kafe2 Documentation

Release 2.7.0

J. Gäßler, C. Verstege, D. Savoiu, G. Quast

Oct 24, 2022

Contents

Welcome to **kafe2**, the *Karlsruhe Fit Environment 2*!

kafe2 is a data fitting framework originally designed for use in undergraduate physics lab courses. It provides a *Python* toolkit for fitting models to data as well as visualizing the fit results. A quick rundown of why you'd want to use *kafe2* can be found **`here https://philfitters.github.io/kafe2/ _**. The gist of it is that *kafe2* provides a simple, user-friendly interface for state-of-the-art statistical methods. It relies on Python packages such as `numpy` and `matplotlib`, and can use the *Python* interface to the minimizer *Minuit* contained in the data analysis framework *ROOT* or in the Python package *iminuit*.

The *first chapter* (page 3) of this documentation gives detailed installation instructions. The *Beginner's Guide* (page 9) explains basic *kafe2* usage to cover simple cases (both Python code and kafe2go). The *User Guide* (page 61) and the *kafe2go Guide* (page 73) describe advanced *kafe2* use with Python code or *kafe2go*. The *next chapter* (page 81) explains the mathematical foundations upon which *kafe* is built. While strictly speaking not required to use *kafe2*, reading the theory chapter is strongly recommended to understand which features to use in a state-of-the-art data analysis (regardless of whether *kafe2* or another data analysis tool is used). The *Developer Guide* (page 103) covers topics that are only relevant if you want to work on *kafe2* as a developer (still very much WIP). Finally, the *API Documentation* (page 107) provides a full description of the user-facing *kafe2* application programming interface.

1 Installing *kafe2*

Warning: *kafe2* versions 2.3.x are the latest versions which support Python 2. Python 2 support will be dropped for all future releases.

1.1 Requirements

kafe2 needs some additional Python packages. When *kafe2* is installed via *pip*, those packages are automatically installed as dependencies:

- NumPy
- Numdifftools
- SciPy
- matplotlib
- tabulate
- PyYAML

Since *kafe2* relies on *matplotlib* for graphics it might be necessary to install external programs:

- Tkinter, the default GUI used by *matplotlib*

Optionally, a function minimizer other than `scipy.optimize.minimize` can be used. *kafe2* implements interfaces to two function minimizers and will use them by default if they're installed:

- *MINUIT*, which is included in *CERN*'s data analysis package ROOT (>= 5.34), or
- iminuit (>= 1.5.2), which is independent of ROOT

1.2 Installation notes (Linux)

The easiest way to install *kafe2* is via pip, which is already included for Python >= 2.7.9. Installing via *pip* will automatically install the minimal dependencies. Please note that commands below should be run as root.

For Python 2:

```
pip2 install kafe2
```

For Python 3:

```
pip3 install kafe2
```

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

In Ubuntu/Mint/Debian, do:

```
apt-get install python-pip python3-pip
```

In Fedora/RHEL/CentOS, do:

```
yum install python2-pip python3-pip
```

or use `easy_install` (included with setuptools):

```
easy_install pip
```

You will also need to install *Tkinter* if it didn't already come with your Python distribution.

For Python 2, Ubuntu/Mint/Debian:

```
apt-get install python-tk
```

For Python 2, Fedora/RHEL/CentOS:

```
yum install tkinter
```

For Python 3, Ubuntu/Mint/Debian:

```
apt-get install python3-tk
```

For Python 3, Fedora/RHEL/CentOS:

```
yum install python3-tkinter
```

1.2.1 Optional: Install *ROOT*

Note: Starting with Ubuntu 16.10, ROOT is no longer available in the official repositories.

In older versions of Ubuntu (and related Linux distributions), ROOT and its Python bindings can be obtained via the package manager via:

```
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.

1.2.2 Optional: Install *iminuit*

iminuit is a Python wrapper for the Minuit minimizer which is independent of ROOT. This minimizer can be used instead of ROOT.

To install the *iminuit* package for Python, the Pip installer is recommended:

```
pip install iminuit
```

Note that the above command does **not** upgrade *iminuit* if it is already installed; To do this add the "–upgrade" option to the above command. *kafe2* officially supports *iminuit* 1.5.4 and the newest version of *iminuit* 2. To install the legacy version of *iminuit*, do:

```
pip install iminuit==1.5.4
```

Note: the last version of *iminuit* that was usable in combination with *Python* 2.7 was *iminuit* 1.3.10. The use of *Python* 2.7 in combination with *iminuit* is therefore not officially supported. You might also need to install the Python headers for *iminuit* to compile properly.

In Ubuntu/Mint/Debian, do:

```
apt-get install libpython2-dev libpython3-dev
```

In Fedora/RHEL/CentOS, do:

```
yum install python2-devel python3-devel
```

1.3 Installation notes (Windows)

Todo: Update and test this section

kafe2 can be installed under Windows, but requires some additional configuration.

The recommended Python distribution for working with *kafe2* 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.

1.3.1 Install *iminuit*

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

```
pip install 'iminuit'
```

1.3.2 Install *kafe2*

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

```
pip install kafe2
```

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 *kafe2*:

```
python setup.py install
```

1.3.3 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 verson 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 in 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.

2 Beginners Guide

This section covers the basics of using *kafe2* for fitting parametric models to data by showing examples. Specifically, it teaches users how to specify measurement data and uncertainties, how to specify model functions, and how to extract the fit results.

An interactive *Jupyter* Notebook teaching the usage of the *kafe2 Python* interface is available in English and German. The use of the aforementioned *Jupyter* notebooks is recommended

More detailed information for the advanced use of *kafe2* is found in the *User Guide* (page 61) and in the *kafe2go Guide* (page 73).

2.1 Basic Fitting Procedure

Generally, any fit performed by *kafe2* requires the specification of some sort of data. This uncertainties of said data usually also need to be defined in order to calculate parameter uncertainties. A specific model function can also be defined; depending on the data type there are defaults (e.g. a straight line for *xy* data).

2.1.1 Using kafe2go

Using *kafe2go* is the simplest way of performing a fit. Here all the necessary information and properties like data and uncertainties are specified in a *YAML* file (*.yml* and *.yaml* file extension). To perform the fit, simply run:

```
kafe2go path/to/fit.yml
```

2.1.2 Using Python

When using *kafe2* via a *Python* script it is possible to precisely control how fits are performed and plotted. However, because there is an inherent tradeoff between complexity and ease of use simplified interfaces for the most common use cases exist. The simplified interfaces in the documentation are functions called directly on the *kafe2* module. The code looks something like this:

```
import kafe2
kafe2.xy_fit(x_data, y_data, y_error=my_y_error)
kafe2.plot()
```

The mode complex object-oriented interface imports the objects from the *kafe2* module instead:

```
from kafe2 import XYFit, Plot
```

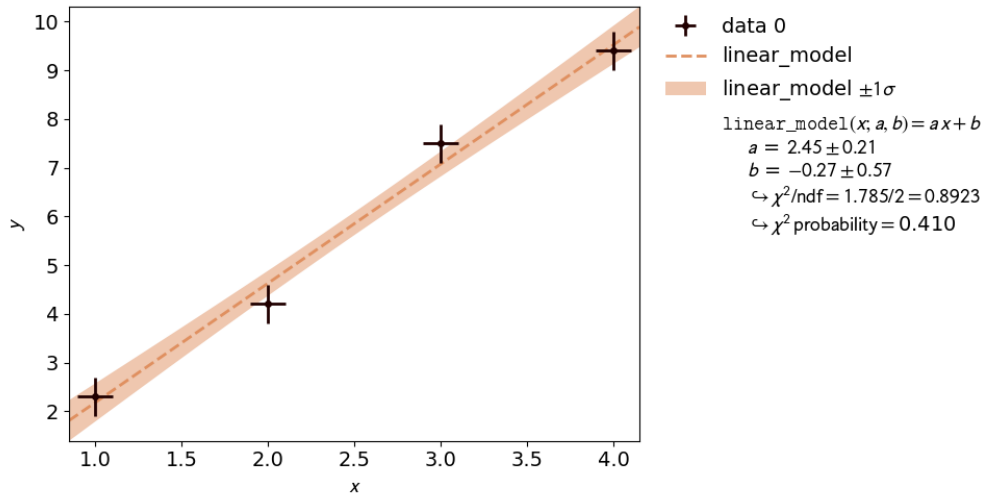
Notice how the objects are capitalized according to CamelCase while the simplified functions are written with snake_case. The difference in how the interfaces are imported is entirely arbitrary and only serves to highlight the difference for educational purposes.

If a code example contains a line similar to `data = XYContainer.from_file("data.yml")` then a *kafe2* object is loaded from a *YAML* file on disk. The corresponding *YAML* files can found in the same directory that contains the example Python script.

All example files are available on GitHub.

2.2 1. Line Fit

The simplest, and also the most common use case of a fitting framework is performing a line fit: A linear function of the form $f(x) = a*x+b$ is made to align with a series of xy data points that have some uncertainty along the x axis and the y axis. This example demonstrates how to perform such a line fit in kafe2 and how to extract the results.



2.2.1 kafe2go

To run this example, open a text editor and save the following file contents as a *YAML* file named `line_fit.yml`.

```yaml
# Example 01: Line fit
# In this example a line f(x) = a * x + b is being fitted
# to some data points with pointwise uncertainties.

# The minimal keywords needed to perform a fit are x_data,
# y_data, and y_errors.
# You might also want to define x_errors.

# Data is defined by lists:
x_data: [1.0, 2.0, 3.0, 4.0]

# For errors lists describe pointwise uncertainties.
# By default the errors will be uncorrelated.
x_errors: [0.05, 0.10, 0.15, 0.20]

# Because x_errors is equal to 5% of x_data we could have also defined it␣
↪like this:
# x_errors: 5%

# In total the following x data will be used for the fit:
# x_0: 1.0 +- 0.05
# x_1: 2.0 +- 0.10
# x_2: 3.0 +- 0.15
# x_3: 4.0 +- 0.20

# In yaml lists can also be written out like this:
y_data:
- 2.3
- 4.2
- 7.5
- 9.4

# The above is equivalent to
# y_data: [2.3, 4.2, 7.5, 9.4]

# For errors a single float gives each data point
# the same amount of uncertainty:
y_errors: 0.4

# The above is equivalent to
# y_errors: [0.4, 0.4, 0.4, 0.4]

# In total the following y data will be used for the fit:
# y_0: 2.3 +- 0.4
# y_1: 4.2 +- 0.4
# y_2: 7.5 +- 0.4
# y_3: 9.4 +- 0.4
```

Then open a terminal, navigate to the directory where the file is located and run

```
kafe2go line_fit.yml
```

2.2.2 Python

The same fit can also be performed by using a *Python* script.

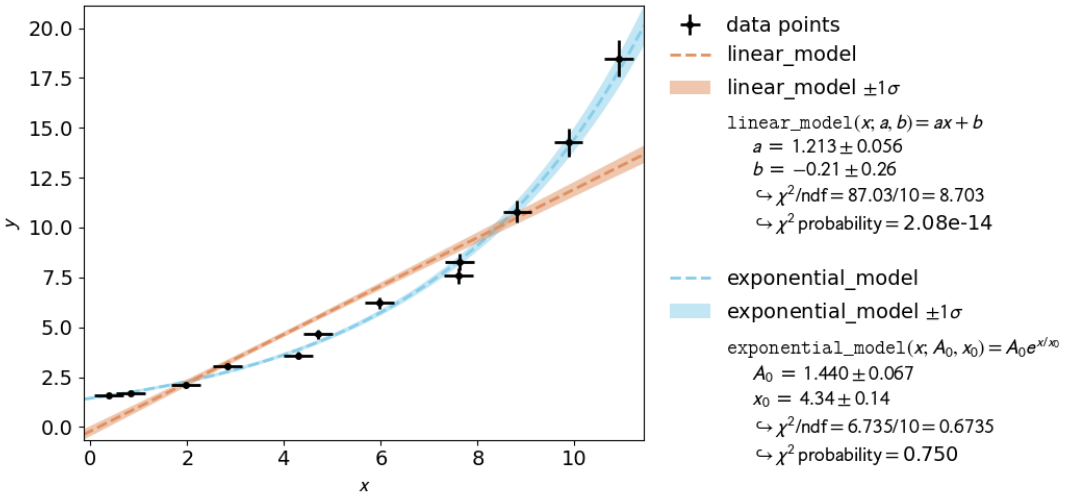
```
import kafe2

# Define or read in the data for your fit:
x_data = [1.0, 2.0, 3.0, 4.0]
y_data = [2.3, 4.2, 7.5, 9.4]
# x_data and y_data are combined depending on their order.
# The above translates to the points (1.0, 2.3), (2.0, 4.2), (3.0, 7.5), and␣
↪(4.0, 9.4).

# Important: Specify uncertainties for the data!
x_error = 0.1
y_error = 0.4

# Pass the information to kafe2:
kafe2.xy_fit(x_data, y_data, x_error=x_error, y_error=y_error)
# Because no model function was specified a line is used by default.

# Call another function to create a plot:
kafe2.plot(
    x_label="x",  # x axis label
    y_label="y",  # y axis label
    data_label="Data",  # label of data in legend
)

# For backwards compatibility with PhyPraKit kafe2 also has a function k2Fit.
# Internally this method simply uses xy_fit and plot:
# kafe2.k2Fit("line", x_data, y_data, sx=x_error, sy=y_error)
# New code should not use this function!
```

If you're performing the fit via *Python* code there should be two plots. The second plot which shows the so-called profile likelihood ("contour plot") will be explained in the next chapter.

2.3 2. Model Functions

In experimental physics a line fit will only suffice for a small number of applications. In most cases you will need a more complex model function with more parameters to accurately model physical reality. This example demonstrates how to specify arbitrary model functions for a *kafe2* fit. When a different function has to be fitted, those functions need to be defined either in the *YAML* file or the *Python* script.

The graphical output itself does not clearly indicate which of the models is acceptable. For this purpose a hypothesis test can be performed which indicates the so-called χ^2 probability - i.e. the probability of obtaining



a worse value for χ^2 at the minimum than the observed one. A higher value corresponds to a better fit. The χ^2 probability of a fit is shown inside the fit info box on the right.

An exponential function is a **non-linear** function! **Non-linear** refers to the linearity of the parameters. Any polynomial like ax^2+bx+c is a linear function of the parameters a, b and c. An exponential function $A_0e^{(x/x_0)}$ is **non-linear** in its parameter x_0. Thus the profile of χ^2 can have a non-parabolic shape. If that is the case, uncertainties of the form $a \pm \delta_a$ won't be accurate. Please refer to *3.1: Profiling* (page 18) for more information.

To see the shape of the profiles and contours, please create a contour plot of the fitted parameters. This can be done by appending the `-c` or `--contours` option to *kafe2go*. Additionally a grid can be added to the contour plots with the `--grid all` flag.

With the simplified *Python* interface a contour plot is created automatically when x errors or relative y errors are specified or when setting the keyword argument `profile=True`. The object-oriented interface uses the `ContoursProfiler` object the usage of which is shown further down.

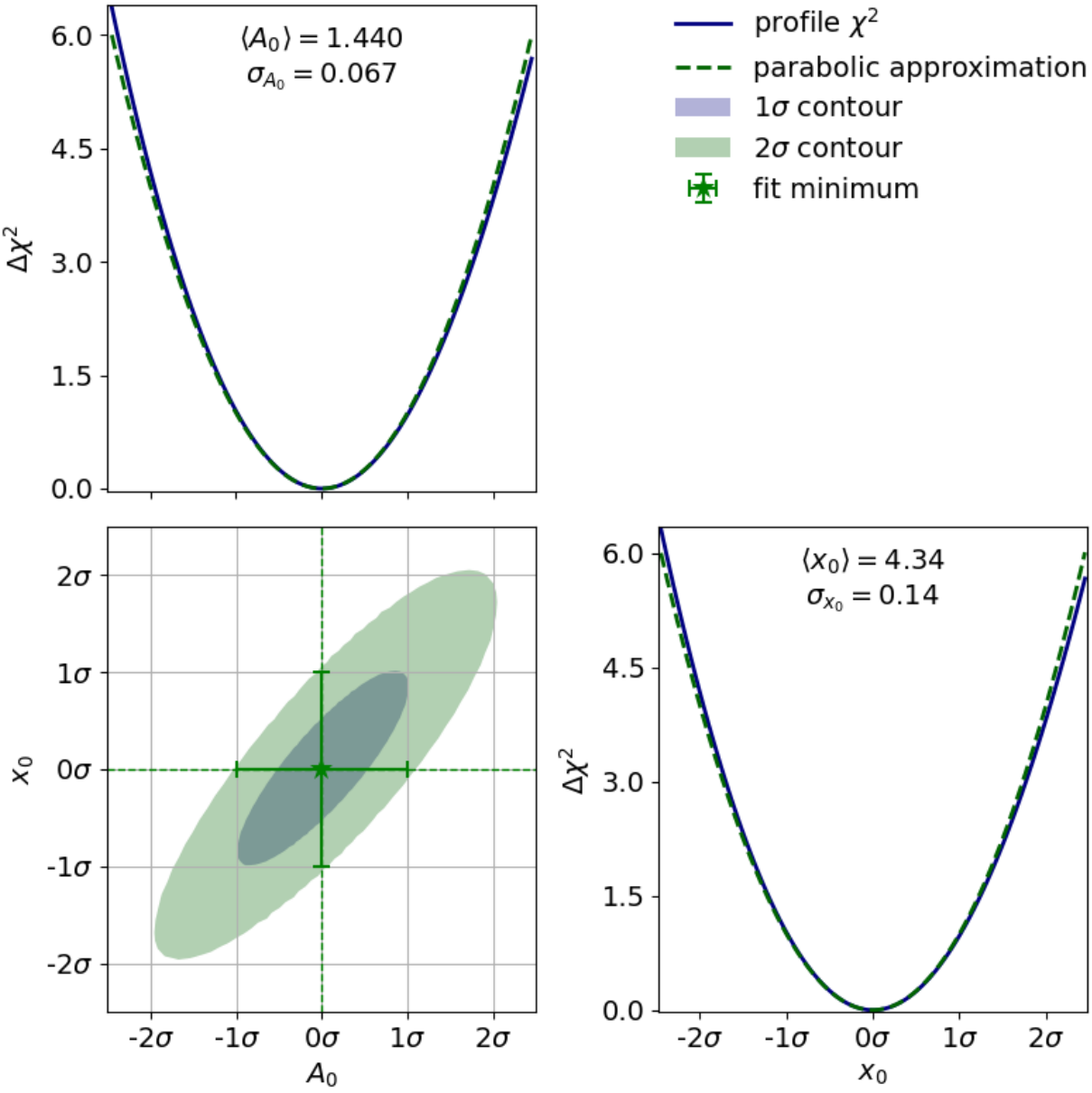
The corresponding contour plot for the exponential fit shown above looks like this:

When looking at the χ^2 profiles of the parameters, a deformation is effectively not present. In this case the fit results and uncertainties are perfectly fine and are can be used as is. See *3.1: Profiling* (page 18) for more information.

2.3.1 kafe2go

Inside a *YAML* file custom fit functions can be defined with the `model_function` keyword. The default way to define said model functions is to write a function *Python* function. *NumPy* and *SciPy* functions are supported without extra import statements, as shown in the example. Alternatively model functions can be defined via *SymPy* (symbolic Python). The part of the model function left of the string `->` is interpreted as *SymPy* symbols, the part on the right is interpreted as a *SymPy* expression using said symbols. The leftmost part of the string up to `:` is interpreted as the model function name (can be omitted).

For more advanced fit functions, consider using *kafe2* inside a *Python* script.


profile χ^2
parabolic approximation
1σ contour
2σ contour
fit minimum
$\langle A_0 \rangle = 1.440$
$\sigma_{A_0} = 0.067$
$\langle x_0 \rangle = 4.34$
$\sigma_{x_0} = 0.14$
$\Delta\chi^2$
x_0
A_0
6.0
4.5
3.0
1.5
0.0
2σ
1σ
0σ
-1σ
-2σ

```
x_data:
- 0.3811286952593707
- 0.8386314422752791
- 1.965701211114396
- 2.823689293793774
- 4.283116179196964
- 4.697874987903564
- 5.971438461333021
- 7.608558039032569
- 7.629881032308029
- 8.818924700702548
- 9.873903963026425
- 10.913590565136278
x_errors:
- correlation_coefficient: 0.0
  error_value: 0.3  # use absolute errors, in this case always 0.3
  relative: false
  type: simple
y_data:
- 1.604521233331755
- 1.6660578688633165
- 2.1251504836493296
- 3.051883842283453
- 3.5790120649006685
- 4.654148130730669
- 6.213711922872129
- 7.576981533273081
- 8.274440603191387
- 10.795366227038528
- 14.272404187046607
- 18.48681513824193
y_errors:
- correlation_coefficient: 0.0
  error_value: 0.05
  relative: true  # use relative errors, in this case 5% of each value
  type: simple

model_function: |
  def exponential_model(x, A_0=1., x_0=5.):
    return A_0 * np.exp(x/x_0)

# If SymPy is installed the model function can also be defined like this:
# model_function: "exponential_model: x A_0 x_0 -> A_0 * exp(x / x_0)"
```

```
x_data:
- 0.3811286952593707
- 0.8386314422752791
- 1.965701211114396
- 2.823689293793774
- 4.283116179196964
- 4.697874987903564
- 5.971438461333021
```

(continues on next page)

(continued from previous page)

```
- 7.608558039032569
- 7.629881032308029
- 8.818924700702548
- 9.873903963026425
- 10.913590565136278
x_errors:
- correlation_coefficient: 0.0
  error_value: 0.3  # use absolute errors, in this case always 0.3
  relative: false
  type: simple
y_data:
- 1.604521233331755
- 1.6660578688633165
- 2.1251504836493296
- 3.051883842283453
- 3.5790120649006685
- 4.654148130730669
- 6.213711922872129
- 7.576981533273081
- 8.274440603191387
- 10.795366227038528
- 14.272404187046607
- 18.48681513824193
y_errors:
- correlation_coefficient: 0.0
  error_value: 0.05
  relative: true  # use relative errors, in this case 5% of each value
  type: simple

model_function: |
  def linear_model(x, a, b):
      return a * x + b

# If SymPy is installed the model function can also be defined like this:
# model_function: "linear_model: x a b -> a * x + b"
```

To use multiple input files with kafe2go, simply run

```
kafe2go path/to/fit1.yml path/to/fit2.yml
```

To plot the fits in two separate figures append the `--separate` flag to the kafe2go command.

```
kafe2go path/to/fit1.yml path/to/fit2.yml --separate
```

For creating a contour plot, simply add `-c` to the command line. This will create contour plots for all given fits.

```
kafe2go path/to/fit1.yml path/to/fit2.yml --separate -c
```

2.3.2 Python

When using *Python* multiple model functions can be defined in the same file. They are plotted together by first calling `kafe2.xy_fit` multiple times and then calling `kafe2.plot`.

```
import numpy as np
import kafe2


# To define a model function for kafe2 simply write it as a Python function.
# Important: The first argument of the model function is interpreted as the␣
↪independent variable
#     of the fit. It is not being modified during the fit and it's the␣
↪quantity represented by
#     the x axis of our fit.

# Our first model is a simple linear function:
def linear_model(x, a, b):
    return a * x + b


# If SymPy is installed you can also define the model like this:
# linear_model = "linear_model: x a b -> a * x + b"


# Our second model is a simple exponential function.
# The kwargs in the function header specify parameter defaults.
def exponential_model(x, A_0=1., x_0=5.):
    return A_0 * np.exp(x/x_0)


# If SymPy is installed you can also define the model like this:
# exponential_model = "exponential_model: x A_0 x_0=5.0 -> A_0 * exp(x / x_0)"


x_data = [0.38, 0.83, 1.96, 2.82, 4.28, 4.69, 5.97, 7.60, 7.62, 8.81, 9.87,␣
↪10.91]
y_data = [1.60, 1.66, 2.12, 3.05, 3.57, 4.65, 6.21, 7.57, 8.27, 10.79, 14.27,␣
↪18.48]
x_error = 0.3
y_error_rel = 0.05

# kafe2.xy_fit needs to be called twice to do two fits:
kafe2.xy_fit(x_data, y_data, model_function=linear_model, x_error=x_error, y_
↪error_rel=y_error_rel)
kafe2.xy_fit(x_data, y_data, model_function=exponential_model,
             x_error=x_error, y_error_rel=y_error_rel, profile=True)
# Make sure to specify profile=True whenever you use a nonlinear model␣
↪function.
# A model function is linear if it is a linear function of each of its␣
↪parameters.
# The model function does not need to be a linear function of the independent␣
↪variable x.
```

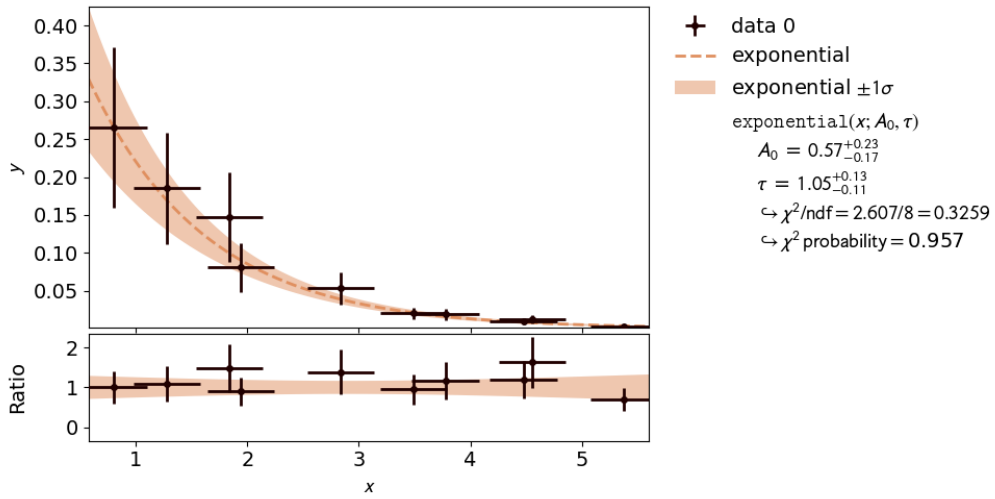
(continues on next page)

(continued from previous page)

```
# Examples: all polynomial model functions are linear, trigonometric␣
↪functions are nonlinear.

# To specify that you want a plot of the last two fits pass -2 as the first␣
↪argument:
kafe2.plot(
    -2,

    # Uncomment the following line to use different names for the parameters:
    # parameter_names=dict(x="t", a=r"\alpha", b=r"\beta", A_0="I_0", x_0="t_0
↪"),
    # Use LaTeX for special characters like greek letters.

    # When Python functions are used as custom model functions kafe2 does not␣
↪know
    # how to express them as LaTeX. The LaTeX can be manually defined like␣
↪this:
    model_expression=["{a}{x} + {b}", "{A_0} e^{{{x}/{x_0}}}"]
    # Parameter names have to be put between {}. To get {} for LaTex double␣
↪them like {{ or }}.
    # When using SymPy to define model function the LaTeX expression can be␣
↪derived automatically.
)
```

2.4 3.1: Profiling

Very often, when the fit model is a non-linear function of the parameters, the χ^2 function is not parabolic around the minimum. A very common example of such a case is an exponential function parametrized as shown in this example.



In the case of a nonlinear fit, the minimum of a χ^2 cost function is no longer shaped like a parabola (with a model parameter on the x axis and χ^2 on the y axis). Now, you might be wondering why you should care about

the shape of the χ^2 function. The reason why it's important is that the common notation of $p \pm \sigma$ for fit results is only valid for a parabola-shaped cost function. If your likelihood function is distorted it will also affect your fit results!

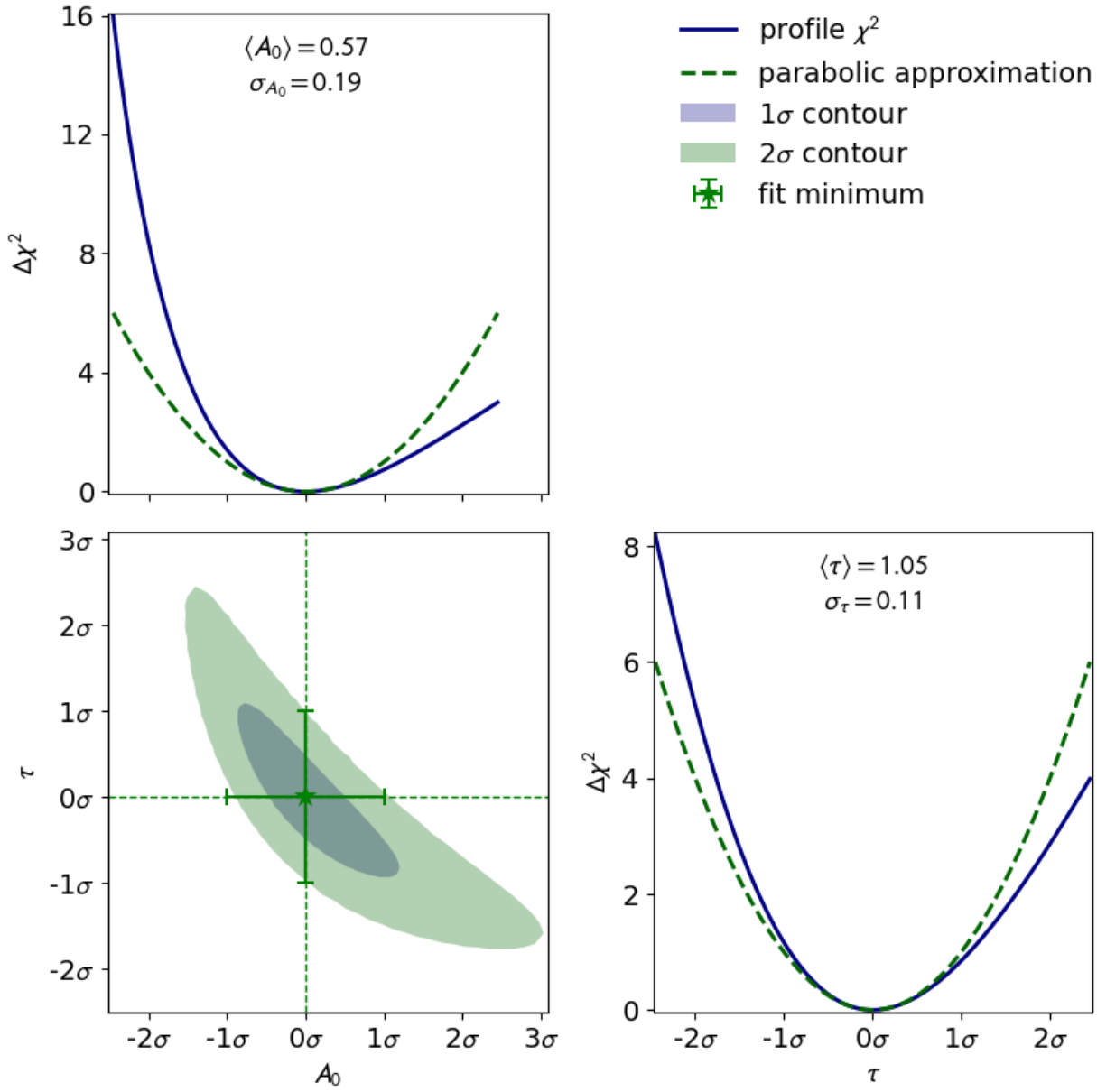
Luckily nonlinear fits oftentimes still produce meaningful fit results as long as the distortion is not too big - you just need to be more careful during the evaluation of your fit results. A common approach for handling nonlinearity is to trace the profile of the cost function (in this case χ^2) in either direction of the cost function minimum and find the points at which the cost function value has increased by a specified amount relative to the cost function minimum. In other words, two cuts are made on either side of the cost function minimum at a specified height. The two points found with this approach span a confidence interval for the fit parameter around the cost function minimum. The confidence level of the interval depends on how high you set the cuts for the cost increase relative to the cost function minimum. The one sigma interval described by conventional parameter errors is achieved by a cut at the fit minimum plus $1^2 = 1$ and has a confidence level of about 68%. The two sigma interval is achieved by a cut at the fit minimum plus $2^2 = 4$ and has a confidence level of about 95%, and so on. The one sigma interval is commonly described by what is called asymmetric errors: the interval limits are described relative to the cost function minimum as $p^{+\sigma_{\text{up}}}_{-\sigma_{\text{down}}}$.

In addition to non-linear function, the usage of x data errors leads to a non-linear fits as well. This is shown in *3.3: x-Errors:* (page 27).

2.4.1 kafe2go

To display asymmetric parameter uncertainties use the flag `-a`. In addition the profiles and contours can be shown by using the `-c` flag. In the *Python* example a ratio between the data and model function is shown below the plot. This can be done by appending the `-r` flag to *kafe2go*.

```
# This is an example for a non-linear fit
# Please run with 'kafe2go -a -c' to show asymmetric uncertainties and the␣
↪contour profiles

x_data: [0.8018943, 1.839664, 1.941974, 1.276013, 2.839654, 3.488302, 3.
↪775855, 4.555187, 4.477186, 5.376026]
x_errors: 0.3
y_data: [0.2650644, 0.1472682, 0.08077234, 0.1850181, 0.05326301, 0.01984233,␣
↪0.01866309, 0.01230001, 0.009694612,
         0.002412357]
y_errors: [0.1060258, 0.05890727, 0.03230893, 0.07400725, 0.0213052, 0.
↪00793693, 0.007465238, 0.004920005, 0.003877845,
           0.0009649427]

model_function: |
  def exponential(x, A_0=1, tau=1):
      return A_0 * np.exp(-x/tau)
```


profile χ^2
parabolic approximation
1σ contour
2σ contour
fit minimum
$\langle A_0 \rangle = 0.57$
$\sigma_{A_0} = 0.19$
$\Delta\chi^2$
τ
A_0
$\langle \tau \rangle = 1.05$
$\sigma_\tau = 0.11$

2.4.2 Python

From this point on the examples use the object-oriented *Python* interface. For a quick intorduction consider this code that is mostly equivalent to the very first example of a straight line with *xy* errors:

```
from kafe2 import XYContainer, Fit, Plot

# Create an XYContainer object to hold the xy data for the fit:
xy_data = XYContainer(x_data=[1.0, 2.0, 3.0, 4.0],
                      y_data=[2.3, 4.2, 7.5, 9.4])
# x_data and y_data are combined depending on their order.
# The above translates to the points (1.0, 2.3), (2.0, 4.2), (3.0, 7.5), and␣
↪(4.0, 9.4).

# Important: Specify uncertainties for the data:
xy_data.add_error(axis='x', err_val=0.1)
xy_data.add_error(axis='y', err_val=0.4)

xy_data.label = 'Data'  # How the data is called in plots

# Create an XYFit object from the xy data container.
# By default, a linear function f=a*x+b will be used as the model function.
line_fit = Fit(data=xy_data)

# Perform the fit: Find values for a and b that minimize the
#     difference between the model function and the data.
line_fit.do_fit()  # This will throw a warning if no errors were specified.

# Optional: Print out a report on the fit results on the console.
line_fit.report()
# With kafe2.xy_fit this gets printed to a file.

# Optional: Create a plot of the fit results using Plot.
plot = Plot(fit_objects=line_fit)  # Create a kafe2 plot object.
plot.x_label = 'x'  # Set x axis label.
plot.y_label = 'y'  # Set y axis label.
plot.plot()  # Do the plot.

plot.save()  # Saves the plot to file 'fit.png' .
# plot.save('my_fit.pdf')  # Saves the plot to a different file / with a␣
↪different file extension.

# Show the fit result.
plot.show()  # Just a convenience wrapper for matplotlib.pyplot.show() .
# NOTE: Calling matplotlib.pyplot.show() closes all figures by default so␣
↪call this AFTER saving.

# Alternatively you could still use the function kafe.plot like this:
# from kafe2 import plot  # Notice that "plot" is not capitalized. "Plot" is␣
↪the plot object.
# kafe2.plot(line_fit)  # Pass the fit object to the plot function.
```

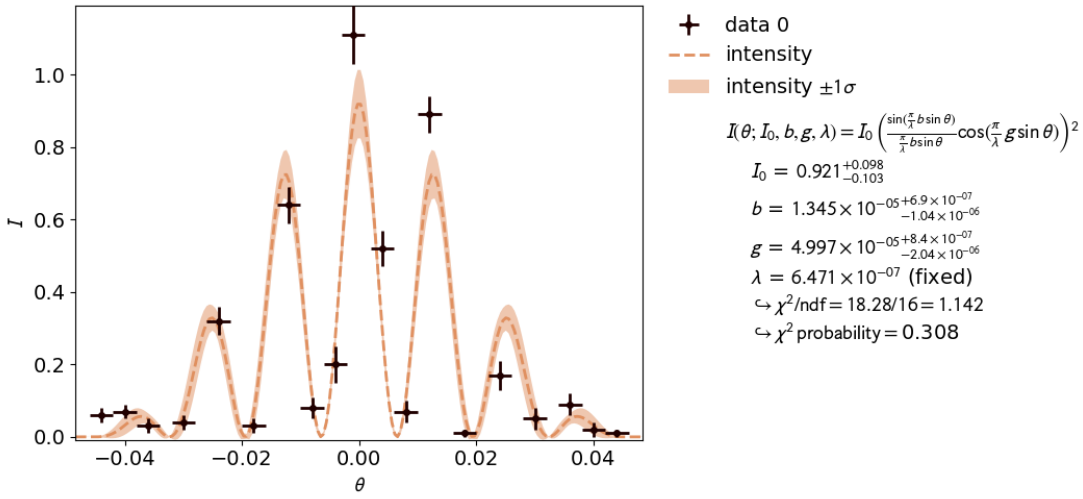
Now for the actual example. The relevant lines to display asymmetric uncertainties and to create the contour

plot are highlighted in the code example below.

```
cost function minimum as par_value+par_err_up-par_err_down.

Note: from this point on the examples use kafe2 objects rather than functions␣
↪like kafe2.xy_fit.
For an introduction to kafe2 objects look at the file 00_object_oriented_
↪programming.py
"""

import numpy as np
from kafe2 import Fit, Plot, ContoursProfiler

def exponential(x, A_0=1, tau=1):
    return A_0 * np.exp(-x/tau)

# define the data as simple Python lists
x = [8.018943e-01, 1.839664e+00, 1.941974e+00, 1.276013e+00, 2.839654e+00, 3.
↪488302e+00,
     3.775855e+00, 4.555187e+00, 4.477186e+00, 5.376026e+00]
xerr = 3.000000e-01
y = [2.650644e-01, 1.472682e-01, 8.077234e-02, 1.850181e-01, 5.326301e-02, 1.
↪984233e-02,
     1.866309e-02, 1.230001e-02, 9.694612e-03, 2.412357e-03]
yerr = [1.060258e-01, 5.890727e-02, 3.230893e-02, 7.400725e-02, 2.130520e-02,␣
↪7.936930e-03,
        7.465238e-03, 4.920005e-03, 3.877845e-03, 9.649427e-04]

# create a fit object from the data arrays
fit = Fit(data=[x, y], model_function=exponential)
fit.add_error(axis='x', err_val=xerr)  # add the x-error to the fit
fit.add_error(axis='y', err_val=yerr)  # add the y-errors to the fit

fit.do_fit()  # perform the fit
fit.report(asymmetric_parameter_errors=True)  # print a report with␣
↪asymmetric uncertainties

# Optional: create a plot
plot = Plot(fit)
plot.plot(asymmetric_parameter_errors=True, ratio=True)  # ratio = data /␣
↪model function

# Optional: create the contours profiler
cpf = ContoursProfiler(fit)
cpf.plot_profiles_contours_matrix()  # plot the contour profile matrix for␣
↪all parameters

plot.show()
```

2.5 3.2: Double Slit

A real world example, when asymmetric parameter uncertainties are needed is the double slit experiment. Here the model function is highly non-linear.



This is also reflected by the highly distorted contours.

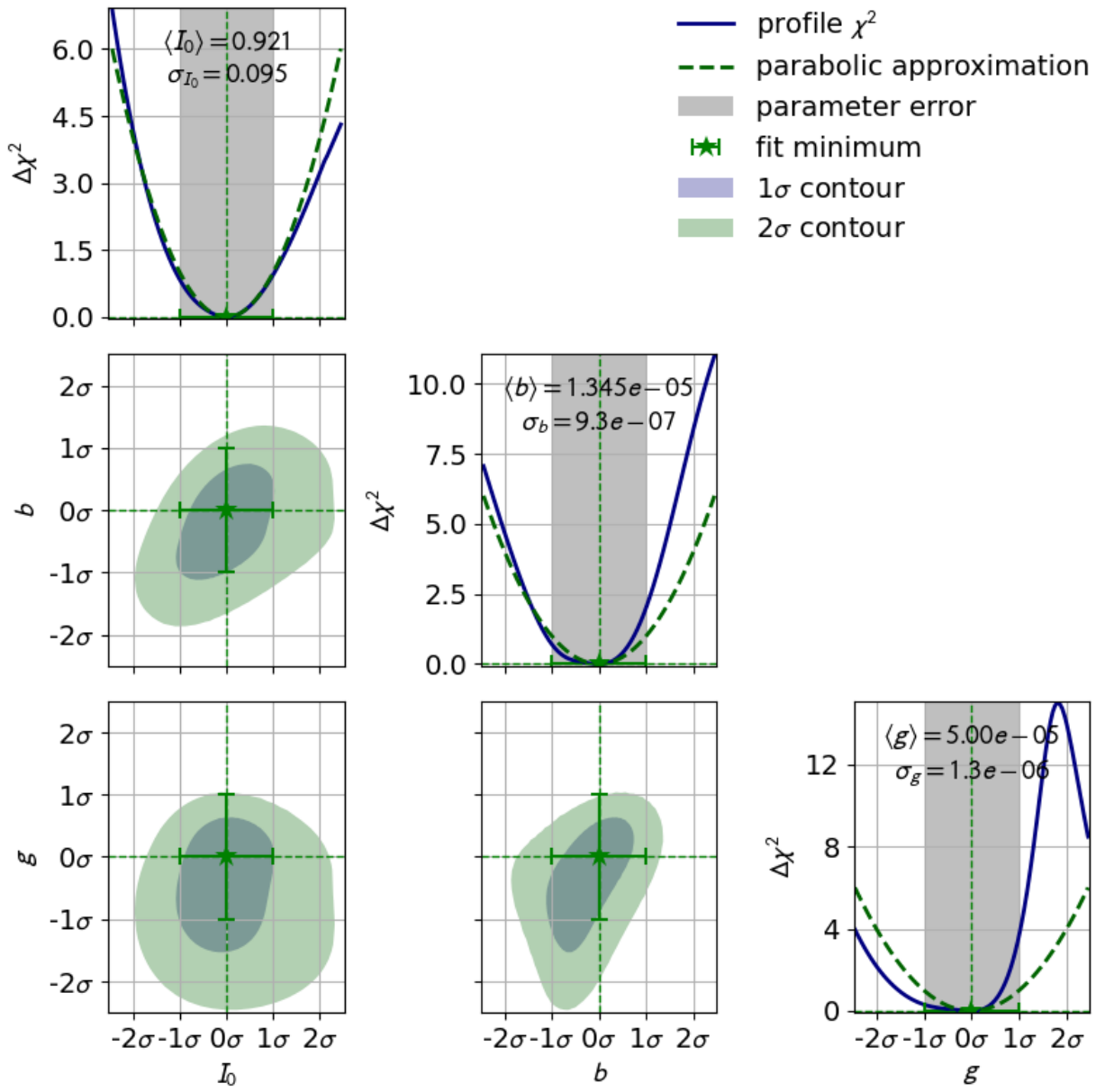
2.5.1 kafe2go

A *YAML* file for this example is available on GitHub. It works perfectly fine with kafe2go, but is not edited to a compact form. It is in fact a direct dump from the fit object used in the python script. *kafe2* supports writing fits and datasets to a kafe2go compatible file with the `data.to_file("my.yml")` and `fit.to_file("filename.yml")` methods as well as reading them with the corresponding classes like `XYContainer.from_file("my.yml")` and `XYFit.from_file("filename.yml")` as shown in the *Python* example.

2.5.2 Python

```
import numpy as np
from kafe2.fit import XYContainer, Fit, Plot
from kafe2.fit.tools import ContoursProfiler


def _generate_dataset(output_filename='02_double_slit_data.yml'):
    """
    Create an XYContainer holding the measurement data
    and the errors and write it to a file.
    """

    xy_data = [[  # x data: position
                -0.044, -0.040, -0.036, -0.030, -0.024, -0.018, -0.012, -0.
↪008, -0.004, -0.001,
```

(continues on next page)


profile χ^2
parabolic approximation
parameter error
fit minimum
1σ contour
2σ contour
$\langle I_0 \rangle = 0.921$
$\sigma_{I_0} = 0.095$
$\langle b \rangle = 1.345e-05$
$\sigma_b = 9.3e-07$
$\langle g \rangle = 5.00e-05$
$\sigma_g = 1.3e-06$
$\Delta\chi^2$
I_0
b
g

(continued from previous page)

```
                0.004,  0.008,  0.012,  0.018,  0.024,  0.030,  0.036,  0.
↪040,  0.044],
               [  # y data: light intensity
                0.06, 0.07, 0.03, 0.04, 0.32, 0.03, 0.64, 0.08, 0.20, 1.11, 0.
↪52, 0.07, 0.89, 0.01,
                0.17, 0.05, 0.09, 0.02, 0.01]]

    data = XYContainer(x_data=xy_data[0], y_data=xy_data[1])

    data.add_error('x', 0.002, relative=False, name='x_uncor_err')
    data.add_error(
        'y',
        [0.02, 0.02, 0.02, 0.02, 0.04, 0.02, 0.05, 0.03, 0.05, 0.08, 0.05, 0.
↪03, 0.05, 0.01, 0.04,
         0.03, 0.03, 0.02, 0.01],
        relative=False,
        name='y_uncor_err'
    )

    data.to_file(output_filename)

# Uncomment this to re-generate the dataset:
# _generate_dataset()


def intensity(theta, I_0, b, g, varlambda):
    """
    In this example our model function is the intensity of diffracted light␣
↪as described by the
    Fraunhofer equation.
    :param theta: angle at which intensity is measured
    :param I_0: intensity amplitude
    :param b: width of a single slit
    :param g: distance between the two slits
    :param varlambda: wavelength of the laser light
    :return: intensity of the diffracted light
    """
    single_slit_arg = np.pi * b * np.sin(theta) / varlambda
    single_slit_interference = np.sin(single_slit_arg) / single_slit_arg
    double_slit_interference = np.cos(np.pi * g * np.sin(theta) / varlambda)
    return I_0 * single_slit_interference ** 2 * double_slit_interference ** 2


# Read in the measurement data from the file generated above:
data = XYContainer.from_file("02_double_slit_data.yml")

# Create fit from data container:
fit = Fit(data=data, model_function=intensity, minimizer="iminuit")

# Optional: assign LaTeX names for prettier fit info box:
fit.assign_model_function_latex_name('I')
fit.assign_model_function_latex_expression(
```

(continues on next page)

(continued from previous page)

```
    r"{I_0}\,\left(\frac{{\sin(\frac{{\pi}}{{{varlambda}}}\,b\,\sin{{{theta}}}
↪)}}"
    r"{{\frac{{\pi}}{{{varlambda}}}\,b\,\sin{{{theta}}}}}"
    r"\cos(\frac{{\pi}}{{{varlambda}}}\,g\,\sin{{{theta}}})\right)^2"
)

# Limit parameters to positive values to better model physical reality:
eps = 1e-8
fit.limit_parameter('I_0', lower=eps)
fit.limit_parameter('b', lower=eps)
fit.limit_parameter('g', lower=eps)

# Set fit parameters to near guesses to improve convergence:
fit.set_parameter_values(I_0=1., b=20e-6, g=50e-6)

# The fit parameters have no preference in terms of values.
# Their profiles are highly distorted, indicating a very non-linear fit.
# You can try constraining them via external measurements to make the fit␣
↪more linear:
# f.add_parameter_constraint('b', value=13.5e-6, uncertainty=1e-6)
# f.add_parameter_constraint('g', value=50e-6, uncertainty=1e-6)

# Fix the laser wavelength to 647.1 nm (krypton laser) since its uncertainty␣
↪is negligible:
fit.fix_parameter('varlambda', value=647.1e-9)

# Fit objects can also be saved to files:
fit.to_file('02_double_slit.yml')
# The generated file can be used as input for kafe2go.

# Alternatively you could load it back into code via:
# f = XYFit.from_file('02_double_slit.yml')

fit.do_fit()

cpf = ContoursProfiler(fit)
cpf.plot_profiles_contours_matrix(parameters=['I_0', 'b', 'g'],
                                  show_grid_for='all',
                                  show_fit_minimum_for='all',
                                  show_error_span_profiles=True,
                                  show_legend=True,
                                  show_parabolic_profiles=True,
                                  show_ticks_for='all',
                                  contour_naming_convention='sigma',
                                  label_ticks_in_sigma=True)

# To see the fit results, plot using Plot:
p = Plot(fit_objects=fit)
p.y_label = r"$I$"
p.plot(asymmetric_parameter_errors=True)

# Show the fit results:
```

(continues on next page)

(continued from previous page)

```
p.show()
```

2.6 3.3: x-Errors:

In addition to non-linear function, the usage of x data errors leads to a non-linear fits as well. kafe2 fits support the addition of x data errors - in fact we've been using them since the very first example. To take them into account the x errors are converted to y errors via multiplication with the derivative of the model function. In other words, kafe2 fits extrapolate the derivative of the model function at the x data values and calculate how a difference in the x direction would translate to the y direction. Unfortunately this approach is not perfect though. Since we're extrapolating the derivative at the x data values, we will only receive valid results if the derivative doesn't change too much at the scale of the x error. Also, since the effective y error has now become dependent on the derivative of the model function it will vary depending on our choice of model parameters. This distorts our likelihood function - the minimum of a χ^2 cost function will no longer be shaped like a parabola (with a model parameter on the x axis and χ^2 on the y axis).

To demonstrate this, the second file `x_errors` will perform a line fit with much bigger uncertainties on the x axis than on the y axis. The non-parabolic shape can be seen in the one-dimensional profile scans.

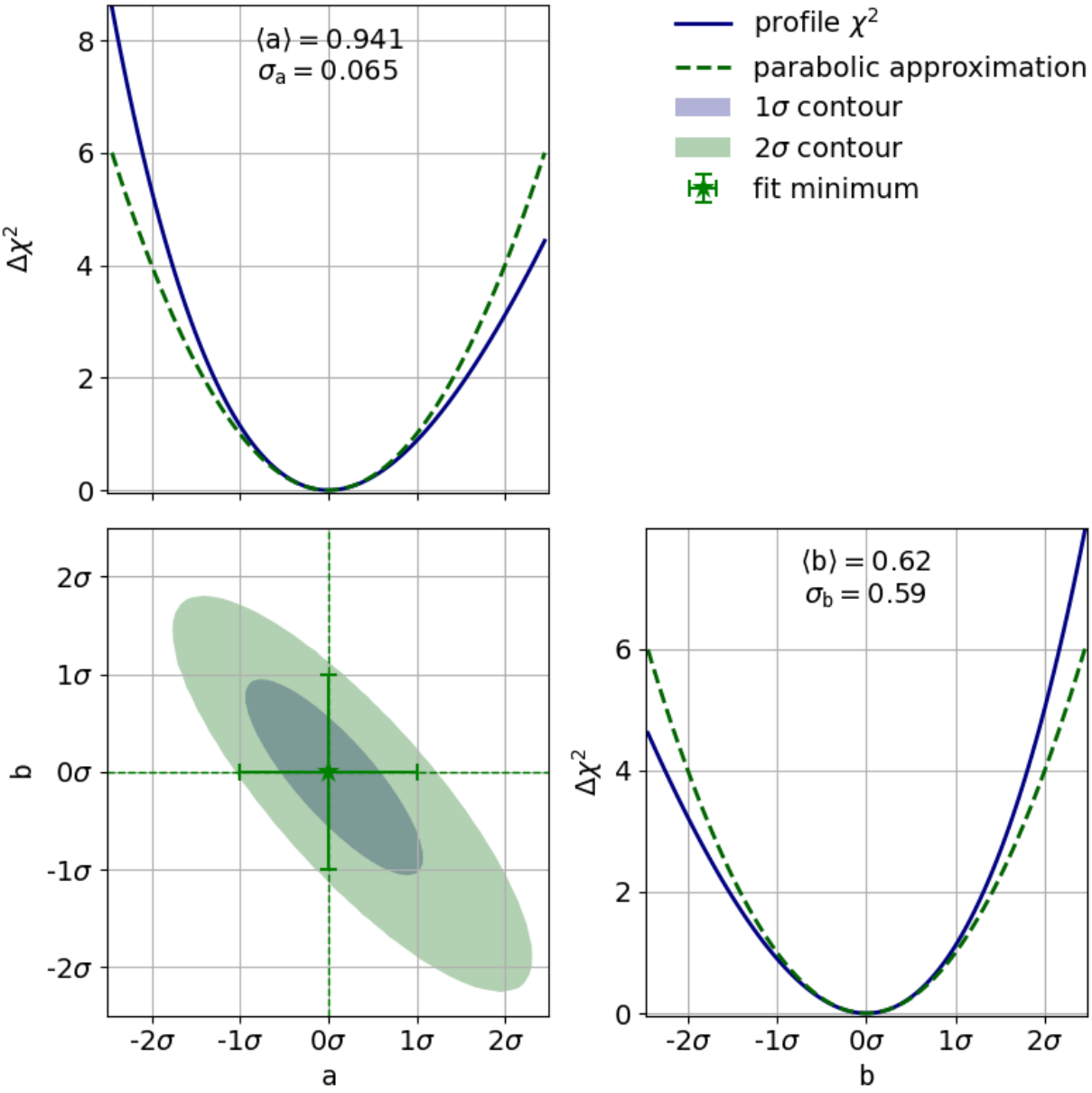
The effects of this deformation are explained in *3.1: Profiling* (page 18).

2.6.1 kafe2go

Keep in mind that *kafe2go* will perform a line fit if no fit function has been specified. In order do add a grid to the contours, run *kafe2go* with the `--grid all` flag. So to plot with asymmetric errors, the profiles and contour as well as a grid run `kafe2go x_errors.yml -a -c --grid all`

```
# kafe2 XYContainer yaml representation written by johannes on 23.07.2019,␣
↪16:57.
type: xy
x_data:
- 0.0
- 1.0
- 2.0
- 3.0
- 4.0
- 5.0
- 6.0
- 7.0
- 8.0
- 9.0
- 10.0
- 11.0
- 12.0
- 13.0
- 14.0
- 15.0
y_data:
```

(continues on next page)


⟨a⟩ = 0.941
σa = 0.065
profile χ²
parabolic approximation
1σ contour
2σ contour
fit minimum
Δχ²
a
b
⟨b⟩ = 0.62
σb = 0.59

(continued from previous page)

```
- 0.2991126116324785
- 2.558050235697161
- 1.2863728164289798
- 3.824686039107114
- 2.843373362329926
- 5.461953737679532
- 6.103072604470123
- 8.166562633164254
- 8.78250807001851
- 8.311966900704014
- 8.980727588512268
- 11.144142620167695
- 11.891326143534158
- 12.126133797209802
- 15.805993018808897
- 15.3488445186788
# The x errors are much bigger than the y errors. This will strongly distort␣
↪the likelihood function.
x_errors: 1.0
y_errors: 0.1
```

2.6.2 Python

The example to show that uncertainties on the x axis can cause a non-linear fit uses the *YAML* dataset given in the *kafe2go* section.

```
from kafe2 import XYContainer, Fit, Plot
from kafe2.fit.tools import ContoursProfiler

# Construct a fit with data loaded from a yaml file. The model function is␣
↪the default of f(x) = a * x + b
nonlinear_fit = Fit(data=XYContainer.from_file('03_x_errors.yml'))

# The x errors are much bigger than the y errors. This will cause a␣
↪distortion of the likelihood function.
nonlinear_fit.add_error('x', 1.0)
nonlinear_fit.add_error('y', 0.1)

# Perform the fit.
nonlinear_fit.do_fit()

# Optional: Print out a report on the fit results on the console.
# Note the asymmetric_parameter_errors flag
nonlinear_fit.report(asymmetric_parameter_errors=True)

# Optional: Create a plot of the fit results using Plot.
# Note the asymmetric_parameter_errors flag
plot = Plot(nonlinear_fit)
plot.plot(fit_info=True, asymmetric_parameter_errors=True)
```

(continues on next page)

(continued from previous page)

```
# Optional: Calculate a detailed representation of the profile likelihood
# Note how the actual chi2 profile differs from the parabolic approximation␣
↪that you would expect with a linear fit.
profiler = ContoursProfiler(nonlinear_fit)
profiler.plot_profiles_contours_matrix(show_grid_for='all')

plot.show()
```

2.7 4: Constraints

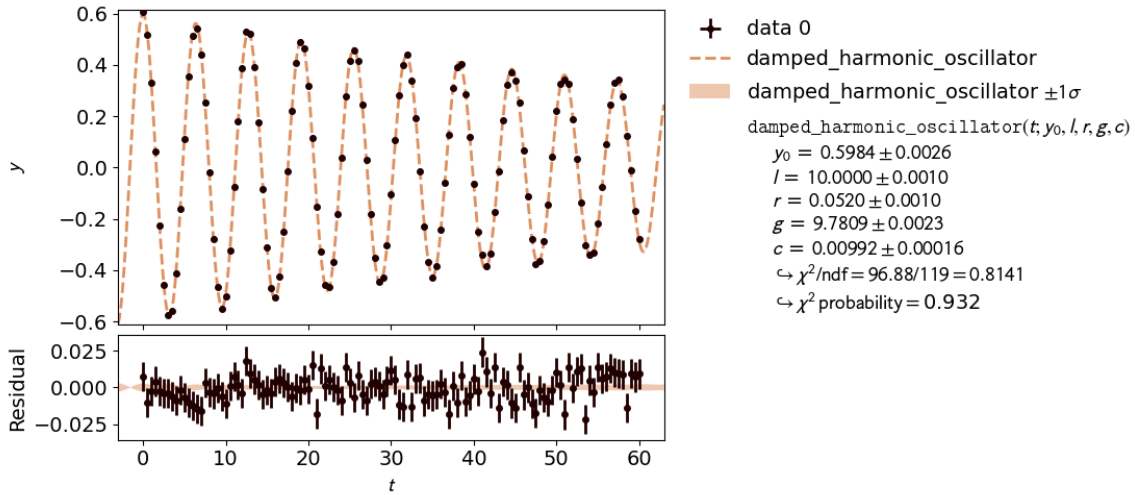
The models used to describe physical phenomena usually depend on a multitude of parameters. However, for many experiments only one of the parameters is of actual interest to the experimenter. Still, because model parameters are generally not uncorrelated the experimenter has to factor in the nuisance parameters for their estimation of the parameter of interest.

Historically this has been done by propagating the uncertainties of the nuisance parameters onto the y-axis of the data and then performing a fit with those uncertainties. Thanks to computers, however, this process can also be done numerically by applying parameter constraints. This example demonstrates the usage of those constraints in the kafe2 framework.

More specifically, this example will simulate the following experiment:

A steel ball of radius r has been connected to the ceiling by a string of length l, forming a pendulum. Due to earth's gravity providing a restoring force this system is a harmonic oscillator. Because of friction between the steel ball and the surrounding air the oscillator is also damped by the viscous damping coefficient c.

The goal of the experiment is to determine the local strength of earth's gravity g. Since the earth is shaped like an ellipsoid the gravitational pull varies with latitude: it's strongest at the poles with $g_p = 9.832\,\mathrm{m/s^2}$ and it's weakest at the equator with $g_e = 9.780\,\mathrm{m/s^2}$. For reference, at Germany's latitude g lies at approximately $9.81\,\mathrm{m/s^2}$.



2.7.1 kafe2go

Parameter constraints are straightforward to use with *kafe2go*. After defining the model function parameter constraints can be set. Simple gaussian constraints can be defined with the parameter name followed by the required information as highlighted in the example. For more information on parameter constraints via a covariance matrix, please refer to *Parameter Constraints* (page 77).

```
x_data: [0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5,
 ↪ 7.0, 7.5, 8.0, 8.5,
          9.0, 9.5, 10.0, 10.5, 11.0, 11.5, 12.0, 12.5, 13.0, 13.5, 14.0, 14.5,
 ↪ 15.0, 15.5, 16.0,
          16.5, 17.0, 17.5, 18.0, 18.5, 19.0, 19.5, 20.0, 20.5, 21.0, 21.5, 22.
 ↪0, 22.5, 23.0, 23.5,
          24.0, 24.5, 25.0, 25.5, 26.0, 26.5, 27.0, 27.5, 28.0, 28.5, 29.0, 29.
 ↪5, 30.0, 30.5, 31.0,
          31.5, 32.0, 32.5, 33.0, 33.5, 34.0, 34.5, 35.0, 35.5, 36.0, 36.5, 37.
 ↪0, 37.5, 38.0, 38.5,
          39.0, 39.5, 40.0, 40.5, 41.0, 41.5, 42.0, 42.5, 43.0, 43.5, 44.0, 44.
 ↪5, 45.0, 45.5, 46.0,
          46.5, 47.0, 47.5, 48.0, 48.5, 49.0, 49.5, 50.0, 50.5, 51.0, 51.5, 52.
 ↪0, 52.5, 53.0, 53.5,
          54.0, 54.5, 55.0, 55.5, 56.0, 56.5, 57.0, 57.5, 58.0, 58.5, 59.0, 59.
 ↪5, 60.0]
x_errors: 0.001

y_data: [0.6055317575993914, 0.5169107902610475, 0.3293169273559535, 0.
 ↪06375859733814328,
          -0.22640917641452488, -0.4558011426302008, -0.5741274235093591, -0.
 ↪5581292464350779,
          -0.4123919729458466, -0.16187396188197636, 0.11128294449725282, 0.
 ↪3557032748758762,
          0.5123991632742368, 0.5426212570679745, 0.441032201346871, 0.
 ↪2530635717934812,
          -0.018454841476861578, -0.27767282674995675, -0.4651897678670606, -0.
 ↪5496969201663507,
          -0.5039723480612176, -0.32491507193649194, -0.07545061334934718, 0.
 ↪17868596924060803,
          0.38816743920465974, 0.5307390166315159, 0.5195411750524918, 0.
 ↪3922352276601664,
          0.1743316249378316, -0.08489324112691513, -0.31307737388477896, -0.
 ↪47061848549403956,
          -0.5059137659253059, -0.4255733083473707, -0.24827692102709906, -0.
 ↪013334081799533964,
          0.21868390151118114, 0.40745363855032496, 0.4900638626421252, 0.
 ↪4639947376885959,
          0.3160179896333209, 0.11661884242058931, -0.15370888958327286, -0.
 ↪32616191426973545,
          -0.45856921225721586, -0.46439857887558106, -0.3685539265001533, -0.
 ↪1830244989818371,
          0.03835530881316253, 0.27636300051486506, 0.4163001580121898, 0.
 ↪45791960756716926,
          0.4133277046353799, 0.2454571744527809, 0.02843505139404641, -0.
 ↪17995259849041545,
```

(continues on next page)

(continued from previous page)

```
        -0.3528170613993332, -0.4432670688586229, -0.4308872405896063, -0.
↪3026153676925856,
        -0.10460001285649327, 0.10576999137260902, 0.2794125634785019, 0.
↪39727960494083775,
        0.44012013094335295, 0.33665573392945614, 0.19241116925921273, -0.
↪01373255454051425,
        -0.2288093781464643, -0.36965277845455136, -0.4281953074871928, -0.
↪3837254105792804,
        -0.2439384794237307, -0.05945292033865969, 0.12714573185667402, 0.
↪31127632080817996,
        0.38957820781361946, 0.4022425989547513, 0.2852587536434507, 0.
↪12097975879552739,
        -0.06442727622110235, -0.25077071692351, -0.3414562050547135, -0.
↪3844271968533089,
        -0.3344326061358699, -0.1749451011511041, -0.016716872087832926, 0.
↪18543436492361495,
        0.322914548974734, 0.37225887677620817, 0.3396788239855797, 0.
↪254332254048965,
        0.0665429944525343, -0.11267963115652598, -0.2795347352658008, -0.
↪37651751469292644,
        -0.3654799155251143, -0.28854786660062454, -0.14328549945642755, 0.
↪04213065692623846,
        0.2201386996949766, 0.32591680654151617, 0.3421258708802321, 0.
↪32458251078266503,
        0.18777342536595865, 0.032971092778669095, -0.13521542014013244, -0.
↪3024666212766573,
        -0.34121930021830144, -0.33201664443451445, -0.21960116119767256, -0.
↪07735846981040519,
        0.09202435934084172, 0.24410808130924241, 0.3310159788871507, 0.
↪34354629209994936,
        0.2765230394100408, 0.12467034370454594, -0.012680322294530635, -0.
↪1690136862958978,
        -0.27753009059653394]
y_errors: 0.01

# Because the model function is oscillating the fit needs to be initialized␣
↪with near guesses for
# unconstrained parameters in order to converge.
# To use starting values for the fit, specify them as default values in the␣
↪fit function.
model_function: |
    def damped_harmonic_oscillator(theta, y_0=1.0, l=1.0, r=1.0, g=9.81, c=0.
↪01):
        # Model function for a pendulum as a 1d, damped harmonic oscillator␣
↪with zero initial speed
        # x = time, y_0 = initial_amplitude, l = length of the string,
        # r = radius of the steel ball, g = gravitational acceleration, c =␣
↪damping coefficient
        # effective length of the pendulum = length of the string + radius of␣
↪the steel ball
        l_total = l + r
        omega_0 = np.sqrt(g / l_total)  # phase speed of an undamped pendulum
```

(continues on next page)

(continued from previous page)

```
        omega_d = np.sqrt(omega_0 ** 2 - c ** 2)  # phase speed of a damped␣
↪pendulum
        return y_0 * np.exp(-c * theta) * (np.cos(omega_d * theta) + c /␣
↪omega_d * np.sin(omega_d * theta))

parameter_constraints:     # add parameter constraints
  l:
    value: 10.0
    uncertainty: 0.001     # l = 10.0+-0.001
  r:
    value: 0.052
    uncertainty: 0.001     # r = 0.052+-0.001
  y_0:
    value: 0.6
    uncertainty: 0.006
    relative: true         # Make constraint uncertainty relative to value, y_
↪0 = 0.6+-0.6%
```

2.7.2 Python

Using *kafe2* inside a *Python* script, parameter constraints can be set with `fit.add_parameter_constraint()`. The according section is highlighted in the code example below.

```
import numpy as np

from kafe2 import XYContainer, Fit, Plot

# Relevant physical magnitudes and their uncertainties:
l, delta_l = 10.0, 0.001  # length of the string, l = 10.0+-0.001 m
r, delta_r = 0.052, 0.001  # radius of the steel ball, r = 0.052+-0.001 m
# Amplitude of the steel ball at t=0 in degrees, y_0 = 0.6+-0.006% degrees:
y_0, delta_y_0 = 0.6, 0.01  # Note that the uncertainty on y_0 is relative to␣
↪y_0
g_0 = 9.81  # Initial guess for g


# Model function for a pendulum as a 1d, damped harmonic oscillator with zero␣
↪initial speed:
# t = time, y_0 = initial_amplitude, l = length of the string,
# r = radius of the steel ball, g = gravitational acceleration, c = damping␣
↪coefficient.
def damped_harmonic_oscillator(t, y_0, l, r, g, c):
    # Effective length of the pendulum = length of the string + radius of the␣
↪steel ball:
    l_total = l + r
    omega_0 = np.sqrt(g / l_total)  # Phase speed of an undamped pendulum.
    omega_d = np.sqrt(omega_0 ** 2 - c ** 2)  # Phase speed of a damped␣
↪pendulum.
    return y_0 * np.exp(-c * t) * (np.cos(omega_d * t) + c / omega_d * np.
↪sin(omega_d * t))
```

(continues on next page)

(continued from previous page)

```
# Load data from yaml, contains data and errors:
data = XYContainer.from_file(filename='data.yml')

# Create fit object from data and model function:
fit = Fit(data=data, model_function=damped_harmonic_oscillator)

# Constrain model parameters to measurements:
fit.add_parameter_constraint(name='l',   value=l,   uncertainty=delta_l)
fit.add_parameter_constraint(name='r',   value=r,   uncertainty=delta_r)
fit.add_parameter_constraint(name='y_0', value=y_0, uncertainty=delta_y_0,␣
↪relative=True)

# Lengths between two points are by definition positive, this can be␣
↪expressed with one-sided limit.
# Note: for technical reasons these limits are inclusive.
fit.limit_parameter("y_0", lower=1e-6)
fit.limit_parameter("l", lower=1e-6)
fit.limit_parameter("r", lower=1e-6)

# Set limits for g that are much greater than the expected deviation but␣
↪still close to 9.81:
fit.limit_parameter("g", lower=9.71, upper=9.91)

# Solutions are real if c < g / (l + r). Set the upper limit for c a little␣
↪lower:
c_max = 0.9 * g_0 / (l + r)
fit.limit_parameter("c", lower=1e-6, upper=c_max)

# Optional: Set the initial values of parameters to our initial guesses.
# This can help with convergence, especially when no constraints or limits␣
↪are specified.
fit.set_parameter_values(y_0=y_0, l=l, r=r, g=g_0)
# Note: this could also be achieved by changing the positional arguments of␣
↪our model function
# into keyword arguments with our initial guesses as the default values.

# Perform the fit:
fit.do_fit()

# Optional: Print out a report on the fit results on the console.
fit.report(asymmetric_parameter_errors=True)

# Optional: plot the fit results.
plot = Plot(fit)
plot.plot(residual=True, asymmetric_parameter_errors=True)

plot.show()
```

2.8 5: Convenience

This part includes examples, designed to be used as cheat sheets.

2.8.1 Plot Customization

This example is a cheat sheet for plot/report customization. It briefly demonstrates methods that modify the optics of kafe2 output.

```
from kafe2 import XYContainer, Fit, Plot

# The same setup as in 001_line_fit/line_fit.py :
xy_data = XYContainer(x_data=[1.0, 2.0, 3.0, 4.0],
                      y_data=[2.3, 4.2, 7.5, 9.4])
xy_data.add_error(axis='x', err_val=0.1)
xy_data.add_error(axis='y', err_val=0.4)
line_fit = Fit(data=xy_data)
line_fit.do_fit()

# Non-LaTeX names are used in reports and other text-based output:
line_fit.assign_parameter_names(x='t', a='alpha', b='beta')
line_fit.assign_model_function_expression('theta')
line_fit.assign_model_function_expression("{a} * {x} + {b}")
# Note: the model function expression is formatted as a Python string.
#    The names of parameters in curly brackets will be replaced with the␣
↪specified latex names.

# You could also just hard-code the parameter names like this:
# line_fit.assign_model_function_expression("alpha * t + beta")

line_fit.report()

# LaTeX names are used in plot info boxes:
line_fit.assign_parameter_latex_names(x='t', a='\\alpha', b='\\beta')
line_fit.assign_model_function_latex_name('\\theta')
line_fit.assign_model_function_latex_expression('{a} \\cdot {x} + {b}')

# Labels can be set for a fit.
# These labels are then used by all Plots created from said fit.
# If a Plot object also defines labels those labels override the fit labels.

# The labels displayed in the info box:
line_fit.data_container.label = "My data label"
line_fit.model_label = "My model label"

# The labels displayed on the x/y axes:
line_fit.data_container.axis_labels = ["My x axis", "My y axis"]

plot = Plot(fit_objects=line_fit)

# Plot objects can be modified with the customize method which sets␣
↪matplotlib keywords.
```

(continues on next page)

(continued from previous page)

```
# The first argument specifies the subplot for which to set keywords.
# The second argument specifies which keyword to set.
# The third argument is a list of values to set for the keyword for each fit␣
↪managed
#     by the plot object.

plot.customize('data', 'label', "My data label 2")  # Overwrite data label in␣
↪info box.
# plot.customize('data', 'label', None)  # Hide data label in info box.
plot.customize('data', 'marker', 'o')  # Set the data marker shape in the␣
↪plot.
plot.customize('data', 'markersize', 5)  # Set the data marker size in the␣
↪plot.
plot.customize('data', 'color', 'blue')  # Set the data marker color in the␣
↪plot.
plot.customize('data', 'ecolor', 'gray')  # Set the data errorbar color in␣
↪the plot.

plot.customize('model_line', 'label', 'My model label 2')  # Set the model␣
↪label in the info box.
# plot.customize('model_line', 'label', None)  # Hide the model label in the␣
↪info box.
plot.customize('model_line', 'color', 'lightgreen')  # Set the model line␣
↪color in the plot.
plot.customize('model_line', 'linestyle', '-')  # Set the style of the model␣
↪line in the plot.
plot.customize('model_error_band', 'label', r'$\pm 1 \sigma$')  # Error band␣
↪label in info box.
# plot.customize('model_error_band', 'label', None)  # Hide error band label.
plot.customize('model_error_band', 'color', 'lightgreen')  # Error band color␣
↪in plot.
# plot.customize('model_error_band', 'hide', True)  # Hide error band in plot␣
↪and info box.

# Available keywords can be retrieved with Plot.get_keywords(subplot_type) .
# subplot_type is for example 'data', 'model_line', or 'model_error_band'.

# In addition to the customize method Plot has a few convenience methods for␣
↪common operations:

plot.x_range = (0.8, 6)  # Set the x range of the plot.
plot.y_range = (1, 11)  # Set the y range of the plot.

plot.x_label = 'My x axis 2'  # Overwrite the label of the x axis.
plot.y_label = 'My y axis 2'  # Overwrite the label of the y axis.

plot.x_scale = 'log'  # Set the x axis to a logarithmic scale.
plot.y_scale = 'log'  # Set the y axis to a logarithmic scale.

# Finally, perform the plot:
plot.plot(ratio=True)
plot.show()
```

2.8.2 Accessing Fit Data via Properties

In the previous kafe2 examples we retrieved kafe2 results in a human-readable form via reports and plots. This example demonstrates how these fit results can instead be retrieved as Python variables.

```
from kafe2 import XYContainer, Fit

# The same setup as in 001_line_fit/line_fit.py :
xy_data = XYContainer(x_data=[1.0, 2.0, 3.0, 4.0],
                      y_data=[2.3, 4.2, 7.5, 9.4])
xy_data.add_error(axis='x', err_val=0.1)
xy_data.add_error(axis='y', err_val=0.4)
line_fit = Fit(data=xy_data)

# First option: retrieve the fit results from the dictionary returned by do_
↪fit.
result_dict = line_fit.do_fit()
# This dictionary contains the same information that would be shown in a␣
↪report or plot.
# It can also be retrieved via
# result_dict = line_fit.get_result_dict()

# Print contents of result dict:
for key in result_dict:
    if "mat" in key:
        print("%s:" % key)
        print(result_dict[key])
    else:
        print("%s = %s" % (key, result_dict[key]))
    print()
# Note: the asymmetric parameter errors are None because computing them is␣
↪relatively expensive.
# To calculate them run
# result_dict = line_fit.do_fit(asymmetric_parameter_errors=True)
# or
# result_dict = line_fit.get_result_dict(asymmetric_parameter_errors=True)

# A comparable output to above can be achieved by manually calling and␣
↪printing fit properties:
print("============ Manual prints below ============")
print()
print("did_fit = %s\n" % line_fit.did_fit)
print("cost = %s\n" % line_fit.cost_function_value)
print("ndf = %s\n" % line_fit.ndf)
print("goodness_of_fit = %s\n" % line_fit.goodness_of_fit)
print("gof/ndf = %s\n" % (line_fit.goodness_of_fit / line_fit.ndf))
print("chi2_probability = %s\n" % line_fit.chi2_probability)
print("parameter_values = %s\n" % line_fit.parameter_values)
print("parameter_name_value_dict = %s\n" % line_fit.parameter_name_value_dict)
print("parameter_cov_mat:\n%s\n" % line_fit.parameter_cov_mat)
print("parameter_errors = %s\n" % line_fit.parameter_errors)
print("parameter_cor_mat:\n%s\n" % line_fit.parameter_cor_mat)
print("asymmetric_parameter_errors:\n%s\n" % line_fit.asymmetric_parameter_
↪errors)
```

(continues on next page)

(continued from previous page)

2.8.3 Saving Fits

Most kafe2 objects can be turned into the human-readable YAML format and written to a file. These files can then be used to load the objects into Python code or as input for kafe2go.

```
from kafe2 import XYContainer, XYFit, Fit, Plot

# The same data as in 001_line_fit/line_fit.py :
xy_data = XYContainer(x_data=[1.0, 2.0, 3.0, 4.0],
                      y_data=[2.3, 4.2, 7.5, 9.4])
xy_data.add_error(axis='x', err_val=0.1)
xy_data.add_error(axis='y', err_val=0.4)

# Save the data container to a file:
xy_data.to_file("data_container.yml")
# Because a data container does not have a model function running kafe2go␣
↪with this file as input
#     will fit a linear model a * x + b .


def quadratic_model(x, a, b, c):
    return a * x ** 2 + b * x + c


line_fit = Fit(data=xy_data, model_function=quadratic_model)

# Save the fit to a file:
line_fit.to_file("fit_before_do_fit.yml")
# Because the fit object contains a model function running kafe2go with this␣
↪file as input will
#     make use of the model function we defined above.
#
# Note: The context in which the model function is defined is NOT saved. If␣
↪your model function
#       depends on things outside the function block it cannot be loaded back␣
↪(NumPy and SciPy
#       are available in the context in which the function is being loaded␣
↪though).

line_fit.do_fit()

# Save the fit with fit results to a file:
line_fit.to_file("fit_after_do_fit.yml")

# Alternatively we could save only the state (parameter values + fit results)␣
↪to a file:
line_fit.save_state("fit_results.yml")
```

(continues on next page)

(continued from previous page)

```
# Load it back into code:
loaded_fit = XYFit.from_file("fit_after_do_fit.yml")
# Note: this requires the use of a specific fit class like XYFit. The generic␣
↪Fit pseudo-class
#     does NOT work.

# Alternatively we could have created a new fit and loaded the fit results:
# loaded_fit = Fit(data=xy_data, model_function=quadratic_model)
# loaded_fit.load_state("fit_results.yml")
#
# Note: Because we defined the model function in regular Python code there␣
↪are no problems with
#       the context in that it's being defined.

loaded_fit.report()

plot = Plot(fit_objects=loaded_fit)

plot.plot()
plot.show()
```

2.9 6.1: Covariance matrix

2.9.1 kafe2go

```
type: indexed
data:
- 80.429
- 80.339
- 80.217
- 80.449
- 80.477
- 80.310
- 80.324
- 80.353
errors:
- - 0.055
  - 0.073
  - 0.068
  - 0.058
  - 0.069
  - 0.091
  - 0.078
  - 0.068
- type: matrix
  matrix_type: covariance
  matrix: [[0.000441, 0.000441, 0.000441, 0.000441, 0.000625, 0.000625, 0.
↪000625, 0.000625],
```

(continues on next page)

(continued from previous page)

```
            [0.000441, 0.000441, 0.000441, 0.000441, 0.000625, 0.000625, 0.
↪000625, 0.000625],
            [0.000441, 0.000441, 0.000441, 0.000441, 0.000625, 0.000625, 0.
↪000625, 0.000625],
            [0.000441, 0.000441, 0.000441, 0.000441, 0.000625, 0.000625, 0.
↪000625, 0.000625],
            [0.000625, 0.000625, 0.000625, 0.000625, 0.001936, 0.001936, 0.
↪001936, 0.001936],
            [0.000625, 0.000625, 0.000625, 0.000625, 0.001936, 0.001936, 0.
↪001936, 0.001936],
            [0.000625, 0.000625, 0.000625, 0.000625, 0.001936, 0.001936, 0.
↪001936, 0.001936],
            [0.000625, 0.000625, 0.000625, 0.000625, 0.001936, 0.001936, 0.
↪001936, 0.001936]]
model_function: |
    def average(a):
        # our model is a simple constant function
        return a
```

2.9.2 Python

```
import numpy as np
from kafe2 import IndexedContainer, Fit, Plot

measurements = np.array([5.3, 5.2, 4.7, 4.8])  # The results we want to␣
↪average.
err_stat = 0.2  # Statistical uncertainty for each measurement.
err_syst_1234 = 0.15  # Systematic uncertainty that affects all measurements.
err_syst_12 = 0.175  # Systematic absolute uncertainty only affecting the␣
↪first two measurements.
err_syst_34 = 0.05  # Systematic relative uncertainty affecting only the last␣
↪two measurements.

# Create an IndexedContainer from our data:
data = IndexedContainer(measurements)

# Start with an empty matrix for our covariance matrix:
covariance_matrix = np.zeros(shape=(4, 4))
# Uncorrelated uncertainties only affect the diagonal of the covariance␣
↪matrix:
covariance_matrix += np.eye(4) * err_stat ** 2
# Fully correlated uncertainties that affect all measurements affect all␣
↪covariance matrix entries:
covariance_matrix += err_syst_1234 ** 2
# Fully correlated uncertainties that affect only a part of the measurements␣
↪result in block-like
# changes to the covariance matrix:
covariance_matrix[0:2, 0:2] += err_syst_12 ** 2  # Magnitude of abs.␣
↪uncertainty is the same.
err_syst_34_abs = err_syst_34 * measurements[2:4]  # Magnitude of abs.␣
↪uncertainty is different.
```

(continues on next page)

(continued from previous page)

```
covariance_matrix[2:4, 2:4] += np.outer(err_syst_34_abs, err_syst_34_abs)

# The covariance matrix can now be simply added to our container to specify␣
↪the uncertainties:
data.add_matrix_error(covariance_matrix, matrix_type="cov")

# To get the same result we could have also added the uncertainties one-by-
↪one like this:
# data.add_error(err_stat, correlation=0)
# data.add_error(err_syst_1234, correlation=1)
# data.add_error([err_syst_12, err_syst_12, 0, 0], correlation=1)
# data.add_error([0, 0, err_syst_34, err_syst_34], correlation=1,␣
↪relative=True)
# See the next example for another demonstration of this approach.


# Just for graphical output:
data.label = 'Test data'
data.axis_labels = [None, 'Measured value (a.o.)']


# The very simple "model":
def average(a):
    return a


# Set up the fit:
fit = Fit(data, average)
fit.model_label = 'average value'

# Perform the fit:
fit.do_fit()

# Report and plot results:
fit.report()
p = Plot(fit)
p.plot()

p.show()
```

2.10 6.2: Error components

Typically, the uncertainties of the measurement data are much more complex than in the examples discussed so far. In most cases there are uncertainties in ordinate and abscissa, and in addition to the independent uncertainties of each data point there are common, correlated uncertainties for all of them.

With the method `add_error()` or `add_matrix_error()` uncertainties can be specified on the 'x' and 'y' data, either in the form of independent or correlated, relative or absolute uncertainties of all or groups of measured values or by specifying the complete covariance or correlation matrix. All uncertainties specified in this way are included in the global covariance matrix for the fit.

As an example, we consider measurements of a cross section as a function of the energy near a resonance. These are combined measurement data from the four experiments at CERN's LEP accelerator, which were corrected for effects caused by photon radiation: Measurements of the hadronic cross section (σ) as a function of the centre-of-mass energy (E).

2.10.1 Python

```
from kafe2 import XYContainer, Fit, Plot, ContoursProfiler

# Center-of-mass energy E (GeV):
E = [88.387, 89.437, 90.223, 91.238, 92.059, 93.004, 93.916]  # x data
E_errors = [0.005, 0.0015, 0.005, 0.003, 0.005, 0.0015, 0.005]  #␣
↪Uncorrelated absolute x errors
ECor_abs = 0.0017  # Correlated absolute x error

# Hadronic cross section with photonic corrections applied (nb):
sig = [6.803, 13.965, 26.113, 41.364, 27.535, 13.362, 7.302]  # y data
sig_errors = [0.036, 0.013, 0.075, 0.010, 0.088, 0.015, 0.045]  #␣
↪Uncorrelated absolute y errors
sigCor_rel = 0.0007  # Correlated relative y error


# Breit-Wigner with s-dependent width:
def BreitWigner(E, s0=41.0, M_Z=91.2, G_Z=2.5):
    s = E*E
    Msq = M_Z*M_Z
    Gsq = G_Z*G_Z
    return s0*s*Gsq/((s-Msq)*(s-Msq)+(s*s*Gsq/Msq))


BW_data = XYContainer(E, sig)  # Create data container.

# Add errors to data container.
# By default errors are assumed to be absolute and uncorrelated.
# For errors that are relative and/or correlated you need to set the␣
↪corresponding kwargs.

# Add independent errors:
error_name_sig = BW_data.add_error(axis='x', name='deltaSig', err_val=E_
↪errors)
error_name_E = BW_data.add_error(axis='y', name='deltaE', err_val=sig_errors)

# Add fully correlated, absolute Energy errors:
error_name_ECor = BW_data.add_error(axis='x', name='Ecor', err_val=ECor_abs,␣
↪correlation=1.)

# Add fully correlated, relative cross section errors:
error_name_sigCor = BW_data.add_error(
    axis='y', name='sigCor', err_val=sigCor_rel, correlation=1.,␣
↪relative=True)
```

(continues on next page)

(continued from previous page)

```
# Note: kafe2 methods that add errors return a name for the added error. If no name is specified
# a random alphanumeric string is assigned automatically. Further down we will use these names to
# enable/disable some of the errors.

# Assign labels for the data and the axes:
BW_data.label = 'QED-corrected hadronic cross-sections'
BW_data.axis_labels = ('CM Energy (GeV)', r'$\sigma_h$ (nb)')
# Note: Because kafe2 containers are copied when a Fit object is created from them assigning labels
# to the original XYContainer after the fit has already been created would NOT work.

BW_fit = Fit(
    BW_data,
    "BreitWigner: x s0 M_Z G_Z -> s0 * x^2 * G_Z^2 / ((x^2 - M_Z^2)^2 + x^4 * G_Z^2 / M_Z^2)"
)

# Uncomment the following two lines to assign data labels after the fit has already been created:
# BW_fit.data_container.label = 'QED-corrected hadronic cross-sections'
# BW_fit.data_container.axis_labels = ('CM Energy (GeV)', r'$\sigma_h$ (nb)')

# Model labels always have to be assigned after the fit has been created:
BW_fit.model_label = 'Beit-Wigner with s-dependent width'

# Set LaTeX names for printout in info-box:
BW_fit.assign_parameter_latex_names(x='E', s0=r'\sigma^0')

# Do the fit:
BW_fit.do_fit()

# Print a report:
BW_fit.report(asymmetric_parameter_errors=True)

# Plot the fit results:
BW_plot = Plot(BW_fit)
BW_plot.y_range = (0, 1.03*max(sig))  # Explicitly set y_range to start at 0.
BW_plot.plot(residual=True, asymmetric_parameter_errors=True)

# Create a contour plot:
ContoursProfiler(BW_fit).plot_profiles_contours_matrix(show_grid_for='contours')

# Investigate the effects of individual error components: disabling the correlated uncertainty on
# energy should decrease the uncertainty of the mass M but have little to no effect otherwise.
print('====== Disabling error component %s ======' % error_name_ECor)
print()
```

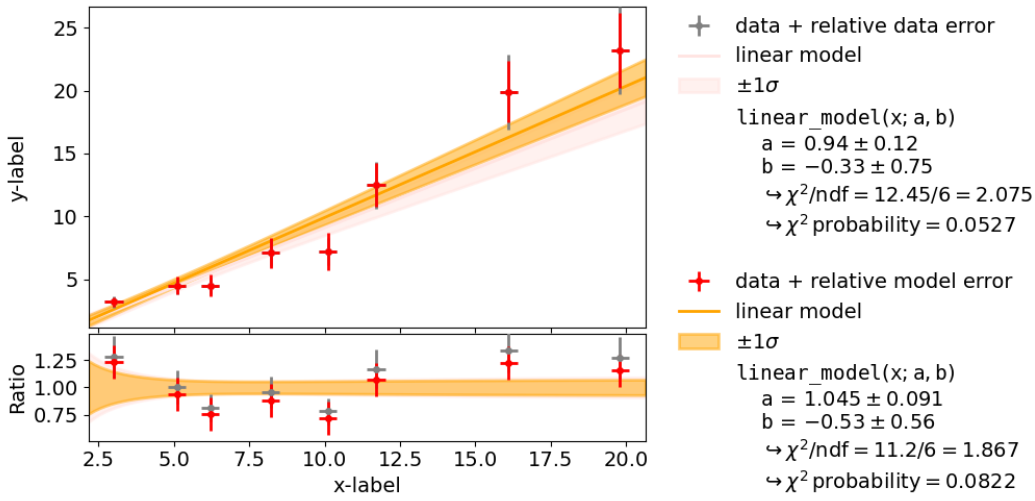
(continues on next page)

(continued from previous page)

```
BW_fit.disable_error(error_name_ECor)
BW_fit.do_fit()
BW_fit.report(show_data=False, show_model=False)

BW_plot.show()
```

2.11 6.3: Relative uncertainties

We had already seen that kafe2 allows the declaration of relative uncertainties which we want to examine more closely in this example. Adjustments with relative uncertainties suffer from the fact that the estimate of the parameter values is distorted. This is because measured values that fluctuate to smaller values have smaller uncertainties; the uncertainties are correspondingly greater when the measured values fluctuate upwards. If the random fluctuations were exactly the other way round, other uncertainties would be assigned. It would be correct to relate the relative uncertainties to the true values, which we do not know. Instead, the option reference='model' allows the uncertainties to be dependent on the model value - still not completely correct, but much better.



As seen in the example, the χ^2 probability improves from around 0.05 to 0.08 and χ^2/ndf improves from 2.1 to 1.9.

2.11.1 kafe2go

For using uncertainties relative to the model, The `parametric_model` section must be added inside the *YAML* file, as highlighted below. For comparison, one cane move the *y*-uncertainty out of the `parametric_model` section. Then the uncertainty will be relative to the data.

```
# This line fit uses uncertainties relative to the model.
# This requires a little bit more user input, but is still very␣
↪straightforward.
```

(continues on next page)

(continued from previous page)

```
x_data: [19.8, 3.0, 5.1, 16.1, 8.2, 11.7, 6.2, 10.1]
y_data: [23.2, 3.2, 4.5, 19.9, 7.1, 12.5, 4.5, 7.2]
x_errors: 0.3
label: data + relative model error
x_label: x-label
y_label: y-label

# In order to define uncertainties relative to the model, a parametric model␣
↪section must be added.
parametric_model:
  model_function: linear_model
  model_label: linear model
  y_errors: 15%
```

2.11.2 Python

Referencing the model as source for relative uncertainties is done by setting the keyword `reference='model'` in the *add_error* (page 124) method.

```
from kafe2 import XYContainer, Fit, Plot, ContoursProfiler

x = [19.8, 3.0, 5.1, 16.1, 8.2,  11.7, 6.2, 10.1]
y = [23.2, 3.2, 4.5, 19.9, 7.1, 12.5, 4.5, 7.2]
data = XYContainer(x_data=x, y_data=y)
data.add_error(axis='x', err_val=0.3)
data.axis_labels = ['x-label', 'y-label']

# create fit with relative data uncertainty
linear_fit1 = Fit(data, model_function='linear_model')
linear_fit1.add_error('y', 0.15, relative=True, reference='data')
linear_fit1.data_container.label = "data + relative data error"
linear_fit1.model_label = "linear model"
linear_fit1.do_fit()

# create fit with relative model uncertainty
linear_fit2 = Fit(data, model_function='linear_model')
linear_fit2.add_error('y', 0.15, relative=True, reference='model')
linear_fit2.data_container.label = "data + relative model error"
linear_fit2.model_label = "linear model"
linear_fit2.do_fit()

plot = Plot((linear_fit2, linear_fit1))  # first fit is shown on top of␣
↪second fit
# assign colors to data...
plot.customize('data', 'marker', ('o', 'o'))
plot.customize('data', 'markersize', (5, 5))
plot.customize('data', 'color', ('red', 'grey'))
plot.customize('data', 'ecolor', ('red', 'grey'))
# ... and model
```

(continues on next page)

(continued from previous page)

```
plot.customize('model_line', 'color', ('orange', 'mistyrose'))
plot.customize('model_error_band', 'label', (r'$\pm 1 \sigma$', r'$\pm 1 \
↪sigma$'))
plot.customize('model_error_band', 'color', ('orange', 'mistyrose'))
plot.plot(ratio=True)

cpf1 = ContoursProfiler(linear_fit1)
cpf1.plot_profiles_contours_matrix(show_grid_for='contours')

cpf2 = ContoursProfiler(linear_fit2)
cpf2.plot_profiles_contours_matrix(show_grid_for='contours')

plot.show()
```

2.12 7: Poisson Cost Function

In data analysis the uncertainty on measurement data is most often assumed to resemble a normal distribution. For many use cases this assumption works reasonably well but there is a problem: to get meaningful fit results you need to know about the uncertainties of your measurements. Now imagine for a moment that the quantity you're measuring is the number of radioactive decays coming from some substance in a given time period. What is your data error in this case? The precision with that you can correctly count the decays? The answer is that due to the inherently random nature of radioactive decay the variance, and therefore the uncertainty on your measurement data directly follows from the mean number of decays in a given time period - the number of decays are following a poisson distribution. In kafe2 this distribution can be modeled by initializing a fit object with a special cost function. In previous examples when no cost function was provided a normal distribution has been assumed by default. It is important to know that for large numbers of events a poisson distribution can be approximated by a normal distribution (y_error = sqrt(y_data)).

For our example on cost functions we imagine the following, admittedly a little contrived scenario: In some remote location on earth archeologists have found the ruins of an ancient civilization. They estimate the ruins to be about 7000 years old. The civilization in question seems to have known about mathematics and they even had their own calendar. Unfortunately we do not know the exact offset of this ancient calendar relative to our modern calendar. Luckily the ancient civilization seems to have mummified their rulers and written down their years of death though. Using a method called radiocarbon dating we can now try to estimate the offset between the ancient and the modern calendar by analyzing the relative amounts of carbon isotopes in the mummified remains of the ancient kings. More specifically, we take small samples from the mummies, extract the carbon from those samples and then measure the number of decaying carbon-14 atoms in our samples. Carbon-14 is a trace radioisotope with a half life of only 5730 years that is continuously being produced in earth's upper atmosphere. In a living organism there is a continuous exchange of carbon atoms with its environment which results in a stable concentration of carbon-14. Once an organism dies, however, the carbon atoms in its body are fixed and the concentration of carbon-14 starts to exponentially decrease over time. If we then measure the concentration of carbon-14 in our samples we can then calculate at which point in time they must have contained atmospheric amounts of carbon-14, i.e. the times of death of the ancient kings.

2.12.1 Python

```
import numpy as np
from kafe2 import XYFit, Plot

# Years of death are our x-data, measured c14 activity is our y-data.
# Note that our data does NOT include any x or y errors.
years_of_death, measured_c14_activity = np.loadtxt('measured_c14_activity.txt
↪')

days_per_year = 365.25  # assumed number of days per year
current_year = 2019  # current year according to the modern calendar
sample_mass = 1.0  # Mass of the carbon samples in g
initial_c14_concentration = 1e-12  # Assumed initial concentration
N_A = 6.02214076e23  # Avogadro constant in 1/mol
molar_mass_c14 = 14.003241  # Molar mass of the Carbon-14 isotope in g/mol

expected_initial_num_c14_atoms = initial_c14_concentration * N_A * sample_
↪mass / molar_mass_c14


# t = years of death in the ancient calendar
# Delta_t = difference between the ancient and the modern calendar in years
# T_12_C14 = half life of carbon-14 in years, read as T 1/2 carbon-14
def expected_activity_per_day(t, Delta_t=5000, T_12_C14=5730):
    # activity = number of radioactive decays
    expected_initial_activity_per_day = expected_initial_num_c14_atoms * np.
↪log(2) / (T_12_C14 * days_per_year)
    total_years_since_death = Delta_t + current_year - t
    return expected_initial_activity_per_day * np.exp(-np.log(2) * total_
↪years_since_death / T_12_C14)


# This is where we tell the fit to assume a poisson distribution for our data.
xy_fit = XYFit(
    xy_data=[years_of_death, measured_c14_activity],
    model_function=expected_activity_per_day,
    cost_function="nll-poisson"
)

# The half life of carbon-14 is only known with a precision of +-40 years
xy_fit.add_parameter_constraint(name='T_12_C14', value=5730, uncertainty=40)

# Perform the fit
# Note that since for a Poisson distribution the data error is directly␣
↪linked to the mean.
# Because of this fits can be performed without explicitly adding data errors.
xy_fit.do_fit()

# Optional: Assign new parameter names:
xy_fit.assign_parameter_latex_names(Delta_t=r"\Delta t", T_12_C14=r"T_{1/2}({}
↪^{14}C)")
```

(continues on next page)

(continued from previous page)

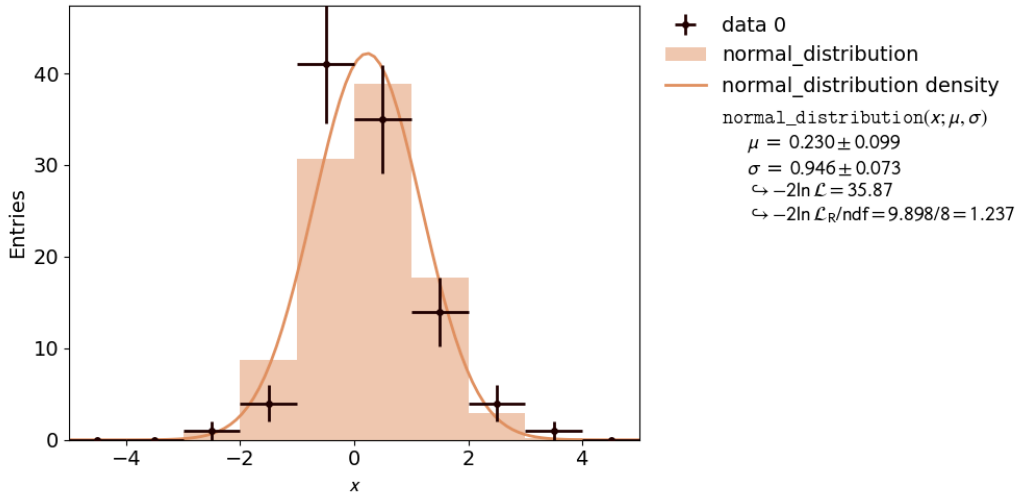
```
# Optional: print out a report on the fit results on the console
xy_fit.report()

# Optional: create a plot of the fit results using Plot
xy_plot = Plot(xy_fit)
xy_plot.plot(fit_info=True)

xy_plot.show()
```

2.13 8: Indexed Fit

In progress…

2.14 9: Histogram Fit

kafe2 is not only capable of performing XY-Fits. One way to handle one-dimensional data with *kafe2* is by fitting a histogram. The distribution of a random stochastic variable follows a probability density function. The fit will determine the parameters of that density function, which the dataset is most likely to follow. To get to the height of a bin, multiply the results of the fitted function with the amount of entries N of the histogram.



2.14.1 kafe2go

In order to tell *kafe2go* that a fit is a histogram fit `type: histogram` has to specified inside the *YAML* file.

```
type: histogram
n_bins: 10
bin_range: [-5, 5]
# alternatively an array for the bin edges can be specified
# bin_edges: [-5.0, -4.0, -3.0, -2.0, -1.0, 0.0, 1.0, 2.0, 3.0, 4.0, 5.0]
raw_data: [-2.7055035067034487, -2.2333305188904347, -1.7435697505734427, -1.
↪6642777356357366,
           -1.5679848184835399, -1.3804124186290108, -1.3795713509086, -1.
↪3574661967681951,
           -1.298552041356647, -1.2520621390613085, -1.1278660506506273, -1.
↪1072596181584253,
           -1.0450542861845105, -0.9892894612864382, -0.9094385163243246, -0.
↪7867219919905643,
           -0.7852735587423539, -0.7735126880952405, -0.6845244856096362, -0.
↪6703508433626327,
           -0.6675916809443503, -0.6566373185397965, -0.5958545249606959, -0.
↪5845389846486857,
           -0.5835331341085711, -0.5281866305050138, -0.5200579414945709, -0.
↪48433400194685733,
           -0.4830779584087414, -0.4810505389213239, -0.43133946043584287, -0.
↪42594442622513157,
           -0.31654403621747557, -0.3001580802272723, -0.2517534764243999, -0.
↪23262171181200844,
           -0.1225371703039085 2, -0.10096229375978451, -0.09647228959040793, -
↪0.09569644689333284,
           -0.08737474828114752, -0.03702037295350094, -0.01793164577601821,␣
↪0.010882324251760892,
           0.02763866634172083, 0.03152268199512073, 0.08347307807870571, 0.
↪08372272370359508,
           0.11250694814984155, 0.11309723105291797, 0.12917440700460459, 0.
↪19490542653497123,
           0.22821618079608103, 0.2524597237143017, 0.2529456172163689, 0.
↪35191851476845365,
           0.3648457146544489, 0.40123846238388744, 0.4100144162294351, 0.
↪44475132818005764,
           0.48371376223538787, 0.4900044970651449, 0.5068133677441506, 0.
↪6009580844076079,
           0.6203740420045862, 0.6221661108241587, 0.6447187743943843, 0.
↪7332139482413045,
           0.7347787168653869, 0.7623603460722045, 0.7632360142894643, 0.
↪7632457205433638,
           0.8088938754947214, 0.817906674531189, 0.8745869551496841, 0.
↪9310849201186593,
           1.0060465798362423, 1.059538506895515, 1.1222643456638215, 1.
↪1645916062658435,
           1.2108747344743727, 1.21215950663896, 1.243754543112755, 1.
↪2521725699532695,
           1.2742444530825068, 1.2747154562887673, 1.2978477018565724, 1.
↪304709072578339,
```

(continues on next page)

(continued from previous page)

```
            1.3228758386699584, 1.3992737494003136, 1.4189560403246424, 1.
↪4920066960096854,
            1.5729404508236828, 1.5805518556565201, 1.6857079988218313, 1.
↪726087558454503,
            1.7511987724150104, 1.8289480135723097, 1.8602081565623672, 2.
↪330215727183154]

model_density_function:
  model_function_formatter:
    latex_name: 'pdf'
    latex_expression_string: "\\exp{{\\frac{{1}}{{2}}(\\frac{{{x}-{mu}}}{{
↪{sigma}}})^2}} /
    \\sqrt{{2\\pi{sigma}^2}}"
    arg_formatters:
      x: '{\tt x}'
      mu: '\mu'
      sigma: '\sigma'
  python_code: |
    def normal_distribution(x, mu, sigma):
        return np.exp(-0.5 * ((x - mu) / sigma) ** 2) / np.sqrt(2.0 * np.pi *␣
↪sigma ** 2)
```

2.14.2 Python

To use a histogram fit in a *Python* script, just import it with `from kafe2 import HistContainer, HistFit`.

The creation a a histogram requires the user to set the limits of the histogram and the amount of bins. Alternatively the bin edges for each bin can be set manually.

```
import numpy as np
from kafe2 import HistContainer, Fit, Plot


def normal_distribution(x, mu, sigma):
    return np.exp(-0.5 * ((x - mu) / sigma) ** 2) / np.sqrt(2.0 * np.pi *␣
↪sigma ** 2)


# random dataset of 100 random values, following a normal distribution with␣
↪mu=0 and sigma=1
data = np.random.normal(loc=0, scale=1, size=100)

# Create a histogram from the dataset by specifying the bin range and the␣
↪amount of bins.
# Alternatively the bin edges can be set.
histogram = HistContainer(n_bins=10, bin_range=(-5, 5), fill_data=data)

# create the Fit object by specifying a density function
fit = Fit(data=histogram, model_function=normal_distribution, density=True)
```

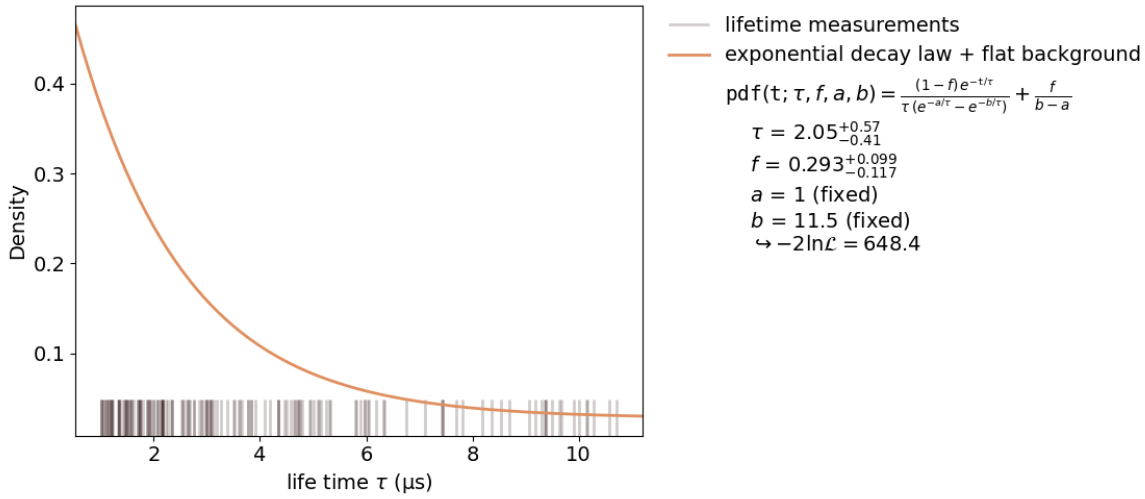
(continues on next page)

(continued from previous page)

```
fit.do_fit()  # do the fit
fit.report(asymmetric_parameter_errors=True)  # Optional: print a report to␣
↪the terminal

# Optional: create a plot and show it
plot = Plot(fit)
plot.plot(asymmetric_parameter_errors=True)
plot.show()
```

2.15 10: Unbinned Fit

An unbinned fit is needed when there are too few data points to create a (good) histogram. If a histogram is created from too few data points information can be lost or even changed by changing the exact value of one data point to the range of a bin. With an unbinned likelihood fit it's still possible to fit the probability density function to the data points, as the likelihood of each data point is fitted.



Inside a *kafe2* fit, single parameters can be fixed as seen in the plot. When fixing a parameter, there must be a good reason to do so. In this case it's the normalization of the probability distribution function. This, of course, could have been done inside the function itself. But if the user wants to change to normalization without touching the distribution function, this is a better way.

2.15.1 kafe2go

Similar to histogram fits, unbinned fits are defined by `type: unbinned` inside the *YAML* file. How to fix single parameters is highlighted in the example below, as well as limiting the background `fbg` to physically correct values.

```
type: unbinned
data: [7.420, 3.773, 5.968, 4.924, 1.468, 4.664, 1.745, 2.144, 3.836, 3.132,␣
↪1.568, 2.352, 2.132,
       9.381, 1.484, 1.181, 5.004, 3.060, 4.582, 2.076, 1.880, 1.337, 3.092,␣
↪2.265, 1.208, 2.753,
       4.457, 3.499, 8.192, 5.101, 1.572, 5.152, 4.181, 3.520, 1.344, 10.29,␣
↪1.152, 2.348, 2.228,
       2.172, 7.448, 1.108, 4.344, 2.042, 5.088, 1.020, 1.051, 1.987, 1.935,␣
↪3.773, 4.092, 1.628,
       1.688, 4.502, 4.687, 6.755, 2.560, 1.208, 2.649, 1.012, 1.730, 2.164,␣
↪1.728, 4.646, 2.916,
       1.101, 2.540, 1.020, 1.176, 4.716, 9.671, 1.692, 9.292, 10.72, 2.164,␣
↪2.084, 2.616, 1.584,
       5.236, 3.663, 3.624, 1.051, 1.544, 1.496, 1.883, 1.920, 5.968, 5.890,␣
↪2.896, 2.760, 1.475,
       2.644, 3.600, 5.324, 8.361, 3.052, 7.703, 3.830, 1.444, 1.343, 4.736,␣
↪8.700, 6.192, 5.796,
       1.400, 3.392, 7.808, 6.344, 1.884, 2.332, 1.760, 4.344, 2.988, 7.440,␣
↪5.804, 9.500, 9.904,
       3.196, 3.012, 6.056, 6.328, 9.064, 3.068, 9.352, 1.936, 1.080, 1.984,␣
↪1.792, 9.384, 10.15,
       4.756, 1.520, 3.912, 1.712, 10.57, 5.304, 2.968, 9.632, 7.116, 1.212,␣
↪8.532, 3.000, 4.792,
       2.512, 1.352, 2.168, 4.344, 1.316, 1.468, 1.152, 6.024, 3.272, 4.960,␣
↪10.16, 2.140, 2.856,
       10.01, 1.232, 2.668, 9.176]
label: "lifetime measurements"
x_label: "life time $\\tau$ (µs)"
y_label: "Density"
model_function:
  name: pdf
  latex_name: '{\tt pdf}'
  python_code: |
    def pdf(t, tau=2.2, fbg=0.1, a=1., b=9.75):
        """Probability density function for the decay time of a myon using the
        Kamiokanne-Experiment. The pdf is normed for the interval (a, b).
        :param t: decay time
        :param fbg: background
        :param tau: expected mean of the decay time
        :param a: the minimum decay time which can be measured
        :param b: the maximum decay time which can be measured
        :return: probability for decay time x"""
        pdf1 = np.exp(-t / tau) / tau / (np.exp(-a / tau) - np.exp(-b / tau))
        pdf2 = 1. / (b - a)
        return (1 - fbg) * pdf1 + fbg * pdf2
  arg_formatters:
```

(continues on next page)

(continued from previous page)

```
    t: '{\tt t}'
    tau: '{\tau}'
    fbg: '{f}'
    a: '{a}'
    b: '{b}'
  latex_expression_string: "\\frac{{ (1-{fbg}) \\, e^{{ -{t}/{tau} }} }}
    {{ {tau} \\, (e^{{ -{a}/{tau} }} - e^{{ -{b}/{tau} }}) }} + \\frac{{ {fbg}
↪ }} {{ {b}-{a} }}"
model_label: "exponential decay law + flat background"
fixed_parameters:
  a: 1
  b: 11.5
limited_parameters:
  fbg: [0.0, 1.0]
```

2.15.2 Python

The fitting procedure is similar to the one of a histogram fit. How to fix single parameters is highlighted in the example below.

```
from kafe2.fit import UnbinnedContainer, Fit, Plot
from kafe2.fit.tools import ContoursProfiler

import numpy as np


def pdf(t, tau=2.2, fbg=0.1, a=1., b=9.75):
    """
    Probability density function for the decay time of a myon using the␣
↪Kamiokanne-Experiment.
    The pdf is normed for the interval (a, b).

    :param t: decay time
    :param fbg: background
    :param tau: expected mean of the decay time
    :param a: the minimum decay time which can be measured
    :param b: the maximum decay time which can be measured
    :return: probability for decay time x
    """
    pdf1 = np.exp(-t / tau) / tau / (np.exp(-a / tau) - np.exp(-b / tau))
    pdf2 = 1. / (b - a)
    return (1 - fbg) * pdf1 + fbg * pdf2


# load the data from the experiment
infile = "tau_mu.dat"
dT = np.loadtxt(infile)

data = UnbinnedContainer(dT)  # create the kafe data object
data.label = 'lifetime measurements'
```

(continues on next page)

(continued from previous page)

```
data.axis_labels = ['life time $\\tau$ (µs)', 'Density']

# create the fit object and set the pdf for the fit
fit = Fit(data=data, model_function=pdf)

# Fix the parameters a and b.
# Those are responsible for the normalization of the pdf for the range (a, b).
fit.fix_parameter("a", 1)
fit.fix_parameter("b", 11.5)
# constrain parameter fbg to avoid unphysical region
fit.limit_parameter("fbg", 0., 1.)

# assign latex names for the parameters for nicer display
fit.model_label = "exponential decay law + flat background"
fit.assign_parameter_latex_names(fbg='f')
# assign a latex expression for the fit function for nicer display
fit.assign_model_function_latex_expression("\\frac{{ (1-{fbg}) \\, e^{{-{t}/
↪{tau}}} }}"
                                           "{{{tau} \\, (e^{{-{a}/{tau}}}-e^{
↪{-{b}/{tau}}}) }}"
                                           "+ \\frac{{ {fbg}}} {{{b}-{a}}}")

fit.do_fit()  # perform the fit
fit.report(asymmetric_parameter_errors=True)  # print a fit report to the␣
↪terminal

plot = Plot(fit)  # create a plot object
plot.plot(fit_info=True, asymmetric_parameter_errors=True)  # plot the data␣
↪and the fit

# Optional: create a contours profile
cpf = ContoursProfiler(fit, profile_subtract_min=False)
# Optional: plot the contour matrix for tau and fbg
cpf.plot_profiles_contours_matrix(parameters=['tau', 'fbg'])

plot.show()  # show the plot(s)
```

2.16 11: Multifit

The premise of this example is deceptively simple: a series of voltages is applied to a resistor and the resulting current is measured. The aim is to fit a model to the collected data consisting of voltage-current pairs and determine the resistance R.

According to Ohm's Law, the relation between current and voltage is linear, so a linear model can be fitted. However, Ohm's Law only applies to an ideal resistor whose resistance does not change, and the resistance of a real resistor tends to increase as the resistor heats up. This means that, as the applied voltage gets higher, the resistance changes, giving rise to nonlinearities which are ignored by a linear model.

To get a hold on this nonlinear behavior, the model must take the temperature of the resistor into account. Thus, the temperature is also recorded for every data point. The data thus consists of triples, instead of the

usual "xy" pairs, and the relationship between temperature and voltage must be modeled in addition to the one between current and voltage.

Here, the dependence $T(U)$ is taken to be quadratic, with some coefficients p_0, p_1, and p_2:

$$T(U) = p_2 U^2 + p_1 U + p_0$$

This model is based purely on empirical observations. The $I(U)$ dependence is more complicated, but takes the "running" of the resistane with the temperature into account:

$$I(U) = \frac{U}{R_0(1 + t \cdot \alpha_T)}$$

In the above, t is the temperature in degrees Celsius, α_T is an empirical "heat coefficient", and R_0 is the resistance at 0 degrees Celsius, which we want to determine.

In essence, there are two models here which must be fitted to the $I(U)$ and $T(U)$ data sets, and one model "incorporates" the other in some way.

2.16.1 Approach 1: parameter constraints

There are several ways to achieve this with *kafe2*. The method chosen here consists of two steps: First, a quadratic model is fitted to the $T(U)$ datasets to estimate the parameters p_0, p_1 and p_2 and their covariance matrix.

Next, the $I(U)$ model is fitted, with the temperature t being explicitly replaced by its parameterization as a function of p_0, p_1 and p_2. The key here is to fit these parameters again from the $I(U)$ dataset, but to constrain them to the values obtained in the previous $T(U)$ fit.

In general, this approach yields different results than the one using a simultaneous multi-model fit, which is demonstrated in the next example.

```
import numpy as np

from kafe2 import XYFit, Plot


# empirical model for T(U): a parabola
def empirical_T_U_model(U, p_2=1.0, p_1=1.0, p_0=0.0):
    # use quadratic model as empirical temperature dependence T(U)
    return p_2 * U**2 + p_1 * U + p_0


# model of current-voltage dependence I(U) for a heating resistor
def I_U_model(U, R_0=1., alpha=0.004, p_2=1.0, p_1=1.0, p_0=0.0):
    # use quadratic model as empirical temperature dependence T(U)
    _temperature = empirical_T_U_model(U, p_2, p_1, p_0)
    # plug the temperature into the model
    return U / (R_0 * (1.0 + _temperature * alpha))


# -- Next, read the data from an external file
```

(continues on next page)

(continued from previous page)

```
# load all data into numpy arrays
U, I, T = np.loadtxt('OhmsLawExperiment.dat', unpack=True)  # data
sigU, sigI, sigT = 0.2, 0.1, 0.5  # uncertainties

T0 = 273.15  # 0 degrees C as absolute Temperature (in Kelvin)
T -= T0  # Measurements are in Kelvin, convert to °C

# -- Finally, go through the fitting procedure

# Step 1: perform an "auxiliary" fit to the T(U) data
auxiliary_fit = XYFit(
    xy_data=[U, T],
    model_function=empirical_T_U_model
)
auxiliary_fit.data_container.axis_labels = ("Voltage (V)", "Temperature (°C)")
auxiliary_fit.data_container.label = "Temperature data"
auxiliary_fit.model_label = "Parametrization"

# (Optional): Assign names for models and parameters
auxiliary_fit.assign_model_function_expression('{1}*{U}^2 + {2}*{U} + {3}')
auxiliary_fit.assign_model_function_latex_expression(r'{1}\,{U}^2 + {2}\,{U}␣
↪+ {3}')

# declare errors on U
auxiliary_fit.add_error(axis='x', err_val=sigU)

# declare errors on T
auxiliary_fit.add_error(axis='y', err_val=sigT)

# perform the auxiliary fit
auxiliary_fit.do_fit()

# (Optional) print the results
auxiliary_fit.report(asymmetric_parameter_errors=True)

# (Optional) plot the results
auxiliary_plot = Plot(auxiliary_fit)
auxiliary_plot.plot(asymmetric_parameter_errors=True)

# Step 2: perform the main fit
main_fit = XYFit(
    xy_data=[U, I],
    model_function=I_U_model
)

# declare errors on U
main_fit.add_error(axis='x', err_val=sigU)
# declare errors on I
main_fit.add_error(axis='y', err_val=sigI)

# constrain the parameters
```

(continues on next page)

(continued from previous page)

```
main_fit.add_matrix_parameter_constraint(
    names=auxiliary_fit.parameter_names,
    values=auxiliary_fit.parameter_values,
    matrix=auxiliary_fit.parameter_cov_mat,
    matrix_type='cov'  # default matrix type is cov, this kwarg is just for␣
↪clarity
)
main_fit.data_container.axis_labels = ("Voltage (V)", "Current (A)")
main_fit.data_container.label = "Current data"
main_fit.model_label = "Temperature-dependent conductance"

# (Optional): Assign names for models and parameters
main_fit.assign_parameter_latex_names(alpha=r'\alpha_\mathrm{T}')
main_fit.assign_model_function_expression('{U} / ({0} * (1 + ({2}*{U}^2 + {3}*
↪{U} + {4}) * {1}))')
main_fit.assign_model_function_latex_expression(r'\frac{{{U}}}{{{0} \cdot (1␣
↪+ ({2}{U}^2 + {3}{U} + {4}) \cdot {1})}}')

# Step 4: do the fit
main_fit.do_fit()

# (Optional) print the results
main_fit.report(asymmetric_parameter_errors=True)

# (Optional) plot the results
plot = Plot(main_fit)
plot.plot(asymmetric_parameter_errors=True)

plot.show()
```

2.16.2 Approach 2: multi-model fit

There are several ways to achieve this with *kafe2*. The method chosen here is to use the `Multifit` functionality to fit both models simultaneously to the $T(U)$ and $I(U)$ datasets.

In general, this approach yields different results than the one using parameter constraints, which is demonstrated in the example called `fit_with_parameter_constraints`.

```
import numpy as np

from kafe2 import XYFit, MultiFit, Plot


# empirical model for T(U): a parabola
def empirical_T_U_model(U, p_2=1.0, p_1=1.0, p_0=0.0):
    # use quadratic model as empirical temperature dependence T(U)
    return p_2 * U**2 + p_1 * U + p_0

# model of current-voltage dependence I(U) for a heating resistor
def I_U_model(U, R_0=1., alpha=0.004, p_2=1.0, p_1=1.0, p_0=0.0):
```

(continues on next page)

(continued from previous page)

```
    # use quadratic model as empirical temperature dependence T(U)
    _temperature = empirical_T_U_model(U, p_2, p_1, p_0)
    # plug the temperature into the model
    return U / (R_0 * (1.0 + _temperature * alpha))


# -- Next, read the data from an external file

# load all data into numpy arrays
U, I, T = np.loadtxt('OhmsLawExperiment.dat', unpack=True)  # data
sigU, sigI, sigT = 0.2, 0.1, 0.5  # uncertainties

T0 = 273.15  # 0 degrees C as absolute Temperature (in Kelvin)
T -= T0  # Measurements are in Kelvin, convert to °C

# -- Finally, go through the fitting procedure

# Step 1: construct the singular fit objects
fit_1 = XYFit(
    xy_data=[U, T],
    model_function="empirical_T_U_model: U p_2 p_1 p_0 -> p_2 * U^2 + p_1 * U␣
↪+ p_0"
)
fit_1.add_error(axis='y', err_val=sigT)  # declare errors on T
fit_1.data_container.axis_labels = ("Voltage (V)", "Temperature (°C)")
fit_1.data_container.label = "Temperature data"
fit_1.model_label = "Parametrization"

fit_2 = XYFit(
    xy_data=[U, I],
    model_function="I_U_model: U R_0 alpha=4e-3 p_2 p_1 p_0 -> U / (R_0 * (1␣
↪+ alpha * (p_2 * U^2 + p_1 * U + p_0)))"
)
fit_2.add_error(axis='y', err_val=sigI)  # declare errors on I
fit_2.data_container.axis_labels = ("Voltage (V)", "Current (A)")
fit_2.data_container.label = "Current data"
fit_2.model_label = "Temperature-dependent conductance"

# Step 2: construct a MultiFit object
multi_fit = MultiFit(fit_list=[fit_1, fit_2], minimizer='iminuit')
#multi_fit.set_parameter_values(alpha=0.004)

# Step 3: Add a shared error error for the x axis.
multi_fit.add_error(axis='x', err_val=sigU, fits='all')

# (Optional): assign names for models and parameters
multi_fit.assign_parameter_latex_names(alpha=r'\alpha_\mathrm{T}')

# Step 4: do the fit
multi_fit.do_fit()

# (Optional): print the results
```

(continues on next page)

(continued from previous page)

```
multi_fit.report(asymmetric_parameter_errors=True)

# (Optional): plot the results
plot = Plot(multi_fit, separate_figures=True)
plot.plot(asymmetric_parameter_errors=True)

plot.save()  # Automatically saves the plots to different files.

plot.show()
```

3

User Guide

For performing fits with *kafe2*, the user need to specify the data, model function and optionally a so-called cost function. In most cases the cost function is either the method of least-squares or the negative-log-likelihood. All this information is then given to a `FitBase` (page 160)-derived object. More information is given in the *Fitting* (page 65)-section.

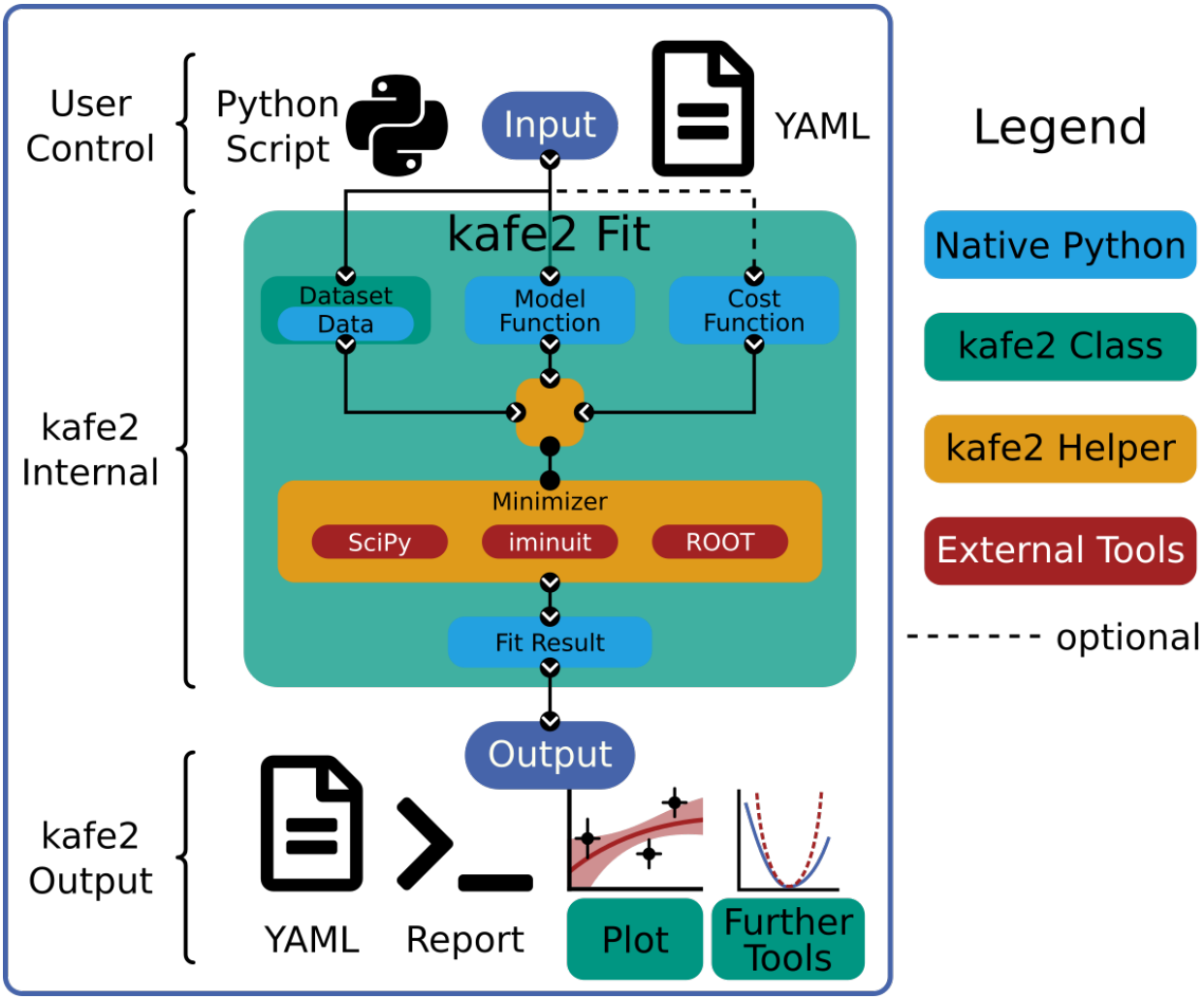
Then there are multiple ways of displaying and using the fit results. The results can either be used directly inside a *Python*-script, printed to the terminal, or *plotted* (page 69). For further analysis, the *Contours Profiler* (page 72) is a very helpful tool to display parameter correlations.

3.1 Datasets

When performing fits with *kafe2*, the data is stored in so-called data containers (`DataContainerBase` (page 156)-derived objects). The difference between the container classes comes down to the type of data they store.

There are two types of data supported by *kafe2*: One-dimensional data like the lifetimes of particles following a probability density function or two-dimensional data like a current-voltage characteristic.

- The most basic example of data is a simple series of one-dimensional data points called indexed data in *kafe2* (`IndexedContainer` (page 134)).
- One-dimensional data can either be kept as-is or filled into a histogram ("*binning*"). In *kafe2* these types of data and their corresponding fits are referred to as unbinned and histogram data/fits (`UnbinnedContainer` and `HistContainer` (page 141)).
- Two-dimensional data and the corresponding fit is referred to as XY data/fit (`XYContainer` (page 114)).


User Control
Python Script
Input
YAML
kafe2 Internal
kafe2 Fit
Dataset Data
Model Function
Cost Function
Minimizer
SciPy
iminuit
ROOT
Fit Result
Output
kafe2 Output
YAML
Report
Plot
Further Tools
Legend
Native Python
kafe2 Class
kafe2 Helper
External Tools
optional

3.1.1 Setting the data

Data containers are created as regular Python objects from iterables (lists, arrays, etc.) of floats.

XY Container

```
from kafe2 import XYContainer
# Create an XYContainer object to hold the xy data for the fit.
xy_data = XYContainer(x_data=[1.0, 2.0, 3.0, 4.0],
                      y_data=[2.3, 4.2, 7.5, 9.4])
```

Unbinned and Indexed Container

```
from kafe2 import IndexedContainer, UnbinnedContainer
idx_data = IndexedContainer([5.3, 5.2, 4.7, 4.8])
unbinned_data = UnbinnedContainer([5.3, 5.2, 4.7, 4.8])
```

Histogram Container

When creating a *HistContainer* (page 141) the binning of the histogram has to be determined. Equidistant bins can be created by using the `n_bins` and `bin_range` keywords.

```
from kafe2 import HistContainer
histogram = HistContainer(n_bins=10, bin_range=(-5, 5))
```

Alternatively the `bin_edges` keyword can be used to directly specify bin edges with arbitrary distances between them:

```
from kafe2 import HistContainer
hist = HistContainer(bin_edges=[-5.0, -4.0, -3.0, -2.0, -1.0, 0.0, 1.0, 2.0,␣
↪3.0, 4.0, 5.0])
```

After setting the bin edges, the histogram can be filled with data points. This can be done directly when creating the container with the `fill_data` keyword or afterwards with the *fill* (page 143) method. Data points lying outside the bin range will be stored in an underflow or overflow bin and are not considered when performing the fit.

```
from kafe2 import HistContainer
histogram = HistContainer(n_bins=10, bin_range=(-5, 5),
                          fill_data=[-7.5, 1.23, 5.74, 1.9, -0.2, 3.1, -2.75,␣
↪...])
# Alternative way
histogram = HistContainer(n_bins=10, bin_range=(-5, 5))
histogram.fill([-7.5, 1.23, 5.74, 1.9, -0.2, 3.1, -2.75, ...])
```

Instead of filling the histogram with raw data, the bin height can be set manually with *set_bins* (page 143). When doing so, rebinning and other options won't be available.

```
from kafe2 import HistContainer
histogram = HistContainer(n_bins=5, bin_range=(0, 5))
histogram.set_bins([1, 3, 5, 2, 0], underflow=2, overflow=0)
```

3.1.2 Data and axis labels

The name of the dataset or its label is set with the *label* (page 156) property. Axis labels can be set with the *x_label* (page 157) and *y_label* (page 157) properties or the *axis_labels* (page 156) property:

```
from kafe2 import XYContainer
xy_data = XYContainer(x_data=[1.0, 2.0, 3.0, 4.0], y_data=[2.3, 4.2, 7.5, 9.
↪4])
xy_data.label = 'My Data'
xy_data.axis_labels = ['Time $\\tau$ (µs)', 'My $y$-label']
```

Text in between dollar signs will be interpreted as latex code. The labels are displayed when plotting the fit results.

3.1.3 Uncertainties

To produce a meaningful fit result most cost functions require the user to specify uncertainties. Independent uncertainties and correlated uncertainties are added using the same methods.

Independent uncertainties

Independent uncertainties can be added to a dataset (*DataContainerBase* (page 156)-derived objects) with the *add_error* (page 157) method:

```
from kafe2 import XYContainer
x = [19.8, 3.0, 5.1, 16.1, 8.2,  11.7, 6.2, 10.1]
y = [23.2, 3.2, 4.5, 19.9, 7.1, 12.5, 4.5, 7.2]
data = XYContainer(x_data=x, y_data=y)
data.add_error(axis='x', err_val=0.3)  # +/-0.3 for all data points in x-
↪direction
data.add_error(axis='y', err_val=0.15, relative=True)  # +/-15% for all␣
↪points in y-direction
```

The `axis` keyword is is only used with XYContainers for the *add_error* (page 116) method. If `err_val` is a single float the same uncertainty is applied to all data points. If `err_val` is a list of floats with the same length as the corresponding data, each entry in `err_val` is applied to the data point with the same index.

Correlated uncertainties

If the correlation between the uncertainties for all data points is the same, the *add_error* (page 157) method can be used with the `correlation` keyword:

```
from kafe2 import IndexedContainer
idx_data = IndexedContainer([5.3, 5.2, 4.7, 4.8])
# independent uncertainties
err_stat = idx_data.add_error([.2, .2, .2, .2])
# uncertainty common to the first two values
err_syst12 = idx_data.add_error([.175, .175, 0., 0.], correlation = 1.)
# relative uncertainty common to the last two values
err_syst34 = idx_data.add_error([0., 0., .05, 0.05], correlation = 1.,␣
↪relative=True)
# uncertainty common to all values
err_syst = idx_data.add_error(0.15, correlation = 1.)
```

Note that the above example does not make use of the `axis` keyword because indexed data is one-dimensional. By calling *add_error* (page 157) multiple times the covariance matrix can be constructed from multiple regular uncertainties. The final covariance matrix can be accessed via the *cov_mat* (page 157) property. It is also possible to directly specify a more complicated uncertainty source as a covariance matrix with the *add_matrix_error* (page 158) method. Please refer to the API documentation for more information.

3.2 Fitting

Creating the correct *FitBase* (page 160) derived object can simply be done with the `Fit` function, which automatically determines the correct fit type for a *DataContainerBase* (page 156) derived object:

```
from kafe2 import XYContainer, Fit
xy_data = XYContainer(x_data=[1.0, 2.0, 3.0, 4.0],
                      y_data=[2.3, 4.2, 7.5, 9.4])
# Create an XYFit object from the xy data container.
# By default, a linear function f=a*x+b will be used as the model function.
line_fit = Fit(data=xy_data)
# further additions like constraints go here
line_fit.do_fit()
```

Alternatively *XYFit* (page 119), *HistFit* (page 145), `UnbinnedFit` or *IndexedFit* (page 137) can be used to create fits with corresponding datasets.

Warning: Always run the *do_fit* (page 166) function of the Fit object when everything is set. Only when calling this function the fit will be performed.

3.2.1 Setting a model function

kafe2 fit objects accept normal Python functions as model functions. The first parameter of those functions will be used as the independent parameter (the parameter on the x axis of plots). The default parameter values of the Python function will be used as starting values for the fit, unless overwritten with the *set_parameter_values* (page 163) method.

```
def linear_model(x, a, b):
    # Our first model is a simple linear function
    return a * x + b

def exponential_model(x, A0=1., x0=5.):
    # Our second model is a simple exponential function
    # The kwargs in the function header specify parameter defaults.
    return A0 * np.exp(x/x0)

xy_data = XYContainer(x_data=[1.0, 2.0, 3.0, 4.0],
                      y_data=[2.3, 4.2, 7.5, 9.4])

# Create 2 Fit objects with the same data but with different model functions
linear_fit = Fit(data=xy_data, model_function=linear_model)
exponential_fit = Fit(data=xy_data, model_function=exponential_model)
```

The display names for the model function and its parameters can be changed like this:

```
linear_fit.assign_model_function_name("line")
linear_fit.assign_parameter_names(a='A', b='b', x='t')
linear_fit.assign_model_function_expression("{a}{x} + {b}")
exponential_fit.assign_model_function_latex_name("\\exp")
exponential_fit.assign_parameter_latex_names(A0='A_0', x0='x_0', x='\\tau')
exponential_fit.assign_model_function_latex_expression("{A0} e^{{{x}/{x0}}}")
```

The latex parameter names and expressions define the graphical output when plotting while the non latex methods define the output names when reporting the fit results to the terminal.

Note: Special characters inside the strings need to be escaped. E.g. a single `\` needs to be `\\`.

Note: Inside the latex expression string, `{` and `}` for latex expressions like `\\frac` need to be doubled, because single curly brackets are used for replacing the parameters with their respective latex names. E.g. kafe2 tries to replace `{x0}` with its latex string `x_0` in this example.

3.2.2 Parameter Constraints

When performing a fit, some values of the model function might have already been determined in previous experiments. Those results and uncertainties can then be used to constrain the given parameters in a new fit. This eliminates the need to manually propagate the uncertainties on the final fit results, as it's now done numerically.

Simple parameter constraints are set with the *add_parameter_constraint* (page 164) method:

```
# Constrain model parameters to measurements
fit.add_parameter_constraint(name='l',   value=l,   uncertainty=delta_l)
fit.add_parameter_constraint(name='r',   value=r,   uncertainty=delta_r)
fit.add_parameter_constraint(name='y_0', value=y_0, uncertainty=delta_y_0,␣
↪relative=True)
```

Note: The names have to be identical to the argument names in the model function. The parameter names can be accessed with the fit *parameter_names* (page 161) property.

If the uncertainties of several parameter constraints are correlated the *add_matrix_parameter_constraint* (page 164) method can be used instead. Please refer to the API Documentation for more information.

3.2.3 Fixing and limiting parameters

Limiting the parameters of a model function can be useful for improving the convergence of a fit by reducing the size of the parameter space in which it searches for the global cost function minimum. This is commonly done when the fit result of one or more parameters is expected to fall in a certain range or when the model function is not valid for some parameter values (e.g. a negative amplitude). For fits with many parameters fixing some of them at first and fitting multiple times might also help.

Fixing parameters is done with the *fix_parameter* (page 163) method and limiting with the *limit_parameter* (page 163) method. Releasing a fixed parameter is performed with *release_parameter* (page 163) and unlimiting a parameter with *unlimit_parameter* (page 163):

```
fit.fix_parameter("a", 1)
fit.fix_parameter("b", 11.5)
fit.release_parameter("a")
# limit parameter fbg to avoid unphysical region
fit.limit_parameter("fbg", 0., 1.)
fit.unlimit_parameter("fbg")
```

Note: The names have to be identical to the argument names in the model function. The parameter names can be accessed with the fit *parameter_names* (page 161) property.

Fixed parameters can be released with the *release_parameter* (page 163) method and limited parameters can be unlimited with the *unlimit_parameter* (page 163) method.

3.2.4 Minimizers

Currently the use of three different minimizers is supported. By default `iminuit` is used. If `iminuit` is not available, *kafe2* falls back to `scipy.optimize.minimize`.

The usage of a specific minimizer can be set during initialization of any *FitBase* (page 160)-object with the *minimizer* keyword. Depending on the installed minimizers this can either be `'iminuit'`, `'scipy'` or `'root'`.

Additional keywords for the instantiation can be passed as a `dict` via the *minimizer_kwargs* keyword when creating a fit object derived from *FitBase* (page 160).

Logging

To enable the output of the minimizer, set up a logger before calling `do_fit`:

```
import logging
logger = logging.getLogger()
logger.setLevel(logging.INFO)
```

This currently only works for the `scipy` and `iminuit` minimizer. For more detailed information increase the logging level to `logging.DEBUG`. This will give a more verbose output when using `iminuit`. The logger level should be reset to `logging.WARNING` before plotting. Otherwise `matplotlib` will create logging messages as well.

3.2.5 Access the fit results

The *do_fit* (page 166) method returns a dictionary containing most of the relevant results. Additionally the results can be printed to the terminal with *report* (page 167). The parameter values can also be accessed via the *parameter_values* (page 161) property as well as the symmetric and asymmetric parameter uncertainties and the correlation and covariance matrices via their respective properties:

```
fit = Fit(my_dataset)  # create a fit object
# perform the fit and calculate asymmetric uncertaintes
result = fit.do_fit(asymmetric_parameter_errors=True)
fit.report()  # print fit results to the terminal
par_vals = fit.parameter_values
par_errs = fit.parameter_errors
par_errs_asym = fit.asymmetric_parameter_errors
par_ocv_mat = fit.parameter_cov_mat
par_cor_mat = fit.parameter_cor_mat
```

A typical dictionary returned by the *do_fit* (page 166) method looks like this:

```
{'did_fit': True,
 'cost': 1.7759115950075888,
 'ndf': 2,
 'goodness_of_fit': 1.7759115950075888,
 'cost/ndf': 0.8879557975037944,
```

(continues on next page)

(continued from previous page)

```
 'chi2_probability': 0.41149607486886164,
 'parameter_values': OrderedDict([('a', 2.468773761415478), ('b', -0.
↪3219331193129483)]),
 'parameter_cov_mat': array([[ 0.0443453 , -0.1108627 ],
                             [-0.1108627 ,  0.33239252]]),
 'parameter_errors': OrderedDict([('a', 0.2105624096609012), ('b', 0.
↪576478065203752)]),
 'parameter_cor_mat': array([[ 1.       , -0.9131448],
                             [-0.9131448,  1.       ]]),
 'asymmetric_parameter_errors': None}
```

Note: Asymmetric parameter uncertainties are only calculated when *do_fit* (page 166) is called with the corresponding keyword `fit.do_fit(asymmetric_parameter_errors=True)`. Otherwise they will be `None`.

3.3 Plotting

For displaying the results of a Fit, *kafe2* provides a *Plot* (page 175)-class. In the background a `matplotlib.pyplot.figure`-object is created. This means that all customization possible with *Matplotlib* can be done with *kafe2*-Plots as well.

The Plot class supports plotting multiple fits at once. By default they will all appear in the same figure. The keyword *separate_figures=True* changes this behaviour, so that each fit will be plotted to a separate figure.

```
import matplotlib.pyplot as plt
from kafe2 import Plot
p = Plot([fit_1, fit_2])
# for separate figures use:
# p = Plot([fit_1, fit_2], separate_figures=True)
# insert customization here
p.plot()
plt.show()
```

Running the *plot* (page 176) function will perform the the plot. Customization should be done before this. After plotting the fits, the according `matplotlib` objects can be accessed via the *figures* (page 175) and *axes* (page 175) properties.

3.3.1 Customize the Plot

Note: The *plot* (page 176) method must be called after all customization is done. Otherwise not all customizations will appear in the plot.

Axis Range

The plot range can be set via the *x_range* (page 176) and *y_range* (page 176) properties:

```
# set the same range for all plots
p.x_range = (0, 10)
p.y_range = (-5, 25)
# set different ranges for each plot, the length must match the number of␣
↪fits handled by the
# plot object.
p.x_range = [(0, 10), (-5, 5)]
p.y_range = [(-5, 25), (10, 100)]
p.plot()  # plot method must come after the customization
```

Axis Scale

Additionally the axis scale can be changed to logarithmic. When changing between a linear and logarithmic x-axis scale, the supporting points for plotting the model function will be updated and evenly spaced on a linear or logarithmic scale.

```
# set the same scale for all fits in this plot object
p.x_scale = "log"
p.y_scale = "linear
# or change the scale for each fit individually
# only use this when `separate_figures=True` is set in the Plot constructor
p.x_scale = ["linear", "log"]
p.y_scale = ["log", "log"]
p.plot()  # plot method must come after the customization
```

Axis Labels

By default, the plot will use the labels specified for each dataset (see *Data and axis labels* (page 64)). If multiple fits are plotted to the same figure, the axis labels from the data containers are concatenated while skipping duplicates.

Alternatively the axis labels can be overwritten for each fit. Again if multiple fits are plotted to the same figure, all labels will be concatenated while skipping duplicates.

```
# set the same axis labels for all fits in this plot object
p.x_label = "My $x$-label"
```

(continues on next page)

(continued from previous page)

```
p.y_label = "Voltage [mV]"
# set different labels for each fit, the length must match the number of fits
p.x_label = ["$x_1$", "My other label for $x_2$"]
p.y_label = ["$Y_1$", "$y_2$"]
p.plot()  # plot method must come after the customization
```

Plot Style

Each graphic element has it's own plotting method and can be customized individually. Available *plot_types* for XYFits are `'data'`, `'model_line'`, `'model_error_band'`, `'ratio'`, `'ratio_error_band'` and 'model' which is hidden by default. The *plot_types* may differ for different types of fits.

The currently set keywords can be obtained with the *get_keywords* (page 177) method. With *customize* (page 178) new values can be added or existing values can be modified. Using `'__del__'` will delete the keyword and `'__default__'` will reset it.

Hiding specific elements from the plot (e.g. the uncertainty band) is done like this:

```
# the array length must match the amount of fits handled by this plot object.
p.customize('model_error_band', 'hide', [True])
```

In order to change the name for the data set and suppress the second output, use the following call:

```
p.customize('data', 'label', [(0, "test data"), (1, '__del__')])
```

Marker type, size and color of the marker and error bars can also be customized:

```
p.customize('data', 'marker', [(0, 'o'), (1,'o')])
p.customize('data', 'markersize', [(0, 5), (1, 5)])
p.customize('data', 'color', [(0, 'blue'), (1,'blue')]) # note: although 2nd␣
↪label is suppressed
p.customize('data', 'ecolor', [(0, 'blue'), (1, 'blue')]) # note: although␣
↪2nd label is suppressed
```

The corresponding values for the model function can also be customized:

```
p.customize('model_line', 'color', [(0, 'orange'), (1, 'lightgreen')])
p.customize('model_error_band', 'label', [(0, r'$\pm 1 \sigma$'), (1, r'$\pm␣
↪1 \sigma$')])
p.customize('model_error_band', 'color', [(0, 'orange'), (1, 'lightgreen')])
```

Additionally it is possible to change parameters using matplotlib functions. Changing the size of the axis labels is done with the following calls:

```
import matplotlib as mpl
mpl.rc('axes', labelsize=20, titlesize=25)
```

3.4 Contours Profiler

Todo: Add this section, examples already use the contours profiler.

4 kafe2go Guide

kafe2go is a standalone program for performing fits which comes with *kafe2*. When using *kafe2go* no programming is required. Instead all necessary information for performing a fit is defined inside a so-called YAML file. Full examples can be found in the *Beginners Guide* (page 9).

To run kafe2go on Linux or MacOS issue this command in the terminal, after installing *kafe2*:

```
kafe2go path/to/fit.yml
```

When using Windows, please use:

```
kafe2go.py path/to/fit.yml
```

The output can be customized via command line arguments. For more information about the command line arguments run:

```
kafe2go --help
```

Additionally, *YAML* files can also be created from fit objects inside a *Python* program by using the `to_file` (page 167) method and loaded from a *YAML* file with `from_file`.

4.1 Setting a fit type

By default *kafe2go* assumes an XYFit, where each data point has an x and a y component. Other types are defined via the `type` keyword inside the *YAML* file:

```
type: histogram
```

Supported types are `type: xy`, `type: histogram`, `type: unbinned` and `type: indexed`.

4.2 Specifying the data

The syntax for specifying fit data differs depending on which fit type is being used.

4.2.1 XY Fits

For XY Fits the x and y values are defined separately:

```
# Data is defined by lists:
x_data: [1.0, 2.0, 3.0, 4.0]
# In yaml lists can also be written out like this:
y_data:
- 2.3
- 4.2
- 7.5
- 9.4
```

The uncertainties in x and y direction can be defined like this:

```
# For errors lists describe pointwise uncertainties.
# By default the errors will be uncorrelated.
x_errors: [0.05, 0.10, 0.15, 0.20]

# Because x_errors is equal to 5% of x_data we could have also defined it␣
↪like this:
# x_errors: 5%

# For errors a single float gives each data point
# the same amount of uncertainty:
y_errors: 0.4
```

In total the above examples represents the following dataset:

X Values	Y Values
1.0 +- 0.05	2.3 +- 0.4
2.0 +- 0.10	4.2 +- 0.4
3.0 +- 0.15	7.5 +- 0.4
4.0 +- 0.20	9.4 +- 0.4

Todo: Add more advanced errors for XY Fits

4.2.2 Histogram Fits

Specifying the data for histogram fits can be done in two different ways: either by specifying the raw data and bins and let *kafe2* handle the binning or by specifying the bins and bin heights.

Like with Python code there are two ways of specifying bins. The first way is to specify equidistant binning with the number of bins and the bin range. A binning with 10 bins in the range from -5 to 5 can be achieved like this:

```
n_bins: 10
bin_range: [-5, 5]
```

Alternatively the `bin_edges` keyword can be used to directly specify bin edges with arbitrary distances between them:

```
bin_edges: [-5.0, -4.0, -3.0, -2.0, -1.0, 0.0, 1.0, 2.0, 3.0, 4.0, 5.0]
```

Filling the bins with raw data is done like this:

```
raw_data: [-7.5, 1.23, 5.74, 1.9, -0.2, 3.1, -2.75, ...]
```

Data points lying outside the bin range will be stored in an underflow or overflow bin and are not considered when performing the fit.

Alternatively the heights of the bins can be set manually. This is done with the `bin_heights` keyword:

```
bin_heights: [7, 21, 25, 42, 54, 51, 39, 28, 20, 12]
```

Warning: The length of `bin_heights` must match the number of bins `n_bins` or the length of `bin_edges` minus one.

The height of the underflow and overflow bin is set via the `underflow` and `overflow` keywords.

4.2.3 Unbinned and Indexed Fits

Setting the data for unbinned and indexed fits is done via the `data` keyword:

```
data: [7.420, 3.773, 5.968, 4.924, 1.468, 4.664, 1.745, 2.144, 3.836, 3.132,␣
↪1.568]
```

4.2.4 Data label and axis labels

The name of the dataset or its label is set with the `label` keyword, axis labels can be set with the `x_label` and `y_label` keywords:

```
label: "My Data"
x_label: "X Values with latex $\\tau$ (µs)"
y_label: "$y_0$-label"
```

Text in between dollar signs will be interpreted as latex code. The labels are used in the graphical output of *kafe2go*.

Todo: Add errors for HistFits and IndexedFits, implement and add for UnbinnedFits

---

4.3 Setting a model function

A model function is defined with the `model_function` keyword, followed by Python code as a string:

```
model_function: |
  def exponential_model(x, A0=1., x0=5.):
      # Our model is a simple exponential function
      # The kwargs in the function header specify parameter defaults.
      return A0 * np.exp(x/x0)
```

Note the block style indicate `|` which indicates a multiline string and keeps line breaks.

Additionally the output names for the model and its parameters can be changed. Then the `model_function` block gains its own keywords and the Python code is moved to the `python_code` sub keyword:

```
model_function:
  python_code: |
    def exponential_model(x, A0=1., x0=5.):
      # Our model is a simple exponential function
      # The kwargs in the function header specify parameter defaults.
      return A0 * np.exp(x/x0)
  name: "exponential function"
  latex_name: "\\exp"
  expression_string: "{A0} * exp({x}/{x0})"
  latex_expression_string: "{A0} e^{{\\frac{{{x}}}{{{x0}}}}}"
  arg_formatters:
    x: 'x'
    x0: "x_0"
    A0:
      - name: A
      - latex_name: A_0
```

All other keywords are pretty much self-explanatory:

- `name` is the model function name for the terminal output. If omitted the function name from the definition will be used.
- `latex_name` is the model function name for the graphical output. If omitted the function name from the definition will be used.
- `expression_string` is the model function expression for the terminal output. If omitted, the output won't have any function expression after its name. Every parameter name inside curly brackets which matches the parameter names from the function definition will be replaced with its formatted version defined in `arg_formatters`.
- `latex_expression_string` is the same as `expression_string` but for the graphical output and supports latex syntax.
- `arg_formatters` defines the replacements for the function parameters. If only one string is given (see x and x0 in the example), the default name will be used for the terminal output and the given string will be used for the graphical latex output. If `name` and `latex_name` are defined, they are used for terminal and graphical outputs (see A0).

Note: Special characters inside the strings need to be escaped. E.g. a single `\` needs to be `\\`.

Note: Inside the latex expression string, `{` and `}` for latex expression like `\\frac` need to be doubled, because single curly brackets are used for replacing the parameters with their respective latex names. E.g. kafe2 tries to replace `{x0}` with its latex string `x_0` in this example.

4.4 Parameter Constraints

The parameter constraints specified with *kafe2go* require the same information as those *specified with Python code* (page 67): parameter names, values, and uncertainties.

4.4.1 Simple Gaussian Constraints

Simple gaussian parameter constraints can be defined by parameter name like this:

```
parameter_constraints:
  a:
    value: 10.0
    uncertainty: 0.001     # a = 10.0+-0.001
  b:
    value: 0.6
    uncertainty: 0.006
    relative: true         # Make constraint uncertainty relative to value, b
↪= 0.6+-0.6%
```

The same can also be done with a list and the name keyword:

```
parameter_constraints:
    - name: a
      value: 10.0
      uncertainty: 0.001     # a = 10.0+-0.001
    - name: b
      value: 0.6
      uncertainty: 0.006
      relative: true         # Make constraint uncertainty relative to value,␣
↪b = 0.6+-0.6%
```

4.4.2 Matrix Gaussian Constraints

Matrix constraints can only be specified using the list format since they constrain multiple parameters, possibly making the first format ambiguous. Inside a list element, the type can be set with the `type: matrix` keyword and a covariance matrix can be given via the `matrix` keyword like this:

```
parameter_constraints:
  - type: matrix
    names: [a, b]
    values: [1.3, 2.5]
    matrix: [[1.1, 0.1], [0.1, 2.4]]
  - type: simple
    name: c
    value: 5.2
    uncertainty: 0.001
```

Here `type: simple` can be omitted, as a simple gaussian constraint is assumed when no type is given.

A covariance matrix is assumed by default for matrix constraints. Correlation matrices are supported as well. The matrix type can be set via the `matrix_type` keyword. Only `matrix_type: cov` for covariance matrices and `matrix_type: cor` for correlation matrices are supported.

4.5 Fixing and limiting parameters

Fixing and limiting parameters is just as simple as the following example. Please use the parameter names as they are defined inside the model function (see: *Setting a model function* (page 76)).

```
fixed_parameters:
  a: 1
  b: 11.5
limited_parameters:
  c: [0.0, 1.0]
  d: [-5, 5]
  e: [0.0, null]
```

Note that a limited parameter needs two arguments: a lower and an upper limit. The first value of the list is the lower limit and the second value is the upper limit. For a one-sided limit simply set the argument for the side that you don't want to limit to `null`. For technical reasons parameter limits are *inclusive*, meaning

that the specified bounds are included in the range of allowed parameter values. For example, in the above configuration the parameter *e* is limited to non-negative numbers and *can* become 0 during the fit. If the final fit result is close to or at the parameter limits, the limits should be reassessed, as the final minimum of the cost function might be a local minimum for the given limits.

5 Mathematical Foundations

This chapter describes the mathematical foundations on which *kafe2* is built, in particular the method of maximum likelihood and the profile likelihood method.

When performing a fit (as a physicist) the problem is as follows: you have some amount of measurement **data** from an experiment that you need to compare to one or more **models** to figure out which of the models - if any - provides the most accurate description of physical reality. You typically also want to know the values of the **parameters** of a model and the degree of (un)certainty on those values.

For very simple problems you can figure this out **analytically**: you take a formula and simply plug in your measurements. However, as your problems become more complex this approach becomes much more difficult - or even straight up impossible. For this reason complex problems are typically solved **numerically**: you use an algorithm to calculate a result that only approximates the analytical solution but in a way that is much easier to solve. *kafe2* is a tool for the latter approach.

5.1 Cost Functions

In the context of **parameter estimation**, a model $m_i(\boldsymbol{p})$ (where i is just some index) is a function of the parameters $\boldsymbol{p}$. During the fitting process the parameters $\boldsymbol{p}$ are varied in such a way that the agreement between the model and the corresponding data d_i becomes "best". This of course means that we need to somehow define "good" or "bad" agreement - we need a metric. This metric is called a loss function or **cost function**. It is defined in such a way that a lower cost function value corresponds with a "better" agreement between a model and our data.

The cost functions implemented by *kafe2* are based on the **method of maximum likelihood**. The idea behind this method is to *maximize* the likelihood function $\mathcal{L}(\boldsymbol{p})$ which represents the probability with which a model $\boldsymbol{m}(\boldsymbol{p})$ would result in the data $\boldsymbol{d}$ that we ended up measuring:

$$\mathcal{L}(\boldsymbol{p}) = \prod_i P(m_i(\boldsymbol{p}), d_i),$$

where $P(m_i(\boldsymbol{p}), d_i)$ describes the probability of measuring the data point d_i given the corresponding model prediction $m_i(\boldsymbol{p})$. This approach allows for a **statistical interpretation** of our fit results as we will see later.

Instead of the likelihood described above however, we are instead using twice its negative logarithm, the so-called **negative log-likelihood** (NLL):

$$\mathrm{NLL}(\boldsymbol{p}) = -2\log\mathcal{L}(\boldsymbol{p}) = -2\log\left(\prod_i P(m_i(\boldsymbol{p}), d_i)\right) = -2\sum_i \log P(m_i(\boldsymbol{p}), d_i).$$

This transformation is allowed because logarithms are **strictly monotonically increasing functions**, and therefore the negative logarithm of any function will have its global minimum at the same place where the original function is maximal. The model $\boldsymbol{m}(\boldsymbol{p})$ that minimizes the NLL will therefore also maximize the likelihood.

While the above transformation may seem nonsensical at first, there are important advantages to calculating the negative log-likelihood over the likelihood:

- The *product* of the probabilities $\prod_i P(m_i(\boldsymbol{p}), d_i)$ is replaced by a *sum* over the logarithms of the probabilities $\sum_i \log P(m_i(\boldsymbol{p}), d_i)$. In the context of a computer program sums are preferable over products because they can be calculated more quickly and because a product of many small numbers can lead to arithmetic underflow.
- Because the probabilities $P(m_i(\boldsymbol{p}), d_i)$ oftentimes contain exponential functions, calculating their logarithm is actually *faster* because it reduces the number of necessary operations.
- Algorithms for numerical optimization **minimize** functions so they can be directly used to optimize the NLL.

As an example, let us consider the likelihood function of data d_i that follows a **normal distribution** around means μ_i with standard deviations (uncertainties) σ_i:

$$\mathcal{L} = \prod_i P(\mu_i, \sigma_i, d_i) = \prod_i \frac{1}{\sqrt{2\pi}\,\sigma_i} \exp\left[-\frac{1}{2}\left(\frac{d_i - \mu_i}{\sigma_i}\right)^2\right].$$

The immediate trouble that we run into with this definition is that we have no idea what the means μ_i are - after all these are the "true values" that our data deviates from. However, we can still use this likelihood function by choosing $\mu_i = m_i(\boldsymbol{p})$. This is because for an infinite amount of data the model values $m_i(\boldsymbol{p})$ converge against the true values μ_i (assuming our model is accurate and our understanding of the uncertainties σ_i is correct).

Note: Conceptually uncertainties are typically associated with the data $\boldsymbol{d}$ even though they represent deviations from the model $\boldsymbol{m}(\boldsymbol{p})$. However, because the normal distribution is symmetric this does not have an effect on the likelihood function $\mathcal{L}$ (as long as the uncertainties $\boldsymbol{\sigma}$ do not depend on the model $\boldsymbol{m}(\boldsymbol{p})$).

For the NLL we now find:

$$\begin{aligned}
\mathrm{NLL}(\boldsymbol{p}) &= -2\log\mathcal{L}(\boldsymbol{p}) \\
&= -2\log\prod_i \frac{1}{\sqrt{2\pi}\,\sigma_i} \exp\left[-\frac{1}{2}\left(\frac{d_i - m_i(\boldsymbol{p})}{\sigma_i}\right)^2\right] \\
&= -2\sum_i \log\frac{1}{\sqrt{2\pi}\,\sigma_i} + \sum_i \left(\frac{d_i - m_i(\boldsymbol{p})}{\sigma_i}\right)^2 \\
&=: -2\log L_{\max} + \chi^2(\boldsymbol{p}).
\end{aligned}$$

As we can see the logarithm cancels out the exponential function of the normal distribution and we are left with two parts: The first is a part represented by $-2\log L_{\max}$ that only depends on the uncertainties σ_i but not

on the model $m_i(\boldsymbol{p})$ or the data d_i. This is the minimum value the NLL could possibly take on if the model $m_i(\boldsymbol{p})$ were to exactly fit the data d_i. The second part can be summed up as

$$\chi^2(\boldsymbol{p}) = \sum_i \left(\frac{d_i - m_i(\boldsymbol{p})}{\sigma_i} \right)^2 .$$

As mentioned before, this is much easier and faster to calculate than the original likelihood function. If the uncertainties $\boldsymbol{\sigma}$ are constant we can ignore the first part and directly use $\chi^2(\boldsymbol{p})$ (chi-squared) as our cost function because we are only interested in differences between cost function values. This special case of the method of maximum likelihood is known as the **method of least squares** and it is by far the most common cost function used for fits.

5.2 Covariance

The $\chi^2(\boldsymbol{p})$ cost function that we discussed in the previous section assumes that our data points d_i are subject to uncertainties σ_i. With this notation we implicitly assumed that our uncertainties are **independent** of one another: that there is no relationship between the uncertainties of any two individual data points. Independent uncertainties are frequently caused by random fluctuations in the measurement values, either because of technical limitations of the experimental setup (electronic noise, mechanical vibrations) or because of the intrinsic statistical nature of the measured observable (as is typical in quantum mechanics, e. g. radioactive decays).

However, there are also **correlated** uncertainties that arise due to effects that distort multiple measurements in the same way. Such uncertainties can for example be caused by a random imperfection of the measurement device which affects all measurements equally. The uncertainties of the measurements taken with such a device are no longer uncorrelated, but instead have one common uncertainty.

Historically uncertainties have been divided into *statistical* and *systematic* uncertainties. While this is appropriate when propagating the uncertainties of the input variables by hand it is not a suitable distinction for a numerical fit. In *kafe2* multiple uncertainties are combined to construct a so-called **covariance matrix**. This is a matrix with the pointwise data **variances** Var_i on its diagonal and the **covariances** Cov_{ij} between two data points outside the diagonal. By using this covariance matrix for our fit we can estimate the uncertainty of our model parameters numerically with no need for propagating uncertainties by hand.

As mentioned before, the diagonal elements of our covariance matrix represent the variances $\mathrm{Var}_i = \sigma_i^2$ of our data points. They simply represent the uncertainty of a single data point d_i while ignoring all other data points. An element outside the diagonal at position (i, j) represents the covariance Cov_{ij} between points d_i and d_j:

$$\mathrm{Cov}_{ij} = E[(d_i - E[d_i])(d_j - E[d_j])] = E[d_i \cdot d_j] - E[d_i] \cdot E[d_j] = E[d_i \cdot d_j] - \mu_i \cdot \mu_j,$$

where E is the expected value of a variable. The covariance Cov_{ij} is a measure of the joint variability of d_i and d_j - but for a meaningful interpretation it needs to be considered relative to the pointwise uncertainties σ_i. We therefore define the so-called **Pearson correlation coefficient** ρ_{ij} as follows:

$$\rho_{ij} = \frac{\mathrm{Cov}_{ij}}{\sigma_i \sigma_j}.$$

The correlation ρ_{ij} is normalized to the interval $[-1, 1]$. Its absolute value is a measure of how strongly the residuals $r_k = d_k - \mu_k$ depend on one another. In other words, the absolute value of ρ_{ij} measures how much information you get about r_i or r_j if you know the other one. For $\rho = 0$ they are completely independent

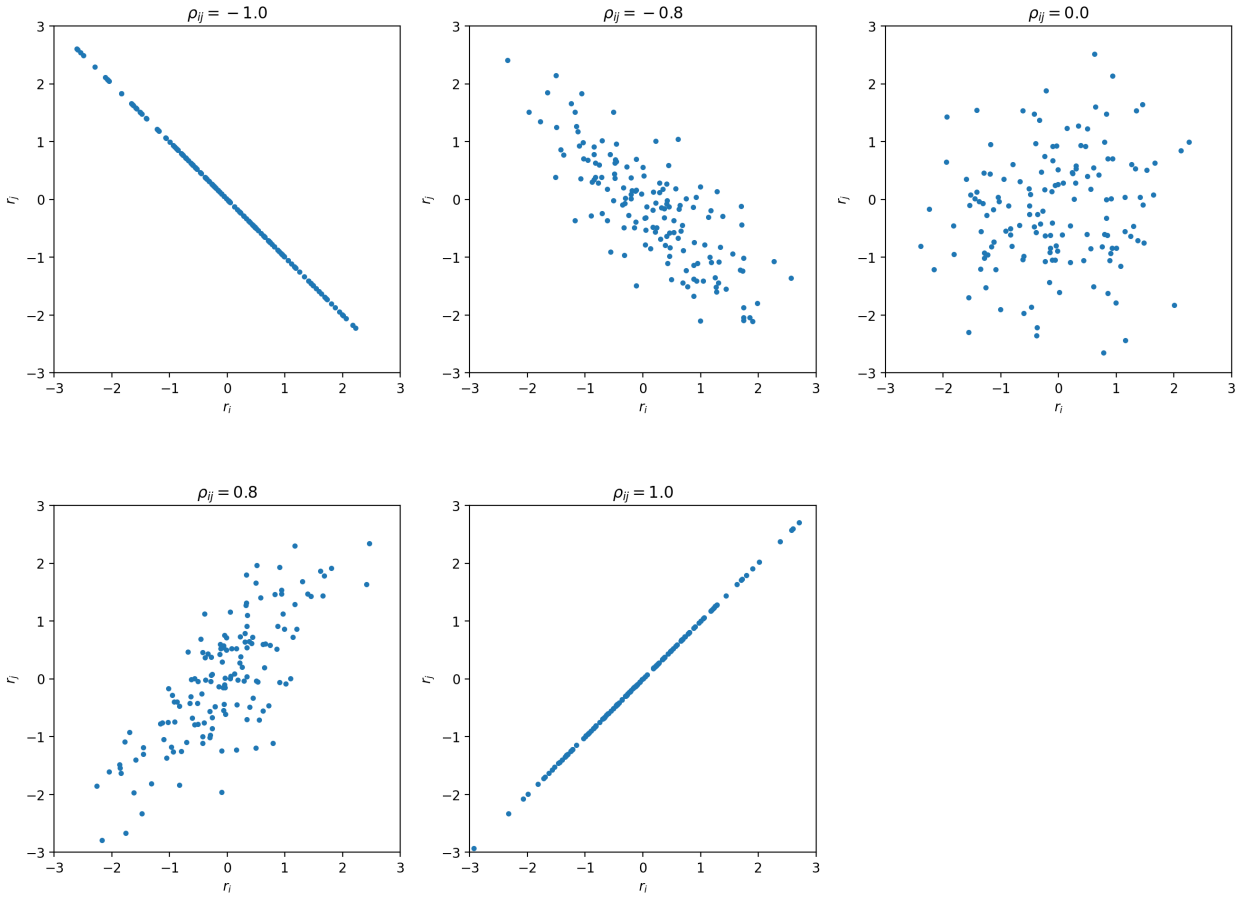


Fig. 5.1: Toy samples for correlation between residuals r_i and r_j for different values of the correlation coefficient ρ_{ij}. With an increasing absolute value the shape changes from a circle to a line.

from one another. For $\rho = \pm 1$ r_i and r_j are directly proportional to one another with a positive (negative) proportional constant for $\rho = +1$ ($\rho = -1$). Toy samples for different values of ρ_{ij} are shown in Fig. 5.1.

For $\rho_{ij} = 0$ the sample forms a circle around (0,0). As the absolute value of ρ_{ij} increases the sample changes its shape to a tilted ellipse - some combinations of r_i and r_j become more likely than others. For $\rho_{ij} = \pm 1$ the ellipse becomes a line - in this degenerate case we really only have one source of uncertainty that affects two data points.

As before, if we have "enough" data we can assume $r_k \approx d_k - m_k(\boldsymbol{p})$. This is useful because it allows us to use a covariance matrix to express the correlations of our uncertainties in our cost function, as we will see shortly.

5.2.1 Covariance Matrix Construction

In a physics experiment it is typically necessary to consider more than one source of uncertainty. Let us consider the following example: we want to measure Earth's gravitational constant g by dropping things from various heights and timing the time they take to hit the ground with a stopwatch. We assume an independent uncertainty of $\sigma_{\text{human}} = 0.5s$ for each data point because humans are not able to precisely align pressing the button of a stopwatch with the actual event. For one reason or another the stopwatch we're using is also consistently off by a few percentage points. To account for this we assume a fully correlated ($\rho_{ij} = 1$) uncertainty of $\sigma_{\text{watch}} = 2\%$ for all data points. To determine the variance of a single data point we can simply add up the variances of the uncertainty sources:

$$\text{Var}_{\text{total}} = \sigma^2_{\text{total}} = \text{Var}_{\text{human}} + \text{Var}_{\text{watch}} = \sigma^2_{\text{human}} + \sigma^2_{\text{watch}}.$$

As it turns out we can use the same approach for the covariances: we can simply add up the covariance matrices of the different uncertainty sources to calculate a total covariance matrix:

$$\boldsymbol{V}_{\text{total}} = \boldsymbol{V}_{\text{human}} + \boldsymbol{V}_{\text{watch}}.$$

The next question would then be how you would determine the covariance matrices for the individual uncertainty sources. A useful approach is to split a covariance matrix into a vector of uncertainty $\boldsymbol{\sigma}$ and the corresponding correlation matrix $\boldsymbol{\rho}$:

$$\boldsymbol{V} = (\boldsymbol{\sigma} \cdot \boldsymbol{\sigma}^T) \circ \boldsymbol{\rho},$$

where $\circ$ is the Hadamard product (a.k.a. Schur product). In other words, the components of $\boldsymbol{V}$ are calculated by simply multiplying the components of $\boldsymbol{\sigma} \cdot \boldsymbol{\sigma}^T$ and $\boldsymbol{\rho}$ at the same position. If we assume that we have three data points we can express the human uncertainty as follows:

$$\boldsymbol{\sigma}_{\text{human}} = \begin{pmatrix} 0.5 \\ 0.5 \\ 0.5 \end{pmatrix}, \quad \boldsymbol{\rho}_{\text{human}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \boldsymbol{V}_{\text{human}} = \begin{pmatrix} 0.25 & 0 & 0 \\ 0 & 0.25 & 0 \\ 0 & 0 & 0.25 \end{pmatrix}.$$

Because the human uncertainties of the individual data points are completely independent from one another the covariance/correlation matrix is a diagonal matrix. On the other hand, given some data points $\boldsymbol{d}$ the watch uncertainty is expressed like this:

$$\boldsymbol{\sigma}_{\text{watch}} = 0.02 \cdot \boldsymbol{d} = 0.02 \cdot \begin{pmatrix} d_1 \\ d_2 \\ d_3 \end{pmatrix}, \quad \boldsymbol{\rho}_{\text{watch}} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}, \quad \boldsymbol{V}_{\text{watch}} = 0.0004 \cdot \begin{pmatrix} d_1^2 & d_1 d_2 & d_1 d_3 \\ d_1 d_2 & d_2^2 & d_2 d_3 \\ d_1 d_3 & d_2 d_3 & d_3^2 \end{pmatrix}.$$

Because the watch uncertainties of the individual data points are fully correlated all components of the correlation matrix are equal to 1. However, this does not necessarily mean that all components of the covariance matrix are also equal. In this example the watch uncertainty per data point is relative, meaning that the absolute uncertainty differs from data point to data point.

If we were to visualize the correlations of the uncertainty components described above, we would find that samples of the human component form a circle while samples from the watch component form a line. If we were to visualize the total uncertainty we would end up with the mixed case where the sample forms an ellipse.

5.2.2 Correlated Least Squares

We previously defined the χ^2 cost function like this:

$$\chi^2(\boldsymbol{p}) = \sum_i \left(\frac{d_i - m_i(\boldsymbol{p})}{\sigma_i} \right)^2 .$$

This definition is only correct if the uncertainties for each data point are independent. If we want to consider the correlations between uncertainties we need to use the covariance matrix $\boldsymbol{V}$ instead of the pointwise uncertainties σ_i:

$$\chi^2(\boldsymbol{p}) = (\boldsymbol{d} - \boldsymbol{m}(\boldsymbol{p}))^T \cdot \boldsymbol{V}^{-1} \cdot (\boldsymbol{d} - \boldsymbol{m}(\boldsymbol{p})).$$

Notably the division by the uncertainties σ_i has been replaced by a matrix inversion. This is because the uncorrelated definition is a special case of the correlated definition. If the uncertainties are completely uncorrelated then $\boldsymbol{V}$ is a diagonal matrix. To invert such a matrix you only need to replace the diagonal elements V_{ii} with $1/V_{ii}$.

5.3 Profile Likelihood

When we perform a fit we are not only interested in the parameter values that fit our data "best", we also want to determine the uncertainty on our result. Fortunately likelihood-based cost functions provide a straightforward solution to our problem: the so-called **profile likelihood method**. By analyzing how a variation of one or more parameters affects the cost function value relative to the global cost function minimum we can determine areas that contain the true values of our parameters with a given **confidence level** of, say 95%.

5.3.1 Profile Likelihood (1 Parameter)

Let's say we performed a fit and found the global cost function minimum of our negative log-likelihood cost function with optimal parameters $\hat{a}, \hat{\boldsymbol{p}}$ (a is just one of the parameters that we consider separately). Because we have some amount of uncertainty on our input data we end up having some amount of uncertainty on our fit result as well. The global cost function minimum is the best fit (according to our cost function). And because our cost function measures how good a fit is given some parameter value a, investigating how flat or steep the cost function minimum is as a function of a tells us something about this parameter: if the cost function value increases very sharply when we move away from the cost function minimum then this tells us that even a small deviation from our fit result would result in a significantly worse fit, making large deviations unlikely. Conversely, if the cost function value increases very slowly when we move away from the cost function

minimum then this tells us that a deviation from our fit result would result in a fit that is only slightly worse than our optimal fit result, making such a deviation from our fit result quite possible.

We are trying to determine a so-called **confidence interval** for a: an interval that we expect to contain the true value of a with a given probability called the **confidence level** CL. The relevant metric for determining these intervals can be derived by considering the cost function increase relative to the global cost function minimum:

$$\Delta\text{NLL}(a, \boldsymbol{p}) = \text{NLL}(a, \boldsymbol{p}) - \text{NLL}(\hat{a}, \hat{\boldsymbol{p}}).$$

The obvious problem with this definition is that we need values not only for a but also for all other parameters $\boldsymbol{p}$ which we aren't actually interested in right now. So how do we determine the values for these parameters? The approach of the profile likelihood method is to choose $\boldsymbol{p}$ in such a way that $\Delta\text{NLL}(a, \boldsymbol{p})$ becomes minimal. In practical terms this means that we fix a to several values near the cost function minimum and then perform a fit over all other parameters for each of these values (this process is called profiling). In this context the parameters $\boldsymbol{p}$ are called **nuisance parameters**: we don't care about their values (right now) but we need to include them in our fits for a statistically correct result. If we were to instead use the optimal parameters $\hat{\boldsymbol{p}}$ then we would save on computing time but we would also be neglecting correlations between our fit parameters. Very often a variation of one fit parameter can (in part) be compensated by varying one of the other fit parameters. If we were to use the optimal parameters $\hat{\boldsymbol{p}}$ instead of performing a fit the cost function would increase more quickly when we vary a so we would end up *understimating* the uncertainty on a.

Now, the question is how to translate a desired confidence level CL to a difference in cost $\Delta\text{NLL}(a, \boldsymbol{p})$. As it turns out the confidence level for a confidence interval can be calculated from the probability density function (PDF) of the standard normal distribution:

$$\text{CL} = \int_{-x_\text{max}}^{x_\text{max}} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}, \quad x_\text{max} = \sqrt{\Delta\text{NLL}(a, \boldsymbol{p})}.$$

What we're actually interested in however, is the inverse case of calculating $\Delta\text{NLL}(a, \boldsymbol{p})$ for a given confidence level. Because the PDF of the normal distribution cannot be integrated analytically we have to resort to numerical integration - SciPy's **percent point function** (*scipy.stats.norm.ppf*) conveniently provides the function we need. We can use it to calculate $\Delta\text{NLL}(a, \boldsymbol{p})$ like this:

$$\Delta\text{NLL}(a, \boldsymbol{p}) = \left(\text{PPF}\left(\frac{1}{2} + \frac{CL}{2}\right)\right)^2.$$

The takeaway of this complicated-looking formula is this: the difference in cost is equal to the square of the "sigma value" of the commonly used confidence intervals of the normal distribution: the 1-σ-interval with CL $\approx 68\%$ corresponds to $\Delta\text{NLL}(a, \boldsymbol{p}) = 1^2 = 1$, the 2-$\sigma$-interval with CL $\approx 95\%$ corresponds to $\Delta\text{NLL}(a, \boldsymbol{p}) = 2^2 = 4$, the 3-$\sigma$-interval with CL $\approx 99.7\%$ corresponds to $\Delta\text{NLL}(a, \boldsymbol{p}) = 3^2 = 9$, and so on.

Note: The above formula is only correct for one dimension. If the shared profile likelihood of more than one parameter is examined we will need to use a different formula (see below).

The profile likelihood method is very expensive in terms of computation. For this reason it is not the default in *kafe2*. Instead the default behavior is to assume that the cost function value increases like

$$\Delta\text{NLL}(a, \hat{\boldsymbol{p}}) = \text{NLL}(\hat{a}, \hat{\boldsymbol{p}}) + \left(\frac{a - \hat{a}}{\sigma_a}\right)^2,$$

where σ_a is the **parabolic parameter uncertainty** of a. These are the standard parameter uncertainties provided by *kafe2* (and in fact most fitting tools). Because every minimum can be approximated by a parabola for sufficiently small scales (Taylor expansion) the parabolic parameter uncertainties are sufficiently accurate for many applications - but if you suspect they are not you should check the profiles of the parameters to make sure the result you extract is actually meaningful.

The easiest way to do this is to set the flag `asymmetric_parameter_errors = True` when calling `FitBase.report()` or `Plot.plot()`. The parabolic parameter uncertainties are then replaced with the edges of the 1-σ-intervals of the corresponding cost function profiles. Because these intervals are not necessarily symmetric around the cost function minimum they are referred to as **asymmetric parameter errors** in *kafe2* (in *Minuit* they are called Minos errors).

Another way to check the profiles is to use the `ContoursProfiler` object. It is capable of plotting the profiles of parameters (and also their contours, see below). Fig. 5.2 shows the profile of the parameter g from the double slit example:

The profile of this parameter is very clearly asymmetric and not even close to the parabolic approximation. If we had only looked at the parabolic parameter uncertainty our idea of the actual confidence intervals would be very wrong.

5.3.2 Profile Likelihood (2 parameters)

In the previous section we learned about the profiles of single fit parameters, which serve as a replacement for the uncertainties of single fit parameters. In this section we will learn about so-called **contours**, which serve as a replacement for the covariance of two fit parameters. Conceptually they are very similar. A profile defines confidence intervals for a single parameter with a certain likelihood of containing the true value of a parameter while a contour defines a **confidence region** with a certain likelihood of containing a *pair* of parameters. Fig. 5.3 shows the contours produced in the double slit example.

In this visualization the confidence region inside the contours is colored. By looking at the legend we find that the contours correspond to 1 σ and 2 σ. Notably the confidence levels of the corresponding confidence regions are *not* the same as in one dimension. In one dimension 1 σ corresponds to roughly 68% while 2 σ corresponds to roughly 95%. We could derive these confidence levels by integrating the probability density function of the standard normal distribution over the interval $[-\sigma, \sigma]$ for a desired σ value. In two dimensions we instead integrate the PDF of the uncorrelated standard bivariate normal distribution over a circle with radius σ around the origin:

$$\mathrm{CL}(\sigma) = \int_0^\sigma dr \int_0^{2\pi} d\varphi r \frac{1}{2\pi} e^{-\frac{r^2}{2}} = \int_0^\sigma dr\, r e^{-\frac{r^2}{2}} = \left[-e^{-\frac{r^2}{2}}\right]_0^\sigma = 1 - e^{-\frac{\sigma^2}{2}}.$$

With this formula we now find $\mathrm{CL}(1) = 39.3\%,\ \mathrm{CL}(2) = 86.4\%,\ \mathrm{CL}(3) = 98.8\%$.

Note: So far there has been no mention of how a contour for a given ΔNLL could be calculated. This is because (efficiently) calculating these contours is not straightforward and even in *kafe2* this is an area of active development.

The parabolic equivalent of a contour is to look at the parameter covariance matrix and to extrapolate the correlated distribution of two parameters. As with the input uncertainties the confidence region calculated this

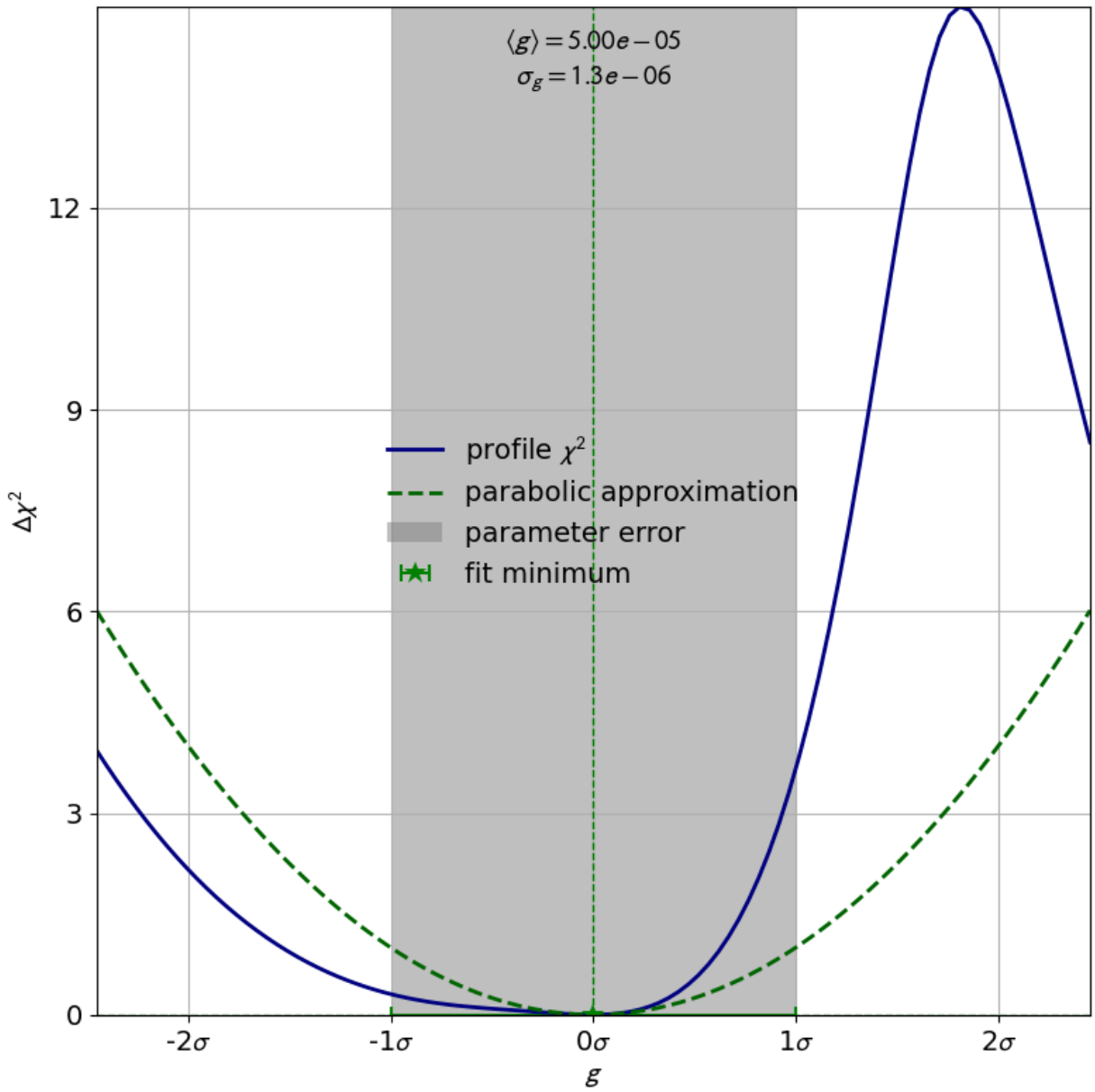


Fig. 5.2: Profile of parameter g from the double slit example. The parabolic approximation of the confidence interval is very inaccurate.

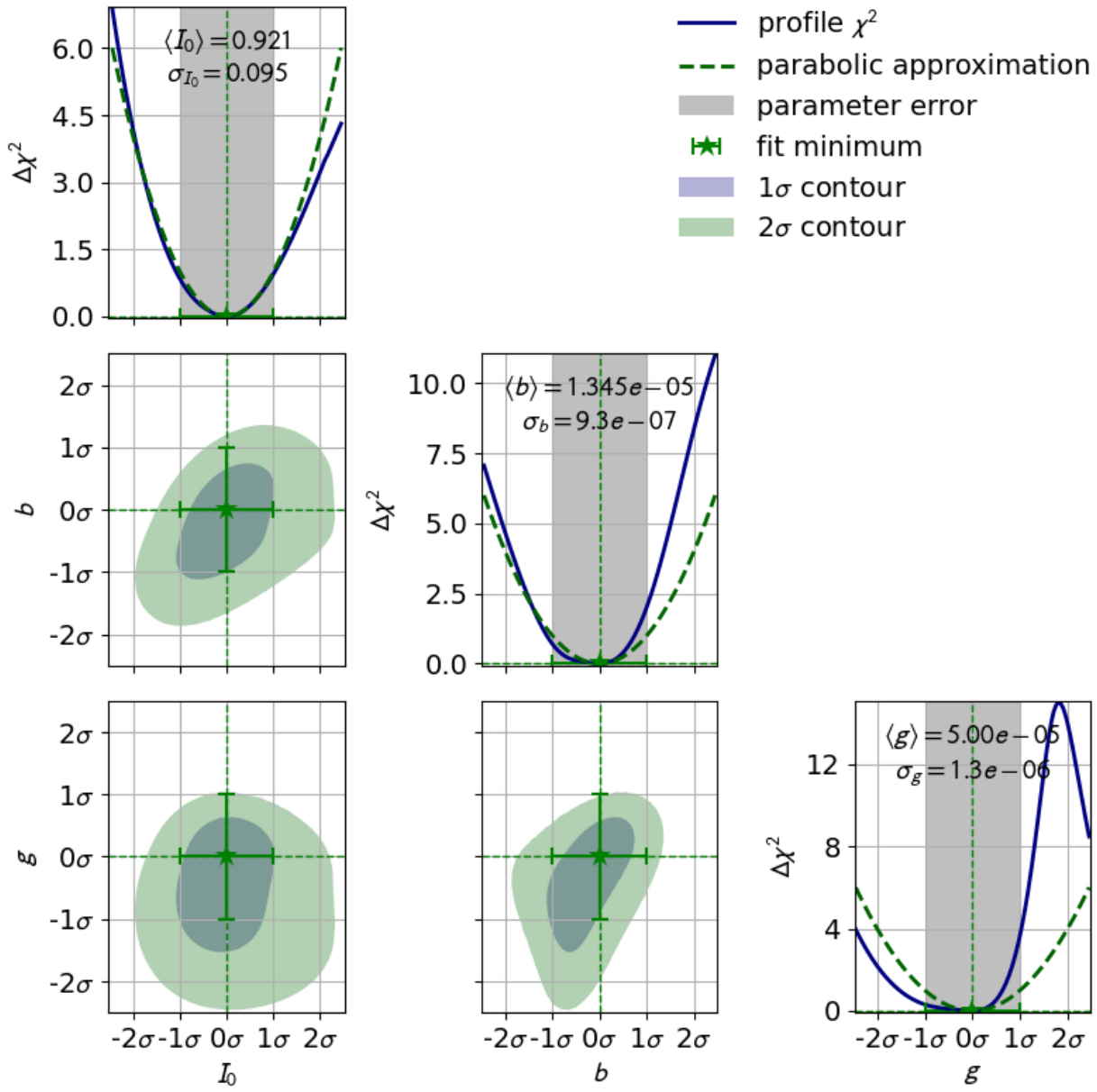


Fig. 5.3: Parameter confidence contours produced in the double slit example. Due to the nonlinear model function the contours are heavily distorted.

way will *always* be an ellipse. Fig. 5.4 shows contours for the nearly linear exponential fit from the model functions example.

If the fit were perfectly linear the 1-σ-contour would reach exactly from $-\sigma$ to $+\sigma$, while the 2-σ-contour would reach exactly from -2σ to $+2\sigma$. As we can see the deviation from this is very small so we can probably use the parameter covariance matrix (or the parameter uncertainties and the parameter correlation matrix) without issue. If we require highly precise confidence intervals for our parameters this might not be acceptable though.

Note: The degree to which confidence intervals/regions are distorted from their parabolic approximation depends on the scale at which the profile likelihood is calculated. Because every function can be accurately approximated by a linear function at infinitesimally small scales (Taylor expansion) the parabolic approximation becomes more accurate for small parameter uncertainties. Conversely, for large parameter uncertainties the parabolic approximation of the profile likelihood becomes less accurate.

5.4 Nonlinear Regression

In the previous section we discussed the profile likelihood method and how it can be used to calculate confidence intervals for our fit parameters. We also discussed the approximation of these confidence intervals through the use of parabolic uncertainties. In this context the term "linear" was used to describe fits where the parabolic uncertainties are accurate. This section will define more precisely what was meant by that.

5.4.1 Linear Regression

Let us assume we have some vector of N data points d_i with corresponding constant Gaussian uncertainties σ_i (that can also be correlated). **Linear regression** is then defined as a regression analysis (fit) using a model $m_i(\boldsymbol{p})$ that is a **linear function** of its M parameters p_j:

$$m_i(\boldsymbol{p}) = b_i + \sum_{j=1}^{M} w_{ij} p_j,$$

where the **weights** w_{ij} and **biases** b_i are simply real numbers. Put another way, each model value m_i is a linear combination of the parameter values p_j plus some bias b_i. We can express the same relationship as above with a weight matrix $\boldsymbol{W}$ and a bias vector $\boldsymbol{b}$:

$$\boldsymbol{m}(\boldsymbol{p}) = \boldsymbol{W}\boldsymbol{p} + \boldsymbol{b}.$$

If we now use the method of least squares (χ^2) to estimate the optimal fit parameters $\hat{\boldsymbol{p}}$ we get a very useful property: the parabolic approximation perfectly describes the uncertainties of the optimal fit parameters $\hat{\boldsymbol{p}}$. We can therefore skip the (relatively) expensive process of profiling the parameters!

Let us look at some examples for linear regression in the context of xy fits since those are the most common. Let us therefore assume that we have some y data $\boldsymbol{d}$ measured at x values $\boldsymbol{x} = (0, 1, 2)^T$. The model function most commonly associated with linear regression is the first degree polynomial $f(x) = a + bx$. We can thus

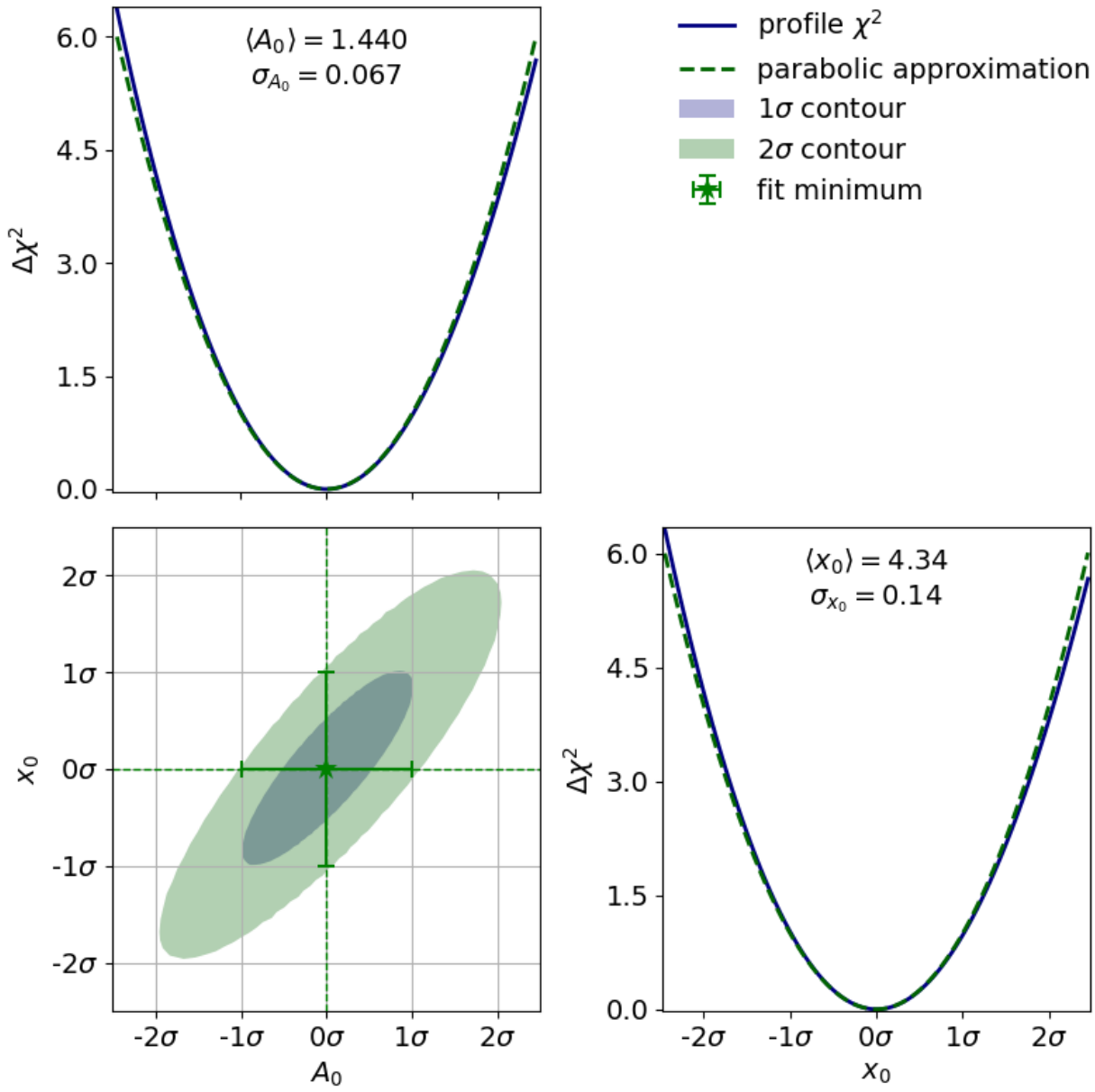


Fig. 5.4: Parameter confidence contours for the exponential fit from the model functions example. The fit is nearly linear on the scale of the uncertainty so the confidence region is close to an ellipse.

express our model like this:

$$\boldsymbol{m}(\boldsymbol{p}) = \boldsymbol{W}\boldsymbol{p} = \left(\boldsymbol{x}^0, \boldsymbol{x}^1\right) \boldsymbol{p} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = a\boldsymbol{x}^0 + b\boldsymbol{x}^1.$$

The upper indices of vectors are to be interpreted as powers of said vectors using the Hadamard/Schur product (component-wise multiplication). In the above equation we only have a weight matrix $W = \left(\boldsymbol{x}^0, \boldsymbol{x}^1\right)$ but no bias vector. We can clearly see that using the first degree polynomial (a line) as our model function results in linear regression. Let's take a look at the third degree polynomial $f(x) = a + bx + cx^2 + dx^3$:

$$\boldsymbol{m}(\boldsymbol{p}) = \boldsymbol{W}\boldsymbol{p} = \left(\boldsymbol{x}^0, \boldsymbol{x}^1, \boldsymbol{x}^2, \boldsymbol{x}^3\right) \boldsymbol{p} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = a\boldsymbol{x}^0 + b\boldsymbol{x}^1 + c\boldsymbol{x}^2 + d\boldsymbol{x}^3.$$

Again we find that the model $\boldsymbol{m}(\boldsymbol{p})$ is a linear function of its parameters $\boldsymbol{p}$. A fit using a third degree polynomial as its model function is therefore also linear regression. This is even though the model function is *not* a linear function of the independent variable x. However, this was never required in our definition of linear regression to begin with because x is not one of our fit parameters. In fact, all *xy* fits using polynomials as model functions fall under linear regression.

5.4.2 Nonlinear Regression

Now that we have defined linear regression, the definition of **nonlinear regression** is rather easy: a regression analysis (fit) that is not linear regression. The natural consequence of this is that the parabolic approximation of the uncertainty of our fit parameters is no longer perfectly accurate. We will therefore need to resort to the profile likelihood method to calculate confidence intervals. The most direct example of nonlinear regression is a fit with a model function that is not a linear function of its parameters, e.g. $f(x) = A \cdot e^{-\lambda x}$. It is simply not possible to express this function using only a finite weight matrix $\boldsymbol{W}$ and a bias vector $\boldsymbol{b}$. We would instead need an infinitely large matrix and infinitely many parameters. With the same *x* vector $\boldsymbol{x} = (0, 1, 2)^T$ as before we find:

$$\boldsymbol{m}(\boldsymbol{p}) = A \cdot e^{-\lambda \boldsymbol{x}} = A \cdot \sum_{k=0}^{\infty} \frac{(-\lambda \boldsymbol{x})^k}{k!}$$

$$= A \cdot \begin{pmatrix} \boldsymbol{x}^0 & -\boldsymbol{x}^1 & \frac{\boldsymbol{x}^2}{2} & -\frac{\boldsymbol{x}^3}{6} & \cdots \end{pmatrix} \begin{pmatrix} \lambda^0 \\ \lambda^1 \\ \lambda^2 \\ \lambda^3 \\ \vdots \end{pmatrix} = A \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & \\ 1 & -1 & \frac{1}{2} & -\frac{1}{6} & \cdots \\ 1 & -2 & 2 & -\frac{4}{3} & \end{pmatrix} \begin{pmatrix} \lambda^0 \\ \lambda^1 \\ \lambda^2 \\ \lambda^3 \\ \vdots \end{pmatrix}.$$

Note: We could of course just cut off the series at some point to approximate the exponential function. This would be equivalent to approximating the exponential function with a polynomial. The parabolic uncertainties of our fit parameters would then be "accurate" but we would only be moving the problem because our model would become less accurate in the process.

Unfortunately, even with a linear model function the fit as a whole can become nonlinear if certain *kafe2* features are used. As of right now these features are uncertainties in *x* direction for *xy* fits and uncertainties relative to the model. This is because when using those features the uncertainties that we feed to our negative log-likelihood are no longer constant. Instead they become a function of the fit parameters: $\sigma_i \rightarrow \sigma_i(\boldsymbol{p})$. As a consequence we have to consider the full Gaussian likelihood rather than just χ^2 to get an unbiased result:

$$\mathrm{NLL}(\boldsymbol{p}) = -2\log L_{\max}(\boldsymbol{p}) + \chi^2(\boldsymbol{p}) = -2\sum_i \log\frac{1}{\sqrt{2\pi}\,\sigma_i(\boldsymbol{p})} + \chi^2(\boldsymbol{p})$$

$$= N\log(2\pi) + 2\sum_i^N \log\sigma_i(\boldsymbol{p}) + \chi^2(\boldsymbol{p}) =: N\log(2\pi) + C_{\mathrm{det}}(\boldsymbol{p}) + \chi^2(\boldsymbol{p}).$$

As with our derivation of χ^2 we end up with a constant term $N\log(2\pi)$ which we can ignore because we are only interested in the differences in cost. We also get a new term $C_{\mathrm{det}}(\boldsymbol{p}) = 2\sum_i^N \log\sigma_i(\boldsymbol{p})$ that we need to consider when our uncertainties depend on our fit parameters. The new term results in higher cost when the uncertainties increase. If we didn't add $C_{\mathrm{det}}(\boldsymbol{p})$ while handling parameter-dependent uncertainties we would end up with a bias towards parameter values for which the uncertainties are increased because those values result in a lower value for χ^2. The subscript "det" is short for determinant, the reason for which should become clear when we look at the full Gaussian likelihood with correlated uncertainties represented by a covariance matrix $\boldsymbol{V}(\boldsymbol{p})$:

$$\mathrm{NLL}(\boldsymbol{p}) = -2\log L_{\max}(\boldsymbol{p}) + \chi^2(\boldsymbol{p}) = -2\log\left[(2\pi)^{-\frac{N}{2}}\frac{1}{\sqrt{\det\boldsymbol{V}(\boldsymbol{p})}}\right] + \chi^2(\boldsymbol{p})$$

$$= N\log(2\pi) + \log\det\boldsymbol{V}(\boldsymbol{p}) + \chi^2(\boldsymbol{p}) =: N\log(2\pi) + C_{\mathrm{det}}(\boldsymbol{p}) + \chi^2(\boldsymbol{p})$$

The constant term is the same as with the uncorrelated uncertainties but term we're interested in has changed to $C_{\mathrm{det}}(\boldsymbol{p}) = \log\det\boldsymbol{V}(\boldsymbol{p})$. If the uncertainties are uncorrelated then the covariance matrix is diagonal and the result is equal to the term we found earlier.

Note: Handling correlated uncertainties that are a function of our fit parameters $\boldsymbol{p}$ is computationally expensive because this means that we need to recalculate the inverse (actually Cholesky decomposition) of our covariance many times which has complexity $O(N^3)$ for N data points - on modern hardware this is typically not an issue though.

Uncertainties In *x* Direction

Now that we know how to handle parameter-dependent uncertainties we can use this knowledge to handle a very common problem: fitting a model with model function $f(x;\boldsymbol{p})$ to data with *x* values x_i and uncertainties in both the *x* and the *y* direction. The uncertainties in the *y* direction $\sigma_{y,i}$ can be used directly. For the *x* uncertainties $\sigma_{x,i}$ we need a trick: we project the uncertainties $\sigma_{x,i}$ onto the *y* axis by multiplying them with the corresponding model function derivative by *x* $f'(x_i;\boldsymbol{p})$:

$$\sigma_{xy,i}(\boldsymbol{p}) = \sqrt{\sigma_{y,i}^2 + (\sigma_{x,i}\cdot f'(x_i;\boldsymbol{p}))^2}.$$

The formula for the pointwise projected *xy* uncertainties $\boldsymbol{\sigma}_{xy}$ can be generalized for the equivalent covariance matrices $\boldsymbol{V}_x$ and $\boldsymbol{V}_y$:

$$\boldsymbol{V}_{xy}(\boldsymbol{p}) = \boldsymbol{V}_y + (f'(\boldsymbol{x};\boldsymbol{p})\cdot f'(\boldsymbol{x};\boldsymbol{p})^T)\circ\boldsymbol{V}_x,$$

where $\circ$ is again the Hadamard product (a.k.a. Schur product) where two matrices are multiplied on a component-by-component basis. We are also implicitly assuming that $f'(\boldsymbol{x}; \boldsymbol{p})$ is a vectorized function à la *NumPy* that returns a vector of derivatives for a vector of x values $\boldsymbol{x}$.

Uncertainties Relative To The Model

Relative uncertainties are very common. For example, the uncertainties of digital multimeters are typically specified as a percentage of the reading. Unfortunately such uncertainties are therefore relative to the true values which we don't know. The standard approach for handling relative uncertainties is therefore to specify them relative to the data points d_i which we do know. However, this approach introduces a bias: if the random fluctuation represented by an uncertainty causes our data d_i to have a reduced (increased) absolute value then the relative uncertainties are underestimated (overestimated). This causes a bias towards models with smaller absolute values in our fit because we are giving data points that randomly happen to have a low absolute value a higher weight than data points with a high absolute value - and this bias increases for large relative uncertainties.

The solution for the bias described above is to specify uncertainties relative to the model $m_i(\boldsymbol{p})$ rather than the data d_i. Because the model "averages" the fluctuations in our data we no longer give a higher weight to data that randomly happens to have a lower absolute value. The price we pay for this is that our total uncertainty becomes a function of our model parameters $\boldsymbol{p}$ which results in an increase in computation time as described above.

Gaussian Approximation Of The Poisson Distribution

kafe2 has a built-in approximation of the Poisson distribution where the Gaussian uncertainty is assumed as:

$$\sigma_i(\boldsymbol{p}) = \sqrt{m_i(\boldsymbol{p})}.$$

The rationale for using the square root of the model $m_i(\boldsymbol{p})$ rather than the square root of the data d_i is the same as with the relative uncertainties described in the previous section. The benefit of using this approximation of the Poisson distribution instead of the Poisson distribution itself is that it is capable of handling additional Gaussian uncertainties on our data.

5.5 Hypothesis Testing

So far we have used cost functions to compare how good or bad certain models and parameter values fit our data relative to each other - but we have never discussed how good or bad a fit is in an absolute sense. Luckily for us there is a metric that we can use: χ^2/NDF, where χ^2 is simply the sum of the squared residuals that we already know and NDF is the **number of degrees of freedom** that our fit has. The basic definition of NDF is that it's simply the number of data points $N_{\boldsymbol{d}}$ minus the number of parameters $N_{\boldsymbol{p}}$:

$$\text{NDF} = N_{\boldsymbol{d}} - N_{\boldsymbol{p}}.$$

Conceptually the number of degrees of freedom are the number of "extra measurements" over the minimum number of data points needed to fully specify a model with $N_{\boldsymbol{p}}$ linearly independent parameters. If our model is not fully specified then our cost function has multiple (or even infinitely many) global minima. For example,

a line with model function $f(x; a, b) = ax + b$ has two linearly independent parameters and as such needs at least two data points to be fully specified.

If our model accurately describes our data, and if our assumptions about the uncertainties of our data are correct, then χ^2/NDF has an expected value of 1. If χ^2/NDF is smaller (larger) than 1 we might be overestimating (underestimating) the uncertainties on our data. If χ^2/NDF is much larger than 1 then our model may not accurately describe our data at all.

To further quantify these rather loose criteria we can make use of **Pearson's chi-squared test**. This is a statistical test that allows us to calculate the probability $P(\chi^2, \text{NDF})$ with which we can expect to observe deviations from our model that are at least as large as the deviations that we saw in our data. To conduct this test we first need to define the so-called $\boldsymbol{\chi^2}$ **distribution**. This distribution has a single parameter k and when sampling from this distribution, the samples from k standard normal distributions x_l are simply squared and then added up:

$$\chi^2(k) = \sum_{l=1}^{k} x_l^2.$$

The deviations of our data relative to its true values (represented by our model) and normalized to its uncertainties follow such standard normal distributions. We can therefore expect the sum of the squares of these deviations $\chi^2(\boldsymbol{p})$ to follow a $\chi^2(k)$ distribution with $k = \text{NDF}$ - if our model and our assumptions about the uncertainties of our data are correct. We can associate the following cumulative distribution function (CDF) $F(k, x)$ with the χ^2 distribution:

$$F(x, k) = \frac{\int_0^{\frac{x}{2}} t^{\frac{k}{2}-1} e^{-t} dt}{\int_0^{\infty} t^{\frac{k}{2}-1} e^{-t} dt}.$$

To calculate the probability $P(\chi^2, \text{NDF})$ with which we would expect a χ^2 value larger than what we got for our fit (i.e. the probability of our fit being worse if we could somehow "reroll" the deviations on our data) we can now simply use:

$$P(\chi^2, \text{NDF}) = 1 - F(\chi^2, \text{NDF}).$$

In *kafe2* $P(\chi^2, \text{NDF})$ is also referred to as the χ^2 probability. We can use this number to determine if deviations from our assumed model are **statistically significant**.

The concept of χ^2/NDF as can be generalized for non-Gaussian likelihoods where the metric becomes **goodness of fit** per degree of freedom GoF/NDF. For a negative log likelihood $\text{NLL}(\boldsymbol{m}(\boldsymbol{p}), \boldsymbol{d})$ with model $\boldsymbol{m}(\boldsymbol{p})$ and data $\boldsymbol{d}$ it is defined like this:

$$\text{GoF/NDF} = \frac{\text{NLL}(\boldsymbol{m}(\hat{\boldsymbol{p}}), \boldsymbol{d}) - \text{NLL}(\boldsymbol{d}, \boldsymbol{d})}{\text{NDF}}.$$

We are subtracting the so-called **saturated likelihood** $\text{NLL}(\boldsymbol{d}, \boldsymbol{d})$ (the minimum value our NLL could have if our model were to perfectly describe our data) from the global cost function minimum $\text{NLL}(\boldsymbol{m}(\hat{\boldsymbol{p}}), \boldsymbol{d})$ and then divide this difference by NDF. As before the expected value of GoF/NDF is 1 if our model and our assumptions about the uncertainties of our data are correct.

5.5.1 Calculating Data Uncertainties from χ^2/NDF

Many fitting tools allow users to fit a model to data without specifying any data uncertainties. This seems to be at odds with our current understanding of Gaussian likelihood-based fits where we always required our data to have some amount of uncertainty. So how does this work? The "solution" is to first give all data points an uncorrelated uncertainty of 1 and to scale these uncertainties *after* the fit in such a way that χ^2/NDF is equal to 1. This approach has a big problem which makes it unsuitable for physics experiments: *we cannot do any hypothesis tests* because we are implicitly assuming that our model is 100% correct. This goes against the very purpose of many physics experiments where experimenters are trying to determine if a theoretical model is consistent with experimental data.

For example, at the Large Hadron Collider the standard model of particle physics has undergone very thorough testing that continues to this day. So far, no statistically significant deviations from the standard model have been found - which is actually a bummer for theoretical physicists. You see, we know for a fact that the standard model is incomplete because (among other things) it does not include gravity. If we were to find an area in which the predictions of the standard model are wrong (beyond the expected uncertainties) this would give theorists an important clue for a new theory that could potentially fix the problems of the standard model.

5.5.2 Fixing And Constraining Parameters

kafe2 allows users to **fix** fit parameters. The practical consequence of this is that one of our fit parameters becomes a constant and is *not* changed during the fit. Because this effectively lowers the number of fit parameters we have to consider the number of fixed parameters N_{fixed} in the calculation of the number of degrees of fredom:

$$\mathrm{NDF} = N_{\boldsymbol{d}} - (N_{\boldsymbol{p}} - N_{\mathrm{fixed}}) = N_{\boldsymbol{d}} - N_{\boldsymbol{p}} + N_{\mathrm{fixed}}.$$

It's also possible to **constrain** fit parameters. Constraints are effectively direct measurements of our fit parameters and they increase the cost of our fit if they are not exactly met. For example, the additional cost C_{con} of a Gaussian constraint for fit parameter a with mean μ_a and standard deviation σ_a can be calculated like this:

$$C_{\mathrm{con}} = \left(\frac{a - \mu_a}{\sigma_a}\right)^2.$$

We can of course generalize this concept to account for correlations between parameters $\boldsymbol{p}$ as defined by a covariance matrix $\boldsymbol{V_p}$:

$$C_{\mathrm{con}} = (\boldsymbol{p} - \boldsymbol{\mu_p})^\intercal \boldsymbol{V_p}^{-1} (\boldsymbol{p} - \boldsymbol{\mu_p}).$$

If we define any constraints we are adding more data to our fit. We therefore also have to increase NDF by the number of constraints N_{con}:

$$\mathrm{NDF} = N_{\boldsymbol{d}} + N_{\mathrm{con}} - N_{\boldsymbol{p}} + N_{\mathrm{fixed}}.$$

A simple parameter constraint that constrains a single parameter counts as one constraint. On the other hand, a matrix parameter constraint that constrains n parameters at once counts as n constraints.

5.6 Data/Fit Types

A large percentage of fits can be expressed as an `XYFit` (page 119). However, there are cases where an `XYFit` (page 119) is not suitable; *kafe2* offers alternatives **fit types** for those cases. Typically these alternative fit types are associated with alternative **data (container) types** so both concepts are explained simultaneously in this section. For example, an `XYFit` (page 119) uses an `XYContainer` (page 114) to hold its *xy* data while a `HistFit` (page 145) uses a `HistContainer` (page 141) to hold and bin its data.

For the following considerations $\boldsymbol{p}$ always describes the vector of fit parameters. Unless mentioned otherwise fits calculate their cost from a data vector $\boldsymbol{d}$ and a model vector $\boldsymbol{m}$.

5.6.1 XYFit

Let's start with the most common fit type: `XYFit` (page 119). The data associated with this fit type consists of two vectors of equal length: a vector of *x* data $\boldsymbol{x}$ and a vector of *y* data $\boldsymbol{d}$. Our model values are calculated as $\boldsymbol{m}(\boldsymbol{x};\boldsymbol{p}) = f(\boldsymbol{x};\boldsymbol{p})$, they are a function of our *x* data and our fit parameters. As the difference in notation implies the *x* and *y* axes are *not* treated in the same way. The *x* axis is interpreted as the **independent variable** of our fit while the *y* data values $\boldsymbol{d}$ and *y* model values $\boldsymbol{m}(\boldsymbol{x};(\boldsymbol{p}))$ are what we ultimately compare to calculate the negative log-likelihood.

Note: Although we only have a few discreet *x* values for which we have to calculate our model $\boldsymbol{m}(\boldsymbol{x};\boldsymbol{p})$, our model function $f(x;\boldsymbol{p})$ is still expected to be a continuous function of x.

A visualization of `XYFit` (page 119) is fairly straightforward: the *xy** axes of our fix directly correspond to the axes of a plot.

5.6.2 IndexedFit

Conceptually `IndexedFit` (page 137) is a simplified version of `XYFit` (page 119): we only have a data vector $\boldsymbol{d}$ and no independent variable at all. Instead we calculate the model vector $\boldsymbol{m}(\boldsymbol{p})$ as a function of just the fit parameters. In *kafe2* `IndexedFit` (page 137) is visualized by interpreting the indices of the data/model vectors as *x* values and the corresponding *xth* entry of those vectors as the *y* value.

5.6.3 HistFit

`HistFit` (page 145) handles N one-dimensional data points $\boldsymbol{x}$ by binning them according to some bin edges $x_0 < ... < x_k < ... < x_K$ to form our data vector $\boldsymbol{d} \in \mathbb{R}^K$. The model function $f(x;\boldsymbol{p})$ that is fitted to these bins is a **probability density function** for the observed values $\boldsymbol{x}$. The bin heights $\boldsymbol{m}(\boldsymbol{p})$ predicted by our model are obtained by integrating $f(x;\boldsymbol{p})$ over a given bin and multiplying the result with N:

$$m_k(\boldsymbol{p}) = N \int_{x_{k-1}}^{x_k} f(t;\boldsymbol{p}) dt.$$

The amplitude of our distribution is therefore *not* one of the fit parameters; we are effectively fitting a density function to a normalized histogram.

Unlike with *XYFit* (page 119) or *IndexedFit* (page 137) the default distribution assumed for the data of a *HistFit* (page 145) is the Poisson distribution rather than the normal distribution.

5.6.4 UnbinnedFit

Just like *HistFit* (page 145) an `UnbinnedFit` accepts a vector of N one-dimensional data points $\boldsymbol{x}$ in conjunction with a probability density function $f(x; \boldsymbol{p})$ for these values as its model function. As the name implies the data is not binned. Instead, because our model function can be interpreted as a probability density we can simply calculate the negative log-likelihood like this:

$$\mathrm{NLL}(\boldsymbol{p}) = -2 \sum_{n=1}^{N} \log f(x_n; \boldsymbol{p}).$$

In *kafe2* `UnbinnedFit` is visualized by interpreting the independent variable as the *x* axis of a plot and the height of the probability density function as the *y* axis. Additionally, a thin, vertical line is added for each data point to indicate the density of our data.

5.6.5 CustomFit

Unlike the other fit types discussed so far, `CustomFit` does not explicitly use data $\boldsymbol{d}$ or a model $\boldsymbol{m}$. Instead the user has to manually define how the cost function value is calculated from the fit parameters $\boldsymbol{p}$. Because any potential data is outside *kafe2* there is no built-in visualization (plotting) available except for the fit parameter profiles/contours calculated by `ContoursProfiler`.

5.6.6 MultiFit

A `MultiFit` is constructed from N regular fits with cost functions $C_i(\boldsymbol{p})$. The idea behind `MultiFit` is rather simple: multiple models that share at least one parameter are simultaneously fitted to their respective data. In accordance with the method of maximum likelihood the optimal fit parameters are those that make the observed combination of individual datasets the most likely. The corresponding cost function can simply be calculated as:

$$C_{\text{multi}}(\boldsymbol{p}) = \sum_{i}^{N} C_i(\boldsymbol{p}).$$

If a `MultiFit` is built from several fits that assume Gaussian uncertainties, it's possible to specify uncertainties that are correlated between those fits. For example, in the case of two fits that have a fully correlated source of uncertainty expressed by a covariance matrix $\boldsymbol{V}_{\text{shared}}$ the effective covariance matrix $\boldsymbol{V}_{\text{multi}}$ for the `MultiFit` becomes:

$$\boldsymbol{V}_{\text{multi}} = \begin{pmatrix} \boldsymbol{V}_{\text{shared}} & \boldsymbol{V}_{\text{shared}} \\ \boldsymbol{V}_{\text{shared}} & \boldsymbol{V}_{\text{shared}} \end{pmatrix}.$$

5.7 Cost Functions

So far we almost universally assumed that the uncertainties of our data can be described with a normal distribution. However, this is not always the case. For example, the number of radioactive decays in a given time interval follows a Poisson distribution. In *kafe2* such distinctions are handled via the **cost function**, the function that in one way or another calculates a scalar cost from the data, model, and uncertainties of a fit. This section describes the built-in cost functions that *kafe2** provides.

5.7.1 χ^2 Cost Function

The by far most common cost function used is the χ^2 cost function that assumes a normal distribution for the uncertainties of our data. In *kafe2* the name is strictly speaking a misnomer because the actual cost calculation considers the full likelihood rather than just χ^2 in order to handle non-constant uncertainties. For N data points d_i with corresponding model values $m_i(\boldsymbol{p})$ and uncorrelated (but possible non-constant) uncertainties $\sigma_i(\boldsymbol{p})$ the cost function value is calculated like this:

$$\mathrm{NLL}(\boldsymbol{p}) = C_{\mathrm{det}}(\boldsymbol{p}) + \chi^2(\boldsymbol{p}) = \sum_i^N 2\log\sigma_i(\boldsymbol{p}) + \left(\frac{d_i - m_i(\boldsymbol{p})}{\sigma_i(\boldsymbol{p})}\right)^2.$$

If the uncertainties are instead correlated as described by a covariance matrix $\boldsymbol{V}(\boldsymbol{p})$ the cost function value becomes:

$$\mathrm{NLL}(\boldsymbol{p}) = C_{\mathrm{det}}(\boldsymbol{p}) + \chi^2(\boldsymbol{p}) = \log\det\boldsymbol{V}(\boldsymbol{p}) + (\boldsymbol{d} - \boldsymbol{m}(\boldsymbol{p}))^T\,\boldsymbol{V}(\boldsymbol{p})^{-1}\,(\boldsymbol{d} - \boldsymbol{m}(\boldsymbol{p})).$$

5.7.2 Poisson Cost Function

The Poisson cost function assumes - as the name implies - a Poisson distribution for our data. Compared to the normal distribution the Poisson distribution has two important features: Firstly the data values d_i (but not the model values $m_i(\boldsymbol{p})$) have to be positive integers, and secondly the mean and variance are inherently linked. We can define the likelihood function $\mathcal{L}(\boldsymbol{p})$ of the Poisson distribution like this:

$$\mathcal{L}(\boldsymbol{p}) = \prod_i^N \frac{m_i(\boldsymbol{p})^{d_i}\; e^{-m_i(\boldsymbol{p})}}{d_i!}.$$

The negative log-likelihood $\mathrm{NLL}(\boldsymbol{p})$ thus becomes:

$$\mathrm{NLL}(\boldsymbol{p}) = -2\log\mathcal{L} = 2\sum_i^N m_i(\boldsymbol{p}) - d_i\log m_i(\boldsymbol{p}) + \frac{d_i(d_i+1)}{2}.$$

Notably $\mathrm{NLL}(\boldsymbol{p})$ depends only on the data d_i and the model $m_i(\boldsymbol{p})$ but *not* on any specified uncertainties $\boldsymbol{\sigma}$. The advantage is that we don't need to specify any uncertainties - but the significant disadvantage is that we *can't* specify any uncertainties either. In such cases the cost function in the following section will need to be used.

5.7.3 Gauss Approximation Cost Function

Because a Poisson distribution cannot handle Gaussian data uncertainties the Poisson distribution is frequently approximated with a normal distribution. The easiest approach is to simply derive the uncertainties σ_i from the data d_i:

$$\sigma_i = \sqrt{d_i}.$$

However, as described in the previous section about nonlinear regression, this leads to a bias towards small model values $m_i(\boldsymbol{p})$. In *kafe2* the uncertainties are therefore derived from the model values:

$$\sigma_i = \sqrt{m_i(\boldsymbol{p})}.$$

Just like before these uncertainties can be easily combined with other sources of uncertainty by simply adding up the (co)variances. However, this approach has an important limitation: it is only valid if the model values $m_i(\boldsymbol{p})$ are large enough (something like $m_i(\boldsymbol{p}) \geq 10$). This is because for small model values the asymmetry of the Poisson distribution and the portion of the normal distribution that resides in the unphysical region with $m_i(\boldsymbol{p}) < 0$ are no longer negligible.

5.8 Numerical Considerations

The mathematical description of χ^2 shown so far makes use of the inverse of the covariance matrix $\boldsymbol{V}^{-1}$. However, *kafe2* does *not* actually calculate $\boldsymbol{V}^{-1}$. Instead the Cholesky decomposition $\boldsymbol{L}\boldsymbol{L}^T = \boldsymbol{V}$ of the covariance matrix is being used where $\boldsymbol{L}$ is a lower triangular matrix. Calculating $\boldsymbol{L}$ is much faster than calculating $\boldsymbol{V}^{-1}$ and it also reduces the rounding error from floating point operations.

We can always calculate a Cholesky decomposition for a matrix that is symmetrical and positive-definite. Obviously a covariance matrix is symmetrical by definition. And because all eigenvalues of a covariance matrix are (typically) positive a covariance matrix is (typically) also positive definite.

Note: The eigenvalues of a covariance matrix represent the equivalent variances in a coordinate system where said variances are uncorrelated (see principal component analysis). The eigenvalues are therefore all positive unless the uncertainties of two or more data points are fully correlated. In this case some of the eigenvalues are 0. However, as a consequence the covariance matrix would also no longer have full rank so we wouldn't be able to invert it either.

Because $\boldsymbol{L}$ is a triangular matrix solving the corresponding system of linear equations for the residual vector $\boldsymbol{r} = \boldsymbol{d} - \boldsymbol{m}$ (difference between data and model) can be done very quickly:

$$\boldsymbol{L}\boldsymbol{x} = \boldsymbol{r}.$$

With $\boldsymbol{x} = \boldsymbol{L}^{-1}\boldsymbol{r}$ we now find:

$$\chi^2 = \boldsymbol{r}^T\boldsymbol{V}^{-1}\boldsymbol{r} = \boldsymbol{r}^T(\boldsymbol{L}\boldsymbol{L}^T)^{-1}\boldsymbol{r} = \boldsymbol{r}^T\boldsymbol{L}^{-T}\boldsymbol{L}^{-1}\boldsymbol{r} = \boldsymbol{x}^T\boldsymbol{x}.$$

Because $\boldsymbol{L}$ is a triangular matrix it can also be used to efficiently calculate $\log\det(\boldsymbol{V})$:

$$\det(\boldsymbol{L}) = \det(\boldsymbol{L}^T) = \prod_i^N L_{ii},$$

$$C_{\text{det}} = \log\det(\boldsymbol{V}) = \log\det(\boldsymbol{L}\boldsymbol{L}^T) = \log(\det\boldsymbol{L}\cdot\det\boldsymbol{L}^T) = \log(\prod_i^N L_{ii}^2) = 2\sum_i^N \log L_{ii}.$$

6 Developer Guide

This developer guide provides information for developers who wish to modify *kafe2*. It currently only covers tools needed for development. A description of the software design will be added at some point in the future.

6.1 Tools

This section covers software dependencies needed for development and how to use them. All command line instructions below assume that the current working directory is the root of the *kafe2* repository.

6.1.1 Requirements

The following software should be installed on your machine:

- git
- Python, both Python 2 *and* Python 3
- ROOT

Additionally, the following Python packages should be installed:

- NumPy
- Scipy
- iminuit
- matplotlib
- numdifftools
- PyYaml
- six
- funcsigs
- tabulate

- coverage
- Sphinx
- Sphinx Bootstrap Theme
- mock

To install all of these packages automatically run:

```
pip2 install -r dev_dependencies.txt
pip3 install -r dev_dependencies.txt
```

6.1.2 Running Unit Tests

To run unit tests for both Python 2 and Python 3 run:

```
python2 -m unittest discover -v -s kafe2/test
python3 -m unittest discover -v -s kafe2/test
```

6.1.3 Determining Test Coverage

To determine the test coverage run:

```
coverage run
```

This will run Python3 unit tests and keep track of the lines that were executed. To print out a general report of the coverage run:

```
coverage report
```

To get the lines of a specific file that were not tested run:

```
coverage report -m path/to/file
```

For a graphical representation of the result an HTML document can be created:

```
coverage html
```

6.1.4 Building the Documentation

To build the documentation run:

```
cd doc
make
```

For creating only the html or pdf documentation run `make html` or `make latex`. Cleaning the output directories can be done with `make clean`.

6.2 Coding Style

In general the code of *kafe2* tries to follow the guidelines of PEP-8. We've decided to use a maximum line length of 100 characters for all files.

But please, do not try to enforce this only for the sake of updating the style. Only update the coding style if you're already performing other changes on a particular section. There might be some sections in the source code which do not follow the general style guidelines, this is okay, as long as the code is working and understandable.

6.2.1 Docstrings

Documenting every method is a very good idea in general, so that a user or developer can quickly and easily understand what a specific code block does and what it is used for. Sphinx is used for creating the documentation, hence we use the Sphinx docstring format.

Since we also support Python 2, we can't use Python's own type hints introduced with Python 3.5. Modern IDEs can also perform type checking with the help of docstrings. That's why, in general, we try to include the types inside the docstrings.

7

API Documentation

This page contains documentation which was automatically extracted from docstrings attached to the *kafe2* source code. All major classes, methods and functions provided by *kafe2* are documented here. For further information, or if in doubt about the exact functionality, users are invited to consult the source code itself. If you notice a mistake in the *kafe2* documentation, or if you think that a particular part needs to be better documented, please open an issue on the kafe2 GitHub page.

7.1 *kafe2* Wrappers

The easiest way to use *kafe2* (as part of a *Python* program) is to use the wrapper functions below. These functions provide pre-configured pipelines for the most common use cases and do not require the user to manually manage objects.

kafe2.fit.util.wrapper.**xy_fit**(*x_data*, *y_data*, *model_function=None*, *p0=None*, *dp0=None*, *x_error=None*, *y_error=None*, *x_error_rel=None*, *y_error_rel=None*, *x_error_cor=None*, *y_error_cor=None*, *x_error_cor_rel=None*, *y_error_cor_rel=None*, *errors_rel_to_model=True*, *limits=None*, *constraints=None*, *report=False*, *profile=None*, *save=True*)

Built-in function for fitting a model function to xy data.

Interpretation of x_error, y_error, x_error_rel, and y_error_rel: If the input error is a simple float it is broadcast across the entire data vector. If the input error is a one-dimensional vector it is interpreted as a pointwise error vector. If the input error is a two-dimensional matrix it is interpreted as a covariance matrix.

Interpretation of x_error_cor, y_error_cor, x_error_cor_rel, and y_error_cor_rel: If the input error is a simple float it is broadcast across the entire data vector. If the input error is a one-dimensional vector then each individual value is added as a separate error that is being broadcast across the entire data vector.

Parameters

- **x_data** (*Sequence[float]*) – the x data values for the fit. Must be one-dimensional.

- **y_data** (*Sequence[float]*) – the y data values for the fit. Must be one-dimensional.
- **model_function** (*Callable*) – The model function as a native Python function where the first argument denotes the independent *x* variable. Alternatively an already defined `XYModelFunction` object. Defaults to a straight line.
- **p0** (*Sequence[float]*) – the initial parameter values for the fit.
- **dp0** (*Sequence[float]*) – the initial parameter step size for the fit.
- **x_error** (*float or Sequence[float]*) – uncorrelated absolute *x* error.
- **y_error** (*float or Sequence[float]*) – uncorrelated absolute *y* error.
- **x_error_rel** (*float or Sequence[float]*) – uncorrelated relative *x* error.
- **y_error_rel** (*float or Sequence[float]*) – uncorrelated relative *y* error.
- **x_error_cor** (*float or Sequence[float]*) – correlated absolute *x* error.
- **y_error_cor** (*float or Sequence[float]*) – correlated absolute *y* error.
- **x_error_cor_rel** (*float or Sequence[float]*) – correlated relative *x* error.
- **y_error_cor_rel** (*float or Sequence[float]*) – correlated relative *y* error.
- **errors_rel_to_model** (*bool*) – whether the relative *y* errors should be relative to the model. Otherwise they are relative to the data.
- **limits** (*Sequence or Sequence[Union[list, tuple]]*) – limits to be applied to the model parameter. The expected format for each limit is an iterable consisting of the parameter name, the lower bound, and then the upper bound. An iterable of limits can be passed to limit multiple parameters.
- **constraints** (*Sequence or Sequence[Union[list, tuple]]*) – constraints to be applied to the model parameter. The expected format for each constraint is an iterable consisting of the parameter name, the parameter mean, and then the parameter uncertainty. An iterable of constraints can be passed to limit multiple parameters.
- **report** (*bool*) – whether a report of the data and fit results should be printed to the console.
- **profile** (*bool*) – whether the profile likelihood method should be used for asymmetric parameter errors and profile/contour plots.
- **save** (*bool*) – whether the fit results should be saved to disk under *results*.

Returns the fit results.

Return type dict

kafe2.fit.util.wrapper.**plot**(*fits=- 1*, *x_label=None*, *y_label=None*, *data_label=None*, *model_label=None*, *error_band_label=None*, *x_range=None*, *y_range=None*, *x_scale=None*, *y_scale=None*, *x_ticks=None*, *y_ticks=None*, *parameter_names=None*, *model_name=None*, *model_expression=None*, *legend=True*, *fit_info=True*, *error_band=True*, *profile=None*, *plot_profile=None*, *show=True*, *save=True*)

Plots kafe2 fits.

Parameters

- **fits** (int or *FitBase* (page 160)) – which kafe2 fits to use for the plot. A positive integer is interpreted as the fit with the given index that has been performed since the program started. A negative integer *-n* is interpreted as the last *n* fits. kafe2 fit objects are used directly.
- **x_label** (*str*) – the *x* axis label.
- **y_label** (*str*) – the *y* axis label.
- **data_label** (*str or Sequence[str]*) – the data label(s) in the legend.
- **model_label** (*str or Sequence[str]*) – the model label(s) in the legend (under data label).
- **error_band_label** (*str or Sequence[str]*) – the error band label(s) in the legend.
- **x_range** (*Sequence[float], len(x_range) == 2*) – *x* range for the plot.
- **y_range** (*Sequence[float], len(y_range) == 2*) – *y* range for the plot.
- **x_scale** (*"linear" or "log"*) – the scale to use for the *x* axis.
- **y_scale** (*"linear" or "log"*) – the scale to use for the *y* axis.
- **x_ticks** (*Sequence[float]*) – the ticks at which to show values on the *x* axis.
- **y_ticks** (*Sequence[float]*) – the ticks at which to show values on the *y* axis.
- **parameter_names** (*dict*) – custom parameter LaTeX names to display in the plot. The dictionary keys are the regular parameter names and the dictionary values are the names to show in the plot.
- **model_name** (*str or Sequence[str]*) – the model LaTeX name(s) in the legend (in the mathematical expression of the model function).
- **model_expression** (*str or Sequence[str]*) – the model LaTeX expression(s) in the legend.

- **legend** (*bool*) – whether the legend should be shown.
- **fit_info** (*bool*) – whether the fit information (fit results, goodness of fit) should be shown.
- **error_band** (*bool*) – whether the model error band should be shown.
- **profile** (*bool*) – whether the profile likelihood method should be used for asymmetric parameter errors and profile/contour plots.
- **plot_profile** (*bool*) – whether the profile plots should be created.
- **show** (*bool*) – whether the plots should be shown.
- **save** (*bool*) – whether the plots should be saved to disk under *results*.

Returns a *kafe2* plot object containing the relevant matplotlib plots.

Return type *Plot* (page 175)

kafe2.fit.util.wrapper.**k2Fit**(*func*, *x*, *y*, *sx=None*, *sy=None*, *srelx=None*, *srely=None*, *xabscor=None*, *yabscor=None*, *xrelcor=None*, *yrelcor=None*, *ref_to_model=True*, *constraints=None*, *p0=None*, *dp0=None*, *limits=None*, *plot=True*, *axis_labels=['x-data', 'y-data']*, *data_legend='data'*, *model_expression=None*, *model_name=None*, *model_legend='model'*, *model_band='$\\pm 1 \\sigma$'*, *fit_info=True*, *plot_band=True*, *asym_parerrs=True*, *plot_cor=False*, *showplots=True*, *quiet=True*)

Legacy function for backwards compatibility with *PhyPraKit*. **New code should not use this function.** Fits a model to *xy* data and plots the results.

Interpretation of sx, sy, srelx, and srely: If the input error is a simple float it is broadcast across the entire data vector. If the input error is a one-dimensional vector it is interpreted as a pointwise error vector. If the input error is a two-dimensional matrix it is interpreted as a covariance matrix.

Interpretation of xabscor, yabscor, xrelcor, and yrelcor: If the input error is a simple float it is broadcast across the entire data vector. If the input error is a one-dimensional vector then each individual value is added as a separate error that is being broadcast across the entire data vector.

Parameters

- **func** (*Callable*) – The model function as a native Python function where the first argument denotes the independent *x* variable. Alternatively an already defined `XYModelFunction` object. Defaults to a straight line.
- **x** (*Sequence[float]*) – the *x* data values for the fit. Must be one-dimensional.
- **y** (*Sequence[float]*) – the *y* data values for the fit. Must be one-dimensional.
- **sx** (*float or Sequence[float]*) – uncorrelated absolute *x* error.
- **sy** (*float or Sequence[float]*) – uncorrelated absolute *y* error.
- **srelx** (*float or Sequence[float]*) – uncorrelated relative *x* error.
- **srely** (*float or Sequence[float]*) – uncorrelated relative *y* error.

- **xabscor** (*float or Sequence[float]*) – correlated absolute *x* error.
- **yabscor** (*float or Sequence[float]*) – correlated absolute *y* error.
- **xrelcor** (*float or Sequence[float]*) – correlated relative *x* error.
- **yrelcor** (*float or Sequence[float]*) – correlated relative *y* error.
- **ref_to_model** (*bool*) – whether the relative *y* errors should be relative to the model. Otherwise they are relative to the data.
- **constraints** (*Sequence or Sequence[Union[list, tuple]]*) – constraints to be applied to the model parameter. The expected format for each constraint is an iterable consisting of the parameter name, the parameter mean, and then the parameter uncertainty. An iterable of constraints can be passed to limit multiple parameters.
- **p0** (*Sequence[float]*) – the initial parameter values for the fit.
- **dp0** (*Sequence[float]*) – the initial parameter step size for the fit.
- **limits** (*Sequence or Sequence[Union[list, tuple]]*) – limits to be applied to the model parameter. The expected format for each limit is an iterable consisting of the parameter name, the lower bound, and then the upper bound. An iterable of limits can be passed to limit multiple parameters.
- **plot** (*bool*) – whether the fit results should be plotted.
- **axis_labels** (*Sequence[str]*) – the labels for the *x* and *y* axis.
- **data_legend** (*str*) – the data label in the legend.
- **model_expression** (*str*) – the model LaTeX expression in the legend.
- **model_name** (*str*) – the model LaTeX name in the legend (in the mathematical expression of the model function).
- **model_legend** (*str*) – the model label in the legend (under data label).
- **model_band** (*str*) – the error band label in the legend.
- **fit_info** (*bool*) – whether the fit information (fit results, goodness of fit) should be shown.
- **plot_band** (*bool*) – whether the model error band should be shown.
- **asym_parerrs** (*bool*) – whether the profile likelihood method should be used for asymmetric parameter errors.
- **plot_cor** (*bool*) – whether the profile plots should be created.
- **showplots** (*bool*) – whether the plots should be shown.
- **report** (*bool*) – whether the report of the data and fit results should be suppressed.

Returns a tuple containing the parameter values, the parameter errors, the parameter correlation matrix, and the minimal χ^2 cost function value.

Return type tuple

7.2 *kafe2* Oject-Oriented Programming

7.3 Parameter Estimation Tools: `fit`

The *kafe2.fit* (page 112) module provides an object-oriented toolkit for estimating model parameters from data ("fitting").

It distinguishes between a number of different data types:

- *xy* data (dedicated submodule: *xy* (page 114)),
- series of indexed measurements (dedicated submodule: *indexed* (page 134)),
- histograms (dedicated submodule: *histogram* (page 141)),
- raw 1D data using the method of maximum likelihood ("unbinned fit", dedicated submodule: *histogram* (page 141)), and
- direct minimization of a cost function (dedicated submodule: `custom`).

Each of the above data types has its own particularities when it comes to fitting. The main difference is due to the way uncertainties can be defined and interpreted for each type of data and how the fit results are presented.

7.3.1 XY Data

For *xy* data, one data set consists of a list of N distinct y measurements d_i with the (discrete) index i ranging from 0 to $N-1$. The measurements were taken at x values x_i. For each measurement in the series, one or more uncertainty sources can be defined, each being a numerical estimate of how much the respective measurement has fluctuated from the "true values". Correlations between uncertainties on separate measurements d_i and d_j can also be taken into account by using *covariance/correlation matrices*.

Additional uncertainites on x_i can also be defined. When fitting an *xy* model to data they are converted to y uncertainties via multiplication with the derivative of the model function by x. When plotting the result of *xy* fits, the model function is displayed as a continuous function of x, and an *error band* can be computed to reflect the model uncertainty, as determined by propagating the parameter uncertainties onto the *y* axis.

The following objects are provided for handling *xy* data:

- *XYContainer* (page 114): data container for storing *xy* data
- *XYParametricModel* (page 129): corresponding model
- *XYFit* (page 119): a fit of a parametric model to *xy* data

7.3.2 Indexed data

Compared to *xy* data indexed data no longer has an explicit x axis. The data simply appears as an indexed list of data points. As a consequence the model function does not expect an indendent variable.

The following objects are provided for handling indexed data, as described above:

- `IndexedContainer` (page 134): data container for storing indexed data
- `IndexedParametricModel` (page 139): corresponding model
- `IndexedFit` (page 137): a fit of a parametric model to *indexed* data

7.3.3 Histograms

kafe2 is also able to handle *histograms*. Histograms organize measurements whose values can fall anywhere across a continuum of values into a number of discrete regions or "bins". Typically, the continuous "measurement space" (a closed real interval $[x_{\text{min}}, x_{\text{max}}]$) is subdivided into a sequence of successive intervals at the "bin edges" $x_{\text{min}} < x_1 < x_2 < \ldots < x_{\text{max}}$. Whenever a measurement falls into one of the bins, the value of that histogram bin is incremented by one. A histogram is completely defined by its bin edges and the bin values.

Note: The bin numbering starts at 1 for the first bin and ends at N, where N is defined as the *size* of the histogram. The bin numbers 0 and $N + 1$ refer to the underflow (below x_{min}) and overflow bin (above x_{max}), respectively.

Defining a parametric model for histograms is not as straightforward as for *xy* and indexed data. Seeing as they keep track of the number of entries in different intervals of the continuum, the bin values can be interpreted using probability theory.

As the number of entries approaches infinity, the number of entries n in the bin covering an interval $[a, b)$, divided by the total number of entries N_{E}, will approach the probablity of an event landing in that bin:

$$\lim_{N_{\text{E}} \to \infty} \frac{n}{N_{\text{E}}} = \int_a^b f(x)\,\mathrm{d}x = F(b) - F(a)$$

In the above formula, $f(x)$ is the *probability density function*, and $F(x)$ is an antiderivative of f (for example the *cumulative distribution function*).

Using the above relation, the model prediction m for the bin $[a, b)$ can be defined as:

$$m = N_{\text{E}} \int_a^b f(x)\,\mathrm{d}x = N_{\text{E}}\,(F(b) - F(a))$$

This means that, for histograms, the model density $f(x)$ needs to be specified as the model function. The model is then calculated by numerically integrating this function over each bin.

An alternative would be to specify the model density *antiderviative* F alongside the model, so that the model can be calculated as a simple difference, rather than as an integral.

The following objects are provided for handling histograms:

- `HistContainer` (page 141): data container for storing histograms

- *HistParametricModel* (page 146): corresponding model
- *HistFit* (page 145): a fit of a parametric model to histograms

7.3.4 Unbinned

If data is treated as *unbinned* the model function $f(x)$ is interpreted as a *model density function*. The cost function value C is then directly calculated as the negative log-likelihood of the data given said PDF:

$$C = -2 \sum_{i=0}^{N-1} \ln f(x_i).$$

An unbinned fit is the edge case of a histogram fit for as the individual bins become infinitessimally thin.

The following objects are provided for handling unbinned data:

- `UnbinnedContainer`: data container for storing unbinned data
- `UnbinnedParametricModel`: corresponding model
- `UnbinnedFit`: a fit of a parametric model to unbinned data

7.3.5 Custom

Lets the user directly define a cost function. Since this fit type does not have explicit data the fit results cannot be plotted automatically.

The following objects are provided for custom fits:

- `CustomFit`: a fit for minimizing a cost function

7.3.6 Plots

For creating graphical representations of fits, the `Plot` is provided. It can be instantiated with any fit object (or list of fit objects) as an argument and will produce one or more plots accordingly using *matplotlib*.

synopsis This module contains specialized objects for storing measurement data, defining and fitting parametric models to these data and producing graphical representations ("plots") of the result. It relies on the `kafe2.core` module for basic functionality.

7.3.7 Tools for Fitting *xy* Data: `xy`

This submodule provides the necessary objects for parameter estimation using data consisting of ordered *xy* pairs. This fit type is used for most cases e.g. when performing fits for the first time or in physics laboratory courses.

synopsis This submodule provides the necessary objects for parameter estimation using data consisting of ordered *xy* pairs.

class kafe2.fit.xy.**XYContainer**(*x_data*, *y_data*, *dtype=<class 'float'>*)

Bases: *kafe2.fit.indexed.container.IndexedContainer* (page 134)

This object is a specialized data container for *xy* data.

Construct a container for *xy* data:

Parameters

- **x_data** (*Sequence[dtype]*) – 1D array of measurement *x* values.
- **y_data** (*Sequence[dtype]*) – 1D array of measurement *y* values.
- **dtype** (*type*) – Data type of the measurements.

property size

Number of data points.

Return type int

property data

2D array with shape `(2, size)` containing a copy of the data stored in this container.

Return type numpy.ndarray

property x

1D array of length `[size]` containing the *x* data.

Return type numpy.ndarray

property x_err

1D array containing the absolute total data *x* uncertainties.

Return type numpy.ndarray

property x_cov_mat

2D array of shape `(size, size)` containing the absolute data *x* covariance matrix.

Return type numpy.ndarray

property x_cov_mat_inverse

2D array of shape `(size, size)` containing the inverse of the absolute data *x* covariance matrix. `None` if singular.

Return type numpy.ndarray or None

property x_cor_mat

2D array of shape `(size, size)` containing the absolute data *x* correlation matrix.

Return type numpy.ndarray

property y

1D array of length `size` containing the *y* data.

Return type numpy.ndarray

property y_err

1D array of length `size` containing the absolute total data *y* uncertainties.

Return type numpy.ndarray

property y_cov_mat

2D array of shape `(size, size)` containing the absolute data *y* covariance matrix.

Return type numpy.ndarray

property y_cov_mat_inverse

2D array of shape `(size, size)` containing the inverse of absolute data *y* covariance matrix. `None` if singular.

Return type numpy.ndarray

property y_cor_mat

2D array of shape `(size, size)` containing the absolute data *y* correlation matrix.

Return type numpy.ndarray

property x_range

Minimum and maximum values of the *x* data.

Return type tuple[float, float]

property y_range

Minimum and maximum values of the *y* data.

Return type tuple[float, float]

add_error(*axis*, *err_val*, *name=None*, *correlation=0*, *relative=False*)

Add an uncertainty source for an axis to the data container.

Parameters

- **axis** (*str or int*) – `'x'`/`0` or `'y'`/`1`
- **err_val** (*float or Sequence[float]*) – Pointwise uncertainties or a single uncertainty for all data points.
- **name** (*str or None*) – Unique name for this uncertainty source. If `None`, the name of the error source will be set to a random alphanumeric string.
- **correlation** (*float*) – Correlation coefficient between any two distinct data points.
- **relative** (*bool*) – If `True`, **err_val** will be interpreted as a *relative* uncertainty.

Returns An error id uniquely identifying the created error source.

Return type str

add_matrix_error(*axis*, *err_matrix*, *matrix_type*, *name=None*, *err_val=None*, *relative=False*)

Add a matrix uncertainty source for an axis to the data container.

Parameters

- **axis** (*str or int*) – 'x'/0 or 'y'/1
- **err_matrix** (*numpy.ndarray*) – 2D array of shape (size, size) containing the covariance or correlation matrix
- **matrix_type** (*str*) – One of 'covariance'/'cov' or 'correlation'/'cor'.
- **name** (*str or None*) – Unique name for this uncertainty source. If None, the name of the error source will be set to a random alphanumeric string.
- **err_val** (*Sequence[float]*) – The pointwise uncertainties. This is mandatory if only a correlation matrix is given.
- **relative** (*bool*) – If True, the covariance matrix and/or **err_val** will be interpreted as a *relative* uncertainty.

Returns An error id uniquely identifying the created error source.

Return type str

get_total_error(*axis*)

Get the error object representing the total uncertainty for a specific axis.

Parameters **axis** (*str or int*) – 'x'/0 or 'y'/1

Returns Error object representing the total uncertainty.

Return type kafe2.core.error.MatrixGaussianError

property has_x_errors

True if at least one *x* uncertainty source is defined for the data container.

Return type bool

property has_uncor_x_errors

True if at least one *x* uncertainty source, which is not fully correlated, is defined for the data container.

Return type bool

property has_y_errors

True if at least one *x* uncertainty source is defined for the data container.

Return type bool

class kafe2.fit.xy.**XYCostFunction_Chi2**(*errors_to_use='covariance'*, *fallback_on_singular=True*, *axes_to_use='xy'*, *add_constraint_cost=True*, *add_determinant_cost=True*)

Bases: *kafe2.fit._base.cost.CostFunction_Chi2* (page 152)

Built-in least-squares cost function for *xy* data.

Parameters

- **errors_to_use** (*str or None*) – Which errors to use when calculating χ^2. This is either *'covariance'*, `'pointwise'` or `None`.
- **axes_to_use** – The errors for the given axes are taken into account when calculating χ^2. Either `'y'` or `'xy'`
- **add_constraint_cost** (*bool*) – If `True`, automatically add the cost for kafe2 constraints.
- **add_determinant_cost** (*bool*) – If `True`, automatically increase the cost function value by the logarithm of the determinant of the covariance matrix to reduce bias.

class kafe2.fit.xy.**XYCostFunction_GaussApproximation**(*errors_to_use='covariance'*, *axes_to_use='xy'*, *add_constraint_cost=True*, *add_determinant_cost=True*)

Bases: *kafe2.fit._base.cost.CostFunction_GaussApproximation* (page 153)

Built-in Gaussian approximation of the Poisson negative log-likelihood cost function for *xy* data.

Parameters

- **errors_to_use** (*str*) – Which errors to use when calculating χ^2. This is either *'covariance'*, `'pointwise'`.
- **axes_to_use** – The errors for the given axes are taken into account when calculating χ^2. Either `'y'` or `'xy'`
- **add_constraint_cost** (*bool*) – If `True`, automatically add the cost for kafe2 constraints.
- **add_determinant_cost** (*bool*) – If `True`, automatically increase the cost function value by the logarithm of the determinant of the covariance matrix to reduce bias.

class kafe2.fit.xy.**XYCostFunction_NegLogLikelihood**(*data_point_distribution='poisson'*, *ratio=False*, *axes_to_use='xy'*)

Bases: *kafe2.fit._base.cost.CostFunction_NegLogLikelihood* (page 155)

Base class for built-in negative log-likelihood cost function.

In addition to the measurement data and model predictions, likelihood-fits require a probability distribution describing how the measurements are distributed around the model predictions. This built-in cost function supports two such distributions: the *Poisson* and *Gaussian* (normal) distributions.

In general, a negative log-likelihood cost function is defined as the double negative logarithm of the product of the individual likelihoods of the data points.

The likelihood ratio is defined as ratio of the likelihood function for each individual observation, divided by the so-called *marginal likelihood*.

Parameters

- **data_point_distribution** (*str*) – Which type of statistics to use for modelling the distribution of individual data points. Either `'poisson'` or `'gaussian'`.
- **ratio** (*bool*) – If `True`, divide the likelihood by the marginal likelihood.

class kafe2.fit.xy.**XYFit**(*xy_data*, *model_function=<function linear_model>*, *cost_function='chi2'*, *minimizer=None*, *minimizer_kwargs=None*, *dynamic_error_algorithm='nonlinear'*)

Bases: *kafe2.fit._base.fit.FitBase* (page 160)

Construct a fit of a model to *xy* data.

Parameters

- **xy_data** (XYContainer (page 114) *or Sequence*) – A *XYContainer* (page 114) or a raw 2D array of shape `(2, N)` containing the measurement data.
- **model_function** (*Callable*) – The model function as a native Python function where the first argument denotes the independent *x* variable. Alternatively an already defined `XYModelFunction` object. Defaults to a straight line.
- **cost_function** (*str or Callable*) – The cost function this fit uses to find the best parameters.
- **minimizer** (*str or None*) – The minimizer to use for fitting. Either `None`, `"iminuit"`, `"tminuit"`, or `"scipy"`.
- **minimizer_kwargs** (*dict*) – Dictionary with kwargs for the minimizer.

CONTAINER_TYPE

alias of *kafe2.fit.xy.container.XYContainer* (page 114)

MODEL_TYPE

alias of *kafe2.fit.xy.model.XYParametricModel* (page 129)

MODEL_FUNCTION_TYPE

alias of *kafe2.fit._base.model.ModelFunctionBase* (page 170)

PLOT_ADAPTER_TYPE

alias of *kafe2.fit.xy.plot.XYPlotAdapter* (page 130)

RESERVED_NODE_NAMES = {'cost', 'total_cor_mat', 'total_cor_mat_inversex_data_cov_mat', 'total_cov_mat', 'total_error', 'x_cor_mat', 'x_cov_mat', 'x_cov_mat_inverse', 'x_error', 'y_data', 'y_data_cor_mat', 'y_data_cov_mat', 'y_data_cov_mat_inverse', 'y_data_error', 'y_model', 'y_model_cor_mat', 'y_model_cov_mat', 'y_model_cov_mat_inverse', 'y_model_error'}

X_ERROR_ALGORITHMS = ('iterative linear', 'nonlinear')

property has_x_errors

`True` if at least one *x* uncertainty source has been defined.

Return type bool

property has_y_errors

`True` if at least one *y* uncertainty source has been defined

Return type bool

property x_data

1D array containing the measurement *x* values.

Return type numpy.ndarray[float]

property x_model

***.x_data* for an** *XYFit* (page 119).

Return type numpy.ndarray[float]

Type 1D array containing the model *x* values. The same as

Type py;obj

property y_data

1D array containing the measurement *y* values.

Return type numpy.ndarray[float]

property model

2D array of shape `(2, N)` containing the *x* and *y* model values

Return type numpy.ndarray

property x_data_error

1D array containing the pointwise *x* data uncertainties

Return type numpy.ndarray[float]

property y_data_error

1D array containing the pointwise *y* data uncertainties

Return type numpy.ndarray[float]

property data_error

1D array containing the pointwise *xy* uncertainties projected onto the *y* axis.

Return type numpy.ndarray[float]

property x_data_cov_mat

2D array of shape `(N, N)` containing the data *x* covariance matrix.

Return type numpy.ndarray

property y_data_cov_mat

2D array of shape `(N, N)` containing the data *y* covariance matrix.

Return type numpy.ndarray

property data_cov_mat

2D array of shape `(N, N)` containing the data *xy* covariance matrix (projected onto the *y* axis).

Return type numpy.ndarray

property x_data_cov_mat_inverse

2D array of shape `(N, N)` containing the inverse of the data *x* covariance matrix or `None` if singular.

Return type numpy.ndarray or None

property y_data_cov_mat_inverse

2D array of shape `(N, N)` containing the inverse of the data *y* covariance matrix or `None` if singular.

Return type numpy.ndarray or None

property data_cov_mat_inverse

2D array of shape `(N, N)` containing the inverse of the data *xy* covariance matrix projected onto the *y* axis. `None` if singular.

Return type numpy.ndarray or None

property x_data_cor_mat

2D array of shape `(N, N)` containing the data *x* correlation matrix.

Return type numpy.ndarray

property y_data_cor_mat

2D array of shape `(N, N)` containing the data *y* correlation matrix.

Return type numpy.ndarray

property data_cor_mat

2D array of shape `(N, N)` containing the data *xy* correlation matrix projected onto the *y* axis.

Return type numpy.ndarray

property y_model

1D array of *y* model predictions for the data points.

Return type numpy.ndarray[float]

property x_model_error

1D array of pointwise model *x* uncertainties.

Return type numpy.ndarray[float]

property y_model_error

1D array of pointwise model *y* uncertainties.

Return type numpy.ndarray[float]

property model_error

1D array of pointwise model *xy* uncertainties projected onto the *y* axis.

Return type numpy.ndarray[float]

property x_model_cov_mat

2D array of shape `(N, N)` containing the model *x* covariance matrix.

Return type numpy.ndarray

property y_model_cov_mat

2D array of shape `(N, N)` containing the model *y* covariance matrix.

Return type numpy.ndarray

property model_cov_mat

2D array of shape `(N, N)` containing the model *xy* covariance matrix projected onto the *y* axis.

Return type numpy.ndarray

property x_model_cov_mat_inverse

2D array of shape `(N, N)` containing the inverse of the model *x* covariance matrix or `None` if singular.

Return type numpy.ndarray or None

property y_model_cov_mat_inverse

2D array of shape `(N, N)` containing the inverse of the model *y* covariance matrix or `None` if singular.

Return type numpy.ndarray

property model_cov_mat_inverse

2D array of shape `(N, N)` containing the inverse of the model *xy* covariance matrix projected onto the *y* axis. `None`` if singular.

Return type numpy.ndarray

property x_model_cor_mat

2D array of shape `(N, N)` containing the model *x* correlation matrix.

Return type numpy.ndarray

property y_model_cor_mat

2D array of shape `(N, N)` containing the model *y* correlation matrix.

Return type numpy.ndarray

property model_cor_mat

2D array of shape `(N, N)` containing the model *xy* correlation matrix projected onto the *y* axis.

Return type numpy.ndarray

property x_total_error

1D array of total pointwise *x* uncertainties.

Return type numpy.ndarray[float]

property y_total_error

1D array of total pointwise *y* uncertainties

Return type numpy.ndarray[float]

property total_error

1D array of the total pointwise *xy* uncertainties projected onto the *y* axis.

Return type numpy.ndarray[float]

property x_total_cov_mat

2D array of shape `(N, N)` containing the total *x* covariance matrix.

Return type numpy.ndarray

property y_total_cov_mat

2D array of shape `(N, N)` containing the total *y* covariance matrix.

Return type numpy.ndarray

property total_cov_mat

2D array of shape `(N, N)` containing the total *xy* covariance matrix projected onto the *y* axis.

Return type numpy.ndarray

property x_total_cov_mat_inverse

2D array of shape `(N, N)` containing inverse of the total *x* covariance matrix. `None` if singular.

Return type numpy.ndarray

property y_total_cov_mat_inverse

2D array of shape `(N, N)` containing inverse of the total *y* covariance matrix. `None` if singular.

Return type numpy.ndarray

property total_cov_mat_inverse

2D array of shape `(N, N)` containing theinverse of the total *xy* covariance matrix projected onto the *y* axis. `None` if singular.

Return type numpy.ndarray

property x_total_cor_mat

2D array of shape `(N, N)` containing the total *x* correlation matrix.

Return type numpy.ndarray

property y_total_cor_mat

2D array of shape `(N, N)` containing the total *y* correlation matrix.

Return type numpy.ndarray

property x_range

Minimum and maximum values of the *x* measurement data.

Return type tuple[float, float]

property y_range

Minimum and maximum values of the *y* measurement data.

Return type tuple[float, float]

add_error(*axis*, *err_val*, *name=None*, *correlation=0*, *relative=False*, *reference='data'*)

Add an uncertainty source for an axis to the data container.

Parameters

- **axis** (*str or int*) – `'x'`/`0` or `'y'`/`1`
- **err_val** (*float or Sequence[float]*) – Pointwise uncertainties or a single uncertainty for all data points.
- **name** (*str or None*) – Unique name for this uncertainty source. If `None`, the name of the error source will be set to a random alphanumeric string.
- **correlation** (*float*) – Correlation coefficient between any two distinct data points.
- **relative** (*bool*) – If `True`, **err_val** will be interpreted as a *relative* uncertainty.
- **reference** (*str*) – Which reference values to use when calculating absolute errors from relative errors. Either `'data'` or `'model'`.

Returns An error id uniquely identifying the created error source.

Return type str

add_matrix_error(*axis*, *err_matrix*, *matrix_type*, *name=None*, *err_val=None*, *relative=False*, *reference='data'*)

Add a matrix uncertainty source for an axis to the data container.

Parameters

- **axis** (*str or int*) – `'x'`/`0` or `'y'`/`1`
- **err_matrix** (*numpy.ndarray*) – 2D array of shape `(size, size)` containing the covariance or correlation matrix
- **matrix_type** (*str*) – One of `'covariance'`/`'cov'` or `'correlation'`/`'cor'`.
- **name** (*str or None*) – Unique name for this uncertainty source. If `None`, the name of the error source will be set to a random alphanumeric string.

- **err_val** (*Sequence[float]*) – The pointwise uncertainties. This is mandatory if only a correlation matrix is given.
- **relative** (*bool*) – If `True`, the covariance matrix and/or **err_val** will be interpreted as a *relative* uncertainty.
- **reference** (*str*) – Which reference values to use when calculating absolute errors from relative errors. Either `'data'` or `'model'`.

Returns An error id uniquely identifying the created error source.

Return type str

eval_model_function(*x=None*, *model_parameters=None*)

Evaluate the model function.

Parameters

- **x** (*numpy.ndarray[float]*) – 1D array containing the values of *x* at which to evaluate the model function. If `None`, the data *x* values *x_data* (page 120) are used.
- **model_parameters** (*Collection[float]*) – The model parameter values. If `None`, the current values `parameter_values` are used.

Returns Model function values at the given *x*-values.

Return type numpy.ndarray[float]

eval_model_function_derivative_by_parameters(*x=None*, *model_parameters=None*, *par_dx=None*)

Evaluate the derivative of the model function with respect to the model parameters.

Parameters

- **x** (*numpy.ndarray[float]*) – 1D array containing the *x* values at which to evaluate the model function. If `None`, the data *x* values *x_data* (page 120) are used.
- **model_parameters** (*Collection[float]*) – 1D array containing the model parameter values. If `None`, the current values `parameter_values` are used.
- **par_dx** (*Collection[float]*) – 1D array with length `pars` containing the numeric differentiation step size for each parameter. If `None` and a fit has been performed, 1% of the parameter uncertainties is used.

Returns 2D array of shape `(par, N)` containing the model function derivatives for each parameter at the given *x* values.

Return type numpy.ndarray[numpy.ndarray[float]]

error_band(*x=None*)

Calculate the symmetric model uncertainty at every given point *x*. This is only possible after a fit has been performed with the *do_fit* (page 166) method.

Parameters **x** (*numpy.ndarray[float]*) – 1D array containing the values of x at which to calculate the model uncertainty.

Returns 1D array containing the model uncertainties at the given x values.

Return type numpy.ndarray[float]

class kafe2.fit.xy.**XYFitEnsemble**(*n_experiments*, *x_support*, *model_function*, *model_parameters*, *cost_function=<kafe2.fit.xy.cost.XYCostFunction_Chi2 object>*, *requested_results=None*)

Bases: *kafe2.fit._base.ensemble.FitEnsembleBase* (page 168)

Object for generating ensembles of fits to *xy* pseudo-data generated according to the specified uncertainty model.

After constructing an `XYFitEnsemble` object, an error model should be added to it. This is done as for `XYFit` objects by using the `add_error` or `add_matrix_error` methods.

Once an uncertainty model is provided, the fit ensemble can be generated by using the `run` method. This method starts by generating a pseudo-dataset in such a way that the empirical distribution of the data corresponds to the specified uncertainty model. It then fits the model to the pseudo-data and extracts information from the fit, such as the resulting parameter values or the value of the cost function at the minimum. This is repeated a large number of times in order to evaluate the whole ensemble in a statistically meaningful way.

The ensemble result can be visualized by using the `plot_results` method.

Construct an `XYFitEnsemble` object.

Parameters

- **n_experiments** (*int*) – Number of pseudo experiments to perform.
- **x_support** (*Sequence[float]*) – x values to use as support for calculating the "true" model ("true" x).
- **model_function** (*Callable*) – The model function. Either a `XYModelFunction` object or an unwrapped native Python function.
- **model_parameters** (*Sequence[float]*) – Parameters of the "true" model
- **cost_function** (*Callable*) – The cost function used for the fits. Either a `CostFunctionBase`-derived object or an unwrapped native Python function.
- **requested_results** (*Sequence[str] or None.*) – List of result variables to collect for each toy fit. If `None` it will default to `('y_pulls', 'parameter_pulls', 'cost')`.

FIT_TYPE

alias of *kafe2.fit.xy.fit.XYFit* (page 119)

AVAILABLE_STATISTICS = {'cor_mat': <property object>, 'cov_mat': <property object>, 'kurtosis': <property object>, 'mean': <property object>, 'mean_error': <property object>, 'skew': <property object>, 'std': <property object>}

property n_exp

The number of pseudo-experiments to perform.

Return type int

property n_par

The number of parameters.

Return type int

property n_dat

The number of data points used for the fit.

Return type int

property n_df

The number of degrees of freedom for the fit

Return type int

add_error(*axis*, *err_val*, *name=None*, *correlation=0*, *relative=False*, *reference='data'*)

Add an uncertainty source for an axis to the data container.

Parameters

- **axis** (*str or int*) – `'x'`/`0` or `'y'`/`1`
- **err_val** (*float or Sequence[float]*) – Pointwise uncertainties or a single uncertainty for all data points.
- **name** (*str or None*) – Unique name for this uncertainty source. If `None`, the name of the error source will be set to a random alphanumeric string.
- **correlation** (*float*) – Correlation coefficient between any two distinct data points.
- **relative** (*bool*) – If `True`, **err_val** will be interpreted as a *relative* uncertainty.
- **reference** (*str*) – Which reference values to use when calculating absolute errors from relative errors. Either `'data'` or `'model'`.

Returns An error id uniquely identifying the created error source.

Return type str

add_matrix_error(*axis*, *err_matrix*, *matrix_type*, *name=None*, *err_val=None*, *relative=False*, *reference='data'*)

Add a matrix uncertainty source for an axis to the data container.

Parameters

- **axis** (*str or int*) – `'x'`/`0` or `'y'`/`1`

- **err_matrix** (*numpy.ndarray*) – 2D array of shape `(size, size)` containing the covariance or correlation matrix
- **matrix_type** (*str*) – One of `'covariance'`/`'cov'` or `'correlation'`/`'cor'`.
- **name** (*str or None*) – Unique name for this uncertainty source. If `None`, the name of the error source will be set to a random alphanumeric string.
- **err_val** (*Sequence[float]*) – The pointwise uncertainties. This is mandatory if only a correlation matrix is given.
- **relative** (*bool*) – If `True`, the covariance matrix and/or **err_val** will be interpreted as a *relative* uncertainty.
- **reference** (*str*) – Which reference values to use when calculating absolute errors from relative errors. Either `'data'` or `'model'`.

Returns An error id uniquely identifying the created error source.

Return type str

run()

Perform the pseudo-experiments. Retrieve and store the requested fit result variables.

get_results(**results*)

Return a dictionary containing the ensembles of result variables.

Parameters **results** (*Iterable[str]*) – Names of result variables to retrieve. Calling without arguments retrieves *all* collected results.

Return type dict

get_results_statistics(*results='all'*, *statistics='all'*)

Return a dictionary containing statistics (e.g. mean) of the result variables.

Parameters

- **results** (*Iterable[str] or str*) – Names of retrieved fit variable for which to return statistics. If `'all'`, get statistics for all retrieved variables
- **statistics** (*Iterable[str] or str*) – Names of statistics to retrieve for each result variable. If `'all'`, get all statistics for each retrieved variable

Return type dict

plot_result_distributions(*results='all'*, *show_legend=True*)

Make plots with histograms of the requested fit variable values across all pseudo-experiments.

Parameters

- **results** (*Iterable[str] or str*) – Names of retrieved fit variable for which to generate plots. If `'all'`, plots for all retrieved variables will be made.
- **show_legend** (*bool*) – If a legend is shown on each figure.

plot_result_scatter(*results='all'*, *show_legend=True*)

Make scatter plots of the requested fit variable values across all pseudo-experiments.

Parameters

- **results** (*Iterable[str] or str*) – Names of retrieved fit variable for which to generate plots. If `'all'`, plots for all retrieved variables will be made.
- **show_legend** (*bool*) – If a legend is shown on each figure.

AVAILABLE_RESULTS = {'cost': <property object>, 'parameter_pulls': <property object>, 'x_data': <property object>, 'y_data': <property object>, 'y_model': <property object>, 'y_pulls': <property object>}

class kafe2.fit.xy.**XYParametricModel**(*x_data*, *model_func=<function linear_model>*, *model_parameters=(1.0, 1.0)*)

Bases: *kafe2.fit._base.model.ParametricModelBaseMixin* (page 174), *kafe2.fit.xy.container.XYContainer* (page 114)

Construct an *XYParametricModel* (page 129) object:

Parameters

- **x_data** (*Collection[float]*) – 1D array containing the *x* values supporting the model
- **model_func** (*Callable*) – Python function handle of the model function.
- **model_parameters** (*Collection[float]*) – 1D array containing the parameter values with which the model function should be initialized.

property data

2D array with shape `(2, N)` containing the model predictions.

Return type numpy.ndarray[numpy.ndarray[float]]

property x

1D array containing the *x* support values.

Return type numpy.ndarray[float]

property y

1D array containing the *y* values calculated from the *x* support values and the current parameters.

Return type numpy.ndarray[float]

eval_model_function(*x=None*, *model_parameters=None*)

Evaluate the model function.

Parameters

- **x** (*numpy.ndarray[float]*) – 1D array containing the *x* values of the support points. If `None`, the model *x* values are used.

- **model_parameters** (*Collection[float] or None*) – 1D array containing the values of the model parameters. If None, the current values are used.

Returns Values of the model function for the given parameters.

Return type numpy.ndarray[float]

eval_model_function_derivative_by_parameters(*x=None*, *model_parameters=None*, *par_dx=None*)

Evaluate the derivative of the model function with respect to the model parameters.

Parameters

- **x** (*numpy.ndarray[float] or None*) – 1D array with length N containing the *x* values of the support points. If None, the model *x* values are used.
- **model_parameters** (*Collection[float] or None*) – 1D array with length pars containing the values of the model parameters. If None, the current values are used.
- **par_dx** (*Collection[float]*) – 1D array with length pars containing the numeric differentiation step size for each parameter.

Returns 2D array with shape (pars, N) containing the values of the model function derivatives with respect to the parameters.

Return type numpy.ndarray[numpy.ndarray[float]]

eval_model_function_derivative_by_x(*x=None*, *model_parameters=None*, *dx=None*)

Evaluate the derivative of the model function with respect to the independent variable.

Parameters

- **x** (*numpy.ndarray[float] or None*) – 1D array containing the *x* values of the support points. If None, the model *x* values are used.
- **model_parameters** (*Collection[float] or None*) – 1D array containing the values of the model parameters. If None, the current values are used.
- **dx** (*float or Collection[float]*) – Step size for numeric differentiation.

Returns 1D array containing the values of the model function derivative for each parameter.

Return type numpy.ndarray[float]

class kafe2.fit.xy.**XYPlotAdapter**(*xy_fit_object*, *from_container=False*)

Bases: *kafe2.fit._base.plot.PlotAdapterBase* (page 179)

Construct an XYPlotContainer for a *XYFit* (page 119) object:

Parameters

- **xy_fit_object** (*kafe2.XYFit*) – The *XYFit* (page 119) object handled by this plot adapter.

- **from_container** (*bool*) – Whether xy_fit_object was created ad-hoc from just a data container.

PLOT_STYLE_CONFIG_DATA_TYPE = 'xy'

PLOT_SUBPLOT_TYPES = {'data': {'container_valid': True, 'plot_adapter_method': 'plot_data', 'target_axes': 'main'}, 'model': {'hide': True, 'plot_adapter_method': 'plot_model', 'target_axes': 'main'}, 'model_error_band': {'plot_adapter_method': 'plot_model_error_band', 'target_axes': 'main'}, 'model_line': {'plot_adapter_method': 'plot_model_line', 'target_axes': 'main'}, 'ratio': {'plot_adapter_method': 'plot_ratio', 'plot_style_as': 'data', 'target_axes': 'ratio'}, 'ratio_error_band': {'plot_adapter_method': 'plot_ratio_error_band', 'plot_style_as': 'model_error_band', 'target_axes': 'ratio'}, 'residual': {'plot_adapter_method': 'plot_residual', 'plot_style_as': 'data', 'target_axes': 'residual'}, 'residual_error_band': {'plot_adapter_method': 'plot_residual_error_band', 'plot_style_as': 'model_error_band', 'target_axes': 'residual'}}

AVAILABLE_X_SCALES = ('linear', 'log')

property data_x

The *x* coordinates of the data (used by *plot_data* (page 132)).

Return type numpy.ndarray

property data_y

The *y* coordinates of the data (used by *plot_data* (page 132)).

Return type numpy.ndarray

property data_xerr

The magnitude of the data *x* error bars (used by *plot_data* (page 132)).

Return type numpy.ndarray

property data_yerr

The magnitude of the data *y* error bars (used by *plot_data* (page 132)).

Return type numpy.ndarray

property model_x

The *x* coordinates of the model (used by *plot_model* (page 132)).

Return type numpy.ndarray

property model_y

The *y* coordinates of the model (used by *plot_model* (page 132)).

Return type numpy.ndarray

property model_xerr

The magnitude of the model *x* error bars (used by *plot_model* (page 132)).

Return type numpy.ndarray

property model_yerr

The magnitude of the model *y* error bars (used by *plot_model* (page 132)).

Return type numpy.ndarray

property x_scale

The *x* axis scale. Available scales are given in AVAILABLE_X_SCALES

Return type str

property model_line_x

x support values for model function. Adapts spacing to *x_scale* (page 132).

Return type numpy.ndarray[float]

property model_line_y

y values of the model function at the support points *model_line_x* (page 132).

Return type numpy.ndarray[float]

property y_error_band

1D array representing the uncertainty band around the model function at the support points *model_line_x* (page 132).

Return type numpy.ndarray[float]

plot_data(*target_axes*, *error_contributions=('data',)*, ***kwargs*)

Plot the measurement data to a specified matplotlib.axes.Axes object.

Parameters

- **target_axes** (*matplotlib.axes.Axes*) – The matplotlib axes used for plotting.
- **error_contributions** (*str or Tuple[str]*) – Which error contributions to include when plotting the data. Can either be data, 'model' or both.
- **kwargs** (*dict*) – Keyword arguments accepted by matplotlib.pyplot.errorbar.

Returns plot handle(s)

plot_model(*target_axes*, *error_contributions=('model',)*, ***kwargs*)

Plot the model data to a specified matplotlib.axes.Axes object.

Parameters

- **target_axes** (*matplotlib.axes.Axes*) – The matplotlib axes used for plotting.

- **error_contributions** (str or Tuple[str] Can either be `data`, `'model'` or both.) – Which error contributions to include when plotting the model.
- **kwargs** (*dict*) – Keyword arguments accepted by `matplotlib.pyplot.errorbar`.

Returns plot handle(s)

plot_model_line(*target_axes*, ***kwargs*)

Plot the model function to a specified `matplotlib.axes.Axes` object.

Parameters

- **target_axes** (*matplotlib.axes.Axes*) – The `matplotlib` axes used for plotting.
- **kwargs** (*dict*) – Keyword arguments accepted by `matplotlib.pyplot.plot`.

Returns plot handle(s)

plot_model_error_band(*target_axes*, ***kwargs*)

Plot an error band around the model model function.

Parameters

- **target_axes** (*matplotlib.axes.Axes*) – The `matplotlib` axes used for plotting.
- **kwargs** (*dict*) – Keyword arguments accepted by `matplotlib.pyplot.fill_between`.

Returns plot handle(s)

plot_ratio_error_band(*target_axes*, ***kwargs*)

Plot model error band around the data/model ratio to specified `matplotlib.axes.Axes` object.

Parameters

- **target_axes** (*matplotlib.axes.Axes*) – The `matplotlib` axes used for plotting.
- **kwargs** (*dict*) – Keyword arguments accepted by `matplotlib.pyplot.fill_between`.

Returns plot handle(s)

plot_residual_error_band(*target_axes*, ***kwargs*)

Plot model error band around the data/model ratio to specified `matplotlib.axes.Axes` object.

Parameters

- **target_axes** (*matplotlib.axes.Axes*) – The `matplotlib` axes used for plotting.

- **kwargs** (*dict*) – Keyword arguments accepted by `matplotlib.pyplot.fill_between`.

Returns plot handle(s)

update_plot_kwargs(*plot_type*, *plot_kwargs*)

Update the value of keyword arguments *plot_kwargs* to be passed to the plot method for for *plot_type*.

If a keyword argument should be removed, the value of the keyword in *plot_kwargs* can be set to the special value `'__del__'`. To indicate that the default value should be used, the special value `'__default__'` can be set as a value.

Parameters

- **plot_type** (*str*) – key identifying a registered plot type for this `PlotAdapter`
- **plot_kwargs** (*dict*) – dictionary containing keywords arguments to override

7.3.8 Tools for Fitting Series of Indexed Measurements: `indexed`

This submodule provides the necessary objects for parameter estimation using data consisting of an indexed series of measurements. This can be useful for calculating weighted mean values or template fits.

synopsis This submodule provides the necessary objects for parameter estimation using data consisting of an indexed series of measurements.

class kafe2.fit.indexed.**IndexedContainer**(*data*, *dtype=<class 'float'>*)

Bases: *kafe2.fit._base.container.DataContainerBase* (page 156)

This object is a specialized data container for series of indexed measurements.

Construct a container for indexed data:

Parameters

- **data** (*iterable of type <dtype>*) – a one-dimensional array of measurements
- **dtype** (*type*) – data type of the measurements

property size

number of data points

property data

container data (one-dimensional `numpy.ndarray`)

property err

absolute total data uncertainties (one-dimensional `numpy.ndarray`)

property cov_mat

absolute data covariance matrix (`numpy.matrix`)

property cov_mat_inverse

inverse of absolute data covariance matrix (`numpy.matrix`), or `None` if singular

property cor_mat

absolute data correlation matrix (`numpy.matrix`)

property data_range

the minimum and maximum value of the data

Type return

add_error(*err_val*, *name=None*, *correlation=0*, *relative=False*)

Add an uncertainty source to the data container. Returns an error id which uniquely identifies the created error source.

Parameters

- **err_val** (*float or iterable of float*) – pointwise uncertainty/uncertainties for all data points
- **name** (str or `None`) – unique name for this uncertainty source. If `None`, the name of the error source will be set to a random alphanumeric string.
- **correlation** (*float*) – correlation coefficient between any two distinct data points
- **relative** (*bool*) – if `True`, **err_val** will be interpreted as a *relative* uncertainty

Returns error name

Return type str

add_matrix_error(*err_matrix*, *matrix_type*, *name=None*, *err_val=None*, *relative=False*)

Add a matrix uncertainty source to the data container. Returns an error id which uniquely identifies the created error source.

Parameters

- **err_matrix** – covariance or correlation matrix
- **matrix_type** (*str*) – one of `'covariance'`/`'cov'` or `'correlation'`/`'cor'`
- **name** (str or `None`) – unique name for this uncertainty source. If `None`, the name of the error source will be set to a random alphanumeric string.
- **err_val** (*iterable of float*) – the pointwise uncertainties (mandatory if only a correlation matrix is given)
- **relative** (*bool*) – if `True`, the covariance matrix and/or **err_val** will be interpreted as a *relative* uncertainty

Returns error name

Return type str

class kafe2.fit.indexed.**IndexedCostFunction**(*cost_function*, *arg_names=None*, *add_constraint_cost=True*, *add_determinant_cost=False*)

Bases: *kafe2.fit._base.cost.CostFunction* (page 149)

Construct `CostFunction` object (a wrapper for a native Python function):

Parameters

- **cost_function** (*Callable*) – function handle
- **arg_names** (*Iterable[str]*) – the names to use for the cost function arguments. If None, detect from function signature.
- **add_constraint_cost** (*bool*) – If `True`, automatically add the cost for kafe2 constraints.
- **add_determinant_cost** (*bool*) – If `True`, automatically increase the cost function value by the logarithm of the determinant of the covariance matrix to reduce bias.

class kafe2.fit.indexed.**IndexedCostFunction_Chi2**(*errors_to_use='covariance'*, *fallback_on_singular=True*, *add_constraint_cost=True*, *add_determinant_cost=True*)

Bases: *kafe2.fit._base.cost.CostFunction_Chi2* (page 152)

Base class for built-in least-squares cost function.

Parameters

- **errors_to_use** (*str or None*) – Which errors to use when calculating χ^2. Either `'covariance'`, `'pointwise'` or `None`.
- **fallback_on_singular** (*bool*) – If `True` and the covariance matrix is singular (or the errors are zero), calculate χ^2 as with `errors_to_use=None`
- **add_constraint_cost** (*bool*) – If `True`, automatically add the cost for kafe2 constraints.
- **add_determinant_cost** (*bool*) – If `True`, automatically increase the cost function value by the logarithm of the determinant of the covariance matrix to reduce bias.

class kafe2.fit.indexed.**IndexedCostFunction_GaussApproximation**(*errors_to_use='covariance'*, *add_constraint_cost=True*, *add_determinant_cost=True*)

Bases: *kafe2.fit._base.cost.CostFunction_GaussApproximation* (page 153)

Base class for built-in Gaussian approximation of the Poisson negative log-likelihood cost function.

Parameters

- **errors_to_use** (*str*) – Which errors to use when calculating χ^2. Either `'covariance'`, `'pointwise'`.

- **add_constraint_cost** (*bool*) – If `True`, automatically add the cost for kafe2 constraints.
- **add_determinant_cost** (*bool*) – If `True`, automatically increase the cost function value by the logarithm of the determinant of the covariance matrix to reduce bias.

class kafe2.fit.indexed.**IndexedCostFunction_NegLogLikelihood**(*data_point_distribution='poisson'*, *ratio=False*)

Bases: *kafe2.fit._base.cost.CostFunction_NegLogLikelihood* (page 155)

Base class for built-in negative log-likelihood cost function.

In addition to the measurement data and model predictions, likelihood-fits require a probability distribution describing how the measurements are distributed around the model predictions. This built-in cost function supports two such distributions: the *Poisson* and *Gaussian* (normal) distributions.

In general, a negative log-likelihood cost function is defined as the double negative logarithm of the product of the individual likelihoods of the data points.

The likelihood ratio is defined as ratio of the likelihood function for each individual observation, divided by the so-called *marginal likelihood*.

Parameters

- **data_point_distribution** (*str*) – Which type of statistics to use for modelling the distribution of individual data points. Either `'poisson'` or `'gaussian'`.
- **ratio** (*bool*) – If `True`, divide the likelihood by the marginal likelihood.

class kafe2.fit.indexed.**IndexedFit**(*data*, *model_function*, *cost_function='chi2'*, *minimizer=None*, *minimizer_kwargs=None*, *dynamic_error_algorithm='nonlinear'*)

Bases: *kafe2.fit._base.fit.FitBase* (page 160)

Construct a fit of a model to a series of indexed measurements.

Parameters

- **data** (*iterable of float*) – the measurement values
- **model_function** (*IndexedModelFunction* (page 138) or unwrapped native Python function) – the model function
- **cost_function** (`CostFunctionBase`-derived or unwrapped native Python function) – the cost function
- **minimizer** (*None, "iminuit", "tminuit", or "scipy".*) – the minimizer to use for fitting.
- **minimizer_kwargs** (*dict*) – dictionary with kwargs for the minimizer.

CONTAINER_TYPE

alias of *kafe2.fit.indexed.container.IndexedContainer* (page 134)

MODEL_TYPE

alias of *kafe2.fit.indexed.model.IndexedParametricModel* (page 139)

MODEL_FUNCTION_TYPE

alias of *kafe2.fit.indexed.model.IndexedModelFunction* (page 138)

PLOT_ADAPTER_TYPE

alias of *kafe2.fit.indexed.plot.IndexedPlotAdapter* (page 140)

RESERVED_NODE_NAMES = {'cost', 'data', 'data_cor_mat', 'data_cov_mat', 'data_error', 'model', 'model_cor_mat', 'model_cov_mat', 'model_error', 'total_cor_mat', 'total_cov_mat', 'total_error'}

property model

array of model predictions for the data points

class kafe2.fit.indexed.**IndexedModelFunction**(*model_function*)

Bases: *kafe2.fit._base.model.ModelFunctionBase* (page 170)

Construct *IndexedModelFunction* (page 138) object (a wrapper for a native Python function):

Parameters **model_function** – function handle

FORMATTER_TYPE

alias of *kafe2.fit.indexed.format.IndexedModelFunctionFormatter* (page 138)

class kafe2.fit.indexed.**IndexedModelFunctionFormatter**(*name*, *latex_name=None*, *index_name='i'*, *latex_index_name='i'*, *arg_formatters=None*, *expression_string=None*, *latex_expression_string=None*)

Bases: *kafe2.fit._base.format.FunctionFormatter* (page 168)

Construct a Formatter for a model function for *indexed* data:

Parameters

- **name** – a plain-text-formatted string indicating the function name
- **latex_name** – a LaTeX-formatted string indicating the function name
- **index_name** – a plain-text-formatted string representing the index
- **latex_index_name** – a LaTeX-formatted string representing the index
- **arg_formatters** – list of ParameterFormatter-derived objects, formatters for function arguments
- **expression_string** – a plain-text-formatted string indicating the function expression

- **latex_expression_string** – a LaTeX-formatted string indicating the function expression

property index_name

The parameter name of the index.

Return type str

property latex_index_name

The LaTeX parameter name of the index.

Return type str

get_formatted(*with_par_values=False*, *n_significant_digits=2*, *format_as_latex=False*, *with_expression=False*)

Get a formatted string representing this model function.

Parameters

- **with_par_values** – if `True`, output will include the value of each function parameter (e.g. `f_i(a=1, b=2, ...)`)
- **n_significant_digits** (*int*) – number of significant digits for rounding
- **format_as_latex** – if `True`, the returned string will be formatted using LaTeX syntax
- **with_expression** – if `True`, the returned string will include the expression assigned to the function

Returns string

class kafe2.fit.indexed.**IndexedParametricModel**(*model_func*, *model_parameters*, *shape_like=None*)

Bases: *kafe2.fit._base.model.ParametricModelBaseMixin* (page 174), *kafe2.fit.indexed.container.IndexedContainer* (page 134)

Construct an *IndexedParametricModel* (page 139) object:

Parameters

- **model_func** – handle of Python function (the model function)
- **model_parameters** – iterable of parameter values with which the model function should be initialized
- **shape_like** – array with the same shape as the model

MODEL_FUNCTION_TYPE

alias of *kafe2.fit.indexed.model.IndexedModelFunction* (page 138)

property data

model predictions (one-dimensional `numpy.ndarray`)

property data_range

tuple containing the minimum and maximum of all model predictions

eval_model_function(*model_parameters=None*)

Evaluate the model function.

Parameters **model_parameters** (list or None) – values of the model parameters (if None, the current values are used)

Returns value(s) of the model function for the given parameters

Return type numpy.ndarray

eval_model_function_derivative_by_parameters(*model_parameters=None*, *par_dx=None*)

Evaluate the derivative of the model function with respect to the model parameters.

Parameters

- **model_parameters** (list or None) – values of the model parameters (if None, the current values are used)
- **par_dx** (*float*) – step size for numeric differentiation

Returns value(s) of the model function derivative for the given parameters

Return type numpy.ndarray

class kafe2.fit.indexed.**IndexedPlotAdapter**(*indexed_fit_object*, *from_container=False*)

Bases: *kafe2.fit._base.plot.PlotAdapterBase* (page 179)

Construct an IndexedPlotContainer for a *IndexedFit* (page 137) object:

Parameters

- **fit_object** – an *IndexedFit* (page 137) object
- **from_container** (*bool*) – Whether indexed_fit_object was created ad-hoc from just a data container.

PLOT_STYLE_CONFIG_DATA_TYPE = 'indexed'

PLOT_SUBPLOT_TYPES = {'data': {'container_valid': True, 'plot_adapter_method': 'plot_data', 'target_axes': 'main'}, 'model': {'hide': True, 'plot_adapter_method': 'plot_model', 'target_axes': 'main'}, 'ratio': {'plot_adapter_method': 'plot_ratio', 'plot_style_as': 'data', 'target_axes': 'ratio'}, 'residual': {'plot_adapter_method': 'plot_residual', 'plot_style_as': 'data', 'target_axes': 'residual'}}

property data_x

data x values

property data_y

data y values

property data_xerr

None for IndexedPlotContainer

Type x error bars for data

property data_yerr

total uncertainty

Type y error bars for data

property model_x

model prediction x values

property model_y

model prediction y values

property model_xerr

x error bars for model (actually used to represent the horizontal step length)

property model_yerr

`None` for `IndexedPlotContainer`

Type y error bars for model

plot_data(*target_axes*, ***kwargs*)

Plot the measurement data to a specified `matplotlib` `Axes` object.

Parameters

- **target_axes** – `matplotlib` `Axes` object
- **kwargs** – keyword arguments accepted by the `matplotlib` methods `errorbar` or `plot`

Returns plot handle(s)

plot_model(*target_axes*, ***kwargs*)

Plot the model predictions to a specified matplotlib `Axes` object.

Parameters

- **target_axes** – `matplotlib` `Axes` object
- **kwargs** – keyword arguments accepted by the `step_fill_between` method

Returns plot handle(s)

7.3.9 Tools for Fitting Histograms: `histogram`

This submodule provides the necessary objects for parameter estimation from histograms. Currently a histogram needs to be filled with all individual data points. A function for setting the bin heights is available but not recommended, as saving and loading those to and from a file is not yet supported.

synopsis This submodule provides the necessary objects for parameter estimation from histograms.

class kafe2.fit.histogram.**HistContainer**(*n_bins*, *bin_range*, *bin_edges=None*, *fill_data=None*, *dtype=<class 'int'>*)

Bases: *kafe2.fit.indexed.container.IndexedContainer* (page 134)

This object is a specialized data container for organizing data into *histograms*.

A histogram is a compact representation of a potentially large number of entries which are distributed along a continuum of values. Histograms divide the continuum into intervals ("bins") and count the number of entries per interval.

Construct a histogram:

Parameters

- **n_bins** (*int*) – number of bins
- **bin_range** (*tuple of floats*) – the lower and upper edges of the entire histogram
- **bin_edges** (*list of floats*) – the bin edges (if `None`, each bin will have the same width)
- **fill_data** (*list of floats*) – entries to fill into the histogram
- **dtype** (*type*) – data type of histogram entries

property size

the number of bins (excluding underflow and overflow bins)

property n_entries

the number of entries

property data

the number of entries in each bin

property raw_data

the number of entries in each bin

property low

the lower edge of the histogram

property high

the upper edge of the histogram

property bin_range

a tuple containing the lower and upper edges of the histogram

property overflow

the number of entries in the overflow bin

property underflow

the number of entries in the underflow bin

property n_bins

the number of bins

property bin_edges

a list of the bin edges (including the outermost ones)

property bin_widths

a list of the bin widths

property bin_centers

a list of the (geometrical) bin centers

fill(*entries*)

Fill new entries into the histogram.

Parameters **entries** (*list of floats*) – list of entries

rebin(*new_bin_edges*)

Change the histogram binning.

Parameters **new_bin_edges** (*list of float*) – list of new bin edges in ascending order

set_bins(*bin_heights*, *underflow=0*, *overflow=0*)

Set the bin heights according to a pre-calculated histogram :param bin_heights: Heights of the bins :type bin_heights: list of int :param underflow: Number of entries in the underflow bin :type underflow: int :param overflow: Number of entries in the overflow bin :type overflow: int

class kafe2.fit.histogram.**HistCostFunction**(*cost_function*, *arg_names=None*, *add_constraint_cost=True*, *add_determinant_cost=False*)

Bases: *kafe2.fit._base.cost.CostFunction* (page 149)

Construct `CostFunction` object (a wrapper for a native Python function):

Parameters

- **cost_function** (*Callable*) – function handle
- **arg_names** (*Iterable[str]*) – the names to use for the cost function arguments. If None, detect from function signature.
- **add_constraint_cost** (*bool*) – If `True`, automatically add the cost for kafe2 constraints.
- **add_determinant_cost** (*bool*) – If `True`, automatically increase the cost function value by the logarithm of the determinant of the covariance matrix to reduce bias.

class kafe2.fit.histogram.**HistCostFunction_Chi2**(*errors_to_use='covariance'*, *fallback_on_singular=True*, *add_constraint_cost=True*, *add_determinant_cost=True*)

Bases: *kafe2.fit._base.cost.CostFunction_Chi2* (page 152)

Base class for built-in least-squares cost function.

Parameters

- **errors_to_use** (*str or None*) – Which errors to use when calculating χ^2. Either `'covariance'`, `'pointwise'` or `None`.
- **fallback_on_singular** (*bool*) – If `True` and the covariance matrix is singular (or the errors are zero), calculate χ^2 as with `errors_to_use=None`
- **add_constraint_cost** (*bool*) – If `True`, automatically add the cost for kafe2 constraints.
- **add_determinant_cost** (*bool*) – If `True`, automatically increase the cost function value by the logarithm of the determinant of the covariance matrix to reduce bias.

class kafe2.fit.histogram.**HistCostFunction_GaussApproximation**(*errors_to_use='covariance'*, *add_constraint_cost=True*, *add_determinant_cost=True*)

Bases: *kafe2.fit._base.cost.CostFunction_GaussApproximation* (page 153)

Base class for built-in Gaussian approximation of the Poisson negative log-likelihood cost function.

Parameters

- **errors_to_use** (*str*) – Which errors to use when calculating χ^2. Either `'covariance'`, `'pointwise'`.
- **add_constraint_cost** (*bool*) – If `True`, automatically add the cost for kafe2 constraints.
- **add_determinant_cost** (*bool*) – If `True`, automatically increase the cost function value by the logarithm of the determinant of the covariance matrix to reduce bias.

class kafe2.fit.histogram.**HistCostFunction_NegLogLikelihood**(*data_point_distribution='poisson'*, *ratio=False*)

Bases: *kafe2.fit._base.cost.CostFunction_NegLogLikelihood* (page 155)

Base class for built-in negative log-likelihood cost function.

In addition to the measurement data and model predictions, likelihood-fits require a probability distribution describing how the measurements are distributed around the model predictions. This built-in cost function supports two such distributions: the *Poisson* and *Gaussian* (normal) distributions.

In general, a negative log-likelihood cost function is defined as the double negative logarithm of the product of the individual likelihoods of the data points.

The likelihood ratio is defined as ratio of the likelihood function for each individual observation, divided by the so-called *marginal likelihood*.

Parameters

- **data_point_distribution** (*str*) – Which type of statistics to use for modelling the distribution of individual data points. Either `'poisson'` or `'gaussian'`.

- **ratio** (*bool*) – If True, divide the likelihood by the marginal likelihood.

class kafe2.fit.histogram.**HistFit**(*data*, *model_function=<function normal_distribution>*, *cost_function=<kafe2.fit._base.cost.CostFunction_NegLogLikelihood object>*, *bin_evaluation='simpson'*, *density=True*, *minimizer=None*, *minimizer_kwargs=None*, *dynamic_error_algorithm='nonlinear'*)

Bases: *kafe2.fit._base.fit.FitBase* (page 160)

Construct a fit of a model to a histogram. If bin_evaluation is a Python function or of a numpy.vectorize object it is interpreted as the antiderivative of model_density_function. If bin_evaluation is equal to "rectangle", "midpoint", "trapezoid", or "simpson" the bin heights are evaluated according to the corresponding quadrature formula. If bin_evaluation is equal to "numerical" the bin heights are evaluated by numerically integrating model_density_function.

Parameters

- **data** (HistContainer) – a HistContainer representing histogrammed data
- **model_function** – the model (density) function
- **cost_function** (CostFunctionBase-derived or unwrapped native Python function) – the cost function
- **bin_evaluation** (*str, callable, or numpy.vectorize*) – how the model evaluates bin heights.
- **density** (*bool*) – if True, scale model function to the number of data points.
- **minimizer** (*None, "iminuit", "tminuit", or "scipy".*) – the minimizer to use for fitting.
- **minimizer_kwargs** (*dict*) – dictionary with kwargs for the minimizer.

CONTAINER_TYPE

alias of *kafe2.fit.histogram.container.HistContainer* (page 141)

MODEL_TYPE

alias of *kafe2.fit.histogram.model.HistParametricModel* (page 146)

MODEL_FUNCTION_TYPE

alias of *kafe2.fit.histogram.model.HistModelFunction* (page 146)

PLOT_ADAPTER_TYPE

alias of *kafe2.fit.histogram.plot.HistPlotAdapter* (page 147)

RESERVED_NODE_NAMES = {'cost', 'data', 'data_cor_mat', 'data_cov_mat', 'data_error', 'model', 'model_cor_mat', 'model_cov_mat', 'model_density', 'model_error', 'total_cor_mat', 'total_cov_mat', 'total_error'}

property model

array of model predictions for the data points

property density

eval_model_function_density(*x*, *model_parameters=None*)

Evaluate the model function density.

Parameters

- **x** (*iterable of float*) – values of *x* at which to evaluate the model function density
- **model_parameters** (*iterable of float*) – the model parameter values (if `None`, the current values are used)

Returns model function density values

Return type `numpy.ndarray`

class kafe2.fit.histogram.**HistModelFunction**(*model_function=None*)

Bases: *kafe2.fit._base.model.ModelFunctionBase* (page 170)

Construct `XYModelFunction` object (a wrapper for a native Python function):

Parameters **model_function** – function handle

class kafe2.fit.histogram.**HistParametricModel**(*n_bins, bin_range, model_density_func=<function normal_distribution>, model_parameters=[1.0, 1.0], bin_edges=None, bin_evaluation='simpson', density=True*)

Bases: *kafe2.fit._base.model.ParametricModelBaseMixin* (page 174), *kafe2.fit.histogram.container.HistContainer* (page 141)

Mixin constructor: sets and initialized the model function.

Parameters

- **model_func** – handle of Python function (the model function)
- **model_parameters** – iterable of parameter values with which the model function should be initialized

MODEL_FUNCTION_TYPE

alias of *kafe2.fit.histogram.model.HistModelFunction* (page 146)

property data

model predictions (one-dimensional `numpy.ndarray`)

property bin_evaluation

how the model evaluates bin heights. :rtype str, callable, or numpy.vectorize

Type return

property bin_evaluation_string

string representation of how the model evaluates bin heights. :rtype str

Type return

property density

eval_model_function_density(*x*, *model_parameters=None*)

Evaluate the model function density.

Parameters

- **x** (*list of float*) – *x* values of the support points
- **model_parameters** (list or None) – values of the model parameters (if None, the current values are used)

Returns value(s) of the model function for the given parameters

Return type numpy.ndarray

fill(*entries*)

Fill new entries into the histogram.

Parameters **entries** (*list of floats*) – list of entries

class kafe2.fit.histogram.**HistPlotAdapter**(*hist_fit_object*, *from_container=False*)

Bases: *kafe2.fit._base.plot.PlotAdapterBase* (page 179)

Construct an HistPlotContainer for a *HistFit* (page 145) object:

Parameters

- **fit_object** – an *HistFit* (page 145) object
- **n_plot_points_model_density** – number of plot points to use for plotting the model density
- **from_container** (*bool*) – Whether hist_fit_object was created ad-hoc from just a data container.

PLOT_STYLE_CONFIG_DATA_TYPE = 'histogram'

PLOT_SUBPLOT_TYPES = {'data': {'container_valid': True, 'plot_adapter_method': 'plot_data', 'target_axes': 'main'}, 'model': {'hide': True, 'plot_adapter_method': 'plot_model', 'target_axes': 'main'}, 'model_density': {'plot_adapter_method': 'plot_model_density', 'target_axes': 'main'}, 'ratio': {'plot_adapter_method': 'plot_ratio', 'plot_style_as': 'data', 'target_axes': 'ratio'}, 'residual': {'plot_adapter_method': 'plot_residual', 'plot_style_as': 'data', 'target_axes': 'residual'}}

AVAILABLE_X_SCALES = ('linear', 'log')

property data_x
data x values

property data_y
data y values

property data_xerr
x error bars for data (actually used to represent the bins)

property data_yerr
total uncertainty

Type y error bars for data

property model_x
model prediction x values

property model_y
model prediction y values

property model_xerr
x error bars for model (actually used to represent the bins)

property model_yerr
`None` for `HistPlotContainer`

Type y error bars for model

property model_density_x
x support points for model density plot

property model_density_y
value of model density at the support points

plot_data(*target_axes*, ***kwargs*)
Plot the measurement data to a specified `matplotlib` `Axes` object.

Parameters

- **target_axes** – `matplotlib` `Axes` object
- **kwargs** – keyword arguments accepted by the `matplotlib` method `errorbar`

Returns plot handle(s)

plot_model(*target_axes*, ***kwargs*)
Plot the model predictions to a specified matplotlib `Axes` object.

Parameters

- **target_axes** – `matplotlib` `Axes` object
- **kwargs** – keyword arguments accepted by the `matplotlib` method `bar`

Returns plot handle(s)

plot_model_density(*target_axes*, ***kwargs*)

Plot the model density to a specified `matplotlib Axes` object.

Parameters

- **target_axes** – `matplotlib Axes` object
- **kwargs** – keyword arguments accepted by the `matplotlib` method `plot`

Returns plot handle(s)

7.3.10 Abstract Base Classes: _base

synopsis This submodule contains the abstract base classes for all objects used by the *kafe2.fit* (page 112) module.

class kafe2.fit._base.**CostFunction**(*cost_function*, *arg_names=None*, *add_constraint_cost=True*, *add_determinant_cost=False*)

Bases: `kafe2.fit.io.file.FileIOMixin`, `object`

Base class for cost functions. Built from a Python function with some extra functionality used by Fit objects.

Any Python function returning a `float` can be used as a cost function, although a number of common cost functions are provided as built-ins for all fit types.

In order to be used as a model function, a native Python function must be wrapped by an object whose class derives from this base class. There is a dedicated *CostFunction* (page 149) specialization for each type of fit.

This class provides the basic functionality used by all *CostFunction* (page 149) objects. These use introspection (`inspect`) for determining the parameter structure of the cost function and to ensure the function can be used as a cost function (validation).

Construct *CostFunction* (page 149) object (a wrapper for a native Python function):

Parameters

- **cost_function** (*Callable*) – function handle
- **arg_names** (*Iterable[str]*) – the names to use for the cost function arguments. If None, detect from function signature.
- **add_constraint_cost** (*bool*) – If `True`, automatically add the cost for kafe2 constraints.
- **add_determinant_cost** (*bool*) – If `True`, automatically increase the cost function value by the logarithm of the determinant of the covariance matrix to reduce bias.

property name

The cost function name (a valid Python identifier)

property func

The cost function handle

property arg_names

The names of the cost function arguments.

property formatter

The `Formatter` object for this function

property argument_formatters

The `Formatter` objects for the function arguments

property needs_errors

Whether the cost function needs errors for a meaningful result

property is_chi2

Whether the cost function is a chi2 cost function.

property saturated

Whether the cost function value is calculated from a saturated likelihood.

property add_determinant_cost

Whether the determinant cost is being added automatically to the cost function value.

property kafe2go_identifier

Short string representation (if any) of this cost function when dumping to file.

property pointwise

True if cost function result does not depend on covariances.

property pointwise_version

Optimized version of cost function that uses pointwise errors, can be None.

property errors_valid

goodness_of_fit (**args*)

How well the model agrees with the data.

chi2_probability (*cost_function_value*, *ndf*)

The chi2 probability associated with this cost function, None for non-chi2 cost functions.

Parameters

- **cost_function_value** (*float*) – the associated cost function value.
- **ndf** (*int*) – the associated number of degrees of freedom.

Returns the associated chi2 probability.

Return type float or None

get_uncertainty_gaussian_approximation (*data*)

Get the gaussian approximation of the uncertainty inherent to the cost function, returns 0 by default.

Parameters **data** – the fit data

Returns the approximated gaussian uncertainty given the fit data

is_data_compatible(*data*)

Tests if model data is compatible with cost function

Parameters **data** (*numpy.ndarray*) – the fit data

Returns if the data is compatible, and if not a reason for the incompatibility

Return type (boo, str)

class kafe2.fit._base.**CostFunctionFormatter**(*name*, *name_saturated=None*, *latex_name=None*, *latex_name_saturated=None*, *arg_formatters=None*, *expression_string=None*, *latex_expression_string=None*)

Bases: *kafe2.fit._base.format.FunctionFormatter* (page 168)

A Formatter class for Cost Functions.

Construct a formatter for a model function:

Parameters

- **name** (*str*) – A plain-text-formatted string indicating the function name.
- **latex_name** (*str*) – A LaTeX-formatted string indicating the function name.
- **arg_formatters** (*list[kafe2.fit._base.ParameterFormatter* (page 172)*]*) – List of *ParameterFormatter* (page 172)-derived objects, formatters for function arguments.
- **expression_string** (*str*) – A plain-text-formatted string indicating the function expression.
- **latex_expression_string** (*str*) – A LaTeX-formatted string indicating the function expression.

property name_saturated

A plain-text-formatted string indicating the saturated function name.

Return type str

property latex_name_saturated

A LaTeX-formatted string indicating the saturated function name.

Return type str

get_formatted(*value=None*, *n_degrees_of_freedom=None*, *with_name=True*, *saturated=False*, *with_value_per_ndf=True*, *format_as_latex=False*)

Get a formatted string representing this cost function.

Parameters

- **value** (*float or None*) – Value of the cost function (if not None, the returned string will include this).

- **n_degrees_of_freedom** (*int or None*) – Number of degrees of freedom (if not None, the returned string will include this).
- **with_name** (*bool*) – If True, the returned string will include the cost function name
- **saturated** (*bool*) – If True, the cost function name for the saturated Likelihood will be used (no effect for chi2).
- **with_value_per_ndf** (*bool*) – If True, the returned string will include the value-ndf ratio as a decimal value
- **format_as_latex** (*bool*) – If True, the returned string will be formatted using LaTeX syntax

Return type str

class kafe2.fit._base.**CostFunction_Chi2**(*errors_to_use='covariance'*, *fallback_on_singular=True*, *add_constraint_cost=True*, *add_determinant_cost=True*)

Bases: *kafe2.fit._base.cost.CostFunction* (page 149)

Base class for built-in least-squares cost function.

Parameters

- **errors_to_use** (*str or None*) – Which errors to use when calculating χ^2. Either 'covariance', 'pointwise' or None.
- **fallback_on_singular** (*bool*) – If True and the covariance matrix is singular (or the errors are zero), calculate χ^2 as with errors_to_use=None
- **add_constraint_cost** (*bool*) – If True, automatically add the cost for kafe2 constraints.
- **add_determinant_cost** (*bool*) – If True, automatically increase the cost function value by the logarithm of the determinant of the covariance matrix to reduce bias.

chi2_no_errors(*data*, *model*)

A least-squares cost function calculated from (y) data and model values, without considering uncertainties:

$$C = \chi^2(\mathbf{d}, \mathbf{m}) = \sum_k (d_k - m_k)^2 + C_{\text{con}}(\mathbf{p}).$$

In the above, $\mathbf{d}$ are the measurements, $\mathbf{m}$ are the model predictions, and $C_{\text{con}}(\mathbf{p})$ is the additional cost resulting from any constrained parameters.

Parameters

- **data** – measurement data $\mathbf{d}$
- **model** – model predictions $\mathbf{m}$

Returns cost function value

chi2_covariance (*data*, *model*, *total_cov_mat_cholesky*)

A least-squares cost function calculated from (y) data and model values, considering the covariance matrix of the (y) measurements. The cost function value can be calculated as follows:

$$C = \chi^2(\mathbf{d}, \mathbf{m}) = (\mathbf{d} - \mathbf{m})^\top \mathbf{V}^{-1} (\mathbf{d} - \mathbf{m}) + C_{\text{con}}(\mathbf{p}) + C_{\text{det}}(\mathbf{V}).$$

In the above, $\mathbf{d}$ are the measurements, $\mathbf{m}$ are the model predictions, $\mathbf{V}^{-1}$ is the inverse of the total covariance matrix, $C_{\text{con}}(\mathbf{p})$ is the additional cost resulting from any constrained parameters, and $C_{\text{det}}(\mathbf{V}) = \ln\det(\mathbf{V})$ is the additional cost to compensate for a non-constant covariance matrix.

Parameters

- **data** – measurement data $\mathbf{d}$
- **model** – model predictions $\mathbf{m}$
- **total_cov_mat_cholesky** – Cholesky decomposition of the total covariance matrix $\mathbf{L}$ with $\mathbf{L}^\top\mathbf{L} = \mathbf{V}$

Returns cost function value

chi2_pointwise_errors (*data*, *model*, *total_error*)

A least-squares cost function calculated from (y) data and model values, considering pointwise (uncorrelated) uncertainties for each data point:

$$C = \chi^2(\mathbf{d}, \mathbf{m}, \sigma) = \sum_k \frac{d_k - m_k}{\sigma_k} + C_{\text{con}}(\mathbf{p}) + C_{\text{det}}(\sigma).$$

In the above, $\mathbf{d}$ are the measurements, $\mathbf{m}$ are the model predictions, σ are the pointwise total uncertainties, $C_{\text{con}}(\mathbf{p})$ is the additional cost resulting from any constrained parameters, and $C_{\text{det}}(\sigma) = \ln\prod_k \sigma_k^2$ is the additional cost to compensate for non-constant errors.

Parameters

- **data** – measurement data $\mathbf{d}$
- **model** – model predictions $\mathbf{m}$
- **total_error** – total error vector σ

Returns cost function value

property pointwise

True if cost function result does not depend on covariances.

property pointwise_version

Optimized version of cost function that uses pointwise errors, can be None.

class kafe2.fit._base.**CostFunction_GaussApproximation**(*errors_to_use='covariance'*, *add_constraint_cost=True*, *add_determinant_cost=True*)

Bases: *kafe2.fit._base.cost.CostFunction* (page 149)

Base class for built-in Gaussian approximation of the Poisson negative log-likelihood cost function.

Parameters

- **errors_to_use** (*str*) – Which errors to use when calculating χ^2. Either `'covariance'`, `'pointwise'`.
- **add_constraint_cost** (*bool*) – If `True`, automatically add the cost for kafe2 constraints.
- **add_determinant_cost** (*bool*) – If `True`, automatically increase the cost function value by the logarithm of the determinant of the covariance matrix to reduce bias.

gaussian_approximation_covariance (*data*, *model*, *total_cov_mat*)

A least-squares cost function calculated from (y) data and model values, considering the covariance matrix of the (y) measurements. The cost function value can be calculated as follows:

$$C = \chi^2(\mathbf{d}, \mathbf{m}, \mathbf{V}) = (\mathbf{d} - \mathbf{m})^\top \tilde{\mathbf{V}}^{-1} (\mathbf{d} - \mathbf{m}) + C_{\text{con}}(\mathbf{p}) + C_{\text{det}}(\tilde{\mathbf{V}}); \quad \tilde{\mathbf{V}}_{ij} = \mathbf{V}_{ij} + \delta_{ij}\mathbf{m}_i.$$

In the above, $\mathbf{d}$ are the measurements, $\mathbf{m}$ are the model predictions, $\mathbf{V}^{-1}$ is the inverse of the total covariance matrix, $C_{\text{con}}(\mathbf{p})$ is the additional cost resulting from any constrained parameters, and $C_{\text{det}}(\tilde{\mathbf{V}}) = \ln\det(\tilde{\mathbf{V}})$ is the additional cost to compensate for a non-constant covariance matrix.

Parameters

- **data** – measurement data $\mathbf{d}$
- **model** – model predictions $\mathbf{m}$
- **total_cov_mat** – The total covariance matrix $\mathbf{V}$

Returns cost function value

gaussian_approximation_pointwise_errors (*data*, *model*, *total_error*)

A least-squares cost function calculated from data and model values, considering pointwise (uncorrelated) uncertainties for each data point:

$$C = \chi^2(\mathbf{d}, \mathbf{m}, \sigma) = \sum_k \left(\frac{d_k - m_k}{\sigma_k + \sqrt{m_k}}\right)^2 + C_{\text{con}}(\mathbf{p}) + C_{\text{det}}(\sigma).$$

In the above, $\mathbf{d}$ are the measurements, $\mathbf{m}$ are the model predictions, σ are the pointwise total uncertainties, $C(\mathbf{p})$ is the additional cost resulting from any constrained parameters, and $C_{\text{det}}(\sigma) = \ln\prod_k m_k + \sigma_k^2$ is the additional cost to compensate for non-constant errors.

Parameters

- **data** – measurement data $\mathbf{d}$
- **model** – model predictions $\mathbf{m}$
- **total_error** – total error vector σ

Returns cost function value

property pointwise

True if cost function result does not depend on covariances.

property pointwise_version

Optimized version of cost function that uses pointwise errors, can be None.

goodness_of_fit(**args*)

How well the model agrees with the data.

get_uncertainty_gaussian_approximation(*data*)

Get the gaussian approximation of the uncertainty inherent to the cost function, returns 0 by default.

Parameters **data** – the fit data

Returns the approximated gaussian uncertainty given the fit data

class kafe2.fit._base.**CostFunction_NegLogLikelihood**(*data_point_distribution='poisson'*, *ratio=False*)

Bases: *kafe2.fit._base.cost.CostFunction* (page 149)

Base class for built-in negative log-likelihood cost function.

In addition to the measurement data and model predictions, likelihood-fits require a probability distribution describing how the measurements are distributed around the model predictions. This built-in cost function supports two such distributions: the *Poisson* and *Gaussian* (normal) distributions.

In general, a negative log-likelihood cost function is defined as the double negative logarithm of the product of the individual likelihoods of the data points.

The likelihood ratio is defined as ratio of the likelihood function for each individual observation, divided by the so-called *marginal likelihood*.

Parameters

- **data_point_distribution** (*str*) – Which type of statistics to use for modelling the distribution of individual data points. Either `'poisson'` or `'gaussian'`.
- **ratio** (*bool*) – If `True`, divide the likelihood by the marginal likelihood.

static nll_gaussian(*data*, *model*, *total_error*)

A negative log-likelihood function assuming Gaussian statistics for each measurement.

The cost function is given by:

$$C = -2\ln\mathcal{L}(\mathbf{d},\mathbf{m},\sigma) = -2\ln\prod_j \mathcal{L}_{\text{Gaussian}}(x=d_j,\mu=m_j,\sigma=\sigma_j) + C_{\text{con}}(\mathbf{p}).$$

$$\rightarrow C = -2\ln\prod_j \frac{1}{\sqrt{2{\sigma_j}^2\pi}}\exp\left(-\frac{(d_j-m_j)^2}{{\sigma_j}^2}\right) + C_{\text{con}}(\mathbf{p}).$$

In the above, **d** are the measurements, **m** are the model predictions, σ are the pointwise total uncertainties, and $C_{\text{con}}(\mathbf{p})$ is the additional cost resulting from any constrained parameters.

Parameters

- **data** – measurement data **d**
- **model** – model predictions **m**

- **total_error** – total error vector σ

Returns cost function value

static nll_poisson(*data*, *model*)

A negative log-likelihood function assuming Poisson statistics for each measurement.

The cost function is given by:

$$C = -2 \ln \mathcal{L}(\mathbf{d}, \mathbf{m}) = -2 \ln \prod_j \mathcal{L}_{\text{Poisson}}(k = d_j, \lambda = m_j) + C_{\text{con}}(\mathbf{p}).$$

$$\rightarrow C = -2 \ln \prod_j \frac{{m_j}^{d_j} \exp(-m_j)}{d_j!} + C_{\text{con}}(\mathbf{p}).$$

In the above, **d** are the measurements, **m** are the model predictions, and $C_{\text{con}}(\mathbf{p})$ is the additional cost resulting from any constrained parameters.

Parameters

- **data** – measurement data **d**
- **model** – model predictions **m**

Returns cost function value

static nllr_gaussian(*data*, *model*, *total_error*)

static nllr_poisson(*data*, *model*)

is_data_compatible(*data*)

Tests if model data is compatible with cost function

Parameters **data** (*numpy.ndarray*) – the fit data

Returns if the data is compatible, and if not a reason for the incompatibility

Return type (boo, str)

get_uncertainty_gaussian_approximation(*data*)

Get the gaussian approximation of the uncertainty inherent to the cost function, returns 0 by default.

Parameters **data** – the fit data

Returns the approximated gaussian uncertainty given the fit data

class kafe2.fit._base.**DataContainerBase**

Bases: kafe2.fit.io.file.FileIOMixin

This is a purely abstract class implementing the minimal interface required by all types of data containers.

It stores measurement data and uncertainties.

property label

The label describing the dataset.

Return type str or None

property axis_labels

The axis labels describing the dataset.

Return type tuple[str or None, str or None]

property x_label

The x-axis label.

Return type str or None

property y_label

The y-axis label.

Return type str or None

abstract property size

The size of the data (number of measurement points).

Return type int

abstract property data

A numpy array containing the data values.

Return type numpy.ndarray[float]

abstract property err

A numpy array containing the pointwise data uncertainties.

Return type numpy.ndarray[float]

abstract property cov_mat

A numpy matrix containing the covariance matrix of the data.

Return type numpy.ndarray[numpy.ndarray[float]]

abstract property cov_mat_inverse

obj`None` if not invertible).

Return type numpy.ndarray[numpy.ndarray[float]] or None

Type A numpy matrix containing inverse of the data covariance matrix (or

Type py

property has_errors

`True` if at least one uncertainty source is defined for the data container.

Return type bool

add_error(*err_val*, *name=None*, *correlation=0*, *relative=False*, *reference=None*)

Add an uncertainty source to the data container.

Parameters

- **err_val** (*float or numpy.ndarray[float]*) – Pointwise uncertainty/uncertainties for all data points.

- **name** (*str or None*) – Unique name for this uncertainty source. If None, the name of the error source will be set to a random alphanumeric string.
- **correlation** (*float*) – Correlation coefficient between any two distinct data points.
- **relative** (*bool*) – If True, **err_val** will be interpreted as a *relative* uncertainty.
- **reference** (*Iterable[float] or None*) – The data values to use when computing absolute errors from relative ones (and vice-versa)

Returns An error id which uniquely identifies the created error source.

Return type str

add_matrix_error(*err_matrix*, *matrix_type*, *name=None*, *err_val=None*, *relative=False*, *reference=None*)

Add a matrix uncertainty source to the data container.

Parameters

- **err_matrix** – Covariance or correlation matrix.
- **matrix_type** (*str*) – One of `'covariance'/'cov'` or `'correlation'/'cor'`.
- **name** (*str or None*) – Unique name for this uncertainty source. If :py:obj`None`, the name of the error source will be set to a random alphanumeric string.
- **err_val** (*Iterable[float]*) – The pointwise uncertainties (mandatory if only a correlation matrix is given).
- **relative** (*bool*) – If True, the covariance matrix and/or **err_val** will be interpreted as a *relative* uncertainty.
- **reference** (*Iterable[float] or None*) – the data values to use when computing absolute errors from relative ones (and vice-versa)

Returns An error id which uniquely identifies the created error source.

Return type str

disable_error(*error_name*)

Temporarily disable an uncertainty source so that it doesn't count towards calculating the total uncertainty.

Parameters **error_name** (*str*) – error name

enable_error(*error_name*)

(Re-)Enable an uncertainty source so that it counts towards calculating the total uncertainty.

Parameters **error_name** (*str*) – error name

get_matching_errors(*matching_criteria=None*, *matching_type='equal'*)

Return a list of uncertainty objects fulfilling the specified matching criteria.

Valid keys for matching_criteria:

- `name` (the unique error name)
- `type` (either `simple` or `matrix`)
- `correlated` (bool, only matches simple errors!)
- `relative` (bool)

Note: The error objects contained in the dictionary are not copies, but the original error objects. Modifying them is possible, but not recommended. If you do modify any of them, the changes will not be reflected in the total error calculation until the error cache is cleared. This can be done by calling the private method `_clear_total_error_cache`.

Parameters

- **matching_criteria** (*dict or None*) – Key-value pairs specifying matching criteria. The resulting error array will only contain error objects matching *all* provided criteria. If `None`, all error objects are returned.
- **matching_type** (*str*) – How to perform the matching. If `'equal'`, the value in **matching_criteria** is checked for equality against the stored value. If `'regex'`, the value in **matching_criteria** is interpreted as a regular expression and is matched against the stored value.

Returns Dict mapping error name to `GaussianErrorBase`-derived error objects.

Return type dict[str, kafe2.core.error.GaussianErrorBase]

get_error(*error_name*)

Return the uncertainty object holding the uncertainty.

Note: If you modify this object, the changes will not be reflected in the total error calculation until the error cache is cleared. This can be forced by calling *enable_error* (page 158).

Parameters **error_name** (*str*) – error name

Returns error object

Return type kafe2.core.error.GaussianErrorBase

get_total_error()

Get the error object representing the total uncertainty.

Returns error object representing the total uncertainty

Return type kafe2.core.error.MatrixGaussianError

class kafe2.fit._base.**FitBase**(*data*, *model_function*, *cost_function*, *minimizer=None*, *minimizer_kwargs=None*, *dynamic_error_algorithm='nonlinear'*)

Bases: kafe2.fit.io.file.FileIOMixin, object

This is a purely abstract class implementing the minimal interface required by all types of fitters.

This is a purely abstract class implementing the minimal interface required by all types of fits.

Parameters

- **minimizer** (*str or None*) – Name of the minimizer to use.
- **minimizer_kwargs** (*dict or None*) – Dictionary wit keywords for initializing the minimizer.
- **dynamic_error_algorithm** (*"nonlinear" or "iterative".*) – how to handle errors that depend on the model parameters.

CONTAINER_TYPE

alias of *kafe2.fit._base.container.DataContainerBase* (page 156)

MODEL_TYPE = None

MODEL_FUNCTION_TYPE = None

PLOT_ADAPTER_TYPE = None

RESERVED_NODE_NAMES = {}

property data

array of measurement values

property data_error

array of pointwise data uncertainties

property data_cov_mat

the data covariance matrix

property data_cov_mat_inverse

inverse of the data covariance matrix (or None if singular)

property data_cor_mat

the data correlation matrix

property data_container

The data container used in this fit.

Return type *kafe2.fit._base.DataContainerBase* (page 156)

abstract property model

property model_error

array of pointwise model uncertainties

property model_cov_mat

the model covariance matrix

property model_cov_mat_inverse

inverse of the model covariance matrix (or `None` if singular)

property model_cor_mat

the model correlation matrix

property total_error

array of pointwise total uncertainties

property total_cov_mat

the total covariance matrix

property total_cov_mat_inverse

inverse of the total covariance matrix (or `None` if singular)

property total_cor_mat

the total correlation matrix

property model_function

The wrapped model function as a *ModelFunctionBase* (page 170) or derived object. This object contains the model function as well as formatting information used for this fit.

Return type *kafe2.fit._base.ModelFunctionBase* (page 170)

property model_label

The label of the model used in this fit.

Return type str or None

property parameter_values

The current parameter values.

Return type numpy.ndarray[float]

property parameter_names

The current parameter names.

Return type tuple[str]

property parameter_errors

The current parameter uncertainties. Can be set to control initial step size during minimization.

Return type numpy.ndarray[float]

property parameter_cov_mat

The current parameter covariance matrix.

Return type None or numpy.ndarray[numpy.ndarray[float]]

property parameter_cor_mat

The current parameter correlation matrix.

Return type None or numpy.ndarray[numpy.ndarray[float]]

property asymmetric_parameter_errors

The current asymmetric parameter uncertainties.

Return type numpy.ndarray[numpy.ndarray[float, float]]

property parameter_name_value_dict

A dictionary mapping each parameter name to its current value.

Return type OrderedDict[str, float]

property parameter_constraints

The gaussian constraints given for the fit parameters.

Return type list[kafe2.core.constraint.GaussianSimpleParameterConstraint or kafe2.core.constraint.GaussianMatrixParameterConstraint]

property cost_function_value

The current value of the cost function.

Return type float

property data_size

The size (number of points) of the data container.

Return type int

property has_model_errors

`True` if at least one uncertainty source is defined for the model.

Return type bool

property has_data_errors

`True` if at least one uncertainty source is defined for the data.

Return type bool

property has_errors

`True` if at least one uncertainty source is defined for either the data or the model.

Return type bool

property model_count

The number of model functions contained in the fit, 1 by default.

Return type int

property did_fit

Whether a fit was performed for the given data and model.

Return type bool

property ndf

The degrees of freedom of this fit.

Return type int

property goodness_of_fit

property dynamic_error_algorithm

The algorithm to use for handling errors that depend on the model parameters. :rtype: str

property chi2_probability

The chi2 probability for the current model values.

property errors_valid

set_parameter_values(***param_name_value_dict*)

Set the fit parameters to new values. Valid keyword arguments are the names of the declared fit parameters.

Parameters **param_name_value_dict** – new parameter values

set_all_parameter_values(*param_value_list*)

Set all the fit parameters at the same time.

Parameters **param_value_list** (*Iterable[float]*) – List of parameter values (mind the order).

fix_parameter(*name*, *value=None*)

Fix a parameter so that its value doesn't change when calling *do_fit* (page 166).

Parameters

- **name** (*str*) – The name of the parameter to be fixed
- **value** (*float or None*) – The value to be given to the fixed parameter. If None the current value from *parameter_values* (page 161) will be used.

release_parameter(*name*)

Release a fixed parameter so that its value once again changes when calling *do_fit* (page 166).

Parameters **name** (*str*) – The name of the fixed parameter to be released

limit_parameter(*name*, *lower=None*, *upper=None*)

Limit a parameter to a given range.

Parameters

- **name** (*str*) – The name of the parameter to limit.
- **lower** (*float*) – The minimum parameter value.
- **upper** (*float*) – The maximum parameter value.

unlimit_parameter(*name*)

Unlimit a parameter.

Parameters **name** (*str*) – The name of the parameter to unlimit.

add_matrix_parameter_constraint(*names*, *values*, *matrix*, *matrix_type='cov'*, *uncertainties=None*, *relative=False*)

Advanced class for applying correlated constraints to several parameters of a fit. The order of **names**, **values**, **matrix**, and **uncertainties** must be aligned. In other words the first index must belong to the first value, the first row/column in the matrix, etc.

Let N be the number of parameters to be constrained.

Parameters

- **names** (*Collection[str]*) – The names of the parameters to be constrained. Must be of shape (N,).
- **values** (*Sized[float]*) – The values to which the parameters should be constrained. Must be of shape shape (N,).
- **matrix** (*Iterable[float]*) – The matrix that defines the correlation between the parameters. By default interpreted as a covariance matrix. Can also be interpreted as a correlation matrix by setting **matrix_type**. Must be of shape shape (N, N).
- **matrix_type** (*str*) – Either `'cov'` or `'cor'`. Defines whether the matrix should be interpreted as a covariance matrix or as a correlation matrix.
- **uncertainties** (*None or Iterable[float]*) – The uncertainties to be used in conjunction with a correlation matrix. Must be of shape (N,)
- **relative** (*bool*) – Whether the covariance matrix/the uncertainties should be interpreted as relative to **values**.

add_parameter_constraint(*name*, *value*, *uncertainty*, *relative=False*)

Apply a simple gaussian constraint to a single fit parameter.

Parameters

- **name** (*str*) – The name of the parameter to be constrained.
- **value** (*float*) – The value to which the parameter should be constrained.
- **uncertainty** (*float*) – The uncertainty with which the parameter should be constrained to the given value.
- **relative** (*bool*) – Whether the given uncertainty is relative to the given value.

get_matching_errors(*matching_criteria=None*, *matching_type='equal'*)

Return a list of uncertainty objects fulfilling the specified matching criteria.

Valid keys for matching_criteria:

- `name` (the unique error name)
- `type` (either `'simple'` or `'matrix'`)
- `correlated` (bool, only matches simple errors!)

- `reference` (either `'model'` or `'data'`)

Note: The error objects contained in the dictionary are not copies, but the original error objects. Modifying them is possible, but not recommended. If you do modify any of them, the changes will not be reflected in the total error calculation until the error cache is cleared. This can be done by calling the private dataset method `_clear_total_error_cache`.

Parameters

- **matching_criteria** (*dict or None*) – Key-value pairs specifying matching criteria. The resulting error array will only contain error objects matching *all* provided criteria. If `None`, all error objects are returned.
- **matching_type** (*str*) – How to perform the matching. If `'equal'`, the value in **matching_criteria** is checked for equality against the stored value. If `'regex'`, the value in **matching_criteria** is interpreted as a regular expression and is matched against the stored value.

Returns Dict mapping error name to `GaussianErrorBase`-derived error objects.

Return type dict[str, kafe2.core.error.GaussianErrorBase]

add_error(*err_val*, *name=None*, *correlation=0*, *relative=False*, *reference='data'*, ***kwargs*)

Add an uncertainty source to the fit.

Parameters

- **err_val** (*float or Iterable[float]*) – Pointwise uncertainty/uncertainties for all data points.
- **name** (*str or None*) – Unique name for this uncertainty source. If `None`, the name of the error source will be set to a random alphanumeric string.
- **correlation** (*float*) – Correlation coefficient between any two distinct data points.
- **relative** (*bool*) – If `True`, **err_val** will be interpreted as a *relative* uncertainty.
- **reference** (*str*) – Either `'data'` or `'model'`. Specifies which reference values to use when calculating absolute errors from relative errors.

Returns An error id which uniquely identifies the created error source.

Return type str

add_matrix_error(*err_matrix*, *matrix_type*, *name=None*, *err_val=None*, *relative=False*, *reference='data'*, ***kwargs*)

Add a matrix uncertainty source for use in the fit.

Parameters

- **err_matrix** – covariance or correlation matrix

- **matrix_type** (*str*) – One of `'covariance'`/`'cov'` or `'correlation'`/`'cor'`
- **name** (*str or None*) – Unique name for this uncertainty source. If `None`, the name of the error source will be set to a random alphanumeric string.
- **err_val** (*Iterable[float]*) – The pointwise uncertainties (mandatory if only a correlation matrix is given).
- **relative** (*bool*) – If `True`, the covariance matrix and/or **err_val** will be interpreted as a *relative* uncertainty.
- **reference** (*str*) – Either `'data'` or `'model'`. Specifies which reference values to use when calculating absolute errors from relative errors.

Returns An error id which uniquely identifies the created error source.

Return type str

disable_error(*err_id*)

Temporarily disable an uncertainty source so that it doesn't count towards calculating the total uncertainty.

Parameters **err_id** (*str*) – error id

enable_error(*err_id*)

(Re-)Enable an uncertainty source so that it counts towards calculating the total uncertainty.

Parameters **err_id** (*str*) – error id

do_fit(*asymmetric_parameter_errors=False*)

Perform the minimization of the cost function.

Parameters **asymmetric_parameter_errors** (*bool*) – If `True`, calculate asymmetric parameter errors.

Returns A dictionary containing the fit results.

Return type dict

assign_model_function_name(*name*)

Assign a string to be the model function name.

Parameters **name** (*str*) – The new name.

assign_model_function_expression(*expression_format_string*)

Assign a plain-text-formatted expression string to the model function.

Parameters **expression_format_string** (*str*) – The plain text string.

assign_parameter_names(***par_names_dict*)

Assign display strings to all model function arguments.

Parameters **par_names_dict** – Dictionary mapping the parameter names to their display names.

assign_model_function_latex_name(*latex_name*)

Assign a LaTeX-formatted string to be the model function name.

Parameters **latex_name** (*str*) – The LaTeX string.

assign_model_function_latex_expression(*latex_expression_format_string*)

Assign a LaTeX-formatted expression string to the model function.

Parameters **latex_expression_format_string** (*str*) – The LaTeX string. Elements like `'{par_name}'` will be replaced automatically with the corresponding LaTeX names for the given parameter. These can be set with *assign_parameter_latex_names* (page 167).

assign_parameter_latex_names(***par_latex_names_dict*)

Assign LaTeX-formatted strings to all model function arguments.

Parameters **par_latex_names_dict** – Dictionary mapping the parameter names to their latex names.

get_result_dict(*asymmetric_parameter_errors=False*)

Return a dictionary of the fit results.

Parameters **asymmetric_parameter_errors** (*bool*) – If `True`, calculate asymmetric parameter errors.

Returns A dictionary containing the fit results.

Return type dict

report(*output_stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*, *show_data=True*, *show_model=True*, *show_fit_results=True*, *asymmetric_parameter_errors=False*)

Print a summary of the fit state and/or results.

Parameters

- **output_stream** (*io.TextIOBase*) – The output stream to which the report should be printed.
- **show_data** (*bool*) – If `True`, print out information about the data.
- **show_model** (*bool*) – If `True`, print out information about the parametric model.
- **show_fit_results** (*bool*) – If `True`, print out information about the fit results.
- **asymmetric_parameter_errors** (*bool*) – If `True`, use two different parameter errors for up/down directions.

to_file(*filename*, *file_format=None*, *calculate_asymmetric_errors=False*)

Write kafe2 object to file

Parameters

- **filename** (*str*) – Filename for the output.

- **file_format** (*str or None*) – A format for the output file. If None, the extension from the filename is used.
- **calculate_asymmetric_errors** (*bool*) – If asymmetric errors should be calculated before saving the results.

save_state(*filename: str*, *file_format: Optional[str] = None*, *calculate_asymmetric_errors: bool = False*)

Write current state of the fit to file. Unlike to *to_file* this does not contain information regarding how the fit is constructed - this is because for complex fit objects a reconstruction from e.g. YAML may not work. In such cases the fit object should be contructed via Python and *save_state* and *load_state* should be used.

Parameters

- **filename** (*str*) – Filename for the output.
- **file_format** (*str or None*) – A format for the output file. If None, the extension from the filename is used.
- **calculate_asymmetric_errors** (*bool*) – If asymmetric errors should be calculated before saving the results.

load_state(*filename: str*, *file_format: Optional[str] = None*)

load current fit state from the specified file that was written with *save_state*.

Parameters

- **filename** (*str*) – Filename for the input.
- **file_format** (*str or None*) – A format for the output file. If None, the extension from the filename is used.

class kafe2.fit._base.**FitEnsembleBase**

Bases: object

Object for generating ensembles of fits to pseudo-data generated according to the specified uncertainty model.

This is a purely abstract class implementing the minimal interface required by all types of fit ensembles.

FIT_TYPE = None

exception kafe2.fit._base.**FitEnsembleException**

Bases: Exception

class kafe2.fit._base.**FunctionFormatter**(*name*, *latex_name=None*, *arg_formatters=None*, *expression_string=None*, *latex_expression_string=None*)

Bases: kafe2.fit.io.file.FileIOMixin, object

Base class for function formatter objects. Requires further specialization for each type of model function. Objects derived from this class store information relevant for constructing plain-text and/or LaTeX string representations of functions.

For this, the function name, formatted as a plain-text/LaTeX string, as well as a list of references to *ParameterFormatter* (page 172) objects which contain information on how to format the model function arguments is stored.

Optionally, plain-text/LaTeX expression strings can be provided. These are strings representing the model function expression (i.e. mathematical formula).

The formatted string is obtained by calling the *get_formatted* (page 170) method.

Construct a formatter for a model function:

Parameters

- **name** (*str*) – A plain-text-formatted string indicating the function name.
- **latex_name** (*str*) – A LaTeX-formatted string indicating the function name.
- **arg_formatters** (*list[kafe2.fit._base.ParameterFormatter* (page 172)*]*) – List of *ParameterFormatter* (page 172)-derived objects, formatters for function arguments.
- **expression_string** (*str*) – A plain-text-formatted string indicating the function expression.
- **latex_expression_string** (*str*) – A LaTeX-formatted string indicating the function expression.

DEFAULT_EXPRESSION_STRING = None

DEFAULT_LATEX_EXPRESSION_STRING = None

property expression_format_string

A plain-text-formatted expression for the function. This function will replace all function parameters with their corresponding strings. For example the string "{a}*{x}+{b}" will turn into "A*x + B" when the name of the parameter a was set to "A", and the name of b is set to "B".

Return type str

property latex_expression_format_string

A LaTeX-formatted expression for the function. This function will replace all function parameters with their corresponding latex string. For example the string `"{a}{x}+{b}"` will turn into `"A_0 x + B"` when the latex name of the parameter `a` was set to `"A_0"`, and the latex name of `b` is set to `"B"`.

Return type str

property name

A plain-text-formatted string indicating the function name.

Return type str

property latex_name

A LaTeX-formatted string indicating the function name.

Return type str

property description

A short plain-text description of the function.

Return type str

property arg_formatters

The list of *ParameterFormatter* (page 172)-derived objects used for formatting all model function arguments.

Return type list[*ParameterFormatter* (page 172)]

property par_formatters

List of *ParameterFormatter* (page 172)-derived objects used for formatting the fit parameters, excluding the independent parameter(s).

Return type list[*ParameterFormatter* (page 172)]

get_formatted(*with_par_values=False*, *n_significant_digits=2*, *format_as_latex=False*, *with_expression=False*)

Get a formatted string representing this model function.

Parameters

- **with_par_values** (*bool*) – If `True`, output will include the value of each function parameter (e.g. `f(a=1, b=2, ...)`).
- **n_significant_digits** (*int*) – Number of significant digits for rounding.
- **format_as_latex** (*bool*) – If `True`, the returned string will be formatted using LaTeX syntax.
- **with_expression** (*bool*) – If `True`, the returned string will include the expression assigned to the function.

class kafe2.fit._base.**ModelFunctionBase**(*model_function=<function linear_model>*, *independent_argcount=1*)

Bases: `kafe2.fit.io.file.FileIOMixin`, `object`

This is a purely abstract class implementing the minimal interface required by all model functions.

In order to be used as a model function, a native Python function must be wrapped by an object whose class derives from this base class. There is a dedicated `ModelFunction` specialization for each type of data container.

This class provides the basic functionality used by all `ModelFunction` objects. These use introspection (`inspect`) for determining the parameter structure of the model function and to ensure the function can be used as a model function (validation).

Construct `ModelFunction` object (a wrapper for a native Python function):

Parameters

- **model_function** – function handle

- **independent_argcount** (*int*) – The amount of independent variables for this model. The first n variables of the model function will be treated as independent variables and will not be fitted.

FORMATTER_TYPE

alias of *kafe2.fit._base.format.ModelFunctionFormatter* (page 171)

property name

The model function name (a valid Python identifier)

property func

The underlying model function handle

property signature

The model function argument specification, as returned by `inspect.signature`

property argcount

The number of arguments the model function accepts. (including any independent variables which are not parameters)

property parcount

The number of fitting parameters in the model function.

property x_name

The name of the independent variable. `None` for 0 independent variables.

property parameter_names

The names of the parameters.

property formatter

The *ModelFunctionFormatter* (page 171)-derived object for this function

property defaults

The default values for model function parameters as a list

property defaults_dict

The default values for model function parameters as a dict

property source_code

class kafe2.fit._base.**ModelFunctionFormatter**(*name*, *latex_name=None*, *arg_formatters=None*, *expression_string=None*, *latex_expression_string=None*)

Bases: *kafe2.fit._base.format.FunctionFormatter* (page 168)

A formatter class for model functions.

This object stores the function name, formatted as a plain-text/LaTeX string, as well as a list of references to *ParameterFormatter* (page 172) objects which contain information on how to format the model function arguments. Additionally formatting information about the independent variable is stored.

Optionally, plain-text/LaTeX expression strings can be provided. These are strings representing the model function expression (i.e. mathematical formula).

The formatted string is obtained by calling the *get_formatted* (page 172) method.

Construct a formatter for a model function:

Parameters

- **name** (*str*) – A plain-text-formatted string indicating the function name.
- **latex_name** (*str*) – A LaTeX-formatted string indicating the function name.
- **arg_formatters** (*list[kafe2.fit._base.ParameterFormatter* (page 172)*]*) – List of *ParameterFormatter* (page 172)-derived objects, formatters for function arguments.
- **expression_string** (*str*) – A plain-text-formatted string indicating the function expression.
- **latex_expression_string** (*str*) – A LaTeX-formatted string indicating the function expression.

property par_formatters

List of *ParameterFormatter* (page 172)-derived objects used for formatting the fit parameters, excluding the independent parameter(s).

Return type list[*ParameterFormatter* (page 172)]

get_formatted(*with_par_values=False*, *n_significant_digits=2*, *format_as_latex=False*, *with_expression=False*)

Create a formatted string representing this model function.

Parameters

- **with_par_values** (*bool*) – If `True`, output will include the value of each function parameter (e.g. `f(a=1, b=2, ...)`).
- **n_significant_digits** (*int*) – number of significant digits for rounding
- **format_as_latex** (*bool*) – If `True`, the returned string will be formatted using LaTeX syntax.
- **with_expression** (*bool*) – If `True`, the returned string will include the expression assigned to the function.

Returns The formatted string representing this model function.

Return type str

class kafe2.fit._base.**ParameterFormatter**(*arg_name*, *value=None*, *error=None*, *asymmetric_error=None*, *name=None*, *latex_name=None*)

Bases: `kafe2.fit.io.file.FileIOMixin`, `object`

Formatter class for model parameter objects.

These objects store the relevant information for constructing plain-text and/or LaTeX string representations of model function parameters.

For this, the original argument name, the name for representation, formatted as a plain-text/LaTeX string, its value and its uncertainty is stored.

The formatted string is obtained by calling the *get_formatted* (page 174) method.

Construct a Parameter Formatter.

Parameters

- **arg_name** (*str*) – A plain string indicating the parameter's signature inside the function call.
- **value** (*float or None*) – The parameter value.
- **error** (*float or None*) – The symmetric parameter error.
- **asymmetric_error** (*tuple[float, float] or None*) – The asymmetric parameter errors.
- **name** (*str or None*) – A plain-text-formatted string indicating the parameter name.
- **latex_name** (*str or None*) – A LaTeX-formatted string indicating the parameter name.

Return type *ParameterFormatter* (page 172)

property arg_name

Name of the function argument this formatter represents.

Return type str

property name

The plain-text-formatted string indicating the parameter name.

Return type str

property latex_name

The LaTeX-formatted string indicating the parameter name.

Return type str

property value

The parameter value.

Return type float or None

property error

The symmetric parameter error.

Return type float or None

property error_rel

The relative parameter error.

Return type float or None

property asymmetric_error

Tuple containing the asymmetric parameter errors.

Return type tuple[float, float] or None

property error_up

The upper uncertainty (only for asymmetric errors).

Return type float or None

property error_down

The lower uncertainty (only for asymmetric errors).

Return type float or None

property fixed

If the parameter has been fixed by the user. `True` when it's fixed, `False` when not.

Return type bool

get_formatted(*with_name=False*, *with_value=True*, *with_errors=True*, *n_significant_digits=2*, *round_value_to_error=True*, *asymmetric_error=False*, *format_as_latex=False*)

Get a formatted string representing this model parameter.

Parameters

- **with_name** (*bool*) – If `True`, the output will include the parameter name.
- **with_value** (*bool*) – If `True`, the output will include the parameter value.
- **with_errors** (*bool*) – If `True`, the output will include the parameter error(s).
- **n_significant_digits** (*int*) – Number of significant digits for rounding.
- **round_value_to_error** (*bool*) – If `True`, the parameter value will be rounded to the same precision as the uncertainty.
- **asymmetric_error** (*bool*) – If `True`, the asymmetric parameter uncertainties are used.
- **format_as_latex** (*bool*) – If `True`, the returned string will be formatted using LaTeX syntax.

Returns The string representation of the parameter.

Return type str

class kafe2.fit._base.**ParametricModelBaseMixin**(*model_func*, *model_parameters*, **args*, ***kwargs*)

Bases: object

A "mixin" class for representing a parametric model. Inheriting from this class in addition to a data container class additionally stores a Python function handle referring to the model function. The argument structure of this function must be compatible with the data container type and it must return a numpy array of the same shape as the *data* (page 157) property of the data container.

This mixin class introduces an additional *parameters* (page 175) property for the object, which can be used to obtain and set the values of the parameter

Derived classes should inherit from *ParametricModelBaseMixin* (page 174) and the relevant data container (in that order).

Mixin constructor: sets and initialized the model function.

Parameters

- **model_func** – handle of Python function (the model function)
- **model_parameters** – iterable of parameter values with which the model function should be initialized

MODEL_FUNCTION_TYPE

alias of *kafe2.fit._base.model.ModelFunctionBase* (page 170)

property ndf

property parameters

Model parameter values

class kafe2.fit._base.**Plot**(*fit_objects*, *separate_figures=False*)

Bases: object

This is a class implementing the creation of Fits from one of more subclasses of :py:obj`PlotAdapterBase`. Consequently a *Plot* (page 175) object manages one or several matplotlib figures. It controls the overall figure layout and is responsible for axes, subplot and legend management.

FIT_INFO_STRING_FORMAT_CHI2 = '{model_function}\n{parameters}\n $\\hookrightarrow${fit_quality}\n $\\hookrightarrow \\chi^2 \\, \\mathrm{{probability =}}${chi2_probability}\n'

FIT_INFO_STRING_FORMAT_SATURATED = '{model_function}\n{parameters}\n $\\hookrightarrow${fit_quality}\n'

FIT_INFO_STRING_FORMAT_NOT_SATURATED = '{model_function}\n{parameters}\n $\\hookrightarrow${cost}\n $\\hookrightarrow${fit_quality}\n'

property figures

The matplotlib figures managed by this object.

property axes

A list of dictionaries (one per figure) mapping names to `matplotlib` *Axes* objects contained in this figure.

property x_range

The plotting x-range for each fit handled by this `Plot` object. :param: List of tuples containing the x_ranges for each fit. :type: list[tuple[float, float]] or tuple[float, float]

property y_range

The plotting y-range for each fit handled by this `Plot` object. :param: List of tuples containing the y_ranges for each fit. :type: list[tuple[float, float]] or tuple[float, float]

property x_scale

The x-scale for each fit used for creating the support values when plotting and axis scaling. :type: list[str] or str

property y_scale

The y-scale for each fit used when plotting. :type: list[str] or str

property x_label

The x-label(s) of the plot. If multiple fits are handled by this plot this is a list of strings. Multiple labels will be separated by a comma in the final plot while skipping duplicates. If a label is `None` or `'__del__'` it will be removed. :type: str or list[str]

property y_label

The y-label(s) of the plot. If multiple fits are handled by this plot this is a list of strings. Multiple labels will be separated by a comma in the final plot while skipping duplicates. If a label is `None` or `'__del__'` it will be removed. :type: str or list[str]

property x_ticks

property y_ticks

static show(**args*, ***kwargs*)

Convenience wrapper for matplotlib.pyplot.show()

plot(*legend=True*, *fit_info=True*, *asymmetric_parameter_errors=False*, *ratio=False*, *ratio_range=None*, *ratio_height_share=0.25*, *residual=False*, *residual_range=None*, *residual_height_share=0.25*, *plot_width_share=0.5*, *figsize=None*)

Plot data, model (and other subplots) for all child `Fit` objects.

Parameters

- **legend** (*bool or Collection[bool]*) – if `True`, a legend is rendered
- **fit_info** – If `True`, fit results will be shown in the legend. This can also be a list of booleans, corresponding to the fits handled by this *Plot* (page 175)-object.
- **asymmetric_parameter_errors** – if `True`, parameter errors in fit results will be asymmetric
- **ratio** – if `True`, a secondary plot containing data/model ratios is shown below the main plot

- **ratio_range** (*tuple of 2 floats*) – the *y* range to set in the secondary plot
- **ratio_height_share** (*float*) – share of the total height to be taken up by the secondary plot
- **plot_width_share** (*float*) – share of the total width to be taken up by the plot(s)
- **figsize** (*tuple of 2 floats*) – the (*width*, *height*) of the figure (in inches) or `None` to use default

Returns dictionary containing information about the plotted objects

Return type dict

get_keywords (*plot_type*)

Retrieve keyword arguments for plots with type *plot_type* as they would be used when calling *plot*.

This is an advanced function. An understanding of how plotting with *matplotlib* and the *PlotAdapter* classes in *kafe2* work is recommended.

The *plot_type* must be one of the plot types registered in the *PlotAdapter* (e.g. `'data'`, `'model_line'` etc.).

Parameters **plot_type** (*str*) – keyword identifying the plot type for which to set a custom keyword argument

Returns list of dictionaries (one per fit instance) containing plot keywords and their values

Return type list of dict

set_keywords (*plot_type*, *keyword_spec*)

Set values for keyword arguments used for plots with type *plot_type*.

This is an advanced function. An understanding of how plotting with *matplotlib* and the *PlotAdapter* classes in *kafe2* work is recommended.

The *plot_type* must be one of the plot types registered in the *PlotAdapter* (e.g. `'data'`, `'model_line'` etc.).

The *keyword_spec* contains dictionaries whose contents will be passed as keyword arguments to the plot adapter method responsible for plotting the *plot_type*. If *keyword* spec contains a key for which a default value is configured, it will be overridden.

Passing the following special values for a keyword will have the following effects:

- `'__del__'`: the value will be removed from the keyword arguments. This includes default values, meaning that the plot call will be made **without** the keyword argument even if a default value for it exists.
- `'__default__'`: the customized value will be replaced by the default value.

Note: No keyword/value validation is done: everything is passed to the underlying plot methods as specified. Incorrect or incompatible keywords may be ignored or lead to errors.

As an example, to override the labels shown in the legend entries for the *data*

```
p = Plot([fit_1, fit_2])
p.customize('data', [dict(label='My Data Label'), dict(label=
↪'Another Data Label')])
```

To set keywords for a single *fit*, pass values as `(index, value)`, where *index* is the index of the *fit* object:

```
p = Plot([fit_1, fit_2])
p.customize('data', [(1, dict(label='Another Data Label'))])
```

Parameters

- **plot_type** (*str*) – keyword identifying the plot type for which to set a custom keyword argument
- **keyword_spec** (list of values or list of 2-tuples like `(index, value)`) – specification of dictionaries containing the keyword arguments to use for fit. Can be either a list of dictionaries with a length corresponding to the number of *fit* objects managed by this *Plot* instance, or a list of tuples of the form `(index, dict)`, where `index` denotes the index of the *fit* object for which the dictionary *dict* should be used.

Returns this *Plot* instance

Return type *Plot*

customize(*plot_type*, *keyword*, *values*)

Set values for keyword arguments used for plots with type *plot_type*.

This is a convenience wrapper around *set_keywords*.

The *keyword* will be passed to the plot adapter method responsible for plotting the *plot_type* as a keyword argument with a value taken from *values*. If a default value for *keyword* is configured, it is overridden.

The *values* can be specified in two ways:

1. as a list with a length corresponding to the number of *fit* objects managed by this *Plot* instance. The special value `'__skip__'` can be used to skip *fit* objects.
2. as a list of tuples of the form `(index, value)`, where *index* denotes the index of the *fit* object for which the *value* should be used.

Passing the following special values for a keyword will have the following effects:

- `'__del__'`: the value will be removed from the keyword arguments. This includes default values, meaning that the plot call will be made **without** the keyword argument even if a default value for it exists.
- `'__default__'`: the customized value will be replaced by the default value.
- `'__skip__'`: the keywords for this *fit* will not be changed.

Note: No keyword/value validation is done: everything is passed to the underlying plot methods as specified. Incorrect or incompatible keywords may be ignored or lead to errors.

As an example, to override the labels shown in the legend entries for the *data*

```
p = Plot([fit_1, fit_2])
p.customize('data', 'label', ['My Data Label', 'Another Data Label'])
```

To set keywords for a single *fit*, pass values as `(index, value)`, where *index* is the index of the *fit* object:

```
p = Plot([fit_1, fit_2])
p.customize('data', 'label', [(1, 'Another Data Label')])
```

Parameters

- **plot_type** (*str*) – keyword identifying the plot type for which to set a custom keyword argument
- **keyword** (*str*) – the keyword argument. The corresponding value in *values* will be passed to the plot adapter method using this keyword argument
- **values** (list of values or list of 2-tuples like `(index, value)`) – values that the keyword argument should take for each fit. Can be a list of values with a length corresponding to the number of *fit* objects managed by this *Plot* instance, or a list of tuples of the form `(index, value)`

Returns this *Plot* instance

Return type *Plot*

save(*fname=None*, *figures='all'*, **args*, ***kwargs*)

Saves the plot figures to files. args and kwargs are passed on to matplotlib.Figure.savefig() .

Parameters

- **fname** (*None or str or iterable of str.*) – Output file name(s), defaults to fit.png or fit_0.png, fit_1.png, …
- **figures** (*'all' or int or iterable of int*) – Which figures to save.

class kafe2.fit._base.**PlotAdapterBase**(*fit_object*, *from_container=False*)

Bases: object

This is a purely abstract class implementing the minimal interface required by all types of plot adapters.

A PlotAdapter object can be constructed for a Fit object of the corresponding type. Its main purpose is to provide an interface for accessing data stored in the Fit object, for the purposes of plotting. Most importantly, it provides methods to call the relevant matplotlib methods for plotting the data, model (and other information, depending on the fit type), and constructs the arrays required by these routines in a meaningful way.

Classes derived from *PlotAdapterBase* (page 179) must at the very least contain properties for constructing the *x* and *y* point arrays for both the data and the fitted model, as well as methods calling the matplotlib routines doing the actual plotting.

Construct a PlotAdapter for a Fit object:

Parameters

- **fit_object** (kafe2.fit._base.FitBase (page 160)) – An object derived from *FitBase* (page 160)
- **from_container** (*bool*) – Whether fit_object was created ad-hoc from just a data container.

PLOT_STYLE_CONFIG_DATA_TYPE = 'default'

PLOT_SUBPLOT_TYPES = {'data': {'container_valid': True, 'plot_adapter_method': 'plot_data', 'target_axes': 'main'}, 'model': {'hide': True, 'plot_adapter_method': 'plot_model', 'target_axes': 'main'}, 'ratio': {'plot_adapter_method': 'plot_ratio', 'plot_style_as': 'data', 'target_axes': 'ratio'}, 'residual': {'plot_adapter_method': 'plot_residual', 'plot_style_as': 'data', 'target_axes': 'residual'}}

AVAILABLE_X_SCALES = ('linear',)

AVAILABLE_Y_SCALES = ('linear', 'log')

call_plot_method(*plot_type*, *target_axes*, ***kwargs*)

Call the registered plot method for *plot_type*.

Parameters

- **plot_type** (*str*) – key identifying a registered plot type for this PlotAdapter
- **target_axes** (*matplotlib.Axes* object) – axes to plot to
- **kwargs** (*dict*) – keyword arguments to pass to the plot method

Returns return value of the plot method

update_plot_kwargs (*plot_type*, *plot_kwargs*)

Update the value of keyword arguments *plot_kwargs* to be passed to the plot method for for *plot_type*.

If a keyword argument should be removed, the value of the keyword in *plot_kwargs* can be set to the special value `'__del__'`. To indicate that the default value should be used, the special value `'__default__'` can be set as a value.

Parameters

- **plot_type** (*str*) – key identifying a registered plot type for this PlotAdapter
- **plot_kwargs** (*dict*) – dictionary containing keywords arguments to override

abstract property data_x

The *x* coordinates of the data (used by *plot_data* (page 182)).

Return type numpy.ndarray

abstract property data_y

The *y* coordinates of the data (used by *plot_data* (page 182)).

Return type numpy.ndarray

abstract property data_xerr

The magnitude of the data *x* error bars (used by *plot_data* (page 182)).

Return type numpy.ndarray

abstract property data_yerr

The magnitude of the data *y* error bars (used by *plot_data* (page 182)).

Return type numpy.ndarray

abstract property model_x

The *x* coordinates of the model (used by *plot_model* (page 182)).

Return type numpy.ndarray

abstract property model_y

The *y* coordinates of the model (used by *plot_model* (page 182)).

Return type numpy.ndarray

abstract property model_xerr

The magnitude of the model *x* error bars (used by *plot_model* (page 182)).

Return type numpy.ndarray

abstract property model_yerr

The magnitude of the model *y* error bars (used by *plot_model* (page 182)).

Return type numpy.ndarray

property x_range

The *x* axis plot range.

Return type tuple[float, float]

property y_range

The *y* axis plot range.

Return type tuple[float, float]

property x_scale

The *x* axis scale. Available scales are given in `AVAILABLE_X_SCALES`

Return type str

property y_scale

The *y* axis scale. Available scales are given in `AVAILABLE_Y_SCALES`

Return type str

property x_label

The *x* axis label of the fit handled by this plot adapter. If `'__del__'` is used, the label will be set to `None`.

Return type str

property y_label

The *x* axis label of the fit handled by this plot adapter. If `'__del__'` is used, the label will be set to `None`.

Return type str

property x_ticks

property y_ticks

property from_container

Whether the contained fit object was created ad-hoc from just a data container.

abstract plot_data(*target_axes*, ***kwargs*)

Method called by the main plot routine to plot the data points to a specified `matplotlib.axes.Axes` object.

Parameters **target_axes** (*matplotlib.axes.Axes*) – The `matplotlib` target axes.

Returns plot handle(s)

abstract plot_model(*target_axes*, ***kwargs*)

Method called by the main plot routine to plot the model to a specified `matplotlib.axes.Axes` object.

Parameters **target_axes** (*matplotlib.axes.Axes*) – The `matplotlib` target axes.

Returns plot handle(s)

plot_ratio(*target_axes*, *error_contributions=('data',)*, ***kwargs*)

Plot the data/model ratio to a specified `matplotlib.axes.Axes` object.

Parameters

- **target_axes** (*matplotlib.axes.Axes*) – The `matplotlib` axes used for plotting.
- **error_contributions** (*str or Tuple[str]*) – Which error contributions to include when plotting the data. Can either be `data`, `'model'` or both.
- **kwargs** (*dict*) – Keyword arguments accepted by `matplotlib.pyplot.errorbar`.

Returns plot handle(s)

plot_residual(*target_axes*, *error_contributions=('data',)*, ***kwargs*)

Plot the residuals to a `matplotlib.axes.Axes` object.

Parameters

- **target_axes** (*matplotlib.axes.Axes*) – The `matplotlib` axes used for plotting.
- **error_contributions** (*str or Tuple[str]*) – Which error contributions to include when plotting the data. Can either be `data`, `'model'` or both.
- **kwargs** (*dict*) – Keyword arguments accepted by `matplotlib.pyplot.errorbar`.

Returns plot handle(s)

get_formatted_model_function(***kwargs*)

return model function string

property model_function_parameter_formatters

The model function parameter formatters, excluding the independent variable.

class kafe2.fit._base.**ScalarFormatter**(*sigma*, *n_significant_digits=2*)

Bases: `object`

Format a scalar to a specified precision, according to the uncertainty.

Parameters

- **sigma** (*float*) – The uncertainty of the parameter.
- **n_significant_digits** (*int*) – Number of significant digits.

kafe2.fit._base.**kc_plot_style**(*data_type*, *subplot_key*, *property_key*)

kafe2.fit._base.**latexify_ascii**(*ascii_string*)

Create a true type latex string of an standard ascii string.

Parameters **ascii_string** (*str*) – The string to be converted

Return type str

Python Module Index

k

Index

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

X

Y