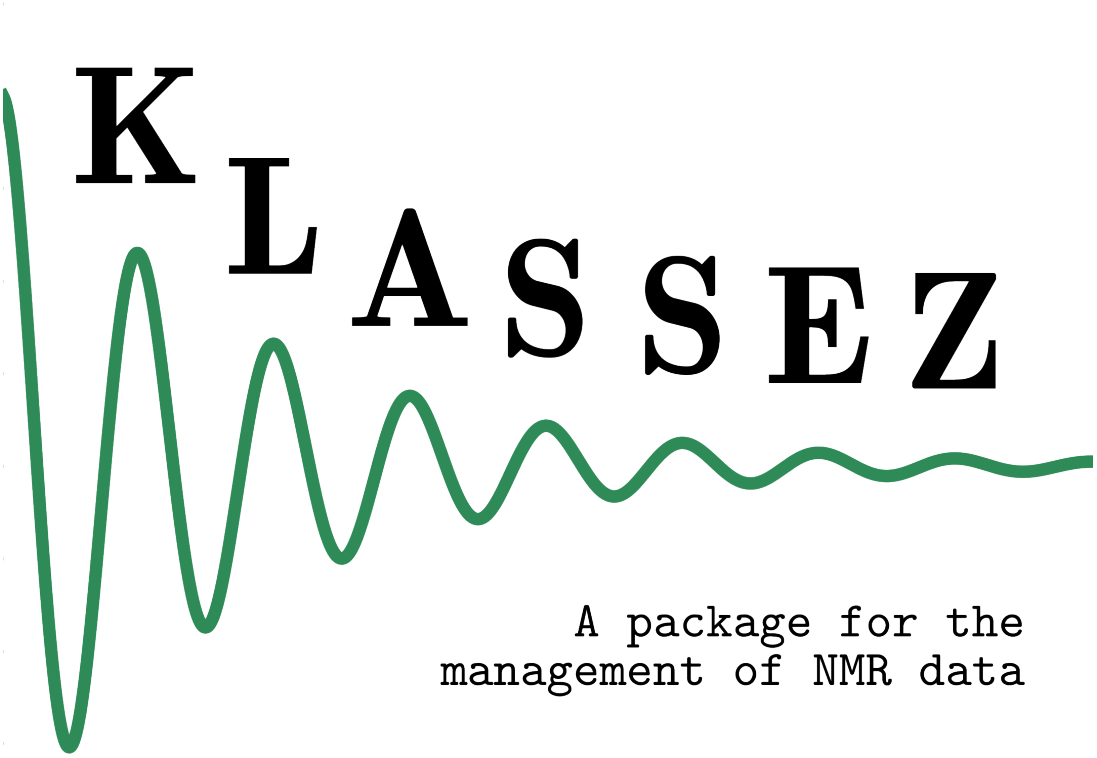

Author:
Francesco Bruno

Major contributors:
Letizia Fiorucci

Version: 0.3a.1

Documentation release date:
July 15, 2023

Contents

1. Introduction

KLASSEZ is a python package written to handle 1D and 2D NMR data. The aim of the project is to provide a toolkit, consisting of 'black-box' functions organized in modules, that could be used to read, process and analyze such data in a flexible manner, so to adapt to the needs of the individual users. However, the open-source nature of the package grants the user the chance to open the lid of these black-boxes and understand the gears that stand behind the function call.

The development of the toolkit started with `python 3.8` and therefore it is compatible with that version. Nevertheless, the use of `python 3.10` is advised.

The key objects provided by KLASSEZ are the classes `Spectrum_1D` and `Spectrum_2D`, that are able to fulfil the aims of the package with a few lines of code. The classes are able to read both simulated (i.e. generated with a custom-made input file) and experimental datasets. The latter feature was tested with Bruker data after the removal of the digital filter (run command `convdta` in TopSpin), but should be compatible with other kind of spectrometers, thanks to the remarkable work made by J. J. Helmus and coworkers with their `nmrglue` package[1]. Either the FID or the spectrum processed with external solver can be read from KLASSEZ by using the classes `Spectrum_nD` or `pSpectrum_nD`, respectively.

The `processing` module, besides the classical functions used for the processing of NMR data (window functions, Fourier transform, etc.), includes denoising algorithms based on Multivariate Curve Resolution[2] and on Cadzow method[3]. Details are illustrated in the description of the functions.

Functions to show and analyze data in real time are provided, with dedicated GUIs. However, it is better to rely on the standalone functions, enclosed in the single modules, to save the figures. In fact, the `figures` module offers a wide plethora of functions (all based on `matplotlib`) to plot the data with a high degree of customization for the appearance.

The fitting functions use `lmfit` to build the initial guess and to minimize the difference between the experimental data and the model, generated with a Voigt profile in the time domain and then Fourier-transformed, in the least-square sense (employing the Levenberg-Marquardt algorithm implemented in `scipy`). For this purpose, the class `Voigt_fit` of the `fit` module includes attribute functions to construct an initial guess interactively, fit the data, and save the parameters in dedicated files.

Regarding the development of the package, I would like to acknowledge Letizia Fiorucci for her contribution in the design and the implementation of several functions, and for the alpha-testing.

[1] `https://www.nmrglue.com/`

[2] Multivariate Curve Resolution: 50 years addressing the mixture analysis problem - A review

[3] Denoising NMR time-domain signal by singular-value decomposition accelerated by graphics processing units

2. User guide

2.1 Initialize the package

Initialize the package by writing, at the top of your file:

```
from klassez import *
```

This line also implies:

```
import os
import sys
import numpy as np
from scipy import linalg, stats
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from pprint import pprint
```

This means these can be not imported in your code, as `KLASSEZ` already does it for you.

2.1.1 Extra variables

Initializing `KLASSEZ` also grants access to `CM` and `COLORS`.

`CM` is a dictionary of colormaps taken from `seaborn` and saved in a dictionary whose keys are their names, so that also `matplotlib` can use them. You can inspect the keys through:

```
print(CM.keys())
```

`COLORS` is:

```
colors = [ 'tab:blue', 'tab:red', 'tab:green', 'tab:orange', 'tab:cyan', 'tab:purple',
    'tab:pink', 'tab:gray', 'tab:brown', 'tab:olive', 'salmon', 'indigo', 'm', 'c', 'g',
    'r', 'b', 'k', ]
```

repeated cyclically ten times and stored as tuple.

Other two 'quality of life' variables are `figures.figsize_small` and `figures.figsize_large`, which correspond to figure panel sizes of 3.59×2.56 inches and 15×8 inches, respectively. The former suits well for saving figures of spectra with font sizes of about 10 pt, whereas the latter are best for GUIs and withstand font sizes of about 14 pt.

For NMR: the variable `sim.gamma` is a dictionary containing the gyromagnetic ratio, in MHz/T, of all the magnetically-active nuclei. For instance:

```
print(sim.gamma['13C'])
>>> 10.70611
```

A decorator function called `cron` is defined in the top-level script `config`, and imported by `__init__`, so that you can use it after writing:

```
from klassez import cron
```

This decorator allows to measure the runtime of a function, and print it on standard output once it ended.

2.2 Processing of a 'raw' 1D spectrum

Let us say that your spectrum is saved in the folder */home/myself/spectra/mydataset/1/*. Initialize the spectrum object through:

```
Path = "/home/myself/spectra/mydataset/1/"
s = Spectrum_1D(Path)
```

This command will do three main tasks:

- read the binary FID of your spectrum and store it in a complex array `s.fid`;
- load the acquisition parameters, read the interesting keys and store them in a dictionary `s.acqus`;
- initialize a dictionary `s.procs` which contains the processing parameters.

If there is the group delay at the beginning of the FID, you are advised to write

```
s.convdta()
```

to remove it. However, this is tested for Bruker data and does not always work, because it depends on the version of TopSpin that the spectrometer ran and several other construction parameters. Therefore, if you see a residual of the digital filter in you spectrum, the easiest solution is to run `CONVDTA` inside TopSpin.

`KLASSEZ` is able to read also Varian and Spinsolve (Magritek) data, by specifying the option `'spect'`.

A detailed description of `acqus` and `procs` is shown in table 2.1 and table 2.2.

Please note that reading the spectrum causes the program to save a file called *'name.procs'*, where 'name' is the path name.

To make the Fourier transform of the FID to obtain the spectrum, you must invoke the `process` method, which reads the `procs` dictionary to get the instructions on the processing you want to make on your spectrum. For instance, if you want to obtain a final spectrum of $8k$ points with an exponential broadening of 25 Hz:

```
s.procs["wf"]["mode"] = "em"
s.procs["wf"]["lb"] = 25
s.procs["zf"] = 8192
s.process()
```

Calling the `process` method generates new attributes of the class:

- `freq`: the frequency scale, in Hz;
- `ppm`: the ppm scale;
- `r`: the real part of the spectrum;

Table 2.1: Description of the `acqus` dictionary of a `Spectrum_1D` object.

Key	Explanation
`B0`	Magnetic field strength /T
`BYTORDA`	Endianness of binary data: `0` little endian, `1` big endian
`DTYPA`	Binary data type: `0` *int32*, `2` *float64*
`GRPDLY`	Number of points of the digital filter
`nuc`	Observed nucleus
`o1p`	Carrier frequency i.e. center of the spectrum, in ppm
`o1`	Same as `o1p`, but in Hz
`SWp`	Sweep width, in ppm
`SW`	Sweep width, in Hz
`SFO1`	Larmor frequency of the observed nucleus at field `B0`
`TD`	Number of sampled complex points
`dw`	Dwell time, i.e. the sampling interval, in seconds
`AQ`	Time duration of the FID
`t1`	Acquisition timescale

- `i`: the imaginary part of the spectrum;
- `S`: the complex spectrum (`S` = `r` + `i`i).

After the Fourier transform, the `process` method applies the phase correction and the calibration using the phase angles and the calibration value saved in the *procs* dictionary automatically. This allows the user to not phase their spectra every time, as well as keeping a record of the processing.

If the spectrum requires phase correction, you can perform it interactively:

```
s.adjph()
```

or by passing the phase angles, in degrees, to `adjph`. Example, if you know you need to phase your spectrum with 30 degrees of $\phi^{(0)}$ and -55 degrees of $\phi^{(1)}$ with the pivot set at 7.32 ppm:

```
s.adjph(p0=30, p1=-55, pv=7.32)
```

In both cases, the phase angles are updated in the `procs` dictionary.

The spectrum can be calibrated using a dedicated GUI:

```
s.cal()
```

or specifying the shift value in ppm or in Hz (in this case, be sure to set the `isHz` keyword to `True`).

```
s.cal(-3)             # Shift of -3 ppm
s.cal(1000, isHz=True) # Shift of +1 kHz
```

Both `ppm` and `freq` are updated according to the given values.

A tool for baseline correction and lineshape fitting are also provided. Regarding the baseline correction, there are now two methods available: the 'old' one, named `baseline_correction`, allows to build and read a manually-designed baseline; the `rpbc` method automatically tries to fit the baseline with a polynomion and correct the phase together.

```
s.basl()          # subtract s.baseline from s.S, then unpack s.r and s.i
```

For the fitting:

Table 2.2: Description of the `procs` dictionary of a `Spectrum_1D` object.

Key	Explanation
`wf`	Window function. This is a dictionary itself: • `'mode'`: choose function between – `'em'`: exponential – `'sin'`: sine – `'qsin'`: squared sine – `'gm'`: mixed lorentzian-gaussian – `'gmb'`: mixed lorentzian-gaussian, Bruker style • `'lb'`: Exponential line-broadening. Read by `em`, `gm` and `gmb` • `'gb'`: Gaussian line-broadening. Read by `gm` and `gmb` • `'gc'`: Center of the gaussian $\in [0, 1]$. Read by `gm` • `'ssb'`: Shift of the sine bell. Read by `sin` and `qsin` • `'sw'`: Sweep width. Automatically set according to `acqus['SW']`
`zf`	Zero-filling. Set the *final* number of points!
`tdeff`	Number of points to be used for processing
`fcor`	Scaling factor for the first point of the FID before Fourier transform
`p0`	Frequency-independent phase correction /degrees
`p1`	First order phase correction /degrees
`pv`	Pivot point for the first order phase correction /ppm
`basl_c`	Set of coefficients of a polynomion to be used as baseline, starting from the 0-order coefficient
`cal`	Offset, in ppm, to be added to the frequency and ppm scales for calibration

```
# Make an initial guess interactively and save all the parameters in a file called
    "myguess.inp"
s.F.iguess(input_file="myguess.inp")
# Save a figure of the initial guess in "myguess.png"
s.F.plot(what='iguess', name="myguess")
# Do the fit
s.F.dofit(
    log_file="myfit.log",     # Write the log of the fit in a file called "myfit.log"
    output_file="myfit.out", # Write the output of my fit in a file called "myfit.out"
    utol=0.5,                # Tolerance on the chemical shift of +/-0.5 ppm
    vary_phi=False,          # Allow/Not allow to fit dephased peaks
    vary_xg=False,           # Allow/Not allow to change fraction of gaussianity
    res_hist_name="myres",   # Make a figure of the residuals called "myres.png"
    test_res=True            # Perform tests on the goodness of the fit
    )
# Plot the output and show it
s.F.plot(what='fit')
```

2.2.1 The class pSpectrum_1D

The class Spectrum_1D does not work if you want to read the processed data directly from TopSpin (or whatever software you used to acquire and process them). Instead, you should use the class pSpectrum_1D, which is designed to perform exactly this task. It inherits most of the attributes and methods of the Spectrum_1D class, therefore its usage closely resembles the example reported in the previous section.

2.3 Processing of a 'raw' 2D spectrum

Let us say that your spectrum is saved in the folder */home/myself/spectra/mydataset/21/*. Initialize the spectrum object through:

```
Path = "/home/myself/spectra/mydataset/21/"
s = Spectrum_2D(Path)
```

The generated acqus and procs dictionaries include informations on both dimensions.

Table 2.3: Description of the acqus dictionary of a Spectrum_2D object.

Key	Explanation
BO	Magnetic field strength /T
BYTORDA	Endianness of binary data: 0 little endian, 1 big endian
DTYPA	Binary data type: 0 *int32*, 2 *float64*
GRPDLY	Number of points of the digital filter
nuc1	Observed nucleus in the indirect dimension
nuc2	Observed nucleus in the direct dimension
o1p	Carrier frequency i.e. center of the indirect dimension, in ppm
o2p	Carrier frequency i.e. center of the direct dimension, in ppm
o1	Same as o1p, but in Hz
o2	Same as o2p, but in Hz
SW1p	Sweep width of the indirect dimension, in ppm
SW2p	Sweep width of the direct dimension, in ppm
SW1	Sweep width of the indirect dimension, in Hz
SW2	Sweep width of the indirect dimension, in Hz
SFO1	Larmor frequency of the observed nucleus in F1 at field BO
SFO2	Larmor frequency of the observed nucleus in F2 at field BO
TD1	Number of t_1-increments
TD2	Number of sampled complex points
dw1	t_1 increments, in seconds
dw2	Dwell time, i.e. the sampling interval, in seconds
AQ1	Sampled timescale of the indirect dimension
AQ2	Time duration of the FID
t1	Evolution timescale
t2	Acquisition timescale

Then, the sequence of commands resembles the ones of the 1D spectra.

```
s.convdta()    # If there is the digital filter
s.process()
# Also in this case, phase correction and calibration are performed automatically with
    the values in procs
```

Table 2.4: Description of the `procs` dictionary of a `Spectrum_2D` object. Each of these dictionary entry is a list of two elements: the first one (index `0`) is the processing to apply on the indirect dimension, the second (index `1`) on the direct dimension. For instance, `procs[tdeff] = [64, 1024]` means to truncate the indirect evolutions to 64 points and the FIDs to 1024 points.

Key	Explanation
`wf`	Window function. This is a dictionary itself: • `'mode'`: choose function between – `'em'`: exponential – `'sin'`: sine – `'qsin'`: squared sine – `'gm'`: mixed lorentzian-gaussian – `'gmb'`: mixed lorentzian-gaussian, Bruker style • `'lb'`: Exponential line-broadening. Read by `em`, `gm` and `gmb` • `'gb'`: Gaussian line-broadening. Read by `gm` and `gmb` • `'gc'`: Center of the gaussian $\in [0, 1]$. Read by `gm` • `'ssb'`: Shift of the sine bell. Read by `sin` and `qsin` • `'sw'`: Sweep width. Automatically set according to `acqus['SW']`
`zf`	Zero-filling. Set the *final* number of points!
`tdeff`	Number of points to be used for processing
`fcor`	Scaling factor for the first point of the FID before Fourier transform
`p0`	Frequency-independent phase correction /degrees
`p1`	First order phase correction /degrees
`pv`	Pivot point for the first order phase correction /ppm
`cal_1`	Calibration offset for F1 /ppm
`cal_2`	Calibration offset for F2 /ppm

```
s.adjph()
s.plot()
```

The keys for `adjph` are of the kind: `pXY`, where `X` is the order of the phase correction (0 or 1) and `Y` is the dimension on which to apply it (1 or 2). Explicative table below:

	F1	**F2**
$\phi^{(0)}$	`p01`	`p02`
$\phi^{(1)}$	`p11`	`p12`
pivot	`pv1`	`pv2`

For further information, rely on the `help` python builtin function.
To read the processed data, use the `pSpectrum_2D` class instead.

2.3.1 Computing projections

While the 2D spectra give an overall look on the whole experiment, the user might want to extract projection of the direct or the indirect dimension, to focus onto particular features in the spectrum. In order to do so, `klassez` offers two commands: `projf1` and `projf2`, which compute the sum projections on the indirect or on the direct dimension, respectively, and store the result in dictionaries called `trf1` and `trf2`, whose keys are the ppm values correspondant to the projections. Actually, the capitalized versions of the two dictionaries (with the same keys), i.e. `Trf1` and `Trf2`, can be more useful, as they are instances of the `pSpectrum_1D` class and therefore are initialized with ppm scales and other parameters.

Example:

```
# Supposed to have a 1H-15N HSQC spectrum

#  Extract the direct dimension trace at 115 ppm, 15N scale
s.projf2(115)
#  Access to it through
Proj_115 = s.Trf2['115']

#  Extract the indirect dimension trace from 6 to 8 ppm, 1H scale
s.projf1(6, 8)
Proj_indim = s.Trf1['6:8']

# You can plot them:
Proj_115.plot()
Proj_indim.plot()
```

2.4 Simulating data

The classes `Spectrum_1D` and `Spectrum_2D` are also able to generate simulated data by reading a custom-written input file. The functions they use are `sim.sim_1D` and `sim.sim_2D`.

2.4.1 Simulate 1D data

The input file you have to write *must* have the following keys:

- `B0`: Magnetic field strength /T;
- `nuc`: Observed nucleus (e.g. `13C`);
- `o1p`: Carrier frequency i.e. centre of the spectrum /ppm;
- `SWp`: Sweep width /ppm. The spectrum will cover the range $[\texttt{o1p} - \texttt{SWp}/2, \texttt{o1p} + \texttt{SWp}/2]$;
- `TD`: Number of sampled (complex) points;
- `shifts`: sequence of peak positions /ppm;
- `fwhm`: Full-width at half-maximum of the peaks /Hz;
- `amplitudes`: Intensity of the peaks in the FID;
- `x_g`: Fraction of gaussianity. $x_g = 0 \implies$ pure Lorentzian peak, $x_g = 1 \implies$ pure Gaussian peak;

and *can* have the following keys:

- `phases`: phases of the peaks /degrees. Default: all zeros;
- `mult`: fine structures of the peaks (e.g. doublets of triplets: `dt`). Default: all singlets;
- `Jconst`: coupling constants of the fine structures /Hz. If more of one coupling is expected, provide them as a sequence. Default: not used as the peaks are all singlets.

Key and value must be separated by a tab character. You are allowed to leave empty rows to improve the readibility and to insert comments using the # character.

Example:

```
B0  16.4    # 700 MHz 1H
nuc 1H
o1p 4.7
SWp 40
TD  8192

shifts 1, 3, 5, 7
fwhm   [10 for k in range(4)]
amplitudes 10, 20, 15, 10
x_g 0, 0.4, 0.6, 1
phases 5, 0, 10, 0

mult   s, t, dt, ddd
Jconst 0, 15, [12, 9.5], [25, 15, 10]
```

This input file generates the spectrum in Figure 2.1.

Code:

```
#! /usr/bin/env python3

from klassez import *

s = Spectrum_1D('sim_in_1D', isexp=False)
s.process()

figures.figure1D(s.ppm, s.r, name='test_1D', X_label='$\delta\, ^1$H /ppm',
    Y_label='Intensity /a.u.')
```

2.4.2 Simulate 2D data

The same procedure can be followed to simulate 2D spectra. The input file to write is very similar to the one for 1D data, except for the quantities that clearly span over two dimensions. As in NMR textbook, the direct and indirect dimensions will be named F2 and F2 respectively, and dimension-specific quantities will feature the 1 or 2 labels accordingly.

- `B0`: Magnetic field strength /T;
- `nuc1`: Observed nucleus in F1(e.g. `13C`);
- `nuc2`: Observed nucleus in F2(e.g. `1H`);
- `o1p`: Carrier frequency i.e. centre of F1 /ppm;

- `o2p`: Carrier frequency i.e. centre of F2 /ppm;
- `SW1p`: Sweep width /ppm. The indirect dimension will cover the range $[\texttt{o1p} - \texttt{SW1p}/2, \texttt{o1p} + \texttt{SW1p}/2]$;
- `SW2p`: Sweep width /ppm. The direct dimension will cover the range $[\texttt{o2p} - \texttt{SW2p}/2, \texttt{o2p} + \texttt{SW2p}/2]$;
- `TD1`: Number of sampled (complex) points in F1;
- `TD2`: Number of sampled (complex) points in F2;
- `shifts_f1`: sequence of peak positions in F1 /ppm;
- `shifts_f2`: sequence of peak positions in F2 /ppm;
- `fwhm_f1`: Full-width at half-maximum of the peaks in F1 /Hz;
- `fwhm_f2`: Full-width at half-maximum of the peaks in F2 /Hz;
- `amplitudes`: Intensity of the peaks in the FID;
- `x_g`: Fraction of gaussianity. $x_g = 0 \implies$ pure Lorentzian peak, $x_g = 1 \implies$ pure Gaussian peak;

Phase distortions and fine structures are not allowed for multidimensional spectra. The indirect dimension will be generated employing the *States-TPPI* sampling scheme.

Example:

```
B0  28.2
nuc1   15N
nuc2   1H
o1p 115
o2p 5
SW1p   40
SW2p   20
TD1 512
TD2 8192

shifts_f1  130.0, 105.0, 120.0, 1.25e2, 130.0, 105.0
shifts_f2  0.0, 0.0, 4.0, 7.0, 1.1e1, 10.5
fwhm_f1 100, 100, 100, 100, 100, 100
fwhm_f2 50, 50, 50, 50, 50, 50
amplitudes 10, 20, 10, 20, 10, 10
x_g 0.0, 0.2, 0.4, 0.6, 0.8, 1.0
```

This input file generates the spectrum in Figure 2.2.

Code:

```
#! /usr/bin/env python3

from klassez import *

s = Spectrum_2D('sim_in_2D', isexp=False)
s.process()

figures.figure2D(s.ppm_f2, s.ppm_f1, s.rr, lvl=0.005, name='test_2D', X_label='$\delta\,
    ^1$H /ppm', Y_label='$\delta\, ^{15}$N /ppm')
```

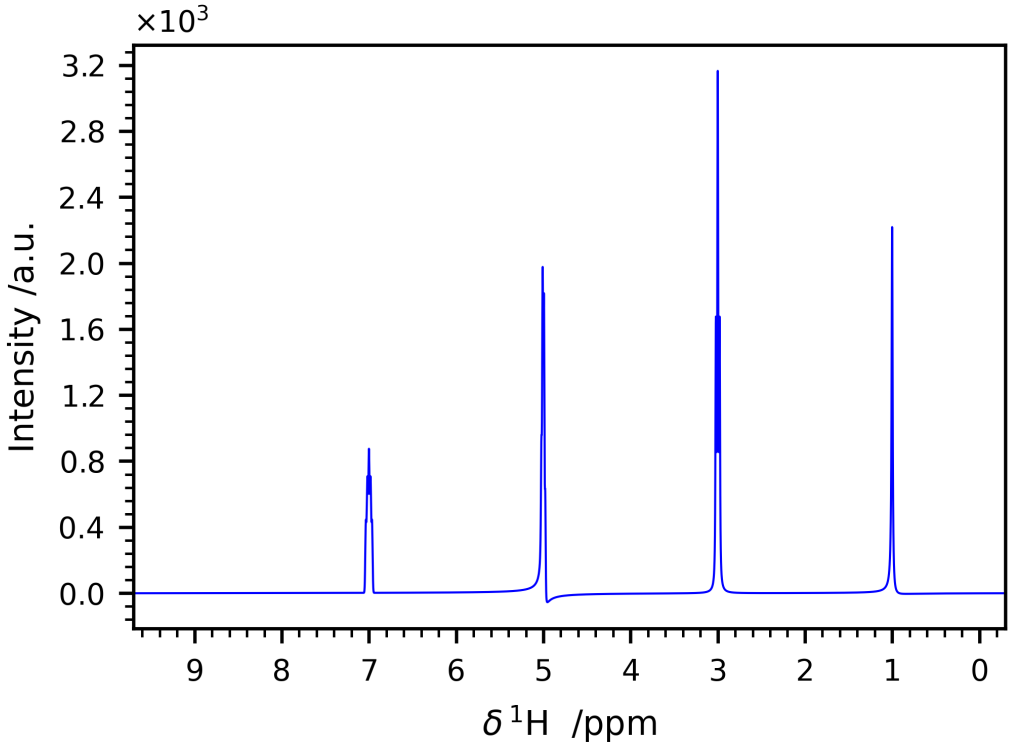

Figure 2.1: Example of a simulated 1D spectrum.

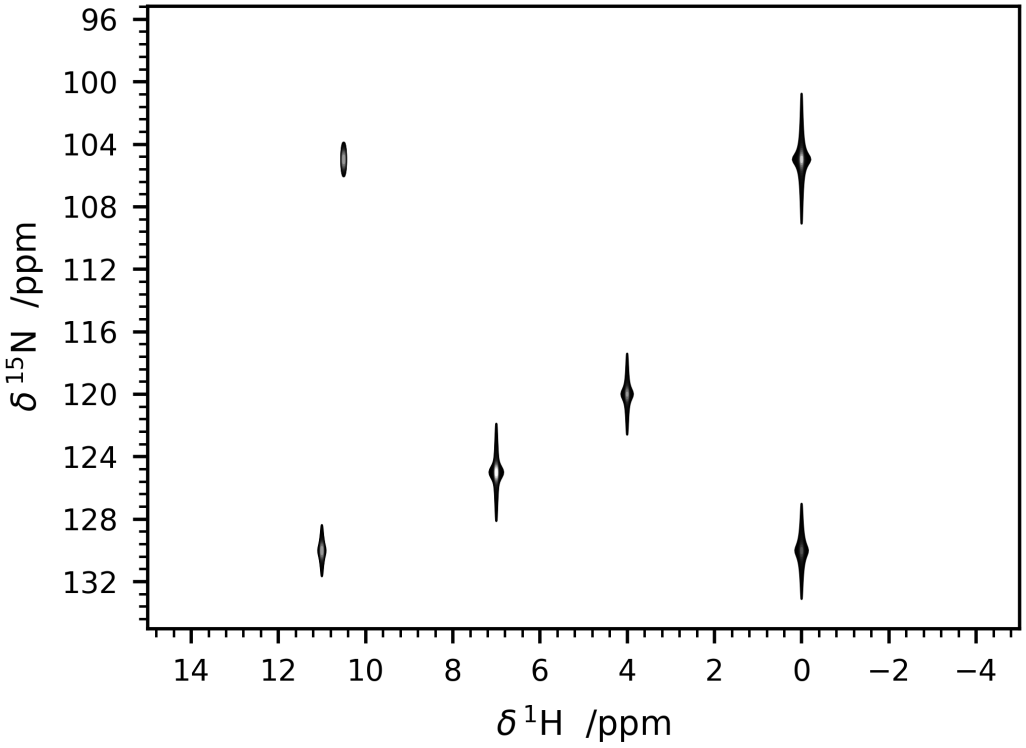

Figure 2.2: Example of a simulated 2D spectrum.

2.5 The Pseudo_2D class

Sometimes, the spectroscopist might find interesting to acquire a series of 1D experiments in which one (or more) parameters are changed according to a certain schedule. This kind of experiments are 2D in principle, but their processing and analysis resemble the one of 1D spectra. Therefore, they lie somewhere in between 1D spectra and 2D spectra, hence they are often referred to as *pseudo_2D*.

Also in this case, `klassez` offers a specific class to deal with this kind of data: `Pseudo_2D`. `Pseudo_2D` is a subclass of `Spectrum_2D`; however, many functions have been adapted to resemble the 1D version.

`Pseudo_2D` does not encode for a routine to automatically simulate data. If you want to, you should give a 1D-like input file (just like the one in section 2.4.1), and replace the attribute `fid` with your FID, generated as you wish. With a real dataset this is not required, as it is able to read everything automatically.

```
path_to_pseudo = "/home/myself/spectra/mydataset/899/"
s = Pseudo_2D(path_to_pseudo)
s.convdta() # If there is the digital filter
```

The `process()` function applies apodization, zero-filling and Fourier transform only on the direct dimension, reading the parameters from a `procs` dictionary like the one of `Spectrum_1D`. The attributes `freq_f1` and `ppm_f1` are initialized with `np.arange(`N`)`, where N is the number of experiments that your FID comprises of.

The phase adjustment is performed on a reference spectrum, then applied on the whole 2D matrix. By default, the chosen spectrum is the first one, but you can choose the one that fits the most your needs.

```
s.process()
s.adjph(expno = 10) # Calls interactive_phase_1D on the 10th experiment
```

The method `plot` shows the 2D contour map of the spectrum, just like the one of `Spectrum_2D`. However, this is not always the most intelligent way to plot the data in order to gather information. This is the reason why this class features two unique additional methods that plot data: `plot_md` and `plot_stacked`. Both rely on the parameter `which`, that is a string of code (i.e. it should be interpreted by `eval`) that identifies which experiment to show by pointing at their index. `which = 'all'` results in pointing at all spectra.

```
s.plot()   # 2D contour map
s.plot_md(which="3, 5, 11")  # Plot the 3rd, the 5th and the 11th spectrum, superimposed
s.plot_stacked(which="np.arange(0,100,5)") # Makes a stacked plot with a spectrum every 5
```

The method `integrate` differs a little bit from the one coded in `Spectrum_1D`.

```
s.integrate(which=2)      # Interactive panel on the 3rd spectrum
```

Even if you select the integration limits on a single spectrum, the method `integrate` will compute the integrals throughout the whole range of experiment. This means that each entry of `integrals` will be an array as long as the number of experiment.

3. List of modules and functions

3.1 MISC package

This package contains miscellaneous functions for the calculation of several properties, and generally for the handling of NMR spectra.

3.1.1 misc.SNR(data, signal=None, n_reg=None)

Computes the signal to noise ratio of a 1D spectrum, as:

$$SNR = \frac{S}{2\sigma_n}$$

Parameters

- data : *1darray*
  The spectrum of which you want to compute the SNR
- signal : *float, optional*
  If provided, uses this value as maximum signal. Otherwise, it is selected as the maximum value in `data`
- n_reg : *list or tuple, optional*
  If provided, contains the points that delimit the noise region. Otherwise, the whole spectrum is used.

Returns

- snr : *float*
  The SNR of the spectrum

3.1.2 misc.SNR_2D(data, n_reg=None)

Computes the signal to noise ratio of a 2D spectrum.

Parameters

- data : *1darray*
  The spectrum of which you want to compute the SNR
- n_reg : *list or tuple*
  If provided, the points of F1 scale and F2 scale, respectively, of which to extract the projections. Otherwise, opens the tool for interactive selection.

Returns

- snr_f1 : *float*
  The SNR of the indirect dimension
- snr_f2 : *float*
  The SNR of the direct dimension

3.1.3 misc.avg_antidiag(X)

Given a matrix `X` without any specific structure, finds the closest Hankel matrix in the Frobenius norm sense by averaging the antidiagonals.

Parameters:

- X: *2darray*
  Input matrix

Returns:

- Xp: *2darray*
  Hankel matrix obtained from `X`

3.1.4 misc.binomial_triangle(n)

Calculates the n-th row of the binomial triangle. The first row is n=1, not 0. Example:

```
binomial_triangle(4)
>>> 1 3 3 1
```

Parameters:

- n: *int*
  Row index

Returns:

- row: *1darray*
  The n-th row of binomial triangle.

3.1.5 misc.calcres(fqscale)

Calculates the frequency resolution of an axis scale, i.e. how many Hz is a 'tick'.

Parameters

- fqscale : *1darray*
  Scale to be processed

Returns

- res : *float*
  The resolution of the scale

3.1.6 misc.cmap2list(cmap, N=10, start=0, end=1)

Extract the colors from a colormap and return it as a list.

Parameters:

- cmap: *matplotlib.Colormap Object*
  The colormap from which you want to extract the list of colors
- N: *int*
  Number of samples to extract
- start: *float*
  Start point of the sampling. 0 = beginning of the cmap; 1 = end of the cmap.
- end: *float*
  End point of the sampling. 0 = beginning of the cmap; 1 = end of the cmap.

Returns:

- colors: *list*
  List of the extracted colors.

3.1.7 misc.edit_checkboxes(checkbox, xadj=0, yadj=0, length=None, height=N color=None)

Edit the size of the box to be checked, and adjust the lines accordingly.

Parameters:

- checkbox: *matplotlib.widgets.CheckBox Object*
  The checkbox to edit
- xadj: *float*
  modifier value for bottom left corner x-coordinate of the rectangle, in checkbox.ax coordinates
- yadj: *float*
  modifier value for bottom left corner y-coordinate of the rectangle, in checkbox.ax coordinates
- length: *float*
  length of the rectangle, in checkbox.ax coordinates
- height: *float*
  height of the rectangle, in checkbox.ax coordinates
- color: *str or list or None*
  If it is not `None`, change color to the lines

3.1.8 misc.find_nearest(array, value)

Finds the value in `array` which is the nearest to `value`.

Parameters

- array : *1darray*
  Self-explanatory
- value : *float*
  Value to be found

Returns

- val : *float*
  The closest value in `array` to `value`

3.1.9 misc.freq2ppm(x, B0=701.125, o1p=0)

Converts `x` from Hz to ppm.

Parameters

- x : *float*
  Value to be converted
- B0 : *float*
  Field frequency, in MHz. Default: 700 MHz
- o1p : *float*
  Carrier frequency, in ppm. Default: 0.

Returns

- y : *float*
  The converted value

3.1.10 misc.get_trace(data, ppm_f2, ppm_f1, a, b=None, column=True)

Takes as input a 2D dataset and the ppm scales of direct and indirect dimensions respectively. Calculates the projection on the given axis summing from `a` (ppm) to `b` (ppm). Default: indirect dimension projection (i.e. `column=True`), change it to '`False`' for the direct dimension projection. Returns the calculated 1D projection.

Parameters

- data : *2darray*
  Spectrum of which to extract the projections
- ppm_f2 : *1darray*
  ppm scale of the direct dimension
- ppm_f1 : *1darray*
  ppm scale of the indirect dimension
- a : *float*
  The ppm value from which to start extracting the projection.
- b : *float, optional*
  If provided, the ppm value at which to stop extracting the projection. Otherwise, returns only the `a` trace.
- column : *bool*
  If `True`, extracts the F1 projection. If `False`, extracts the F2 projection.

Returns

- y : *1darray*
  Computed projection

3.1.11 misc.get_ylim(data_inp)

Calculates the y-limits of a plot as follows:

- Bottom: `min(data) - 5% max(data)`
- Top: `max(data) + 5% max(data)`

Parameters:

- data_inp: *ndarray or list*
  Input data. If it is a list, data_inp is converted to array.

3.1.12 misc.hankel(data, n=None)

Computes a Hankel matrix from `data`. If `data` is a `1darray` of length N, computes the correspondant Hankel matrix of dimensions `(`$N - n + 1$`,` n`)`. If `data` is a `2darray`, computes the closest Hankel matrix in the Frobenius norm sense by averaging the values on the antidiagonals.

Parameters:

- data: *1darray*
  Vector to be Hankel-ized, of length N
- n: *int*
  Number of columns that the Hankel matrix will have

Returns:

- H: *2darray*
  Hankel matrix of dimensions `(`$N - n + 1$`,` n`)`

3.1.13 misc.hz2pt(fqscale, hz)

Converts `hz` from frequency units to points, on the basis of its scale.

Parameters

- fqscale : *1darray*
  Scale to be processed
- hz : *float*
  Value to be converted

Returns

- pt : *float*
  The frequency value converted in points

3.1.14 misc.in2px(*in_args)

Converts a sequence of numbers from inches to pixels by multiplying ×96.

Parameters:

- in_args: *sequence of floats*
  Values in inches to convert

Returns:

- px_args: *tuple of ints*
  Values in pixels

3.1.15 misc.makeacqus_1D(dic)

Given a NMRGLUE dictionary from a 1D spectrum (generated by `ng.bruker.read`), this function builds the `acqus` file with only the 'important' parameters.

Parameters

- dic : *dict*
  NMRglue dictionary returned by `ng.bruker.read`

Returns

- acqus : *dict*
  Dictionary with only few parameters

3.1.16 misc.makeacqus_2D(dic)

Given a NMRGLUE dictionary from a 2D spectrum (generated by `ng.bruker.read`), this function builds the `acqus` file with only the 'important' parameters.

Parameters

- dic : *dict*
  NMRglue dictionary returned by `ng.bruker.read`

Returns

- acqus : *dict*
  Dictionary with only few parameters

3.1.17 misc.mathformat(ax, axis='y', limits=(-2,2))

Apply exponential formatting to the given axis of the given figure panel. The offset text size is uniformed to the tick labels' size.

Parameters:

- ax: *matplotlib.Subplot Object*
  Panel of the figure to edit
- axis: *str*
  `'x'`, `'y'` or `'both'`.
- limits: *tuple*
  tuple of ints that indicate the order of magnitude range outside which the exponential format is applied.

3.1.18 misc.molfrac(n)

Computes the 'molar fraction' `x` of the array `n`. Returns also the total amount.

$$x_i = \frac{n_i}{N}, \qquad N = \sum_i n_i$$

Parameters

- n: *list or 1darray*
  list of values

Returns

- x: *list or 1darray*
  molar fraction array
- N: *float*
  sum of all the elements in `n`

3.1.19 misc.noise_std(y)

Calculates the standard deviation of the noise using the Bruker formula. Taken y as an array of N points, and y_i its i-th entry:

$$\sigma_n = \frac{1}{\sqrt{N-1}}\sqrt{\sum_{i=1}^{N} y_i^2 - \frac{1}{N}\left[\left(\sum_{i=1}^{N} y_i\right)^2 + \frac{3\left(\sum_{i=1}^{N/2} i(y_{N-i+1} - y_i)\right)^2}{N^2-1}\right]}$$

Parameters

- y : *1darray*
  The spectral region you would like to use to calculate the standard deviation of the noise.

Returns

- noisestd : *float*
  The standard deviation of the noise.

3.1.20 misc.nuc_format(nuc)

Converts the `nuc` key you may find in `acqus` in the formatted label, e.g. `'13C'` → `'${^13}$C'`

Parameters:

- nuc: *str*
  Unformatted string

Returns:

- fnuc: *str*
  Formatted string.

3.1.21 misc.polyn(x, c)

Computes $p(x)$, polynomion of degree $n-1$, where n is the number of provided coefficients.

Parameters

- x : *1darray*
  Scale upon which to build the polynomion
- c : *list or 1darray*
  Sequence of the polynomion coeffiecient, starting from the 0-th order coefficient

Returns

- px : *1darray*
  Polynomion of degree $n-1$.

3.1.22 misc.ppm2freq(x, B0=701.125, o1p=0)

Converts `x` from ppm to Hz.

Parameters

- x : *float*
  Value to be converted
- B0 : *float*
  Field frequency, in MHz. Default: 700 MHz
- o1p : *float*
  Carrier frequency, in ppm. Default: 0.

Returns

- y : *float*
  The converted value

3.1.23 misc.ppmfind(ppm_scale, value)

Finds the exact `value` in `ppm scale`.

Parameters

- ppm_scale : *1darray*
  Self-explanatory
- value : *float*
  The value to be found

Returns

- I : *int*
  The index correspondant to `V` in `ppm_scale`
- V : *float*
  The closest value to `value` in `ppm_scale`

3.1.24 misc.pretty_scale(ax, limits, axis='x', n_major_ticks=10)

This function computes a pretty scale for your plot. Calculates and sets a scale made of `n_major_ticks` numbered ticks, spaced by `5 * n_major_ticks` unnumbered ticks. After that, the plot borders are trimmed according to the given limits.

Parameters

- ax: *matplotlib.AxesSubplot object*
  Panel of the figure of which to calculate the scale
- limits: *tuple*
  limits to apply of the given axis. `(left, right)`
- axis: *str*
  `'x'` for x-axis, `'y'` for y-axis
- n_major_ticks: *int*
  Number of numbered ticks in the final scale. An oculated choice gives very pleasant results.

3.1.25 misc.print_dict(mydict)

Prints a dictionary one entry per row, in the format `key:  value`. Nested dictionaries are printed with an indentation

Parameters:

- mydict: *dict*
  The dictionary you want to print

Returns:

- outstring: *str*
  The printed text formatted as single string

3.1.26 misc.print_list(mylist)

Prints a list, one entry per row.

Parameters:

- mylist: *list*
  The list you want to print

Returns:

- outstring: *str*
  The printed text formatted as single string

3.1.27 misc.procpar(txt)

Takes as input the path of a file containing a 'key' in the first column and a 'value' in the second column. Returns a dictionary of shape `key : value`.

Parameters

- txt : *str*
  Path to a file that contains 'key' in first column and 'value' in the second

Returns

- procpars : *dict*
  Dictionary of shape `key : value`.

3.1.28 misc.px2in(*px_args)

Converts a sequence of numbers from inches to pixels by multiplying ×96.

Parameters:

- px_args: *sequence of ints*
  Values in pixels to convert

Returns:

- in_args: *tuple of floats*
  Values in inches

3.1.29 misc.readlistfile(datafile)

Takes as input the path of a file containing one entry for each row. Returns a list of the aforementioned entries.

Parameters

- datafile: *str*
  Path to a file that contains one entry for each row

Returns

- files : *list*
  List of the entries contained in the file

3.1.30 misc.select_for_integration(ppm_f1, ppm_f2, data, Neg=True)

Select the peaks of a 2D spectrum to integrate. First, select the area where your peak is located by dragging the red square. Then, select the center of the peak by right_clicking. Finally, click 'ADD' to store the peak. Repeat the procedure for as many peaks as you want.

Parameters:

- ppm_f1 : *1darray*
  ppm scale of the indirect dimension
- ppm_f2 : *1darray*
  ppm scale of the direct dimension
- data : *2darray*
  Spectrum
- Neg : *bool*
  Choose if to show the negative contours (`True`) or not (`False`)

Returns:

- peaks: *list of dict*
  For each peak there are two keys, `'f1'` and `'f2'`, whose meaning is obvious. For each of these keys, you have `'u'`: center of the peak /ppm, and `'lim'`: the limits of the square you drew before.

3.1.31 misc.select_traces(ppm_f1, ppm_f2, data, Neg=True, grid=False)

Select traces from a 2D spectrum, save the coordinates in a list. Left click to select a point, right click to remove it.

Parameters

- ppm_f1 : *1darray*
  ppm scale of the indirect dimension
- ppm_f2 : *1darray*
  ppm scale of the direct dimension
- data : *2darray*
  Spectrum
- Neg : *bool*
  Choose if to show the negative contours (`True`) or not (`False`)
- grid : *bool*
  Choose if to display the grid (`True`) or not (`False`)

Returns

- coord: *list*
  List containing the `[x,y]` coordinates of the selected points.

3.1.32 misc.set_fontsizes(ax, fontsize=10)

Automatically adjusts the fontsizes of all the figure elements. In particular:

- title = `fontsize`
- axis labels = `fontsize` -2
- ticks labels = `fontsize` -3
- legend entries = `fontsize` -4

Parameters:

- ax: *matplotlib.Subplot Object*
  Subplot of interest
- fontsize: *float*
  Starting fontsize

3.1.33 misc.set_ylim(ax, data_inp)

Sets the y-limits of `ax` as follows:

- Bottom: `min(data) − 5% max(data)`
- Top: `max(data) + 5% max(data)`

Parameters:

- ax: *matplotlib.Subplot Object*
  Panel of the figure where to apply this scale
- data_inp: *ndarray or list*
  Input data. If it is a list, `data_inp` is converted to array.

3.1.34 misc.show_cmap(cmap, N=10, start=0, end=1)

Plot the colors extracted from a colormap.

Parameters:

- cmap: *matplotlib.Colormap Object*
  The colormap from which you want to extract the list of colors
- N: *int*
  Number of samples to extract
- start: *float*
  Start point of the sampling. 0 = beginning of the cmap; 1 = end of the cmap.
- end: *float*
  End point of the sampling. 0 = beginning of the cmap; 1 = end of the cmap.

3.1.35 misc.split_acqus_2D(acqus)

Split the acqus dictionary of a 2D spectrum into two separate 1D-like acqus dictionaries.

Parameters:

- acqus: *dict*
  acqus dictionary of a 2D spectrum

Returns:

- acqu1s: *dict*
  acqus dictionary of the indirect dimension
- acqu2s: *dict*
  acqus dictionary of the direct dimension

3.1.36 misc.split_procs_2D(procs)

Split the procs dictionary of a 2D spectrum into two separate 1D-like procs dictionaries.

Parameters

- procs: *dict*
  procs dictionary of a 2D spectrum

Returns

- proc1s: *dict*
  procs dictionary of the indirect dimension
- proc2s: *dict*
  procs dictionary of the direct dimension

3.1.37 misc.trim_data(ppm_scale, y, sx, dx)

Trims the frequency scale and correspondant 1D dataset `y` from `sx` (ppm) to `dx` (ppm).

Parameters

- ppm_scale : *1darray*
  ppm scale of the spectrum
- y : *1darray*
  spectrum
- sx : *float*
  ppm value where to start trimming
- dx : *float*
  ppm value where to finish trimming

Returns

- xtrim : *1darray*
  Trimmed ppm scale
- ytrim : *1darray*
  Trimmed spectrum

3.1.38 misc.trim_data_2D(x_scale, y_scale, data, xlim=None, ylim=None)

Trims data and scales according to `xlim` and `ylim`. Returns the trimmed data and the correspondant trimmed scales.

Parameters:

- x_scale: *1darray*
  Scale for the rows of `data`
- y_scale: *1darray*
  Scale for the columns of `data`
- data: *2darray*
  Data to be trimmed
- xlim: *tuple*
  Limits for `x_scale (L, R)`
- ylim: *tuple*
  Limits for `y_scale (L, R)`

Returns:

- trimmed_x: *1darray*
  Trimmed x-scale
- trimmed_y: *1darray*
  Trimmed y-scale
- trimmed_data: *2darray*
  Trimmed data

3.1.39 misc.unhankel(H)

Concatenates the first row and the last column of the matrix `H`, which should have Hankel-like structure, so to build the array of independent parameters.

Parameters:

- H: *2darray*
  Hankel-like matrix

Returns:

- h: *1darray*
  First row and last column of `H`, concatenated

3.1.40 misc.write_acqus_1D(acqus, path='./', filename=None)

Writes the input file for a simulated spectrum, basing on a dictionary of parameters.

Parameters

- acqus : *dict*
  The dictionary containing the parameters for the simulation
- path : *str, optional*
  Directory where the file will be saved.
- filename: *str, optional*
  Name of the file to be saved in `path`. The default name is `sim_in_1D`

3.1.41 misc.write_acqus_2Dacqus, path='sim_in_2D')

Writes the input file for a simulated spectrum, basing on a dictionary of parameters.

Parameters

- acqus : *dict*
  The dictionary containing the parameters for the simulation
- path : *str, optional*
  Directory where the file will be saved. The default name is `sim_in_1D`.

3.1.42 misc.write_help(request, file=None)

Gets the documentation of `request`, and tries to save it in a text file.

Parameters:

- request: *function or class or package*
  Whatever you need documentation of
- file: *str or None or False*
  Name of the output documentation file. If it is `None`, a default name is given. If it is `False`, the output is printed on screen.

3.1.43 misc.write_ser(fid, path='./', BYTORDA=0, DTYPA=0, overwrite=True)

Writes the FID file in directory 'path', in a TopSpin-readable way (i.e. little endian, int32). The binary file is named 'fid' if 1D, 'ser' if multiD. The parameters BYTORDA and DTYPA can be found in the acqus file.

- `BYTORDA` = 1 ⇒ big endian ⇒ '>'
- `BYTORDA` = 0 ⇒ little endian ⇒ '<'
- `DTYPA` = 0 ⇒ int32 ⇒ 'i4'
- `DTYPA` = 2 ⇒ float64 ⇒ 'f8'

Parameters:

- fid : *ndarray*
  FID array to be written
- path : *str*
  Directory where to save the file

3.2 PROCESSING package

This package contains functions for the processing of NMR spectra, either in time domain or in frequency domain, and the transition between the two domains.

3.2.1 processing.Cadzow(data, n, nc, print_head=True)

This functions performs Cadzow denoising on `data`, which is a 1D array of N points. The algorithm works as follows:

1. Transform `data` in a Hankel matrix $\mathbb{H}$ of dimensions $(N-n, n)$
2. Make SVD on $\mathbb{H} = \mathbb{U}\mathbb{S}\mathbb{V}^\dagger$
3. Keep only the first `nc` singular values, and put all the rest to 0 ($\mathbb{S} \to \mathbb{S}'$)
4. Rebuild $\mathbb{H}' = \mathbb{U}\mathbb{S}'\mathbb{V}^\dagger$
5. Average the antidiagonals to rebuild the Hankel-type structure, then make 1D array

Parameters

- data: *1darray*
  Input data
- n: *int*
  Number of columns of the Hankel matrix.
- nc: *int*
  Number of singular values to keep.
- print_head: *bool*
  Set it to `True` to display the fancy heading.

Returns

- datap: *1darray*
  Denoised data

3.2.2 processing.Cadzow_2D(data, n, nc, i=True, itermax=100, f=0.005, print_time=True)

Performs the Cadzow denoising method on a 2D spectrum, one transient at the time. This function calls `Cadzow` if `i=False`, or `iterCadzow` if `i=True`.

Parameters

- data: *2darray*
  Input data
- n: *int*
  Number of columns of the Hankel matrix.
- nc: *int*
  Number of singular values to keep.
- i: *bool*
  Calls `processing.Cadzow` if `i=False`, or `processing.iterCadzow` if `i=True`.
- itermax: *int*
  Maximum number of iterations allowed.
- f: *float*
  Factor for the arrest criterion.
- print_time: *bool*
  Set it to `True` to display the time spent.

Returns

- datap: *2darray*
  Denoised data

3.2.3 processing.EAE(data)

Shuffles data if the spectrum is acquired with *'FnMODE':'Echo-Antiecho'*. **NOTE:** introduces -90° phase shift in F1, to be corrected after the processing.

```
pdata = np.zeros_like(data)
pdata[::2] = (data[::2].real - data[1::2].real) + 1j*(data[::2].imag - data[1::2].imag)
pdata[1::2] = -(data[::2].imag + data[1::2].imag) + 1j*(data[::2].real + data[1::2].real)
```

Parameters

- data : *ndarray*
  Data to be shuffled.

Returns

- pdata : *ndarray*
  Shuffled data.

3.2.4 processing.LRD(data, nc)

Denoising method based on Low-Rank Decomposition. The algorithm performs a singular value decomposition on `data`, then keeps only the first `nc` singular values while setting all the others to 0. Finally, rebuilds the data matrix using the modified singular values.

Parameters:

- data: *2darray*
  Data to be denoised
- nc: *int*
  Number of components, i.e. number of singular values to keep

Returns:

- data_out: *2darray*
  Denoised data

3.2.5 processing.MCR(input_data, nc, f=10, tol=1e-5, itermax=1e4, H=True, oncols=True)

This is an implementation of Multivariate Curve Resolution for the denoising of 2D NMR data.

Let us consider a matrix $\mathbb{D}$, of dimensions $m \times n$, where the starting data are stored. The final purpose of MCR is to decompose the $\mathbb{D}$ matrix as follows:

$$\mathbb{D} = \mathbb{CS} + \mathbb{E}$$

where $\mathbb{C}$ and $\mathbb{S}$ are matrices of dimension $m \times$ `nc` and `nc` $\times n$, respectively, and $\mathbb{E}$ contains the part of the data that are not reproduced by the factorization.

Being $\mathbb{D}$ the FID of a NMR spectrum, $\mathbb{C}$ will contain time evolutions of the indirect dimension, and $\mathbb{S}$ will contain transients in the direct dimension.

The total MCR workflow can be separated in two parts: a first algorithm that produces an initial guess for the three matrices $\mathbb{C}$, $\mathbb{S}$ and $\mathbb{E}$ (`SIMPLISMA`), and an optimization step that aims at the removal of the unwanted features of the data by iteratively filling the $\mathbb{E}$ matrix (`MCR ALS`).

This function returns the denoised datasets, $\mathbb{CS}$, and the single $\mathbb{C}$ and $\mathbb{S}$ matrices.

Parameters

- input_data: *2darray or 3darray*
  a 3D array containing the set of 2D NMR datasets to be coprocessed stacked along the first dimension. A single 2D array can be passed, if the denoising of a single dataset is desired.
- nc: *int*
  number of purest components to be looked for;
- f: *float*
  percentage of allowed noise;
- tol: *float*
  tolerance for the arrest criterion;
- itermax: *int*
  maximum number of allowed iterations
- H: *bool*
  `True` for horizontal stacking of data (default), `False` for vertical;
- oncols: *bool*
  `True` to estimate $\mathbb{S}$ with `processing.SIMPLISMA`, `False` to estimate $\mathbb{C}$.

Returns

- CS_f: *2darray or 3darray*
  Final denoised data matrix
- C_f: *2darray or 3darray*
  Final $\mathbb{C}$ matrix
- S_f: *2darray or 3darray*
  Final $\mathbb{S}$ matrix

3.2.6 processing.MCR_ALS(D, C, S, itermax=10000, tol=1e-5)

Performs alternating least squares to get the final $\mathbb{C}$ and $\mathbb{S}$ matrices. Being the fundamental MCR equation:

$$\mathbb{D} = \mathbb{C}\mathbb{S} + \mathbb{E}$$

At the k-th step of the iterative cycle:

1. $\mathbb{C}^{(k)} = \mathbb{D}\mathbb{S}^{+(k-1)}$
2. $\mathbb{S}^{(k)} = \mathbb{C}^{+(k)}\mathbb{D}$
3. $\mathbb{E}^{(k)} = \mathbb{D} - \mathbb{C}^{(k)}\mathbb{S}^{(k)}$

Defined r_C and r_S as the Frobenius norm of the difference of $\mathbb{C}$ and $\mathbb{S}$ matrices between two subsequent steps:

$$r_C = \left\|\mathbb{C}^{(k)} - \mathbb{C}^{(k-1)}\right\|_F \qquad r_S = \left\|\mathbb{S}^{(k)} - \mathbb{S}^{(k-1)}\right\|_F$$

The convergence is reached when:

$$r_C \leq \texttt{tol} \quad \wedge \quad r_S \leq \texttt{tol}$$

Parameters

- D: *2darray*
  Input data, of dimensions $m \times n$
- C: *2darray*
  Estimation of the $\mathbb{C}$ matrix, of dimensions $m \times$ `nc`.
- S: *2darray*
  Estimation of the $\mathbb{S}$ matrix, of dimensions `nc` $\times n$.
- itermax: *int*
  Maximum number of iterations
- tol: *float*
  Threshold for the arrest criterion.

Returns

- C: *2darray*
  Optimized $\mathbb{C}$ matrix, of dimensions $m \times$ `nc`.
- S: *2darray*
  Optimized $\mathbb{S}$ matrix, of dimensions `nc` $\times n$.

3.2.7 processing.MCR_unpack(C, S, nds, H=True)

Reverts matrix augmentation of `stack_MCR`. If `H=True`, converts `C` from dimensions $(Y, \texttt{nds})$ to $(X, Y, \texttt{nds})$ and `S` from dimensions $(\texttt{nds}, X * Z)$ to $(X, \texttt{nds}, Z)$; if `H=False` converts `C` from dimensions $(Y, \texttt{nds})$ to $(X, Y, \texttt{nds})$ and `S` from dimensions $(\texttt{nds}, X * Z)$ to $(X, \texttt{nds}, Z)$.

Parameters

- C: *2darray*
  MCR $\mathbb{C}$ matrix
- S: *2darray*
  MCR $\mathbb{S}$ matrix
- nds: *int*
  Number of datasets to be unpacked.
- H: *bool*
  `True` for horizontal stacking, `False` for vertical stacking.

Returns

- C_f: *3darray*
  Not-augmented $\mathbb{C}$ matrix.
- S_f: *3darray*
  Not-augmented $\mathbb{S}$ matrix.

3.2.8 processing.SIMPLISMA(D, nc, f=10, oncols=True)

Finds the first `nc` purest components of matrix `D` using the SIMPLISMA algorithm, proposed by Windig and Guilment (*DOI: 10.1021/ac00014a016*). If `oncols=True`, this function estimates $\mathbb{S}$ with SIMPLISMA, then calculates $\mathbb{C} = \mathbb{D}\mathbb{S}^{+}$. If `oncols=False`, this function estimates $\mathbb{C}$ with SIMPLISMA, then calculates $\mathbb{S} = \mathbb{C}^{+}\mathbb{D}$. `f` defines the percentage of allowed noise.

Parameters

- D: *2darray*
  Input data, of dimensions $m \times n$
- nc: *int*
  Number of components to be found. This determines the final size of the $\mathbb{C}$ and $\mathbb{S}$ matrices.
- f: *float*
  Percentage of allowed noise.
- oncols: *bool*
  If `True`, `SIMPLISMA` estimates the $\mathbb{S}$ matrix, otherwise estimates $\mathbb{C}$.

Returns

- C: *2darray*
  Estimation of the $\mathbb{C}$ matrix, of dimensions $m \times$ `nc`.
- S: *2darray*
  Estimation of the $\mathbb{S}$ matrix, of dimensions `nc` $\times n$.

3.2.9 processing.acme(data, m=1, a=5e-05)

Automated phase Correction based on Minimization of Entropy. This algorithm allows for automatic phase correction by minimizing the entropy of the m-th derivative of the spectrum, as explained in detail by L. Chen et.al. in *Journal of Magnetic Resonance 158 (2002) 164-168*.

Defined the entropy of h as:

$$S = -\sum_j h_j \ln h_j$$

and

$$h = \frac{\left|R_j^{(m)}\right|}{\sum_j \left|R_j^{(m)}\right|}$$

where

$$R = \mathrm{Re}\{\texttt{spectrum}\, e^{-\mathrm{i}\phi}\}$$

and $R^{(m)}$ is the m-th derivative of R, the objective function to minimize is:

$$Q = S + P(R)$$

where $P(R)$ is a penalty function for negative values of the spectrum.

The phase correction is applied using `processing.ps`. The values `p0` and `p1` are fitted using Nelder-Mead algorithm.

Parameters:

- data: *1darray*
  Spectrum to be phased, complex
- m: *int*
  Order of the derivative to be computed
- a: *float*
  Weighting factor for the penalty function

Returns:

- p0f: *float*
  Fitted zero-order phase correction, in degrees
- p1f: *float*
  Fitted first-order phase correction, in degrees

3.2.10 processing.align(ppm_scale, data, lims, u_off=0.5, ref_idx=0)

Performs the calibration of a pseudo-2D experiment by circular-shifting the spectra of an appropriate amount. The target function aims to minimize the superimposition between a reference spectrum and the others using a brute-force method.

Parameters:

- ppm_scale: *1darray*
  ppm scale of the spectrum to calibrate
- data: *2darray*
  Complex-valued spectrum
- lims: *tuple*
  (ppm sx, ppm dx) of the calibration region
- u_off: *float*
  Maximum offset for the circular shift, in ppm
- ref_idx: *int*
  Index of the spectrum to be used as reference

Returns:

- data_roll: *2darray*
  Calibrated data
- u_cal: *list*
  Number of point of which the spectra have been circular-shifted
- u_cal_ppm: *list*
  Correction for the ppm scale of each experiment

3.2.11 processing.baseline_correction(ppm, data, basl_file='spectrum.basl', winlim=None)

Interactively corrects the baseline of a given spectrum and saves the parameters in a file. The program starts with an interface to partition the spectrum in windows to correct separately. Then, for each window, an interactive panel opens to allow the user to compute the baseline.

Parameters:

- ppm: *1darray*
  PPM scale of the spectrum
- data: *1darray*
  The spectrum of which to adjust the baseline
- basl_file: *str*
  Name for the baseline parameters file
- winlim: *list or str or None*
  List of the breakpoints for the window. If it is `str`, it points to a file to be read with `np.loadtxt`. If it is `None`, the partitioning is done interactively.

3.2.12 processing.calc_nc(data, s_n)

Calculates the optimal number of components, given the standard deviation of the noise. The threshold value is calculated as stated in Theorem 1 of reference: https://arxiv.org/abs/1710.09787v2

Parameters:

- data: *2darray*
  Input data
- s_n: *float*
  Noise standard deviation

Returns:

- n_c: *int*
  Number of components

3.2.13 processing.calibration(ppmscale, S)

Scroll the ppm scale of spectrum to make calibration. The interface offers two guidelines: the red one, labelled 'reference signal' remains fixed, whereas the green one ('calibration value') moves with the ppm scale.

The ideal calibration procedure consists in placing the red line on the signal you want to use as reference, and the green line on the ppm value that the reference signal must assume in the calibrated spectrum. Then, scroll with the mouse until the two lines are superimposed.

Parameters

- ppmscale: *1darray*
  The ppm scale to be calibrated
- S: *1darray*
  The spectrum to calibrate

Returns

- offset: *float*
  Difference between original scale and new scale. This must be summed up to the original ppm scale to calibrate the spectrum.

3.2.14 processing.convdta(data, grpdly=0, scaling=1)

Removes the digital filtering to obtain a spectrum similar to the command CONVDTA performed by TopSpin. However, they will differ a little bit because of the digitization. These differences are not invisible to human's eye.

Parameters:

- data: *ndarray*
  FID with digital filter
- grpdly: *int*
  Number of points that the digital filter consists of. Key *$GRPDLY* in *acqus* file
- scaling: *float*
  Scaling factor of the resulting FID. Needed to match TopSpin's intensities.

Returns:

- data_in: *ndarray*
  FID without the digital filter. It will have `grpdly` points less than `data`.

3.2.15 processing.em(data, lb, sw)

Exponential apodization.

Being the FID an array of N points in its last dimension, and taken $x = \texttt{np.arange(N)}/N$:

$$\texttt{apod} = \exp\left[-\pi \frac{\texttt{lb}}{2\texttt{sw}} x\right]$$

Parameters

- data : *ndarray*
  FID to be apodized
- lb : *float*
  Lorentzian broadening, in Hz. $\texttt{lb} > 0$
- sw : *float*
  Spectral width in Hz, used for normalization.

Returns

- apod * data : *ndarray*
  Apodized FID.

3.2.16 processing.fp(data, wf=None, zf=None, fcor=0.5, tdeff=0)

Performs the full processing of a 1D NMR FID.

Parameters

- data: *1darray*
  Data to be processed.
- wf: *dict*
  {'mode': function to be used, 'parameters': see window functions documentation}
- zf: *int*
  final size of spectrum
- fcor: *float*
  Weighting factor for the first point of the FID.
- tdeff: *int*
  Number of FID points to be employed for the processing. `tdeff = 0` means whole FID.

Returns

- data: *ndarray*
  Processed spectrum.

3.2.17 processing.ft(data, alt=False, fcor=0.5, Numpy=True)

Fourier transform in NMR sense. This means it returns the reversed spectrum.

Parameters

- data : *ndarray*
  FID to be Fourier-transformed.
- alt: *bool*
  Negates the sign of the odd points, then takes the complex conjugate. Required for States-TPPI processing.
- fcor: *float*
  Weighting factor for FID 1-st point. Default value (0.5) prevents baseline offset
- Numpy: *bool*
  If `True` (***STRONGLY ADVISED***) performs the FT using the FFT algorithm encoded in Numpy. Otherwise, performs it manually using the definition of discrete FT.

Returns

- dataft : *ndarray*
  Transformed FID.

3.2.18 processing.gm(data, lb, gb, sw, gc=0)

Gaussian apodization.

Being the FID an array of N points in its last dimension, and taken $x = \texttt{np.arange(N)}$:

$$\texttt{apod} = \exp\big[a - b^2\big], \qquad a = \pi\frac{\texttt{lb}}{\texttt{sw}}x, \qquad b = 0.6\pi\frac{\texttt{gb}}{\texttt{sw}}\big(\texttt{gc}(N-1) - x\big)$$

Parameters

- data : *ndarray*
  FID to be apodized
- lb : *float*
  Lorentzian broadening, in Hz. $\texttt{lb} < 0$
- gb : *float*
  Gaussian broadening, in Hz. $\texttt{gb} > 0$
- sw : *float*
  Spectral width in Hz, used for normalization.
- gc: *float*
  Gaussian center, relatively to the FID length: $0 \leq \texttt{gc} \leq 1$

Returns

- apod * data : *ndarray*
  Apodized FID.

3.2.19 processing.gmb(data, lb, gb, sw)

Gaussian apodization, Bruker-like. It does not work very well.

Being the FID an array of N points in its last dimension, and taken $t = \texttt{np.arange(N)/sw}$:

$$\texttt{apod} = \exp\bigl[\alpha t - (\beta t)^2\bigr], \qquad \alpha = \pi\texttt{lb}, \qquad \beta = \frac{-\alpha N}{2\,\texttt{gb}\,\texttt{sw}})$$

Parameters

- data : *ndarray*
  FID to be apodized
- lb : *float*
  Lorentzian broadening, in Hz. $\texttt{lb} < 0$
- gb : *float*
  Gaussian broadening, in Hz. $\texttt{gb} > 0$
- sw : *float*
  Spectral width in Hz, used for normalization.
- gc: *float*
  Gaussian center, relatively to the FID length: $0 \leq \texttt{gc} \leq 1$

Returns

- apod * data : *ndarray*
  Apodized FID.

3.2.20 processing.ift(data, alt=False, fcor=0.5, Numpy=True)

Inverse Fourier transform in NMR sense. This means that the spectrum is reversed before to be inverse-Fourier transformed.

Parameters

- data : *ndarray*
  FID to be inverse Fourier-transformed.
- alt: *bool*
  Negates the sign of the odd points, then takes the complex conjugate. Required for States-TPPI processing.
- fcor: *float*
  Weighting factor for FID 1-st point. Default value (0.5) prevents baseline offset
- Numpy: *bool*
  If `True` (***STRONGLY ADVISED***) performs the FT using the FFT algorithm encoded in Numpy. Otherwise, performs it manually using the definition of discrete FT.

Returns

- dataft : *ndarray*
  Inverse transformed FID.

3.2.21 processing.integral(fx, x=None, lims=None)

Calculates the primitive of `fx`. If `fx` is a multidimensional array, the integrals are computed along the last dimension.

Parameters:

- fx: *ndarray*
  Function (array) to integrate
- x: *1darray or None*
  Independent variable. Determines the integration step. If `None`, it is set as the point scale
- lims: *tuple or None*
  Integration range. If `None`, the whole function is integrated

Returns:

- Fx: *ndarray*
  Integrated function.

3.2.22 processing.integral_2D(ppm_f1, t_f1, SFO1, ppm_f2, t_f2, SFO2, u_1=None, fwhm_1=200, utol_1=0.5, u_2=None, fwhm_2=200, utol_2=0.5, plot_result=False)

Calculate the integral of a 2D peak. The idea is to extract the traces correspondent to the peak center and fit them with a gaussian function in each dimension. Then, once got the intensity of each of the two gaussians, multiply them together in order to obtain the 2D integral. This procedure should be equivalent to what CARA does.

Parameters:

- ppm_f1: *1darray*
  PPM scale of the indirect dimension
- t_f1: *1darray*
  Trace of the indirect dimension, real part
- SFO1: *float*
  Larmor frequency of the nucleus in the indirect dimension
- ppm_f2: *1darray*
  PPM scale of the direct dimension
- t_f2: *1darray*
  Trace of the direct dimension, real part
- SFO2: *float*
  Larmor frequency of the nucleus in the direct dimension
- u_1: *float*
  Chemical shift in F1 /ppm. Defaults to the center of the scale
- fwhm_1: *float*
  Starting FWHM /Hz in the indirect dimension
- utol_1: *float*
  Allowed tolerance for `u_1` during the fit. `(u_1-utol_1, u_1+utol_1)`
- u_2: *float*
  Chemical shift in F2 /ppm. Defaults to the center of the scale
- fwhm_2: *float*
  Starting FWHM /Hz in the direct dimension
- utol_2: *float*
  Allowed tolerance for `u_2` during the fit. `(u_2-utol_2, u_2+utol_2)`
- plot_result: *bool*
  True to show how the program fitted the traces.

Returns:

- I_tot: *float*
  Computed integral.

3.2.23 processing.interactive_basl_windows(ppm, data)

Allows for interactive partitioning of a spectrum in windows. Double left click to add a bar, double right click to remove it. Returns the location of the red bars as a list.

Parameters:

- ppm: *1darray*
  PPM scale of the spectrum
- data: *1darray*
  Spectrum to be partitioned

Returns:

- coord: *list*
  List containing the coordinates of the windows, plus `ppm[0]` and `ppm[-1]`

3.2.24 processing.interactive_echo_param(data0)

Interactive plot that allows to select the parameters needed to process a CPMG-like FID. Use the TextBox or the arrow keys to adjust the values. You can call `processing.sum_echo_train` or `processing.split_echo_train` by starring the return statement of this function, i.e.:

```
processing.sum_echo_train(data0, *interactive_echo_train(data0))
```

as they are in the correct order to be used in this way.

Parameters:

- data0: *ndarray*
  CPMG FID

Returns:

- n: *int*
  Distance between one echo and the next one
- n_echoes: *int*
  Number of echoes to sum/split
- i_p: *int*
  Offset points from the start of the FID

3.2.25 processing.interactive_fp(fid0, acqus, procs)

Perform the processing of a 1D NMR spectrum interactively. The GUI offers the opportunity to test different window functions, as well as different `tdeff` values and final sizes. The active parameters appear as blue text.

Parameters:

- fid0: *1darray*
  FID to process
- acqus: *dict*
  Dictionary of acquisition parameters
- procs: *dict*
  Dictionary of processing parameters

Returns:

- pdata: *1darray*
  Processed spectrum
- procs: *dict*
  Updated dictionary of processing parameters

3.2.26 processing.interactive_phase_1D(ppmscale, S)

Allows for interactive phase adjustment of 1D NMR spectra. Employs `processing.ps` for the actual phase correction. Press the **z** key on the keyboard to toggle the automatic adjustment of vertical scale on or off.

Parameters

- ppmscale: *1darray*
  PPM scale of the spectrum
- S: *1darray*
  The spectrum to be phased

Returns

- phased_data: *1darray*
  Phased spectrum.

3.2.27 processing.interactive_phase_2D(ppm_f1, ppm_f2, S)

Interactively adjust the phases of a 2D spectrum. `S` must be hypercomplex, therefore must be passed before to unpack it into the 4 real files. The phase correction is done using `processing.ps` as follows:

1. `S = processing.ps(S, p0=p0_f2, p1=p1_f2, pivot=pivot_f2)`
2. Transpose: normal if *FnMODE='QF'*, hypercomplex otherwise
3. `S = phase(S, p0=p0_f1, p1=p1_f1, pivot=pivot_f1)`
4. Transpose back

Parameters

- ppm_f1: *1darray*
  Indirect dimension ppm scale
- ppm_f1: *1darray*
  Direct dimension ppm scale
- S: *2darray*
  Hypercomplex spectrum

Returns

- S:*2darray*
  Phased spectrum

3.2.28 processing.interactive_qfil(ppm, data_in)

Interactive function to design a gaussian filter with the aim of suppressing signals in the spectrum. You can adjust position and width of the filter scrolling with the mouse.

Parameters:

- ppm: *1darray*
  Scale on which the filter will be built
- data_in: *1darray*
  Spectrum on which to apply the filter.

Returns:

- u: *float*
  Position of the gaussian filter
- s: *float*
  Width of the gaussian filter (Standard deviation)

3.2.29 processing.interactive_xfb(fid0, acqus, procs, lvl0=0.1, show_cnt=True)

Perform the processing of a 2D NMR spectrum interactively. The GUI offers the opportunity to test different window functions, as well as different `tdeff` values and final sizes. The active parameters appear as blue text. When changing the parameters, give it some time to compute. The figure panel is quite heavy.

Parameters:

- fid0: *2darray*
  FID to process
- acqus: *dict*
  Dictionary of acquisition parameters
- procs: *dict*
  Dictionary of processing parameters
- lvl0: *float*
  Starting level of the contours
- show_cnt: *bool*
  Choose if to display data using contours (`True`) or heatmap (`False`)

Returns:

- pdata: *2darray*
  Processed spectrum
- procs: *dict*
  Updated dictionary of processing parameters

3.2.30 processing.inv_fp(data, wf=None, size=None, fcor=0.5)

Performs the full inverse processing of a 1D NMR spectrum (`data`). Required parameters are:

Parameters

- data: *1darray*
  Input data
- wf: *dict*
  {'mode': function to be used, 'parameters': different from each function}
- size: *int*
  initial size of the FID
- fcor: *float*
  weighting factor for the FID first point

Returns

- data: *1darray*
  Processed data

3.2.31 processing.inv_xfb(data, wf=[None, None], size=[None, None], fcor=[0.5, 0.5], FnMODE='States-TPPI')

Performs the full processing of a 2D NMR FID (`data`). Required parameters are:

Parameters

- data: *2darray*
  Input data
- wf: *list of dict*
  list of two entries `[F1, F2]`. Each entry is a dictionary of window functions
- zf: *list*
  list of two entries `[zf F1, zf F2]`
- fcor: *list*
  first fid point weighting factor `[F1, F2]`
- u : *bool*
  If `True`, unpacks the hypercomplex spectrum and returns the 4 real files, using `processing.unpack_2D`
- tdeff: *list of int*
  number of points of the FID to be used for the processing, `[F1, F2]`

Returns

- data: *2darray*
  Processed data

3.2.32 processing.iterCadzow(data, n, nc, itermax=100, f=0.005, print_head=T print_time=True)

This functions performs Cadzow denoising on `data`, which is a 1D array of N points, in an iterative manner. The algorithm works as follows:

1. Transform `data` in a Hankel matrix $\mathbb{H}$ of dimensions $(N-n, n)$
2. Make SVD on $\mathbb{H} = \mathbb{U}\mathbb{S}\mathbb{V}^\dagger$
3. Keep only the first `nc` singular values, and put all the rest to 0 ($\mathbb{S} \to \mathbb{S}'$)
4. Rebuild $\mathbb{H}' = \mathbb{U}\mathbb{S}'\mathbb{V}^\dagger$
5. Average the antidiagonals to rebuild the Hankel-type structure, then make 1D array
6. Check arrest criterion: if it is not reached, go to step 1, otherwise exit from the cycle and return the processed data.

The arrest criterion is on the array of singular values S, which is the main diagonal of the matrix $\mathbb{S}$. At step k and Python indexing system:

$$\left| \frac{S^{(k-1)}[\texttt{nc}-1]}{S^{(k-1)}[0]} - \frac{S^{(k)}[\texttt{nc}-1]}{S^{(k)}[0]} \right| < f \frac{S^{(0)}[\texttt{nc}-1]}{S^{(0)}[0]}$$

Parameters

- data: *1darray*
  Input data
- n: *int*
  Number of columns of the Hankel matrix.
- nc: *int*
  Number of singular values to keep.
- itermax: *int*
  Maximum number of iterations allowed.
- f: *float*
  Factor for the arrest criterion.
- print_head: *bool*
  Set it to `True` to display the fancy heading.
- print_time: *bool*
  Set it to `True` to display the time spent.

Returns

- datap: *1darray*
  Denoised data

3.2.33 processing.load_baseline(filename, ppm, data)

Read the baseline parameters from a file and builds the baseline itself.

Parameters:

- filename: *str*
  Location of the baseline file
- ppm: *1darray*
  PPM scale of the spectrum
- data: *1darray*
  Spectrum of which to correct the baseline

Returns:

- baseline: *1darray*
  Computed baseline

3.2.34 processing.make_polynomion_baseline(ppm, data, limits)

Interactive baseline correction with 4th degree polynomion.

Parameters:

- ppm: *1darray*
  PPM scale of the spectrum
- data: *1darray*
  spectrum
- limits: *tuple*
  Window limits `(left, right)`.

Returns:

- mode: *str*
  Baseline correction mode: `'polynomion'` as default, `'spline'` if you press the button
- C_f: *1darray or str*
  Baseline polynomion coefficients, or `'callintsmooth'` if you press the spline button

---

3.2.35 processing.make_scale(size, dw, rev=True)

Computes the frequency scale of the NMR spectrum, given the number of points and the employed dwell time (the REAL one, not the TopSpin one!). `rev=True` is required for the correct frequency arrangement in the NMR sense.

Parameters

- size: *int*
  Number of points of the frequency scale
- dw : *float*
  Time spacing in the time dimension
- rev: *bool*
  Reverses the scale

Returns

- fqscale: *1darray*
  The computed frequency scale.

3.2.36 processing.new_MCR(input_data, nc, f=10, tol=1e-05, itermax=10000. H=True, oncols=True, our_function=None, fargs=[], our_function2=N f2args=[])

This is an implementation of Multivariate Curve Resolution # for the denoising of 2D NMR data. It requires: # - input_data: a tensor containing the set of 2D NMR datasets to be coprocessed # stacked along the first dimension; # - nc : number of purest components; # - f : percentage of allowed noise; # - tol : tolerance for the arrest criterion; # - itermax : maximum number of allowed iterations, default 10000 # - H : True for horizontal stacking of data (default), False for vertical; # - oncols : True to estimate S with purest components, False to estimate C # This function returns the denoised datasets, 'CS', and the 'C' and 'S' matrices.

3.2.37 processing.new_MCR_ALS(D, C, S, itermax=10000, tol=1e-05, reg_f=None, reg_fargs=[])

Modified function to do ALS

3.2.38 processing.pknl(data, grpdly=0, onfid=False)

Compensate for the Bruker group delay at the beginning of FID through a first-order phase correction of

$$\phi^{(1)} = 360 * \texttt{GRPDLY}$$

This should be applied after apodization and zero-filling.

Parameters:

- data: *ndarray*
  Input data. Be sure it is complex!
- grpdly: *int*
  Number of points that make the group delay.
- onfid: *bool*
  If it is `True`, performs FT before to apply the phase correction, and IFT after.

Returns:

- datap: *ndarray*
  Corrected data

3.2.39 processing.ps(data, ppmscale=None, p0=None, p1=None, pivot=None, interactive=False)

Applies phase correction on the last dimension of data. The pivot is set at the center of the spectrum by default. Missing parameters will be inserted interactively.

Being `data` a 1D array of N points, as well as `ppmscale`, the following parameters are defined:

$p_V =$ (index of `pivot` on `ppmscale`)$/N$

$x =$ `np.arange`$(N)/N - p_V$

Basically, p_V and x are `pivot` and `ppmscale`, normalized in order to fit a scale that ranges from 0 to 1. This is done with the aim to 'standardize' the behaviour of the function, making it nucleus-independent.

The following phase-correction function is applied:

$$\Phi = \exp\big[\mathrm{i}(\mathtt{p0} + \mathtt{p1}x)\big]$$

If `data` is a 2D array, the function Φ is applied row-wise.

Parameters:

- data: *ndarray*
  Input data
- ppmscale: *1darray or None*
  PPM scale of the spectrum. Required for pivot and interactive phase correction
- p0: *float*
  Zero-order phase correction angle /degrees
- p1: *float*
  First-order phase correction angle /degrees
- pivot: *float or None.*
  First-order phase correction pivot /ppm. If `None`, it is set at the center of the spectrum.
- interactive: *bool*
  If `True`, all the parameters will be ignored and the interactive phase correction panel will be opened.

Returns:

- datap: *ndarray*
  Phased data
- final_values: *tuple*
  Employed values of the phase correction. `(p0, p1, pivot)`

3.2.40 processing.qfil(ppm, data, u, s)

Suppress signals in the spectrum using a gaussian filter.

Parameters:

- ppm: *1darray*
  Scale on which to build the filter
- data: *ndarray*
  Data to be processed. The filter is applied on the last dimension
- u: *float*
  Position of the filter
- s: *float*
  Width of the filter (standard deviation)

Returns:

- pdata: *ndarray*
  Filtered data

3.2.41 processing.qpol(fid)

Fits the FID with a 4-th degree polynomion, then subtracts it from the original FID. The real and imaginary channels are treated separately.

Parameters

- fid : *ndarray*
  Self-explanatory.

Returns

- fid : *ndarray*
  Processed FID.

3.2.42 processing.qsin(data, ssb)

Sine-squared apodization.

Being `data` an array of N points in its last dimension, and taken $x = \texttt{np.arange(N)}/N$, if $\texttt{ssb} = 0$ or $\texttt{ssb} = 1$:

$$\texttt{apod} = \sin^2\left(\pi\, x\right)$$

else if $\texttt{ssb} \geq 2$:

$$\texttt{apod} = \sin^2\left(\pi\frac{1}{\texttt{ssb}} + \pi(1 - \frac{1}{\texttt{ssb}})x\right)$$

Parameters

- data : *ndarray*
  FID to be apodized.
- ssb : *int*
  Shifting parameter for the sine bell.
  - $\texttt{ssb} = 0$: from 0 to π
  - $\texttt{ssb} = 1$: same as $\texttt{ssb} = 0$
  - $\texttt{ssb} = 2$: from $\pi/2$ to π
  - $\texttt{ssb} = 3$: from $\pi/3$ to π

Returns

- apod * data : *ndarray*
  Apodized FID.

3.2.43 processing.quad(fid)

Subtracts from the FID the arithmetic mean of its last quarter. The real and imaginary channels are treated separately.

Parameters

- fid : *ndarray*
  Self-explanatory.

Returns

- fid : *ndarray*
  Processed FID.

3.2.44 processing.repack_2D(rr, ir, ri, ii)

Renconstruct hypercomplex 2D NMR data given the 4 real arrays. See `processing.unpack_2D` for details.

Returns

- data

3.2.45 processing.rev(data)

Reverses the last dimension of `data`.

Parameters

- data : *ndarray*
  matrix to be reverted

Returns

- data[...,::-1] : *ndarray*
  Self-explanatory

3.2.46 processing.rpbc(data, split_imag=False, n=5, basl_method='tq', basl_thresh=0.1, basl_itermax=500, **phase_kws)

Reversed Phase and Baseline Correction. Allows for the automatic phase correction and baseline subtraction of NMR spectra. It is called 'reversed' because the baseline is actually computed and subtracted before to perform the phase correction.

The baseline is computed using a low-order polynomion, built on a scale that goes from -1 to 1, whose coefficients are obtained minimizing a non-quadratic cost function. It is recommended to use either 'tq' (truncated quadratic, much faster) or 'huber' (Huber function, slower but sometimes more accurate). The user is requested to choose between separating the real and imaginary channel in this step. The order of the polynomion and the threshold value are the key parameters for obtaining a good baseline. The used function is processing.polyn_basl

The phase correction is computed on the baseline-subtracted complex data as described in the SINC algorithm (M. Sawall et al., *Journal of Magnetic Resonance* 289 (2018), 132-141). The default parameters are generally fine, but in case of data with poor SNR (approximately SNR < 10) better results can be obtained by increasing the value of the e1 parameter. The employed function is processing.SINC_phase

Parameters:

- data: *1darray*
  Data to be processed, complex-valued
- split_imag: *bool*
  If True, computes the baseline on the real and imaginary part separately; else, the set of polynomion coefficients are forced to be the same for both
- n: *int*
  Number of coefficients of the polynomion, i.e. it will be of degree n-1
- basl_method: *str*
  Cost function to be minimized for the baseline computation. Look for fit.CostFunc, 'method' attribute
- basl_thresh: *float*
  Relative threshold value for the non-quadratic behaviour of the cost function. Look for fit.CostFunc, 's' attribute
- basl_itermax: *int*
  Maximun number of iterations allowed during the baseline fitting procedure
- phase_kws: *keyworded arguments*
  Optional arguments for the phase correction. Look for fit.SINC_phase keyworded arguments for details.

Returns:

- y: *1darray*
  Processed data
- p0: *float*
  Zero-order phase correction angle, in degrees
- p1: *float*
  First-order phase correction angle, in degrees

- c: *1darray*
  Set of coefficients to be used for the baseline computation, starting from the 0-order coefficient

---

3.2.47 processing.sin(data, ssb)

Sine apodization.

Being `data` an array of N points in its last dimension, and taken $x = \texttt{np.arange(N)}/N$, if $\texttt{ssb} = 0$ or $\texttt{ssb} = 1$:

$$\texttt{apod} = \sin\left(\pi\, x\right)$$

else if $\texttt{ssb} \geq 2$:

$$\texttt{apod} = \sin\left(\pi\frac{1}{\texttt{ssb}} + \pi(1 - \frac{1}{\texttt{ssb}})x\right)$$

Parameters

- data : *ndarray*
  FID to be apodized.
- ssb : *int*
  Shifting parameter for the sine bell.
  - $\texttt{ssb} = 0$: from 0 to π
  - $\texttt{ssb} = 1$: same as $\texttt{ssb} = 0$
  - $\texttt{ssb} = 2$: from $\pi/2$ to π
  - $\texttt{ssb} = 3$: from $\pi/3$ to π

Returns

- apod * data : *ndarray*
  Apodized FID.

3.2.48 processing.split_echo_train(datao, n, n_echoes, i_p=0)

Separate a CPMG echo-train FID into echoes so to be processed separately. The first decay, i.e. the native FID, is extracted, and corresponds to echo number 0. Then, for each echo, the left side (reversed) is summed up to its right part. You can use `processing.interactive_echo_param` to calculate the parameters interactively.

Parameters:

- datao: *ndarray*
  FID with an echo train on its last dimension
- n: *int*
  number of points that separate one echo from the next
- n_echoes: *int*
  number of echoes to extract. If it is 0, extracts only the first decay
- i_p: *int*
  Number of offset points

Returns:

- data_p: *(n+1)darray*
  Separated echoes

3.2.49 processing.stack_MCR(input_data, H=True)

Performs matrix augmentation converting `input_data` from dimensions (X, Y, Z) to $(Y, X * Z)$ if `H=True`, or $(X * Y, Z)$ if `H=False`.

Parameters

- input_data: *3darray*
  Contains the spectra to be stacked together. The index that runs on the datasets must be the first one.
- H: *bool*
  `True` for horizontal stacking, `False` for vertical stacking.

Returns

- data: *2darray*
  Augmented data matrix.

3.2.50 processing.sum_echo_train(datao, n, n_echoes, i_p=0)

Sum up a CPMG echo-train FID into echoes so to enchance the SNR. This function calls `processing.split_e` with the same parameters. You can use `processing.interactive_echo_param` to calculate the parameters interactively.

Parameters:

- datao: *ndarray*
  FID with an echo train on its last dimension
- n: *int*
  number of points that separate one echo from the next
- n_echoes: *int*
  number of echoes to sum
- i_p: *int*
  Number of offset points

Returns:

- data_p: *ndarray*
  Summed echoes

3.2.51 processing.tabula_rasa(data, lvl=0.05, cmap=cm.Blues_r)

This function is to be used in SIFT algorithm. Allows interactive selection using a Lasso widget of the region of the spectrum which contain signal. Returns a masking matrix, of the same shape as data, whose entries are 1 inside the selection and 0 outside.

Parameters

- data : *2darray*
  The data to be plotted
- lvl : *float*
  Level of the contours, expressed as fraction of the maximum intensity
- cmap : *matplotlib.cm Object*
  Color of the contours

Returns

- mask: *2darray*
  Matrix that contains 1 inside the selection and 0 outside.

3.2.52 processing.td_eff(data, tdeff)

Uses only the first `tdeff` points of data. It is applied before any other processing. The length of the list `tdeff` must match the dimension of `data`, if `data` is multidimensional.

Parameters

- data: *ndarray*
  The FID to be trimmed.

- tdeff: *int or list*
  The number of points to be used. If `data` is 1D, `tdeff` can be passed as integer, otherwise it has to be: `[F1, F2, ..., Fn]`. A 0 entry in the list means to not trim the corresponding dimension.

Returns

- data: *ndarray*
  Trimmed data according to `tdeff`.

3.2.53 processing.tp_hyper(data)

Computes the hypercomplex transpose of data. Needed for the processing of data acquired in a phase-sensitive manner in the indirect dimension.

To explain how a hypercomplex transposition works, we will focus on the simpler example of a 4×6 matrix, $\mathbb{A}$, with complex coefficients $a_{i,j} = x_{i,j} + \mathrm{i}y_{i,j}$.

$$\mathbb{A} = \begin{pmatrix} x_{1,1} + \mathrm{i}y_{1,1} & x_{1,2} + \mathrm{i}y_{1,2} & x_{1,3} + \mathrm{i}y_{1,3} & x_{1,4} + \mathrm{i}y_{1,4} \\ x_{2,1} + \mathrm{i}y_{2,1} & x_{2,2} + \mathrm{i}y_{2,2} & x_{2,3} + \mathrm{i}y_{2,3} & x_{2,4} + \mathrm{i}y_{2,4} \\ x_{3,1} + \mathrm{i}y_{3,1} & x_{3,2} + \mathrm{i}y_{3,2} & x_{3,3} + \mathrm{i}y_{3,3} & x_{3,4} + \mathrm{i}y_{3,4} \\ x_{4,1} + \mathrm{i}y_{4,1} & x_{4,2} + \mathrm{i}y_{4,2} & x_{4,3} + \mathrm{i}y_{4,3} & x_{4,4} + \mathrm{i}y_{4,4} \\ x_{5,1} + \mathrm{i}y_{5,1} & x_{5,2} + \mathrm{i}y_{5,2} & x_{5,3} + \mathrm{i}y_{5,3} & x_{5,4} + \mathrm{i}y_{5,4} \\ x_{6,1} + \mathrm{i}y_{6,1} & x_{6,2} + \mathrm{i}y_{6,2} & x_{6,3} + \mathrm{i}y_{6,3} & x_{6,4} + \mathrm{i}y_{6,4} \end{pmatrix}$$

The hypercomplex tranpose of the matrix $\mathbb{A}$, here referred as $\mathbb{B}$, is:

$$\mathbb{B} = \begin{pmatrix} x_{1,1} + \mathrm{i}x_{2,1} & x_{3,1} + \mathrm{i}x_{4,1} & x_{5,1} + \mathrm{i}x_{6,1} \\ y_{1,1} + \mathrm{i}y_{2,1} & y_{3,1} + \mathrm{i}y_{4,1} & y_{5,1} + \mathrm{i}y_{6,1} \\ x_{1,2} + \mathrm{i}x_{2,2} & x_{3,2} + \mathrm{i}x_{4,2} & x_{5,2} + \mathrm{i}x_{6,2} \\ y_{1,2} + \mathrm{i}y_{2,2} & y_{3,2} + \mathrm{i}y_{4,2} & y_{5,2} + \mathrm{i}y_{6,2} \\ x_{1,3} + \mathrm{i}x_{2,3} & x_{3,3} + \mathrm{i}x_{4,3} & x_{5,3} + \mathrm{i}x_{6,3} \\ y_{1,3} + \mathrm{i}y_{2,3} & y_{3,3} + \mathrm{i}y_{4,3} & y_{5,3} + \mathrm{i}y_{6,3} \\ x_{1,4} + \mathrm{i}x_{2,4} & x_{3,4} + \mathrm{i}x_{4,4} & x_{5,4} + \mathrm{i}x_{6,4} \\ y_{1,4} + \mathrm{i}y_{2,4} & y_{3,4} + \mathrm{i}y_{4,4} & y_{5,4} + \mathrm{i}y_{6,4} \end{pmatrix}$$

Parameters

- data : *2darray*
  Complex data matrix.

Returns

- datatp : *2darray*
  Hypercomplex transpose of data.

3.2.54 processing.unpack_2D(data)

Separates fully processed 2D NMR data into 4 distinct ser files:

`rr` = Re{`data`}[:: 2]

`ir` = Im{`data`}[:: 2]

`ri` = Re{`data`}[1 :: 2]

`ii` = Im{`data`}[1 :: 2]

Parameters

- data : *2darray*
  Complex data matrix.

Returns

- rr, ir, ri, ii: *2darray*, *2darray*, *2darray*, *2darray*
  See above.

3.2.55 processing.whittaker_smoother(data, n=2, s_f=1, w=None)

Adapted from P.H.C. Eilers, *Anal. Chem* 2003, 75, 3631-3636. Implementation of the smoothing algorithm proposed by Whittaker in 1923.

Parameters:

- data: *1darray*
  Data to be smoothed
- n: *int*
  Order of the difference to be computed
- s_f: *float*
  Smoothing factor
- w: *1darray or None*
  Array of weights. If `None`, no weighting is applied.

Returns:

- z: *1darray*
  Smoothed data

3.2.56 processing.write_basl_info(f, limits, mode, data)

Writes the baseline parameters of a certain window in a file.

Parameters:

- f: *TextIO object*
  File where to write the parameters
- limits: *tuple*
  Limits of the spectral window. `(left, right)`
- mode: *str*
  Baseline correction mode: `'polynomion'` or `'spline'`
- data: *float or 1darray*
  It can be either the spline smoothing factor or the polynomion coefficients

---

3.2.57 processing.xfb(data, wf=[None, None], zf=[None, None], fcor=[0.5,0.5], tdeff=[0,0], u=True, FnMODE='States-TPPI')

Performs the full processing of a 2D NMR FID.

Parameters

- data: *2darray*
  FID data matrix
- wf: *list*
  List of two entries `[F1, F2]`. Each entry is a dictionary: {'mode': function to be used, 'parameters': different from each function}
- zf: *list*
  List of two entries `[zf F1, zf F2]`
- fcor: *list*
  Weighting factor for the first point of the FID. `[F1, F2]`
- tdeff: *list*
  Number of FID points to be employed for the processing. `tdeff = [0,0]` means whole FID. `[F1, F2]`
- u : *bool*
  if `True`, unpacks the hypercomplex spectrum and returns the 4 files.

Returns

- data : *2darray*
  Processed data.

3.2.58 processing.zf(data, size)

Zero-filling of `data` up to `size`.

Parameters

- data : *ndarray*
  FID / Spectrum to be zero-filled
- size : *int*
  Final size of the FID / Spectrum

Returns

- datazf : *ndarray*
  Zero-filled FID / Spectrum

3.3 FIGURES package

This package contains a series of functions to make plots of various nature.

3.3.1 figures.ax1D(ax, ppm, datax, norm=False, xlims=None, ylims=None, c='tab:blue', lw=0.5, X_label='$ delta $ F1 /ppm', Y_label='Intensity /a.u.', n_xticks=10, n_yticks=10, label=None, fontsize=10)

Makes the figure of a 1D NMR spectrum, placing it in a given figure panel. This allows the making of modular figures.

The plot can be customized in a very flexible manner by setting the function keywords properly.

Parameters:

- ax: *matplotlib.subplot Object*
  panel where to put the figure
- ppm: *1darray*
  ppm scale of the spectrum
- data: *1darray*
  spectrum to be plotted
- norm: *bool*
  if True, normalizes the intensity to 1.
- xlims: *list or tuple*
  Limits for the x-axis. If None, the whole scale is used.
- ylims: *list or tuple*
  Limits for the y-axis. If None, the whole scale is used.
- c: *str*
  Colour of the line.
- lw: *float*
  linewidth
- X_label: *str*
  text of the x-axis label;
- Y_label: *str*
  text of the y-axis label;
- n_xticks: *int*
  Number of numbered ticks on the x-axis of the figure
- n_yticks: *int*
  Number of numbered ticks on the x-axis of the figure
- label: *str*
  label to be put in the legend.

- fontsize: *float*
  Biggest font size in the figure.

Returns:

- line: *Line2D Object*
  Line object returned by plt.plot.

3.3.2 figures.ax2D(ax, ppm_f2, ppm_f1, datax, xlims=None, ylims=None, cmap='Greys_r', c_fac=1.4, lvl=0.1, lw=0.5, X_label='$ delta ,$F2 /ppm', Y_label='$ delta ,$F1 /ppm', title=None, n_xticks=10, n_yticks=10, fontsize=10)

Makes a 2D contour plot like the one in `figures.figure2D`, but in a specified panel. Allows for the buildup of modular figures. The contours are drawn according to the formula:

```
cl = contour_start * contour_factor ** np.arange(contour_num)
```

where

```
contour_start = np.max(data) * lvl
contour_num = 16
contour_factor = c_fac.
```

Increasing the value of c_fac will decrease the number of contour lines, whereas decreasing the value of c_fac will increase the number of contour lines.

Parameters:

- ax: *matplotlib.subplot Object*
  panel where to put the figure
- ppm_f2: *1darray*
  ppm scale of the direct dimension
- ppm_f1: *1darray*
  ppm scale of the indirect dimension
- datax: *2darray*
  the 2D NMR spectrum to be plotted
- xlims: *tuple*
  limits for the x-axis (left, right). If None, the whole scale is used.
- ylims: *tuple*
  limits for the y-axis (left, right). If None, the whole scale is used.
- cmap: *str*
  Colormap identifier for the contour
- c_fac: *float*
  Contour factor parameter
- lvl: *float*
  height with respect to maximum at which the contour are computed
- X_label: *str*
  text of the x-axis label;
- Y_label: *str*
  text of the y-axis label;

- lw: *float*
  linewidth of the contours

- title: *str*
  Figure title.

- n_xticks: *int*
  Number of numbered ticks on the x-axis of the figure

- n_yticks: *int*
  Number of numbered ticks on the x-axis of the figure

- fontsize: *float*
  Biggest font size in the figure.

Returns:

- cnt: *matplotlib.QuadContour object*
  Drawn contour lines

3.3.3 figures.ax_heatmap(ax, data, zlim='auto', z_sym=True, cmap=None, xscale=None, yscale=None, rev=(False, False), n_xticks=10, n_yticks=1 n_zticks=10, fontsize=10)

Computes a heatmap of data on the given 'ax'

Parameters:

- ax: *matplotlib.Subplot object*
  Panel where to draw the heatmap
- data: *2darray*
  Input data
- zlim: *tuple or 'auto' or 'abs'*
  Vertical limits of the heatmap, that determines the extent of the colorbar. 'auto' means `(min(data), max(data))`, 'abs' means `(min(|data|), max(|data|))`.
- z_sym: *bool*
  True to symmetrize the vertical scale around 0.
- cmap: *matplotlib.cm object*
  Colormap of the heatmap.
- xscale: *1darray or None*
  x-scale. None means `np.arange(data.shape[1])`
- yscale: *1darray or None*
  y-scale. None means `np.arange(data.shape[0])`
- rev: *tuple of bool*
  Reverse scale (x, y).
- n_xticks: *int*
  Number of ticks of the x axis
- n_yticks: *int*
  Number of ticks of the y axis
- n_zticks: *int*
  Number of ticks of the color bar
- fontsize: *float*
  Biggest font size to apply to the figure.

Returns:

- im: *matplotlib.AxesImage*
  The heatmap
- cax: *figure panel where the colorbar is drawn*

3.3.4 figures.dotmd(ppmscale, S, labels=None, n_xticks=10)

Interactive display of multiple 1D spectra.

Parameters

- ppmscale: *1darray or list*
  ppm scale of the spectra. If only one scale is supplied, all the spectra are plotted using the same scale. Otherwise, each spectrum is plotted using its scale. There is a 1:1 correspondance between `ppmscale` and `S`.
- S: *list*
  spectra to be plotted
- labels: *list*
  labels to be put in the legend.
- n_xticks: *int*
  Number of numbered ticks on the x-axis of the figure

3.3.5 figures.dotmd_2D(ppm_f1, ppm_f2, S, labels=None, name='dotmd_2D X_label='$\delta\, $ F2 /ppm', Y_label='$\delta\, $ F1 /ppm', n_xticks=10, n_yticks=10, Neg=True)

Interactive display of multiple 2D spectra. They have to share the same scales.

- ppm_f1: *1darray*
  ppm scale of the indirect dimension. If only one scale is supplied, all the spectra are plotted using the same scale. Otherwise, each spectrum is plotted using its scale. There is a 1:1 correspondance between `ppm_f1` and `S`.

- ppm_f2: *1darray*
  ppm scale of the direct dimension. If only one scale is supplied, all the spectra are plotted using the same scale. Otherwise, each spectrum is plotted using its scale. There is a 1:1 correspondance between `ppm_f2` and `S`.

- S: *list*
  spectra to be plotted

- labels: *list*
  labels to be put in the legend.

- name: *str*
  If you choose to save the figure, this is its filename.

- X_label: *str*
  text of the x-axis label;

- Y_label: *str*
  text of the y-axis label;

- n_xticks: *int*
  Number of numbered ticks on the x-axis of the figure

- n_yticks: *int*
  Number of numbered ticks on the x-axis of the figure

- Neg: *bool*
  If `True`, show the negative contours.

3.3.6 figures.figure1D(ppm, datax, norm=False, xlims=None, ylims=None, c='tab:blue', lw=0.5, X_label='$ delta $ F1 /ppm', Y_label='Intensity /a.u.', n_xticks=10, n_yticks=10, fontsize=10, name=None, ext='png', dpi=600)

Makes the figure of a 1D NMR spectrum.

The plot can be customized in a very flexible manner by setting the function keywords properly.

Parameters:

- ppm: *1darray*
  ppm scale of the spectrum
- datax: *1darray*
  spectrum to be plotted
- norm: *bool*
  if True, normalizes the intensity to 1.
- xlims: *list or tuple*
  Limits for the x-axis. If None, the whole scale is used.
- ylims: *list or tuple*
  Limits for the y-axis. If None, the whole scale is used.
- c: *str*
  Colour of the line.
- lw: *float*
  linewidth
- X_label: *str*
  text of the x-axis label;
- Y_label: *str*
  text of the y-axis label;
- n_xticks: *int*
  Number of numbered ticks on the x-axis of the figure
- n_yticks: *int*
  Number of numbered ticks on the x-axis of the figure
- fontsize: *float*
  Biggest font size in the figure.
- name: *str or None*
  Filename for the figure to be saved. If None, the figure is shown instead.
- ext: *str*
  Format of the image
- dpi: *int*
  Resolution of the image in dots per inches

3.3.7 figures.figure1D_multi(ppm0, data0, xlims=None, ylims=None, norm=Fa c=None, X_label='$ delta $ F1 /ppm', Y_label='Intensity /a.u.', n_xticks=10, n_yticks=10, fontsize=10, labels=None, name=None, ext='png', dpi=600)

Creates the superimposed plot of a series of 1D NMR spectra.

Parameters:

- ppm0: *sequence of 1darray or 1darray*
  ppm scale of the spectra. If only one scale is supplied, it is assumed to be the same for all the spectra
- data0: *sequence of 1darray*
  List containing the spectra to be plotted
- xlims: *tuple or None*
  Limits for the x-axis. If None, the whole scale is used.
- ylims: *tuple or None*
  Limits for the y-axis. If None, they are automatically set.
- norm: *False or float or str*
  If it is False, it does nothing. If it is float, divides all spectra for that number. If it is str('#'), normalizes all the spectra to the '#' spectrum (python numbering). If it is whatever else string, normalizes all spectra to themselves.
- c: *tuple or None*
  List of the colors to use for the traces. None uses the default ones.
- X_label: *str*
  text of the x-axis label
- Y_label: *str*
  text of the y-axis label
- n_xticks: *int*
  Number of numbered ticks on the x-axis of the figure
- n_yticks: *int*
  Number of numbered ticks on the x-axis of the figure
- fontsize: *float*
  Biggest fontsize in the picture
- labels: *list or None or False*
  List of the labels to be shown in the legend. If it is None, the default entries are used (i.e., '1, 2, 3,...'). If it is False, the legend is not shown.
- name: *str or None*
  Filename of the figure, if it has to be saved. If it is None, the figure is shown instead.
- ext: *str*
  Format of the image

- dpi: *int*
  Resolution of the image in dots per inches

3.3.8 figures.figure2D(ppm_f2, ppm_f1, datax, xlims=None, ylims=None, cmap='Greys_r', c_fac=1.4, lvl=0.09, X_label='$ delta $ F2 /ppm', Y_label='$ delta $ F1 /ppm', lw=0.5, cmapneg=None, n_xticks=10, n_yticks=10, fontsize=10, name=None, ext='png', dpi=600)

Makes a 2D contour plot. Allows for the buildup of modular figures. The contours are drawn according to the formula:

```
cl = contour_start * contour_factor ** np.arange(contour_num)
```

where

```
contour_start = np.max(data) * lvl
contour_num = 16
contour_factor = c_fac.
```

Increasing the value of c_fac will decrease the number of contour lines, whereas decreasing the value of c_fac will increase the number of contour lines.

Parameters:

- ppm_f2: *1darray*
  ppm scale of the direct dimension
- ppm_f1: *1darray*
  ppm scale of the indirect dimension
- datax: *2darray*
  the 2D NMR spectrum to be plotted
- xlims: *tuple*
  limits for the x-axis (left, right). If None, the whole scale is used.
- ylims: *tuple*
  limits for the y-axis (left, right). If None, the whole scale is used.
- cmap: *str*
  Colormap identifier for the contour
- c_fac: *float*
  Contour factor parameter
- lvl: *float*
  height with respect to maximum at which the contour are computed
- X_label: *str*
  text of the x-axis label;
- Y_label: *str*
  text of the y-axis label;

- lw: *float*
  linewidth of the contours
- cmapneg: *str or None*
  Colormap identifier for the negative contour. If None, they are not computed at all
- n_xticks: *int*
  Number of numbered ticks on the x-axis of the figure
- n_yticks: *int*
  Number of numbered ticks on the x-axis of the figure
- fontsize: *float*
  Biggest font size in the figure.
- name: *str*
  Filename for the figure
- ext: *str*
  Format of the image
- dpi: *int*
  Resolution of the image in dots per inches

3.3.9 figures.figure2D_multi(ppm_f2, ppm_f1, datax, xlims=None, ylims=Nor lvl='default', c_fac=1.4, Negatives=False, X_label='$ delta $ F2 /ppm', Y_label='$ delta $ F1 /ppm', lw=0.5, n_xticks=10, n_yticks=10, labels=None, name=Nor ext='png', dpi=600)

Generates the figure of multiple, superimposed spectra, using figures.ax2D.

Parameters:

- ppm_f2: *1darray*
  ppm scale of the direct dimension
- ppm_f1: *1darray*
  ppm scale of the indirect dimension
- datax: *sequence of 2darray*
  the 2D NMR spectra to be plotted
- xlims: *tuple*
  limits for the x-axis (left, right). If None, the whole scale is used.
- ylims: *tuple*
  limits for the y-axis (left, right). If None, the whole scale is used.
- lvl: *'default' or list*
  height with respect to maximum at which the contour are computed. If 'default', each spectrum is at 10% of maximum height. Otherwise, each entry of the list corresponds to the contour height of the respective spectrum.
- c_fac: *float*
  Contour factor
- Negatives: *bool*
  set it to True if you want to see the negative part of the spectrum
- X_label: *str*
  text of the x-axis label;
- Y_label: *str*
  text of the y-axis label;
- lw: *float*
  linewidth of the contours
- n_xticks: *int*
  Number of numbered ticks on the x-axis of the figure
- n_yticks: *int*
  Number of numbered ticks on the x-axis of the figure
- labels: *list*
  entries of the legend. If None, the spectra are numbered.

- name: *str*
  Filename for the figure. If None, it is shown instead of saved
- ext: *str*
  Format of the image
- dpi: *int*
  Resolution of the image in dots per inches

3.3.10 figures.fitfigure(S, ppm_scale, t_AQ, V, C=False, SFO1=701.125, o1p=0, limits=None, s_labels=None, X_label='$ delta ,$ F1 /ppm', n_xticks=10, name=None)

Makes the figure to show the result of a quantitative fit.

Parameters:

- S : *1darray*
  Spectrum to be fitted
- ppm_scale : *1darray*
  Self-explanatory
- V : *2darray*
  matrix (# signals, parameters)
- C : *1darray or False*
  Coefficients of the polynomion to be used as baseline correction. If the 'baseline' checkbox in the interactive figure panel is not checked, C_f is False.
- limits : *tuple or None*
  Trim limits for the spectrum (left, right). If None, the whole spectrum is used.
- s_labels : *list or None or False*
  Legend entries for the single components. If None, they are computed automatically as 1, 2, 3, etc. If False, they are not shown in the legend.
- X_label : *str*
  label for the x-axis.
- n_xticks : *int*
  number of numbered ticks that will appear in the ppm scale. An oculated choice can be very satisfying.
- name : *str or None*
  Name with which to save the figure. If None, the picture is shown instead of being saved.

3.3.11 figures.heatmap(data, zlim='auto', z_sym=True, cmap=None, xscale=None, yscale=None, rev=(False, False), n_xticks=10, n_yticks=10 n_zticks=10, fontsize=10, name=None)

Computes a heatmap of data.

Parameters:

- data: *2darray*
  Input data
- zlim: *tuple or 'auto' or 'abs'*
  Vertical limits of the heatmap, that determines the extent of the colorbar. 'auto' means (min(data), max(data)), 'abs' means(min(|data|), max(|data|)).
- z_sym: *bool*
  True to symmetrize the vertical scale around 0.
- cmap: *matplotlib.cm object*
  Colormap of the heatmap.
- xscale: *1darray or None*
  x-scale. None means np.arange(data.shape[1])
- yscale: *1darray or None*
  y-scale. None means np.arange(data.shape[0])
- rev: *tuple of bool*
  Reverse scale (x, y).
- n_xticks: *int*
  Number of ticks of the x axis
- n_yticks: *int*
  Number of ticks of the y axis
- n_zticks: *int*
  Number of ticks of the color bar
- fontsize: *float*
  Biggest font size to apply to the figure.
- name: *str or None*
  Filename for the figure. Set to None to show the figure.

3.3.12 figures.plot_fid(fid, name=None, ext='png', dpi=600)

Makes a two-panel figure that shows on the left the real part of the FID, on the right the imaginary part. The x-scale and y-scale are automatically adjusted.

3.3.13 figures.plot_fid_re(fid, scale=None, c='b', lims=None, name=None, ext='png', dpi=600)

Makes a single-panel figure that shows either the real or the imaginary part of the FID. The x-scale and y-scale are automatically adjusted.

Parameters:

- fid: *ndarray*
  FID to be plotted
- scale: *1darray or None*
  x-scale of the figure
- c: *str*
  Color
- lims: *tuple or None*
  Limits
- name: *str*
  Name of the figure
- ext: *str*
  Format of the image
- dpi: *int*
  Resolution of the image in dots per inches

3.3.14 figures.redraw_contours(ax, ppm_f2, ppm_f1, S, lvl, cnt, Neg=False, Ncnt=None, lw=0.5, cmap=[None, None])

Redraws the contours in interactive 2D visualizations.

Parameters:

- ax: *matplotlib.Subplot Object*
  Panel of the figure where to draw the contours
- ppm_f2: *1darray*
  ppm scale of the direct dimension
- ppm_f1: *1darray*
  ppm scale of the indirect dimension
- S: *2darray*
  Spectrum
- lvl: *float*
  Level at which to draw the contours
- cnt: *matplotlib.contour.QuadContourSet object*
  Pre-existing contours
- Neg: *bool*
  Choose if to draw the negative contours (`True`) or not (`False`)
- Ncnt: *matplotlib.contour.QuadContourSet object*
  Pre-existing negative contours
- lw: *float*
  Linewidth
- cmap: *list*
  Colour of the contours. Format: `[cmap +, cmap -]`

Returns:

- cnt: *matplotlib.contour.QuadContourSet object*
  Updated contours
- Ncnt: *matplotlib.contour.QuadContourSet object or None*
  Updated negative contours if `Neg` is `True`, `None` otherwise

3.3.15 figures.sns_heatmap(data, name=None, ext='png', dpi=600)

Computes a heatmap of data, which is a matrix. This function employs the *seaborn* package. Specify 'name' if you want to save the figure.

Parameters:

- data: *2darray*
  Data of which to compute the heatmap. Make sure the entries are real numbers.
- name: *str or None*
  Filename of the figure to be saved. If None, the figure is shown instead.
- ext: *str*
  Format of the image
- dpi: *int*
  Resolution of the image in dots per inches

3.3.16 figures.stacked_plot(ppmscale, S, xlims=None, lw=0.5, X_label='$ delta $ F1 /ppm', Y_label='Normalized intensity /a.u.', n_xticks=10, labels=None, name=None, ext='png', dpi=600)

Creates a stacked plot of all the spectra contained in the list 'S'. Note that 'S' MUST BE a list. All the spectra must share the same scale.

Parameters:

- ppmscale: *1darray*
  ppm scale of the spectrum
- S: *list*
  spectra to be plotted
- xlims: *list or tuple*
  Limits for the x-axis. If None, the whole scale is used.
- lw: *float*
  linewidth
- name: *str*
  filename of the figure, if it has to be saved;
- X_label: *str*
  text of the x-axis label;
- Y_label: *str*
  text of the y-axis label;
- n_xticks: *int*
  Number of numbered ticks on the x-axis of the figure
- labels: *list*
  labels to be put in the legend.

3.4 SIM package

This package contains function for the simulation of various features of NMR spectra, being them monodimensional or bidimensional. Functions for the simulation of whole spectra are also provided.

3.4.1 sim.calc_splitting(u0, I0, m=1, J=0)

Calculate the frequency and the intensities of a NMR signal splitted by scalar coupling.

Parameters:

- u0: *float*
  Frequency of the non-splitted signal (Hz)
- I0: *float*
  Total intensity of the non-splitted signal.
- m: *int*
  Multiplicity, i.e. number of expected signals after the splitting
- J: *float*
  Scalar coupling constant (Hz)

Returns:

- u_s: *1darray*
  Frequencies of the splitted signal (Hz)
- I_s: *1darray*
  Intensities of the splitted signal

3.4.2 sim.f_gaussian(x, u, s, A=1)

Gaussian function in the frequency domain:

$$\mathcal{G}(x) = \frac{A}{\sqrt{2\pi}s} \exp\left[-\frac{1}{2}\left(\frac{x-u}{s}\right)^2\right]$$

Parameters

- x: *1darray*
  Independent variable
- u: *float*
  Peak position
- s: *float*
  Standard deviation
- A: *float*
  Intensity

Returns

- f: *1darray*
  Gaussian function.

3.4.3 sim.f_lorentzian(x, u, fwhm, A=1)

Lorentzian function in the frequency domain:

$$\mathcal{L}(x) = \frac{A}{\pi} \frac{\gamma}{(x-u)^2 + \gamma^2}$$

Parameters

- x: *1darray*
  Independent variable
- u: *float*
  Peak position
- fwhm: *float*
  Full-width at half-maximum, $\Gamma = 2\gamma$
- A: *float*
  Intensity

Returns

- f: *1darray*
  Lorentzian function.

3.4.4 sim.f_pvoigt(x, u, fwhm, A=1, x_g=0)

Pseudo-Voigt function in the frequency domain:

$$S(x) = x_g \mathcal{G}(x) + (1 - x_g)\mathcal{L}(x)$$

This is practically done by:

```
s = fwhm / 2.355
S = A* (sim.f_gaussian(x, u, s, A=x_g) + sim.f_lorentzian(x, u, fwhm, A=1-x_g))
```

Parameters

- x: *1darray*
  Independent variable
- u: *float*
  Peak position
- fwhm: *float*
  Full-width at half-maximum
- A: *float*
  Intensity
- x_g: *float*
  Fraction of gaussianity

Returns

- S: *1darray*
  Pseudo-Voigt function.

3.4.5 sim.gaussian_filter(ppm, u, s)

Compute a gaussian filter to be used in order to suppress signals in the spectrum.

Parameters:

- ppm: *1darray*
  Scale on which to build the filter
- u: *float*
  Position of the filter
- s: *float*
  Width of the filter (standard deviation)

Returns:

- G: *1darray*
  Computed gaussian filter

3.4.6 sim.load_sim_1D(File)

Creates a dictionary from the spectral parameters listed in the input file.

Template of the input file:

```
B0  16.4              # Magnetic field strength /Tesla
nuc 1H                # Observed nucleus
o1p 4.7               # Pulse carrier frequency /ppm
SWp 30                # Spectral width /ppm
TD  4096              # Number of sampled complex points
shifts 0, 4.5, 3      # Peak shifts /ppm, separated by commas
fwhm   100, 100, 100  # Full-width at half maximum of the peaks /Hz, separated by commas
amplitudes 1, 4, 3    # Amplitudes of the peaks, separated by commas
x_g 0.0, 0.5, 1.0     # Fraction of gaussianity of the peaks (1 = 100% gaussian, 0 = 100%
    lorentzian)
```

Use the `tab` character to separate the variable from its value. Comments are placed here to explain what the variables stand for, they are not needed in a real input file. However, if you want to put them, use the `#` character to denote them. Note that the variable names **CANNOT** be changed for any reason: the penalty is a massive sequence of errors. If only one value is supplied for the fields `shifts`, `fwhm`, `amplitudes` and `x_g`, that value must be followed by a comma in order to allow the program to recognize the list as a tuple.

Parameters

- File: *str*
  Path to the input file location

Returns

- dic: *dict*
  Dictionary of the parameters, ready to be read from the simulation functions.

3.4.7 sim.load_sim_2D(File, states=True)

Creates a dictionary from the spectral parameters listed in the input file.

Template of the input file:

```
B0  28.2            # Magnetic field strength /Tesla
nuc1    15N         # Observed nucleus in indirect dimension (F1)
nuc2    1H          # Observed nucleus in direct dimension (F2)
o1p 115             # Pulse carrier frequency /ppm in F1
o2p 5               # Pulse carrier frequency /ppm in F2
SW1p    40          # F1 Spectral width /ppm
SW2p    20          # F2 Spectral width /ppm
TD1 64              # Number of t1 increment in indirect dimension
TD2 256             # Number of sampled complex points for each transient
shifts_f1  130.0, 105.0, 120.0      # Peak F1 shifts /ppm, separated by commas
shifts_f2  0.0, 0.0, 7.0            # Peak F2 shifts /ppm, separated by commas
fwhm_f1 100, 100, 100               # Full-width at half maximum of the peaks in F1
    (/Hz), separated by commas
fwhm_f2 500, 500, 500               # Full-width at half maximum of the peaks in F2
    (/Hz), separated by commas
amplitudes 100.0, 200.0, 100.0      # Intensity of the peaks, separated by commas
x_g 0.0, 0.5, 1.0                   # # Fraction of gaussianity of the peaks (1 = 100%
    gaussian, 0 = 100% lorentzian)
```

Use the `tab` character to separate the variable from its value. Comments are placed here to explain what the variables stand for, they are not needed in a real input file. However, if you want to put them, use the `#` character to denote them. Note that the variable names **CANNOT** be changed for any reason: the penalty is a massive sequence of errors. If only one value is supplied for the fields `shifts`, `fwhm`, `amplitudes` and `x_g`, that value must be followed by a comma in order to allow the program to recognize the list as a tuple.

Parameters

- File: *str*
  Path to the input file location
- states: *bool*
  If *FnMODE* is *States* or *States-TPPI*, set it to `True` to get the correct timescale.

Returns

- dic: *dict*
  Dictionary of the parameters, ready to be read from the simulation functions.

3.4.8 sim.mult_noise(data_size, mean, s_n)

Multiplicative noise model.

3.4.9 sim.multiplet(u, I, m='s', J=[])

Split a given signal according to a scalar coupling pattern.

Parameters:

- u: *float*
  Frequency of the non-splitted signal (Hz)
- I: *float*
  Intensity of the non-splitted signal
- m: *str*
  Organic chemistry-like multiplet, i.e. s, d, dqt, etc.
- J: *float or list*
  Scalar coupling constants. The number of constants should match the number of coupling branches

Returns:

- u_in: *list*
  List of the splitted frequencies (Hz)
- I_in: *list*
  Intensities of the splitted signal

3.4.10 sim.noisegen(size, o2, t2, s_n=1)

Simulates additive noise in the time domain, in the form of a matrix of dimensions `size`.

This model for the noise depicts it as a white noise vector (i.e. random number that fit a gaussian distribution with 0 mean and standard deviation equal to `s_n`), modulated for the carrier frequency `o2`.

We consider the noise as the sum of a correlated contribution, due to the fact that the signal in a real spectrometer travels through the same cables until the ADC, and of a non-correlated contribution, which arises from the separation into real channel and imaginary channel.

This translates in the following code:

```
# correlated part of noise until ADC
white_corr = np.random.normal(0, s_n, size)
# white noise in FID has to be centered on the offset frequency
noise_corr = white_corr * np.exp(1j* 2 * np.pi * o2 * t2)

# uncorrelated part of noise: quadrature detection
white_re = np.random.normal(0, s_n, size)
white_im = np.random.normal(0, s_n, size)
# cosine-modulated in the real channel and sine-modulated in the imaginary channel
noise_re = white_re * np.cos( 2* np.pi * o2 * t2)
noise_im = white_im * np.sin( 2* np.pi * o2 * t2)

# final noise is sum of the two parts
noise = noise_corr + (noise_re + 1j*noise_im)
```

Parameters

- size: *int or tuple*
  Dimension of the noise matrix
- o2: *float*
  Carrier frequency, in Hz.
- t2: *1darray*
  Time scale of the last temporal dimension.
- s_n: *float*
  Standard deviation of the noise.

Returns

- noise: *2darray*
  Noise matrix, of dimensions `size`.

3.4.11 sim.sim_1D(File, pv=False)

Simulates a 1D NMR spectrum from the instructions written in `File`.

The instructions on how to write the input file are reported in the caption of `sim.load_sim_1D`.

Parameters

- File: *str*
  Path to the input file location
- pv: *bool*
  `True` for pseudo-Voigt model, `False` for Voigt model.

Returns

- fid: *1darray*
  FID of the simulated spectrum.

3.4.12 sim.sim_2D(File, states=True, alt=True, pv=False)

Simulates a 2D NMR spectrum from the instructions written in `File`. The indirect dimension is sampled with *FnMODE=States-TPPI* as default.

The instructions on how to write the input file are reported in the caption of `sim.load_sim_2D`.

Parameters

- File: *str*
  Path to the input file location
- states: *bool*
  Set it to `True` to allow for correct spectral arrangement in the indirect dimension.
- alt: *bool*
  Set it to `True` to allow for correct spectral arrangement in the indirect dimension.
- pv: *bool*
  `True` for pseudo-Voigt model, `False` for Voigt model.

Returns

- fid: *2darray*
  FID of the simulated spectrum.

3.4.13 sim.t_2Dgaussian(t1, t2, v1, v2, s1, s2, A=1, states=True, alt=True)

Bidimensional gaussian peak in the time domain. The working code requires `states=True` and `alt=True`.

The signal is generated as follows:

```
# States acquires twice the same point of the indirect dimension time domain
t1[1::2] = t1[::2]
# TPPI cycles the receiver phase of 90 degrees at each transient acquisition
freq_1 = np.zeros(len(t1), dtype='complex64')
for k in range(4):
    t1t = t1[k::4]
    freq_1[k::4] = np.cos( (2 * np.pi * v1 * t1t) - (0.5 * np.pi * np.mod(k,4) ))

# NMR signal in the direct dimension
F2 = np.exp(1j*2*np.pi*v2*t2) * np.exp(-(s2**2 * t2**2)/2)
# NMR signal in the indirect dimension
F1 = freq_1 * np.exp(-(s1**2 * t1**2)/2)

# The full FID is reconstructed by doing the external product between the two vectors
S = A * F1.reshape(-1,1) @ F2.reshape(1,-1)
```

Parameters

- t1: *1darray*
  Indirect evolution timescale
- t2: *1darray*
  Timescale of the direct dimension
- v1: *float*
  Peak position in the indirect dimension, in Hz
- v2: *float*
  Peak position in the direct dimension, in Hz
- s1: *float*
  Standard deviation in the indirect dimension, in rad/s
- s2: *float*
  Standard deviation in the direct dimension, in rad/s
- A: *float*
  Intensity
- states: *bool*
  Set to `True` for *FnMODE = States-TPPI*
- alt: *bool*
  Set to `True` for *FnMODE = States-TPPI*

Returns

- S: *2darray*
  2D Gaussian function.

3.4.14 sim.t_2Dlorentzian(t1, t2, v1, v2, fwhm1, fwhm2, A=1, states=True, alt=True)

Bidimensional Lorentzian peak. The working code requires `states=True` and `alt=True`.

The signal is generated as follows:

```
hwhm1 = fwhm1 / 2
hwhm2 = fwhm2 / 2

# States acquires twice the same point of the indirect dimension time domain
t1[1::2] = t1[::2]
# TPPI cycles the receiver phase of 90 degrees at each transient acquisition
freq_1 = np.zeros(len(t1), dtype='complex64')
for k in range(4):
    t1t = t1[k::4]
    freq_1[k::4] = np.cos( (2 * np.pi * v1 * t1t) - (0.5 * np.pi * np.mod(k,4) ))

# NMR signal in the direct dimension
F2 = np.exp(1j*2*np.pi*v2*t2) * np.exp(-(hwhm2 * t2))
# NMR signal in the indirect dimension
F1 = freq_1 * np.exp(-(hwhm1 * t1))


# The full FID is reconstructed by doing the external product between the two vectors
S = A * F1.reshape(-1,1) @ F2.reshape(1,-1)
```

Parameters

- t1: *1darray*
  Indirect evolution timescale
- t2: *1darray*
  Timescale of the direct dimension
- v1: *float*
  Peak position in the indirect dimension, in Hz
- v2: *float*
  Peak position in the direct dimension, in Hz
- fwhm1: *float*
  Full-width at half maximum in the indirect dimension, in rad/s
- fwhm2: *float*
  Full-width at half maximum in the direct dimension, in rad/s
- A: *float*
  Intensity
- states: *bool*
  Set to `True` for 'FnMODE':'States-TPPI
- alt: *bool*
  Set to `True` for 'FnMODE':'States-TPPI

Returns

- S: *2darray*
  Lorentzian function.

3.4.15 sim.t_2Dpvoigt(t1, t2, v1, v2, fwhm1, fwhm2, A=1, x_g=0.5, states=True, alt=True)

Generates a 2D pseudo-voigt signal in the time domain. `x_g` states for the fraction of gaussianity, whereas `A` defines the overall amplitude of the total peak. Indexes '1' and '2' on the variables stand for 'F1' and 'F2', respectively.

```
# stdev computed for the gaussian part.
s1 = fwhm1 / 2.355
s2 = fwhm2 / 2.355
# Passing 's' to 'gaussian' and 'fwhm' to 'lorentzian' makes the two parts of the
    pseudo-voigt signal to have the same width and allow proper summation
G = sim.t_2Dgaussian(t1, t2, v1, v2, s1, s2, A=x_g, states=states, alt=alt)
L = sim.t_2Dlorentzian(t1, t2, v1, v2, fwhm1, fwhm2, A=(1-x_g), states=states, alt=alt)
fid = A * (G + L)
```

Parameters

- t1: *1darray*
  Indirect evolution timescale
- t2: *1darray*
  Timescale of the direct dimension
- v1: *float*
  Peak position in the indirect dimension, in Hz
- v2: *float*
  Peak position in the direct dimension, in Hz
- fwhm1: *float*
  Full-width at half maximum in the indirect dimension, in rad/s
- fwhm2: *float*
  Full-width at half maximum in the direct dimension, in rad/s
- A: *float*
  Intensity
- x_g: *float*
  Fraction of gaussianity
- states: *bool*
  Set to `True` for *FnMODE=States-TPPI*
- alt: *bool*
  Set to `True` for *FnMODE=States-TPPI*

Returns

- fid: *2darray*
  2D Pseudo-Voigt function.

3.4.16 sim.t_2Dvoigt(t1, t2, v1, v2, fwhm1, fwhm2, A=1, x_g=0.5, states=True alt=True)

Generates a 2D Voigt signal in the time domain. `x_g` states for the fraction of gaussianity, whereas `A` defines the overall amplitude of the total peak. Indexes '1' and '2' on the variables stand for 'F1' and 'F2', respectively.

```
# stdev computed for the gaussian part.
s1 = fwhm1 / 2.355
s2 = fwhm2 / 2.355
# hwhm computed for the lorentzian part.
hwhm1 = fwhm1 / 2
hwhm2 = fwhm2 / 2

# States acquires twice the same point of the indirect dimension time domain
t1[1::2] = t1[::2]

# direct dimension
#   frequency
freq_2 = np.exp(1j * 2 * np.pi * v2 * t2)

#   Add line-broadening, fist lorentzian then gaussian, using:
#   hwhm' = (1 - x_g) * hwhm     for L
#   s' = x_g * s                 for G
F2 = freq_2 * np.exp(-(1-x_g)*hwhm2 * t2) * np.exp(-((x_g*s2)**2 * t2**2)/2)

# indirect dimension
# Redfield cycles the receiver phase of 90 degrees at each transient acquisition
freq_1 = np.zeros(len(t1), dtype='complex64')
for k in range(4):
    t1t = t1[k::4]
    freq_1[k::4] = np.cos( (2 * np.pi * v1 * t1t) - (0.5 * np.pi * np.mod(k,4) ))

#   Add line-broadening, fist lorentzian then gaussian, using:
#   hwhm' = (1 - x_g) * hwhm     for L
#   s' = x_g * s                 for G
F1 = freq_1 * np.exp(-(1-x_g) * hwhm1 * t1) * np.exp(-((x_g*s1)**2 * t1**2)/2)

# The full FID is reconstructed by doing the external product between the two vectors
S = A * F1.reshape(-1,1) @ F2.reshape(1,-1)
return S
```

Parameters

- t1: *1darray*
  Indirect evolution timescale
- t2: *1darray*
  Timescale of the direct dimension
- v1: *float*
  Peak position in the indirect dimension, in Hz
- v2: *float*
  Peak position in the direct dimension, in Hz

- fwhm1: *float*
  Full-width at half maximum in the indirect dimension, in rad/s
- fwhm2: *float*
  Full-width at half maximum in the direct dimension, in rad/s
- A: *float*
  Intensity
- x_g: *float*
  Fraction of gaussianity
- states: *bool*
  Set to `True` for 'FnMODE':'States-TPPI
- alt: *bool*
  Set to `True` for 'FnMODE':'States-TPPI

Returns

- S: *2darray*
  Voigt function.

3.4.17 sim.t_gaussian(t, u, s, A=1, phi=0)

Gaussian function in the time domain.

$$g(x) = Ae^{\mathrm{i}\phi}\, e^{\mathrm{i}2\pi ut} e^{-s^2t^2/2}$$

Parameters

- t: *1darray*
  Independent variable
- u: *float*
  Peak position, in Hz
- s: *float*
  Standard deviation, in rad/s
- A: *float*
  Intensity
- phi: *float*
  Phase, in radians

Returns

- S: *1darray*
  Gaussian function.

3.4.18 sim.t_lorentzian(t, u, fwhm, A=1, phi=0)

Lorentzian function in the time domain.

$$\ell(x) = Ae^{\mathrm{i}\phi}\, e^{\mathrm{i}2\pi ut} e^{-\gamma t}$$

Parameters

- t: *1darray*
  Independent variable
- u: *float*
  Peak position, in Hz
- fwhm: *float*
  Full-width at half-maximum, $\Gamma = 2\gamma$, in rad/s
- A: *float*
  Intensity
- phi: *float*
  Phase, in radians

Returns

- S: *1darray*
  Lorentzian function.

3.4.19 sim.t_pvoigt(t, u, fwhm, A=1, x_g=0, phi=0)

Pseudo-Voigt function in the time domain:

$$s(x) = x_g g(x) + (1 - x_g)\ell(x)$$

```
s = fwhm / 2.355
S = A * (sim.t_gaussian(t, u, s, A=x_g, phi=phi) + sim.t_lorentzian(t, u, fwhm, A=1-x_g,
    phi=phi))
```

Parameters

- t: *1darray*
  Independent variable
- u: *float*
  Peak position, in Hz
- fwhm: *float*
  Full-width at half-maximum, in rad/s
- A: *float*
  Intensity
- x_g: *float*
  Fraction of gaussianity
- phi: *float*
  Phase, in radians

Returns

- S: *1darray*
  Pseudo-Voigt function.

3.4.20 sim.t_voigt(t, u, fwhm, A=1, x_g=0, phi=0)

Voigt function in the time domain. The parameter `x_g` affects the linewidth of the lorentzian and gaussian contributions.

```
s = fwhm / 2.355
S = A * np.exp(1j*phi) * sim.t_gaussian(t, u/2, s*x_g) * sim.t_lorentzian(t, u/2,
    fwhm*(1-x_g))
```

Parameters

- t: *1darray*
  Independent variable
- u: *float*
  Peak position, in Hz
- fwhm: *float*
  Full-width at half-maximum, in rad/s
- A: *float*
  Intensity
- x_g: *float*
  Fraction of gaussianity
- phi: *float*
  Phase, in radians

Returns

- S: *1darray*
  Voigt function.

3.4.21 sim.water7(N, t2, vW, fwhm=300, A=1, spread=701.125)

Simulates a feature like the water ridge in HSQC spectra, in the time domain.

This signal is modelled as a gaussian signal which does not encode for any frequency in the indirect dimension, and whose chemical shift moves due to field drifts through the various transients according to a gaussian distribution. This signal is on-phase in the even transients and 90°-dephased in the odd transients.

Parameters

- N: *int*
  Number of transients
- t2: *1darray*
  Time scale of the last temporal dimension.
- vW: *float*
  Nominal peak position, in Hz.
- fwhm: *float*
  Nominal full-width at half maximum of the peak, in rad/s
- A: *float*
  Signal intensity.
- spread: *float*
  Standard deviation of the peak position distribution, in Hz.

Returns

- ridge: *2darray*
  Matrix of the ridge.

3.5 FIT package

Functions for performing fits.

3.5.1 fit.CostFunc `class`

Class that groups several ways to compute the target of the minimization in a fitting procedure. It includes the classic squared sum of the residuals, as well as some other non-quadratic cost functions. Let x be the residuals and s the chosen threshold value. Then the objective value R is computed as:

$$R = \sum\_if(x_i)$$

where $f(x)$ can be chosen between the following options:

- Quadratic:

$$f(x) = x^2$$

- Truncated Quadratic:

$$f(x) = \begin{cases} x^2 & \text{if } |x| < s \\ s^2 & \text{otherwise} \end{cases}$$

- Huber function:

$$f(x) = \begin{cases} x^2 & \text{if } |x| < s \\ 2s|x| - s^2 & \text{otherwise} \end{cases}$$

- Asymmetric Truncated Quadratic:

$$f(x) = \begin{cases} x^2 & \text{if } x < s \\ s^2 & \text{otherwise} \end{cases}$$

- Asymmetric Huber function:

$$f(x) = \begin{cases} x^2 & \text{if } x < s \\ 2sx - s^2 & \text{otherwise} \end{cases}$$

Attributes:

- method: *function*
  Function to be used for the computation of the objective value. It must take as input the array of the residuals and the threshold, no matter if the latter is actually used or not.

- s: *float*
  Threshold value

Methods:

__init__(self, method='q', s=None)

Initialize the method according to your choice, then stores the threshold value in the attribute 's'. Allowed choices are:

- 'q': Quadratic
- 'tq': Truncated Quadratic
- 'huber': Huber function
- 'atq': Asymmetric Truncated Quadratic
- 'ahuber': Asymmetric Huber function

Parameters:

- method: *str*
  Label for the method selection
- s: *float*
  Threshold value

__call__(self, x)

Computes the objective value according to the chosen method and the residuals array `x`.

Parameters:

- x: *1darray*
  Array of the residuals

Returns:

- R: *float*
  Computed objective value

---

asymm_huber(r, s)

Linear behaviour above s, penalizes negative entries

---

asymm_truncated_quadratic(r, s)

Constant behaviour above s, penalizes negative entries

---

huber(r, s)

Linear behaviour above s

---

method_selector(self, method)

Performs the selection of the method according to the identifier string.

Parameters:

- method: *str*
  Method label

Returns:

- f: *function*
  Selected model

---

squared_sum(r, s=0)

Quadratic everywhere

---

truncated_quadratic(r, s)

Constant behaviour above s

3.5.2 fit.LR(y, x=None)

Performs a linear regression of y with a model $y_c = mx + q$.

Parameters:

- y: *1darray*
  Data to be fitted
- x: *1darray*
  Independent variable. If `None`, the point indexes are used.

Returns:

- y_c: *1darray*
  Fitted trend
- values: *tuple*
  (m, q)

3.5.3 fit.LSP(y, x, n=5)

Make a polynomial fit on the experimental data y by solving the linear system

$$\vec{y} = \mathbb{T}\vec{c}$$

where $\mathbb{T}$ is the Vandermonde matrix of the x-scale and c is the set of coefficients that minimize the problem in the least-squares sense.

Parameters:

- y: *1darray*
  Experimental data
- x: *1darray*
  Independent variable (better if normalized)
- n: *int*
  Order of the polynomion + 1, i.e. number of coefficients

Returns:

- c: *1darray*
  Set of minimized coefficients

3.5.4 fit.SINC_ObjFunc `class`

Computes the objective function as explained in 'M. Sawall et al., *Journal of Magnetic Resonance* 289 (2018), 132-141'.

The cost function is computed as:

$$f(d) = \sum_{i=1}^{3} \gamma_i g_i(d|\varepsilon_i)$$

where d is the real part of the NMR spectrum.

Attributes:

- gamma1: *float*
  Weighting factor for function g1
- gamma2: *float*
  Weighting factor for function g2
- gamma3: *float*
  Weighting factor for function g3
- e1: *float*
  Tolerance value for function g1
- e2: *float*
  Tolerance value for function g2

Methods:

__init__(self, gamma1=10, gamma2=0.01, gamma3=0, e1=0, e2=0)

Initialize the coefficients used to weigh the objective function.

Parameters:

- gamma1: *float*
  Weighting factor for function g1
- gamma2: *float*
  Weighting factor for function g2
- gamma3: *float*
  Weighting factor for function g3
- e1: *float*
  Tolerance value for function g1
- e2: *float*
  Tolerance value for function g2

__call__(self, d)

Computes the objective function f as explained in the paper

g1(d, e1=0)

Penalty function for negative entries of the spectrum

Parameters:

- d: *1darray*
  Spectrum
- e1: *float*
  Tolerance for negative entries

---

g2(d, e2=0)

Regularization function that favours the smallest integral.

Parameters:

- d: *1darray*
  Spectrum
- e2: *float*
  Tolerance for ideal baseline

---

g3(d)

Regularization function for the smoothing.

Parameters:

- d: *1darray*
  Spectrum

3.5.5 fit.SINC_phase(data, gamma1=10, gamma2=0.01, gamma3=0, e1=0, e2=0, **fit_kws)

Perform automatic phase correction according to the SINC algorithm, as described in 'M. Sawall et al., *Journal of Magnetic Resonance* 289 (2018), 132-141'. The fitting method defaults to 'least_squares'.

Parameters:

- data: *1darray*
  Spectrum to phase-correct
- gamma1: *float*
  Weighting factor for function g_1: non-negativity constraint
- gamma2: *float*
  Weighting factor for function g_2: smallest-integral constraint
- gamma3: *float*
  Weighting factor for function g_3: smoothing constraint
- e1: *float*
  Tolerance factor for function g_1: adjustment for noise
- e2: *float*
  Tolerance factor for function g_2: adjustment for non-ideal baseline
- fit_kws: *keyworded arguments*
  Additional parameters for the fit function. See lmfit.Minimizer.minimize for details. Do not use 'leastsq' because the cost function returns a scalar value!

Returns:

- p0: *float*
  Fitted zero-order phase correction angle, in degrees
- p1: *float*
  Fitted first-order phase correction angle, in degrees

3.5.6 fit.Voigt_Fit `class`

This class offers an 'interface' to fit a 1D NMR spectrum.

Attributes:

- ppm_scale: *1darray*
  Self-explanatory
- S : *1darray*
  Spectrum to fit. Only real part
- t_AQ: *1darray*
  acquisition timescale of the spectrum
- SFO1: *float*
  Larmor frequency of the nucleus
- o1p : *float*
  Pulse carrier frequency
- nuc : *str or None*
  Nucleus. Used to write the X_scale of the plot.
- input_file : *str*
  filename of the input file
- output_file : *str*
  filename of the output file
- log_file : *str*
  filename of the log file
- limits : *tuple*
  borders of the fitting window
- Vi : *2darray*
  array with the values of the signals used as initial guess
- Ci : *1darray*
  coefficients of the baseline polynomion as initial guess
- Vf : *2darray*
  array with the values of the signals after the fit
- Cf : *1darray*
  coefficients of the baseline polynomion after the fit
- s_labels : *list*
  legend entries for the single signals.

Methods:

__init__(self, ppm_scale, S, t_AQ, SFO1, o1p, nuc=None)

Initialize the class with common values.

Parameters:

- ppm_scale: *1darray*
  ppm scale of the spectrum
- S: *1darray*
  Spectrum to be fitted
- t_AQ: *1darray*
  Acquisition timescale
- SFO1: *float*
  Larmor frequency of the observed nucleus, in MHz
- o1p: *float*
  Carrier position, in ppm
- nuc: *str*
  Observed nucleus. Used to customize the x-scale of the figures.

---

dofit(self, input_file=None, output_file=None, log_file='fit.log', on_cplex=True, **kwargs)

Performs the fit using fit.voigt_fit with self.Vi and self.Ci as initial parameters.

Parameters:

- input_file: *str*
  Path to the input file. If None, self.input_file is used.
- output_file: *str*
  Path to the output file. If None, self.output_file is used.
- log_file: *str*
  Path to the log file.
- on_cplex: *bool*
  Choose if to fit on the complex data (True) or on the real data (False).
- kwargs: *keyworded parameters*
  See fit.voigt_fit.

---

get_fit_lines(self, what='fit')

Calculates the components, total fit curve and baseline used for iguess or fit. Calls fit.calc_fit_lines to do it.

Parameters:

- what: *str*
  'iguess' or 'fit'. self.Vi and self.Ci or self.Vf and self.Cf are read, respectively.

Returns:

- signals: *list of 1darray*
  Components used for the fit
- Total: *1darray*
  Sum of all the signals
- baseline: *1darray*
  Baseline.

---

iguess(self, input_file=None, limits=None)

Reads, or computes, the initial guess for the fit. The window limits can be set interactively or specified through the parameter 'limits'.

Parameters:

- input_file: *str or None*
  Path to the input file. If None, the default name 'ppm1:ppm2.inp' is set.
- limits: *tuple or None*
  (lim_sx, lim_dx) in ppm. If None and input_file does not exist yet, they are set interactively.

---

load_fit(self, output_file=None)

Reads the output file of a fit and stores the values in self.Vf, self.Cf and self.limits. self.output_file is also replaced with output_file.

Parameters:

- output_file: *str*
  Path to the output file to be read

---

plot(self, what='fit', s_labels=None, **kwargs)

Makes a figure of either the initial guess or the fit. Calls figures.fitfigure.

Parameters:

- what: *str*
  'iguess' or 'fit'. self.Vi and self.Ci or self.Vf and self.Cf are read, respectively.
- s_labels: *list*
  Labels for the components that will appear in the legend of the figure.
- kwargs: *keyworded parameters*
  See figures.fitfigure for the additional parameters.

3.5.7 fit.ax_histogram(ax, data0, nbins=100, density=True, f_lims=None, xlabel=None, x_symm=False, barcolor='tab:blue', fontsize=10)

Computes an histogram of 'data' and tries to fit it with a gaussian lineshape. The parameters of the gaussian function are calculated analytically directly from 'data' using 'scipy.stats.norm'

Parameters:

- ax : *matplotlib.subplot Object*
  panel of the figure where to put the histogram
- data0 : *ndarray*
  the data to be binned
- nbins : *int*
  number of bins to be calculated
- density : *bool*
  True for normalize data
- f_lims : *tuple or None*
  limits for the x axis of the figure
- xlabel : *str or None*
  Text to be displayed under the x axis
- x_symm : *bool*
  set it to True to make symmetric x-axis with respect to 0
- barcolor: *str*
  Color of the bins
- fontsize: *float*
  Biggest fontsize in the figure

Returns:

- m : *float*
  Mean of data
- s : *float*
  Standard deviation of data.

3.5.8 fit.bin_data(data0, nbins=100, density=True, x_symm=False)

Computes the histogram of `data`, sampling it into `nbins` bins. If `data` is multidimensional, it is flattened.

Parameters:

- data : *ndarray*
  the data to be binned
- nbins : *int*
  number of bins to be calculated
- density : *bool*
  True for normalize `data`
- x_symm : *bool*
  set it to `True` to make symmetric x-axis with respect to 0

Returns:

- hist: *1darray*
  The bin intensity
- bin_scale: *1darray*
  Scale built with the mean value of the bin widths.

3.5.9 fit.build_2D_sgn(parameters, acqus, N=None, procs=None)

Create a 2D signal according to the final parameters returned by `fit.gen_iguess_2D`. Process it according to `procs`.

Parameters:

- parameters: *list or 2darray*
  sequence of the parameters: `u1, u2, fwhm1, fwhm2, I, x_g`. Multiple components are allowed
- acqus: *dict*
  2D-like acqus dictionary containing the acquisition timescales (keys: `t1` and `t2`)
- N: *tuple of int*
  Zero-filling values `(F1, F2)`. Read only if `procs` is `None`
- procs: *dict*
  2D-like procs dictionary.

Returns:

- peak: *2darray*
  rr part of the generated signal

3.5.10 fit.build_baseline(ppm_scale, C, L=None)

Builds the baseline calculating the polynomion with the given coefficients, and summing up to the right position.

Parameters:

- ppm_scale: *1darray*
  ppm scale of the spectrum
- C: *list*
  Baseline coefficients. No baseline corresponds to `False`.
- L: *list*
  List of window regions. If it is `None`, the baseline is built on the whole `ppm_scale`

Returns:

- baseline: *1darray*
  Self-explanatory.

3.5.11 fit.calc_fit_lines(ppm_scale, limits, t_AQ, SFO1, o1p, N, V, C=False)

Given the values extracted from a fit input/output file, calculates the signals, the total fit function, and the baseline.

Parameters:

- ppm_scale: *1darray*
  PPM scale of the spectrum
- limits: *tuple*
  `(left, right)` in ppm
- t_AQ: *1darray*
  Acquisition timescale
- SFO1: *float*
  Larmor frequency of the nucleus /ppm
- o1p: *float*
  Pulse carrier frequency /ppm
- N: *int*
  Size of the final spectrum.
- V: *2darray*
  Matrix containing the values to build the signals.
- C: *1darray*
  Baseline polynomion coefficients. `False` to not use the baseline

Returns:

- sgn: *list*
  Voigt signals built using `V`
- Total: *1darray*
  sum of all the arrays in `sgn`
- baseline: *1darray*
  Polynomion built using `C` as coefficients. False if `C` is `False`.

3.5.12 fit.dic2mat(dic, peak_names, ns, A=None)

This is used to make the matrix of the parameters starting from a dictionary like the one produced by *lmfit*. The column of the total intensity is not added, unless the parameter `A` is passed. In this case, the third column (which is the one with the relative intesities) is corrected using the function `misc.molfrac`.

Parameters:

- dic : *dict*
  input dictionary
- peak_names : *list*
  list of the parameter entries to be looked for
- ns : *int*
  number of signals to unpack
- A : *float or None*
  Total intensity.

Returns:

- V : *2darray*
  Matrix containing the parameters.

3.5.13 fit.fit_int(y, y_c)

Computes the optimal intensity and intercept of a linear model in the least squares sense. Let y be the experimental data and y_c the model, and let $\langle w \rangle$ the mean of variable w. Then:

$$A = \frac{\langle y\, y_c \rangle - \langle y \rangle \langle y_c \rangle}{\langle y_c^2 \rangle - \langle y_c \rangle^2}$$

$$q = \frac{\langle y \rangle \langle y_c^2 \rangle - \langle y\, y_c \rangle \langle y_c \rangle}{\langle y_c^2 \rangle - \langle y_c \rangle^2}$$

Parameters:

- y: *1darray*
  Experimental data
- y_c: *1darray*
  Model data

Returns:

- A: *float*
  Optimized intensity
- q: *float*
  Optimized intercept

3.5.14 fit.gaussian_fit(x, y)

Fit `y` with a gaussian function, built using `x` as independent variable. The gaussian function is built with `sim.f_gaussian`.

Parameters:

- x : *1darray*
  x-scale
- y : *1darray*
  data to be fitted

Returns:

- u : *float*
  mean
- s : *float*
  standard deviation
- A : *float*
  Integral

3.5.15 fit.gen_iguess(x, experimental, param, model, model_args=[])

GUI for the interactive setup of a Parameters object to be used in a fitting procedure. Once you initialized the Parameters object with the name of the parameters and a dummy value, you are allowed to set the value, minimum, maximum and vary status through the textboxes given in the right column, and see their effects in real time. Upon closure of the figure, the Parameters object with the updated entries is returned. A maximum of 18 parameters will fit the figure.

Parameters:

- x: *1darray*
  Independent variable
- experimental: *1darray*
  The objective values you are trying to fit
- param: *lmfit.Parameters Object*
  Initialized parameters object
- model: *function*
  Function to be used for the generation of the fit model. `param` must be the first argument.
- model_args: *list*
  List of args to be passed to `model`, after `param`

Returns:

- param: *lmfit.Parameters Object*
  Updated Parameters Object

3.5.16 fit.gen_iguess_2D(ppm_f1, ppm_f2, tr1, tr2, u1, u2, acqus, fwhm0=100 procs=None)

Generate the initial guess for the fit of a 2D signal. The employes model is the one of a 2D Voigt signal, acquired with the States-TPPI scheme in the indirect dimension (i.e. `sim.t_2DVoigt`). The program allows for the inclusion of up to 10 components for the signal, in order to improve the fit. The `acqus` dictionary must contain the following keys:

- `t1`: acquisition timescale in the indirect dimension (States)
- `t2`: acquisition timescale in the direct dimension
- `SFO1`: Larmor frequency of the nucleus in the indirect dimension
- `SFO2`: Larmor frequency of the nucleus in the direct dimension
- `o1p`: carrier position in the indirect dimension /ppm
- `o2p`: carrier position in the direct dimension /ppm

The signals will be processed according to the values in the `procs` dictionary, if given; otherwise, they will be just zero-filled up to the data size (i.e. `(len(ppm_f1), len(ppm_f2))`).

Parameters:

- ppm_f1: *1darray*
  ppm scale for the indirect dimension
- ppm_f2: *1darray*
  ppm scale for the direct dimension
- tr1: *1darray*
  Trace of the original 2D peak in the indirect dimension
- tr2: *1darray*
  Trace of the original 2D peak in the direct dimension
- u1: *float*
  Chemical shift of the original 2D peak in the indirect dimension /ppm
- u2: *float*
  Chemical shift of the original 2D peak in the direct dimension /ppm
- acqus: *dict*
  Dictionary of acquisition parameters
- fwhm0: *float*
  Initial value for FWHM in both dimensions
- procs: *dict*
  Dictionary of processing parameters

Returns:

- final_parameters: *2darray*
  Matrix of dimension `(# signals, 6)` that contains, for each row: ν_1(Hz), ν_2(Hz), Γ_1(Hz), Γ_2(Hz), A, x_g
- fit_interval: *tuple of tuple*
  Fitting window. `( (left_f1, right_f1), (left_f2, right_f2) )`

3.5.17 fit.get_region(ppmscale, S, rev=True)

Interactively select the spectral region to be fitted. Returns the border ppm values.

Parameters:

- ppmscale: *1darray*
  The ppm scale of the spectrum
- S: *1darray*
  The spectrum to be trimmed
- rev: *bool*
  Choose if to reverse ppm scale and data (`True`) or not (`False`).

Returns:

- left: *float*
  Left border of the selected spectral window
- right: *float*
  Right border of the selected spectral window

3.5.18 fit.histogram(data, nbins=100, density=True, f_lims=None, xlabel=None, x_symm=False, barcolor='tab:blue', fontsize=10, name=Nor ext='png', dpi=600)

Computes an histogram of 'data' and tries to fit it with a gaussian lineshape. The parameters of the gaussian function are calculated analytically directly from 'data' using 'scipy.stats.norm'

Parameters:

- data : *ndarray*
  the data to be binned
- nbins : *int*
  number of bins to be calculated
- density : *bool*
  True for normalize data
- f_lims : *tuple or None*
  limits for the x axis of the figure
- xlabel : *str or None*
  Text to be displayed under the x axis
- x_symm : *bool*
  set it to True to make symmetric x-axis with respect to 0
- barcolor: *str*
  Color of the bins
- fontsize: *float*
  Biggest fontsize in the figure
- name : *str*
  name for the figure to be saved
- ext: *str*
  Format of the image
- dpi: *int*
  Resolution of the image in dots per inches

Returns:

- m : *float*
  Mean of data
- s : *float*
  Standard deviation of data.

3.5.19 fit.integrate(ppm0, data0, X_label='$\delta\,$F1 /ppm')

Allows interactive integration of a NMR spectrum through a dedicated GUI. Returns the values as a dictionary, where the keys are the selected regions truncated to the 2nd decimal figure. The returned dictionary contains some predefined keys, as follows:

- `total`: total integrated area
- `ref_pos`: location of the reference peak (`ppm1 : ppm2`)
- `ref_int`: absolute integral of the reference peak
- `ref_val`: for how many nuclei the reference peak integrates

The absolute integral of the x-th peak, I_x, must be calculated according to the formula:

$$I_x = I_x^{\text{relative}} \cdot \frac{\texttt{ref\_int}}{\texttt{ref\_val}}$$

Parameters:

- ppm: *1darray*
  PPM scale of the spectrum
- data: *1darray*
  Spectrum to be integrated.
- X_label: *str*
  Label of the x-axis

Returns:

- f_vals: *dict*
  Dictionary containing the values of the integrated peaks.

3.5.20 fit.integrate_2D(ppm_f1, ppm_f2, data, SFO1, SFO2, fwhm_1=200, fwhm_2=200, utol_1=0.5, utol_2=0.5, plot_result=False)

Function to select and integrate 2D peaks of a spectrum, using dedicated GUIs. Calls integral_2D to do the dirty job.

Parameters:

- ppm_f1: *1darray*
  PPM scale of the indirect dimension
- ppm_f2: *1darray*
  PPM scale of the direct dimension
- data: *2darray*
  real part of the spectrum
- SFO1: *float*
  Larmor frequency of the nucleus in the indirect dimension
- SFO2: *float*
  Larmor frequency of the nucleus in the direct dimension
- fwhm_1: *float*
  Starting FWHM /Hz in the indirect dimension
- fwhm_2: *float*
  Starting FWHM /Hz in the direct dimension
- utol_1: *float*
  Allowed tolerance for `u_1` during the fit. `(u_1-utol_1, u_1+utol_1)`
- utol_2: *float*
  Allowed tolerance for `u_2` during the fit. `(u_2-utol_2, u_2+utol_2)`
- plot_result: *bool*
  True to show how the program fitted the traces.

Returns:

- I: *dict*
  Computed integrals. The keys are `'<ppm f1>:<ppm f2>'` with 2 decimal figures.

3.5.21 fit.interactive_smoothing(x, y, cmap='RdBu')

Interpolate the given data with a 3rd-degree spline. Type the desired smoothing factor in the box and see the outcome directly on the figure. When the panel is closed, the smoothed function is returned.

Parameters:

- x: *1darray*
  Scale of the data
- y: *1darray*
  Data to be smoothed
- cmap: *str*
  Name of the colormap to be used to represent the weights

Returns:

- sx: *1darray*
  Location of the spline points
- sy: *1darray*
  Smoothed y
- s_f: *float*
  Employed smoothing factor for the spline
- weights: *1darray*
  Weights vector

3.5.22 fit.join_par(filenames, ppm_scale, joined_name=None)

Load a series of parameters fit files. Join them together, returning a unique array of signal parameters, a list of coefficients for the baseline, and a list of tuples for the regions. Also, uses the coefficients and the regions to directly build the baseline according to the ppm windows.

Parameters:

- filenames: *list*
  List of directories of the input files.
- ppm_scale: *1darray*
  ppm scale of the spectrum. Used to build the baseline
- joined_name: *str or None*
  If it is not `None`, concatenates the files in the list `filenames` and saves them in a single file named `joined_name`.

Returns:

- V: *2darray*
  Array of joined signal parameters
- C: *list*
  Baseline polynomion coefficients. No baseline corresponds to `False`.
- L: *list*
  List of window regions.
- baseline: *1darray*
  Baseline built from `C` and `L`.

3.5.23 fit.make_iguess(S, ppm_scale, t_AQ, limits=None, SFO1=701.125, o1p=0, rev=True, name='i_guess.inp')

Compute the initial guess for the quantitative fit of 1D NMR spectrum in an interactive manner. When the panel is closed, the values are saved in a file.

Parameters:

- S : *1darray*
  Spectrum to be fitted
- ppm_scale : *1darray*
  Self-explanatory
- t_AQ : *1darray*
  Acquisition timescale
- limits : *tuple or None*
  Trim limits for the spectrum `(left, right)`. If `None`, the whole spectrum is used.
- SFO1 : *float*
  Larmor frequency /MHz
- o1p : *float*
  pulse carrier frequency /ppm
- rev : *bool*
  choose if you want to reverse the x-axis scale (`True`) or not (`False`).
- name : *str*
  name of the file where to save the parameters

Returns:

- V_f : *2darray*
  matrix (# signals, parameters)
- C_f : *1darray or False*
  Coefficients of the polynomion to be used as baseline correction. If the 'baseline' checkbox in the interactive figure panel is not checked, `C_f` is `False`.

3.5.24 fit.make_signal(t, u, s, k, x_g, phi, A, SFO1=701.125, o1p=0, N=None)

Generates a Voigt signal using `sim.t_voigt` on the basis of the specified parameters. Then, makes the Fourier transform and returns it.

Parameters:

- t : *ndarray*
  acquisition timescale
- u : *float*
  chemical shift, in ppm
- s : *float*
  full-width at half-maximum, in hertz
- k : *float*
  relative intensity
- x_g : *float*
  fraction of gaussianity
- phi : *float*
  phase of the signal, in degrees
- A : *float*
  total intensity
- SFO1 : *float*
  Larmor frequency, in MHz
- o1p : *float*
  pulse carrier frequency, in ppm
- N : *int or None*
  length of the final signal array. If `None`, the signal is not zero-filled before to be transformed.

Returns:

- sgn : *1darray*
  generated signal in the frequency domain (just the real part).

3.5.25 fit.peak_pick(ppm_f1, ppm_f2, data, coord_filename='coord.tmp')

Make interactive peak-picking on 2D spectra. The position of the selected signals are saved in `coord_filename`. If `coord_filename` already exists, the new signals are appended at its bottom: nothing is overwritten. Calls `misc.select_traces` for the selection.

Parameters:

- ppm_f1: *1darray*
  ppm scale for the indirect dimension
- ppm_f2: *1darray*
  ppm scale for the direct dimension
- data: *2darray*
  Spectrum to peak-pick. The dimension should match the scale sizes.
- coord_filename: *str*
  Path to the file where to save the peak coordinates

Returns:

- coord: *list*
  List of `(u2, u1)` for each peak

3.5.26 fit.polyn_basl(y, n=5, method='tq', s=0.2, itermax=10000)

Fit the baseline of a spectrum with a low-order polynomion using a non-quadratic objective function. Let y be an array of N points. The polynomion is generated on a normalized scale that goes from -1 to 1 in N steps, and the coefficients are initialized with the ordinary least squares fit. Then, the guess is refined using the objective function of choice employing the trust-region reflective least-squares algorithm.

Parameters:

- y: *1darray*
  Experimental data
- n: *int*
  Order of the polynomion + 1, i.e. number of coefficients
- method: *str*
  Objective function of choice. 'q': quadratic, 'tq': truncated quadratic, 'huber': Huber, 'atq': asymmetric truncated quadratic, 'ahuber': asymmetric huber
- s: *float*
  Relative threshold value for the non-quadratic behaviour of the objective function
- itermax: *int*
  Number of maximum iterations

Returns:

- px: *1darray*
  Fitted polynomion
- c: *list*
  Set of coefficients of the polynomion

3.5.27 fit.print_par(V, C, limits=[None,None])

Prints on screen the same thing that `fit.write_par` writes in a file.

Parameters:

- V : *2darray*
  matrix (# signals, parameters)
- C : *1darray or False*
  Coefficients of the polynomion to be used as baseline correction. If it is `False`, they are not printed.
- limits : *tuple or None*
  Trim limits for the spectrum `(left, right)`.

3.5.28 fit.read_par(filename)

Reads the input file of the fit and returns the values.

Parameters:

- filename: *str*
  directory and name of the input file to be read

Returns:

- V : *2darray*
  matrix (# signals, parameters)
- C : *1darray or False*
  Coefficients of the polynomion to be used as baseline correction. If the corresponding flag is not found in the file, `C_f` is set to `False`.
- limits : *tuple*
  Trim limits for the spectrum `(left, right)`.

3.5.29 fit.smooth_spl(x, y, s_f=1, size=0, weights=None)

Fit the input data with a 3rd-order spline, given the smoothing factor to be applied.

Parameters:

- x: *1darray*
  Location of the experimental points
- y: *1darray*
  Input data to be fitted
- s_f: *float*
  Smoothing factor of the spline. 0=best straight line, 1=native spline.
- size: *int*
  Size of the spline. If `size` is 0, the same dimension as `y` is chosen.
- weights: *1darray*
  Array of weights of the spline points. `None` assign the same weight to all points in `x`.

Returns:

- x_s: *1darray*
  Location of the spline data points.
- y_s: *1darray*
  Spline that fits the data.

3.5.30 fit.test_residuals(R, nbins=100, density=False)

Test the residuals of a fit to see if it was reliable. Returns two values: `SYSDEV` and `Q_G`.

`SYSDEV` is inspired by Svergun's Gnom, and it tells if there are systematic deviations basing on the number of sign changes in the residual. Optimal value must be 1. Values greater than 1 show positive bias, values lesser than 1 show negative bias.
Let `R` be an array of N points, and let N_s be the number of sign changes in `R`. Then:

$$\texttt{SYSDEV} = \frac{N_s}{N/2}$$

`Q_G` is to see the discrepancy between a gaussian function built with the mean and standard deviation of the residuals and the gaussian built fitting the histogram of the residuals. Values go from 0 (worst case) to 1 (best case).
Let x be the scale of the bin edges, and let $G_T(x)$ and $G_F(x)$ be the theoretical and the fitted gaussians respectively.

$$G_T(x) = \frac{A_T}{\sqrt{2\pi}\sigma_T}\exp\left[-\frac{1}{2}\left(\frac{x-\mu_T}{\sigma_T}\right)^2\right] \qquad G_F(x) = \frac{A_F}{\sqrt{2\pi}\sigma_F}\exp\left[-\frac{1}{2}\left(\frac{x-\mu_F}{\sigma_F}\right)^2\right]$$

Then:

$$\texttt{Q\_G} = \frac{\int |G_T(x) - G_F(x)|\,dx}{\int G_T(x)\,dx + \int G_F(x)\,dx}$$

The numerator of this fraction expresses the area of the superposition region of the two gaussians. This value is then related to the worst possible case, i.e. when the two gaussians are totally disjointed.

Parameters:

- R : *1darray*
  Array of the residuals
- nbins : *int*
  number of bins of the histogram, which is also the length of x.
- density : *bool*
  `True` to normalize the histogram, `False` otherwise.

Returns:

- SYSDEV : *float*
  Read full caption
- Q_G : *float*
  Read full caption

3.5.31 fit.voigt_fit(S, ppm_scale, V, C, t_AQ, limits=None, SFO1=701.125, o1p=0, utol=0.5, vary_phi=False, vary_xg=True, hist_name=None, write_out='fit.out', test_res=True)

Fits an NMR spectrum with a set of signals, whose parameters are specifed in the `V` matrix. There is the possibility to use a baseline through the parameter `C`. The signals are computed in the time domain and then Fourier transformed.

Parameters:

- S : *1darray*
  Spectrum to be fitted
- ppm_scale : *1darray*
  Self-explanatory
- V : *2darray*
  matrix (# signals, parameters)
- C : *1darray or False*
  Coefficients of the polynomion to be used as baseline correction. If it is `False`, the baseline correction is not used.
- t_AQ : *1darray*
  Acquisition timescale
- limits : *tuple or None*
  Trim limits for the spectrum `(left, right)`. If `None`, the whole spectrum is used.
- SFO1 : *float*
  Larmor frequency /MHz
- o1p : *float*
  pulse carrier frequency /ppm
- utol : *float*
  tolerance for the chemical shift. The peak center can move in the range $[\mu - \texttt{utol}, \mu + \texttt{utol}]$.
- vary_xg: *bool*
  If it is `False`, the parameter `x_g` cannot be varied during the fitting procedure. Useful when fitting with pure Gaussians or pure Lorentzians.
- vary_basl: *bool*
  If it is False, the baseline is kept fixed at the initial parameters.

Returns:

- C_f : *1darray or False*
  Coefficients of the polynomion to be used as baseline correction, or just `False` if not used.
- V_f : *2darray*
  matrix (# signals, parameters) after the fit
- result : *lmfit.fit_result Object*
  container of all information on the fit

3.5.32 fit.voigt_fit_2D(x_scale, y_scale, data, parameters, lim_f1, lim_f2, acqus, N=None, procs=None, utol=(1,1), s1tol=(0,500), s2tol=(0,500), vary_xg=False, logfile=None)

Function that performs the fit of a 2D peak using multiple components. The program reads a parameter matrix, that contains:

```
u1 /ppm, u2 /ppm, fwhm1 /Hz, fwhm2 /Hz, I /a.u., x_g
```

in each row. The number of rows corresponds to the number of components used for the computation of the final signal. The function returns the analogue version of the parameters matrix, but with the optimized values.

Parameters:

- x_scale: *1darray*
  ppm_f2 of the spectrum, full
- y_scale: *1darray*
  ppm_f1 of the spectrum, full
- data: *2darray*
  spectrum, full
- parameters: *1darray or 2darray*
  Matrix `(# signals, 6)`. Read main caption.
- lim_f2: *tuple*
  Trimming limits for x_scale
- lim_f1: *tuple*
  Trimming limits for y_scale
- acqus: *dict*
  Dictionary of acquisition parameters.
- N: *tuple of ints*
  `len(y_scale), len(x_scale)`. Used only if procs is None
- procs: *dict*
  Dictionary of processing parameters.
- utol: *tuple of floats*
  Tolerance for the chemical shifts `(utol_f1, utol_f2)`. Values will be set to $u_1 \pm$`utol_f1`, $u_2 \pm$ `utol_f2`.
- s1tol: *tuple of floats*
  Range of variations for the fwhm in f1, in Hz
- s2tol: *tuple of floats*
  Range of variations for the fwhm in f2, in Hz
- vary_xg: *bool*
  Choose if to fix the x_g value or not
- logfile: *str or None*
  Path to a file where to write the fit information. If it is `None`, they will be printed into standard output.

Returns:

- out_parameters: *2darray*
  parameters, but with the optimized values.

3.5.33 fit.write_log(input_file, output_file, limits, V_i, C_i, V_f, C_f, result, runtime, test_res=True, log_file='fit.log')

Write a log file with all the information of the fit.

Parameters:

- input_file: *str*
  Location and filename of the input file
- output_file: *str*
  Location and filename of the output file
- limits: *tuple*
  Delimiters of the spectral region that was fitted. `(left, right)`
- V_i: *2darray*
  Initial parameters of the fit
- C_i: *1darray or False*
  Coefficients of the starting polynomion used for baseline correction. If `False`, they are not written.
- V_f: *2darray*
  Final parameters of the fit
- C_f: *1darray or False*
  Coefficients of the final polynomion used for baseline correction. If `False`, they are not written.
- result: *lmfit.FitResult Object*
  Object returned by *lmfit* after the fit.
- runtime: *datetime.datetime Object*
  Time taken for the fit
- test_res: *bool*
  Choose if to test the residual with the `fit.test_residual function` (`True`) or not (`False`)
- log_file: *str*
  Filename of the log file to be saved.

3.5.34 fit.write_par(V, C, limits, filename='i_guess.inp')

Write the parameters of the fit, whether they are input or output.

Parameters:

- V : *2darray*
  matrix (# signals, parameters)
- C : *1darray or False*
  Coefficients of the polynomion to be used as baseline correction. If it is `False`, they will not appear in the written file.
- limits : *tuple*
  Trim limits for the spectrum `(left, right)`.
- filename: *str or TextIO Object*
  directory and name of the file to be written, or directly a file opened with `open` and writing rights.

3.6 SPECTRA package

All the classes in the `Spectra` module are automatically imported together with `klassez` itself.

Refer to the examples reported in the *User guide* section to understand how to use them, or use the functions `help()`, `vars()`, `dir()` to get detailed info on how they exactly work.

3.6.1 Spectra.Pseudo_2D `class`

Subclass of Spectrum_2D to simulate and handle pseudo-2D experiments. Basically, they share more or less the same attributes, but some methods were adapted in order to suit well with a not-Fourier-transformed indirect dimension.

Attributes:

- datadir: *str*
  Path to the input file/dataset directory
- filename: *str*
  Base of the name of the file, without extensions
- fid: *2darray*
  FID. For simulated data, this must be explicitely set!
- acqus: *dict*
  Dictionary of acqusition parameters
- ngdic: *dict*
  Created only if it is an experimental spectrum. Generated by nmrglue.bruker.read, contains all the information on the spectrometer and on the spectrum.
- procs: *dict*
  Dictionary of processing parameters
- S: *2darray*
  Complex spectrum
- rr: *2darray*
  Real part F2, real part F1
- ii: *2darray*
  Imaginary part F2, imaginary part F1
- freq_f1: *1darray*
  Indeces of the experiments, works as placeholder
- freq_f2: *1darray*
  Frequency scale of the direct dimension, in Hz
- ppm_f1: *1darray*
  Indeces of the experiments, works as placeholder
- ppm_f2: *1darray*
  ppm scale of the direct dimension
- trf1: *dict*
  Projections of the indirect dimension, as 1darrays. Keys: 'ppm_f2' where they were taken

- trf2: *dict*
  Projections of the direct dimension, as 1darrays. Keys: 'ppm_f1' where they were taken
- Trf1: *dict*
  Projections of the indirect dimension, as pSpectrum_1D objects. Keys: 'ppm_f2' where they were taken
- Trf2: *dict*
  Projections of the direct dimension, as pSpectrum_1D objects. Keys: 'ppm_f1' where they were taken
- integrals: *dict*
  Dictionary where to save the regions and values of the integrals.

Methods:

__init__(self, in_file, fid=None, pv=False, isexp=True)

Initialize the class.

Parameters:

- in_file: *str*
  path to file to read, or to the folder of the spectrum
- fid: *2darray or None*
  Array that replaces self.fid.
- pv: *bool*
  True if you want to use pseudo-voigt lineshapes for simulation, False for Voigt
- isexp: *bool*
  True if this is an experimental dataset, False if it is simulated

---

adjph(self, expno=0, p0=None, p1=None, pv=None, update=True)

Adjusts the phases of the spectrum according to the given parameters, or interactively if they are left as default.

Parameters:

- expno: *int*
  Index of the experiment (python numbering) to use in the interactive panel
- p0: *float or None*
  0-th order phase correction /°
- p1: *float or None*
  1-st order phase correction /°
- pv: *float or None*
  1-st order pivot /ppm
- update: *bool*
  Choose if to upload the procs dictionary or not

---

align(self, lims=None, u_off=0.5, ref_idx=0)

Aligns the spectrum to a reference signal in the reference spectrum (default: first one).

Parameters:

- lims: *tuple or None*
  Reference signal region, in ppm. If None, you can select it interactively.
- u_off: *float*
  Maximum displacement allowed, in ppm
- ref_idx: *int*
  Index of the spectrum to be used as a reference (python numbering)

---

basl(self, from_procs=False, phase=True)

Apply baseline correction to the whole pseudo-2D by subtracting self.baseline from self.S. Then, self.S is unpacked in self.rr and self.ii.

Parameters:

- from_procs: *bool*
  If True, computes the baseline using the polynomion model reading self.procs['basl_c'] as coefficients
- phase: *bool*
  Choose if to apply the same phase correction of the spectrum to the baseline. This should be done if the baseline was computed before the phase adjustment!

---

cal(self, offset=[None, None], isHz=False, update=True)

Calibration of the ppm and frequency scales according to a given value, or interactively. In this latter case, a reference peak must be chosen. Calls processing.calibration

Parameters:

- offset: *tuple*
  (scale shift F1, scale shift F2)
- isHz: *tuple of bool*
  True if offset is in frequency units, False if offset is in ppm
- update: *bool*
  Choose if to update the procs dictionary or not

---

calf1(self, value=None, isHz=False)

Calibrates the ppm and frequency scale of the indirect dimension according to a given value, or interactively. Calls self.cal on F1 only.

Parameters:

- value: *float or None*
  scale shift value
- isHz: *bool*
  True if offset is in frequency units, False if offset is in ppm

---

calf2(self, value=None, isHz=False)

Calibrates the ppm and frequency scale of the direct dimension according to a given value, or interactively. Calls self.cal on F2 only

Parameters:

- value: *float or None*
  scale shift value
- isHz: *bool*
  True if offset is in frequency units, False if offset is in ppm

---

convdta(self, scaling=1)

Calls processing.convdta

---

eae(self)

Calls processing.EAE to shuffle the data and make a States-like FID. Sets self.eaeflag to 0.

---

integrate(self, which=0, lims=None)

Integrate the spectrum with a dedicated GUI. Calls processing.integral on each experiment, then saves the results in self.integrals. Therefore, the entries of self.integrals are sequences! If lims is not given, calls fit.integrate on the trace to select the regions to integrate.

Parameters:

- which: *int*
  Experiment index to show in interactive panel
- lims: *tuple*
  Region of the spectrum to integrate (ppm1, ppm2)

---

inv_process(self)

Performs the inverse processing of the spectrum according to the given parameters. Overwrites the S attribute!! Calls inv_xfb.

---

mc(self)

Computes the magnitude of the spectrum on self.S. Then, updates rr, ri, ir, ii.

---

plot(self, Neg=True, lvl0=0.2, Y_label='')

Plots the real part of the spectrum as a 2D contour plot.

Parameters:

- Neg: *bool*
  Plot (True) or not (False) the negative contours.
- lvl0: *float*
  Starting contour value.
- Y_label: *str*
  Custom label for vertical axis.

---

plot_md(self, which=None, lims=None)

Plot a number of experiments, superimposed.

Parameters:

- which: *str or None*
  List of experiment indexes, so that eval(which) is meaningful. None plots all of them
- lims: *tuple*
  Region of the spectrum to show (ppm1, ppm2)

---

plot_stacked(self, which=None, lims=None)

Plot a number of experiments, stacked.

Parameters:

- which: *str or None*
  List of experiment indexes, so that eval(which) is meaningful. None plots all of them.
- lims: *tuple*
  Region of the spectrum to show (ppm1, ppm2)

---

process(self)

Process only the direct dimension. Calls processing.fp on each transient. The parameters are read from the procs dictionary

---

projf1(self, a, b=None)

Calculates the sum trace of the indirect dimension, from a to b in F2. Store the trace in the dictionary trf1 and as 1D spectrum in Trf1. The key is 'a' or 'a:b' Updates the Trf1[label].freq and Trf1[label].ppm with self.freq_f1 and self.ppm_f1 respectively.

Parameters:

- a: *float*
  ppm F2 value where to extract the trace.
- b: *float or None.*
  If it is None, extract the trace in a. Else, sum from a to b in F2.

---

projf2(self, a, b=None)

Calculates the sum trace of the direct dimension, from a to b in F1. Store the trace in the dictionary trf2 and as 1D spectrum in Trf2. The key is 'a' or 'a:b'

Parameters:

- a: *float*
  ppm F1 value where to extract the trace.
- b: *float or None.*
  If it is None, extract the trace in a. Else, sum from a to b in F1.

---

qfil(self, which=None, u=None, s=None)

Gaussian filter to suppress signals. Tries to read self.procs['qfil'], which is { 'u': u, 's': s } Otherwise, these are set interactively by processing.interactive_qfil and then added to self.procs. Calls processing.qfil

Parameters:

- which: *int or None*
  Index of the F2 trace to be used for interactive_qfil. If None, a suitable trace can be selected using misc.select_traces.
- u: *float*
  Position /ppm
- s: *float*
  Width (standard deviation) /ppm

---

read_procs(self, other_dir=None)

Reads the procs dictionary from a file named 'filename.procs' in the same directory of the input file.

Parameters:

- other_dir: *str or None*
  Different location for the procs dictionary to look into. If None, self.datadir is used instead. W! Do not put the trailing slash!

Returns:

- procs: *dict*
  Dictionary of processing parameters

---

rpbc(self, ref_exp=0, **rpbc_kws)

Computes the phase angles and the baseline using processing.rpbc on a reference spectrum taken from self.S. Then applies the phase correction and subtracts the baseline, automatically, to all experiments of the pseudo-2D. The procs dictionary is then updated and saved. The polynomial baseline is computed according to the given coefficients and stored in self.baseline

Parameters:

- ref_exp: *int*
  Index of the reference experiment on which to apply the algorithm
- rpbc_kws: *keyworded arguments*
  See processing.RPBC for details.

---

write_acqus(self, other_dir=None)

Write the acqus dictionary in a file named 'filename.acqus'. Calls misc.write_acqus_1D

Parameters:

- other_dir: *str or None*
  Different location for the acqus dictionary to write into. If None, self.datadir is used instead.

---

write_integrals(self, filename='integrals.dat')

Write the integrals in a file named filename.

Parameters:

- filename: *str*
  name of the file where to write the integrals.

---

write_procs(self, other_dir=None)

Writes the actual procs dictionary in a file named 'filename.procs' in the same directory of the input file.

Parameters:

- other_dir: *str or None*
  Different location for the procs dictionary to write into. If None, self.datadir is used instead. W! Do not put the trailing slash!

---

write_ser(ser, acqus, path=None)

Writes a real/complex array in binary format. Calls misc.write_ser. Be sure that acqus contains the BYTORDA and DTYPA keys. See misc.write_ser to understand the meaning of these values.

Parameters:

- ser: *ndarray*
  Array that you want to convert in binary format.
- acqus: *dict*
  Dictionary of acquisition parameters. It must contain BYTORDA and DTYPA.
- path: *str*
  Path where to save the binary file.

---

xf1(self)

Process only the indirect dimension. Transposes the spectrum in hypermode or normally if FnMODE != QF, then calls for processing.fp using self.procs[keys][0], finally transposes it back. The result is stored in self.S, then self.rr and self.ii are written. freq_f1 and ppm_f1 are assigned with the indexes of the transients.

---

xf2(self)

Process only the direct dimension. Calls processing.fp using procs[keys][1] The result is stored in self.S, then self.rr and self.ii are written. freq_f1 and ppm_f1 are assigned with the indexes of the transients.

3.6.2 Spectra.Spectrum_1D `class`

Class: 1D NMR spectrum

Attributes:

- datadir: *str*
  Path to the input file/dataset directory
- filename: *str*
  Base of the name of the file, without extensions
- fid: *1darray*
  FID
- acqus: *dict*
  Dictionary of acqusition parameters
- ngdic: *dict*
  Created only if it is an experimental spectrum. Generated by nmrglue.bruker.read, contains all the information on the spectrometer and on the spectrum.
- procs: *dict*
  Dictionary of processing parameters
- S: *1darray*
  Complex spectrum
- r: *1darray*
  Real part of the spectrum
- i: *1darray*
  Imaginary part of the spectrum
- freq: *1darray*
  Frequency scale of the spectrum, in Hz
- ppm: *1darray*
  ppm scale of the spectrum
- F: *fit.Voigt_Fit object*
  Used for deconvolution. See fit.Voigt_fit.
- baseline: *1darray*
  Baseline of the spectrum.
- integrals: *dict*
  Dictionary where to save the regions and values of the integrals.

Methods:

__init__(self, in_file, pv=False, isexp=True, spect='bruker')

Initialize the class. Simulation of the dataset (i.e. isexp=False) employs sim.sim_1D.

Parameters:

- in_file: *str*
  path to file to read, or to the folder of the spectrum
- pv: *bool*
  True if you want to use pseudo-voigt lineshapes for simulation, False for Voigt
- isexp: *bool*
  True if this is an experimental dataset, False if it is simulated
- spect: *str*
  Data file format. Allowed: 'bruker', 'varian', 'magritek'

---

acme(self, **method_kws)

Automatic phase correction based on entropy minimization It calculates the phase angles using the algorithm specified in method, then calls self.adjph with those values.

Parameters:

- method_kws: *keyworded arguments*
  Additional parameters for the chosen method.

---

adjph(self, p0=None, p1=None, pv=None, update=True)

Adjusts the phases of the spectrum according to the given parameters, or interactively if they are left as default. Calls for processing.ps

Parameters:

- p0: *float or None*
  0-th order phase correction /°
- p1: *float or None*
  1-st order phase correction /°
- pv: *float or None*
  1-st order pivot /ppm
- update: *bool*
  Choose if you want to update the procs dictionary or not

---

baseline_correction(self, basl_file='spectrum.basl', winlim=None)

Correct the baseline of the spectrum, according to a pre-existing file or interactively. Calls processing.baseline_correction or processing.load_baseline

Parameters:

- basl_file: *str*
  Path to the baseline file. If it already exists, the baseline will be built according to this file; otherwise this will be the destination file of the baseline.

- winlim: *tuple or None*
  Limits of the baseline. If it is None, it will be interactively set. If basl_file exists, it will be read from there. Else, (ppm1, ppm2).

---

basl(self, from_procs=False, phase=True)

Apply the baseline correction by subtracting self.baseline from self.S. Then, self.S is unpacked in self.r and self.i

Parameters:

- from_procs: *bool*
  If True, computes the baseline using the polynomion model reading self.procs['basl_c'] as coefficients

- phase: *bool*
  Choose if to apply the same phase correction of the spectrum to the baseline. This should be done if the baseline was computed before the phase adjustment!

---

cal(self, offset=None, isHz=False, update=True)

Calibrates the ppm and frequency scale according to a given value, or interactively. Calls processing.calibration

Parameters:

- offset: *float or None*
  scale shift value

- isHz: *bool*
  True if offset is in frequency units, False if offset is in ppm

- update: *bool*
  Choose if to update the procs dictionary or not

---

convdta(self, scaling=1)

Call processing.convdta using self.acqus['GRPDLY']

---

integrate(self, lims=None)

Integrate the spectrum with a dedicated GUI. Calls fit.integrate and writes in self.integrals with keys [ppm1:ppm2]

Parameters:

- lims: *tuple*
  Integrates from lims[0] to lims[1]. If it is None, calls for interactive integration.

---

inv_process(self)

Performs the inverse processing of the spectrum according to the given parameters. Overwrites the S attribute!! Calls processing.inv_fp

---

mc(self)

Calculates the magnitude of the spectrum and overwrites self.S, self.r, self.i

---

plot(self, name=None, ext='png', dpi=600)

Plots the real part of the spectrum.

Parameters:

- name: *str*
  Filename for the figure. If None, it is shown instead.
- ext: *str*
  Format of the image
- dpi: *int*
  Resolution of the image in dots per inches

---

process(self, interactive=False)

Performs the processing of the FID. The parameters are read from self.procs. Calls processing.interactive_fp or processing.fp using self.acqus and self.procs Writes the result is self.S, then unpacks it in self.r and self.i Calculates frequency and ppm scales. Also initializes self.F with fit.Voigt_Fit class using the current parameters

Parameters:

- interactive: *bool*
  True if you want to open the interactive panel, False to read the parameters from self.procs.

---

qfil(self, u=None, s=None)

Gaussian filter to suppress signals. Tries to read self.procs['qfil'], which is { 'u': u, 's': s } Otherwise, these are set interactively by processing.interactive_qfil and then added to self.procs. Calls processing.qfil

Parameters:

- u: *float*
  Position of the filter /ppm
- s: *float*
  Width (standard deviation) of the filter /ppm

---

read_procs(self, other_dir=None)

Reads the procs dictionary from a file named 'filename.procs' in the same directory of the input file.

Parameters:

- other_dir: *str or None*
  Different location for the procs dictionary to look into. If None, self.datadir is used instead. W! Do not put the trailing slash!

Returns:

- procs: *dict*
  Dictionary of processing parameters

---

rpbc(self, **rpbc_kws)

Computes the phase angles and the baseline using processing.RPBC on self.S. Then applies the phase correction and subtracts the baseline, automatically. The procs dictionary is then updated and saved. The polynomial baseline is computed according to the given coefficients and stored in self.baseline

Parameters:

- rpbc_kws: *keyworded arguments*
  See processing.RPBC for details.

---

write_acqus(self, other_dir=None)

Write the acqus dictionary in a file named 'filename.acqus'. Calls misc.write_acqus_1D

Parameters:

- other_dir: *str or None*
  Different location for the acqus dictionary to write into. If None, self.datadir is used instead.

---

write_integrals(self, other_dir=None)

Write the integrals in a file named '{self.filename}.int'.

Parameters:

- other_dir: *str or None*
  Different location for the integrals file to write into. If None, self.datadir is used instead.

---

write_procs(self, other_dir=None)

Writes the actual procs dictionary in a file named 'filename.procs' in the same directory of the input file.

Parameters:

- other_dir: *str or None*
  Different location for the procs dictionary to write into. If None, self.datadir is used instead. W! Do not put the trailing slash!

---

write_ser(ser, acqus, path=None)

Writes a real/complex array in binary format. Calls misc.write_ser. Be sure that acqus contains the BYTORDA and DTYPA keys. See misc.write_ser to understand the meaning of these values.

Parameters:

- ser: *ndarray*
  Array that you want to convert in binary format.
- acqus: *dict*
  Dictionary of acquisition parameters. It must contain BYTORDA and DTYPA.
- path: *str*
  Path where to save the binary file.

3.6.3 Spectra.Spectrum_2D `class`

Class: 2D NMR spectrum

Attributes:

- datadir: *str*
  Path to the input file/dataset directory
- filename: *str*
  Base of the name of the file, without extensions
- fid: *2darray*
  FID
- acqus: *dict*
  Dictionary of acqusition parameters
- ngdic: *dict*
  Created only if it is an experimental spectrum. Generated by nmrglue.bruker.read, contains all the information on the spectrometer and on the spectrum.
- procs: *dict*
  Dictionary of processing parameters
- eaeflag: *int*
  If FnMODE is Echo-Antiecho, keeps track of the manipulation of the data so to not repeat the same process twice
- S: *2darray*
  Complex (or hypercomplex, depending on FnMODE) spectrum
- rr: *2darray*
  Real part F2, real part F1
- ii: *2darray*
  Imaginary part F2, imaginary part F1
- ir: *2darray*
  Real part F2, imaginary part F1. Only exist if F1 is acquired in phase-sensitive mode
- ri: *2darray*
  Imaginary part F2, real part F1. Only exist if F1 is acquired in phase-sensitive mode
- freq_f1: *1darray*
  Frequency scale of the indirect dimension, in Hz
- freq_f2: *1darray*
  Frequency scale of the direct dimension, in Hz
- ppm_f1: *1darray*
  ppm scale of the indirect dimension
- ppm_f2: *1darray*
  ppm scale of the direct dimension
- trf1: *dict*
  Projections of the indirect dimension, as 1darrays. Keys: 'ppm_f2' where they were taken

- trf2: *dict*
  Projections of the direct dimension, as 1darrays. Keys: 'ppm_f1' where they were taken

- Trf1: *dict*
  Projections of the indirect dimension, as pSpectrum_1D objects. Keys: 'ppm_f2' where they were taken

- Trf2: *dict*
  Projections of the direct dimension, as pSpectrum_1D objects. Keys: 'ppm_f1' where they were taken

- integrals: *dict*
  Dictionary where to save the regions and values of the integrals.

Methods:

__init__(self, in_file, pv=False, isexp=True, is_pseudo=False)

Initialize the class.

Parameters:

- in_file: *str*
  path to file to read, or to the folder of the spectrum

- pv: *bool*
  True if you want to use pseudo-voigt lineshapes for simulation, False for Voigt

- isexp: *bool*
  True if this is an experimental dataset, False if it is simulated

- is_pseudo: *bool*
  True if it is a pseudo-2D. Legacy option

---

adjph(self, p01=None, p11=None, pv1=None, p02=None, p12=None, pv2=None, update=True)

Adjusts the phases of the spectrum according to the given parameters, or interactively if they are left as default. The non-interactive workflow is to apply processing.ps on F2, transpose according to FnMODE, apply processing.ps on F1, transpose back. If FnMODE is 'No', the phase correction is applied only on F2, as it should be done in a pseudo-2D experiment. Once self.S was updated and unpacked, the phase values are added to the procs dictionary to keep track of multiple phase adjustments.

Parameters:

- p01: *float or None*
  0-th order phase correction /° of the indirect dimension

- p11: *float or None*
  1-st order phase correction /° of the indirect dimension

- pv1: *float or None*
  1-st order pivot /ppm of the indirect dimension

- p02: *float or None*
  0-th order phase correction /° of the direct dimension
- p12: *float or None*
  1-st order phase correction /° of the direct dimension
- pv2: *float or None*
  1-st order pivot /ppm of the direct dimension
- update: *bool*
  Choose if to update the procs dictionary or not

---

cal(self, offset=[None, None], isHz=False, update=True)

Calibration of the ppm and frequency scales according to a given value, or interactively. In this latter case, a reference peak must be chosen. Calls processing.calibration

Parameters:

- offset: *tuple*
  (scale shift F1, scale shift F2)
- isHz: *tuple of bool*
  True if offset is in frequency units, False if offset is in ppm
- update: *bool*
  Choose if to update the procs dictionary or not

---

calf1(self, value=None, isHz=False)

Calibrates the ppm and frequency scale of the indirect dimension according to a given value, or interactively. Calls self.cal on F1 only.

Parameters:

- value: *float or None*
  scale shift value
- isHz: *bool*
  True if offset is in frequency units, False if offset is in ppm

---

calf2(self, value=None, isHz=False)

Calibrates the ppm and frequency scale of the direct dimension according to a given value, or interactively. Calls self.cal on F2 only

Parameters:

- value: *float or None*
  scale shift value
- isHz: *bool*
  True if offset is in frequency units, False if offset is in ppm

---

convdta(self, scaling=1)

Calls processing.convdta to compensate for the group delay. It does not always work, depends on TopSpin version and planets alignment.

Parameters:

- scaling: *float*
  Scaling factor for processingconvdta.

---

eae(self)

Calls processing.EAE to shuffle the data and make a States-like FID. Sets self.eaeflag to 0.

---

integrate(self, **kwargs)

Integrates the spectrum with a dedicated GUI. Calls fit.integrate_2D

Parameters:

- kwargs: *keyworded arguments*
  Additional parameters for fit.integrate_2D

---

inv_process(self)

Performs the inverse processing of the spectrum according to the given parameters. Overwrites the S attribute!! Calls inv_xfb.

---

mc(self)

Computes the magnitude of the spectrum on self.S. Then, updates rr, ri, ir, ii.

---

plot(self, Neg=True, lvl0=0.2)

Plots the real part of the spectrum. Use the mouse scroll to adjust the contour starting level.

Parameters:

- Neg: *bool*
  Plot (True) or not (False) the negative contours.
- lvl0: *float*
  Starting contour value with respect to the maximum of the spectrum

---

process(self, interactive=False, **int_kwargs)

Performs the full processing of the FID on both dimensions. The parameters are read from self.procs. If FnMODE is Echo-Antiecho and you did not call self.eae before, the FID is converted to States with processing.EAE before to start. If interactive is True, calls processing.interactive_xfb with int_kwargs, else calls processing.xfb. The complex/hypercomplex spectrum is stored in self.S, then unpacked into self.rr, self.ri, self.ir, self.ii. If FnMODE is Echo-Antiecho, a phase correction of -90 degrees is applied on the indirect dimension.

Parameters:

- interactive: *bool*
  True if you want to open the interactive panel, False to read the parameters from self.procs.
- int_kwargs: *keyworded arguments*
  Additional parameters for processing.interactive_xfb, if interactive=True.

---

projf1(self, a, b=None)

Calculates the sum trace of the indirect dimension, from a ppm to b ppm in F2. Store the trace in the dictionary trf1 and as 1D spectrum in Trf1. The key is 'a' or 'a:b' Calls misc.get_trace on self.rr with column=True

Parameters:

- a: *float*
  ppm F2 value where to extract the trace.
- b: *float or None.*
  If it is None, extract the trace in a. Else, sum from a to b in F2.

---

projf2(self, a, b=None)

Calculates the sum trace of the direct dimension, from a ppm to b ppm in F1. Store the trace in the dictionary trf2 and as 1D spectrum in Trf2. The key is 'a' or 'a:b' Calls misc.get_trace on self.rr with column=False

Parameters:

- a: *float*
  ppm F1 value where to extract the trace.
- b: *float or None.*
  If it is None, extract the trace in a. Else, sum from a to b in F1.

---

qfil(self, which=None, u=None, s=None)

Gaussian filter to suppress signals. Tries to read self.procs['qfil'], which is { 'u': u, 's': s } Otherwise, these are set interactively by processing.interactive_qfil and then added to self.procs. Calls processing.qfil

Parameters:

- which: *int or None*
  Index of the F2 trace to be used for interactive_qfil. If None, a suitable trace can be selected using misc.select_traces.
- u: *float*
  Position /ppm
- s: *float*
  Width (standard deviation) /ppm

---

read_procs(self, other_dir=None)

Reads the procs dictionary from a file named 'filename.procs' in the same directory of the input file.

Parameters:

- other_dir: *str or None*
  Different location for the procs dictionary to look into. If None, self.datadir is used instead. W! Do not put the trailing slash!

Returns:

- procs: *dict*
  Dictionary of processing parameters

---

write_acqus(self, other_dir=None)

Write the acqus dictionary in a file named 'filename.acqus'. Calls misc.write_acqus_1D

Parameters:

- other_dir: *str or None*
  Different location for the acqus dictionary to write into. If None, self.datadir is used instead.

---

write_integrals(self, other_dir=None)

Write the integrals in a file named '{self.filename}.int'.

Parameters:

- other_dir: *str or None*
  Different location for the integrals file to write into. If None, self.datadir is used instead.

---

write_procs(self, other_dir=None)

Writes the actual procs dictionary in a file named 'filename.procs' in the same directory of the input file.

Parameters:

- other_dir: *str or None*
  Different location for the procs dictionary to write into. If None, self.datadir is used instead. W! Do not put the trailing slash!

---

write_ser(ser, acqus, path=None)

Writes a real/complex array in binary format. Calls misc.write_ser. Be sure that acqus contains the BYTORDA and DTYPA keys. See misc.write_ser to understand the meaning of these values.

Parameters:

- ser: *ndarray*
  Array that you want to convert in binary format.
- acqus: *dict*
  Dictionary of acquisition parameters. It must contain BYTORDA and DTYPA.
- path: *str*
  Path where to save the binary file.

---

xf1(self)

Process only the indirect dimension. Transposes the spectrum in hypermode or normally if FnMODE != QF, then calls for processing.fp using self.procs[keys][0], finally transposes it back. The result is stored in self.S, then self.rr and self.ii are written. freq_f1 and ppm_f1 are assigned with the indexes of the transients.

---

xf2(self)

Process only the direct dimension. Calls processing.fp using procs[keys][1] The result is stored in self.S, then self.rr and self.ii are written. freq_f1 and ppm_f1 are assigned with the indexes of the transients.

3.6.4 Spectra.pSpectrum_1D `class`

Subclass of Spectrum_1D that allows to handle processed 1D NMR spectra. Useful when dealing with traces of 2D spectra. Shares the same attributes with Spectrum_1D.

Attributes:

- datadir: *str*
  Path to the input file/dataset directory
- filename: *str*
  Base of the name of the file, without extensions
- acqus: *dict*
  Dictionary of acqusition parameters
- ngdic: *dict*
  Created only if it is an experimental spectrum. Generated by nmrglue.bruker.read, contains all the information on the spectrometer and on the spectrum.
- procs: *dict*
  Dictionary of processing parameters
- S: *1darray*
  Complex spectrum
- r: *1darray*
  Real part of the spectrum
- i: *1darray*
  Imaginary part of the spectrum
- freq: *1darray*
  Frequency scale of the spectrum, in Hz
- ppm: *1darray*
  ppm scale of the spectrum
- F: *fit.Voigt_Fit object*
  Used for deconvolution. See fit.Voigt_fit.
- baseline: *1darray*
  Baseline of the spectrum.
- integrals: *dict*
  Dictionary where to save the regions and values of the integrals.

Methods:

__init__(self, in_file, acqus=None, procs=None, istrace=False, filename='T')

Initialize the class.

Parameters:

- in_file: *str or 1darray*
  If istrace is True, in_file is the NMR spectrum. Else, it is the directory of the processed data.
- acqus: *dict or None*
  If istrace is True, you must supply the associated 'acqus' dictionary. Else, it is not necessary as it is read from the input directory
- procs: *dict or None*
  You can pass the dictionary of processing parameters, if you want. Otherwise, it is initialized with standard values.
- istrace: *bool*
  Declare the object as trace extracted from a 2D (True) or as true experimental spectrum (False)
- filename: *str*
  If istrace is True, this will be the filename of self.acqus and self.procs

---

acme(self, **method_kws)

Automatic phase correction based on entropy minimization It calculates the phase angles using the algorithm specified in method, then calls self.adjph with those values.

Parameters:

- method_kws: *keyworded arguments*
  Additional parameters for the chosen method.

---

adjph(self, p0=None, p1=None, pv=None, update=True)

Adjusts the phases of the spectrum according to the given parameters, or interactively if they are left as default. Calls for processing.ps

Parameters:

- p0: *float or None*
  0-th order phase correction /°
- p1: *float or None*
  1-st order phase correction /°
- pv: *float or None*
  1-st order pivot /ppm
- update: *bool*
  Choose if you want to update the procs dictionary or not

---

baseline_correction(self, basl_file='spectrum.basl', winlim=None)

Correct the baseline of the spectrum, according to a pre-existing file or interactively. Calls processing.baseline_correction or processing.load_baseline

Parameters:

- basl_file: *str*
  Path to the baseline file. If it already exists, the baseline will be built according to this file; otherwise this will be the destination file of the baseline.

- winlim: *tuple or None*
  Limits of the baseline. If it is None, it will be interactively set. If basl_file exists, it will be read from there. Else, (ppm1, ppm2).

---

basl(self, from_procs=False, phase=True)

Apply the baseline correction by subtracting self.baseline from self.S. Then, self.S is unpacked in self.r and self.i

Parameters:

- from_procs: *bool*
  If True, computes the baseline using the polynomion model reading self.procs['basl_c'] as coefficients

- phase: *bool*
  Choose if to apply the same phase correction of the spectrum to the baseline. This should be done if the baseline was computed before the phase adjustment!

---

cal(self, offset=None, isHz=False, update=True)

Calibrates the ppm and frequency scale according to a given value, or interactively. Calls processing.calibration

Parameters:

- offset: *float or None*
  scale shift value

- isHz: *bool*
  True if offset is in frequency units, False if offset is in ppm

- update: *bool*
  Choose if to update the procs dictionary or not

---

convdta(self, scaling=1)

Call processing.convdta using self.acqus['GRPDLY']

---

integrate(self, lims=None)

Integrate the spectrum with a dedicated GUI. Calls fit.integrate and writes in self.integrals with keys [ppm1:ppm2]

Parameters:

- lims: *tuple*
  Integrates from lims[0] to lims[1]. If it is None, calls for interactive integration.

---

inv_process(self)

Performs the inverse processing of the spectrum according to the given parameters. Overwrites the S attribute!! Calls processing.inv_fp

---

mc(self)

Calculates the magnitude of the spectrum and overwrites self.S, self.r, self.i

---

plot(self, name=None, ext='png', dpi=600)

Plots the real part of the spectrum.

Parameters:

- name: *str*
  Filename for the figure. If None, it is shown instead.
- ext: *str*
  Format of the image
- dpi: *int*
  Resolution of the image in dots per inches

---

process(self, interactive=False)

Performs the processing of the FID. The parameters are read from self.procs. Calls processing.interactive_fp or processing.fp using self.acqus and self.procs Writes the result is self.S, then unpacks it in self.r and self.i Calculates frequency and ppm scales. Also initializes self.F with fit.Voigt_Fit class using the current parameters

Parameters:

- interactive: *bool*
  True if you want to open the interactive panel, False to read the parameters from self.procs.

---

qfil(self, u=None, s=None)

Gaussian filter to suppress signals. Tries to read self.procs['qfil'], which is { 'u': u, 's': s } Otherwise, these are set interactively by processing.interactive_qfil and then added to self.procs. Calls processing.qfil

Parameters:

- u: *float*
  Position of the filter /ppm
- s: *float*
  Width (standard deviation) of the filter /ppm

---

read_procs(self, other_dir=None)

Reads the procs dictionary from a file named 'filename.procs' in the same directory of the input file.

Parameters:

- other_dir: *str or None*
  Different location for the procs dictionary to look into. If None, self.datadir is used instead. W! Do not put the trailing slash!

Returns:

- procs: *dict*
  Dictionary of processing parameters

---

rpbc(self, **rpbc_kws)

Computes the phase angles and the baseline using processing.RPBC on self.S. Then applies the phase correction and subtracts the baseline, automatically. The procs dictionary is then updated and saved. The polynomial baseline is computed according to the given coefficients and stored in self.baseline

Parameters:

- rpbc_kws: *keyworded arguments*
  See processing.RPBC for details.

---

write_acqus(self, other_dir=None)

Write the acqus dictionary in a file named 'filename.acqus'. Calls misc.write_acqus_1D

Parameters:

- other_dir: *str or None*
  Different location for the acqus dictionary to write into. If None, self.datadir is used instead.

---

write_integrals(self, other_dir=None)

Write the integrals in a file named '{self.filename}.int'.

Parameters:

- other_dir: *str or None*
  Different location for the integrals file to write into. If None, self.datadir is used instead.

---

write_procs(self, other_dir=None)

Writes the actual procs dictionary in a file named 'filename.procs' in the same directory of the input file.

Parameters:

- other_dir: *str or None*
  Different location for the procs dictionary to write into. If None, self.datadir is used instead. W! Do not put the trailing slash!

---

write_ser(ser, acqus, path=None)

Writes a real/complex array in binary format. Calls misc.write_ser. Be sure that acqus contains the BYTORDA and DTYPA keys. See misc.write_ser to understand the meaning of these values.

Parameters:

- ser: *ndarray*
  Array that you want to convert in binary format.
- acqus: *dict*
  Dictionary of acquisition parameters. It must contain BYTORDA and DTYPA.
- path: *str*
  Path where to save the binary file.

3.6.5 Spectra.pSpectrum_2D `class`

Subclass of Spectrum_2D that allows to handle processed 2D NMR spectra. Reads the processed spectrum from Bruker.

Attributes:

- datadir: *str*
  Path to the input file/dataset directory
- filename: *str*
  Base of the name of the file, without extensions
- acqus: *dict*
  Dictionary of acqusition parameters
- ngdic: *dict*
  Generated by nmrglue.bruker.read, contains all the information on the spectrometer and on the spectrum.
- procs: *dict*
  Dictionary of processing parameters
- S: *2darray*
  Complex (or hypercomplex, depending on FnMODE) spectrum
- rr: *2darray*
  Real part F2, real part F1
- ii: *2darray*
  Imaginary part F2, imaginary part F1
- ir: *2darray*
  Real part F2, imaginary part F1. Only exist if F1 is acquired in phase-sensitive mode
- ri: *2darray*
  Imaginary part F2, real part F1. Only exist if F1 is acquired in phase-sensitive mode
- freq_f1: *1darray*
  Frequency scale of the indirect dimension, in Hz
- freq_f2: *1darray*
  Frequency scale of the direct dimension, in Hz
- ppm_f1: *1darray*
  ppm scale of the indirect dimension
- ppm_f2: *1darray*
  ppm scale of the direct dimension
- trf1: *dict*
  Projections of the indirect dimension, as 1darrays. Keys: 'ppm_f2' where they were taken
- trf2: *dict*
  Projections of the direct dimension, as 1darrays. Keys: 'ppm_f1' where they were taken

- Trf1: *dict*
  Projections of the indirect dimension, as pSpectrum_1D objects. Keys: 'ppm_f2' where they were taken

- Trf2: *dict*
  Projections of the direct dimension, as pSpectrum_1D objects. Keys: 'ppm_f1' where they were taken

- integrals: *dict*
  Dictionary where to save the regions and values of the integrals.

Methods:

__init__(self, in_file)

Initialize the class.

Parameters:

- in_file: *str*
  Path to the spectrum. Here, the 'pdata/#' folder must be specified.

---

adjph(self, p01=None, p11=None, pv1=None, p02=None, p12=None, pv2=None, update=True)

Adjusts the phases of the spectrum according to the given parameters, or interactively if they are left as default. The non-interactive workflow is to apply processing.ps on F2, transpose according to FnMODE, apply processing.ps on F1, transpose back. If FnMODE is 'No', the phase correction is applied only on F2, as it should be done in a pseudo-2D experiment. Once self.S was updated and unpacked, the phase values are added to the procs dictionary to keep track of multiple phase adjustments.

Parameters:

- p01: *float or None*
  0-th order phase correction /° of the indirect dimension

- p11: *float or None*
  1-st order phase correction /° of the indirect dimension

- pv1: *float or None*
  1-st order pivot /ppm of the indirect dimension

- p02: *float or None*
  0-th order phase correction /° of the direct dimension

- p12: *float or None*
  1-st order phase correction /° of the direct dimension

- pv2: *float or None*
  1-st order pivot /ppm of the direct dimension

- update: *bool*
  Choose if to update the procs dictionary or not

---

cal(self, offset=[None, None], isHz=False, update=True)

Calibration of the ppm and frequency scales according to a given value, or interactively. In this latter case, a reference peak must be chosen. Calls processing.calibration

Parameters:

- offset: *tuple*
  (scale shift F1, scale shift F2)
- isHz: *tuple of bool*
  True if offset is in frequency units, False if offset is in ppm
- update: *bool*
  Choose if to update the procs dictionary or not

---

calf1(self, value=None, isHz=False)

Calibrates the ppm and frequency scale of the indirect dimension according to a given value, or interactively. Calls self.cal on F1 only.

Parameters:

- value: *float or None*
  scale shift value
- isHz: *bool*
  True if offset is in frequency units, False if offset is in ppm

---

calf2(self, value=None, isHz=False)

Calibrates the ppm and frequency scale of the direct dimension according to a given value, or interactively. Calls self.cal on F2 only

Parameters:

- value: *float or None*
  scale shift value
- isHz: *bool*
  True if offset is in frequency units, False if offset is in ppm

---

convdta(self, scaling=1)

Calls processing.convdta to compensate for the group delay. It does not always work, depends on TopSpin version and planets alignment.

Parameters:

- scaling: *float*
  Scaling factor for processingconvdta.

---

eae(self)

Calls processing.EAE to shuffle the data and make a States-like FID. Sets self.eaeflag to 0.

---

integrate(self, **kwargs)

Integrates the spectrum with a dedicated GUI. Calls fit.integrate_2D

Parameters:

- kwargs: *keyworded arguments*
  Additional parameters for fit.integrate_2D

---

inv_process(self)

Performs the inverse processing of the spectrum according to the given parameters. Overwrites the S attribute!! Calls inv_xfb.

---

mc(self)

Computes the magnitude of the spectrum on self.S. Then, updates rr, ri, ir, ii.

---

plot(self, Neg=True, lvl0=0.2)

Plots the real part of the spectrum. Use the mouse scroll to adjust the contour starting level.

Parameters:

- Neg: *bool*
  Plot (True) or not (False) the negative contours.
- lvl0: *float*
  Starting contour value with respect to the maximum of the spectrum

---

process(self, interactive=False, **int_kwargs)

Performs the full processing of the FID on both dimensions. The parameters are read from self.procs. If FnMODE is Echo-Antiecho and you did not call self.eae before, the FID is converted to States with processing.EAE before to start. If interactive is True, calls processing.interactive_xfb with int_kwargs, else calls processing.xfb. The complex/hypercomplex spectrum is stored in self.S, then unpacked into self.rr, self.ri, self.ir, self.ii. If FnMODE is Echo-Antiecho, a phase correction of -90 degrees is applied on the indirect dimension.

Parameters:

- interactive: *bool*
  True if you want to open the interactive panel, False to read the parameters from self.procs.
- int_kwargs: *keyworded arguments*
  Additional parameters for processing.interactive_xfb, if interactive=True.

---

projf1(self, a, b=None)

Calculates the sum trace of the indirect dimension, from a ppm to b ppm in F2. Store the trace in the dictionary trf1 and as 1D spectrum in Trf1. The key is 'a' or 'a:b' Calls misc.get_trace on self.rr with column=True

Parameters:

- a: *float*
  ppm F2 value where to extract the trace.
- b: *float or None.*
  If it is None, extract the trace in a. Else, sum from a to b in F2.

---

projf2(self, a, b=None)

Calculates the sum trace of the direct dimension, from a ppm to b ppm in F1. Store the trace in the dictionary trf2 and as 1D spectrum in Trf2. The key is 'a' or 'a:b' Calls misc.get_trace on self.rr with column=False

Parameters:

- a: *float*
  ppm F1 value where to extract the trace.
- b: *float or None.*
  If it is None, extract the trace in a. Else, sum from a to b in F1.

---

qfil(self, which=None, u=None, s=None)

Gaussian filter to suppress signals. Tries to read self.procs['qfil'], which is { 'u': u, 's': s } Otherwise, these are set interactively by processing.interactive_qfil and then added to self.procs. Calls processing.qfil

Parameters:

- which: *int or None*
  Index of the F2 trace to be used for interactive_qfil. If None, a suitable trace can be selected using misc.select_traces.
- u: *float*
  Position /ppm
- s: *float*
  Width (standard deviation) /ppm

---

read_procs(self, other_dir=None)

Reads the procs dictionary from a file named 'filename.procs' in the same directory of the input file.

Parameters:

- other_dir: *str or None*
  Different location for the procs dictionary to look into. If None, self.datadir is used instead. W! Do not put the trailing slash!

Returns:

- procs: *dict*
  Dictionary of processing parameters

---

write_acqus(self, other_dir=None)

Write the acqus dictionary in a file named 'filename.acqus'. Calls misc.write_acqus_1D

Parameters:

- other_dir: *str or None*
  Different location for the acqus dictionary to write into. If None, self.datadir is used instead.

---

write_integrals(self, other_dir=None)

Write the integrals in a file named '{self.filename}.int'.

Parameters:

- other_dir: *str or None*
  Different location for the integrals file to write into. If None, self.datadir is used instead.

---

write_procs(self, other_dir=None)

Writes the actual procs dictionary in a file named 'filename.procs' in the same directory of the input file.

Parameters:

- other_dir: *str or None*
  Different location for the procs dictionary to write into. If None, self.datadir is used instead. W! Do not put the trailing slash!

---

write_ser(ser, acqus, path=None)

Writes a real/complex array in binary format. Calls misc.write_ser. Be sure that acqus contains the BYTORDA and DTYPA keys. See misc.write_ser to understand the meaning of these values.

Parameters:

- ser: *ndarray*
  Array that you want to convert in binary format.
- acqus: *dict*
  Dictionary of acquisition parameters. It must contain BYTORDA and DTYPA.
- path: *str*
  Path where to save the binary file.

---

xf1(self)

Process only the indirect dimension. Transposes the spectrum in hypermode or normally if FnMODE != QF, then calls for processing.fp using self.procs[keys][0], finally transposes it back. The result is stored in self.S, then self.rr and self.ii are written. freq_f1 and ppm_f1 are assigned with the indexes of the transients.

---

xf2(self)

Process only the direct dimension. Calls processing.fp using procs[keys][1] The result is stored in self.S, then self.rr and self.ii are written. freq_f1 and ppm_f1 are assigned with the indexes of the transients.
