ANALOG DEVICES
AHEAD OF WHAT'S POSSIBLE™

Virtual Electronics Lab: How to Create an Oscilloscope Using Python and the ADALM2000

Christian Jason Garcia, Software Systems Engineer, and
Arnie Mae Baes, Software Systems Engineer

Abstract

A virtual electronics laboratory is a collection of software-based instruments. It is a simulated electronics lab environment implemented as a software application, and it allows users to perform a multitude of electronic experiments. Having a physical, fully functional laboratory can be expensive and hard to manage. Imagine having the capability of an electronics laboratory that can fit inside your pocket; the possibilities are endless!

This article aims to demonstrate how users can develop their own virtual laboratory instruments using the ADALM2000. The Python programming language will be used in this article due to its simplicity, and also because it is open source. With the combination of Python and the ADALM2000, it is possible to develop several virtual laboratory instruments such as an oscilloscope, signal generator, digital multimeter, and many more. However, this article will only concentrate on one instrument—the oscilloscope. This is a good instrument to start with since it is one of the most fundamental instruments that we use in an actual electronics laboratory.

Introduction

The instrumentation industry is moving steadily and rapidly toward virtualization. Software-based instruments are hosted on a PC that uses as minimal specialized hardware as possible to link it to the devices it must measure/control. This hardware typically includes plug-in boards for digitizing a signal directly or for controlling standalone instruments.

Virtual instrumentation is known for its flexibility, modularity, and portability. Analog Devices provides its customers with electronic modules that cater to almost every use case there is, including this one. A good example of this module is the ADALM2000.

The ADALM2000 enables engineers or developers to create their own virtual electronics laboratory depending on their specific needs. Via the libm2k library, users can develop software applications for controlling the ADALM2000 using C++, C#, or Python. More detailed discussions about the ADALM2000 and libm2k will follow in later sections.

What Is an Oscilloscope?

Oscilloscopes are an essential part of electronics engineering due to the value they bring to signal analysis of common and complex circuits. In addition to that, oscilloscopes nowadays feature computer connectivity so that the signal captured in the oscilloscope can be digitally stored for later analysis.

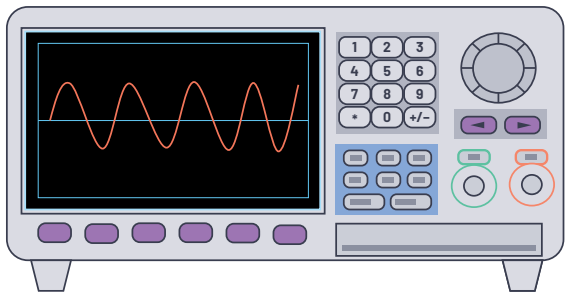

Figure 1. Representation of an oscilloscope.

An oscilloscope is used to visualize the voltage and time characteristics of an analog or digital waveform. The front panel controls—amplifier triggering, sweep timing, and display—are used to tweak the display to better visualize the signal.

It shows us the behavior of a signal input over a certain period, which is essential for analyzing common circuits. It also helps in verifying the functionality of these circuits. This is a major reason why an oscilloscope is an indispensable part of any electronics lab equipment. In addition to this, we can improve the analysis of certain electronic circuits by allowing engineers to customize their very own oscilloscope to suit their needs.

What Is the ADALM2000?

The ADALM2000 is an active learning module that features a digital oscilloscope, function generator, logic analyzer, voltmeter, spectrum, and digital bus analyzer, along with two programmable power supplies. For basic users or students, Scopy can be used to interface with the ADALM2000. For application developers, an application interface can be developed using the libm2k library. There is also an option for firmware developers to develop a custom software or HDL that can run directly on the ADALM2000.

Get Started

Installation of Python and PyCharm

Python is a powerful and easy to learn open-source programming language. Python can be downloaded from the official Python website. Select Python 3.7 if you are unsure which version to use.

Python can be used without an integrated development environment (IDE), but to make downloading libraries and debugging hassle free, PyCharm is can be used. PyCharm is an IDE providing several essential tools for developers, making it the most popular IDE used for Python development. Download the latest version of PyCharm Community on the official JetBrains website.

Installation of Libraries

Python libraries contain methods or functions that can be used for specific applications. In this article, we will use libm2k, matplotlib, and NumPy.

Libm2k

To interface with the ADALM2000 using Python, you need to install the libm2k library. This is a C++ library with available bindings for Python, C#, MATLAB®, and LabVIEW®, and has the following functionalities:

- **AnalogIn** is for an oscilloscope or voltmeter. We will focus on this one.
- **AnalogOut** is for a signal generator.
- **Digital** is for a logic analyzer or pattern generator.
- **PowerSupply** is for a constant voltage generator.
- **DMM** is for a digital multimeter.

Detailed information about this library can be found on the libm2k wiki page.

Installation of Libm2k

One way to install this library is by following these steps:

- Go to the release page.
  - Download the latest executable version of the library. Example: Libm2k-0.4.0-Windows-Setup.exe
- Run the executable. Make sure to select **Install libm2k Python bindings** when the Setup window prompts you to select additional tasks.

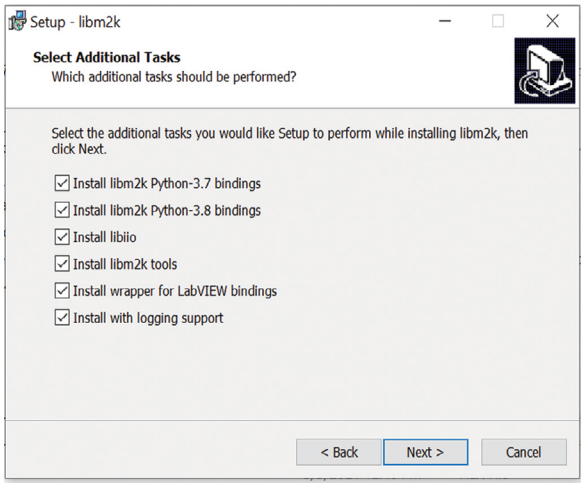

Figure 2. Libm2k installation window.

- Finish the installation. Libm2k will be installed in the default environment of Python.

Matplotlib

To create the oscilloscope display, you need to use the matplotlib library. This library is popular and easy to use for customizing and displaying visualizations in Python. Detailed information about this library can be found on the matplotlib website.

NumPy

A simple oscilloscope will still require a lot of mathematical computations. The NumPy library can help by providing simple functions for complex computations. Detailed information about this library can be found on the NumPy website.

Installation of Matplotlib and NumPy

To install both matplotlib and NumPy, follow these steps in PyCharm:

- Go to File > Settings > Project Interpreter.
- Click the + icon located on the right side of the Settings Window.
- The Available Packages window will appear. In the search box, search for matplotlib and NumPy.
- Specify the version to be installed (select the latest version).
- Click the **Install Package** button.

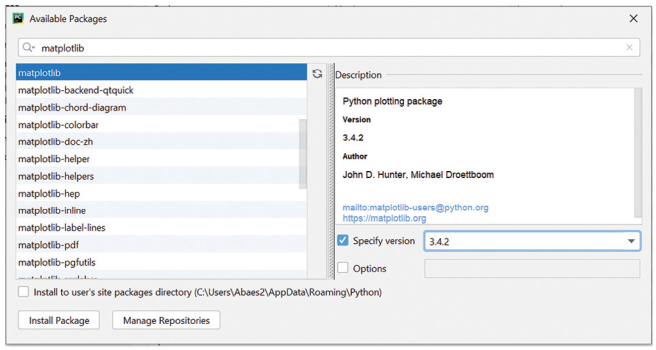

Figure 3. Installing library packages in PyCharm.

Hardware Setup

Before we start coding, let's set up the hardware components. The following hardware components are required:

- Signal source (or a signal generator, if available)
- ADALM2000
- Probes and clippers

If a signal generator is available, connect the ADALM2000 device to Channel 1 and Channel 2 with probes and/or clippers using the configuration shown in Figure 4.

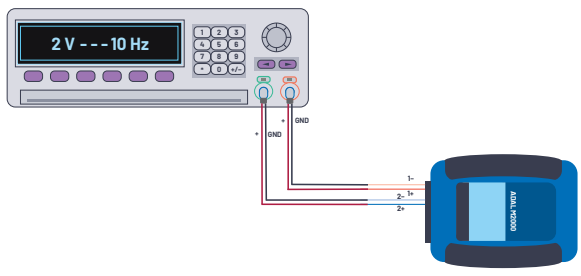

Figure 4. An actual setup with a signal generator and ADALM2000.

Table 1. Pin Configuration

Signal Generator	ADALM2000
Ch1 Positive Wire (+)	1+
Ch1 Ground	1-
Ch2 Positive Wire (+)	2+
Ch2 Ground	2-

You could also follow the same configuration for other available signal sources. Lastly, connect the ADALM2000 device to your PC via the USB port.

Simple Virtual Oscilloscope

In this section, we will go through the program block per block. We will also discuss what the codes do and the reasons for how they are written. We will demonstrate additional examples in the succeeding sections where we modify this base code to show that we can add more functionalities to best fit the use case of developers.

First, import the three libraries (libm2k, matplotlib, and NumPy) that we will use to develop our virtual oscilloscope.

```
import libm2k
import matplotlib.pyplot as plt
import numpy as np
```

A uniform resource identifier (URI) is a unique identifier for each ADALM2000 connected to the PC. This block of code ensures that an ADALM2000 is connected to the PC. The code will automatically exit if there is no ADALM2000 device plugged in to your PC.

```
# Detect ADALM2000 device connected to PC
uri = libm2k.getAllContexts()
if uri == []:
    print("No ADALM2000 found. Please replug ADALM2000 device.")
    exit(1)
else:
    print("Successfully connected to ADALM2000.")
```

Connect to the ADALM2000 with the detected URI. The **uri[0]** pertains to the URI of the first ADALM2000 device detected, in case there is more than one device connected.

```
# Connect to ADALM2000 with the detected uri
adalm2000_dev = libm2k.m2kOpen(uri[0])
```

Run the calibration for ADC and DAC. This is an important step to ensure we will get accurate measurements.

```
# Run the calibration for ADC and DAC
adalm2000_dev.calibrateADC()
adalm2000_dev.calibrateDAC()
```

Set the sample rate and time duration. The available sample rates are 1 kHz, 10 kHz, 100 kHz, 1 MHz, 10 MHz, and 100 MHz. The sample rate is the number of times we get a sample in 1 second, and the duration is how long we get these samples. For example, if we set the sample rate to 1000 and duration as 3, we're going to get 1000 samples per second for 3 seconds. This makes a total of 3000 samples.

```
# Set the sample rate and time duration
sample_rate = 1000  # Hz or sample/sec
duration = 3  # seconds (time duration of our data)
```

Enable and set up Channel 1 as analog input for the oscilloscope.

```
# Enable and setup channel 1 as analog input for our oscilloscope
ocsi = adalm2000_dev.getAnalogIn()
ocsi.setSampleRate(sample_rate)
# Channel 1
ocsi.enableChannel(libm2k.ANALOG_IN_CHANNEL_1, True)
ocsi.setRange(libm2k.ANALOG_IN_CHANNEL_1, -10, 10) # range of voltage (from -10 to 10)
```

Linspace is used to create an array of evenly spaced samples. We will use this NumPy function to create the time x-axis data array. The first and second parameter of this function indicates the start and end value of the array, respectively. The last parameter is the number of samples that we want to generate within the start and end value.

In this example, the start value is 0 and the end value is the set duration, which is 3. For the number of samples, we multiply the **duration** and **sample_rate** to get the total sample that we need, which is 3000 samples. These 3000 samples will be evenly placed between 0 and 3. This array will be stored in **time_x**.

The **data_y** stores the waveform samples that we gathered using the ADALM2000 device. The samples for Channel 1 are stored in **data_y[0]**, while samples from Channel 2 are stored in **data_y[1]**. In order for us to display the accurate frequency of the waveform, we have to use the same number of samples that we used in **time_x**.

```
# x-axis data
time_x = np.linspace(0, duration, (duration * sample_rate))
# y-axis data
data_y = ocsi.getSamples(duration * sample_rate)
```

Create the figure that we will manipulate. The **plt.subplots** function will return the figure object (stored in **fig**) and the axes object (stored in **ax**), which will be used to customize the whole plot.

We can add grids that will serve as guidelines to the waveform. Add axes labels and y limit to add more details about the plot.

```
# Create the figure that we will manipulate
fig, ax = plt.subplots()
plt.plot(time_x, data_y[0])
ax.grid()
ax.set_xlabel("Time (s)")
ax.set_ylim([-4, 4])
ax.set_ylabel("Voltage")
```

Show the graph.

```
plt.show()
```

Destroy the context at the end of the code.

```
libm2k.contextClose(adalm2000_dev)
```

Run the code and expect to see a figure similar to Figure 5.

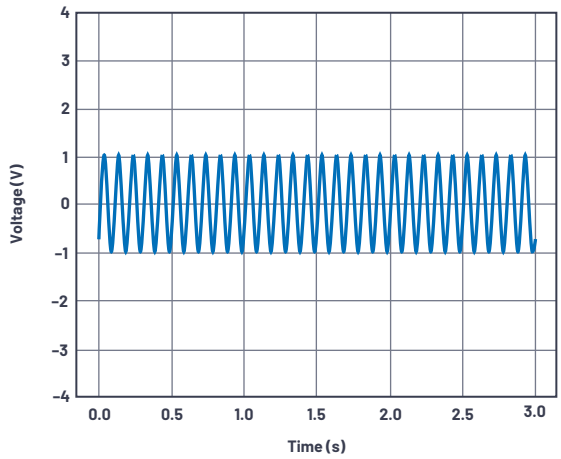

Figure 5. A single-channel sine wave output; one signal generator output: 10 Hz, 2 V p-p.

2-Channel Virtual Oscilloscope

In this section, we will use the code from the previous section and add more code blocks to make it a 2-channel virtual oscilloscope.

To add another channel, duplicate the **ocsi.enableChannel** and **ocsi.setRange** lines and change the first parameter from **libm2k.ANALOG_IN_CHANNEL_1** to **libm2k.ANALOG_IN_CHANNEL_2**.

```
# Enable and setup channel 1 and 2 as analog input for our oscilloscope
ocsi = adalm2000_dev.getAnalogIn()
ocsi.setSampleRate(sample_rate)
# Channel 1
ocsi.enableChannel(libm2k.ANALOG_IN_CHANNEL_1, True)
ocsi.setRange(libm2k.ANALOG_IN_CHANNEL_1, -10, 10) # range of voltage (from -10 to 10)
# Channel 2
ocsi.enableChannel(libm2k.ANALOG_IN_CHANNEL_2, True)
ocsi.setRange(libm2k.ANALOG_IN_CHANNEL_2, -10, 10) # range of voltage (from -10 to 10)
```

In creating the graph, add another plot for Channel 2. The data for Channel 2 is in array **data_y[1]**. We can also customize the colors of the two plots to easily distinguish them from each other. In this example, we used light coral for Channel 1 and steel blue for Channel 2.

```
# Create the figure that we will manipulate
fig, ax = plt.subplots()
plt.plot(time_x, data_y[0], color='lightcoral') # channel 1 plot
plt.plot(time_x, data_y[1], color='steelblue') # channel 2 plot
ax.grid()
ax.set_xlabel("Time (s)")
ax.set_ylim([-4, 4])
ax.set_ylabel("Voltage")
```

Run the code and you should see results similar to Figure 6.

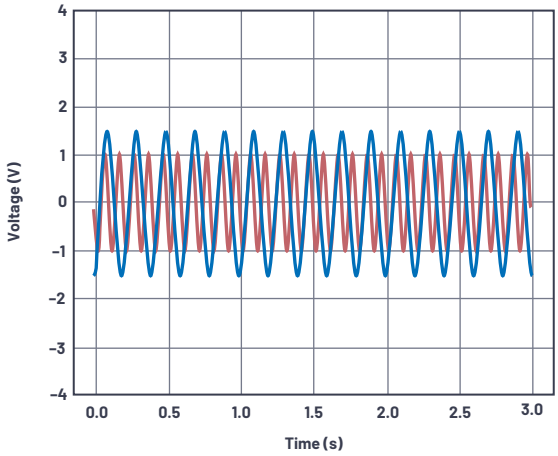

Figure 6. Two-channel sine wave output. Channel 1 signal generator output: 10 Hz, 2 V p-p; Channel 2 signal generator output: 5 Hz, 3 V p-p.

Additional Functionalities for Virtual Oscilloscope

In this section, we will add additional functionalities to our virtual oscilloscope to make it more interactive. Matplotlib provides several widgets that we could use. In this example, we will use the text label and slider widgets. We will also continue the code from the last section.

Add another import for the matplotlib slider.

```
import libm2k
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.widgets import Slider
```

Convert the time and data arrays to NumPy arrays. These will be used in the computations we will do on the next code block.

```
# x-axis data
time_x = np.linspace(0, duration, (duration * sample_rate))
# y-axis data
data_y = ocsi.getSamples(duration * sample_rate)

# Convert to numpy arrays
data_y_np1 = np.array(data_y[0]) # data from ch1
data_y_np2 = np.array(data_y[1]) # data from ch2
time_x_np = np.array(time_x) # time data for x axis
```

Since we acquired all the waveform data, what's stopping us from extracting the properties of these waveforms? In the code block below, we extracted the V_{pp}, V_{ave}, and V_{rms} from the acquired data of both channels. To compute for the V_{pp}, we added the absolute value of the maximum and minimum values found in the data_y numpy arrays. To compute for V_{ave}, we just need to divide the V_{pp} to pi. To compute for V_{rms}, we need to divide the V_{pp} to 2 multiplied to square root of 2.

```
# Compute for Vpp, Vave, and Vrms
v_pp_1 = abs(np.min(data_y_np1)) + abs(np.max(data_y_np1))
v_ave_1 = v_pp_1 / np.pi
v_rms_1 = v_pp_1 / (2 * np.sqrt(2))

v_pp_2 = abs(np.min(data_y_np2)) + abs(np.max(data_y_np2))
v_ave_2 = v_pp_2 / np.pi
v_rms_2 = v_pp_2 / (2 * np.sqrt(2))
```

This block of code is similar to the previous sections. The only difference is that we used the NumPy arrays for the plot instead of using the original arrays. We also created waveform objects from the plots. We will use these objects later.

```
# Create the figure and the waveforms that we will manipulate
fig, ax = plt.subplots()
wave1, = plt.plot(time_x_np, data_y_np1, color='lightcoral') # channel 1 plot
wave2, = plt.plot(time_x_np, data_y_np2, color='steelblue') # channel 2 plot
ax.grid()
ax.set_xlabel("Time (s)")
ax.set_ylim([-4, 4])
ax.set_ylabel("Voltage")
```

To display the computed V_{pp}, V_{ave}, and V_{rms} in the figure, we will utilize the text label widget from the matplotlib library. Create the string labels, **label_ch1** and **label_ch2**, and then concatenate the two strings to create the final label, **fin_label**. We will use the **plt.text** to create the text label. The first and second parameter **(0.2, 3)** is the x and y position of the text. The third parameter is the string to be displayed. The fourth and fifth parameters are the style of text and box, respectively.

```
# Make a text label at the top of graph to show the computed Vpp, Vave and Vrms
label_ch1 = "Channel 1 : Vpp = {:.2f}  Vave = {:.2f}  Vrms = {:.2f}".format(v_pp_1, v_ave_1, v_rms_1)
label_ch2 = "\nChannel 2 : Vpp = {:.2f}  Vave = {:.2f}  Vrms = {:.2f}".format(v_pp_2, v_ave_2, v_rms_2)
fin_label = label_ch1 + label_ch2
plt.text(0.2, 3, fin_label, style='italic', bbox={'facecolor': 'paleturquoise', 'alpha': 0.5, 'pad': 5})
```

Next, let's create the offset slider. The purpose of this slider is to adjust the reference level of the waveform. Adjust the main plot to the left to give space for the slider. The **plt.axes** defines the dimensions, position, and face color of the slider. The **Slider** function is used to create an object for the offset slider with specific properties.

```
# Adjust plot position so we can place the slider
plt.subplots_adjust(left=0.2)

# Make a vertically oriented slider to control the offset
ax_offset = plt.axes([0.05, 0.2, 0.0225, 0.63], facecolor='lemonchiffon')
offset_slider = Slider(ax=ax_offset, label="Offset", valmin=-2, valmax=2,
                       valinit=0, orientation="vertical")
```

Create **update_offset** function and register it to the **offset_slider** object. This function adds offset to the waveform every time we change the value of the slider.

```
# The function to be called anytime a slider's value changes
def update_offset(val):
    wave1.set_ydata(data_y_np1 + offset_slider.val)
    wave2.set_ydata(data_y_np2 + offset_slider.val)
    fig.canvas.draw_idle()

# Register the update_offset function with each slider
offset_slider.on_changed(update_offset)
```

Run the code and expect to see a figure similar to Figure 7.

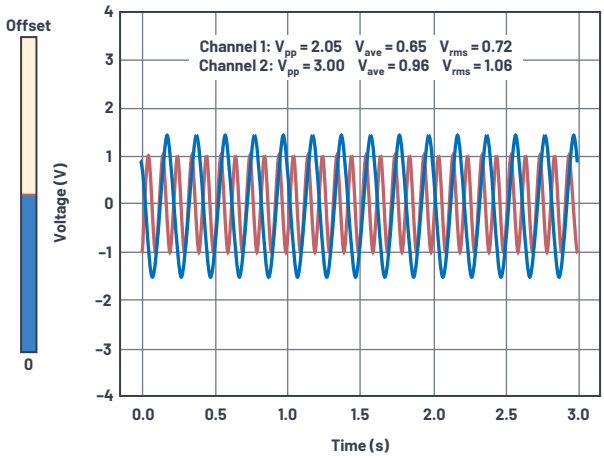

Figure 7. Default 2-channel sine wave output with offset slider.

Try to adjust the offset by using the slider. You will see the waveform move up or down in real time.

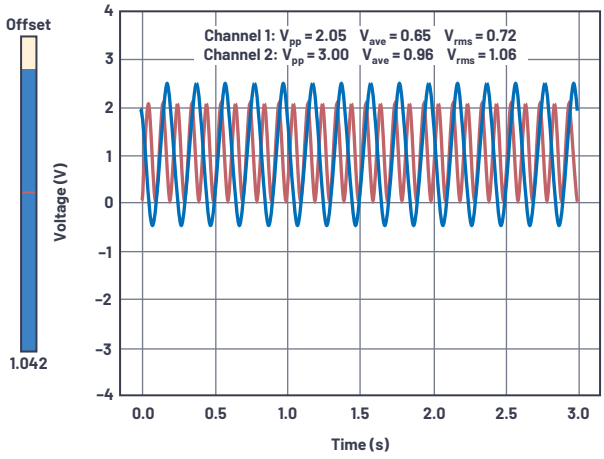

Figure 8. Adjusted offset slider (left slide) to adjust offset of both channel outputs.

Summary

This article explains the importance and convenience of having a virtual electronics laboratory. It also demonstrates how to develop a virtual oscilloscope using the ADALM2000 and Python. Software requirements and hardware setup were discussed before presenting three examples.

References

"ADALM2000 Overview." Analog Devices, March 2021.

Bhunia, Chandan, Saikat Giri, Samrat Kar, and Sudarshan Haldar. "A Low-Cost PC-Based Virtual Oscilloscope." *IEEE Transactions on Education*, Vol. 47, No. 2, May 2004.

"Limb2k Examples: analog.py." Analog Devices, Inc.

Tegen, Amelia and Wright, Jeremy. "Oscilloscopes: The Digital Alternative: The Digital Scope's Capability in Measurement, Transient Capture, and Data Storage Is a Significant Improvement Over its Analog Counterpart." *IEEE Potentials*, Vol. 2, 1983.

"What Is libm2k?" Analog Devices Wiki, April 2021.

About the Author

Arnie Mae Baes joined Analog Devices in December 2019 as a firmware engineer. During her first year, she focused on GUI and firmware development. In December 2020, she joined the Consumer Software Engineering Group and is now focusing on firmware test development. She graduated from Batangas State University with a bachelor's degree in electronics engineering. She can be reached at arniemae.baes@analog.com.

Christian Jason Garcia is a firmware verification engineer in Analog Devices General Trias, Philippines. He joined Analog Devices in November 2018 after graduating from University of Santo Tomas with a bachelor's degree in electronics and communications engineering. He specializes in software testing and system verification of SmartMesh networks in the e-Mobility Group. He can be reached at christian.garcia@analog.com.