

---

# **kafe Documentation**

***Release 1.2.0***

**D. Savoiu, G. Quast**

October 16, 2016



## CONTENTS

<b>1</b>	<b><i>kafe</i> Overview</b>	<b>3</b>
1.1	Code Structure . . . . .	3
1.2	Fitting in a Nutshell . . . . .	4
1.3	Example . . . . .	5
<b>2</b>	<b>Installing <i>kafe</i></b>	<b>7</b>
2.1	Requirements . . . . .	7
2.2	Installation notes (Linux) . . . . .	7
2.3	Installation notes (Windows) . . . . .	9
<b>3</b>	<b>Fit examples, utilities, tips and tricks</b>	<b>11</b>
3.1	Example 1 - model comparison . . . . .	11
3.2	Example 2 - two fits and models . . . . .	12
3.3	Example 3 - non-linear fit with non-parabolic errors . . . . .	14
3.4	Example 4 - average of correlated measurements . . . . .	16
3.5	Example 5 - non-linear multi-parameter fit (damped oscillation) . . . . .	17
3.6	Example 6 - linear multi-parameter fit . . . . .	18
3.7	Example 7 - another non-linear multi-parameter fit (double-slit spectrum) . . . . .	20
3.8	Example 8 - fit of a Breit-Wigner resonance to data with correlated errors . . . . .	21
3.9	Example 9 - fit of a function to histogram data . . . . .	23
3.10	Example 10 - plotting with <i>kafe</i> : properties of a Gauss curve . . . . .	25
<b>4</b>	<b><i>kafe</i> reference</b>	<b>27</b>
4.1	<i>kafe</i> in a nutshell . . . . .	27
4.2	Main modules: <code>dataset</code> , <code>fit</code> and <code>plot</code> . . . . .	28
4.3	Interfaces to external packages . . . . .	41
4.4	<i>kafe</i> configuration . . . . .	46
4.5	Modules with helper tools . . . . .	47
4.6	Auxilliary modules . . . . .	55
	<b>Python Module Index</b>	<b>57</b>
	<b>Index</b>	<b>59</b>



**kafe** is a data fitting framework designed for use in undergraduate physics lab courses. It provides a basic Python toolkit for fitting models to data as well as visualisation of the data and the model function. It relies on Python packages such as `numpy` and `matplotlib`, and uses the Python interface to the minimizer *Minuit* contained in the data analysis framework *ROOT* or in the Python package *iminuit*.



## KAFE OVERVIEW

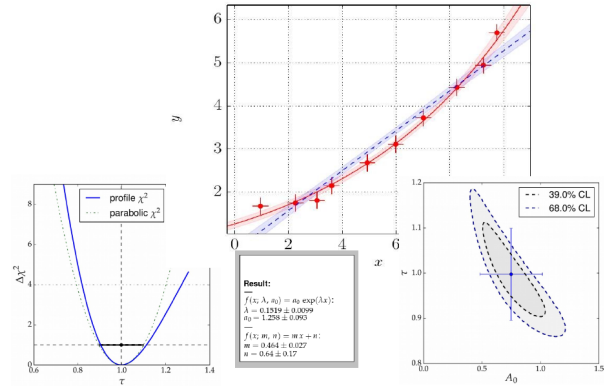


Fig. 1.1: Graphical output generated with *kafe*.

The *kafe* package provides a rather general approach to fitting of a model function to two-dimensional data points with correlated uncertainties in both dimensions. The Python API guarantees full flexibility for data input. Helper functions for file-based input and some examples are available for own applications.

Applications range from performing a simple average of measurements to complex situations with both correlated (systematic) and uncorrelated (statistical) uncertainties on the measurements of the  $x$  and  $y$  values described by a non-linear model function depending on a large number of parameters.

The model function describes the  $y$  values as a function of the  $x$ -values and a set of model parameters  $\{p\}$ ,  $y=f(x; \{p\})$ . Full flexibility exists as model functions are implemented as Python code. Again, examples are provided, but user implementations are supported as well.

Fitting is based on the  $\chi^2$ -method, assuming Gaussian errors and correlations described by covariance matrices. The level of agreement between data points and the fit model is expressed in terms of the  $\chi^2$  probability, i. e. the probability to find less agreement between data and model than actually observed. Full access to the covariance matrix of the - typically correlated - model parameters is provided.

The graphical output visualises the data and the fit model at the best-fit-point of the parameters and also shows the uncertainty of the fit model as a light band surrounding the line representing the model function. Plotting of confidence level contours for pairs of parameters or profiling of the  $\chi^2$  curves for each of the fit parameters are also provided.

### 1.1 Code Structure

The code of *kafe* is centred around very few classes to handle Data input, fitting and plotting, as illustrated in the figure on the right-hand side.

Data, their uncertainties, and, optionally, the correlations of the uncertainties - are passed through the interface of the *kafe* class *Dataset* (page 28). Input can be included in the Python code or is read from files in standardised or user-defined formats. The representation of the data within the *Dataset* (page 28) class is minimalistic,

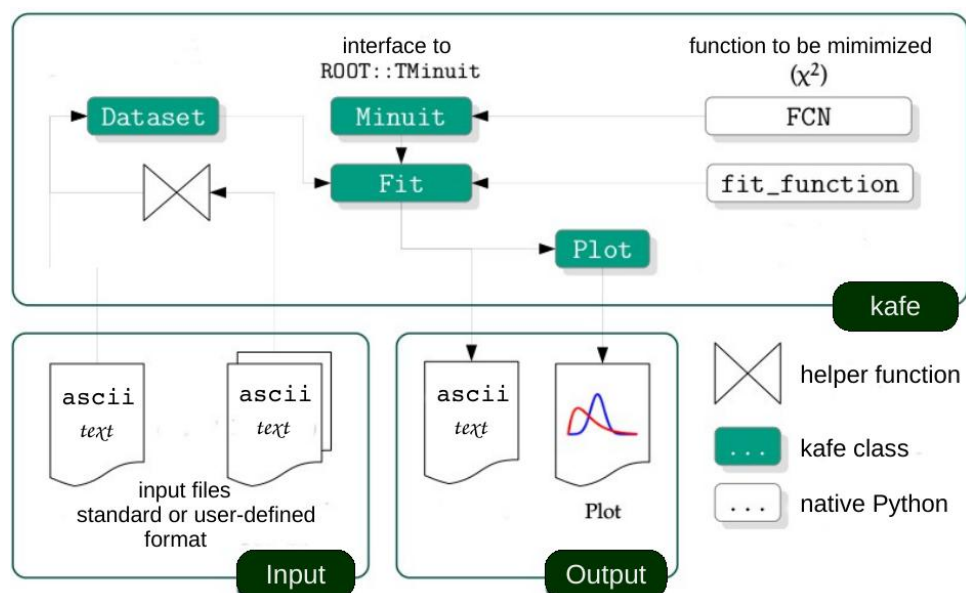


Fig. 1.2: Code structure of the kafe package

consisting of the  $x$  and  $y$  values and the full covariance matrices of their uncertainties. Correlated errors between  $x$  and  $y$  values are not supported yet, as such use cases are rare.

A helper function, `build_dataset()` (page 47), is available to transform various error models, like a combination of independent and correlated errors or common absolute or relative errors, to this basic format.

Adding a model function, taken either from a prepared set of fit functions within kafe or from a user's own Python implementation, results in a `Fit` (page 33) object, which controls the minimizer `Minuit` (page 41). Access to the final results of the fitting procedure is provided by data members of the `Fit` class.

One or multiple fit objects, i. e. the input data and model functions(s) at the best-fit point in parameter-space, are visualised by the class `Plot` (page 40) with the help of `matplotlib` functionality. The `plot` (page 40) module also contains functionality to display the model uncertainty by surrounding the model function at the best-fit values of the parameters by a light band, the one- $\sigma$  uncertainty band, which is obtained by propagation of the uncertainties of the fit parameters taking into account their correlations.

Two-dimensional contour lines of pairs of parameters are obtained with the method `plot_contour()` (page 36) of the `Fit` class, which internally relies on the `mncont` method of the `Minuit` package. Contour curves are obtained from a scan of the  $\chi^2$ -function around a fixed value, where each point on the curve represents the minimum with respect to all other free parameters in the fit, thus taking into account the correlation of the considered pair of parameters with all other parameters of the model.

In a similar way, the method `plot_profile()` (page 37) provides profiled  $\chi^2$  curves, i. e. the value of the minimal  $\chi^2$  as a function of one parameter while all other parameters are allowed to vary.

## 1.2 Fitting in a Nutshell

Fitting with **kafe** in a nutshell goes like this:

1. create a `Dataset` (page 28) object from your measurement data:

```
>>> my_d = kafe.Dataset(data=[[0., 1., 2.], [1.23, 3.45, 5.62]])
```

2. add errors (uncertainties) to your `Dataset` (page 28):

```
>>> my_d.add_error_source('y', 'simple', 0.5) # y errors, all +/- 0.5
```

3. import a model function from `kafe.function_library` (page 55) (or define one yourself):



```
>>> from kafe.function_library import linear_2par
```

4. create a *Fit* (page 33) object from your *Dataset* (page 28) and your model function:

```
>>> my_f = kafe.Fit(my_d, linear_2par)
```

5. do the fit:

```
>>> my_f.do_fit()
```

6. (optional) if you want to see a plot of the result, use the *Plot* (page 40) object:

```
>>> my_p = kafe.Plot(my_f)
>>> my_p.plot_all()
>>> my_p.show()
```

## 1.3 Example

Only very few lines of Python code are needed to perform fits with kafe. The snippet of code shown below performs a fit of a quadratic function to some data points with uncertainties:

```
from kafe import *
from kafe.function_library import quadratic_3par

#### build a Dataset instance:
myDataset = build_dataset(
    [0.05,0.36,0.68,0.80,1.09,1.46,1.71,1.83,2.44,2.09,3.72,4.36,4.60],
    [0.35,0.26,0.52,0.44,0.48,0.55,0.66,0.48,0.75,0.70,0.75,0.80,0.90],
    yabserr=[0.06,0.07,0.05,0.05,0.07,0.07,0.09,0.1,0.11,0.1,0.11,0.12,0.1],
    title='some data',
    axis_labels=['$x$', '$y=f(x)$'])

#### Create the Fit object
myFit = Fit(myDataset, quadratic_3par)
# Set initial values and error estimates
myFit.set_parameters((0., 1., 0.2), (0.5, 0.5, 0.5))
# Do the Fit
myFit.do_fit()

#### Create result plots and output them
myPlot = Plot(myFit)
myPlot.plot_all()
myPlot.save('kafe_example0.pdf') # to file

myPlot.show()                  # to screen
```

The output in text form (also available via various `get_...()` methods of the *Fit* (page 33) class) contains the values of the parameters at the best-fit point, their correlation matrix and the fit probability. The example produces the following graphical output:

The parametrisation chosen in this example leads to a strong correlation of the fit parameters. This can be graphically visualised by adding the following lines at the end of the example:

```
### Create and save contour plots
contour1 = myFit.plot_contour(0, 1, dchi2=[1., 2.3])
contour2 = myFit.plot_contour(0, 2, dchi2=[1., 2.3])
contour3 = myFit.plot_contour(1, 2, dchi2=[1., 2.3])
contour1.savefig('kafe_example0_contour1.pdf')
contour2.savefig('kafe_example0_contour2.pdf')
contour3.savefig('kafe_example0_contour3.pdf')
```

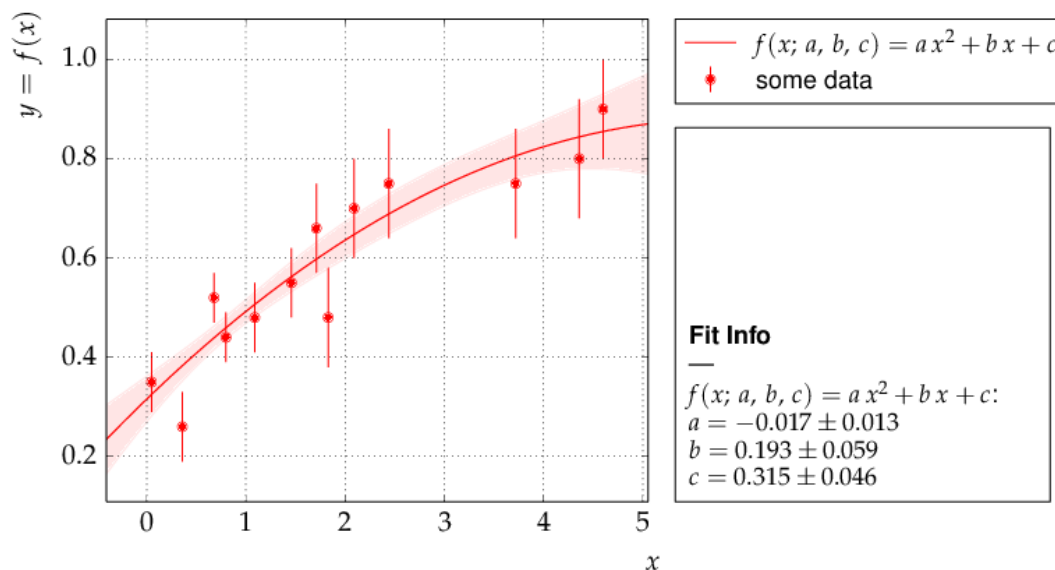


Fig. 1.3: Example: *Data points with one-dimensional error bars compared to a quadratic model function with kafe.*

The example code produces two confidence-level contours for each pair of parameters (with  $id=0$ ,  $id=1$  and  $id=2$ ), corresponding to an increase of the  $\chi^2$ -function with respect to the minimum by the values given in the list passed as the third parameter to the method `myFit.plot_contour()`. The resulting graphical representation, as shown below, displays the 39% contours, corresponding to the one-sigma errors, and the 68% contours. The uncertainties on each parameter, indicated by the error bars, are also shown. They correspond to the projections of the one-sigma contours on the axes.

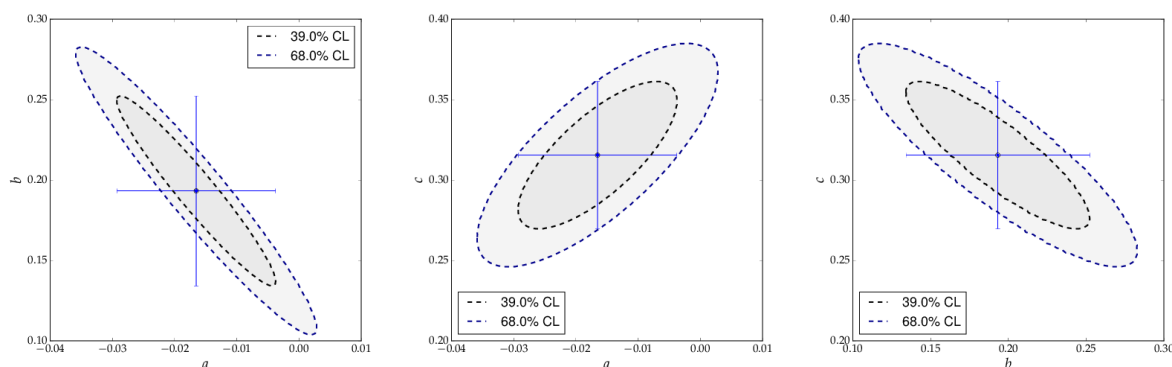


Fig. 1.4: *Contour curves of a pairs of paramters a, b and c of the example above, calculated with kafe.*

More and advanced examples - like fitting different models to one data set, comparison of different data sets with model functions, averaging of correlated measurements or fits with a large number of parameters - are provided as part of the *kafe* distribution and are described in the section *Examples* below. They may serve as a starting point for own applications.

## INSTALLING KAFE

### 2.1 Requirements

*kafe* needs some additional Python packages. The recommended versions of these are as follows:

- `SciPy`  $\geq 0.12.0$
- `NumPy`  $\geq 1.10.4$
- `matplotlib`  $\geq 1.5.0$

Additionally, a function minimizer is needed. *kafe* implements interfaces to two function minimizers and requires at least one of them to be installed:

- *MINUIT*, which is included in *CERN*'s data analysis package *ROOT* ( $\geq 5.34$ ), or
- *iminuit* ( $\geq 1.1.1$ ), which is independent of *ROOT* (this is the default)

Finally, *kafe* requires a number of external programs:

- *Qt4* ( $\geq 4.8.5$ ) and the Python bindings *PyQt4* ( $\geq 3.18.1$ ) are needed because *Qt* is the supported interactive frontend for *matplotlib*. Other frontends are not supported and may cause unexpected behavior.
- A *LaTeX* distribution (tested with *TeX Live*), since *LaTeX* is used by *matplotlib* for typesetting labels and mathematical expressions.
- *dvipng* for converting DVI files to PNG graphics

### 2.2 Installation notes (Linux)

Most of the above packages and programs can be installed through the package manager on most Linux distributions.

*kafe* was developed for use on Linux desktop systems. Please note that all commands below should be run as root.

#### 2.2.1 Install *NumPy*, *SciPy* and *matplotlib*

These packages should be available in the package manager.

In Ubuntu/Mint/Debian:

```
apt-get install python-numpy python-scipy python-matplotlib
```

In Fedora/RHEL/CentOS:

```
yum install numpy scipy python-matplotlib
```

## 2.2.2 Install *ROOT*

ROOT and its Python bindings can be obtained via the package manager in Ubuntu/Mint/Debian:

```
apt-get install root-system libroot-bindings-python5.34 libroot-bindings-python-dev
```

Or, in Fedora/RHEL/CentOS:

```
yum install root root-python
```

This setup is usually sufficient. However, you may decide to build ROOT yourself. In this case, be sure to compile with *PyROOT* support. Additionally, for Python to see the *PyROOT* bindings, the following environment variables have to be set correctly (:

```
export ROOTSYS=<directory where ROOT is installed>
export LD_LIBRARY_PATH=$ROOTSYS/lib:$PYTHONDIR/lib:$LD_LIBRARY_PATH
export PYTHONPATH=$ROOTSYS/lib:$PYTHONPATH
```

For more info, refer to <http://root.cern.ch/drupal/content/pyroot>.

## 2.2.3 Install *iminuit*

*iminuit* is a Python wrapper for the Minuit minimizer which is independent of ROOT. If compiling/installing ROOT is not possible, this minimizer can be used instead.

To install the *iminuit* package for Python, the [Pip installer](#) is recommended:

```
pip install iminuit
```

If you don't have *Pip* installed, get it from the package manager.

In Ubuntu/Mint/Debian, do:

```
apt-get install python-pip
```

In Fedora/RHEL/CentOS, do:

```
yum install python-pip
```

or use *easy\_install* (included with [setuptools](#)):

```
easy_install pip
```

You might also need to install the Python headers for *iminuit* to compile properly.

In Ubuntu/Mint/Debian, do:

```
apt-get install libpython2.7-dev
```

In Fedora/RHEL/CentOS, do:

```
yum install python-devel
```

Read the README file for more information on other dependencies (there should be adequate packages for your Linux distribution to satisfy these).

## 2.2.4 Install *kafe*

To install *kafe* using *Pip*, simply run the helper script as root:

```
./install.sh
```

To remove *kafe* using *Pip*, just run the helper script:

```
./uninstall.sh
```

Alternatively, installing using Python’s *setuptools* also works, but may not provide a clean uninstall. Use this method if installing with *Pip* is not possible:

```
python setup.py install
```

## 2.3 Installation notes (Windows)

*kafe* can be installed under Windows, but requires some additional configuration.

The recommended Python distribution for working with *kafe* under Windows is [WinPython](#), which has the advantage that it is portable and comes with a number of useful pre-installed packages. Particularly, *NumPy*, *SciPy* and *matplotlib* are all pre-installed in *WinPython*, as are all *Qt*-related dependencies.

Be sure to install *WinPython* version 2.7, since *kafe* does not currently run under Python 3.

### 2.3.1 Install *iminuit*

After installing *WinPython*, start ‘WinPython Command Prompt.exe’ in the *WinPython* installation directory and run

```
pip install iminuit
```

### 2.3.2 Install *kafe*

Now *kafe* can be installed from PyPI by running:

```
pip install kafe
```

Alternatively, it may be installed directly using *setuptools*. Just run the following in ‘WinPython Command Prompt.exe’ after switching to the directory into which you have downloaded *kafe*:

```
python setup.py install
```

### 2.3.3 Using *kafe* with ROOT under Windows

If you want *kafe* to work with ROOT’s *TMinuit* instead of using *iminuit*, then ROOT has to be installed. Please note that ROOT releases for Windows are 32-bit and using the PyROOT bindings on a 64-bit *WinPython* distribution will not work.

A pre-built version of ROOT for Windows is available on the ROOT homepage as a Windows Installer package. The recommended version is [ROOT 5.34](#). During the installation process, select “Add ROOT to the system PATH for all users” when prompted. This will set the `PATH` environment variable to include the relevant ROOT directories. The installer also sets the `ROOTSYS` environment variable, which points to the directory where ROOT is installed. By default, this is `C:\root_v5.34.34`.

Additionally, for Python to find the *PyROOT* bindings, the `PYTHONPATH` environment variable must be modified to include the `bin` subdirectory of path where ROOT is installed. On Windows 10, assuming ROOT has been installed in the default directory (`C:\root_v5.34.34`), this is achieved as follows:

1. open the Start Menu and start typing “environment variables”
2. select “Edit the system environment variables”
3. click the “Environment Variables...” button
4. in the lower part, under “System variables”, look for the “PYTHONPATH” entry

5. modify/add the “PYTHONPATH” entry:

- if it doesn’t exist, create it by choosing “New...”, enter PYTHONPATH as the variable name and `C:\root_v5.34.34\bin` as the variable value
- if it already exists and contains only one path, edit it via “Edit...” and insert `C:\root_v5.34.34\bin;` at the beginning of the variable value. (Note the semicolon!)
- if the variable already contains several paths, choosing “Edit...” will show a dialog box to manage them. Choose “New” and write `C:\root_v5.34.34\bin`

6. close all opened dialogs with “OK”

Now you may try to `import ROOT` in the *WinPython* interpreter to check if everything has been set up correctly.

For more information please refer to ROOT’s official [PyROOT Guide](#).

## FIT EXAMPLES, UTILITIES, TIPS AND TRICKS

A wide range of applications of the *kafe* core and the usage of the helper functions is exemplified below. All of them are contained in the sub-directory `examples/` of the *kafe* distribution and are intended to serve as a basis for user projects.

### 3.1 Example 1 - model comparison

To decide whether a model is adequate to describe a given set of data, typically several models have to be fit to the same data. Here is the code for a comparison of a data set to two models, namely a linear and an exponential function:

```
# import everything we need from kafe
from kafe import *
# additionally, import the two model functions we want to fit:
from kafe.function_library import linear_2par, exp_2par

#####
# Load the Dataset from the file
my_dataset = Dataset(input_file='dataset.dat', title="Example Dataset")
### Create the Fits
my_fits = [Fit(my_dataset, exp_2par),
           Fit(my_dataset, linear_2par)]
### Do the Fits
for fit in my_fits:
    fit.do_fit()
### Create the plots, save and show output
my_plot = Plot(my_fits[0], my_fits[1])
my_plot.plot_all(show_data_for=0) # show data only once (it's the same data)
my_plot.save('plot.pdf')
my_plot.show()
```

The file *dataset.dat* contains x and y data in the standard *kafe* data format, where values and errors (and optionally also correlation coefficients) are given for each axis separately. `#` indicates a comment line, which is ignored when reading the data:

```
# axis 0: x
# datapoints uncor. err.
0.957426 3.0e-01
2.262212 3.0e-01
3.061167 3.0e-01
3.607280 3.0e-01
4.933100 3.0e-01
5.992332 3.0e-01
7.021234 3.0e-01
8.272489 3.0e-01
9.250817 3.0e-01
9.757758 3.0e-01
```

```
# axis 1: y
# datapoints uncor. err.
1.672481 2.0e-01
1.743410 2.0e-01
1.805217 2.0e-01
2.147802 2.0e-01
2.679615 2.0e-01
3.110055 2.0e-01
3.723173 2.0e-01
4.430122 2.0e-01
4.944116 2.0e-01
5.698063 2.0e-01
```

The resulting output is shown below. As can be seen already from the graph, the exponential model better describes the data. The  $\chi^2$  probability in the printed output shows, however, that the linear model would be marginally acceptable as well:

```
linear_2par
chi2prob 0.052
HYPTTEST accepted (CL 5%)

exp_2par
chi2prob 0.96
HYPTTEST accepted (CL 5%)
```

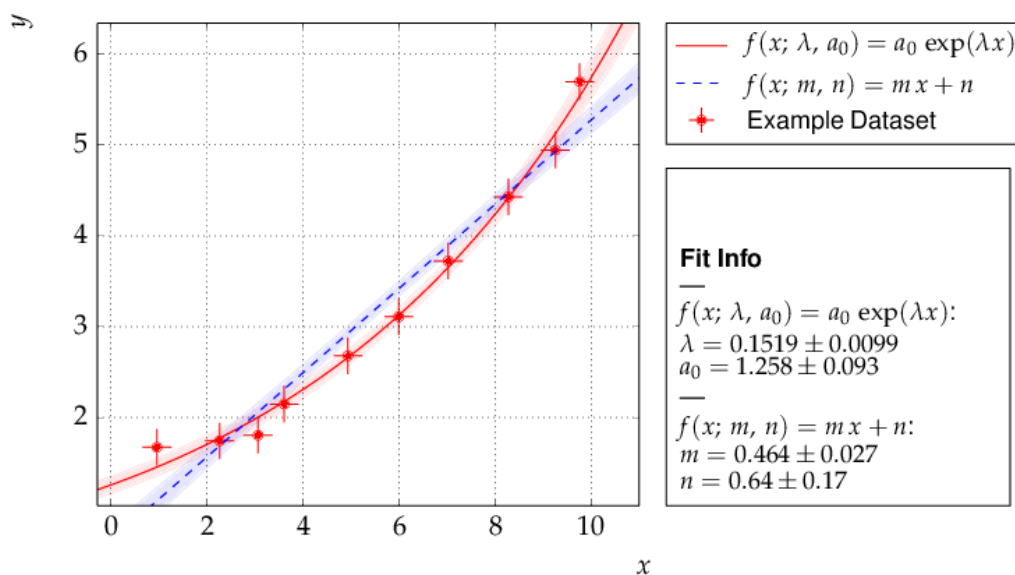


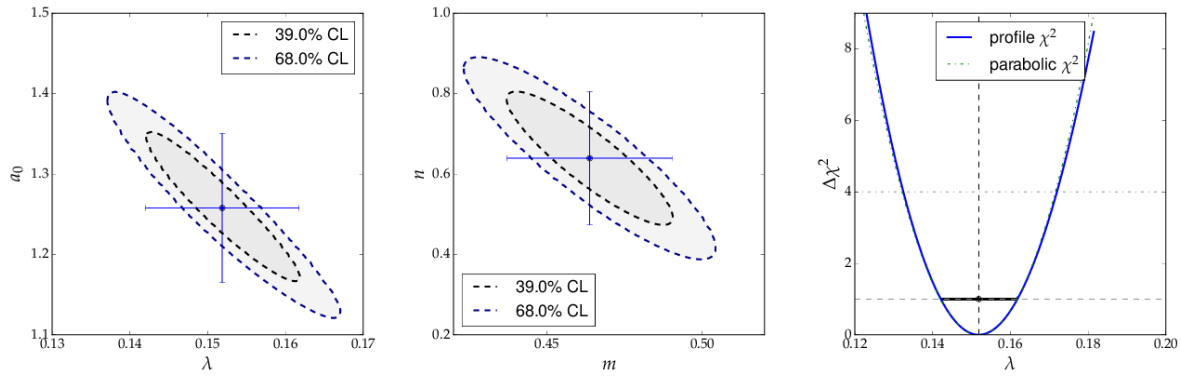
Fig. 3.1: Output of example1 - compare two models

The contour curves of the two fits are shown below and reflect the large correlations between the fit parameters. The right plot of the profile  $\chi^2$  curve shows that there is a slight deviation from the parabolic curve in the first fit of a non-linear (exponential) function. For more details on the profiled  $\chi^2$  curve see the discussion of example 3, where the difference is more prominent.

## 3.2 Example 2 - two fits and models

Another typical use case consists of comparing two sets of measurements and the models derived from them. This is very similar to the previous example with minor modifications:




 Fig. 3.2: Contour curves and a profile  $\chi^2$  curve for the fits in example 1

```

...
#####
# Workflow #
#####
# Load two Datasets from files
my_datasets = [Dataset(input_file='dataset1.dat', title="Example Dataset 1"),
               Dataset(input_file='dataset2.dat', title="Example Dataset 2")]
# Create the Fits
...
# Do the Fits
...
# Create the plots
my_plot.plot_all() # this time without any arguments, i.e. show everything
...
    
```

This results in the following output:

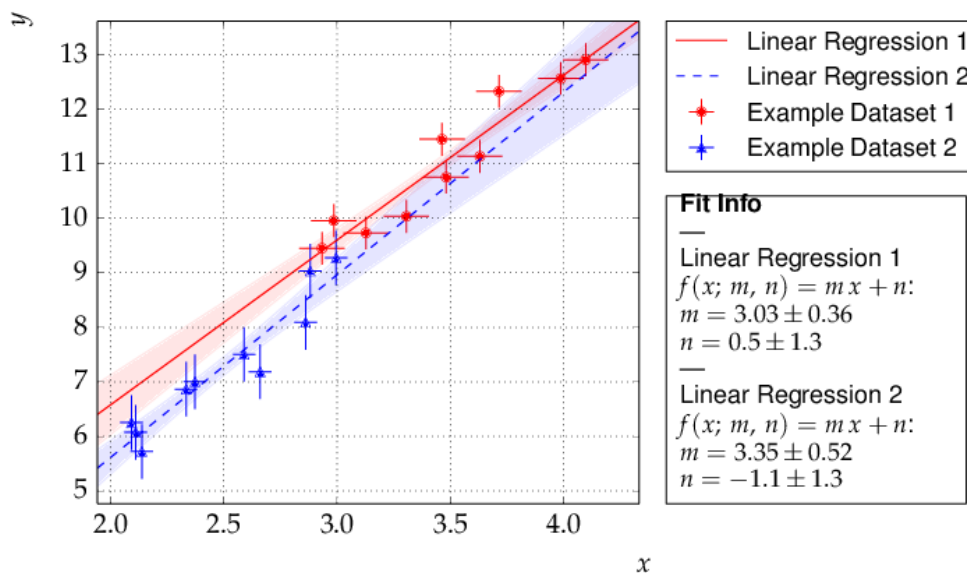


Fig. 3.3: Output of example2 - comparison of two linear fits.

Although the parameters extracted from the two data sets agree within errors, the uncertainty bands of the two functions do not overlap in the region where the data of Dataset 2 are located, so the data are most probably

incompatible with the assumption of an underlying single linear model.

### 3.3 Example 3 - non-linear fit with non-parabolic errors

Very often, when the fit model is a non-linear function of the parameters, the  $\chi^2$  function is not parabolic around the minimum. A very common example of such a case is an exponential function parametrised as shown in the code fragment below. *Minuit* contains a spacial algorithm, *Minos*, which returns correct errors also in this case. Instead of using the curvature the minimum, *Minos* follows the  $\chi^2$  function from the minimum to the point where it crosses the value *minimum+up*, where *up=1* corresponds to one standard deviation in  $\chi^2$  fits. During the scan of the  $\chi^2$  function at different values of each parameter the minimum with respect to all other parameters in the fit is determined, thus making sure that all correlations among the parameters are taken into account. In case of a parabolic  $\chi^2$  function, the *Minos* errors are identical to those obtained by the *Hesse* algorithm, but are typically larger or asymmetric in other cases.

The method `kafe.do_fit()` executes the *Minos* algorithm after completion of a fit and prints the *Minos* errors if the deviation from the parabolic result are larger than 5% .

A graphical visualisation is provided by the method `plot_profile()` , which displays the profile  $\chi^2$  curve for the parameter with name or index passed as an argument to the method.

The relevant code fragments and the usage of the method `kafe.fit.plot_profile()` are illustrated here:

```
...
# definition of the fit function
@ASCII(x_name="t", expression="A0*exp(-t/tau)")
# Set some LaTeX-related parameters for this function
@LaTeX(name='A', x_name="t",
        parameter_names=('A_0', '\\tau{}'),
        expression="A_0\\,\\exp(\\frac{-t}{\\tau})")
@FitFunction
def exponential(t, A0=1, tau=1):
    return A0 * exp(-t/tau)
...
# Load the data, perform fit and plot
my_dataset = Dataset(input_file='dataset.dat', title="Example Dataset")
my_fit = Fit(my_dataset, exponential)
my_fit.do_fit()
my_plot = Plot(my_fit)
my_plot.plot_all()
# --> display contours and profile
contour = my_fit.plot_contour(0, 1, dchi2=[1.,2.3])
profile1=my_fit.plot_profile(0)
profile2=my_fit.plot_profile(1)
# Show the plots
my_plot.show()
```

The data points were generated using a normalisation factor of  $A0=1$ . and a lifetime  $\tau=1$ .. The resulting fit output below demonstrates that this is well reproduced within uncertainties:

The contour  $A_0$  vs  $\tau$ , however, is not an ellipse, as shown in the figure below. The profiled  $\chi^2$  curves are also shown; they deviate significantly from parabolas. The proper one-sigma uncertainty in the sense of a 68% confidence interval is read from these curves by determining the parameter values where the  $\chi^2$  curves cross the horizontal lines at a value of  $\Delta\chi^2=1$  above the minimum. The two-sigma uncertainties correspond to the intersections with the horizontal line at  $\Delta\chi^2=4$ .

Note: a more parabolic behaviour is achieved by using the width parameter  $\lambda=1/\tau$  in the parametrisation of the exponential function.

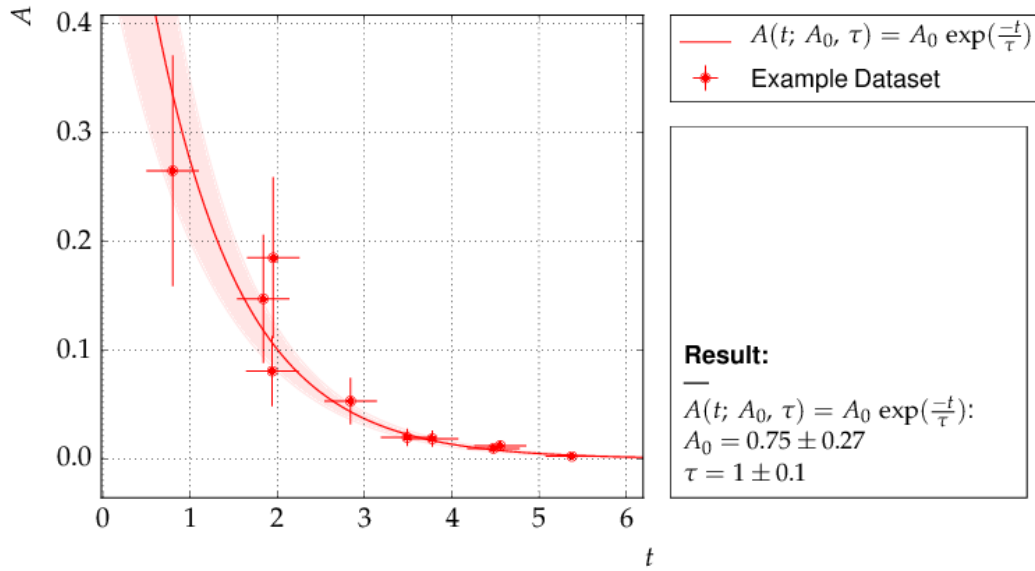
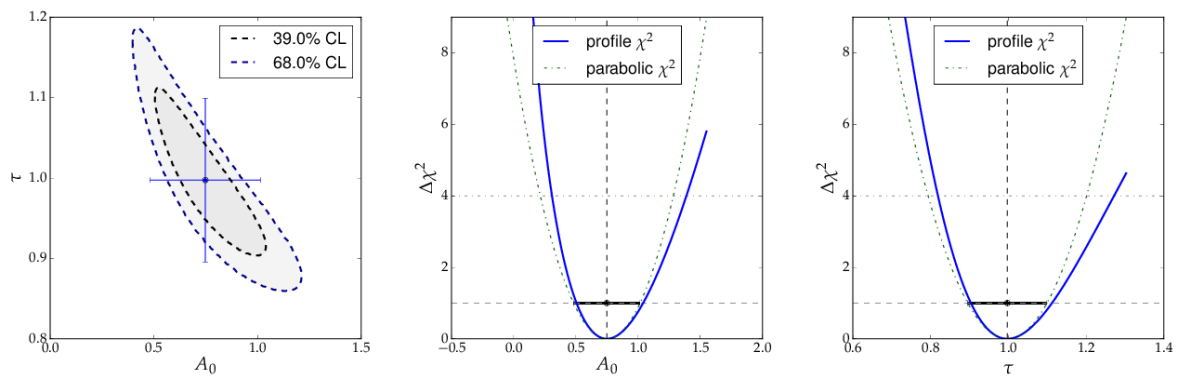


Fig. 3.4: Output of example 3 - Fit of an exponential


 Fig. 3.5: Contour and profile  $\chi^2$  curves of example 3

### 3.4 Example 4 - average of correlated measurements

The average of a set of measurements can be considered as a fit of a constant to a set of input data. This example illustrates how correlated errors are handled in *kafe*. Measurements can have a common error, which may be absolute or relative, i. e. depend on the input value. In more complicated cases the full covariance matrix must be constructed.

*kafe* has a helper function, `build_dataset()` in module *fit* (page 33), which aids in setting up the covariance matrix and transforming the input to the default format used by the *Dataset* (page 28) and *Fit* (page 33) classes. Two further helper functions in module *file\_tools* (page 48) aid in reading the appropriate information from data files.

1. The function `parse_column_data()` (page 48) reads the input values and their independent errors from one file, and optionally covariance matrices for the x and y axes from additional files. The field ordering is defined by a control string.
2. Another helper function, `buildDataset_fromFile()` (page 48), specifies input values or blocks of input data from a single file with keywords.

The second version needs only very minimal additional user code, as illustrated here:

```
from kafe import *
from kafe.function_library import constant_lpar
from kafe.file_tools import buildDataset_fromFile
#
# =====
fname = 'WData.dat'
curDataset = buildDataset_fromFile(fname) # Dataset from input file
curFit = Fit(curDataset, constant_lpar)   # set up the fit object
curFit.do_fit()

myPlot = Plot(curFit)
myPlot.plot_all()
myPlot.save("plot.pdf")
myPlot.show()
```

The input file is necessarily more complicated, but holds the full information on the data set in one place. Refer to the documentation of the function `parse_general_inputfile()` (page 49) in module *file\_tools* (page 48) for a full description of the currently implemented keywords. The input file for the averaging example is here:

```
# Measurements of W boson mass (combined LEP2, 2013)
# =====
# example to use parse_general_inputfile from kafe;
# covariance matrix build from common errors
# ==
# Meta data for plotting
*TITLE measurements of the W boson mass
*xLabel number of measurement
*yLabel  $m_W$ 
*yUnit GeV/c2

# x data need not be given for averaging

# =====
# Measurements of W mass by ALEPH, DELPHI, L3 and OPAL
# from from LEP2 Report Feb. 2013
# common errors within channels
# 2q2l: 0.021 GeV,
# 4q: 0.044 GeV,
# and between channels: 0.025 GeV
# =====
```

```

*yData_SCOV
# W_mass err      syst      sqrt of the off-diagonal
# 2q2l channel          elements of the
80.429  0.055  0.021      #          covariance matrix
80.339  0.073  0.021  0.021
80.217  0.068  0.021  0.021  0.021
80.449  0.058  0.021  0.021  0.021  0.021
# 4q channel
80.477  0.069  0.044  0.025  0.025  0.025  0.025  0.044
80.310  0.091  0.044  0.025  0.025  0.025  0.025  0.044  0.044
80.324  0.078  0.044  0.025  0.025  0.025  0.025  0.044  0.044  0.044
80.353  0.068  0.044  0.025  0.025  0.025  0.025  0.044  0.044  0.044  0.044

```

### 3.5 Example 5 - non-linear multi-parameter fit (damped oscillation)

This example shows the fitting of a more complicated model function to data collected from a damped harmonic oscillator. In such non-linear fits, setting the initial values is sometimes crucial to let the fit converge at the global minimum. The *Fit* (page 33) object provides the method *set\_parameters()* (page 38) for this purpose. As the fit function for this problem is not a standard one, it is defined explicitly making use of the decorator functions available in *kafe* to provide nice type setting of the parameters. This time, the function *parse\_column\_data()* (page 48) is used to read the input, which is given as separate columns with the fields

```
<time> <Amplitude> <error on time> <error on Amplitude>
```

Here is the example code:

```

...
from kafe import *
from numpy import exp, cos
# Model function definition #
# =====
# Set an ASCII expression for this function
@ASCII(x_name="t", expression="A0*exp(-t/tau)*cos(omega*t+phi)")
# Set some LaTeX-related parameters for this function
@LaTeX(name='A', x_name="t",
        parameter_names=('a_0', '\\tau', '\\omega', '\\varphi'),
        expression="a_0\\exp(-\\frac{t}{\\tau})\\cos(\\omega t+\\varphi)")
@FitFunction
def damped_oscillator(t, a0=1, tau=1, omega=1, phi=0):
    return a0 * exp(-t/tau) * cos(omega*t + phi)

# ---- Workflow #
# load the experimental data from a file
my_dataset = parse_column_data('damped_oscillation.dat',
    field_order="x,y,xabserr,yabserr", title="Damped Oscillator",
    axis_labels=['Time t', 'Amplitude'])
# --- Create the Fit
my_fit = Fit(my_dataset, damped_oscillator)
# Set the initial values for the fit:
#           a_0 tau omega phi
my_fit.set_parameters((1., 2., 6.28, 0.8))
my_fit.do_fit()
# --- Create and output the plots
my_plot = Plot(my_fit)
my_plot.plot_all()
#my_plot.save('plot.pdf')
my_fit.plot_correlations() # all contours and profiles
my_plot.show()

```

This is the resulting output:

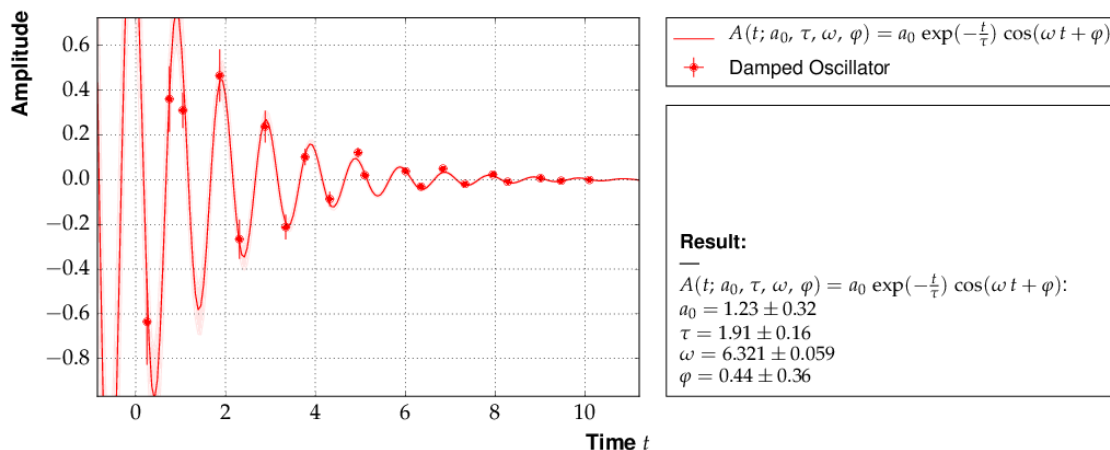


Fig. 3.6: Example 5 - fit of the time dependence of the amplitude of a damped harmonic oscillator.

The fit function is non-linear, and, furthermore, there is not a single local minimum - e.g. a shift in phase of  $180^\circ$  corresponds to a change in sign of the amplitude, and valid solutions are also obtained for multiples of the base frequency. Checking of the validity of the fit result is therefore important. The method `plot_correlations()` (page 37) provides the contours of all pairs of parameters and the profiles for each of the parameters and displays them in a matrix-like arrangement. Distorted contour-ellipses show whether the result is affected by near-by minima, and the profiles allow to correctly assign the parameter uncertainties in cases where the parabolic approximation is not precise enough.

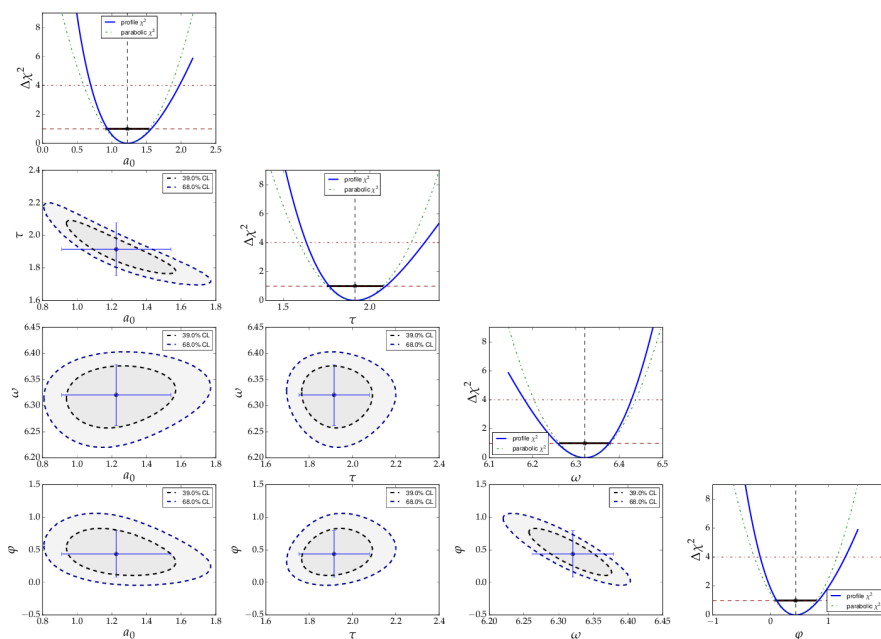


Fig. 3.7: Confidence contours and profiles for example 5.

### 3.6 Example 6 - linear multi-parameter fit

This example is not much different from the previous one, except that the fit function, a standard fourth-degree polynomial from the module `function_library` (page 55), is modified to reflect the names of the problem given, and `matplotlib` functionality is used to influence the output of the plot, e.g. axis names and linear or

logarithmic scale.

It is also shown how to circumvent a problem that often arises when errors depend on the measured values. For a counting rate, the (statistical) error is typically estimated as the square root of the (observed) number of entries in each bin. For large numbers of entries, this is not a problem, but for small numbers, the correlation between the observed number of entries and the error derived from it leads to a bias when fitting functions to the data. This problem can be avoided by iterating the fit procedure:

In a pre-fit, a first approximation of the model function is determined, which is then used to calculate the expected errors, and the original errors are replaced before performing the final fit. Note that the numbers of entries in the bins must be sufficiently large to justify a replacement of the (asymmetric) Poisson uncertainties by the symmetric uncertainties implied by the  $\chi^2$ -method.

The implementation of this procedure needs accesses some more fundamental methods of the *Dataset*, *Fit* and *FitFunction* classes. The code shown below demonstrates how this can be done with *kafe*, using some of its lower-level, internal interfaces:

```
from kafe.function_library import poly4
# modify function's independent variable name to reflect its nature:
poly4.x_name = 'x=cos(t)'
poly4.latex_x_name = 'x=\\cos(\\theta)'
...

# Set the axis labels appropriately
my_plot.axis_labels = ['$\\cos(\\theta)$', 'counting rate']
...
# load the experimental data from a file
my_dataset = parse_column_data(
    'counting_rate.dat',
    field_order="x,y,yabserr",
    title="Counting Rate per Angle")

### pre-fit
# error for bins with zero contents is set to 1.
covmat = my_dataset.get_cov_mat('y')
for i in range(0, len(covmat)):
    if covmat[i, i] == 0.:
        covmat[i, i] = 1.
my_dataset.set_cov_mat('y', covmat) # write it back

# Create the Fit
my_fit = Fit(my_dataset, poly4)
#         fit_label="Linear Regression " + dataset.data_label[-1])

# perform an initial fit with temporary errors (minimal output)
my_fit.call_minimizer(final_fit=False, verbose=False)

# set errors using model at pre-fit parameter values:
#         sigma_i^2=cov[i, i]=n(x_i)
fdata = my_fit.fit_function.evaluate(my_fit.xdata,
                                     my_fit.current_parameter_values)
np.fill_diagonal(covmat, fdata)
my_fit.current_cov_mat = covmat # write new covariance matrix
### end pre-fit - rest is as usual
my_fit.do_fit()
# Create the plots and ==
my_plot = Plot(my_fit)
# -- set the axis labels
my_plot.axis_labels = ['$\\cos(\\theta)$', 'counting rate']
# -- set scale linear / log
my_plot.axes.set_yscale('linear')
...
```

This is the resulting output:

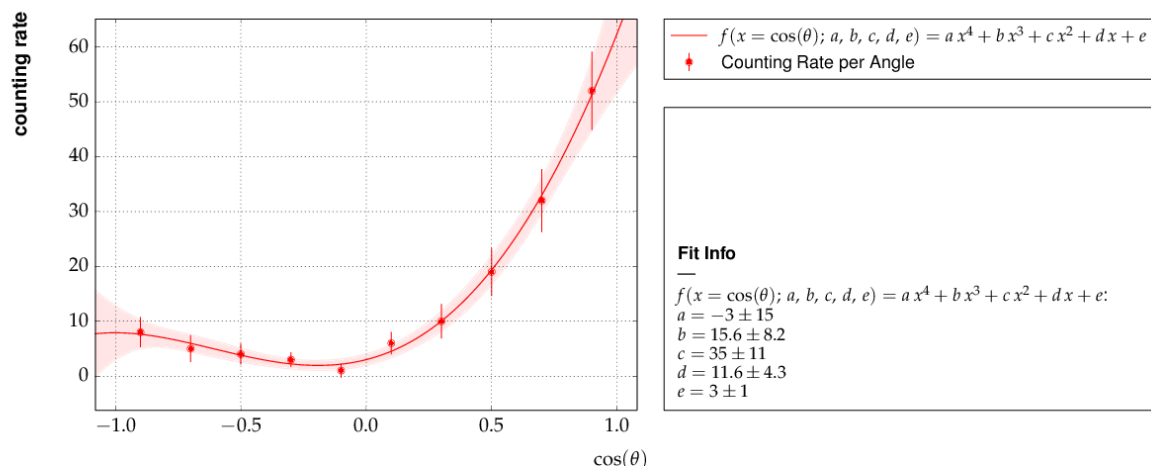


Fig. 3.8: Output of example 6 - counting rate.

### 3.7 Example 7 - another non-linear multi-parameter fit (double-slit spectrum)

Again, not much new in this example, except that the model is now very non-linear, the intensity distribution of light after passing through a double-slit. The non-standard model definition again makes use of the decorator mechanism to provide nice output - the decorators (expressions beginning with '@') can safely be omitted if *LaTeX* output is not needed. Setting of appropriate initial conditions is absolutely mandatory for this example, because there exist many local minima of the  $\chi^2$  function.

Another problem becomes obvious when carefully inspecting the fit function definition: only two of the three parameters  $g$ ,  $b$  or  $k$  can be determined, and therefore one must be kept fixed, or an external constraint must be applied. Failing to do so will result in large, correlated errors on the parameters  $g$ ,  $b$  and  $k$  as an indication of the problem.

Fixing parameters of a model function is achieved by the method `fix_parameters()` (page 35), and a constraint within a given uncertainty is achieved by the method `constrain_parameters()` (page 34) of the `Fit` (page 33) class.

Here are the interesting pieces of code:

```
...
# Model function definition #
# Set an ASCII expression for this function
@ASCII(x_name="x", expression="I0*(sin(k/2*b*sin(x))/(k/2*b*sin(x))"
      "*cos(k/2*g*sin(x))^2")
# Set some LaTeX-related parameters for this function
@LaTeX(name='I', x_name="\\alpha{}",
      parameter_names=('I_0', 'b', 'g', 'k'),
      expression="I_0\\,\\left(\\frac{\\sin(\\frac{k}{2}\\,b\\,\\sin{\\alpha})}{\\frac{k}{2}\\,b\\,\\sin{\\alpha}}\\right)"
      "\\cos(\\frac{k}{2}\\,g\\,\\sin{\\alpha})^2")
@FitFunction
def double_slit(alpha, I0=1, b=10e-6, g=20e-6, k=1.e7):
    k_half_sine_alpha = k/2*sin(alpha) # helper variable
    k_b = k_half_sine_alpha * b
    k_g = k_half_sine_alpha * g
    return I0 * (sin(k_b)/(k_b) * cos(k_g))**2
...
# Set the initial values for the fit
```



```
#           I      b      g      k
my_fit.set_parameters((1., 20e-6, 50e-6, 9.67e6))
# fix one of the (redundant) parameters, here 'k'
my_fit.fix_parameters('k')
...
```

If the parameter  $k$  in the example above has a (known) uncertainty, it is more appropriate to constrain it within its uncertainty (which may be known from an independent measurement or from the specifications of the laser used in the experiment). To take into account a wave number  $k$  known with a precision of  $10^4$  the last line in the example above should be replaced by:

```
...
my_fit.constrain_parameters(['k'], [9.67e6], [1.e4])
...
```

This is the resulting output:

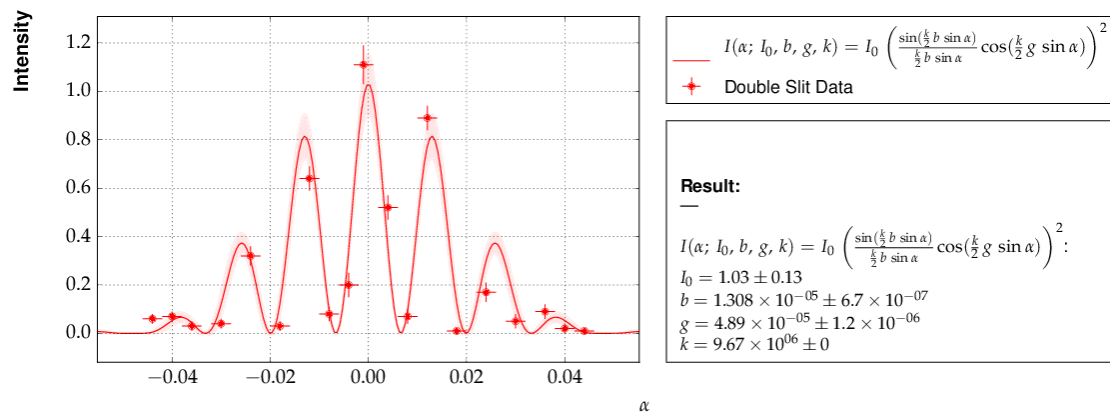


Fig. 3.9: Example 7 - fit of the intensity distribution of light behind a double slit with fixed or constrained wave length.

### 3.8 Example 8 - fit of a Breit-Wigner resonance to data with correlated errors

This example illustrates how to define the data and the fit function in a single file - provided by the helper function `buildFit_fromFile()` (page 48) in module `file_tools` (page 48). Parsing of the input file is done by the function `parse_general_inputfile()` (page 49), which had already been introduced in Example 4. The definition of the fit function as *Python* code including the *kafe* decorators in the input file, however, is new. Note: because spaces are used to separate data fields in the input file, spaces needed for proper *Python* indentation have to be replaced by '~'. The last key in the file defines the start values of the parameters and their initial ranges.

The advantage of this approach is the location of all data and the fit model in one place, which is strictly separated from the *Python* code. The *Python* code below is thus very general and can handle a large large variety of problems without modification (except for the file name, which could easily be passed on the command line):

```
from kafe import *
from kafe.file_tools import buildFit_fromFile
# -----
fname = 'LEP-Data.dat'
# initialize fit object from file
BWfit = buildFit_fromFile(fname)
BWfit.do_fit()
```

```
#
BWplot = Plot(BWfit)
BWplot.plot_all()
BWplot.save("plot.pdf")
BWplot.show()
```

The magic happens in the input file, which now has to provide all the information needed to perform the fit:

```
# Measurements of hadronic Z cross sections at LEP
# -----
# this file is to be parsed with
#         kafe.file_tools.buildFit_fromFile()

# Meta data for plotting
*TITLE   LEP Hadronic Cross Section ( $\sigma^0_{\mathrm{had}}$ )
*Basename example8_BreitWigner
*xLabel   $E_{\mathrm{CM}}$ 
*xUnit    $\mathrm{GeV}$ 
*yLabel   $\sigma^0_{\mathrm{had}}$ 
*yUnit    $\mathrm{nb}$ 

#-----
# DATA: average of hadronic cross sections measured by
# ALEPH, DELPHI, L3 and OPAL around 7 energy points at the Z resonance
#-----

# CMenergy E err
*xData
  88.387  0.005
  89.437  0.0015
  90.223  0.005
  91.238  0.003
  92.059  0.005
  93.004  0.0015
  93.916  0.005

# Centre-of-mass energy has a common uncertainty
*xAbsCor 0.0017

# sig^0_h  sig err      #  rad.cor  sig_h measured
*yData
  6.803  0.036      #  1.7915    5.0114
  13.965  0.013      #  4.0213    9.9442
  26.113  0.075      #  7.867     18.2460
  41.364  0.010      #  10.8617   30.5022
  27.535  0.088      #  3.9164    23.6187
  13.362  0.015      # -0.6933    14.0552
  7.302   0.045      # -1.8181     9.1196

# cross-sections have a common relative error
*yRelCor 0.0007

*FITLABEL Breit-Wigner-Fit \, {\large{(with~s-dependent~width)}}

*FitFunction
# Breit-Wigner with s-dependent width
@ASCII(name='sigma', expression='s0*x^2*G^2/[ (E^2-M^2)^2+(E^4*G^2/M^2) ]')
@LaTeX(name='\sigma', parameter_names=('\\sigma^0', 'M_Z', '\\Gamma_Z'),
expression='\\frac{\\sigma^0}{M_Z^2\\Gamma_Z^2} \\{ ((E^2-M_Z^2)^2+(E^4\\Gamma_Z^2 / M_Z^2)) \\}')
@FitFunction
def BreitWigner(E, s0=41.0, M=91.2, G=2.5):
    return s0*E*E*G*G/((E*E-M*M)**2+(E**4*G*G/(M*M)))
```

```

*InitialParameters # initial values and range
41.  0.5
91.2 0.1
2.5  0.1

#-----
#### official results (LEP Electroweak Working Group):
#### s0=41.540+/-0.037nb  M=91.1875+/-0.0021GeV/c^2  G=2.4952+/-0.0023 GeV
#### uses all decay modes of the Z and full cross-section list
#-----

```

Here is the output:

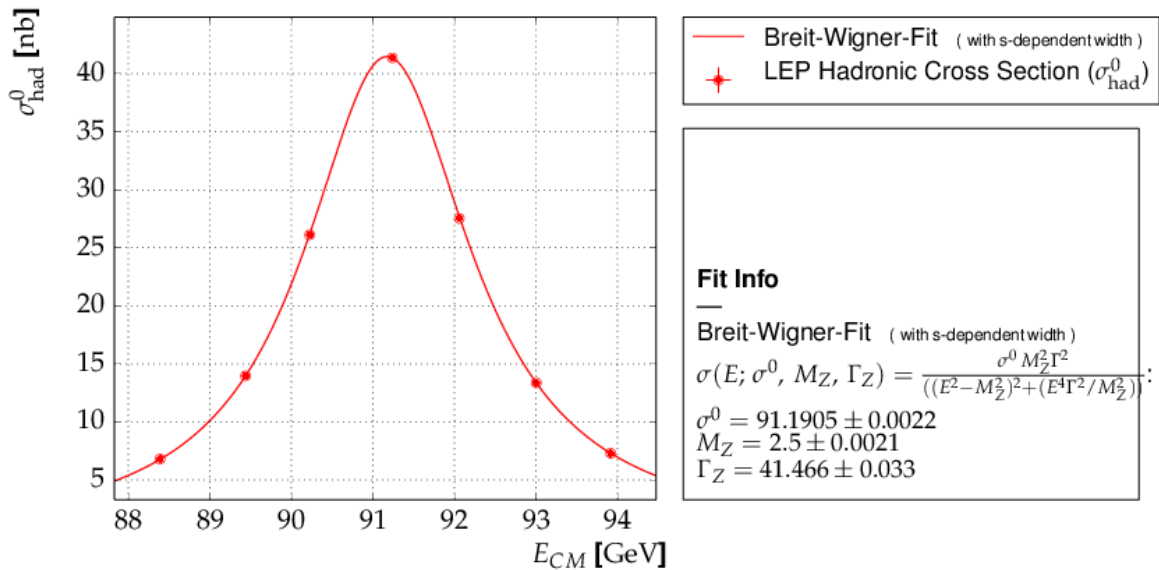


Fig. 3.10: Output of example 8 - Fit of a Breit-Wigner function.

This example also contains a code snippet demonstrating how to plot contours by calling the `Fit` (page 33) object's `plot_contour()` (page 36) method. This is the code:

```

# plot pairs of contours at 1 sigma, 68%, 2 sigma and 95%
cont_fig1 = BWfit.plot_contour(0, 1, dchi2=[1.,2.3,4.,5.99])
cont_fig2 = BWfit.plot_contour(0, 2, dchi2=[1.,2.3,4.,5.99])
cont_fig3 = BWfit.plot_contour(1, 2, dchi2=[1.,2.3,4.,5.99])
# save to files
cont_fig1.savefig("kafe_BreitWignerFit_contour12.pdf")
cont_fig2.savefig("kafe_BreitWignerFit_contour13.pdf")
cont_fig3.savefig("kafe_BreitWignerFit_contour23.pdf")

```

The resulting pictures show that parameter correlations are relatively small:

### 3.9 Example 9 - fit of a function to histogram data

This example brings us to the limit of what is currently possible with *kafe*. Here, the data represent the center of a histogram bins and the number of entries,  $n_i$ , in each bin. The (statistical) error is typically estimated as the square root of the (observed) number of entries in each bin. For large numbers of entries, this is not a problem, but for small numbers, especially for bins with 0 entries, the correlation between the observed number of entries and the error derived from it leads to a bias when fitting functions to the histogram data. In particular, bins with

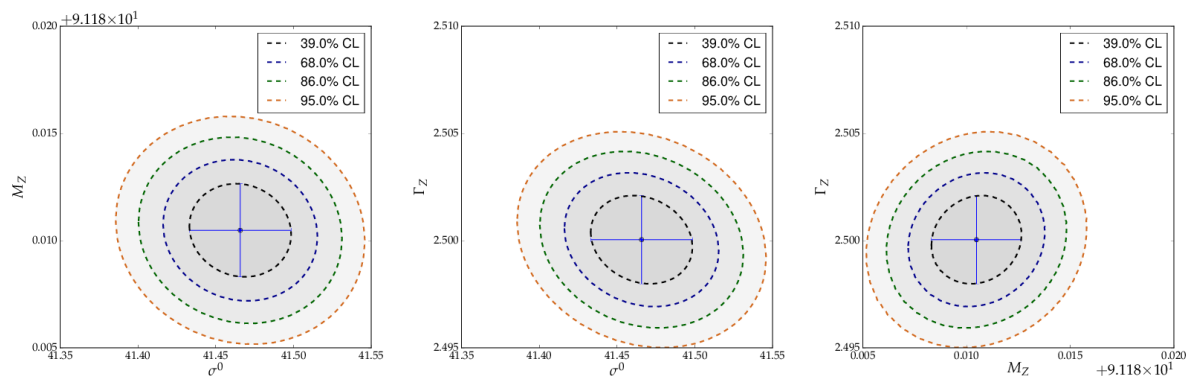


Fig. 3.11: Contours generated in example 8 - Fit of a Breit-Wigner function.

zero entries cannot be handled in the  $\chi^2$ -function, and are typically omitted to cure the problem. However, a bias remains, as bins with downward fluctuations of the observed numbers of events get assigned smaller errors and hence larger weights in the fitting procedure - leading to the aforementioned bias.

These problems are avoided by using a likelihood method for such use cases, where the Poisson distribution of the uncertainties and their dependence on the values of the fit model is properly taken into account. However, the  $\chi^2$ -method can be saved to some extent if the fitting procedure is iterated. In a pre-fit, a first approximation of the model function is determined, where the error in bins with zero entries is set to one. The model function determined from the pre-fit is then used to calculate the expected errors for each bin, and the original errors are replaced before performing the final fit. Note that the numbers of entries in the bins must be sufficiently large to justify a replacement of the (asymmetric) Poisson uncertainties by the symmetric uncertainties implied by the  $\chi^2$ -method.

The code shown below demonstrates how to get a grip on such more complex procedures with more fundamental methods of the *Dataset*, *Fit* and *FitFunction* classes:

```
...
# Load Dataset from file
hdataset = Dataset(input_file='hdataset.dat', title="Data for example 9")

# error for bins with zero contents is set to 1.
covmat = hdataset.get_cov_mat('y')
for i in range(0, len(covmat)):
    if covmat[i, i] == 0.:
        covmat[i, i] = 1.
hdataset.set_cov_mat('y', covmat) # write it back

# Create the Fit instance
hfit = Fit(hdataset, gauss, fit_label="Fit of a Gaussian to histogram data")
#
# perform an initial fit with temporary errors (minimal output)
hfit.call_minimizer(final_fit=False, verbose=False)
#
# re-set errors using model at pre-fit parameter values:
#     sigma_i^2=cov[i, i]=n(x_i)
fdata=hfit.fit_function.evaluate(hfit.xdata, hfit.current_parameter_values)
np.fill_diagonal(covmat, fdata)
hfit.current_cov_mat = covmat # write back new covariance matrix
#
# now do final fit with full output
hfit.do_fit()
# and create, draw, save and show plot
...
```

Here is the output, which shows that the parameters of the standard normal distribution, from which the data were generated, are reproduced well by the fit result:

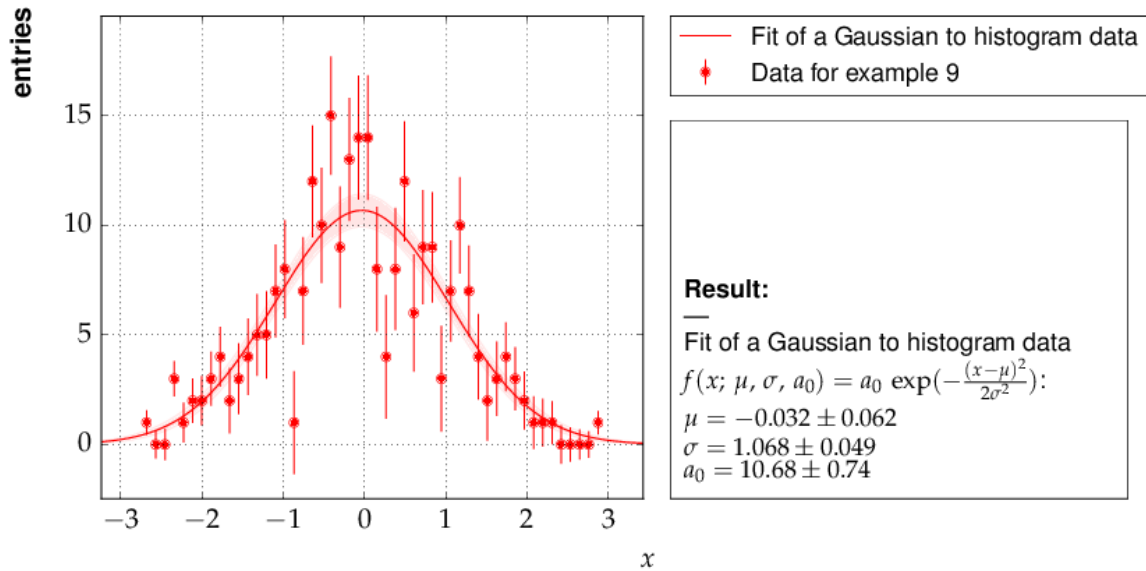


Fig. 3.12: Output of example 9 - Fit of a Gaussian distribution to histogram data

### 3.10 Example 10 - plotting with *kafe*: properties of a Gauss curve

This example shows how to access the *kafe* plot objects to annotate plots with `matplotlib` functionality.

A dummy object *Dataset* (page 28) is created with points lying exactly on a Gaussian curve. The *Fit* (page 33) will then converge toward that very same Gaussian. When plotting, the data points used to “support” the curve can be omitted.

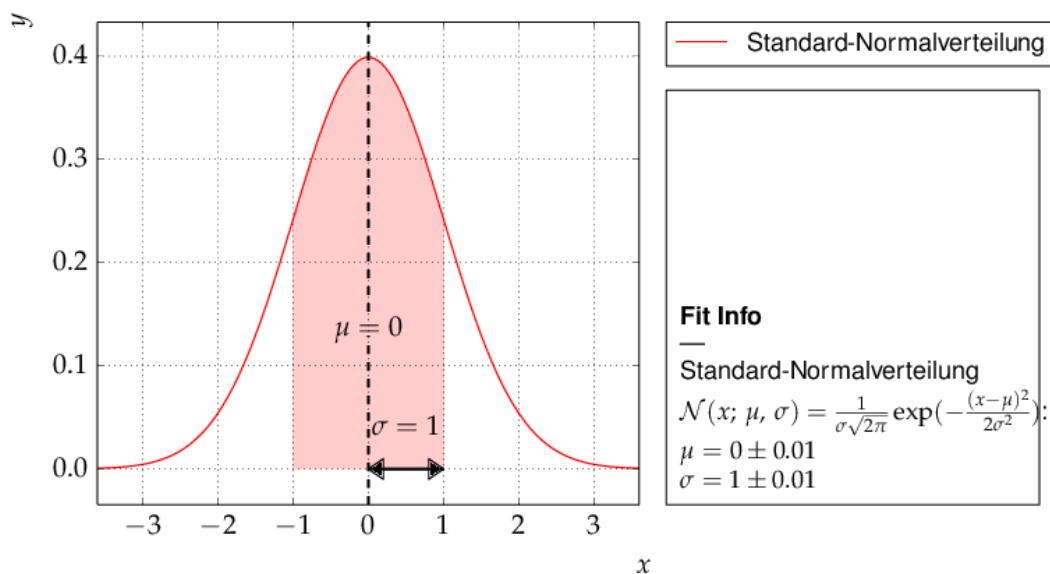


Fig. 3.13: Output of example 10 - properties of a Gauss curve.



## KAFE REFERENCE

This page contains documentation which was automatically extracted from docstrings in the source code. All major classes, methods and functions provided by *kafe* are documented here. For further information, or if in doubt about the exact functionality, users are invited to consult the source code itself.

### 4.1 *kafe* in a nutshell

**kafe** – 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 (errors) of the data are also stored in the *Dataset* as a list of one or more *ErrorSource* objects, each of which stores a part of the uncertainty information as a so-called *covariance matrix* (also called an *error matrix*). This allows **kafe** to work with uncertainties of different kinds for a *Dataset*, particularly when there is a degree of correlation between the uncertainties of the datapoints.

Fitting with **kafe** in a nutshell goes like this:

1. create a *Dataset* object from your measurement data

```
>>> my_d = kafe.Dataset(data=[[0., 1., 2.], [1.23, 3.45, 5.62]])
```

2. add errors (uncertainties) to your *Dataset*

```
>>> my_d.add_error_source('y', 'simple', 0.5) # y errors, all +/- 0.5
```

3. import a model function from *kafe.function\_library* (or define one yourself)

```
>>> from kafe.function_library import linear_2par
```

4. create a *Fit* object from your *Dataset* and your model function

```
>>> my_f = kafe.Fit(my_d, linear_2par)
```

5. do the fit

```
>>> my_f.do_fit()
```

6. (optional) if you want to see a plot of the result, use the *Plot* object

```
>>> my_p = kafe.Plot(my_f)
>>> my_p.plot_all()
>>> my_p.show()
```

For more in-depth information on **kafe**'s features, feel free to consult the documentation.

## 4.2 Main modules: dataset, fit and plot

### 4.2.1 Dataset: Collection of data points (kafe.dataset)

```
class kafe.dataset.Dataset (data=None, title='Untitled Dataset', axis_labels=['x', 'y'],
                             axis_units=['', ''], **kwargs)
```

Bases: object

The *Dataset* object is a data structure for storing measurement and error data.

It contains the measurement *data* as *NumPy* arrays and the error information as a list of *ErrorSource* (page 33) objects for each axis, each of which represents a separate contribution to the uncertainty of the measurement data, expressed as a *covariance matrix*.

The *Dataset* object calculates a *total covariance matrix* by adding up all the individual *ErrorSource* covariance matrices. This *total covariance matrix* is the one used for fitting.

A *Dataset* can be constructed directly from the measurement data, and can optionally be given a *title*, *axis labels* and *axis units*, as well as a *base name* for log or output files:

```
>>> my_d = kafe.Dataset(data=[[0., 1., 2.], [1.23, 3.45, 5.62]])
```

After constructing the *Dataset*, an error model may be added using *add\_error\_source()* (page 28) (here, an absolute y-uncertainty of 0.5):

```
>>> my_d.add_error_source('y', 'simple', 0.5) # y errors, all +/- 0.5
```

The *Dataset* may then be used for fitting. For more information, see the *Fit* (page 33) object documentation.

#### Keyword Arguments

- data** (*iterable, optional*) – the measurement data. Either of the form (xdata, ydata) or [(x1, y1), (x2, y2),... (xn, yn)]
- title** (*string, optional*) – the name of the *Dataset*. If omitted, the *Dataset* will be given the generic name 'Untitled Dataset'.
- axis\_labels** (*list of strings, optional*) – labels for the *x* and *y* axes. If omitted, these will be set to 'x' and 'y', respectively.
- axis\_units** (*list of strings, optional*) – units for the *x* and *y* axes. If omitted, these will be assumed to be dimensionless, i.e. the unit will be an empty string.
- basename** (*string*) – base name of files generated by this Dataset/subsequent Fits...

**add\_error\_source** (*axis, err\_type, err\_val, relative=False, correlated=False, recompute\_cov\_mat=True*)

Add an error source for the data. A *Dataset* can have many error sources for each axis, each corresponding to a covariance matrix. The total error model for the axis is represented by the sum of these matrices.

Note: whenever an *ErrorSource* is added, the total covariance matrix is (re-)calculated, unless *recompute\_cov\_mat* is *False*.

#### Parameters

- axis** ('x' or 'y') – axis for which to add error source.
- err\_type** ('simple' or 'matrix') – a 'simple' error source is constructed from a single float or a list of *N* floats (*N* being the size of the *Dataset*), representing the uncertainty of the corresponding data points.



A 'matrix' error source is a user-constructed covariance matrix.

- err\_val** (float/list of floats *or* `numpy.matrix`) – for a 'simple' error source, a float of a list of  $N$  floats ( $N$  being the size of the *Dataset*). The float/each float in the list represents the uncertainty of the corresponding data point.

For a 'matrix' error source, the user-constructed covariance matrix (type: `numpy.matrix`).

#### Keyword Arguments

- relative** (boolean, optional, default `False`) – errors relative to the data (`True`) or absolute (`False`).
- correlated** (boolean, optional, default `False`) – errors fully correlated (`True`) or totally uncorrelated (`False`).
- recompute\_cov\_mat** (boolean, optional, default `True`) – recalculate the total covariance matrix after adding the error source

**Return**this integer may later be used to remove or disable/enable the error source using `remove_error_source()` (page 32), `disable_error_source()` (page 29) or `enable_error_source()` (page 29).

#### Return type

**calc\_cov\_mats** (*axis*='all')

(Re-)Calculate the covariance matrix from the enabled error sources.

**Keyword Arguments***axis* ('x' or 'y' or 'all') – axis/axes for which to (re-)calculate covariance matrix.

**cov\_mat\_is\_regular** (*axis*)

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

**Parameters***axis* ('x' or 'y') – Axis for which to check for regularity of the covariance matrix.

**Returns**`True` if covariance matrix is regular

**Return type**boolean

**cov\_mats** = `None`

covariance matrices for axes

**disable\_error\_source** (*axis*, *err\_src\_id*)

Disables an `ErrorSource` by excluding it from the calculation of the total covariance matrix.

#### Parameters

- axis** ('x' or 'y') – axis for which to add error source.
- err\_src\_id** (*int*) – error source ID, as returned by `add_error_source()` (page 28).

**enable\_error\_source** (*axis*, *err\_src\_id*)

Enables an `ErrorSource` by excluding it from the calculation of the total covariance matrix.

#### Parameters

- axis** ('x' or 'y') – axis for which to add error source.
- err\_src\_id** (*int*) – error source ID, as returned by `add_error_source()` (page 28).

**err\_src** = `None`

lists of `ErrorSource` objects

**error\_source\_is\_enabled** (*axis*, *err\_src\_id*)

Returns `True` if an `ErrorSource` is enabled, that is if it is included in the total covariance matrix.

**Parameters**

- axis** ('x' or 'y') – Axis for which to load the error matrix.
- err\_src\_id** (int) – error source ID, as returned by `add_error_source()` (page 28).

**Returns**

- bool** – *True* if the specified error source is enables
- TODO** (`##DocString##`)

**get\_axis** (axis\_alias)

Get axis id from an alias.

**Parameters****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).

**Return**the axis ID

**Return type**int

**get\_cov\_mat** (axis, fallback\_on\_singular=None)

Get the error matrix for an axis.

**Parameters****axis** ('x' or 'y') – Axis for which to load the error matrix.

**Keyword Arguments****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 instead. Alternatively, the shortcuts 'identity' or 1 and 'zero' or 0 can be used to return the identity and zero matrix respectively.

**Return**the current covariance matrix

**Return type***numpy.matrix*

**get\_data** (axis)

Get the measurement data for an axis.

**Parameters****axis** (string) – Axis for which to get the measurement data. Can be 'x' or 'y'.

**Return**the measurement data for the axis

**Return type***numpy.array*

**get\_data\_span** (axis, include\_errorBars=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.

**Parameters****axis** ('x' or 'y') – Axis for which to get the data span.

**Keyword Arguments****include\_errorBars** (boolean, optional) – *True* if the returned span should be enlarged to contain the error bars of the smallest and largest datapoints (default: *False*)

**Return**the data span for the axis

**Return type**a tuple (*min*, *max*)

**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!

#### Keyword Arguments

- 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.

**Returns** a plain-text representation of the *Dataset*

**Return type** `str`

**get\_size()**

Get the size of the *Dataset*. This is equivalent to the length of the  $x$ -axis data.

**Returns** the number of datapoints in the *Dataset*.

**Return type** `int`

**has\_correlations** (*axis=None*)

Returns *True* if the specified axis has correlation data, *False* if not.

**Parameters** **axis** (`'x'` or `'y'` or *None*, optional) – Axis for which to check for correlations. If *None*, returns *true* if there are correlations for at least one axis.

**Returns** *True* if the specified axis has correlation data

**Return type** `bool`

**has\_errors** (*axis=None*)

Returns *True* if the specified axis has any kind of error data.

**Parameters** **axis** (`'x'` or `'y'` or *None*, optional) – Axis for which to check for error data. If *None*, returns *true* if there are errors for at least one axis.

**Returns** *True* if the specified axis has any kind of error data.

**Return type** `bool`

**n\_datapoints = None**

number of data points in the *Dataset*

**read\_from\_file** (*input\_file*)

Reads the *Dataset* object from a file.

One way to construct a *Dataset* is to specify an input file containing a plain-text representation of the dataset:

```
>>> my_dataset.read_from_file('/path/to/file')
```

or

```
>>> my_dataset.read_from_file(my_file_object)
```

For details on the format, see [get\\_formatted\(\)](#) (page 30)

**Parameters**`input_file` (*str*) – path to the file

**Returns**`True` if the read succeeded, `False` if not.

**Return type**`boolean`

**remove\_error\_source** (*axis*, *err\_src\_id*, *recompute\_cov\_mat=True*)

Remove the error source from the *Dataset*.

**Parameters**

• **axis** (*'x'* or *'y'*) – axis for which to add error source.

• **err\_src\_id** (*int*) – error source ID, as returned by [add\\_error\\_source\(\)](#) (page 28).

**Keyword Arguments**`recompute_cov_mat` (*boolean*, optional, default `True`) – recalculate the total covariance matrix after removing the error source

**set\_axis\_data** (*axis*, *data*)

Set the measurement data for a single axis.

**Parameters**

• **axis** (*'x'* or *'y'*) – Axis for which to set the measurement data.

• **data** (*iterable*) – Measurement data for axis.

**set\_cov\_mat** (*axis*, *mat*)

Forcibly set the error matrix for an axis, ignoring [ErrorSource](#) (page 33) objects. This is useful for adjusting the covariance matrix during the fit process.

**Parameters**

• **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** (*data*)

Set the measurement data for both axes.

Each element of **data** 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.set_data(([0., 1., 2.], [1.23, 3.45, 5.62]))
```

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

```
>>> my_dataset.set_data(([0.0, 1.23], [1.0, 3.45], [2.0, 5.62]))
```

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.

**Parameters**`data` (*iterable*) – the measurement data. Either of the form (*xdata*, *ydata*) or [(*x1*, *y1*), (*x2*, *y2*),... (*xn*, *yn*)]

**write\_formatted** (*file\_path*, *format\_string='%.06e'*, *delimiter='\t'*)

Writes the dataset to a plain-text file. For details on the format, see [read\\_from\\_file\(\)](#) (page 31).

**Parameters**`file_path` (*string*) – Path of the file object to write. **WARNING:** *overwrites existing files!*

### Keyword Arguments

•**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.

**class** `kafe.dataset.ErrorSource`

Bases: `object`

This object stores the error information for a `Dataset` as a *covariance matrix*  $C$  (sometimes also referred to as the *error matrix*). This has several advantages: it allows calculating the function to minimize (e.g. the chi-square) for a fit as a matrix product, and it allows specifying multiple error sources for a `Dataset` by simply adding up the corresponding matrices.

The object contains methods to generate a covariance matrix for some simple cases, such as when all points have the same relative or absolute errors and the errors are either not correlated or fully correlated. For more complicated error models, a covariance matrix can be specified directly.

**get\_matrix** (*size=None*)

Returns/Generates the covariance matrix for this `ErrorSource`.

If the user specified the matrix using `make_from_matrix()` (page 33), returns that matrix. If a simple error model is specified, a matrix is constructed as follows:

For *uncorrelated* errors, the covariance matrix is always diagonal.

If a single float  $\sigma$  is given as the error, the diagonal entries will be equal to  $\sigma^2$ . In this case, the matrix size needs to be specified via the `size` parameter.

If a list of floats  $\sigma_i$  is given as the error, the  $i$ -th entry will be equal to  $\sigma_i^2$ . In this case, the size of the matrix is inferred from the size of the list.

For *fully correlated* errors, the covariance matrix is the outer product of the error array  $\sigma_i$  with itself, i.e. the  $(i, j)$ -th matrix entry will be equal to  $\sigma_i \sigma_j$ .

**Keyword Argument****size** (*int (sometimes required)*) – Size of the matrix to return. Only relevant if the error value is a single float, since in that case there is no way to deduce the matrix size.

**make\_from\_matrix** (*cov\_mat, check\_singular=False*)

Sets the covariance matrix manually.

**Parameters****cov\_mat** (*numpy.matrix*) – A square, symmetric (and usually *regular*) matrix.

**Keyword Arguments****check\_singular** (*boolean, optional*) – Whether to force singularity check. Defaults to `False`.

**make\_from\_val** (*err\_val, fully\_correlated=False*)

Sets information required to construct the covariance matrix.

**Parameters****err\_val** (*float or sequence of floats*) – If all data points have the same uncertainty

**Keyword Arguments****fully\_correlated** (*boolean, optional*) – Whether the errors are fully correlated. Defaults to `False`.

## 4.2.2 Fit: Fit of model functions to a Dataset (`kafe.fit`)

`kafe.fit.CL2Chi2` (*CL*)

Helper function to calculate DeltaChi2 from confidence level CL

`kafe.fit.Chi22CL` (*dc2*)

Helper function to calculate confidence level CL from DeltaChi2

```
class kafe.fit.Fit (dataset, fit_function, external_fcn=<function chi2>, fit_name=None,
                  fit_label=None, minimizer_to_use='iminuit')
```

Bases: object

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* (the minimum of which should be located to find the best fit) can be specified. If not given, the *FCN* function defaults to  $\chi^2$ .

#### Parameters

- dataset** (*Dataset*) – A *Dataset* object containing all information about the data
- fit\_function** (*function*) – A user-defined Python function to fit 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.

#### Keyword Arguments

- 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, parameter_values)
```

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

- fit\_name** (*string, optional*) – An ASCII name for this fit. This is used as a label for the the matplotlib figure window and for naming the fit output file. If omitted, the fit will take over the name of the parent dataset.
- fit\_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 fit function's *LaTeX* expression.
- minimizer\_to\_use** (*'ROOT' or 'minuit', optional*) – Which minimizer to use. This defaults to whatever is set in the config file, but can be specifically overridden for some fits using this keyword argument

**call\_external\_fcn** (*\*parameter\_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.

**Parameters****parameter\_values** (*sequence of values*) – the parameter values at which *FCN* is to be evaluated

**call\_minimizer** (*final\_fit=True, verbose=False, quiet=False*)

Instructs the minimizer to do a minimization.

**constrain\_parameters** (*parameters, parvals, parerrs, cor\_mat=None*)

Constrain the parameter with the given name to  $c \pm \sigma$ . This is achieved by adding an appropriate *penalty term* to the  $\chi^2$  function, see function *chi2()* (page 39).

#### Parameters

- parameters** (*list of int*) – list of paramter id's or names to constrain

- parvals** (*list of float*) – list of parameter values

- parerrs** (*list of float*) – list of errors on parameters

**Keyword Arguments\*\*cor\_mat\*\*** (*numpy.matrix* optional) – correlation matrix of the parameters

**contours = None**

Parameter Contours [id1, id2, dchi2, [xc], [yc]]

**current\_cov\_mat = None**

the current covariance matrix used for the *Fit*

**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 repeated until the change in the error matrix caused by the projection becomes negligible.

#### Keyword Arguments

- 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*

**final\_fcn = None**

Final minimum of fcn (chi2)

**final\_parameter\_errors = None**

Final parameter errors

**final\_parameter\_values = None**

Final parameter values

**fit\_function = None**

the fit function used for this *Fit*

**fix\_parameters** (*\*parameters\_to\_fix*)

Fix the given parameters so that the minimizer works without them when `do_fit()` (page 35) is called next. Parameters can be given by their names or by their IDs.

**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**A function of a single variable corresponding to the fit function at the current parameter values.

**Return type**function handle

**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**The covariance matrix of the parameters.

**Return type***numpy.matrix*

**get\_function\_error** (*x*)

This method uses the parameter error matrix of the fit to calculate a symmetric (parabolic) error on the function value itself. Note that this method takes the entire parameter error matrix into account, so that it also accounts for correlations.

The method is useful if, e.g., you want to draw a confidence band around the function in your plot routine.

**Parameters***x* (*float* or sequence of *float*) – the values at which the function error is to be estimated

**Returns**the estimated error at the given point(s)

**Return type***float* or sequence of *float*

**get\_parameter\_errors** (*rounding=False*)

Get the current parameter uncertainties from the minimizer.

**Keyword Arguments***rounding* (*boolean, optional*) – Whether or not to round the returned values to significance.

**Returns**A tuple of the parameter uncertainties

**Return type***tuple*

**get\_parameter\_values** (*rounding=False*)

Get the current parameter values from the minimizer.

**Keyword Arguments***rounding* (*boolean, optional*) – Whether or not to round the returned values to significance.

**Returns**A tuple of the parameter values

**Return type***tuple*

**get\_results** ()

Return results from Fit

**latex\_parameter\_names** = *None*

*LaTeX* parameter names

**minos\_errors** = *None*

MINOS Errors [err, err+, err-, gcor]

**number\_of\_parameters** = *None*

the total number of parameters

**par\_cov\_mat** = *None*

Parameter covariance matrix (*numpy.matrix*)

**parabolic\_errors** = *None*

True if  $\chi^2$  is approx. parabolic (*boolean*)

**parameter\_is\_fixed** (*parameter*)

Check whether a parameter is fixed. Accepts a parameter's name or ID and returns a *boolean* value.

**parameter\_names** = *None*

the names of the parameters

**plot\_contour** (*parameter1, parameter2, dchi2=2.3, n\_points=100, color='gray', alpha=0.1, show=False, axes=None*)

Plots one or more two-dimensional contours for this fit into a separate figure and returns the figure object.

**Parameters**

• **parameter1** (*int* or *string*) – ID or name of the parameter to appear on the *x*-axis.

• **parameter2** (*int* or *string*) – ID or name of the parameter to appear on the *y*-axis.



**Keyword Arguments**

- **dchi2** (*float or list of floats (optional)*) – delta- $\chi^2$  value(s) used to evaluate contour(s) 1. = 1 sigma 2.3 = 68.0% (default) 4. = 2 sigma 5.99 = 95.0%
- **n\_points** (*int, optional*) – Number of plot points to use for the contour. Higher values yield smoother contours but take longer to render. Default is 100.
- **color** (*string, optional*) – A `matplotlib` color identifier specifying the fill color of the contour. Default is 'gray'.
- **alpha** (*float, optional*) – Transparency of the contour fill color ranging from 0. (fully transparent) to 1. (fully opaque). Default is 0.25
- **show** (*boolean, optional*) – Specify whether to show the figure before returning it. Defaults to `False`.
- **axes** (*matplotlib.pyplot.axes*) – Sub-plot axes to add plot to

**Returns** A figure object containing the contour plot.

**Return type** `matplotlib` figure object if no axes given

**plot\_correlations()**

Plots two-dimensional contours for all pairs of parameters and profile for all parameters, arranges as a matrix.

**Returns** A figure object containing the matrix of plots.

**Return type** `matplotlib` figure object

**plot\_profile(parid, n\_points=21, color='blue', alpha=0.5, show=False, axes=None)**

Plots a profile

$\chi^2$  for this fit into a separate figure and returns the figure object.

**Parameters** **parid** (*int or string*) – ID or name of parameter

**Keyword Arguments**

- **n\_points** (*int, optional*) – Number of plot points to use for the profile curve.
- **color** (*string, optional*) – A `matplotlib` color identifier specifying the line color. Default is 'blue'.
- **alpha** (*float, optional*) – Transparency of the contour fill color ranging from 0. (fully transparent) to 1. (fully opaque). Default is 0.25
- **show** (*boolean, optional*) – Specify whether to show the figure before returning it. Defaults to `False`.
- **axes** (*sub-plot axes to put plot*)

**Returns** A figure object containing the profile plot.

**Return type** `matplotlib` figure object if axes is `None`

**print\_fit\_details()**

prints some fit goodness details

**print\_fit\_results()**

prints fit results

**print\_raw\_results()**

unformatted print-out of all fit results in

**print\_rounded\_fit\_parameters()**

prints the fit parameters

**profiles = None**

Parameter Profiles [`id1`, [`xp`], [`dchi1(xp)`]]

**project\_x\_covariance\_matrix()**

Project elements of the  $x$  covariance matrix onto the total matrix.

This is done element-wise, according to the formula:

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

**release\_parameters(\*parameters\_to\_release)**

Release the given parameters so that the minimizer begins to work with them when `do_fit()` is called next. Parameters can be given by their names or by their IDs. If no arguments are provided, then release all parameters.

**set\_parameters(\*args, \*\*kwargs)**

Sets the parameter values (and optionally errors) for this fit. This is usually called just before the fit is done, to establish the initial parameters. If a parameter error is omitted, it is set to 1/10th of the parameter values themselves. If the default value of the parameter is 0, it is set, by exception, to 0.1.

This method accepts up to two positional arguments and several keyword arguments.

**Parameters**

- **args[0]** (*tuple/list of floats, optional*) – The first positional argument is expected to be a tuple/list containing the parameter values.
- **args[1]** (*tuple/list of floats, optional*) – The second positional argument is expected to be a tuple/list of parameter errors, which can also be set as an approximate estimate of the problem's uncertainty.

**Keyword Arguments**

- **no\_warning** (*boolean, optional*) – Whether to issue warnings (`False`) or not (`True`) when communicating with the minimizer fails. Defaults to `False`.
- **Valid keyword argument names are parameter names. The keyword arguments themselves may be floats (parameter values) or 2-tuples containing the parameter values and the parameter error in that order**
- **<parameter\_name>\*** (*float or 2-tuple of floats, optional*) – Set the parameter with the name `<'parameter_name'>` to the value given. If a 2-tuple is given, the first element is understood to be the value and the second to be the parameter error.

**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*

**class kafe.fit.GaussianConstraint (constraint, cov\_mat=None)**

Bases: `object`

Object used to constrain parameters. The object stores for each constrain the constrained parameters, the errors, the id of the parameter (the place at which each parameter is located in `parameter_constrain`) and the inverse covariance matrix of the constrained parameters. The class gives a tool to calculate the chi2 penalty term for the given constrained parameters, where the fitted `parameter_values` must be given.

**Parametersconstraint** (*list of two iterables*) – The first iterable ( $c_i$ ) contains the constrained parameters' expected values and the second iterable ( $\sigma_i$ ) contains the constraint uncertainties. A parameter with constraint uncertainty set to 0 remains unconstrained.

**Keyword Argumentscov\_mat** (*'numpy matrix'*) – Contains the covariance matrix of the constraints. The inverse covariance matrix will be saved to save computing time.

**calculate\_chi2\_penalty(parameter\_values)**

Calculates the  $\chi^2$  penalty for the given constraint parameter. This function gets called in the  $\chi^2$  function and returns a penalty term.

**Parameters****parameter\_values** (*list/tuple*) – The values of the parameters at which  $f(x)$  should be evaluated.

`kafe.fit.build_fit(dataset, fit_function, fit_label='untitled', fit_name=None, initial_fit_parameters=None, constrained_parameters=None)`

This helper function creates a *Fit* (page 33) from a series of keyword arguments.

#### Parameters

- **dataset** (a *kafe Dataset* (page 28)) –
- **fit\_function** (a *Python function*, optionally with) – `@FitFunction`, `@LATEX` and `@FitFunction` decorators

#### Keyword Arguments

- **fit\_label** (*LaTeX label for this fit, optional*) – Defaults to “untitled”
- **fit\_name** (*name for this fit, optional*) – Defaults to the dataset name
- **initial\_fit\_parameters** (*None or 2-tuple of list, sequence of floats*) – specifying initial parameter values and errors
- **constrained\_parameters** (*None or 3-tuple of list, tuple/np.array*) – of one string and 2 floats specifying the names, values and uncertainties of constraints to apply to model parameters

#### Returns

**Return type***Fit* (page 33) object

`kafe.fit.chi2(xdata, ydata, cov_mat, fit_function, parameter_values, constrain=None)`

The  $\chi^2$  implementation. Calculates  $\chi^2$  according to the formula:

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

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

If a constraint  $c_i \pm \sigma_i$  is applied to a parameter  $p_i$ , a *penalty term* is added for each constrained parameter:

$$\chi_{\text{cons}}^2 = \chi^2 + \sum_i \left( \frac{p_i - c_i}{\sigma_i} \right)^2$$

#### Parameters

- **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  $f(x)$
- **parameter\_values** (*list/tuple*) – The values of the parameters at which  $f(x)$  should be evaluated.

**Keyword Arguments****constrain** (*None or dictionary, optional*) – The key of the dictionary holds the parameter ids, while the values are *GaussianConstraint* objects with values, errors and correlation of the parameters.

`kafe.fit.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.

#### Parameters

- **value** (*float*) – value to round to significance
- **error** (*float*) – uncertainty of the value

**Keyword Argument****significance** (*int, optional*) – number of significant digits of the error to consider

### 4.2.3 Plot: Graphical representation of a Fit (`kafe.plot`)

`class kafe.plot.Plot (*fits, **kwargs)`

Bases: `object`

The constructor accepts a series of *Fit* objects as positional arguments. Some keyword arguments can be provided to override the defaults.

**axis\_labels = None**

axis labels

**compute\_plot\_range** (*include\_errorBars=True*)

Compute the span of all child datasets and sets the plot range to that

**draw\_fit\_parameters\_box** (*plot\_spec=0, force\_show\_uncertainties=False*)

Draw the parameter box to the canvas

#### Parameters

- **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.

- **force\_show\_uncertainties** (*boolean (optional, default: False)*) – If True, shows uncertainties even for Datasets without error data. Note that in that case these values are meaningless!

**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* objects to plot

**init\_plots** (*\*\*kwargs*)

Initialize the plots for each fit.

**on\_draw** (*event*)

Function to call when a draw event occurs.

**plot** (*p\_id, show\_data=True, show\_function=True, show\_band=True*)

Plot the *Fit* object with the number *p\_id* to its figure.

**plot\_all** (*show\_info\_for='all', show\_data\_for='all', show\_function\_for='all', show\_band\_for='meaningful'*)

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.

**set\_axis\_scale** (*axis, scale\_type, \*\*kwargs*)

Set the scale for an axis.

#### Parameters

- **axis** ('x' or 'y') – Axis for which to set the scale.

•**scale\_type** ('linear' or 'log') – Type of scale to set for the axis.

#### Keyword Arguments

•**basex** (*int*) – Base of the 'x' axis scale logarithm. Only relevant for log scales.

•**basey** (*int*) – Base of the 'y' axis scale logarithm. Only relevant for log scales.

**show()**

Show graphics in one or more matplotlib interactive windows.

---

**Note:** This shows all figures/plots generated before it is called. Because of the way `matplotlib` handles some plotting parameters (`matplotlib.rcParams`) these cannot be set individually for each figure before it is displayed. This means that all figures will be shown with the same plot style: that of the *Plot* object from which `show()` is called.

---

**show\_legend = None**

whether to show the plot legend (`True`) or not (`False`)

**class kafe.plot.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 overridden 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.

**kafe.plot.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.

**Parameters***label* (*string*) – Plain-text string to convert to LaTeX.

**kafe.plot.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 center) and negative amounts will shorten it (move the interval end towards the interval center).

**kafe.plot.pad\_span\_log** (*span*, *pad\_coeff*=1, *additional\_pad*=None, *base*=10)

## 4.3 Interfaces to external packages

### 4.3.1 Interface to ROOT's TMinuit (kafe.minuit)

**kafe.minuit.D\_MATRIX\_ERROR** = {0: 'Error matrix not calculated', 1: 'Error matrix approximate!', 2: 'Error matrix Error matrix status codes

**class** kafe.minuit.**Minuit** (*number\_of\_parameters, function\_to\_minimize, parameter\_names, start\_parameters, parameter\_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**[C array] If the user chooses to calculate the first derivative of the function inside the *FCN*, this value should be written here. This interface to *Minuit* ignores this derivative, however, so calculating this inside the *FCN* has no effect (yet).

**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.

**fix\_parameter** (*parameter\_number*)

Fix parameter number <*parameter\_number*>.

**parameter\_number**[int] Number of the parameter to fix.

**function\_to\_minimize** = **None**

the actual *FCN* called in *FCN\_wrapper*

**get\_chi2\_probability** (*n\_deg\_of\_freedom*)

Returns the probability that an observed  $\chi^2$  exceeds the calculated value of  $\chi^2$  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=21*)

Returns a list of points (2-tuples) representing a sampling of the  $1\sigma$  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 21.

**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] (parameter\_name, parameter\_val, parameter\_error)

**get\_parameter\_name** (parameter\_nr)  
Gets the name of parameter number parameter\_nr  
**parameter\_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

**get\_profile** (parid, n\_points=21)  
Returns a list of points (2-tuples) the profile the  $\chi^2$  of the TMinuit fit.  
**parid**[int] ID of the parameter to be displayed on the  $x$ -axis.  
**n\_points**[int (optional)] number of points used for profile. Default is 21.  
**returns**[two arrays, par. values and corresp.  $\chi^2$ ] containing n\_points sampled profile points.

**max\_iterations** = None  
maximum number of iterations until TMinuit gives up

**minimize** (final\_fit=True, log\_print\_level=2)  
Do the minimization. This calls *Minuit*'s algorithms MIGRAD for minimization and, if *final\_fit* is *True*, also HESSE for computing/checking the parameter error matrix.

**minos\_errors** (log\_print\_level=1)  
Get (asymmetric) parameter uncertainties from MINOS algorithm. This calls *Minuit*'s algorithms MINOS, which determines parameter uncertainties using profiling of the chi2 function.  
**returns**[tuple] A tuple of [err+, err-, parabolic error, global correlation]

**name** = None  
the name of this minimizer type

**number\_of\_parameters** = None  
number of parameters to minimize for

**release\_parameter** (parameter\_number)  
Release parameter number <parameter\_number>.  
**parameter\_number**[int] Number of the parameter to release.

**reset** ()  
Execute TMinuit's *mnrset* method.

**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** (parameter\_errors=None)  
Sets the fit parameter errors. If parameter\_values='None', sets the error to 10% of the parameter value.

**set\_parameter\_names** (parameter\_names)  
Sets the fit parameters. If parameter\_values='None', tries to infer defaults from the function\_to\_minimize.

**set\_parameter\_values** (parameter\_values)

Sets the fit parameters. If `parameter_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

**update\_parameter\_data** (*show\_warnings=False*)

(Re-)Sets the parameter names, values and step size on the C++ side of Minuit.

`kafe.minuit.P_DETAIL_LEVEL = 1`

default level of detail for TMinuit's output (typical range: -1 to 3, default: 1)

### 4.3.2 Interface to `iminuit` (`kafe.iminuit_wrapper`)

`kafe.iminuit_wrapper.D_MATRIX_ERROR = {0: 'Error matrix not calculated', 1: 'Error matrix approximate!', 2: 'Error matrix singular'}`  
Error matrix status codes

**class** `kafe.iminuit_wrapper.IMinuit` (*number\_of\_parameters*, *function\_to\_minimize*, *parameter\_names*, *start\_parameters*, *parameter\_errors*, *quiet=True*, *verbose=False*)

A wrapper class for `iminuit`.

**FCN\_wrapper** (*\*\*kw\_parameters*)

This wrapper converts from the “keyword argument” way of calling the function to a “positional argument” way, taking into account the order of the parameters as they appear in *self.parameter\_names*.

This mapping is done for each call, so it's quite resource intensive, but this is unavoidable, since external FCNs to minimize expect positional arguments.

**kw\_parameters**[dict] Map of parameter name to parameter value.

**errordef = None**

`iminuit` errordef

**fix\_parameter** (*parameter*)

Fix parameter *<parameter>*.

**parameter**[string] Name of the parameter to fix.

**function\_to\_minimize = None**

the actual *FCN* called in `FCN_wrapper`

**get\_chi2\_probability** (*n\_deg\_of\_freedom*)

Returns the probability that an observed  $\chi^2$  exceeds the calculated value of  $\chi^2$  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=21*)

Returns a list of points (2-tuples) representing a sampling of the  $1\sigma$  contour of the `iminuit` 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 21.

**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** (*correlation=False*)  
Retrieves the parameter error matrix from *iminuit*.

**correlation**[boolean (optional, default `False`)] If `True`, return correlation matrix, else return covariance matrix.

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 *iminuit*.

**return**[tuple] Current *Minuit* parameter errors

**get\_parameter\_info** ()  
Retrieves parameter information from *iminuit*.

**return**[list of tuples] (*parameter\_name*, *parameter\_val*, *parameter\_error*)

**get\_parameter\_name** (*parameter\_nr*)  
Gets the name of parameter number *parameter\_nr*

**parameter\_nr**[int] Number of the parameter whose name to get.

**get\_parameter\_values** ()  
Retrieves the parameter values from *iminuit*.

**return**[tuple] Current *Minuit* parameter values

**get\_profile** (*parameter*, *n\_points=21*)  
Returns a list of points (2-tuples) the profile the  $\chi^2$  of the *iminuit* fit.

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

**n\_points**[int (optional)] number of points used for profile. Default is 21.

**returns**[two arrays, par. values and corresp.  $\chi^2$ ] containing `n_points` sampled profile points.

**max\_iterations = None**  
maximum number of iterations until *iminuit* gives up

**minimize** (*final\_fit=True*, *log\_print\_level=2*)  
Do the minimization. This calls *Minuit*'s algorithms MIGRAD for minimization and, if *final\_fit* is `True`, also HESSE for computing/checking the parameter error matrix.

**minos\_errors** (*log\_print\_level=1*)  
Get (asymmetric) parameter uncertainties from MINOS algorithm. This calls *Minuit*'s algorithms MINOS, which determines parameter uncertainties using profiling of the chi2 function.

**returns**[tuple] A tuple of (err+, err-, parabolic error, global correlation)

**name** = None  
the name of this minimizer type

**number\_of\_parameters** = None  
number of parameters to minimize for

**release\_parameter** (*parameter*)  
Release parameter <*parameter*>.

**parameter**[string] Name of the parameter to release.

**reset** ()  
Resets iminuit by re-creating the minimizer.

**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** (*parameter\_errors=None, update\_iminuit=True*)  
Sets the fit parameter errors. If *parameter\_values='None'*, sets the error to 10% of the parameter value.

**set\_parameter\_names** (*parameter\_names, update\_iminuit=True*)  
Sets the fit parameter names.

**set\_parameter\_values** (*parameter\_values, update\_iminuit=True*)  
Sets the fit parameters. If *parameter\_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 iminuit 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 iminuit to use a certain strategy. Refer to iminuit's documentation for available strategies.

**set\_tolerance** (*tol*)  
Sets the tolerance value for Minuit.

**tol**[float] The tolerance

**update\_parameter\_data** (*show\_warnings=False*)  
(Re-)Sets the parameter names, values and step size in iminuit.

**kafe.iminuit\_wrapper.P\_DETAIL\_LEVEL = 1**  
default level of detail for iminuit's output (typical range: -1 to 3, default: 1)

## 4.4 kafe configuration

### 4.4.1 kafe configuration (kafe.config)

**kafe.config.create\_config\_file** (*config\_type, force=False*)  
Create a kafe config file.

**config\_type**['user' or 'local'] Create a 'user' config file in '~/.config/kafe' or a 'local' one in the current directory.

**force**[boolean (optional)] If true, overwrites existing files.

**kafe.config.log\_file** (*file\_relative\_path*)  
Returns correct location for placing log files.

## 4.5 Modules with helper tools

### 4.5.1 For building datasets (`kafe.dataset_tools`)

`kafe.dataset_tools.build_dataset` (*xdata*, *ydata*, *cov\_mats=None*, *xabserr=0.0*, *xrelerr=0.0*, *xabscor=0.0*, *xrelcor=0.0*, *yabserr=0.0*, *yrelerr=0.0*, *yabscor=0.0*, *yrelcor=0.0*, *title=None*, *axis\_labels=None*, *axis\_units=None*, *\*\*kwargs*)

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

#### Parameters

- **xdata** (list/tuple/*np.array* of floats) – This keyword argument is mandatory and should be an iterable containing *x*-axis the measurement data.
- **ydata** (list/tuple/*np.array* of floats) – This keyword argument is mandatory and should be an iterable containing *y*-axis the measurement data.
- **cov\_mats** (*None* or 2-tuple, optional) – This argument defaults to *None*, which means no covariance matrices are used. If covariance matrices are needed, a tuple with two entries (the first for *x* covariance matrices, the second for *y*) must be passed.

Each element of this tuple may be either *None* or a NumPy matrix object containing a covariance matrix for the respective axis.

#### Keyword Arguments

- **error specification keywords** (*iterable or numeric (see below)*) – In addition to covariance matrices, errors can be specified for each axis (*x* or *y*) according to a simplified error model.

In this respect, a valid keyword is composed of an axis, an error relativity specification (*abs* or *rel*) and error correlation type (*err* or *cor*). The errors are then set as follows:

##### 1. For totally uncorrelated errors (*err*):

- 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 fully correlated errors (*cor*):

- keyword argument *must* be a single number. The global correlated error for the axis is then set to that.

So, for example:

```
>>> my_dataset = build_dataset(..., yabserr=0.3, yrelcor=0.1)
```

creates a *Dataset* with an uncorrelated error of 0.3 for each *y* coordinate and a fully correlated (systematic) error of *y* of 0.1.

- **title** (*string, optional*) – The title of the *Dataset*.
- **axis\_labels** (*2-tuple of strings, optional*) – a 2-tuple containing the axis labels for the *Dataset*. This is relevant when plotting *Fits* of the *Dataset*, but is ignored when plotting more than one *Fit* in the same *Plot*.
- **axis\_units** (*2-tuple of strings, optional*) – a 2-tuple containing the axis units for the *Dataset*. This is relevant when plotting *Fits* of the *Dataset*, but is ignored when plotting more than one *Fit* in the same *Plot*.

**Returns** *Dataset* object constructed from data and error information

**Return type** *Dataset* (page 28)

## 4.5.2 For parsing files (`kafe.file_tools`)

`kafe.file_tools.buildDataset_fromFile` (*file\_to\_parse*)

Build a kafe *Dataset* (page 28) object from input file with key words and file format defined in *parse\_general\_inputfile* () (page 49)

**Parameters***file\_to\_parse* (file-like object or string containing a file path) – The file to parse.

**Returns**a *Dataset* (page 28) object constructed with the help of the method `kafe.dataset.Dataset.build_dataset()`

**Return type**`py:class:~kafe.dataset.Dataset`

`kafe.file_tools.buildFit_fromFile` (*file\_to\_parse*)

Build a kafe *Fit* (page 33) object from input file with keywords and file format defined in *parse\_general\_inputfile* () (page 49)

**Parameters***file\_to\_parse* (file-like object or string containing a file path) – The file to parse.

**Returns**a *Fit* (page 33) object constructed with the help of the methods `build_dataset()` and `build_fit()`

**Return type**`py:class:~kafe.fit.Fit`

`kafe.file_tools.parse_column_data` (*file\_to\_parse*, *field\_order*='x,y', *delimiter*=' ', *cov\_mat\_files*=None, *title*='Untitled Dataset', *base\_name*=None, *axis\_labels*=['x', 'y'], *axis\_units*=['', ''])

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*, *xabserr*, *yabserr*, *xrelerr*, *yrelerr*. Another valid field name is *ignore* which can be used to skip a field.

A certain type of field can appear several times. If this is the case, all specified errors are added in quadrature:

$$\sigma_{\text{tot}} = \sqrt{\sigma_1^2 + \sigma_2^2 + \dots}$$

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

For more complex error models, errors and correlations may be specified as covariance matrices. If this is desired, then any number of covariance matrices (stored in separate files) may be specified for an axis by using the *cov\_mat\_files* argument.

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.

### Parameters

- **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** (several (see below), optional) – This argument defaults to None, which means no covariance matrices are used. If covariance matrices are needed, a tuple with two entries (the first for *x* covariance matrices, the second for *y*) must be passed.

Each element of this tuple may be either None, a file or file-like object, or an iterable containing files and file-like objects. Each file should contain a covariance matrix for the respective axis.

When creating the *Dataset*, all given matrices are summed over.

- **title** (string, optional) – The title of the *Dataset*.

- basename** (string or None, optional) – A basename for the Dataset. All output files related to this dataset will use this as a basename. If this is None (default), the basename will be inferred from the filename.
- axis\_labels** (2-tuple of strings, optional) – a 2-tuple containing the axis labels for the Dataset. This is relevant when plotting Fits of the Dataset, but is ignored when plotting more than one Fit in the same Plot.
- axis\_units** (2-tuple of strings, optional) – a 2-tuple containing the axis units for the Dataset. This is relevant when plotting Fits of the Dataset, but is ignored when plotting more than one Fit in the same Plot.

**Returns**A Dataset built from the parsed file.

**Return type**Dataset (page 28)

`kafe.file_tools.parse_general_inputfile (file_to_parse)`

This function can be used to specify *kafe Dataset* (page 28) or *Fit* (page 33) objects in a single input file, thus requiring minimal Python code. Keywords as specified in a dictionary `tokens` specify all objects and parameters needed by the functions `build_dataset()` (page 47) in module `dataset` (page 28) and `build_fit()` (page 39) in module `fit` (page 33).

**Parameters**`file_to_parse` (file-like object or string containing a file path) – The file to parse.

**Returns**keyword lists to build a *kafe Dataset* (page 28) or *Fit* (page 33) object with the helper functions `build_dataset` or `build_fit`

**Return type**(dataset\_kwargs, fit\_kwargs)

### Input file format

The interpretation of the input data is driven by *keywords*. All data following a key must be of the same kind. A block of data ends when a new key is specified. Comments can be introduced by #.

Some keys only expect a single float or string-type value, given on the same line, separated by a space (' '):

```
<key> <value>
```

Other keys require multiple lines of input. For instance, the keys `*xData` and `*yData` expect the following lines to be a table where the first column corresponds to the data values and the second column corresponds to the uncertainties:

```
<key>
<value1> <uncertainty1>
<value2> <uncertainty2>
...
<valueN> <uncertaintyN>
```

The column separator is space (' '). For more details about input data specification, see [below](#) (page 49).

### Specifying metadata

Key	Description
*TITLE	name of the dataset
*BASENAME	name from which output file names is derived
*FITLABEL	fit label
*xLabel	x axis label
*xUnit	x axis unit
*yLabel	y axis label
*yUnit	y axis unit

The fit label may be set using the key `*FITLABEL`, followed by the desired name for the fit. **Specifying input data**

Input data are given as a list of values (one datapoint per row). For a simple uncertainty model (no correlations), the keys `*xData` and `*yData` are used. The second column indicates the uncertainty of the measurement:

```
*xData
1.2
3.4
6.9

*yData
2.1      0.2
3.9      0.3
8.2      0.5
```

**Note:** Uncertainties always have to be specified for `*yData`. For `*xData`, they are optional.

For input data with correlated uncertainties, the alternative keys `*xData_COR` and `*yData_COR` are provided. For these, additional columns must be given. The second and third column indicate the uncorrelated and correlated uncertainties, respectively. The subsequent columns contain the correlation matrix (a lower triangular matrix containing the correlation coefficients):

```
*yData_COR
# value indep.uncert. syst.uncert. elements of corr. matrix.
2.1      0.2          0.1
3.9      0.3          0.2          1.0
8.2      0.5          0.3          1.0          1.0
```

**Note:** Only elements below the main diagonal of the correlation matrix have to be specified. Since the matrix is symmetric by construction, the elements above the main diagonal can be inferred from those below. Additionally, since the diagonal elements of a correlation matrix are always equal to 1 by definition, they are also omitted.

As an alternative to specifying the correlation matrix, the covariance matrix may be specified directly. There are two ways to do this:

The keys `*xData_SCOV` and `*yData_SCOV` allow specifying the covariance matrix by providing a correlated uncertainty (third column) and the square root of the elements below the main diagonal. This is useful if the pairwise covariance of two measurements cannot be expressed using the correlation coefficient and needs to be provided explicitly.

In the example below, there is a correlation between the first two and the last two measurements, which is estimated under the assumption that the smaller of the two uncertainties represents a common error:

```
*yData_SCOV
# mH      err      syst      sqrt(cov)
124.51    0.52     0.06
125.60    0.40     0.20    0.06
125.98    0.42     0.28    0.    0.
124.70    0.31     0.15    0.    0.    0.15
```

A second possibility is specifying the full covariance matrix directly. This is achieved using the `*xData_COV` and `*yData_COV` keywords. In this case, only the data values and the uncorrelated uncertainties (first and second columns, respectively) must be specified in addition to the covariance matrix (all other columns). All entries starting with the third column are assumed to be covariance matrix elements. The matrix is symmetric, so elements above the diagonal are omitted. Note that the diagonal must be specified and corresponds to the squares of the correlated errors:

```
*yData_COV
# mH      err      cov_ij
124.51    0.52     0.0036
125.60    0.40     0.0036    0.04
125.98    0.42     0.        0.        0.0784
```

124.70	0.31	0.	0.	0.0225	0.0225
--------	------	----	----	--------	--------

### Specifying additional uncertainties

In addition to the uncertainties already specified in the *input data table* (page 49), other systematic uncertainties may be provided. These are assumed be fully correlated and common to all data points. This can be achieved by using the following keys:

Key	Description
*xAbsCor	common fully correlated x-uncertainty (absolute)
*yAbsCor	common fully correlated y-uncertainty (absolute)
*xRelCor	common fully correlated x-uncertainty (relative)
*yRelCor	common fully correlated y-uncertainty (relative)

### Specifying a fit function

To specify the fit function, the key `*FitFunction` is provided. This key should be followed by *Python* code:

```
def fitf(x, ...):
    ...
    return ...
```

**Note:** Only one Python function may be defined after the `*FitFunction` keyword. Also, any function name can be used instead of `fitf`.

Additionally, the decorators `@ASCII`, `@LaTeX` and `@FitFunction` are supported (see *ASCII* (page 52), *LaTeX* (page 53) and *FitFunction* (page 52))

### Specifying initial values for parameters

Initial values for fit parameters may be set using the keyword `*InitialParameters`. This keyword expects to be followed by a table with two columns containing floating-point values.

Each line in the table corresponds to one fit parameter, in the order they are given in the fit function signature. The first column should contain the initial value of the parameters and the second column the “initial uncertainty”, which controls the initial variation range of the parameter at the beginning of the fit:

```
*InitialParameters
<initial value par 1> <initial uncert par 1>
<initial value par 2> <initial uncert par 2>
...
<initial value par N> <initial uncert par N>
```

### Constraining parameters

If there is any prior knowledge about model parameters’ values on uncertainties, these may be constrained during the fit.

During the fit, model parameters can be constrained within their uncertainties if there is any prior knowledge about their values and uncertainties.

This may be specified using the keyword `*ConstrainedParameters`, followed by a table containing the parameter name, value and uncertainty for each parameter to be constrained:

<parameter name>	<parameter value>	<parameter uncert.> ,
------------------	-------------------	-----------------------

**Note:** The parameter name must be the one specified in the fit function definition.

### Example

Here is an example of an input file to calculate the average of four partly correlated measurements (see *Example 8* (page 21)):

```
# Meta data for plotting
*TITLE Higgs-mass measurements
*xLabel number of measurement
*yLabel  $m_{\mathrm{H}}$ 
*yUnit GeV/ $c^2$ 

#*xData # commented out, as not needed for simple average

*yData_SCOV # assume that minimum of syst. errors is a common error
# mH      err      syst as sqrt(cov)
124.51    0.52     0.06
125.60    0.40     0.20  0.06
125.98    0.42     0.28  0.   0.
124.70    0.31     0.15  0.   0.  0.15

*FitFunction # Python code of fit function

# kafe fit function decorators are supported
@ASCII(expression='av')
@LaTeX(name='f', parameter_names=('av'), expression='av')
@FitFunction
def fitf(x, av=1.0): # fit an average
    return av

*FITLABEL Average

*InitialParameters
120. 1.
```

### 4.5.3 For specifying and decorating fit functions (`kafe.function_tools`)

`kafe.function_tools.ASCII(**kwargs)`

Optional decorator for fit functions. This overrides a `FitFunction`'s plain-text (ASCII) attributes. The new values for these attributes must be passed as keyword arguments to the decorator. Possible arguments:

**name**[string] Plain-text representation of the function name.

**parameter\_names**[list of strings] List of plain-text representations of the function's arguments. The length of this list must be equal to the function's argument number. The argument names should be in the same order as in the function definition.

**x\_name**[string] Plain-text representation of the independent variable's name.

**expression**[string] Plain-text-formatted expression representing the function's formula.

**class** `kafe.function_tools.FitFunction(f)`

Decorator class for fit functions. If a function definition is decorated using this class, some information is collected about the function which is relevant to the fitting process, such as the number of parameters, their names and default values. Some details pertaining to display and representation are also set, such as *LaTeX* representations of the parameter names and the function name. Other decorators can be applied to a function object to specify things such as a *LaTeX* or plain-text expression for the fit function.

**derive\_by\_parameters** (*x\_0*, *precision\_spec*, *parameter\_list*)

Returns the gradient of *func* with respect to its parameters, i.e. with respect to every variable of *func* except the first one.

**precision\_spec**[float or iterable of floats] An array of floats indicating the initial point spacing for numerically evaluating the derivative. Can be a single float value to use the same spacing for every derivation.

**derive\_by\_x** (*x\_0*, *precision\_list*, *parameter\_list*)

If *x\_0* is iterable, gives the array of derivatives of a function  $f(x, par_1, par_2, \dots)$  around  $x = x_i$  at



every  $x_i$  in  $\vec{x}$ . If  $x_0$  is not iterable, gives the derivative of a function  $f(x, par_1, par_2, \dots)$  around  $x = x_0$ .

**evaluate** ( $x_0$ , *parameter\_list*)

Evaluate the fit function at an x-value or at an array of x-values for the parameter values in *parameter\_list*.

**x\_0** float or array of floats

**parameter\_list** values of function parameters

**returns** function value(s)

**expression = None**

a math expression (string) representing the function's result

**get\_function\_equation** (*equation\_format*='latex', *equation\_type*='full', *ensuremath*=True)

Returns a string representing the function equation. Supported formats are *LaTeX* and ASCII inline math. Note that *LaTeX* math is wrapped by default in an `\ensuremath{}` expression. If this is not desired behaviour, the flag `ensuremath` can be set to `False`.

**equation\_format**[string (optional)] Can be either "latex" (default) or "ascii".

**equation\_type**[string (optional)] Can be either "full" (default), "short" or "name". A "name"-type equation returns a representation of the function name:

```
f
```

A "short"-type equation limits itself to the function name and variables:

```
f(x, par1, par2)
```

A "full"-type equation includes the expression which the function calculates:

```
f(x, par1, par2) = par1 * x + par2
```

**ensuremath**[boolean (optional)] If a *LaTeX* math equation is requested, `True` (default) will wrap the resulting expression in an `\ensuremath{}` tag. Otherwise, no wrapping is done.

**latex\_expression = None**

a *LaTeX* math expression, the function's result

**latex\_name = None**

The function's name in *LaTeX*

**latex\_parameter\_names = None**

A list of parameter names in *LaTeX*

**latex\_x\_name = None**

A *LaTeX* symbol for the independent variable.

**name = None**

The name of the function

**number\_of\_parameters = None**

The number of parameters

**parameter\_defaults = None**

The default values of the parameters

**parameter\_names = None**

The names of the parameters

**x\_name = None**

The name given to the independent variable

`kafe.function_tools.LaTeX(**kwargs)`

Optional decorator for fit functions. This overrides a `FitFunction`'s `latex_` attributes. The new values for the `latex_` attributes must be passed as keyword arguments to the decorator. Possible arguments:

**name**[string] *LaTeX* representation of the function name.

**parameter\_names**[list of strings] List of *LaTeX* representations of the function's arguments. The length of this list must be equal to the function's argument number. The argument names should be in the same order as in the function definition.

**x\_name**[string] *LaTeX* representation of the independent variable's name.

**expression**[string] *LaTeX*-formatted expression representing the function's formula.

`kafe.function_tools.derivative(func, derive_by_index, variables_tuple, derivative_spacing)`

Gives  $\frac{\partial f}{\partial x_k}$  for  $f = f(x_0, x_1, \dots)$ . *func* is *f*, *variables\_tuple* is  $\{x_i\}$  and *derive\_by\_index* is *k*.

`kafe.function_tools.outer_product(input_array)`

Takes a *NumPy* array and returns the outer (dyadic, Kronecker) product with itself. If *input\_array* is a vector *x*, this returns  $xx^T$ .

## 4.5.4 For working with LaTeX strings (kafe.latex\_tools)

`kafe.latex_tools.ascii_to_latex_math(str_ascii, monospace=True, ensuremath=True)`

Escapes certain characters in an ASCII input string so that the result can be included in math mode without error.

**str\_ascii**[string] A plain-text string containing characters to be escaped for *LaTeX* math mode.

**monospace**[boolean (optional)] Whether to render the whole expression as monospace. Defaults to `True`.

**ensuremath**[boolean (optional)] If this is `True`, the resulting formula is wrapped in an `\ensuremath{\}` tag. Defaults to `True`.

## 4.5.5 For routine numeric tasks (kafe.numeric\_tools)

`kafe.numeric_tools.MinuitCov_to_cor(cov_mat)`

Converts a covariance matrix as returned by Minuit to the corresponding correlation matrix; note that the Minuit covariance matrix may contain lines/rows with zeroes if parameters are fixed

**cov\_mat**[*numpy.matrix*] The Minuit covariance matrix to convert.

`kafe.numeric_tools.cor_to_cov(cor_mat, error_list)`

Converts a correlation matrix to a covariance matrix according to the formula

$$\text{Cov}_{ij} = \text{Cor}_{ij} \sigma_i \sigma_j$$

**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*.

`kafe.numeric_tools.cov_to_cor(cov_mat)`

Converts a covariance matrix to a correlation matrix according to the formula

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

**cov\_mat**[*numpy.matrix*] The covariance matrix to convert.

`kafe.numeric_tools.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*

`kafe.numeric_tools.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.

`kafe.numeric_tools.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.

## 4.6 Auxilliary modules

### 4.6.1 Collection of ready-to-use fit functions (`kafe.function_library`)

Collection of model functions

### 4.6.2 File/stream manipulation (`kafe.stream`)

**class** `kafe.stream.StreamDup(out_file, suppress_stdout=False)`

Bases: `object`

Object for simultaneous logging to stdout and files. This object provides a file-like object for the output to be written to. Writing to this object will write to stdout (usually the console) and to a file.

**out\_file**[file path or file-like object or list of file paths ...] File(s) to which to log the output, along with stdout. If a file exists on disk, it will be appended to.

**suppress\_stdout**[boolean] Whether to log to stdout simultaneously (`False`) or suppress output to stdout (`True`). Default to `False`.

**fileno()**

Returns the file handler id of the main (first) output file.

**flush()**

**write(message)**

**write\_timestamp(prefix)**

**write\_to\_file(message)**

**write\_to\_stdout(message, check\_if\_suppressed=False)**

Explicitly write to stdout. This method will not check by default whether `suppress_stdout` is set for this *StreamDup*. If `check_if_suppressed` is explicitly set to `True`, then this check occurs.



**c**

`config` (*Unix*), 46

**d**

`dataset` (*Unix*), 28

`dataset_tools` (*Unix*), 47

**f**

`file_tools` (*Unix*), 48

`fit` (*Unix*), 33

`function_library` (*Unix*), 55

`function_tools` (*Unix*), 52

**i**

`iminuit` (*Unix*), 44

**k**

`kafe.__init__`, 27

`kafe.config`, 46

`kafe.dataset`, 28

`kafe.dataset_tools`, 47

`kafe.file_tools`, 48

`kafe.fit`, 33

`kafe.function_library`, 55

`kafe.function_tools`, 52

`kafe.iminuit_wrapper`, 44

`kafe.latex_tools`, 54

`kafe.minuit`, 41

`kafe.numeric_tools`, 54

`kafe.plot`, 40

`kafe.stream`, 55

**l**

`latex_tools` (*Unix*), 54

**m**

`minuit` (*Unix*), 41

**n**

`numeric_tools` (*Unix*), 54

**p**

`plot` (*Unix*), 40

**s**

`stream` (*Unix*), 55



## A

add\_error\_source() (kafe.dataset.Dataset method), 28  
 ASCII() (in module kafe.function\_tools), 52  
 ascii\_to\_latex\_math() (in module kafe.latex\_tools), 54  
 axis\_labels (kafe.plot.Plot attribute), 40

## B

build\_dataset() (in module kafe.dataset\_tools), 47  
 build\_fit() (in module kafe.fit), 39  
 buildDataset\_fromFile() (in module kafe.file\_tools), 48  
 buildFit\_fromFile() (in module kafe.file\_tools), 48

## C

calc\_cov\_mats() (kafe.dataset.Dataset method), 29  
 calculate\_chi2\_penalty() (kafe.fit.GaussianConstraint method), 38  
 call\_external\_fcn() (kafe.fit.Fit method), 34  
 call\_minimizer() (kafe.fit.Fit method), 34  
 chi2() (in module kafe.fit), 39  
 Chi22CL() (in module kafe.fit), 33  
 CL2Chi2() (in module kafe.fit), 33  
 compute\_plot\_range() (kafe.plot.Plot method), 40  
 config (module), 46  
 constrain\_parameters() (kafe.fit.Fit method), 34  
 contours (kafe.fit.Fit attribute), 35  
 cor\_to\_cov() (in module kafe.numeric\_tools), 54  
 cov\_mat\_is\_regular() (kafe.dataset.Dataset method), 29  
 cov\_mats (kafe.dataset.Dataset attribute), 29  
 cov\_to\_cor() (in module kafe.numeric\_tools), 54  
 create\_config\_file() (in module kafe.config), 46  
 current\_cov\_mat (kafe.fit.Fit attribute), 35

## D

D\_MATRIX\_ERROR (in module kafe.iminuit\_wrapper), 44  
 D\_MATRIX\_ERROR (in module kafe.minuit), 41  
 Dataset (class in kafe.dataset), 28  
 dataset (kafe.fit.Fit attribute), 35  
 dataset (module), 28  
 dataset\_tools (module), 47  
 derivative() (in module kafe.function\_tools), 54  
 derive\_by\_parameters() (kafe.function\_tools.FitFunction method), 52  
 derive\_by\_x() (kafe.function\_tools.FitFunction method), 52

disable\_error\_source() (kafe.dataset.Dataset method), 29  
 do\_fit() (kafe.fit.Fit method), 35  
 draw\_fit\_parameters\_box() (kafe.plot.Plot method), 40  
 draw\_legend() (kafe.plot.Plot method), 40

## E

enable\_error\_source() (kafe.dataset.Dataset method), 29  
 err\_src (kafe.dataset.Dataset attribute), 29  
 error\_source\_is\_enabled() (kafe.dataset.Dataset method), 29  
 errordef (kafe.iminuit\_wrapper.IMinuit attribute), 44  
 ErrorSource (class in kafe.dataset), 33  
 evaluate() (kafe.function\_tools.FitFunction method), 53  
 expression (kafe.function\_tools.FitFunction attribute), 53  
 extend\_span() (kafe.plot.Plot method), 40  
 external\_fcn (kafe.fit.Fit attribute), 35  
 extract\_statistical\_errors() (in module kafe.numeric\_tools), 54

## F

FCN\_wrapper() (kafe.iminuit\_wrapper.IMinuit method), 44  
 FCN\_wrapper() (kafe.minuit.Minuit method), 42  
 file\_tools (module), 48  
 fileno() (kafe.stream.StreamDup method), 55  
 final\_fcn (kafe.fit.Fit attribute), 35  
 final\_parameter\_errors (kafe.fit.Fit attribute), 35  
 final\_parameter\_values (kafe.fit.Fit attribute), 35  
 Fit (class in kafe.fit), 33  
 fit (module), 33  
 fit\_function (kafe.fit.Fit attribute), 35  
 FitFunction (class in kafe.function\_tools), 52  
 fits (kafe.plot.Plot attribute), 40  
 fix\_parameter() (kafe.iminuit\_wrapper.IMinuit method), 44  
 fix\_parameter() (kafe.minuit.Minuit method), 42  
 fix\_parameters() (kafe.fit.Fit method), 35  
 flush() (kafe.stream.StreamDup method), 55  
 function\_library (module), 55  
 function\_to\_minimize (kafe.iminuit\_wrapper.IMinuit attribute), 44  
 function\_to\_minimize (kafe.minuit.Minuit attribute), 42  
 function\_tools (module), 52

## G

GaussianConstraint (class in kafe.fit), 38  
get\_axis() (kafe.dataset.Dataset method), 30  
get\_chi2\_probability() (kafe.iminuit\_wrapper.IMinuit method), 44  
get\_chi2\_probability() (kafe.minuit.Minuit method), 42  
get\_contour() (kafe.iminuit\_wrapper.IMinuit method), 44  
get\_contour() (kafe.minuit.Minuit method), 42  
get\_cov\_mat() (kafe.dataset.Dataset method), 30  
get\_current\_fit\_function() (kafe.fit.Fit method), 35  
get\_data() (kafe.dataset.Dataset method), 30  
get\_data\_span() (kafe.dataset.Dataset method), 30  
get\_error\_matrix() (kafe.fit.Fit method), 35  
get\_error\_matrix() (kafe.iminuit\_wrapper.IMinuit method), 45  
get\_error\_matrix() (kafe.minuit.Minuit method), 42  
get\_fit\_info() (kafe.iminuit\_wrapper.IMinuit method), 45  
get\_fit\_info() (kafe.minuit.Minuit method), 42  
get\_formatted() (kafe.dataset.Dataset method), 30  
get\_function\_equation() (kafe.function\_tools.FitFunction method), 53  
get\_function\_error() (kafe.fit.Fit method), 36  
get\_line() (kafe.plot.PlotStyle method), 41  
get\_linecolor() (kafe.plot.PlotStyle method), 41  
get\_marker() (kafe.plot.PlotStyle method), 41  
get\_markercolor() (kafe.plot.PlotStyle method), 41  
get\_matrix() (kafe.dataset.ErrorSource method), 33  
get\_parameter\_errors() (kafe.fit.Fit method), 36  
get\_parameter\_errors() (kafe.iminuit\_wrapper.IMinuit method), 45  
get\_parameter\_errors() (kafe.minuit.Minuit method), 43  
get\_parameter\_info() (kafe.iminuit\_wrapper.IMinuit method), 45  
get\_parameter\_info() (kafe.minuit.Minuit method), 43  
get\_parameter\_name() (kafe.iminuit\_wrapper.IMinuit method), 45  
get\_parameter\_name() (kafe.minuit.Minuit method), 43  
get\_parameter\_values() (kafe.fit.Fit method), 36  
get\_parameter\_values() (kafe.iminuit\_wrapper.IMinuit method), 45  
get\_parameter\_values() (kafe.minuit.Minuit method), 43  
get\_pointsize() (kafe.plot.PlotStyle method), 41  
get\_profile() (kafe.iminuit\_wrapper.IMinuit method), 45  
get\_profile() (kafe.minuit.Minuit method), 43  
get\_results() (kafe.fit.Fit method), 36  
get\_size() (kafe.dataset.Dataset method), 31

## H

has\_correlations() (kafe.dataset.Dataset method), 31  
has\_errors() (kafe.dataset.Dataset method), 31

## I

IMinuit (class in kafe.iminuit\_wrapper), 44

iminuit (module), 44  
init\_plots() (kafe.plot.Plot method), 40

## K

kafe.\_\_init\_\_ (module), 27  
kafe.config (module), 46  
kafe.dataset (module), 28  
kafe.dataset\_tools (module), 47  
kafe.file\_tools (module), 48  
kafe.fit (module), 33  
kafe.function\_library (module), 55  
kafe.function\_tools (module), 52  
kafe.iminuit\_wrapper (module), 44  
kafe.latex\_tools (module), 54  
kafe.minuit (module), 41  
kafe.numeric\_tools (module), 54  
kafe.plot (module), 40  
kafe.stream (module), 55

## L

label\_to\_latex() (in module kafe.plot), 41  
LaTeX() (in module kafe.function\_tools), 53  
latex\_expression (kafe.function\_tools.FitFunction attribute), 53  
latex\_name (kafe.function\_tools.FitFunction attribute), 53  
latex\_parameter\_names (kafe.fit.Fit attribute), 36  
latex\_parameter\_names (kafe.function\_tools.FitFunction attribute), 53  
latex\_tools (module), 54  
latex\_x\_name (kafe.function\_tools.FitFunction attribute), 53  
log\_file() (in module kafe.config), 46

## M

make\_from\_matrix() (kafe.dataset.ErrorSource method), 33  
make\_from\_val() (kafe.dataset.ErrorSource method), 33  
make\_symmetric\_lower() (in module kafe.numeric\_tools), 54  
max\_iterations (kafe.iminuit\_wrapper.IMinuit attribute), 45  
max\_iterations (kafe.minuit.Minuit attribute), 43  
minimize() (kafe.iminuit\_wrapper.IMinuit method), 45  
minimize() (kafe.minuit.Minuit method), 43  
minos\_errors (kafe.fit.Fit attribute), 36  
minos\_errors() (kafe.iminuit\_wrapper.IMinuit method), 45  
minos\_errors() (kafe.minuit.Minuit method), 43  
Minuit (class in kafe.minuit), 41  
minuit (module), 41  
MinuitCov\_to\_cor() (in module kafe.numeric\_tools), 54

## N

n\_datapoints (kafe.dataset.Dataset attribute), 31



name (kafe.function\_tools.FitFunction attribute), 53  
 name (kafe.iminuit\_wrapper.IMinuit attribute), 45  
 name (kafe.minuit.Minuit attribute), 43  
 number\_of\_parameters (kafe.fit.Fit attribute), 36  
 number\_of\_parameters  
   (kafe.function\_tools.FitFunction attribute), 53  
 number\_of\_parameters (kafe.iminuit\_wrapper.IMinuit attribute), 46  
 number\_of\_parameters (kafe.minuit.Minuit attribute), 43  
 numeric\_tools (module), 54

## O

on\_draw() (kafe.plot.Plot method), 40  
 outer\_product() (in module kafe.function\_tools), 54

## P

P\_DETAIL\_LEVEL (in module kafe.iminuit\_wrapper), 46  
 P\_DETAIL\_LEVEL (in module kafe.minuit), 44  
 pad\_span() (in module kafe.plot), 41  
 pad\_span\_log() (in module kafe.plot), 41  
 par\_cov\_mat (kafe.fit.Fit attribute), 36  
 parabolic\_errors (kafe.fit.Fit attribute), 36  
 parameter\_defaults (kafe.function\_tools.FitFunction attribute), 53  
 parameter\_is\_fixed() (kafe.fit.Fit method), 36  
 parameter\_names (kafe.fit.Fit attribute), 36  
 parameter\_names (kafe.function\_tools.FitFunction attribute), 53  
 parse\_column\_data() (in module kafe.file\_tools), 48  
 parse\_general\_inputfile() (in module kafe.file\_tools), 49  
 Plot (class in kafe.plot), 40  
 plot (module), 40  
 plot() (kafe.plot.Plot method), 40  
 plot\_all() (kafe.plot.Plot method), 40  
 plot\_contour() (kafe.fit.Fit method), 36  
 plot\_correlations() (kafe.fit.Fit method), 37  
 plot\_profile() (kafe.fit.Fit method), 37  
 plot\_range (kafe.plot.Plot attribute), 40  
 plot\_style (kafe.plot.Plot attribute), 40  
 PlotStyle (class in kafe.plot), 41  
 print\_fit\_details() (kafe.fit.Fit method), 37  
 print\_fit\_results() (kafe.fit.Fit method), 37  
 print\_raw\_results() (kafe.fit.Fit method), 37  
 print\_rounded\_fit\_parameters() (kafe.fit.Fit method), 37  
 profiles (kafe.fit.Fit attribute), 37  
 project\_x\_covariance\_matrix() (kafe.fit.Fit method), 37

## R

read\_from\_file() (kafe.dataset.Dataset method), 31  
 release\_parameter() (kafe.iminuit\_wrapper.IMinuit method), 46  
 release\_parameter() (kafe.minuit.Minuit method), 43  
 release\_parameters() (kafe.fit.Fit method), 38

remove\_error\_source() (kafe.dataset.Dataset method), 32  
 reset() (kafe.iminuit\_wrapper.IMinuit method), 46  
 reset() (kafe.minuit.Minuit method), 43  
 round\_to\_significance() (in module kafe.fit), 39

## S

save() (kafe.plot.Plot method), 40  
 set\_axis\_data() (kafe.dataset.Dataset method), 32  
 set\_axis\_scale() (kafe.plot.Plot method), 40  
 set\_cov\_mat() (kafe.dataset.Dataset method), 32  
 set\_data() (kafe.dataset.Dataset method), 32  
 set\_err() (kafe.iminuit\_wrapper.IMinuit method), 46  
 set\_err() (kafe.minuit.Minuit method), 43  
 set\_parameter\_errors() (kafe.iminuit\_wrapper.IMinuit method), 46  
 set\_parameter\_errors() (kafe.minuit.Minuit method), 43  
 set\_parameter\_names() (kafe.iminuit\_wrapper.IMinuit method), 46  
 set\_parameter\_names() (kafe.minuit.Minuit method), 43  
 set\_parameter\_values() (kafe.iminuit\_wrapper.IMinuit method), 46  
 set\_parameter\_values() (kafe.minuit.Minuit method), 43  
 set\_parameters() (kafe.fit.Fit method), 38  
 set\_print\_level() (kafe.iminuit\_wrapper.IMinuit method), 46  
 set\_print\_level() (kafe.minuit.Minuit method), 44  
 set\_strategy() (kafe.iminuit\_wrapper.IMinuit method), 46  
 set\_strategy() (kafe.minuit.Minuit method), 44  
 set\_tolerance() (kafe.iminuit\_wrapper.IMinuit method), 46  
 show() (kafe.plot.Plot method), 41  
 show\_legend (kafe.plot.Plot attribute), 41  
 stream (module), 55  
 StreamDup (class in kafe.stream), 55

## T

tolerance (kafe.minuit.Minuit attribute), 44

## U

update\_parameter\_data()  
   (kafe.iminuit\_wrapper.IMinuit method), 46  
 update\_parameter\_data() (kafe.minuit.Minuit method), 44

## W

write() (kafe.stream.StreamDup method), 55  
 write\_formatted() (kafe.dataset.Dataset method), 32  
 write\_timestamp() (kafe.stream.StreamDup method), 55  
 write\_to\_file() (kafe.stream.StreamDup method), 55  
 write\_to\_stdout() (kafe.stream.StreamDup method), 55

## X

`x_name` (`kafe.function_tools.FitFunction` attribute), [53](#)  
`xdata` (`kafe.fit.Fit` attribute), [38](#)

## Y

`ydata` (`kafe.fit.Fit` attribute), [38](#)

## Z

`zero_pad_lower_triangle()` (in `kafe.numeric_tools` module), [55](#)