Coding Framework and Implementation for Resident Space Object Observation

AE8900 MS Special Problems Report
Space Systems Design Lab (SSDL)
Guggenheim School of Aerospace Engineering
Georgia Institute of Technology
Atlanta, GA

Author:
Nicole M. Tyman

Advisor:
Marcus J. Holzinger

May 1, 2015

Coding Framework and Implementation for Resonant Space Object Observation

Nicole M. Tyman *

Georgia Institute of Technology, Atlanta, GA, 30332, US

The detection, classification, and tracking of Earth-orbiting space objects is of great importance for the safety of orbiting spacecraft, motivating the improvement of the current space situational awareness (SSA) model. A revolutionary concept for SSA introduced by DARPA includes allowing independent observers to contribute observational data in order to reduce cost and introduce more space object data for the SSN. With the installment of a Raven-class telescope at the Georgia Institute of Technology, this report seeks to provide a coding framework for observations that allows for relevent contributions to the SSA concept. The information and coding architecture are illustrated, and the coding performance is analyzed. Improvements to the current architecture are introduced as future work.

I. Introduction

With more than 500,000 estimated space debris objects orbiting Earth, NORAD has cataloged roughly 40,000 space objects as of 2015. At relative velocities of 15 km/s, this debris can potentially collide with Earth-orbiting spacecraft, including the ISS and thousands of commercial satellites, and create more debris particulates in the process. The risks associated with these collisions include not only the loss of capital investment but also the loss of human life. It is no question that the safety of current and future space objects resides in the detection, classification, and tracking of the orbiting space debris so that the space environment is adequately modeled.[1]

For decades, the Space Surveillance Network (SSN) has utilized a network of sensors scattered worldwide in order to detect and catalog space objects. DARPA is currently seeking to revolutionize space situational awareness concepts by leading away from a sensor-centric model to a more data-centric model, effectively introducing more data sources and reducing cost in the process.[2] This concept seeks to modify the current SSN architecture to a more distributed and coordinated architecture by introducing more data sources. These data sources, whether it be backyard astronomers or research institutions, can potentially contribute to the SSN if the observation capabilities are well-matched with requirements for space object observation.

The observatory at the Georgia Institute of Technology recently acquired a Raven-class telescope to aide in the efforts of detection, classification, and tracking of space objects. Using commercial-off-the-shelf components optimized for specific observation campaigns, the Raven-class telescope can meaningfully contribute to this new SSA concept in a cost-effective build. This report seeks to describe a code implementation for the observation of resonant space object using Python and will investigate the required information architecture, python setup, code architecture, and coding performance. Future work on the code is also investigated to further improve the architecture and include a discussion for potential instrumentation change.

II. Information Architecture

Figure 1 shows the information architecture for the main Python code, which is used for space object observations. This figure shows the information flow from the user, weather station, computer system, and instrumentation to the main Python executable code. The user defines the observation frequency and duration as well as the save file location for the observation campaign, and the user is responsible for ensuring

*Graduate Researcher, Guggenheim School of Aerospace Engineering, 270 Ferst Drive, AIAA Non-Member

connectivity of the required instrumentation. The goal of the software is to collect measurements from each instrument at the user-specified frequency and export the information as both XML and text files that may later be stored in an information database.

The main Python code defined in this report will be converted to an executable file that will govern the whole observation process. Though a graphical user interface (GUI), the user is able to select the observation frequency and duration inputs, test the Internet connection, initiate the observation, and save the image and sensor data. Once the observation is initiated, the executable will utilize event threading in order to ensure synchronized measurements at the required frequency and duration. The executable will then store instrumentation and weather station data as an XML and TXT files that may later be converted to a single XML file with fields defined by the O2SDK XML standard for space object observations.

The XML file may then be uploaded to BaseX, which is an XML database module for Python. If a web host allows for database access, users may access the XML file through secure authentication over the web. Using this information architecture, a database of observations can be maintained and used for improvement of the current space situational awareness model.

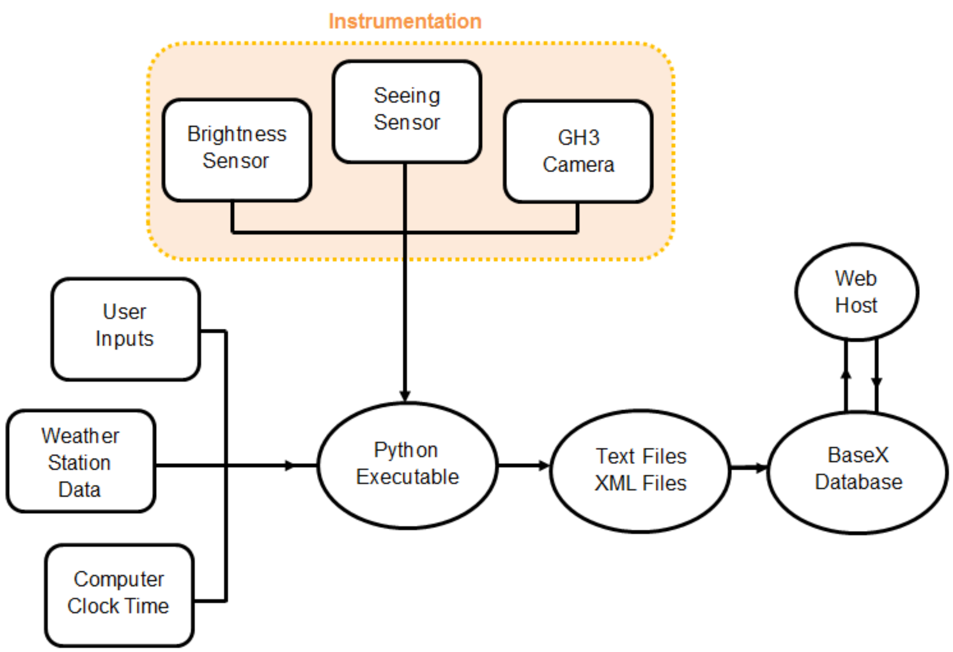

Figure 1. Code architecture for the main Python code. The user block signifies all that is available to or operable by the user, and the function call flows are illustrated

III. Python Setup

Python packages are available online from various universities and independent software developers. These packages have the benefit of offering optimized functions in terms of computation speed that potential developers may use in their own coding algorithm. These packages allow users to spend less time overcoming the learning curve associated with working in a new computing language and more time formulating and developing a working code. Creating working code more efficiently with respect to time and computational speed using these Python extensions and packages follows the cost-effective goal of the data-centric changes to the space domain awareness model.

The following Python packages have offered substantial contribution to the project:

- BeautifulSoup
- ElementTree
- wxpython
- pyflycapture2

BeautifulSoup is Python library that is currently licensed under MIT. This package extracts and parses data available from an HTML or XML document, and allows for the developer to parse efficiently through the elements and its attributes. The project uses this package to extract the weather station information by parsing through the station's weather data attributes. The extracted fields can then be processed and passed into an XML structure.

ElementTree is a package that allows the storage of hierarchical data structures using simplified functions and is used to wrap the structured weather information into an XML format. While the structure of the file is pre-defined based on the information headers, this package is used to easily create the file structure that is later saved using the save2xml() function.

Wxpython is a Python-specific GUI toolkit that wraps a cross-platform GUI library written in C++. This package introduces a graphical user interface to the main Python code, which allows input and event-triggered control from the user. The package includes many basic GUI items, including radioboxes, buttons, and text boxes, and allows for the developer to bind events and functions that seem intuitive to the user.

Pyflycapture2 is Python binding for the FlyCapture API used for Point Grey cameras. While the camera can operate though Point Grey's own software, this package allows for Python to access functions defined in the FlyCapture SDK library documentation. This allows the developer to access various camera commands and settings, including camera connection, image capture, and frame rate and image resolution settings. This code is used to connect and access the camera when storing the image data.

All of the aforementioned packages are available for free use, and example code using some of the packages has been supplied under the samples folder on GitHub. For this project, Python version 2.6 for Windows is used to develop the code.

IV. Code Architecture

The code architecture for the project is shown in Figure 2. The user is able to define the measurement variables, the save file directory, and url of the weather station into the GUI Panel of the code, while ensuring that all of the required instrumentation is connected and operating. Through the graphical interface, the user can select whether to ensure Internet connection, change the current file saving directory, or start the measurements. Testing the Internet connection commands the code to test whether the user-supplied url is reachable, which will then trigger a successful or erroneous dialog box. Changing the current file directory opens a directory dialog, and the choice of a folder triggers the change of the save file location. When the user selects "start," the event triggers the initiation of three measurements. Threading is used to simultaneously initiate three functions: weatherobs(), camobs(), and runtime().

The weatherobs() function is divided into two subfunctions: collect_weather() and save2xml(). The prior subfunction connects to the weather station via the user-supplied url and extracts relevant data from the HTML. A filename is then created based on the time of the measurement, and then save2xml() is executed. This subfunction saves the data into an XML file into the specified directory using the aforementioned filename. The files are then indented using the IndentPrintXml() function after the file is saved as an XML file, which is purely done for the aesthetics of the XML file.

The camobs() function connects to the Point Grey camera, takes an image from the camera using the default values for frame rate and image size, disconnects the camera, and saves the image array as a TEXT file. The filename for the TEXT file is derived similarly as in the weatherobs() function, where the time of the meaurement is included.

The runtime() function allows the threading execution to occur for the user-defined frequency. While each of the threads initiate at the same time, each thread must wait until the other threads have executed before stopping if the join() function is included. Even if the two measurement threads execute below the user-defined frequency, this function allows for a new iteration of measurements to occur after the desired time has elapsed for each measurement.

All save files currently are constructed by using the time of the measurement in the name of the file, along with the notation of w for weather station data and c for image data. After an XML file and text file are created from the observation campaign, the user can then upload the XML file into BaseX, which is a Python database. If the web host allows pulls from databases, the XML files can then be pulled using specific authentication, allowing information from the file to be made available upon inquiry.

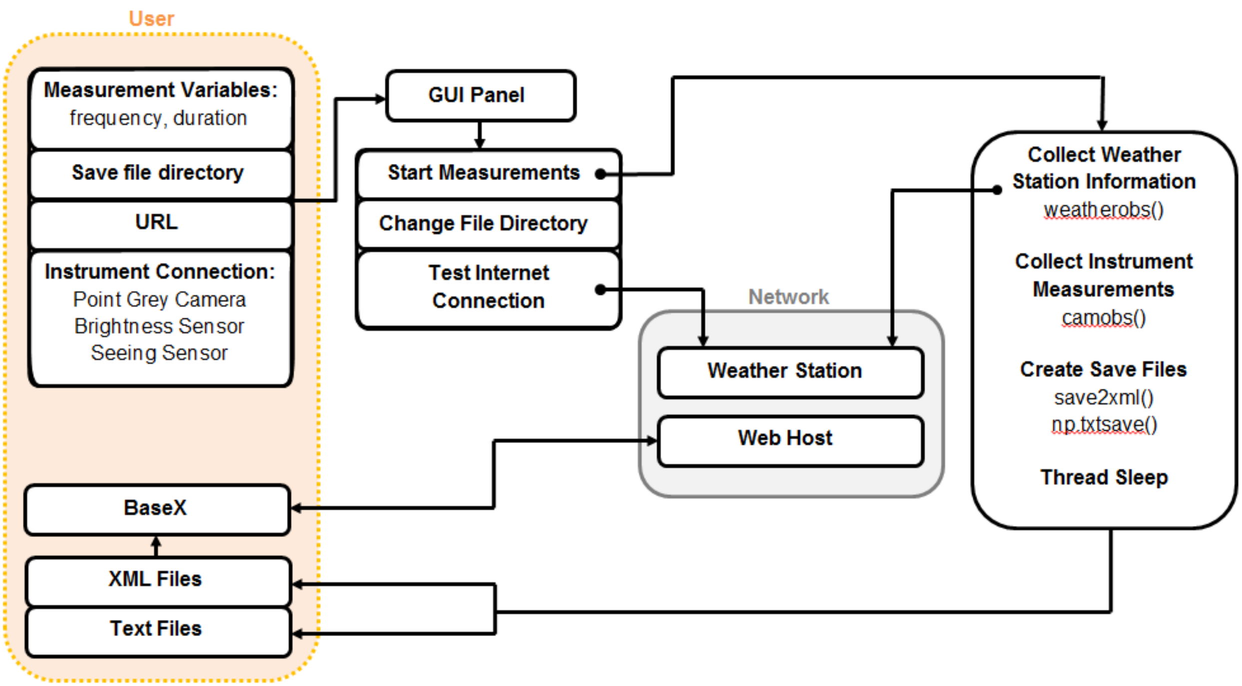

Figure 2. Code architecture for the main code, showing the functional progression from user, system, url, and instrumentation data to the Python executable.

V. Coding Performance

The performance of the main code is investigated in terms of CPU time using the timeit() module and the results are shown in Table 1. This test was performed on a Lenovo Yoga tablet PC with wireless Internet connection for the test frequency of 25 Hz using a Point Grey Grasshopper 2 camera (model number GRAS-14S3C-C).

Table 1. Run time of each function in the thread.

Function	Run Time (s)
weatherobs()	0.208
collect_weather()	0.198
save2xml()	0.005
camobs()	0.308
start_capture()	0.018
runtime()	0.040

Since the code is divided into three threads, each thread is examined for run times. The thread that calls weatherobs() runs at nearly five times the required speed. The weatherobs() function is composed of the collect_weather() function and the save2xml() function, and the bulk of the time is spent collecting the weather information. The high run time which may be attributed to the limited Internet speed, and may increase given a faster connection. The thread that calls camobs() runs into a similar problem, but the bulk of the time is spent initiating the connection to the camera. Image capture itself only took 0.018 seconds, which would match the required time for each observation. Overall, the code does show promise to execute at the maximum frequency of the camera as long as a fast Internet connection is secured and the camera connection and frame rate settings are initiated before the observation occurs.

VI. Conclusions and Future Work

By providing a coding framework and implementation for space object observations, the telescope at Georgia Tech may contribute to the SSA concept and provide relevant data to the SSN. The code has been attached in the Appendix, and the report outlines the information and coding architecture as well as the Python setup and coding performance. The code allows for the measurements to be taken at the desired frequency and observation duration. While the current camera code shows the required code to get the camera operational, the code shows potential timing improvement through the initiation of the camera before measurements, setting the required frame rate for image capture, and securing a fast and reliable Internet connection.

The issue of improving the camera code is saved for future work as the camera options may not be finalized. With the impending addition of a USB Point Grey camera, there have been concerns whether there will be issues of compatibility. While Point Grey cameras do have Direct Show driver support, which should allow compatibility with the software, drivers can be created using the X2 standard for SkyX to ensure compatibility. If SkyX were to be used for camera and mount control, the Python code can then be modified for SkyX interfacing and optimized in performance.

Another future work option includes setting up the Python database using BaseX. While it is possible for BaseX to allow user-authenticated web access to the database, more investigation can be made into setting up and testing this process. Overall, the future work aims to build up the current coding framework in order for the telescope at Georgia Tech to be fully capable of providing a database solution to the SSA concept.

Appendix

```
# -*- coding: utf-8 -*-
"""
Created on Tue Mar 31 14:57:26 2015

# Program starts when user has moved the mount and wants to initiate taking
# pictures, program will save data and take information from computer
# After process is finished, can have another program to process through and
# create an xml file


# Purpose: Allow the user to start observations set at certain frequencies
# All user-defined items must be defined in the class (TimerClass)

# List of modules used/need to install:
# numpy, BeautifulSoup, ElementTree, wxpython, py2flycapture2
@author: Jitterbug (Nicole Tyman)
"""
# Import all modules
import os
import timeit
import sys
import time
import datetime
import threading
import numpy as np

from bs4 import BeautifulSoup
import urllib2

from xml.etree import ElementTree
from xml.etree.ElementTree import Element
from xml.etree.ElementTree import SubElement
from lxml import etree

global freq

# User input
url="http://www.weatherlink.com/user/gtmaui/index.php?view=summary&headers=1"
page=urllib2.urlopen(url)
start_date = datetime.datetime.now()
```

```
def collect_weather():
    page=urllib2.urlopen(url)
    soup = BeautifulSoup(page.read())
    data=soup.findAll('td',{'class':'summary_data'})
    return(data)

    # Makes the xml file indented
def IndentPrintXml(fn):
    assert fn is not None
    parser = etree.XMLParser(resolve_entities=False, strip_cdata=False)
    document = etree.parse(fn, parser)
    document.write(fn, pretty_print=True, encoding='utf-8')

def save2xml(data, filename):
    # Create the xml file
    # Create the root, WeatherData
    # The root, elements, etc. cannot include spaces
    weatherdata = Element( 'GTWeatherData' )

    # Create a subelement: Temperature
    tem = SubElement( weatherdata, 'OutsideTemp' )

    # List all the information extracted from the image
    tempcurr = SubElement( tem, 'Current')
    tempcurr.text = data[1].string

    temphigh = SubElement( tem, 'TodaysHigh')
    temphigh.text =  data[2].string

    temphightime = SubElement( tem, 'TodaysHighTime')
    temphightime.text = data[3].string

    templow = SubElement( tem, 'TodaysLow')
    templow.text =  data[4].string

    templowtime = SubElement( tem, 'TodaysLowTime')
    templowtime.text = data[5].string

    ## Create the subelement: Outside humidity
    oh = SubElement( weatherdata, 'OutsideHumidity' )
    ohcurr = SubElement( oh, 'Current')
    ohcurr.text = data[7].string
    ohhigh = SubElement( oh, 'TodaysHigh')
    ohhigh.text =  data[8].string
    ohhightime = SubElement( oh, 'TodaysHighTime')
    ohhightime.text = data[9].string
    ohlow = SubElement( oh, 'TodaysLow')
    ohlow.text =  data[10].string
    ohlowtime = SubElement( oh, 'TodaysLowTime')
    ohlowtime.text = data[11].string

    # Create the subelement: Inside Temp
    it = SubElement( weatherdata, 'InsideTemperature' )
    itcurr = SubElement( it, 'Current')
    itcurr.text = data[13].string
    ithigh = SubElement( it, 'TodaysHigh')
    ithigh.text =  data[14].string
    ithightime = SubElement( it, 'TodaysHighTime')
    ithightime.text = data[15].string
    itlow = SubElement( it, 'TodaysLow')
    itlow.text =  data[16].string
    itlowtime = SubElement( it, 'TodaysLowTime')
    itlowtime.text = data[17].string

    # Create the subelement: InsideHumidity
    ih = SubElement( weatherdata, 'InsideHumidity' )
    ihcurr = SubElement( ih, 'Current')
    ihcurr.text = data[19].string
    ihhigh = SubElement( ih, 'TodaysHigh')
```

```
    ihhigh.text =  data[20].string
    ihhightime = SubElement( ih, 'TodaysHighTime')
    ihhightime.text = data[21].string
    ihlow = SubElement( ih, 'TodaysLow')
    ihlow.text =  data[22].string
    ihlowtime = SubElement( ih, 'TodaysLowTime')
    ihlowtime.text = data[23].string

    # Create the subelement: Heat Index
    hi = SubElement( weatherdata, 'HeatIndex' )
    hicurr = SubElement( hi, 'Current')
    hicurr.text = data[25].string
    hihigh = SubElement( hi, 'TodaysHigh')
    hihigh.text =  data[26].string
    hihightime = SubElement( hi, 'TodaysHighTime')
    hihightime.text = data[27].string

    # Create the subelement: Wind Chill
    wc = SubElement( weatherdata, 'WindChill' )
    wccurr = SubElement( wc, 'Current')
    wccurr.text = data[31].string
    wclow = SubElement( wc, 'TodaysLow')
    wclow.text =  data[34].string
    wclowtime = SubElement( wc, 'TodaysLowTime')
    wclowtime.text = data[35].string

    # Create the subelement: Dew Point
    dp = SubElement( weatherdata, 'DewPoint' )
    dpcurr = SubElement( dp, 'Current')
    dpcurr.text = data[37].string
    dphigh = SubElement( dp, 'TodaysHigh')
    dphigh.text =  data[38].string
    dphightime = SubElement( dp, 'TodaysHighTime')
    dphightime.text = data[39].string
    dplow = SubElement( dp, 'TodaysLow')
    dplow.text =  data[40].string
    dplowtime = SubElement( dp, 'TodaysLowTime')
    dplowtime.text = data[41].string

    # Create the subelement: Barometer
    bm = SubElement( weatherdata, 'Barometer' )
    bmcurr = SubElement( bm, 'Current')
    bmcurr.text = data[43].string
    bmhigh = SubElement( bm, 'TodaysHigh')
    bmhigh.text =  data[44].string
    bmhightime = SubElement( bm, 'TodaysHighTime')
    bmhightime.text = data[45].string
    bmlow = SubElement( bm, 'TodaysLow')
    bmlow.text =  data[46].string
    bmlowtime = SubElement( bm, 'TodaysLowTime')
    bmlowtime.text = data[47].string

    # Create the subelement: Bar Trend
    bt = SubElement( weatherdata, 'BarTrend' )
    btcurr = SubElement( bt, 'Current')
    btcurr.text = data[49].string #55

    # Create the subelement: Wind Speed
    ws = SubElement( weatherdata, 'WindSpeed')
    wscurr = SubElement( ws, 'Current')
    wscurr.text = data[55].string
    wshigh = SubElement( ws, 'TodaysHigh')
    wshigh.text =  data[56].string
    wshightime = SubElement( ws, 'TodaysHighTime')
    wshightime.text = data[57].string

    # Create the subelement: Wind Direction
    wd = SubElement( weatherdata, 'WindDirection')
    wdcurr = SubElement( wd, 'Current')
    wdcurr.text = data[61].string
```

```
    # Create the subelement: 12 Hour Forecast
    hrfc = SubElement( weatherdata, 'TwelveHrForecast')
    hrfc.text = data[66].string

    # Create the subelement: Average Wind Speed
    aws = SubElement( weatherdata, 'AvgWindSpeed')
    awstm = SubElement( aws, 'TwoMinute')
    awstm.text = data[67].string
    awstenm = SubElement( aws, 'TenMinute')
    awstenm.text = data[68].string

    # Create the subelement: Wind Gust Speed
    wgs = SubElement( weatherdata, 'WindGustSpeed')
    wgstm = SubElement( wgs, 'TwoMinute')
    wgstm.text = data[73].string
    wgstenm = SubElement( wgs, 'TenMinute')
    wgstenm.text = data[74].string

    # Create the subelement: Rain
    rain = SubElement( weatherdata, 'Rain')
    rr = SubElement( rain, 'Rate')
    rr.text = data[81].string
    rd = SubElement( rain, 'Day')
    rd.text = data[82].string
    rs = SubElement( rain, 'Storm')
    rs.text = data[83].string
    rm = SubElement( rain, 'Month')
    rm.text = data[84].string
    ry = SubElement( rain, 'Year')
    ry.text = data[85].string

    # Create the subelement: Last Hour Rain
    lhr = SubElement( weatherdata, 'LastHourRain')
    lhrr = SubElement( lhr, 'Rate')
    lhrr.text = data[87].string

    # Save the file
    output_file = open( filename, 'w' )
    output_file.write( '<?xml version="1.0"?>' )
    output_file.write( ElementTree.tostring( weatherdata ) )
    output_file.close()

    IndentPrintXml(filename)

def weatherobs():
    #Check to see if each thread starts at same time
#    time = datetime.datetime.now()
#    print time

    #t0 = time.clock()
    weatherdata = collect_weather()
    #t1 = time.clock()
    #print t1-t0

    now = datetime.datetime.now()
    nowstr = str(now.year) + '_' + str(now.month) + '_' + str(now.day) + '_' + \
    str(now.hour) + '_' + str(now.minute) + '_' + str(now.second) + '_' + \
    str(now.microsecond)
    # Generate filename and save to file
    filename = nowstr
    filename += 'w'
    #filename += str(time.time()).replace('.','_')
    filename += '.xml'

    save2xml(weatherdata, filename) #Puts data into xml


def camobs():
    #Check to see if each thread starts at same time
```

```
    #time = datetime.datetime.now()
    #print time
    import flycapture2 as fc2
    import numpy as np

    c = fc2.Context()
    c.connect(*c.get_camera_from_index(0))
    c.set_video_mode_and_frame_rate(fc2.VIDEOMODE_1280x960Y16,
            fc2.FRAMERATE_7_5)
    c.start_capture()
    im = fc2.Image()
    b = [np.array(c.retrieve_buffer(im)).sum() for i in range(80)]
    a = np.array(im)
    c.stop_capture()
    c.disconnect()

    now = datetime.datetime.now()
    nowstr = str(now.year) + '_' + str(now.month) + '_' + str(now.day) + '_' + \
    str(now.hour) + '_' + str(now.minute) + '_' + str(now.second) + '_' + \
    str(now.microsecond)
    # Generate filename and save to file
    filename = nowstr
    filename += 'c'
    #filename += str(time.time()).replace('.','_')
    filename += '.txt'

    np.savetxt(filename, a, delimiter=',')

def runtime(freq):
    #Since all threads need to start after 1/freq seconds, use the join function
    # This way, the thread waits until this function is done, then allows for
    # the next thread to occur
    #time.sleep(1.0/freq)
    time.sleep(1.0/freq)


# Measure time each code operates
#print min(timeit.Timer(weatherobs).repeat(3, 50))/50
#print min(timeit.Timer(collect_weather).repeat(3, 50))/50
#print min(timeit.Timer(runtime).repeat(3,50))/50

def is_number(x):
    try:
        float(x)
        return True
    except ValueError:
        pass

    try:
        import unicodedata
        unicodedata.numeric(x)
        return True
    except (TypeError, ValueError):
        pass
    return False

class TimerClass(threading.Thread):
    timearray = np.array([])

    def __init__(self, freq, obsdur):
        self.freqinput = freq
        self.obsdur = obsdur
        threading.Thread.__init__(self)
        self.event = threading.Event()
        self.count = 1
        self.isrunning = True

    def run(self):

        while self.count < self.freqinput and self.isrunning:
```

```
            #print self.count, print_time(), time.clock(), 'Time: %f' % time.time()
            self.timearray = np.r_[self.timearray, time.clock()]
            self.weatherdata = collect_weather()

            # Generate filename and save to file
            filename = 'w'
            filename += str(self.count)
            #filename += str(time.time()).replace('.','_')
            filename += '.xml'

            save2xml(self.weatherdata, filename) #Puts data into excelsheet
            #self.event.wait(0.04)
            self.count += 1
            #time.sleep(0.04)
            #print self.count
        print self.timearray

    def stop(self):
        self.event.set()

import wx

class ObsPanel(wx.Panel):
    def __init__(self, parent):
        wx.Panel.__init__(self, parent)
        self.newpath = os.getcwd();

        # create some sizers
        mainSizer = wx.BoxSizer(wx.VERTICAL)
        grid = wx.GridBagSizer(hgap=5, vgap=5)
        hSizer = wx.BoxSizer(wx.HORIZONTAL)

        # A multiline TextCtrl - This is here to show how the events work in this program, don't pay too much attention to it
        #self.logger = wx.TextCtrl(self, size=(200,300), style=wx.TE_MULTILINE | wx.TE_READONLY)

        # Frequency input
        self.freq = wx.StaticText(self, label="Frequency (Hz):")
        self.obsdur = wx.StaticText(self, label="Observation Duration (s):", size= (180,-1))
        self.freqbox = wx.TextCtrl(self, size=(80,-1))
        self.obsdurbox = wx.TextCtrl(self, size=(80,-1))
        grid.Add(self.freq, pos=(1,0))
        grid.Add(self.freqbox, pos=(1,1))
        grid.Add(self.obsdur, pos=(2,0))
        grid.Add(self.obsdurbox, pos=(2,1))


        # Save file input
        # Shows the current save file location and allows the user to change
        # the location
        self.fltext = wx.StaticText(self, label="Save File Path:")
        grid.Add(self.fltext, pos=(3,0))
        self.fl = wx.TextCtrl(self, value=str(self.newpath), size=(250,-1))
        grid.Add(self.fl, pos=(4,0), span=(1,2))

        # Create a change save directory button
        self.buttonsave = wx.Button(self, label="Change Directory")
        self.Bind(wx.EVT_BUTTON, self.OnChangeDirClick, self.buttonsave)
        grid.Add(self.buttonsave, pos=(5,0))

        # Test Connection button
        self.button =wx.Button(self, label="Test Connection")
        self.Bind(wx.EVT_BUTTON, self.OnConClick,self.button)

        # Start button
        self.button2 =wx.Button(self, label="Start")
        self.Bind(wx.EVT_BUTTON, self.OnStartClick,self.button2)
```

```
        #Spacer
        grid.Add((-1, 5), pos=(7,0))

        hSizer.Add(grid, 0, wx.ALL, 1)
        #hSizer.Add(self.logger)
        mainSizer.Add(hSizer, 0, wx.ALL, 3)
        mainSizer.Add(self.button, 0, wx.CENTER)
        mainSizer.Add(self.button2, 0, wx.CENTER)
        self.SetSizerAndFit(mainSizer)

    def OnChangeDirClick(self, event):

        dlg = wx.DirDialog(self, "Choose a directory:",
                           style=wx.DD_DEFAULT_STYLE
                           #| wx.DD_DIR_MUST_EXIST
                           #| wx.DD_CHANGE_DIR
                           )
        if dlg.ShowModal() == wx.ID_OK:
            print "New Save File Path Location: %s" % dlg.GetPath()
            self.newpath = dlg.GetPath()
            self.fl.SetValue(dlg.GetPath())
            #os.chdir(dlg.GetPath())
        dlg.Destroy()

    def OnConClick(self,event):
        # Test the connection. If fails, return dialog box
        #self.logger.AppendText(" Click on object with Id %d\n" %event.GetId())
        try:
            urllib2.urlopen('http://www.google.com');
            dlg = wx.MessageDialog(self, "Test Connection Successful", "Internet Connectivity?", wx.OK)  #create a dialog (dlg) box to display the message, and ok button
            dlg.ShowModal()  #show the dialog box, modal means cannot do anything on the program until clicks ok or cancel
            dlg.Destroy()  #destroy the dialog box when its not needed
        except IOError:
            dlg = wx.MessageDialog(self, "Test Connection Unsuccessful", "Internet Connectivity?", wx.OK)  #create a dialog (dlg) box to display the message, and ok button
            dlg.ShowModal()  #show the dialog box, modal means cannot do anything on the program until clicks ok or cancel
            dlg.Destroy()  #destroy the dialog box when its not needed
    def OnStartClick(self,event):
        # Check if inputs are correct
        self.freqinput = float(self.freqbox.GetValue().strip())
        self.obsdurinput = float(self.obsdurbox.GetValue().strip())

        # Start the observations
        if is_number(self.freqinput) & is_number(self.obsdurinput):
            if self.freqinput > 25 or self.freqinput < 0:
                dlg = wx.MessageDialog(self, "Please provide a measurement frequency between 0 and 25.", "Error", wx.OK)  #create a dialog (dlg) box to display the message, and ok button
                dlg.ShowModal()
                dlg.Destroy()
            elif self.obsdurinput <0:
                dlg = wx.MessageDialog(self, "Please provide a positive observation duration.", "Error", wx.OK)  #create a dialog (dlg) box to display the message, and ok button
                dlg.ShowModal()
                dlg.Destroy()

            else:
                #print datetime.datetime.now()

                for i in range(int(self.obsdurinput*self.freqinput)):
                    # Append the threads
                    thread1= threading.Thread(target=weatherobs)
                    thread2 = threading.Thread(target=camobs)
                    thread3 = threading.Thread(target=runtime, args=(self.freqinput,))

                    thread1.start()
                    thread2.start()
```

```python
                    thread3.start()

                    # Wait until all threads are done
                    thread1.join()
                    thread2.join()
                    thread3.join()

                #print datetime.datetime.now()


        else:
            dlg = wx.MessageDialog(self, "Please provide numerical inputs", "Error", wx.OK) #create a dialog (dlg) box to display the message, and ok button
            dlg.ShowModal()
            dlg.Destroy()

    def OnSave(self,event):
        self.logger.AppendText(" Click on object with Id %d\n" %event.GetId())
        print self.newpath
    def EvtText(self, event):
        self.logger.AppendText('EvtText: %s\n' % event.GetString())
    def EvtChar(self, event):
        self.logger.AppendText('EvtChar: %d\n' % event.GetKeyCode())
        event.Skip()
    def EvtCheckBox(self, event):
        self.logger.AppendText('EvtCheckBox: %d\n' % event.Checked())


app = wx.App(False)
frame = wx.Frame(None)
panel = ObsPanel(frame)
frame.Fit()
frame.Show()
app.MainLoop()
```

Compiled.py

References

[1]Kelso, T. S., Alfano, S. "Satellite Orbital Conjunction Reports Assessing Threatening Encounters in Space (SOCRATES)" *SPIE*, 2006.

[2]Blake, Travis, Sanchez, M., Krassner, Georgen M. J., and Sundbeck, S. "Space Domain Awareness," 2012.