Use Python to perform swept-sine analysis

The python open-source language can control an oscilloscope and a function generator to run frequency-response tests.

Sinay Goldberg, Ra’anana, Israel

Engineers often perform swept-sine analysis (SSA) on electronic or mechanical systems to measure frequency response. From the resulting frequency response function, engineers can calculate the system's transfer function and design suitable control systems. Engineers often use dynamic signal analyzers (DSAs) or network analyzers for these measurements, but these instruments are expensive, and have limited uses. A system consisting of an oscilloscope, a function generator, and a computer running open-source automation software form a low-cost alternative.

To measure frequency response in a mechanical system driven by a piezoelectric actuator, I used a Rigol Technologies 1064 oscilloscope and a Tekronix AFG 3022B arbitrary function generator. Both instruments connect to a PC through USB. My application program for data collection and instrument control is written in Python 2.65, an open-source language, and the pyvisa library <http://sourceforge.net/projects/pyvisa>, which is Python incarnation to VISA (Virtual Instrument Software Architecture) for controlling test instruments.

A voltage from the function generator, connected to the electrodes, excites the piezoelectric actuator (Figure 1). A vibrometer measures the DUT's mechanical movement, transforming movement speed into voltage. Channel 1 of the oscilloscope measures the vibrometer's output voltage. Channel 2, connected to the waveform-generator's output, measures the system's input signal amplitude and phase. Channel 3 measures current passing through the piezoelectric actuator.

Figure 1. An input signal excites a transducer to cause vibration in a mechanical system, which the oscilloscope measures.
Figure 2 shows the steps that the system takes to perform SSA. The code in Listing 1 sets the function generator’s excitation frequency. Similar code in Listing 2 sets the signal amplitude while the code in Listing 3 sets and scans the excitation frequency.

**Figure 2. The process involves initializing the AFG and sweeping though the frequency range.**

Collecting data from the RIGOL oscilloscope is tricky because the waveform data is returned as ASCII string of characters between header and footer. The data has 600 samples and 8-bit resolution. To overcome the vertical-resolution limit and quiet a noisy signal, you can take multiple samples and average them. The code in Listing 4 reads the captured waveform and converts the data into voltage values.

Because the measured voltage may span a few orders of magnitude, the code regularly checks that the signal isn’t too small or out of scale and automatically optimizes the oscilloscope’s the vertical scale. The code in Listing 5 auto scales the vertical-measurement axis so the measured signal will be between 25% and 100% of the full oscilloscope’s scale.

Scaling the oscilloscope’s horizontal scale lets the system achieve the maximum possible resolution because the signal’s voltage range spans more than three orders of magnitude, from 2 mV to 10 V.

Measuring wide range of frequencies make take a long time because slow signals take a long time to acquire. To shorten the measurement time I used two complementary signal processing methods:
1) For relatively fast samples (50 Hz and above), I set the oscilloscope's time base to collect 30-to-60 signal periods. The amplitude, frequency, and phase of signals from all channels was calculated using an FFT (fast Fourier transform).

2) For slow signals (2 Hz to below 50 Hz), I set the oscilloscope's time base to collect two to four periods and fitted the waveform to sine wave function. Using a python SciPy curve-fitting function, I fit a sine function to all channels.

Post processing:
The two-tier signal-processing sequence described above yields amplitude, phase, frequency and, in the case of curve fitting, DC bias for each oscilloscope channel. Extracting frequency response function or Bode plot data requires the following post-processing steps:

1) Calculate the signal intensity by dividing the system output signal amplitude (vibrometer or current) by the input signal.
2) Calculate the relative phase by subtracting system input signal phase (vibrometer or current) from the output signal phase.
3) Compare of the input and output signals frequencies to confirm the quality of measurement.

Collecting many periods of slow signal, for example 50 periods of 10 Hz signal takes 5 s. Unfortunately, using an FFT at those frequencies is very slow. For signals at frequencies of 50 kHz and higher, the FFT is a must.

The code in Listing 6 sets the scope time base for up to 60 periods. The code uses the excitation frequency, and set the time base accordingly. The factor 6.0 is derived from 60 periods divided by 10 grid lines. The oscilloscope rounds the calculated time base is “rounded” to the next existing slower time scale.

With the above procedures in place, the system runs the code in Listing 7 to sweep the waveform-generator's frequency and to collect relative amplitude and phase for each frequency. The software then saves the data to a text or CSV (comma-separated variable) file for offline analysis and graphic representation.

I've also used Python and in applications such as:

- Acquiring data with National Instruments data-acquisition cards, both PCI and USB. Using A/D card enable collection of waveforms with 12-bit to 16-bit resolution and 4096 up to $2^{20}$ points per measurement compared to 8 bit, 600 points on the oscilloscope. That lets me make high resolution, very low-frequency measurements,
- Switching a Rigol power supply to control peripheral devices,
- Sending e-mail notifications with measurement results when a sweep is completed or in case of an error. Such a feature let you get back to your work while the system measures, and
- Taking images and video of the apparatus using webcam during measurement.

The system has some limitations. First, Python and VISA don't operate in real time, so expect a delay of about 50 ms from the time the computer issues a command until it executes. Furthermore, long USB cables used to connect devices to host computer make the system susceptible to electromagnetic noise, which can create errors that crashe the pyvisa driver. Take care during programming and running your code to handle errors resulting from device
communication errors. As a precaution, place the USB cables away from fluorescent lighting and high-voltage instruments.

**Acknowledgement**

Thanks to Python open source community for creating an easy and straightforward programming environment. Special thanks to the VISA, numpy, and scipy groups for excellent functionality. I’m also grateful to the developers of pydaqtools and videocapture for very helpful modules. Your contributions made my job a lot easier, fun and successful.

Comments on data acquisition in python versus LabView can be found at http://www.japh.de/blog/python-for-measurement-and-data-acquisition/.
Listing 1: Code to connect to set the excitation frequency for the AFG:

```python
import visa
#-------------------------------------------------------------------
def set_afg_freq(freq): ## set afg freq.
    try :
        afg = visa.instrument('USB0::0x0699::0x0347::C031273')
        ret = True
    except :
        print('Could not find AFG')
        ret = False
    afg.write("SOURce1:FREQuency:CW "+str(freq)+" Hz")
    afg.write("Source2:FREQuency:CW "+str(freq)+" Hz")
    return ret
```

Listing 2

Similar code is used to set the amplitude.

```python
import visa
#-------------------------------------------------------------------
def set_afg_amp(amp): ## set afg amplitude.
    try :
        afg = visa.instrument('USB0::0x0699::0x0347::C031273')
        ret = True
    except :
        print('Could not find AFG')
        ret = False
    afg.write("SOURce1:VOLTAGE:AMPLITUDE "+str(amp[0]))
    afg.write("Source2:VOLTAGE:AMPLITUDE "+str(amp[1]))
    return ret
```

Listing 3: The following code is used to set and scan the excitation frequency

```python
for freq in range(100,250000,100):
    if set_afg_freq(freq):
        # read data from scope
```

Listing 4: To overcome the resolution limit one can take multiple samples and average. Reading a waveform is done with the following
def get_waveform(ch=1): # default channel = 1
    scope = visa.instrument('USB0::0x1AB1::0x0488::DS1BE122400144')
    scope.write("CHANnel"+str(ch)+"::SCALe?") # get vertical scale
    curr_scale = float(scope.read())
    scope.write("::WAVeform:DATA? CHANnel"+str(ch)+"\n") # read waveform
    char_wf = scope.read()
    char_wf = char_wf[:-1] # remove footer
    int_wf = [] # integer values
    for i in range(len(char_wf)): # convert char array to integer
        int_wf.append(100-ord(char_wf[i])) # scope vertical full scale is +-100 pix
    scale = curr_scale/25 # 25 pixels between 2 grid lines
    wf = []
    j=11 # ignore the header
    while j<len(char_wf): # convert to voltage values
        wf.append(int_wf[j]*scale) #
        j=j+1
    return(wf)

Listing 5: The following code was used to auto scale the vertical measurement axis.

def Auto_Vscl(ch=1): # auto scale scope vertical axis
    scope = visa.instrument('USB0::0x1AB1::0x0488::DS1BE122400144')
    Vscales = [0.002, 0.005, 0.01,
               0.02, 0.05, 0.1,
               0.2, 0.5, 1.0,
               2.0, 5.0, 10.0,
               20.0, 50.0, 100.0]
    ChS = ":CHANnel"+str(ch)
    Vpp = get_ch_Vpp(ch) # current signal span V peak to peak
    scope.write(ChS+"::SCALe?")
    curr_scale = float(scope.read()) # get the vertical scale
    while (Vpp > curr_scale*8.0) : # check if the signal is too high
        new_scale = Vscales[Vscales.index(curr_scale)+1] #
        scope.write(ChS+"::SCALe "+str(new_scale))
        time.sleep(1.0)
        curr_scale = new_scale
        Vpp = get_ch_Vpp(ch)
    while (Vpp < curr_scale*2.0): # check if the signal is too low
        if curr_scale > 0.002:
            new_scale = Vscales[Vscales.index(curr_scale)-1]
            scope.write(ChS+"::SCALe "+str(new_scale))
time.sleep(0.2)
curr_scale = new_scale
Vpp = get_ch_Vpp(ch)
else:
    Vpp = 5*c curr_scale

**Listing 6:** The following lines sets the scope timebase to have up to 60 periods. The algorithm takes the excitation frequency, and set the time base accordingly. The factor 6.0 is derived from 60 periods divided by 10 grid lines. The calculated time base is “rounded” by the scope to the next lower time scale.

```
    if 6.0/freq > curr_tb:
        curr_tb = 1/freq
        scope1.set_scope_freq(freq)
```

Measurement:
With the above procedures in place, frequency sweep is run using the following code:

**Listing 7**

```
for freq in range(100, 250000, 100):
    if 6.0/freq > curr_tb: # set scope time base
        curr_tb = 1/freq
        scope.set_scope_freq(freq) # set the timebase
    if set_afg_freq(freq): # set excitation frequency
        scope.Auto_Vscal(1) # set the vertical scope scale
        excit_data = scope.get_waveform (1) # Read waveform data
        scope.Auto_Vscal(2)
        out_data = scope.get_waveform (2)
        scope.Auto_Vscal(3)
        current_dat = scope.get_waveform (3)
```