PyQt-Fit Documentation

Release 1.3.1

Barbier de Reuille, Pierre

August 08, 2014

Contents

PyQt-Fit is a regression toolbox in Python with simple GUI and graphical tools to check your results. It currently handles regression based on user-defined functions with user-defined residuals (i.e. parametric regression) or non-parametric regression, either local-constant or local-linear, with the option to provide your own. The GUI currently provides an interface only to parametric regression.

Contents:

CHAPTER 1

Introduction to PyQt-Fit

The GUI for 1D data analysis is invoked with:

```
$ pyqt_fit1d.py
```

PyQt-Fit can also be used from the python interpreter. Here is a typical session:

```
>>> import pyqt_fit
>>> from pyqt_fit import plot_fit
>>> import numpy as np
>>> from matplotlib import pylab
>>> x = np.arange(0,3,0.01)
>>> y = 2*x + 4*x**2 + np.random.randn(*x.shape)
>>> def fct((a0, a1, a2), x):
...     return a0 + a1*x + a2*x*x
>>> fit = pyqt_fit.CurveFitting(x, y, (0,1,0), fct)
>>> result = plot_fit.fit_evaluation(fit, x, y)
>>> print fit(x) # Display the estimated values
>>> plot_fit.plot1d(result)
>>> pylab.show()
```

PyQt-Fit is a package for regression in Python. There are two set of tools: for parametric, or non-parametric regression.

For the parametric regression, the user can define its own vectorized function (note that a normal function wrappred into numpy's "vectorize" function is perfectly fine here), and find the parameters that best fit some data. It also provides bootstrapping methods (either on the samples or on the residuals) to estimate confidence intervals on the parameter values and/or the fitted functions.

The non-parametric regression can currently be either local constant (i.e. spatial averaging) in nD or local-linear in 1D only. There is a version of the bootstrapping adapted to non-parametric regression too.

The package also provides with four evaluation of the regression: the plot of residuals vs. the X axis, the plot of normalized residuals vs. the Y axis, the QQ-plot of the residuals and the histogram of the residuals. All this can be output to a CSV file for further analysis in your favorite software (including most spreadsheet programs).

CHAPTER 2

Regression using the GUI - tutorial

2.1 Using the interface

The script is starting from the command line with:

```
$ pyqt_fit1d.py
```

Once starting the script, the interface will look like this:

The interface is organised in 4 sections:

1. the top-left of the window to define the data to load and process;
2. the bottom-left to define the function to be fitted and its parameters;
3. the top-right to define the options to compute confidence intervals;
4. the bottom-right to define the output options.

2.1.1 Loading the Data

The application can load CSV files. The first line of the file must be the name of the available datasets. In case of missing data, only what is available on the two selected datasets are kept.

Once loaded, the available data sets will appear as option in the combo-boxes. You need to select for the X axis the explaining variable and the explained variable on the Y axis.

2.1.2 Defining the regression function

First, you will want to choose the function. The available functions are listed in the combo box. When selecting a function, the list of parameters appear in the list below. The value presented are estimated are a quick estimation from the data. You can edit them by double-clicking. It is also where you can specify if the parameter is of known value, and should therefore be fixed.

If needed, you can also change the computation of the residuals. By default there are two kind of residuals:

Standard residuals are simply the difference between the estimated and observed value.

Difference of the logs residual are the difference of the log of the values.

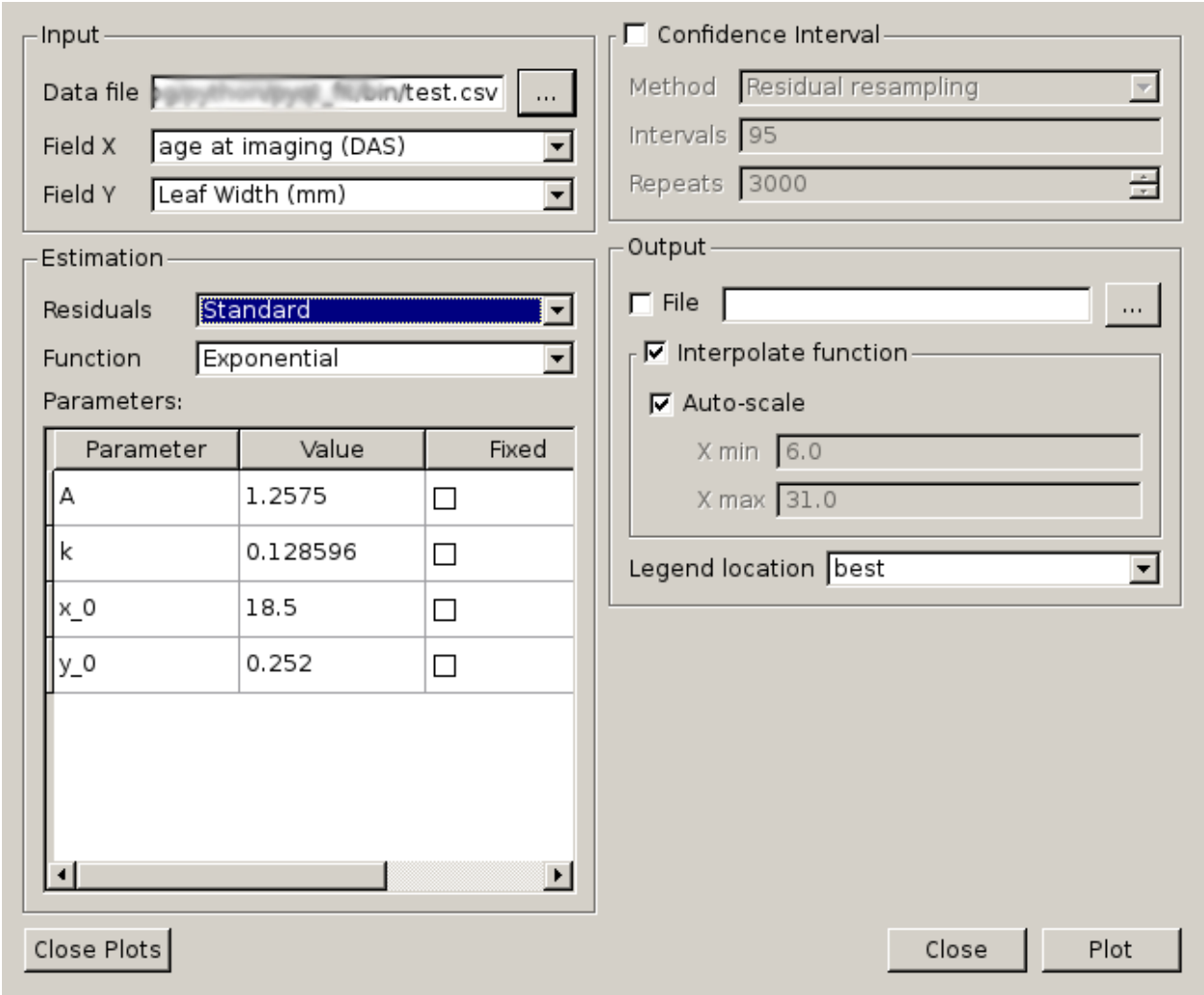
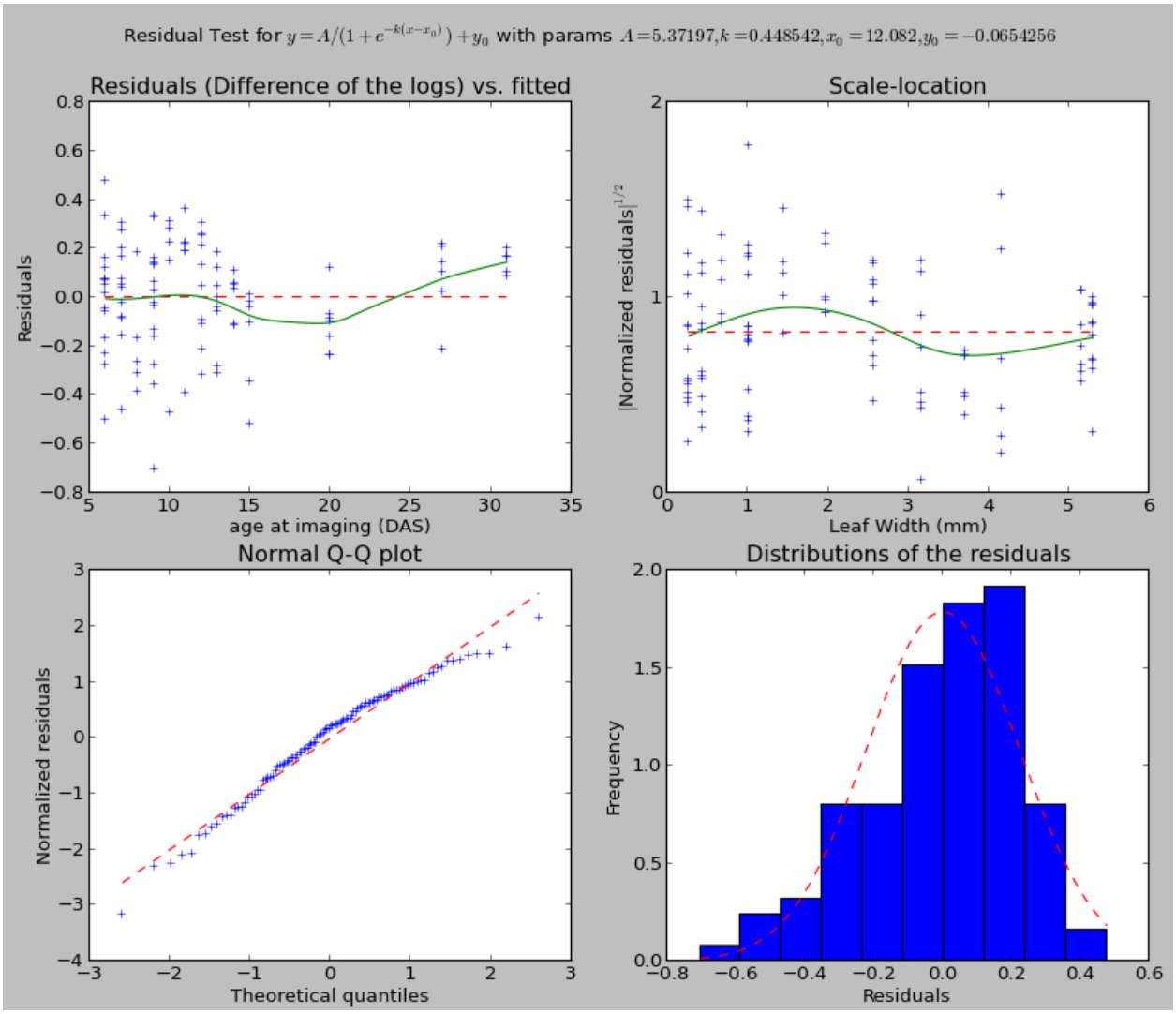


Figure 2.1: Main GUI of PyQt-Fit

2.1.3 Plotting and output

By default, the output consists in the data points, and the fitted function, interpolated on the whole range of the input data. If is, however, possible to both change the range of data, or even evaluate the function on the existing data points rather than interpolated ones.

The output also presents a window to evaluate the quality of the fitting:



In general, the dashed red line is the target to achieve for a good fitting. When present the green line is the estimates that should match the red line.

The top-left graph presents the distribution of the residuals against the explaining variable. The green line shows a local-linear regression of the residuals. It should be aligned with the dashed red line.

The top-right graph presents the distribution of the square root of the standardized residuals against the explained variable. The purpose of this graph is to test the uniformity of the distribution. The green line is again a local-linear regression of the points. The line should be as flat and horizontal as possible. If the distribution is normal, the green line should match the dashed red line.

The bottom right graph presents a histogram of the residuals. For parametric fitting, the residuals should globally be normal.

The bottom left graph presents a QQ-plot, matching the theoretical quantiles and the standardized residuals. If the residuals are normally distributed, the points should be on the dashed red line.

The result of the fitting can also be output. What is written correspond exactly to what is displayed. The output is also a CSV file, and is meant to be readable by a human.

2.1.4 Confidence interval

Confidence interval can be computed using bootstrapping. There are two kinds of boostrapping implemented:

regular bootstrapping The data are resampled, the pairs (x, y) are kept. There is no assumption made. But is is often troublesome in regression, tending to flatten the results.

residual resampling After the first evaluation, for each pair (x, y), we find the estimated value $\hat{y}$. Then, the residuals are re-sampled, and new pairs $(x, \hat{y}+r')$ are recreated, r' being the resampled residual.

The intervals ought to be a list of semi-colon separated values of percentages. At last, the number of repeats will define how many re-sampling there will be.

2.2 Defining your own function

First, you need to define the environment variable `PYQTFIT_PATH` and add a list of colon-separated folders. In each folder, you can add python modules in a `functions` sub-folder. For example, if the path `~/.pyqtfit` is in `PYQTFIT_PATH`, then you need to create a folder `~/.pyqtfit/functions`, in which you can add your own python modules.

Which module will be loaded, and the functions defined in it will be added in the interface. A function is a class or an object with the following properties:

`name` Name of the function

`description` Equation of the function

`args` List of arguments

`__call__(args, x)` Compute the function. The `args` argument is a tuple or list with as many elements are in the `args` attribute of the function.

`init_args(x,y)` Function guessing some initial arguments from the data. It must return a list or tuple of values, one per argument to the function.

`Dfun(args, x)` Compute the jacobian of the function at the points `c`. If the function is not provided, the attribute should be set to None, and the jacobian will be estimated numerically.

As an example, here is the definition of the cosine function:

```
import numpy as np

class Cosine(object):
  name = "Cosine"
  args = "y0 C phi t".split()
  description = "y = y0 + X cos(phi x + t)"

  @staticmethod
  def __call__((y0,C,phi,t), x):
    return y0 + C*np.cos(phi*x+t)

  Dfun = None
```

```
  @staticmethod
  def init_args(x, y):
    C = y.ptp()/2
    y0 = y.min() + C
    phi = 2*np.pi/x.ptp()
    t = 0
    return (y0, C, phi, t)
```

2.3 Defining your own residual

Similarly to the functions, it is possible to implement your own residual. The rediduals need to be in a `residuals` folder. And they need to be object or classes with the following properties:

`name` Name of the residuals

`description` Formula used to compute the residuals

`__call__(y1, y0)` Function computing the residuals, `y1` being the original data and `y0` the estimated data.

`invert(y, res)` Function applying the residual to the estimated data.

`Dfun(y1, y0, dy)` Compute the jacobian of the residuals. `y1` is the original data, `y0` the estaimted data and `dy` the jacobian of the function at `y0`.

As an example, here is the definition of the log-residuals:

```
class LogResiduals(object):
  name = "Difference of the logs"
  description = "log(y1/y0)"

  @staticmethod
  def __call__(y1, y0):
    return log(y1/y0)

  @staticmethod
  def Dfun(y1, y0, dy):
    """
    J(log(y1/y0)) = -J(y0)/y0
    where J is the jacobian and division is element-wise (per row)
    """
    return -dy/y0[newaxis,:]

  @staticmethod
  def invert(y, res):
    """
    Multiply the value by the exponential of the residual
    """
    return y*exp(res)
```

CHAPTER 3

Parametric regression tutorial

3.1 Introduction

Given a set of observations (x_i, y_i), with $x_i = (x_{i1}, \ldots, x_{ip})^T \in \mathbb{R}^p$. We assume, there exists a function $f(\theta, x)$ and a set of parameters $\theta \in \mathbb{R}^q$ such that:

$$y_i = f(\theta, x_i) + \epsilon_i$$

with $\epsilon_i \in \mathbb{R}$ such that $E(\epsilon) = 0$.

The objective is to fine the set of parameters *theta*. Obviously, the real function is inaccesible. Instead, we will try to find an estimate of the parameters, $\hat{\theta}$ using the least square estimator, which is:

$$\hat{\theta} = \text{argmin}_{\theta \in \mathbb{R}^q} \left(f(\theta, x_i) - y_i\right)^2$$

The method is based on the SciPy function `scipy.optimize.leastsq`, which relies on the MINPACK's functions `lmdif` and `lmder`. Both functions implement a modified Levenberg-Marquardt algorithm to solve the least-square problem. Most of the output of the main curve fitting option will be the output of the least-square function in scipy.

3.2 A simple example

As a simple example, we will take the function f to be:

$$f((a_0, a_1, a_2), x) = a_0 + a_1 x + a_2 x^2$$

Let's assume the points look like this:

The data points have been generated by that script:

```
>>> import numpy as np
>>> from matplotlib import pylab as plt
>>> x = np.arange(0,3,0.01)
>>> y = 2*x + 4*x**2 + 3*np.random.randn(*x.shape)
>>> plt.plot(x,y,'+',label='data')
>>> plt.legend(loc=0)
>>> plt.xlabel('X'); plt.ylabel('Y')
```

So we will expect to find something close to $(0, 2, 4)$.

To perform the analysis, we first need to define the function to be fitted:

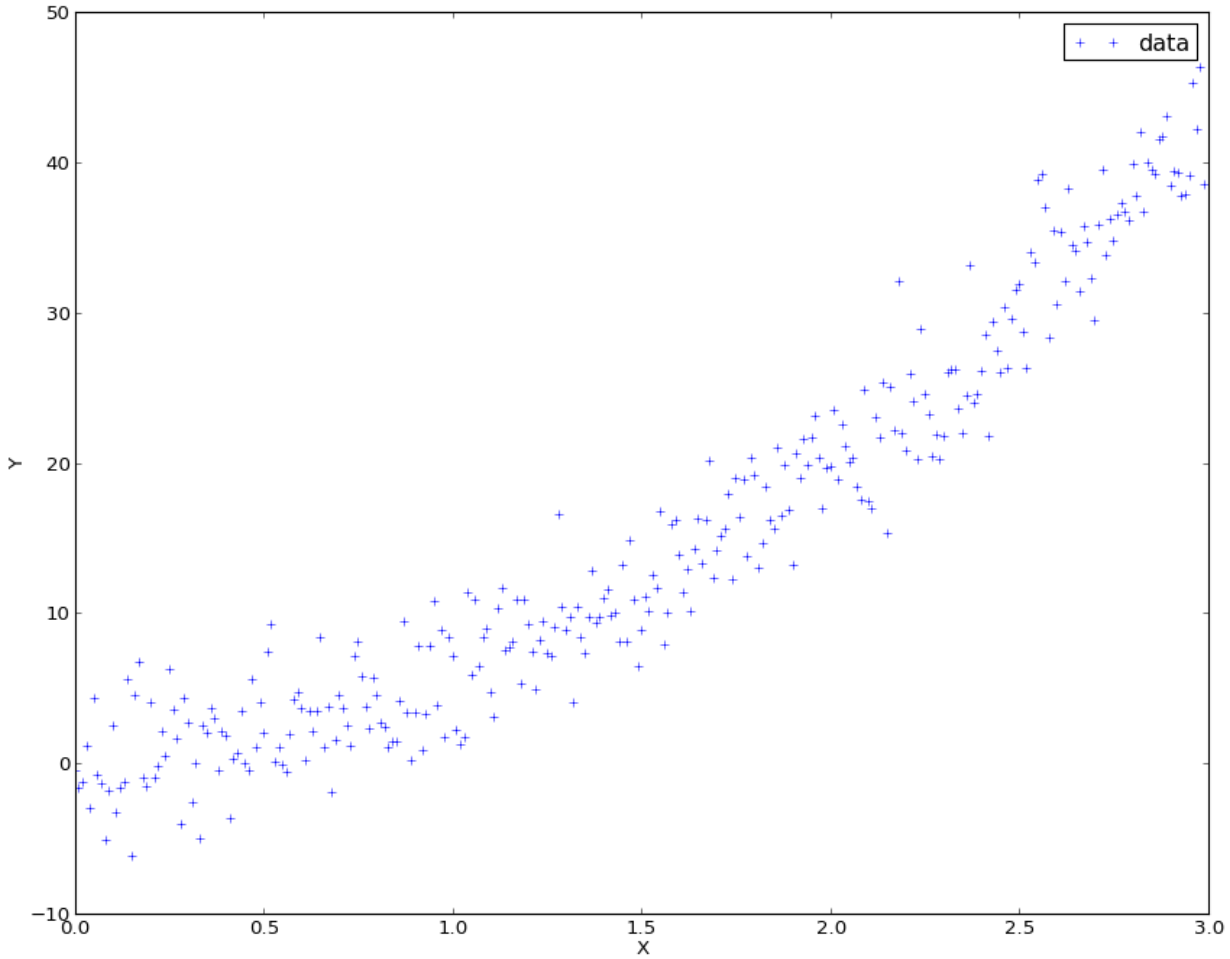


Figure 3.1: Raw data for curve fitting

```
>>> def f(params, x):
...   a0, a1, a2 = params
...   return a0 + a1*x+ a2*x**2
```

Then, we construct a `CurveFitting` object, which computes and stores the optimal parameters, and also behaves as a function for the fitted data:

```
>>> import pyqt_fit
>>> fit = pyqt_fit.CurveFitting(x,y,(0,1,0),f)
>>> print "The parameters are: a0 = {0}, a1 = {1}, a2 = {2}".format(*fit.popt)
The parameters are: a0 = 0.142870141922, a1 = 1.33420587099, a2 = 4.27241667343
>>> yfitted = fit(x)
```

The `fit` object, beside being a callable object to evaluate the fitting function as some points, contain the following properties:

fct Function being fitted (e.g. the one given as argument)

popt Optimal parameters for the function

res Residuals of the fitted data

pcov Covariance of the parameters around the optimal values.

infodict Additional estimation outputs, as given by `scipy.optimize.leastsq()`

3.2.1 Fitting analysis

PyQt-Fit also has tools to evaluate your fitting. You can use them as a whole:

```
>>> from pyqt_fit import plot_fit
>>> result = plot_fit.fit_evaluation(fit, x, y,
...                                  fct_desc = "$y = a_0 + a_1 x + a_2 x^2$",
...                                  param_names=['a_0', 'a_1', 'a_2'])
```

You can then examine the `result` variable. But you can also perform only the analysis you need. For example, you can compute the data needed for the residual analysis with:

```
>>> rm = plot_fit.residual_measures(fit.res)
```

`rm` is a named tuple with the following fields:

scaled_res Scaled residuals, sorted in ascending values for residuals. The scaled residuals are computed as $sr_i = \frac{r_i}{\sigma_r}$, where σ_r is the variance of the residuals.

res_IX Ordering indices for the residuals in scaled_res. This orders the residuals in an ascending manner.

prob List of quantiles used to compute the normalized quantiles.

normq Value expected for the quantiles in `prob` if the distribution is normal. The foluma is: $\Phi(p) = \sqrt{2}\,\mathrm{erf}^{-1}(2p-1), p \in [0;1]$

3.2.2 Plotting the results

At last, you can use the display used for the GUI:

```
>>> handles = plot_fit.plot1d(result)
```

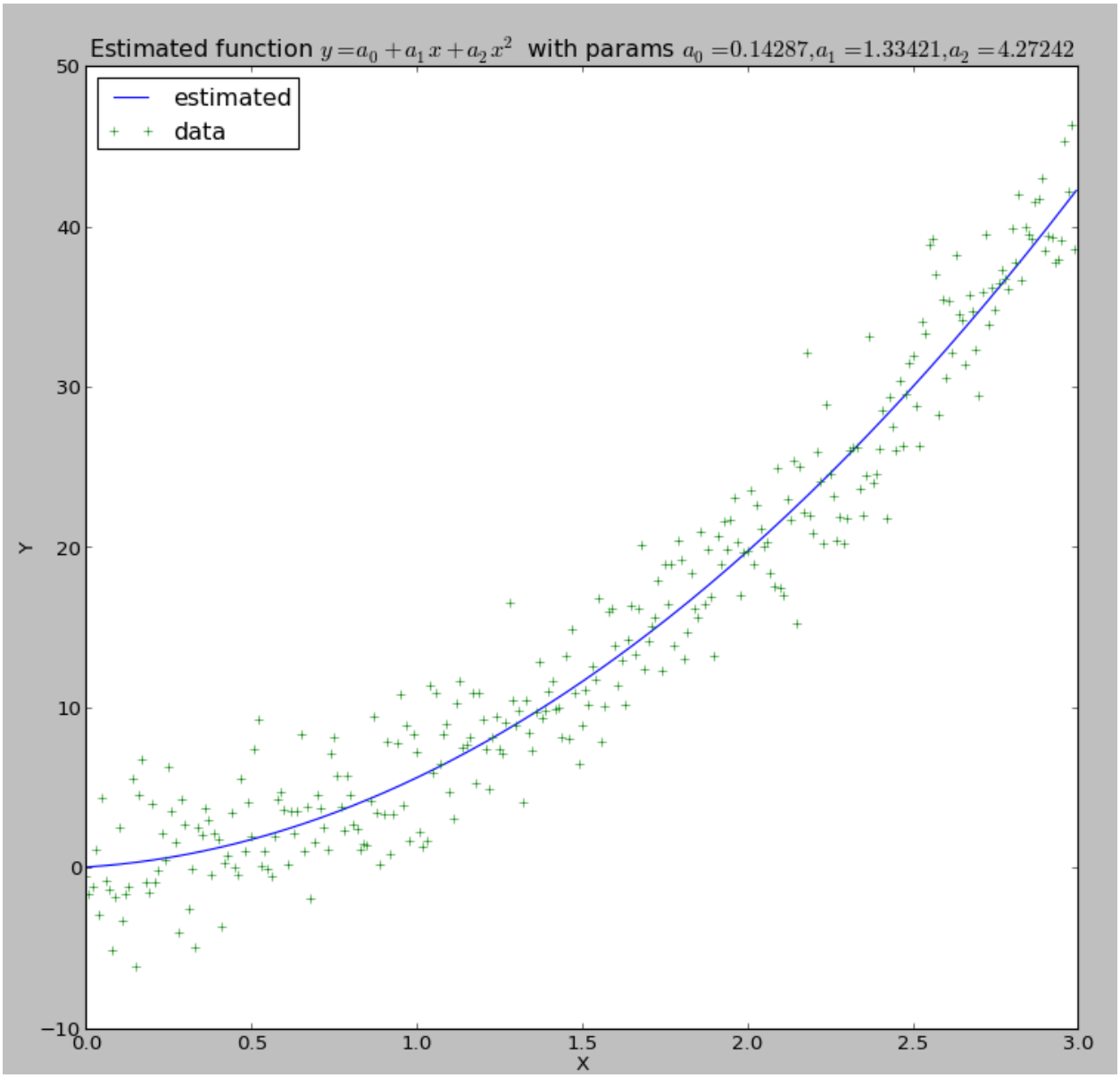


Figure 3.2: Curve fitting output

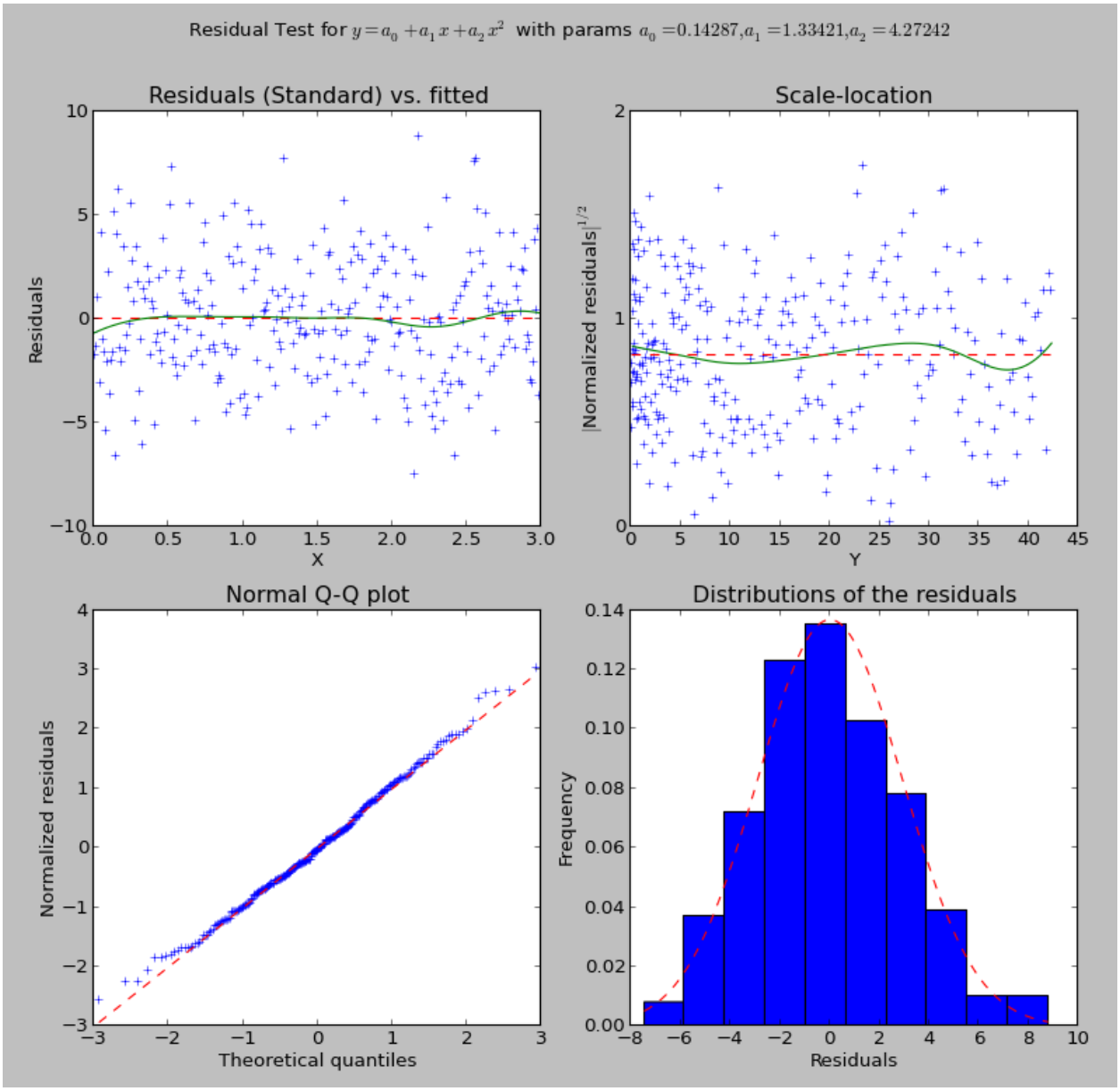


Figure 3.3: Residual checking output

What you will obtain are these two graphs:

Do not hesitate to look at the code for `pyqt_fit.plot_fit.plot1d()` to examine how things are plotted. The function should return all the handles you may need to tune the presentation of the various curves.

3.2.3 Speeding up the fitting: providing the jacobian

The least-square algorithm uses the jacobian (i.e. the derivative of the function with respect to each parameter on each point). By default, the jacobian is estimated numerically, which can be quite expensive (if the function itself is). But in many cases, it is fairly easy to compute. For example, in our case we have:

$$\begin{aligned}\frac{\partial f(x)}{\partial a_0} &= 1\\ \frac{\partial f(x)}{\partial a_1} &= x\\ \frac{\partial f(x)}{\partial a_2} &= x^2\end{aligned}$$

By default, the derivatives should be given in columns (i.e. each line correspond to a parameter, each column to a point):

```
>>> def df(params, x):
...   result = np.ones((3, x.shape[0]), dtype=float)
...   result[1] = x
...   result[2] = x**2
...   return result # result[0] is already 1
>>> fit.Dfun = df
>>> fit.fit()
```

Of course there is no change in the result, but it should be slightly faster (note that in this case, the function is so fast that to make it worth it, you need a lot of points as input).

3.3 Confidence Intervals

PyQt-Fit provides bootstrapping methods to compute confidence intervals. Bootstrapping is a method to estimate confidence interval and probability distribution by resampling the data provided. For our problem, we will call:

```
>>> import pyqt_fit.bootstrap as bs
>>> xs = np.arange(0, 3, 0.01)
>>> result = bs.bootstrap(pyqt_fit.CurveFitting, x, y, eval_points = xs, fit_kwrds = dict(p0 = (0,1,(
```

This will compute the 95% and 99% confidence intervals for the curves and for the optimised parameters (`popt`). The result is a named tuple `pyqt_fit.bootstrap.BootstrapResult`. The most important field are `y_est` and `CIs` that provide the estimated values and the confidence intervals for the curve and for the parameters.

On the data, the result can be plotted with:

```
>>> plt.plot(xs, result.y_fit(xs), 'r', label="Fitted curve")
>>> plt.plot(xs, result.CIs[0][0,0], 'g--', label='95% CI')
>>> plt.plot(xs, result.CIs[0][0,1], 'g--')
>>> plt.fill_between(xs, result.CIs[0][0,0], result.CIs[0][0,1], color='g', alpha=0.25)
>>> plt.legend(loc=0)
```

The result is:

The bounds for the parameters are obtained with:

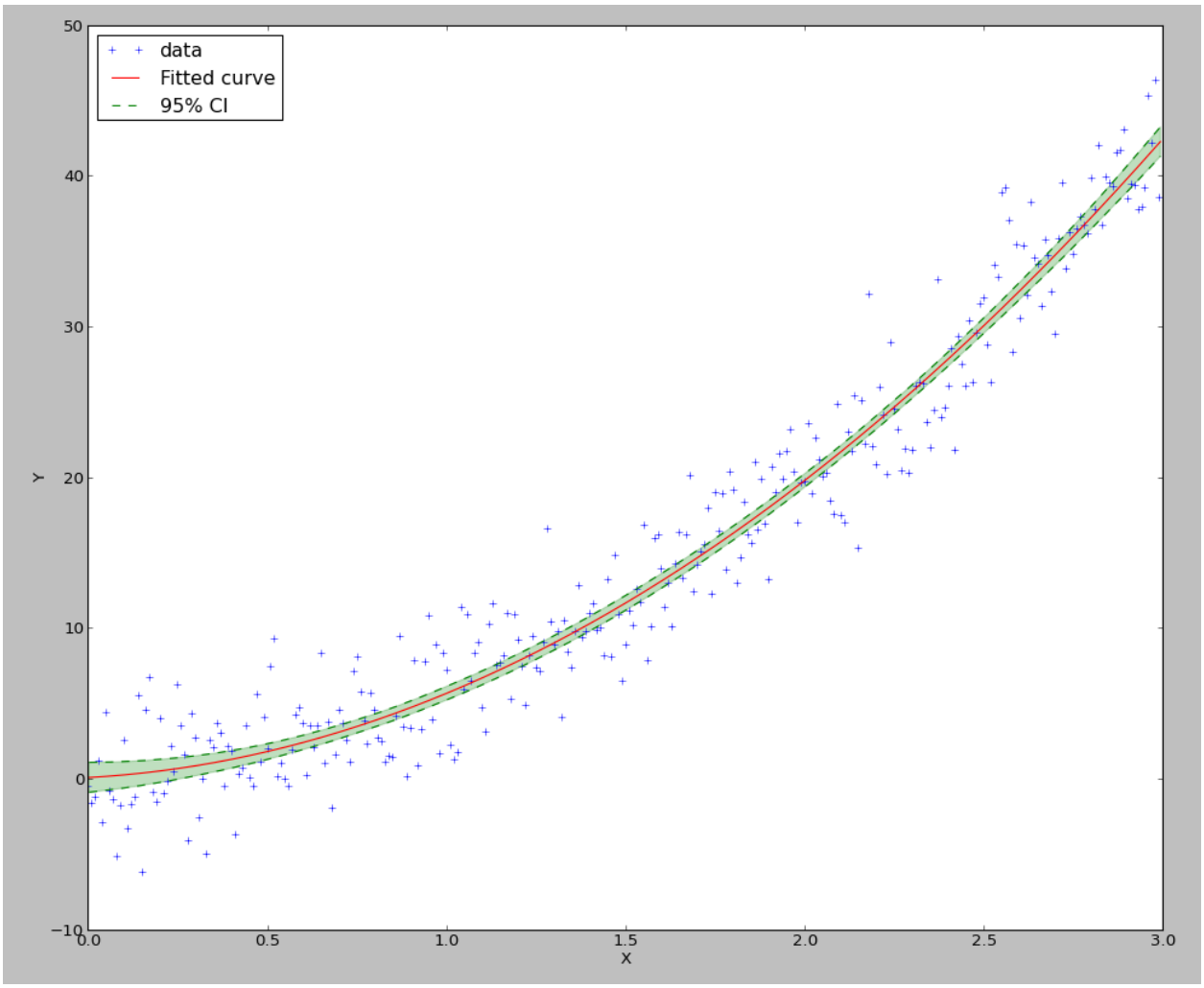


Figure 3.4: Drawing of the 95% confidence interval

```
>>> print "95% CI for p0 = {}-{}".format(*result.CIs[1][0])
>>> print "99% CI for p0 = {}-{}".format(*result.CIs[1][1])
95% CI for p0 = [-0.84216998 -0.20389559  3.77950689]-[ 1.14753806  2.8848943   4.7557855 ]
99% CI for p0 = [-1.09413524 -0.62373955  3.64217184]-[ 1.40142123  3.32762714  4.91391328]
```

It is also possible to obtain the full distribution of the values for the curve and for the parameters by providing the argument `full_results=True` and by looking at `result.full_results`.

3.4 Defining the functions and residuals

3.4.1 User-defined function

The function must be a two argument python function:

1. the parameters of the function, provided either as a tuple or a ndarray
2. the values on which the function is to be evaluated, provided as a single value or a ndarray

If the second argument is a ndarray of shape `(...,N)`, the output must be a ndarray of shape `(N,)`.

If is also possible to provide the function computing the Jacobian of the estimation function. The arguments are the same as for the function, but the shape of the output must be `(P,N)`, where `P` is the number of parameters to be fitted, unless the option `col_deriv` is set to 0, in which case the shape of the output must be `(N,P)`.

3.4.2 User-defined residuals

It is also possible to redefine the notion of residuals. A common example is to use the log of the residuals. It is most applicable if the standard deviation of the residuals is proportional to the fitted quantity. The residual should be a function of two arguments:

1. the measured data
2. the fitted data

For example, the log residuals would be:

```
>>> def log_residuals(y1, y0):
...   return np.log(y1/y0)
```

As for the user-defined function, it is possible to provide the jacobian of the residuals. It must be provided as a function of 3 arguments:

1. the measured data
2. the fitted data
3. the jacobian of the function on the fitted data

The shape of the output must be the same as the shape of the jacobian of the function. For example, if `col_deriv` is set to True, the jacobian of the log-residuals will be defined as:

```
>>> def Dlog_residual(y1, y0):
...   return -1/y0[np.newaxis,:]
```

This is because:

$$\mathcal{J}\left(\log\frac{y_1}{y_0}\right) = -\frac{\mathcal{J}(y_0)}{y_0}$$

as y_1 is a constant, and y_0 depend on the parameters.

Also, methods like the residuals bootstrapping will require a way to apply residuals on fitted data. For this, you will need to provide a function such as:

```
>>> def invert_log_residuals(y, res):
...   return y*np.exp(res)
```

This function should be such that this expression returns always true:

```
>>> all(log_residuals(invert_log_residuals(y, res), y) == res)
```

Of course, working with floating point values, this is usually not happening. So a better test function would be:

```
>>> sum((log_residuals(invert_log_residuals(y, res), y) - res)**2) < epsilon
```

3.5 Using the functions/residuals defined for the GUI

It is also possible to use the functions and residuals defined for the GUI. The interface for this are via the modules `pyqt_fit.functions` and `pyqt_fit.residuals`.

The list of available functions can be retrieved with:

```
>>> pyqt_fit.functions.names()
['Power law', 'Exponential', 'Linear', 'Logistic']
```

And a function is retrieved with:

```
>>> f = pyqt_fit.functions.get('Logistic')
```

The function is an object with the following properties:

`__call__` Evaluate the function on a set of points, as described in the previous section

`Dfun` Evaluate the jacobian of the function. If not available, this property is set to `None`

`args` Name of the arguments

`description` Formula or description of the evaluated function

`init_args` Function provided a reasonnable first guess for the parameters. Should be called with `f.init_args(x,y)`.

In the same way, the list of available residuals can be retrieved with:

```
>>> pyqt_fit.residuals.names()
['Difference of the logs', 'Standard']
```

And a residuals function is retrieved with:

```
>>> r = pyqt_fit.residuals.get('Difference of the logs')
```

The residuals is an object with the following properties:

`__call__` Evaluate the residuals, as described in the previous section

`Dfun` Evaluate the jacobian of the residuals. If not available, this property is set to `None`

`invert` Function that apply the residuals to a set of fitted data. It will be called as `r.invert(y, res)`. It should have the properties of the invert function described in the previous section.

`description` Description of the kind of residuals

name Name of the residuals.

CHAPTER 4

Non-Parametric regression tutorial

4.1 Introduction

In general, given a set of observations (x_i, y_i), with $x_i = (x_{i1}, \ldots, x_{ip})^T \in \mathbb{R}^p$. We assume there exists a function $f(x)$ such that:

$$y_i = f(x_i) + \epsilon_i$$

with $\epsilon_i \in \mathbb{R}$ such that $E(\epsilon) = 0$. This function, however, is not accessible. So we will consider the function $\hat{f}$ such that:

$$\hat{f}(x) = \text{argmin}_f \left(y_i - f(x_i)\right)^2$$

The various methods presented here consists in numerical approximations finding the minimum in a part of the function space. The most general method offered by this module is called the local-polynomial smoother. It uses the Taylor-decomposition of the function f on each point, and a local weigthing of the points, to find the values. The function is then defined as:

$$\hat{f}_n(x) = \text{argmin}_{a_0} \sum_i K\left(\frac{x - x_i}{h}\right) \left(y_i - \mathcal{P}_n(x_i)\right)^2$$

Where $\mathcal{P}_n$ is a polynomial of order n whose constant term is a_0, K is a kernel used for weighing the values and h is the selected bandwidth. In particular, in 1D:

$$\hat{f}_n(x) = \text{argmin}_{a_0} \sum_i K\left(\frac{x - x_i}{h}\right) \left(y_i - a_0 - a_1(x - x_i) - \ldots - a_n \frac{(x - x_i)^n}{n!}\right)^2$$

In general, higher polynomials will reduce the error term but will overfit the data, in particular at the boundaries.

4.2 A simple example

For our example, lets first degine our target function:

```
>>> import numpy as np
>>> def f(x):
...     return 3*np.cos(x/2) + x**2/5 + 3
```

Then, we will generate our data:

```
>>> xs = np.random.rand(200) * 10
>>> ys = f(xs) + 2*np.random.randn(*xs.shape)
```

We can then visualize the data:

```
>>> import matplotlib.pyplot as plt
>>> grid = np.r_[0:10:512j]
>>> plt.plot(grid, f(grid), 'r--', label='Reference')
>>> plt.plot(xs, ys, 'o', alpha=0.5, label='Data')
>>> plt.legend(loc='best')
```

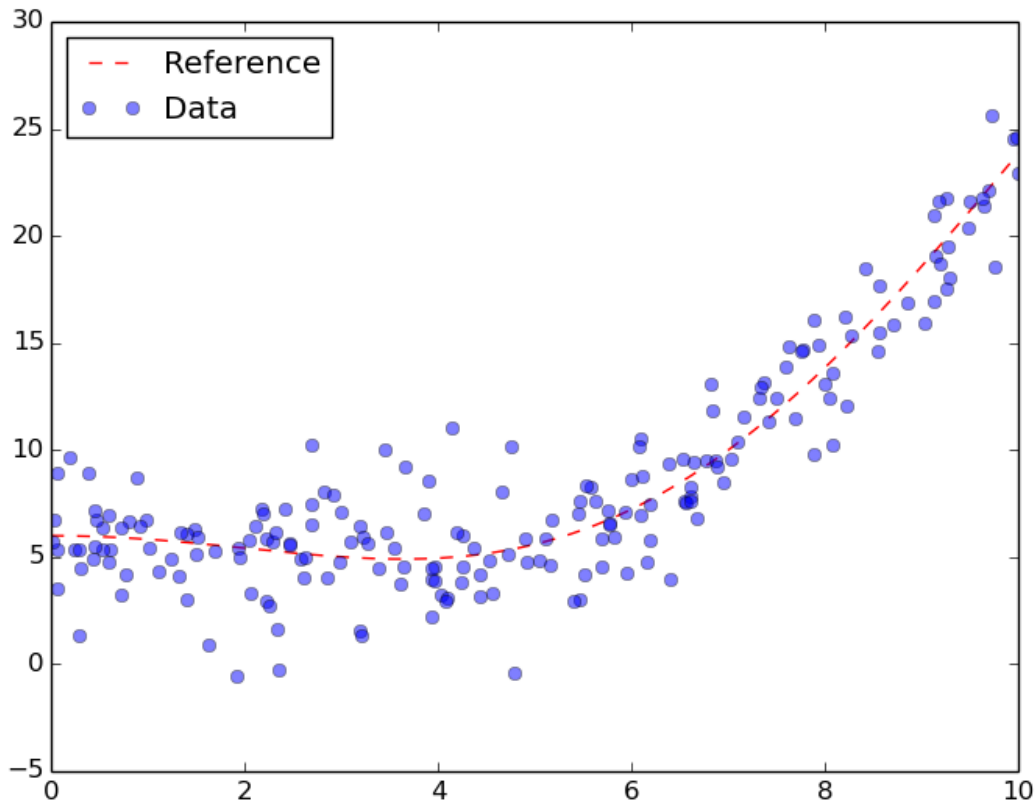


Figure 4.1: Generated data with generative function.

At first, we will try to use a simple Nadaraya-Watson method, or spatial averaging, using a gaussian kernel:

```
>>> import pyqt_fit.nonparam_regression as smooth
>>> from pyqt_fit import npr_methods
>>> k0 = smooth.NonParamRegression(xs, ys, method=npr_methods.SpatialAverage())
>>> k0.fit()
>>> plt.plot(grid, k0(grid), label="Spatial Averaging", linewidth=2)
>>> plt.legend(loc='best')
```

As always during regressionm we need to look at the residuals:

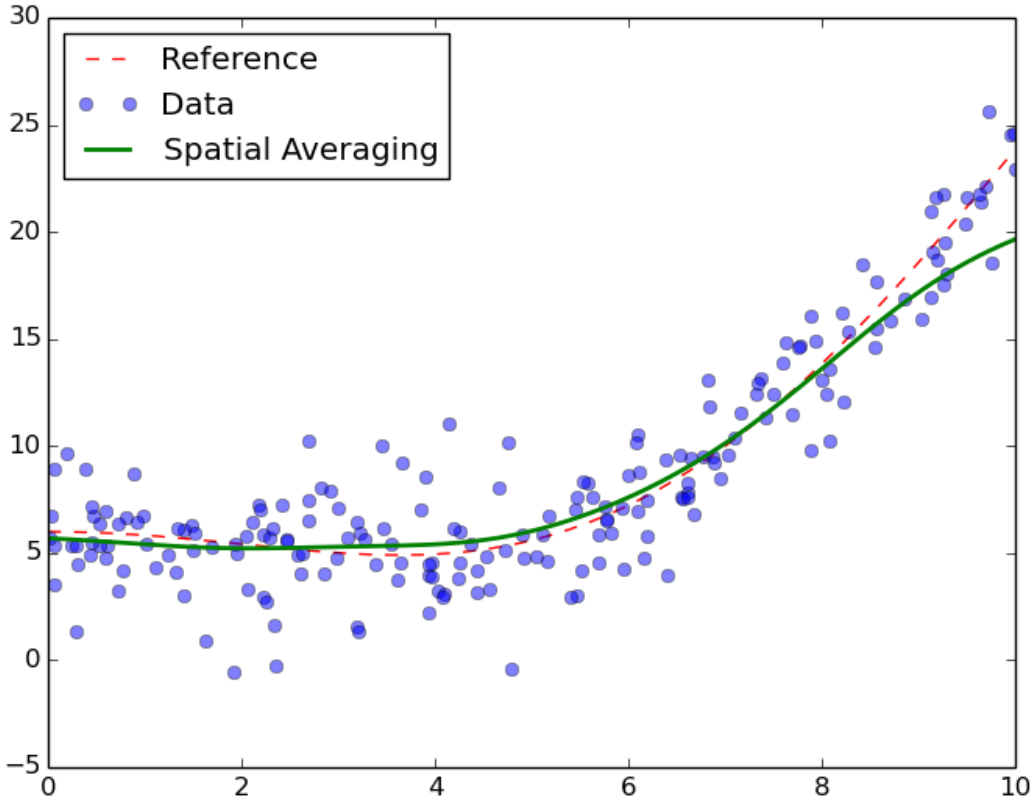


Figure 4.2: Result of the spatial averaging.

```
>>> from pyqt_fit import plot_fit
>>> yopts = k0(xs)
>>> res = ys - yopts
>>> plot_fit.plot_residual_tests(xs, yopts, res, 'Spatial Average')
```

We can see from the data that the inside of the curve is well-fitted. However, the boundaries are not. This is extremely visible on the right boundary, where the data is clearly under-fitted. This is a typical problem with spatial averaging, as it doesn't cope well with strong maxima, especially on the boundaries. As an improvement, we can try local-linear or local-polynomial. The process is exactly the same:

```
>>> k1 = smooth.NonParamRegression(xs, ys, method=npr_methods.LocalPolynomialKernel(q=1))
>>> k2 = smooth.NonParamRegression(xs, ys, method=npr_methods.LocalPolynomialKernel(q=2))
>>> k3 = smooth.NonParamRegression(xs, ys, method=npr_methods.LocalPolynomialKernel(q=3))
>>> k12 = smooth.NonParamRegression(xs, ys, method=npr_methods.LocalPolynomialKernel(q=12))
>>> k1.fit(); k2.fit(); k3.fit(); k12.fit()
>>> plt.figure()
>>> plt.plot(xs, ys, 'o', alpha=0.5, label='Data')
>>> plt.plot(grid, k12(grid), 'b', label='polynom order 12', linewidth=2)
>>> plt.plot(grid, k3(grid), 'y', label='cubic', linewidth=2)
>>> plt.plot(grid, k2(grid), 'k', label='quadratic', linewidth=2)
>>> plt.plot(grid, k1(grid), 'g', label='linear', linewidth=2)
>>> plt.plot(grid, f(grid), 'r--', label='Target', linewidth=2)
>>> plt.legend(loc='best')
```

In this example, we can see that linear, quadratic and cubic give very similar result, while a polynom of order 12 is clearly over-fitting the data. Looking closer at the data, we can see that the quadratic and cubic fits seem to be better adapted, as quadratic and cubic both seem to over-fit the data. Note that this is not to be generalise and is very dependent on the data you have! We can now redo the residual plots:

```
>>> yopts = k1(xs)
>>> res = ys - yopts
>>> plot_fit.plot_residual_tests(xs, yopts, res, 'Local Linear')
```

We can also look at the residuals for the quadratic polynomial:

```
>>> yopts = k2(xs)
>>> res = ys - yopts
>>> plot_fit.plot_residual_tests(xs, yopts, res, 'Local Quadratic')
```

We can see from the structure of the noise that the quadratic curve seems indeed to fit much better the data. Unlike in the local linear regression, we do not have significant bias along the X axis. Also, the residuals seem "more normal" (i.e. the points in the QQ-plot are better aligned) than in the linear case.

4.3 Confidence Intervals

Confidence intervals can be computed using bootstrapping. Based on the previous paragraph, you can get confidence interval on the estimation with:

```
>>> import pyqt_fit.bootstrap as bs
>>> grid = np.r_[0:10:512j]
>>> def fit(xs, ys):
...   est = smooth.NonParamRegression(xs, ys, method=npr_methods.LocalPolynomialKernel(q=2))
...   est.fit()
...   return est
>>> result = bs.bootstrap(fit, xs, ys, eval_points = grid, CI = (95,99))
```

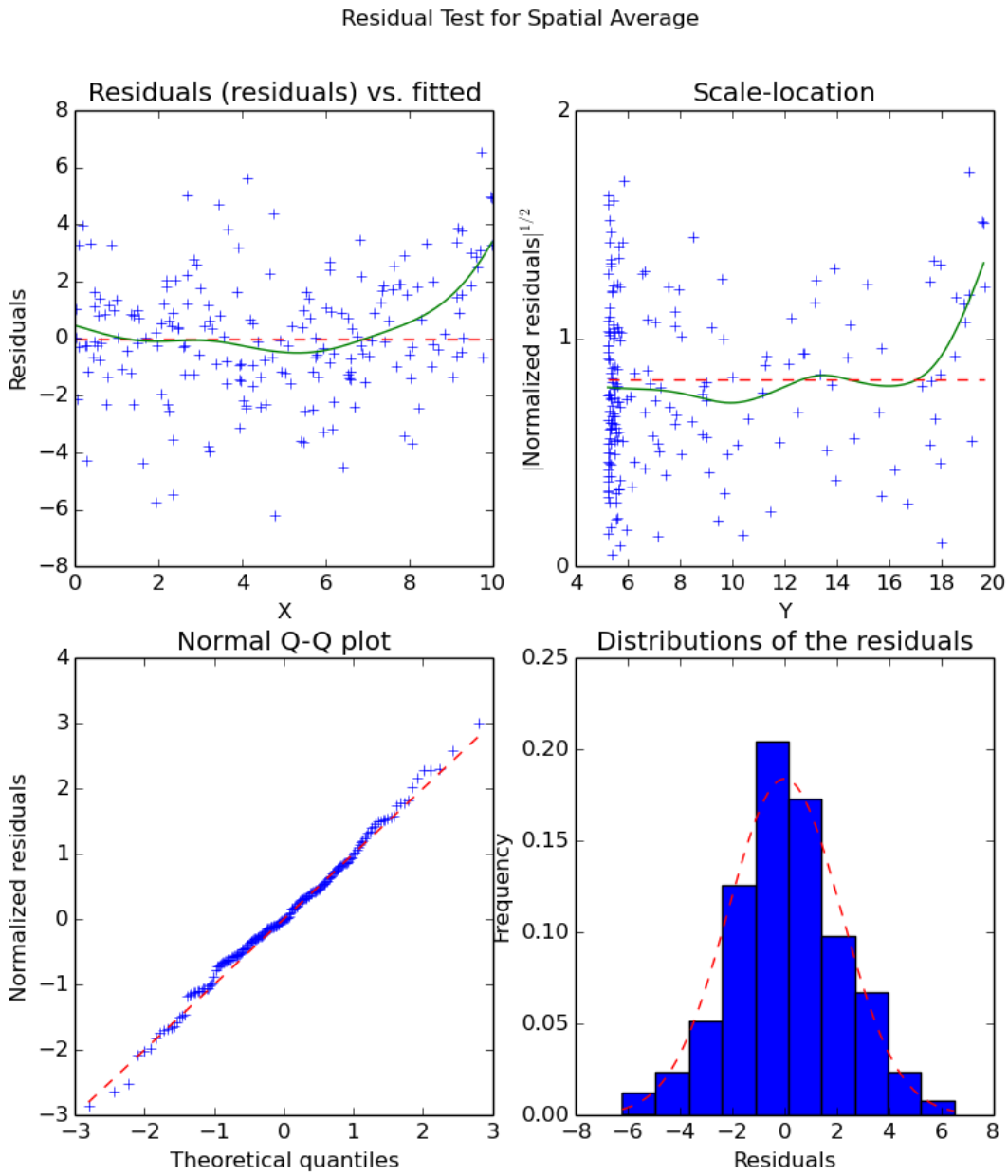


Figure 4.3: Residuals of the Spatial Averaging regression

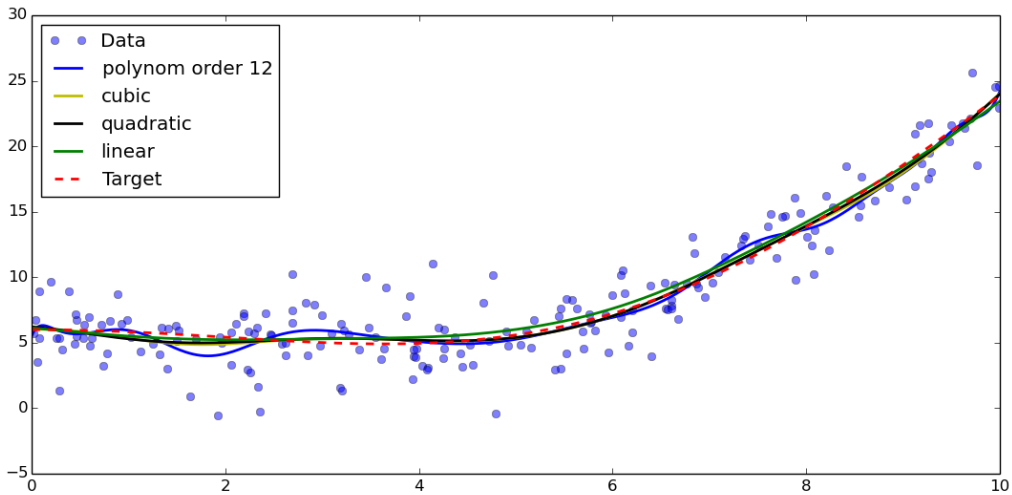


Figure 4.4: Result of polynomial fitting with orders 1, 2, 3 and 12

This will compute the 95% and 99% confidence intervals for the quadratic fitting. The result is a named tuple `pyqt_fit.bootstrap.BootstrapResult`. The most important field are `y_est` and `CIs` that provide the estimated values and the confidence intervals for the curve.

The data can be plotted with:

```
>>> plt.plot(xs, ys, 'o', alpha=0.5, label='Data')
>>> plt.plot(grid, result.y_fit(grid), 'r', label="Fitted curve", linewidth=2)
>>> plt.plot(grid, result.CIs[0][0,0], 'g--', label='95% CI', linewidth=2)
>>> plt.plot(grid, result.CIs[0][0,1], 'g--', linewidth=2)
>>> plt.fill_between(grid, result.CIs[0][0,0], result.CIs[0][0,1], color='g', alpha=0.25)
>>> plt.legend(loc=0)
```

4.4 Types of Regressions

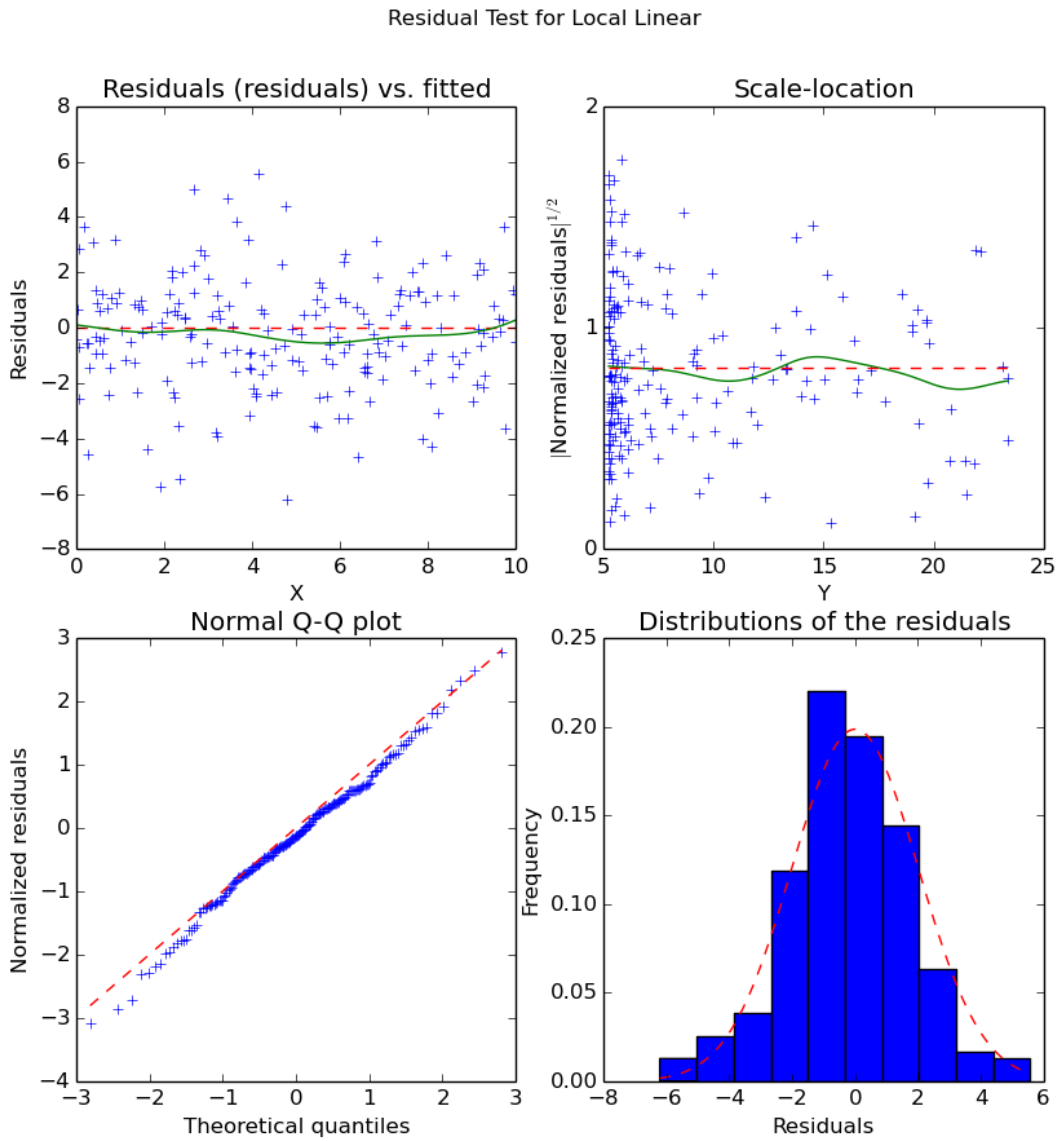


Figure 4.5: Residuals of the Local Linear Regression

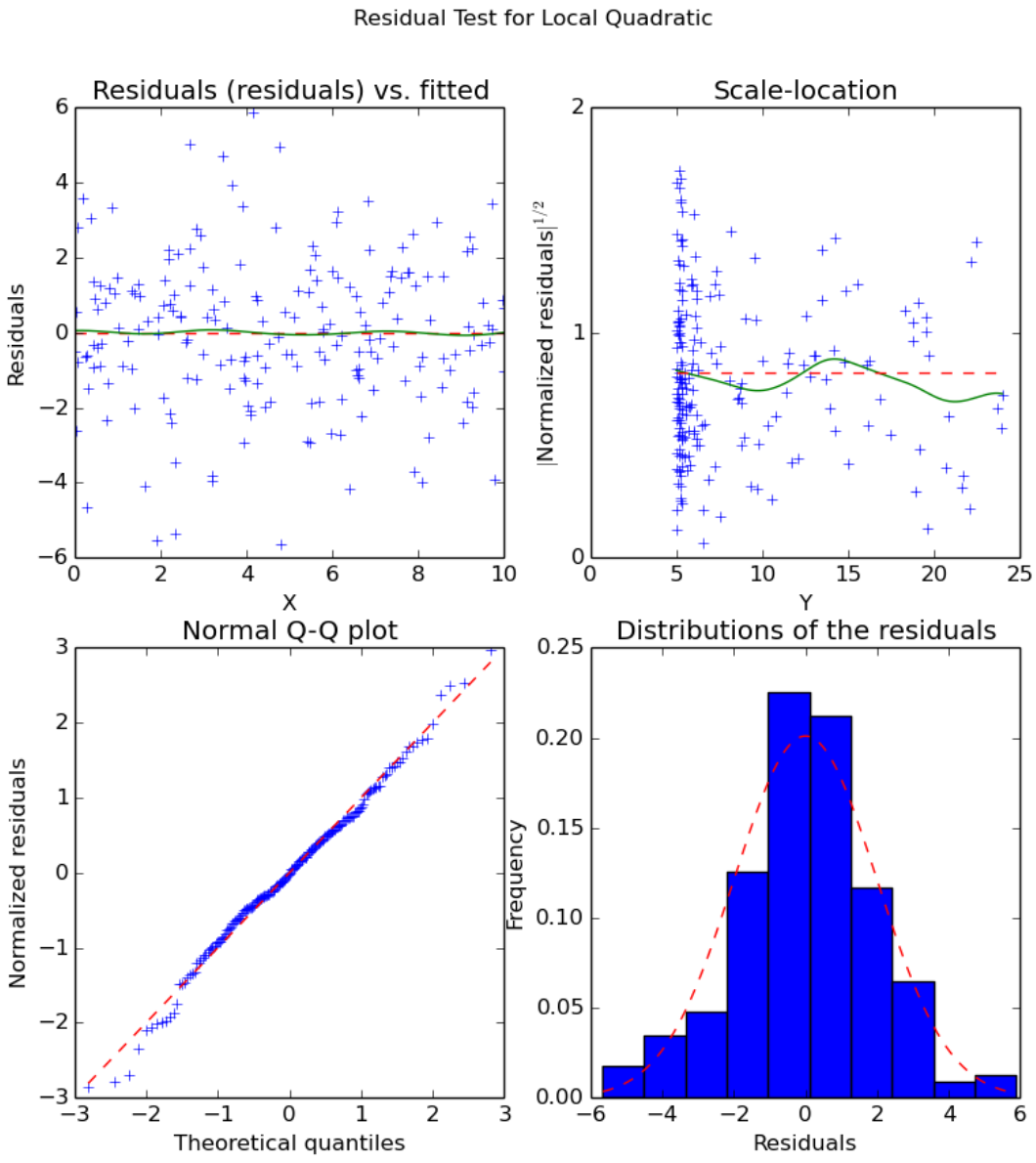


Figure 4.6: Residuals of the Local Quadratic Regression

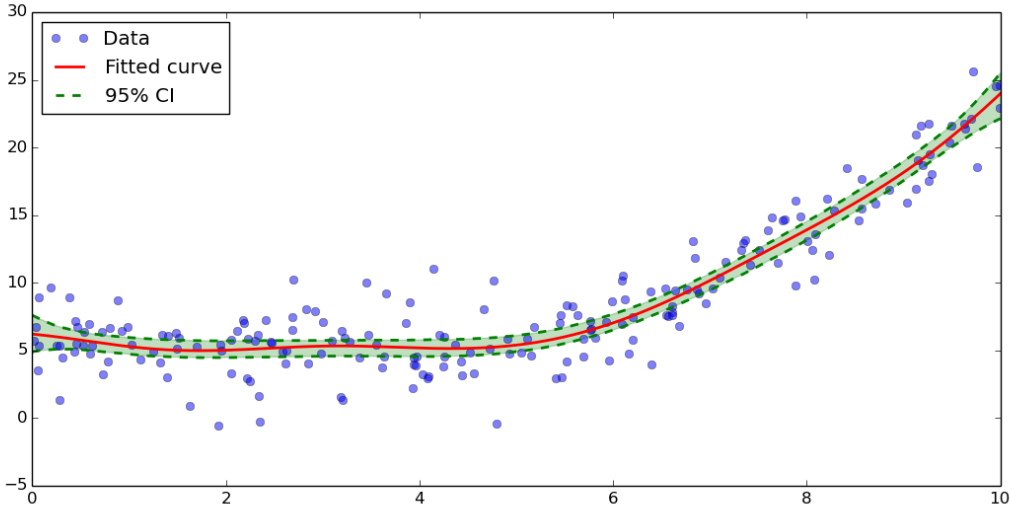


Figure 4.7: Confidence intervals

CHAPTER 5

Kernel Density Estimation tutorial

5.1 Introduction

Kernel Density Estimation is a method to estimate the frequency of a given value given a random sample.

Given a set of observations $(x_i)_{1 \leq i \leq n}$. We assume the observations are a random sampling of a probability distribution f. We first consider the kernel estimator:

$$\hat{f}(x) = \frac{1}{Wnh} \sum_{i=1}^{n} \frac{w_i}{\lambda_i} K\left(\frac{x_i - x}{h\lambda_i}\right)$$

Where:

1. $K : \mathbb{R}^p \to \mathbb{R}$ is the kernel, a function centered on 0 and that integrates to 1;
2. math:*h* is the bandwidth, a smoothing parameter that would typically tend to 0 when the number of samples tend to ∞;
3. (w_i) are the weights of each of the points, and W is the sum of the weigths;
4. (λ_i) are the adaptation factor of the kernel.

Also, it is desirable if the second moment of the kernel (i.e. the variance) is 1 for the bandwidth to keep a uniform meaning across the kernels.

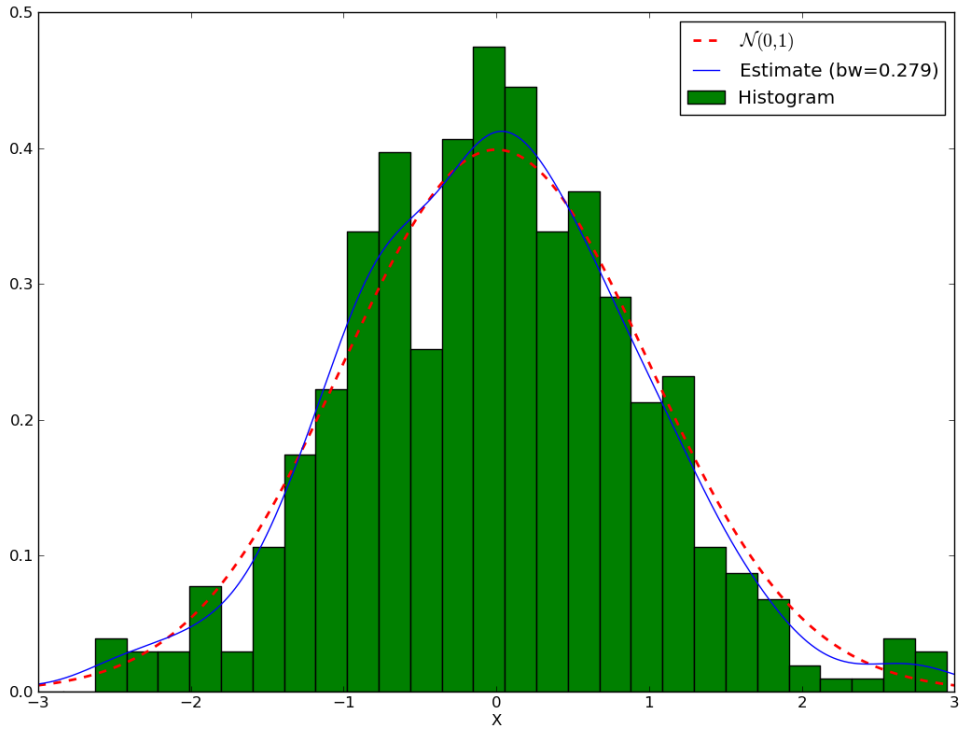
5.2 A simple example

First, let's assume we have a random variable following a normal law $\mathcal{N}(0, 1)$, and let's plot its histogram:

```
>>> import numpy as np
>>> from scipy.stats import norm
>>> from matplotlib import pylab as plt
>>> f = norm(loc=0, scale=1)
>>> x = f.rvs(500)
>>> xs = np.r_[-3:3:1024j]
>>> ys = f.pdf(xs)
>>> h = plt.hist(x, bins=30, normed=True, color=(0,.5,0,1), label='Histogram')
>>> plt.plot(xs, ys, 'r--', linewidth=2, label='$\mathcal{N}(0,1)$')
>>> plt.xlim(-3,3)
>>> plt.xlabel('X')
```

We can get estimate the density with:

```
>>> from pyqt_fit import kde
>>> est = kde.KDE1D(x)
>>> plot(xs, est(xs), label='Estimate (bw={:.3g})'.format(est.bandwidth))
>>> plt.legend(loc='best')
```



You may wonder why use KDE rather than a histogram. Let's test the variability of both method. To that purpose, let first generate a set of a thousand datasets and the corresponding histograms and KDE, making sure the width of the KDE and the histogram are the same:

```
>>> import numpy as np
>>> from scipy.stats import norm
>>> from pyqt_fit import kde
>>> f = norm(loc=0, scale=1)
>>> xs = np.r_[-3:3:1024j]
>>> nbins = 20
>>> x = f.rvs(1000*1000).reshape(1000,1000)
>>> hs = np.empty((1000, nbins), dtype=float)
>>> kdes = np.empty((1000, 1024), dtype=float)
>>> hs[0], edges = np.histogram(x[0], bins=nbins, range=(-3,3), density=True)
>>> mod = kde.KDE1D(x[0])
>>> mod.fit()  # Force estimation of parameters
>>> mod.bandwidth = mod.bandwidth  # Prevent future recalculation
>>> kdes[0] = mod(xs)
>>> for i in range(1, 1000):
>>>   hs[i] = np.histogram(x[i], bins=nbins, range=(-3,3), density=True)[0]
>>>   mod.xdata = x[i]
>>>   kdes[i] = mod(xs)
```

Now, let's find the mean and the 90% confidence interval:

```
>>> h_mean = hs.mean(axis=0)
>>> h_ci = np.array(np.percentile(hs, (5, 95), axis=0))
>>> h_err = np.empty(h_ci.shape, dtype=float)
>>> h_err[0] = h_mean - h_ci[0]
>>> h_err[1] = h_ci[1] - h_mean
>>> kde_mean = kdes.mean(axis=0)
>>> kde_ci = np.array(np.percentile(kdes, (5, 95), axis=0))
>>> width = edges[1:]-edges[:-1]
>>> fig = plt.figure()
>>> ax1 = fig.add_subplot(1,2,1)
>>> ax1.bar(edges[:-1], h_mean, yerr=h_err, width = width, label='Histogram',
...         facecolor='g', edgecolor='k', ecolor='b')
>>> ax1.plot(xs, f.pdf(xs), 'r--', lw=2, label='$\mathcal{N}(0,1)$')
>>> ax1.set_xlabel('X')
>>> ax1.set_xlim(-3,3)
>>> ax1.legend(loc='best')
>>> ax2 = fig.add_subplot(1,2,2)
>>> ax2.fill_between(xs, kde_ci[0], kde_ci[1], color=(0,1,0,.5), edgecolor=(0,.4,0,1))
>>> ax2.plot(xs, kde_mean, 'k', label='KDE (bw = {:.3g})'.format(mod.bandwidth))
>>> ax2.plot(xs, f.pdf(xs), 'r--', lw=2, label='$\mathcal{N}(0,1)$')
>>> ax2.set_xlabel('X')
>>> ax2.legend(loc='best')
>>> ymax = max(ax1.get_ylim()[1], ax2.get_ylim()[1])
>>> ax2.set_ylim(0, ymax)
>>> ax1.set_ylim(0, ymax)
>>> ax1.set_title('Histogram, max variation = {:.3g}'.format((h_ci[1] - h_ci[0]).max()))
>>> ax2.set_title('KDE, max variation = {:.3g}'.format((kde_ci[1] - kde_ci[0]).max()))
>>> fig.suptitle('Comparison Histogram vs. KDE')
```

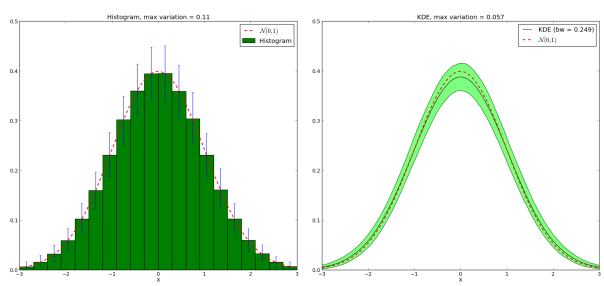


Figure 5.1: Comparison Histogram / KDE – KDE has less variability

Note that the KDE doesn't tend toward the true density. Instead, given a kernel K, the mean value will be the convolution of the true density with the kernel. But for that price, we get a much narrower variation on the values. We also avoid boundaries issues linked with the choices of where the bars of the histogram start and stop.

5.3 Boundary Conditions

5.3.1 Simple Boundary

One of the main focus of the implementation is the estimation of density on bounded domain. As an example, let's try to estimate the KDE of a dataset following a χ^2_2 distribution. As a reminder, the PDF of this distribution is:

$$\chi^2_2(x) = \frac{1}{2}e^{-\frac{x}{2}}$$

This distribution is only defined for $x > 0$. So first let's look at the histogram and the default KDE:

```
>>> from scipy import stats
>>> from matplotlib import pylab as plt
>>> from pyqt_fit import kde, kde_methods
>>> import numpy as np
>>> chi2 = stats.chi2(2)
>>> x = chi2.rvs(1000)
>>> plt.hist(x, bins=20, range=(0,8), color=(0,.5,0), label='Histogram', normed=True)
>>> est = kde.KDE1D(x)
>>> xs = np.r_[0:8:1024j]
>>> plt.plot(xs, est(xs), label='KDE (bw = {:.3g})'.format(est.bandwidth))
>>> plt.plot(xs, chi2.pdf(xs), 'r--', lw=2, label=r'$\chi^2_2$')
>>> plt.legend(loc='best')
```

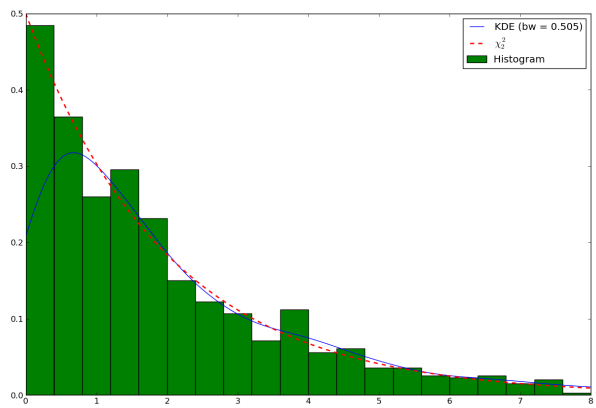


Figure 5.2: Standard estimation of the χ^2_2 distribution

We can see that the estimation is correct far from the 0, but when closer than twice the bandwidth, the estimation becomes incorrect. The reason is that the method "sees" there are no points below 0, and therefore assumes the density continuously decreases to reach 0 in slightly negative values. Moreover, if we integrate the KDE in the domain $[0, \infty]$:

```
>>> from scipy import integrate
>>> integrate.quad(est, 0, np.inf)
(0.9138087148449997, 2.7788548831933142e-09)
```

we can see the distribution sums up only to about 0.91, instead of 1. In short, we are "loosing weight".

There are a number of ways to take into account the bounded nature of the distribution and correct with this loss. A common one consists in truncating the kernel if it goes below 0. This is called "renormalizing" the kernel. The method can be specified setting the `method` attribute of the KDE object to `pyqt_fit.kde_methods.renormalization`:

```
>>> est_ren = kde.KDE1D(x, lower=0, method=kde_methods.renormalization)
>>> plt.plot(xs, est_ren(xs), 'm', label=est_ren.method.name)
>>> plt.legend(loc='best')
```

It can be shown that the convergence at the boundary with the renormalization method is slower than in the rest of the dataset. Another method is a linear approximation of the density toward the boundaries. The method, being an approximation, will not sum up to exactly 1. However, it often approximate the density much better:

```
>>> from pyqt_fit import kde_methods
>>> est_lin = kde.KDE1D(x, lower=0, method=kde_methods.linear_combination)
```

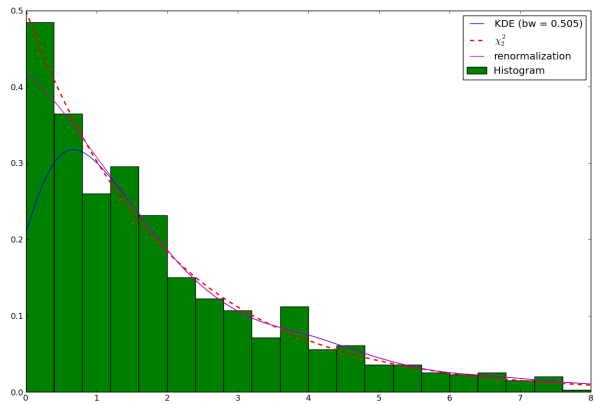


Figure 5.3: Renormalized estimation of the χ^2_2 distribution

```
>>> plt.plot(xs, est_lin(xs), 'y', label=est_lin.method.name)
>>> plt.legend(loc='best')
```

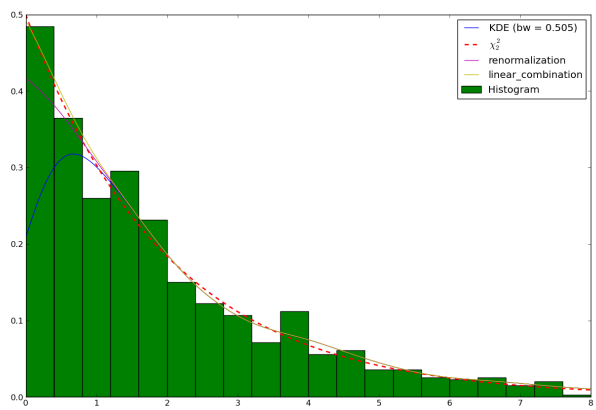


Figure 5.4: Linear combination estimation of the χ^2_2 distribution

5.3.2 Reflective Boundary

Sometimes, not only do we have a boundary, but we expect the density to be reflective, that is the derivative on the boundary is 0, we expect the data to behave the same as being repeated by reflection on the boundaries. An example is the distribution of the distance from a 2D point taken from a 2D gaussian distribution to the center:

$$Z = |X - Y| \qquad X \sim \mathcal{N}(0,1), Y \sim \mathcal{N}(0,1)$$

First, let's look at the histogram:

```
>>> from scipy import stats, integrate
>>> from matplotlib import pylab as plt
>>> from pyqt_fit import kde, kde_methods
>>> import numpy as np
>>> f = stats.norm(loc=0, scale=1)
>>> x = f.rvs(1000)
>>> y = f.rvs(1000)
```

```
>>> z = np.abs(x-y)
>>> plt.hist(z, bins=20, facecolor=(0,.5,0), normed=True)
```

Then, the KDE assume reflexive boundary conditions:

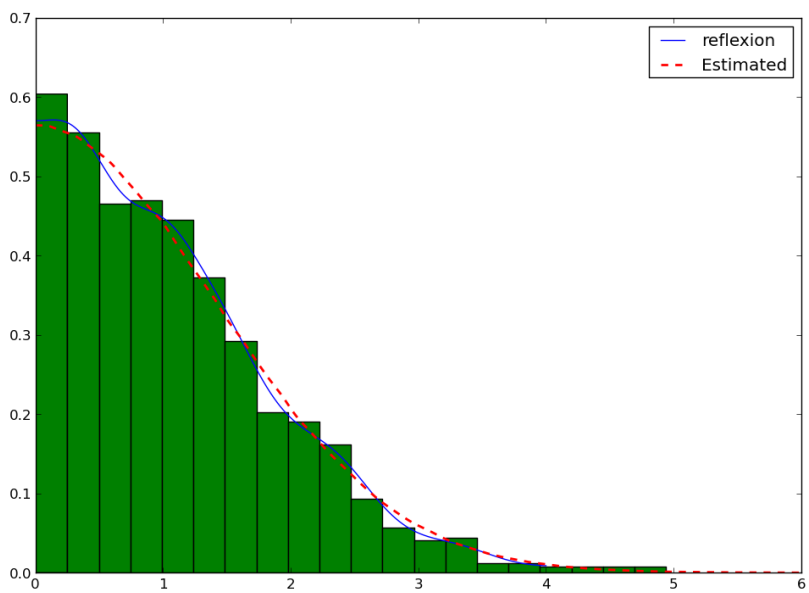
```
>>> xs = np.r_[0:8:1024j]
>>> est = kde.KDE1D(z, lower=0, method=kde_methods.reflection)
>>> plot(xs, est(xs), color='b', label=est.method.name)
```

To estimate the "real" distribution, we will increase the number of samples:

```
>>> xx = f.rvs(1000000)
>>> yy = f.rvs(1000000)
>>> zz = np.abs(xx-yy)
```

If you try to estimate the KDE, it will now be very slow. To speed up the process, you can use the `grid` method. The `grid` method will compute the result using DCT or FFT if possible. It will work only if you don't have variable bandwidth and boundary conditions are either reflexive, cyclic, or non-existent (i.e. unbounded):

```
>>> est_large = kde.KDE1D(zz, lower=0, method=kde_methods.reflection)
>>> xxs, yys = est_large.grid()
>>> plt.plot(xxs, yys, 'r--', lw=2, label='Estimated')
>>> plt.xlim(0, 6)
>>> plt.ylim(ymin=0)
>>> plt.legend(loc='best')
```



5.3.3 Cyclic Boundaries

Cyclic boundaries work very much like reflexive boundary. The main difference is that they require two bounds, as reflexive conditions can be only with one bound.

5.4 Methods for Bandwidth Estimation

There are a number of ways to estimate the bandwidth. The simplest way is to specify it numerically, either during construction or after:

```
>>> est = kde.KDE1D(x, bandwidth=.1)
>>> est.bandwidth = .2
```

If is sometimes more convenient to specify the variance of the kernel (which is the square bandwidth). So these are equivalent to the two previous lines:

```
>>> est = kde.KDE1D(x, bandwidth=.01)
>>> est.covariance = .04
```

But often you will want to use a pre-defined method:

```
>>> est = kde.KDE1D(x, covariance = kde.scotts_covariance)
```

At last, if you want to define your own method, you simply need to define a function. For example, you can re-define and use the function for the Scotts rule of thumb (which compute the variance of the kernel):

```
>>> def my_scotts(x, model=None):
...   return (0.75 * len(x))**-.2 * x.var()
>>> est = kde.KDE1D(x, covariance=my_scotts)
```

The model object is a reference to the KDE object and is mostly useful if you need to know about the boundaries of the domain.

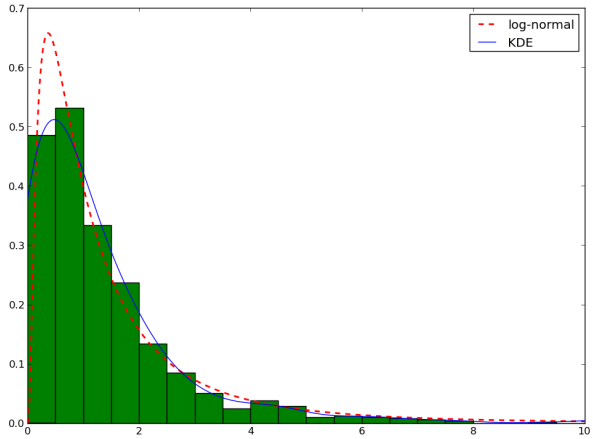
5.5 Transformations

Sometimes, it is not really possible to estimate correctly the density is the current domain. A transformation is required. As an example, let's try to estimate a log-normal distribution, i.e. the distribution of a variable whose logarithm is normally distributed:

```
>>> from scipy import stats
>>> from matplotlib import pylab as plt
>>> from pyqt_fit import kde, kde_methods
>>> import numpy as np
>>> f = stats.lognorm(1)
>>> x = f.rvs(1000)
>>> xs = r_[0:10:4096j]
>>> plt.hist(x, bins=20, range=(0,10), color='g', normed=True)
>>> plt.plot(xs, f.pdf(xs), 'r--', lw=2, label='log-normal')
>>> est = kde.KDE1D(x, method=kde_methods.linear_combination, lower=0)
>>> plt.plot(xs, est(xs), color='b', label='KDE')
>>> plt.legend(loc='best')
```

You can note that even the histogram doesn't reflect very well the distribution here. The linear recombination method, although not perfect also gives a better idea of what is going on. But really, we should be working in log space:

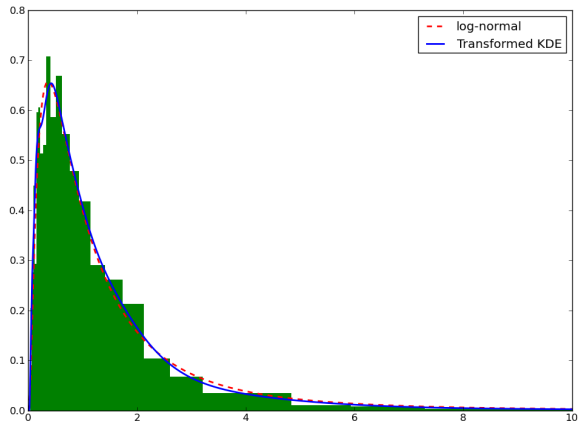
```
>>> plt.figure()
>>> lx = np.log(x)
>>> h, edges = np.histogram(lx, bins=30, range=(-np.log(30), np.log(10)))
>>> width = np.exp(edges[1:]) - np.exp(edges[:-1])
>>> h = h / width
>>> h /= len(x)
>>> plt.bar(np.exp(edges[:-1]), h, width = width, facecolor='g', linewidth=0, ecolor='b')
```




```
>>> plt.plot(xs, f.pdf(xs), 'r--', lw=2, label='log-normal')
>>> plt.xlim(xmax=10)
>>> plt.legend(loc='best')
```

We can do the same for the KDE by using the *transformKDE* method. This method requires the transformation as argument:

```
>>> trans = kde.KDE1D(x, method=kde_methods.transformKDE1D(kde_methods.LogTransform))
>>> plt.plot(xs, trans(xs), color='b', lw=2, label='Transformed KDE')
>>> plt.legend(loc='best')
```



CHAPTER 6

Modules of PyQt-Fit

6.1 Module pyqt_fit.plot_fit

Author Pierre Barbier de Reuille <pierre.barbierdereuille@gmail.com>

This modules implement functions to test and plot parametric regression.

6.1.1 Analyses of the residuals

pyqt_fit.plot_fit.**fit_evaluation**(*fit*, *xdata*, *ydata*, *eval_points=None*, *CI=()*, *CIresults=None*, *xname='X'*, *yname='Y'*, *fct_desc=None*, *param_names=()*, *residuals=None*, *res_name='Standard'*)

This function takes the output of a curve fitting experiment and store all the relevant information for evaluating its success in the result.

Parameters

- **fit** (*fitting object*) – object configured for the fitting
- **xdata** (*ndarray of shape (N,) or (k,N) for function with k prefictors*) – The independent variable where the data is measured
- **ydata** (*ndarray*) – The dependant data
- **eval_points** (*ndarray or None*) – Contain the list of points on which the result must be expressed. It is used both for plotting and for the bootstrapping.
- **CI** (*tuple of int*) – List of confidence intervals to calculate. If empty, none are calculated.
- **xname** (*string*) – Name of the X axis
- **yname** (*string*) – Name of the Y axis
- **fct_desc** (*string*) – Formula of the function
- **param_names** (*tuple of strings*) – Name of the various parameters
- **residuals** (*callable*) – Residual function
- **res_desc** (*string*) – Description of the residuals

Return type `ResultStruct`

Returns Data structure summarising the fitting and its evaluation

pyqt_fit.plot_fit.**residual_measures**(*res*)

Compute quantities needed to evaluate the quality of the estimation, based solely on the residuals.

Return type `ResidualMeasures`

Returns the scaled residuals, their ordering, the theoretical quantile for each residuals, and the expected value for each quantile.

6.1.2 Plotting the residuals

`pyqt_fit.plot_fit.`**`plot_dist_residuals`**(*res*)

Plot the distribution of the residuals.

Returns the handle toward the histogram and the plot of the fitted normal distribution

`pyqt_fit.plot_fit.`**`plot_residuals`**(*xname*, *xdata*, *res_desc*, *res*)

Plot the residuals against the X axis

Parameters

- **xname** (*str*) – Name of the X axis
- **xdata** (*ndarray*) – 1D array with the X data
- **res_desc** (*str*) – Name of the Y axis
- **res** (*ndarray*) – 1D array with the residuals

The shapes of `xdata` and `res` must be the same

Returns The handles of the the plots of the residuals and of the smoothed residuals.

`pyqt_fit.plot_fit.`**`scaled_location_plot`**(*yname*, *yopt*, *scaled_res*)

Plot the scaled location, given the dependant values and scaled residuals.

Parameters

- **yname** (*str*) – Name of the Y axis
- **yopt** (*ndarray*) – Estimated values
- **scaled_res** (*ndarray*) – Scaled residuals

Returns the handles for the data and the smoothed curve

`pyqt_fit.plot_fit.`**`qqplot`**(*scaled_res*, *normq*)

Draw a Q-Q Plot from the sorted, scaled residuals (i.e. residuals sorted and normalized by their standard deviation)

Parameters

- **scaled_res** (*ndarray*) – Scaled residuals
- **normq** (*ndarray*) – Expected value for each scaled residual, based on its quantile.

Returns handle to the data plot

`pyqt_fit.plot_fit.`**`plot_residual_tests`**(*xdata*, *yopts*, *res*, *fct_name*, *xname='X'*, *yname='Y'*, *res_name='residuals'*, *sorted_yopts=None*, *scaled_res=None*, *normq=None*, *fig=None*)

Plot, in a single figure, all four residuals evaluation plots: `plot_residuals()`, `plot_dist_residuals()`, `scaled_location_plot()` and `qqplot()`.

Parameters

- **xdata** (*ndarray*) – Explaining variables
- **yopt** (*ndarray*) – Optimized explained variables

- **fct_name** (*str*) – Name of the fitted function
- **xname** (*str*) – Name of the explaining variables
- **yname** (*str*) – Name of the dependant variables
- **res_name** (*str*) – Name of the residuals
- **sorted_yopts** (*ndarray*) – `yopt`, sorted to match the scaled residuals
- **scaled_res** (*ndarray*) – Scaled residuals
- **normq** (*ndarray*) – Estimated value of the quantiles for a normal distribution
- **fig** (*handle or None*) – Handle of the figure to put the plots in, or None to create a new figure

Return type `ResTestResult`

Returns The handles to all the plots

6.1.3 General plotting

`pyqt_fit.plot_fit.`**`plot1d`**(*result*, *loc=0*, *fig=None*, *res_fig=None*)

Use matplotlib to display the result of a fit, and return the list of plots used

Return type `Plot1dResult`

Returns hangles to the various figures and plots

6.1.4 Output to a file

`pyqt_fit.plot_fit.`**`write1d`**(*outfile*, *result*, *res_desc*, *CImethod*)

Write the result of a fitting and its evaluation to a CSV file.

Parameters

- **outfile** (*str*) – Name of the file to write to
- **result** (*ResultStruct*) – Result of the fitting evaluation (e.g. output of `fit_evaluation()`)
- **res_desc** (*str*) – Description of the residuals (in more details than just the name of the residuals)
- **CImethod** (*str*) – Description of the confidence interval estimation method

6.1.5 Return types

Most function return a tuple. For easier access, there are named tuple, i.e. tuples that can be accessed by name.

class `pyqt_fit.plot_fit.`**`ResultStruct`**(...)

Note: This is a class created with `pyqt_fit.utils.namedtuple()`.

`fct`

Fitted function (i.e. result of the fitted function)

`fct_desc`

Description of the function being fitted

param_names
Name of the parameters fitted

xdata
Explaining variables used for fitting

ydata
Dependent variables observed during experiment

xname
Name of the explaining variables

yname
Name of the dependent variabled

res_name
Name of the residuals

residuals
Function used to compute the residuals

popt
Optimal parameters

res
Residuals computed with the parameters `popt`

yopts
Evaluation of the optimized function on the observed points

eval_points
Points on which the function has been interpolated (may be equal to xdata)

interpolation
Interpolated function on `eval_points` (may be equal to `yopt`)

sorted_yopts
Evaluated function for each data points, sorted in increasing residual order

scaled_res
Scaled residuals, ordered by increasing residuals

normq
Expected values for the residuals, based on their quantile

CI
List of confidence intervals evaluated (in percent)

CIs
List of arrays giving the confidence intervals for the dependent variables and for the parameters.

CIresults
Object returned by the confidence interval method

class `pyqt_fit.plot_fit.`**`ResidualMeasures`**(*scaled_res*, *res_IX*, *prob*, *normq*)

Note: This is a class created with `pyqt_fit.utils.namedtuple()`.

---

scaled_res
Scaled residuals, sorted

res_IX
Sorting indices for the residuals

prob
Quantiles of the scaled residuals

normq
Expected values of the quantiles for a normal distribution

class `pyqt_fit.plot_fit.`**`ResTestResult`**(*res_figure*, *residuals*, *scaled_residuals*, *qqplot*, *dist_residuals*)

Note: This is a class created with `pyqt_fit.utils.namedtuple()`.

res_figure
Handle to the figure

residuals
Handles created by `plot_residuals()`

scaled_residuals
Handles created by `scaled_location_plot()`

qqplot
Handles created by `qqplot()`

dist_residuals
Handles created by `plot_dist_residuals()`

class `pyqt_fit.plot_fit.`**`Plot1dResult`**(*figure*, *estimate*, *data*, *CIs*, **ResTestResult*)

Note: This is a class create with `pyqt_fit.utils.namedtuple()`. Also, it contains all the first of `ResTestResult` at the end of the tuple.

figure
Handle to the figure with the data and fitted curve

estimate
Handle to the fitted curve

data
Handle to the data

CIs
Handles to the confidence interval curves

6.2 Module `pyqt_fit.curve_fitting`

Author Pierre Barbier de Reuille <pierre.barbierdereuille@gmail.com>

This module specifically implement the curve fitting, wrapping the default scipy.optimize.leastsq function. It allows for parameter value fixing, different kind of residual and added constraints function.

The main class of the module is the `CurveFitting` class.

class `pyqt_fit.curve_fitting.`**`CurveFitting`**(*xdata*, *ydata*, ***kwords*)
Fit a curve using the `scipy.optimize.leastsq()` function

Parameters

- **xdata** (*ndarray*) – Explaining values
- **ydata** (*ndarray*) – Target values

Once fitted, the following variables contain the result of the fitting:

Variables

- **popt** (*ndarray*) – The solution (or the result of the last iteration for an unsuccessful call)
- **pcov** (*ndarray*) – The estimated covariance of popt. The diagonals provide the variance of the parameter estimate.
- **res** (*ndarray*) – Final residuals
- **infodict** (*dict*) – a dictionary of outputs with the keys:

  `nfev` the number of function calls

  `fvec` the function evaluated at the output

  `fjac` A permutation of the R matrix of a QR factorization of the final approximate Jacobian matrix, stored column wise. Together with ipvt, the covariance of the estimate can be approximated.

  `ipvt` an integer array of length N which defines a permutation matrix, `p`, such that `fjac*p = q*r`, where `r` is upper triangular with diagonal elements of nonincreasing magnitude. Column `j` of `p` is column `ipvt(j)` of the identity matrix.

  `qtf` the vector `(transpose(q) * fvec)`

  `CI` list of tuple of parameters, each being the lower and upper bounds for the confidence interval in the CI argument at the same position.

  `est_jacobian` True if the jacobian is estimated, false if the user-provided functions have been used

---

Note: In this implementation, residuals are supposed to be a generalisation of the notion of difference. In the end, the mathematical expression of this minimisation is:

$$\hat{\theta} = \text{argmin}_{\theta \in \mathbb{R}^p} \sum_i r(y_i, f(\theta, x_i))^2$$

Where θ is the vector of p parameters to optimise, r is the residual function and f is the function being fitted.

`__call__`(*xdata*)

Return the value of the fitted function for each of the points in `xdata`

6.3 Module `pyqt_fit.bootstrap`

Author Pierre Barbier de Reuille <pierre.barbierdereuille@gmail.com>

This modules provides function for bootstrapping a regression method.

6.3.1 Bootstrap Shuffling Methods

`pyqt_fit.bootstrap.`**`bootstrap_residuals`**(*fct*, *xdata*, *ydata*, *repeats=3000*, *residuals=None*, *add_residual=None*, *correct_bias=False*, ***kwrds*)

This implements the residual bootstrapping method for non-linear regression.

Parameters

- **fct** (*callable*) – Function evaluating the function on xdata at least with `fct(xdata)`
- **xdata** (*ndarray of shape (N,) or (k,N) for function with k predictors*) – The independent variable where the data is measured
- **ydata** (*ndarray*) – The dependant data
- **residuals** (*ndarray or callable or None*) – Residuals for the estimation on each xdata. If callable, the call will be `residuals(ydata, yopt)`.
- **repeats** (*int*) – Number of repeats for the bootstrapping
- **add_residual** (*callable or None*) – Function that add a residual to a value. The call `add_residual(ydata, residual)` should return the new ydata, with the residuals 'applied'. If None, it is considered the residuals should simply be added.
- **correct_bias** (*boolean*) – If true, the additive bias of the residuals is computed and restored
- **kwrds** (*dict*) – Dictionnary present to absorbed unknown named parameters

Return type (ndarray, ndarray)

Returns

1. xdata, with a new axis at position -2. This correspond to the 'shuffled' xdata (as they are *not* shuffled here)

2.Second item is the shuffled ydata. There is a line per repeat, each line is shuffled independently.

`pyqt_fit.bootstrap.`**`bootstrap_regression`**(*fct*, *xdata*, *ydata*, *repeats=3000*, ***kwrds*)

This implements the shuffling of standard bootstrapping method for non-linear regression.

Parameters

- **fct** (*callable*) – This is the function to optimize
- **xdata** (*ndarray of shape (N,) or (k,N) for function with k predictors*) – The independent variable where the data is measured
- **ydata** (*ndarray*) – The dependant data
- **repeats** (*int*) – Number of repeats for the bootstrapping
- **kwrds** (*dict*) – Dictionnary to absorbed unknown named parameters

Return type (ndarray, ndarray)

Returns

1. The shuffled x data. The axis -2 has one element per repeat, the other axis are shuffled independently.

2. The shuffled ydata. There is a line per repeat, each line is shuffled independently.

6.3.2 Main Boostrap Functions

`pyqt_fit.bootstrap.`**`bootstrap`**(*fit*, *xdata*, *ydata*, *CI*, *shuffle_method=<function bootstrap_residuals at 0x7fe6847c47d0>*, *shuffle_args=()*, *shuffle_kwrds={}*, *repeats=3000*, *eval_points=None*, *full_results=False*, *nb_workers=None*, *extra_attrs=()*, *fit_args=()*, *fit_kwrds={}*)

This function implement the bootstrap algorithm for a regression algorithm. It is capable of spreading the load across many threads using shared memory and the `multiprocess` module.

Parameters

- **fit** (*callable*) – Method used to compute regression. The call is:

  ```
  f = fit(xdata, ydata, *fit_args, **fit_kwrds)
  ```

  Fit should return an object that would evaluate the regression on a set of points. The next call will be:

  ```
  f(eval_points)
  ```

- **xdata** (*ndarray of shape (N,) or (k,N) for function with k predictors*) – The independent variable where the data is measured
- **ydata** (*ndarray*) – The dependant data
- **CI** (*tuple of float*) – List of percentiles to extract
- **shuffle_method** (*callable*) – Create shuffled dataset. The call is:

  ```
  shuffle_method(xdata, ydata, y_est, repeat=repeats, *shuffle_args,
                 **shuffle_kwrds)
  ```

  where `y_est` is the estimated dependant variable on the xdata.
- **shuffle_args** (*tuple*) – List of arguments for the shuffle method
- **shuffle_kwrds** (*dict*) – Dictionnary of arguments for the shuffle method
- **repeats** (*int*) – Number of repeats for the bootstraping
- **eval_points** (*ndarray or None*) – List of points to evaluate. If None, eval_point is xdata.
- **full_results** (*bool*) – if True, output also the whole set of evaluations
- **nb_worders** – Number of worker threads. If None, the number of detected CPUs will be used. And if 1 or less, a single thread will be used.
- **extra_attrs** (*tuple of str*) – List of attributes of the fitting method to extract on top of the y values for confidence intervals
- **fit_args** (*tuple*) – List of extra arguments for the fit callable
- **fit_kwrds** (*dict*) – Dictionnary of extra named arguments for the fit callable

Return type `BootstrapResult`

Returns Estimated y on the data, on the evaluation points, the requested confidence intervals and, if requested, the shuffled X, Y and the full estimated distributions.

class `pyqt_fit.bootstrap.`**`BootstrapResult`**(*y_fit*, *y_est*, *y_eval*, *CIs*, *shuffled_xs*, *shuffled_ys*, *full_results*)

Note: This is a class created with `pyqt_fit.utils.namedtuple()`.

`y_fit`
Y estimated on xdata

`y_est: ndarray`
Y estimated on eval_points

CIs
List of confidence intervals. The first element is for the estimated values on `eval_points`. The others are for the extra attributes specified in `extra_attrs`. Each array is a 3-dimensional array (Q,2,N), where Q is the number of confidence interval and N is the number of data points. Values (x,0,y) give the lower bounds and (x,1,y) the upper bounds of the confidence intervals.

shuffled_xs
if full_results is True, the shuffled x's used for the bootstrapping

shuffled_ys
if full_results is True, the shuffled y's used for the bootstrapping

full_results
if full_results is True, the estimated y's for each shuffled_ys

6.4 Module `pyqt_fit.nonparam_regression`

Author Pierre Barbier de Reuille <pierre.barbierdereuille@gmail.com>

Module implementing non-parametric regressions using kernel methods.

class `pyqt_fit.nonparam_regression.`**`NonParamRegression`**(*xdata*, *ydata*, ***kwords*)
Class performing kernel-based non-parametric regression.

The calculation is split in three parts:

- The kernel (`kernel`)
- Bandwidth computation (`bandwidth`, `covariance`)
- Regression method (`method`)

N
Number of points in the dataset (set by the fitting)

bandwidth
Bandwidth of the kernel.

This is defined as the square root of the covariance matrix

covariance
Covariance matrix of the kernel.

It must be of the right dimension!

dim
Dimension of the domain (set by the fitting)

fit()
Method to call to fit the parameters of the fitting

fitted
Check if the fitting needs to be performed.

fitted_method
Method actually used after fitting.

The main method may choose to provide a more tuned method during fitting.

kernel
Kernel object. Should provide the following methods:

kernel.pdf(xs) Density of the kernel, denoted $K(x)$

kernel_type
Type of the kernel. The kernel type is a class or function accepting the dimension of the domain as argument and returning a valid kernel object.

lower
Lower bound of the domain for each dimension

method
Regression method itself. It should be an instance of the class following the template `pyqt_fit.npr_methods.RegressionKernelMethod`.

need_fit()
Calling this function will mark the object as needing fitting.

set_actual_bandwidth(*bandwidth*, *covariance*)
Method computing the bandwidth if needed (i.e. if it was defined by functions)

upper
Lower bound of the domain for each dimension

xdata
2D array (D,N) with D the dimension of the domain and N the number of points.

ydata
1D array (N,) of values for each point in xdata

6.5 Module `pyqt_fit.npr_methods`

Author Pierre Barbier de Reuille <pierre.barbierdereuille@gmail.com>

Module implementing non-parametric regressions using kernel methods.

6.5.1 Non-Parametric Regression Methods

Methods must either inherit or follow the same definition as the `pyqt_fit.npr_methods.RegressionKernelMethod`.

`pyqt_fit.npr_methods.`**compute_bandwidth**(*reg*)
Compute the bandwidth and covariance for the model, based of its xdata attribute

class `pyqt_fit.npr_methods.`**RegressionKernelMethod**
Base class for regression kernel methods

The following methods are interface methods that should be overriden with ones specific to the implemented method.

fit(*reg*)
Fit the method and returns the fitted object that will be used for actual evaluation.

The object needs to call the `pyqt_fit.nonparam_regression.NonParamRegression.set_actual_bandwi` method with the computed bandwidth and covariance.

Default Compute the bandwidth based on the real data and set it in the regression object

evaluate(*points*, *out*)
Evaluate the regression of the provided points.

Parameters

- **points** (*ndarray*) – 2d-array of points to compute the regression on. Each column is a point.
- **out** (*ndarray*) – 1d-array in which to store the result

Return type ndarray

Returns The method must return the `out` array, updated with the regression values

6.5.2 Provided methods

Only extra methods will be described:

class `pyqt_fit.npr_methods.`**`SpatialAverage`**

Perform a Nadaraya-Watson regression on the data (i.e. also called local-constant regression) using a gaussian kernel.

The Nadaraya-Watson estimate is given by:

$$f_n(x) \triangleq \frac{\sum_i K\left(\frac{x-X_i}{h}\right) Y_i}{\sum_i K\left(\frac{x-X_i}{h}\right)}$$

Where $K(x)$ is the kernel and must be such that $E(K(x)) = 0$ and h is the bandwidth of the method.

Parameters

- **xdata** (*ndarray*) – Explaining variables (at most 2D array)
- **ydata** (*ndarray*) – Explained variables (should be 1D array)
- **cov** (*ndarray or callable*) – If an ndarray, it should be a 2D array giving the matrix of covariance of the gaussian kernel. Otherwise, it should be a function `cov(xdata, ydata)` returning the covariance matrix.

`q`

Degree of the fitted polynom

`correction``()`

The correction coefficient allows to change the width of the kernel depending on the point considered. It can be either a constant (to correct globaly the kernel width), or a 1D array of same size as the input.

`set_density_correction``()`

Add a correction coefficient depending on the density of the input

class `pyqt_fit.npr_methods.`**`LocalLinearKernel1D`**

Perform a local-linear regression using a gaussian kernel.

The local constant regression is the function that minimises, for each position:

$$f_n(x) \triangleq \operatorname{argmin}_{a_0 \in \mathbb{R}} \sum_i K\left(\frac{x-X_i}{h}\right) \left(Y_i - a_0 - a_1(x-X_i)\right)^2$$

Where $K(x)$ is the kernel and must be such that $E(K(x)) = 0$ and h is the bandwidth of the method.

`q`

Degree of the fitted polynom

class `pyqt_fit.npr_methods.`**`LocalPolynomialKernel1D`**`(`*q=3*`)`

Perform a local-polynomial regression using a user-provided kernel (Gaussian by default).

The local constant regression is the function that minimises, for each position:

$$f_n(x) \triangleq \operatorname{argmin}_{a_0 \in \mathbb{R}} \sum_i K\left(\frac{x - X_i}{h}\right)\left(Y_i - a_0 - a_1(x - X_i) - \ldots - a_q \frac{(x - X_i)^q}{q!}\right)^2$$

Where $K(x)$ is the kernel such that $E(K(x)) = 0$, q is the order of the fitted polynomial and h is the bandwidth of the method. It is also recommended to have $\int_{\mathbb{R}} x^2 K(x) dx = 1$, (i.e. variance of the kernel is 1) or the effective bandwidth will be scaled by the square-root of this integral (i.e. the standard deviation of the kernel).

Parameters

- **xdata** (*ndarray*) – Explaining variables (at most 2D array)
- **ydata** (*ndarray*) – Explained variables (should be 1D array)
- **q** (*int*) – Order of the polynomial to fit. **Default:** 3
- **cov** (*float or callable*) – If an float, it should be a variance of the gaussian kernel. Otherwise, it should be a function `cov(xdata, ydata)` returning the variance. **Default:** `scotts_covariance`

q

Degree of the fitted polynom

class `pyqt_fit.npr_methods.`**`LocalPolynomialKernel`**(*q=3*)

Perform a local-polynomial regression in N-D using a user-provided kernel (Gaussian by default).

The local constant regression is the function that minimises, for each position:

$$f_n(x) \triangleq \operatorname{argmin}_{a_0 \in \mathbb{R}} \sum_i K\left(\frac{x - X_i}{h}\right)(Y_i - a_0 - \mathcal{P}_q(X_i - x))^2$$

Where $K(x)$ is the kernel such that $E(K(x)) = 0$, q is the order of the fitted polynomial, $\mathcal{P}_q(x)$ is a polynomial of order d in x and h is the bandwidth of the method.

The polynomial $\mathcal{P}_q(x)$ is of the form:

$$\mathcal{F}_d(k) = \left\{ \boldsymbol{n} \in \mathbb{N}^d \middle| \sum_{i=1}^d n_i = k \right\}$$

$$\mathcal{P}_q(x_1, \ldots, x_d) = \sum_{k=1}^q \sum_{\boldsymbol{n} \in \mathcal{F}_d(k)} a_{k,\boldsymbol{n}} \prod_{i=1}^d x_i^{n_i}$$

For example we have:

$$\mathcal{P}_2(x, y) = a_{110}x + a_{101}y + a_{220}x^2 + a_{211}xy + a_{202}y^2$$

Parameters

- **xdata** (*ndarray*) – Explaining variables (at most 2D array). The shape should be (N,D) with D the dimension of the problem and N the number of points. For 1D array, the shape can be (N,), in which case it will be converted to (N,1) array.
- **ydata** (*ndarray*) – Explained variables (should be 1D array). The shape must be (N,).
- **q** (*int*) – Order of the polynomial to fit. **Default:** 3
- **kernel** (*callable*) – Kernel to use for the weights. Call is `kernel(points)` and should return an array of values the same size as `points`. If `None`, the kernel will be `normal_kernel(D)`.

- **cov** (*float or callable*) – If an float, it should be a variance of the gaussian kernel. Otherwise, it should be a function `cov(xdata, ydata)` returning the variance. **Default:** `scotts_covariance`

q
Degree of the fitted polynomials

`pyqt_fit.npr_methods.`**`default_method`**
Defaut non-parametric regression method. :Default: LocalPolynomialKernel(q=1)

6.5.3 Utility functions and classes

class `pyqt_fit.npr_methods.`**`PolynomialDesignMatrix1D`**(*degree*)

class `pyqt_fit.npr_methods.`**`PolynomialDesignMatrix`**(*dim*, *deg*)
Class used to create a design matrix for polynomial regression

`__call__`(*x*, *out=None*)
Creates the design matrix for polynomial fitting using the points x.

Parameters

- **x** (*ndarray*) – Points to create the design matrix. Shape must be (D,N) or (N,), where D is the dimension of the problem, 1 if not there.
- **deg** (*int*) – Degree of the fitting polynomial
- **factors** (*ndarray*) – Scaling factor for the columns of the design matrix. The shape should be (M,) or (M,1), where M is the number of columns of the out. This value can be obtained using the `designMatrixSize()` function.

Returns The design matrix as a (M,N) matrix.

6.6 Module `pyqt_fit.kde`

Author Pierre Barbier de Reuille <pierre.barbierdereuille@gmail.com>

Module implementing kernel-based estimation of density of probability.

Given a kernel K, the density function is estimated from a sampling $X = \{X_i \in \mathbb{R}^n\}_{i\in\{1,\dots,m\}}$ as:

$$f(\mathbf{z}) \triangleq \frac{1}{hW}\sum_{i=1}^{m}\frac{w_i}{\lambda_i}K\left(\frac{X_i-\mathbf{z}}{h\lambda_i}\right)$$
$$W = \sum_{i=1}^{m} w_i$$

where h is the bandwidth of the kernel, w_i are the weights of the data points and λ_i are the adaptation factor of the kernel width.

The kernel is a function of $\mathbb{R}^n$ such that:

$$\begin{array}{rrcll} & \int\cdots\int_{\mathbb{R}^n} f(\mathbf{z})d\mathbf{z} & = & 1 & \iff & f \text{ is a probability} \\ & \int\cdots\int_{\mathbb{R}^n} \mathbf{z}f(\mathbf{z})d\mathbf{z} & = & \mathbf{0} & \iff & f \text{ is centered} \\ \forall \mathbf{u}\in\mathbb{R}^n, \|\mathbf{u}\|=1 & \int_{\mathbb{R}} t^2 f(t\mathbf{u})dt & \approx & 1 & \iff & \text{The co-variance matrix of } f \text{ is close to be the identity.} \end{array}$$

The constraint on the covariance is only required to provide a uniform meaning for the bandwidth of the kernel.

If the domain of the density estimation is bounded to the interval $[L, U]$, the density is then estimated with:

$$f(x) \triangleq \frac{1}{hW} \sum_{i=1}^{n} \frac{w_i}{\lambda_i} \hat{K}(x; X, \lambda_i h, L, U)$$

where $\hat{K}$ is a modified kernel that depends on the exact method used. Currently, only 1D KDE supports bounded domains.

6.6.1 Kernel Density Estimation Methods

class `pyqt_fit.kde.`**`KDE1D`**(*xdata*, ***kwords*)

Perform a kernel based density estimation in 1D, possibly on a bounded domain $[L, U]$.

Parameters

- **data** (*ndarray*) – 1D array with the data points
- **kwords** (*dict*) – setting attributes at construction time. Any named argument will be equivalent to setting the property after the fact. For example:

```
>>> xs = [1,2,3]
>>> k = KDE1D(xs, lower=0)
```

will be equivalent to:

```
>>> k = KDE1D(xs)
>>> k.lower = 0
```

The calculation is separated in three parts:

•The kernel (`kernel`)

•The bandwidth or covariance estimation (`bandwidth`, `covariance`)

•The estimation method (`method`)

`__call__`(*points*, *out=None*)

This method is an alias for `BoundedKDE1D.evaluate()`

`bandwidth`

Bandwidth of the kernel. Can be set either as a fixed value or using a bandwidth calculator, that is a function of signature `w(xdata)` that returns a single value.

Note: A ndarray with a single value will be converted to a floating point value.

`cdf_grid`(*N=None*, *cut=None*)

Compute the cdf from the lower bound to the points given as argument.

`closed`

Returns true if the density domain is closed (i.e. lower and upper are both finite)

`copy`()

Shallow copy of the KDE object

`covariance`

Covariance of the gaussian kernel. Can be set either as a fixed value or using a bandwidth calculator, that is a function of signature `w(xdata)` that returns a single value.

Note: A ndarray with a single value will be converted to a floating point value.

evaluate (*points*, *out=None*)
Compute the PDF of the distribution on the set of points `points`

fit ()
Compute the various parameters needed by the kde method

grid (*N=None*, *cut=None*)
Evaluate the density on a grid of N points spanning the whole dataset.

Returns a tuple with the mesh on which the density is evaluated and the density itself

icdf_grid (*N=None*, *cut=None*)
Compute the inverse cumulative distribution (quantile) function on a grid.

kernel
Kernel object. This must be an object modeled on `pyqt_fit.kernels.Kernel1D`. It is recommended to inherit this class to provide numerical approximation for all methods.

By default, the kernel is an instance of `pyqt_fit.kernels.normal_kernel1d`

lambdas
Scaling of the bandwidth, per data point. It can be either a single value or an array with one value per data point.

When deleted, the lamndas are reset to 1.

lower
Lower bound of the density domain. If deleted, becomes set to $-\infty$

method
Select the method to use. The method should be an object modeled on `pyqt_fit.kde_methods.KDE1DMethod`, and it is recommended to inherit the model.

Available methods in the `pyqt_fit.kde_methods` sub-module.

Default `pyqt_fit.kde_methods.default_method`

upper
Upper bound of the density domain. If deleted, becomes set to ∞

weights
Weigths associated to each data point. It can be either a single value, or an array with a value per data point. If a single value is provided, the weights will always be set to 1.

6.6.2 Bandwidth Estimation Methods

`pyqt_fit.kde.`**variance_bandwidth** (*factor*, *xdata*)
Returns the covariance matrix:

$$\mathcal{C} = \tau^2 cov(X)$$

where τ is a correcting factor that depends on the method.

`pyqt_fit.kde.`**silverman_covariance** (*xdata*, *model=None*)
The Silverman bandwidth is defined as a variance bandwidth with factor:

$$\tau = \left(n\frac{d+2}{4}\right)^{\frac{-1}{d+4}}$$

`pyqt_fit.kde.`**`scotts_covariance`**(*xdata*, *model=None*)

The Scotts bandwidth is defined as a variance bandwidth with factor:

$$\tau = n^{\frac{-1}{d+4}}$$

`pyqt_fit.kde.`**`botev_bandwidth`**(*N=None*, ***kword*)

Implementation of the KDE bandwidth selection method outline in:

Z. I. Botev, J. F. Grotowski, and D. P. Kroese. Kernel density estimation via diffusion. The Annals of Statistics, 38(5):2916-2957, 2010.

Based on the implementation of Daniel B. Smith, PhD.

The object is a callable returning the bandwidth for a 1D kernel.

6.7 Module `pyqt_fit.kde_methods`

Author Pierre Barbier de Reuille <pierre.barbierdereuille@gmail.com>

This module contains a set of methods to compute univariate KDEs. See the objects in the `pyqt_fit.kde` module for more details on these methods.

These methods provide various variations on $\hat{K}(x; X, h, L, U)$, the modified kernel evaluated on the point x based on the estimation points X, a bandwidth h and on the domain $[L, U]$.

The definitions of the methods rely on the following definitions:

$$\begin{array}{rcl} a_0(l,u) & = & \int_l^u K(z)dz \\ a_1(l,u) & = & \int_l^u zK(z)dz \\ a_2(l,u) & = & \int_l^u z^2K(z)dz \end{array}$$

These definitions correspond to:

- $a_0(l, u)$ – The partial cumulative distribution function
- $a_1(l, u)$ – The partial first moment of the distribution. In particular, $a_1(-\infty, \infty)$ is the mean of the kernel (i.e. and should be 0).
- $a_2(l, u)$ – The partial second moment of the distribution. In particular, $a_2(-\infty, \infty)$ is the variance of the kernel (i.e. which should be close to 1, unless using higher order kernel).

6.7.1 References:

6.7.2 Univariate KDE estimation methods

The exact definition of such a method is found in `pyqt_fit.kde.KDE1D.method`

`pyqt_fit.kde_methods.`**`generate_grid`**(*kde*, *N=None*, *cut=None*)

Helper method returning a regular grid on the domain of the KDE.

Parameters

- **kde** (*KDE1D*) – Object describing the KDE computation. The object must have been fitted!
- **N** (*int*) – Number of points in the grid
- **cut** (*float*) – for unbounded domains, how far past the maximum should the grid extend to, in term of KDE bandwidth

Returns A vector of N regularly spaced points

`pyqt_fit.kde_methods.`**`compute_bandwidth`**(*kde*)

Compute the bandwidth and covariance for the model, based of its xdata attribute

class `pyqt_fit.kde_methods.`**`KDE1DMethod`**

Base class providing a default grid method and a default method for unbounded evaluation of the PDF and CDF. It also provides default methods for the other metrics, based on PDF and CDF calculations.

Note

- It is expected that all grid methods will return the same grid if used with the same arguments.
- It is fair to assume all array-like arguments will be at least 1D arrays.

The following methods are interface methods that should be overriden with ones specific to the implemented method.

`fit`(*kde*)

Method called by the KDE1D object right after fitting to allow for one-time calculation.

Parameters **kde** (*pyqt_fit.kde.KDE1D*) – KDE object being fitted

Default Compute the bandwidth and covariance if specified as functions

`pdf`(*kde*, *points*, *out*)

Compute the PDF of the estimated distribution.

Parameters

- **kde** (*pyqt_fit.kde.KDE1D*) – KDE object
- **points** (*ndarray*) – Points to evaluate the distribution on
- **out** (*ndarray*) – Result object. If must have the same shapes as `points`

Return type ndarray

Returns Returns the `out` variable, updated with the PDF.

Default Direct implementation of the formula for unbounded pdf computation.

`__call__`(*kde*, *points*, *out*)

Call the `pdf()` method.

`grid`(*kde*, *N=None*, *cut=None*)

Evaluate the PDF of the distribution on a regular grid with at least `N` elements.

Parameters

- **kde** (*pyqt_fit.kde.KDE1D*) – KDE object
- **N** (*int*) – minimum number of element in the returned grid. Most methods will want to round it to the next power of 2.
- **cut** (*float*) – for unbounded domains, how far from the last data point should the grid go, as a fraction of the bandwidth.

Return type (ndarray, ndarray)

Returns The array of positions the PDF has been estimated on, and the estimations.

Default Evaluate $pdf(x)$ on a grid generated using `generate_grid()`

cdf (*kde*, *points*, *out*)

Compute the CDF of the estimated distribution, defined as:

$$cdf(x) = P(X \leq x) = \int_l^x p(t)dt$$

where l is the lower bound of the distribution domain and p the density of probability

Parameters

- **kde** (*pyqt_fit.kde.KDE1D*) – KDE object
- **points** (*ndarray*) – Points to evaluate the CDF on
- **out** (*ndarray*) – Result object. If must have the same shapes as `points`

Return type ndarray

Returns Returns the `out` variable, updated with the CDF.

Default Direct implementation of the formula for unbounded CDF computation.

cdf_grid (*kde*, *N=None*, *cut=None*)

Evaluate the CDF of the distribution on a regular grid with at least `N` elements.

Parameters

- **kde** (*pyqt_fit.kde.KDE1D*) – KDE object
- **N** (*int*) – minimum number of element in the returned grid. Most methods will want to round it to the next power of 2.
- **cut** (*float*) – for unbounded domains, how far from the last data point should the grid go, as a fraction of the bandwidth.

Return type (ndarray, ndarray)

Returns The array of positions the CDF has been estimated on, and the estimations.

Default Evaluate $cdf(x)$ on a grid generated using `generate_grid()`

icdf (*kde*, *points*, *out*)

Compute the inverse cumulative distribution (quantile) function, defined as:

$$icdf(p) = \inf \{x \in \mathbb{R} : cdf(x) \geq p\}$$

Parameters

- **kde** (*pyqt_fit.kde.KDE1D*) – KDE object
- **points** (*ndarray*) – Points to evaluate the iCDF on
- **out** (*ndarray*) – Result object. If must have the same shapes as `points`

Return type ndarray

Returns Returns the `out` variable, updated with the iCDF.

Default First approximate the result using linear interpolation on the CDF and refine the result numerically using the Newton method.

icdf_grid (*kde*, *N=None*, *cut=None*)

Compute the inverse cumulative distribution (quantile) function on a grid.

Note The default implementation is not as good an approximation as the plain icdf default method.

Parameters

- **kde** (*pyqt_fit.kde.KDE1D*) – KDE object
- **N** (*int*) – minimum number of element in the returned grid. Most methods will want to round it to the next power of 2.
- **cut** (*float*) – for unbounded domains, how far from the last data point should the grid go, as a fraction of the bandwidth.

Return type (ndarray, ndarray)

Returns The array of positions the CDF has been estimated on, and the estimations.

Default Linear interpolation of the inverse CDF on a grid

sf(*kde*, *points*, *out*)

Compute the survival function, defined as:

$$sf(x) = P(X \geq x) = \int_x^u p(t)dt = 1 - cdf(x)$$

Parameters

- **kde** (*pyqt_fit.kde.KDE1D*) – KDE object
- **points** (*ndarray*) – Points to evaluate the survival function on
- **out** (*ndarray*) – Result object. If must have the same shapes as `points`

Return type ndarray

Returns Returns the `out` variable, updated with the survival function.

Default Compute explicitly $1 - cdf(x)$

sf_grid(*kde*, *N=None*, *cut=None*)

Compute the survival function on a grid.

Parameters

- **kde** (*pyqt_fit.kde.KDE1D*) – KDE object
- **N** (*int*) – minimum number of element in the returned grid. Most methods will want to round it to the next power of 2.
- **cut** (*float*) – for unbounded domains, how far from the last data point should the grid go, as a fraction of the bandwidth.

Return type (ndarray, ndarray)

Returns The array of positions the survival function has been estimated on, and the estimations.

Default Compute explicitly $1 - cdf(x)$

isf(*kde*, *points*, *out*)

Compute the inverse survival function, defined as:

$$isf(p) = \sup\{x \in \mathbb{R} : sf(x) \leq p\}$$

Parameters

- **kde** (*pyqt_fit.kde.KDE1D*) – KDE object
- **points** (*ndarray*) – Points to evaluate the iSF on
- **out** (*ndarray*) – Result object. If must have the same shapes as `points`

Return type ndarray

Returns Returns the `out` variable, updated with the inverse survival function.

Default Compute $icdf(1-p)$

`isf_grid`(*kde*, *N=None*, *cut=None*)

Compute the inverse survival function on a grid.

Note The default implementation is not as good an approximation as the plain isf default method.

Parameters

- **kde** (*pyqt_fit.kde.KDE1D*) – KDE object
- **N** (*int*) – minimum number of element in the returned grid. Most methods will want to round it to the next power of 2.
- **cut** (*float*) – for unbounded domains, how far from the last data point should the grid go, as a fraction of the bandwidth.

Return type (ndarray, ndarray)

Returns The array of positions the CDF has been estimated on, and the estimations.

Default Linear interpolation of the inverse survival function on a grid

`hazard`(*kde*, *points*, *out*)

Compute the hazard function evaluated on the points.

The hazard function is defined as:

$$h(x) = \frac{p(x)}{sf(x)}$$

where $p(x)$ is the probability density function and $sf(x)$ is the survival function.

Parameters

- **kde** (*pyqt_fit.kde.KDE1D*) – KDE object
- **points** (*ndarray*) – Points to evaluate the hazard function on
- **out** (*ndarray*) – Result object. If must have the same shapes as `points`

Return type ndarray

Returns Returns the `out` variable, updated with the hazard function

Default Compute explicitly $pdf(x)/sf(x)$

`hazard_grid`(*kde*, *N=None*, *cut=None*)

Compute the hazard function on a grid.

Parameters

- **kde** (*pyqt_fit.kde.KDE1D*) – KDE object
- **N** (*int*) – minimum number of element in the returned grid. Most methods will want to round it to the next power of 2.
- **cut** (*float*) – for unbounded domains, how far from the last data point should the grid go, as a fraction of the bandwidth.

Return type (ndarray, ndarray)

Returns The array of positions the hazard function has been estimated on, and the estimations.

Default Compute explicitly $pdf(x)/sf(x)$

cumhazard(*kde*, *points*, *out*)

Compute the cumulative hazard function evaluated on the points.

The hazard function is defined as:

$$ch(x) = \int_l^x h(t)dt = -\ln sf(x)$$

where l is the lower bound of the domain, h the hazard function and sf the survival function.

Parameters

- **kde** (*pyqt_fit.kde.KDE1D*) – KDE object
- **points** (*ndarray*) – Points to evaluate the cumulative hazard function on
- **out** (*ndarray*) – Result object. If must have the same shapes as `points`

Return type ndarray

Returns Returns the `out` variable, updated with the cumulative hazard function

Default Compute explicitly $-\ln sf(x)$

cumhazard_grid(*kde*, *N=None*, *cut=None*)

Compute the hazard function on a grid.

Parameters

- **kde** (*pyqt_fit.kde.KDE1D*) – KDE object
- **N** (*int*) – minimum number of element in the returned grid. Most methods will want to round it to the next power of 2.
- **cut** (*float*) – for unbounded domains, how far from the last data point should the grid go, as a fraction of the bandwidth.

Return type (ndarray, ndarray)

Returns The array of positions the hazard function has been estimated on, and the estimations.

Default Compute explicitly $-\ln sf(x)$

name

Type str

Specify a human-readable name for the method, for presentation purposes.

But the class also provide a number of utility methods to help implementing you own:

numeric_cdf(*kde*, *points*, *out*)

Provide a numeric approximation of the CDF based on integrating the pdf using `scipy.integrate.quad()`.

numeric_cdf_grid(*kde*, *N=None*, *cut=None*)

Compute the CDF on a grid using a trivial, but fast, numeric integration of the pdf.

Estimation methods

Here are the methods implemented in pyqt_fit. To access these methods, the simplest is to use the instances provided.

`pyqt_fit.kde_methods.`**unbounded**

Instance of the `KDE1DMethod` class.

`pyqt_fit.kde_methods.`**`renormalization`**
Instance of the `RenormalizationMethod` class.

`pyqt_fit.kde_methods.`**`reflection`**
Instance of the `ReflectionMethod` class.

`pyqt_fit.kde_methods.`**`linear_combination`**
Instance of the `LinearCombinationMethod` class.

`pyqt_fit.kde_methods.`**`cyclic`**
Instance of the `CyclicMethod` class.

`pyqt_fit.kde_methods.`**`transformKDE1D`**(*trans*, *method=None*, *inv=None*, *Dinv=None*)
Creates an instance of `TransformKDE1DMethod`

`pyqt_fit.kde_methods.`**`default_method`**
Method used by `pyqt_fit.kde.KDE1D` by default. :Default: `reflection`

Classes implementing the estimation methods

class `pyqt_fit.kde_methods.`**`RenormalizationMethod`**
This method consists in using the normal kernel method, but renormalize to only take into account the part of the kernel within the domain of the density [1].

The kernel is then replaced with:

$$\hat{K}(x; X, h, L, U) \triangleq \frac{1}{a_0(u, l)} K(z)$$

where:

$$z = \frac{x - X}{h} \qquad l = \frac{L - x}{h} \qquad u = \frac{U - x}{h}$$

class `pyqt_fit.kde_methods.`**`ReflectionMethod`**
This method consist in simulating the reflection of the data left and right of the boundaries. If one of the boundary is infinite, then the data is not reflected in that direction. To this purpose, the kernel is replaced with:

$$\hat{K}(x; X, h, L, U) \triangleq K(z) + K\left(\frac{x + X - 2L}{h}\right) + K\left(\frac{x + X - 2U}{h}\right)$$

where:

$$z = \frac{x - X}{h}$$

See the `pyqt_fit.kde_methods` for a description of the various symbols.

When computing grids, if the bandwidth is constant, the result is computing using CDT.

class `pyqt_fit.kde_methods.`**`LinearCombinationMethod`**
This method uses the linear combination correction published in [1].

The estimation is done with a modified kernel given by:

$$\hat{K}(x; X, h, L, U) \triangleq \frac{a_2(l, u) - a_1(-u, -l)z}{a_2(l, u)a_0(l, u) - a_1(-u, -l)^2} K(z)$$

where:

$$z = \frac{x - X}{h} \qquad l = \frac{L - x}{h} \qquad u = \frac{U - x}{h}$$

[1] Jones, M. C. 1993. Simple boundary correction for kernel density estimation. Statistics and Computing 3: 135–146.

class `pyqt_fit.kde_methods.`**`CyclicMethod`**

This method assumes cyclic boundary conditions and works only for closed boundaries.

The estimation is done with a modified kernel given by:

$$\hat{K}(x; X, h, L, U) \triangleq K(z) + K\left(z - \frac{U-L}{h}\right) + K\left(z + \frac{U-L}{h}\right)$$

where:

$$z = \frac{x - X}{h}$$

When computing grids, if the bandwidth is constant, the result is computing using FFT.

`pyqt_fit.kde_methods.`**`create_transform`**(*obj*, *inv=None*, *Dinv=None*)

Create a transform object.

Parameters

- **obj** (*fun*) – This can be either simple a function, or a function-object with an 'inv' and/or 'Dinv' attributes containing the inverse function and its derivative (respectively)
- **inv** (*fun*) – If provided, inverse of the main function
- **Dinv** (*fun*) – If provided, derivative of the inverse function

Return type Transform

Returns A transform object with function, inverse and derivative of the inverse

The inverse function must be provided, either as argument or as attribute to the object. The derivative of the inverse will be estimated numerically if not provided.

Note All the functions should accept an `out` argument to store the result.

class `pyqt_fit.kde_methods.`**`TransformKDE1DMethod`**(*trans*, *method=None*, *inv=None*, *Dinv=None*)

Compute the Kernel Density Estimate of a dataset, transforming it first to a domain where distances are "more meaningful".

Often, KDE is best estimated in a different domain. This object takes a KDE1D object (or one compatible), and a transformation function.

Given a random variable X of distribution f_X, the random variable $Y = g(X)$ has a distribution f_Y given by:

$$f_Y(y) = \left|\frac{1}{g'(g^{-1}(y))}\right| \cdot f_X(g^{-1}(y))$$

In our term, Y is the random variable the user is interested in, and X the random variable we can estimate using the KDE. In this case, g is the transform from Y to X.

So to estimate the distribution on a set of points given in x, we need a total of three functions:

•Direct function: transform from the original space to the one in which the KDE will be perform (i.e. $g^{-1} : y \mapsto x$)

•Invert function: transform from the KDE space to the original one (i.e. $g : x \mapsto y$)

•Derivative of the invert function

If the derivative is not provided, it will be estimated numerically.

Parameters

- **trans** – Either a simple function, or a function object with attributes *inv* and *Dinv* to use in case they are not provided as arguments. The helper `create_transform()` will provide numeric approximation of the derivative if required.
- **method** – instance of KDE1DMethod used in the transformed domain. Default is `pyqt_fit.kde_methods.KDE1DMethod`
- **inv** – Invert of the function. If not provided, *trans* must have it as attribute.
- **Dinv** – Derivative of the invert function.

Note all given functions should accept an optional `out` argument to get a pre-allocated array to store its result. Also the `out` parameter may be one of the input argument.

6.8 Module `pyqt_fit.kernels`

Author Pierre Barbier de Reuille <pierre.barbierdereuille@gmail.com>

Module providing a set of kernels for use with either the `pyqt_fit.kde` or the `kernel_smoothing` module.

Kernels should be created following this template:

6.8.1 Helper class

This class is provided with default implementations of everything in term of the PDF.

class `pyqt_fit.kernels.`**`Kernel1D`**

A 1D kernel $K(z)$ is a function with the following properties:

$$\begin{array}{rcll} \int_{\mathbb{R}} K(z) & = & 1 & \\ \int_{\mathbb{R}} zK(z)dz & = & 0 & \\ \int_{\mathbb{R}} z^2K(z)dz & < & \infty & (\approx 1) \end{array}$$

Which translates into the function should have:

- a sum of 1 (i.e. a valid density of probability);
- an average of 0 (i.e. centered);
- a finite variance. It is even recommanded that the variance is close to 1 to give a uniform meaning to the bandwidth.

`cut`

Type float

Cutting point after which there is a negligeable part of the probability. More formally, if c is the cutting point:

$$\int_{-c}^{c} p(x)dx \approx 1$$

`lower`

Type float

Lower bound of the support of the PDF. Formally, if l is the lower bound:

$$\int_{-\infty}^{l} p(x)dx = 0$$

upper

Type float

Upper bound of the support of the PDF. Formally, if u is the upper bound:

$$\int_u^\infty p(x)dx = 0$$

cdf (z, *out=None*)

Returns the cumulative density function on the points z, i.e.:

$$K_0(z) = \int_{-\infty}^z K(t)dt$$

dct (z, *out=None*)

DCT of the kernel on the points of `z`. The points will always be provided as a grid with 2^n points, representing the whole frequency range to be explored.

fft (z, *out=None*)

FFT of the kernel on the points of `z`. The points will always be provided as a grid with 2^n points, representing the whole frequency range to be explored. For convenience, the second half of the points will be provided as negative values.

pdf (z, *out=None*)

Returns the density of the kernel on the points z. This is the funtion $K(z)$ itself.

Parameters

- **z** (*ndarray*) – Array of points to evaluate the function on. The method should accept any shape of array.
- **out** (*ndarray*) – If provided, it will be of the same shape as z and the result should be stored in it. Ideally, it should be used for as many intermediate computation as possible.

pm1 (z, *out=None*)

Returns the first moment of the density function, i.e.:

$$K_1(z) = \int_{-\infty}^z zK(t)dt$$

pm2 (z, *out=None*)

Returns the second moment of the density function, i.e.:

$$K_2(z) = \int_{-\infty}^z z^2K(t)dt$$

6.8.2 Gaussian Kernels

class `pyqt_fit.kernels.`**`normal_kernel`**(*dim*)

Returns a function-object for the PDF of a Normal kernel of variance identity and average 0 in dimension `dim`.

pdf (*xs*)

Return the probability density of the function.

Parameters **xs** (*ndarray*) – Array of shape (D,N) where D is the dimension of the kernel and N the number of points.

Returns an array of shape (N,) with the density on each point of `xs`

class `pyqt_fit.kernels.`**`normal_kernel1d`**

1D normal density kernel with extra integrals for 1D bounded kernel estimation.

`cdf` (*z*, *out=None*)

Cumulative density of probability. The formula used is:

$$\mathrm{cdf}(z) \triangleq \int_{-\infty}^{z} \phi(z)dz = \frac{1}{2}\mathrm{erf}\left(\frac{z}{\sqrt{2}}\right) + \frac{1}{2}$$

`dct` (*z*, *out=None*)

Returns the DCT of the normal distribution

`fft` (*z*, *out=None*)

Returns the FFT of the normal distribution

`pdf` (*z*, *out=None*)

Return the probability density of the function. The formula used is:

$$\phi(z) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$$

Parameters **xs** (*ndarray*) – Array of any shape

Returns an array of shape identical to `xs`

`pm1` (*z*, *out=None*)

Partial moment of order 1:

$$\mathrm{pm1}(z) \triangleq \int_{-\infty}^{z} z\phi(z)dz = -\frac{1}{\sqrt{2\pi}} e^{-\frac{z^2}{2}}$$

`pm2` (*z*, *out=None*)

Partial moment of order 2:

$$\mathrm{pm2}(z) \triangleq \int_{-\infty}^{z} z^2\phi(z)dz = \frac{1}{2}\mathrm{erf}\left(\frac{z}{2}\right) - \frac{z}{\sqrt{2\pi}} e^{-\frac{z^2}{2}} + \frac{1}{2}$$

6.8.3 Tricube Kernel

class `pyqt_fit.kernels.`**`tricube`**

Return the kernel corresponding to a tri-cube distribution, whose expression is. The tri-cube function is given by:

$$f_r(x) = \begin{cases} \left(1 - |x|^3\right)^3 & \text{, if } x \in [-1; 1] \\ 0 & \text{, otherwise} \end{cases}$$

As f_r is not a probability and is not of variance 1, we use a normalized function:

$$f(x) = abf_r(ax)$$
$$a = \sqrt{\frac{35}{243}}$$
$$b = \frac{70}{81}$$

cdf (*z*, *out=None*)

CDF of the distribution:

$$\mathrm{cdf}(x) = \begin{cases} \frac{1}{162}\left(60(ax)^7 - 7\left(2(ax)^{10} + 15(ax)^4\right)\mathrm{sgn}\,(ax) + 140ax + 81\right) & \text{, if} x \in [-1/a; 1/a] \\ 0 & \text{, if} x < -1/a \\ 1 & \text{, if} x > 1/a \end{cases}$$

pm1 (*z*, *out=None*)

Partial moment of order 1:

$$\mathrm{pm1}(x) = \begin{cases} \frac{7}{3564a}\left(165(ax)^8 - 8\left(5(ax)^{11} + 33(ax)^5\right)\mathrm{sgn}\,(ax) + 220(ax)^2 - 81\right) & \text{, if} x \in [-1/a; 1/a] \\ 0 & \text{, otherwise} \end{cases}$$

pm2 (*z*, *out=None*)

Partial moment of order 2:

$$\mathrm{pm2}(x) = \begin{cases} \frac{35}{486a^2}\left(4(ax)^9 + 4(ax)^3 - \left((ax)^{12} + 6(ax)^6\right)\mathrm{sgn}\,(ax) + 1\right) & \text{, if} x \in [-1/a; 1/a] \\ 0 & \text{, if } x < -1/a \\ 1 & \text{, if } x > 1/a \end{cases}$$

6.8.4 Epanechnikov Kernel

class `pyqt_fit.kernels.`**`Epanechnikov`**

1D Epanechnikov density kernel with extra integrals for 1D bounded kernel estimation.

cdf (*xs*, *out=None*)

CDF of the distribution. The CDF is defined on the interval $[-\sqrt{5} : \sqrt{5}]$ as:

$$\mathrm{cdf}(x) = \begin{cases} \frac{1}{2} + \frac{3}{4\sqrt{5}}x - \frac{3}{20\sqrt{5}}x^3 & \text{, if } x \in [-\sqrt{5} : \sqrt{5}] \\ 0 & \text{, if } x < -\sqrt{5} \\ 1 & \text{, if } x > \sqrt{5} \end{cases}$$

pdf (*xs*, *out=None*)

The PDF of the kernel is usually given by:

$$f_r(x) = \begin{cases} \frac{3}{4}\left(1 - x^2\right) & \text{, if} x \in [-1 : 1] \\ 0 & \text{, otherwise} \end{cases}$$

As f_r is not of variance 1 (and therefore would need adjustments for the bandwidth selection), we use a normalized function:

$$f(x) = \frac{1}{\sqrt{5}} f\left(\frac{x}{\sqrt{5}}\right)$$

pm1 (*xs*, *out=None*)

First partial moment of the distribution:

$$\mathrm{pm1}(x) = \begin{cases} -\frac{3\sqrt{5}}{16}\left(1 - \frac{2}{5}x^2 + \frac{1}{25}x^4\right) & \text{, if } x \in [-\sqrt{5} : \sqrt{5}] \\ 0 & \text{, otherwise} \end{cases}$$

pm2 (*xs*, *out=None*)

Second partial moment of the distribution:

$$\mathrm{pm2}(x) = \begin{cases} \frac{5}{20}\left(2 + \frac{1}{\sqrt{5}}x^3 - \frac{3}{5^{5/2}}x^5\right) & \text{, if } x \in [-\sqrt{5} : \sqrt{5}] \\ 0 & \text{, if } x < -\sqrt{5} \\ 1 & \text{, if } x > \sqrt{5} \end{cases}$$

6.8.5 Higher Order Kernels

High order kernels are kernel that give up being valid probabilities. We will say a kernel $K_{[n]}$ is of order n if:

$$\begin{array}{rcl} \int_{\mathbb{R}} K_{[n]}(x)dx & = & 1 \\ \forall 1 \leq k < n \int_{\mathbb{R}} x^k K_{[n]} dx & = & 0 \\ \int_{\mathbb{R}} x^n K_{[n]} dx & \neq & 0 \end{array}$$

PyQt-Fit implements two high order kernels.

class `pyqt_fit.kernels.`**`Epanechnikov_order4`**

Order 4 Epanechnikov kernel. That is:

$$K_{[4]}(x) = \frac{3}{2}K(x) + \frac{1}{2}xK'(x) = -\frac{15}{8}x^2 + \frac{9}{8}$$

where K is the non-normalized Epanechnikov kernel.

class `pyqt_fit.kernels.`**`normal_order4`**

Order 4 Normal kernel. That is:

$$\phi_{[4]}(x) = \frac{3}{2}\phi(x) + \frac{1}{2}x\phi'(x) = \frac{1}{2}(3 - x^2)\phi(x)$$

where ϕ is the normal kernel.

6.9 Module `pyqt_fit.utils`

Author Pierre Barbier de Reuille <pierre.barbierdereuille@gmail.com>

Module contained a variety of small useful functions.

`pyqt_fit.utils.`**`namedtuple`**(*typename*, *field_names*, *verbose=False*, *rename=False*)

Returns a new subclass of tuple with named fields.

```
>>> Point = namedtuple('Point', 'x y')
>>> Point.__doc__                   # docstring for the new class
'Point(x, y)'
>>> p = Point(11, y=22)             # instantiate with positional args or keywords
>>> p[0] + p[1]                     # indexable like a plain tuple
33
>>> x, y = p                        # unpack like a regular tuple
>>> x, y
(11, 22)
>>> p.x + p.y                       # fields also accessable by name
33
>>> d = p._asdict()                 # convert to a dictionary
>>> d['x']
11
>>> Point(**d)                      # convert from a dictionary
Point(x=11, y=22)
>>> p._replace(x=100)               # _replace() is like str.replace() but targets named fields
Point(x=100, y=22)
```

`pyqt_fit.utils.`**`approx_jacobian`**(*x*, *func*, *epsilon*, **args*)

Approximate the Jacobian matrix of callable function func

Parameters

- **x** (*ndarray*) – The state vector at which the Jacobian matrix is desired

- **func** (*callable*) – A vector-valued function of the form f(x,*args)
- **epsilon** (*ndarray*) – The peturbation used to determine the partial derivatives
- **args** (*tuple*) – Additional arguments passed to func

Returns An array of dimensions (lenf, lenx) where lenf is the length of the outputs of func, and lenx is the number of

---

Note: The approximation is done using forward differences

CHAPTER 7

Indices and tables

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

p