```
# File: QYUI.py
#
# Last modified: 04/18/20
# This file contains a graphic user interface program and four modules written for it.
# Single_Motor_Control and _2D interacts with an applied motion stepper
# LeCroy_Scope interacts with a Lecroy Osciloscope
# find_ip_addr helps to detect ip address of the oscilloscope, when there's no user input.
#
#
# This graphic user interface allows user to
# (1) set up data point positions, channel description and start data acquisition
#     (by calling Data_Run_2D.py or Data_Run_3D.py)
# (2) control the motor (by calling Motor_Control_2D.py or Motor_Control_3D.py)
# (3) view graphic display of the current probe position and data point positions in the chamber
#
#
# Author: Yuchen Qian
#
#

import numpy
import math
import sys
import os
import os.path
import time
import datetime
from Motor_Control_2D_xy import Motor_Control_2D
from LeCroy_Scope import LeCroy_Scope, WAVEDESC_SIZE
from LeCroy_Scope import EXPANDED_TRACE_NAMES
import tkinter
from tkinter import filedialog
import tkinter.messagebox
import h5py as h5py

dir_path=os.path.dirname(os.path.realpath(__file__))
version_number="03/01/2018 12:37pm"      # update this when a change has been made

from PyQt5 import QtCore
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *

from mpl_toolkits.mplot3d import axes3d
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
from matplotlib.figure import Figure
import matplotlib.patches as patches
from scipy.linalg import norm

data_running = False # to keep track of the data acquisition thread
############################################################################################
############################################################################################


class MyMplCanvas(FigureCanvas):
  '''
  A FigureCanvas that plots the user setup data taking positions, current probe positions,
  and finished positions
  '''

  def __init__(self, parent=None, width=6, height=3, dpi=100):
    fig = Figure(figsize=(width, height), dpi=dpi)
    self.ax = fig.add_subplot(111)
    self.ax.set_xlim(-35, 35)
    self.ax.set_ylim(-35, 35)

    FigureCanvas.__init__(self, fig)
    self.setParent(parent)

    FigureCanvas.setSizePolicy(self,
                               QSizePolicy.Expanding,
                               QSizePolicy.Expanding)
    FigureCanvas.updateGeometry(self)

    self.ax.grid(which = 'both')
    self.ax.add_patch(patches.Rectangle((-38, -50), 76, 100, fill = False, edgecolor = 'red'))

    self.matrix = self.ax.scatter(0, 0, 0, color = 'blue', marker = 'o')
    self.point = self.ax.scatter(0, 0, 0, color = 'red', marker = '*')
    self.initialize_visited_points() # initialize an array to record finished positions

  def update_figure(self, param):
    # update the points on the graph when user input a new set of positions
    # param: user input that contains max x, min x, max y, min y, and number points in x and y
    # which defines a rectangle on the grid

    self.parameters = param

    self.xmax = self.parameters['xmax']
    self.xmin = self.parameters['xmin']
    self.ymax = self.parameters['ymax']
    self.ymin = self.parameters['ymin']
    self.nx = self.parameters['nx']
    self.ny = self.parameters['ny']
    self.xpos = numpy.linspace(self.xmin,self.xmax,self.nx)
    self.ypos = numpy.linspace(self.ymin,self.ymax,self.ny)

    # Create two arrays that correspond x and y of each point
    self.X = numpy.zeros(self.nx*self.ny)
    self.Y = numpy.zeros(self.nx*self.ny)

    index = 0
    for xx in self.xpos:
      for yy in self.ypos:
        self.X[index] = xx
        self.Y[index] = yy
        index += 1

    # Plot the points on the grid
    self.matrix = self.ax.scatter(self.X, self.Y, color = 'blue', marker = 'o')
    self.draw()
    print(self.parameters)

  def update_probe(self, xnow, ynow):
    # Update the current position of the probe
    # xnow, ynow: x,y positions of the probe, calculated from the feedback of the motor
    self.point = self.ax.scatter(xnow, ynow, color = 'red', marker = '*')
    self.draw()

  def update_axis(self, x1, y1, x2, y2):
    # Change the range of the axis. See class Axis_Controls.
    # x1, y1, x2, y2: x min, y min, x max, y max from user input.
    self.ax.set_xlim(x2, x1)
    self.ax.set_ylim(y2, y1)

  def finished_positions(self, x, y):
    # Denote the points where data has been taken with green
    # x,y: finished positions
    self.finished_x.append(x)
    self.finished_y.append(y)
    self.visited_points = self.ax.scatter(self.finished_x, self.finished_y, color = 'green', marker = 'o')
    self.draw()

  def initialize_visited_points(self):
    self.finished_x = []
    self.finished_y = []
    self.visited_points = self.ax.scatter(self.finished_x, self.finished_y, color = 'green', marker = 'o')



class Axis_Controls(QGroupBox):
  '''
  A widget that allows user to change the range of x, y axis in class MyMplCanvas
  through spinboxes or directly putting in number.
  '''

  def __init__(self):
    super().__init__()
    self.xupInput = QSpinBox()
    self.yupInput = QSpinBox()
    self.xlowInput = QSpinBox()
    self.ylowInput = QSpinBox()

    # Set default value and range for the axis
    self.xupInput.setRange(-60, 60)
    self.yupInput.setRange(-60, 60)
    self.xlowInput.setRange(-60, 60)
    self.ylowInput.setRange(-60, 60)

    self.xupInput.setValue(35)
    self.yupInput.setValue(35)
    self.xlowInput.setValue(-35)
    self.ylowInput.setValue(-35)


    self.xaxisLabel = QLabel("x axis range:")
    self.yaxisLabel = QLabel("y axis range:")
    self.toLabel = QLabel("to")
    self.toLabel2 = QLabel("to")
    self.blankLabel = QLabel("  ")

    # Set up widget layout
    axisLayout = QGridLayout()
    axisLayout.addWidget(self.xaxisLabel, 0, 0)
    axisLayout.addWidget(self.xlowInput, 0, 1)
    axisLayout.addWidget(self.toLabel, 0, 2)
    axisLayout.addWidget(self.xupInput, 0, 3)
    axisLayout.addWidget(self.blankLabel, 0, 4)
    axisLayout.addWidget(self.yaxisLabel, 0, 5)
    axisLayout.addWidget(self.ylowInput, 0, 6)
    axisLayout.addWidget(self.toLabel2, 0, 7)
    axisLayout.addWidget(self.yupInput, 0, 8)
    self.setLayout(axisLayout)



#############################################################################################
#############################################################################################
```

```
class Position_Controls(QGroupBox):
  '''
  Allows user to set up data taking positions
  '''
  def __init__(self):
    super().__init__()
    self.setTitle("Set up DAQ position")

    self.xMaxLabel = QLabel("Max x:")
    self.xMinLabel = QLabel("Min x:")
    self.yMaxLabel = QLabel("Max y:")
    self.yMinLabel = QLabel("Min y:")
    self.nxLabel = QLabel("nx:")
    self.nyLabel = QLabel("ny:")

    self.xMaxInput = QLineEdit()
    self.xMinInput = QLineEdit()
    self.yMaxInput = QLineEdit()
    self.yMinInput = QLineEdit()
    self.nxInput = QLineEdit()
    self.nyInput = QLineEdit()

    self.xMaxInput.setText("0")
    self.xMinInput.setText("0")
    self.yMaxInput.setText("0")
    self.yMinInput.setText("0")
    self.nxInput.setText("1")
    self.nyInput.setText("1")


    self.ConfirmButton = QPushButton("Confirm Input",self)

    # Set up widget layout
    controlsLayout = QGridLayout()
    controlsLayout.addWidget(self.xMaxLabel, 0, 0)
    controlsLayout.addWidget(self.xMinLabel, 1, 0)
    controlsLayout.addWidget(self.yMaxLabel, 2, 0)
    controlsLayout.addWidget(self.yMinLabel, 3, 0)
    controlsLayout.addWidget(self.nxLabel, 4, 0)
    controlsLayout.addWidget(self.nyLabel, 5, 0)


    controlsLayout.addWidget(self.xMaxInput, 0, 1)
    controlsLayout.addWidget(self.xMinInput, 1, 1)
    controlsLayout.addWidget(self.yMaxInput, 2, 1)
    controlsLayout.addWidget(self.yMinInput, 3, 1)
    controlsLayout.addWidget(self.nxInput, 4, 1)
    controlsLayout.addWidget(self.nyInput, 5, 1)

    controlsLayout.addWidget(self.ConfirmButton, 6, 1)

    self.setLayout(controlsLayout)



##########################################################################################################
##########################################################################################################


class Acquisition_Controls(QGroupBox):
  '''
  Allows users to set up and start a data run, or take a test shot
  '''

  def __init__(self):
    super().__init__()
    self.DataRun = QPushButton("Start Data Acquisition", self)
    self.TestShot = QPushButton("Take Single Test Shot", self)
    self.num_run = QSpinBox()
    self.num_shots = QSpinBox()
    self.num_run_label = QLabel("Number of total runs:")
    self.num_shots_label = QLabel("Shots per position:")

    self.num_run.setRange(1, 100)
    self.num_shots.setRange(1, 200)
    self.num_run.setValue(1)
    self.num_shots.setValue(1)

    ACLayout = QGridLayout()
    ACLayout.addWidget(self.DataRun, 0, 0)
    ACLayout.addWidget(self.TestShot, 0, 1)
    ACLayout.addWidget(self.num_run_label, 1, 0)
    ACLayout.addWidget(self.num_shots_label, 2, 0)
    ACLayout.addWidget(self.num_run, 1, 1)
    ACLayout.addWidget(self.num_shots, 2, 1)

    self.setLayout(ACLayout)


##########################################################################################################
##########################################################################################################


class Motor_Movement(QGroupBox):
  '''
  Allow users to control the motor
  '''

  def __init__(self, x_ip_addr = None, y_ip_addr = None, MOTOR_PORT = None):
    super().__init__()
    self.setTitle("Motor Movement Control")

    self.x_ip_addr = x_ip_addr
    self.y_ip_addr = y_ip_addr
    self.MOTOR_PORT = MOTOR_PORT

    # (cm) Move probe to absolute position along the shaft counted by motor encoder
    self.xMoveLabel = QLabel("Move x motor to:")
    self.yMoveLabel = QLabel("Move y motor to:")
    self.xMoveInput = QLineEdit()
    self.yMoveInput = QLineEdit()

    # For 3D acquisition, need another feature to move the probe to absolute position.
    # this should be done by calling "move_to_position" function in Motor_Control_3D,
    #    with corresponding geometry calculation

    # Set velocity.
    self.xvLabel = QLabel("Set x velocity:")
    self.yvLabel = QLabel("Set y velocity:")
    self.xvInput = QLineEdit()
    self.yvInput = QLineEdit()


    self.MoveButton     = QPushButton("Move Motor", self)
    self.StopNowButton  = QPushButton("BUG don't click", self)
    self.SetZero        = QPushButton("Set Zero", self)
    self.SetVelocity = QPushButton("Set Velocity", self)
    self.MoveButton.clicked.connect(self.move_to_position)
    self.StopNowButton.clicked.connect(self.stop_now)
    self.SetZero.clicked.connect(self.zero)
    self.SetVelocity.clicked.connect(self.set_velocity)

    self.CurposLabel = QLabel("Current probe position (cm, cm):")
    self.CurposInput = QLineEdit(readOnly = True)
    self.velocityButton = QPushButton("Get motor speed (rpm):")
    self.velocityInput = QLineEdit(readOnly = True)
    self.velocityButton.clicked.connect(self.update_current_speed)

    MMLayout = QGridLayout()
    MMLayout.addWidget(self.xMoveLabel, 0, 0)
    MMLayout.addWidget(self.yMoveLabel, 0, 1)
    MMLayout.addWidget(self.xMoveInput, 1, 0)
    MMLayout.addWidget(self.yMoveInput, 1, 1)
    MMLayout.addWidget(self.MoveButton, 1, 2)
    MMLayout.addWidget(self.xvLabel, 2, 0)
    MMLayout.addWidget(self.yvLabel, 2, 1)
    MMLayout.addWidget(self.xvInput, 3, 0)
    MMLayout.addWidget(self.yvInput, 3, 1)
    MMLayout.addWidget(self.SetVelocity, 3, 2)
    MMLayout.addWidget(self.SetZero, 4, 0)
    MMLayout.addWidget(self.StopNowButton, 4, 1)
    MMLayout.addWidget(self.CurposLabel, 5, 0)
    MMLayout.addWidget(self.CurposInput, 5, 1, 1, 2)
    MMLayout.addWidget(self.velocityButton, 6, 0)
    MMLayout.addWidget(self.velocityInput, 6, 1, 1, 2)


    self.setLayout(MMLayout)

    # Initialize Motor_Control_2D
    self.mc = Motor_Control_2D(x_ip_addr = self.x_ip_addr, y_ip_addr = self.y_ip_addr)

#-----------------------------------------------------------------------

  def move_to_position(self):
    # Directly move the motor to their absolute position
    try:
      x_pos = float(self.xMoveInput.text())
      y_pos = float(self.yMoveInput.text())
      self.mc.enable()
      self.mc.move_to_position(x_pos, y_pos)
      self.mc.disable()
    except ValueError:
      QMessageBox.about(self, "Error", "Position should be valid numbers.")

  def disable():
    self.mc.disable()

  def stop_now(self):
    # Stop motor movement now
    self.mc.stop_now()


  def zero(self):
    zeroreply=QMessageBox.question(self, "Set Zero",
      "You are about to set the current probe position to (0,0). Are you sure?",
      QMessageBox.Yes, QMessageBox.No)
    if zeroreply == QMessageBox.Yes:
      QMessageBox.about(self, "Set Zero", "Probe position is now (0,0).")
```

```
        self.mc.set_zero()


    def ask_velocity(self):
      return self.mc.ask_velocity()


    def set_velocity(self):
      xv = self.xvInput.text()
      yv = self.yvInput.text()
      self.mc.set_velocity(xv, yv)


    def current_probe_position(self):
      return self.mc.current_probe_position()

    def update_current_speed(self):
      self.speedx, self.speedy = self.ask_velocity()
      self.velocityInput.setText("(" + str(self.speedx) + " ," + str(self.speedy) +")")

    def set_input_usage(self, usage):
      self.mc.set_input_usage(usage)

    def set_steps_per_rev(self, stepsx, stepsy):
      self.mc.set_steps_per_rev(stepsx, stepsy)



##############################################################################################
##############################################################################################


class Scope_Channel(QGroupBox):
  '''
  Allow users to write comments for each scope channel before data run starts
  '''

  def __init__(self):
    super().__init__()
    self.titleLabel = QLabel("Enter channel descriptions")
    self.c1Label = QLabel("Channel 1:")
    self.c2Label = QLabel("Channel 2:")
    self.c3Label = QLabel("Channel 3:")
    self.c4Label = QLabel("Channel 4:")
    self.c1Input = QLineEdit()
    self.c2Input = QLineEdit()
    self.c3Input = QLineEdit()
    self.c4Input = QLineEdit()


    SCLayout = QGridLayout()
    SCLayout.addWidget(self.titleLabel, 0, 0, 1, 2)
    SCLayout.addWidget(self.c1Label, 1, 0)
    SCLayout.addWidget(self.c2Label, 2, 0)
    SCLayout.addWidget(self.c3Label, 3, 0)
    SCLayout.addWidget(self.c4Label, 4, 0)
    SCLayout.addWidget(self.c1Input, 1, 1)
    SCLayout.addWidget(self.c2Input, 2, 1)
    SCLayout.addWidget(self.c3Input, 3, 1)
    SCLayout.addWidget(self.c4Input, 4, 1)
    self.setLayout(SCLayout)


update_pos = None

##############################################################################################
##############################################################################################

class Software_Version(QGroupBox):
  def __init__(self):
    super().__init__()
    self.mod_timestr=(os.path.getmtime(dir_path))
    self.mod_datetime=datetime.datetime.fromtimestamp(self.mod_timestr).strftime('%Y-%b-%d %H:%M:%S %p')
    self.vLabel = QLabel("Last Modified: ")
    self.lastmodified = QLabel(self.mod_datetime)
    self.version = QLabel("Version: "+version_number)

    SVLayout = QGridLayout()
    SVLayout.addWidget(self.vLabel, 0, 0)
    SVLayout.addWidget(self.lastmodified, 1, 0)
    SVLayout.addWidget(self.version, 2, 0)
    self.setLayout(SVLayout)

##############################################################################################
##############################################################################################
class Signals(QObject):
  '''
  Five signals to keep track of / update the program during the data run
  '''
  finished = pyqtSignal() # sent when a run is finished
  updated_position = pyqtSignal(float, float) # sent the position info when the probe moves to a new position
  new_screen_dump = pyqtSignal() # sent when asking for an up-to-date screenshot from the scope
  finished_position = pyqtSignal(float, float) # sent the position info when data at that position has been recorded
  cancel = pyqtSignal() # sent if the user click to cancel the current data run
```

```
class Data_Run_Thread(QRunnable):
  '''
  The main data recording thread that starts when users click to start a data run.
  '''

  def __init__(self, hdf5_filename, pos_param, channel_description, ip_addrs):
    super(Data_Run_Thread, self).__init__()

    self.hdf5_filename = hdf5_filename
    self.pos_param = pos_param
    self.channel = channel_description
    self.ip_addrs = ip_addrs
    self.signals = Signals()

  def get_channel_description(self, tr) -> str:
    #callback function to return a string containing a description of the data in each recorded channel

    #user: assign channel description text here to override the default:
    if tr == 'C1':
      return self.channel["C1"]
    if tr == 'C2':
      return self.channel["C2"]
    if tr == 'C3':
      return self.channel["C3"]
    if tr == 'C4':
      return self.channel["C4"]

    # otherwise, program-generated default description strings follow
    if tr in EXPANDED_TRACE_NAMES.keys():
      return 'no entered description for ' + EXPANDED_TRACE_NAMES[tr]

    return '**** get_channel_description(): unknown trace indicator "'+tr+'". How did we get here?'



  def get_positions(self) -> ([(),(),(),()], numpy.array, numpy.array, numpy.array):
    # callback function to return the positions array

    xmax = self.pos_param["xmax"]
    xmin = self.pos_param["xmin"]
    ymax = self.pos_param["ymax"]
    ymin = self.pos_param["ymin"]
    nx = self.pos_param["nx"]
    ny = self.pos_param["ny"]

    xpos = numpy.linspace(xmin,xmax,nx)
    ypos = numpy.linspace(ymin,ymax,ny)

    num_duplicate_shots = self.pos_param["num_shots"]       # number of duplicate shots recorded at the ith location
    num_run_repeats = self.pos_param["num_run"]           # number of times to repeat sequentially over all locations

    # allocate the positions array, fill it with zeros
    positions = numpy.zeros((nx*ny*num_duplicate_shots*num_run_repeats), dtype=[('Line_number', '>u4'), ('x', '>f4'),
('y', '>f4')])

    #create rectangular shape position array
    index = 0
    for repeat_cnt in range(num_run_repeats):
      for y in ypos:
        for x in xpos:
          for dup_cnt in range(num_duplicate_shots):
            positions[index] = (index+1, x, y)
            index += 1

    # print(positions)       # for debugging

    return positions, xpos, ypos, num_duplicate_shots

  def get_hdf5_filename(self) -> str:

    avoid_overwrite = True     # <-- setting this to False will allow overwriting an existing file without a prompt

    fn = self.hdf5_filename
    if fn == None  or len(fn) == 0  or  (avoid_overwrite  and  os.path.isfile(fn)):
      # if we are not allowing possible overwrites as default, and the file already exists, use file open dialog
      tk = tkinter.Tk()
      tk.withdraw()    # prevent tk GUI from popping up
      fnoptions={}
      fnoptions['title'] = 'Save file as ...'
      fnoptions['defaultextension'] = '.hdf5'
      fnoptions['filetypes'] = [("Hierarchical Data Format",'*.hdf5'), ("All files",'*.*')]
      fn = filedialog.asksaveasfilename(**fnoptions)
      if not fn:     # if user pressed 'cancel', fn = None
        print("\nUser cancelled save file input.")
        # if len(fn) == 0:
        #   #raise SystemExit(0)
        #   fn = exit
      tk.destroy()

    self.hdf5_filename = fn    # save it for later
    return fn

  def acquire_displayed_traces(self, scope, datasets, hdr_data, pos_ndx):
    """ worker for below :
      acquire enough sweeps for the averaging, then read displayed scope trace data into HDF5 datasets
    """
    timeout = 2000 # seconds
    timed_out, N = scope.wait_for_max_sweeps(str(pos_ndx)+': ', timeout)  # leaves scope not triggering

    if timed_out:
      print('**** averaging timed out: got '+str(N)+' at %.6g s' % timeout)

    traces = scope.displayed_traces()

    for tr in traces:
      try:
        NPos,NTimes = datasets[tr].shape
        # sometimes for 10000 the scope hardware returns 10001 samples, so we have to specify [0:NTimes]:
        datasets[tr][pos_ndx,0:NTimes] = scope.acquire(tr)[0:NTimes]
        #?# datasets[tr].flush()
      except KeyError:
        print(tr + ' is displayed on the scope but not recorded.' +
                   ' To record this channel, please display the trace before starting the data run.')
        continue

    for tr in traces:
      try:
        hdr_data[tr][pos_ndx] = numpy.void(scope.header_bytes())    # valid after scope.acquire()
        #?# hdr_data[tr].flush()
        #?# are there consequences in timing or compression size if we do the flush()s recommend for the SWMR function?
      except KeyError:
        continue

    scope.set_trigger_mode('NORM')   # resume triggering

  #-----------------------------------------------------------------------------------------

  def create_sourcefile_dataset(self, grp, fn):
    """ worker for below:
      create an HDF5 dataset containing the contents of the specified file
      add attributes file name and modified time
    """
    fds_name = os.path.basename(fn)
    fds = grp.create_dataset(fds_name, data=open(fn, 'r').read())
    fds.attrs['filename'] = fn
    fds.attrs['modified'] = time.ctime(os.path.getmtime(fn))

  #-----------------------------------------------------------------------------------------

  def run(self):
    # The main data acquisition routine
    #
    #   Arguments are user-provided callback functions that return the following:
    #     get_hdf5_filename()          the output HDF5 filename,
    #     get_positions()              the positions array,
    #     get_channel_description(c)   the individual channel descriptions (c = 'C1', 'C2', 'C3', 'C4'),
    #     get_ip_addresses()           a dict of the form {'scope':'10.0.0.12', 'x':'10.0.0.13', 'y':'10.0.0.14', 'z':''}
    #                                       if a key is not specified, no motion will be attempted on that axis
    #
    #   Creates the HDF5 file, creates the various groups and datasets, adds metadata (see "HDF5 OUTPUT FILE SETUP")
    #
    #   Iterates through the positions array (see "MAIN ACQUISITION LOOP"):
    #       calls motor_control.set_position(pos)
    #       Waits for the scope to average the data, as per scope settings
    #       Writes the acquired scope data to the HDF5 output file
    #
    #   Closes the HDF5 file when done
    #
    #============================
    # list of files to include in the HDF5 data file
    thispath = os.path.realpath(__file__)
    src_files = [thispath,           # ASSUME this file is in the same directory as the next two:
          os.path.dirname(thispath)+os.sep+'LeCroy_Scope.py',
          os.path.dirname(thispath)+os.sep+'Motor_Control_2D_xy.py'
           ]
    #for testing, list these:s
    print('Files to record in the hdf5 archive:')
    print('    invoking file (this file)     =', src_files[0])
    print('    LeCroy_Scope file  =', src_files[1])
    print('    motor control file =', src_files[2])

    #============================
    # position array given by Data_Run_GUI_xy.py:
    positions, xpos, ypos, num_duplicate_shots = self.get_positions()

    # Create empty position arrays
    if xpos is None:
      xpos = numpy.array([])
    if ypos is None:
      ypos = numpy.array([])

    #============================

    mc = Motor_Control_2D(x_ip_addr = self.ip_addrs['x'], y_ip_addr = self.ip_addrs['y'])


    ######### HDF5 OUTPUT FILE SETUP #########

    # Open hdf5 file for writing (user callback for filename):
    ofn = self.get_hdf5_filename()      # callback arg to the current function
    if not ofn:    #if user pressed cancel during input file name
      self.signals.cancel.emit()
      pass
    else:
      # 'w' - overwrite (we should have determined whether we want to overwrite in get_hdf5_filename()):
      f = h5py.File(ofn,  'w')
      # f = h5py.File(ofn,  'x')  # 'x' - no overwrite

      #===========================
      # create HDF5 groups similar to those in the legacy format:

      acq_grp    = f.create_group('/Acquisition')              # /Acquisition
      acq_grp.attrs['run_time'] = time.ctime()                                             # not legacy
      scope_grp  = acq_grp.create_group('LeCroy_scope')        # /Acquisition/LeCroy_scope
      header_grp = scope_grp.create_group('Headers')                                       # not legacy

      ctl_grp    = f.create_group('/Control')                  # /Control
      pos_grp    = ctl_grp.create_group('Positions')           # /Control/Positions

      meta_grp   = f.create_group('/Meta')                     # /Meta                  not legacy
      script_grp = meta_grp.create_group('Python')             # /Meta/Python
      scriptfiles_grp = script_grp.create_group('Files')       # /Meta/Python/Files

      # in the /Meta/Python/Files group:
      for src_file in src_files:
        self.create_sourcefile_dataset(scriptfiles_grp, src_file)

      # I don't know how to get this information from the scope:
      scope_grp.create_dataset('LeCroy_scope_Setup_Arrray', data=numpy.array('Sorry, this is not included', dtype='S'))

      pos_ds = pos_grp.create_dataset('positions_setup_array', data=positions)
      pos_ds.attrs['xpos'] = xpos                                                           # not legacy
      pos_ds.attrs['ypos'] = ypos                                                           # not legacy
      pos_ds.attrs['shotperpos'] = num_duplicate_shots                                      # not legacy

      # create the scope access object, and iterate over positions
      with LeCroy_Scope(self.ip_addrs['scope'], verbose=False) as scope:
        if not scope:
          # I think we have raised an exception if this is the case, so we never get here
          print('Scope not found at '+self.ip_addrs['scope'])
          return

        scope_grp.attrs['ScopeType'] = scope.idn_string

        NPos = len(positions)
        NTimes = scope.max_samples()

        datasets = {}
        hdr_data = {}

        # create 4 default data sets, empty.
        # These will all be populated for compatibility with legacy format hdf5 files.

        # datasets['C1'] = scope_grp.create_dataset('Channel1', shape=(NPos,NTimes), fletcher32=True,
        #                  compression='gzip', compression_opts=9)
        # datasets['C2'] = scope_grp.create_dataset('Channel2', shape=(NPos,NTimes), fletcher32=True,
        #                  compression='gzip', compression_opts=9)
        # datasets['C3'] = scope_grp.create_dataset('Channel3', shape=(NPos,NTimes), fletcher32=True,
        #                  compression='gzip', compression_opts=9)
        # datasets['C4'] = scope_grp.create_dataset('Channel4', shape=(NPos,NTimes), fletcher32=True,
        #                  compression='gzip', compression_opts=9)

        # create other datasets, one for each displayed trace (but not C1-4, which we just did)
        # todo: should we maybe just ignore these?  or have a user option to include them?

        traces = scope.displayed_traces()
        for tr in traces:
          name = scope.expanded_name(tr)
          # ds = scope_grp.create_dataset(name, (NPos,NTimes), chunks=(1,NTimes), fletcher32=True,
          #                               compression='gzip', compression_opts=9)
          ds = scope_grp.create_dataset(name, shape=(NPos,NTimes), fletcher32=True,
                                          compression='gzip', compression_opts=9)
          datasets[tr] = ds

        # For each trace we are storing, we will write one header per position (immediately after
        #    the data for that position has been acquired); these compress to an insignificant size
        # For whatever stupid reason we need to write the header as a binary blob using an "HDF5 opaque" type
        #    - here void type 'V346'  (otherwise I could not manage to avoid invisible string processing and
        #    interpretation)
        for tr in traces:
          name = scope.expanded_name(tr)
          hdr_data[tr] = header_grp.create_dataset(name, shape=(NPos,), dtype="V%i"%(WAVEDESC_SIZE), fletcher32=True,
                                 compression='gzip', compression_opts=9)  # V346 = void type, 346 bytes long

        # create "time" dataset
        time_ds = scope_grp.create_dataset('time', shape=(NTimes,), fletcher32=True, compression='gzip',
                         compression_opts=9)

        # at this point all datasets should be created, so we can
        # switch to SWMR mode
        #?# f.swmr_mode = True    # SWMR MODE: DO NOT CREATE ANY MORE DATASETS AFTER THIS
        #?# check effects of flushing...see above

        try:  # try-catch for Ctrl-C keyboard interrupt
```

```
          ######### BEGIN MAIN ACQUISITION LOOP #########
          print('starting acquisition loop at', time.ctime())
          acquisition_loop_start_time = time.time()

          nowx, nowy = (-999, -999)
          for pos in positions:
            # prevent motor from enabling/disabling when taking data at the same position
            # this stops the motor noise from being picked up by the data in between shots
            if nowx!=pos[1] or nowy!=pos[2]:
              # enable motor
              mc.enable()

              # move to next position
              print('position index =', pos[0], '  x =', pos[1], '  y =', pos[2], end='\n')
              mc.move_to_position(pos[1], pos[2])
              self.signals.updated_position.emit(pos[1], pos[2])
              nowx, nowy = (pos[1], pos[2])
              x_encoder, y_encoder = mc.current_probe_position()
              self.signals.updated_position.emit(x_encoder, y_encoder)

              # Disable the motor current output when taking the data
              mc.disable()


            if pos[0] > 1:
              t_remain = (len(positions) - pos[0]) * (time.time()-acquisition_loop_start_time)/pos[0] / 3600
              print ('Estimated remaining time:%6.2f'%(t_remain))
            else:
              print ('')

            print('------------------', scope.gaaak_count, '-------------------- ',pos[0],sep='')

    #        scope.autoscale('C3')  # for now can only _increase_ the V/div

            # do averaging, and copy scope data for each trace on the screen to the output HDF5 file
            self.acquire_displayed_traces(scope, datasets, hdr_data, pos[0]-1)   # argh the pos[0] index is 1-based

            # Show plot traces on GUI
            try:
              scope.screen_dump()
              self.signals.new_screen_dump.emit()
            # except VisaIOError: #VisaIOError undefined?
            #   print ('Unable to grab screen due to VisaIOError')
            #   continue
            except:
              print ('Unable to grab screen due to unknown Error')
              continue

            self.signals.finished_position.emit(x_encoder, y_encoder)

            # at least get one time array recorded for swmr functions
            if pos[0] == 1:
              time_ds[0:NTimes] = scope.time_array()[0:NTimes]
              #time_ds.flush()

          ######### END MAIN ACQUISITION LOOP #########

        except KeyboardInterrupt:
          print('\n______Halted due to Ctrl-C______', '  at', time.ctime())

        # copy the array of time values, corresponding to the last acquired trace, to the times_dataset
        time_ds[0:NTimes] = scope.time_array()[0:NTimes]      # specify number of points, sometimes scope return extras
        if type(time_ds) == 'stupid':
          print(' this is only included to make the linter happy, otherwise it thinks time_ds is not used')

        # Set any unused datasets to 0 (e.g. any C1-4 that was not acquired); when compressed
        #      they require negligible space
        # Also add the text descriptions.    Do these together to be able to be able to make a note in the description
        for tr in traces:
          if datasets[tr].len() == 0:
            datasets[tr] = numpy.zeros(shape=(NPos,NTimes))
            datasets[tr].attrs['description'] = 'NOT RECORDED: ' + self.get_channel_description(tr)           # callback\
arg to the current function
            datasets[tr].attrs['recorded']    = False
          else:
            datasets[tr].attrs['description'] = self.get_channel_description(tr)                              # callback\
arg to the current function
            datasets[tr].attrs['recorded']    = True
            datasets[tr].attrs['shots per position']    = self.pos_param["num_shots"]


      f.close()  # close the HDF5 file

      self.signals.finished.emit()
      #done



class Test_Shot_Thread(QRunnable):
  '''
  The thread that runs when users click to take a test data shot.
  '''

  def __init__(self, ip_addrs):
    super(Test_Shot_Thread, self).__init__()
    self.signals = Signals()
    self.ip_addrs = ip_addrs

  def acquire_displayed_traces(self, scope):
    # acquire enough sweeps for the averaging, then read displayed scope trace data into HDF5 datasets

    timeout = 2000 # seconds
    timed_out, N = scope.wait_for_max_sweeps('Test shot: ', timeout)  # leaves scope not triggering

    if timed_out:
      print('**** averaging timed out: got '+str(N)+' at %.6g s' % timeout)

    scope.screen_dump()
    self.signals.new_screen_dump.emit()
    scope.set_trigger_mode('NORM')   # resume triggering

  def run(self):
    with LeCroy_Scope(self.ip_addrs['scope'], verbose=False) as scope:
      if not scope:
        # I think we have raised an exception if this is the case, so we never get here
        print('Scope not found at '+self.ip_addrs['scope'])
        return
      self.acquire_displayed_traces(scope)   # argh the pos[0] index is 1-based
    self.signals.finished.emit()

###############################################################################################
###############################################################################################


class Window(QWidget):
  '''
  The main GUI interface
  '''

  def __init__(self):
    super(Window, self).__init__()

    self.pc = Position_Controls()
    self.canvas = MyMplCanvas()
    self.ac = Acquisition_Controls()
    self.axc = Axis_Controls()
    self.sv = Software_Version()
    self.sc = Scope_Channel()
    self.x_ip = "192.168.0.70"
    self.y_ip = "192.168.0.80"
    self.scope_ip = "192.168.0.60"
    self.port_ip = int(7776)
    self.mm = Motor_Movement(x_ip_addr = self.x_ip, y_ip_addr = self.y_ip, MOTOR_PORT = self.port_ip)
    self.mm.set_input_usage(3)
    self.mm.set_steps_per_rev(20000, 20000)

    # when the values in axis control change, trigger axis_change function
    self.axc.xupInput.valueChanged.connect(self.axis_change)
    self.axc.yupInput.valueChanged.connect(self.axis_change)
    self.axc.xlowInput.valueChanged.connect(self.axis_change)
    self.axc.ylowInput.valueChanged.connect(self.axis_change)

    # when users confirms a new set up of positions, trigger update_geometry function
    self.pc.ConfirmButton.clicked.connect(self.update_geometry)

    # when users click data run or test shot, trigger the corresponding thread to run
    self.ac.DataRun.clicked.connect(self.start_data_run)
    self.ac.TestShot.clicked.connect(self.start_test_shot)

    self.ScopeScreen = QLabel(self) # make a QLabel to present scope screenshot
    self.update_screen_dump() # the default will be the last taken screenshot


    layout = QGridLayout()
    layout.addWidget(self.canvas, 0, 0, 1, 2)
    layout.addWidget(self.axc, 1, 0, 1, 2)      #axes control
    layout.addWidget(self.mm, 2, 0, 2, 1)          #motor movement
    layout.addWidget(self.pc, 2, 1, 2, 1)          #position control
    layout.addWidget(self.ac, 2, 2)          #acquisition control
    layout.addWidget(self.sc, 2, 3, 2, 1)          #scope channel comments
    layout.addWidget(self.sv, 3, 2)
    layout.addWidget(self.ScopeScreen, 0, 2 , 2, 2)
    self.setLayout(layout)

    self.setWindowTitle("180E Data Acquisition System for XY Probe Drives")
    self.resize(1600, 700)

    self.threadpool = QThreadPool()

    # Set timer to update current probe position and instant motor velocity
    self.timer = QtCore.QTimer(self)
    self.timer.timeout.connect(self.update_current_position)
    self.timer.start(500)


  def axis_change(self):
    xup = self.axc.xupInput.value()
    yup = self.axc.yupInput.value()
    xlow = self.axc.xlowInput.value()
    ylow = self.axc.ylowInput.value()
    self.canvas.update_axis(xup,yup,xlow,ylow)


  def update_current_position(self):
    # update the current position of the probe when the data run thread is not running
    if data_running == False:
      self.xnow, self.ynow = self.mm.current_probe_position()
      self.canvas.point.remove()
      self.canvas.update_probe(self.xnow, self.ynow)
      self.mm.CurposInput.setText("(" + str(round(self.xnow, 2)) + " ," + str(round(self.ynow, 2)) +")")

    else:
      pass


  def update_current_position_during_data_run(self, xnow, ynow):
    # update the current position of the probe when the data run thread is running
    # xnow, ynow: current position from the info sent back by the thread
    if data_running == True:
      self.xnow = xnow
      self.ynow = ynow
      self.canvas.point.remove()
      self.canvas.update_probe(self.xnow, self.ynow)
      self.mm.CurposInput.setText("(" + str(round(self.xnow, 2)) + " ," + str(round(self.ynow, 2)) +")")
    else:
      print("Why is this called when data_running == False ?")

  def update_screen_dump(self):
    # present the 'scope_screen_dump.png' picture (saved by LeCroy_Scope)
    self.pixmap = QPixmap("scope_screen_dump.png")
    #Rescale the picture to fit the screen. However this makes the picture from a HD scope blurry, so don’t:
    #self.pixmapscaled = self.pixmap.scaledToHeight(800)
    self.ScopeScreen.setPixmap(self.pixmap)

  def mark_finished_positions(self, x, y):
    if data_running == True:
      self.xdone = x
      self.ydone = y
      self.canvas.visited_points.remove()
      self.canvas.finished_positions(self.xdone, self.ydone)
    else:
      print("Why is this called when data_running == False ?")


  def update_current_speed(self):
    # Ask for current motor speed through Motor_Control_2D
    self.speedx, self.speedy = self.mm.ask_velocity()
    self.velocityInput.setText("(" + str(self.speedx) + " ," + str(self.speedy) +")")

  def update_parameters(self):
    # Update the user setup for the data positions
    self.parameters = {}
    self.update = True
    try:
      self.parameters["xmax"] = float(self.pc.xMaxInput.text())
      self.parameters["xmin"] = float(self.pc.xMinInput.text())
      self.parameters["ymax"] = float(self.pc.yMaxInput.text())
      self.parameters["ymin"] = float(self.pc.yMinInput.text())
      self.parameters["nx"] = int(self.pc.nxInput.text())
      self.parameters["ny"] = int(self.pc.nyInput.text())
      return self.parameters
    except ValueError:
      QMessageBox.about(self, "Error", "Position should be valid numbers.")
      self.update = False

  def update_geometry(self):
    self.param = self.update_parameters()

    if self.update == True:
      self.canvas.matrix.remove()
      self.canvas.update_figure(self.param)
    else:
      pass

  def update_channel_information(self):
    self.channel_info = {}

    self.channel_info["C1"] = self.sc.c1Input.text()
    self.channel_info["C2"] = self.sc.c2Input.text()
    self.channel_info["C3"] = self.sc.c3Input.text()
    self.channel_info["C4"] = self.sc.c4Input.text()

    return self.channel_info

  def start_data_run(self):
    # Start data_run threading

    self.hdf5_filename = None

    # Obtain user setup for positions, number of shots and number of run, etc.
    self.pos_param = self.update_parameters()
    self.pos_param["num_shots"] = self.ac.num_shots.value()
    self.pos_param["num_run"] = self.ac.num_run.value()
    self.channel_description = self.update_channel_information()

    self.ip_addrs = {}
    self.ip_addrs['x'] = self.x_ip
    self.ip_addrs['y'] = self.y_ip
    self.ip_addrs['scope'] = self.scope_ip

    # start data run
    self.data_run = Data_Run_Thread(self.hdf5_filename, self.pos_param, self.channel_description, self.ip_addrs)
    self.freeze_all_controls() # Disable all the buttons on GUI
    self.data_run.signals.finished.connect(self.data_run_finished) # Call data_run_finished when signal emits
    self.data_run.signals.cancel.connect(self.acquisition_canceled) # Cancel the run when signal emits
    self.data_run.signals.updated_position.connect(self.update_current_position_during_data_run)
                 # Update probe position when signal emits
    self.data_run.signals.finished_position.connect(self.mark_finished_positions) # Mark position when signal emits
    self.data_run.signals.new_screen_dump.connect(self.update_screen_dump) # Get new scope screenshot
    self.threadpool.start(self.data_run)

  def acquisition_canceled(self):
    QMessageBox.about(self, "Acquisition Status", "Data acquisition cancelled.")
    self.enable_all_controls()


  def data_run_finished(self):
    QMessageBox.about(self, "Acquisition Status", "Data acquisition complete.")
    self.enable_all_controls()
    self.canvas.visited_points.remove() # Clear canvas
    self.canvas.initialize_visited_points() # Initialize a new array for visited positions


  def test_shot_finished(self):
    QMessageBox.about(self, "Take Test Shot", "Test shot is finished.")
    self.enable_all_controls()


  def freeze_all_controls(self):
    # Disable all the buttons on GUI

    global data_running
    data_running = True
    self.pc.setEnabled(False)
    self.ac.setEnabled(False)
    self.sc.setEnabled(False)
    self.mm.MoveButton.setEnabled(False)
    self.mm.SetZero.setEnabled(False)
    self.mm.SetVelocity.setEnabled(False)
    self.mm.velocityButton.setEnabled(False)

  def enable_all_controls(self):
    global data_running
    data_running = False
    self.pc.setEnabled(True)
    self.ac.setEnabled(True)
    self.sc.setEnabled(True)
    self.mm.MoveButton.setEnabled(True)
    self.mm.SetZero.setEnabled(True)
    self.mm.SetVelocity.setEnabled(True)
    self.mm.velocityButton.setEnabled(True)


  def start_test_shot(self):
    self.ip_addrs = {}
    self.ip_addrs['scope'] = self.scope_ip
    self.test_shot = Test_Shot_Thread(self.ip_addrs)
    self.test_shot.signals.finished.connect(self.test_shot_finished)
    self.test_shot.signals.new_screen_dump.connect(self.update_screen_dump)
    self.threadpool.start(self.test_shot)


  def fileQuit(self):
    self.close()

  def closeEvent(self, ce):
    self.fileQuit()



if __name__ == '__main__':

  app = QApplication(sys.argv)
  window = Window()
  window.show()

  sys.exit(app.exec_())
```

```
"""
This file should be named as LeCroy_Scope.py

This file defines the class that implements communication with a LeCroy X-Stream scope.
It expects National Instruments Visa to be installed on the computer, and pyvisa to be installed in the Python library.

See inline comments for the function of the member functions.

LeCroy Header interpretation is based on PP's previous internet scrapings

Created on Wed Aug 31, 2016

@author: Patrick Pribyl

PyVisa documentation:                                   https://media.readthedocs.org/pdf/pyvisa/1.6/pyvisa.pdf
LeCroy Automation Command Reference Manual:
                                       http://cdn.teledynelecroy.com/files/manuals/automation_command_ref_manual_ws.pdf
LeCroy Remote Control Manual for "X-Stream" scopes:  http://cdn.teledynelecroy.com/files/manuals/wm-rcm-e_rev_d.pdf
National Instruments Visa at                          http://www.ni.com/download/ni-visa-16.0/6184/en/     (in Aug 2016)
NOTE: PyVisa FAQ Points to an old version
    setting up the LeCroy scope "Passport":           http://forums.ni.com/ni/attachments/ni/170/579106/1/VICP-NI-MAX.doc
LeCroy "passport" for NI-Visa:                        http://teledynelecroy.com/support/softwaredownload/home.aspx

Anaconda:
Installing pyvisa from the conda-forge channel can be achieved by adding conda-forge to your channels with:
  c:\>conda config --add channels conda-forge
Once the conda-forge channel has been enabled, pyvisa can be installed with:
  c:\>conda install pyvisa

other notes:
NI VISA Manuals:
  NI-VISA User Manual                   http://digital.ni.com/manuals.nsf/websearch/266526277DFF74F786256ADC0065C50C
  NI-VISA Programmer Reference Manual   http://digital.ni.com/manuals.nsf/websearch/87E52268CF9ACCEE86256D0F006E860D

"""

import numpy
import visa
from pyvisa.resources import MessageBasedResource
from pyvisa.errors import VisaIOError
import collections
import struct
import sys
import pylab as plt
import matplotlib.image as mpimg
import time

# the header recorded for each trace contains 63 entries, and occupies 346 bytes

WAVEDESC = collections.namedtuple('WAVEDESC',
['descriptor_name', 'template_name', 'comm_type', 'comm_order',
 'wave_descriptor', 'user_text', 'res_desc1', 'trigtime_array', 'ris_time_array',
 'res_array1', 'wave_array_1', 'wave_array_2', 'res_array2', 'res_array3',
 'instrument_name', 'instrument_number', 'trace_label', 'reserved1', 'reserved2',
 'wave_array_count', 'pnts_per_screen', 'first_valid_pnt', 'last_valid_pnt',
 'first_point', 'sparsing_factor', 'segment_index', 'subarray_count', 'sweeps_per_acq',
 'points_per_pair', 'pair_offset', 'vertical_gain', 'vertical_offset', 'max_value',
 'min_value', 'nominal_bits', 'nom_subarray_count', 'horiz_interval', 'horiz_offset',
 'pixel_offset', 'vertunit', 'horunit', 'horiz_uncertainty',
 'tt_second', 'tt_minute', 'tt_hours', 'tt_days', 'tt_months', 'tt_year', 'tt_unused',
 'acq_duration', 'record_type', 'processing_done', 'reserved5', 'ris_sweeps',
 'timebase', 'vert_coupling', 'probe_att', 'fixed_vert_gain', 'bandwidth_limit',
 'vertical_vernier', 'acq_vert_offset', 'wave_source'])

WAVEDESC_SIZE = 346
"""
 The header should be 346 bytes (with correct packing); it is preceded by 15 bytes for the def9header etc.
 note: for simplicity I expanded the leCroy_time struct into explicit fields above, labeled tt_xxx

 To get floating values from the stored raw data: y[i] = vertical_gain * data[i] - vertical_offset

 some entries:
   horiz_offset:    Seconds between trigger and first data point (note this is <= 0 if trigger is visible on screen)
   bandwidth_limit: 0 = off, 1 = on
   record_type:     see below
   processing_done: see below
   timebase:        see below
   fixed_vert_gain: see below
   vert_coupling:   see below
   wave_source:     0=CH1, 1=CH2, 2=CH3, 3=CH4, 9=Unknown
"""
WAVEDESC_FMT = '=16s16shhllllllllll16sl16shhllllllllllhhffffhhfdd48s48sfdBBBBhhfhhhhhhfhhffh'
#    The initial '=' character specifies native byte order, with standard (C) alignment.

RECORD_TYPES = ['single_sweep', 'interleaved', 'histogram', 'graph', 'filter_coefficient',
                'complex', 'extrema', 'sequence_obsolete', 'centered_RIS', 'peak_detect']

PROCESSING_TYPES = ['no_processing', 'fir_filter', 'interpolated', 'sparsed',
                    'autoscaled', 'no_result', 'rolling', 'cumulative']

TIMEBASE_IDS = ['1 ps', '2 ps', '5 ps', '10 ps', '20 ps', '50 ps', '100 ps', '200 ps', '500 ps',
                '1 ns', '2 ns', '5 ns', '10 ns', '20 ns', '50 ns', '100 ns', '200 ns', '500 ns',
                '1 us', '2 us', '5 us', '10 us', '20 us', '50 us', '100 us', '200 us', '500 us',
                '1 ms', '2 ms', '5 ms', '10 ms', '20 ms', '50 ms', '100 ms', '200 ms', '500 ms',
                '1 s',  '2 s',  '5 s',  '10 s',  '20 s',  '50 s',  '100 s',  '200 s',  '500 s',
                '1 ks', '2 ks', '5 ks']   # these are per division; ALSO: 100 corresponds to EXTERNAL

VERT_GAIN_IDS = ['1 uV', '2 uV', '5 uV', '10 uV', '20 uV', '50 uV', '100 uV', '200 uV', '500 uV',
                 '1 mV', '2 mV', '5 mV', '10 mV', '20 mV', '50 mV', '100 mV', '200 mV', '500 mV',
                 '1 V',  '2 V',  '5 V',  '10 V',  '20 V',  '50 V',  '100 V',  '200 V',  '500 V',
                 '1 kV', '2 kV', '5 kV', '10 kV']   # these are per division; pp added the last 3

VERT_COUPLINGS = ['DC 50 Ohms', 'ground', 'DC 1 MOhm', 'ground', 'AC 1 MOhm']

EXPANDED_TRACE_NAMES = {'F1': 'Math1'   , 'F2': 'Math2'   , 'F3': 'Math3'   , 'F4': 'Math4'   ,
                        'F5': 'Math5'   , 'F6': 'Math6'   , 'F7': 'Math7'   , 'F8': 'Math8'   ,
                        'TA': 'ChannelA', 'TB': 'ChannelB', 'TC': 'ChannelC', 'TD': 'ChannelD',
                        'M1': 'Memory1' , 'M2': 'Memory2' , 'M3': 'Memory3' , 'M4': 'Memory4' ,
                        'C1': 'Channel1', 'C2': 'Channel2', 'C3': 'Channel3', 'C4': 'Channel4' }
# note: the documentation indicates these are possible, but some of them result in errors
KNOWN_TRACE_NAMES = sorted(list(EXPANDED_TRACE_NAMES.keys()))

class LeCroy_Scope:
  """ implements communication with a LeCroy X-Stream scope """
  scope     = None        # the common scope instance
  rm        = None        # the common resource manager instance
  rm_status = False
  valid_trace_names = ()  # list of trace names recognized by the scope (filled in on first call)
  gaaak_count = 0         # peculiar error described below (see wait_for_sweeps())
  idn_string = ''         # scope *idn response
  trace_bytes = numpy.zeros(shape=(WAVEDESC_SIZE), dtype='b')   # buffer for trace data; set to the correct size later
  offscale_fraction = .005 # fraction of off-scale samples that results in auto-scale rescaling

  def __init__(self, ipv4_addr, verbose=True, timeout=5000):
    """ Opens the NI-VISA resource manager, then attempts to open the resource 'VICP::'+ipv4_addr+'::INSTR'.
      The resource-manager-open function rm_open() verifies that the scope is communicating.
      Checks with the scope to determine valid trace names (which often causes the scope to beep due to
      queries about invalid names; this is useful to confirm that communication is established, when it
      happens).
    """
    self.verbose = verbose
    # use resource manager to open 'VICP::'+ipv4_addr+'::INSTR';
    self.rm_status = self.rm_open(ipv4_addr)   # also assigns self.scope to this "instrument"
    if not self.rm_status:
      err = '**** program exiting'
      raise(RuntimeError(err))
    self.scope.timeout    = timeout
    self.scope.chunk_size = 1000000
    self.scope.write('COMM_HEADER OFF')

    if len(self.valid_trace_names) == 0:
      for tr in KNOWN_TRACE_NAMES:
        self.scope.write(tr+':TRACE?')
          # this makes a characteristic set of beeps on the scope, since it fails for several of the entries in the list
        self.scope.write('CMR?')           # read (and clear) the Command Status Register to check for errors
        error_code = int(self.scope.read())
        if error_code == 0:
          self.valid_trace_names += (tr,)  # no error, assume ok

  def __repr__(self):
    """ return a printable version: not a useful function """
    return self.scope.__repr__()


  def __str__(self):
    """ return a string representation: slightly less useless than __repr__(), but still not useful """
    txt = self.scope.__repr__() + '\n'
    trs = self.displayed_traces()
    for tr in trs:
      txt += self.scope.query(tr+':VOLT_DIV?')
    txt += self.scope.query('TIME_DIV?')
    txt += self.scope.query('VBS? "return=app.Acquisition.Horizontal.NumPoints"')
    return txt

  def __bool__(self):
    """ boolean test if valid - assumes valid if the resource manager status is True """
    return self.rm_status

  def __enter__(self):
    """ no special processing after __init__() """
    return self

  def __exit__(self, exc_type, exc_value, traceback):
    """ checks for how many times the peculiar error described below was detected (see wait_for_sweeps()),
      then calls __del__() """
    print('LeCroy_Scope:__exit__() called', end='')
    if self.gaaak_count != 0:
      print(' with', self.gaaak_count, '"gaaak" type errors', end='')
    print('  at', time.ctime())
    self.__del__()

  def __del__(self):
    """ cleanup: close the scope resource, close the resource manager """
    if self.scope != None:
      self.scope.close()
      self.scope = None
    if self.rm != None:
      self.rm.close()
      self.rm = None
    self.rm_status = False
```

```
#---------------------------------------------------------------------------

def rm_list_resources(self):
  """ this is a very slow process --AND-- LeCroy scopes using VISA Passport do not show up in this list, anyway """
  if self.verbose: print('<:> searching for VISA resources')
  t0 = time.time()
  self.rm.list_resources()
  t1 = time.time()
  if self.verbose and (t1-t0 > 1): print('    ............................%6.3g sec' % (t1-t0))

#---------------------------------------------------------------------------
#todo: how to find VICP address automatically?

def rm_open(self, ipv4_addr)  -> bool:
  """ open the NI-VISA resource manager
    then open the scope resource 'VICP::'+ipv4_addr+'::INSTR'
    once open, attempt to communicate with the scope
    throw an exception if any of the above fails
    eventually we need to call rm_close()
  """
  if self.rm != None:
    return True

  if self.verbose: print('<:> constructing resource manager')
  t0 = time.time()
  self.rm = visa.ResourceManager()
  t1 = time.time()
  if self.verbose and (t1-t0 > 1): print('    ............................%6.3g sec' % (t1-t0), end='')

  if self.verbose: print('<:> attempting to open resource VICP::'+ipv4_addr+'::INSTR')

  # attempt to open a connection to the scope
  try:
    self.scope = self.rm.open_resource('VICP::'+ipv4_addr+'::INSTR', resource_pyclass=MessageBasedResource)
    print('...ok')
  except Exception:
    print('\n**** Scope not found at "', ipv4_addr, '"\n')
    #return False,0,0
    raise

  # send a (standard) *IDN? query as a way of testing whether we have a scope:
  try:
    self.idn_string = self.scope.query('*IDN?')
    if self.verbose: print('<:>', self.idn_string)  # returns scope type, name, version info
  except Exception:
    print('\n**** Scope at "', ipv4_addr,'" did not respond to "*IDN?" query\n')
    self.rm.close()
    return False,0,0
  return True

#---------------------------------------------------------------------------

def rm_close(self):
  """ close the resource manager; should eventually be called any time rm_open is called """
  if self.rm != None:
    self.rm.close()
    self.rm = None

#---------------------------------------------------------------------------

def screen_dump(self, white_background = False, png_fn = 'scope_screen_dump.png', full_screen = True):
  """ obtain a screen dump from the scope, in the form of a .png file
    write the file with filenam png_fn (=argument)
    read the file and display it on the screen using matplotlib imshow() function
  """
  if white_background:
    bckg = 'WHITE'
  else:
    bckg = 'BLACK'
  if full_screen:
    area = 'DSOWINDOW'
  else:
    area = 'GRIDAREAONLY'
  # write "hardcopy" setup information:
  self.scope.write('COMM_HEADER OFF')
  self.scope.write('HARDCOPY_SETUP DEV, PNG, BCKG, '+bckg+', DEST, "REMOTE", AREA, '+area)
  # send screen dump command
  self.scope.write('SCREEN_DUMP')
  # read screen dump information: this is exactly the contents of a .png file, typically < 40 kB
  screen_image_png = self.scope.read_raw()
  # write the .png file
  file = open(png_fn, 'wb')      # Can this be achieved without having to go to disk??
  # actually, this is not a bug, it's a feature, since we probably want the image in a file anyway
  file.write(screen_image_png)
  file.close()
  # x = mpimg.imread(png_fn)
  # (h,w,d) = numpy.shape(x)
  # plt.figure(num=None, figsize=(w/100, h/100), dpi=100, facecolor='w', edgecolor='k')
  # plt.subplots_adjust(left=0.0, right=1.0, bottom=0.0, top=1.0)
  # plt.imshow(x)

#---------------------------------------------------------------------------

def write_status_msg(self, msg):
  """ send a message to the status line on the scope;
      nominally this should be < 50 chars
  """
  if len(msg) > 49:   # specs say 49 chars max: todo: is this still the limit?
    self.scope.write('MESSAGE "'+msg[0:46]+'..."')
  else:
    self.scope.write('MESSAGE "'+msg+'"')

#--------------------------------------------------------------------------

def validate_channel(self, Cn)  -> str:
  """ convenience function, returns canonical channel label, C1, C2, C3, or C4
    works correctly if Cn is a string or integer
    throws a runtime error if the argument is not a proper channel label
  """
  # channel should be 'C1','C2','C3','C4' (as opposed to the more general trace labels)
  if type(Cn) == str and (Cn == 'C1' or Cn == 'C2' or Cn == 'C3' or Cn == 'C4'):
    return Cn
  if type(Cn) == int and (Cn >= 1 and Cn <= 4):
    return 'C'+str(Cn)
  err = '**** validate_channel(): channel = "' + Cn + '" is not allowed, must be C1-4'
  raise(RuntimeError(err)).with_traceback(sys.exc_info()[2])

#--------------------------------------------------------------------------

def validate_trace(self, tr)  -> str:
  """ convenience function, returns canonical trace label, which is broader than a channel label
    see valid_trace_names defined at top of file
    if Cn is an integer, assumes we want a channel name
    throws a runtime error if the argument is not a proper trace label
  """
  if type(tr) == int and (tr >= 1 and tr <= 4):
    return 'C'+str(tr)
  for trn in self.valid_trace_names:
    if tr == trn:
      return trn
  err = '**** validate_trace(): trace name "' + tr + '" is unknown'
  raise(RuntimeError(err)).with_traceback(sys.exc_info()[2])

#--------------------------------------------------------------------------

def max_samples(self, N = 0) -> int:
  """ mostly used for determining the number of samples the scope expects to acquire.
    If the argument N is given, this routine can also be used to attempt to set the number of samples
    to one of the following:
          500, 1000, 2500, 5000, 10000, 25000, 50000, 100000, 250000, 500000, etc.
    except many scopes won't accept all of these
    The return value is the actual number of samples that the scope will acquire
  """
  if N > 0:
    self.scope.write('VBS "app.Acquisition.Horizontal.MaxSamples='+str(N)+'"')
  # find out what happened:
  return int(self.scope.query('VBS? "return=app.Acquisition.Horizontal.NumPoints"'))

#--------------------------------------------------------------------------


def displayed_channels(self)  -> ():    # returns a tuple of channel names, e.g. ('C1', 'C4')
  """ return displayed CHANNELS only, ignoring math, memory, etc """
  channels = ()
  self.scope.write('COMM_HEADER OFF')

  if self.scope.query('C1:TRACE?')[0:2] == 'ON':
    channels += ('C1',)
  if self.scope.query('C2:TRACE?')[0:2] == 'ON':
    channels += ('C2',)
  if self.scope.query('C3:TRACE?')[0:2] == 'ON':
    channels += ('C3',)
  if self.scope.query('C4:TRACE?')[0:2] == 'ON':
    channels += ('C4',)
  return channels


def displayed_traces(self)  -> ():    # returns a tuple of trace names, e.g. ('C1', 'C4', 'F1')
  """ return displayed TRACES, including math, memory, etc. """
  traces = ()
  self.scope.write('COMM_HEADER OFF')

  for tr in self.valid_trace_names:
    if self.scope.query(tr+':TRACE?')[0:2] == 'ON':
      traces += (tr,)
  return traces

#--------------------------------------------------------------------------

def vertical_scale(self, trace) -> float:
  """ get vertical scale setting for the trace
  """
  Tn = self.validate_trace(trace)
  scale = float(self.scope.query('VBS? "Return=app.Acquisition.'+Tn+'.VerScale"'))
  return scale


def set_vertical_scale(self, trace, scale) -> float:
  """ set vertical scale setting for the trace
  """
  Tn = self.validate_trace(trace)
  self.scope.write('VBS "app.Acquisition.'+Tn+'.VerScaleVariable=True"')
  self.scope.write('VBS "app.Acquisition.'+Tn+'.VerScale='+str(scale)+'"')
  return self.vertical_scale(trace)   # it may not be what we asked for

#--------------------------------------------------------------------------

def averaging_count(self, channel='C1') -> int:
  """ get count of averages specified for the channel, default = read from channel 'C1'
  """
  Cn = self.validate_channel(channel)
  NSweeps = int(self.scope.query('VBS? "Return=app.Acquisition.'+Cn+'.AverageSweeps"'))
  #todo: should this deal with traces rather than channels?
  return NSweeps


def set_averaging_count(self, channel='C1', NSweeps=1):
  """ set count of averages for a given channel (not used)
  """
  Cn = self.validate_channel(channel)
  if NSweeps < 1:
    NSweeps = 1
  if NSweeps > 1000000:
    NSweeps = 1000000
  self.scope.write('VBS "app.Acquisition.'+Cn+'.AverageSweeps='+str(NSweeps)+'"')


def max_averaging_count(self) -> (int,int):
  """ get maximum averaging count across all displayed channels
    returns #sweeps and corresponding channel. To display progress, use
    self.averaging_count(cc) where cc is the returned channel
  """
  NSweeps = 0
  ach = None
  for ch in self.displayed_channels():
    n = self.averaging_count(ch)
    if n > NSweeps:
      NSweeps = n
      ach = ch
  if ach == None:
    # throw an exception if no channels had sweeps
    err = '**** max_averaging_count(): no displayed channels'
    raise(RuntimeError(err)).with_traceback(sys.exc_info()[2])
  return NSweeps, ach

#--------------------------------------------------------------------------

def wait_for_max_sweeps(self, aux_text='', timeout=100)  -> (bool,int):
  """ determine maximum averaging count across all displayed channels, then wait for that many sweeps
  """
  NSweeps, ach = self.max_averaging_count()
  self.write_status_msg(aux_text + 'Waiting for averaging('+str(NSweeps)+') to complete')

  timed_out,N = self.wait_for_sweeps(ach, NSweeps, timeout)

  if timed_out:
    msg = 'averaging('+str(NSweeps)+') timed out: '+str(N)+' at %.6g s' % timeout
  else:
    msg = 'averaging('+str(NSweeps)+'), completed, got '+str(N)
  self.write_status_msg(aux_text + msg)
  return timed_out,N

def wait_for_sweeps(self, channel, NSweeps, timeout=100, sleep_interval=0.1) -> (bool,int):
  """ Worker for above: wait for a given channel to trigger NSweeps times
    This polls the scope to determine number of sweeps that have occurred, so may overshoot
    (to get faster polling, set sleep_interval(seconds) to a smaller number)
  """
  #todo: instead of a timeout, generate a linear fit to sweeps per second, then fail if t_dropout > 6*sigma delayed
  #      In addition, it would be possible to actually project when the process will complete, and
  #      stop at nearly the exact time.  Then take an extra sweep if necessary. If we cared.
  channel = self.validate_channel(channel)

  print_interval = 3  #seconds

  #17-07-11 self.scope.write('TRIG_MODE AUTO')   # try to make sure it is triggering
  self.set_trigger_mode('AUTO')
  self.scope.write('CLEAR_SWEEPS')     # clear sweeps
  self.set_trigger_mode('NORM')
  time.sleep(0.05)  # 17-07-11 sometimes is not clearing sweeps
  self.scope.write('COMM_FORMAT DEF9,BYTE,BIN')             # set byte data transfer
  self.scope.write('WAVEFORM_SETUP SP,0,NP,1,FP,1,SN,0')    # read 1 data points
  self.scope.write('COMM_HEADER OFF')

  # Some time is apparently required to allow for the scope to propagate the requested
  #   settings through to the hardware. This matters at the beginning of polling after
  #   CLEAR_SWEEPS should set have the number to 0. (Aug 2016 - Tested on LeCroy HDO4104)
  # Eliminating this delay causes intermittent "gaaak" fails as per below
  time.sleep(0.25)
  #    0.1 second still results in a gaaak remediation maybe 0.3% of the time (in verbose mode),
  #    and 0.01% (1e-4) in non-verbose mode
  #    2017-07-11 - 0.1 second -> 600 gaaak errors out of 1200 shots; changed to 0.25, much more infrequent (1/1800)
  t = time.time()
  timeout += t
  next_print_time = t+print_interval
```

```
    if self.verbose: print('<:> waiting for averaging to complete')

    timed_out = True
    gaaak = 0
    while time.time() < timeout:

      #  -----------  bug 2  ----------------
      # scope goes crazy: says "Processing" for maybe 20 seconds, then comes back to life
      # pyvisa.errors.VisaIOError: VI_ERROR_TMO (-1073807339): Timeout expired before operation completed.
      #   so I added this pyvisa_error_count try-except business:

      pyvisa_error_count = 0
      while pyvisa_error_count < 99:
        # 17-07-13 this is failing on 12.5MS scope -> (stupid scope) error; stop-trigger by hand fixes problem somehow
        time.sleep(sleep_interval)
        t0 = time.time()
        try:
          #print("wait_for_sweeps(): attempting to read waveform data")
          self.scope.write(channel+':WAVEFORM?')              ### this is all we really want to do here:
          hdr_bytes = self.scope.read_raw()                   ###    get the scope waveform data
          break                                               ###    and stop trying
        except VisaIOError as err:
          pyvisa_error_count += 1
          if pyvisa_error_count > 98:
            raise
          print('pyvisa.errors.VisaIOError:',err, '  at', time.ctime())   # todo add line# information
          print('(stupid scope)')
          for c in "will try again.":
            print(c,end='',flush=True)
            time.sleep(0.33333)
          print(" now (", pyvisa_error_count, ")",sep='')
          timeout += time.time()-t0

      # self.scope.read_raw()    -----------  bug 1  ----------------
      # NOTE: THERE IS SOMETHING SLIGHTLY OFF ABOUT THE SCOPE PROCESSING HERE
      #       The next value of sweeps_per_acq CAN BE WRONG
      #             -- I discovered this because it occasionally registers as > NSweeps on the first time through
      # NOTE2: delay AND verbose=False reduces the number of errors to ~ 1/10000, so there is possibly a
      #        communications interaction problem, rather than simply a scope issue
      # At any rate, we detect this and tail-recurse to try again if we find the problem occurred

      # desired field is a long int at offset 148 in the header;
      #    note: there 15 bytes of non-header at beginning of buffer
      sweeps_per_acq = struct.unpack('=l', hdr_bytes[15+148:15+148+4])[0]   # note: struct.unpack returns a tuple

      #print("wait_for_sweeps(): sweeps_per_acq = ",sweeps_per_acq)
      gaaak = sweeps_per_acq   # for catching error, below
      if sweeps_per_acq >= NSweeps:
        #print('done waiting because sweeps_per_acq =', sweeps_per_acq,'/',NSweeps)
        timed_out = False
        break
      if time.time() > next_print_time:
        next_print_time += print_interval
        if self.verbose: print(sweeps_per_acq, '/', NSweeps)
      if self.verbose: print('.', sep='', end='', flush=True)

    #17-07-11 self.scope.write('TRIG_MODE STOP')  # stop triggering
    self.set_trigger_mode('STOP')

    # get final number after we stop triggering:
    self.scope.write(channel+':WAVEFORM?')
    hdr_bytes = self.scope.read_raw()
    sweeps_per_acq = struct.unpack('=l', hdr_bytes[15+148:15+148+4])[0]

    if gaaak > sweeps_per_acq:      # check for scope error described above
      self.gaaak_count += 1
      print('=o=o=o=o=o=o=o================================================gaaak, read', gaaak, 'then', sweeps_per_acq)
      return self.wait_for_sweeps(channel, NSweeps, timeout, sleep_interval)  # tail recurse to try again

    if self.verbose: print(sweeps_per_acq, '/', NSweeps)
    return timed_out, sweeps_per_acq

  #--------------------------------------------------------------------------



  def acquire(self, trace, raw=False)  -> numpy.array:
    """ Read a trace from the scope, and return a numpy array of floats corresponding to the data displayed.
      Saves the header.
      if raw==True returns the raw word or byte data, otherwise floating point values
      Uses the current header information to compute the floats from the returned raw data, which can
         be either short integers (16 bits signed), or signed chars (8 bits signed). However the latter
         should not happen because we specify the 2-byte version in the COM_FORMAT command about 5 lines
         down from here.
    """
    trace = self.validate_trace(trace)

    #waveform_setup:   SP=NP=0 -> send all points, for first point FP=1, segment# SN=0 - send all segments
    self.scope.write('WAVEFORM_SETUP SP,0,NP,0,FP,1,SN,0')
    #no header, WORD length data, binary
    self.scope.write('COMM_HEADER OFF')
    self.scope.write('COMM_FORMAT DEF9,WORD,BIN')

    # read raw data from scope
    if self.verbose: print('\n<:> reading',trace,'from scope')
    t0 = time.time()

    self.scope.write(trace+':WAVEFORM?')
    self.trace_bytes = self.scope.read_raw()

    t1 = time.time()
    if self.verbose and (t1-t0 > 1): print('    .............................%6.3g sec' % (t1-t0))

    # parse the header:
    # Note: The first 15 bytes are not part of the WAVEDESC header, as determined by inspection of the data

    self.hdr = WAVEDESC._make(struct.unpack(WAVEDESC_FMT, self.trace_bytes[15:15+WAVEDESC_SIZE]))

    NSamples = int(0)
    if self.hdr.comm_type == 0:
      # data returned as signed chars
      NSamples = self.hdr.wave_array_1
    elif self.hdr.comm_type == 1:
      # data returned as shorts
      NSamples = int(self.hdr.wave_array_1/2)
    else:
      # throw an exception if we don't recognize comm_type
      err = '**** hdr.comm_type = ' + str(self.hdr.comm_type) + '; expected value is either 0 or 1'
      raise(RuntimeError(err)).with_traceback(sys.exc_info()[2])

    if self.verbose: print('<:> NSamples =',NSamples)
    if NSamples == 0:
      # throw an exception if there are no samples (i.e. scope not triggered, trace not displayed)
      err = '**** fail because NSamples = 0 (possible cause: trace has no data? scope not triggered?)' +
                  '\nIF SCOPE IS IN 2-CHANNEL MODE BUT CHANNEL 1 or 4 ARE SELECTED, they have no data'
      raise(RuntimeError(err)).with_traceback(sys.exc_info()[2])

    if self.verbose:
      print('<:> record type:      ', RECORD_TYPES[self.hdr.record_type])
      print('<:> timebase:         ', TIMEBASE_IDS[self.hdr.timebase], 'per div')
      print('<:> vertical gain:    ', VERT_GAIN_IDS[self.hdr.fixed_vert_gain], 'per div')
      print('<:> vertical coupling:', VERT_COUPLINGS[self.hdr.vert_coupling])
      print('<:> processing:       ', PROCESSING_TYPES[self.hdr.processing_done])
      print('<:> #sweeps:          ', self.hdr.sweeps_per_acq)
      print('<:> enob:             ', self.hdr.nominal_bits)
      vert_units = str(self.hdr.vertunit).split('\\x00')[0][2:]    # for whatever reason this prepends "b'" to string
      horz_units = str(self.hdr.horunit).split('\\x00')[0][2:]     # so ignore first 2 chars
      print('<:> data scaling      gain = %6.3g, offset = %8.5g' % (self.hdr.vertical_gain, self.hdr.vertical_offset),
vert_units)
      print('<:> sample timing       dt = %6.3g,   offset = %8.5g' % (self.hdr.horiz_interval, self.hdr.horiz_offset),
horz_units)

    # compute the data values

    if self.verbose: print('<:> computing data values')
    t0 = time.time()   # accurate to about 1 ms on my windows 8.1 system

    ndx0 = (15+WAVEDESC_SIZE) + self.hdr.user_text + self.hdr.trigtime_array + self.hdr.ris_time_array +
self.hdr.res_array1

    if self.hdr.comm_type == 1:       # data returned in words (short integers)
      ndx1 = ndx0 + NSamples*2
      wdata = struct.unpack(str(NSamples)+'h', self.trace_bytes[ndx0:ndx1])              # unpack returns a tuple, so
      if raw:
        data = wdata
      else:
        data = numpy.array(wdata) * self.hdr.vertical_gain - self.hdr.vertical_offset      # we need to convert tuple to
array in order to work with it
    if self.hdr.comm_type == 0:       # data returned in bytes (signed char)
      ndx1 = ndx0 + NSamples
      cdata = struct.unpack(str(NSamples)+'b', self.trace_bytes[ndx0:ndx1])              # unpack returns a tuple
      if raw:
        data = cdata
      else:
        data = numpy.array(cdata) * self.hdr.vertical_gain - self.hdr.vertical_offset

    t1 = time.time()

    if self.verbose and (t1-t0 > 1): print('    .............................%6.3g sec' % (t1-t0))
    return data

  #--------------------------------------------------------------------------

  def time_array(self) -> numpy.array:
    """ return a numpy array containing sample times
      note: only valid after a call to acquire
    """
    NSamples = int(0)
    if self.hdr.comm_type == 0:
      NSamples = self.hdr.wave_array_1             # data returned as signed chars
    elif self.hdr.comm_type == 1:
      NSamples = int(self.hdr.wave_array_1/2)      # data returned as shorts
    t0 = self.hdr.horiz_offset
    return numpy.linspace(t0, t0+NSamples*self.hdr.horiz_interval, NSamples, endpoint=False)
    #note on linspace construction here: suppose we have 2 samples and the trace is 10ms, the samples should be at 0 and
5 ms,
    #                                     rather than 0 and 10ms as linspace(0,N*dt,N) would return
    # Assume this is the case, because when requesting 10000 samples the scope actually returns 10001.
    #    todo: test this, e.g. sample 1 kHz with 1000 pts, look at aliasing. Need the 1 kHz to be referenced to same
frequency as scope

  #--------------------------------------------------------------------------

  def set_trigger_mode(self, trigger_mode)  -> str:
    """ set the scope trigger mode to: 'AUTO', 'NORM', 'SINGLE', or 'STOP'
      if the argument is not one of these, does not change trigger mode
    """
    self.scope.write('COMM_HEADER OFF')
    # prev_trigger_mode = self.scope.query('TRIG_MODE?') #This oftentimes causes time out error
    if trigger_mode == 'AUTO':
      self.scope.write('TRIG_MODE AUTO')
    elif trigger_mode == 'NORM':
      self.scope.write('TRIG_MODE NORM')
    elif trigger_mode == 'SINGLE':
      self.scope.write('TRIG_MODE SINGLE')
    elif trigger_mode == 'STOP':
      self.scope.write('TRIG_MODE STOP')
    else:
      print('set_trigger_mode function receives trigger commands other than AUTO, NORM, SINGLE or STOP.')

    # for i in range(25):   #17-07-11 added verification
    #   txt = self.scope.query('TRIG_MODE?')
    #   if txt[0:3] == trigger_mode[0:3]:   # '\n' stuck on end it seems
    #     break
    #   print('set_trigger_mode(',trigger_mode,')    attempt',i,':  TRIG_MODE is',txt)
    #   time.sleep(0.1)

    # return prev_trigger_mode

  #--------------------------------------------------------------------------

  def expanded_name(self, tr) -> str:
    """ Returns a long version of a trace name; e.g. C1 -> Channel1,  F2 -> Math2, etc """
    if tr in EXPANDED_TRACE_NAMES.keys():
      return EXPANDED_TRACE_NAMES[tr]
    return "unknown_trace_name"

  #--------------------------------------------------------------------------

  def header_bytes(self) -> numpy.array:
    """ return a numpy byte array containing the header """
    #invalid literal for int():  return numpy.array(self.trace_bytes[15:15+WAVEDESC_SIZE], dtype='B')
    return self.trace_bytes[15:15+WAVEDESC_SIZE]

  #--------------------------------------------------------------------------

  def dumtest(self):
    r1 = self.scope.query('PANEL_SETUP?')
    self.scope.write('*SAV 1')    # save entire front panel state in nonvolatile #1
    print(len(r1))

    self.scope.write('VBS app.SaveRecall.Setup.PanelFilename="REMOTE"')
    r2 = self.scope.query('app.SaveRecall.Setup.DoSavePanel')
    print(len(r2))

  #--------------------------------------------------------------------------

  def autoscale(self, trace):
    averaging_count = self.averaging_count(trace)
    self.set_averaging_count(trace, 1)
    time.sleep(0.05)

    #print("autoscale() called")

    status = False
    while True:

      #print("wait for sweeps, trace ", trace)
      self.wait_for_sweeps(trace, 1, timeout=100, sleep_interval=.1)

      #print("acquire")
      data = self.acquire(trace, True)     # read raw scope data

      #print("trig mode normal")
      self.scope.write('TRIG_MODE NORM')   # try to make sure it is triggering

      #print("max =",self.hdr.min_value, "min =",self.hdr.max_value)         # edges of the grid

      delta = (self.hdr.max_value - self.hdr.min_value) * self.offscale_fraction
      #print("delta=",delta)

      bins=(-2**15, self.hdr.min_value+delta, self.hdr.max_value-delta, 2**15-1)
      if bins[0] == bins[1]:
        bins[1] = bins[0]+1   # make bottom bin has finite width
      if bins[2] == bins[3]:
        bins[2] = bins[3]-1   # make sure top bin has finite width
      # 3 bins: low, ok, high

      hist,e = numpy.histogram(data, bins=bins)

      if self.verbose:
        print('hist=',hist, end="     ")
        print('edges=',e)
      NSamples = numpy.size(data)
      if hist[1] > (1-self.offscale_fraction)*NSamples:   # e.g. we want 99% of all samples in the OK bin
        status = True
        break

      # if here, more than offscale_fraction of the samples are defined as "saturated" (d >.98*max or d <.98*min)
      scale = self.vertical_scale(trace)
      #print('scale=',scale,end=' -> ')

      # for now limit = 1 V/div as per 50ohm setting, todo: fix
      if scale == 1:
        print("autoscale(): can't go any larger than 1V/div on 50ohm setting")
        status = False
        break

      scale *= 1.41421356237   # sqrt(2)
      if scale > 1:
        scale = 1

      scale = self.set_vertical_scale(trace, scale)
      print("                                 autoscale(",trace,"): setting vertical scale = ",scale, sep='')

      # now loop to try again

    #print("resetting averaging count for", trace, "and returning status =", status)
    self.set_averaging_count(trace, averaging_count)
    return status
```

```
'''
This file should be named as Motor_Control_2D_xy.py

Motor_Control_2D controls two motors (x and y), using Single_Motor_Control.py
In this case, the x motor moves in x direction and y motor moves in y direction.
It can be easily modified to accommodate 3D motion.

Modified by: Yuchen Qian
Oct 2017
'''

import math
from Single_Motor_Control import Motor_Control
import time
import numpy


############################################################################################
############################################################################################


class Motor_Control_2D:

  def __init__(self, x_ip_addr = None, y_ip_addr = None):

    self.x_mc = Motor_Control(verbose=True, server_ip_addr= x_ip_addr)
    self.y_mc = Motor_Control(verbose=True, server_ip_addr= y_ip_addr)


    self.steps_per_cm = 200000.0 # For EG=20000steps/rev motor and 1mm/rev shaft
    # self.steps_per_degree = 33333.0

    self.motor_moving = False

    self.d_outside = 71.0 #cm distance from the ball valve to the motor's motion channel
    self.d_inside = 35.5 #cm distance from the ball valve to the center of the chamber (0,0) point



  def set_steps_per_rev(self, xsteps, ysteps):
    # Set up the steps per revolution
    self.x_mc.steps_per_rev(xsteps)
    self.y_mc.steps_per_rev(ysteps)

  # Translate the input cartesian positions to motor positions. Here the distance from the ball valve to
  # the motor's motion channel is 29.5 inch (75cm), and the distance from the ball valve to the center of
  # the chamber (0,0) is approximately 14 inch (35.5cm)
  def translate_to_motor_coordinate(self, dx, dy): #dx, dy is the user input coordinate. mx, my is the motor coordinate.
    my = self.d_outside * dy / (dx + self.d_inside)
    mx = numpy.sqrt(dy**2 + (dx + self.d_inside)**2) - self.d_inside
    print('for test: move to motor coordinate', mx, my)
    return mx, my

  def translate_to_user_coordinate(self, mx, my):
    dx = numpy.sqrt((mx + self.d_inside)**2 / ((my / self.d_outside)**2 + 1)) - self.d_inside
    dy = my * (dx + self.d_inside) / self.d_outside
    return dx, dy

  def move_to_position(self, x_pos, y_pos):
    # Directly move the motor to their absolute position
    #mx_pos, my_pos = self.translate_to_motor_coordinate(x_pos, y_pos)
    x_step = self.cm_to_steps(x_pos)
    y_step = self.cm_to_steps(y_pos)
    self.x_mc.set_position(x_step)
    self.y_mc.set_position(y_step)
    self.motor_moving = True
    self.wait_for_motion_complete()


#----------------------------------------------------------------------------------------------------


  def stop_now(self):
    # Stop motor movement now
    self.x_mc.stop_now()
    self.y_mc.stop_now()


  def set_zero(self):
    self.x_mc.set_zero()
    self.y_mc.set_zero()


  def reset_motor(self):
    self.x_mc.reset_motor()
    self.y_mc.reset_motor()


#----------------------------------------------------------------------------------------------------


  def ask_velocity(self):
    self.speedx = self.x_mc.motor_velocity()
    self.speedy = self.y_mc.motor_velocity()
    return self.speedx, self.speedy
  def set_velocity(self, vx, vy):
    self.x_mc.set_speed(vx)
    self.y_mc.set_speed(vy)


#--------------------------------------------------------------------------------------------



  def cm_to_steps(self, d:float) -> int:
    # convert distance d in cm to motor position
    return int(d * self.steps_per_cm)

  # def degree_to_steps(self, d:float) -> int: # This feature applies to Z-Th probe drive
  #   # convert angle d in degree to motor position
  #   return int(d * self.steps_per_degree)

#--------------------------------------------------------------------------------------------



  def current_probe_position(self):
    # Obtain encoder feedback and calculate probe position
    """ Might need a encoder_unit_per_step, if encoder feedback != input step """
    mx_pos = self.x_mc.current_position() / self.steps_per_cm *5
    my_pos = self.y_mc.current_position() / self.steps_per_cm *5 #Seems that 1 encoder unit = 5 motor step unit
    #dx_pos, dy_pos = self.translate_to_user_coordinate(mx_pos,my_pos)

    return mx_pos, my_pos


#--------------------------------------------------------------------------------------------


  def wait_for_motion_complete(self):

    timeout = time.time() + 300

    while True :

      x_stat = self.x_mc.check_status()
      y_stat = self.y_mc.check_status()
      time.sleep(0.2)

      x_not_moving = x_stat.find('M') == -1
      y_not_moving = y_stat.find('M') == -1


#        print ('x:', x_stat)
#        print ('y:', y_stat)
#        print ('z:', z_stat)
#        print (x_not_moving, y_not_moving, z_not_moving)

      if x_not_moving and y_not_moving:
        break
      elif time.time() > timeout:
        raise TimeoutError("Motor has been moving for over 5min???")
    self.motor_moving = False
    print ("Motor stopped")


#--------------------------------------------------------------------------------------------


  def disable(self):
    self.x_mc.inhibit()
    self.y_mc.inhibit()

  def enable(self):
    self.x_mc.enable()
    self.y_mc.enable()

  def set_input_usage(self, usage):
    self.x_mc.set_input_usage(usage)
    self.y_mc.set_input_usage(usage)
```

```
# This file should be named as Single_Motor_Control.py
# This program translates ASCII commands and sends messages to an Applied Motion motor.
# It can be used in Motor_Control_3D.py and Motor_Control_2D.py for multi-dimensional movement.
# Author: Yuchen Qian



import sys
if sys.version_info[0] < 3: raise RuntimeError('This script should be run under Python 3')

import socket
import select
import time
from find_ip_addr import find_ip_addr





############################################################################################################
############################################################################################################


class Motor_Control:

  MSIPA_CACHE_FN = 'motor_server_ip_address_cache.tmp'
  MOTOR_SERVER_PORT = 7776
  BUF_SIZE = 1024
  # server_ip_addr = '10.10.10.10' # for direct ethernet connection to PC

  # - - - - - - - - - - - - - - - - - -
  # To search IP address:
  last_pos = 999

  def __init__(self, server_ip_addr = None, msipa_cache_fn = None, verbose = True):
    self.verbose = verbose
    if msipa_cache_fn == None:
      self.msipa_cache_fn = self.MSIPA_CACHE_FN
    else:
      self.msipa_cache_fn = msipa_cache_fn

    # if we get an ip address argument, set that as the suggest server IP address, otherwise look in cache file
    if server_ip_addr != None:
      self.server_ip_addr = server_ip_addr
    else:
      try:
        # later: save the successfully determined motor server IP address in a file on disk
        # now: read the previously saved file as a first guess for the motor server IP address:
        self.server_ip_addr = None
        with open(self.msipa_cache_fn, 'r') as f:
          self.server_ip_addr = f.readline()
      except FileNotFoundError:
        self.server_ip_adddr = None

    # - - - - - - - - - - - - - - - - - - - - - - - - -
    if self.server_ip_addr != None  and  len(self.server_ip_addr) > 0:
      try:
        print('looking for motor server at', self.server_ip_addr,end=' ',flush=True)
        t = self.send_text('RS')
        print ('status =', t[5:])
        if t != None: #TODO: link different response to corresponding motor status
          print('...found')
#          self.reset_motor()
          self.inhibit(inh=False)
          self.send_text('IFD') #set response format to decimal

        else:
          print('motor server returned', t, sep='')
          print('todo: why not the correct response?')

      except TimeoutError:
        print('...timed out')
      except (KeyboardInterrupt,SystemExit):
        print('...stop finding')
        raise

    with open(self.msipa_cache_fn, 'w') as f:
      f.write(self.server_ip_addr)

#    encoder_resolution = self.send_text('ER')
#    if float(encoder_resolution[5:]) != 4000:
#      print('Encoder step/rev is not equal to motor step/rev. Check!!!')
############################################################################################################
############################################################################################################
  def __repr__(self):
    """ return a printable version: not a useful function """
    return self.server_ip_addr + '; ' + self.msipa_cache_fn + '; ' + self.verbose


  def __str__(self):
    """ return a string representation: """
    return self.__repr__()

  def __bool__(self):
    """ boolean test if valid - assumes valid if the server IP address is defined """
    return self.server_ip_addr != None

  def __enter__(self):
    """ no special processing after __init__() """
    return self

  def __exit__(self, exc_type, exc_value, traceback):
    """ no special processing after __init__() """

  def __del__(self):
    """ no special processing after __init__() """

##############################################################################################
##############################################################################################

  def send_text(self, text, timeout:int=None) -> str:
    """worker for below - opens a connection to send commands to the motor control server, closes when done"""
    """ note: timeout is not working - needs some MS specific iocontrol stuff (I think) """
    RETRIES = 30
    retry_count = 0
    while retry_count < RETRIES:  # Retries added 17-07-11
      try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        ##if timeout is not None:
        ##  #not on windows: socket.settimeout(timeout)
        ##  s.setsockopt(socket.SOL_SOCKET, socket.SO_RCVTIMEO, struct.pack('LL', timeout, 0))
        s.connect((self.server_ip_addr, self.MOTOR_SERVER_PORT))
        break
      except ConnectionRefusedError:
        retry_count += 1
        print('...connection refused, at',time.ctime(),' Is motor_server process running on remote machine?',
              '  Retry', retry_count, '/', RETRIES, "on", str(self.server_ip_addr))
      except TimeoutError:
        retry_count += 1
        print('...connection attempt timed out, at',time.ctime(),
              '  Retry', retry_count, '/', RETRIES, "on", str(self.server_ip_addr))

    if retry_count >= RETRIES:
      input(" pausing in motor_control.py send_text() function, hit Enter to try again, or ^C: ")
      s.close()
      return self.send_text(text, timeout)  # tail-recurse if retry is requested


    message = bytearray(text, encoding = 'ASCII')
    buf = bytearray(2)
    buf[0] = 0
    buf[1] = 7
    for i in range(len(message)):
        buf.append(message[i])
    buf.append(13)

    s.send(buf)

    BUF_SIZE = 1024
    data = s.recv(BUF_SIZE)
    s.close()
    return_text = data.decode('ASCII')
    #print(text,' ', return_text)
    return return_text

#    if timeout is not None:
#      s.setblocking(0)
#      ready = select.select([s],[],[],timeout)
#      if ready[0]:
#        data = s.recv(BUF_SIZE)
#        s.close()
#        #decipher returned text here
#        return_text = data.decode('ASCII')
#        return return_text
#      else:
#        print ("No message heard from motor")




#    # For utf-8 encoding commands
#    buf = bytes(text, encoding='utf-8')
#    s.send(buf)
#    data = s.recv(self.BUF_SIZE)
#    s.close()
#    return_text = data.decode('utf-8')
#    if self.verbose:
#       print(' | response is', return_text)
#       #print(' ',type(data), len(data), ' ', end='')
#    return return_text




#-------------------------------------------------------------------------------------------

  def instant_velocity(self):

    resp = self.send_text('IV')
    # return rpm
    rpm = float(resp[5:])
    return (rpm)

#-------------------------------------------------------------------------------------------

  def motor_velocity(self):

    resp = self.send_text('VE')
    # return rpm
    rpm = float(resp[5:])
    return (rpm)

#-------------------------------------------------------------------------------------------

  def current_position(self):
    resp = self.send_text('EP')
    r = 0
    while r < 30:
      try:
        pos = float(resp[5:])
        self.last_pos = pos
#        print ('\n ,,,,,,,Motor at:', pos)
        return pos
        break

      except ValueError:
#        print ('Not right')
        time.sleep(2)
        r += 1


#-------------------------------------------------------------------------------------------

  def set_position(self, step):

    try:
      #self.send_text('DI'+str(step))
      self.send_text('FP'+str(step))
      time.sleep(0.5)
#      print ('Finish moving')

    except ConnectionResetError as err:
      print('*** connection to server failed: "'+err.strerror+'"')
      return False
    except ConnectionRefusedError as err:
      print('*** could not connect to server: "'+err.strerror+'"')
      return False
    except KeyboardInterrupt:
      print('\n______Halted due to Ctrl-C______')
      return False


    # todo: see http://code.activestate.com/recipes/408859/  recv_end() code
    #       We need to include a terminating character for reliability, e.g.: text += '\n'


#-------------------------------------------------------------------------------------------

  def stop_now(self):
    self.send_text('ST')

#-------------------------------------------------------------------------------------------

  def steps_per_rev(self, stepsperrev):
    self.send_text('EG' + str(stepsperrev))
    print('set stpes/rev = ' + str(stepsperrev) +'\n')

#-------------------------------------------------------------------------------------------

  def set_zero(self):
    self.send_text('EP0')  # Set encoder position to zero
    resp = self.send_text('IE')
    if int(resp[5:]) == 0:
      print ('Set encoder to zero\n')
      self.send_text('SP0')  # Set position to zero
      resp = self.send_text('IP')
      if int(resp[5:]) == 0:
        print ('Set current position to zero\n')
      else :
        print ('Fail to set current position to zero\n')
    else :
      print ('Fail to set encoder to zero\n')


  def set_acceleration(self, acceleration):
    self.send_text('AC'+str(acceleration))


  def set_decceleration(self, decceleration):
    self.send_text('DE'+str(decceleration))


  def set_speed(self, speed):
    try:
      self.send_text('VE'+str(speed))
```

```
#      resp = self.send_text('VE')
#      print (resp)
    except ConnectionResetError as err:
      print('*** connection to server failed: "'+err.strerror+'"')
      return False
    except ConnectionRefusedError as err:
      print('*** could not connect to server: "'+err.strerror+'"')
      return False
    except KeyboardInterrupt:
      print('\n______Halted due to Ctrl-C______')
      return False

#-----------------------------------------------------------------------------------------------

  def check_status(self):
    # print("""
    #   # A = An Alarm code is present (use AL command to see code, AR command to clear code)
    #   # D = Disabled (the drive is disabled)
    #   # E = Drive Fault (drive must be reset by AR command to clear this fault)
    #   # F = Motor moving
    #   # H = Homing (SH in progress)
    #   # J = Jogging (CJ in progress)
    #   # M = Motion in progress (Feed & Jog Commands)
    #   # P = In position
    #   # R = Ready (Drive is enabled and ready)
    #   # S = Stopping a motion (ST or SK command executing)
    #   # T = Wait Time (WT command executing)
    #   # W = Wait Input (WI command executing)
    #   """)
    return self.send_text('RS')

#-----------------------------------------------------------------------------------------------

  def reset_motor(self):

    self.send_text('RE',timeout=5)
    print("reset motor\n")

  def clear_alarm(self):

    self.send_text('AR')
    print('Clear alarm. Check LED light to see if the fault condition persists.')


#-----------------------------------------------------------------------------------------------

  def inhibit(self, inh=True):
     """ inh = True:  Raises the disable line on the PWM controller to disable the output
               False: Lowers the inhibit line
     """
     if inh:
       cmd = 'MD'
       print('motor disabled\n', sep='', end='', flush=True)
     else:
       cmd = 'ME'
       print('motor enabled\n', sep='', end='', flush=True)

     try:
       self.send_text(cmd)  # INHIBIT or ENABLE

     except ConnectionResetError as err:
       print('*** connection to server failed: "'+err.strerror+'"')
       return False
     except ConnectionRefusedError as err:
       print('*** could not connect to server: "'+err.strerror+'"')
       return False
     except KeyboardInterrupt:
       print('\n______Halted due to Ctrl-C______')
       return False

     # todo: see http://code.activestate.com/recipes/408859/  recv_end() code
     #       We need to include a terminating character for reliability, e.g.: text += '\n'
     return True



  def enable(self, en=True):
     """ en = True:  Lowers the inhibit line on the PWM controller to disable the output
              False: Raises the inhibit line
     """
     return self.inhibit(not en)

  def set_input_usage(self, usage):
    self.send_text('SI'+str(usage))
    print('set x3 input usage to SI' + str(usage) + '\n')
```

```
"""
This file should be named as find_ip_addr.py

find_ip_addr(search_str, ip_range='') -> ()

Returns a list of ip addresses associated with a particular adapter manufacturer
See example below for usage

Uses the program nmap to accomplish the results. This should be installed on the system,
but here we use hardwired paths:
  Linux:   /usr/bin/nmap
  Windows: looks in C:\Program Files (x86) for an nmap directory


@author: Patrick
"""
import sys
if sys.version_info[0] < 3: raise RuntimeError('This script should be run under Python 3')

import os
import socket
import subprocess

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# throws this exception if search fails

class find_ip_addr_error(Exception):
  """ our local error exception; has no behavior of its own"""
  pass

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

def my_ip_addr() -> str:
  """ worker for below: used to determine the 24 bit subnet address of our local lan"""
  if sys.platform == 'linux':
    test_ip_addr = '8.8.8.8'
    r = [l for l in ([ip for ip in socket.gethostbyname_ex(socket.gethostname())[2] if not ip.startswith("127.")][:1],
[[(s.connect((test_ip_addr, 53)), s.getsockname()[0], s.close()) for s in [socket.socket(socket.AF_INET,
socket.SOCK_DGRAM)]][0][1]]) if l][0][0]
    if len(r) == 0:
      raise find_ip_addr_error('*** my_ip_addr(): This computer does not seem to have an IP address')
    return r
    # this opaque technique is from
    #          http://stackoverflow.com/questions/166506/finding-local-ip-addresses-using-pythons-stdlib
    # I'm not sure it works, since it returns the same result when the RJ45 connection is unplugged and
    #    there is no internet connection, AND also when the wifi subsequently connects with a different IP address
    # Also, on my subnet at home, it (cleverly) returns the internet facing IP, rather than e.g. '192.169.1.5';
    #    unfortunately the latter is what we need in this instance
  else:
    # on the other hand, the following works on windows but not on Linux:
    ip_addrs = socket.gethostbyname_ex(socket.gethostname())[2]
#??? Behavior changed again? 3/27/2017 had to remove this because  len(ip_addrs) is 1 now
#???    if len(ip_addrs) < 2:
#???      raise find_ip_addr_error('*** my_ip_addr(): This computer does not seem to have an IP address')
    return ip_addrs[-1]                # = current ipv4 address, e.g. '192.168.1.100'
    # was return ip_addrs[1]                # = current ipv4 address, e.g. '192.168.1.100'

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# main routine

def find_ip_addr(search_str, ip_range='') -> ():
  """ uses nmap to find the ip address associated with search_str. e.g. "Raspberry PI" or "Avalue Technology" """
  if len(ip_range) == 0:
    ip_addr = my_ip_addr()
    if len(ip_addr) == 0:
      raise find_ip_addr_error('*** find_ip_addr(): local ip address not determined')
    ip_range = ip_addr[0:ip_addr.rfind('.')] + '.*'   # change last byte to '*'

  if sys.platform == 'linux':
    nmap_path = '/usr/bin/nmap'
  else:
    # assume windows, and that nmap is installed in a directory like "C:\Program Files (x86)\nmap-7.12"
    WPF = os.getenv("programfiles(x86)")   # I guess fix this if not 64 bit windows
    dl = [d for d in os.listdir(WPF) if d.lower().find('nmap') == 0]
    if len(dl) == 0:
      WPF = os.getenv("programfiles")
    dl = [d for d in os.listdir(WPF) if d.lower().find('nmap') == 0]
    if len(dl) == 0:
      print("*** find_ip_addr(): cannot find nmap.exe (see https://nmap.org)")  # note: this is windows we are failing on
      raise(find_ip_addr_error("cannot find nmap.exe (see https://nmap.org)"))
    nmap_path = os.path.join(WPF, dl[-1], 'nmap.exe')     # use last occurrence, which is presumably the highest version

  args = [nmap_path, "-n", "-sn", ip_range]        # e.g.   nmap -n -sn 192.168.1.*

  if sys.platform == 'linux':
    args = ['sudo'] + args  # needs to be root to get MAC and vendor info, for some reason

  print('Running nmap to search '+ip_range+' for "'+search_str+'"', flush=True, end='')

  # this works, but then complains about nonsense (Python 3.5 on Windows):
  #       t = os.spawnv(os.P_WAIT, args[0], args)
  # instead, do this (and capture the output):

  nmap_results = ''
  if sys.hexversion < 0x03050000:
    # ugh bumped into this because Pi ships with Python 3.4, which does not have subprocess.run()
    sco = subprocess.check_output(args)
    # will raise exception CalledProcessError if return code != 0
    nmap_results = str(sco)
  else:
    cpo = subprocess.run(args, stdout=subprocess.PIPE)
    if cpo.returncode != 0:
      raise find_ip_addr_error('Failed to run "'+args[0]+'"')
    nmap_results = str(cpo.stdout)

  """ nmap output is a series of 3-line descriptions of the form
     ...
  Nmap scan report for 192.168.1.5
  Host is up (0.080s latency).
  MAC Address: B8:27:EB:A5:B1:91 (Raspberry Pi Foundation)
     ...
  """
  addrs = ()
  while len(nmap_results) > 0:
    i = nmap_results.find(search_str)    # find our target
    if i < 0:
      break
    k = i + len(search_str)
    i = nmap_results[0:i].rfind("report for ")   # backwards search starting at our target
    if i < 0:
      raise find_ip_addr_error('nmap results are funny: found "'+search_str+'" but could not find the text "report for"')
      return addrs
    i += len("report for ")
    j = nmap_results[i:].find("\\r\\n")
    if j < 0:
      j = len(nmap_results[i:])
    addrs += (nmap_results[i:i+j],)
    nmap_results = nmap_results[k:]   # repeat, starting after the found text

  if len(addrs) == 0:
    print(' -> not found')
    raise find_ip_addr_error('nmap did not find "'+search_str+'"')
  else:
    print(' ->',addrs)
  return addrs
```