numkit Documentation

Release 1.2.2+6.g85d0ece

Oliver Beckstein

March 13, 2023

Contents

Release 1.2.2+6.g85d0ece

Date March 13, 2023

numkit contains numerical helper functions and classes that have been proven to be useful for analysis of molecular dynamics simulation. The package used to be part of GromacsWrapper.

`numkit` consists of a collection of functions and classes built on top of NumPy and SciPy to aid the numerical analysis of data. It is geared towards the use of data coming from molecular simulations, namely time series. It is used in `gromacs.formats.XVG`.

Please note that these functions are provided "as is" and no guarantee is given that they are accurate or free from error. Bug reports and test cases are very welcome. Please submit them through the Github Issue Tracker.

See also:

Core functionality is based on SciPy (`scipy` module). For more general time series analysis see pandas.

CHAPTER 1

Contents

1.1 Installation

You can install from *Source Code* or as a package.

1.1.1 pip installation

Install with **pip** from the numkit PyPi repository with

```
pip install numkit
```

Python 2.7 and recent Python 3 are all supported.

1.1.2 conda installation

New in version 1.2.1.

Changed in version 1.2.2: moved from *bioconda* to *conda-forge* channel

Install the conda-forge numkit package with conda package manager from the conda-forge channel with

```
conda install -c conda-forge numkit
```

1.2 Source Code

The source code itself is available in the numkit git repository under the BSD 3-clause license (see file `LICENSE`).

1.2.1 Public git repository

Get the sources with

```
git clone https://github.com/Becksteinlab/numkit
```

and install with

```
pip install ./numkit
```

1.2.2 Contributing

Contributions in the form of feedback, bug reports, suggestions and pull requests are very welcome.

1.3 `numkit.fitting` — Fitting data

The module contains functions to do least square fits of functions of one variable f(x) to data points (x,y).

1.3.1 Example

For example, to fit a un-normalized Gaussian with `FitGauss` to data distributed with mean 5.0 and standard deviation 3.0:

```
from numkit.fitting import FitGauss
import numpy, numpy.random

# generate suitably noisy data
mu, sigma = 5.0, 3.0
Y,edges = numpy.histogram(sigma*numpy.random.randn(10000), bins=100, density=True)
X = 0.5*(edges[1:]+edges[:-1]) + mu

g = FitGauss(X, Y)

print(g.parameters)
# [ 4.98084541  3.00044102  1.00069061]
print(numpy.array([mu, sigma, 1]) - g.parameters)
# [ 0.01915459 -0.00044102 -0.00069061]

import matplotlib.pyplot as plt
plt.plot(X, Y, 'ko', label="data")
plt.plot(X, g.fit(X), 'r-', label="fit")
```

If the initial parameters for the least square optimization do not lead to a solution then one can provide customized starting values in the *parameters* keyword argument:

```
g = FitGauss(X, Y, parameters=[10, 1, 1])
```

The *parameters* have different meaning for the different fit functions; the documentation for each function shows them in the context of the fit function.

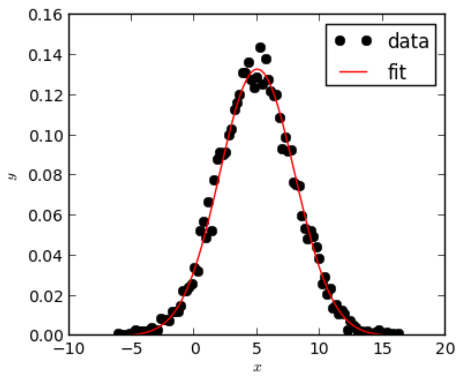

Fig. 1: A Gaussian (red) was fit to the data points (black circles) with the `numkit.fitting.FitGauss` class.

1.3.2 Creating new fit functions

New fit function classes can be derived from `FitFunc`. The documentation and the methods `FitFunc.f_factory()` and `FitFunc.initial_values()` must be overriden. For example, the class `FitGauss` is implemented as

```
class FitGauss(FitFunc):
    '''y = f(x) = p[2] * 1/sqrt(2*pi*p[1]**2) * exp(-(x-p[0])**2/(2*p[1]**2))'''
    def f_factory(self):
        def fitfunc(p,x):
            return p[2] * 1.0/(p[1]*numpy.sqrt(2*numpy.pi)) * numpy.exp(-(x-p[0])**2/
↪(2*p[1]**2))
        return fitfunc
    def initial_values(self):
        return [0.0,1.0,0.0]
```

The function to be fitted is defined in `fitfunc()`. The parameters are accessed as `p[0]`, `p[1]`, ... For each parameter, a suitable initial value must be provided.

1.3.3 Functions and classes

`numkit.fitting.`**`Pearson_r`**(*x*, *y*)

Pearson's r (correlation coefficient).

Pearson(x,y) –> correlation coefficient

x and *y* are arrays of same length.

Historical note – Naive implementation of Pearson's r :: Ex = scipy.stats.mean(x) Ey = scipy.stats.mean(y) covxy = numpy.sum((x-Ex)*(y-Ey)) r = covxy/math.sqrt(numpy.sum((x-Ex)**2)*numpy.sum((y-Ey)**2))

`numkit.fitting.`**`linfit`**(*x*, *y*, *dy=None*)

Fit a straight line y = a + bx to the data in *x* and *y*.

Errors on y should be provided in dy in order to assess the goodness of the fit and derive errors on the parameters.

linfit(x,y[,dy]) –> result_dict

Fit y = a + bx to the data in x and y by analytically minimizing chi^2. dy holds the standard deviations of the individual y_i. If dy is not given, they are assumed to be constant (note that in this case Q is set to 1 and it is meaningless and chi2 is normalised to unit standard deviation on all points!).

Returns the parameters a and b, their uncertainties sigma_a and sigma_b, and their correlation coefficient r_ab; it also returns the chi-squared statistic and the goodness-of-fit probability Q (that the fit would have chi^2 this large or larger; Q < 10^-2 indicates that the model is bad — Q is the probability that a value of chi-square as _poor_ as the calculated statistic chi2 should occur by chance.)

Returns result_dict with components

intercept, sigma_intercept a +/- sigma_a

slope, sigma_slope b +/- sigma_b

parameter_correlation correlation coefficient r_ab between a and b

chi_square chi^2 test statistic

Q_fit goodness-of-fit probability

Based on 'Numerical Recipes in C', Ch 15.2.

class `numkit.fitting.`**`FitFunc`**(*x*, *y*, *parameters=None*)

Fit a function f to data (x,y) using the method of least squares.

The function is fitted when the object is created, using `scipy.optimize.leastsq()`. One must derive from the base class *`FitFunc`* and override the *`FitFunc.f_factory()`* (including the definition of an appropriate local `fitfunc()` function) and *`FitFunc.initial_values()`* appropriately. See the examples for a linear fit *`FitLin`*, a 1-parameter exponential fit *`FitExp`*, or a 3-parameter double exponential fit *`FitExp2`*.

The object provides two attributes

`FitFunc.parameters` list of parameters of the fit

`FitFunc.message` message from `scipy.optimize.leastsq()`

After a successful fit, the fitted function can be applied to any data (a 1D-numpy array) with *`FitFunc.fit()`*.

`f_factory`()

Stub for fit function factory, which returns the fit function. Override for derived classes.

`fit`(*x*)

Applies the fit to all *x* values

`initial_values`()

List of initital guesses for all parameters p[]

class `numkit.fitting.`**`FitLin`**(*x*, *y*, *parameters=None*)

y = f(x) = p[0]*x + p[1]

class `numkit.fitting.`**`FitExp`**(*x*, *y*, *parameters=None*)

y = f(x) = exp(-p[0]*x)

class `numkit.fitting.`**`FitExp2`**(*x*, *y*, *parameters=None*)

y = f(x) = p[0]*exp(-p[1]*x) + (1-p[0])*exp(-p[2]*x)

class `numkit.fitting.`**`FitGauss`**(*x*, *y*, *parameters=None*)

y = f(x) = p[2] * 1/sqrt(2*pi*p[1]**2) * exp(-(x-p[0])**2/(2*p[1]**2))

Fits an un-normalized gaussian (height scaled with parameter p[2]).

- p[0] == mean mu
- p[1] == standard deviation $sigma$
- p[2] == scale a

1.4 `numkit.timeseries` — Time series manipulation and analysis

A time series contains of a sequence of time points (typically spaced equally) and a value for each time point. This module contains some standard scientific time series manipulations.

See also:

pandas

1.4.1 Correlations

Autocorrelation time (time when ACF becomes 0 for the first time; uses `gromacs.formats.XVG` to read the data and `numkit.timeseries.autocorrelation_fft()` to calculate the ACF):

```
R = gromacs.formats.XVG("./md.xvg")
acf = autocorrelation_fft(R.array[1])
numpy.where(acf <= 0)[0][0]
```

Alternatively, fit an exponential to the ACF and extract the time constant (see `numkit.timeseries.tcorrel()`).

`numkit.timeseries.`**`tcorrel`**(*x*, *y*, *nstep=100*, *debug=False*)

Calculate the correlation time and an estimate of the error of the mean <y>.

The autocorrelation function f(t) is calculated via FFT on every *nstep* of the **fluctuations** of the data around the mean (y-<y>). The normalized ACF f(t)/f(0) is assumed to decay exponentially, f(t)/f(0) = exp(-t/tc) and the decay constant tc is estimated as the integral of the ACF from the start up to its first root.

See [FrenkelSmit2002] p526 for details.

Note: *nstep* should be set sufficiently large so that there are less than ~50,000 entries in the input.

Arguments

x 1D array of abscissa values (typically time)

y 1D array of the ibservable y(x)

nstep only analyze every *nstep* datapoint to speed up calculation [100]

Returns dictionary with entries *tc* (decay constant in units of *x*), *t0* (value of the first root along x (y(t0) = 0)), *sigma* (error estimate for the mean of y, <y>, corrected for correlations in the data).

`numkit.timeseries.`**`autocorrelation_fft`**(*series*, *remove_mean=True*, *paddingcorrection=True*, *normalize=False*, ***kwargs*)

Calculate the auto correlation function.

autocorrelation_fft(series,remove_mean=False,**kwargs) –> acf

The time series is correlated with itself across its whole length. Only the [0,len(series)[interval is returned.

By default, the mean of the series is subtracted and the correlation of the fluctuations around the mean are investigated.

For the default setting remove_mean=True, acf[0] equals the variance of the series, acf[0] = Var(series) = <(series - <series>)**2>.

Optional:

- The series can be normalized to its 0-th element so that acf[0] == 1.
- For calculating the acf, 0-padding is used. The ACF should be corrected for the 0-padding (the values for larger lags are increased) unless mode='valid' is set (see below).

Note that the series for mode='same'|'full' is inaccurate for long times and should probably be truncated at 1/2*len(series)

Arguments

series (time) series, a 1D numpy array of length N

remove_mean `False`: use series as is; `True`: subtract mean(series) from series [`True`]

paddingcorrection `False`: corrected for 0-padding; `True`: return as is it is. (the latter is appropriate for periodic signals). The correction for element 0=<i<N amounts to a factor N/(N-i). Only applied for modes != "valid" [`True`]

normalize `True` divides by acf[0] so that the first element is 1; `False` leaves un-normalized [`False`]

mode "full" | "same" | "valid": see `scipy.signal.fftconvolve()` ["full"]

kwargs other keyword arguments for `scipy.signal.fftconvolve()`

1.4.2 Coarse graining time series

The functions in this section are all based on `regularized_function()`. They reduce the number of datapoints in a time series to *maxpoints* by histogramming the data into *maxpoints* bins and then applying a function to reduce the data in each bin. A number of commonly used functions are predefined but it is straightforward to either use `apply_histogrammed_function()` or `regularized_function()` directly. For instance, `mean_histogrammed_function()` is implemented as

```
def mean_histogrammed_function(t, y, maxpoints):
    return apply_histogrammed_function(numpy.mean, t, y, maxpoints)
```

More complicated functions can be defined; for instance, one could use *`numkit.timeseries.tcorrel()`* to compute the correlation time of the data in short blocks:

```
def tc_histogrammed_function(t, y, maxpoints):
   dt = numpy.mean(numpy.diff(t))
   def get_tcorrel(y):
      t = numpy.cumsum(dt*numpy.ones_like(y)) - dt
      results = tcorrel(t, y, nstep=1)
      return results['tc']
   return apply_histogrammed_function(get_tcorrel, t, y, bins=maxpoints)
```

(This particular function (implemented as *`numkit.timeseries.tc_histogrammed_function()`*) is not very robust, for instance it has problems when there are only very few data points in each bin because in this case the auto correlation function is not well defined.)

`numkit.timeseries.`**`mean_histogrammed_function`**(*t*, *y*, ***kwargs*)

Compute mean of data *y* in bins along *t*.

Returns the mean-regularised function *F* and the centers of the bins.

See also:

`regularized_function()` with *func* = `numpy.mean()`

`numkit.timeseries.`**`rms_histogrammed_function`**`(`*t*, *y*, ***kwargs*`)`
Compute root mean square of data *y* in bins along *t*.

Returns the RMS-regularised function *F* and the centers of the bins. *demean* = `True` removes the mean first.

`regularized_function()` with *func* = `sqrt(mean(y*y))`

`numkit.timeseries.`**`min_histogrammed_function`**`(`*t*, *y*, ***kwargs*`)`
Compute minimum of data *y* in bins along *t*.

Returns the min-regularised function *F* and the centers of the bins.

`regularized_function()` with *func* = `numpy.min()`

`numkit.timeseries.`**`max_histogrammed_function`**`(`*t*, *y*, ***kwargs*`)`
Compute maximum of data *y* in bins along *t*.

Returns the max-regularised function *F* and the centers of the bins.

`regularized_function()` with *func* = `numpy.max()`

`numkit.timeseries.`**`median_histogrammed_function`**`(`*t*, *y*, ***kwargs*`)`
Compute median of data *y* in bins along *t*.

Returns the median-regularised function *F* and the centers of the bins.

`regularized_function()` with *func* = `numpy.median()`

`numkit.timeseries.`**`percentile_histogrammed_function`**`(`*t*, *y*, ***kwargs*`)`
Compute the percentile *per* of data *y* in bins along *t*.

Returns the percentile-regularised function *F* and the centers of the bins.

Keywords

per percentile as a percentage, e.g. 75 is the value that splits the data into the lower 75% and upper 25%; 50 is the median [50.0]

demean `True`: remove the mean of the bin data first [`False`]

`regularized_function()` with `scipy.stats.scoreatpercentile()`

`numkit.timeseries.`**`error_histogrammed_function`**`(`*t*, *y*, ***kwargs*`)`
Calculate the error in each bin using *`tcorrel()`*.

Warning: Not well tested and fragile.

`numkit.timeseries.`**`circmean_histogrammed_function`**`(`*t*, *y*, ***kwargs*`)`
Compute circular mean of data *y* in bins along *t*.

Returns the circmean-regularised function *F* and the centers of the bins.

kwargs are passed to `scipy.stats.morestats.circmean()`, in particular set the lower bound with *low* and the upper one with *high*. The default is [-pi, +pi].

`regularized_function()` with *func* = `scipy.stats.morestats.circmean()`

Note: Data are interpreted as angles in radians.

`numkit.timeseries.`**`circstd_histogrammed_function`**`(`*t*, *y*, ***kwargs*`)`
Compute circular standard deviation of data *y* in bins along *t*.

Returns the circstd-regularised function *F* and the centers of the bins.

kwargs are passed to `scipy.stats.morestats.circmean()`, in particular set the lower bound with *low* and the upper one with *high*. The default is [-pi, +pi].

`regularized_function()` with *func* = `scipy.stats.morestats.circstd()`

Note: Data are interpreted as angles in radians.

`numkit.timeseries.`**`tc_histogrammed_function`**(*t*, *y*, ***kwargs*)

Calculate the correlation time in each bin using `tcorrel()`.

Warning: Not well tested and fragile.

`numkit.timeseries.`**`apply_histogrammed_function`**(*func*, *t*, *y*, ***kwargs*)

Compute *func* of data *y* in bins along *t*.

Returns the *func* -regularised function *F(t')* and the centers of the bins *t'*.

`numkit.timeseries.`**`func`**(*y*) → float

func takes exactly one argument, a numpy 1D array *y* (the values in a single bin of the histogram), and reduces it to one scalar float.

`numkit.timeseries.`**`regularized_function`**(*x*, *y*, *func*, *bins=100*, *range=None*)

Compute *func()* over data aggregated in bins.

`(x,y) --> (x', func(Y'))` with `Y' = {y:  y(x) where x in x' bin}`

First the data is collected in bins x' along x and then *func* is applied to all data points Y' that have been collected in the bin.

`numkit.timeseries.`**`func`**(*y*) → float

func takes exactly one argument, a numpy 1D array *y* (the values in a single bin of the histogram), and reduces it to one scalar float.

Note: *x* and *y* must be 1D arrays.

Arguments

x abscissa values (for binning)

y ordinate values (func is applied)

func a numpy ufunc that takes one argument, func(Y')

bins number or array

range limits (used with number of bins)

Returns

F,edges function and edges (`midpoints = 0.5*(edges[:-1]+edges[1:])`)

(This function originated as `recsql.sqlfunctions.regularized_function()`.)

1.4.3 Smoothing time series

Function *`numkit.timeseries.smooth()`* applies a window kernel to a time series and smoothes fluctuations. The number of points in the time series stays the same.

`numkit.timeseries.`**`smooth`**`(`*x*, *window_len=11*, *window='flat'*`)`

smooth the data using a window with requested size.

This method is based on the convolution of a scaled window with the signal. The signal is prepared by introducing reflected copies of the signal (with the window size) in both ends so that transient parts are minimized in the begining and end part of the output signal.

Arguments

x the input signal, 1D array

window_len the dimension of the smoothing window, always converted to an integer (using `int()`) and must be odd

window the type of window from 'flat', 'hanning', 'hamming', 'bartlett', 'blackman'; flat window will produce a moving average smoothing. If *window* is a `numpy.ndarray` then this array is directly used as the window (but it still must contain an odd number of points) ["flat"]

Returns the smoothed signal as a 1D array

Example

Apply a simple moving average to a noisy harmonic signal:

```
>>> import numpy as np
>>> t = np.linspace(-2, 2, 201)
>>> x = np.sin(t) + np.random.randn(len(t))*0.1
>>> y = smooth(x)
```

Source: based on http://www.scipy.org/Cookbook/SignalSmooth

`numkit.timeseries.`**`smoothing_window_length`**`(`*resolution*, *t*`)`

Compute the length of a smooting window of *resolution* time units.

Arguments

resolution length in units of the time in which *t* us supplied

t array of time points; if not equidistantly spaced, the mean spacing is used to compute the window length

Returns odd integer, the size of a window of approximately *resolution*

See also:

`smooth()`

1.4.4 Exceptions

`exception` `numkit.timeseries.`**`LowAccuracyWarning`**

Warns that results may possibly have low accuracy.

1.5 `numkit.integration` — Numerical integration of data

See also:

`scipy.integrate`

`numkit.integration.`**`simps_error`**(*dy*, *x=None*, *dx=1*, *axis=-1*, *even='avg'*)

Error on integral evaluated with Simpson's rule from errors of points, *dy*.

Evaluate the integral with `scipy.integrate.simps()`. For a given vector *dy* of errors on the function values, the error on the integral is calculated via propagation of errors from the Newton-Cotes formula for the 3rd Lagrange interpolating polynomial. The results are exact for the cases of even spacing *dx*; for uneven spacing we currently average all spacings (exact solution is in the works. . .)

Arguments

dy errors for the tabulated values of the integrand f

x values of abscissa at which f was tabulated (can be `None` and then *dx* should be provided)

dx constant spacing of the abscissa

axis axis in *dy* along which the data lies

even see `scipy.integrate.simps()` ('avg', 'first', 'last')

1.6 `numkit.observables` — Observables as quantities with errors

Example showing how to use *`QuantityWithError`*:

```
>>> from numkit.observables import QuantityWithError
>>> a = QuantityWithError(2.0, 1.0)
>>> a2 = QuantityWithError(2.0, 1.0)   # 2nd independent measurement of a
>>> a3 = QuantityWithError(2.0, 1.0)   # 3rd independent measurement of a
>>> b = QuantityWithError(-1, 0.5)
>>> a+a
4 (2)
>>> a+a2
4 (1.41421)
>>> (a+a+a)/3
2 (1)
>>> (a+a2+a3)/3
2 (0.57735)
>>> a/b
-2 (1.41421)
```

Note that each quantity has an identity: it makes a difference to the error of a combined quantity such as a+a if the inputs are independent measurements of the same.

See also:

Various packages that describe quantities with units, in particular quantities.

1.6.1 Classes

`class` `numkit.observables.`**`QuantityWithError`**(*value*, *error=None*, *qid=None*, ***kwargs*)

Number with error and basic error propagation arithmetic.

The quantity is assumed to be a mean of an observable (`value`) with an associated (Gaussian) error `error` (which is the sqrt of the variance `variance` of the data).

The covariance is not taken into account in the error propagation (i.e. all quantities are assumed to be uncorrelated) with the exception of the case of binary relations of the quantity with itself. For instance, a*a is correctly interpreted as a**2). However, this behaviour is not guaranteed to work for any complicated expression.

`value`
Value of the observable A, typically the mean of a number of observations, $\langle A \rangle$.

`variance`
Variance $\langle (A - \langle A \rangle)^2 \rangle$ of the observable. Changing the variance automatically changes the `error`.

`static asQuantityWithError``(`*other*`)`
Return a *`QuantityWithError`*.

If the input is already a *`QuantityWithError`* then it is returned itself. This is important because a new quantity x' would be considered independent from the original one x and thus lead to different error estimates for quantities such as x*x versus x*x'.

`astuple``()`
Return tuple (value,error).

`copy``()`
Create a new quantity with the same value and error.

`deepcopy``()`
Create an exact copy with the same identity.

`error`
Error of the observable.

The error is taken as the square root of the `variance` of the observations, $\sqrt{\langle (A - \langle A \rangle)^2 \rangle}$.

Changing the error automatically changes the `variance`.

`isSame``(`*other*`)`
Check if *other* is 100% correlated with *self*.

`True` if

- *other* is the same observable (instance)
- *other* was derived from *self* without using any other independent quantities with errors, e.g.

```
>>> a = QuantityWithError(1.0, 0.5)
>>> b = a**2 - a*2
>>> a.isSame(b)
True
```

`False` if

- *other* is a scalar (without an error), or
- *other* was computed from *self* without involvement of any other observables.

Limitations: How should one treat the case when a quantity is used again in an operation, e.g.

```
c = a + b
d = c/a
```

How to compute the error on d? What should the result for `c.isSame(a)` be?

class numkit.observables.**QID**

Identity of a *QuantityWithError*.

The basic idea:

```
QID(iterable) --> identity
QID() --> ``None``
```

The anonymous identity is `None`, anything else is a `frozenset()`.

The QID can contain arbitray (but unique) identifiers in the *iterable*; however, strings are treated as individual objects and *not* as iterables.

The error arithmetic encapsulated by the operator-overloading of *QuantityWithError* builds new QIDs by accumulating QIDs of the terms of the expression. In a certain sense, the "history" of a quantity becomes its "identity".

union(*x*)

Return the union of sets as a new set.

(i.e. all elements that are in either set.)

1.6.2 Functions

numkit.observables.**iterable**(*obj*)

Returns `True` if *obj* can be iterated over and is *not* a string.

numkit.observables.**asiterable**(*obj*)

Returns obj so that it can be iterated over; a string is *not* treated as iterable

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Bibliography

[FrenkelSmit2002] D. Frenkel and B. Smit, Understanding Molecular Simulation. Academic Press, San Diego 2002

Python Module Index

Index