profiletools Documentation

Release 0.1

Mark Chilenski

Sep 17, 2017

Contents

Source home: https://github.com/markchil/profiletools

CHAPTER 1

Overview

profiletools is a Python package that provides a convenient, powerful and extensible way of working with multivariate data, particularly profile data from magnetic plasma confinement devices. *profiletools* features deep integration with `gptools` to support Gaussian process regression (GPR).

CHAPTER 2

Notes

profiletools has been developed and tested on Python 2.7 and scipy 0.14.0. It may work just as well on other versions, but has not been tested.

profiletools uses the module `gptools` for GPR. You can find the source at https://github.com/markchil/gptools/ and the documentation at http://gptools.readthedocs.org/

profiletools uses the module `eqtools` for tokamak coordinate transformations. You can find the source at https://github.com/PSFCPlasmaTools/eqtools/ and the documentation at http://eqtools.readthedocs.org/

If you find this software useful, please be sure to cite it:

M.A. Chilenski (2014). profiletools: Classes for working with profile data of arbitrary dimension, GNU General Public License. github.com/markchil/profiletools

Once I put together a formal publication on this software and its applications, this readme will be updated with the relevant citation.

CHAPTER 3

Contents

The `profiletools` data model

The `Profile` class

The core class of `profiletools` is the `Profile`. This class is designed primarily to hold point measurements of some quantity, which may depend on an arbitrary number of variables and can be sampled at arbitrary locations – there is no implicit assumption that observations lie on an orderly grid. Internally, a `Profile` instance stores the independent variables in attribute *X*. *X* is an array with shape (*M*, *X_dim*), where *M* is the number of observations and *X_dim* is the number of independent variables. The observations themselves are stored in the attribute *y*, which is an array of shape (*M*,). This is essentially how a sparse matrix is stored and is how `profiletools` can be so flexible about how many independent variables there are and where they are sampled. There can be uncertainties on both the independent variables (stored in the attribute *err_X*) and on the dependent variable (stored in the attribute *err_y*).

Channels

`profiletools` understands that particular data should be treated as a unit during averaging and so forth. Such a unit could correspond to all of the points taken at a given time, or all of the points taken by a given instrument. The attribute *channels* is an array with shape (*M*, *X_dim*). By default this array is just a copy of *X* such that measurements at the exact same locations are grouped together. But, suppose you have sensors at different locations taking time-resolved measurements. Hence, *X_dim* is two: the first column of *X* is the time and the second is the spatial coordinate of the sensor. But say each sensor has a coordinate that varies slightly in time: just using the default choice for *channels* will cause each individual measurement from each sensor to be treated as an independent channel, and time averaging will not have the desired effect. Instead, the second column of *channels* can be set such that all measurements from a given sensor have the same value and are hence treated together when averaging data.

Linearly transformed quantities

`Profile` objects can also incorporate quantities which are linear transformations of the underlying point measurements stored in *X* and *y*. Each channel of a transformed sensor is stored in a `Channel` object. This object stores the data values in attribute *y* which has shape (*M*,) along with the associated uncertainty *err_y*. Each measurement y is taken to be a linear transformation $y = Tf(X)$ where X is a collection of *N* points and $f(X)$ refers to the latent variables (i.e., what is stored as *y* in the `Profile` itself). The transformation matrices associated with each of the observations in *y* are stored in the attribute *T* which is an array with shape (*M*, *N*). The locations used are stored in the attribute *X* which has shape (*M*, *N*, *X_dim*), with the associated uncertainties stored in *err_X*. The `Channel` instances associated with a given `Profile` instance are stored in the attribute *transformed*.

Averaging data

Many different techniques for averaging the data and computing the associated uncertainties are supported, refer to `average_points()` for more details. By carrying out all averaging within a given channel using this function, it is straightforward to add additional capabilities as needed.

`gptools` integration

`profiletools` features very tight integration with the `gptools` package (https://github.com/markchil/gptools/, http://gptools.readthedocs.org/) to perform Gaussian process fits. Creating a Gaussian process (GP) for data of arbitrary *X_dim* is as simple as calling the `create_gp()` method of the `Profile` instance. The GP can then be trained by calling `find_gp_MAP_estimate()`. Once this is complete, the smoothed curve can be obtained using `smooth()`. If additional adjustments to the `GaussianProcess` instance are needed, it is kept in the *gp* attribute of the `Profile` instance.

Plasma profile data

`profiletools` is primarily designed for working with profile data from magnetic confinement fusion devices, namely the Alcator C-Mod tokamak at MIT. The `BivariatePlasmaProfile` class is an extension of `Profile` designed for this particular use case.

Data model

Plasma profile data are functions of space (1, 2 or 3 coordinates) and time (hence the term “bivariate” even when *X_dim* is greater than 2). Time is always the first column in *X*, with the remaining spatial coordinates forming the other columns.

Tokamak coordinate systems

`BivariatePlasmaProfile` uses `eqtools` (https://github.com/PSFCPlasmaTools/eqtools/, http://eqtools.readthedocs.org/) to support the myriad coordinate systems used in tokamak research. Coordinate transforms are handled using the `convert_abscissa()` method.

Constraints for Gaussian process regression

`BivariatePlasmaProfile` provides two methods for adding constraints to the Gaussian process created with `create_gp()`: `constrain_slope_on_axis()` applies a zero slope constraint at the magnetic axis and `constrain_at_limiter()` applies approximate zero slope and value constraints at the location of the limiter. Note, however, that both of these constraints are applied automatically when calling `create_gp()`. You can disable them using the *constrain_slope_one_axis* and *constrain_at_limiter* keywords to `create_gp()`, and you can influence their behavior with the *axis_constraint_kwargs* and *limiter_constraint_kwargs* keywords.

Accessing Alcator C-Mod data

`profiletools` provides a collection of functions to access Alcator C-Mod data. This prevents the user from having to remember the diverse set of `MDSplus` calls needed to load the data from the tree and delivers the data in the standard `BivariatePlasmaProfile` class. Notice that each of these are implemented as a function and not a class – that way all of the instances for a given quantity are the same class.

Example

Loading the data

To load the electron density profile from shot 1101014006, simply call the `ne()` function:

```
p = ne(1101014006, include=['CTS', 'ETS'])
```

The optional keyword *include* specifies which signal are included – in this case core and edge Thomson scattering. If you want the data expressed in a specific coordinate system, use the *abscissa* keyword:

```
p = ne(1101014006, abscissa='r/a')
```

Or, call `convert_abscissa()`:

```
p = ne(1101014006)
p.convert_abscissa('r/a')
```

Selecting a time window or specific time points

To request data only from a certain time window, use the *t_min* and *t_max* keywords. For instance, to get the data from 1.0s to 1.5s, you would type:

```
p = ne(1101014006, t_min=1.0, t_max=1.5)
```

If you want to remove points after having created the `BivariatePlasmaProfile`, then you can use the `remove_points()` method:

```
p.remove_points((p.X[:, 0] < t_min) | (p.X[:, 0] > t_max))
```

If you want to only keep points at specific times (such as points at a specific sawtooth phase), you can use the `keep_times()` method. For each time point designated, this will find the point in the profile which is closest. If there are many missing datapoints, blindly applying this technique can result in data far from the desired point being included. Hence, the *tol* keyword will cause `keep_times()` to only keep points that are within *tol* of the target. So, to keep the points within 1ms of 1.0s, 1.1s and 1.3s, you would type:

```
p.keep_times([1.0, 1.1, 1.3], tol=1e-3)
```

Time averaging or using all points

Once the data are loaded and confined to the desired window, you can time-average them. Thomson scattering data have computed uncertainties in the tree, so you can (and should) use a weighted average:

```
p.time_average(weighted=True)
```

There are a wide variety of options for how the data are averaging depending on the specific application – see `average_points()` for more details.

If instead you want to keep all of the points within the designated time window, you can simply drop that axis from *X*. Recall that time is always the first column, so you would call:

```
p.drop_axis(0)
```

Plotting the data and smoothing it with a Gaussian process

You can plot the data simply by calling `plot_data()`.

Once you have picked the slices you want and/or time-averaged the data, you can fit a Gaussian process with the following steps:

```
p.create_gp()
p.find_gp_MAP_estimate()
p.plot_gp(ax='gca')
```

This will plot the smoothed profile on a somewhat sensible grid on the axis created in the previous call to `plot_data()`. `plot_data()` is a convenience method to get a quick look at the smoothed profile. To evaluate the profile on a specific grid, use the `smooth()` method:

```
roa = scipy.linspace(0, 1.2, 100)
mean, stddev = p.smooth(roa)
```

You can also have `smooth()` plot the fit at the same time using the *plot* keyword:

```
ax, mean, stddev = p.smooth(roa, plot=True)
```

Gradients and linear transformations

You can compute gradients simply by passing the *n* keyword:

```
mean_gradient, stddev_gradient = p.smooth(roa, n=1)
```

You can even compute a mixture of values and gradients at once:

```
roa2 = scipy.concatenate((roa, roa))
n = scipy.concatenate((scipy.zeros_like(roa), scipy.ones_like(roa)))
mean, stddev = p.smooth(roa2, n=n)
```

You can even get the covariances by using the *return_cov* keyword:

```
mean, cov = p.smooth(roa2, n=n, return_cov=True)
```

See the documentation for `gptools.GaussianProcess.predict()` for more details (http://gptools.readthedocs.org/en/latest/gptools.html#gptools.gaussian_process.GaussianProcess.predict).

To compute linearly-transformed quantities (such as line or volume integrals), pass your transformation matrix into the *output_transform* keyword:

```
mean, stddev = p.smooth(roa_vals, output_transform=T)
```

Here, *roa_vals* are the M points the density is evaluated at and T is a transformation matrix with shape (N, M) that transforms the values at those M points into the N transformed outputs. `compute_volume_average()` is a convenience method that uses this approach to compute the volume average and its uncertainty.

`compute_a_over_L()` is a convenience method to compute the normalized inverse gradient scale length. This calculation uses the covariance between values and gradients to properly propagate the uncertainty. Since the error propagation equation breaks down in the edge where the value goes to zero, you can set *full_MC* = True to use full Monte Carlo error propagation.

When computing gradients (either directly with `smooth()` or indirectly with `compute_a_over_L()`) it is important to use Markov chain Monte Carlo (MCMC) to integrate over the possible hyperparameters of the model in order to fully capture the uncertainty in the fit. This is accomplished by leaving out the call to `find_gp_MAP_estimate()` and instead setting *use_MCMC=True* when calling `smooth()` or `compute_a_over_L()`. You can control the properties of the MCMC sampler using the keywords for `gptools.GaussianProcess.compute_from_MCMC()` (http://gptools.readthedocs.org/en/latest/gptools.html#gptools.gaussian_process.GaussianProcess.compute_from_MCMC) and `gptools.GaussianProcess.sample_hyperparameter_posterior()` (http://gptools.readthedocs.org/en/latest/gptools.html#gptools.gaussian_process.GaussianProcess.sample_hyperparameter_posterior).

Complete example

The complete example to load and plot the electron density data as a function of r/a from shot 1101014006 averaged over 1.0s to 1.5s is:

```
p = ne(1101014006, t_min=1.0, t_max=1.5, abscissa='r/a')
p.time_average()
p.plot_data()
p.create_gp()
p.find_gp_MAP_estimate()
roa = scipy.linspace(0, 1.2, 100)
ax, mean, std = p.smooth(roa, plot=True, ax='gca')
```

Signals supported

Electron density

The following diagnostics are supported:

- `neCTS()`: Core Thomson scattering.
- `neETS()`: Edge Thomson scattering.
- `neTCI()`: Two-color interferometer. This is a line- integrated diagnostic. Loading the data is rather slow because the quadrature weights must be computed. Fitting the data is rather slow because of the computational cost of including all of the quadrature points in the Gaussian process. There are several parameters that let you adjust the tradeoff between computational time and accuracy, see the documentation for more details.
- `neReflect()`: Scape-off layer reflectometer. Because of how these data are stored and processed you need to be very careful about how you include them in your fits.

Electron temperature

The following diagnostics are supported:

- `TeCTS()`: Core Thomson scattering.

- *TeETS()*: Edge Thomson scattering.
- *TeFRCECE()*: High spatial resolution ECE system.
- *TeGPC()*: Grating polychromator ECE system.
- *TeGPC2()*: Second grating polychromator ECE system.
- *TeMic()*: Michelson interferometer. High frequency space resolution but low temporal resolution.

X-ray emissivity

You must be careful when interpreting the uncertainties on these fits since they are already inverted/smoothed. This is mostly useful for getting a rough look at the results of combining the two AXUV systems.

emissAX() supports both AXA and AXJ through use of the required *system* argument.

Additional patterns and examples

Weighted versus unweighted averaging

Diagnostics like CTS and ETS have computed uncertainties that can be used to weight the data during averaging to give a better representation of the sample statistics. But, the other diagnostics do not: an assumed value (typically 10%) is used when the data are loaded. This should be replaced with the unweighted sample standard deviation when the data are averaged in order to give an honest assessment of the variability in the quantity. To combine weighted and unweighted averaging, you should create the profiles separately:

```
p = Te(1101014006, include=['CTS', 'ETS'], abscissa='r/a', t_min=1.0, t_max=1.5)
p.time_average(weighted=True)
p_ECE = Te(1101014006, include=['GPC', 'GPC2', 'FRCECE'], abscissa='r/a', t_min=1.0,␣
↪t_max=1.5)
p_ECE.time_average(weighted=False)
p.add_profile(p_ECE)
```

This example uses the *add_profile()* method to merge the data from *p_ECE* into *p*.

Multiple time slices

There is considerable overhead associated with loading the data from the tree and performing coordinate conversions. Since time averaging mutates the *BivariatePlasmaProfile* instance in place, it is necessary to keep a copy of the master profile with all of the data. This is accomplished using `copy.deepcopy()`:

```
p_master = ne(1101014006, include=['CTS', 'ETS'], abscissa='r/a')
windows = [(1.0, 1.1), (1.1, 1.2)]
for w in windows:
    p = copy.deepcopy(p_master)
    p.remove_points((p.X[:, 0] < w[0]) | (p.X[:, 0] > w[1]))
    p.time_average(weighted=True)
    p.find_gp_MAP_estimate()
    mean, std = p.smooth(roa)
```

Unless the plasma is changing rapidly you can probably save some time by setting the optimal hyperparameters from one time slice as the initial guess for the next time slice and setting *random_starts* to zero.

profiletools package

Submodules

profiletools.CMod module

Provides classes for working with Alcator C-Mod data via MDSplus.

class `profiletools.CMod.`**`BivariatePlasmaProfile`**(*X_dim=1*, *X_units=None*, *y_units=''*, *X_labels=None*, *y_label=''*, *weightable=True*)

Bases: `profiletools.core.Profile`

Class to represent bivariate (y=f(t, psi)) plasma data.

The first column of *X* is always time. If the abscissa is 'RZ', then the second column is *R* and the third is *Z*. Otherwise the second column is the desired abscissa (psinorm, etc.).

`remake_efit_tree`()

Remake the EFIT tree.

This is needed since EFIT tree instances aren't pickleable yet, so to store a `BivariatePlasmaProfile` in a pickle file, you must delete the EFIT tree.

`convert_abscissa`(*new_abscissa*, *drop_nan=True*, *ddof=1*)

Convert the internal representation of the abscissa to new coordinates.

The target abcissae are what are supported by *rho2rho* from the *eqtools* package. Namely,

psinorm	Normalized poloidal flux
phinorm	Normalized toroidal flux
volnorm	Normalized volume
Rmid	Midplane major radius
r/a	Normalized minor radius

Additionally, each valid option may be prepended with 'sqrt' to return the square root of the desired normalized unit.

Parameters **new_abscissa** : str

The new abscissa to convert to. Valid options are defined above.

drop_nan : bool, optional

Set this to True to drop any elements whose value is NaN following the conversion. Default is True (drop NaN elements).

ddof : int, optional

Degree of freedom correction to use when time-averaging a conversion.

`time_average`(***kwargs*)

Compute the time average of the quantity.

Stores the original bounds of *t* to *self.t_min* and *self.t_max*.

All parameters are passed to `average_data()`.

`drop_axis`(*axis*)

Drops a selected axis from *X*.

Parameters **axis** : int

The index of the axis to drop.

keep_times(*times*, ***kwargs*)

Keeps only the nearest points to vals along the time axis for each channel.

Parameters **times** : array of float

The values the time should be close to.

****kwargs** : optional kwargs

All additional kwargs are passed to `keep_slices()`.

add_profile(*other*)

Absorbs the data from another profile object.

Parameters **other** : `Profile`

`Profile` to absorb.

remove_edge_points(*allow_conversion=True*)

Removes points that are outside the LCFS.

Must be called when the abscissa is a normalized coordinate. Assumes that the last column of *self.X* is space: so it will do the wrong thing if you have already taken an average along space.

Parameters **allow_conversion** : bool, optional

If True and self.abscissa is 'RZ', then the profile will be converted to psinorm and the points will be dropped. Default is True (allow conversion).

constrain_slope_on_axis(*err=0*, *times=None*)

Constrains the slope at the magnetic axis of this Profile's Gaussian process to be zero.

Note that this is accomplished approximately for bivariate data by specifying the slope to be zero at the magnetic axis for a number of points in time.

It is assumed that the Gaussian process has already been created with a call to `create_gp()`.

It is required that the abscissa be either Rmid or one of the normalized coordinates.

Parameters **err** : float, optional

The uncertainty to place on the slope constraint. The default is 0 (slope constraint is exact). This could also potentially be an array for bivariate data where you wish to have the uncertainty vary in time.

times : array-like, optional

The times to impose the constraint at. Default is to use the unique time values in *X[:, 0]*.

constrain_at_limiter(*err_y=0.01*, *err_dy=0.1*, *times=None*, *n_pts=4*, *expansion=1.25*)

Constrains the slope and value of this Profile's Gaussian process to be zero at the GH limiter.

The specific value of *X* coordinate to impose this constraint at is determined by finding the point of the GH limiter which has the smallest mapped coordinate.

If the limiter location is not found in the tree, the system will instead use R=0.91m, Z=0.0m as the limiter location. This is a bit outside of where the limiter is, but will act as a conservative approximation for cases where the exact position is not available.

Note that this is accomplished approximately for bivariate data by specifying the slope and value to be zero at the limiter for a number of points in time.

It is assumed that the Gaussian process has already been created with a call to `create_gp()`.

The abscissa cannot be 'Z' or 'RZ'.

Parameters **err_y** : float, optional

The uncertainty to place on the value constraint. The default is 0.01. This could also potentially be an array for bivariate data where you wish to have the uncertainty vary in time.

err_dy : float, optional

The uncertainty to place on the slope constraint. The default is 0.1. This could also potentially be an array for bivariate data where you wish to have the uncertainty vary in time.

times : array-like, optional

The times to impose the constraint at. Default is to use the unique time values in *X[:, 0]*.

n_pts : int, optional

The number of points outside of the limiter to use. It helps to use three or more points outside the plasma to ensure appropriate behavior. The constraint is applied at *n_pts* linearly spaced points between the limiter location (computed as discussed above) and the limiter location times *expansion*. If you set this to one it will only impose the constraint at the limiter. Default is 4.

expansion : float, optional

The factor by which the coordinate of the limiter location is multiplied to get the outer limit of the *n_pts* constraint points. Default is 1.25.

`remove_quadrature_points_outside_of_limiter``()`

Remove any of the quadrature points which lie outside of the limiter.

This is accomplished by setting their weights to zero. When `create_gp()` is called, it will call `GaussianProcess.condense_duplicates()` which will remove any points for which all of the weights are zero.

This only affects the transformed quantities in *self.transformed*.

`get_limiter_locations``()`

Retrieve the location of the GH limiter from the tree.

If the data are not there (they are missing for some old shots), use R=0.91m, Z=0.0m.

`create_gp``(`*constrain_slope_on_axis=True*, *constrain_at_limiter=True*, *axis_constraint_kwargs={}*, *limiter_constraint_kwargs={}*, ***kwargs*`)`

Create a Gaussian process to handle the data.

Calls `create_gp()`, then imposes constraints as requested.

Defaults to using a squared exponential kernel in two dimensions or a Gibbs kernel with tanh warping in one dimension.

Parameters **constrain_slope_on_axis** : bool, optional

If True, a zero slope constraint at the magnetic axis will be imposed after creating the gp. Default is True (constrain slope).

constrain_at_limiter : bool, optional

If True, a zero slope and value constraint at the GH limiter will be imposed after creating the gp. Default is True (constrain at axis).

axis_constraint_kwargs : dict, optional

The contents of this dictionary are passed as kwargs to `constrain_slope_on_axis()`.

limiter_constraint_kwargs : dict, optional

The contents of this dictionary are passed as kwargs to `constrain_at_limiter()`.

****kwargs** : optional kwargs

All remaining kwargs are passed to `Profile.create_gp()`.

`compute_a_over_L`(*X*, *force_update=False*, *plot=False*, *gp_kwargs={}*, *MAP_kwargs={}*, *plot_kwargs={}*, *return_prediction=False*, *special_vals=0*, *special_X_vals=0*, *compute_2=False*, ***predict_kwargs*)

Compute the normalized inverse gradient scale length.

Only works on data that have already been time-averaged at the moment.

Parameters X : array-like

The points to evaluate a/L at.

force_update : bool, optional

If True, a new Gaussian process will be created even if one already exists. Set this if you have added data or constraints since you created the Gaussian process. Default is False (use current Gaussian process if it exists).

plot : bool, optional

If True, a plot of a/L is produced. Default is False (no plot).

gp_kwargs : dict, optional

The entries of this dictionary are passed as kwargs to `create_gp()` if it gets called. Default is {}.

MAP_kwargs : dict, optional

The entries of this dictionary are passed as kwargs to `find_gp_MAP_estimate()` if it gets called. Default is {}.

plot_kwargs : dict, optional

The entries of this dictionary are passed as kwargs to *plot* when plotting the mean of a/L. Default is {}.

return_prediction : bool, optional

If True, the full prediction of the value and gradient are returned in a dictionary. Default is False (just return value and stddev of a/L).

special_vals : int, optional

The number of special return values incorporated into *output_transform* that should be dropped before computing a/L. This is used so that things like volume averages can be efficiently computed at the same time as a/L. Default is 0 (no extra values).

special_X_vals : int, optional

The number of special points included in the abscissa that should not be included in the evaluation of a/L. Default is 0 (no extra values).

compute_2 : bool, optional

If True, the second derivative and some derived quantities will be computed and added to the output structure (if *return_prediction* is True). You should almost always have r/a for your abscissa when using this: the expressions for other coordinate systems are not as well-vetted. Default is False (don't compute second derivative).

****predict_kwargs** : optional parameters

All other parameters are passed to the Gaussian process' `predict()` method.

`compute_volume_average`(*return_std=True*, *grid=None*, *npts=400*, *force_update=False*, *gp_kwargs={}*, *MAP_kwargs={}*, ***predict_kwargs*)

Compute the volume average of the profile.

Right now only supports data that have already been time-averaged.

Parameters **return_std** : bool, optional

If True, the standard deviation of the volume average is computed and returned. Default is True (return mean and stddev of volume average).

grid : array-like, optional

The quadrature points to use when finding the volume average. If these are not provided, a uniform grid over volnorm will be used. Default is None (use uniform volnorm grid).

npts : int, optional

The number of uniformly-spaced volnorm points to use if *grid* is not specified. Default is 400.

force_update : bool, optional

If True, a new Gaussian process will be created even if one already exists. Set this if you have added data or constraints since you created the Gaussian process. Default is False (use current Gaussian process if it exists).

gp_kwargs : dict, optional

The entries of this dictionary are passed as kwargs to `create_gp()` if it gets called. Default is {}.

MAP_kwargs : dict, optional

The entries of this dictionary are passed as kwargs to `find_gp_MAP_estimate()` if it gets called. Default is {}.

****predict_kwargs** : optional parameters

All other parameters are passed to the Gaussian process' `predict()` method.

Returns **mean** : float

The mean of the volume average.

std : float

The standard deviation of the volume average. Only returned if *return_std* is True. Note that this is only sufficient as an error estimate if you separately verify that the integration error is less than this!

`compute_peaking`(*return_std=True*, *grid=None*, *npts=400*, *force_update=False*, *gp_kwargs={}*, *MAP_kwargs={}*, ***predict_kwargs*)

Compute the peaking of the profile.

Right now only supports data that have already been time-averaged.

Uses the definition from Greenwald, et al. (2007): $w(\psi_n = 0.2)/\langle w\rangle$.

Parameters **return_std** : bool, optional

If True, the standard deviation of the volume average is computed and returned. Default is True (return mean and stddev of peaking).

grid : array-like, optional

The quadrature points to use when finding the volume average. If these are not provided, a uniform grid over volnorm will be used. Default is None (use uniform volnorm grid).

npts : int, optional

The number of uniformly-spaced volnorm points to use if *grid* is not specified. Default is 400.

force_update : bool, optional

If True, a new Gaussian process will be created even if one already exists. Set this if you have added data or constraints since you created the Gaussian process. Default is False (use current Gaussian process if it exists).

gp_kwargs : dict, optional

The entries of this dictionary are passed as kwargs to `create_gp()` if it gets called. Default is {}.

MAP_kwargs : dict, optional

The entries of this dictionary are passed as kwargs to `find_gp_MAP_estimate()` if it gets called. Default is {}.

****predict_kwargs** : optional parameters

All other parameters are passed to the Gaussian process' `predict()` method.

`profiletools.CMod.`**`neCTS`**(*shot*, *abscissa='RZ'*, *t_min=None*, *t_max=None*, *electrons=None*, *efit_tree=None*, *remove_edge=False*, *remove_zeros=True*, *Z_shift=0.0*)

Returns a profile representing electron density from the core Thomson scattering system.

Parameters **shot** : int

The shot number to load.

abscissa : str, optional

The abscissa to use for the data. The default is 'RZ'.

t_min : float, optional

The smallest time to include. Default is None (no lower bound).

t_max : float, optional

The largest time to include. Default is None (no upper bound).

electrons : MDSplus.Tree, optional

An MDSplus.Tree object open to the electrons tree of the correct shot. The shot of the given tree is not checked! Default is None (open tree).

efit_tree : eqtools.CModEFITTree, optional

An eqtools.CModEFITTree object open to the correct shot. The shot of the given tree is not checked! Default is None (open tree).

remove_edge : bool, optional

If True, will remove points that are outside the LCFS. It will convert the abscissa to psinorm if necessary. Default is False (keep edge).

remove_zeros: bool, optional

If True, will remove points that are identically zero. Default is True (remove zero points). This was added because clearly bad points aren't always flagged with a sentinel value of errorbar.

Z_shift: float, optional

The shift to apply to the vertical coordinate, sometimes needed to correct EFIT mapping. Default is 0.0.

`profiletools.CMod.`**`neETS`**(*shot*, *abscissa='RZ'*, *t_min=None*, *t_max=None*, *electrons=None*, *efit_tree=None*, *remove_edge=False*, *remove_zeros=True*, *Z_shift=0.0*)

Returns a profile representing electron density from the edge Thomson scattering system.

Parameters **shot** : int

The shot number to load.

abscissa : str, optional

The abscissa to use for the data. The default is 'RZ'.

t_min : float, optional

The smallest time to include. Default is None (no lower bound).

t_max : float, optional

The largest time to include. Default is None (no upper bound).

electrons : MDSplus.Tree, optional

An MDSplus.Tree object open to the electrons tree of the correct shot. The shot of the given tree is not checked! Default is None (open tree).

efit_tree : eqtools.CModEFITTree, optional

An eqtools.CModEFITTree object open to the correct shot. The shot of the given tree is not checked! Default is None (open tree).

remove_edge : bool, optional

If True, will remove points that are outside the LCFS. It will convert the abscissa to psinorm if necessary. Default is False (keep edge).

remove_zeros: bool, optional

If True, will remove points that are identically zero. Default is True (remove zero points). This was added because clearly bad points aren't always flagged with a sentinel value of errorbar.

Z_shift: float, optional

The shift to apply to the vertical coordinate, sometimes needed to correct EFIT mapping. Default is 0.0.

`profiletools.CMod.`**`neTCI`**(*shot*, *abscissa='r/a'*, *t_min=None*, *t_max=None*, *electrons=None*, *efit_tree=None*, *quad_points=20*, *Z_point=-3.0*, *theta=0.7853981633974483*, *thin=1*, *flag_threshold=0.001*, *ds=0.001*)

Returns a profile representing electron density from the two color interferometer system.

Parameters **shot** : int

The shot number to load.

abscissa : str, optional

The abscissa to use for the data. The default is 'r/a'.

t_min : float, optional

The smallest time to include. Default is None (no lower bound).

t_max : float, optional

The largest time to include. Default is None (no upper bound).

electrons : `MDSplus.Tree`, optional

An `MDSplus.Tree` instance open to the electrons tree of the correct shot. The shot of the given tree is not checked! Default is None (open tree).

efit_tree : :py:class'eqtools.CModEFITTree', optional

An `eqtools.CModEFITTree` instance open to the correct shot. The shot of the given tree is not checked! Default is None (open tree).

quad_points : int or array of float, optional

The quadrature points to use. If an int, then *quad_points* linearly- spaced points between 0 and 1.2 will be used. Otherwise, *quad_points* must be a strictly monotonically increasing array of the quadrature points to use.

Z_point : float

Z coordinate of the starting point of the rays (should be well outside the tokamak). Units are meters.

theta : float

Angle of the chords. Units are radians.

thin : int

Amount by which the data are thinned before computing weights and averages. Default is 1 (no thinning).

flag_threshold : float, optional

The threshold below which points are considered bad. Default is 1e-3.

ds : float, optional

The step size TRIPPy uses to form the beam. Default is 1e-3

`profiletools.CMod.`**`neTCI_old`**(*shot*, *abscissa='RZ'*, *t_min=None*, *t_max=None*, *electrons=None*, *efit_tree=None*, *npts=100*, *flag_threshold=0.001*)

Returns a profile representing electron density from the two color interferometer system.

Parameters shot : int

The shot number to load.

abscissa : str, optional

The abscissa to use for the data. The default is 'RZ'.

t_min : float, optional

The smallest time to include. Default is None (no lower bound).

t_max : float, optional

The largest time to include. Default is None (no upper bound).

electrons : MDSplus.Tree, optional

An MDSplus.Tree object open to the electrons tree of the correct shot. The shot of the given tree is not checked! Default is None (open tree).

efit_tree : eqtools.CModEFITTree, optional

An eqtools.CModEFITTree object open to the correct shot. The shot of the given tree is not checked! Default is None (open tree).

npts : int, optional

The number of points to use for the line integral. Default is 20.

flag_threshold : float, optional

The threshold below which points are considered bad. Default is 1e-3.

`profiletools.CMod.`**`neReflect`**(*shot*, *abscissa='Rmid'*, *t_min=None*, *t_max=None*, *electrons=None*, *efit_tree=None*, *remove_edge=False*, *rf=None*)

Returns a profile representing electron density from the LH/SOL reflectometer system.

Parameters **shot** : int

The shot number to load.

abscissa : str, optional

The abscissa to use for the data. The default is 'Rmid'.

t_min : float, optional

The smallest time to include. Default is None (no lower bound).

t_max : float, optional

The largest time to include. Default is None (no upper bound).

electrons : MDSplus.Tree, optional

An MDSplus.Tree object open to the electrons tree of the correct shot. The shot of the given tree is not checked! Default is None (open tree).

efit_tree : eqtools.CModEFITTree, optional

An eqtools.CModEFITTree object open to the correct shot. The shot of the given tree is not checked! Default is None (open tree).

remove_edge : bool, optional

If True, will remove points that are outside the LCFS. It will convert the abscissa to psinorm if necessary. Default is False (keep edge).

rf : MDSplus.Tree, optional

An MDSplus.Tree object open to the RF tree of the correct shot. The shot of the given tree is not checked! Default is None (open tree).

`profiletools.CMod.`**`ne`**(*shot, include=['CTS', 'ETS'], TCI_quad_points=None, TCI_flag_threshold=None, TCI_thin=None, TCI_ds=None, **kwargs*)

Returns a profile representing electron density from both the core and edge Thomson scattering systems.

Parameters **shot** : int

The shot number to load.

include : list of str, optional

The data sources to include. Valid options are:

CTS	Core Thomson scattering
ETS	Edge Thomson scattering
TCI	Two color interferometer
reflect	SOL reflectometer

The default is to include all TS data sources, but not TCI or the reflectometer.

****kwargs**

All remaining parameters are passed to the individual loading methods.

`profiletools.CMod.`**`neTS`**(*shot*, ***kwargs*)

Returns a profile representing electron density from both the core and edge Thomson scattering systems.

`profiletools.CMod.`**`TeCTS`**(*shot*, *abscissa='RZ'*, *t_min=None*, *t_max=None*, *electrons=None*, *efit_tree=None*, *remove_edge=False*, *remove_zeros=True*, *Z_shift=0.0*)

Returns a profile representing electron temperature from the core Thomson scattering system.

Parameters **shot** : int

The shot number to load.

abscissa : str, optional

The abscissa to use for the data. The default is 'RZ'.

t_min : float, optional

The smallest time to include. Default is None (no lower bound).

t_max : float, optional

The largest time to include. Default is None (no upper bound).

electrons : MDSplus.Tree, optional

An MDSplus.Tree object open to the electrons tree of the correct shot. The shot of the given tree is not checked! Default is None (open tree).

efit_tree : eqtools.CModEFITTree, optional

An eqtools.CModEFITTree object open to the correct shot. The shot of the given tree is not checked! Default is None (open tree).

remove_edge : bool, optional

If True, will remove points that are outside the LCFS. It will convert the abscissa to psinorm if necessary. Default is False (keep edge).

remove_zeros: bool, optional

If True, will remove points that are identically zero. Default is True (remove zero points). This was added because clearly bad points aren't always flagged with a sentinel value of errorbar.

Z_shift: float, optional

The shift to apply to the vertical coordinate, sometimes needed to correct EFIT mapping. Default is 0.0.

`profiletools.CMod.`**`TeETS`**(*shot*, *abscissa='RZ'*, *t_min=None*, *t_max=None*, *electrons=None*, *efit_tree=None*, *remove_edge=False*, *remove_zeros=False*, *Z_shift=0.0*)

Returns a profile representing electron temperature from the edge Thomson scattering system.

Parameters **shot** : int

The shot number to load.

abscissa : str, optional

The abscissa to use for the data. The default is 'RZ'.

t_min : float, optional

The smallest time to include. Default is None (no lower bound).

t_max : float, optional

The largest time to include. Default is None (no upper bound).

electrons : MDSplus.Tree, optional

An MDSplus.Tree object open to the electrons tree of the correct shot. The shot of the given tree is not checked! Default is None (open tree).

efit_tree : eqtools.CModEFITTree, optional

An eqtools.CModEFITTree object open to the correct shot. The shot of the given tree is not checked! Default is None (open tree).

remove_edge : bool, optional

If True, will remove points that are outside the LCFS. It will convert the abscissa to psinorm if necessary. Default is False (keep edge).

remove_zeros: bool, optional

If True, will remove points that are identically zero. Default is False (keep zero points). This was added because clearly bad points aren't always flagged with a sentinel value of errorbar.

Z_shift: float, optional

The shift to apply to the vertical coordinate, sometimes needed to correct EFIT mapping. Default is 0.0.

`profiletools.CMod.`**`TeFRCECE`**(*shot*, *rate='s'*, *cutoff=0.15*, *abscissa='Rmid'*, *t_min=None*, *t_max=None*, *electrons=None*, *efit_tree=None*, *remove_edge=False*)

Returns a profile representing electron temperature from the FRCECE system.

Parameters **shot** : int

The shot number to load.

rate : {'s', 'f'}, optional

Which timebase to use – the fast or slow data. Default is 's' (slow).

cutoff : float, optional

The cutoff value for eliminating cut-off points. All points with values less than this will be discarded. Default is 0.15.

abscissa : str, optional

The abscissa to use for the data. The default is 'Rmid'.

t_min : float, optional

The smallest time to include. Default is None (no lower bound).

t_max : float, optional

The largest time to include. Default is None (no upper bound).

electrons : MDSplus.Tree, optional

An MDSplus.Tree object open to the electrons tree of the correct shot. The shot of the given tree is not checked! Default is None (open tree).

efit_tree : eqtools.CModEFITTree, optional

An eqtools.CModEFITTree object open to the correct shot. The shot of the given tree is not checked! Default is None (open tree).

remove_edge : bool, optional

If True, will remove points that are outside the LCFS. It will convert the abscissa to psinorm if necessary. Default is False (keep edge).

`profiletools.CMod.`**`TeGPC2`**(*shot*, *abscissa='Rmid'*, *t_min=None*, *t_max=None*, *electrons=None*, *efit_tree=None*, *remove_edge=False*)

Returns a profile representing electron temperature from the GPC2 system.

Parameters **shot** : int

The shot number to load.

abscissa : str, optional

The abscissa to use for the data. The default is 'Rmid'.

t_min : float, optional

The smallest time to include. Default is None (no lower bound).

t_max : float, optional

The largest time to include. Default is None (no upper bound).

electrons : MDSplus.Tree, optional

An MDSplus.Tree object open to the electrons tree of the correct shot. The shot of the given tree is not checked! Default is None (open tree).

efit_tree : eqtools.CModEFITTree, optional

An eqtools.CModEFITTree object open to the correct shot. The shot of the given tree is not checked! Default is None (open tree).

remove_edge : bool, optional

If True, will remove points that are outside the LCFS. It will convert the abscissa to psinorm if necessary. Default is False (keep edge).

`profiletools.CMod.`**`TeGPC`**(*shot*, *cutoff=0.15*, *abscissa='Rmid'*, *t_min=None*, *t_max=None*, *electrons=None*, *efit_tree=None*, *remove_edge=False*)

Returns a profile representing electron temperature from the GPC system.

Parameters **shot** : int

The shot number to load.

cutoff : float, optional

The cutoff value for eliminating cut-off points. All points with values less than this will be discarded. Default is 0.15.

abscissa : str, optional

The abscissa to use for the data. The default is 'Rmid'.

t_min : float, optional

The smallest time to include. Default is None (no lower bound).

t_max : float, optional

The largest time to include. Default is None (no upper bound).

electrons : MDSplus.Tree, optional

An MDSplus.Tree object open to the electrons tree of the correct shot. The shot of the given tree is not checked! Default is None (open tree).

efit_tree : eqtools.CModEFITTree, optional

An eqtools.CModEFITTree object open to the correct shot. The shot of the given tree is not checked! Default is None (open tree).

remove_edge : bool, optional

If True, will remove points that are outside the LCFS. It will convert the abscissa to psinorm if necessary. Default is False (keep edge).

`profiletools.CMod.`**`TeMic`**(*shot*, *cutoff=0.15*, *abscissa='Rmid'*, *t_min=None*, *t_max=None*, *electrons=None*, *efit_tree=None*, *remove_edge=False*, *remove_zeros=True*)

Returns a profile representing electron temperature from the Michelson interferometer.

Parameters **shot** : int

The shot number to load.

abscissa : str, optional

The abscissa to use for the data. The default is 'Rmid'.

t_min : float, optional

The smallest time to include. Default is None (no lower bound).

t_max : float, optional

The largest time to include. Default is None (no upper bound).

electrons : MDSplus.Tree, optional

An MDSplus.Tree object open to the electrons tree of the correct shot. The shot of the given tree is not checked! Default is None (open tree).

efit_tree : eqtools.CModEFITTree, optional

An eqtools.CModEFITTree object open to the correct shot. The shot of the given tree is not checked! Default is None (open tree).

remove_edge : bool, optional

If True, will remove points that are outside the LCFS. It will convert the abscissa to psinorm if necessary. Default is False (keep edge).

`profiletools.CMod.`**`Te`**(*shot, include=['CTS', 'ETS', 'FRCECE', 'GPC2', 'GPC', 'Mic'], FRCECE_rate='s', FRCECE_cutoff=0.15, GPC_cutoff=0.15, remove_ECE_edge=True, **kwargs*)

Returns a profile representing electron temperature from the Thomson scattering and ECE systems.

Parameters **shot** : int

The shot number to load.

include : list of str, optional

The data sources to include. Valid options are:

CTS	Core Thomson scattering
ETS	Edge Thomson scattering
FRCECE	FRC electron cyclotron emission
GPC	Grating polychromator
GPC2	Grating polychromator 2

The default is to include all data sources.

FRCECE_rate : {'s', 'f'}, optional

Which timebase to use for FRCECE – the fast or slow data. Default is 's' (slow).

FRCECE_cutoff : float, optional

The cutoff value for eliminating cut-off points from FRCECE. All points with values less than this will be discarded. Default is 0.15.

GPC_cutoff : float, optional

The cutoff value for eliminating cut-off points from GPC. All points with values less than this will be discarded. Default is 0.15.

remove_ECE_edge : bool, optional

If True, the points outside of the LCFS for the ECE diagnostics will be removed. Note that this overrides remove_edge, if present, in kwargs. Furthermore, this may lead to abscissa being converted to psinorm if an incompatible option was used.

****kwargs**

All remaining parameters are passed to the individual loading methods.

`profiletools.CMod.`**`TeTS`**`(`*shot*, ***kwargs*`)`

Returns a profile representing electron temperature data from the Thomson scattering system.

Includes both core and edge system.

`profiletools.CMod.`**`emissAX`**`(`*shot*, *system*, *abscissa='Rmid'*, *t_min=None*, *t_max=None*, *tree=None*, *efit_tree=None*, *remove_edge=False*`)`

Returns a profile representing emissivity from the AXA system.

Parameters **shot** : int

The shot number to load.

system : {AXA, AXJ}

The system to use.

abscissa : str, optional

The abscissa to use for the data. The default is 'Rmid'.

t_min : float, optional

The smallest time to include. Default is None (no lower bound).

t_max : float, optional

The largest time to include. Default is None (no upper bound).

tree : MDSplus.Tree, optional

An MDSplus.Tree object open to the cmod tree of the correct shot. The shot of the given tree is not checked! Default is None (open tree).

efit_tree : eqtools.CModEFITTree, optional

An eqtools.CModEFITTree object open to the correct shot. The shot of the given tree is not checked! Default is None (open tree).

remove_edge : bool, optional

If True, will remove points that are outside the LCFS. It will convert the abscissa to psinorm if necessary. Default is False (keep edge).

`profiletools.CMod.`**`emiss`**(*shot, include=['AXA', 'AXJ'], **kwargs*)

Returns a profile representing emissivity.

Parameters shot : int

The shot number to load.

include : list of str, optional

The data sources to include. Valid options are: {AXA, AXJ}. The default is to include both data sources.

****kwargs**

All remaining parameters are passed to the individual loading methods.

`profiletools.CMod.`**`read_plasma_csv`**(**args*, ***kwargs*)

Returns a profile containing the data from a CSV file.

If your data are bivariate, you must ensure that time ends up being the first column, either by putting it first in your CSV file or by specifying its name first in *X_names*.

The CSV file can contain metadata lines of the form “name data” or “name data,data,...”. The following metadata are automatically parsed into the correct fields:

shot	shot number
times	comma-separated list of times included in the data
t_min	minimum time included in the data
t_max	maximum time included in the data
coordinate	the abscissa the data are represented as a function of

If you don’t provide *coordinate* in the metadata, the program will try to use the last entry in X_labels to infer the abscissa. If this fails, it will simply set the abscissa to the title of the last entry in X_labels. If you provide your data as a function of R, Z it will look for the last two entries in X_labels to be R and Z once surrounding dollar signs and spaces are removed.

Parameters are the same as `read_csv()`.

`profiletools.CMod.`**`read_plasma_NetCDF`**(**args*, ***kwargs*)

Returns a profile containing the data from a NetCDF file.

The file can contain metadata attributes specified in the *metadata* kwarg. The following metadata are automatically parsed into the correct fields:

shot	shot number
times	comma-separated list of times included in the data
t_min	minimum time included in the data
t_max	maximum time included in the data
coordinate	the abscissa the data are represented as a function of

If you don’t provide *coordinate* in the metadata, the program will try to use the last entry in X_labels to infer the abscissa. If this fails, it will simply set the abscissa to the title of the last entry in X_labels. If you provide your data as a function of R, Z it will look for the last two entries in X_labels to be R and Z once surrounding dollar signs and spaces are removed.

Parameters are the same as `read_NetCDF()`.

profiletools.core module

Provides the base `Profile` class and other utilities.

`profiletools.core.`**`average_points`**(*X*, *y*, *err_X*, *err_y*, *T=None*, *ddof=1*, *robust=False*, *y_method='sample'*, *X_method='sample'*, *weighted=False*)

Find the average of the points with the given uncertainties using a variety of techniques.

Parameters **X** : array, (*M*, *D*) or (*M*, *N*, *D*)

Abscissa values to average.

y : array, (*M*)

Data values to average.

err_X : array, same shape as *X*

Uncertainty in *X*.

err_y : array, same shape as *y*

Uncertainty in *y*.

T : array, (*M*, *N*), optional

Transform for *y*. Default is None (*y* is not transformed).

ddof : int, optional

The degree of freedom correction used in computing the standard deviation. The default is 1, the standard Bessel correction to give an unbiased estimate of the variance.

robust : bool, optional

Set this flag to use robust estimators (median, IQR). Default is False.

y_method : {'sample', 'RMS', 'total', 'of mean', 'of mean sample'}, optional

The method to use in computing the uncertainty in the averaged *y*.

- 'sample' computes the sample standard deviation.
- 'RMS' computes the root-mean-square of the individual error bars.
- 'total' computes the square root of the sum of the sample variance and the mean variance. This is only statistically reasonable if the points represent sample means/variances already.
- 'of mean' computes the uncertainty in the mean using error propagation with the given uncertainties.
- 'of mean sample' computes the uncertainty in the mean using error propagation with the sample variance. Should not be used with weighted estimators!

Default is 'sample' (use sample variance).

X_method : {'sample', 'RMS', 'total', 'of mean', 'of mean sample'}, optional

The method to use in computing the uncertainty in the averaged *X*. Options are the same as *y_method*. Default is 'sample' (use sample variance).

weighted : bool, optional

Set this flag to use weighted estimators. The weights are 1/err_y^2. Default is False (use unweighted estimators).

Returns **mean_X** : array, (*D*,) or (*N*, *D*)

Mean of abscissa values.

mean_y : float

Mean of data values.

err_X : array, same shape as *mean_X*

Uncertainty in abscissa values.

err_y : float

Uncertainty in data values.

T : array, (*N*,) or None

Mean of transformation.

class `profiletools.core.`**`Channel`**(*X*, *y*, *err_X=0*, *err_y=0*, *T=None*, *y_label=''*, *y_units=''*)

Bases: `object`

Class to store data from a single channel.

This is particularly useful for storing linearly transformed data, but should work for general data just as well.

Parameters **X** : array, (*M*, *N*, *D*)

Abscissa values to use.

y : array, (*M*,)

Data values.

err_X : array, same shape as *X*

Uncertainty in *X*.

err_y : array, (*M*,)

Uncertainty in data.

T : array, (*M*, *N*), optional

Linear transform to get from latent variables to data in *y*. Default is that *y* represents untransformed data.

y_label : str, optional

Label for the *y* data. Default is empty string.

y_units : str, optional

Units of the *y* data. Default is empty string.

`keep_slices`(*axis*, *vals*, *tol=None*, *keep_mixed=False*)

Only keep the indices closest to given *vals*.

Parameters **axis** : int

The column in *X* to check values on.

vals : float or 1-d array

The value(s) to keep the points that are nearest to.

keep_mixed : bool, optional

Set this flag to keep transformed quantities that depend on multiple values of *X[:, :, axis]*. Default is False (drop mixed quantities).

Returns **still_good** : bool

Returns True if there are still any points left in the channel, False otherwise.

average_data(*axis=0*, ***kwargs*)

Average the data along the given *axis*.

Parameters **axis** : int, optional

Axis to average along. Default is 0.

****kwargs** : optional keyword arguments

All additional kwargs are passed to `average_points()`.

remove_points(*conditional*)

Remove points satisfying *conditional*.

Parameters **conditional** : array, same shape as *self.y*

Boolean array with True wherever a point should be removed.

Returns **bad_X** : array

The removed *X* values.

bad_err_X : array

The uncertainty in the removed *X* values.

bad_y : array

The removed *y* values.

bad_err_y : array

The uncertainty in the removed *y* values.

bad_T : array

The transformation matrix of the removed *y* values.

class `profiletools.core.`**Profile**(*X_dim=1*, *X_units=None*, *y_units=''*, *X_labels=None*, *y_label=''*, *weightable=True*)

Bases: `object`

Object to abstractly represent a profile.

Parameters **X_dim** : positive int, optional

Number of dimensions of the independent variable. Default value is 1.

X_units : str, list of str or None, optional

Units for each of the independent variables. If *X_dim'=1, this should given as a single string, if 'X_dim*>1, this should be given as a list of strings of length *X_dim*. Default value is *None*, meaning a list of empty strings will be used.

y_units : str, optional

Units for the dependent variable. Default is an empty string.

X_labels : str, list of str or None, optional

Descriptive label for each of the independent variables. If *X_dim'=1, this should be given as a single string, if 'X_dim*>1, this should be given as a list of strings of length *X_dim*. Default value is *None*, meaning a list of empty strings will be used.

y_label : str, optional

Descriptive label for the dependent variable. Default is an empty string.

weightable : bool, optional

Whether or not it is valid to use weighted estimators on the data, or if the error bars are too suspect for this to be valid. Default is True (allow use of weighted estimators).

Attributes

y	(`Array`, (*M*,)) The *M* dependent variables.
X	(`Matrix`, (*M*, *X_dim*)) The *M* independent variables.
err_y	(`Array`, (*M*,)) The uncertainty in the *M* dependent variables.
err_X	(`Matrix`, (*M*, *X_dim*)) The uncertainties in each dimension of the *M* independent variables.
channels	(`Matrix`, (*M*, *X_dim*)) The logical groups of points into channels along each of the independent variables.
X_dim	(positive int) The number of dimensions of the independent variable.
X_units	(list of str, (X_dim,)) The units for each of the independent variables.
y_units	(str) The units for the dependent variable.
X_labels	(list of str, (X_dim,)) Descriptive labels for each of the independent variables.
y_label	(str) Descriptive label for the dependent variable.
weightable	(bool) Whether or not weighted estimators can be used.
trans-formed	(list of `Channel`) The transformed quantities associated with the `Profile` instance.
gp	(`gptools.GaussianProcess` instance) The Gaussian process with the local and transformed data included.

add_data (*X*, *y*, *err_X=0*, *err_y=0*, *channels=None*)

Add data to the training data set of the `Profile` instance.

Will also update the Profile's Gaussian process instance (if it exists).

Parameters X : array-like, (*M*, *N*)

M independent variables of dimension *N*.

y : array-like, (*M*,)

M dependent variables.

err_X : array-like, (*M*, *N*), or scalar float, or single array-like (*N*,), optional

Non-negative values only. Error given as standard deviation for each of the *N* dimensions in the *M* independent variables. If a scalar is given, it is used for all of the values. If a single array of length *N* is given, it is used for each point. The default is to assign zero error to each point.

err_y : array-like (*M*,) or scalar float, optional

Non-negative values only. Error given as standard deviation in the *M* dependent variables. If *err_y* is a scalar, the data set is taken to be homoscedastic (constant error). Otherwise, the length of *err_y* must equal the length of *y*. Default value is 0 (noiseless observations).

channels : dict or array-like (*M*, *N*)

Keys to logically group points into "channels" along each dimension of *X*. If not passed, channels are based simply on which points have equal values in *X*. If only certain dimensions have groupings other than the simple default equality conditions, then you can pass a dict with integer keys in the interval [0, *X_dim*-1] whose values are the arrays of length *M* indicating the channels. Otherwise, you can pass in a full (*M*, *N*) array.

Raises ValueError

Bad shapes for any of the inputs, negative values for *err_y* or *n*.

`add_profile`(*other*)

Absorbs the data from one profile object.

Parameters other : `Profile`

`Profile` to absorb.

`drop_axis`(*axis*)

Drops a selected axis from *X*.

Parameters axis : int

The index of the axis to drop.

`keep_slices`(*axis*, *vals*, *tol=None*, ***kwargs*)

Keeps only the nearest points to vals along the given axis for each channel.

Parameters axis : int

The axis to take the slice(s) of.

vals : array of float

The values the axis should be close to.

tol : float or None

Tolerance on nearest values – if the nearest value is farther than this, it is not kept. If None, this is not applied.

****kwargs** : optional kwargs

All additional kwargs are passed to `keep_slices()`.

`average_data`(*axis=0*, ***kwargs*)

Computes the average of the profile over the desired axis.

If *X_dim* is already 1, this returns the average of the quantity. Otherwise, the `Profile` is mutated to contain the desired averaged data. *err_X* and *err_y* are populated with the standard deviations of the respective quantities. The averaging is carried out within the groupings defined by the *channels* attribute.

Parameters axis : int, optional

The index of the dimension to average over. Default is 0.

****kwargs** : optional kwargs

All additional kwargs are passed to `average_points()`.

`plot_data`(*ax=None*, *label_axes=True*, ***kwargs*)

Plot the data stored in this Profile. Only works for X_dim = 1 or 2.

Parameters ax : axis instance, optional

Axis to plot the result on. If no axis is passed, one is created. If the string 'gca' is passed, the current axis (from plt.gca()) is used. If X_dim = 2, the axis must be 3d.

label_axes : bool, optional

If True, the axes will be labelled with strings constructed from the labels and units set when creating the Profile instance. Default is True (label axes).

****kwargs** : extra plotting arguments, optional

Extra arguments that are passed to errorbar/errorbar3d.

Returns The axis instance used.

`remove_points`(*conditional*)

Remove points where conditional is True.

Note that this does NOT remove anything from the GP – you either need to call `create_gp()` again or act manually on the `gp` attribute.

Also note that this does not include any provision for removing points that represent linearly-transformed quantities – you will need to operate directly on `transformed` to remove such points.

Parameters **conditional** : array-like of bool, (*M*,)

Array of booleans corresponding to each entry in *y*. Where an entry is True, that value will be removed.

Returns **X_bad** : matrix

Input values of the bad points.

y_bad : array

Bad values.

err_X_bad : array

Uncertainties on the abcissa of the bad values.

err_y_bad : array

Uncertainties on the bad values.

`remove_outliers`(*thresh=3*, *check_transformed=False*, *force_update=False*, *mask_only=False*, *gp_kwargs={}*, *MAP_kwargs={}*, ***predict_kwargs*)

Remove outliers from the Gaussian process.

The Gaussian process is created if it does not already exist. The chopping of values assumes that any artificial constraints that have been added to the GP are at the END of the GP's data arrays.

The values removed are returned.

Parameters **thresh** : float, optional

The threshold as a multiplier times *err_y*. Default is 3 (i.e., throw away all 3-sigma points).

check_transformed : bool, optional

Set this flag to check if transformed quantities are outliers. Default is False (don't check transformed quantities).

force_update : bool, optional

If True, a new Gaussian process will be created even if one already exists. Set this if you have added data or constraints since you created the Gaussian process. Default is False (use current Gaussian process if it exists).

mask_only : bool, optional

Set this flag to return only a mask of the non-transformed points that are flagged. Default is False (completely remove bad points). In either case, the bad transformed points will ALWAYS be removed if *check_transformed* is True.

gp_kwargs : dict, optional

The entries of this dictionary are passed as kwargs to *create_gp()* if it gets called. Default is {}.

MAP_kwargs : dict, optional

The entries of this dictionary are passed as kwargs to *find_gp_MAP_estimate()* if it gets called. Default is {}.

****predict_kwargs** : optional parameters

All other parameters are passed to the Gaussian process' `predict()` method.

Returns X_bad : matrix

Input values of the bad points.

y_bad : array

Bad values.

err_X_bad : array

Uncertainties on the abcissa of the bad values.

err_y_bad : array

Uncertainties on the bad values.

transformed_bad : array of *Channel*

Transformed points that were removed.

remove_extreme_changes(*thresh=10*, *logic='and'*, *mask_only=False*)

Removes points at which there is an extreme change.

Only for univariate data!

This operation is performed by looking for points who differ by more than *thresh* * *err_y* from each of their neighbors. This operation will typically only be useful with large values of thresh. This is useful for eliminating bad channels.

Note that this will NOT update the Gaussian process.

Parameters thresh : float, optional

The threshold as a multiplier times *err_y*. Default is 10 (i.e., throw away all 10-sigma changes).

logic : {'and', 'or'}, optional

Whether the logical operation performed should be an and or an or when looking at left-hand and right-hand differences. 'and' is more conservative, but 'or' will help if you have multiple bad channels in a row. Default is 'and' (point must have a drastic change in both directions to be rejected).

mask_only : bool, optional

If True, only the boolean mask indicated where the bad points are will be removed, and it is up to the user to remove them. Default is False (actually remove the bad points).

create_gp(*k=None*, *noise_k=None*, *upper_factor=5*, *lower_factor=5*, *x0_bounds=None*, *mask=None*, *k_kwargs={}*, ***kwargs*)

Create a Gaussian process to handle the data.

Parameters k : `Kernel` instance, optional

Covariance kernel (from `gptools`) with the appropriate number of dimensions, or None. If None, a squared exponential kernel is used. Can also be a string from the following table:

SE	Squared exponential
gibbstanh	Gibbs kernel with tanh warping
RQ	Rational quadratic
SEsym1d	1d SE with symmetry constraint

The bounds for each hyperparameter are selected as follows:

sigma_f	[1/lower_factor, upper_factor]*range(y)
l1	[1/lower_factor, upper_factor]*range(X[:, 1])
...	And so on for each length scale

Here, eps is sys.float_info.epsilon. The initial guesses for each parameter are set to be halfway between the upper and lower bounds. For the Gibbs kernel, the uniform prior for sigma_f is used, but gamma priors are used for the remaining hyperparameters. Default is None (use SE kernel).

noise_k : `Kernel` instance, optional

The noise covariance kernel. Default is None (use the default zero noise kernel, with all noise being specified by *err_y*).

upper_factor : float, optional

Factor by which the range of the data is multiplied for the upper bounds on both length scales and signal variances. Default is 5, which seems to work pretty well for C-Mod data.

lower_factor : float, optional

Factor by which the range of the data is divided for the lower bounds on both length scales and signal variances. Default is 5, which seems to work pretty well for C-Mod data.

x0_bounds : 2-tuple, optional

Bounds to use on the x0 (transition location) hyperparameter of the Gibbs covariance function with tanh warping. This is the hyperparameter that tends to need the most tuning on C-Mod data. Default is None (use range of X).

mask : array of bool, optional

Boolean mask of values to actually include in the GP. Default is to include all values.

k_kwargs : dict, optional

All entries are passed as kwargs to the constructor for the kernel if a kernel instance is not provided.

****kwargs** : optional kwargs

All additional kwargs are passed to the constructor of `gptools.GaussianProcess`.

`find_gp_MAP_estimate`(*force_update=False*, *gp_kwargs={}*, ***kwargs*)

Find the MAP estimate for the hyperparameters of the Profile's Gaussian process.

If this `Profile` instance does not already have a Gaussian process, it will be created. Note that the user is responsible for manually updating the Gaussian process if more data are added or the `Profile` is otherwise mutated. This can be accomplished directly using the *force_update* keyword.

Parameters **force_update** : bool, optional

If True, a new Gaussian process will be created even if one already exists. Set this if you have added data or constraints since you created the Gaussian process. Default is False (use current Gaussian process if it exists).

gp_kwargs : dict, optional

The entries of this dictionary are passed as kwargs to `create_gp()` if it gets called. Default is {}.

****kwargs** : optional parameters

All other parameters are passed to the Gaussian process' `optimize_hyperparameters()` method.

`plot_gp` (*force_update=False*, *gp_kwargs={}*, *MAP_kwargs={}*, ***kwargs*)
Plot the current state of the Profile's Gaussian process.

If this `Profile` instance does not already have a Gaussian process, it will be created. Note that the user is responsible for manually updating the Gaussian process if more data are added or the `Profile` is otherwise mutated. This can be accomplished directly using the *force_update* keyword.

Parameters **force_update** : bool, optional

If True, a new Gaussian process will be created even if one already exists. Set this if you have added data or constraints since you created the Gaussian process. Default is False (use current Gaussian process if it exists).

gp_kwargs : dict, optional

The entries of this dictionary are passed as kwargs to `create_gp()` if it gets called. Default is {}.

MAP_kwargs : dict, optional

The entries of this dictionary are passed as kwargs to `find_gp_MAP_estimate()` if it gets called. Default is {}.

****kwargs** : optional parameters

All other parameters are passed to the Gaussian process' `plot()` method.

`smooth` (*X*, *n=0*, *force_update=False*, *plot=False*, *gp_kwargs={}*, *MAP_kwargs={}*, ***kwargs*)
Evaluate the underlying smooth curve at a given set of points using Gaussian process regression.

If this `Profile` instance does not already have a Gaussian process, it will be created. Note that the user is responsible for manually updating the Gaussian process if more data are added or the `Profile` is otherwise mutated. This can be accomplished directly using the *force_update* keyword.

Parameters **X** : array-like (*N*, *X_dim*)

Points to evaluate smooth curve at.

n : non-negative int, optional

The order of derivative to evaluate at. Default is 0 (return value). See the documentation on `gptools.GaussianProcess.predict()`.

force_update : bool, optional

If True, a new Gaussian process will be created even if one already exists. Set this if you have added data or constraints since you created the Gaussian process. Default is False (use current Gaussian process if it exists).

plot : bool, optional

If True, `gptools.GaussianProcess.plot()` is called to produce a plot of the smoothed curve. Otherwise, `gptools.GaussianProcess.predict()` is called directly.

gp_kwargs : dict, optional

The entries of this dictionary are passed as kwargs to `create_gp()` if it gets called. Default is {}.

MAP_kwargs : dict, optional

The entries of this dictionary are passed as kwargs to `find_gp_MAP_estimate()` if it gets called. Default is {}.

****kwargs** : optional parameters

All other parameters are passed to the Gaussian process' `plot()` or `predict()` method according to the state of the *plot* keyword.

Returns **ax** : axis instance

The axis instance used. This is only returned if the *plot* keyword is True.

mean : `Array`, (*M*,)

Predicted GP mean. Only returned if *full_output* is False.

std : `Array`, (*M*,)

Predicted standard deviation, only returned if *return_std* is True and *full_output* is False.

full_output : dict

Dictionary with fields for mean, std, cov and possibly random samples. Only returned if *full_output* is True.

write_csv(*filename*)

Writes this profile to a CSV file.

Parameters **filename** : str

Path of the file to write. If the file exists, it will be overwritten without warning.

`profiletools.core.`**read_csv**(*filename*, *X_names=None*, *y_name=None*, *metadata_lines=None*)

Reads a CSV file into a `Profile`.

If names are not provided for the columns holding the *X* and *y* values and errors, the names are found automatically by looking at the header row, and are used in the order found, with the last column being *y*. Otherwise, the columns will be read in the order specified. The column names should be of the form "name [units]", which will be automatically parsed to populate the `Profile`. In either case, there can be a corresponding column "err_name [units]" which holds the 1-sigma uncertainty in that quantity. There can be an arbitrary number of lines of metadata at the beginning of the file which are read into the `metadata` attribute of the `Profile` created. This is most useful when using `BivariatePlasmaProfile` as you can store the shot and time window.

Parameters **X_names** : list of str, optional

Ordered list of the column names containing the independent variables. The default behavior is to infer the names and ordering from the header of the CSV file. See the discussion above. Note that if you provide *X_names* you must also provide *y_name*.

y_name : str, optional

Name of the column containing the dependent variable. The default behavior is to infer this name from the header of the CSV file. See the discussion above. Note that if you provide *y_name* you must also provide *X_names*.

metadata_lines : non-negative int, optional

Number of lines of metadata to read from the beginning of the file. These are read into the `metadata` attribute of the profile created.

`profiletools.core.`**`read_NetCDF`**(*filename*, *X_names*, *y_name*, *metadata=[]*)

Reads a NetCDF file into a `Profile`.

The file must contain arrays of equal length for each of the independent and the dependent variable. The units of each variable can either be specified as the units attribute on the variable, or the variable name can be of the form "name [units]", which will be automatically parsed to populate the `Profile`. For each independent and the dependent variable there can be a corresponding column "err_name" or "err_name [units]" which holds the 1-sigma uncertainty in that quantity. There can be an arbitrary number of metadata attributes in the file which are read into the corresponding attributes of the `Profile` created. This is most useful when using `BivariatePlasmaProfile` as you can store the shot and time window. Be careful that you do not overwrite attributes needed by the class, however!

Parameters **X_names** : list of str

Ordered list of the column names containing the independent variables. See the discussion above regarding name conventions.

y_name : str

Name of the column containing the dependent variable. See the discussion above regarding name conventions.

metadata : list of str, optional

List of attribute names to read into the corresponding attributes of the `Profile` created.

`profiletools.core.`**`parse_column_name`**(*name*)

Parse a column header *name* into label and units.

`profiletools.core.`**`errorbar3d`**(*ax*, *x*, *y*, *z*, *xerr=None*, *yerr=None*, *zerr=None*, ***kwargs*)

Draws errorbar plot of z(x, y) with errorbars on all variables.

Parameters **ax** : 3d axis instance

The axis to draw the plot on.

x : array, (*M*,)

x-values of data.

y : array, (*M*,)

y-values of data.

z : array, (*M*,)

z-values of data.

xerr : array, (*M*,), optional

Errors in x-values. Default value is 0.

yerr : array, (*M*,), optional

Errors in y-values. Default value is 0.

zerr : array, (M,), optional

Errors in z-values. Default value is 0.

****kwargs** : optional

Extra arguments are passed to the plot command used to draw the datapoints.

`profiletools.core.`**`unique_rows`**(*arr*)

Returns a copy of arr with duplicate rows removed.

From Stackoverflow "Find unique rows in numpy.array."

Parameters **arr** : `Array`, (m, n). The array to find the unique rows of.

Returns **unique** : `Array`, (p, n) where $p <= m$

The array *arr* with duplicate rows removed.

`profiletools.core.`**`get_nearest_idx`**(*v*, *a*)

Returns the array of indices of the nearest value in *a* corresponding to each value in *v*.

Parameters **v** : Array

Input values to match to nearest neighbors in *a*.

a : Array

Given values to match against.

Returns Indices in *a* of the nearest values to each value in *v*. Has the same shape as *v*.

class `profiletools.core.`**`RejectionFunc`**(*mask*, *positivity=True*, *monotonicity=True*)

Bases: `object`

Rejection function for use with *full_MC* mode of `GaussianProcess.predict()`.

Parameters **mask** : array of bool

Mask for the values to include in the test.

positivity : bool, optional

Set this to True to impose a positivity constraint on the sample. Default is True.

monotonicity : bool, optional

Set this to True to impose a positivity constraint on the samples. Default is True.

`__call__`(*samp*)

Returns True if the sample meets the constraints, False otherwise.

`profiletools.core.`**`leading_axis_product`**(*w*, *x*)

Perform a product along the leading axis, as is needed when applying weights.

`profiletools.core.`**`meanw`**(*x*, *weights=None*, *axis=None*)

Weighted mean of data.

Defined as

$$\mu = \frac{\sum_i w_i x_i}{\sum_i w_i}$$

Parameters **x** : array-like

The vector to find the mean of.

weights : array-like, optional

The weights. Must be broadcastable with *x*. Default is to use the unweighted mean.

axis : int, optional

The axis to take the mean along. Default is to use the whole data set.

`profiletools.core.`**`varw`**(*x*, *weights=None*, *axis=None*, *ddof=1*, *mean=None*)

Weighted variance of data.

Defined (for *ddof* = 1) as

$$s^2 = \frac{\sum_i w_i}{(\sum_i w_i)^2 - \sum_i w_i^2} \sum_i w_i (x_i - \mu)^2$$

Parameters **x** : array-like

The vector to find the mean of.

weights : array-like, optional

The weights. Must be broadcastable with *x*. Default is to use the unweighted mean.

axis : int, optional

The axis to take the mean along. Default is to use the whole data set.

ddof : int, optional

The degree of freedom correction to use. If no weights are given, this is the standard Bessel correction. If weights are given, this uses an approximate form based on the assumption that the weights are inverse variances for each data point. In this case, the value has no effect other than being True or False. Default is 1 (apply correction assuming normal noise dictated weights).

mean : array-like, optional

The weighted mean to use. If you have already computed the weighted mean with `meanw()`, you can pass the result in here to save time.

`profiletools.core.`**`stdw`**(**args*, ***kwargs*)

Weighted standard deviation of data.

Defined (for *ddof* = 1) as

$$s = \sqrt{\frac{\sum_i w_i}{(\sum_i w_i)^2 - \sum_i w_i^2} \sum_i w_i (x_i - \mu)^2}$$

Parameters **x** : array-like

The vector to find the mean of.

weights : array-like, optional

The weights. Must be broadcastable with *x*. Default is to use the unweighted mean.

axis : int, optional

The axis to take the mean along. Default is to use the whole data set.

ddof : int, optional

The degree of freedom correction to use. If no weights are given, this is the standard Bessel correction. If weights are given, this uses an approximate form based on the assumption that the weights are inverse variances for each data point. In this case, the value has no effect other than being True or False. Default is 1 (apply correction assuming normal noise dictated weights).

mean : array-like, optional

The weighted mean to use. If you have already computed the weighted mean with `meanw()`, you can pass the result in here to save time.

`profiletools.core.`**`robust_std`**(*y*, *axis=None*)

Computes the robust standard deviation of the given data.

This is defined as $IQR/(2\Phi^{-1}(0.75))$, where IQR is the interquartile range and Φ is the inverse CDF of the standard normal. This is an approximation based on the assumption that the data are Gaussian, and will have the effect of diminishing the effect of outliers.

Parameters **y** : array-like

The data to find the robust standard deviation of.

axis : int, optional

The axis to find the standard deviation along. Default is None (find from whole data set).

`profiletools.core.`**`scoreatpercentilew`**(*x*, *p*, *weights*)

Computes the weighted score at the given percentile.

Does not work on small data sets!

Parameters **x** : array

Array of data to apply to. Only works properly on 1d data!

p : float or array of float

Percentile(s) to find.

weights : array, same shape as *x*

The weights to apply to the values in *x*.

`profiletools.core.`**`medianw`**(*x*, *weights=None*, *axis=None*)

Computes the weighted median of the given data.

Does not work on small data sets!

Parameters **x** : array

Array of data to apply to. Only works properly on 1d, 2d and 3d data.

weights : array, optional

Weights to apply to the values in *x*. Default is to use an unweighted estimator.

axis : int, optional

The axis to take the median along. Default is None (apply to flattened array).

`profiletools.core.`**`robust_stdw`**(*x*, *weights=None*, *axis=None*)

Computes the weighted robust standard deviation from the weighted IQR.

Does not work on small data sets!

Parameters **x** : array

Array of data to apply to. Only works properly on 1d, 2d and 3d data.

weights : array, optional

Weights to apply to the values in *x*. Default is to use an unweighted estimator.

axis : int, optional

The axis to take the robust standard deviation along. Default is None (apply to flattened array).

Module contents

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index