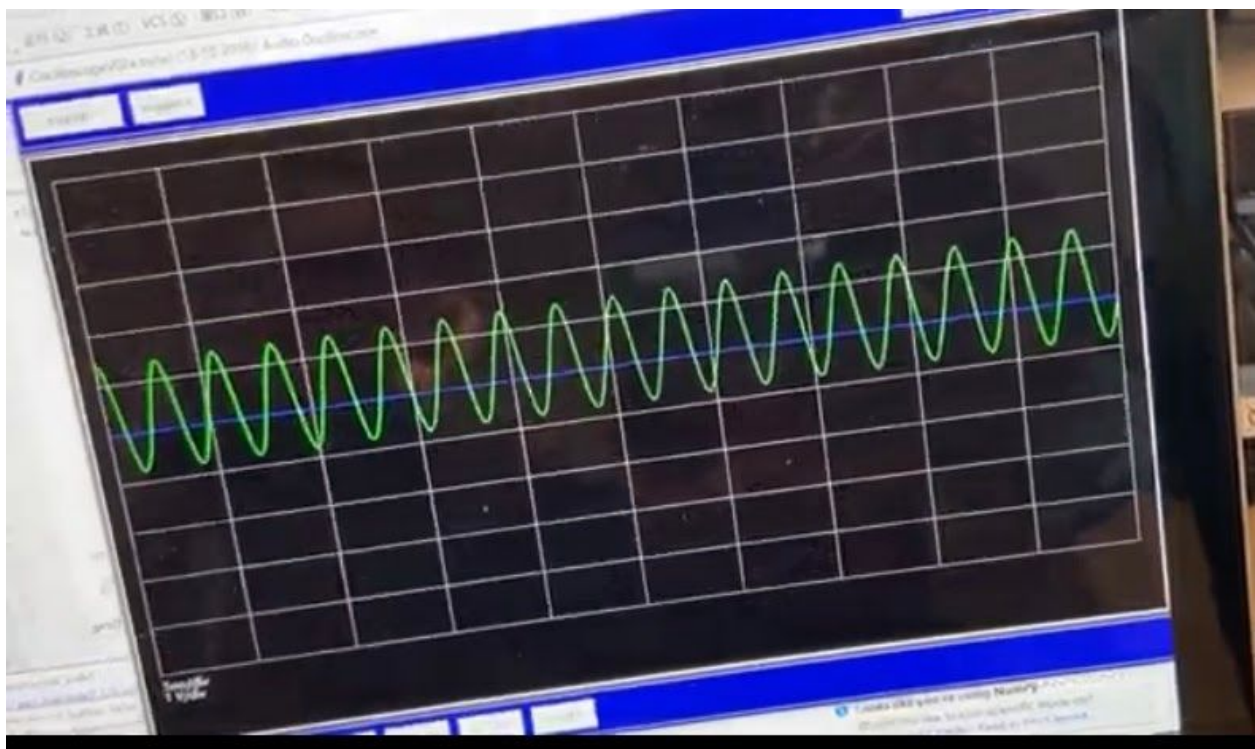


Fig 5 (a) sin waves input on python oscilloscope

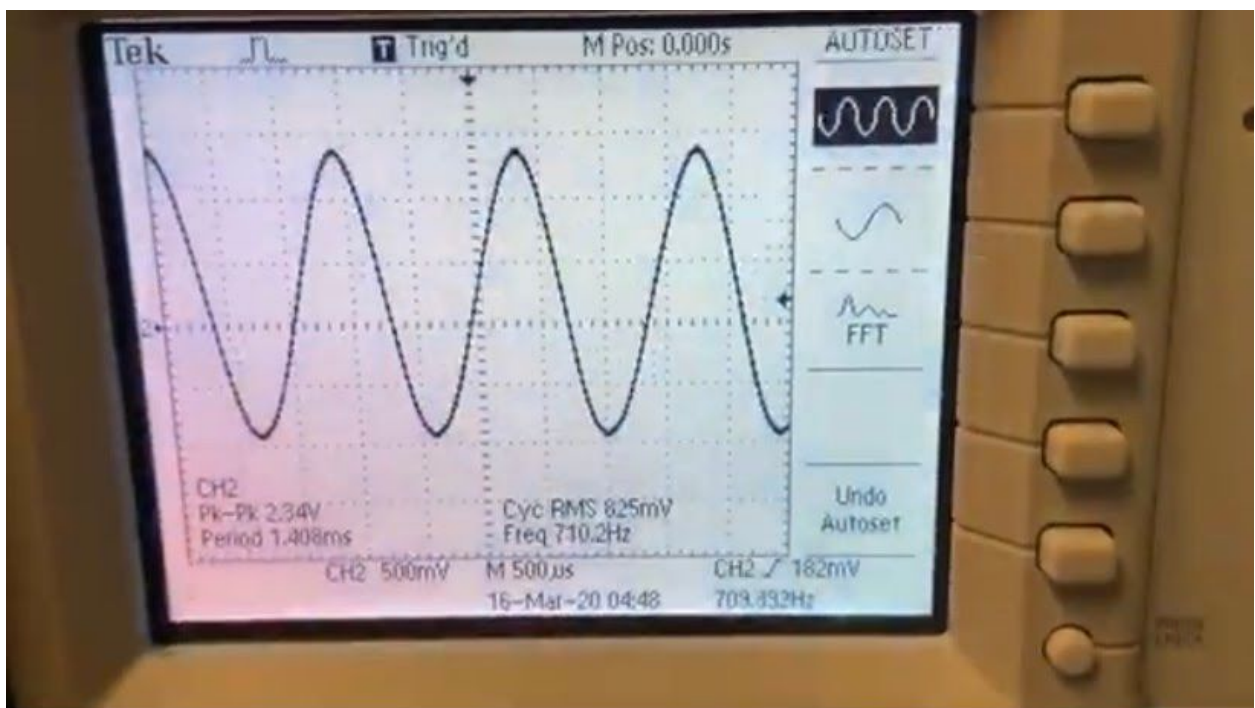


Fig 5 (b) sin waves input on oscilloscope

Additionally, we can also generate the digital signal waveform that was entailed by our client, which is a square wave with a width of 10ms and it repeats this pulse three times at the beginning of one second. The output is depicted as below.

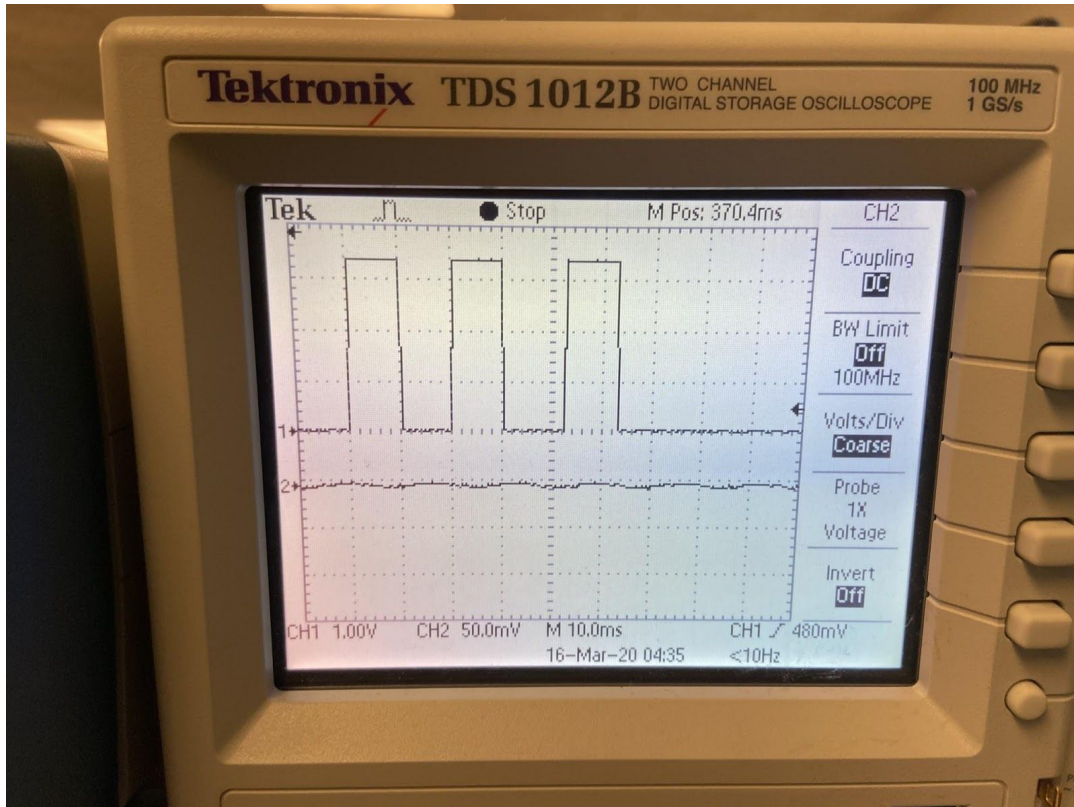


Fig 6 (a) output digital signal with time scale of 10ms

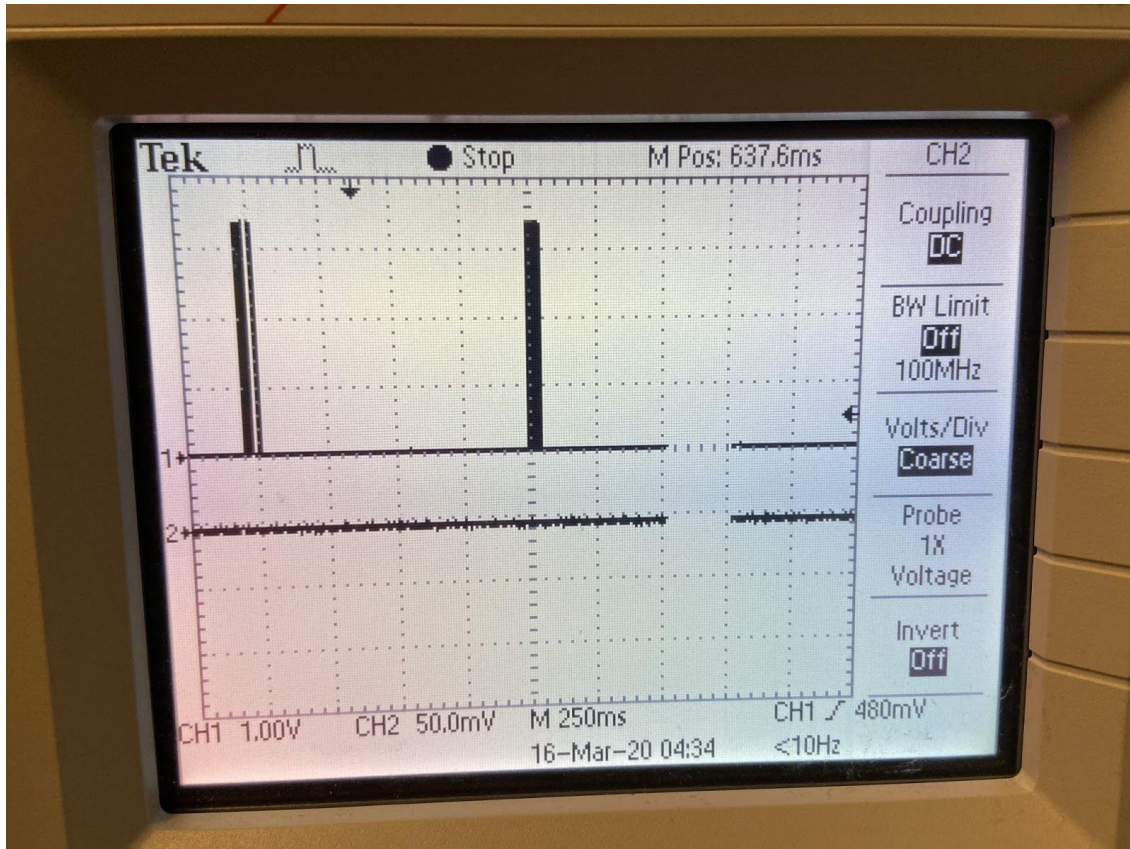


Fig 6 (b) output digital signal with time scale of 250ms

Future scope

Considering the project had to be abruptly ended there are many things that can be added to its scope. And as such, we have the following recommendations from our experience.

Trigger

We had left the project with having a user selected trigger level to trigger the data acquisition of the input signal. However, as per the requirements of the client for feedback, we can have another input pulse signal trigger the primary signal. Having this enabled means using the digital output generated to be used as the trigger signal for the primary signal without needing external pulse signal generators.

Functions implementation

Here we list the functions we could be adding to the project.

The first thing we can do is to make the software able to do the fourier transform. Since we already have the data stream, we can perform FFT on the data stream. The numpy library has the API to call fft. However, we really need to study what the return data stream means. For now, we did a little experiment on just input the sine wave and call that function. We directly output the result and it is not the peak we want. There needs to be more research done on this topic to move further.

Another thing is about a proxy. For an oscillator, we should have proxies to point to the point on x-axis or y-axis we want. As a real oscillator, we need to add up to 4 proxies. The difficulty is that since our scale of axis can be changed, the value of the proxy pointing at also needs to be changed. The convert of the coordinate to the real voltage or frequency still needs to be discussed.

User Interface Optimize

Since this code is for other users, we need to optimize the interface. We are reading through the Tkinter library documentation looking for some optimization into the program so that the buttons, the font can be looked more comfortable and friendly to users. And we met some problems when arranging our interface. In Matlab, the arrangement of each panel and component can be easier. You can directly allocate components or the panel to the coordinate

of the desired place. There's a `uniconrol` function for all the components or panels that you can add almost any properties you can come up with to it. However, in Tkinter, the component (like a button or a label) cannot be initialized with a coordinate. And once the component is put into a panel, the arrange order of components depends on the order you put the component in the panel. And the arrangement of rows and columns of the components is not easy to change. Actually, we did not find a way to put things to the location of the window arbitrarily. This is the problem we are going to finish in the next step.

Except for the aesthetic aspects, we are also going to work on optimizing the time to draw, to read in data. In the current state, when we read in from peripheral devices, we open a explore task each read and let the garbage collection mechanism in python to help us release this part of memory when we finish each read. There should be a way to process multiple reads in one task. This can save time and memory. Also, we are going to check the base code of how the Tkinter library draws each point on the screen. As we know, a released library performs some error checks that sometimes may not be necessary to ordinary users. We will simplify the library to make it lighter so that we can gain a higher performance.

Also, considering the software was meant to be used by the client in his coursework, it needs to be as versatile as possible while being stable throughout. Having consistent crashes or running into open ended loops might be a huge risk. Therefore, a thorough testing must be carried out before releasing it as a final product.

Mac OS

The next step should be making the code work on macOS and finding out how to overcome the overflow of buffers. The second step is to find another eligible ADC converter that has compatible software to manipulate such converters on macOS. However, this is not plausible as we have mentioned before.

Multi Platform Integration

Since we have used tkinter for the demo and most of the gui, we need to consider the possibility of the upgrade of python so as to make sure our program will be usable for future versions of the operating system. After running some tests on tkinter, which turned out to be good so far, we also found that if we run python 3.7, namely, the latest version of python, there will be system crash on macOS, and such error is unanimous on all systems after 10.14. Currently, Apple Inc. has not released the bug-fixed version of system and whenever we run a tkinter method from this library the navigation bar on macOS system will disappear, system will automatically log out current user account and the resolution of Mac screen will change into a very small window size. We can test and develop this program under the python 3.5

environment, but python language is updating pretty fast these days, thus, we cannot be sure if such a GUI library still has its robustness after several years.

Dr. Johnson currently requires NI USB6001 device as data sampler, i.e. ADC. However, we encountered this problem of finding compatible NI devices for the driver software, since the latest driver only supports macOS 10.9.x. Note that this project should be done by May 2020, we can wait for NI to release the latest driver, but such waiting might be futile eventually. And for the semester we need to come up with a backup plan such as using another comparable ADC that has a driver that can be easily implemented on macOS. Obviously, this driver and ADC need to be robust due to the fact that this project is for educational purposes.

Conclusion

We first finished the basic functions of an oscilloscope including displaying audio input, land input, and input from the ADC converter of USB6001 NI. So far, we can plot any waveform from an ADC converter on Windows platform, unfortunately, we cannot solve the problem of buffer overflow when we run the same program on macOS. And the current version of NI interface software for data acquisition device we used does not support any macOS that is later than 10.10v. Hence, using a NI device for data reading is not plausible on macOS. But the oscilloscope we built can support any python editor on PC including but not limited to spyder and pycharm.

The input scale precision can be configured from 1ms to 1s, and voltage level can be configured from 10mV to 5V. Trigger level is also adjustable. For the output digital signal we can have a precision at around 10ms with respect to the built-in function's execution time bottleneck in python. We then built three modes in this program to switch from input and output. Theoretically, simultaneous input and output is possible but requires multi thread programming, and in this scenario, the execution time of each function we used in this program is highly unpredictable. This is why we set up three different modes controlled by three buttons in order to transform from different input and output scenarios. Apparently, using simple input or output mode alone at one time is guaranteed to give the precision we want and applying input and output simultaneously is not recommended.

Unfortunately, we cannot meet the full requirements from our client because of COVID-19, but we have finished about 80% of the original task. The output and input mode in this project is thoroughly done. According to the original plan we should have rudimental operation functions added to this program such as fourier transform to do some basic signal processing. But without access to the laboratory to test the waveforms' correction, we paused right here.

In conclusion, we design and test an oscilloscope with python as source code. And this oscilloscope can be used and configured with an NI data acquisition device on PC, which is free of authorization fee and robust in GUI. Future update and expansion of this oscilloscope can be performed by adding more signal processing functions and data saving functions to it.

References

- Lott GK, Johnson BR, Bonow RH, Land BR, Hoy RR, "g-PRIME: A Free, Windows Based Data Acquisition and Event Analysis Software Package for Physiology in Classrooms and Research Labs" J Undergrad Neurosci Ed, Fall 2009, 8(1):A50-A54
- GUI Programming in Python, <https://wiki.python.org/moin/GuiProgramming>, last accessed on 15th Dec 2019.
- How Soundcard works, <https://computer.howstuffworks.com/sound-card2.htm>, last accessed on 15th Dec 2019.
- Audio Oscilloscope With Soundcard And Software Written In Python, <https://www.qsl.net/pa2ohh/11oscilloscope.htm>, last accessed on 15th Dec 2019.
- National instruments model USB 6008, <https://www.ni.com/en-us/support/model.usb-6008.html>, last accessed on 15th Dec 2019.

Appendix

Code:

```
import numpy
import pyaudio
import nidaqmx as dq
import math
import time
from tkinter import *
from tkinter import messagebox
from tkinter import filedialog
from tkinter import simpledialog
from tkinter import font
from tkinter import ttk
import _thread
#-----#
GRW = 1000
GRH = 500
XOL = 20
YOT = 25
CANVASwidth = GRW + 2 * XOL
CANVASheight = GRH + 2 * YOT + 48
Ymin = YOT
Ymax = YOT + GRH
LONGsweep = False
LONGchunk = LONGsweep
Samplelist = [1000, 2000, 2500, 5000, 7500, 10000, 20000, 40000]
CHvdiv = [0.01, 0.1, 1.0, 10.0, 20.0, 50.0, 100.0, 1000.0] # Sensitivity list in
mv/div
TIMEdiv = [0.1, 0.2, 0.5, 1.0, 2.0, 5.0, 10.0, 20.0, 50.0, 100.0] # Time list in
ms/div
ADzero = False
COLORframes = "#000080" # Color = "#rrggbb" rr=red gg=green bb=blue, Hexadecimal
values 00 - ff
COLORcanvas = "#000000"
COLORgrid = "#808080"
COLORzeroline = "#0000ff"
COLORtrace1 = "#00ff00"
COLORtrace2 = "#ff8000"
COLORtext = "#ffffff"
```

```

COLORtrigger = "#ff0000"
Buttonwidth1 = 12
Buttonwidth2 = 12
TRACES = 1
TRACESread = 0
RUNstatus = 1
SAMPLErate = 15000
Devicedict = {}
output = True
flag = True
mode = 0
#-----
Devicename1 = []
Devicename2 = []
Devicedict = {}

```

```

class channels:
    global NItask
    def __init__(self):
        self.timediv = 6
        self.chdiv = len(CHvdiv) - 1
        self.Tline = []
        self.AUDIOsignal = []
        self.Triggerline = []
        self.SHOWsamples = GRW
        self.NItask = None
        self.stream = None
        self.CurDeviceName = None
        self.offset = 0
        self.AUDIOdevin = None
        self.ADsens = 1000
        self.audiosize = int (SAMPLErate * TIMEdiv[self.timediv] * 10 /1000)
        self.PA = None
        self.triggerlevel = -200
        self.outputtask = dq.Task()
    def setaudiosize(self):
        self.audiosize = int (SAMPLErate * TIMEdiv[self.timediv] * 10 / 1000)
    def maketrace(self, c):
        global XOL
        global YOT
        global GRW

```

```

global GRH
global Ymin
global Ymax
global TRACES
global RUNstatus
global CHvdiv
global TIMEdiv
global SAMPLERate
TRACEsize = len(self.AUDIOsignal)
# print(TRACEsize)
# print(self.AUDIOsignal)
if TRACEsize == 0:
    self.Tline = []
    return()
Yconv = float(GRH / 10) * 1000 / (self.ADsens * CHvdiv[self.chdiv])
self.SHOWsamples = SAMPLERate * 10 * TIMEdiv[self.timediv] / 1000
self.Tline = []
t = 0
x = 0
Tstep = 0
if (self.SHOWsamples >= GRW):
    self.SHOWsamples = GRW
    Tstep = (int)(self.SHOWsamples/ GRW)
if (self.SHOWsamples < GRW):
    expand = []
    n = int(GRW / self.SHOWsamples) + 1
    for o in range(0, len(self.AUDIOsignal)):
        for i in range(0,n):
            expand.append(self.AUDIOsignal[o])
    self.AUDIOsignal = expand
    Tstep = 1
Xstep = 1
x1 = 0
y1 = 0.0
while(x <= GRW):
    x1 = x + XOL
    y = float(self.AUDIOsignal[t])
    ytemp = int(c - Yconv * y)
    if (ytemp < Ymin):
        ytemp = Ymin
    if (ytemp > Ymax):
        ytemp = Ymax
    self.Tline.append(int(x1))

```

```

        self.Tline.append(int(ytemp))
        t = t + Tstep
        x = x + Xstep
    x1 = XOL
    y1 = int(c - Yconv * float(0))
    if (y1 < Ymin):
        y1 = Ymin
    if (y1 > Ymax):
        y1 = Ymax
    self.Triggerline.append(int(XOL - 5))
    self.Triggerline.append(int(y1))
    self.Triggerline.append(int(XOL + 5))
    self.Triggerline.append(int(y1))
channel1 = channels()
# channel2 = channels()

def generate():
    global channel1
    global mode
    global RUNstatus
    while True:
        if mode == 1 or mode == 2:
            width = 0.01
            period = 0.1 - (width*6)
            if (RUNstatus == 0):
                while True:
                    a = 0
            if RUNstatus == 1:
                while channel1.CurDeviceName == None or channel1.outputtask == None:
                    a = 0
                try:
                    channel1.outputtask.close()
                except:
                    pass
                try:
                    channel1.outputtask = dq.Task()

channel1.outputtask.do_channels.add_do_chan(channel1.CurDeviceName[:-3] +
"port0/line0")
                except:
                    pass
            if RUNstatus == 2 :
                channel1.outputtask.start()

```

```

        start0 = time.time()
        for i in range(0, 3):
            start = time.time()
            channel1.outputtask.write(True)
            while (time.time() - start < width):
                a = 1
                # time.sleep(width - 0.0025)
            start = time.time()
            channel1.outputtask.write(False)
            while (time.time() - start < width):
                a = 1

        while (time.time() - start0 < period):
            a = 1
            # time.sleep( period - 2*width*3)
            channel1.outputtask.stop()
    if RUNstatus == 3 or RUNstatus == 4:
        channel1.outputtask.close()
        channel1.outputtask = None

def BTrigger1():
    channel1.triggerlevel -= CHvdiv[channel1.chdiv] / 10
    print(channel1.triggerlevel)
    RUNstatus = 1
    UpdateScreen()

def BTrigger2():
    channel1.triggerlevel += CHvdiv[channel1.chdiv] / 10
    print(channel1.triggerlevel)
    RUNstatus = 1
    UpdateScreen()

def BStart():
    global RUNstatus
    if (RUNstatus == 0):
        RUNstatus = 1
    UpdateScreen()

def BTime1(channel : channels):
    global RUNstatus
    if (channel.timediv >= 1):

```



```

        channel.timediv = channel.timediv - 1
    if RUNstatus == 2:
        RUNstatus = 4
    UpdateTrace()

def BTime2(channel : channels):
    global RUNstatus
    global TIMEdiv
    if (channel.timediv < len(TIMEdiv) - 1):
        channel.timediv = channel.timediv + 1
    if RUNstatus == 2:
        RUNstatus = 4
    UpdateTrace()

def BCHlevel1(channel : channels):
    global RUNstatus
    if (channel.chdiv >= 1):
        channel.chdiv = channel.chdiv - 1
    UpdateTrace()

def BCHlevel2(channel : channels):
    global RUNstatus
    if (channel.chdiv < len(CHvdiv) - 1):
        channel.chdiv = channel.chdiv + 1
    UpdateTrace()

def BSetup(*args):
    global ADzero
    global SAMPLErate
    global RUNstatus

    s = title.get()
    if (s == None): # If Cancel pressed, then None
        return ()

    try: # Error if for example no numeric characters or OK pressed without input (s
= ""), then v = 0
        v = int(s)
    except:
        v = 0

```

```

if v != 0:
    SAMPLERate = v
    print(SAMPLERate)
if (RUNstatus == 2):
    RUNstatus = 4
UpdateScreen()

def AUDIOin():
    global channel1
    global channel2
    global RUNstatus
    global mode
    while(True):
        channel1.setaudiosize()
        # channel2.setaudiosize()
        if (channel1.AUDIOdevin == None):
            RUNstatus = 0
        if (RUNstatus == 1):
            Action1(channel1)
        #UpdateScreen()
        if (RUNstatus == 2 ):
            if mode == 2 or mode == 0:
                Action2(channel1)
            elif mode == 1:
                channel1.AUDIOsignal = numpy.zeros(int(1.5 *
channel1.audiosize),dtype=numpy.bool)
                MakeScreen()
                # if (TRACES == 2):
                #     Action2(channel2)
        if (RUNstatus == 3) or (RUNstatus == 4):
            Action3(channel1)
            if (TRACES == 2):
                Action3(channel2)
        UpdateAll()
        root.update_idletasks()
        root.update()

def Action1(channel : channels):
    global RUNstatus
    global output
    global flag
    if (channel.AUDIOdevin != None):
        PA = pyaudio.PyAudio()

```

```

FORMAT = pyaudio.paInt16
chunkbuffer = int(3000)
if (channel.AUDIOdevin < 15):
    try:
        channel.stream = PA.open(format=FORMAT,
                                channels=TRACES,
                                rate=SAMPLErate,
                                input=True,
                                output=False,
                                frames_per_buffer=int(chunkbuffer),
                                input_device_index=channel.AUDIOdevin)

        RUNstatus = 2
    except:
        RUNstatus = 0
        txt = "Something wrong when creating stream for the device"
        messagebox.showerror("Cannot open Audio Stream", txt)
else:
    try:
        channel.NItask.close()
    except:
        pass
    channel.NItask = dq.Task()
    for i in Devicedict.keys():
        if Devicedict.get(i) == channel.AUDIOdevin:
            channel.CurDeviceName = i
            break
    try:
        addedChan = False
        for i in channel.NItask.ai_channels:
            if i.name == channel.CurDeviceName:
                addedChan = True
        if addedChan == False:

channel.NItask.ai_channels.add_ai_voltage_chan(channel.CurDeviceName)
        channel.NItask.start()
        flag = True
        RUNstatus = 2
    except Exception as e:
        RUNstatus = 0
        txt = "Task cannot be created, check the device connection and
channel name"
        messagebox.showerror("Error when creating NI tasks", txt)
        print(e)

```

```

def Action2(channel : channels):
    global RUNstatus
    global output
    AUDIOsignals = []
    if (channel.AUDIOdevin != None):
        if (channel.AUDIOdevin < 15):
            while len(AUDIOsignals) < channel.audiosize:
                buffervalue = channel.stream.get_read_available()
                if buffervalue > 1024:
                    signals = channel.stream.read(buffervalue)
                    AUDIOsignals.extend(numpy.fromstring(signals, "Int16"))
            else:
                while len(AUDIOsignals) < 1.5 * channel.audiosize:
                    signals = channel.NItask.read(80)
                    for i in range(0, len(signals)):
                        signals[i] = signals[i] * 1000
                    AUDIOsignals.extend(signals)
                channel.NItask.stop()
        while (len(AUDIOsignals) > 0 and AUDIOsignals[0] < channel.triggerlevel):
            del AUDIOsignals[0]
        channel.AUDIOsignal = AUDIOsignals
        # MakeScreen()
def Action3(channel : channels):
    global RUNstatus
    global flag
    if (channel.AUDIOdevin != None):
        if (channel.AUDIOdevin < 15):
            channel.stream.stop_stream()
            channel.stream.close()
        else:
            channel.NItask.close()
            channel.NItask = None
            # flag = False

    if (channel.PA != None):
        channel.PA.terminate()
    if RUNstatus == 3:
        RUNstatus = 0
    if RUNstatus == 4:
        RUNstatus = 1

```

```

def UpdateAll():
    CalculateData()
    MakeTrace()
    UpdateScreen()

def UpdateTrace():
    MakeTrace()
    UpdateScreen()

def UpdateScreen():
    MakeScreen()
    root.update()

def MakeScreen():
    global XOL
    global YOT
    global GRW
    global GRH
    global Ymin
    global Ymax
    global CHvdiv
    global TIMEdiv
    global CANVASwidth
    global CANVASheight
    global ADSens
    global SAMPLErate
    global channel1
    global channel2
    de = ca.find_enclosed(0, 0, CANVASwidth + 1000, CANVASheight + 1000)
    for n in de:
        ca.delete(n)
    i = 0
    x1 = XOL
    x2 = XOL + GRW
    while (i < 11):
        y = YOT + i * GRH / 10
        Dline = [x1, y, x2, y]
        ca.create_line(Dline, fill=COLORgrid)
        i = i + 1
    if TRACES == 1:
        y = YOT + 5 * GRH / 10
        Dline = [x1, y, x2, y]

```

```

        ca.create_line(Dline, fill = COLORzeroline)
if TRACES == 2:
    y = YOT + GRH / 4
    Dline = [x1,y,x2,y]
    ca.create_line(Dline, fill=COLORzeroline)          # Blue horizontal line 1 for
2 traces
    y = YOT + 3 * GRH / 4
    Dline = [x1,y,x2,y]
    ca.create_line(Dline, fill=COLORzeroline)
i = 0
y1 = YOT
y2 = YOT + GRH
while (i < 11):
    x = XOL + i * GRW / 10
    Dline = [x, y1, x, y2]
    ca.create_line(Dline, fill=COLORgrid)
    i = i + 1
vx = TIMEdiv[channel1.timediv]
if vx >= 1000:
    txt = str(int(vx / 1000)) + "s/div"
if vx < 1000 and vx >= 1:
    txt = str(int(vx)) + "ms/div"
if vx < 1:
    txt = "0." + str(int(vx * 10)) + "ms/div"
if vx <= 0.01:
    txt = "0." + "0" + str(int(vx * 100)) + "ms/div"
x = XOL
y = YOT + GRH + 12
ca.create_text(x, y, text= txt, anchor = W, fill = COLORtext)
vy = CHvdiv[channel1.chdiv]
txt = ""
if vy >= 1000:
    txt = txt + str(int(vy/1000)) + " V/div"
if vy < 1000 and vy >= 1:
    txt = txt + str(int(vy)) + " mV/div"
if vy < 1:
    txt = txt + "0." + str(int(vy * 10)) + " mV/div"
x = XOL
y = YOT+GRH+24
idTXT = ca.create_text (x, y, text=txt, anchor=W, fill=COLORtext)
txt = ""
# vx = TIMEdiv[channel2.timediv]
# if vx >= 1000:

```

```

#     txt = str(int(vx / 1000)) + "s/div"
# if vx < 1000 and vx >= 1:
#     txt = str(int(vx)) + "ms/div"
# if vx < 1:
#     txt = "0." + str(int(vx * 10)) + "ms/div"
# if vx <= 0.01:
#     txt = "0." + "0" + str(int(vx * 100)) + "ms/div"
# x = XOL
# y = YOT + GRH + 36
# ca.create_text(x, y, text= txt, anchor = W, fill = COLORtext)
# txt = ""
# vy = CHvdiv[channel2.chdiv]
# txt = ""
# if vy >= 1000:
#     txt = txt + str(int(vy/1000)) + " V/div"
# if vy < 1000 and vy >= 1:
#     txt = txt + str(int(vy)) + " mV/div"
# if vy < 1:
#     txt = txt + "0." + str(int(vy * 10)) + " mV/div"
# x = XOL
# y = YOT+GRH+48
# idTXT = ca.create_text (x, y, text=txt, anchor=W, fill=COLORtext)
if (len(channel1.Tline) > 4):
    ca.create_line(channel1.Tline, fill = COLORtrace1)
# if (TRACES == 2):
#     if (len(channel2.Tline) > 4):
#         ca.create_line(channel2.Tline, fill = COLORtrace2)
i = 0
def MakeTrace():
    global channel1
    global channel2
    global XOL
    global YOT
    global GRW
    global GRH
    global Ymin
    global Ymax
    global TRACES
    global RUNstatus
    global CHvdiv
    global TIMEdiv
    global SAMPLErate
    if len(channel1.AUDIOsignal) == 0:

```

```

        return
Yconv1 = float(GRH / 10) * 1000 / (channel1.ADsens * CHvdiv[channel1.chdiv])
if (TRACES == 1):
    c1 = GRH / 2 + YOT - channel1.offset
if (TRACES == 2):
    c1 = GRH / 4 + YOT - channel1.offset
    c2 = 3 * GRH / 4 + YOT - channel2.offset
    channel2.maketrace(c = c2)
channel1.maketrace(c = c1)

def ReadInDevice():
    global Devicename1
    global Devicename2
    global Devicedict
    Devicename1 = []
    Devicename2 = []
    PA = pyaudio.PyAudio()
    s = PA.get_device_info_by_index(0)
    Devicedict[s['name']] = s['index']
    Devicename1.append(s['name'])
    nisystem = dq.system.System.local()
    nidenum = len(nisystem.devices)
    n = 15
    for i in range(nidenum):
        channels = nisystem.devices[i].ai_physical_chans
        cnum = len(channels)
        for j in range(cnum):
            Devicename1.append(channels[j].name)
            Devicedict[channels[j].name] = n
            n = n + 1
        PA.terminate()
    Devicename2 = Devicename1
    Devicebox1['values'] = Devicename1
    # Devicebox2['values'] = Devicename2

def change(*args):
    global channel1
    global RUNstatus
    s = Devicebox1.get()

```



```

        channel1.AUDIOdevin = Devicedict[s]
        print(s + ":" + str(channel1.AUDIOdevin))
        if (RUNstatus == 2) or RUNstatus == 1:
            RUNstatus = 4
        RUNstatus = 1

def Bmode0():
    global mode
    global RUNstatus
    mode = 0
    if RUNstatus == 2 or RUNstatus == 1:
        RUNstatus = 4
    RUNstatus = 1

def Bmode1():
    global mode
    global RUNstatus
    mode = 1
    if RUNstatus == 2 or RUNstatus == 1:
        RUNstatus = 4
    RUNstatus = 1

def Bmode2():
    global mode
    global RUNstatus
    mode = 1
    if RUNstatus == 2 or RUNstatus == 1:
        RUNstatus = 4
    RUNstatus = 1
    mode = 2

# def change_1(*args):
#     global channel2
#     global RUNstatus
#     s = Devicebox2.get()
#     channel2.AUDIOdevin = Devicedict[s]
#     print(s + ":" + str(channel2.AUDIOdevin))
#     if (RUNstatus == 2):
#         RUNstatus = 4
#     RUNstatus = 1

def CalculateData():
    return()

```

```

def BStop():
    global RUNstatus
    if (RUNstatus == 1):
        RUNstatus = 0
    elif (RUNstatus == 2):
        RUNstatus = 3
    elif (RUNstatus == 3):
        RUNstatus = 3
    elif (RUNstatus == 4):
        RUNstatus = 3

# def BTraces():
#     global TRACES
#     global RUNstatus
#
#     if (TRACES == 1):
#         TRACES = 2
#     else:
#         TRACES = 1
#
#     if RUNstatus == 2: # Restart if running
#         RUNstatus = 1

root = Tk()
root.title("OscilloscopeV02a.py(w) (13-10-2018): Audio Oscilloscope")
root.minsize(100, 100)
title = StringVar()
title.set(10000)
frame1 = Frame(root, background=COLORframes, borderwidth=5, relief=RIDGE)
frame1.pack(side=LEFT, expand=1, fill=Y)

frame2 = Frame(root, background="black", borderwidth=5, relief=RIDGE)
frame2.pack(side=TOP, expand=1, fill=X)

frame3 = Frame(root, background=COLORframes, borderwidth=5, relief=RIDGE)
frame3.pack(side=TOP, expand=1, fill=X)

ca = Canvas(frame2, width=CANVASwidth, height=CANVASheight, background=COLORcanvas)
ca.pack(side=TOP)

# b = Button(frame1, text="1/2 Channels", width=Buttonwidth1, command=BTraces)
# b.pack(side=LEFT, padx=5, pady=5)

```

```

b = Button(frame1, text = "trigger+", width = Buttonwidth2, command = BTrigger2)
b.pack(side = TOP, padx = 10, pady = 10)
b = Button(frame1, text = "trigger-", width = Buttonwidth2, command = BTrigger1)
b.pack(side = TOP, padx = 10, pady = 10)
b = Button(frame1, text="mode1", width=Buttonwidth2, command=Bmode0)
b.pack(side= TOP, padx=10, pady=10)
b = Button(frame1, text="mode2", width=Buttonwidth2, command=Bmode1)
b.pack(side= TOP, padx=10, pady=10)
b = Button(frame1, text="mode3", width=Buttonwidth2, command=Bmode2)
b.pack(side= TOP, padx=10, pady=10)
b = Label(frame1, text = "Samplerate", width = Buttonwidth2,bg = COLORframes, fg =
COLORtext)
b.pack(side = TOP, padx = 5, pady = 2)
m=OptionMenu(frame1,title,*Samplelist)
m.config(width=Buttonwidth2-5)
m.pack(side=TOP, padx=5, pady=5)
b = Label(frame1, text = "Devices", width = Buttonwidth2,bg = COLORframes, fg =
COLORtext)
b.pack(side = TOP, padx = 5, pady = 2)
Devicebox1=ttk.Combobox(frame1, width=Buttonwidth2 - 2, postcommand=ReadInDevice,
values=Devicename1, )
Devicebox1.pack(side=TOP, padx=5, pady=2)
Devicebox1.bind("<<ComboboxSelected>>",change)

```

```

b = Button(frame3, text="Start", width=Buttonwidth2, command=BStart)
b.pack(side=LEFT, padx=5, pady=5)
b = Button(frame3, text="Stop", width=Buttonwidth2, command=BStop)
b.pack(side=LEFT, padx=5, pady=5)

```

```

b = Button(frame3, text="-Time", width=Buttonwidth2, command=lambda :
BTime1(channel1))
b.pack(side=LEFT, padx=5, pady=5)

```

```

b = Button(frame3, text="+Time", width=Buttonwidth2, command=lambda :
BTime2(channel1))
b.pack(side=LEFT, padx=5, pady=5)

```

```

b = Button(frame3, text="-CH1", width=Buttonwidth2, command= lambda :
BCHlevel1(channel1))
b.pack(side=LEFT, padx=5, pady=5)

```

```

b = Button(frame3, text="+CH1", width=Buttonwidth2, command= lambda :
BCHlevel2(channel1))
b.pack(side=LEFT, padx=5, pady=5)
# Devicebox2=ttk.Combobox(frame1, width=Buttonwidth1, postcommand=ReadInDevice,
values=Devicename2, )
# Devicebox2.pack(side=RIGHT, padx=5, pady=5)
# Devicebox2.bind("<<ComboboxSelected>>",change_1)
#l = Label(frame1,text="Devices:",background="#000080",fg="ffffff")
#l.pack(side=RIGHT, padx=5, pady=5)
title.trace_variable('w', BSetup)
if __name__ == '__main__':
    root.update()
    _thread.start_new(generate, ())
    AUDIOin()

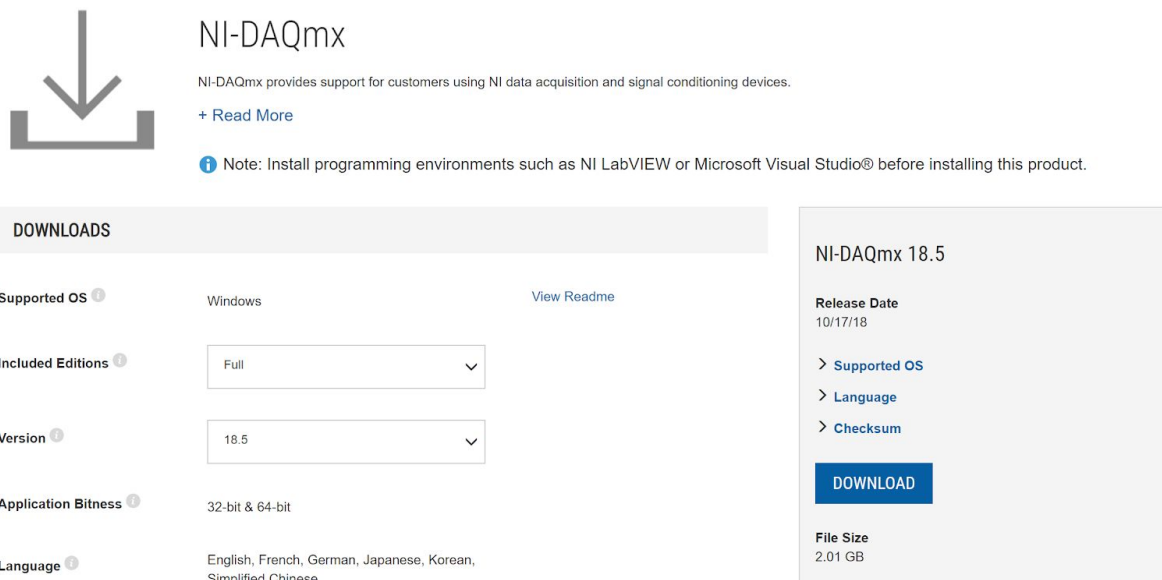
```

User Manual

Install package:

Go to <https://www.ni.com/en-us/support/downloads/drivers/download.ni-daqmx.html#288283> to install needed NI data acquisition devices drivers. It should contain the drivers for all the type of NI data acquisition devices.

Choose the version for your system and click the blue button “Download”.



NI-DAQmx

NI-DAQmx provides support for customers using NI data acquisition and signal conditioning devices.

[+ Read More](#)

Note: Install programming environments such as NI LabVIEW or Microsoft Visual Studio® before installing this product.

DOWNLOADS

Supported OS ¹ Windows [View Readme](#)

Included Editions ¹ Full

Version ¹ 18.5

Application Bitness ¹ 32-bit & 64-bit

Language ¹ English, French, German, Japanese, Korean, Simplified Chinese

NI-DAQmx 18.5

Release Date
10/17/18

[> Supported OS](#)

[> Language](#)

[> Checksum](#)

DOWNLOAD

File Size
2.01 GB

Follow the Installer to install. No extra configuration needs to be set.

Install nidaqmx library for python. If you are using Anaconda, you can use this command:

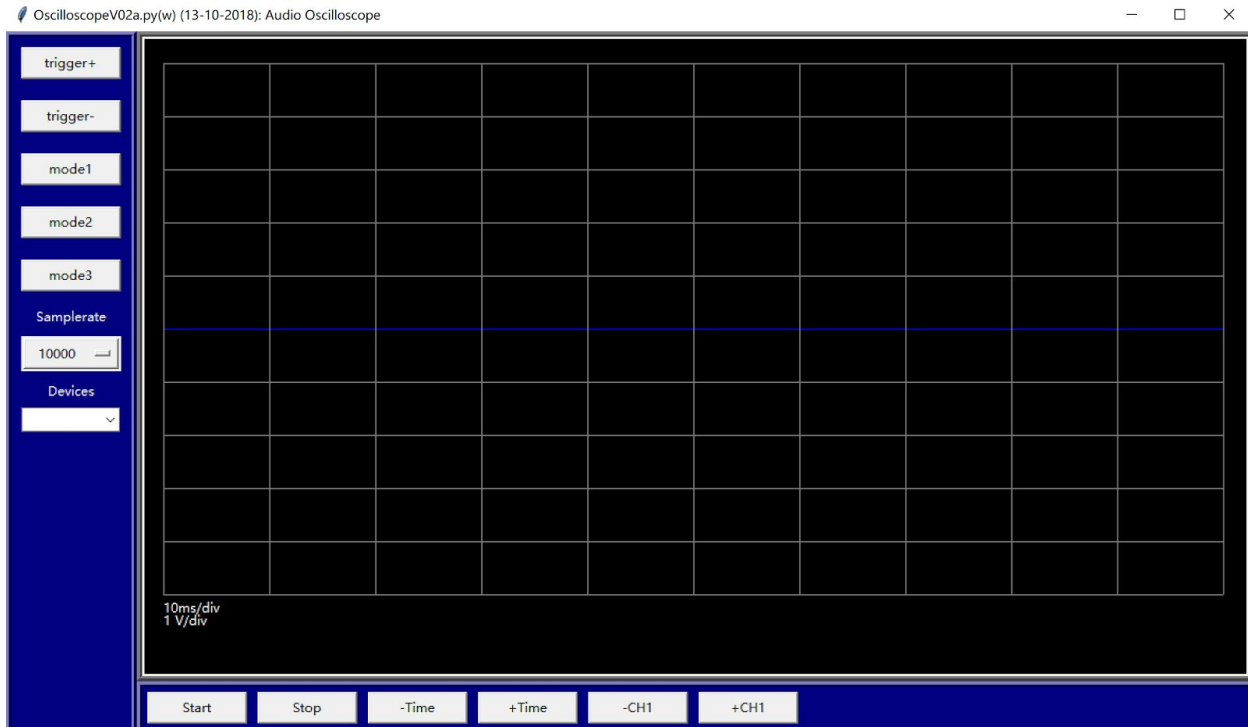
```
conda install -c conda-forge nidaqmx-python
```

Or just the traditional pip install command:

```
pip install nidaqmx
```

Interface:

The whole interface is like this



1. "Devices" combo box allows us to choose the available devices. Click on the right of the combo box, it will show the available devices like. The default device is only the sound card input. The format for the name of NI daq devices will be like "Dev0/ai0". The Dev0 is the virtual name of the connected device. The "ai0" after the "/" is the port number on the device. In the current version, we only take the analog input ports.
2. The sample rate can choose the number of sample points each read. There may be some confusion here. The reading speed for the NI daq devices is fixed. So this sample rate here means the number of sample points per working loop.
3. There are three mode buttons: mode1, mode2, mode3. "mode1" is the mode only read in signals and display. "mode2" is the mode that only outputs the generated signal and there will be no update on the screen. "mode3" can do the signal display and output signal at the same time. But mode3 may not be stable because read-in and output threads will affect each other and influence the performance.
4. Trigger buttons are to modify the inner trigger. Usually a higher trigger level will make the display of the signal more stable. "trigger+" is to increase the trigger level. "trigger-" is to lower the trigger level.
5. Start button is to restart the program from the exceptions(may be the abrupt disconnection of the device or other unexpected errors) and the status after you clicked the stop button.
6. Stop button is to pause the program
7. "-Time" is to lower the time scope we are looking at. "+Time" is to enlarge the time scope we are looking at.
8. "-CH1" is to observe the signal in a smaller voltage scale. "+CH1" is to enlarge the voltage scale per division.