# Maxim LaPlante’s Super Cool Process Data Analysis Tool for Dummies

Tool for correlating time-series process data. Overview:

1. Combine, compare, and analyze data
2. Identify, isolate, and cleanse by events
3. Super fast (especially compared to PARCscript and Excel)
4. Intuitive
5. Everything is easily plottable
6. *Neural-Network-Compatible!*

**How it works:**

The tool is written in Python, a programming language. It uses modules (things that add features to Python):

* Pandas to hold and manipulate data
* Numpy to do math on arrays (data)
* Matplotlib to plot data

The tool itself is also a module, called DataPARsing. It does the job of combining, simplifying, and specifying these abilities from these modules so that they can be applied to data from dataPARC PARCview.

The data which the tool looks at is collected using the Excel dataPARC Add-On which is available on the top ribbon of Excel (Desktop version) on GPI computers.

**How to set it up:**

1. Make sure Python is installed on your computer. On windows, the easiest way to do this is:
   1. Open cmd (Command Prompt) and type in “python”. This will direct you to install python from the Windows app store
   2. Once installed, go back to cmd and type:
      1. pip install numpy
      2. pip install matplotlib
      3. pip install pandas
   3. You might have to do more pip installs. You will be told what you are missing when you try to run the program.
   4. Install IDLE from the windows app store.
2. Make a copy of the folder with DataPARSing.py and main.py in it
3. In this folder, create an Excel workbook. In this workbook, go to the dataPARC Add-On on the top ribbon. Do “Get Normalized Data.” Copy your tags to the top row from PARCview. Set your time increment between 1min and 30mins (more data = slower), and “get data”
4. To start writing your code, type “IDLE main.py” in your cmd window
5. To run the code at any time, press f5 or save the file and run it directly. This will run the code, point our your errors, and allow you to run functions directly from the IDLE shell.

**Start writing code:**

1. You have to specify the Excel file in the code. Copy the filepath (go to the .xlsx file in File Explorer, click on it, and copy with Ctrl-C) into the line:
   1. xlfile = "HD\_K1\_v2.xlsx"
2. See the example code in main.py to get a lay of the land
3. Reference the following info to understand what you can do

**Guide to DataPARsing**

There are two important types of objects (the type is called a class):

One is PARC\_TimeSeries, which stores a time-series data from a tag. It has a bunch of functions to plot, modify, analyze, etc. When you pull data from an Excel file, you will have a dictionary (a fancy list with labels) with a bunch of these “Pts” objects stored inside. You can reference each one individually and give them nicer, shorter names.

The other is Events, which is created when you pull events from a PARC\_TimeSeries object. It stores the timestamps of events. It also has some functions for plotting, analysis, etc.

**PARC\_TimeSeries**

Say you do:

speed = data["KAL.KAL\_DeltaV.DRV25219 speed"]

(assuming that tag is in the data you pulled), you can now play with a Pts object called “speed”

Anything you do with the Pts functions (called “methods”) you will do by:

speed.some\_function( arguments, if any )

Here are the functions for Pts:

get\_average() – gets the average

get\_standardDeviation() – self-explanatory

corr( other Pts ) – get the R2 correlation of two time-series variables.

plot( ax ) – plot the Pts onto a MatPlotLib axis object

smooth( windowsize = 10 by default) – returns a smoothed version of the data. To permanently smooth, do speed= speed.smooth()

arithmetic – you can add (,etc.) Pts objects to other Pts objects, or to numbers for offset or scaling. Note: when adding a Pts object to a number, put the number on the right, e.g. speed = speed \* 5 or speed + 5, whatever

slope() – get the rate of change as a series. Again, to store that data, do speed\_roc = speed.slope().

round(width = 1 by default) – round to the nearest whatever. Again, to store that data, do speed = speed.round(5) for round to the nearest 5.

without\_outliers() – quickly remove outliers (data more the 3stddevs away from the mean), again speed = speed.without\_outliers()

normalize() – set the average to zero and the stddev to 1. Makes it really easy to plot things on the same axis.

cleanse\_timestamps( Events) – remove all timestamps within the events passed as an argument. Again, speed = speed. etc etc

keep\_timestamps( Events) – Same as above, but the opposite

get\_event\_timestamps( function, function argument ) – get an Events object. This is a bit weird – the first argument is the name of a function, the second is a way to pass arguments. See the code for more details, but the functions are: outside, under, over, not\_within, within. To store events to do things with them, do some\_events = some\_Pts. get\_event\_timestamps( within, [5, 100] ) or whatever condition. See the example code.

**Events** – get these with get\_event\_timestamps (see above)

Some information is accessed as a property, not a function. To access these, do some\_events.some\_property

true\_duration – get the total time these events take up

true\_duration\_percent – duration but as a percent of total time

total\_duration – the time from the beginning to the end of the data

Ok, now the methods / functions:

pad( number of mins OR [ number mins before, number mins after] ) increase the size of each event by x minutes in each direction. NOTE: when you do this, any overlapping events will be merged.

nudge( number of mins ) push all events forward or backward by x mins. Positive = Forward. This is useful if you think there is a delay in effect, e.g. a machine speed increase might cause a change elsewhere *in x minutes*.

plot( ax ) – similar to Pts’ plot, but with highlighting.

combine( other Events, logic type = “AND” by default but can be specified “OR” ) – combine some events object with another. With “AND”, only times when both events are simultaneous are kept, with “OR” any time either event is true is kept.

getDeltas( Pts, return format (list by default, may be pd.Series), percents (False by default, may be specified True) ) – look at some data during these events. For instance, if you have an event for “speed is increasing quickly”, you can look at how much some\_flow increased during each event. With the list, there will be a delta for each event. With the pd.Series (pd.Series is a class which is at the core of the Pts), you will get the same exact data as a list but in pd.Series format. *This does strip timestamps, which is not an ideal behavior and should be patched.*