# How I added Real-time Plotting to OpenSVP

**Prepared by:**  
Eugène Desjardins Couture

**Present to:**

OpenSVP Developers

Date: August 23, 2019

**Disclaimer**

Natural Resources Canada (NRCan) is not responsible for the accuracy or completeness of the information contained in the reproduced material. NRCan shall at all times be indemnified and held harmless against any and all claims whatsoever arising out of negligence or other fault in the use of the information contained in this publication or product.

**Third-Party Materials**

Some of the information contained in this publication or product may be subject to copyrights held by other individuals or organizations. To obtain information concerning copyright ownership and restrictions, contact us by:

E-mail:

Telephone:

Facsimile:

Note: The two texts above are optional and may be added to the mandatory text set forth above. You have the option of either adding both or any one of the two.

**Copyright**

Information contained in this publication or product may be reproduced, in part or in whole, and by any means, for personal or public non-commercial purposes, without charge or further permission, unless otherwise specified.

You are asked to:

* exercise due diligence in ensuring the accuracy of the materials reproduced;
* indicate the complete title of the materials reproduced, and the name of the author organization; and
* indicate that the reproduction is a copy of an official work that is published by Natural Resources Canada (NRCan) and that the reproduction has not been produced in affiliation with, or with the endorsement of, NRCan.

Commercial reproduction and distribution is prohibited except with written permission from NRCan. For more information, contact NRCan at copyright.droitdauteur@nrcan-rncan.gc.ca.

*© Her Majesty the Queen in Right of Canada, as represented by the Minister of Natural Resources Canada, [year of publication].*

Summary

I added the real-time plotting feature as a process of multiprocessing, the same way the run process is implemented and I link those two processes together to send the real-time flow of data.

Table of Contents

[Introduction 3](#_Toc17464497)

[General idea 4](#_Toc17464498)

[What has been done? 5](#_Toc17464499)

[How to start? 5](#_Toc17464500)

[Modification to openSVP files 6](#_Toc17464501)

[Modification to app.py 6](#_Toc17464502)

[Modification to ui.py 8](#_Toc17464503)

[Modification to the Drivers 10](#_Toc17464504)

[RealTimePlotting.py 10](#_Toc17464505)

[Das.py 11](#_Toc17464506)

[Therefore, the Dataframe reinitialise itself at each test in the data\_capture method of the DAS Class: 12](#_Toc17464507)

[What is next? 13](#_Toc17464508)

[View tab with preference window 13](#_Toc17464509)

[The Real-time plotting GUI 13](#_Toc17464510)

[The Run GUI 14](#_Toc17464511)

## Introduction

In the field of research, it is an advantage to be able to assess data fast. Therefore, some features can be interesting to add to a software like OpenSVP. For example, it would be great to have waveform analysis to effectively parse through the data in a visual way. In this report, I will explain how I introduce a real-time plotting feature to OpenSVP. With this report, someone will be able to understand how he/she can add to the feature and make it better.

## General idea

This report is to give a detail view of what were the changes in the OpenSVP code. Therefore, new developpers can understand how it was added and how it works.

## How it works

The idea of Real-time plotting is a concurrent process from the test on going that query data from the test to then illustrate it. Therefore, since a python execution is only one thread of code and that the run process must not be interrupted, the real-time plotting feature has his own process or thread that is executed in parallel.

### RTP vs SVP Concept

The code is adding processes from the multiprocessing library. Multiprocessing is used to resolve a CPU bound problem. The application can then split up his cpu load into multiple CPU cores. The limitation of this method is the number of CPUs on the computer. Following is a flowchart of some major function of RTP through SVP.

### Modification to openSVP files

#### Modification to app.py

To add real-time plotting, we need concurrent processes since it needs to run at the same time as the run process. Therefore, I added the real-time plotting feature the same way as a run process.

In fact, Python is a language that only allows a program to do one thing at a time. However, there is a way to cheat this feature of Python and it is called multiprocessing. Multiprocessing creates another python interpreter that execute your new process, and then this process runs at the same time as the main process.

These are the main modifications I did to app.py to add Real-time plotting:

1. I added a new function named real\_time\_plotting(), that takes the current library path (lib\_path) as a parameter to access the RealTimePlotting.py script, which is the real-time plotting feature. Moreover, this function also has a pipe connection (rtp\_conn) as a parameter to communicate with the run process.

def real\_time\_plotting(lib\_path, rtp\_conn):  
 sys.path.insert(0, lib\_path + '\\svpelab')  
 try:  
 n = importlib.import\_module('RealTimePlotting')  
 if n is not None:  
 n.RealTimePlottingDialog(rtp\_conn)  
 except Exception, e:  
 raise  
 finally:  
 if lib\_path is not None and sys.path[0] == lib\_path:  
 del sys.path[0]

1. Next, I initialise a sub process in the start method of the RunContext Class. This sub process call the real\_time\_plotting function above when the process is started. In this way, the real-time plotting start at the same time as a run process in openSVP. Moreover, the run process is activated when you run a test or a suite.

self.process = MultiProcess(name='svp\_process', target=process\_run, args=(filename, env, script\_config, params, self.lib\_path, self.test\_conn, self.run\_conn))  
self.subprocess = MultiProcess(name='rtp\_process', target=real\_time\_plotting, args=(self.lib\_path, self.rtp\_conn))  
  
self.process.start()  
self.subprocess.start()

1. As you can see in the initialisation of the two processes above, there is two connection named self.run\_conn and self.rtp\_conn. These connections are used to communicate between the two processes. Therefore, these two connections are linked together like a pipe and can send messages and data at anytime.

self.rtp\_conn, self.run\_conn = multiprocessing.Pipe()

1. We already saw that the rtp\_conn is passed to the real\_time\_plotting function of app.py, and then it is passed to the RealTimePlottingDialog function of the RealTimePlotting.py script in the drivers directory. For the run\_conn, it is pass to the process\_run function of app.py. Then, it goes to the RunScript initialisation for the test\_script variable, which is accessible in almost every driver and is different for each test.

def process\_run(filename, env, config, params, lib\_path, conn, run\_conn):  
 name = script\_path = None  
 try:  
 sys.stdout = sys.stderr = open(os.path.join(trace\_dir(), 'sunssvp\_script.log'), "w", buffering=0)  
 script\_path, name = os.path.split(filename)  
 name, ext = os.path.splitext(name)  
 if lib\_path is not None:  
 sys.path.insert(0, lib\_path)  
 sys.path.insert(0, script\_path)  
 try:  
 m = importlib.import\_module(name)  
 info = m.script\_info()  
 test\_script = RunScript(env=env, info=info, config=config, config\_file=None, params=params, conn=conn, run\_conn=run\_conn)  
 m.run(test\_script)  
 except Exception, e:  
 raise  
 finally:  
 if name in sys.modules:  
 del sys.modules[name]  
 if sys.path[0] == script\_path:  
 del sys.path[0]  
 if lib\_path is not None and sys.path[0] == lib\_path:  
 del sys.path[0]

1. After this, I added some lines of code to properly dispose of the sub process when it is finished. I did the same way as the run process. Therefore, I will let you investigate on your own for the other changes. (Little tip: If you have SVN or something similar, you can always replace the app.py of a version without real-time plotting and compare the changes with the diff feature of SVN. Just remember to not Commit)

#### Modification to ui.py

You have to know that ui.py mainly handles the GUI of OpenSVP. However, the way I added Real-time plotting is that I created a new GUI in the RealTimePlotting.py script in the driver.

For now, I did not add anything significant to the OpenSVP GUI. Just a button that does nothing. Which I initialise in multiple function of the RunUI (You can search for the key words : Real-time plotting changes):

1. There is the line where I add the button to the RunPanel:

# Real-time plotting changes  
status\_bar\_ctrl\_sizer.Add(self.run\_ctrl.rtp\_button, 0, wx.TOP|wx.BOTTOM|wx.LEFT|wx.RIGHT, 5)

1. This where I add the logo to the button inside the RunCtrl Class:

# Real-time plotting changes  
self.rtp\_bitmap = wx.Bitmap(os.path.join(images\_path, 'rtp\_96.gif'), wx.BITMAP\_TYPE\_GIF)

1. This is where I created the button and linked it with the self.rtp function that does nothing :

self.rtp\_button = wx.BitmapButton(self.parent, bitmap=self.rtp\_bitmap)  
self.rtp\_button.Bind(wx.EVT\_BUTTON, self.rtp)  
self.rtp\_button.Enable()  
self.rtp\_process = None

1. This is the self.rtp method of the RunCtrl function, where I tried to start the real-time plotting process but didn’t work:

def rtp(self, event):  
 self.rtp\_process = svp.MultiProcess(name='Real-time Plotting', target=RealTimePlotting, args=(self,))  
 self.rtp\_process.start()

Next, in the ui.py script, I also added the View tab and it’s Real-Time plotting options. It was intended that I would add an option to activate the feature and also add a new window where we could ajust the parameters of the feature before running a test. Therefore, this is the beginning of this work:

1. So to add the View menu to the main menu bar, I started by adding the next line of code to the ToolFrame Class. Which is the parameters of the menu items :

menu\_rtp\_items = [(wx.ID\_ANY, 'active', '', None, OP\_RTP\_ACT),  
 (wx.ID\_ANY, 'Preference', '', None, OP\_RTP\_PREF)]  
  
menu\_view\_items = [(wx.ID\_ANY, 'Real-time Plotting', '', menu\_rtp\_items, OP\_VIEW\_RTP)]

1. Then, these menu items are use in the create\_menu\_bar method of the ToolFrame class :

def create\_menu\_bar(self):  
 ops = self.update\_menu\_ops()  
 menu\_bar = wx.MenuBar()  
 file\_menu, enabled = self.create\_menu(ToolFrame.menu\_file\_items, ops)  
 menu\_bar.Append(file\_menu, 'File')  
 edit\_menu, enabled = self.create\_menu(ToolFrame.menu\_edit\_items, ops)  
 menu\_bar.Append(edit\_menu, 'Edit')  
 view\_menu, enabled = self.create\_menu(ToolFrame.menu\_view\_items, ops)  
 menu\_bar.Append(view\_menu, 'View')  
 help\_menu, enabled = self.create\_menu(ToolFrame.menu\_help\_items, ops)  
 menu\_bar.Append(help\_menu, 'Help')  
 self.SetMenuBar(menu\_bar)  
 return menu\_bar

### Modification to the Drivers

#### RealTimePlotting.py

The big modification I did to the drivers is adding the RealTimePlotting.py script, which implement the real-time plotting feature. I added some comment to the code to let you understand my thinking better.

1. It is composed of the main function RealTimePlottingDialog(rtp\_conn). This function initialize a new GUI and let it run.

def RealTimePlottingDialog(rtp\_conn):  
 app = wx.App()  
 app.frame = GraphFrame(rtp\_conn)  
 alignToBottomRight(app.frame)  
 app.SetTopWindow(app.frame)  
 app.frame.Show()  
 app.MainLoop()

1. I will not go in details in how the GUI is done. Moreover, the GraphFrame class initialize the GUI with his create functions.

class GraphFrame(wx.Frame):  
 *""" The main frame of the application  
 """* title = 'Demo: dynamic matplotlib graph'  
  
 def \_\_init\_\_(self, rtp\_conn = None):  
 wx.Frame.\_\_init\_\_(self, None, -1, self.title)  
  
 self.rtp\_conn = rtp\_conn  
 self.df = pd.DataFrame()  
 self.df\_x = pd.DataFrame()  
 self.df\_y = pd.DataFrame()  
 self.xy\_df = pd.DataFrame()  
 self.x\_list = []  
 self.info = None  
 self.show\_frame = False  
 self.xy = None

1. Next, a timer refresh the graph and it is set in the initialisation.

self.redraw\_timer = wx.Timer(self)  
self.Bind(wx.EVT\_TIMER, self.on\_redraw\_timer, self.redraw\_timer)  
self.redraw\_timer.Start(40)

1. Then, this timer call the on\_redraw\_timer method of GraphFrame, which call the data\_read() function, that add refresh the different Dataframe of the graph and refresh the plot.

def on\_redraw\_timer(self, event):  
 # if paused do not add data, but still redraw the plot  
 # (to respond to scale modifications, grid change, etc.)  
 #  
 if not self.paused and self.rtp\_conn.poll():  
 self.data\_read()  
  
 self.draw\_plot()

1. Now, you need to understand that the run process collects the data at the same time, as you want to plot it. This is where the connection between the run process and the real-time plotting process is useful. Therefore, by the rtp\_conn the data\_read() function reads the data sent through the run\_conn and update the DataFrames of the graph.

def data\_read(self):  
 data = None  
 try:  
 if self.rtp\_conn:  
 if self.rtp\_conn.poll() is True:  
 data = self.rtp\_conn.recv()  
 except Exception, e:  
 raise  
 if data is not None:

1. Warning, the code is much more complex, so to better understand the code see the comments added.

#### Das.py

In das.py, I send the data as they are collected in the data\_sample method of the DAS class by the run\_conn of the ts (test script) attribute of DAS in the form of series of a panda Dataframe. Then, it is received by the rtp\_conn as seen above and it reconstruct the Dataframe in the RealTimePlotting.py script. Moreover, at the beginning of each test, there is an info dictionary that is send before the new Dataframe is made.

def data\_sample(self):  
 *"""  
 Read the current data values directly from the DAS and place in the current dataset.  
 """* if self.\_capture is True:  
  
 self.\_last\_datarec = self.device\_data\_read()  
 if self.start\_time is None:  
 self.start\_time = [self.device.start\_time]  
 self.\_ds2 = pd.DataFrame(np.column\_stack(self.\_last\_datarec[1:]), index=self.start\_time,  
 columns=self.data\_points[1:])  
 self.ts.run\_conn.send({'name': self.ts.name, 'phases': self.ts.param\_value('eut.phases')})  
 else:  
 self.\_ds2 = self.\_ds2.append(pd.Series(self.\_last\_datarec[1:], index=self.data\_points[1:],  
 name=self.device.current\_time))  
 self.ts.run\_conn.send(self.\_ds2.tail(1))  
 self.\_ds.append(self.\_last\_datarec)  
 return self.\_last\_datarec

## Therefore, the Dataframe reinitialise itself at each test in the data\_capture method of the DAS Class:

self.\_ds2 = pd.DataFrame()  
self.start\_time = None

## What is next?

In this section, I will give some ideas that can added to the real-time plotting feature. Some of these ideas might have already been introduced in the feature.

### View tab with preference window

We could have a view tab in the main GUI with a real-time plotting sub-option that has an active option and a preference option.

The active option would just ask the user if he wants the real-time plotting feature active or not.

The preference option would open a preference window, where you can set initiate parameters to the Real-time plotting GUI. These features could be the color, the grid precision, the graph overlay, the ranges, which graph you want to see, etc. It would show an example of the graph in action with random data.

### The Real-time plotting GUI

For the GUI of the real-time plotting feature, we could adjust it to be as the same format as other windows of OpenSVP.

Then, we could add the options that we set in the preference window. Therefore, we could change certain parameters as the graph is drawn.

One of the parameters would be to decide what kind of graph you want.

We could have two graphs based on the data. The first graph would contain two Y-axis that represent the electrical values that we test (Volts and Watts, for a Volt-Watt test) and the X-axis would be time. The second graph would depends on the test you do. For example, in a Volt-Watt test, you have the Volts for the X-axis and the Watts for the Y-axis and we could put a graph overlay as another parameter on this graph and then see if the test pass in real-time.

We could decide to pause the graph, but continue to collect data in the background.

We could have a full screen option, since real-time plotting can be interesting when doing demonstrations.

### The Run GUI

I already added a new button next to the play and the stop button, but it does nothing for now.

The purpose of this button could be like the active option of the view tab.