kafe Documentation

Release 0.3alpha46

Daniel Savoiu

August 21, 2013

CONTENTS

kafe is a data fitting framework designed for use in undergraduate physics lab courses. It provides a basic *Python* toolkit for fitting and plotting using already available *Python* packages such as *NumPy* and *matplotlib*, as well as *CERN ROOT*'s version of the *Minuit* minimizer.

Contents

CHAPTER

ONE

SUMMARY

The package provides a simple approach to fitting using variance-covariance matrices, thus allowing for error correlations to be taken into account. This implementation's error model assumes the measurement data (dependent variable) is distributed according to a *Gaussian* distribution centered at its "true" value. The spread of the distribution is given as a (1σ)-error.

An "errors-in-variables" model is also implemented to take uncertainties in the independent variable (x errors) into account. This is done by specifying/constructing a separate variance-covariance matrix for the x axis and "projecting" it onto the y error matrix. If the fit function is approximated in each point by its tangent line, the *Gaussian* errors in the x direction are not warped by this projection.

...

For examples on how to use **kafe**, see the `examples` folder. Consulting the API can also be helpful.

CHAPTER

TWO

API DOCUMENTATION

2.1 kafe Package

2.1.1 `kafe` Package

A Python package for fitting and plotting for use in physics lab courses.

This Python package allows fitting of user-defined functions to data. A dataset is represented by a *Dataset* object which stores measurement data as *NumPy* arrays. The uncertainties of the data are also stored in the *Dataset* as an *error matrix*, allowing for both correlated and uncorrelated errors to be accurately represented.

The constructor of a *Dataset* object accepts several keyword arguments and can be used to construct a *Dataset* out of data which has been loaded into *Python* as *NumPy* arrays. Alternatively, a plain-text representation of a *Dataset* can be read from a file.

Also provided are helper functions which construct a *Dataset* object from a file containing column data (one measurement per row, column order can be specified).

2.1.2 `_version_info` Module

2.1.3 `constants` Module

`F_SIGNIFICANCE = 2`
: Set significance for returning results and errors N = rounding error to N significant digits and value to the same order of magnitude as the error.

`G_PADDING_FACTOR_X = 1.2`
: factor by which to expand x data range

`G_PADDING_FACTOR_Y = 1.2`
: factor by which to expand y data range

`G_PLOT_POINTS = 200`
: number of plot points for plotting the function

`M_CONFIDENCE_LEVEL = 0.05`
: Confidence level for hypythesis test. A fit is rejected it χ^2_{prob} is smaller than this constant

`M_MAX_ITERATIONS = 6000`
: Maximum *Minuit* iterations until aborting the process

`M_MAX_X_FIT_ITERATIONS = 2`
: Number of maximal additional iterations for x fit (o disregards x errors)

M_TOLERANCE = 0.1
Minuit tolerance level

2.1.4 `dataset` Module

class `Dataset`(***kwargs*)
The *Dataset* object is a data structure for storing measurement and error data. In this implementation, the *Dataset* has the compulsory field *data*, which is used for storing the measurement data, and another field *cov_mats*, used for storing the covariance matrix for each axis.

There are several ways a *Dataset* can be constructed. The most straightforward way is to specify an input file containing a plain-text representation of the dataset:

```
>>> my_dataset = Dataset(input_file='/path/to/file')
```

or

```
>>> my_dataset = Dataset(input_file=my_file_object)
```

If an *input_file* keyword is provided, all other input is ignored. The *Dataset* plain-text representation format is as follows:

```
# x data
x_1  sigma_x_1
x_2  sigma_x_2  cor_x_12
...  ...        ...       ...
x_N  sigma_x_N  cor_x_1N  ...  cor_x_NN

# y data
y_1  sigma_y_1
y_2  sigma_y_2  cor_y_12
...  ...        ...       ...
y_N  sigma_y_N  cor_y_1N  ...  cor_y_NN
```

Here, the *sigma_...* represents the statistical error of the data point and *cor_..._ij* is the correlation coefficient between the *i*-th and *j*-th data point.

Alternatively, field data can be set by passing iterables as keyword arguments. Available keywords for this purpose are:

data : tuple/list of tuples/lists/arrays of floats

a tuple/list of measurement data. Each element of the tuple/list must be iterable and be of the same length. The first element of the **data** tuple/list is assumed to be the *x* data, and the second to be the *y* data:

```
>>> my_dataset = Dataset(data=([0., 1., 2., 3., 4.], [1.23, 3.45, 5.62, 7.88, 9.64]))
```

Alternatively, x-y value pairs can also be passed as **data**. The following is equivalent to the above:

```
>>> my_dataset = Dataset(data=([0.0, 1.23], [1.0, 3.45], [2.0, 5.62], [3.0, 7.88], [4.0, 9.64]))
```

In case the *Dataset* contains two data points, the ordering is ambiguous. In this case, the first ordering (*x* data first, then *y* data) is assumed.

cov_mats : tuple/list of *numpy.matrix*

a tuple/list of two-dimensional iterables containing the covariance matrices for *x* and *y*, in that order. Covariance matrices can be any sort of two-dimensional NxN iterables, assuming N is the number of data points.

```
>>> my_dataset = Dataset(data=([0., 1., 2.], [1.23, 3.45, 5.62]), cov_mats=(my_cov_mat_x, my_cov_mat_
```

This keyword argument can be omitted, in which case covariance matrices of zero are assumed. To specify a covariance matrix for a single axis, replace the other with `None`.

```
>>> my_dataset = Dataset(data=([0., 1., 2.], [1.23, 3.45, 5.62]), cov_mats=(None, my_cov_mat_y))
```

title : string

the name of the *Dataset*. If omitted, the *Dataset* will be given the generic name 'Untitled Dataset'.

`axis_labels` = **None**
axis labels

`axis_units` = **None**
units to assume for axis

`cov_mat_is_regular`(*axis*)
Returns *True* if the covariance matrix for an axis is regular and `False` if it is singular.

axis ['x' or 'y'] Axis for which to check for regularity of the covariance matrix.

`cov_mats` = **None**
list of covariance matrices

`data` = **None**
list containing measurement data (axis-ordering)

`get_axis`(*axis_alias*)
Get axis id from an alias.

axis_alias [string or int] Alias of the axis whose id should be returned. This is for example either '0' or 'x' for the *x*-axis (id 0).

`get_cov_mat`(*axis*, *fallback_on_singular=None*)
Get the error matrix for an axis.

axis ['x' or 'y'] Axis for which to load the error matrix.

fallback_on_singular [*numpy.matrix* or string (optional)] What to return if the matrix is singular. If this is `None` (default), the matrix is returned anyway. If this is a *numpy.matrix* object or similar, that is returned istead. Alternatively, the shortcuts 'identity' or 1 and 'zero' or 0 can be used to return the identity and zero matrix respectively.

`get_data`(*axis*)
Get the measurement data for an axis.

axis [string] Axis for which to get the measurement data. Can be 'x' or 'y'.

`get_data_span`(*axis*, *include_error_bars=False*)
Get the data span for an axis. The data span is a tuple (*min*, *max*) containing the smallest and highest coordinates for an axis.

axis ['x' or 'y'] Axis for which to get the data span.

include_error_bars [boolean (optional)] `True` if the returned span should be enlarged to contain the error bars of the smallest and largest datapoints (default: `False`)

`get_formatted`(*format_string='.06e'*, *delimiter='t'*)
Returns the dataset in a plain-text format which is human-readable and can later be used as an input file for the creation of a new *Dataset*. The format is as follows:

```
# x data
x_1  sigma_x_1
x_2  sigma_x_2  cor_x_12
```

```
...  ...        ...       ...
x_N  sigma_x_N  cor_x_1N  ...  cor_x_NN

# y data
y_1  sigma_y_1
y_2  sigma_y_2  cor_y_12
...  ...        ...       ...
y_N  sigma_y_N  cor_y_1N  ...  cor_y_NN
```

Here, the `x_i` and `y_i` represent the measurement data, the `sigma_?_i` are the statistical uncertainties of each data point, and the `cor_?_ij` are the correlation coefficients between the *i*-th and *j*-th data point.

If the `x` or `y` errors are not correlated, then the entire correlation coefficient matrix can be omitted. If there are no statistical uncertainties for an axis, the second column can also be omitted. A blank line is required at the end of each data block!

format_string [string (optional)] A format string with which each entry will be rendered. Default is `'.06e'`, which means the numbers are represented in scientific notation with six significant digits.

delimiter [string (optional)] A delimiter used to separate columns in the output.

`get_size()`
Get the size of the *Dataset*. This is equivalent to the length of the *x*-axis data.

`has_correlations`(*axis*)
Returns *True* if the specified axis has correlation data, `False` if not. singular.

axis [`'x'` or `'y'`] Axis for which to check for correlations.

`has_errors`(*axis*)
Returns *True* if the specified axis has statistical error data.

axis [`'x'` or `'y'`] Axis for which to check for error data.

`n_axes` = **None**
dimensionality of the *Dataset*. Currently, only 2D *Datasets* are supported

`n_datapoints` = **None**
number of data points in the *Dataset*

`read_from_file`(*input_file*)
Reads the *Dataset* object from a file.

returns [boolean] `True` if the read succeeded, `False` if not.

`set_cov_mat`(*axis*, *mat*)
Set the error matrix for an axis.

axis [`'x'` or `'y'`] Axis for which to load the error matrix.

mat [*numpy.matrix* or `None`] Error matrix for the axis. Passing `None` unsets the error matrix.

`set_data`(*axis*, *data*)
Set the measurement data for an axis.

axis [`'x'` or `'y'`] Axis for which to set the measurement data.

data [iterable] Measurement data for axis.

`write_formatted`(*file_path*, *format_string='.06e'*, *delimiter='t'*)
Writes the dataset to a plain-text file. For details on the format, see get_formatted.

file_path [string] Path of the file object to write. **WARNING**: *overwrites existing files*!

format_string [string (optional)] A format string with which each entry will be rendered. Default is `'.06e'`, which means the numbers are represented in scientific notation with six significant digits.

delimiter [string (optional)] A delimiter used to separate columns in the output.

`build_dataset`(*xdata*, *ydata*, ***kwargs*)

This helper function creates a *Dataset* from a series of keyword arguments.

Valid keyword arguments are:

xdata and ydata [list/tuple/*np.array* of floats] These keyword arguments are mandatory and should be iterables containing the measurement data.

error specification keywords [iterable or numeric (see below)] A valid keyword is composed of an axis (*x* or *y*), an error relativity specification (*abs* or *rel*) and error correlation type (*stat* or *syst*). The errors are then set as follows:

1. **For statistical errors:**
   - if keyword argument is iterable, the error list is set to that
   - if keyword argument is a number, an error list with identical entries is generated
2. **For systematic errors:**
   - keyword argument *must* be a single number. The global correlated error for the axis is then set to that.

So, for example:

```
>>> myDataset = build_dataset(..., yabsstat=0.3, yrelsyst=0.1)
```

creates a dataset where the statistical error of each *y* coordinate is set to 0.3 and the overall systematic error of *y* is set to 0.1.

`debug_print`(*message*)

2.1.5 `file_tools` Module

`parse_column_data`(*file_to_parse*, *field_order='x,y'*, *delimiter=' '*, *cov_mat_files=None*, *title='Untitled Dataset'*)

Parses a file which contains measurement data in a one-measurement-per-row format. The field (column) order can be specified. It defaults to *x,y'. Valid field names are 'x*, *y*, *xabsstat*, *yabsstat*, *xrelstat*, *yrelstat*. Another valid field name is *ignore* which can be used to skip a field.

Every valid measurement data file *must* have an *x* and a *y* field.

Additionally, a delimiter can be specified. If this is a whitespace character or omitted, any sequence of whitespace characters is assumed to separate the data.

If the measurement errors and correlations are given as covariance matrices (in a separate file), these files can be specified using the *cov_mat_files* argument.

file_to_parse [file-like object or string containing a file path] The file to parse.

field_order [string (optional)] A string of comma-separated field names giving the order of the columns in the file. Defaults to `'x,y'`.

delimiter [string (optional)] The field delimiter used in the file. Defaults to any whitespace.

cov_mat_files [`None` or tuple of strings/file-like objects (optional)] Files which contain x- and y-covariance matrices, in that order. Defaults to `None`.

return [*Dataset*] A Dataset built from the parsed file.

`parse_matrix_file`(*file_like*, *delimiter=None*)

Read a matrix from a matrix file. The format of the matrix file should be:

```
# comment row
a_11  a_12  ...  a_1M
a_21  a_22  ...  a_2M
...   ...   ...  ...
a_N1  a_N2  ...  a_NM
```

file_like [string or file-like object] File path or file object to read matrix from.

delimiter [`None` or string (optional)] Column delimiter use in the matrix file. Defaults to `None`, meaning any whitespace.

2.1.6 `fit` Module

class `Fit`(*dataset*, *fit_function*, *external_fcn=<function chi2 at 0x28452a8>*, *function_label=None*, *function_equation=None*)

Object representing a fit. This object references the fitted *Dataset*, the fit function and the resulting fit parameters.

Necessary arguments are a *Dataset* object and a fit function (which should be fitted to the *Dataset*). Optionally, an external function *FCN* (whose minima should be located to find the best fit) can be specified. If not given, the *FCN* function defaults to χ^2.

dataset [*Dataset*] A *Dataset* object containing all information about the data

fit_function [function] A user-defined Python function to be fitted to the data. This function's first argument must be the independent variable *x*. All other arguments *must* be named and have default values given. These defaults are used as a starting point for the actual minimization. For example, a simple linear function would be defined like:

```
>>> def linear_2par(x, slope=1, y_intercept=0):
...     return slope * x + y_intercept
```

Be aware that choosing sensible initial values for the parameters is often crucial for a succesful fit, particularly for functions of many parameters.

external_fcn [function (optional)] An external *FCN* (function to minimize). This function must have the following call signature:

```
>>> FCN(xdata, ydata, cov_mat, fit_function, param_values)
```

It should return a float. If not specified, the default χ^2 *FCN* is used. This should be sufficient for most fits.

function_label [LaTeX-formatted string (optional)] A name/label/short description of the fit function. This appears in the legend describing the fitter curve. If omitted, this defaults to the function's Python name.

function_equation [LaTeX-formatted string (optional)] The fit function's equation.

`call_external_fcn`(**param_values*)

Wrapper for the external *FCN*. Since the actual fit process depends on finding the right parameter values and keeping everything else constant, we can use the *Dataset* object to pass known, fixed information to the external *FCN*, varying only the parameter values.

param_values [sequence of values] the parameter values at which *FCN* is to be evaluated

`current_cov_mat` = **None**

the current covariance matrix used for the *Fit*

`current_param_errors` = **None**
the current uncertainties of the parameters

`current_param_values` = **None**
the current values of the parameters

`dataset` = **None**
this Fit instance's child *Dataset*

`do_fit`(*quiet=False*, *verbose=False*)
Runs the fit algorithm for this *Fit* object.

First, the *Dataset* is fitted considering only uncertainties in the *y* direction. If the *Dataset* has no uncertainties in the *y* direction, they are assumed to be equal to 1.0 for this preliminary fit, as there is no better information available.

Next, the fit errors in the *x* direction (if they exist) are taken into account by projecting the covariance matrix for the *x* errors onto the *y* covariance matrix. This is done by taking the first derivative of the fit function in each point and "projecting" the *x* error onto the resulting tangent to the curve.

This last step is repeater until the change in the error matrix caused by the projection becomes negligible.

quiet [boolean (optional)] Set to `True` if no output should be printed.

verbose [boolean (optional)] Set to `True` if more output should be printed.

`external_fcn` = **None**
the (external) function to be minimized for this *Fit*

`fit_function` = **None**
the fit function used for this *Fit*

`fit_one_iteration`(*verbose=False*)
Instructs the minimizer to do a minimization.

`function_equation` = **None**
LaTeX function equation

`function_label` = **None**
a label to use in the legend when plotting

`get_current_fit_function`()
This method returns a function object corresponding to the fit function for the current parameter values. The returned function is a function of a single variable.

returns [function] A function of a single variable corresponding to the fit function at the current parameter values.

`get_error_matrix`()
This method returns the covariance matrix of the fit parameters which is obtained by querying the minimizer object for this fit

returns [*numpy.matrix*] The covariance matrix of the parameters.

`get_parameter_errors`(*rounding=False*)
Get the current parameter uncertainties from the minimizer.

rounding [boolean (optional)] Whether or not to round the returned values to significance.

returns [tuple] A tuple of the parameter uncertainties

`get_parameter_values`(*rounding=False*)
Get the current parameter values from the minimizer.

rounding [boolean (optional)] Whether or not to round the returned values to significance.

returns [tuple] A tuple of the parameter values

`minimizer = None`
this *Fit*'s minimizer (*Minuit*)

`number_of_parameters = None`
the number of parameters

`param_names = None`
the names of the parameters

`param_names_latex = None`
LaTeX parameter names

`print_fit_details()`
prints some fit goodness details

`print_fit_results()`
prints fit results

`print_rounded_fit_parameters()`
prints the fit parameters

`project_x_covariance_matrix()`
Project the x errors from the x covariance matrix onto the total matrix.

This is done elementwise, according to the formula:

$$C_{tot,ij} = C_{y,ij} + C_{x,ij} \frac{\partial f}{\partial x_i} \frac{\partial f}{\partial x_j}$$

`xdata = None`
the x coordinates of the data points used for this *Fit*

`ydata = None`
the y coordinates of the data points used for this *Fit*

`chi2`(*xdata*, *ydata*, *cov_mat*, *fit_function*, *param_values*)
A simple χ^2 implementation. Calculates χ^2 according to the formula:

$$\chi^2 = \lambda^T C^{-1} \lambda$$

Here, λ is the residual vector $\lambda = \vec{y} - \vec{f}(\vec{x})$ and C is the covariance matrix.

xdata [iterable] The x measurement data

ydata [iterable] The y measurement data

cov_mat [*numpy.matrix*] The total covariance matrix

fit_function [function] The fit function $f(x)$

param_values [list/tuple] The values of the parameters at which $f(x)$ should be evaluated.

`round_to_significance`(*value*, *error*, *significance=2*)
Rounds the error to the established number of significant digits, then rounds the value to the same order of magnitude as the error.

value [float] value to round to significance

error [float] uncertainty of the value

significance [int (optional)] number of significant digits of the error to consider

2.1.7 `function_library` Module

`constant_1par`(*x*, *constant=1.0*)

`exp_2par`(*x*, *growth=1.0*, *constant_factor=0.0*)

`exp_3par`(*x*, *growth=1.0*, *constant_factor=0.0*, *y_offset=0.0*)

`exp_3par2`(*x*, *growth=1.0*, *constant_factor=0.0*, *x_offset=0.0*)

`exp_4par`(*x*, *growth=1.0*, *constant_factor=0.0*, *x_offset=0.0*, *y_offset=0.0*)

`gauss`(*x*, *mean=0.0*, *sigma=1.0*, *scale=1.0*)

`linear_1par`(*x*, *slope=1.0*)

`linear_2par`(*x*, *slope=1.0*, *y_intercept=0.0*)

`linear_2par2`(*x*, *slope=1.0*, *x_offset=0.0*)

`poisson`(*x*, *mean=0.0*, *scale=1.0*)

`poly3`(*x*, *coeff3=1.0*, *coeff2=0.0*, *coeff1=0.0*, *coeff0=0.0*)

`poly4`(*x*, *coeff4=1.0*, *coeff3=0.0*, *coeff2=0.0*, *coeff1=0.0*, *coeff0=0.0*)

`poly5`(*x*, *coeff5=1.0*, *coeff4=0.0*, *coeff3=0.0*, *coeff2=0.0*, *coeff1=0.0*, *coeff0=0.0*)

`quadratic_1par`(*x*, *quad_coeff=1.0*)

`quadratic_2par`(*x*, *quad_coeff=1.0*, *constant=0.0*)

`quadratic_2par2`(*x*, *quad_coeff=1.0*, *x_offset=0.0*)

`quadratic_3par`(*x*, *quad_coeff=1.0*, *lin_coeff=0.0*, *constant=0.0*)

`quadratic_3par2`(*x*, *quad_coeff=1.0*, *x_offset=0.0*, *constant=0.0*)

2.1.8 `function_tools` Module

`derivative`(*func*, *derive_by_index*, *variables_tuple*, *derivative_spacing*)
Gives $\frac{\partial f}{\partial x_k}$ for $f = f(x_0, x_1, \ldots)$. *func* is f, *variables_tuple* is $\{x_i\}$ and *derive_by_index* is k.

`derive_by_parameters`(*func*, *x_0*, *param_list*, *derivative_spacing*)
Returns the gradient of *func* with respect to its parameters, i.e. with respect to every variable of *func* except the first one.

`derive_by_x`(*func*, *x_0*, *param_list*, *derivative_spacing*)
If *x_0* is iterable, gives the array of derivatives of a function $f(x, par_1, par_2, \ldots)$ around $x = x_i$ at every x_i in $\vec{x}$. If *x_0* is not iterable, gives the derivative of a function $f(x, par_1, par_2, \ldots)$ around $x =$ `x_0`.

`get_function_property`(*func*, *prop*)
Returns a specific property of the function. This assumes that the function is defined as

```
>>> def func(x, par1=1.0, par2=3.14, par3=2.71, ...): ...
```

func [function] A function object from which to extract the property.

prop [any of `'name'`, `'parameter names'`, `'parameter defaults'`, `'number of parameters'`]
A string representing a property.

`outer_product`(*input_array*)
Takes a *NumPy* array and returns the outer (dyadic, Kronecker) product with itself. If *input_array* is a vector $\mathbf{x}$, this returns $\mathbf{x}\mathbf{x}^T$.

2.1.9 `minuit` Module

`D_MATRIX_ERROR` = {0: 'Error matrix not calculated', 1: 'Error matrix approximate!', 2: 'Error matrix forced positive d

Error matrix status codes

class `Minuit`(*number_of_parameters*, *function_to_minimize*, *par_names*, *start_params*, *param_errors*, *quiet=True*, *verbose=False*)

A class for communicating with ROOT's function minimizer tool Minuit.

`FCN_wrapper`(*number_of_parameters*, *derivatives*, *f*, *parameters*, *internal_flag*)

This is actually a function called in *ROOT* and acting as a C wrapper for our *FCN*, which is implemented in Python.

This function is called by *Minuit* several times during a fit. It doesn't return anything but modifies one of its arguments (*f*). This is *ugly*, but it's how *ROOT*'s `TMinuit` works. Its argument structure is fixed and determined by *Minuit*:

number_of_parameters [int] The number of parameters of the current fit

derivatives [??] Computed gradient (??)

f [C array] The desired function value is in f[0] after execution.

parameters [C array] A C array of parameters. Is cast to a Python list

internal_flag [int] A flag allowing for different behaviour of the function. Can be any integer from 1 (initial run) to 4(normal run). See *Minuit*'s specification.

`function_to_minimize` = **None**

the actual *FCN* called in `FCN_wrapper`

`get_chi2_probability`(*n_deg_of_freedom*)

Returns the probability that an observed χ^2 exceeds the calculated value of χ^2 for this fit by chance, even for a correct model. In other words, returns the probability that a worse fit of the model to the data exists. If this is a small value (typically <5%), this means the fit is pretty bad. For values below this threshold, the model very probably does not fit the data.

n_def_of_freedom [int] The number of degrees of freedom. This is typically $n_extdatapoints - n_extparameters$.

`get_contour`(*parameter1*, *parameter2*, *n_points=20*)

Returns a list of points (2-tuples) representing a sampling of the 1σ contour of the TMinuit fit. The `FCN` has to be minimized before calling this.

parameter1 [int] ID of the parameter to be displayed on the *x*-axis.

parameter2 [int] ID of the parameter to be displayed on the *y*-axis.

n_points [int (optional)] number of points used to draw the contour. Default is 20.

returns [2-tuple of tuples] a 2-tuple (x, y) containing `n_points+1` points sampled along the contour. The first point is repeated at the end of the list to generate a closed contour.

`get_error_matrix`()

Retrieves the parameter error matrix from TMinuit.

return : *numpy.matrix*

`get_fit_info`(*info*)

Retrieves other info from *Minuit*.

info [string]

Information about the fit to retrieve. This can be any of the following:

- `'fcn'`: *FCN* value at minimum,

- `'edm'`: estimated distance to minimum
- `'err_def'`: *Minuit* error matrix status code
- `'status_code'`: *Minuit* general status code

`get_parameter_errors()`
Retrieves the parameter errors from TMinuit.

return [tuple] Current *Minuit* parameter errors

`get_parameter_info()`
Retrieves parameter information from TMinuit.

return [list of tuples] `(param_name, param_val, param_error)`

`get_parameter_name(param_nr)`
Gets the name of parameter number `param_nr`

param_nr [int] Number of the parameter whose name to get.

`get_parameter_values()`
Retrieves the parameter values from TMinuit.

return [tuple] Current *Minuit* parameter values

`max_iterations = None`
maximum number of iterations until `TMinuit` gives up

`minimize(log_print_level=3)`
Do the minimization. This calls *Minuit*'s algorithms `MIGRAD` for minimization and `HESSE` for computing/checking the parameter error matrix.

`number_of_parameters = None`
number of parameters to minimize for

`reset()`

`set_err(up_value=1.0)`
Sets the `UP` value for Minuit.

up_value [float (optional, default: 1.0)] This is the value by which *FCN* is expected to change.

`set_parameter_errors(param_errors=None)`
Sets the fit parameter errors. If param_values='None', sets the error to 1% of the parameter value.

`set_parameter_names(param_names)`
Sets the fit parameters. If param_values='None', tries to infer defaults from the function_to_minimize.

`set_parameter_values(param_values)`
Sets the fit parameters. If param_values='None', tries to infer defaults from the function_to_minimize.

`set_print_level(print_level=1)`
Sets the print level for Minuit.

print_level [int (optional, default: 1 (frugal output))] Tells `TMinuit` how much output to generate. The higher this value, the more output it generates.

`set_strategy(strategy_id=1)`
Sets the strategy Minuit.

strategy_id [int (optional, default: 1 (optimized))] Tells `TMinuit` to use a certain strategy. Refer to `TMinuit`'s documentation for available strategies.

`tolerance` = **None**
`TMinuit` tolerance

2.1.10 `numeric_tools` Module

`cor_to_cov`(*cor_mat*, *error_list*)
Converts a correlation matrix to a covariance matrix according to the formula

$$\mathrm{Cov}_{ij} = \mathrm{Cor}_{ij}\,\sigma_i\,\sigma_j$$

cor_mat [*numpy.matrix*] The correlation matrix to convert.

error_list [sequence of floats] A sequence of statistical errors. Must be of the same length as the diagonal of *cor_mat*.

`cov_to_cor`(*cov_mat*)
Converts a covariance matrix to a correlation matrix according to the formula

$$\mathrm{Cor}_{ij} = \frac{\mathrm{Cov}_{ij}}{\sqrt{\mathrm{Cov}_{ii}\,\mathrm{Cov}_{jj}}}$$

cov_mat The covariance matrix to convert. Type: *numpy.matrix*

`extract_statistical_errors`(*cov_mat*)
Extracts the statistical errors from a covariance matrix. This means it returns the (elementwise) square root of the diagonal entries

cov_mat The covariance matrix to extract errors from. Type: *numpy.matrix*

`make_symmetric_lower`(*mat*)
Copies the matrix entries below the main diagonal to the upper triangle half of the matrix. Leaves the diagonal unchanged. Returns a *NumPy* matrix object.

mat [*numpy.matrix*] A lower diagonal matrix.

returns [*numpy.matrix*] The lower triangle matrix.

`zero_pad_lower_triangle`(*triangle_list*)
Converts a list of lists into a lower triangle matrix. The list members should be lists of increasing length from 1 to N, N being the dimension of the resulting lower triangle matrix. Returns a *NumPy* matrix object.

For example:

```
>>> zero_pad_lower_triangle([ [1.0], [0.2, 1.0], [0.01, 0.4, 3.0] ])
matrix([[ 1.  ,  0.  ,  0.  ],
        [ 0.2 ,  1.  ,  0.  ],
        [ 0.01,  0.4 ,  3.  ]])
```

triangle_list [list] A list containing lists of increasing length.

returns [*numpy.matrix*] The lower triangle matrix.

2.1.11 `plot` Module

class `Plot`(**fits*, ***kwargs*)

`axis_labels` = **None**
axis labels

`compute_plot_range`(*include_error_bars=True*)
Compute the span of all child datasets and sets the plot range to that

`draw_fit_parameters_box`(*plot_spec=0*)
Draw the parameter box to the canvas

plot_spec [int, list of ints, string or None (optional, default: 0)] Specify the plot id of the plot for which to draw the parameters. Passing 0 will only draw the parameter box for the first plot, and so on. Passing a list of ints will only draw the parameters for plot ids inside the list. Passing `'all'` will print parameters for all plots. Passing `None` will return immediately doing nothing.

`draw_legend`()
Draw the plot legend to the canvas

`extend_span`(*axis*, *new_span*)
Expand the span of the current plot.

This method extends the current plot span to include *new_span*

`fits` = **None**
list of 'Fit's to plot

`init_plots`()
Initialize the plots for each fit.

`plot`(*p_id*, *show_data=True*)
Plot the *Fit* object with the number *p_id* to its figure.

`plot_all`(*show_info_for='all'*, *show_data_for='all'*)
Plot every *Fit* object to its figure.

`plot_range` = **None**
plot range

`plot_style` = **None**
plot style

`save`(*output_file*)
Save the *Plot* to a file.

`show`()
Show the *Plot* in a matplotlib interactive window.

`show_legend` = **None**
whether to show the plot legend (`True`) or not (`False`)

class `PlotStyle`
Class for specifying a style for a specific plot. This object stores a progression of marker and line types and colors, as well as preferences relating to point size and label size. These can be overriden by overwriting the instance variables directly. A series of *get_...* methods are provided which go through these lists cyclically.

`get_line`(*idm*)
Get a specific line type. This runs cyclically through the defined defaults.

`get_linecolor`(*idm*)
Get a specific line color. This runs cyclically through the defined defaults.

`get_marker`(*idm*)
Get a specific marker type. This runs cyclically through the defined defaults.

`get_markercolor`(*idm*)
Get a specific marker color. This runs cyclically through the defined defaults.

`get_pointsize`(*idm*)
Get a specific point size. This runs cyclically through the defined defaults.

`label_to_latex`(*label*)
Generates a simple LaTeX-formatted label from a plain-text label. This treats isolated characters and words beginning with a backslash as mathematical expressions and surround them with $ signs accordingly.

label [string] Plain-text string to convert to LaTeX.

`pad_span`(*span*, *pad_coeff=1*, *additional_pad=None*)
Enlarges the interval *span* (list of two floats) symmetrically around its center to length *pad_coeff*. Optionally, an *additional_pad* argument can be specified. The returned span is then additionally enlarged by that amount.

additional_pad can also be a list of two floats which specifies an asymmetric amount by which to enlarge the span. Note that in this case, positive entries in *additional_pad* will enlarge the span (move the interval end away from the interval's center) and negative amounts will shorten it (move the interval end towards the interval's center).

2.1.12 `stream` Module

class `StreamDup`(*out_file*)
Bases: `object`

Object for simultaneous logging to stdout and a file.

`fileno`()

`flush`()

`write`(*message*)

`write_timestamp`(*prefix*)

`write_to_file`(*message*)

`write_to_stdout`(*message*)

2.1.13 Indices and tables

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

INDEX

H

I

K

L

M

N

O

P

Q

R

S

T

W

X

Y

Z