pytplot

Sep 10, 2020

Contents

CHAPTER 1

Introduction

1.1 What is pytplot?

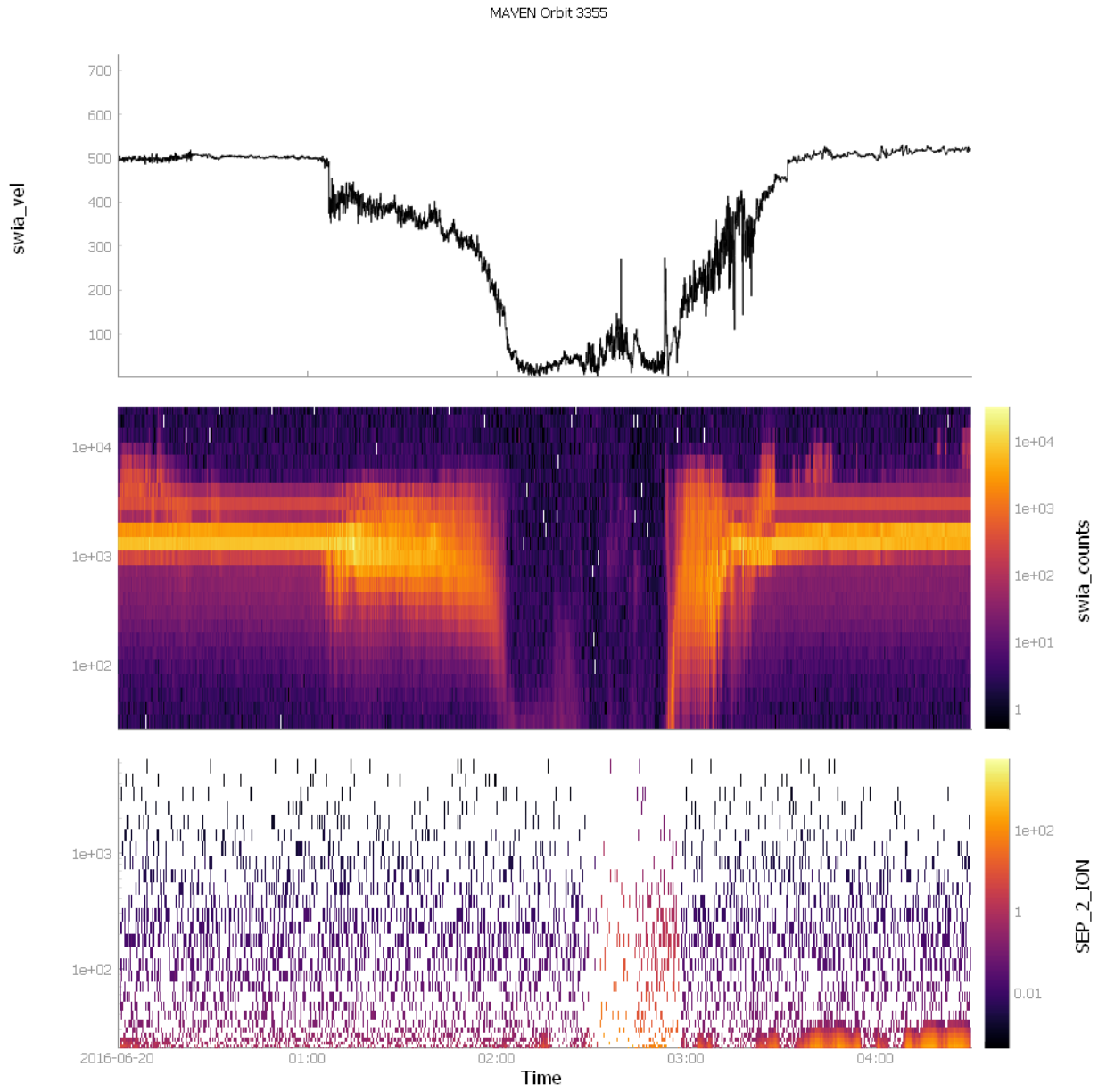
Pytplot is an effort by the Laboratory for Atmospheric and Space Physics to replicate the functionality IDL tplot library. It is a tool for manipulating, analyzing, and plotting time series data. Primarily, it is focused on handling lines and spectrograms from satellite data. It can plot using either Qt via pytqtgraph, or using HTML files via Bokeh.

1.2 What does it do?

Because the tplot library evolved over several decades with new features being added depending on what scientists needed, there is a variety of things it does:

- Reads in data from a variety of sources (including netCDF and CDF file readers)
- Stores data in a common format, alongside all of its metadata and plot options.
- Plots the data in a stacked time series plot with time as the common axis.
- Easily add new axes
- Provides a list of simple commands to modify the plots (line styles, colors, etc), or overplot two different variables
- Provides time series data analysis/manipulation routines
- Provides tools to enable mouse interactions with other python routines

1.3 Pyqtgraph Sample



1.4 Interactive Bokeh Sample

1.5 Version History

1.4.8 Changes:

- The coordinates for 3+ dimensional objects should now be read in correctly.

1.4.7 Changes:

- Added two more ancillary plots, Mars 2D Map and Mars 3D Map

1.4.6 Changes:

- Occasionally logged spec plots would not be properly displayed, this is a hopefully a quick fix for that

1.4.5 Changes:

- Fixing typescript errors with bokeh

1.4.4 Changes:

- Fixing pyqtgraph's collections.abc imports

1.4.3 Changes:

- Bug fix where ylog could not be unset

1.4.1 Changes:

- Fixed small bug in the spec plots for certain file types
- Added PySPEDAS changes to the cdf_to_tplot routine

1.4.0 Changes:

- Added documentation
- Redid tplot variables as xarrays
- Added many unit tests
- Numerous bug fixes

1.3.3 Changes:

- Added power spectrum calculation routine

1.3.2 Changes:

- Updates to the spectrogram plots and CDF reader

1.3.0 Changes:

- Added interactive plots for spectrograms, documentation coming soon

1.2.11 Changes:

- Commenting out tplot_math stuff

1.2.9 Changes:

- Added a netcdf_to_tplot reader
- Changed date axis to show more relevant times

1.2.8 Changes:

- Adding merge functionality to the cdf_to_tplot routine

1.2.5 Changes:

- Adding tplot_math, with various basic functions to begin data analysis
- Crosshairs now implemented in pyqtgraph
- Timebars work in alt/map plots

1.2.4 Changes:

- Fixed for latest version of Anaconda

1.2.1 Changes:

- Fixed a bug in the pyqtgraph spec plots with time varying bins

1.2.0 Changes:

- Added ability to display an arbitrarily large number of qt plot windows, if done from ipython

1.1.13 Changes:

- Added overplot capabilities to the Qt Plotting routines

1.1.12 Changes:

- Fixed major issue with pip installer
- Added ability to use pytplot without a graphics interface, if building only html files are desired.

1.1.6 Changes:

- Fixed a spot where python warnings were changed to change back after the function was over

1.1.4 Changes:

- Added a qt option to tplot, which will allow users to just open the HTML file in a browser window

1.1.3 Changes:

- Bug fix, pyqtgraph was creating a layout every time which eventually caused a crash
- Still a known error where bokeh will no longer plot more than once

1.1.2 Changes:

- Added support for bokeh 0.12.13

1.1.0 Changes:

- Added the ability to plot directly in the Qt Window with pyqtgraph. This may entirely replace the bokeh plotting routines at some point.

1.0.15 Changes:

- Changing tplot to use QtWebKitWidgets by default, but attempt to use QWebEngineView if not found

1.0.14 Changes:

- Fixed a bug in cdf_to_tplot

1.0.11 Changes:

- Bug fixes in the last couple of revisions

1.0.8 Changes:

- Reverting back QWebEngineView changes from 1.0.6

1.0.7 Changes:

- Should be able to export to HTML properly now.

1.0.6 Changes:

- Qt is getting rid of support for QtWebView. QWebEngineView will replace it, but has great difficulty viewing html greater than 2GB.
- As a temporary solution, a local html file is saved, and then read into QWebEngineView.

1.0.5 Changes:

- Fixed a memory leak

1.0.2 Changes:

- Added cdf_to_tplot routine
- Made a version checker

CHAPTER 2

Tplot Variables

Every piece of data read into tplot is stored into global "tplot variable" (sometimes called "tvar" in the documentation).

Every routine that deals with these variables uses simply their names to reference them, you do not need to pass the actual variables themselves around from function to function.

If you would like to access the variable directly, they are stored in a global OrderedDict object called "data_quants", and can be found like so:

```
pytplot.data_quants['tplot_variable_name_here']
```

Tplot variables are kept global because it is not unheard of for >50 variables to be read in from a single file satellite data file, and naming/keeping track of each variable can become cumbersome.

2.1 Internal Structure

Users do not necessarily need to know the internal details of the tplot variables to use the library, but if you'd like to use tplot variables in other libraries this is good information to know.

"Tplot variable" is really just a fancy name for an Xarray DataArrray object. Tplot variables are just a type of DataArray that have standardized attributes so that PyTplot knows how to work with them

- There is always a "time" coordinate, and that time is in seconds since 1970.
- For spectogram data, there is always a "spec_bins" coordinate that keeps track of the bin sizes on the yaxis
- There is a "plot_options" attribute given to each tplot variable that is a large nested dictionary that keeps track of all plotting objects for the variable.
- The dimensions stored are always "time", "y", "v", and for more mutlidimensional data there is "v2", "v3", etc

You can also access the pure underlying numpy arrays like so:

```
#Data
pytplot.data_quants['variable_name'].values
```

(continues on next page)

(continued from previous page)

```
#Time
pytplot.data_quants['variable_name'].coords['time'].values
#Spec bins
pytplot.data_quants['variable_name'].coords['spec_bins'].values
```

CHAPTER 3

Reading in Data

3.1 Manual Input

pytplot.**store_data**(*name*, *data=None*, *delete=False*, *newname=None*)

This function creates a "Tplot Variable" based on the inputs, and stores this data in memory. Tplot Variables store all of the information needed to generate a plot.

Parameters

- **name** – str Name of the tplot variable that will be created
- **data** – dict A python dictionary object.

  'x' should be a 1-dimensional array that represents the data's x axis. Typically this data is time, represented in seconds since epoch (January 1st 1970)

  'y' should be the data values. This can be 2 dimensions if multiple lines or a spectrogram are desired.

  'v' is optional, and is only used for spectrogram plots. This will be a list of bins to be used. If this is provided, then 'y' should have dimensions of x by z.

  'v1/v2/v3/etc' are also optional, and are only used for to spectrogram plots. These will act as the coordinates for 'y' if 'y' has numerous dimensions. By default, 'v2' is plotted in spectrogram plots.

  'x' and 'y' can be any data format that can be read in by the pandas module. Python lists, numpy arrays, or any pandas data type will all work.
- **delete** – bool, optional Deletes the tplot variable matching the "name" parameter
- **newname** – str Renames TVar to new name

Note: If you want to combine multiple tplot variables into one, simply supply the list of tplot variables to the "data" parameter. This will cause the data to overlay when plotted.

Returns None

Examples

```
>>> # Store a single line
>>> import pytplot
>>> x_data = [1,2,3,4,5]
>>> y_data = [1,2,3,4,5]
>>> pytplot.store_data("Variable1", data={'x':x_data, 'y':y_data})
```

```
>>> # Store a two lines
>>> x_data = [1,2,3,4,5]
>>> y_data = [[1,5],[2,4],[3,3],[4,2],[5,1]]
>>> pytplot.store_data("Variable2", data={'x':x_data, 'y':y_data})
```

```
>>> # Store a spectrogram
>>> x_data = [1,2,3]
>>> y_data = [ [1,2,3] , [4,5,6], [7,8,9] ]
>>> v_data = [1,2,3]
>>> pytplot.store_data("Variable3", data={'x':x_data, 'y':y_data, 'v':v_data})
```

```
>>> # Combine two different line plots
>>> pytplot.store_data("Variable1and2", data=['Variable1', 'Variable2'])
```

```
>>> #Rename TVar
>>> pytplot.store_data('a', data={'x':[0,4,8,12,16], 'y':[1,2,3,4,5]})
>>> pytplot.store_data('a',newname='f')
```

3.2 CDF Reader

pytplot.**cdf_to_tplot**(*filenames*, *varformat=None*, *get_support_data=False*, *prefix=''*, *suffix=''*, *plot=False*, *merge=False*, *center_measurement=False*, *notplot=False*, *varnames=[]*)

This function will automatically create tplot variables from CDF files.

Note: Variables must have an attribute named "VAR_TYPE". If the attribute entry is "data" (or "support_data"), then they will be added as tplot variables. Additionally, data variables should have attributes named "DEPEND_TIME" or "DEPEND_0" that describes which variable is x axis. If the data is 2D, then an attribute "DEPEND_1" must describe which variable contains the secondary axis.

Parameters

- **filenames** – str/list of str The file names and full paths of CDF files.
- **varformat** – str The file variable formats to load into tplot. Wildcard character "*" is accepted. By default, all variables are loaded in.
- **get_support_data** – bool Data with an attribute "VAR_TYPE" with a value of "support_data" will be loaded into tplot. By default, only loads in data with a "VAR_TYPE" attribute of "data".

- **prefix** – str The tplot variable names will be given this prefix. By default, no prefix is added.
- **suffix** – str The tplot variable names will be given this suffix. By default, no suffix is added.
- **plot** – bool The data is plotted immediately after being generated. All tplot variables generated from this function will be on the same plot.
- **merge** – bool If True, then data from different cdf files will be merged into a single pytplot variable.
- **center_measurement** – bool If True, the CDF epoch variables are time-shifted to the middle of the accumulation interval by their DELTA_PLUS_VAR and DELTA_MINUS_VAR variable attributes
- **notplot** – bool If True, then data are returned in a hash table instead of being stored in tplot variables (useful for debugging, and access to multi-dimensional data products)
- **varnames** – list Load these variables only. If [] or ['*'], then load everything.

Returns List of tplot variables created (unless notplot keyword is used).

3.3 NetCDF Reader

pytplot.**netcdf_to_tplot**(*filenames*, *time=''*, *prefix=''*, *suffix=''*, *plot=False*, *merge=False*)

This function will automatically create tplot variables from CDF files.

Parameters

- **filenames** – str/list of str The file names and full paths of netCDF files.
- **time** – str The name of the netCDF file's time variable.
- **prefix** – str The tplot variable names will be given this prefix. By default, no prefix is added.
- **suffix** – str The tplot variable names will be given this suffix. By default, no suffix is added.
- **plot** – bool The data is plotted immediately after being generated. All tplot variables generated from this function will be on the same plot. By default, a plot is not created.
- **merge** – bool If True, then data from different netCDF files will be merged into a single pytplot variable.

Returns List of tplot variables created.

Examples

```
>>> #Create tplot variables from a GOES netCDF file
>>> import pytplot
>>> file = "/Users/user_name/goes_files/g15_epead_a16ew_1m_20171201_20171231.nc"
>>> pytplot.netcdf_to_tplot(file, prefix='mvn_')
```

```
>>> #Add a prefix, and plot immediately.
>>> import pytplot
>>> file = "/Users/user_name/goes_files/g15_epead_a16ew_1m_20171201_20171231.nc"
>>> pytplot.netcdf_to_tplot(file, prefix='goes_prefix_', plot=True)
```

3.4 IDL Restore

pytplot.**tplot_restore**(*filename*)

This function will restore tplot variables that have been saved with the "tplot_save" command.

Note: This function is compatible with the IDL tplot_save routine. If you have a ".tplot" file generated from IDL, this procedure will restore the data contained in the file. Not all plot options will transfer over at this time.

Parameters **filename** – str The file name and full path generated by the "tplot_save" command.

Returns None

Examples

```
>>> # Restore the saved data from the tplot_save example
>>> import pytplot
>>> pytplot.restore('C:/temp/variable1.pytplot')
```

CHAPTER 4

Options

pytplot.**options**(*name*, *option=None*, *value=None*, *opt_dict=None*)

This function allows the user to set a large variety of options for individual plots.

Parameters

- **name** – str Name or number of the tplot variable
- **option** – str The name of the option. See section below.
- **value** – str/int/float/list The value of the option. See section below.
- **dict** – dict This can be a dictionary of option:value pairs. Option and value will not be needed if this dictionary item is supplied.

Options:

Options	Value type	Notes
Color	str/list	red, green, blue, etc. Also takes in RGB tuples, i.e. (0,255,0) for green
Colormap	str/list	https://matplotlib.org/examples/color/colormaps_reference.html.
Spec	int	1 sets the Tplot Variable to spectrogram mode, 0 reverts.
Alt	int	1 sets the Tplot Variable to altitude plot mode, 0 reverts.
Map	int	1 sets the Tplot Variable to latitude/longitude mode, 0 reverts.
link	list	Allows a user to reference one tplot variable to another.
ylog	int	1 sets the y axis to log scale, 0 reverts.
zlog	int	1 sets the z axis to log scale, 0 reverts (spectrograms only).
legend_names	list	A list of strings that will be used to identify the lines.
xlog_slice	bool	Sets x axis on slice plot to log scale if True.
ylog	bool	Set y axis on main plot window to log scale if True.
ylog_slice	bool	Sets y axis on slice plot to log scale if True.
zlog	bool	Sets z axis on main plot window to log scale if True.
line_style	str	scatter (to make scatter plots), or solid_line, dot, dash, dash_dot, dash_dot_dot_dot, long_dash.
char_size	int	Defines character size for plot labels, etc.
name	str	The title of the plot.

Table 1 – continued from previous page

Options	Value type	Notes
panel_size	flt	Number between (0,1], representing the percent size of the plot.
basemap	str	Full path and name of a background image for "Map" plots.
alpha	flt	Number between [0,1], gives the transparancy of the plot lines.
thick	flt	Sets plot line width.
yrange	flt list	Two numbers that give the y axis range of the plot.
zrange	flt list	Two numbers that give the z axis range of the plot.
xrange_slice	flt list	Two numbers that give the x axis range of spectrogram slicing plots.
yrange_slice	flt list	Two numbers that give the y axis range of spectrogram slicing plots.
ytitle	str	Title shown on the y axis.
ztitle	str	Title shown on the z axis. Spec plots only.
ysubtitle	str	Subtitle shown on the y axis.
zsubtitle	str	Subtitle shown on the z axis. Spec plots only.
plotter	str	Allows a user to implement their own plotting script in place of the ones herein.
crosshair_x	str	Title for x-axis crosshair.
crosshair_y	str	Title for y-axis crosshair.
crosshair_z	str	Title for z-axis crosshair.
static	str	Datetime string that gives desired time to plot y and z values from a spec plot.
static_tavg	str	Datetime string that gives desired time-averaged y and z values to plot from a spec plot.
t_average	int	Seconds around which the cursor is averaged when hovering over spectrogram plots.
'spec_plot_dim'	int	If variable two dimensions, this sets which dimension the variable will have on on the y axis. All ot

Returns None

Examples

```
>>> # Change the y range of Variable1
>>> import pytplot
>>> x_data = [1,2,3,4,5]
>>> y_data = [1,2,3,4,5]
>>> pytplot.store_data("Variable1", data={'x':x_data, 'y':y_data})
>>> pytplot.options('Variable1', 'yrange', [2,4])
```

```
>>> # Change Variable1 to use a log scale
>>> pytplot.options('Variable1', 'ylog', 1)
```

CHAPTER 5

Tplot Options

pytplot.**tplot_options**(*option*, *value*)

This function allows the user to set GLOBAL options for the generated plots.

Parameters

- **option** – str The name of the option. See section below
- **value** – str/int/float/list The value of the option. See section below.

Options:

Options	Value type	Notes
title	str	Title of the the entire output
title_size	int	Font size of the output
wsize	[int, int]	[height, width], pixel size of the plot window
title_align	int	Offset position in pixels of the title
var_label	srt	Name of the tplot variable to be used as another x axis
alt_range	[flt, flt]	The min and max altitude to be plotted on all alt plots
map_x_range	[int, int]	The min and max longitude to be plotted on all map plots
map_y_range	[int, int]	The min and max latitude to be plotted on all map plots
x_range	[flt, flt]	The min and max x_range (usually time) to be plotted on all Spec/1D plots
data_gap	int	Number of seconds with consecutive nan values allowed before no interp should occur
roi	[str, str]	Times between which there's a region of interest for a user
crosshair	bool	Option allowing crosshairs and crosshair legend
vertical_spacing	int	The space in pixels between two plots
show_all_axes	bool	Whether or not to just use one axis at the bottom of the plot
black_background	bool	Whether or not to make plot backgrounds black w/ white text
axis_font_size	int	The font size of the axis ticks. Default is 10.

Returns None

Examples

```
>>> # Set the plot title
>>> import pytplot
>>> pytplot.tplot_options('title', 'SWEA Data for Orbit 1563')
```

```
>>> # Set the window size
>>> pytplot.tplot_options('wsize', [1000,500])
```

CHAPTER 6

Helper Functions

These are functions that help get or set attributes of the data. All of these were basically mapped one-to-one from IDL routines.

pytplot.**del_data**(*name=None*)

This function will delete tplot variables that are already stored in memory.

Parameters **name** – str Name of the tplot variable to be deleted. If no name is provided, then all tplot variables will be deleted.

Returns None

Examples

```
>>> # Delete Variable 1
>>> import pytplot
>>> pytplot.del_data("Variable1")
```

pytplot.**get_data**(*name*, *xarray=False*)

This function extracts the data from the tplot Variables stored in memory.

Parameters **name** – str Name of the tplot variable

Returns: tuple of data/dimensions stored in pytplot time_val : numpy array of seconds since 1970 data_val : n-dimensional array of data spec_bins_val (if exists) : spectral bins if the plot is a spectrogram v1_val (if exists) : numpy array of v1 dimension coordinates v2_val {if exists} : numpy array of v2 dimension coordinates v3_val (if exists) : numpy array of v3 dimension coordinates

Examples

```
>>> # Retrieve the data from Variable 1
>>> import pytplot
>>> x_data = [1,2,3,4,5]
```

(continues on next page)

(continued from previous page)

```
>>> y_data = [1,2,3,4,5]
>>> pytplot.store_data("Variable1", data={'x':x_data, 'y':y_data})
>>> time, data = pytplot.get_data("Variable1")
```

pytplot.**get_timespan**(*name*)

This function extracts the time span from the Tplot Variables stored in memory.

Parameters **name** – str Name of the tplot variable

Returns

float The beginning of the time series

time_end [float] The end of the time series

Return type time_begin

Examples

```
>>> # Retrieve the time span from Variable 1
>>> import pytplot
>>> x_data = [1,2,3,4,5]
>>> y_data = [1,2,3,4,5]
>>> pytplot.store_data("Variable1", data={'x':x_data, 'y':y_data})
>>> time1, time2 = pytplot.get_timespan("Variable1")
```

pytplot.**get_ylimits**(*name*, *trg=None*)

This function will get extract the y-limits from the Tplot Variables stored in memory.

Parameters

- **name** – str Name of the tplot variable
- **trg** – list, optional The time range that you would like to look in

Returns

float The minimum value of y

ymax [float] The maximum value of y

Return type ymin

Examples

```
>>> # Retrieve the y-limits from Variable 1
>>> import pytplot
>>> x_data = [1,2,3,4,5]
>>> y_data = [1,2,3,4,5]
>>> pytplot.store_data("Variable1", data={'x':x_data, 'y':y_data})
>>> y1, y2 = pytplot.get_ylimits("Variable1")
```

pytplot.**timebar**(*t*, *varname=None*, *databar=False*, *delete=False*, *color='black'*, *thick=1*, *dash=False*)

This function will add a vertical bar to all time series plots. This is useful if you want to bring attention to a specific time.

Parameters

- **t** – flt/list The time in seconds since Jan 01 1970 to place the vertical bar. If a list of numbers are supplied, multiple bars will be created. If "databar" is set, then "t" becomes the point on the y axis to place a horizontal bar.
- **varname** – str/list, optional The variable(s) to add the vertical bar to. If not set, the default is to add it to all current plots.
- **databar** – bool, optional This will turn the timebar into a horizontal data bar. If this is set True, then variable "t" becomes the point on the y axis to place a horizontal bar.
- **delete** – bool, optional If set to True, at lease one varname must be supplied. The timebar at point "t" for variable "varname" will be removed.
- **color** – str The color of the bar
- **thick** – int The thickness of the bar
- **dash** – bool If set to True, the bar is dashed rather than solid

Returns None

Examples

```
>>> # Place a green time bar at 2017-07-17 00:00:00
>>> import pytplot
>>> pytplot.timebar(1500249600, color='green')
```

```
>>> # Place a dashed data bar at 5500 on the y axis
>>> pytplot.timebar(5500, dashed=True, databar=True)
```

```
>>> Place 3 magenta time bars of thickness 5
    at [2015-12-26 05:20:01, 2015-12-26 08:06:40, 2015-12-26 08:53:19]
    for variable 'sgx' plot
>>> pytplot.timebar([1451107201,1451117200,1451119999],'sgx',color='m',thick=5)
```

pytplot.**timespan**(*t1*, *dt*, *keyword='days'*)

This function will set the time range for all time series plots. This is a wrapper for the function "xlim" to better handle time axes.

Parameters

- **t1** – flt/str The time to start all time series plots. Can be given in seconds since epoch, or as a string in the format "YYYY-MM-DD HH:MM:SS"
- **dt** – flt The time duration of the plots. Default is number of days.
- **keyword** – str Sets the units of the "dt" variable. Days, hours, minutes, and seconds are all accepted.

Returns None

Examples

```
>>> # Set the timespan to be 2017-07-17 00:00:00 plus 1 day
>>> import pytplot
>>> pytplot.timespan(1500249600, 1)
```

```
>>> # The same as above, but using different inputs
>>> pytplot.timespan("2017-07-17 00:00:00", 24, keyword='hours')
```

pytplot.**timestamp**(*val*)

This function will turn on a time stamp that shows up at the bottom of every generated plot.

Parameters

val str A string that can either be 'on' or 'off'.

Returns None

Examples # Turn on the timestamp import pytplot pytplot.timestamp('on')

pytplot.**tplot_names**()

This function will print out and return a list of all current Tplot Variables stored in the memory.

Parameters **None** –

Returns

list of str A list of all Tplot Variables stored in the memory

Return type list

Examples

```
>>> import pytplot
>>> x_data = [1,2,3,4,5]
>>> y_data = [1,2,3,4,5]
>>> pytplot.store_data("Variable1", data={'x':x_data, 'y':y_data})
>>> tnames = pyplot.tplot_names()
0 : Variable 1
```

pytplot.**tplot_rename**(*old_name*, *new_name*)

This function will rename tplot variables that are already stored in memory.

Parameters

- **old_name** – str Old name of the Tplot Variable
- **new_name** – str New name of the Tplot Variable

Returns None

Examples

```
>>> # Rename Variable 1 to Variable 2
>>> import pytplot
>>> pytplot.tplot_rename("Variable1", "Variable2")
```

pytplot.**xlim**(*min*, *max*)

This function will set the x axis range for all time series plots

Parameters

- **min** – flt The time to start all time series plots. Can be given in seconds since epoch, or as a string in the format "YYYY-MM-DD HH:MM:SS"

- **max** – flt The time to end all time series plots. Can be given in seconds since epoch, or as a string in the format "YYYY-MM-DD HH:MM:SS"

Returns None

Examples

```
>>> # Set the timespan to be 2017-07-17 00:00:00 plus 1 day
>>> import pytplot
>>> pytplot.xlim(1500249600, 1500249600 + 86400)
```

```
>>> # The same as above, but using different inputs
>>> pytplot.xlim("2017-07-17 00:00:00", "2017-07-18 00:00:00")
```

pytplot.**ylim**(*name*, *min*, *max*)

This function will set the y axis range displayed for a specific tplot variable.

Parameters

- **name** – str The name of the tplot variable that you wish to set y limits for.
- **min** – flt The start of the y axis.
- **max** – flt The end of the y axis.

Returns None

Examples

```
>>> # Change the y range of Variable1
>>> import pytplot
>>> x_data = [1,2,3,4,5]
>>> y_data = [1,2,3,4,5]
>>> pytplot.store_data("Variable1", data={'x':x_data, 'y':y_data})
>>> pytplot.ylim('Variable1', 2, 4)
```

pytplot.**zlim**(*name*, *min*, *max*)

This function will set the z axis range displayed for a specific tplot variable. This is only used for spec plots, where the z axis represents the magnitude of the values in each bin.

Parameters

- **name** – str The name of the tplot variable that you wish to set z limits for.
- **min** – flt The start of the z axis.
- **max** – flt The end of the z axis.

Returns None

Examples

```
>>> # Change the z range of Variable1
>>> import pytplot
>>> x_data = [1,2,3]
>>> y_data = [ [1,2,3] , [4,5,6], [7,8,9] ]
>>> v_data = [1,2,3]
```

(continues on next page)

(continued from previous page)

```
>>> pytplot.store_data("Variable3", data={'x':x_data, 'y':y_data, 'v':v_data})
>>> pytplot.zlim('Variable1', 2, 3)
```

CHAPTER 7

Plotting

7.1 tplot function

All plotting is called through the "tplot()" function. If you'd like to customize how you plots are displayed, it is assume that you have set them up prior to calling this function.

pytplot.**tplot**(*name*, *var_label=None*, *slice=False*, *combine_axes=True*, *nb=False*, *save_file=None*, *gui=False*, *qt=False*, *bokeh=False*, *save_png=None*, *display=True*, *testing=False*, *extra_functions=[]*, *extra_function_args=[]*, *vert_spacing=None*, *pos_2d=False*, *pos_3d=False*, *exec_qt=True*, *window_name='Plot'*, *interactive=False*)

This is the function used to display the tplot variables stored in memory. The default output is to show the plots stacked on top of one another inside of a qt window

Parameters

- **name** – str / list List of tplot variables that will be plotted
- **var_label** – str, optional The name of the tplot variable you would like as a second x axis.
- **slice** – bool, optional If True, a secondary interactive plot will be generated next to spectrogram plots. Mousing over the spectrogram will display a slice of data from that time on the interactive chart.
- **combine_axes** – bool, optional If True, the axes are combined so that they all display the same x range. This also enables scrolling/zooming/panning on one plot to affect all of the other plots simultaneously.
- **nb** – bool, optional If True, the plot will be displayed inside of a current Jupyter notebook session.
- **save_file** – str, optional A full file name and path. If this option is set, the plot will be automatically saved to the file name provided in an HTML format. The plots can then be opened and viewed on any browser without any requirements.
- **bokeh** – bool, optional If True, plots data using bokeh Else (bokeh=False or omitted), plots data using PyQtGraph

- **extra_functions** – func, optional This is an extra function that gets called just prior to the data being plotted. This is useful if you'd like to build your own Qt display that reacts to the mouse movement. Built in displays can be found in the AncillaryPlots folder.
- **extra_function_args** – list of tuples, optional These are the arguments to give your extra_functions
- **vert_spacing** – int Thes distance in pixels you'd like the plots to be
- **gui** – bool, optional If True, then this function will output the 2 HTML components of the generated plots as string variables. This is useful if you are embedded the plots in your own GUI. For more information, see http://bokeh.pydata.org/en/latest/docs/user_guide/embed.html
- **qt** – bool, optional If True, then this function will display the plot inside of the Qt window. From this window, you can choose to export the plots as either an HTML file, or as a PNG.
- **save_png** – str, optional A full file name and path. If this option is set, the plot will be automatically saved to the file name provided in a PNG format.
- **display** – bool, optional If True, then this function will display the plotted tplot variables. Necessary to make this optional so we can avoid it in a headless server environment.
- **testing** – bool, optional If True, doesn't run the '(hasattr(sys, 'ps1'))' line that makes plots interactive - i.e., avoiding issues

Returns None

Examples

```
>>> #Plot a single line in bokeh
>>> import pytplot
>>> x_data = [2,3,4,5,6]
>>> y_data = [1,2,3,4,5]
>>> pytplot.store_data("Variable1", data={'x':x_data, 'y':y_data})
>>> pytplot.tplot("Variable1",bokeh=True)
```

```
>>> #Display two plots
>>> x_data = [1,2,3,4,5]
>>> y_data = [[1,5],[2,4],[3,3],[4,2],[5,1]]
>>> pytplot.store_data("Variable2", data={'x':x_data, 'y':y_data})
>>> pytplot.tplot(["Variable1", "Variable2"])
```

```
>>> #Display 2 plots, using Variable1 as another x axis
>>> x_data = [1,2,3]
>>> y_data = [ [1,2,3] , [4,5,6], [7,8,9] ]
>>> v_data = [1,2,3]
>>> pytplot.store_data("Variable3", data={'x':x_data, 'y':y_data, 'v':v_data})
>>> pytplot.options("Variable3", 'spec', 1)
>>> pytplot.tplot(["Variable2", "Variable3"], var_label='Variable1')
```

```
>>> #Plot all 3 tplot variables, sending the output to an HTML file
>>> pytplot.tplot(["Variable1", "Variable2", "Variable3"], save_file='C:/temp/
↪pytplot_example.html')
```

```
>>> #Plot all 3 tplot variables, sending the HTML output to a pair of strings
>>> div, component = pytplot.tplot(["Variable1", "Variable2", "Variable3"],␣
↪gui=True)
```

(continues on next page)

(continued from previous page)

7.2 Oveplotting

To combine two or more variables in the same plot, you need to create a new variable like so:

```
pytplot.store_data("new_variable", data=["variable1_to_overplot", "variable2_to_
↪overplot"])
```

Then when you plot this new variable, it will be a combination of the two variables given in "data".

Note: Each variable should still retain the plot options you set for it, but I am still working out the kinks.

7.3 Extra X axes

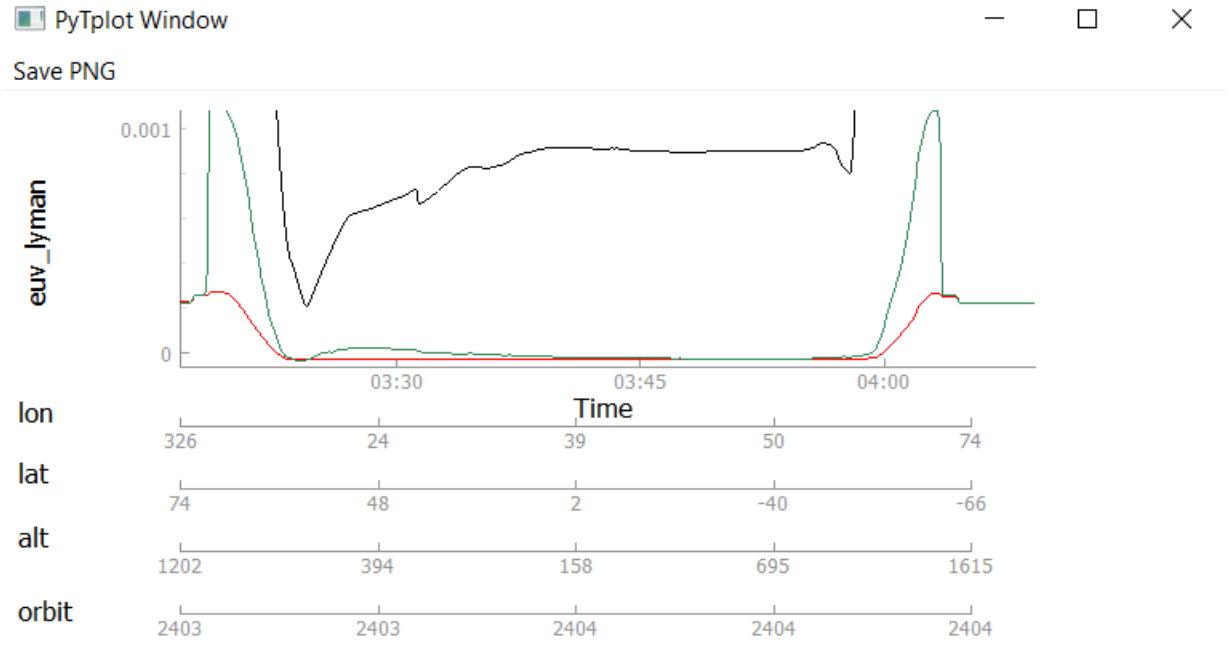
A commonly used feature of tplot is adding extra x axes (in addition to time), on the bottom of the plot.

To do so in pytplot, specify which tplot variable(s) you'd like to to be included on the axis by passing them to the "var_label" option in tplot:

```
pytplot.tplot("variable1", var_label = ["variable2", "variable3"])
```

Note: Unfortunately, in the Bokeh plots currently the extra x axes must be linearly increasing in order to display properly. Hopefully we can determine a way to map variables onto the axes at some point in the future.

A common use case would be orbit numbers or spacecraft position. Here is an example of multiple x axes below:



CHAPTER 8

Linking Two Variables and Non-Time Series Plots

Sometimes plots other than time series may be desired. Typically, this is either altitude or latitude/longitude plots.

Because tplot variables always have time on their x-axis, in order to get these types of plots you need to tell pytplot that two variables are related to one another.

In other words, if you have a tplot variable for altitude named "spacecraft_alt" and while the data is stored in the variable "spacecraft_data", you can tell pytplot that the two are related with the following command:

```
pytplot.link("spacecraft_data", "spacecraft_alt", link_type = 'alt')
```

If the data are not on the same time cadence, the linked variable will be interpolated to match that of the data variable at the time of plotting.

8.1 Why Linking?

One may ask, why not just allow altitude/latitude/longitude to be the xaxis of the tplot variables? The answer, aside from the fact that this is a time-series manipulation library, is that time is still retained in the "backend" of the plots. For instance, the time value will display when you hover over the data, and if you mark specific times with either a vertical Time Bar or Region of Interest, then that point/area will be highlighted on the plot.

8.2 Altitude Plot Example

The following uses data taken from the MAVEN mission:

```
# Store Data from a dictionary variable named "insitu" (read into python from␣
↪elsewhere)
pytplot.store_data('sc_alt', data={'x':insitu['Time'] , 'y':insitu['SPACECRAFT'][
↪'ALTITUDE']})
pytplot.store_data('euv_low', data={'x':insitu['Time'] , 'y':insitu['EUV'][
↪'IRRADIANCE_LOW']})
```

(continues on next page)

(continued from previous page)

```
pytplot.store_data('euv_lyman', data={'x':insitu['Time'] , 'y':insitu['EUV'][
↪'IRRADIANCE_LYMAN']})
pytplot.store_data('euv_mid', data={'x':insitu['Time'] , 'y':insitu['EUV'][
↪'IRRADIANCE_MID']})

# Link the EUV variables to "sc_alt"
pytplot.link(["euv_mid"], "sc_alt", link_type='alt')
pytplot.link(["euv_low"], "sc_alt", link_type='alt')
pytplot.link(["euv_lyman"], "sc_alt", link_type='alt')
#Specify that you'd like to overplot
pytplot.store_data('euv', data=["euv_low","euv_mid","euv_lyman"])

#Set Plot options
pytplot.options("euv", 'alt', 1)
pytplot.options("euv", 'ylog', 1)
pytplot.ylim("euv", .000001, .02)
pytplot.options("euv", "legend_names", ["Low", "Mid", "Lyman"])

#Add a big blue marker at 3:23:00
pytplot.timebar('2015-12-25 03:23:00', varname='euv', color='blue', thick=10)

#Plot!
pytplot.tplot("euv")
```

8.3 Map Plot Example

The following is the same example as above, but this time plotted vs latitude and longitude over the surface of Mars. It also plots only the euv_lyman variable:

```
# Store Data from a dictionary variable named "insitu" (read into python from␣
↪elsewhere)
pytplot.store_data('sc_lon', data={'x':insitu['Time'] , 'y':insitu['SPACECRAFT']['SUB_
↪SC_LONGITUDE']})
pytplot.store_data('sc_lat', data={'x':insitu['Time'] , 'y':insitu['SPACECRAFT']['SUB_
↪SC_LATITUDE']})
pytplot.store_data('euv_lyman', data={'x':insitu['Time'] , 'y':insitu['EUV'][
↪'IRRADIANCE_LYMAN']})

# Link latitude and longitude
pytplot.link(["euv_lyman"], "sc_lat", link_type='lat')
pytplot.link(["euv_lyman"], "sc_lon", link_type='lon')

# Set Plot options
pytplot.options("euv_lyman", 'map', 1)
pytplot.options("euv_lyman", 'zlog', 1)
pytplot.options("euv_lyman", 'basemap', 'C:/maps/MOLA_BW_2500x1250.jpg'))

#Add a big blue marker at 3:23:00
pytplot.timebar('2015-12-25 03:23:00', varname='euv_lyman', color='blue', thick=20)

# Plot!
pytplot.tplot("euv_lyman")
```

If pyqtgraph is used to plot the map plot instead, a little marker will appear on the map at the time the user is hovering over with their mouse.

CHAPTER 9

Interactive Plots

Part of the appeal of IDL tplot is that its easy to make plots that interact with the primary plotting window, primarily based on the time that the user is hovering the mouse over.

There are several interactive plots that are in pytplot, specifically ones that are useful for the MAVEN spacecraft, but this page will also describe how to create your own interactive plots.

9.1 Spectrogram Slicing

Slicing a spectrogram can be called in a few different ways. The easiest way to do it would be to simply specify "slice=True" when calling tplot

```
pytplot.tplot("spectrogram_data", slice=True)
```

Below is what the spectrogram slicer command looks like in a Bokeh graph.

There are a few options one can set with the options command to modify this plot (see the options section):

Options	Value type	Notes
xlog_slice	bool	Sets x axis on interactive plot to log scale if True.
ylog_slice	bool	Sets y axis on interactive plot to log scale if True.
xrange_slice	flt list	Two numberes that give the x axis range of interactive plots.
yrange_slice	flt list	Two numberes that give the y axis range of interactive plots.
static	str	Creates a non-interactive spec-slicing plot. Datetime string that gives desired time to plot y and z values from a specplot.
static_tavg	str	Creates a non-interactive spec-slicing plot. Datetime string that gives desired time-averaged y and z values to plot from a spec plot.
t_average	int	Seconds around which the cursor is averaged when hovering over spectrogram plots.

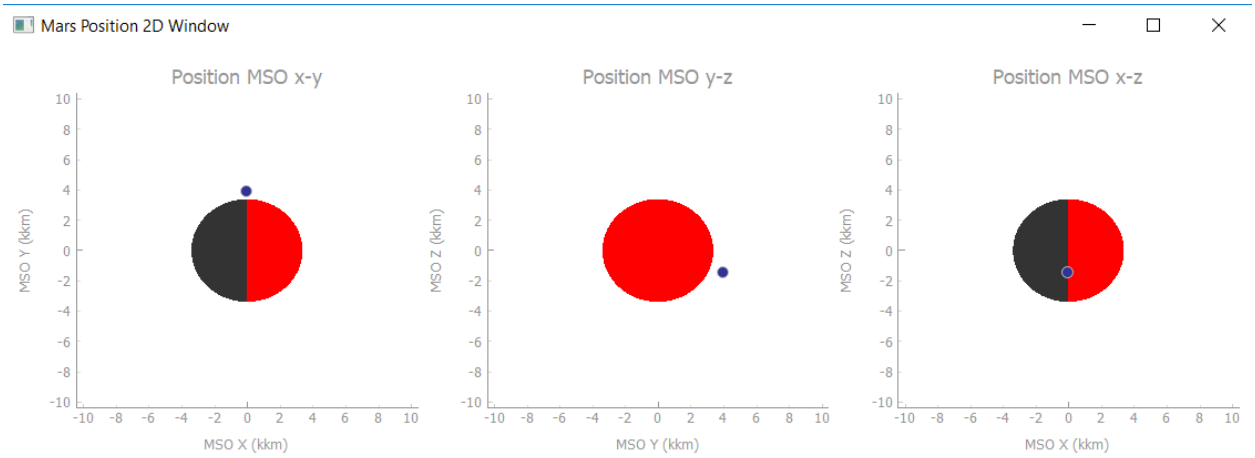
9.2 Mars 2D Position

By linking a data tplot variable to tplot variables that contain spacecraft position information, one can bring up an interactive visualization of the spacecraft's position in MSO coordinates that changes depending on the time the user is hovering over.

For example, suppose you have data in a tplot variable called "maven_data" and position information in the variables "mvn::kp::mso_x/y/z"

```
pytplot.link("maven_data", "mvn::kp::mso_x", "x")
pytplot.link("maven_data", "mvn::kp::mso_y", "y")
pytplot.link("maven_data", "mvn::kp::mso_z", "z")
pytplot.tplot("maven_data", pos_3d=True)
```

A window like the one below would appear, with a dot that moves around depending on where your mouse is hovered over on the main window.



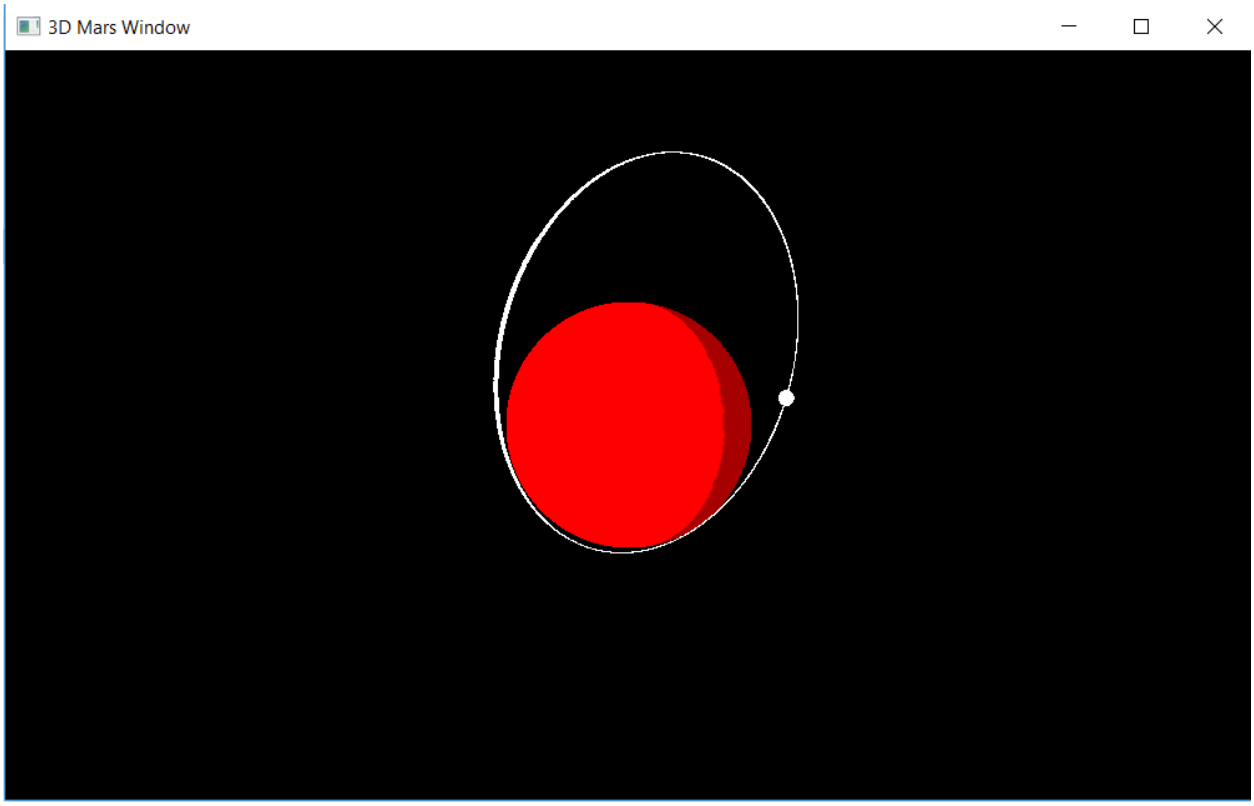
9.3 Mars 3D Position

Similar to the above interactive plot, there is also a 3D viewer that is available if the user has pyopengl installed, which can be installed with the command

```
pip install pyopengl
```

In this case, you can create a 3D spacecraft position viewer of mars after linking x/y/z position information to the data, and plotting it with the command

```
pytplot.tplot("maven_data", pos_3d=True)
```



9.4 Adding your Own Supplementary Qt Plots

Note: This only works when plotting with pyqtgraph. Bokeh only creates static HTML files, so it cannot communicate back to python once created.

If you using pytplot in an IPython environment (including Jupyter notebooks), you can have multiple Qt windows open without issue. This is because IPython continually runs a qt "event loop" in the background, similar to IDL.

However, if you are using python in a non-interactive environment (say just running a script), pyqtgraph needs to start an event loop to run, which is done as the last thing in a call to tplot(). Python will "freeze" and continue looking for events in the tplot window until the tplot window closes.

This means that if you want multiple plots to appear at the same time, you need to supply a function to the tplot() command to call before it starts its event loop. Lets say you have a custom plot you'd like to appear at the same time as the pytplot plot, you would make sure your function gets called with the following addition to the tplot command:

```
pytplot.tplot(['variables_to_plot'], extra_functions=[your_func])
```

Or, if you need to supply arguements to your function:

```
pytplot.tplot(['variables_to_plot'], extra_functions=[your_func], extra_function_
↪args=[(arg1, arg2, arg3)])
```

Why would you want to ensure plots appear at the same time? Mainly for interactivity purposes, as described in the following section.

Alternatively, you can tell the tplot command to not begin the event loop at all, and you can call it yourself later. This can be done simply by specifying:

```
pytplot.tplot(['variables_to_plot'], exec_qt=False)
```

Then you can run your own functions and begin the event loop after calling tplot.

9.5 Adding Interactivity

The way that the interactive plots work is that any time a mouse is moved in a pyqtgraph plotting window, pytplot will call all functions "registered" to its HoverTime class. This class will supply the function with the new time the user is hovering over, and the tplot variable that the user is hovering over.

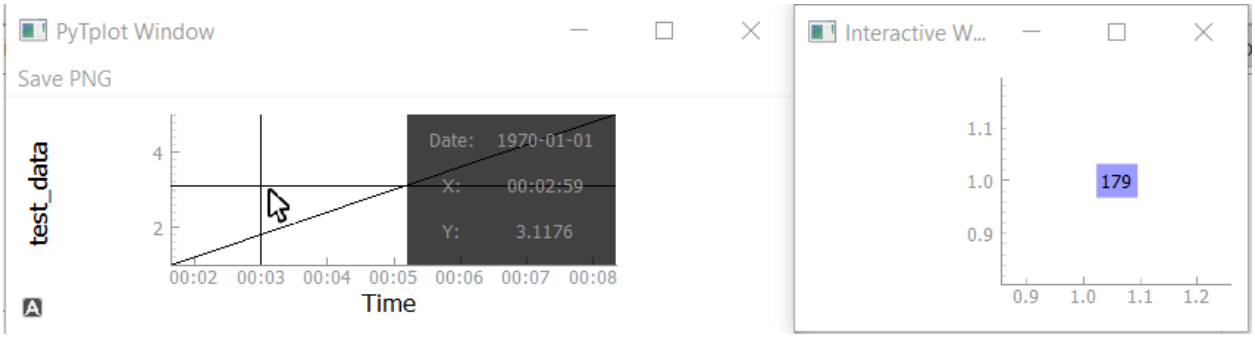
If you have a function that you'd like called whenever the user hovers over a spot on the plots (say it updates your own personal plot window), you can register it like so:

```
pytplot.hover_time.register_listener(your_update_func)
```

9.6 Custom Interactive Example

This is a very simple script that will create a qt window using pyqtgraph that displays what time the user is hovering over.

```
import pytplot
import pyqtgraph as pg

window = pg.GraphicsWindow()

def text_window():

    # Set up plotting window
    window.setWindowTitle('Interactive Window')
    plot = window.addPlot()

    # Add the text item (with crazy HTML inserted, this was the textitem example on␣
↪pyqtgraph's website)
    textitem = pg.TextItem(html='<div style="text-align: center"><span style="color:
↪#FFF;">This is the</span><br><span style="color: #FF0; font-size: 32pt;">PEAK</span>
↪</div>', anchor=(-0.3,0.5), border='w', fill=(0, 0, 255, 100))
    plot.addItem(textitem)
    textitem.setPos(1,1)

    # Define what the plot will do when the user hovers over a new time
    def update(time, name):
        textitem.setText(str(time))

    # Register to update function above to pytplot
    pytplot.hover_time.register_listener(update)

pytplot.store_data("test_data", data={'x':[100,200,300,400,500], 'y':[1,2,3,4,5]})

pytplot.tplot("test_data", extra_functions=[text_window], extra_function_args=[()])
```



For a more complex example, you can take a look at the spec_slicer, 2d_position_mars, and 3d_position_mars functions.

9.7 GUI Creation

When plots are created in pytplot using pyqtgraph, they are actually stored in a list alongside a list of names at the global level, which can be accessed via

```
pytplot.pytplotWindow_names
pytplot.pytplotWindows
```

You can specify window names when you call pytplot so that you can reference them later to build a GUI. For example

```
pytplot.tplot('mvn_kp::spacecraft::altitude', exec_qt=False, window_name='PYDIVIDE_
↪MAP2D')
```

The above command will a) stop the qt event loop from starting and b) name the window created "PYDIVIDE_MAP2D". Then you can find the window again by

```
for i, plot_name in enumerate(pytplot.pytplotWindow_names):
            if plot_name == 'PYDIVIDE_MAP2D':
                    The_window_I_need = pytplot.pytplotWindows[i]
```

This can be useful if you want to add the window to a GUI you are creating.

9.8 GUI Creation Example

Below is some sample code of how the MAVEN SDC creates a simple GUI from pytplot

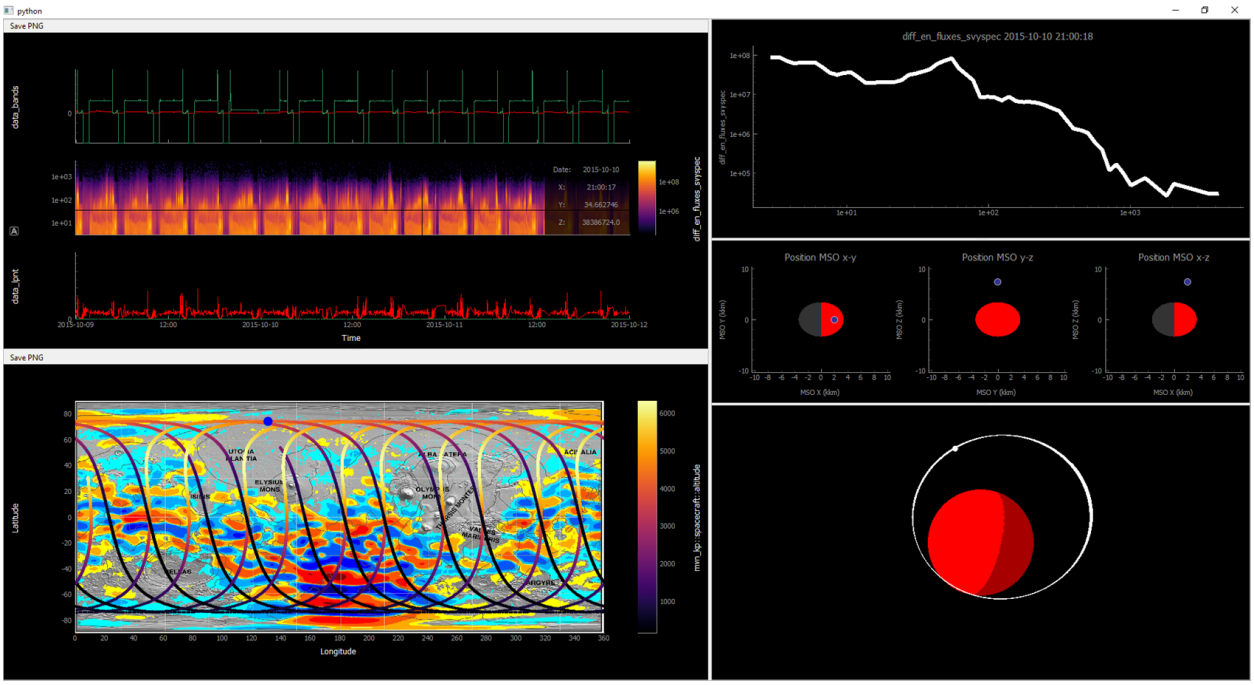
```
    # Load in MAVEN Data from PySPEDAS (variables assumed to be filled in from␣
↪elsewhere)
    tplot_names = pyspedas.maven_load(filenames=filenames, instruments=instruments,␣
↪level=level, type=type, start_date=start_date, end_date=end_date)

    # Change Altitude information to plot as a Map
    pytplot.options('mvn_kp::spacecraft::altitude', 'map', 1)

    # Grab local Map file and specify it as the background map
    map_file = os.path.join(os.path.dirname(__file__), 'basemaps', 'MAG_Connerny_2005.
↪jpg')
    pytplot.options('mvn_kp::spacecraft::altitude', 'basemap', map_file)
```

(continues on next page)

(continued from previous page)

```
    # Plot items without execution of the qt event loop
    pytplot.tplot(tplot_names, pos_2d=True, pos_3d=True, interactive=True, exec_
↪qt=False, window_name='PYDIVIDE_PLOT')
    pytplot.tplot('mvn_kp::spacecraft::altitude', exec_qt=False, window_name=
↪'PYDIVIDE_MAP2D', extra_functions=[],
                                extra_function_args=[])

    # Build up the GUI from QSplitter items
    app = QtGui.QApplication([])
win = QtGui.QMainWindow()
app.setStyle("Fusion")

plot_splitter = QtGui.QSplitter(QtCore.Qt.Vertical, frameShape=QtGui.QFrame.
↪StyledPanel,
                                    frameShadow=QtGui.QFrame.Plain)
ancillary_splitter = QtGui.QSplitter(QtCore.Qt.Vertical, frameShape=QtGui.QFrame.
↪StyledPanel,
                                        frameShadow=QtGui.QFrame.Plain)
main_splitter = QtGui.QSplitter(QtCore.Qt.Horizontal, frameShape=QtGui.QFrame.
↪StyledPanel,
                                    frameShadow=QtGui.QFrame.Plain)
main_splitter.addWidget(plot_splitter)
main_splitter.addWidget(ancillary_splitter)

for i, plot_name in enumerate(pytplot.pytplotWindow_names):
    if plot_name == 'PYDIVIDE_PLOT':
        plot_splitter.addWidget(pytplot.pytplotWindows[i])

for i, plot_name in enumerate(pytplot.pytplotWindow_names):
    if plot_name == 'PYDIVIDE_MAP2D':
        plot_splitter.addWidget(pytplot.pytplotWindows[i])

for i, plot_name in enumerate(pytplot.pytplotWindow_names):
    if plot_name == 'Spec_Slice':
        ancillary_splitter.addWidget(pytplot.pytplotWindows[i])
for i, plot_name in enumerate(pytplot.pytplotWindow_names):
    if plot_name == '2D_MARS':
        ancillary_splitter.addWidget(pytplot.pytplotWindows[i])
for i, plot_name in enumerate(pytplot.pytplotWindow_names):
    if plot_name == '3D_MARS':
        ancillary_splitter.addWidget(pytplot.pytplotWindows[i])

main_splitter.show()

    # Finally, start the event loop
    import sys
if (sys.flags.interactive != 1) or not hasattr(QtCore, 'PYQT_VERSION'):
    app.exec_()
```

Below is a sample of what the created interactive GUI looks like.


python
Save PNG
data_bands
data_bpnt
Time
diff_en_fluxes_svyspec
diff_en_fluxes_svyspec 2015-10-10 21:00:18
Position MSO x-y
Position MSO y-z
Position MSO x-z
MSO X (km)
MSO Y (km)
MSO Z (km)
Save PNG
Latitude
Longitude
mvn_kp::spacecraft::altitude

CHAPTER 10

Math Routines

10.1 Arithmetic

pytplot.tplot_math.**add**(*tvar1*, *tvar2*, *new_tvar=None*)

Adds two tplot variables together. Will interpolate if the two are not on the same time cadence.

Parameters

- **tvar1** – str Name of first tplot variable.
- **tvar2** – int/float Name of second tplot variable
- **new_tvar** – str Name of new tvar for added data. If not set, then the data in tvar1 is replaced.

Returns None

Examples

```
>>> pytplot.store_data('a', data={'x':[0,4,8,12,16], 'y':[1,2,3,4,5]})
>>> pytplot.store_data('c', data={'x':[0,4,8,12,16,19,21], 'y':[1,4,1,7,1,9,1]})
>>> pytplot.add('a','c','a+c')
```

pytplot.tplot_math.**subtract**(*tvar1*, *tvar2*, *new_tvar=None*)

Subtracts two tplot variables. Will interpolate if the two are not on the same time cadence.

Parameters

- **tvar1** – str Name of first tplot variable.
- **tvar2** – int/float Name of second tplot variable
- **new_tvar** – str Name of new tvar for added data. If not set, then the data in tvar1 is replaced.

Returns None

Examples

```
>>> pytplot.store_data('a', data={'x':[0,4,8,12,16], 'y':[1,2,3,4,5]})
>>> pytplot.store_data('c', data={'x':[0,4,8,12,16,19,21], 'y':[1,4,1,7,1,9,1]})
>>> pytplot.subtract('a','c','a-c')
```

pytplot.tplot_math.**multiply**(*tvar1*, *tvar2*, *new_tvar=None*)

Multiplies two tplot variables. Will interpolate if the two are not on the same time cadence.

Parameters

- **tvar1** – str Name of first tplot variable.
- **tvar2** – int/float Name of second tplot variable
- **new_tvar** – str Name of new tplot variable. If not set, then the data in tvar1 is replaced.

Returns None

Examples

```
>>> pytplot.store_data('a', data={'x':[0,4,8,12,16], 'y':[1,2,3,4,5]})
>>> pytplot.store_data('c', data={'x':[0,4,8,12,16,19,21], 'y':[1,4,1,7,1,9,1]})
>>> pytplot.multiply('a','c','a_x_c')
```

pytplot.tplot_math.**divide**(*tvar1*, *tvar2*, *new_tvar=None*)

Divides two tplot variables. Will interpolate if the two are not on the same time cadence.

Parameters

- **tvar1** – str Name of first tplot variable.
- **tvar2** – int/float Name of second tplot variable
- **new_tvar** – str Name of new tvar for divided data. If not set, then the data in tvar1 is replaced.

Returns None

Examples

```
>>> pytplot.store_data('a', data={'x':[0,4,8,12,16], 'y':[1,2,3,4,5]})
>>> pytplot.store_data('c', data={'x':[0,4,8,12,16,19,21], 'y':[1,4,1,7,1,9,1]})
>>> pytplot.divide('a','c','a_over_c')
```

pytplot.tplot_math.**crop**(*tvar1*, *tvar2*, *replace=True*)

Crops both tplot variable so that their times are the same. This is done automatically by other processing routines if times do not match up.

Parameters

- **tvar1** – str Name of the first tplot variable
- **tvar2** – str Name of the second tplot variable
- **replace** – bool, optional If true, the data in the original tplot variables are replaced. Otherwise, new variables are created.

Returns None

Examples

```
>>> pytplot.store_data('a', data={'x':[0,4,8,12,16], 'y':[1,2,3,4,5]})
>>> pytplot.store_data('b', data={'x':[2,5,8,11,14,17,20], 'y':[[1,1,1,1,1,1,1],[2,
↪2,5,4,1,1],[100,100,3,50,1,1],[4,4,8,58,1,1],[5,5,9,21,1,1],[6,6,2,2,1,1],[7,7,
↪1,6,1,1]]})
>>> pytplot.crop('a','b')
```

pytplot.tplot_math.**tinterp**(*tvar1*, *tvar2*, *replace=False*)

Interpolates one tplot variable to another one's time cadence. This is done automatically by other processing routines.

Parameters

- **tvar1** – str Name of first tplot variable whose times will be used to interpolate tvar2's data.
- **tvar2** – str Name of second tplot variable whose data will be interpolated.
- **replace** – bool, optional If true, the data in the original tplot variable is replaced. Otherwise, a variable is created.

Returns new_var2, the name of the new tplot variable

Examples

```
>>> pytplot.store_data('a', data={'x':[0,4,8,12,16], 'y':[1,2,3,4,5]})
>>> pytplot.store_data('c', data={'x':[0,4,8,12,16,19,21], 'y':[1,4,1,7,1,9,1]})
>>> pytplot.tinterp('a','c')
>>> print(pytplot.data_quants['c_interp'].data)
```

10.2 Add Across Columns

pytplot.tplot_math.**add_across**(*tvar*, *column_range=None*, *new_tvar=None*)

Adds across columns in the tplot variable

Note: This analysis routine assumes the data is no more than 2 dimensions. If there are more, they may become flattened!

Parameters

- **tvar** – str Name of tplot variable.
- **column_range** – list of ints The columns to add together. For example, if [1,4] is given here, columns 1, 2, 3, and 4 will be added together. If not set, then every column is added.
- **new_tvar** – str Name of new tvar for averaged data. If not set, then the variable is replaced

Returns None

Examples

```
>>> #Add across every column in the data
>>> pytplot.store_data('d', data={'x':[2,5,8,11,14,17,21], 'y':[[1,1,50],[2,2,3],
↪[100,4,47],[4,90,5],[5,5,99],[6,6,25],[7,7,-5]]})
>>> pytplot.add_across('d',new_tvar='d_aa')
>>> print(pytplot.data_quants['d_aa'].data)
```

```
>>> #Add across specific columns in the data
>>> pytplot.store_data('b', data={'x':[2,5,8,11,14,17,20], 'y':[[1,1,1,1,1,1],[2,
↪2,5,4,1,1],[100,100,3,50,1,1],[4,4,8,58,1,1],[5,5,9,21,1,1],[6,6,2,2,1,1],[7,7,
↪1,6,1,1]]})
>>> pytplot.add_across('b',column_range=[[1,2],[3,4]],new_tvar='b_aap')
>>> print(pytplot.data_quants['b_aap'].data)
```

10.3 Average over time

pytplot.tplot_math.**avg_res_data**(*tvar*, *res*, *new_tvar=None*)

Averages the variable over a specified period of time.

Parameters

- **tvar1** – str Name of tplot variable.
- **res** – int/float The new data resolution
- **new_tvar** – str Name of new tvar for averaged data. If not set, then the data in tvar is replaced.

Returns None

Examples

```
>>> #Average the data over every two seconds
>>> pytplot.store_data('d', data={'x':[2,5,8,11,14,17,21], 'y':[[1,1,50],[2,2,3],
↪[100,4,47],[4,90,5],[5,5,99],[6,6,25],[7,7,-5]]})
>>> pytplot.avg_res_data('d',2,'d2res')
>>> print(pytplot.data_quants['d'].values)
```

10.4 Clip Data

pytplot.tplot_math.**clip**(*tvar*, *ymin*, *ymax*, *new_tvar=None*)

Change out-of-bounds data to NaN.

Parameters

- **tvar1** – str Name of tvar to use for data clipping.
- **ymin** – int/float Minimum value to keep (inclusive)
- **ymax** – int/float Maximum value to keep (inclusive)
- **newtvar** – str Name of new tvar for clipped data storage. If not specified, tvar will be replaced

Returns None

Examples

```
>>> Make any values below 2 and above 6 equal to NaN.
>>> pytplot.store_data('d', data={'x':[2,5,8,11,14,17,21], 'y':[[1,1],[2,2],[100,
↪100],[4,4],[5,5],[6,6],[7,7]]})
>>> pytplot.clip('d',2,6,'e')
```

10.5 Deflag Data

pytplot.tplot_math.**deflag**(*tvar*, *flag*, *new_tvar=None*)

Change specified 'flagged' data to NaN.

Parameters

- **tvar1** – str Name of tplot variable to use for data clipping.
- **flag** – int,list Flagged data will be converted to NaNs.
- **newtvar** – str Name of new tvar for deflagged data storage. If not specified, then the data in tvar1 will be replaced.

Returns None

Examples

```
>>> # Remove any instances of [100,90,7,2,57] from 'd', store in 'e'.
>>> pytplot.store_data('d', data={'x':[2,5,8,11,14,17,21], 'y':[[1,1],[2,2],[100,
↪4],[4,90],[5,5],[6,6],[7,7]]})
>>> pytplot.deflag('d',[100,90,7,2,57],'e')
```

10.6 Degap Data

pytplot.tplot_math.**degap**(*tvar*, *dt*, *margin*, *func='nan'*, *new_tvar=None*)

Fills gaps in the data either with NaNs or the last number.

Parameters

- **tvar** – str Name of tplot variable to modify
- **dt** – int/float Step size of the data in seconds
- **margin** – int/float, optional The maximum deviation from the step size allowed before degapping occurs. In other words, if you'd like to fill in data every 4 seconds but occasionally the data is 4.1 seconds apart, set the margin to .1 so that a data point is not inserted there.
- **func** – str, optional Either 'nan' or 'ffill', which overrides normal interpolation with NaN substitution or forward-filled values.
- **new_tvar** – str, optional The new tplot variable name to store the data into. If None, then the data is overwritten.

Returns None

Examples

```
>>> # TODO
```

10.7 Derivative

pytplot.tplot_math.**derive**(*tvar*, *new_tvar=None*)

Takes the derivative of the tplot variable.

Parameters

- **tvar** – str Name of tplot variable.
- **new_tvar** – str Name of new tplot variable. If not set, then the data in tvar is replaced.

Returns None

Examples

```
>>> pytplot.store_data('b', data={'x':[2,5,8,11,14,17,20], 'y':[[1,1,1,1,1,1],[2,
↪2,5,4,1,1],[100,100,3,50,1,1],[4,4,8,58,1,1],[5,5,9,21,1,1],[6,6,2,2,1,1],[7,7,
↪1,6,1,1]]})
>>> pytplot.derive('b','dbdt')
>>> print(pytplot.data_quants['dbdt'].values)
```

10.8 Flatten Data

pytplot.tplot_math.**flatten**(*tvar*, *range=None*, *new_tvar=None*)

Divides the column by an average over specified time

Note: This analysis routine assumes the data is no more than 2 dimensions. If there are more, they may become flattened!

Parameters

- **tvar** – str Name of first tplot variable.
- **range** – [int, int], optional The time range to average over. The default is the whole range.
- **new_tvar** – str Name of new tvar for added data. If not set, then a name is made up.

Returns None

Examples

```
>>> # Divide each column by the average of the data between times 8 and 14
>>> pytplot.store_data('d', data={'x':[2,5,8,11,14,17,21], 'y':[[1,1,50],[2,2,3],
↪[100,4,47],[4,90,5],[5,5,99],[6,6,25],[7,7,-5]]})
>>> pytplot.flatten('d',[8,14],'d_flatten')
>>> print(pytplot.data_quants['d_flatten'].values)
```

10.9 Interpolate through NaN values

pytplot.tplot_math.**interp_nan**(*tvar*, *new_tvar=None*, *s_limit=None*)

Interpolates the tplot variable through NaNs in the data. This is basically just a wrapper for xarray's interpolate_na function.

Note: This analysis routine assumes the data is no more than 2 dimensions. If there are more, they may become flattened!

Parameters

- **tvar** – str Name of tplot variable.
- **s_limit** – int or float, optional The maximum size of the gap in seconds to not interpolate over. I.e. if there are too many NaNs in a row, leave them there.
- **new_tvar** – str Name of new tvar for added data. If not set, then the original tvar is replaced.

Returns None

Examples

```
>>> # Interpolate through the np.NaN values
>>> pytplot.store_data('e', data={'x':[2,5,8,11,14,17,21], 'y':[[np.nan,1,1],[np.
↪nan,2,3],[4,np.nan,47],[4,np.nan,5],[5,5,99],[6,6,25],[7,np.nan,-5]]})
>>> pytplot.interp_nan('e','e_nonan',s_limit=5)
>>> print(pytplot.data_quants['e_nonan'].values)
```

10.10 Join/Split Data

pytplot.tplot_math.**join_vec**(*tvars*, *new_tvar=None*, *merge=False*)

Joins 1D tplot variables into one tplot variable.

Note: This analysis routine assumes the data is no more than 2 dimensions. If there are more, they may become flattened!

Parameters

- **tvars** – list of str Name of tplot variables to join together
- **new_tvar** – str, optional The name of the new tplot variable. If not specified, a name will be assigned.
- **merge** – bool, optional Whether or not to merge the created variable into an older variable

Returns None

Examples

```
>>> pytplot.store_data('d', data={'x':[2,5,8,11,14,17,21], 'y':[[1,1,50],[2,2,3],
↪[100,4,47],[4,90,5],[5,5,99],[6,6,25],[7,7,-5]]})
>>> pytplot.store_data('e', data={'x':[2,5,8,11,14,17,21], 'y':[[np.nan,1,1],[np.
↪nan,2,3],[4,np.nan,47],[4,np.nan,5],[5,5,99],[6,6,25],[7,np.nan,-5]]})
>>> pytplot.store_data('g', data={'x':[0,4,8,12,16,19,21], 'y':[[8,1,1],[100,2,3],
↪[4,2,47],[4,39,5],[5,5,99],[6,6,25],[7,-2,-5]]})
>>> pytplot.join_vec(['d','e','g'],'deg')
>>> print(pytplot.data_quants['deg'].values)
```

pytplot.tplot_math.**split_vec**(*tvar*, *new_name=None*, *columns='all'*, *suffix=None*)
Splits up 2D data into many 1D tplot variables.

Note: This analysis routine assumes the data is no more than 2 dimensions. If there are more, they may become flattened!

Parameters

- **tvar** – str Name of tplot variable to split up
- **newtvars** – int/list, optional The names of the new tplot variables. This must be the same length as the number of variables created.
- **columns** – list of ints, optional The specific column numbers to grab from the data. The default is to split all columns.

Returns None

Examples

```
>>> pytplot.store_data('b', data={'x':[2,5,8,11,14,17,20], 'y':[[1,1,1,1,1,1],[2,
↪2,5,4,1,1],[100,100,3,50,1,1],[4,4,8,58,1,1],[5,5,9,21,1,1],[6,6,2,2,1,1],[7,7,
↪1,6,1,1]]})
>>> pytplot.tplot_math.split_vec('b',['b1','b2','b3'],[0,[1,3],4])
>>> print(pytplot.data_quants['b2'].values)
```

10.11 Power Spectrum

pytplot.tplot_math.**pwr_spec**(*tvar*, *nbp=256*, *nsp=128*, *name=None*)
Calculates the power spectrum of a line, and adds a tplot variable for this new spectrogram

Parameters

- **tvar** – str Name of tvar to use
- **nbp** – int, optional The number of points to use when calculating the FFT
- **nsp** – int, optional The number of points to shift over to calculate the next FFT
- **name** – str, optional The name of the new tplot variable created,

Returns None

Examples

```
>>> pytplot.cdf_to_tplot("/path/to/pytplot/testfiles/mvn_euv_l2_bands_20170619_
↪v09_r03.cdf")
>>> pytplot.tplot_math.split_vec('data')
>>> pytplot.pwr_spec('data_0')
>>> pytplot.tplot('data_0_pwrspec')
```

10.12 Resample Data

pytplot.tplot_math.**resample**(*tvar*, *times*, *new_tvar=None*)

Linearly interpolates data to user-specified values. To interpolate one tplot variable to another, use tinterp.

Note: This analysis routine assumes the data is no more than 2 dimensions. If there are more, they may become flattened!

Parameters

- **tvar** – str Name of tvar whose data will be interpolated to specified times.
- **times** – int/list Desired times for interpolation.
- **new_tvar** – str Name of new tvar in which to store interpolated data. If none is specified, tvar will be overwritten

Returns None

Examples

```
>>> # Interpolate data for 'd' to values [3,4,5,6,7,18].
>>> pytplot.store_data('d', data={'x':[2,5,8,11,14,17,21], 'y':[[1,1],[2,2],[100,
↪100],[4,4],[5,5],[6,6],[7,7]]})
>>> pytplot.tplot_resample('d',[3,4,5,6,7,18],'d_resampled')
```

10.13 Spectrum Multiplication

pytplot.tplot_math.**spec_mult**(*tvar*, *new_tvar=None*)

Multiplies the data by the stored spectrogram bins and created a new tplot variable

---

Note: This analysis routine assumes the data is no more than 2 dimensions. If there are more, they may become flattened!

Parameters

- **tvar** – str Name of tplot variable
- **times** – int/list Desired times for interpolation.
- **new_tvar** – str Name of new tvar in which to store interpolated data. If none is specified, a name will be created.

Returns None

Examples

```
>>> pytplot.store_data('h', data={'x':[0,4,8,12,16,19,21], 'y':[[8,1,1],[100,2,3],
↪[4,2,47],[4,39,5],[5,5,99],[6,6,25],[7,-2,-5]],'v':[[1,1,50],[2,2,3],[100,4,47],
↪[4,90,5],[5,5,99],[6,6,25],[7,7,-5]]})
>>> pytplot.spec_mult('h','h_specmult')
>>> print(pytplot.data_quants['h_specmult'].data)
```

Index