Thompson Lab

University of Colorado-Boulder

Thompsonlib User Manual

Author: Graham Greve
graham.greve@colorado.edu

October 17, 2016

Contents

Overview

At the time of writing, there are a number of Python classes which are meant to facilitate quick and easy plotting and analysis. The inspiration for much of the organization comes from Igor Pro. In particular:

- A `Wave()` is a 1D array of points with optional metadata.
- A `Trace()` is a set of two waves typically viewed as a set of (x, y) points.
- A `Fit()` is an optimized fit to a set of (x, y) points conforming to some `FitFunc()` model.
- A `Line()` is a simple object for quickly including a vertical or horizontal line on a plot.
- A `Dataset()` is a mostly an array of data.

These classes form the basis for the analysis suite so they are described first. For information on importing the data from, for example, a LabVIEW data log, see Chapter 4. Unforunately, this manual may not remain completely up to date, but it should provide intuition for the structure and some helpful options. I hope you will then be able to use the code to explore all options available.

Dependencies

The code relies on Python 3, NumPy for numeric operations and importing data, SciPy for curve fitting, matplotlib for plotting, and a few other default packages like os. These packages are all installed automatically when downloading the Anaconda distribution with Spyder editor.

TODO: How to install the TL package so it doesn't have to be included via relative path??? :-)

Waves, Traces, Fits, and Lines

Waves

A `Wave` object contains a 1D set of N points, but it has helpful metadata which finds its way into several other objects. Since it is very simple, it can often be replaced with a normal Python array (`wave=[3,6,2]`), but doing so bypasses several conveniences, like automatic generation of wave statistics.

Like in IGOR, a `Wave` can contain its own x-axis scaling information through the $x0$ and $deltax$ properties, so that plotting a lone `Wave` will use these to generate x points. By default, $x0 = 0$ and the separation between points is 1.

Listing 2.1: Wave() properties

```
self.pts       # array of the wave values
self.errbars   # wave or array() of errbars
self.name      # string name for identification
self.x0        # x value of first point
self.deltax    # spacing between x-points

len(self)      # number of points
self.mean      # mean of pts
self.std       # standard deviation of pts
self.var       # variance of pts
self.min       # minimum value
self.max       # maximum value
```

A good `Wave` definition might look like the following:

Listing 2.2: Wave() initiation

```
pts = data[:, 1] # all rows, second column
wLAPD_mV = Wave(pts,
                errbars=[],
                name='LAPD Power [mV]',
                x0=0,
                deltax=10.0)
```

Wave arithmetic should be handled for call cases, but errors may not be propagated correctly, and you should always double-check the code is working properly.

Wave.**setScaling**(Wave **wave**) *or* Wave.**setScaling**(float **x0**, float **deltax**)

Listing 2.3: Reset scaling information after Wave creation

```
wLAPD_mV.setScaling(wave0) # copy scaling from wave0
wLAPD_mV.setScaling(x0=0, deltax=0.001) # set points to be 0.001 apart
```

The scaling `Wave` can then be retrieved with `wLAPD_mV.getScalingWave()`.

Wave.**setErrs**(list(float **errbars**))

Listing 2.4: Set error bars or remove them

```
wLAPD_mV.setErrs([.1, .1, .5, ...]) # set to an ordered list of error values
wLAPD_mV.setErrs([lower_error, upper_error]) # set lower and upper errors
wLAPD_mV.setErrs([]) # remove error bars
```

Note on error bars: the number of error bar points must equal the number of total points, but assymetric errorbars (low and high) can be added via standard MATPLOTLIB format: `[[lower errors], [upper errors]]`

Wave.**pruneWave**([float **low**, float **high**], Wave **judgeWave**)

Remove point p from a wave *targetWave* if the corresponding point $x = judgeWave[p]$ satisfies $x < low$ or $x > high$.

Listing 2.5: pruneWave format

```
wLAPD_mV.pruneWave([0, 9999], targetWave) # remove points outside this range
```

Traces

A `Trace` can be thought of as an (x, y) relationship of points, but it can actually be made in two ways:

- Provide an *xwave* and a *ywave* of equal length
- Provide a *ywave* and use automatically generated *xwave* points. In Igor, this is like using the automatic x scaling, and you can define a x_0 and Δx within the wave.

The full set of attributes for a `Trace` object are presented below.

Listing 2.6: Trace() properties

```
self.xwave            # wave of x points
self.ywave            # equal length wave of y points
self.name             # str to identify the (x, y) relationship later
self.linestyle (ls)   # include iff you want a line drawn ('-', '--', '-., ':')
self.marker           # default: 'o', could be None
self.markersize       # default: 8
self.color            # str for point or line color
self.cmapstr          # str for colormap, include only if desired
self.stroke           # float thickness of stroke, default: 0.5
self.errbarcol        # str for error bar color may be different
self.plotxerr         # bool for whether to plot x errorbars
self.plotyerr         # bool for whether to plot y errorbars
```

Examples for how to quickly create a 1-wave trace or properly create the 2-wave trace:

Listing 2.7: Trace() creation

```
xs = Wave(data[:, 0], errbars=data[:, 2])
ys = Wave(data[:, 1], errbars=data[:, 3])
myTrace1 = Trace(ys)
myTrace2 = Trace(xs, ys,
                 color='#994444',
                 errbarcol='#cc6666',
                 name='My Xs vs. Ys',
                 ls=':', # dotted line
                 marker='o',
                 markersize=8,
                 stroke=0.5)
```

Trace.**addFit**(FitFunc **fitFunc**, array(float **params**), [optional Fit() arguments])

Quickly create a `Fit` object (see Sect. 2) corresponding to your trace by providing a `FitFunc` and a list of parameter guesses. A separate `Fit()` object is returned which no longer has explicit ties to the `Trace` used so it will not update. Can use any optional `Fit()` arguments like *ls* and *usexerr*.

Listing 2.8: Add Fit to Trace

```
myFit = myTrace.addFit(FitLin, [4.4, 1.5], ls='-', color='#AABBFF')
```

Trace.**plotErrorbars**([bool **plotxerr**, bool **plotyerr**, str **color**])

Error bars are automatically turned on and plotted if they're attached to a wave, but sometimes a particular trace may not look good with error bars, so use this function to disable plotting x or y errorbars, or to change their color.

Listing 2.9: Error Bars on Trace After Creation

```
myTrace.plotErrorbars(plotxerr=True, plotyerr=False, color='#CCDDFF'):
```

Trace.**sort**()

Sort a Trace's *xwave* and *ywave* at the same time based on the *xwave*, such that a line drawn from point to point will go from left to right and never cross back.

Fits

A `Fit` object uses SciPy to optimally fit a set of (x, y) points to a `FitFunc` model (See Section regarding `FitFunc` models). It maintains information about what was fitted as well as metadata for plotting a nice curve. An error should be given if the fit is completely unsuccessful or if the number of points is not equal. By default, the error bars from a `Trace` are used to constrain the fit unless you set them not to.

Listing 2.10: Fit() properties

```
self.xwave          # array/Wave of x points used in fit
self.ywave          # array/Wave of y points used in fit
self.fitFunc        # FitFunc model object used
self.p_guess        # array of floats used as parameter guesses
self.pruneX         # False or (min, max) for slice of x points to fit to
self.pruneY         # False or (min, max) for slice of y points to fit to
self.usexerr        # bool whether to use x errorbars in fit
self.useyerr        # bool whether to use y errorbars in fit
self.name           # string identifier name
self.ls             # what time of line to draw: '-', '--', '-., ':'
self.color          # str color used in plotting
self.fitxvals       # array of linearly spaced x points for the output fit
self.popt           # array of N optimized fit parameters
self.pcov           # N-by-N covariance matrix
self.stdevs         # array of standard deviation uncertainties from the cov. matrix diags.
self.fittedParams   # dictionary of form {'param': (opt-value, stdev)}
self.usedErrBars    # (bool, bool) for whether error bars were used to constrain fit
```

Here are a few example fits:

Listing 2.11: Fit() creation

```
xs, ys = zip(*data[:, 0:2]) # waves are better than arrays though

fit1 = Fit(xs,
           ys,
           FitLor, # predefined fit model
           p_guess=[200, 3, 1, 78], # requires knowledge of param ordering
           name='Fit of my voltages',
           pruneX=(-10, 10), # only fit points with x in (-10, 10)
           pruneY=False, # fit any y points though (default)
           usexerr=True, # use error bars, if any exist
           useyerr=True,
           ls='-',
           color='#ff8888',
           verbose=True) # print output message after fit
fit1 = Fit(xs[0:10], ys[0:10], FitGauss, [200, 1, 1, 0]) # Fit first ten points
```

Standard printing of a `Fit` object, `print(fit1)`, should reprint the output message including nicely formatted optimal parameter values.

FitFuncs

The `FitFunc` fit model object is a little out of place. It contains the following, mostly for self-documentation:

Listing 2.12: FitFunc properties

```
self.name          # useful string name for the form of fit
self.args          # list of parameters in string form
self.helpStr       # str of helpful advice
self.latexFormula  # str for printing out formula when help() is called
self.f             # the FUNCTION defining the fit model
```

Note that the args array always starts with the dependent variable ("x"). To create your own `FitFunc` model, remember to attach your model function f. Use a template like below:

Listing 2.13: Wave properties

```
myFitModel = FitFunc("Special Fit",
                     ["x", "A", "tau"],
                     "Unhelpful help string",
                     "\\cos(A * \\sin(\\tau x))")
def fit_specialf(x, m, b):
  return np.cos(A * np.sin( tau * x ))
myFitModel.f = fit_specialf
```

Important: to maintain self-documentation, please use the fitFunc.py library file and add your function into the *AllFits* object. This object has a built in `AllFits.help()` function which should print out a handy chart of all fit models (See Fig. 2.1).

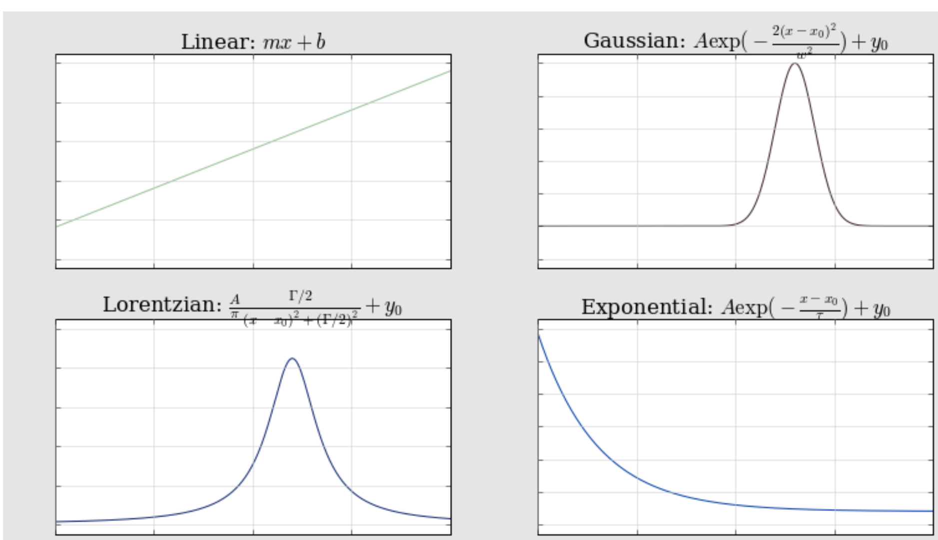

Figure 2.1: Truncated output of calling `AllFits.help()` (with some formatting error).

Lines

A `Line` object is a very simple object which allows MATPLOTLIB to draw a line.

Listing 2.14: Line properties and initiation

```
self.p          # point to draw the line
self.ls         # maybe solid '-' or dashed '--' or dotted ':'
self.vert       # bool for whether it is vertical. Default is horizontal.
self.color      # str of color
self.name       # str identifier name (maybe useless)

horLine  = Line(-34.0, '--', '#f0c8c8', 'The Minimum')
vertLine = Line(33.3,
                ls='-',
                vert=True,
                color='#99ccaa',
                name='The Maximum')
```

Plotting

Plot

The whole point of the preceding structures is to be able to quickly add them to a plot via a list of things to plot. The general plotting function has several options which will be examined through examples. The logic in the code should be pretty easy to follow. The `Plot()` command has the following parameters:

listOfTraces	*list of plot objects*	The main list of `Traces`, `Waves`, `Lines`, and `Fits`. Can be a single item.
title	*str*	Plot title at top of figure
xlabel	*str*	Plot x-axis label
ylabel	*str*	Plot y-axis label
xlog	*bool*	Set to *True* for log scale along x-axis
ylog	*bool*	Set to *True* for log scale along y-axis
xlim	*(float, float) or None*	Manual x-axis range of form *(min, max)*
ylim	*(float, float) or None*	Manual y-axis range of form *(min, max)*
gridlines	*bool*	Set to *False* to hide gridlines
legend	*bool or (str, [])*	*False* to hide; or *("upper left", [trace, numbers, to, label])*
fc	*str*	Background color of figure
fs	*(float, float0)*	The *(width, height)* of the image in inches
dpi	*int*	Pixels per inch (default is 80)

The procedure for figure creation is currently: create the MATPLOTLIB figure, add gridlines, add one plot axis object, set up axis scales, convert `Waves` to `Traces`, plot all `Traces` and `Fits`, plot error bars for traces with error bars, plot all `Lines`, set title, labels, and legends. An example:

Listing 3.1: Using Plot()

```
Plot([pts1, pts2, fit3, fit1, fit2, line1, line2],
     title='Gauss/Lor Fit Test',
     xlabel='Time (ms)',
     ylabel='Frequency (MHz)',
     xlog=False,
     ylog=True,
     xlim=[0, 10],
     gridlines=True,
     fc='#e9e9cc',
     fs=(6.0, 6.0),
     dpi=80)
```

Note that you can use LaTeX in the title and labels, but you will need to enter math mode, and inside of strings, you should double-escape everything so newline (\n), tab (\t), and other characters aren't replaced. For example: `xlabel='$\\tau$ (ms)'`.

Colormaps

A colormap changes point color based on $f(z)$, where the *zwave* array needs to be stored in the *ywave*'s *clist* property. Then the trace has to have the *cmapstr* set to a valid color map string (see Fig. 3.1). Color maps are automatically scaled, so you have to deal with that yourself.

Listing 3.2: Error Bars on Trace After Creation

```
colormask = [0, 0, 1, 1, .6, 1, 0, 0, 0, 1, .6, 1]
pts = TL.Wave([0, 1, 2, 6, 4, 3, 7, 5, 12, 2, 3, 5], name='MyWave', clist=colormask)
trace = TL.Trace(pts, cmapstr='plasma')
TL.Plot(trace, title="Color wave test")
```

Figure 3.1: Default colormaps.

Importing Data

Importing data is handled through the fairly versatile NumPy `genfromtxt()` function.

Datasets

The `Dataset` object is mostly a 2D numpy array $\mathcal{D}$ with some metadata, created as a result of importing a data file of rows of different points, columns of different fields. The default ordering has element $\mathcal{D}$`[i][j]` for row `i` and column `j`, but this can be transposed using the `unpack=True` parameter. Note that for a `Dataset` that contains both strings and floats, a numpy *structured array* is created, and this cannot be indexed numerically except by row. Instead, *headers* can be used to reference a column of interest by name. For example, suppose our data set looks like:

Listing 4.1: Example data contents of a .txt file

```
time   AI0    GenFit1   StrVar   ...   AOM1
1.00   0.01   0.01E-3    NaN     ...    5.0
1.01   0.02   0.00E-3    NaN     ...    5.0
1.02   0.01   0.01E-3    NaN     ...    0.0
                    ...
5.00   0.01   3.14E-1    NaN     ...    0.0
```

Importing this (either with the TSV or Datalog import functions) will create a structured array because of the "NaN" column, allowing access to the first point ($time = 1.00$) with `data[0]` or access to all *AI0* points through `data['AI0']`. But we cannot hope to use `data[:, 0]` or anything to get column 0 because a structured array cannot be indexed in this manner. If the data contained in the file is entirely numeric, a regular, indexable 2D array is created (still compatible with header strings).

Finally, `Waves` can be attached to `Datasets`, see the next section. The `Dataset` object structure is (see file for arguments that may go into `genfromtxt`):

Listing 4.2: Dataset structure

```
self.data        # np.array from np.genfromtext()
self.name        # string identifier
self.filename    # string filename with extension
self.folder      # string of last folder (eg the date)
self.rows        # number of "data points"
self.cols        # number of "data fields"
self.headers     # column headers (in numpy, called "names")
self.isDatalog   # boolean for whether file was Master VI datalog
```

Importing Datasets

Slightly specialized functions for importing common datasets are detailed here.

importDatalog([Rb style Datalogs from Master VI])

Listing 4.3: importDatalog

```
importDatalog(filename,       # automatically appends "_XYGraphDataLog0.txt"
              name='First Dataset',
              headers=True,  # automatic from LabVIEW
              skip_header=0, # use all data points
              skip_footer=0,
              usecols=[],    # import all columns
              unpack=True,   # transpose data
              max_rows=0)    # use all data points
```

importTimeTrace([Rb style AI Time Traces from Master VI])

Imports 7 column AI trace file and automatically names them AI0 through AI6.

Listing 4.4: importTimeTrace

```
importTimeTrace(filename,
                name='',
                skip_header=3, # three metadata lines for Igor
                skip_footer=3, # three metadata lines for Igor
                usecols=[],    # import all AI traces
                unpack=True,   # transpose data
                max_rows=0)    # use all data points
```

importAIPoints([Sr style AI log file])

Creates columns for time (first column) and AI7 through AI0 (from columns $2, 4, ..., 16$).

Listing 4.5: importAIPoints

```
importAIPoints(filename,
               name='',
               headers=False, # will be automatic, but ["could", "overwrite", "here"]
               skip_header=0, # use all points
               skip_footer=3, # use all points
               unpack=True,   # transpose data
               max_rows=0)    # use all points
```

importCSV([Regular comma-separated table])

Listing 4.6: importCSV

```
importCSV(filename,
          name='',
          headers=False, # no column names, but ["could", "set", "here"]
          skip_header=0, # use all points
          skip_footer=0, # use all points
          usecols=[],    # use all columns
          unpack=True,   # transpose data
          max_rows=0):   # use all points
```

importTSV([Regular tab-separated table])

Listing 4.7: importTSV

```
importTSV(filename,
          name='',
          headers=False, # no column names, but ["could", "set", "here"]
          skip_header=0, # use all points
          skip_footer=0, # use all points
          usecols=[],    # use all columns
          unpack=True,   # transpose data
          max_rows=0):   # use all points
```