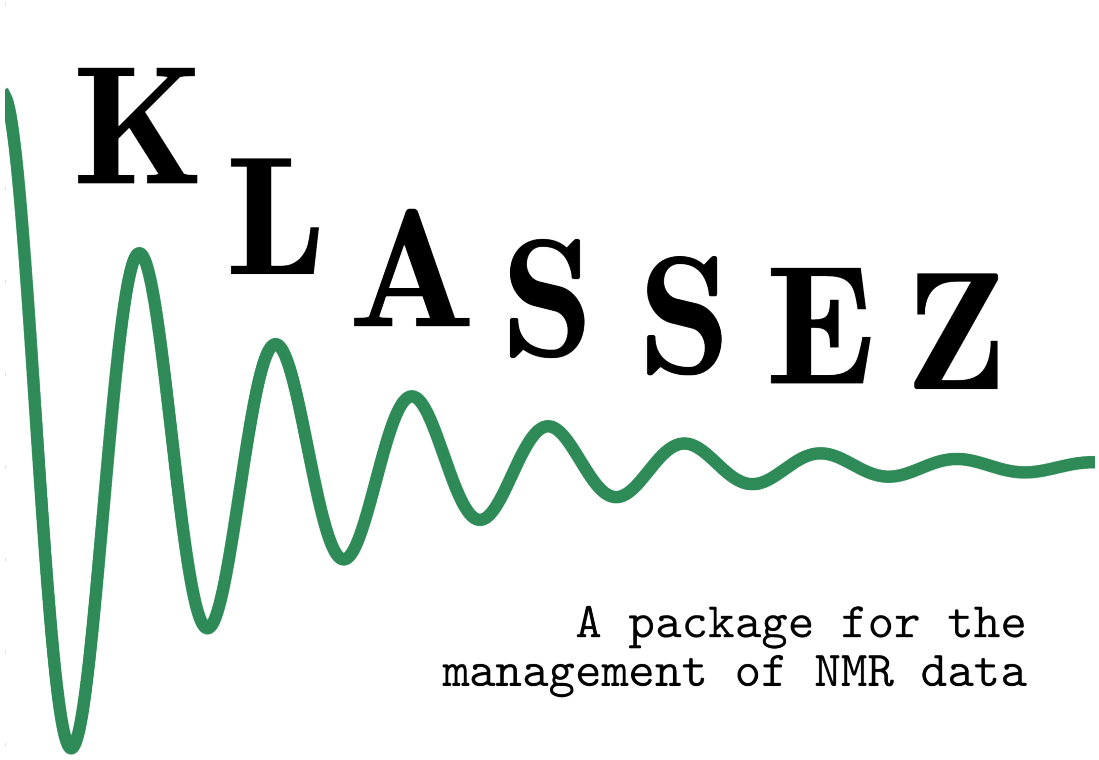

Author:
Francesco Bruno

Major contributors:
Letizia Fiorucci

Version: 0.4a.10

Documentation release date:
October 6, 2025

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

KLASSEZ can be installed from PyPI through:

```
pip install klassez
```

The required dependencies are sorted out automatically.

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

```
from klassez import *
```

This line executes the following code:

```
import os
import sys
import numpy as np
from numpy import linalg
from scipy import stats
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import seaborn as sns
from copy import deepcopy
from pprint import pprint

from . import fit, misc, sim, figures, processing
from .Spectra import Spectrum_1D, pSpectrum_1D, Spectrum_2D, pSpectrum_2D, Pseudo_2D

# Use seaborn's colormaps and save it to a dictionary
from .config import CM, CM_2D, COLORS, cron
```

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

An alternative, safer version to prevent overwriting of custom functions is:

```
import klassez as kz
```

In this case, additional packages for the main script must be declared explicitely.

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())
```

There is a restricted list of colormaps, `CM_2D`, that should be used for visualizing 2D spectra.

`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.

`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()
s.pknl()   # Tries to remove the digital filter through a first-order phase correction
```

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;
- `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.

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.

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

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 – `'no'`: no apodization – `'em'`: exponential – `'sin'`: sine – `'qsin'`: squared sine – `'gm'`: mixed lorentzian-gaussian – `'gmb'`: mixed lorentzian-gaussian, Bruker style • `'lb'`: Exponential line-broadening. Read by `em` and `gm` • `'lb_gm'`: Exponential line-broadening. Read by `gm` • `'gb'`: Gaussian line-broadening. Read by `gmb` • `'gb_gm'`: Gaussian line-broadening. Read by `gm` • `'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

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
`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
`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 `B0`
`SFO2`	Larmor frequency of the observed nucleus in F2 at field `B0`
`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.process()
s.pknl()   # Remove the digital filter
# Also in this case, phase correction and calibration are performed automatically with
    the values in procs
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:

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 – `'no'`: no apodization – `'em'`: exponential – `'sin'`: sine – `'qsin'`: squared sine – `'gm'`: mixed lorentzian-gaussian – `'gmb'`: mixed lorentzian-gaussian, Bruker style • `'lb'`: Exponential line-broadening. Read by `em` and `gmb` • `'lb_gm'`: Exponential line-broadening. Read by `gm` • `'gb'`: Gaussian line-broadening. Read by `gmb` • `'gb_gm'`: Gaussian line-broadening. Read by `gm` • `'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
`p02`	Frequency-independent phase correction /degrees, direct dimension
`p12`	First order phase correction /degrees, direct dimension
`pv2`	Pivot point for the first order phase correction /ppm, direct dimension
`p01`	Frequency-independent phase correction /degrees, indirect dimension
`p11`	First order phase correction /degrees, indirect dimension
`pv1`	Pivot point for the first order phase correction /ppm, indirect dimension
`cal_1`	Calibration offset for F1 /ppm
`cal_2`	Calibration offset for F2 /ppm

	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;
- `beta`: Fraction of gaussianity. $\beta = 0 \implies$ pure Lorentzian peak, $\beta = 1 \implies$ 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
beta   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;
- `beta`: Fraction of gaussianity. $\beta = 0 \implies$ pure Lorentzian peak, $\beta = 1 \implies$ 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
beta    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')
```

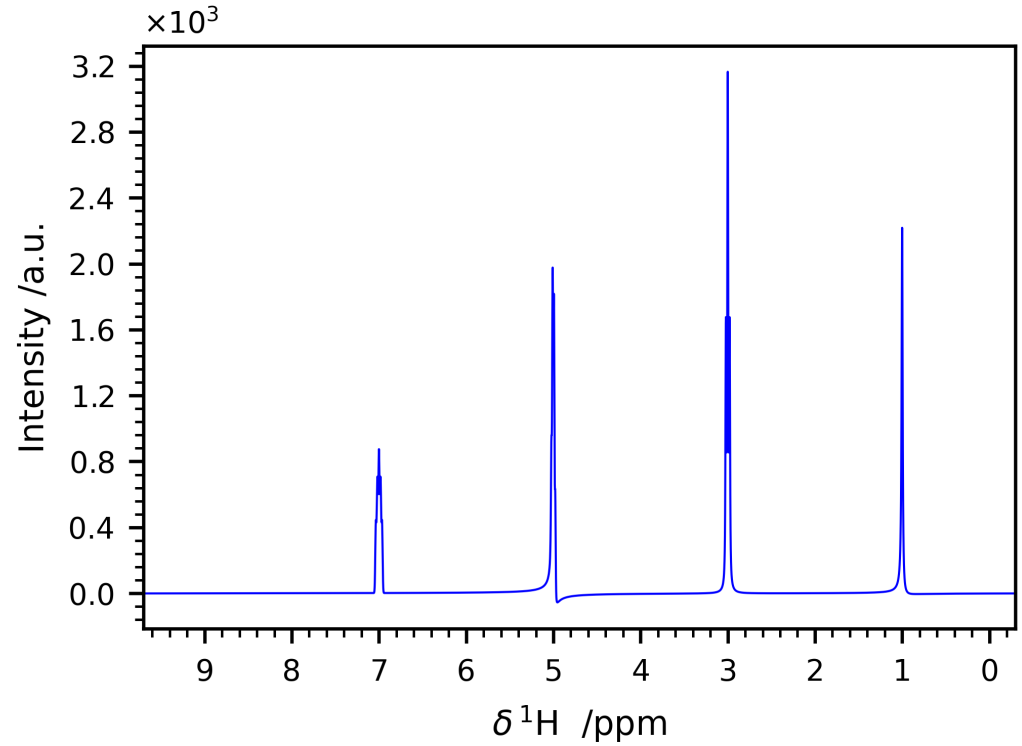

Figure 2.1: Example of a simulated 1D spectrum.

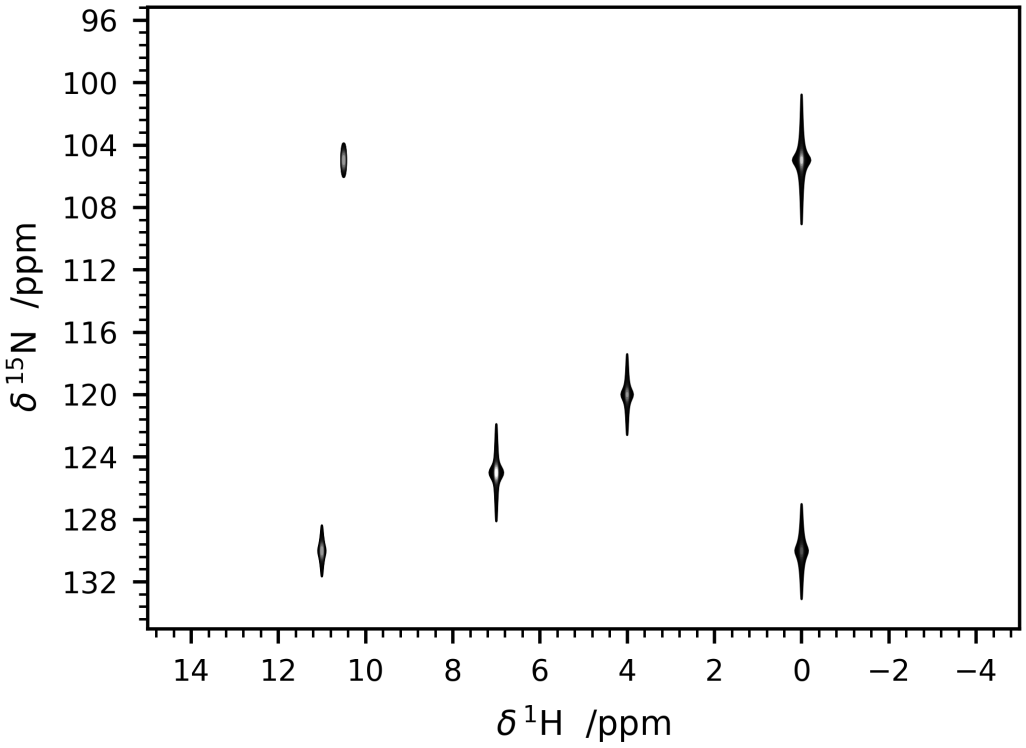

Figure 2.2: Example of a simulated 2D spectrum.

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 by using the method `mount`, 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)
```

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. In particular, `freq_f1` numbers the experiments sequentially from 0 to $N-1$, whereas `ppm_f1` does it from 1 to N. Therefore, when calling the method `projf2` to extract the experiments as `Spectrum_1D` objects, the argument must follow `ppm_f1`. As an example, to project the first experiment, one should type

```
s.projf2(1)
```

The user can replace this 'standard' numbering with the actual parameter that is varied during the evolution of the indirect dimension, by substitution of the `ppm_f1` attribute. As a result, the projection must be performed according to this new scale.

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.pknl()       # Tries to remove the digital filter
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.

2.6 Deconvolution of 1D datasets

The class `fit.Voigt_Fit` in KLASSEZ offers a very convenient interface to deconvolve a spectrum by fitting. A shortcut to the class, which initializes the parameters automatically, is implemented in the attribute `F` of `Spectrum_1D`.

To generate the input guess for the fit, you have to call the method `iguess` of the class. This can work in two different modes: the default one, which allows to build the guess peak-by-peak, and with `auto=True`, that features a peak-picker for the selection. The former is more precise, the second is much faster.

Whatever the employed method, the building of the initial guess is a two-stage process. First, you must zoom in with the matplotlib interactive viewer on the region of the spectrum you are interested in. Then, you can build the guess following the instructions in the GUI. When you press 'SAVE', your guess is stored, and the spectrum returns to the original view.

The 'manual' mode allows to optimize a polynomial baseline for each interval. A button labelled 'SET BASL' must be pressed when a satisfying region is highlighted in the GUI: this allows the scale on which the baseline is computed to be correctly aligned to the region itself. When this step is correctly performed, the box next to the button turns from red to green. Should the region be moved during the optimization of the initial guess, the box turns back to red, and the 'SET BASL' button must be pressed again to adjust the baseline scale accordingly.

The information on the peaks is saved in a `.vf` file, which can be imported with the function `fit.read_vf`. There are two kind of `.vf` file: `.ivf`, that marks initial guesses, and `.fvf`, for the results of the fit. However, this is a human-only distinction, as the structure of the files is the same.

An example of `.vf` file is shown here:

```
! Initial guess computed by francesco on 11/11/2024 at 15:48:44

         Region;     Intensity
-------------------------------------------------------------------------------------------
 193.317:168.041; 8.08246575e+00

   #;             u;          fwhm;    Rel. I.;     Phase;      Beta;      Group
-------------------------------------------------------------------------------------------
   1;   179.94060191;    172.500000;  1.000000;   -10.000;   0.00000;          0
-------------------------------------------------------------------------------------------

===========================================================================================

         Region;     Intensity
-------------------------------------------------------------------------------------------
    59.936:6.662;  5.02908980e+01

   #;             u;          fwhm;    Rel. I.;     Phase;      Beta;      Group
-------------------------------------------------------------------------------------------
   2;    40.29851786;    150.000000;  0.214286;     0.000;   0.00000;          0
   3;    24.98695246;    140.000000;  0.785714;    10.000;   0.00000;          0
-------------------------------------------------------------------------------------------

===========================================================================================
```

The header line, that starts with a `!`, is a comment, and acts as a separator between different attempts of the fit. In fact, `.vf` files are never overwritten: working again on the same file appends the information at the bottom. Hence, there is a parameter `n` in the `fit.read_vf` function that allows to select which attempt to read.

Then, a series of blocks follow. Each block marks a region of selection: the keys 'Region' and 'Intensity' mark the limits of the fitting window, and the total intensity of the peaks. Under this line, there is a table that collects the peak parameters. As a final information there might be the baseline coefficients for the given region, which start with the key 'BASL_C'. Should this line be missing, it means that the baseline was not optimized during the computation of the guess, and the coefficients will all be set to 0 when the file is read. The end of the block is marked with a line of '='.

The method `iguess` automatically search for the existing input file. If it finds it, it is automatically loaded. Otherwise, the GUI for the computation of the initial guess opens up.

The fit can be performed by calling the method `dofit`, which returns a list of `lmfit.MinimizerResult` objects (one for each region) for a detailed inspection on how the fit performed. The behavior of the fit can be customized by setting the parameters of the method (see examples or the dedicated page of the manual). The fit goes region-by-region, and the results are saved in a `.fvf` file.

A `.fvf` file can be loaded using the method `load_fit`.

Either the initial guess or the result of the fit can be conveniently visualized by using the method `plot`. Alternatively, the arrays of the model can be retrieved by calling `calc_fit_lines`. The method `res_histogram` computes the histogram of the residuals, for a better understanding of the outcome of the fit procedure.

2.7 Example scripts

2.7.1 Reading and processing of 1D spectra

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

from klassez import *

# Be aware that this is a BASIC processing
# Read the documentation of the functions to see the full powers

if 1:
    # This example is for the simulated data
    s = Spectrum_1D('acqus_1D', isexp=False)
    s.to_vf()  # You can convert info on peaks to .ivf for fitting
else:
    # Use the following to read experimentals:
    spect = 'bruker', 'jeol', 'varian', 'magritek', 'oxford' # One of these
    s = Spectrum_1D(path_to_dataset, spect=spect)

# Setup the processing
#   Apodization
#       Follow the table in the user manual to see what reads what
s.procs['wf']['mode'] = 'em'
s.procs['wf']['lb'] = 5
#   Zero-filling
s.procs['zf'] = 2**14

#   Apply processing and do FT
```

```
s.process()
# Remove the digital filter
s.pknl()
# Phase correction
s.adjph()
# Plot the data
s.plot()
```

2.7.2 Fit 1D spectrum

The beginning of the script is the same of the reading example.

```
# s.F is a fit.Voigt_Fit object
filename = 'test_1D_fit' # base filename for everything fit-related
# Compute the initial guess
auto = False      # True for peak-picker, False for manual
s.F.iguess(filename=filename, auto=auto)

if 0:  # Do the fit
    lmfit_result = s.F.dofit( ### Parameters of the fitting ###
            u_lim=5,          # movement for chemical shift /ppm
            f_lim=50,         # movement for linewidth /Hz
            k_lim=(0, 3),     # limits for intensity
            vary_phase=True,  # optimize the phase of the peak
            vary_b=True,      # optimize the lineshape (L/G ratio)
            method='leastsq', # optimization method
            itermax=10000,    # max. number of iterations
            fit_tol=1e-10,    # arrest criterion threshold (see lmfit for details)
        basl_fit='fixed'  # how to handle the baseline during the fit
            filename=filename, # filename for the .fuf file
            )
else:
    # Load an existing .fuf file
    s.F.load_fit(filename=filename)

# Plot the results
s.F.plot(what='result',  # what='iguess' for initial guess
        show_total=True, # Show the total trace or not
        show_res=True,   # Show the residuals
        res_offset=0.1,  # Displacement of the residuals (plots residuals - res_offset)
        labels=None,     # Labels for the peaks
        filename=filename, # Filename for the figures
        ext='png',       # format of the figure
        dpi=300,         # Resolution of the figure
        )

# Compute histogram of the residuals
s.F.res_histogram(what='result',
            nbins=500,   # Number of bins of the histogram
            density=True, # Normalize them
            f_lims=None, # Limits for x axis
            xlabel='Residuals', # Guess what!
            x_symm=True, # Symmetrize the x-scale
            barcolor='tab:green',   # Color of the bars
```

```
            fontsize=20, # Guess what!
            filename=filename, ext='png', dpi=300)

# Convert the tables of numbers in arrays
peaks, total, limits, whole_basl = s.F.get_fit_lines(what='result')
```

2.7.3 Read and process 2D spectrum

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

from klassez import *

# Be aware that this is a BASIC processing
# Read the documentation of the functions to see the full powers

if 1:
    # This example is for the simulated data
    s = Spectrum_2D('acqus_2D', isexp=False)
else:
    # For experimentals, at version 0.4a.7 klassez reads only 2D bruker
    s = Spectrum_2D(path_to_dataset)

# Setup the processing
#   Apodization
#       Follow the table in the user manual to see what reads what
#       REMEMBER: index 0 is F1, index 1 is F2, for procs
s.procs['wf'][1]['mode'] = 'em'
s.procs['wf'][1]['lb'] = 5
s.procs['wf'][0]['mode'] = 'qsin'
s.procs['wf'][0]['ssb'] = 2
#   Zero-filling
s.procs['zf'] = 512, 2048

#   Apply processing and do FT
s.process()
# Remove the digital filter
s.pknl()
# Phase correction
s.adjph()
# Plot the data
s.plot()

# Extract projections
ppm_f2 = 180
ppm_f1 = 10
s.projf1(ppm_f2)  # Extract F1 trace @ ppm_f2 ppm
f1 = s.Trf1[f'{ppm_f2:.2f}'] # Call it back: it is a Spectrum_1D object!
f1.plot()
s.projf2(ppm_f1)  # Extract F2 trace @ ppm_f1 ppm
f2 = s.Trf2[f'{ppm_f1:.2f}'] # Call it back: it is a Spectrum_1D object!
f2.plot()
```

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.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

3.1.2 misc.binomial_triangle(n)

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

```
In: > binomial\_triangle(4)
    > 1 3 3 1
```

Parameters:

- n: *int*
  Row index

Returns:

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

3.1.3 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

3.1.4 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.

3.1.5 misc.data2wav(data, filename='audiofile', cutoff=None, rate=44100)

Converts an array of data in a .wav file. The data are converted in float32 format, then normalized to fit the (-1, 1) interval

Parameters:

- data: *ndarray*
  Data to listen to
- filename: *str*
  Filename for the .wav file, without extension
- cutoff: *float or None*
  Clipping borders for the audio. If None, no clipping is performed
- rate: *int*
  Sample rate in samples/sec

3.1.6 misc.edit_checkboxes(checkbox, xadj=0, yadj=0, dim=100, 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
- dim: *float*
  Area of the square, in pixels. Default value is 25
- color: *str or list or None*
  If it is not None, change color to the lines

3.1.7 misc.extend_taq(old_taq, newsize=None)

Extend the acquisition timescale to a longer size, using the same dwell time

Parameters:

- old_taq: *1darray*
  Old timescale
- newsize: *int*
  New size of acqusition timescale, in points

Returns:

- new_taq: *1darray*
  Extended timescale

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 tovalue

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

Converts xfrom 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.

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 ax as follows:

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

where height = max(data) - min(data)

Parameters:

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

Returns:

- lims: *tuple*
  Bottom, Top

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 id 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 times 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.lenslice(a)

Calculates the length of a slice, i.e. the length an array would have when sliced with this slice.

Parameters:

- a: *slice*
  Slice of which to calculate the length

Returns:

- length: *int*
  Length of the slice

3.1.16 misc.load_ser(path, TD1=1, BYTORDA=0, DTYPA=0, cplx=True)

Reads a binary file and transforms it in an array. 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:

- path : *str*
  Path to the file to read
- TD1: *int*
  Number of experiments in the indirect dimension
- BYTORDA: *int*
  Endianness of data
- DTYPA: *int*
  Data type format
- cplx: *bool*
  If True, the input data are interpreted as complex, which means that in the direct dimension there will be real and imaginary parts alternated.

Returns:

- data: *2darray*
  Array of data.

3.1.17 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

3.1.18 misc.makeacqus_1D_jeol(dic)

Given a dictionary from a 1D spectrum (generated by jeol_parser.parse), this function builds the acqus file with only the 'important' parameters.

Parameters:

- dic: *dict*
  Dictionary generated with jeol_parser.parse

Returns:

- acqus : *dict*
  Dictionary with only few parameters

3.1.19 misc.makeacqus_1D_oxford(dic)

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

Parameters:

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

Returns:

- acqus : *dict*
  Dictionary with only few parameters

3.1.20 misc.makeacqus_1D_spinsolve(dic)

Given a NMRGLUE dictionary from a 1D spectrum (generated by ng.spinsolve.read), this function builds the acqus file with only the 'important' parameters. Be sure to get the info from all the configuration files!

Parameters:

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

Returns:

- acqus : *dict*
  Dictionary with only few parameters

3.1.21 misc.makeacqus_1D_varian(dic)

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

Parameters:

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

Returns:

- acqus : *dict*
  Dictionary with only few parameters

3.1.22 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

3.1.23 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.

3.1.24 misc.merge_dict(*dics)

Merge a sequence of dictionaries in a single dictionary.

Parameters:

- dics: *sequence of dict*
  Dictionaries to merge

Returns:

- merged_dict: *dict*
  Merged dictionary

3.1.25 misc.molfrac(n)

Computes the 'molar fraction' 'x' of the array 'n'. Also computes the total amount.

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'

3.1.26 misc.noise_std(y)

Calculates the standard deviation of the noise using the Bruker formula. Taken $\mathtt{y}$ as an array of r points, and $\mathtt{y}[k]$ its k-th entry:

$$\sigma_N = \frac{1}{\sqrt{r-1}}\sqrt{\sum_{k=0}^{r-1}(\mathtt{y}[k]^2) - \frac{1}{r}\left[\left(\sum_{k=0}^{r-1}\mathtt{y}[k]\right)^2 + \frac{3}{r^2-1}\left(\sum_{k=0}^{\lfloor r/2\rfloor-1}(k+1)(\mathtt{y}[\lfloor r/2\rfloor+k] - \mathtt{y}[\lfloor r/2\rfloor-k-1])\right)^2\right]}$$

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.

3.1.27 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.

3.1.28 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.

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

Converts xfrom 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

3.1.30 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'

3.1.31 misc.pretty_scale(ax, limits, axis='x', n_major_ticks=10, *, minor_each=5, fmt=None)

This function computes a pretty scale for your plot. Calculates and sets a scale made of 'n_major_ticks' numbered ticks, spaced by 'minor_each' 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, 'z' for z-axis
- n_major_ticks: *int*
  Number of numbered ticks in the final scale. An oculated choice gives very pleasant results.
- minor_each: *int*
  Number of divisions for each interval between two major ticks
- fmt: *str*
  String-formatting for the numbers on the axis. Should be given as e.g. '.3f'

3.1.32 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

3.1.33 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

3.1.34 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 the correspondant dictionary

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'

3.1.35 misc.px2in(*px_args)

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

Parameters:

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

Returns:

- in_args: *tuple of floats*
  Values in inches

3.1.36 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

3.1.37 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.

3.1.38 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.

3.1.39 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

3.1.40 misc.set_ylim(ax, data_inp)

Set the limits on the y-axis on the ax subplot. The values are computed using misc.get_ylim.

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.

3.1.41 misc.show_cmap(cmap, N=10, start=0, end=1, filename=None)

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.
- filename: *str or None*
  Filename of the figure to be saved. The '.png' extension is added automatically. If None, the figure is shown instead

---

3.1.42 misc.snr(data, signal=None, n_reg=None)

Computes the signal to noise ratio of a 1D spectrum.

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

3.1.43 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

3.1.44 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

3.1.45 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

3.1.46 misc.sum_overlay(y1, y2, x0, x=None)

Compute the sum of two arrays y_1 and y_2 of different dimensions. The sum starts at x_0 (on the scale x) as left anchor point.

Parameters:

- y1: *1darray*
  Original array
- y2: *1darray*
  Array to sum
- x0: *float*
  Value on the x scale as left anchor point
- x: *1darray*
  Reference scale. If None, the index points is used.

Returns:

- ym: *1darray*
  Summed arrays

3.1.47 misc.trim_data(ppm_scale, y, lims)

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
- lims: *tuple*
  ppm values where to start and stop trimming

Returns:

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

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

Trims data and the 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

3.1.49 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, concatenated

3.1.50 misc.write_acqus_1D(acqus, path='sim_in_1D')

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.

3.1.51 misc.write_acqus_2D(acqus, 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.

3.1.52 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.

3.1.53 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.

- 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

3.1.54 misc.zero_crossing(array, after=False)

Find the indices where the elements in the array change sign. The identified positions are the ones before the sign changes. This behavior can be modified by setting 'after=True'.

Parameters:

- array: *1darray*
  Data to analyze
- after: *bool*
  If True, returns the indices of the element after the sign change; if False, the indices before.

Returns:

- zerocross: *1darray*
  Position of the zero-crossing, according to 'before'

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.abc(ppm, data, n=5, lims=None, alpha=2.75, qfil=False, qfilp={'u': 4.7, 's': 10})

Automatic computation of a baseline for a spectrum using a thresholding-based method for the detection of the baseline-only region, followed by a weighted linear least squares optimization with a polynomion of degree n-1. The weights are computed on the absolute value of the first derivative of the spectrum. Set qfil to True if there is a very intense solvent peak that would hamper the computation of the threshold.

Parameters:

- ppm: *1darray*
  PPM scale of the spectrum
- data: *1darray*
  The spectrum to baseline-correct
- n: *int*
  Number of coefficients of the polynomial baseline
- lims: *tuple or None*
  Limits for the region on which to compute the baseline, in ppm
- alpha: *float*
  The threshold will be set as thr = alpha * np.std(np.gradient(data))
- qfil: *bool*
  Choose whether to apply a filter on the solvent region (True) or not (False)
- qfilp: *dict*
  - 'u' = center of the filter in ppm
  - 's' = width of the filter in Hz

Returns:

- baseline: *1darray*
  Computed baseline

3.2.2 processing.abs(ppm, data, n=5, lims=None, alpha=2.75, qfil=False, qfilp={'u': 4.7, 's': 10})

Computes the baseline correction on data using processing.abc, and gives back the subtracted spectrum. The imaginary part of the spectrum is reconstructed using processing.hilbert.

Parameters:

- ppm: *1darray*
  PPM scale of the spectrum
- data: *1darray*
  The spectrum to baseline-correct
- n: *int*
  Number of coefficients of the polynomial baseline
- lims: *tuple or None*
  Limits for the region on which to compute the baseline, in ppm
- alpha: *float*
  The threshold will be set as thr = alpha * np.std(np.gradient(data))
- qfil: *bool*
  Choose whether to apply a filter on the solvent region (True) or not (False)
- qfilp: *dict*
  - 'u' = center of the filter in ppm
  - 's' = width of the filter in Hz

Returns:

- S: *1darray*
  Baseline-subtracted spectrum

3.2.3 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 $\mathbb{h}$ as:

$$S = -\sum_j \mathbb{h}[j] \ln(\mathbb{h}[j])$$

and

$$h = \frac{\left|\mathbb{r}[j]^{(m)}\right|}{\sum_j |\mathbb{r}[j]^{(m)}|}$$

where

$$\mathbb{r} = \mathrm{Re}\{\mathrm{spectrum}\, e^{-\mathrm{i}\phi}\}$$

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

$$S + P(\mathbb{r})$$

where $P(\mathbb{r})$ 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

3.2.4 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

3.2.5 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, indicates the location of a file to be read with np.loadtxt. If it is None, the partitioning is done interactively.

3.2.6 processing.blp(data, pred=1, order=8)

Applies backward linear prediction by calling processing.lp with mode='b'.

Parameters:

- data: *1darray*
  FID to be linear-predicted
- pred: *int*
  Number of points to predict
- order: *int*
  Number of coefficients to use for the prediction

Returns:

- lpdata: *1darray*
  FID with linear prediction applied.

3.2.7 processing.blp_ng(data, pred=1, order=8, N=2048)

Performs backwards linear prediction on data. This function calls nmrglue.process.proc_lp.lp with most of the parameters set automatically. The algorithm predicts 'pred' points of the FID using 'order' coefficient for the linear interpolation. Only the first N points of the FID are used in the LP equation, because the computational cost scales with n**2, making the use of more than 8k points not effective: using more points brings negligible contiribution to the final result. For Oxford spectra, set 'pred' to half the value written in 'TDoff'.

Parameters:

- data: *ndarray*
  Data on which to perform the linear prediction. For 2d data, it is performed row-by-row
- pred: *int*
  Number of points to be predicted
- order: *int*
  Number of coefficients to be used for the prediction
- N: *int*
  Number of points of the FID to be used in the calculation

Returns:

- datap: *ndarray*
  Data with the predicted points appended at the beginning

3.2.8 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 $\mathbb{H} = \mathbb{U}\mathbb{S}\mathbb{V}^H$
3. Keep only the first `nc` singular values, and put all the rest to 0 ($\mathbb{S} \to \mathbb{S}'$)
4. Rebuild $\mathbb{H}' = \mathbb{U}\mathbb{S}'\mathbb{V}^H$
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

3.2.9 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

3.2.10 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

3.2.11 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.

3.2.12 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.

3.2.13 processing.convolve(in1, in2)

Perform the convolution of the two array by multiplying their inverse Fourier transform. The two arrays must have the same dimension.

Parameters:

- in1: *ndarray*
  First array
- in2: *ndarray*
  Second array

Returns:

- cnv: *ndarray*
  Convolved array

3.2.14 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)
```

3.2.15 processing.em(data, lb, sw)

Exponential apodization

Parameters:

- data: *ndarray*
  Input data
- lb: *float*
  Lorentzian broadening. It should be positive.
- sw: *float*
  Spectral width /Hz

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

Performs the full processing of a 1D NMR FID (data).

Parameters:

- data: *1darray*
  Input data
- wf: *dict*
  {'mode': function to be used, 'parameters': different from each function}
- zf: *int*
  final size of spectrum
- fcor: *float*
  weighting factor for the FID first point
- tdeff: *int*
  number of points of the FID to be used for the processing.

Returns:

- datap: *1darray*
  Processed data

3.2.17 processing.ft(data0, alt=False, fcor=0.5)

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

Parameters:

- data0: *ndarray*
  Array to Fourier-transform
- alt: *bool*
  negates the sign of the odd points, then take their complex conjugate. Required for States-TPPI processing.
- fcor: *float*
  weighting factor for FID 1st point. Default value (0.5) prevents baseline offset

Returns:

- dataft: *ndarray*
  Transformed data

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

Gaussian apodization. The parameter 'lb' controls the sharpening factor of a rising exponential, and behaves exactly as in processing.em. In contrast, 'gb' controls the gaussian decay factor. Apply this function VERY CAREFULLY. Choose the right values through the interactive processing.

Parameters:

- data: *ndarray*
  Input data
- lb: *float*
  Lorentzian sharpening /Hz. It should be negative.
- gb: *float*
  Gaussian broadening. It should be positive.
- gc: *float*
  Gaussian center, relatively to the FID length: 0 <= gc <= 1
- sw: *float*
  Spectral width /Hz

Returns:

- pdata: *ndarray*
  Processed data

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

Bruker-style Gaussian apodization. Apply this function VERY CAREFULLY. Choose the right values through the interactive processing.

Parameters:

- data: *ndarray*
  Input data
- lb: *float*
  Lorentzian sharpening /Hz. It should be negative.
- gb: *float*
  Gaussian broadening. It should be positive.
- sw: *float*
  Spectral width /Hz

Returns:

- pdata: *ndarray*
  Processed data

3.2.20 processing.hilbert(f)

Computes the Hilbert transform of real vector f in order to retrieve its imaginary part. Make sure that the original spectrum was zero-filled to at least twice the original size of the FID. The algorithm computes the convolution by means of FT, as follows:

- make IFT of $f = a$
- compute $h =$ `[1j for x in range(N)if x<N/2 else -1j]`
- Compute $b = ha$
- Build $d = a + \mathrm{i}b$
- make FT of $d = F$
- replace Re(F) with f

Parameters:

- f: *ndarray*
  Array of which you want to compute the imaginary part

Returns:

- f_cplx: *ndarray*
  Complex version of f

3.2.21 processing.hilbert2(data)

Retrieve the imaginary parts of a hypercomplex dataset, when you only have the rr part.

- rr = Re(Ht{rr})
- ir = Im(Ht{rr})
- ri = - Im(Ht{rr.T}.T)
- ii = Im(Ht{ri})

Parameters:

- data: *2darray*
  rr part

Returns:

- rr: *2darray*
  Real part in f2, real part in f1
- ir: *2darray*
  Imaginary part in f2, real part in f1
- ri: *2darray*
  Real part in f2, imaginary part in f1
- ii: *2darray*
  Imaginary part in f2, imaginary part in f1

3.2.22 processing.ift(data0, alt=False, fcor=0.5)

Inverse Fourier transform in NMR sense. This means that the input dataset is reversed before to do iFT.

Parameters:

- data0: *ndarray*
  Array to Fourier-transform
- alt: *bool*
  negates the sign of the odd points, then take their complex conjugate. Required for States-TPPI processing.
- fcor: *float*
  weighting factor for FID 1st point. Default value (0.5) prevents baseline offset

Returns:

- dataft: *ndarray*
  Transformed data

3.2.23 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 the point scale
- lims: *tuple or None*
  Integration range. If None, the whole function is integrated.

Returns:

- Fx: *ndarray*
  Integrated function.

3.2.24 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.

3.2.25 processing.integrate(fx, x=None, lims=None)

Calculates the definite integral of fx as I = F[-1] - F[0]. 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 the point scale
- lims: *tuple or None*
  Integration range. If None, the whole function is integrated.

Returns:

- I: *float*
  Integrated function.

3.2.26 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]

3.2.27 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

3.2.28 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:

3.2.29 processing.interactive_phase_1D(ppmscale, S)

This function allow to adjust the phase of 1D spectra interactively. Use the mouse scroll to regulate the values.

Parameters:

- ppmscale: *1darray*
  ppm scale of the spectrum. Used to regulate the pivot position
- S: *1darray*
  Spectrum to be phased. Must be complex!

Returns:

- phased_data: *1darray*
  Phased spectrum

3.2.30 processing.interactive_phase_2D(ppm_f1, ppm_f2, S, hyper=True)

Interactively adjust the phases of a 2D spectrum S must be complex or hypercomplex, so BEFORE TO UNPACK

Parameters:

- ppm_f1: *1darray*
  ppm scale of the indirect dimension
- ppm_f2: *1darray*
  ppm scale of the direct dimension
- S: *2darray*
  Data to be phase-adjusted
- hyper: *bool*
  True if S is hypercomplex, False if S is just complex

Returns:

- S: *2darray*
  Phased data
- final_values_f1: *tuple*
  (p0_f1, p1_f1, pivot_f1)
- final_values_f2: *tuple*
  (p0_f2, p1_f2, pivot_f2)

3.2.31 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)

3.2.32 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

3.2.33 processing.inv_convolve(in1, in2)

Perform the inverse-convolution of the two array by dividing their inverse Fourier transform. The two arrays must have the same dimension.

Parameters:

- in1: *ndarray*
  First array
- in2: *ndarray*
  Second array

Returns:

- cnv: *ndarray*
  Convolved array

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

Performs the full inverse processing of a 1D NMR spectrum (data).

Parameters:

- data: *1darray*
  Spectrum
- 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:

- pdata: *1darray*
  FID

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

Reverts the full processing of a 2D NMR FID (data).

Parameters:

- data: *2darray*
  Input data, hypercomplex
- wf: *list of dict*
  list of two entries [F1, F2]. Each entry is a dictionary of window functions
- size: *list of int*
  Initial size of FID
- fcor: *list of float*
  first fid point weighting factor [F1, F2]
- FnMODE: *str*
  Acquisition mode in F1

Returns:

- data: *2darray*
  Processed data

3.2.36 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

3.2.37 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

3.2.38 processing.lp(data, pred=1, order=8, mode='b')

Apply linear prediction on the dataset. This method solves the linear system

$$\mathbb{D}\mathbb{a} = \mathbb{d}$$

where $\mathbb{a}$ is the array of lp coefficients.

Parameters:

- data: *1darray*
  FID to be linear-predicted
- pred: *int*
  Number of points to predict
- order: *int*
  Number of coefficients to use for the prediction
- mode: *str*
  'f' for forward linear prediction, 'b' for backward linear prediction

Returns:

- newdata: *1darray*
  FID with linear prediction applied.

3.2.39 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

3.2.40 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

---

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

Computes the frequency scale of the NMR spectrum, given the # 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.

3.2.42 processing.mcr(input_data, nc, f=10, tol=1e-05, itermax=10000.0, P='H', 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, CS, and the single C and 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
- P: *str or 2darray*
  'H' for horizontal stacking, 'V' for vertical stacking, or custom matrix as explained in the description of mcr_stack
- oncols: *bool*
  True to estimate S with processing.simplisma, False to estimate C.

Returns:

- CS_f: *2darray or 3darray*
  Final denoised data matrix
- C_f: *2darray or 3darray*
  Final C matrix
- S_f: *2darray or 3darray*
  Final S matrix

3.2.43 processing.mcr_als(D, C, S, itermax=10000, tol=1e-05)

Performs alternating least squares to get the final $\mathbb{C}$ and $\mathbb{S}$ matrices. Being the fundamental MCR equation: $\mathbb{D} = \mathrm{CS} + \mathrm{E}$ 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 C and S matrices between two subsequent steps:

$$rC = \|\mathbb{C}_k - \mathbb{C}_{k-1}\| \quad rS = \|\mathbb{S}_k - \mathbb{S}_{k-1}\|$$

The convergence is reached when: `rC <= tol && rS <= tol`

Parameters:

- D: *2darray*
  Input data, of dimensions m x n
- C: *2darray*
  Estimation of the C matrix, of dimensions m x nc.
- S: *2darray*
  Estimation of the S matrix, of dimensions nc x n.
- itermax: *int*
  Maximum number of iterations
- tol: *float*
  Threshold for the arrest criterion.

Returns:

- C: *2darray*
  Optimized C matrix, of dimensions m x nc.
- S: *2darray*
  Optimized S matrix, of dimensions nc x n.

3.2.44 processing.mcr_stack(input_data, P='H')

Performs matrix augmentation by assembling input_data according to the positioning matrix P. P has two default modes: 'H' = horizontal stacking; 'V' = vertical stacking. Otherwise, a custom $\mathbb{P}$ matrix can be given as follows. The entries of the $\mathbb{P}$ matrix are the indices of the data in input_data. The shape of the matrix determines the final arrangement.

Example: if input_data is `[a, b, c, d, e, f]`, and one wants to obtain `[[a, b], [d,c], [f, e]]` the correspondant $\mathbb{P}$ matrix is:

```
P = [[0, 1], [3, 2], [5, 4]]
```

If each dataset in input_data has dimensions (m, n) and P has dimensions (u,v), then the returned data matrix will have dimensions (mu, nv).

Parameters:

- input_data: *3darray*
  Contains the spectra to be stacked together. The index that runs on the datasets must be the first one.

- P: *str or 2darray*
  'H' for horizontal stacking, 'V' for vertical stacking, or custom matrix as explained in the description

Returns:

- data: *2darray*
  Augmented data matrix.

3.2.45 processing.mcr_unpack(C, S, nds, P='H')

Reverts matrix augmentation of mcr_stack. The denoised spectra can be calculated by matrix multiplication: `D[k] = C_f[k] S_f[k] #for k = 0,..., nds-1`

Parameters:

- C: *2darray*
  MCR C matrix
- S: *2darray*
  MCR S matrix
- nds: *int*
  number of experiments
- P: *str or 2darray*
  'H' for horizontal stacking, 'V' for vertical stacking, or custom matrix as explained in the description of mcr_stack

Returns:

- C_f: *list of 2darray*
  Disassembled MCR C matrix
- S_f: *list of 2darray*
  Disassembled MCR C matrix

3.2.46 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 p1 = 360 * 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

3.2.47 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.

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 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)

3.2.48 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

3.2.49 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_corr : *ndarray*
  Processed FID

3.2.50 processing.qsin(data, ssb)

Sine-squared apodization.

Parameters:

- ssb: *int*
  Sine bell shift.

3.2.51 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.

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

Renconstruct hypercomplex 2D NMR data given the 4 ser files

Parameters:

- rr: *2darray*
  Real F2, Real F1
- ir: *2darray*
  Imaginary F2, Real F1
- ri: *2darray*
  Real F2, Imaginary F1
- ii: *2darray*
  Imaginary F2, Imaginary F1

Returns:

- data: *2darray*
  Hypecomplex matrix

3.2.53 processing.rev(data)

Reverse data over its last dimension

3.2.54 processing.rpbc(data, split_imag=False, n=5, basl_method='huber', basl_thresh=0.2, basl_itermax=2000, **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 (ref.). 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

---

3.2.55 processing.simplisma(D, nc, f=10, oncols=True)

Finds the first nc purest components of matrix $\mathbb{D}$ 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 x n
- nc: *int*
  Number of components to be found. This determines the final size of the C and S matrices.
- f: *float*
  Percentage of allowed noise.
- oncols: *bool*
  If True, simplisma estimates the S matrix, otherwise estimates C.

Returns:

- C: *2darray*
  Estimation of the C matrix, of dimensions m x nc.
- S: *2darray*
  Estimation of the S matrix, of dimensions nc x n.

3.2.56 processing.sin(data, ssb)

Sine apodization.

Parameters:

- ssb: *int*
  Sine bell shift.

3.2.57 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.

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

3.2.58 processing.stack_fids(*fids, filename=None)

Stacks together FIDs in order to create a pseudo-2D experiment. This function can handle either arrays or Spectrum_1D objects.

Parameters:

- fids: *sequence of 1darrays or Spectrum_1D objects*
  Input data.
- filename: *str*
  Location for a .npy file to be saved. If None, no file is created.

Returns:

- p2d: *2darray*
  Stacked FIDs.

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

Sum up a CPMG echo-train FID into echoes so to be enchance the SNR. This function calls processing.split_echo_train with the same parameters.

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

3.2.60 processing.td_eff(data, tdeff)

Uses only the first tdeff points of data. tdeff must be a list as long as the dimensions: tdeff = [F1, F2, ..., Fn]

Parameters:

- data: *ndarray*
  Data to be trimmed
- tdeff: *list of int*
  Number of points to be used in each dimension

3.2.61 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.

3.2.62 processing.unpack_2D(data)

Separates hypercomplex data into 4 distinct ser files

Parameters:

- data: *2darray*
  Hypercomplex matrix

Returns:

- rr: *2darray*
  Real F2, Real F1
- ir: *2darray*
  Imaginary F2, Real F1
- ri: *2darray*
  Real F2, Imaginary F1
- ii: *2darray*
  Imaginary F2, Imaginary F1

3.2.63 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

3.2.64 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

---

3.2.65 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 (data). The returned values depend on u: it is True, returns a sequence of 2darrays depending on FnMODE, otherwise just the complex/hypercomplex data after FT in both dimensions

Parameters:

- data: *2darray*
  Input data
- wf: *sequence of dict*
  (F1, F2); {'mode': function to be used, 'parameters': different from each function}
- zf: *sequence of int*
  final size of spectrum, (F1, F2)
- fcor: *sequence of float*
  weighting factor for the FID first point, (F1, F2)
- tdeff: *sequence of int*
  number of points of the FID to be used for the processing, (F1, F2)
- u: *bool*
  choose if to unpack the hypercomplex spectrum into separate arrays or not
- FnMODE: *str*
  Acquisition mode in F1

Returns:

- datap: *2darray or tuple of 2darray*
  Processed data or tuple of 2darray

3.2.66 processing.zf(data, size)

Zero-filling of data up to size in its last dimension.

Parameters:

- data: *ndarray*
  Array to be zero-filled
- size: *int*
  Number of points of the last dimension after zero-filling

Returns:

- datazf: *ndarray*
  Zero-filled data

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=, 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=, Y_label=, 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` and `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, lw=0.8, 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.
- 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

Returns:

- scale_factor: *list*
  Intensity of the spectra with respect to the original when the figure is closed

---

3.3.5 figures.dotmd_2D(ppm_f1, ppm_f2, S0, labels=None, name='dotmd_2D X_label='$ delta $ F2 /ppm', Y_label='$ delta $ F1 /ppm', n_xticks=10, n_yticks=10, Neg=False)

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

Parameters:

- 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.

Returns:

- lvl: *list*
  Intensity factors when the figure is closed

3.3.6 figures.figure1D(ppm, datax, norm=False, xlims=None, ylims=None, c='tab:blue', lw=0.5, X_label=, Y_label='Intensity /a.u.', n_xticks=10, n_yticks=10, fontsize=10, name=None, ext='tiff', 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=, Y_label='Intensity /a.u.', n_xticks=10, n_yticks=1 fontsize=10, labels=None, name=None, ext='tiff', 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=, Y_label=, lw=0.5, cmapneg=None, n_xticks=10, n_yticks=10, fontsize=10, name=None, ext='tiff', 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 and 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=, Y_label=, lw=0.5, n_xticks=10, n_yticks=10, labels=None, name=None, ext='tiff', 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
- 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=, 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.ongoing_fit(exp, calc, residual, ylims=None, filename=None, dpi=100)

Makes a figure of an ongoing fit. It displays the experimental data and the model, and the residuals in a separate window. The figure can be either saved or shown.

Parameters:

- exp: *1darray*
  Experimental data
- calc: *1darray*
  Current model
- residual: *1darray*
  Residuals of the fit
- ylims: *tuple*
  Optional limits for y-axis
- filename: *str or None*
  Filename of the figure to be saved. If None, the figure is shown instead
- dpi: *int*
  Resolution of the figure in dots per inches

3.3.13 figures.plot_fid(fid, name=None, ext='tiff', 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.

3.3.14 figures.plot_fid_re(fid, scale=None, c='b', lims=None, name=None, ext='tiff', 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

3.3.15 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. [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

3.3.16 figures.sns_heatmap(data, name=None, ext='tiff', 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

3.3.17 figures.stacked_plot(ppmscale, S, xlims=None, lw=0.5, X_label=, Y_label='Normalized intensity /a.u.', n_xticks=10, labels=None, name=None, ext='tiff', 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.cron(func, *args, **kwargs)

Decorator: use it to monitor the runtime of a function.

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

Gaussian function in the frequency domain:

Parameters:

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

Returns:

- f: *1darray*
  Gaussian function.

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

Lorentzian function in the time domain:

Parameters:

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

Returns:

- f: *1darray*
  Lorentzian function.

3.4.5 sim.f_pvoigt(x, u, fwhm, A=1, b=0)

Pseudo-Voigt function in the frequency domain:

Parameters:

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

Returns:

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

3.4.6 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

3.4.7 sim.load_sim_1D(File)

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

Parameters:

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

Returns:

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

3.4.8 sim.load_sim_2D(File, states=True)

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

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.

3.4.9 sim.mult_noise(data_size, mean, s_n)

Multiplicative noise model.

3.4.10 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

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

Simulates additive noise in the time domain.

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.

3.4.12 sim.sim_1D(File, pv=False)

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

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.

3.4.13 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 states-TPPI as default.

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.

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

Bidimensional gaussian function.

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*
  Gaussian function.

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

Bidimensional lorentzian function.

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.

3.4.16 sim.t_2Dpvoigt(t1, t2, v1, v2, fwhm1, fwhm2, A=1, b=0, states=True, alt=True)

Generates a 2D pseudo-voigt signal in the time domain. b 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.

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
- b: *float*
  Fraction of gaussianity
- states: *bool*
  Set to True for 'FnMODE':'States-TPPI
- alt: *bool*
  Set to True for 'FnMODE':'States-TPPI46

Returns:

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

3.4.17 sim.t_2Dvoigt(t1, t2, v1, v2, fwhm1, fwhm2, A=1, b=0, states=True, alt=True)

Generates a 2D Voigt signal in the time domain. b 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.

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
- b: *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.

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

Gaussian function in the time domain.

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.

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

Lorentzian function in the time domain.

Parameters:

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

Returns:

- S: *1darray*
  Lorentzian function.

3.4.20 sim.t_pvoigt(t, u, fwhm, A=1, b=0, phi=0)

Pseudo-Voigt function in the time domain:

Parameters:

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

Returns:

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

3.4.21 sim.t_voigt(t, u, fwhm, A=1, b=0, phi=0)

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

Parameters:

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

Returns:

- S: *1darray*
  Voigt function.

3.4.22 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.

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: *1darray*
  Modified residuals according to the chosen target function

---

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.Peak `class`

Class to represent the characteristic parameters of an NMR peak, and to compute it.

Attributes:

- t: *1darray*
  Timescale for the FID
- SFO1: *float*
  Nucleus Larmor frequency
- o1p: *float*
  Carrier position
- N: *int*
  Number of points of the spectrum, i.e. after eventual zero-filling
- u: *float*
  Chemical shift /ppm
- fwhm: *float*
  Linewidth /Hz
- k: *float*
  Intensity, relative
- b: *float*
  Fraction of gaussianity (b=0 equals pure lorentzian)
- phi: *float*
  Phase /degrees
- group: *int*
  Identifier for the component of a multiplet

Methods:

__init__(self, acqus, u=None, fwhm=5, k=1, b=0, phi=0, N=None, group=0)

Initialize the class with the configuration parameters, and with defauls values, if not given.

Parameters:

- acqus: *dict*
  It should contain 't', 'SFO1', 'o1p', and 'N'
- u: *float*
  Chemical shift /ppm
- fwhm: *float*
  Linewidth /Hz
- k: *float*
  Intensity, relative

- b: *float*
  Fraction of gaussianity (b=0 equals pure lorentzian)
- phi: *float*
  Phase /degrees
- N: *int*
  Number of points of the spectrum, i.e. after eventual zero-filling. None means to not zero-fill
- group: *int*
  Identifier for the component of a multiplet

__call__(self, A=1, cplx=False, get_fid=False)

Generates a voigt signal on the basis of the stored attributes, in the time domain. Then, makes the Fourier transform and returns it after the eventual zero-filling.

Parameters:

- A: *float*
  Absolute intensity value
- cplx: *bool*
  Returns the complex (True) or only the real part (False) of the signal
- get_fid: *bool*
  If True, returns the FID instead of the transformed signal

Returns:

- sgn : *1darray*
  generated signal in the frequency domain

---

get_fid(self, A=1)

Compute and returns the FID encoding for that signal.

Parameters:

- A: *float*
  Absolute intensity value

Returns:

- sgn : *1darray*
  generated signal in the time domain

---

par(self)

Creates a dictionary with the currently stored attributes and returns it.

Returns:

- dic: *dict*
  Dictionary of parameters

3.5.3 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:

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

where d is the real part of the NMR spectrum.

Attributes:

- gamma1: *float*
  Weighting factor for function g_1
- gamma2: *float*
  Weighting factor for function g_2
- gamma3: *float*
  Weighting factor for function g_3
- e1: *float*
  Tolerance value for function g_1
- e2: *float*
  Tolerance value for function g_2

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

3.5.4 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
- SW: *float*
  Spectral width /Hz
- SFO1: *float*
  Larmor frequency of the nucleus
- o1p : *float*
  Pulse carrier frequency
- filename: *str*
  Root of the names of the files that will be saved
- X_label: *str*
  Label for the chemical shift axis in the figures
- i_guess: *list*
  Initial guess for the fit, read by a .ivf file with fit.read_vf
- result: *list*
  Result the fit, read by a .fvf file with fit.read_vf

Methods:

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

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.

- filename: *str or None*
  Root of the name of the files that will be saved

---

dofit(self, indep=True, u_lim=1, f_lim=10, k_lim=(0, 3), vary_phase=False, vary_b=True, itermax=10000, fit_tol=1e-08, filename=None, method='leastsq', basl_fit='no')

Perform a lineshape deconvolution fitting. The initial guess is read from the attribute self.i_guess. The components can be considered to be all independent from one to another by setting 'indep' to True: this means that the fit will be done using fit.voigt_fit_indep. The indep=False option has not been implemented yet.

Parameters:

- indep: *bool*
  True to consider all the components to be independent

- u_lim: *float*
  Determines the displacement of the chemical shift (in ppm) from the starting value.

- f_lim: *float*
  Determines the displacement of the linewidth (in Hz) from the starting value.

- k_lim: *float or tuple*
  If tuple, minimum and maximum allowed values for k during the fit. If float, maximum displacement from the initial guess

- vary_phase: *bool*
  Allow the peaks to change phase (True) or not (False)

- vary_b: *bool*
  Allow the peaks to change Lorentzian/Gaussian ratio

- itermax: *int*
  Maximum number of allowed iterations

- fit_tol: *float*
  Value of the target function to be set as x_tol and f_tol

- filename: *str*
  Path to the output file. If None, '<self.filename>.fvf' is used

- method: *str or list of str*
  Method to be used for the optimization. See lmfit for details. There is the option to run multiple optimizations in series.

- basl_fit: *str*
  How to address the baseline fit. The options are:

  - 'no': Do not use baseline (default)

  - 'fixed': The baseline is computed once and kept fixed during the optimization
  - 'fit': The baseline coefficients enter as fit parameters during the nonlinear optimization
  - 'calc': The baseline coefficients are calculated during the optimization via linear least-squares optimization

Returns:

- lmfit_results: *list of lmfit.minimizer.MinimizerResult*
  Sequence of the fit results, ordered as the regions dictionary

---

get_fit_lines(self, what='result')

Calculates the components, and the total fit curve used as initial guess, or as fit results.. The components will be returned as a list, not split by region.

Parameters:

- what: *str*
  'iguess' or 'result'

Returns:

- signals: *list of 1darray*
  Components used for the fit
- total: *1darray*
  Sum of all the signals
- limits_list: *list*
  List of region delimiters, in ppm
- whole_basl: *1darray*
  Computed baseline

---

iguess(self, filename=None, n=-1, ext='ivf', auto=False)

Reads, or computes, the initial guess for the fit. If the file is there already, it just reads it with fit.read_vf. Otherwise, it calls fit.make_iguess to make it.

Parameters:

- filename: *str or None*
  Path to the input file. If None, '<self.filename>.ivf' is used
- n: *int*
  Index of the initial guess to be read (default: last one)
- ext: *str*
  Extension of the file to be used
- auto: *bool*
  If True, uses the GUI for automatic peak picking, if False, the manual one

---

load_fit(self, filename=None, n=-1, ext='fvf')

Reads a file with fit.read_vf and stores the result in self.result.

Parameters:

- filename: *str*
  Path to the .fvf file to be read. If None, '<self.filename>.fvf' is used.
- n: *int*
  Index of the fit to be read (default: last one)
- ext: *str*
  Extension of the file to be used

---

plot(self, what='result', show_total=True, show_res=False, res_offset=0, show_basl=False, labels=None, filename=None, ext='tiff', dpi=600, dim=None)

Plots either the initial guess or the result of the fit, and saves all the figures. Calls fit.plot_fit. The figure <filename>_full will show the whole model and the whole spectrum. The figures labelled with _R<k> will depict a detail of the fit in the k-th fitting region. Optional labels for the components can be given: in this case, the structure of 'labels' should match the structure of self.result (or self.i_guess). This means that the length of the outer list must be equal to the number of fitting region, and the length of the inner lists must be equal to the number of peaks in that region.

Parameters:

- what: *str*
  'iguess' to plot the initial guess, 'result' to plot the fitted data
- show_total: *bool*
  Show the total trace (i.e. sum of all the components) or not
- show_res: *bool*
  Show the plot of the residuals
- res_offset: *float*
  Displacement of the residuals plot from 0, to be given as a fraction of the height of the experimental spectrum. res_offset > 0 will move the residuals BELOW the zero-line!
- show_basl: *bool*
  If True, displays the baseline on the spectrum and uses it to compute the total trace.
- labels: *list of list*
  Optional labels for the components. The structure of this parameter must match the structure of self.result
- filename: *str*
  Root of the name of the figures that will be saved. If None, <self.filename> is used
- ext: *str*
  Format of the saved figures
- dpi: *int*
  Resolution of the figures, in dots per inches

- dim: *tuple or None* Dimensions of the figures to be saved, in inches. If None, the default value of (15, 8) is used.

---

res_histogram(self, what='result', nbins=500, density=True, f_lims=None, xlabel='Residuals x_symm=True, barcolor='tab:green', fontsize=20, filename=None, ext='tiff', dpi=300)

Computes the histogram of the residuals and saves it. Employs fit.histogram to make the figure.

Parameters:

- what: *str*
  'iguess' or 'result'
- 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

---

to_tragico(self, which='iguess', filename=None)

Writes input 1 and input 2 for a TrAGICo run, on the basis of either the initial guess or the results of a fit. The files will be named '<filename>_inp1' and '<filename>_inp2', respectively.

Parameters:

- which: *str*
  'iguess' or 'result'
- filename: *str*
  Name of the file that will be saved. If None, the file will be saved in the spectrum directory

---

3.5.5 fit.Voigt_Fit_P2D `class`

This class offers an 'interface' to fit a pseudo 2D NMR spectrum.

Attributes:

- ppm_scale: *1darray*
  Self-explanatory
- S : *2darray*
  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
- filename: *str*
  Root of the names of the files that will be saved
- X_label: *str*
  Label for the chemical shift axis in the figures
- i_guess: *list*
  Initial guess for the fit, read by a .ivf file with fit.read_vf_P2D
- result: *list*
  Result the fit, read by a .fvf file with fit.read_vf_P2D

Methods:

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

Initialize the class with common values.

Parameters:

- ppm_scale: *1darray*
  ppm scale of the spectrum
- S: *2darray*
  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.

- filename: *str or None*
  Root of the name of the files that will be saved

---

dofit(self, u_tol=1, f_tol=10, vary_phase=False, vary_b=True, itermax=10000, filename=None)

Perform a lineshape deconvolution fitting by calling fit.voigt_fit_P2D. The initial guess is read from the attribute self.i_guess.

Parameters:

- u_tol: *float*
  Determines the displacement of the chemical shift (in ppm) from the starting value.

- f_tol: *float*
  Determines the displacement of the linewidth (in Hz) from the starting value.

- vary_phase: *bool*
  Allow the peaks to change phase (True) or not (False)

- vary_b: *bool*
  Allow the peaks to change Lorentzian/Gaussian ratio

- itermax: *int*
  Maximum number of allowed iterations

- filename: *str*
  Path to the output file. If None, '<self.filename>.fvf' is used

---

get_fit_lines(self, what='result')

Calculates the components, and the total fit curve used as initial guess, or as fit results.. The components will be returned as a list, not split by region.

Parameters:

- what: *str*
  'iguess' or 'result'

Returns:

- signals: *list of list of 1darray*
  Components used for the fit

- total: *2darray*
  Sum of all the signals

- limits_list: *list*
  List of the region delimiters, in ppm

---

iguess(self, input_file=None, expno=0, n=-1)

Reads, or computes, the initial guess for the fit. If the file is there already, it just reads it with fit.read_vf. Otherwise, it calls fit.make_iguess to make it.

Parameters:

- input_file: *str or None*
  Path to the input file. If None, '<self.filename>.ivf' is used
- expno: *int*
  Number of the experiment on which to compute the initial guess, in python numbering
- n: *int*
  Index of the initial guess to be read (default: last one)

---

load_fit(self, output_file=None, n=-1)

Reads a file with fit.read_vf_P2D and stores the result in self.result.

Parameters:

- output_file: *str*
  Path to the .fvf file to be read. If None, '<self.filename>.fvf' is used.
- n: *int*
  Index of the fit to be read (default: last one)

---

plot(self, what='result', show_total=True, show_res=False, res_offset=0, labels=None, filename=None, ext='tiff', dpi=600)

Plots either the initial guess or the result of the fit, and saves all the figures. Calls fit.plot_fit_P2D. The figures <filename>_full will show the whole model and the whole spectrum. The figures labelled with _R<k> will depict a detail of the fit in the k-th fitting region. Optional labels for the components can be given: in this case, the structure of 'labels' should match the structure of self.result (or self.i_guess). This means that the length of the outer list must be equal to the number of fitting region, and the length of the inner lists must be equal to the number of peaks in that region.

Parameters:

- what: *str*
  'iguess' to plot the initial guess, 'result' to plot the fitted data
- show_total: *bool*
  Show the total trace (i.e. sum of all the components) or not
- show_res: *bool*
  Show the plot of the residuals
- res_offset: *float*
  Displacement of the residuals plot from 0, to be given as a fraction of the height of the experimental spectrum. res_offset > 0 will move the residuals BELOW the zero-line!

- labels: *list of list*
  Optional labels for the components. The structure of this parameter must match the structure of self.result
- filename: *str*
  Root of the name of the figures that will be saved. If None, <self.filename> is used
- ext: *str*
  Format of the saved figures
- dpi: *int*
  Resolution of the figures, in dots per inches

---

res_histogram(self, what='result', nbins=500, density=True, f_lims=None, xlabel='Residuals x_symm=True, barcolor='tab:green', fontsize=20, filename=None, ext='tiff', dpi=300)

Computes the histogram of the residuals and saves it in the same folder of the fit figures. Employs fit.histogram to make the figure.

Parameters:

- what: *str*
  'iguess' or 'result'
- 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

3.5.6 fit.ax_histogram(ax, data0, nbins=100, density=True, f_lims=None, xlabel=None, x_symm=False, fitG=True, 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
- fitG: *bool*
  Shows the gaussian approximation
- 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.

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

Computes the histogram of data, sampling it into nbins bins.

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.

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

Create a 2D signal according to the final parameters returned by make_iguess_2D. Process it according to procs.

Parameters:

- parameters: *list or 2darray*
  sequence of the parameters: u1, u2, fwhm1, fwhm2, I, b. 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

3.5.9 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*
  Parameters 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.

3.5.10 fit.calc_R2(y, y_c)

Computes the R-squared coefficient of a linear regression as:

$$R^2 = 1 - \frac{\sum(y - y_{mean})^2}{\sum(y - y_c)^2}$$

Parameters:

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

Returns:

- R2: *float*
  R-squared coefficient

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 sgn
- baseline: *1darray*
  Polynomion built using C. 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 l. 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 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, q=True)

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 <w> the mean of variable w. Then: A = (<y_c y> - <y_c><y>) / (<y_c$\hat{2}$> - <y_c>$\hat{2}$) q = (<y_c>$\hat{2}$<y> - <y_c><y_c y>) / (<y_c$\hat{2}$> - <y_c>$\hat{2}$)

Parameters:

- y: *1darray*
  Experimental data
- y_c: *1darray*
  Model data
- q: *bool*
  If True, includes the offset in the calculation. If False, only the intensity factor is computed.

Returns:

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

3.5.14 fit.gaussian_fit(x, y, s_in=None)

Fit 'y' with a gaussian function, built using 'x' as independent variable

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=[], sens0=1)

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.

Keybinding:

- '>': increase sensitivity
- '<': decrease sensitivity
- 'up': increase value
- 'down': decrease value
- 'left': change parameter
- 'right': change parameter
- 'v': change 'vary' status
- '<': toggle automatic zoom adjustment

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
- sens0: *float*
  Default sensitivity for the change of the parameters with the mouse

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: v1(Hz), v2(Hz), fwhm1(Hz), fwhm2(Hz), A, b
- 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 the 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, fitG=True, barcolor='tab:blue', fontsize=10, name=None, ext='tiff', 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
- fitG: *bool*
  Shows the gaussian approximation
- 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=)

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 pre-defined 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(relative) * ref_int / 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*
  Parameters coefficients. No baseline corresponds to False.
- L: *list*
  List of window regions.
- baseline: *1darray*
  Baseline built from C and L.

3.5.23 fit.lr(y, x=None, force_intercept=False)

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.
- force_intercept: *bool*
  If True, forces the intercept to be zero.

Returns:

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

3.5.24 fit.lsp(y, x, n=5, w=None)

Linear-System Polynomion Make a polynomial fit on the experimental data y by solving the linear system

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

where $\mathbb{T}$ is the Vandermonde matrix of the x-scale and $\mathbb{c}$ is the set of coefficients that minimize the problem in the least-squares sense. It is also possible to make it weighted by using an array of weights $\mathbb{w}$.

Parameters:

- y: *1darray*
  Experimental data
- x: *1darray*
  Independent variable (better if normalized)
- n: *int*
  Order of the polynomion + 1, i.e. number of coefficients
- w: *1darray*
  Array of weights for the data. If None, the nonweighted approach is used

Returns:

- c: *1darray*
  Set of minimized coefficients

3.5.25 fit.make_iguess(S_in, ppm_scale, t_AQ, SFO1=701.125, o1p=0, filename='i_guess')

Creates the initial guess for a lineshape deconvolution fitting procedure, using a dedicated GUI. The GUI displays the experimental spectrum in black and the total function in blue. First, select the region of the spectrum you want to fit by focusing the zoom on it using the lens button. Then, use the '+' button to add components to the spectrum. The black column of text under the textbox will be colored with the same color of the active peak. Use the mouse scroll to adjust the parameters of the active peak. Write a number in the 'Group' textbox to mark the components of the same multiplet. Group 0 identifies independent peaks, not part of a multiplet (default). The sensitivity of the mouse scroll can be regulated using the 'up arrow' and 'down arrow' buttons. The active peak can be changed in any moment using the slider.

The baseline can be computed first by initializing the x-scale on the selected window through the 'SET BASL' button. The informer light next to the button becomes green if it is properly set. The baseline coefficients can be set with the mouse scroll analogously to any other parameter.

When you are satisfied with your fit, press 'SAVE' to write the information in the output file. Then, the GUI is brought back to the initial situation, and the region you were working on will be marked with a green rectangle. You can repeat the procedure as many times as you wish, to prepare the guess on multiple spectral windows.

Keyboard shortcuts:

- 'increase sensitivity' : '>'
- 'decrease sensitivity' : '<'
- mouse scroll up: 'up arrow key'
- mouse scroll down: 'down arrow key'
- 'add a component': '+'
- 'remove the active component': '-'
- 'change component, forward': 'page up'
- 'change component, backward': 'page down'

Parameters:

- S_in: *1darray*
  Experimental spectrum
- ppm_scale: *1darray*
  PPM scale of the spectrum
- t_AQ: *1darray*
  Acquisition timescale
- SFO1: *float*
  Nucleus Larmor frequency /MHz
- o1p: *float*
  Carrier frequency /ppm
- filename: *str*
  Path to the filename where to save the information. The '.ivf' extension is added automatically.

3.5.26 fit.make_iguess_P2D(S_in, ppm_scale, expno, t_AQ, SFO1=701.125, o1p=0, filename='i_guess')

Creates the initial guess for a lineshape deconvolution fitting procedure of a pseudo-2D experiment, using a dedicated GUI. It will be donw on only one experiment of the whole pseudo-2D. The GUI displays the experimental spectrum in black and the total function in blue. First, select the region of the spectrum you want to fit by focusing the zoom on it using the lens button. Then, use the '+' button to add components to the spectrum. The black column of text under the textbox will be colored with the same color of the active peak. Use the mouse scroll to adjust the parameters of the active peak. Write a number in the 'Group' textbox to mark the components of the same multiplet. Group 0 identifies independent peaks, not part of a multiplet (default). The sensitivity of the mouse scroll can be regulated using the 'up arrow' and 'down arrow' buttons. The active peak can be changed in any moment using the slider.

When you are satisfied with your fit, press 'SAVE' to write the information in the output file. Then, the GUI is brought back to the initial situation, and the region you were working on will be marked with a green rectangle. You can repeat the procedure as many times as you wish, to prepare the guess on multiple spectral windows.

Keyboard shortcuts:

- 'increase sensitivity' : '>'
- 'decrease sensitivity' : '<'
- mouse scroll up: 'up arrow key'
- mouse scroll down: 'down arrow key'
- 'add a component': '+'
- 'remove the active component': '-'
- 'change component, forward': 'page up'
- 'change component, backward': 'page down'

Parameters:

- S_in: *1darray*
  Experimental spectrum
- ppm_scale: *1darray*
  PPM scale of the spectrum
- expno: *int*
  Index of experiment of the pseudo 2D on which to compute the initial guess, in python numbering
- t_AQ: *1darray*
  Acquisition timescale
- SFO1: *float*
  Nucleus Larmor frequency /MHz
- o1p: *float*
  Carrier frequency /ppm
- filename: *str*
  Path to the filename where to save the information. The '.ivf' extension is added automatically.

3.5.27 fit.make_iguess_auto(ppm, data, SW, SFO1, o1p, filename='iguess')

GUI to create a .ivf file, used as initial guess for Voigt_Fit. The computation of the peak positions and linewidths employs scipy.signal.find_peaks and scipy.signal.peak_widths, respectively. In addition, peak features may be added manually by clicking with the left button twice. Unwanted features can be removed with right clicks. If the FWHM of a peak cannot be computed automatically, a dummy FWHM of 1 Hz is assigned automatically. The file <filename>.ivf is written upon pressing the SAVE button. Press Z to activate/deactivate the cursor snap.

Parameters:

- ppm: *1darray*
  PPM scale of the spectrum
- data: *1darray*
  real part of the spectrum to fit
- SW: *float*
  Spectral width /Hz
- SFO1: *float*
  Nucleus Larmor Frequency /MHz
- o1p: *float*
  Carrier position /ppm
- filename: *str*
  Path to the file where to save the initial guess. The .ivf extension is added automatically.

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

Generates a voigt signal on the basis of the passed parameters in the time domain. Then, makes the Fourier transform and returns it.

Parameters:

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

Returns:

- sgn : *1darray*
  generated signal in the frequency domain

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

Make interactive peak_picking. 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

3.5.30 fit.plot_fit(S, ppm_scale, regions, t_AQ, SFO1, o1p, show_total=False, show_res=False, res_offset=0, show_basl=False, X_label='$ delta$ /ppm', labels=None, filename='fit', ext='tiff', dpi=600, dim=None)

Plots either the initial guess or the result of the fit, and saves all the figures. The figure <filename>_full will show the whole model and the whole spectrum. The figures labelled with _R<k> will depict a detail of the fit in the k-th fitting region. Optional labels for the components can be given: in this case, the structure of 'labels' should match the structure of 'regions'. This means that the length of the outer list must be equal to the number of fitting region, and the length of the inner lists must be equal to the number of peaks in that region.

Parameters:

- S: *1darray*
  Spectrum to be fitted
- ppm_scale: *1darray*
  ppm scale of the spectrum
- regions: *dict*
  Generated by fit.read_vf
- t_AQ: *1darray*
  Acquisition timescale
- SFO1: *float*
  Larmor frequency of the observed nucleus, in MHz
- o1p: *float*
  Carrier position, in ppm
- show_total: *bool*
  Show the total trace (i.e. sum of all the components) or not
- show_res: *bool*
  Show the plot of the residuals
- res_offset: *float*
  Displacement of the residuals plot from 0, to be given as a fraction of the height of the experimental spectrum. res_offset > 0 will move the residuals BELOW the zero-line!
- show_basl: *bool*
  If True, displays the baseline on the spectrum and uses it to compute the total trace.
- X_label: *str*
  Text to show as label for the chemical shift axis
- labels: *list of list*
  Optional labels for the components. The structure of this parameter must match the structure of self.result
- filename: *str*
  Root of the name of the figures that will be saved. If None, <self.filename> is used

- ext: *str*
  Format of the saved figures
- dpi: *int*
  Resolution of the figures, in dots per inches
- dim: *tuple*
  Size of the figure in inches (length, height)

3.5.31 fit.plot_fit_P2D(S, ppm_scale, regions, t_AQ, SFO1, o1p, show_total=False, show_res=False, res_offset=0, X_label=, labels=None, filename='fit', ext='tiff', dpi=600)

Plots either the initial guess or the result of the fit, and saves all the figures. A new folder named <filename>_fit will be created. The figure <filename>_full will show the whole model and the whole spectrum. The figures labelled with _R<k> will depict a detail of the fit in the k-th fitting region. Optional labels for the components can be given: in this case, the structure of 'labels' should match the structure of 'regions'. This means that the length of the outer list must be equal to the number of fitting region, and the length of the inner lists must be equal to the number of peaks in that region.

Parameters:

- S: *2darray*
  Spectrum to be fitted
- ppm_scale: *1darray*
  ppm scale of the spectrum
- regions: *list of dict*
  Generated by fit.read_vf_P2D
- 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.
- show_total: *bool*
  Show the total trace (i.e. sum of all the components) or not
- show_res: *bool*
  Show the plot of the residuals
- res_offset: *float*
  Displacement of the residuals plot from 0, to be given as a fraction of the height of the experimental spectrum. res_offset > 0 will move the residuals BELOW the zero-line!
- X_label: *str*
  Text to show as label for the chemical shift axis
- labels: *list of list*
  Optional labels for the components. The structure of this parameter must match the structure of self.result
- filename: *str*
  Root of the name of the figures that will be saved.

- ext: *str*
  Format of the saved figures

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

3.5.32 fit.polyn_basl(y, n=5, method='huber', s=0.2, c_i=None, itermax=1000)

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 either from outside through the parameter c_i or 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
- c_i: *sequence or None*
  Initial guess for the polynomion coefficient. If None, the least-squares fit is used
- itermax: *int*
  Number of maximum iterations

Returns:

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

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

Prints on screen the same thing that 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 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.

3.5.34 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 '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.

3.5.35 fit.read_vf(filename, n=-1)

Reads a .ivf (initial guess) or .fvf (final fit) file, containing the parameters for a lineshape deconvolution fitting procedure. The file is separated and unpacked into a list of dictionaries, each of which contains the limits of the fitting window, the total intensity value, and a dictionary for each peak with the characteristic values to compute it with a Voigt line.

Parameters:

- filename: *str*
  Path to the filename to be read

- n: *int*
  Number of performed fit to be read. Default: last one. The breakpoints are lines that start with '!'. For this reason, n=0 returns an empty dictionary, hence the first fit is n=1.

Returns:

- regions: *list*
  List of dictionaries for running the fit.

3.5.36 fit.read_vf_P2D(filename, n=-1)

Reads a .ivf (initial guess) or .fvf (final fit) file, containing the parameters for a lineshape deconvolution fitting procedure. The file is separated and unpacked into a list of list of dictionaries, each of which contains the limits of the fitting window, and a dictionary for each peak with the characteristic values to compute it with a Voigt line.

Parameters:

- filename: *str*
  Path to the filename to be read

- n: *int*
  Number of performed fit to be read. Default: last one. The breakpoints are lines that start with '!'. For this reason, n=0 returns an empty dictionary, hence the first fit is n=1.

Returns:

- regions: *list of list of dict*
  List of dictionaries for running the fit.

3.5.37 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 g1: non-negativity constraint
- gamma2: *float*
  Weighting factor for function g2: smallest-integral constraint
- gamma3: *float*
  Weighting factor for function g3: smoothing constraint
- e1: *float*
  Tolerance factor for function g1: adjustment for noise
- e2: *float*
  Tolerance factor for function g2: 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

3.5.38 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=0, the same dimension as y is chosen.

Returns:

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

3.5.39 fit.test_correl(data, subtract_mean=True)

Tests an array of residuals for their correlation. It compares the unit-lag autocorrelation P of the data (see below) with the theoretical value for non-correlated data T_P:

$$P = \sum_i \mathbb{r}[i+1]; T_P = (N-1)^(0.5) \sum_i \mathbb{r}[i]^2$$

If $P < T_P$, the residuals are not correlated, and the result is True.

Parameters:

- data: *1darray*
  Residuals to be test
- subtract_mean: *bool*
  If True, subtracts from the residuals their mean.

Returns:

- test: *bool*
  True if the residuals are non correlated, False otherwise

3.5.40 fit.test_ks(data, thresh=0.05)

Performs the Kolmogorov-Smirnov test on the residuals to check if they are drawn from a normal distribution. The implementation is scipy.stats.kstest. The result is True if the residuals are Gaussian.

Parameters:

- data: *1darray*
  Residuals to test
- thresh: *float*
  Significance level for the test. Default is 5

Returns:

- test: *bool*
  True if the residuals are Gaussian, False otherwise

3.5.41 fit.test_randomsign(data, thresh=1.96)

Test an array of residuals for the randomness of the sign changes. The result it True if the sequence is recognized as random.

Parameters:

- data: *1darray*
  Residuals to test
- thresh: *float*
  Significance level. The default is 1.96, which corresponds to 5

Returns:

- test: *bool*
  True if the signs are random, False otherwise

3.5.42 fit.test_residuals(res, alpha=0.05)

Tests an array of residuals for their randomness, correlation, and underlying distribution. To do this, it uses the functions 'fit.test_randomsign', 'fit.test_correl', 'fit.test_ks'. The results of the tests will be print in standard output and returned.

Parameters:

- res: *ndarray*
  Residuals to be tested
- alpha: *float*
  Significance level

Returns:

- test_random: *bool*
  Randomness of the residuals (True = random)
- test_correlation: *bool*
  Correlation of the residuals (True = non-correlated)
- test_gaussian: *bool*
  Normal-distribution of the residuals (True = normally-distributed)

3.5.43 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

3.5.44 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.

3.5.45 fit.voigt_fit_P2D(S, ppm_scale, regions, t_AQ, SFO1, o1p, u_tol=1, f_tol=10, vary_phase=False, vary_b=False, itermax=10000, filename='fit')

Performs a lineshape deconvolution fit on a pseudo-2D experiment using a Voigt model. The initial guess must be read from a .ivf file. All components are treated as independent, regardless from the value of the 'group' attribute. The fitting procedure operates iteratively one window at the time. During the fit routine, the peak positions and lineshapes will be varied consistently on all the experiments; only the intensities are allowed to change in a different way.

Parameters:

- S: *2darray*
  Experimental spectrum
- ppm_scale: *1darray*
  PPM scale of the spectrum
- regions: *dict*
  Generated by fit.read_vf_P2D
- t_AQ: *1darray*
  Acquisition timescale
- SFO1: *float*
  Nucleus Larmor frequency /MHz
- o1p: *float*
  Carrier frequency /ppm
- u_tol: *float*
  Maximum allowed displacement of the chemical shift from the initial value /ppm
- f_tol: *float*
  Maximum allowed displacement of the linewidth from the initial value /ppm
- vary_phase: *bool*
  Allow the peaks to change phase
- vary_b: *bool*
  Allow the peaks to change Lorentzian/Gaussian ratio
- itermax: *int*
  Maximum number of allowed iterations
- filename: *str*
  Name of the file where the fitted values will be saved. The .fvf extension is added automatically

---

3.5.46 fit.voigt_fit_indep(S, ppm_scale, regions, t_AQ, SFO1, o1p, u_lim=1, f_lim=10, k_lim=(0, 3), vary_phase=False, vary_b=True, itermax=10000, fit_tol=1e-08, filename='fit', method='leastsq', basl_fit='n

Performs a lineshape deconvolution fit using a Voigt model. The initial guess must be read from a .ivf file. All components are treated as independent, regardless from the value of the 'group' attribute. The fitting procedure operates iteratively one window at the time.

Parameters:

- S: *1darray*
  Experimental spectrum
- ppm_scale: *1darray*
  PPM scale of the spectrum
- regions: *dict*
  Generated by fit.read_vf
- t_AQ: *1darray*
  Acquisition timescale
- SFO1: *float*
  Nucleus Larmor frequency /MHz
- o1p: *float*
  Carrier frequency /ppm
- u_lim: *float*
  Maximum allowed displacement of the chemical shift from the initial value /ppm
- f_lim: *float*
  Maximum allowed displacement of the linewidth from the initial value /ppm
- k_lim: *float or tuple*
  If tuple, minimum and maximum allowed values for k during the fit. If float, maximum displacement from the initial guess
- vary_phase: *bool*
  Allow the peaks to change phase
- vary_b: *bool*
  Allow the peaks to change Lorentzian/Gaussian ratio
- itermax: *int*
  Maximum number of allowed iterations
- fit_tol: *float*
  Target value to be set for x_tol and f_tol
- filename: *str*
  Name of the file where the fitted values will be saved. The .fvf extension is added automatically
- method: *str or list of str*
  Method to be used for the optimization. See lmfit for details. There is the option to run multiple optimizations in series.

- basl_fit: *str*
  How to address the baseline fit. The options are:
  - 'no': Do not use baseline (default)
  - 'fixed': The baseline is computed once and kept fixed during the optimization
  - 'fit': The baseline coefficients enter as fit parameters during the nonlinear optimization
  - 'calc': The baseline coefficients are calculated during the optimization via linear least-squares optimization

Returns:

- lmfit_results: *list of lmfit.minimizer.MinimizerResult*
  Sequence of the fit results, ordered as the regions dictionary

---

3.5.47 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, it was not used.
- V_f: *2darray*
  Final parameters of the fit
- C_f: *1darray or False*
  Coefficients of the final polynomion used for baseline correction. If False, it was not used.
- 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.

3.5.48 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 the 'baseline' checkbox in the interactive figure panel is not checked, C_f is False.
- limits : *tuple*
  Trim limits for the spectrum (left, right).
- filename: *str*
  directory and name of the file to be written

3.5.49 fit.write_vf(filename, peaks, lims, I, prev=0, header=False, bas_c=None

Write a section in a fit report file, which shows the fitting region and the parameters of the peaks to feed into a Voigt lineshape model.

Parameters:

- filename: *str*
  Path to the file to be written
- peaks: *dict*
  Dictionary of fit.Peak objects
- lims: *tuple*
  (left limit /ppm, right limit /ppm)
- I: *float*
  Absolute intensity value
- prev: *int*
  Number of previous peaks already saved. Increases the peak index
- header: *bool*
  If True, adds a '!' starting line to separate fit trials
- bas_c: *None or 1darray*
  Baseline coefficients

3.5.50 fit.write_vf_P2D(filename, peaks, lims, prev=0)

Write a section in a fit report file, which shows the fitting region and the parameters of the peaks to feed into a Voigt lineshape model.

Parameters:

- filename: *str*
  Path to the file to be written
- peaks: *list of dict*
  list of dictionares of fit.Peak objects, one per experiment
- lims: *tuple*
  (left limit /ppm, right limit /ppm)
- prev: *int*
  Number of previous peaks already saved. Increases the peak index

---

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.

- F: *fit.Voigt_Fit_P2D object*
  Interface for lineshape deconvolution.

Methods:

__init__(self, in_file, pv=False, isexp=True)

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

---

add_noise(self, s_n=1)

Adds noise to the FID, using the function sim.noisegen.

Parameters:

- s_n: *float*
  Standard deviation of the noise

---

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, 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: *float*
  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.

---

mount(self, fids=[], filename=None, newacqus=None)

Replaces the FID of the experiment with a custom one, made by stacking 1D experiments. If the default filename exists (i.e. '<self.filename>.npy'), the function loads it, otherwise calls processing.stack_fids to create it. The 'fid' attribute is overwritten. The key TD1 of the acqus dictionary is updated to match the first dimension of the new FID.

Parameters:

- fids: *sequence of 1darray or Spectrum_1D objects*
  FIDs to be stacked. It can be empty if the .npy file already exists.
- filename: *str or None*
  Path to the filename, without the .npy extension. If it is None, the default filename is used.
- newacqus: *dict*
  New acqus dictionary that replaces the actual one. If it is not a dictionary, no actions are performed.

---

pknl(self)

Reverses the effect of the digital filter by applying a first order phase correction. To be called after having processed the data by 'self.process()'

---

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.

---

scan(self, ns=1, s_n=1)

Simulates the acquisition of ns scans, by adding a different realization of noise at each iteration. The function is supposed to start with the FID without noise at all. If not, the results will be biased.

Parameters:

- ns: *int*
  Number of scans to accumulate
- s_n: *float*
  Standard deviation of the noise

---

to_wav(self, filename=None, cutoff=None, rate=44100)

Converts the FID in an audio file by using misc.data2wav.

Parameters:

- filename: *str*
  Path where to save the file. If None, self.filename is used
- cutoff: *float*
  Clipping limits for the FID
- rate: *int*
  Sampling rate in samples/sec

---

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', 'oxford', 'jeol'

---

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.

---

add_noise(self, s_n=1)

Adds noise to the FID, using the function sim.noisegen.

Parameters:

- s_n: *float*
  Standard deviation of the noise

---

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!

---

blp(self, pred=8, order=8)

Call processing.blp on self.fid for the application of backward linear prediction to the data. Important for Oxford benchtop data, where you have to predict 8 points to have a usable spectrum.

Parameters:

- pred: *int*
  Number of points to be predicted
- order: *int*
  Number of coefficients to be used for the prediction
- N: *int*
  Number of FID points to be used for calculation; used to decrease computation time

---

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

---

pknl(self)

Reverses the effect of the digital filter by applying a first order phase correction. To be called after having processed the data by 'self.process()'

---

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.

---

scan(self, ns=1, s_n=1)

Simulates the acquisition of ns scans, by adding a different realization of noise at each iteration. The function is supposed to start with the FID without noise at all. If not, the results will be biased.

Parameters:

- ns: *int*
  Number of scans to accumulate
- s_n: *float*
  Standard deviation of the noise

---

to_vf(self, filename=None, Hs=None, fvf=True)

Transform a simulated spectrum in a .ivf or .fvf file to be used in a deconvolution procedure. To do this, it reads the peak parameters saved in acqus. The number of signals is determined by acqus['amplitudes']. Multiplets are splitted according to their Js, and saved as components.

Parameters:

- filename: *str or None*
  Path to the filename to be saved, without extension. If None, self.filename is used
- Hs: *int or None*
  Number of nuclei the spectrum integrates for. If None, the sum of the amplitudes is used.
- fvf: *bool*
  If True, adds the '.fvf' extension to the filename, if False, adds '.ivf'

---

to_wav(self, filename=None, cutoff=None, rate=44100)

Converts the FID in an audio file by using misc.data2wav.

Parameters:

- filename: *str*
  Path where to save the file. If None, self.filename is used
- cutoff: *float*
  Clipping limits for the FID
- rate: *int*
  Sampling rate in samples/sec

---

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

---

add_noise(self, s_n=1)

Adds noise to the FID, using the function sim.noisegen.

Parameters:

- s_n: *float*
  Standard deviation of the noise

---

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.

---

pknl(self)

Reverses the effect of the digital filter by applying a first order phase correction. To be called after having processed the data by 'self.process()'

---

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

---

scan(self, ns=1, s_n=1)

Simulates the acquisition of ns scans, by adding a different realization of noise at each iteration. The function is supposed to start with the FID without noise at all. If not, the results will be biased.

Parameters:

- ns: *int*
  Number of scans to accumulate
- s_n: *float*
  Standard deviation of the noise

---

to_wav(self, filename=None, cutoff=None, rate=44100)

Converts the FID in an audio file by using misc.data2wav.

Parameters:

- filename: *str*
  Path where to save the file. If None, self.filename is used
- cutoff: *float*
  Clipping limits for the FID
- rate: *int*
  Sampling rate in samples/sec

---

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.

---

add_noise(self, s_n=1)

Adds noise to the FID, using the function sim.noisegen.

Parameters:

- s_n: *float*
  Standard deviation of the noise

---

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!

---

blp(self, pred=8, order=8)

Call processing.blp on self.fid for the application of backward linear prediction to the data. Important for Oxford benchtop data, where you have to predict 8 points to have a usable spectrum.

Parameters:

- pred: *int*
  Number of points to be predicted
- order: *int*
  Number of coefficients to be used for the prediction
- N: *int*
  Number of FID points to be used for calculation; used to decrease computation time

---

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

---

pknl(self)

Reverses the effect of the digital filter by applying a first order phase correction. To be called after having processed the data by 'self.process()'

---

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.

---

scan(self, ns=1, s_n=1)

Simulates the acquisition of ns scans, by adding a different realization of noise at each iteration. The function is supposed to start with the FID without noise at all. If not, the results will be biased.

Parameters:

- ns: *int*
  Number of scans to accumulate
- s_n: *float*
  Standard deviation of the noise

---

to_vf(self, filename=None, Hs=None, fvf=True)

Transform a simulated spectrum in a .ivf or .fvf file to be used in a deconvolution procedure. To do this, it reads the peak parameters saved in acqus. The number of signals is determined by acqus['amplitudes']. Multiplets are splitted according to their Js, and saved as components.

Parameters:

- filename: *str or None*
  Path to the filename to be saved, without extension. If None, self.filename is used
- Hs: *int or None*
  Number of nuclei the spectrum integrates for. If None, the sum of the amplitudes is used.
- fvf: *bool*
  If True, adds the '.fvf' extension to the filename, if False, adds '.ivf'

---

to_wav(self, filename=None, cutoff=None, rate=44100)

Converts the FID in an audio file by using misc.data2wav.

Parameters:

- filename: *str*
  Path where to save the file. If None, self.filename is used
- cutoff: *float*
  Clipping limits for the FID
- rate: *int*
  Sampling rate in samples/sec

---

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.

---

add_noise(self, s_n=1)

Adds noise to the FID, using the function sim.noisegen.

Parameters:

- s_n: *float*
  Standard deviation of the noise

---

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.

---

pknl(self)

Reverses the effect of the digital filter by applying a first order phase correction. To be called after having processed the data by 'self.process()'

---

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

---

scan(self, ns=1, s_n=1)

Simulates the acquisition of ns scans, by adding a different realization of noise at each iteration. The function is supposed to start with the FID without noise at all. If not, the results will be biased.

Parameters:

- ns: *int*
  Number of scans to accumulate
- s_n: *float*
  Standard deviation of the noise

---

to_wav(self, filename=None, cutoff=None, rate=44100)

Converts the FID in an audio file by using misc.data2wav.

Parameters:

- filename: *str*
  Path where to save the file. If None, self.filename is used
- cutoff: *float*
  Clipping limits for the FID
- rate: *int*
  Sampling rate in samples/sec

---

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.