```
#!/usr/bin/env python

try:
    import tkinter as tk
except:
    import Tkinter as tk
import pygubu
import matplotlib
from tkinter import messagebox
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from matplotlib.figure import Figure
from bode import Bode

class GUI:
    def __init__(self, master):

        self.builder = builder = pygubu.Builder()
        builder.add_from_file("GUI.ui")
        master.title("USB Gain/Phase Analyzer")
        #master.rowconfigure(0, weight=1)
        #master.columnconfigure(0, weight=1)
        self.top = builder.get_object("top", master)

        #Set up entry boxes
        self.min_var = tk.DoubleVar()
        self.max_var = tk.DoubleVar()
        self.min_entry = tk.Entry(self.top, name="entry_min",textvariable=self.min_var, w
idth=10)
        self.min_entry.grid(column=1,row=1)
        self.max_entry = tk.Entry(self.top, name="entry_max", textvariable=self.max_var,w
idth=10)
        self.max_entry.grid(column=1,row=2)

        #set up canvas to hold plots
        self.freq_fig = Figure()
        self.phase_fig = Figure()
        self.freq_canvas = FigureCanvasTkAgg(self.freq_fig, master)
        self.phase_canvas = FigureCanvasTkAgg(self.phase_fig, master)

        #connect callbacks
        builder.connect_callbacks(self)
        self.calibrate = False

        self.bode = Bode()

    def on_plot(self):
        self.freq_fig.clear()
        self.phase_fig.clear()
        phase = self.builder.get_variable('do_phase').get()
        linear = self.builder.get_variable('do_linear').get()


        #If we have not calibrated, then we need to make sure that these
        #Bode attributes have been set
        if not self.calibrate:
            lower_bound = self.min_var.get()
            upper_bound = self.max_var.get()
            # need error here
            #while lower_bound != upper_bound:
            if lower_bound >= upper_bound:
                error = tk.messagebox.showerror("error","These values are not valid.\n Tr
y again min must be lower than max" )
            #       break
            self.bode.set_lower_bound(lower_bound)
            self.bode.set_upper_bound(upper_bound)
            self.bode.set_do_phase(phase)
            self.bode.generate_freqs()

        #Have bode generate data
        self.bode.run()
```

```
        #Get frequency response data from bode.
        freq_response = self.bode.get_freq_response_data()
        if self.calibrate:
            freq_plt = self.freq_fig.add_subplot(111, xlabel="Frequency", ylabel="Gain, c
alibrated")
            freq_calibrate_data = self.bode.get_freq_calibration_data()
            assert len(freq_calibrate_data) == len(freq_response), (len(freq_calibrate_da
ta), len(freq_response))
            for i in range(len(freq_response)):
                freq_response[i] = freq_response[i] - freq_calibrate_data[i]
        else:
            freq_plt = self.freq_fig.add_subplot(111, xlabel="Frequency", ylabel="Gain, u
ncalibrated")

        #Generate frequency response plot
        freqs_f = self.bode.get_freqs_f()
        if not linear:
            freq_plt.semilogx(freqs_f, freq_response)
        else:
            freq_plt.plot(freqs_f, freq_response)
        freq_plt.grid(True)
        self.freq_canvas.show()
        self.freq_canvas.get_tk_widget().place(relx=.26, rely=0.01,relheight=0.49,relwidt
h=0.7)

        #If we want phase, we go through the process again
        if phase:
            phase_response = self.bode.get_phase_response_data()
            if self.calibrate:
                phase_plt = self.phase_fig.add_subplot(111, xlabel="Frequency", ylabel="P
hase, calibrated")
                phase_calibrate_data = self.bode.get_phase_calibration_data()
                for i in range(len(phase_response)):
                    phase_response[i] = phase_response[i] - phase_calibrate_data[i]
            else:
                phase_plt = self.phase_fig.add_subplot(111, xlabel="Frequency", ylabel="P
hase, uncalibrated")

            freqs_p = self.bode.get_freqs_p()

            if not linear:
                phase_plt.semilogx(freqs_p, phase_response)
            else:
                phase_plt.plot(freqs_p, phase_response)

            phase_plt.grid(True)
            self.phase_canvas.show()
            self.phase_canvas.get_tk_widget().place(relx=.26,rely=0.505,relheight=0.49,re
lwidth=0.7)

    def on_calibrate(self):
        #set lower and upper bounds for bode
        lower_bound = self.min_var.get()
        upper_bound = self.max_var.get()
        self.bode.set_lower_bound(lower_bound)
        self.bode.set_upper_bound(upper_bound)
        #need error box here
        #while lower_bound != upper_bound:
        if lower_bound >= upper_bound:
            error = tk.messagebox.showerror("error","These values are not valid.\n Try ag
ain min must be lower than max")
        #        break
        #tell bode whether or not it will do a phase plot
        phase = self.builder.get_variable('do_phase').get()
        self.bode.set_do_phase(phase)

        #generate the frequencies to sample across
        self.bode.generate_freqs()
```

```
        #have bode perform calibration
        self.bode.calibrate()
        self.calibrate = True

if __name__ == "__main__":
    root = tk.Tk()
    gui = GUI(root)
    root.mainloop()
```

```
from serial_comm import *
from response import *
import numpy as np

class Bode:
    def __init__(self):
        self.s = connect_gpa()
        mc_init(self.s)
        synth_init(self.s)
        frontend_init(self.s)
        self.freq_calibration = []
        self.phase_calibration = []
        self.freq_response = []
        self.phase_response = []
        self.freqs_f = []
        self.freqs_p = []
        self.upper_bound = 0.0
        self.lower_bound = 0.0
        self.do_phase = False

    #set the upper bound of sampling frequencies
    def set_upper_bound(self, upper_bound):
        self.upper_bound = upper_bound

    #sets lower bound of sampling frequencies
    def set_lower_bound(self, lower_bound):
        self.lower_bound = lower_bound

    def set_do_phase(self, do_phase):
        self.do_phase = do_phase

    def get_freq_calibration_data(self):
        return self.freq_calibration

    def get_phase_calibration_data(self):
        return self.phase_calibration

    def get_freq_response_data(self):
        return self.freq_response

    def get_phase_response_data(self):
        return self.phase_response

    #generate lists of all sampling frequencies
    def generate_freqs(self):
        self.freqs_f = np.logspace(np.log10(self.lower_bound), np.log10(self.upper_bound)
, 60) # 1 kHz to 150 MHz
        if self.do_phase:
            self.freqs_p = np.logspace(np.log10(self.lower_bound), np.log10(self.upper_bo
und), 30) # 1 kHz to 150 MHz

    def get_freqs_f(self):
        return self.freqs_f

    def get_freqs_p(self):
        return self.freqs_p

    def calibrate(self):
        self.freq_calibration = get_freq_response(self.s, self.freqs_f)
        if self.do_phase:
            self.phase_calibration = get_phase_response(self.s, self.freqs_p)

    def run(self):
        self.freq_response = get_freq_response(self.s, self.freqs_f)
        if self.do_phase:
            self.phase_response = get_phase_response(self.s, self.freqs_p)
```

```
#!/usr/bin/python
import serial
import time
import sys
from serial_comm import getline
import serial_comm as defs

# The first few samples come out wrong. Repeat them
N_REPEAT = 6

def get_freq_response(s, freqs):
    print ("Collecting data... ")
    data = []

    freqs = list(freqs[:N_REPEAT]) + list(freqs)

    for i in freqs:
        nSamples = max(((1/i)*50)//1000000, 2048)
        s.write (("T:FREQ %d, %f\r\n" % (defs.CH_MAIN, i)).encode ('ascii'))
        getline (s)
        s.write(b"LOW:CLR GPIO_ATTEN\r\n")
        time.sleep(.005)
        s.write (("T:SAM %d\r\n" % nSamples).encode ('ascii'))
        level = float (getline (s))

        #if level >= 2900, attenuate the input signal
        if level >= 3500:
            print("this is running...")
            s.write(b"LOW:SET GPIO_ATTEN\r\n")
            time.sleep(.005)
            s.write (("T:SAM %d\r\n" % nSamples).encode ('ascii'))
            level = float (getline (s))
            db = level / (4095 * 24e-3 / 3.3) + 15
        else:
            db = level / (4095 * 24e-3 / 3.3)

        print ("%.2f Hz\t%.2f dB" % (i, db))
        data.append (db)

    # Remove repeated measurements
    data = data[N_REPEAT:]
    #data = [i-data[0] for i in data]
    return data


def get_phase_response(s, freqs):
    print ("Collecting phase data...")
    N_POINTS_PER_RANGE = 16
    PRECISION = 1.
    data = []
    freqs = list(freqs[:N_REPEAT]) + list(freqs)
    for i in freqs:
        # Set frequency. Then, search phases for a null
        s.write (("T:FREQ %d, %f\r\n" % (defs.CH_PHASE, i)).encode ('ascii'))
        getline (s)
        s.write (("T:FREQ %d, %f\r\n" % (defs.CH_MAIN, i)).encode ('ascii'))
        getline (s)
        s.write (("T:AMP %d, 1.0\r\n" % (defs.CH_PHASE)).encode ('ascii'))
        getline (s)
        s.write(b"LOW:SET GPIO_ATTEN\r\n")
        s.write (("T:AMP %d, 0.178\r\n" % (defs.CH_MAIN)).encode ('ascii'))
        getline (s)

        phase_bound_left = 0
        phase_bound_right = 360
        while phase_bound_right - phase_bound_left > PRECISION:
            width = phase_bound_right - phase_bound_left
            pitch = width / N_POINTS_PER_RANGE
            phases = [(i * pitch) + phase_bound_left for i in range (N_POINTS_PER_RANGE)]
            lowest_phase = None
```

```
            lowest_amp = float("inf")
            lowest_i = None
            for phase_i, phase in enumerate(phases):
                nSamples = max(((1/i)*50)//1000000, 2048)
                phase = phase % 360.
                s.write (("T:PHASE %d, %f\r\n" % (defs.CH_PHASE, phase)).encode ('ascii')
)
                getline (s)
                s.write (("T:SAM %d\r\n" % nSamples).encode ('ascii'))
                level = float (getline (s))
                if level < lowest_amp:
                    lowest_amp = level
                    lowest_phase = phase
                    lowest_i = phase_i
                print (".", end='')
                sys.stdout.flush ()
            if lowest_i == 0:
                # Slide the range left
                width = phase_bound_right - phase_bound_left
                phase_bound_left -= width / 2
                phase_bound_right -= width / 2
            elif lowest_i == len(phases) - 1:
                # Slide the range right
                width = phase_bound_right - phase_bound_left
                phase_bound_left += width / 2
                phase_bound_right += width / 2
            else:
                # Narrow the range
                phase_bound_left = phases[lowest_i - 1]
                phase_bound_right = phases[lowest_i + 1]

        phases = list (range (0, 360, 40))
        lowest_phase -= 180.
        phase = lowest_phase

        # Smooth phase discontinuities
        if len(data):
            last_phase = data[-1]
            offset_phases = [phase + (360. * i) for i in range(-3,4)]
            phase_errors = [abs(i - last_phase) for i in offset_phases]

            best_phase, best_error = min (zip (offset_phases, phase_errors), key=lambda x
: x[1])
            phase = best_phase


        print ("%.2f Hz\t%f deg" % (i, phase))
        data.append (phase)
    data = data[N_REPEAT:]
    return data
```

```
#!/usr/bin/python
import serial
import time
import os

CH_MAIN = 0
CH_PHASE = 1
CH_IN_1 = 1
CH_IN_2 = 0

USB_ID = "1209:4757"


def getline (s):
    return s.readline ().decode ('ascii').strip ()

def printline (s, indent=2):
    """Print a line from serial, indented."""
    line = getline (s)
    print ((" " * indent) + line)
    return line

def read(fn):
    with open (fn) as f:
        return f.read().strip()

def usb_id(path):
    if not os.path.isfile (path + "/idProduct"):
        return None
    if not os.path.isfile (path + "/idVendor"):
        return None
    product = read(path + "/idProduct")
    vendor = read(path + "/idVendor")
    return vendor.lower() + ":" + product.lower()

def get_tty(path):
    for subdir in os.listdir(path):
        if not os.path.isdir(path + "/" + subdir):
            continue
        if os.path.isdir(path + "/" + subdir + "/tty"):
            this_subdir = subdir
            break
    return "/dev/" + os.listdir(os.path.join(path, subdir, "tty"))[0]


def find_device():
    DEVS = "/sys/bus/usb/devices"
    for dev_dir in os.listdir(DEVS):
        path = DEVS + "/" + dev_dir
        if usb_id(path) == USB_ID:
            return DEVS + "/" + dev_dir

def connect_gpa():
    devnode = find_device()
    if devnode is None:
        raise Exception("No GPA found!")
    tty = get_tty(devnode)
    print ("Connecting to GPA...")
    return serial.Serial (tty, 1, timeout=1)

def mc_init(s):
    print ("Initializing microcontroller...")
    s.write (b"*IDN?\r\n")
    idnstr = getline (s)
    print ("  identity string = " + idnstr)
    while not idnstr.startswith ("WCP52"):
        print ("  bad response, retrying")
        s.write (b"\r\n\r\n\r\n")
        time.sleep (0.5)
        s.flushOutput ()
```

```
        s.flushInput ()
        s.write (b"*IDN?\r\n")
        idnstr = getline (s)
        print ("  identity string = " + idnstr)
    print ("  OK")

def synth_init(s):
    print ("Initializing synthesizer...")
    s.write (b"T:INIF\r\n")
    s.write (b"*OPC?\r\n")
    getline (s)
    time.sleep (0.25)
    s.write (b"T:INCK\r\n")
    s.write (b"*OPC?\r\n")
    getline (s)
    time.sleep (0.25)
    print ("  OK")

def frontend_init(s):
    print ("Initializing frontend... ")
    s.write (("T:FREQ %d, 0\r\n" % (CH_PHASE)).encode('ascii'))
    printline (s)
    s.write (("T:AMP %d, 0\r\n" % (CH_PHASE)).encode('ascii'))
    printline (s)
    s.write (("T:CH %d\r\n" % (CH_IN_1)).encode('ascii'))
    s.write (b"*OPC?\r\n")
    getline (s)
    print ("  OK")
```