Table of Contents

# spectrometer.py

# Set DEBUG to True to produce output while running to help DEBUGging  
DEBUG = False  
SHOW\_TEST\_BUTTON = True  
  
######################################################################  
## import required packages  
######################################################################  
  
import PySimpleGUI as sg  
import numpy as np  
import os  
import sensor  
import model  
import table  
import usb  
  
######################################################################  
## Setting display parameters  
######################################################################  
  
sg.theme('DefaultNoMoreNagging')  
sg.set\_options(font=('latin modern roman', 16, 'bold'))  
  
def debugprint(x):  
 if DEBUG:  
 print(x)  
  
######################################################################  
## Set initial values for global variables  
######################################################################  
  
measured\_samples = [] #[sample id, volts, absorbance, concentration]  
  
# Set spectro to Sensor object to access Spectrophotometer through Raspberry Pi  
spectro = sensor.Sensor('@Volts')  
  
# Initialize model  
model = model.Model('@calibration-plot', '@measuring-plot')  
  
# functions to call given key events  
key\_function = {}  
  
######################################################################  
#### Layouts and associated functions  
######################################################################  
  
# Main Screen  
# Layouts  
  
heading = [[sg.Text("Seth's Spectacular Spectrophotometer", size=(50, 1), justification='center',  
 font=('latin modern roman', 24, 'bold'))],  
 [sg.Text('Charles W. Anderson, Seth H. Frisbie, Erika J. Mitchell, and Kenneth R. Sikora',  
 size=(80, 1),  
 justification='right', font=('latin modern roman', 12, 'bold'))],  
 [sg.Text('\"It\'s the questions that drive us, Mr. Anderson" -- Agent Smith (The Matrix)',  
 size=(75, 1),  
 justification='right', font=('latin modern roman', 14, 'italic'))],  
 [sg.Text('Spectrophotometer is not connected. Random voltage values will be used.',  
 key='@random-volts-used', visible=False, justification='center',  
 size=(75, 1), font=('latin modern roman', 16, 'bold'), text\_color='red')]]  
  
layout\_voltage = [[sg.Column([[sg.Frame('', [  
 [sg.Text(size=(8, 2),  
 justification='right', key='@Volts'),  
 sg.Text('binary volts', size=(15, 2),   
 justification='left')]], size=(340, 50))]])]]  
  
######################################################################  
### Calibration Panel 1, choose expected equation  
  
# Layouts  
  
all\_calibration\_layouts = []  
all\_calibration\_keys = []  
  
panel\_calibration\_equation\_form = [[sg.pin(sg.Column([  
 [sg.Text('Do you expect a linear or quadratic calibration equation?')],  
 [sg.Radio('Linear', 'CalForm', default=False, key='@linear', enable\_events=True)],  
 [sg.Radio('Quadratic', 'CalForm', default=False, key='@quadratic', enable\_events=True)],  
 [sg.Text('', key='@n-samples-needed', visible=True)],  
 [sg.Text('Put the zero concentration standard in the holder.\n'  
 'Use the potentiometers on the spectrophotometer to carefully\n'  
 'adjust the voltage to approximately 1,000 binary volts.\n'  
 'Click OK when this is done.',  
 key='@put-zero-standard', visible=False),  
 sg.Button('Ok', key='@zero-conc-ok', visible=False)]],  
 visible=True,  
 key='@cal-equation-form'))]]  
  
all\_calibration\_layouts += panel\_calibration\_equation\_form  
all\_calibration\_keys.append('@cal-equation-form')  
  
# Functions  
  
want\_linear\_calibration = False  
want\_quadratic\_calibration = False  
   
def equation\_form\_chosen(form):  
 global want\_linear\_calibration, want\_quadratic\_calibration  
   
 want\_linear\_calibration = want\_quadratic\_calibration = False  
  
 if form == 'linear':  
 window['@n-samples-needed'].update('You will need a minimum of 4 calibration standards,\n'  
 'including the blank, for a statistical line.')  
 want\_linear\_calibration = True  
 else:  
 window['@n-samples-needed'].update('You will need a minimum of 5 calibration standards,\n'  
 'including the blank, for a statistical line.')  
 want\_quadratic\_calibration = True  
 window['@put-zero-standard'].update(visible=True)  
 window['@zero-conc-ok'].update(visible=True)  
   
key\_function['@linear'] = lambda event\_not\_used, values\_not\_used: equation\_form\_chosen('linear')  
key\_function['@quadratic'] = lambda event\_not\_used, values\_not\_used: equation\_form\_chosen('quadratic')  
  
def zero\_conc\_ok(event, values\_not\_used):  
 volts = spectro.read\_multiple\_voltages()  
 spectro.set\_volts\_zero\_concentration(volts)  
  
 model.clear\_calibration\_samples()  
 model.add\_calibration\_sample(0, volts, spectro.volts\_to\_absorbance(volts))  
 calibration\_samples\_table.set\_samples(model.samples)  
 make\_calibration\_layout\_visible('@cal2')  
 window['@record-cal'].update(disabled=False)  
 calibration\_samples\_table.refresh()  
  
  
key\_function['@zero-conc-ok'] = zero\_conc\_ok  
  
######################################################################  
### Calibration Panel 2, collecting samples  
  
# Layouts  
  
calibration\_samples\_table = table.Table('@cal-table',  
 [],   
 ('Concentration', 'Binary Volts', 'Absorbance'),  
 spectro)  
  
panel\_calibration\_samples = [[sg.pin(sg.Column([  
 [sg.Column([[sg.Button('Load Test Samples', key='@load-samples'),  
 sg.Text('to test with predefined samples.')]],  
 visible=True)],   
 [sg.Text('Place a new standard in the holder.')],  
 [sg.Text('What is the concentration of this standard?'),  
  
  
 sg.Input(key='@concentration', size=(10, 2)),  
 sg.Text(' ', key='@concentration-error')],  
 [sg.Column([  
 [sg.Text(' '),  
 sg.Button('Record', key='@record-cal', disabled=True),  
 sg.Button('Done Recording', key='@done-recording-cal', disabled=True)]],  
 element\_justification='right', expand\_x=True)],  
 [sg.Text('', key='@nmore')],  
 [calibration\_samples\_table.make\_sgTable()]], visible=False, key='@cal2'))]]  
  
all\_calibration\_layouts += panel\_calibration\_samples  
all\_calibration\_keys.append('@cal2')  
  
# Functions  
  
def make\_calibration\_layout\_visible(visible\_layout\_key):  
 for layout\_key in all\_calibration\_keys:  
 debugprint('layout\_key={}'.format(layout\_key))  
 window[layout\_key].update(visible=False)  
 window[visible\_layout\_key].update(visible=True)  
  
### Add ability to preload testing data for calibration samples.  
  
simulate\_trend = None  
  
# Functions  
  
def set\_trend\_test(key, values\_not\_used):  
 global simulate\_trend  
 simulate\_trend = key[1:key.index('-test')]  
  
def load\_samples(key, values\_not\_used):  
 global window\_load\_samples  
 layout\_select\_testing\_data = [  
 [sg.Text('TESTING. Choose test data to load.')],  
 [sg.Radio('linear no intercept', 'TrendForm', default=False,  
 key='@linear-no-intercept-test', enable\_events=True),  
 sg.Radio('linear with intercept', 'TrendForm', default=False,  
 key='@linear-intercept-test', enable\_events=True)],  
 [sg.Radio('quadratic no intercept', 'TrendForm', default=False,  
 key='@quadratic-no-intercept-test', enable\_events=True),  
 sg.Radio('quadratic with intercept', 'TrendForm', default=False,  
 key='@quadratic-intercept-test', enable\_events=True)],  
 [sg.Radio('no trend', 'TrendForm', default=False,  
 key='@no-trend-test', enable\_events=True),  
 sg.Radio('cubic', 'TrendForm', default=False,  
 key='@cubic-test', enable\_events=True)],  
 [sg.Button('Done', key='@done-loading-samples')]]  
 window\_load\_samples = sg.Window("Load Test Samples", layout\_select\_testing\_data,  
 finalize=True)  
  
def done\_loading\_samples(key, values):  
 global window\_load\_samples  
 test\_data = [k for (k, v) in values.items() if v][0]  
 # Trim leading @ and trailing '-test'  
 test\_data = test\_data[1:test\_data.index('-test')]  
 model.set\_calibration\_samples\_for\_testing(test\_data)  
 calibration\_samples\_table.set\_samples(model.samples)  
 spectro.set\_volts\_zero\_concentration(model.samples[0][1])  
 calibration\_samples\_table.refresh()   
 window\_load\_samples.close()  
 main\_window['@done-recording-cal'].update(disabled=False)  
   
key\_function['@no-trend-test'] = set\_trend\_test  
key\_function['@linear-no-intercept-test'] = set\_trend\_test  
key\_function['@linear-intercept-test'] = set\_trend\_test  
key\_function['@quadratic-no-intercept-test'] = set\_trend\_test  
key\_function['@quadratic-intercept-test'] = set\_trend\_test  
key\_function['@cubic-test'] = set\_trend\_test  
key\_function['@done-loading-samples'] = done\_loading\_samples  
key\_function['@load-samples'] = load\_samples  
  
### End of testing data loading  
  
## Check for valid numerical input in text boxes  
  
def check\_input(input\_key, convert\_f, error\_key, error\_message):  
 inputbox\_text = values[input\_key]  
 errorbox = window[error\_key]  
 if len(inputbox\_text) > 0:  
 try:  
 n\_measurements = convert\_f(inputbox\_text)  
 errorbox.update('')  
 return n\_measurements  
 except:  
 errorbox.update(error\_message, text\_color='red')  
 return None  
  
def record\_calibration\_sample(event, values\_not\_used):  
 # global recording\_calibration\_samples  
  
 # recording\_calibration\_samples = True  
 conc = check\_input('@concentration', float, '@concentration-error', 'Must be a number.')   
 if conc and event == '@record-cal':  
 # Read mean of multiple voltages  
 mean\_volts = spectro.read\_multiple\_voltages()  
 # Add sample to calibration\_samples\_table  
 model.add\_calibration\_sample(conc, mean\_volts, spectro.volts\_to\_absorbance(mean\_volts))  
 calibration\_samples\_table.set\_samples(model.samples)  
 calibration\_samples\_table.refresh()  
 n\_calibration\_recordings = len(model.samples)  
 # Enable the "Done Recording" button if enough calibration samples  
 # Clear the concentration field window  
 window['@concentration']('')  
 # have been collected for   
 if ((want\_quadratic\_calibration and n\_calibration\_recordings >= 5)  
 or  
 (want\_linear\_calibration and n\_calibration\_recordings >= 4)):  
 window['@done-recording-cal'].update(disabled=False)  
 if want\_quadratic\_calibration:  
 window['@nmore'].update('You need a minimum of {} more calibration standards for a '  
 'statistical curve.'.format(max(5 - n\_calibration\_recordings, 0)))  
 elif want\_linear\_calibration:  
 window['@nmore'].update('You need a minimum of {} more calibration standards for a '  
 'statistical curve.'.format(max(4 - n\_calibration\_recordings, 0)))  
  
key\_function['@record-cal'] = record\_calibration\_sample  
  
def done\_recording(event, values\_not\_used):  
 # recording\_calibration\_samples = False  
 calibration\_samples = np.array(model.samples) # np.array(calibration\_samples)  
 debugprint(calibration\_samples)  
  
 ### Algorithm for checking trends and y intercepts.  
   
 # If quadratic wanted:  
 # Test for cubic trend.  
 # If cubic found:  
 # "Please recalibrate"  
 # Else cubic not found:  
 # Test for quadratic.  
 # If quadratic found:  
 # Test for y-intercept non-zero.  
 # If y-intercept is non-zero;  
 # Ask "Use or recalibrate?"  
 # Else y-intercept is zero:  
 # Use this calibration and move to "Measure Concentration".  
 # Else quadratic not found:  
 # Test for linear.  
 # If linear found:  
 # Test for y-intercept non-zero.  
 # if y-intercept is non-zero:  
 # Ask "Use or Recalibrate?"  
 # Else y-intercept is zero:  
 # Ask "Use or Recalibrate?"  
 # Else linear not found:  
 # "Please recalibrate"  
  
 # If linear wanted:  
 # Test for quadratic.  
 # If quadratic found:  
 # "Please recalibrate."  
 # Else quadratic not found:  
 # Test for linear.  
 # If linear found:  
 # Test for y-intercept non-zero.  
 # if y-intercept is non-zero:  
 # Ask "Use or Recalibrate?"  
 # Else y-intercept is zero:  
 # Ask "Use or Recalibrate?"  
 # (previous step here was: Move to "Measure Concentration")  
 # Else linear not found:  
 # "Please recalibrate"  
  
 if want\_quadratic\_calibration:  
  
 debugprint('WANT QUADRATIC')  
  
 if model.significant\_cubic(): # calls model.train([0, 1, 2, 3])  
  
 debugprint('cubic found')  
 model.update\_calibration\_plot()  
 ask\_to\_recalibrate(  
 'Your calibration equation has a signficant cubic trend.') #. Please recalibrate')  
  
 else:  
  
 if model.significant\_quadratic(): # calls model.train([0, 1, 2])  
  
 debugprint('quadratic found')  
 if model.significant\_y\_intercept():  
 debugprint('y\_intercept found')  
 model.update\_calibration\_plot()  
 ask\_use\_or\_recalibrate(  
 'Your calibration equation does not go through the origin (0,0);\n'  
 'more specifically, it has a statistically significant y-intercept\n'  
 'at the 95% confidence level.')  
 else:  
 debugprint('no y intercept')  
 model.update\_calibration\_plot()  
 ask\_use\_or\_recalibrate(  
 'Your calibration equation has a significant quadratic trend and the y-intercept is zero.')  
  
 elif model.significant\_linear(): # calls model.train([0, 1])  
  
 debugprint('linear found')  
 if model.significant\_y\_intercept():  
 debugprint('y\_intercept found')  
 model.update\_calibration\_plot()  
 ask\_use\_or\_recalibrate(  
 'Your calibration equation is linear and does not go through the origin (0,0).\n'  
 'More specifically, your calibration does not have a statistically significant\n'  
 'quadratic trend at the 95% confidence level; however it does have a statistically\n'  
 'significant linear trend at the 95% confidence level. And it has a statistically\n'  
 'significant y-intercept at 95% confidence level.')  
 else:  
  
 debugprint('no y\_intercept')  
 model.update\_calibration\_plot()  
 ask\_use\_or\_recalibrate(  
 'Your calibration equation is linear, not quadratic. More specifically, your calibration\n'  
 'does not have a statistically significant quadratic trend at the 95% confidence level;\n'  
 'however, it does have a statistically significant linear trend at the 95% confidence level.')  
 else:  
  
 debugprint('no trend found')  
 model.update\_calibration\_plot()  
 ask\_to\_recalibrate(  
 'Your calibration equation does not have a significant trend.') # Please recalibrate')  
  
 if want\_linear\_calibration:  
 debugprint('WANT LINEAR')  
 if model.significant\_quadratic(): # calls model.train([0, 1, 2])  
  
 debugprint('quadratic found')  
 model.update\_calibration\_plot()  
 ask\_to\_recalibrate(  
 'Your calibration equation has a signficant quadratic trend.') # Please recalibrate')  
  
 else:  
 if model.significant\_linear(): # calls model.train([0, 1])  
  
 debugprint('linear found')  
 if model.significant\_y\_intercept():  
 debugprint('y intercept found')  
 model.update\_calibration\_plot()  
 ask\_use\_or\_recalibrate(  
 'Your calibration equation does not go through the origin (0,0);\n'  
 'more specifically, it has a statistically significant y-intercept\n'  
 'at the 95% confidence level.')  
 else:  
  
 debugprint('no intercept found')  
 model.update\_calibration\_plot()  
 ask\_use\_or\_recalibrate(  
 'Your calibration equation has a significant linear trend and the y-intercept is zero.')  
 # move\_to\_measure\_concentration()  
 else:  
  
 debugprint('no trend found')  
 model.update\_calibration\_plot()  
 ask\_to\_recalibrate(  
 'Your calibration equation does not have a signficant linear trend.') # Please recalibrate')  
  
key\_function['@done-recording-cal'] = done\_recording  
  
######################################################################  
### Calibration Panel 3, ask to accept or recalibrate if not immediately accepted  
  
# Layouts  
  
calibration\_panel\_ask = [[sg.pin(sg.Column([  
 [sg.pin(sg.Column([[sg.Text('', key='@trend-status-ok')],  
 [sg.Text('Please recalibrate.'), sg.Button('Ok', key='@recalibrate1')]],  
 key='@just-recalibrate'))],  
 [sg.pin(sg.Column([[sg.Text('', key='@trend-status-ask')],  
 [sg.Text('Do you want to use this calibration equation,\n'  
 'or do you want to recalibrate?'), sg.Button('Use', key='@use'),  
 sg.Button('Recalibrate', key='@recalibrate2')]],  
 key='@use-or-recalibrate'))],  
 [sg.Canvas(key='@calibration-plot')]],  
 visible=False, key='@ask-recalibrate'))]]  
all\_calibration\_layouts += calibration\_panel\_ask  
all\_calibration\_keys.append('@ask-recalibrate')  
  
# Functions  
  
def ask\_to\_recalibrate(text):  
 window['@trend-status-ok'].update(text)  
 make\_calibration\_layout\_visible('@ask-recalibrate')  
 window['@use-or-recalibrate'].update(visible=False)  
 window['@just-recalibrate'].update(visible=True)  
 debugprint("window['@calibration-plot'])")  
 debugprint(window['@calibration-plot'])  
   
def ask\_use\_or\_recalibrate(text):  
 window['@trend-status-ask'].update(text)  
 make\_calibration\_layout\_visible('@ask-recalibrate')  
 window['@use-or-recalibrate'].update(visible=True)  
 window['@just-recalibrate'].update(visible=False)  
   
first\_time\_measuring = True  
def use\_calibration(event=None, values\_not\_used=None):  
 global first\_time\_measuring  
 first\_time\_measuring = True  
 debugprint('changing selection to measure tab')  
 window['@measure-tab'].update(disabled=False) # Set to false when calibration done  
 window['@measure-tab'].select()  
   
def move\_to\_measure\_concentration():  
 use\_calibration()  
   
key\_function['@use'] = use\_calibration  
  
def recalibrate(event, values\_not\_used):  
   
 model.clear\_calibration\_samples()  
 calibration\_samples\_table.set\_samples(model.samples)  
 make\_calibration\_layout\_visible('@cal-equation-form')  
 window['@done-recording-cal'].update(disabled=True)  
 window['@cal-equation-form'].update(visible=True)  
   
key\_function['@recalibrate1'] = recalibrate  
key\_function['@recalibrate2'] = recalibrate  
  
######################################################################  
### Measuring Panel  
  
# Layouts  
  
measured\_samples = [] #[sample id, volts, absorbance, concentration]  
  
measured\_samples\_table = table.Table('@measure-table',  
 measured\_samples,  
 ('Sample ID', 'Binary Volts', 'Absorbance', 'Concentration'),  
 spectro)  
  
measuring\_panel = [[sg.Column([  
 [sg.Canvas(key='@measuring-plot')],  
 [sg.Text('Sample ID'), sg.Input(key='@ID', size=15), sg.Button('Record', key='@measure')],  
 [measured\_samples\_table.make\_sgTable()],  
 [sg.Text('When you are ready to save these values, please insert a USB drive into the Raspberry Pi.', key='@insert-usb', visible=False)],  
 # when inserted, the above row will disappear and the following row will appear  
 [sg.Text('Enter file name:', key='@enter-filename', visible=True),  
 sg.Input(key='@filename', size=20, default\_text='', visible=True),  
 sg.Button('Save', key='@save', disabled=True, visible=True)],  
 [sg.pin(sg.Text(key='@lines-saved', visible=False))]])]]  
  
# Functions  
  
def measure(key\_not\_used, values\_not\_used):  
 # Read multiple voltages and return their average.  
 mean\_volts = spectro.read\_multiple\_voltages()  
 # convert volts to absorbances and display.  
 absorbance = spectro.volts\_to\_absorbance(mean\_volts)  
 # use model to convert absorbance to concentration, and display  
 concentration = model.use(absorbance)  
 # The "Measure" button clicked, so add this sample to the measured\_samples\_table.  
 id = window['@ID'].get()  
 if DEBUG:  
 debugprint(id, mean\_volts, absorbance, concentration)  
 measured\_samples\_table.add([id, mean\_volts, absorbance, concentration])  
 # Refresh the table display  
 window['@ID'].update('')  
 measured\_samples\_table.refresh()  
  
def save(event, values\_not\_used):  
 filename = window['@filename'].get()  
 if not filename.endswith('.csv'):  
 filename += '.csv'  
 if os.path.exists(filename):  
 main\_window['@lines-saved'].update(visible=False)  
 else:  
 success, msg = measured\_samples\_table.save(filename, ('Sample ID', 'Binary Volts', 'Absorbance', 'Concentration'))  
 main\_window['@lines-saved'].update(msg)  
 main\_window['@lines-saved'].update(visible=True)  
  
key\_function['@measure'] = measure  
key\_function['@save'] = save  
  
######################################################################  
#### Set up the TabGroup for the Calibration tab panel and the Measure Concentration tab panel  
  
tab\_group = sg.TabGroup([[  
 sg.Tab('Calibration', all\_calibration\_layouts, key='@calibration-tab', border\_width=5),  
 sg.Tab('Measure Concentration', measuring\_panel, key='@measure-tab', border\_width=5)]],  
 tab\_location='topleft', border\_width=5)  
  
tabbed\_layout = [[tab\_group], [sg.Push(), sg.Button('Quit')]]  
  
######################################################################  
#### Define screen as heading, voltage, and tabbed panel layouts  
  
screen = heading + layout\_voltage + tabbed\_layout # layout\_calibrate + layout\_sample  
  
  
######################################################################  
## Define PySimpleGUI main window and initialize calibration and measured sample tables.  
  
main\_window = sg.Window('Spectrophotometer', screen, finalize=True)  
spectro.set\_window(main\_window)  
model.set\_window(main\_window)  
calibration\_samples\_table.set\_window(main\_window)  
measured\_samples\_table.set\_window(main\_window)  
  
# recording\_calibration\_samples = False  
main\_window['@measure-tab'].update(disabled=True) # Set to false when calibration done  
  
if simulate\_trend is not None:  
 main\_window['@done-recording-cal'].update(disabled=False)  
  
######################################################################  
######################################################################  
## Start main loop  
######################################################################  
######################################################################  
  
while True:  
  
 # Read events and values from PySimpleGUI components that have changed state  
 # event, values = window.read(timeout=500) # milliseconds  
 window, event, values = sg.read\_all\_windows(timeout=500) # milliseconds  
  
 if DEBUG and event != '\_\_TIMEOUT\_\_':  
 print('event={} values={} tab\_group.Get()= {}'.format(event, values, tab\_group.Get()))  
  
 # Close application if window is closed or Quit is clicked  
 if event in (sg.WIN\_CLOSED, 'Quit'): # Closed window or clicked Quit button  
 window.close()  
 try:  
 if window == window\_load\_samples:  
 window\_load\_samples = None  
 else:  
 break  
 except:  
 break  
  
 # Determine which tab panel is showing  
 if tab\_group.Get() == '@calibration-tab':  
 tab = 'calibrating'  
 else:  
 tab = 'measuring'  
  
 # Read and display current volts  
 volts = spectro.read\_and\_display\_voltage()  
 if tab == 'measuring':  
 # Display equation found by inverting calibration curve.  
 if first\_time\_measuring:  
 model.update\_measuring\_plot()  
 first\_time\_measuring = False  
 # Read multiple voltages and return their average.  
 mean\_volts = spectro.read\_multiple\_voltages()  
 # convert volts to absorbances and display.  
 absorbance = spectro.volts\_to\_absorbance(mean\_volts)  
 # use model to convert absorbance to concentration, and display  
 concentration = model.use(absorbance)  
  
 # Handle editting table cells  
 if isinstance(event, tuple):  
 if event[0] == '@cal-table':  
 row, col = event[2]  
 if row is not None:  
 calibration\_samples\_table.edit\_cell('@cal-table', row + 1, col)  
  
 elif event[0] == '@measure-table':  
 row, col = event[2]  
 if row is not None:  
 measured\_samples\_table.edit\_cell('@measure-table', row + 1, col)  
  
 continue  
  
 #### Handle all other events by calling function sassociated with event in  
 #### key\_function dictionary  
  
 if event != '\_\_TIMEOUT\_\_':  
 key\_function[event](event, values)  
  
 if tab == 'measuring':  
 usb\_found = usb.usb\_inserted()  
 debugprint('usb\_found {}'.format(usb\_found))  
 main\_window['@insert-usb'].update(visible=not usb\_found)  
 main\_window['@enter-filename'].update(visible=usb\_found)  
 main\_window['@filename'].update(visible=usb\_found)  
 main\_window['@save'].update(visible=usb\_found)  
 main\_window['@save'].update(disabled=False)  
  
window.close()

# sensor.py

import numpy as np  
import Adafruit\_MCP3008 # for the analog-to-digital converter MCP3008  
import random  
import time  
  
# Average over this many voltage recordings for each measurement, with the  
# given delay between each recording.  
# The continuous voltage at the top are single recordings.  
  
N\_MEASUREMENTS\_EACH\_SAMPLE = 32  
DELAY\_BETWEEN\_READINGS = 0.02 # seconds  
  
DEBUG = False  
def debugprint(x):  
 if DEBUG:  
 print(x)  
  
######################################################################  
## class Sensor  
##  
## Interface with MCP3008 A/D chip.  
## Methods read voltage from MCP3008, updates the continuous voltage display, and returns  
## average of multiple samples  
######################################################################  
  
class Sensor():  
  
 def \_\_init\_\_(self, key):  
 self.key = key  
  
 # Software SPI configuration:  
 self.pin\_on\_MCP3008\_for\_spectro\_volts = 0  
 CLK = 18  
 MISO = 23  
 MOSI = 24  
 CS = 25  
  
 try:  
 self.mcp = Adafruit\_MCP3008.MCP3008(clk=CLK, cs=CS, miso=MISO, mosi=MOSI)  
 self.connected = True  
 except:  
 self.mcp = None  
 self.connected = False  
   
 def read\_and\_display\_voltage(self):  
 if self.mcp:  
 volts = self.mcp.read\_adc(self.pin\_on\_MCP3008\_for\_spectro\_volts)  
 self.window['@random-volts-used'](visible=False)  
 else:  
 volts = random.normalvariate(500, 10)  
 self.window['@random-volts-used'](visible=True)  
 try:  
 self.mcp = Adafruit\_MCP3008.MCP3008(clk=CLK, cs=CS, miso=MISO, mosi=MOSI)  
 except:  
 self.mcp = None  
  
 # Update display of binary volts  
 self.window[self.key].update('{:.2f}'.format(volts))  
 return volts  
  
 def read\_multiple\_voltages(self):  
 voltage\_sum = 0  
 for values in range(N\_MEASUREMENTS\_EACH\_SAMPLE):  
 time.sleep(DELAY\_BETWEEN\_READINGS)  
 if self.mcp:  
 volts = self.mcp.read\_adc(self.pin\_on\_MCP3008\_for\_spectro\_volts)  
 self.window['@random-volts-used'](visible=False)  
 else:  
 volts = random.normalvariate(500, 10)  
 self.window['@random-volts-used'](visible=True)  
 try:  
 self.mcp = Adafruit\_MCP3008.MCP3008(clk=CLK, cs=CS, miso=MISO, mosi=MOSI)  
 except:  
 self.mcp = None  
 voltage\_sum += volts  
 voltage\_mean = voltage\_sum / N\_MEASUREMENTS\_EACH\_SAMPLE  
 return voltage\_mean  
  
 def set\_volts\_zero\_concentration(self, volts):  
 self.volts\_for\_zero\_concentration = volts  
  
 def volts\_to\_absorbance(self, volts):  
 # volts\_for\_zero\_concentration set during recording of 0 concentration  
 try:  
 debugprint('{} {}'.format(self.volts\_for\_zero\_concentration, volts))  
 debugprint(self.volts\_for\_zero\_concentration/ volts)  
 debugprint(np.log10(self.volts\_for\_zero\_concentration/volts))  
 absorb = np.log10(self.volts\_for\_zero\_concentration / volts)  
 except Exception as ex:  
 print(ex)  
 print('WARNING: Ratio of zero concentrations volts to volts just measured is negative!')  
 print('Using absorbance of 0.')  
 absorb = 0  
 return absorb  
  
 def set\_window(self, window):  
 self.window = window

# model.py

DEBUG = False  
  
import numpy as np  
import math  
import scipy.stats as ss  
import matplotlib  
matplotlib.use('TkAgg')  
import matplotlib.pyplot as plt  
import matplotlib.ticker as mticker  
from matplotlib.backends.backend\_tkagg import FigureCanvasTkAgg  
  
plt.rcParams.update({  
 "figure.facecolor": (1.0, 0.0, 0.0, 0.0),  
 "axes.facecolor": (0.5, 0.5, 0.5, 0.0), # green with alpha = 50%  
 })  
  
def debugprint(x):  
 if DEBUG:  
 print(x)  
  
######################################################################  
## class Model  
##  
## Fits calibration model to samples.  
## Checks for signficance of coefficients.  
## Produces plots and equations.  
######################################################################  
  
class Model():  
  
 def \_\_init\_\_(self, calibration\_key, measuring\_key):  
 self.model = None  
 self.samples = []  
 self.calibration\_figure = plt.figure(figsize=(7, 4), dpi=100, facecolor=(0.8, 0.8, 0.8))  
 self.calibration\_axis = self.calibration\_figure.add\_subplot(111) #, facecolor=(0.6, 0.8, 0.6))  
 self.measuring\_figure = plt.figure(figsize=(9, 0.3), facecolor=(0.8, 0.8, 0.8))  
 self.measuring\_axis = self.measuring\_figure.add\_subplot(111)  
 self.calibration\_key = calibration\_key  
 self.measuring\_key = measuring\_key  
 self.calibration\_canvas = None  
 self.measuring\_canvas = None  
   
 def set\_window(self, window):  
 self.window = window  
   
 def clear\_calibration\_samples(self):  
 self.samples = []  
   
 def add\_calibration\_sample(self, concentration, mean\_volts, absorbance):  
 self.samples.append([concentration, mean\_volts, absorbance])  
   
 def get\_samples(self):  
 return np.array(self.samples)  
   
 def get\_n\_samples(self):  
 return len(self.samples)  
   
 def set\_calibration\_samples\_for\_testing(self, simulate\_trend):  
 self.samples = calibration\_samples\_for\_testing[simulate\_trend]  
   
 def set\_calibration\_samples(self, samples):  
 self.samples = samples  
   
 def make\_X\_powers(self, X, powers):  
 return np.hstack([X[:, c:c+1] \*\* powers for c in range(X.shape[1])])  
  
 def train(self, powers=[0, 1, 2, 3]):  
 # each sample is concentration, volts, absorbance  
 samps = np.array(self.samples)  
 X = samps[:, 0:1] # concentration  
 T = samps[:, 2:3] # absorbance  
 X = self.make\_X\_powers(X, powers)  
 W = np.linalg.lstsq(X, T, rcond=None)[0]  
 Y = X @ W  
 n, p = X.shape  
 MSE = np.sum((T - Y)\*\*2) / (n - p)  
 Sxx\_inv = np.linalg.pinv(X.T @ X)  
 SE\_W = np.sqrt(MSE \* Sxx\_inv.diagonal().reshape(-1, 1))  
 t\_W = W / SE\_W  
 p\_values = np.array([2 \* (1 - ss.t.cdf(np.abs(tw), n - p)) for tw in t\_W]).reshape(-1, 1)  
 t\_ppf = ss.t.ppf(1 - 0.025, n - p)  
 half\_interval = t\_ppf \* SE\_W  
 CI = np.hstack((W - half\_interval, W + half\_interval))  
 R2 = 1 - np.sum((T - Y)\*\*2) / np.sum((T - np.mean(T))\*\*2)  
 if n > p:  
 R2adj = 1 - (1 - R2) \* (n - 1) / (n - p)  
 else:  
 R2adj = np.nan  
 self.model = {'powers': powers,  
 'coef': W.ravel(),  
 'std err': SE\_W.ravel(),  
 't': t\_W.ravel(),  
 'P>|t|': p\_values.ravel(),  
 '[0.025': CI[:, 0],  
 '0.975]': CI[:, 1],  
 'R-squared': R2,  
 'Adj. R-squared': R2adj}  
 return self.model  
  
 def confidence\_interval\_does\_not\_include\_zero(self, coefficient\_index):  
 low = self.model['[0.025'][coefficient\_index]  
 high = self.model['0.975]'][coefficient\_index]  
 return low \* high > 0 # different signs return True  
  
 def significant\_linear(self):  
 self.train([0, 1])  
 return self.confidence\_interval\_does\_not\_include\_zero(1)  
  
 def significant\_quadratic(self):  
 self.train([0, 1, 2])  
 return self.confidence\_interval\_does\_not\_include\_zero(2)  
  
 def significant\_cubic(self):  
 self.train([0, 1, 2, 3])  
 return self.confidence\_interval\_does\_not\_include\_zero(3)  
  
 def significant\_y\_intercept(self):  
 sig\_y = self.confidence\_interval\_does\_not\_include\_zero(0)   
 if not sig\_y:  
 self.model['coef'][0] = 0 # set to zero because y intercept not significant  
 return sig\_y  
   
 def use(self, absorbance):  
 if len(self.model['coef']) == 3:  
 # absorbance is quadratic function of concentration  
 # To solve for concentration, use quadratic discriminant function  
 # ab = a conc^2 + b conc + c  
 # conc = (-b +- sqrt(b^2 - 4 a (c - ab))) / 2a  
 c, b, a = self.model['coef']  
 try:  
 sqrt = math.sqrt(b \* b - 4 \* a \* (c - absorbance))  
 except:  
 debugprint('In model.use. sqrt is nan. Returning 0 for concentration.')  
 return 0  
 conc1, conc2 = (-b + sqrt) / (2 \* a), (-b - sqrt) / (2 \* a)  
  
 samps = np.array(self.samples)  
 self.min\_concentration = samps[:, 0].min()  
 self.max\_concentration = samps[:, 0].max()  
  
 test1 = self.min\_concentration <= conc1 <= self.max\_concentration  
 test2 = self.min\_concentration <= conc2 <= self.max\_concentration  
 if test1 and not test2:  
 return conc1  
 if test2 and not test1:  
 return conc2  
 else:  
 debugprint('Both concentrations within range. Using first one')  
 return conc1  
  
 elif len(self.model['coef']) == 2:  
 # linear model, absorb = a conc + b  
 # so conc = (absorb - b) / a  
 b, a = self.model['coef']  
 conc = (absorbance - b) / a  
 return conc  
  
 else:  
 debugprint('Cannot use cubic model')  
 return None  
  
 def format\_model\_equation(self):  
 f = mticker.ScalarFormatter(useMathText=True)  
 f.set\_powerlimits((-3, 3))  
 coefficients = self.model['coef']  
 powers = self.model['powers']  
 s = 'Absorbance = '  
 make\_two\_lines = False  
 for i in reversed(range(len(self.model['powers']))):  
 p = self.model['powers'][i]  
 if p > 1:  
 c = coefficients[i]  
 if s[-2] == '=':  
 # first term  
 sgn = ''  
 else:  
 sgn = '\;-\;' if c < 0 else '\;+\;'  
 c = np.abs(c)  
 s += '${} {:.3f} \; (Concentration)^{} $'.format(sgn, c, p)  
 elif p == 1:  
 c = coefficients[i]  
 if s[-2] == '=':  
 # first term  
 sgn = ''  
 else:  
 sgn = '\;-\;' if c < 0 else '\;+\;'  
 c = np.abs(c)  
 s += '${} {:.3f} \; (Concentration) $'.format(sgn, c)  
 else:  
 c = coefficients[i]  
 debugprint(c)  
 if np.abs(c) > 0.0005:  
 if s[-2] == '=':  
 # first term  
 sgn = ''  
 else:  
 sgn = '\;-\;' if c < 0 else '\;+\;'  
 c = np.abs(c)  
 s += '${} {:.3f}$'.format(sgn, c)  
 return s  
  
 def format\_concentration\_equation(self):  
 f = mticker.ScalarFormatter(useMathText=True)  
 f.set\_powerlimits((-3, 3))  
 pm = r'\pm'  
 sqrt = r'\sqrt'  
 s = 'Concentration ='  
 if len(self.model['coef']) == 3:  
 c, b, a = self.model['coef']  
 stra = '{:.3f}'.format(a)  
 strb = '{:.3f}'.format(b)  
 if c != 0:  
 strc = '{:.3f}'.format(c)  
 else:  
 strc = ''  
 s += ' $( -{} {} {}{{({})^2 - 4 ({}) ({} - absorbance)}})\; / \;( 4 ({})^2)$'.format(strb, pm, sqrt, strb, stra, strc, stra)  
  
 elif len(self.model['coef']) == 2:  
 # linear  
 b, a = self.model['coef']  
 stra = '({:.3f})'.format(a) if a < 0 else '{:.3f}'.format(a)  
 if b != 0:  
 sgn = '\;+\;' if b < 0 else '\;-\;'  
 strb = '{} {:.3f}'.format(sgn, b)  
 s += '$(Absorbance {})\; /\; {}$'.format(strb, stra)  
 else:  
 strb = ''  
 # s += f' $(absorbance) \; /\; {stra}$'  
 s += ' $(Absorbance \; / \; {})$'.format(stra)  
  
 return s  
  
 def get\_canvas(self, figure, key):  
 return FigureCanvasTkAgg(figure, self.window[key].TKCanvas).get\_tk\_widget()  
  
 def update\_calibration\_plot(self):  
 if self.calibration\_canvas:  
 self.calibration\_canvas.forget()  
 self.calibration\_canvas = self.get\_canvas(self.calibration\_figure, self.calibration\_key)  
 samps = np.array(self.samples)  
 X = samps[:, 0:1] # concentration  
 T = samps[:, 2:3] # absorbance  
  
 self.calibration\_axis.cla()  
 self.calibration\_axis.plot(X, T, 'o')  
 self.calibration\_axis.set\_xlabel('Concentration')  
 self.calibration\_axis.set\_ylabel('Absorbance')  
 n = 20  
 xs = np.linspace(X.min(), X.max(), 20).reshape(-1, 1)  
 xs\_powers = self.make\_X\_powers(xs, self.model['powers'])  
 self.calibration\_axis.plot(xs, xs\_powers @ self.model['coef'], 'r')  
 equation = self.format\_model\_equation()  
 r2 = '$R^2$ = {:.2f}\n$R^2$ adj = {:.2f}'.format(self.model["R-squared"], self.model["Adj. R-squared"])  
 self.calibration\_axis.text(0.05, 0.8, equation + '\n' + r2,  
 transform=self.calibration\_axis.transAxes, fontsize='small')  
 self.calibration\_canvas.pack(side='top')  
 self.calibration\_figure.tight\_layout()  
  
  
 def update\_measuring\_plot(self):  
 if self.measuring\_canvas:  
 self.measuring\_canvas.forget()  
 equation = self.format\_concentration\_equation()  
 self.measuring\_canvas = self.get\_canvas(self.measuring\_figure, self.measuring\_key)  
 self.measuring\_axis.text(0.02, 0.1, equation, fontsize='large')  
 self.measuring\_axis.axis('off')  
 self.measuring\_canvas.pack(side='top')  
  
# Test data for quickly filling in calibration table.  
# Only available if spectrophotometer is not connected.  
  
calibration\_samples\_for\_testing = {  
 'cubic': [  
 [0.0, 994.28, 0.0],  
 [1.16, 749.75, 0.123],  
 [2.9, 566.9, 0.244],  
 [6.84, 414, 0.381],  
 [11.15, 313.03, 0.502],  
 [12.9, 234.37, 0.628],  
 [14.035, 182.75, 0.736]],  
 'quadratic-no-intercept': [  
 [0.0, 987.75, 0.0],  
 [0.576, 741.96, 0.124],  
 [1.22, 558.21, 0.248],  
 [1.942, 401.06, 0.391],  
 [2.836, 305.21, 0.510],  
 [3.969, 229.06, 0.635],  
 [6.473, 178.65, 0.743]],  
 'quadratic-intercept': [  
 [0.0, 995.87, 0.0],  
 [0.001, 740, 0.12314],  
 [0.002, 680, 0.1657],  
 [0.724, 411.28, 0.384],  
 [1.618, 312.03, 0.504],  
 [2.751, 234.62, 0.628],  
 [5.255, 182.31, 0.600]], # 737]],  
 'linear-no-intercept': [  
 [0.0, 987.56, 0.0],  
 [0.875, 741.93, 0.124],  
 [1.681, 556.46, 0.249],  
 [2.514, 394, 0.399],  
 [3.380, 304.84, 0.510],  
 [4.214, 228.53, 0.636],  
 [5.015, 178.28, 0.743]],  
 'linear-intercept': [  
 [0.0, 994.46, 0.0],  
 [0.001, 747.31, 0.124],  
 [0.847, 563.15, 0.247],  
 [1.68, 401.84, 0.394],  
 [2.546, 312.34, 0.503],  
 [3.379, 233.40, 0.629],  
 [4.181, 181.31, 0.739]],  
 'no-trend': [  
 [0.0, 994.46, 0.0],  
 [0.001, 747.31, 1],  
 [0.847, 563.15, 1],  
 [1.68, 401.84, 1],  
 [2.546, 312.34, 1],  
 [3.379, 233.40, 1],  
 [4.181, 181.31, 1]]  
 }

# table.py

import PySimpleGUI as sg  
import os  
import usb  
  
######################################################################  
## class Table  
##  
## Used for table of calibration samples and also for table of measured samples  
######################################################################  
  
class Table():  
  
 def \_\_init\_\_(self, key, data, headings, spectro): # data, headings):  
  
 self.key = key  
 self.data = data  
 self.headings = headings  
 self.spectro = spectro  
 self.dec\_places = 3  
   
 def make\_sgTable(self):  
  
 self.sgtable = sg.Table(key=self.key, values=self.data, headings=self.headings,  
 justification='right', enable\_click\_events=True,  
 expand\_x=True, expand\_y=True,  
 col\_widths=120, row\_height=25) #300  
 return self.sgtable  
  
 def set\_window(self, window):  
 self.window = window  
  
 def set\_samples(self, samples):  
 self.data = []  
 dp = self.dec\_places  
 for row in samples:  
 row\_decimals = []  
 for r in row:  
 try:  
 rconv = int(float(r) \* 10\*\*dp) / 10\*\*dp  
 except:  
 rconv = r  
 row\_decimals.append(rconv)  
 self.data.append(row\_decimals)  
   
 def add(self, row):  
 '''Used only for table of measured values, not calibration samples.'''  
 dp = self.dec\_places  
 row\_decimals = []  
 for r in row:  
 try:  
 rconv = int(float(r) \* 10\*\*dp) / 10\*\*dp  
 except:  
 rconv = r  
 row\_decimals.append(rconv)  
 self.data.append(row\_decimals)  
   
 def cell\_callback(self, event, row, col, text, keypressed):  
 table = self.window[self.key].Widget  
 widget = event.widget  
 if keypressed == 'Return':  
 text = widget.get()  
 values = list(table.item(row, 'values'))  
 values[col] = text  
 table.item(row, values=values)  
 self.data[row-1][col] = float(text)  
 widget.destroy()  
  
 if self.data[row-1][0] == 0:  
 self.spectro.set\_volts\_zero\_concentration(self.data[row-1][1])  
 self.data[row-1][2] = self.spectro.volts\_to\_absorbance(self.data[row-1][1])  
  
 widget.master.destroy()  
 self.refresh()  
   
 def refresh(self):  
 self.sgtable.update(values=self.data)  
 table = self.window[self.key].Widget  
 self.sgtable.update(visible=True)  
   
 def edit\_cell(self, key, row, col):  
 root = self.window.TKroot  
 table = self.window[self.key].Widget  
 text = table.item(row, 'values')[col]  
 x, y, width, height = table.bbox(row, col)  
 wx, wy = table.winfo\_x(), table.winfo\_y()  
 frame = sg.tk.Frame(table)  
 frame.place(x=x, y=y, anchor='nw', width=width, height=height)  
 textvariable = sg.tk.StringVar()  
 textvariable.set(text)  
 entry = sg.tk.Entry(frame, textvariable=textvariable, justify='right',  
 font=('latin modern roman', 16, 'bold'))  
 entry.pack()  
 entry.select\_range(0, sg.tk.END)  
 entry.icursor(sg.tk.END)  
 entry.focus\_force()  
 entry.bind('<Return>', lambda e, r=row, c=col, t=text,  
 k='Return':self.cell\_callback(e, r, c, t, k))  
   
 def save(self, filename, columns):  
  
 # Look for USB devcies. Exit if not found.  
   
 path = usb.get\_usb\_path()  
 if path is None:  
 return False, 'USB drive not found. Save was not successful.'  
   
 if not filename.endswith('.csv'):  
 filename += '.csv'  
  
 filename = path + '/' + filename   
  
 if os.path.exists(filename):  
 answer = sg.popup\_yes\_no('The file ' + filename + ' already exists. Do you want to replace it with this new data?')  
 if answer == 'No':  
 return False, 'Please enter a different filename.'  
   
 # Open file and write each row from this table's data.  
  
 try:  
 with open(filename, 'w') as f:  
 f.write(', '.join(columns))  
 f.write('\n')  
 for row in self.data:  
 for i, col in enumerate(row):  
 if isinstance(col, str):  
 f.write(col)  
 elif isinstance(col, int):  
 f.write(str(col))  
 elif isinstance(col, float):  
 f.write('{:.7g}'.format(col))  
 if i + 1 < len(row):  
 f.write(', ')  
 f.write('\n')  
  
 return True, '{} lines written to {}'.format(len(self.data), filename)  
  
 except:  
 return False, 'Save to file ' + filename + ' was not successful. Please try again or choose different file name.'

# usb.py

import os  
import glob  
import subprocess  
  
# Functions to find and access USB drive  
# from http://stackoverflow.com/questions/22615750/how-can-the-directory-of-a-usb-drive-connected-to-a-system-be-obtained  
  
DEBUG = False  
  
def debugprint(x):  
 if DEBUG:  
 print(x)  
  
def get\_usb\_devices():  
 sdb\_devices = map(os.path.realpath, glob.glob('/dev/sd\*'))  
 usb\_devices = [d for d in sdb\_devices if len(d) > 0]  
 debugprint('{} {}'.format(list(sdb\_devices), list(usb\_devices)))  
 return dict((os.path.basename(dev), dev) for dev in usb\_devices)  
  
def get\_usb\_path():  
 devices = get\_usb\_devices()  
 output = subprocess.check\_output(['mount']).splitlines()  
 is\_usb = lambda path: any(dev in path.decode('utf8') for dev in devices)  
 usb\_info = (line for line in output if is\_usb(line.split()[0]))  
 fullInfo = []  
 for info in usb\_info:  
 mountURI = info.split()[0]  
 usbURI = info.split()[2]  
 for x in range(3, info.split().\_\_sizeof\_\_()):  
 if info.split()[x].\_\_eq\_\_("type"):  
 for m in range(3, x):  
 usbURI += " "+info.split()[m]  
 break  
 fullInfo.append([mountURI.decode('utf8'), usbURI.decode('utf8')])  
 debugprint('devices {}'.format(devices))  
 debugprint('usb\_info {}'.format(list(usb\_info)))  
 debugprint('fullInfo {}'.format(fullInfo))  
 if not fullInfo:  
 return None  
 for dev in fullInfo[0]:  
 if 'media' in dev:  
 return dev  
 return None  
  
def usb\_inserted():  
 path = get\_usb\_path()  
 return path is not None  
   
if \_\_name\_\_ == '\_\_main\_\_':  
 print('get\_usb\_path() returns', get\_usb\_path())