NMR-EsPy

Release 2.0.0

Simon Hulse & Mohammadali Foroozandeh

Dec 14, 2023

CONTENTS

What is NMR-EsPy?

NMR-EsPy (**N**uclear **M**agnetic **R**esonance **Es**timation in **Py**thon) is a Python package for the estimation of parameters that describe NMR signals.

As with many packages that do number crunching in Python, NMR-EsPy makes heavy use of the NumPy ecosystem.

NMR-EsPy package is a product of work carried out during Simon's Ph.D within the former Foroozandeh group in the University of Oxford's Chemistry Department.

The primary goal behind the NMR-EsPy package is to quantify NMR datasets. It is assumed that raw NMR datasets (called FIDs) comprise a number of sinusoidal oscillators; NMR-EsPy uses numerical techniques in order to estimate the amplitudes, phases, frequencies, and damping rates of these oscillators. Such information can be made use of in various applications.

This version of NMR-EsPy provides the means to estimate both 1D NMR datasets and 2D J-Resolved datasets. Available features include:

- Importing data acquired on Bruker specrometers.
- Simulating data using the Spinach Matlab package.
- Basic pre-processing of NMR data (phasing, baseline correction).
- Estimation of NMR data using numerical optimisation.
- Generating parameter tables.
- Generating result figures.

With 2D-J Resolved datasets, NMR-EsPy can be employed to construct homodecoupled (pure shift) spectra via estimation (a paper outlining this is yet to be released).

In future versions, support for other NMR data types and applications will be included.

Using the Documentation

New to NMR-EsPy? Look at the following:

If you wish to use the API:

1. Read the *installation instructions*. If you are not interested in using the GUI, it is possible to skip the section titled *Installing the GUI to TopSpin*
2. Read the *tutorial on 1D data estimation*.
3. Read *further tutorials* that are relevant to your interests.
4. Whenever a moment arises that you wish to understand more about particular features in the package, consult the *API reference*.

If you wish to use the GUI:

1. Read the *installation instructions*. If you are going to be loading the GUI from within TopSpin, it is important to read the section titled *Installing the GUI to TopSpin*.
2. Read the *instructions on using the GUI*.

In you are comfortable with writing Python code, it is recommended that you make use of NMR-EsPy via the API, rather than the GUI, as it is more feature rich.

Issues

If you come across any unexpected behavior/bugs, please get in touch with Simon, via email (see the email icon in the sidebar), or file an issue. This project is no longer actively worked on, but if there are bugs that prevent use of the package I will try to fix them.

Publications

- Simon G. Hulse, Mohammadali Foroozandeh. Newton meets Ockham: Parameter estimation and model selection of NMR data with NMR-EsPy. *J. Magn. Reson.* 338 (2022) 107173. https://doi.org/10.1016/j.jmr.2022.107173

CHAPTER

ONE

INSTALLATION

Note: On this page, `<pyexe>` denotes the symbolic link/path to the Python executable you are using. **You need to be using Python 3.8 or above.**

NMR-EsPy is available via the Python Package Index. The latest stable version can be installed using:

```
$ <pyexe> -m pip install nmrespy
```

If you want to install the development and documentation-building requirements, use the following command:

```
$ <pyexe> -m pip install nmrespy[dev,docs]
```

NMR-EsPy has the follwoing dependencies which are installed automatically when you run `pip install`.

Package	Version	Details
Numpy	1.19+	Ubiquitous
Scipy	1.5+	Ubiquitous
Matplotlib	3.3+	Required for result plotting, manual phase-correction, and the GUI.
bruker_uti	0.0.1+	Required for loading Bruker data.
pybaselines	1.0.0+	Required for baseline correction.
Colorama	0.4+	Required on Windows only. Enables ANSI escape character sequences to work, allowing coloured terminal output.

1.1 Installing the GUI to TopSpin

NMR-EsPy has an accompanying Graphical User Interface (GUI) which is accessible via both the terminal and Bruker's TopSpin software. The following two sections describe ways to install the scripts for accessing the GUI via TopSpin.

1.1.1 Automatic TopSpin installation

Note: Use this method if you can, it is less error prone!

Enter the following into a terminal:

```
$ <pyexe> -m nmrespy --install-to-topspin
```

Directories matching the following glob pattern will be searched for on your system:

- UNIX: `/opt/topspin*`
- Windows: `C:\Bruker\TopSpin*`

If there are valid directories, you will see a message similar to this:

```
The following TopSpin path(s) were found on your system:
    [1] /opt/topspin4.0.8
For each TopSpin version you would like to install the nmrespy app to,
provide the corresponding numbers, separated by whitespaces.
If you want to cancel the install to TopSpin, enter 0.
If you want to install to all the listed TopSpin installations, press <Return>:
```

In this example, pressing `1` or `<Return>` would install the scripts to TopSpin 4.0.8. Pressing `0` would cancel the operation.

For each specified path, two files will be generated:

- `/<path to>/topspin<x.y.z>/exp/stan/nmr/py/user/espy1d.py`
- `/<path to>/topspin<x.y.z>/exp/stan/nmr/py/user/espy2dj.py`

where `<x.y.z>` is the TopSpin version number.

```
SUCCESS:
    /opt/topspin4.0.8/exp/stan/nmr/py/user/espy1d.py

SUCCESS:
    /opt/topspin4.0.8/exp/stan/nmr/py/user/espy2dj.py
```

If you get the following message instead, you either need to install TopSpin, or manually install the script (see the next section):

```
No TopSpin installations were found on your system! If you don't have
TopSpin, I guess that makes sense. If you do have TopSpin, perhaps it is
installed in a non-default location? You'll have to perform a manual
installation in this case. See the documentation for details.
```

1.1.2 Manual TopSpin installation

If automatic installation failed, perhaps because TopSpin isn't installed in the default location, you can get the TopSpin GUI loader up-and-running with the following steps.

Copying the loader scripts

Load a Python REPL and enter the following to determine where the GUI loading scripts are located:

```
$ <pyexe>
>>> from nmrespy import TOPSPINPATHS as tp
>>> for file in tp:
...     print(file)
...
/home/simon/.venv/lib/python3.9/site-packages/nmrespy/app/topspin_scripts/espy2dj.py
/home/simon/.venv/lib/python3.9/site-packages/nmrespy/app/topspin_scripts/espy1d.py
```

Copy these files to your TopSpin installation.

- UNIX:

  You may need `sudo` depending on where your TopSpin directory is.

  ```
  $ cp /home/simon/.venv/lib/python3.9/site-packages/nmrespy/app/topspin_scripts/espy2dj.
    py \
  > /path/to/.../topspinx.y.z/exp/stan/nmr/py/user/
  $ cp /home/simon/.venv/lib/python3.9/site-packages/nmrespy/app/topspin_scripts/espy1d.py
    \
  > /path/to/.../topspinx.y.z/exp/stan/nmr/py/user/
  ```

- Windows:

  ```
  > copy C:\Users\simon\.venv\Lib\site-packages\nmrespy\app\topspin_scripts\espy2dj.py ^
  More? C:\path\to\...\TopSpinx.y.z\exp\stan\nmr\py\user\
  > copy C:\Users\simon\.venv\Lib\site-packages\nmrespy\app\topspin_scripts\espy1d.py ^
  More? C:\path\to\...\TopSpinx.y.z\exp\stan\nmr\py\user\
  ```

Editing the loader scripts

Now you need to open the newly created files and make some edits to configure path information.

1. Load TopSpin
2. Enter `edpy` in the bottom-left command prompt
3. Select the `user` subdirectory from `Source`

Then do the following things for both `espy1d.py` and `espy2dj.py`:

1. Double click the file
2. You need to set `py_exe` (which is `None` initially) with the path to your Python executable. One way to determine this which should be independent of Operating System is to load a Python REPL and enter the following lines (below is an example on Windows):

```
>>> import sys
>>> exe = sys.executable.replace('\\', '\\\\') # replace is needed for Windows
>>> print(f"\"{exe}\"")
"C:\\Users\\simon\\.venv\\Scripts\\python.exe"
```

You should set `py_exe` as the **EXACT** output you get from this:

```
py_exe = "C:\\Users\\simon\\.venv\\Scripts\\python.exe"
```

3. (Optional) If you have `pdflatex` on your system (see the *LaTeX* section below), and you want to be able to produce PDF result files, you will also have to specify the path to the `pdflatex` executable, given by the variable `pdflatex_exe`, which is set to `None` by default. To find this path, enter the following into a REPL:

   - *UNIX*

   ```
   >>> from subprocess import check_output
   >>> exe = check_output(["which", "pdflatex"])
   >>> exe = str(exe, 'utf-8').rstrip()
   >>> print(f"\"{exe}\"")
   "/usr/bin/pdflatex"
   ```

   - *Windows*

   ```
   >>> from subprocess import check_output
   >>> exe = check_output("where pdflatex", shell=True)
   >>> exe = str(exe, 'utf-8').rstrip().replace("\\", "\\\\")
   >>> print(f"\"{exe}\"")
   "C:\\texlive\\2020\\bin\\win32\\pdflatex.exe"
   ```

   You should set `pdflatex_exe` as the **EXACT** output you get from this:

   ```
   pdflatex_exe = "C:\\texlive\\2020\\bin\\win32\\pdflatex.exe"
   ```

1.2 LaTeX (Optional)

NMR-EsPy provides functionality to save result files to PDF format using LaTeX. The easiest way to get LaTeX is probably to install TexLive.

As a simple check that your system has LaTeX available, the command `pdflatex` should exist. Open a terminal.

Enter the following command:

```
$ pdflatex -v
```

If you see something similar to the following:

```
pdfTeX 3.14159265-2.6-1.40.20 (TeX Live 2019/Debian)
kpathsea version 6.3.1
Copyright 2019 Han The Thanh (pdfTeX) et al.

--snip--
```

things should work fine. If you get an error indicating that `pdflatex` isn't recognised, you probably haven't got LaTeX installed.

The following is a full list of packages that your LaTeX installation will need to successfully compile the `.tex` files generated by this module:

- amsmath
- array
- booktabs
- cmbright
- geometry
- hyperref
- longtable
- nicefrac
- siunitx
- tcolorbox
- varwidth
- xcolor

If you wish to check the packages are available, use `kpsewhich`:

```
$ kpsewhich booktabs.sty
/usr/share/texlive/texmf-dist/tex/latex/booktabs/booktabs.sty
```

If a pathname appears, the package is installed to that path. These packages are pretty ubiquitous, so it is likely that they have been installed already.

1.3 MATLAB and Spinach (Optional)

Spinach is a highly sophisticated library for spin dynamics simulations using MATLAB. NMR-EsPy provides some routines that enable the generation of datasets via Spinach. For this you will need:

- MATLAB installed
- Spinach downloaded, and present in the MATLAB path list (see the installation instructions)
- The MATLAB Engine for Python installed (see the installation instructions)

CHAPTER

TWO

TUTORIALS

In this chapter, tutorials are provided to help you get up-and-running with the main features of the NMR-EsPy API. For a rigorous description of the API, you should consult the *Reference* afterwards.

2.1 Using `Estimator1D`

The `nmrespy.Estimator1D` class is provided for the consideration of 1D NMR data.

2.1.1 Generating an instance

There are a few ways to create a new instance of the estimator depending on the source of the data.

Bruker data

It is possible to load both raw FID data and processed spectral data from Bruker using `new_bruker()`. All that is needed is the path to the dataset:

1. If you wish to import an FID, set the path as `"<path_to_data>/<expno>/"`. There should be an `fid` file and an `acqus` file directly under this directory. The data in the `fid` file will be imported, and the artefact from digital filtering will be removed by a first-order phase shift.

   Note: If you import FID data, there is a high chance that you will need to phase the data, and apply baseline correction before proceeding to run estimation. Look at `phase_data()` and `baseline_correction()`, respectively.

   ```
   >>> import nmrespy as ne
   >>> estimator = ne.Estimator1D.new_bruker("/home/simon/nmr_data/andrographolide/1")
   >>> estimator.phase_data(p0=2.653, p1=-5.686, pivot=13596)
   >>> estimator.baseline_correction()
   ```

2. To import processed data, set the path as `"<path_to_data>/<expno>/pdata/<procno>"`. There should be a `1r` file and a `procs` file directly under this directory. The data in `1r` will be Inverse Fourier Transformed, and the resulting time-domain signal is sliced so that only the first half is retained.

   Note: It can be more convienient to provide processed data, even though the data will be converted to the time-domain for estimation, as you can then rely on TopSpin's automated processing scripts

to phase and baseline correct. However, **you should not apply any window function to the data other than exponential line broadening.**

```
>>> import nmrespy as ne
>>> estimator = ne.Estimator1D.new_bruker("home/simon/nmr_data/andrographolide/1/pdata/1
  ")
>>> # Note there is no need for extra data-processing steps
```

Simulated data from a set of signal parameters

You can create an estimator with synthetic data constructed from known parameters using `new_from_parameters()`. The parameters must be provided as a 2D NumPy array with `params.shape[1] == 4`. Each row should contain a signal's amplitude, phase (rad), frequency (Hz), and damping factor (s^{-1}).

```
>>> import nmrespy as ne
>>> import numpy as np
>>> # Using frequencies of 2,3-Dibromopropanoic acid @ 500MHz
>>> params = np.array([
...     [1., 0., 1864.4, 7.],
...     [1., 0., 1855.8, 7.],
...     [1., 0., 1844.2, 7.],
...     [1., 0., 1835.6, 7.],
...     [1., 0., 1981.4, 7.],
...     [1., 0., 1961.2, 7.],
...     [1., 0., 1958.8, 7.],
...     [1., 0., 1938.6, 7.],
...     [1., 0., 2265.6, 7.],
...     [1., 0., 2257.0, 7.],
...     [1., 0., 2243.0, 7.],
...     [1., 0., 2234.4, 7.],
... ])
>>> sfo = 500.
>>> estimator = ne.Estimator1D.new_from_parameters(
...     params=params,
...     pts=2048,  # FID made with 2048 points
...     sw=1.2 * sfo,  # sweep width set to 1.2 ppm
...     offset=4.1 * sfo,  # transmitter offset set to 4.1 ppm
...     sfo=sfo,  # transmitter frequecy set to 500 MHz
...     snr=40.,  # signal-to-noise ratio of the FID set to 40 dB
... )
```

Note: For the rest of this section, we will be using the estimator created in the above code snippet.

Simulated data from Spinach

Assuming you have installed the *relevant requirements*, you can create an estimator instance with data simulated using Spinach with `new_spinach()`. The spin system is defined by a specification of isotropic chemical shifts and scalar couplings:

- For the chemical shifts, a list of floats is required.
- For J-couplings, a list with 3-element tuples of the form `(spin1, spin2, coupling)` is required. **N.B. the spin indices start at "1" rather than "0"**.

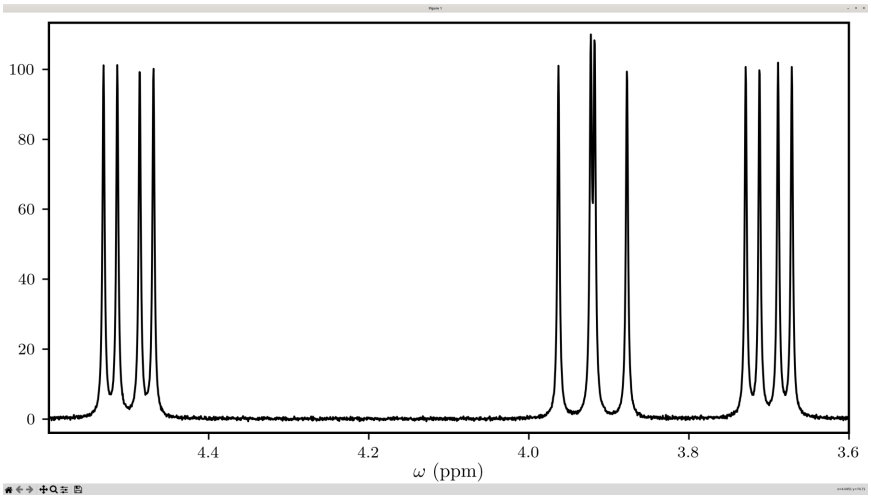
It can take some time to run this function is it involves (a) starting up MATLAB and (b) running a simulation of the experiment.

```
>>> import nmrespy as ne
>>> # 2,3-Dibromopropanoic acid
>>> shifts = [3.7, 3.92, 4.5]
>>> couplings = [(1, 2, -10.1), (1, 3, 4.3), (2, 3, 11.3)]
>>> sfo = 500.
>>> estimator = ne.Estimator1D.new_spinach(
...     shifts=shifts,
...     couplings=couplings,
...     pts=2048,
...     sw=1.2 * sfo,
...     offset=4.1 * sfo,
...     sfo=sfo,
... )
```

2.1.2 Viewing and accessing the dataset

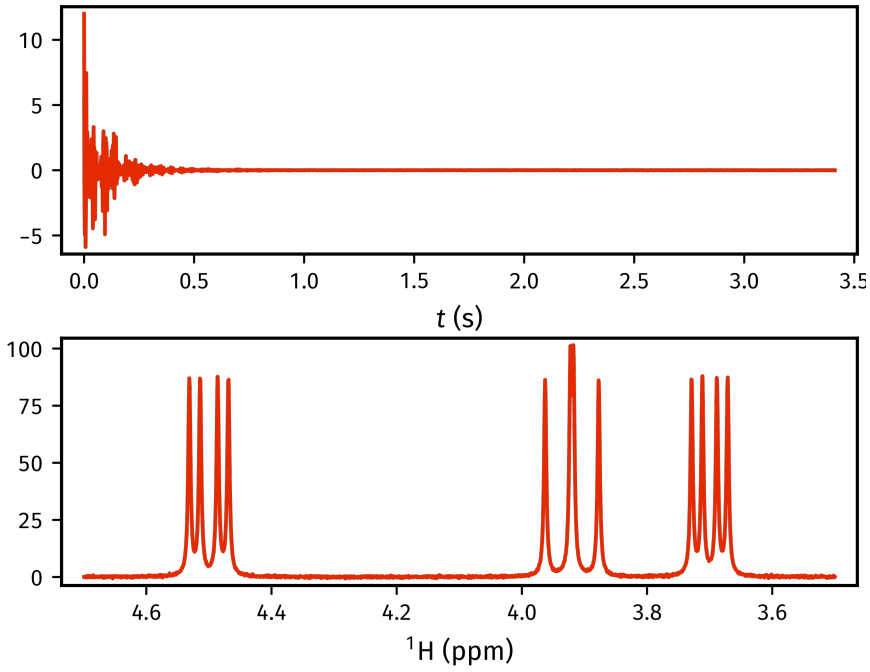
You can inspect the data associated with the estimator with `view_data()`, which loads an interactive matplotlib figure:

```
>>> estimator.view_data(freq_unit="ppm")
```



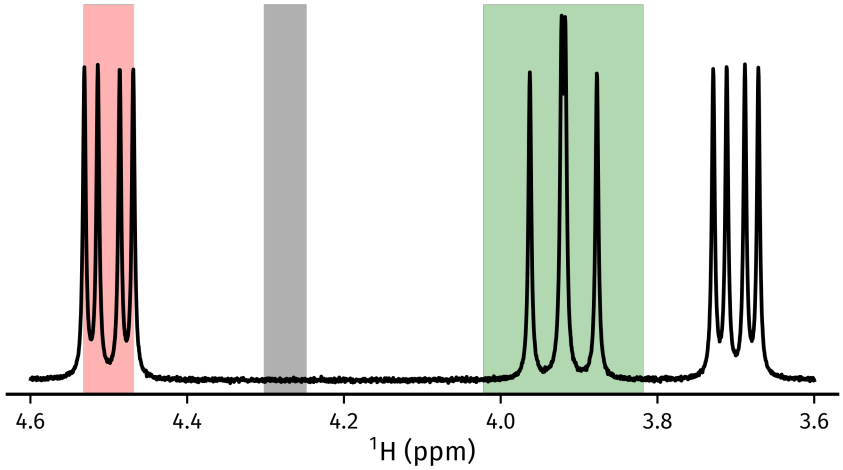
You can access the time-domain data with the `data()` property, and the associated time-points can be retrieved using `get_timepoints()`. The spectral data is accessed with `spectrum()`, and the corresponding chemical shifts with `get_shifts()`.

```
>>> import matplotlib.pyplot as plt
>>> fid = estimator.data
>>> tp = estimator.get_timepoints()[0]
>>> spectrum = estimator.spectrum
>>> shifts = estimator.get_shifts(unit="ppm")[0]
>>> fig, axs = plt.subplots(nrows=2)
>>> axs[0].plot(tp, fid.real)
[<matplotlib.lines.Line2D object at 0x7f2c0e0aa680>]
>>> axs[0].set_xlabel("$t$ (s)")
Text(0.5, 0, '$t$ (s)')
>>> axs[1].plot(shifts, spectrum.real)
[<matplotlib.lines.Line2D object at 0x7f2c0e0aa8c0>]
>>> # Flip x-axis limits (ensure plotting from high to low shifts)
>>> axs[1].set_xlim(reversed(axs[1].get_xlim()))
(4.759384765624999, 3.4400292968749997)
>>> axs[1].set_xlabel("$^1$H (ppm)")
Text(0.5, 0, '$^1$H (ppm)')
>>> # The exact appearence of the generated figure may slightly differ to
>>> # the one below; that'll just be due to customisations I have made to mpl.
>>> plt.show()
```



2.1.3 Estimating the dataset

The generation of parameter estimates is facilitated using the `estimate()` method. In most scenarios, it will not be computationally feasible to estimate the entire FID at once, due to the number of constituent datapoints and signals. For this reason, NMR-EsPy generates frequency-filtered "sub-FIDs" to break the problem down into more manageable chunks. To create suitable sub-FIDs, it is important to select regions in which all signals of interest fully reside within its bounds. As well as this, a region that is devoid of signals (the "noise region") must be indicated. In the figure below, the red region would be inappropriate as the signals clearly do not reside within it fully. The green region is acceptable, as the signals do abide by this. Finally, the grey region is a suitable noise region as it only comprises points in the baseline.



For our dataset, we will estimate three regions, encompassing each multiplet structure in the spectrum. Each region must be given as a tuple of 2 floats, specifying the left and right boundaries of the region of interest (the order of these doesn't matter). By default, these are assumed to be given in Hz, unless `region_unit` is set to `"ppm"`.

```
>>> regions = [(4.6, 4.4), (4.02, 3.82), (3.8, 3.6)]
>>> noise_region = (4.3, 4.25)
>>> for region in regions:
...     estimator.estimate(
...         region=region, noise_region=noise_region, region_unit="ppm",
...     )
...
```

```
ESTIMATING REGION: 4.6 - 4.4 ppm (F1)
```

```
MPM STARTED
```

```
--> Pencil Parameter: 125
--> Hankel data matrix constructed:
    Size:   251 x 126
    Memory: 0.4826MiB
--> Performing Singular Value Decomposition...
--> Computing number of oscillators...
    Number of oscillators will be estimated using MDL
    Number of oscillations: 4
--> Computing signal poles...
--> Computing complex amplitudes...
--> Checking for oscillators with negative damping...
    None found
```

```
MPM COMPLETE
```

```
Time elapsed: 0 mins, 0 secs, 45 msecs
```

```
OPTIMISATION STARTED
```

```
TRUST NCG ALGORITHM STARTED
```

(continues on next page)

(continued from previous page)

```
└──────────────────────────────┘

┌─────────┬──────────────┬──────────────┬─────────────┐
│  Iter.  │  Objective   │  Grad. Norm  │ Trust Radius│
├─────────┼──────────────┼──────────────┼─────────────┤
│  0      │ 0.0106245    │ 0.0653396    │ 1           │
│  6      │ 0.00529026   │ 4.06591e-09  │ 1           │
└─────────┴──────────────┴──────────────┴─────────────┘
Optimiser successfully converged.

┌──────────────────────────────┐
│TRUST NCG ALGORITHM COMPLETE  │
└──────────────────────────────┘
Time elapsed: 0 mins, 0 secs, 34 msecs

┌──────────────────────────────┐
│OPTIMISATION COMPLETE         │
└──────────────────────────────┘
Time elapsed: 0 mins, 0 secs, 101 msecs

┌──────────────────────────────────────────┐
│ESTIMATING REGION: 4.02 - 3.82 ppm (F1)   │
└──────────────────────────────────────────┘

--snip--

┌──────────────────────────────────────────┐
│ESTIMATING REGION: 3.8 - 3.6 ppm (F1)     │
└──────────────────────────────────────────┘

--snip--
```

2.1.4 Inspecting estimation results

Note: Result indices

Each time the `estimate()` method is called, the result is appended to a list containing all the generated results. For many methods that make use of estimation results, an argument called `indices` exists. This lets you specify the results you are interested in. By default (`indices = None`) all results will be used. A subset of the results can be considered by including a list of integers. For example `indices = [0, 2]` would mean only the 1st and 3rd results acquired with the estimator are considered.

A NumPy array of the generated results can be acquired using `get_params()`. The corresponding errors associated with the signal parameters are obtained with `get_errors()`.

```
>>> # All params, frequencies in Hz:
>>> estimator.get_params()
[[ 1.0018e+00  1.5921e-03  1.8356e+03  7.0187e+00]
 [ 1.0003e+00  2.4881e-03  1.8442e+03  6.9968e+00]
 [ 1.0024e+00  1.5817e-03  1.8558e+03  7.0281e+00]
 [ 1.0008e+00  9.1591e-04  1.8644e+03  7.0007e+00]
 [ 1.0022e+00  7.1936e-04  1.9386e+03  7.0109e+00]
 [ 9.9470e-01 -7.4609e-04  1.9588e+03  6.9866e+00]
 [ 1.0080e+00 -1.0112e-03  1.9612e+03  7.0448e+00]
 [ 1.0009e+00 -7.1398e-04  1.9814e+03  7.0131e+00]
```

(continues on next page)

(continued from previous page)

```
 [ 1.0003e+00  1.1306e-03  2.2344e+03  7.0095e+00]
 [ 1.0011e+00  6.0150e-04  2.2430e+03  7.0011e+00]
 [ 9.9902e-01  2.8231e-04  2.2570e+03  6.9856e+00]
 [ 1.0004e+00 -1.8229e-03  2.2656e+03  7.0057e+00]]
>>>
>>> # All errors, frequencies in Hz
>>> estimator.get_errors()
[[0.0013 0.0013 0.0019 0.0121]
 [0.0014 0.0014 0.002  0.0124]
 [0.0014 0.0014 0.002  0.0125]
 [0.0013 0.0013 0.0019 0.012 ]
 [0.0012 0.0012 0.0018 0.0114]
 [0.0036 0.0036 0.0034 0.0212]
 [0.0036 0.0036 0.0034 0.0213]
 [0.0012 0.0012 0.0018 0.0114]
 [0.0013 0.0013 0.0019 0.0116]
 [0.0013 0.0013 0.0019 0.0118]
 [0.0013 0.0013 0.0019 0.0118]
 [0.0013 0.0013 0.0018 0.0116]]
>>>
>>> # Params for first region, frequencies in ppm
>>> estimator.get_params(indices=[0], funit="ppm")
[[ 1.0003e+00  1.1306e-03  4.4688e+00  7.0095e+00]
 [ 1.0011e+00  6.0150e-04  4.4860e+00  7.0011e+00]
 [ 9.9902e-01  2.8231e-04  4.5140e+00  6.9856e+00]
 [ 1.0004e+00 -1.8229e-03  4.5312e+00  7.0057e+00]]
>>>
>>> # Params for second and third regions, split up
>>> estimator.get_params(indices=[1, 2], merge=False, funit="ppm")
[array([[ 1.0022e+00,  7.1936e-04,  3.8772e+00,  7.0109e+00],
       [ 9.9470e-01, -7.4609e-04,  3.9176e+00,  6.9866e+00],
       [ 1.0080e+00, -1.0112e-03,  3.9224e+00,  7.0448e+00],
       [ 1.0009e+00, -7.1398e-04,  3.9628e+00,  7.0131e+00]]),
 array([[1.0018e+00, 1.5921e-03, 3.6712e+00, 7.0187e+00],
       [1.0003e+00, 2.4881e-03, 3.6884e+00, 6.9968e+00],
       [1.0024e+00, 1.5817e-03, 3.7116e+00, 7.0281e+00],
       [1.0008e+00, 9.1591e-04, 3.7288e+00, 7.0007e+00]])]
```

Writing result tables

Tables of parameters can be saved to `.txt` and `.pdf` formats. using `write_result()`. For PDF generation, you will need a working LaTeX installation. See the *installation instructions*.

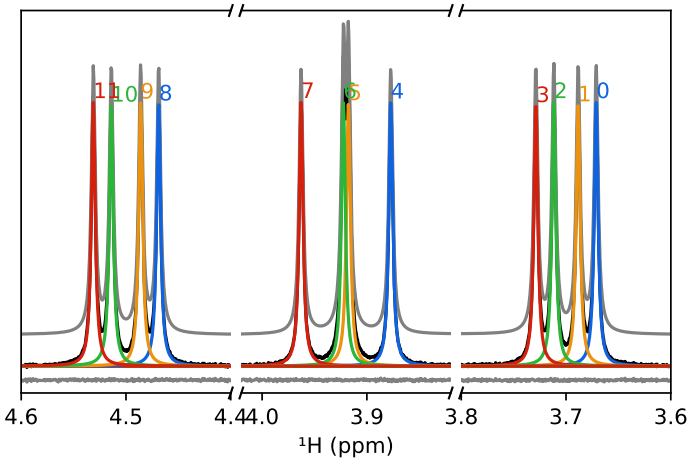
```
>>> for fmt in ("txt", "pdf"):
...     estimator.write_result(
...         path="tutorial_1d",
...         fmt=fmt,
...         description="Simulated 2,3-Dibromopropanoic acid signal.",
...     )
...
Saved file tutorial_1d.txt.
Saved file tutorial_1d.tex.
Saved file tutorial_1d.pdf.
You can view and customise the corresponding TeX file at tutorial_1d.tex.
```

Creating result plots

Figures giving an overview of the estimation result can be generated using `plot_result()`.

```
>>> for (txt, indices) in zip(("complete", "index_1"), (None, [1])):
...     fig, ax = estimator.plot_result(
...         indices=indices,
...         figure_size=(4.5, 3.),
...         region_unit="ppm",
...         axes_left=0.03,
...         axes_right=0.97,
...         axes_top=0.98,
...         axes_bottom=0.09,
...     )
...     fig.savefig(f"tutorial_1d_{txt}_fig.pdf")
...
```

Below is the figure `tutorial_1d_complete_fig.pdf`:



Saving the estimator

The estimator object itself can be saved and reloaded for future use with the `to_pickle()` and `from_pickle()` methods, respectively:

```
>>> estimator.to_pickle("tutorial_1d")
Saved file tutorial_1d.pkl.
>>> # Load the estimator and assign to the `estimator_cp` variable
>>> estimator_cp = ne.Estimator1D.from_pickle("tutorial_1d")
```

Saving a logfile

A logfile listing all the methods called on the estimator can be created using `save_log()`:

```
>>> estimator.save_log("tutorial_1d")
Saved file tutorial_1d.log.
```

2.2 Using `Estimator2DJ`

The `nmrespy.Estimator2DJ` class enables the estimation of 2D J-Resolved (2DJ) spectroscopy datasets. This facilitates use of **CUPID** (**C**omputer-assisted **U**ndiminished-sensitivity **P**rotocol for **I**deal **D**ecoupling) which can be used to generate homodecoupled (*pure shift*) spectra and to predict multiplet structures.

Many methods in this class have analogues in `Estimator1D`. You are advised to read through the *1D tutorial* before continuing, as minimal descriptions will be provided of concepts covered in that.

2.2.1 Generating an instance

Bruker data

Use `new_bruker()`. Unlike `Estimator1D`, you must import time-domain 2DJ data. The path should be set as `"<path_to_data>/<expno>/"`. There should be a `ser` file, an `acqus` file, and an `acqu2s` file directly under this directory. Phasing in the direct-dimension will be needed. If deemed necessary, baseline correction can be performed too.

```
>>> import nmrespy as ne
>>> estimator = ne.Estimator2DJ.new_bruker("/home/simon/nmr_data/qunine/dexamethasone/2")
>>> estimator.phase_data(p0=0.041, p1=-6.383, pivot=1923)
>>> estimator.baseline_correction()
<Estimator2DJ object at 0x7f4b7e1f5cd0>

Experiment Information
──────────────────────

Parameter                    │F1   │F2
─────────────────────────────┼─────┼──────
Nucleus                      │N/A  │¹H
Transmitter Frequency (MHz)  │N/A  │600.18
Sweep Width (Hz)             │50   │7211.5
Sweep Width (ppm)            │N/A  │12.016
Transmitter Offset (Hz)      │0    │2815.4
Transmitter Offset (ppm)     │N/A  │4.691


No estimation performed yet.
```

Simulated data from Spinach

Use `new_spinach()`

```
>>> # Sucrose DFT calculation shifts and couplings
>>> # Note that the dataset generated will not be reminiscent of what
>>> # the actual solution-state dataset for sucrose looks like!
>>> shifts = [
...     6.005, 3.510, 3.934, 3.423, 4.554, 3.891, 4.287, 3.332, 1.908, 1.555, 0.644,
...     4.042, 4.517, 3.889, 4.635, 4.160, 4.021, 4.408, 0.311, 1.334, 0.893, 0.150,
... ]
>>> couplings = [
...     (1, 2, 2.285), (2, 3, 4.657), (2, 8, 4.828), (3, 4, 4.326),
...     (4, 5, 4.851), (5, 6, 5.440), (5, 7, 2.288), (6, 7, -6.210),
...     (7, 11, 7.256), (12, 13, -4.005), (12, 19, 1.460), (14, 15, 4.253),
...     (15, 16, 4.448), (15, 21, 3.221), (16, 18, 4.733), (17, 18, -4.182),
...     (18, 22, 1.350),
... ]
>>> estimator = ne.Estimator2DJ.new_spinach(
...     shifts=shifts,
...     couplings=couplings,
...     pts=(64, 4096),
...     sw=(30., 2200.),
...     offset=1000.,
...     field=300.,
...     field_unit="MHz",
...     snr=20.,
... )
```

Note: We will be using the estimator generated above for the rest of this tutorial. If you do not have access to MATLAB/Spinach, you can construct the estimator by using an FID I made earlier:

```
>>> import nmrespy as ne
>>> from pathlib import Path
>>> import pickle
>>> estimator_path = Path(ne.__file__).expanduser().parents[1] \
...     / "samples/jres_sucrose_sythetic/sucrose_jres_synthetic.pkl"
>>> with open(fid_path, "rb") as fh:
...     estimator = pickle.load(fh)
...
```

2.2.2 Estimating the dataset

The procedure for estimating 2DJ data is very similar to that of 1D data. You need to specify regions in the direct dimension that are of interest for generating filtered sub-FIDs. No filtering is done in the indirect dimension. In our example, it turns out that for a couple of the regions selected, the number of oscillators automatically predicted is slightly smaller that the "true" number, and so the true number has been hard-coded (see the lines involving `initial_guesses`).

```
>>> regions = (
...     (6.08, 5.91), (4.72, 4.46), (4.46, 4.22), (4.22, 4.1), (4.09, 3.98),
...     (3.98, 3.83), (3.58, 3.28), (2.08, 1.16), (1.05, 0.0),
... )
```

(continues on next page)

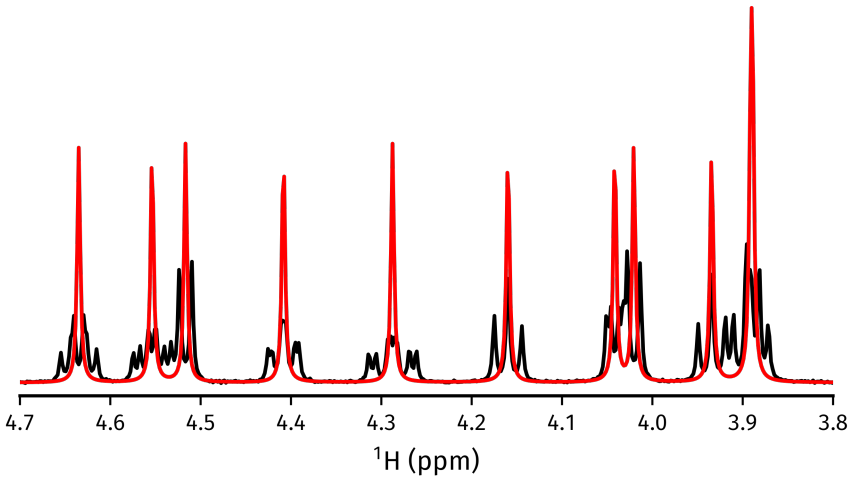
(continued from previous page)

```
>>> n_regions = len(regions)
>>> initial_guesses = n_regions * [None]
>>> initial_guesses[1:3] = [16, 16]
>>> # kwargs common to estimation of each region
>>> common_kwargs = {
...     "noise_region": (5.5, 5.33),
...     "region_unit": "ppm",
...     "max_iterations": 200,
...     "phase_variance": True,
... }
>>> for init_guess, region in zip(initial_guesses, regions):
...     kwargs = {**{"region": region, "initial_guess": init_guess}, **common_kwargs}
...     estimator.estimate(**kwargs)
>>> # It is a good idea to pickle the estimator after estimation
>>> estimator.to_pickle("sucrose")
```

2.2.3 Acquiring a pure shift spectrum

The `cupid_spectrum()` method produces a pure shift spectrum using the 2DJ parameter estimate. In the code snippet below, a figure is made which compares the pure shift spectrum with the spectrum of the first direct-dimension slice in the 2DJ data, i.e. a normal 1D spectrum.

```
>>> # Normal 1D spectrum
>>> init_spectrum = estimator.spectrum_direct.real
>>> # Homodecoupled spectrum produced using CUPID
>>> cupid_spectrum = estimator.cupid_spectrum().real
>>> # Get direct-dimension shifts
>>> shifts = estimator.get_shifts(unit="ppm", meshgrid=False)[-1]
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(figsize=(4.5, 2.5))
>>> ax.plot(shifts, init_spectrum, color="k")
>>> ax.plot(shifts, cupid_spectrum, color="r")
>>> # The most interesting region of the spectrum
>>> ax.set_xlim(4.7, 3.8)
>>> # ========================
>>> # These lines are just for plot aesthetics
>>> for x in ("top", "left", "right"):
>>>     ax.spines[x].set_visible(False)
>>> ax.set_xticks([4.7 - 0.1 * i for i in range(10)])
>>> ax.set_yticks([])
>>> ax.set_position([0.03, 0.175, 0.94, 0.83])
>>> ax.set_xlabel(f"{estimator.unicode_nuclei[1]} (ppm)")
>>> # ========================
>>> fig.savefig("cupid_spectrum.png")
```



2.2.4 Multiplet prediction

Oscillators belonging to the same multiplet can be predicted based on the fact that in a 2DJ FID any pair of signals i, j should satisfy the following:

$$\left|\left(f_i^{(2)} - f_i^{(1)}\right) - \left(f_j^{(2)} - f_j^{(1)}\right)\right| < \epsilon,$$

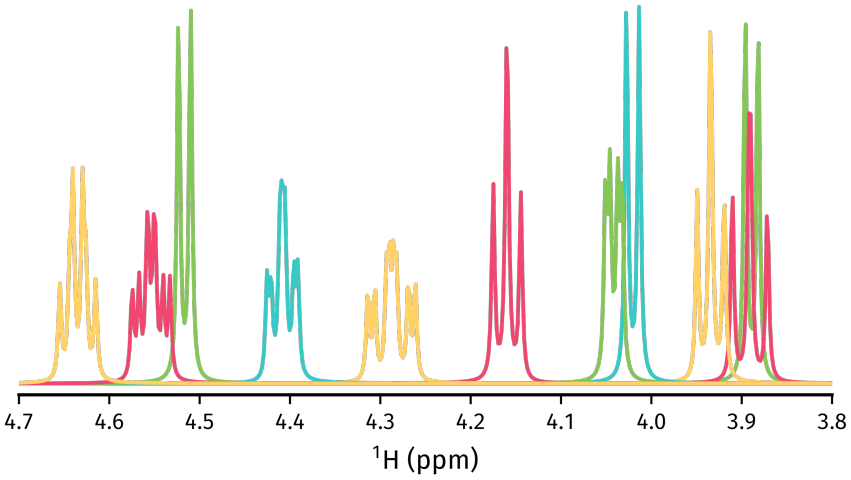
where ϵ is an error threshold, and $f^{(1)}$ and $f^{(2)}$ are the estimated indirect- and direct-dimension frequencies, respectively. `predict_multiplets()` generates groups of oscillator indices satisfying the above criterion. A key parameter for this is `thold`, which sets the error threshold ϵ. By default, this is set to be $\min\left(f_{sw}^{(1)}/N^{(1)}, f_{sw}^{(2)}/N^{(2)}\right)$, i.e whichever is larger out of the indirect- and direct-dimension spectral resolutions. However, especially when considering real data, this threshold can be a little optimistic. For good multiplet groupings, you may need to manually provide a larger threshold.

In the example below, multiplet groups are determined for regions with indices 1-5 (covering the region plotted above).

```
>>> indices = [1, 2, 3, 4, 5]
>>> multiplets = estimator.predict_multiplets(indices=indices)
>>> for (freq, idx) in multiplets.items():
...     print(f"{freq / estimator.sfo[1]:.4f}ppm: {idx}")
...
3.8890ppm: [1, 4]
3.8910ppm: [0, 2, 3, 5]
3.9344ppm: [6, 7, 8]
4.0205ppm: [9, 10]
4.0416ppm: [11, 12, 13, 14]
4.1598ppm: [15, 16, 17]
4.2876ppm: [18, 19, 20, 21, 22, 23, 24, 25]
4.4083ppm: [26, 27, 28, 29, 30, 31, 32, 33]
4.5167ppm: [34, 35]
4.5537ppm: [36, 37, 38, 39, 40, 41, 42, 43]
4.6349ppm: [44, 45, 46, 47, 48, 49]
```

To generate FIDs corresponding to each multiplet structure, use the `construct_multiplet_fids()` method. In the following code snippet, each generated FID undergoes FT, with all the spectra being plotted.

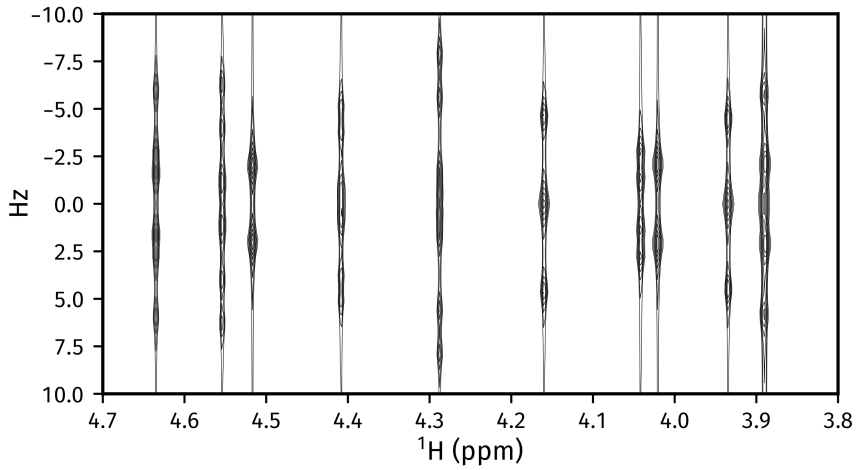
```
>>> # Direct-dimension shifts
>>> shifts_f2 = estimator.get_shifts(unit="ppm", meshgrid=False)[-1]
>>> fids = estimator.construct_multiplet_fids(indices=indices)
>>> # Create an iterator which cycles through values infinitely
>>> from itertools import cycle
>>> colors = cycle(["#84c757", "#ef476f", "#ffd166", "#36c9c6"])
>>> fig, ax = plt.subplots(figsize=(4.5, 2.5))
>>> for fid in fids:
...     # Halve first point prior to FT to prevent vertical baseline shift
...     fid[0] *= 0.5
...     # FT and retrieve real component
...     spectrum = ne.sig.ft(fid).real
...     ax.plot(shifts_f2, spectrum, color=next(colors))
...
>>> ax.set_xlim(4.7, 3.8)
>>> # ========================
>>> # These lines are just for plot aesthetics
>>> for x in ("top", "left", "right"):
...     ax.spines[x].set_visible(False)
...
>>> ax.set_xticks([4.7 - 0.1 * i for i in range(10)])
>>> ax.set_yticks([])
>>> ax.set_position([0.03, 0.175, 0.94, 0.83])
>>> ax.set_xlabel(f"{estimator.unicode_nuclei[1]} (ppm)")
>>> # ========================
>>> fig.savefig("multiplets.png")
```



2.2.5 Generating tilted spectra

The well-known 45° shear (commonly called a tilt) that is applied to 2DJ spectra for orthogonal separation of chemical shifts and scalar couplings effectively maps the frequencies in the direct dimension $f^{(2)}$ to $f^{(2)} - f^{(1)}$. Armed with a parameter estimate of an FID, a synthetic signal with these adjusted frequencies can be constructed. As well as this, generating a pair of phase- or amplitude-modulated FIDs enables the construction of absorption-mode spectra (cf the issues involved in generating nice spectra from hypercomplex 2DJ datasets). Use `nmrespy.Estimator2DJ.sheared_signal()`, with `indirect_modulation` set to either `"amp"` or `"phase"` to generate the desired spectrum. Then, use either `nmrespy.sig.proc_phase_modulated()` or `nmrespy.sig.proc_amp_modulated()` as appropriate to construct the spectrum:

```
>>> # Generate P- and N- type FIDs with "sheared" frequencies
>>> sheared_fid = estimator.sheared_signal(indirect_modulation="phase")
>>> # sheared_fid[0] -> P-type, sheared_fid[1] -> N-type
>>> sheared_fid.shape
(2, 64, 4096)
>>> # Generates 2rr, 2ri, 2ir, 2ii spectra
>>> sheared_spectrum = ne.sig.proc_phase_modulated(sheared_fid)
>>> sheared_spectrum.shape
(4, 64, 4096)
>>> spectrum_2rr = sheared_spectrum[0]
>>> # Note the `meshgrid` kwarg is True here to make 2D shift arrays
>>> shifts_f1, shifts_f2 = estimator.get_shifts(unit="ppm")
>>> fig, ax = plt.subplots(figsize=(4.5, 2.5))
>>> # Contour levels
>>> base, factor, nlevels = 25, 1.3, 10
>>> levels = [base * factor ** i for i in range(nlevels)]
>>> ax.contour(
...     shifts_f2,
...     shifts_f1,
...     spectrum_2rr,
...     colors="k",
...     levels=levels,
...     linewidths=0.3,
... )
>>> ax.set_xlim(4.7, 3.8)
>>> ax.set_ylim(10., -10.)
>>> # ========================
>>> # These lines are just for plot aesthetics
>>> ax.set_xticks([4.7 - 0.1 * i for i in range(10)])
>>> ax.set_position([0.12, 0.175, 0.85, 0.79])
>>> ax.set_xlabel(f"{estimator.unicode_nuclei[1]} (ppm)", labelpad=1)
>>> ax.set_ylabel("Hz", labelpad=1)
>>> # ========================
>>> fig.savefig("sheared_spectrum.png")
```

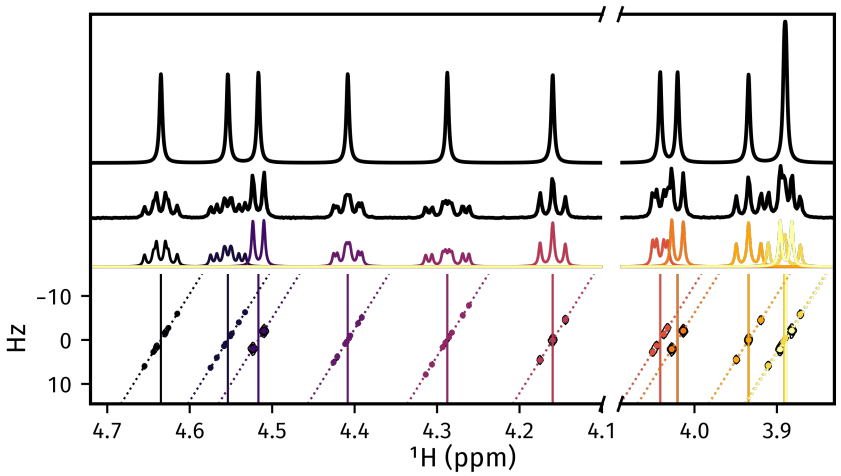


2.2.6 Plotting result figures

The `plot_result()` method enables the generation of a figure which gives an overview of the estimation result. The figure comprises the following, from top to bottom:

- The homodecoupled spectrum generated using `cupid_spectrum()`.
- The 1D spectrum generated from the first direct-dimension FID in the 2DJ dataset.
- The multiplet structures predicted. Note that to get decent multiplet assignments, you may need to increase the value of the `multiplet_thold` argument manually.
- A contour plot of the 2DJ spectrum, with points indicating the positions of estimated peaks.

```
>>> fig, axs = estimator.plot_result(
...     indices=[1, 2, 3, 4, 5],
...     region_unit="ppm",
...     marker_size=5.,
...     figsize=(4.5, 2.5),
...     # There is a lot of scope for editing the colours of
...     # multiplets. See the reference!
...     # Here I specify the name of a colormap in matplotlib.
...     multiplet_colors="inferno",
...     # Argumnets of the position of the plot in the figure
...     axes_left=0.1,
...     axes_bottom=0.15,
... )
>>> fig.savefig("plot_result_2dj.png")
```



2.2.7 Writing data to Bruker format

Note: I am aware that it is currently not possible to analyse the pdata that is generated by NMR-EsPy (you'll get an error along the lines of *"This application requires frequency domain data"* if you try to peak pick, integrate etc). As a workaround for now, you can run a processing script on the FID to generate processed data that is readable by TopSpin. All that is done to get the spectrum from the FID is halve the initial point and FT (there is no apodisation).

Multiplets

To write individual multiplet structures to separate Bruker experiments, you can use `write_multiplets_to_bruker()`. It is a good idea to set a prefix for the experiment numbers, especially if the directory you are saving to already has data directories, so you can easily remember which of the directories correspond to multiplet structures. In the example below, as 22 multiplets were resolved, and `expno_prefix` was set to `99`, the directories `9901/`, `9902/`, ..., `9922/` are created.

```
>>> estimator.write_multiplets_to_bruker(
...     path=".",
...     expno_prefix=99,
...     pts=16384,
...     force_overwrite=True,
... )
Saved multiplets to folders ./[9901-9922]/
```

CUPID data

To save the homodecoupled signal generated by our CUPID method, use `write_cupid_to_bruker()`:

```
>>> estimator.write_cupid_to_bruker(
...     path=".",
...     expno=1111,
...     pts=16384,
... )
Saved CUPID signal to ./1111/
```

2.2.8 Miscellaneous

For writing result tables to text and PDF files, saving estimators to binary files for later use, and saving log files, look at the relevant sections in the *Using Estimator1D* tutorial.

CHAPTER

THREE

USING THE GRAPHICAL USER INTERFACE

A Graphical User Interface (GUI) is available for using NMR-EsPy without the requirement of writing Python scripts. The GUI provides the majority of functionality that you can achieve with the NMR-EsPy API, though for greater flexibility, if you are competent at Python, you might prefer to stick to script writing.

Before an in-depth description of using the GUI, you should note the following general points:

Using entry boxes

In many scenarios you can insert information using an entry box. Most of these require validation before the input you provide is accepted. Once you enter text into an entry box, it will be highlighted in red. Once you have entered the desired value, press `<Enter>`. Once you do this, the value you have provided will be checked to ensure it is valid. If it is, the value will be accepted. Otherwise, the entry box will revert its value back to the previous value. The red highlight will also disappear.

3.1 1D GUI

3.1.1 Loading the 1D GUI

From a terminal

Enter the following:

```
$ <pyexe> -m nmrespy --setup1d /path/to/data
```

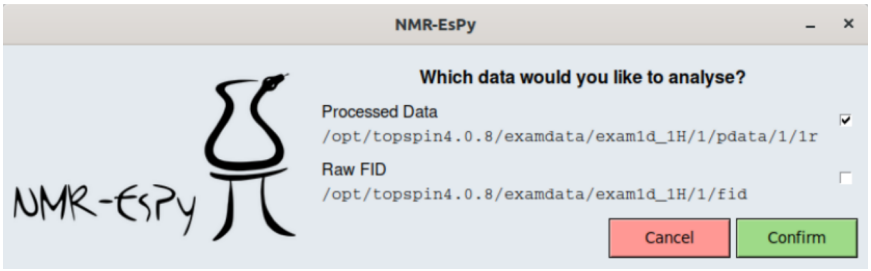
The data path can be of either of the following forms:

- `<root>/<expno>`, where `<expno>` is an experiment number. Directly under this directory should be an `fid` file, as well as an `acqus` file.
- `<root>/<expno>/pdata/<procno>`, where `<procno>` is a processed data number. Directly under this directory should be a `1r` file, as well as a `procs` file.

From TopSpin

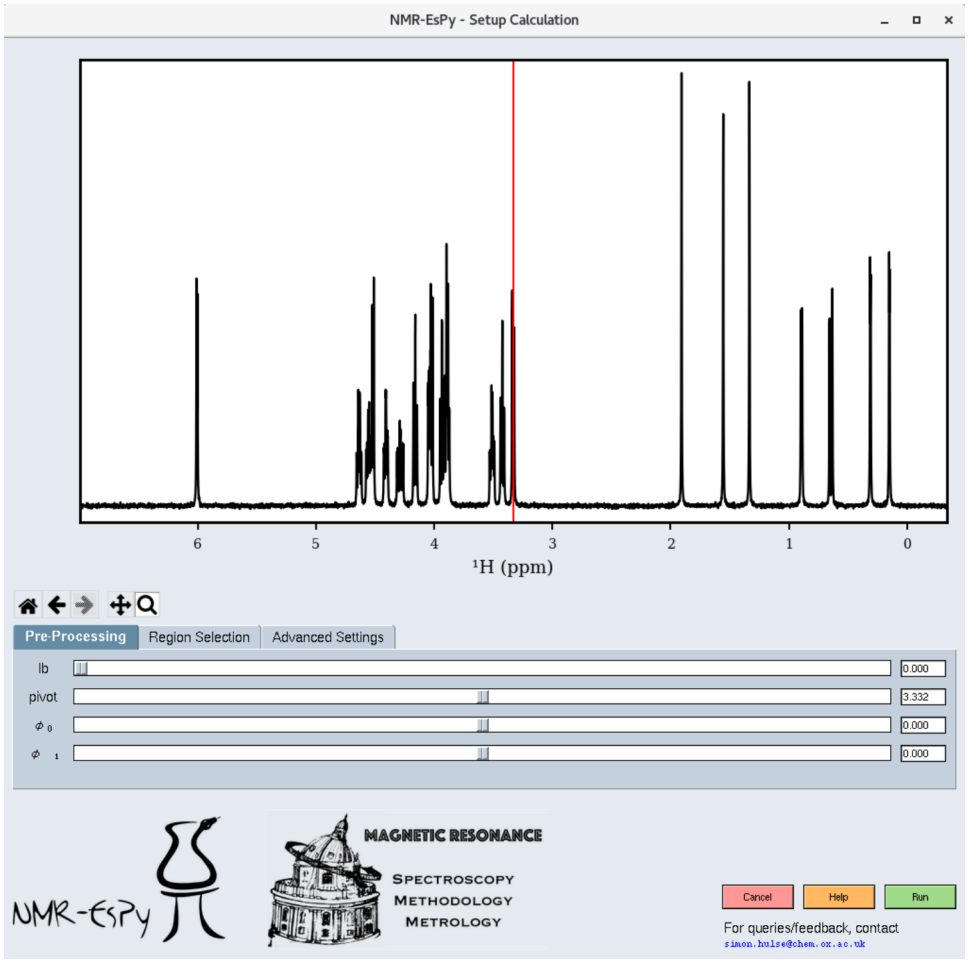
Assuming you have *installed the TopSpin scripts already*, load the 1D dataset of interest. Then, in the bottom left command prompt, enter `espy1d`.

At this point, you will be prompted on whether you would like to import the raw FID data (stored in the `../../fid` file) or the processed data (stored in `1r`).



3.1.2 Setting up a 1D estimation

Assuming the path you have specified contains valid data, a window similar to the following will appear:



You can explore the dataset with the following navigation icons located to the bottom left of the plot:

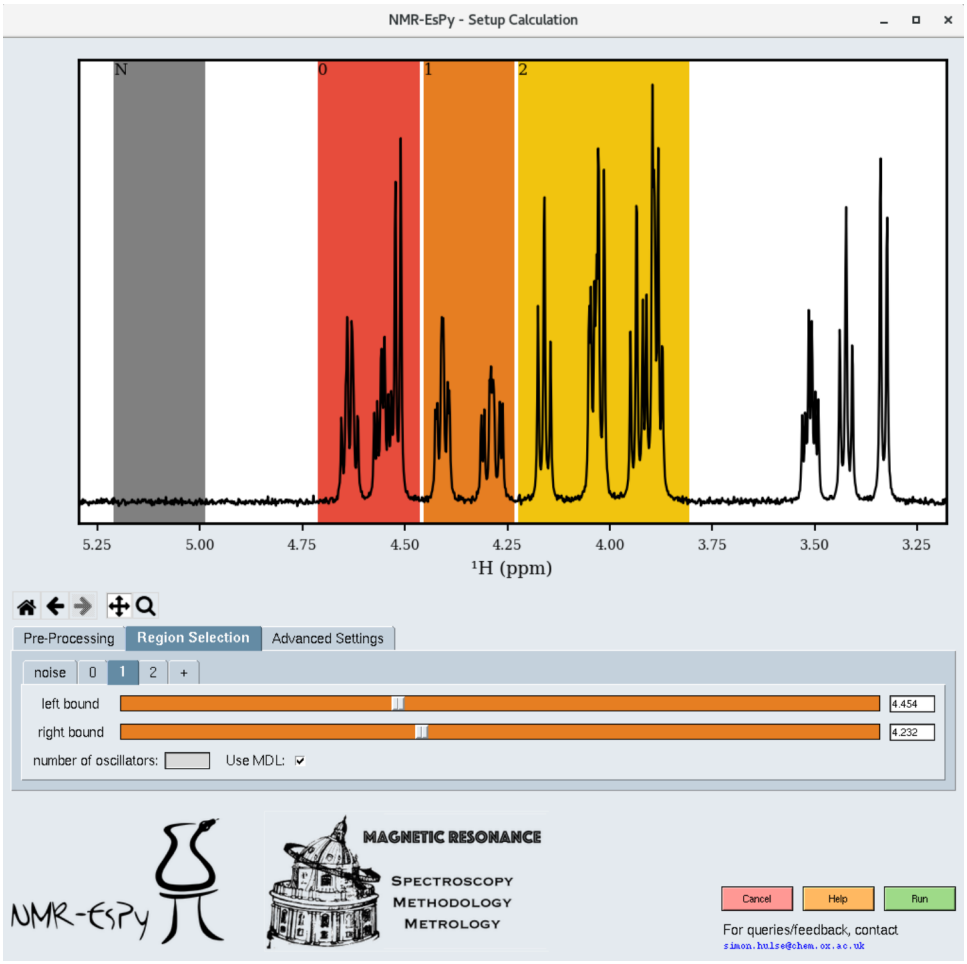
Icon	Role
	Return to the original plot view.
	Return to the previous plot view.
	Undo a return to a previous view
	Pan. Note that panning outside the spectral window is not possible.
	Zoom.

Pre-processing

You will get the best results out of NMR-EsPy is you provide data that has been phase-corrected, and has been baseline corrected. The pre-processing tab allows you to carry out any required phase correction. You are likely to need to do this if you have imported raw FID data. Baseline correction is applied to the data automatically prior to estimation.

Region Selection

When you click on the “Region Selection” tab, you will see a few shaded regions appear in the plot:



- Coloured regions indicate the parts of the data you wish to estimate. The smaller you can keep these regions, the better, in terms of computational cost and result efficacy. However note that the bounds of each region should be at regions of the spectrum which are baseline (see the figure for appropriate examples).
- The grey region (called the noise region) is a special region which should not contain any signals.
- You can adjust each region by selecting the appropriate tab. To add an extra region, press the "+" tab.
- By default, NMR-EsPy uses a technique called the Minimum Description Length to estimate an appropriate number of oscillators to fit to a particular region. If you instead wish to provide a hard-coded value, untick the "Use MDL" checkbox, and insert an integer into the "number of oscillators" box.

Advanced Settings

The "Advanced Settings" tab enables a few customisations to the optimisation routine:

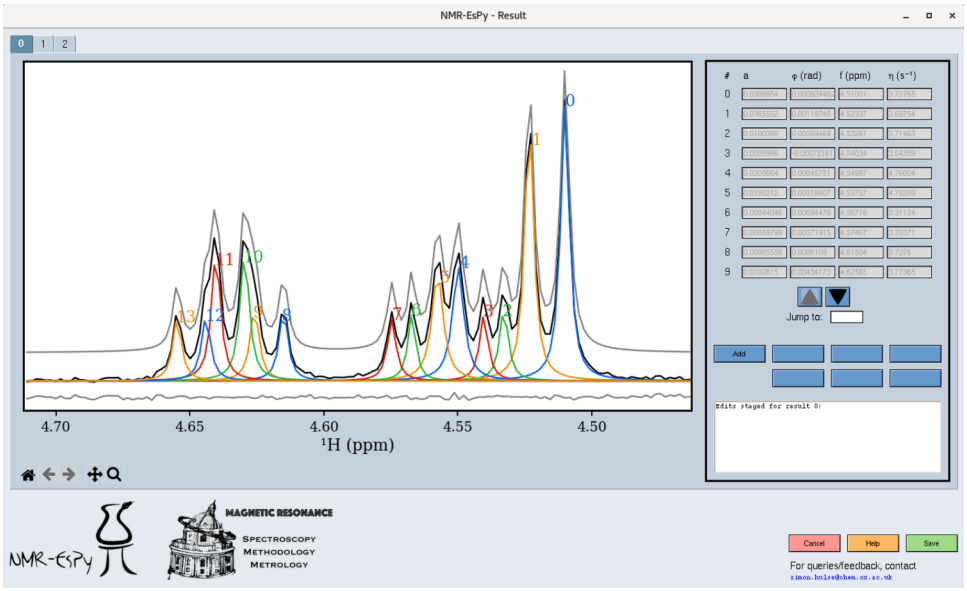
- The optimisation routine can either use the exact Hessian matrix, or a more efficient approximation, based on the Gauss-Newton method (default). This can be adjusted with the "optimisation method" drop-down. In general, we recommend you stick with "Gauss-Newton".
- The "maximum iterations" box allows you to adjust the number of iterations the optimiser is allowed to undertake before it is forced to terminate.
- The "optimise phase variance" checkbox allows you to specify whether you want the variance of oscillator phases to be included in the function to be optimised (fidelity). By default, assuming you have phased the data, you should have this on.

The bottom-right buttons

- After you have set up the estimation routine, click the green "Run" button to execute it.
- The "Help" button loads this documentation.
- The "Cancel" button closes the GUI down.

3.1.3 Reviewing estimation results

Once the estimation routine is complete, you will see a window similar to this one:



- In the top left are numbered tabs. These allow you to inspect each region in turn.
- The plot region shows the oscillators generated by the routine (coloured and numbered) along with the data itself (black). Above and below the plot in grey are the summation of all the oscillators (the model) and the residual, respectively.
- On the right-hand side is a table with all the parameters which make up the oscillators. You can navigate using the up and down arrow buttons, or by specifying an oscillator label to jump to.

Todo: Further discussion to be made here.

CHAPTER

FOUR

REFERENCE

This chapter provides detailed descriptions of classes and functions available in the user-facing API of NMR-EsPy.

The average user is likely to only be concerned with the estimator object they wish to use (either `Estimator1D` or `Estimator2DJ`). Also of potential interest is `nmrespy.sig`, a module of functions for processing signals, including FT/IFT, apodisation, phasing, etc.

4.1 Estimator1D

class nmrespy.**Estimator1D**(*data: numpy.ndarray*, *expinfo: nmrespy.expinfo.ExpInfo*, *datapath: pathlib.Path | None = None*)

Estimator class for 1D data. For a tutorial on the basic functionailty this provides, see *Using Estimator1D*.

Note: To create an instance of `Estimator1D`, you are advised to use one of the following methods if any are appropriate:

- `new_bruker()`
- `new_from_parameters()`
- `new_spinach()`
- `from_pickle()` (re-loads a previously saved estimator).

Parameters

- **data** – The data associated with the binary file in *path*.
- **datapath** – The path to the directory containing the NMR data.
- **expinfo** – Experiment information.

baseline_correction(*min_length: int = 50*) → None

Apply baseline correction to the estimator's data.

The algorithm applied is desribed in[1]. This uses an implementation provided by pybaselines.

[1] Cobas, J., et al. A new general-purpose fully automatic baseline-correction procedure for 1D and 2D NMR data. Journal of Magnetic Resonance, 2006, 183(1), 145-151.

Parameters

min_length – *From the pybaseline docs:* Any region of consecutive baseline points less than `min_length` is considered to be a false positive and all points in the region are converted to peak points. A higher `min_length` ensures less points are falsely assigned as baseline points.

References

property bf: Iterable[float | None] | None

Get the basic frequency (MHz).

For each dimension where `sfo()` is not `None`, this is equivalent to `self.sfo[i] - self.offset()[i]`

property bruker_params: dict | None

Return a dictionary of Bruker parameters.

If the class instance was generated by `new_bruker()`, a dictionary of experiment parameters will be returned. Otherwise, `None` will be returned.

property data: numpy.ndarray

Return the data associated with the estimator.

property data_direct: numpy.ndarray

Generate a 1D FID of the first signal in the direct dimension.

property default_pts: Iterable[int]

Get default points associated with each dimension.

edit_result(*index: int = -1*, *add_oscs: numpy.ndarray | None = None*, *rm_oscs: Iterable[int] | None = None*, *merge_oscs: Iterable[Iterable[int]] | None = None*, *split_oscs: Dict[int, Dict | None] | None = None*, ***estimate_kwargs*) → None

Manipulate an estimation result. After the result has been changed, it is subjected to optimisation.

There are four types of edit that you can make:

- *Add* new oscillators with defined parameters.
- *Remove* oscillators.
- *Merge* multiple oscillators into a single oscillator.
- *Split* an oscillator into many oscillators.

Parameters

- **index** – See *index*.
- **add_oscs** – The parameters of new oscillators to be added. Should be of shape `(n, 2 * (1 + self.dim))`, where `n` is the number of new oscillators to add. Even when one oscillator is being added this should be a 2D array, i.e.
  - 1D data:

```
params = np.array([[a, φ, f, η]])
```

  - 2D data:

```
params = np.array([[a, φ, f₁, f₂, η₁, η₂]])
```

- **rm_oscs** – An iterable of ints for the indices of oscillators to remove from the result.
- **merge_oscs** – An iterable of iterables. Each sub-iterable denotes the indices of oscillators to merge together. For example, `[[0, 2], [6, 7]]` would mean that oscillators 0 and 2 are merged, and oscillators 6 and 7 are merged. A merge involves removing all the oscillators, and creating a new oscillator with the sum of amplitudes, and the average of phases, freqeuncies and damping factors.
- **split_oscs** – A dictionary with ints as keys, denoting the oscillators to split. The values should themselves be dicts, with the following permitted key/value pairs:
  - `"separation"` - An list of length equal to `self.dim`. Indicates the frequency separation of the split oscillators in Hz. If not specified, this will be the spectral resolution in each dimension.
  - `"number"` - An int indicating how many oscillators to split into. If not specified, this will be `2`.
  - `"amp_ratio"` A list of floats with length equal to the number of oscillators to be split into (see `"number"`). Specifies the relative amplitudes of the oscillators. If not specified, the amplitudes will be equal.

  As an example for a 1D estimator:

```
split_oscs = {
    2: {
        "separation": 1.,  # if 1D, don't need a list
    },
    5: {
        "number": 3,
        "amp_ratio": [1., 2., 1.],
    },
}
```

  Here, 2 oscillators will be split.
  - Oscillator 2 will be split into 2 (default) oscillators with equal amplitude (default). These will be separated by 1Hz.
  - Oscillator 5 will be split into 3 oscillators with relative amplitudes 1:2:1. These will be separated by `self.sw()[0] / self.default_pts()[0]` Hz (default).
- **estimate_kwargs** – Keyword arguments to provide to the call to `estimate()`. Note that `"initial_guess"` and `"region_unit"` are set internally and will be ignored if given.

estimate(*region: Iterable[Tuple[float, float]] | None = None*, *noise_region: Iterable[Tuple[float, float]] | None = None*, *region_unit: str = 'hz'*, *initial_guess: numpy.ndarray | int | None = None*, *mode: str = 'apfd'*, *amp_thold: float | None = None*, *phase_variance: bool = True*, *cut_ratio: float | None = 1.1*, *mpm_trim: Iterable[int] | None = None*, *nlp_trim: Iterable[int] | None = None*, *hessian: str = 'gauss-newton'*, *max_iterations: int | None = None*, *negative_amps: str = 'remove'*, *output_mode: int | None = 10*, *save_trajectory: bool = False*, *epsilon: float = 1e-08*, *eta: float = 0.15*, *initial_trust_radius: float = 1.0*, *max_trust_radius: float = 4.0*, *check_neg_amps_every: int = 10*, *_log: bool = True*, ***optimiser_kwargs*)

Estimate a specified region of the signal.

The basic steps that this method carries out are:

- (Optional, but highly advised) Generate a frequency-filtered "sub-FID" corresponding to a specified region of interest.
- (Optional) Generate an initial guess using the Minimum Description Length (MDL)[2] and Matrix Pencil Method (MPM)[3][4][5][6]
- Apply numerical optimisation to determine a final estimate of the signal parameters. The optimisation routine employed is the Trust Newton Conjugate Gradient (NCG) algorithm ([7], Algorithm 7.2).

Parameters

- **region** – The frequency range of interest. Should be of the form `[left, right]` where `left` and `right` are the left and right bounds of the region of interest in Hz or ppm (see `region_unit`). If `None`, the full signal will be considered, though for sufficently large and complex signals it is probable that poor and slow performance will be realised.
- **noise_region** – If `region` is not `None`, this must be of the form `[left, right]` too. This should specify a frequency range where no noticeable signals reside, i.e. only noise exists.
- **region_unit** – One of `"hz"` or `"ppm"` Specifies the units that `region` and `noise_region` have been given as.
- **initial_guess** –
  - If `None`, an initial guess will be generated using the MPM with the MDL being used to estimate the number of oscillators present.
  - If an int, the MPM will be used to compute the initial guess with the value given being the number of oscillators.
  - If a NumPy array, this array will be used as the initial guess.
- **hessian** – Specifies how to construct the Hessian matrix.
  - If `"exact"`, the exact Hessian will be used.
  - If `"gauss-newton"`, the Hessian will be approximated as is done with the Gauss-Newton method. See the *"Derivation from Newton's method"* section of this article.
- **mode** – A string containing a subset of the characters `"a"` (amplitudes), `"p"` (phases), `"f"` (frequencies), and `"d"` (damping factors). Specifies which types

[2] Yingbo Hua and Tapan K Sarkar. "Matrix pencil method for estimating parameters of exponentially damped/undamped sinusoids in noise". In: IEEE Trans. Acoust., Speech, Signal Process. 38.5 (1990), pp. 814–824.

[3] Yung-Ya Lin et al. "A novel detection–estimation scheme for noisy NMR signals: applications to delayed acquisition data". In: J. Magn. Reson. 128.1 (1997), pp. 30–41.

[4] Yingbo Hua. "Estimating two-dimensional frequencies by matrix enhancement and matrix pencil". In: [Proceedings] ICASSP 91: 1991 International Conference on Acoustics, Speech, and Signal Processing. IEEE. 1991, pp. 3073–3076.

[5] Fang-Jiong Chen et al. "Estimation of two-dimensional frequencies using modified matrix pencil method". In: IEEE Trans. Signal Process. 55.2 (2007), pp. 718–724.

[6] M. Wax, T. Kailath, Detection of signals by information theoretic criteria, IEEE Transactions on Acoustics, Speech, and Signal Processing 33 (2) (1985) 387–392.

[7] Jorge Nocedal and Stephen J. Wright. Numerical optimization. 2nd ed. Springer series in operations research. New York: Springer, 2006.

of parameters should be considered for optimisation. In most scenarios, you are likely to want the default value, `"apfd"`.

- **amp_thold** – A value that imposes a threshold for deleting oscillators of negligible ampltiude.
  - If `None`, does nothing.
  - If a float, oscillators with amplitudes satisfying $a_m < a_{\text{thold}} \|a\|_2$ will be removed from the parameter array, where $\|a\|_2$ is the Euclidian norm of the vector of all the oscillator amplitudes. It is advised to set `amp_thold` at least a couple of orders of magnitude below 1.
- **phase_variance** – Whether or not to include the variance of oscillator phases in the cost function. This should be set to `True` in cases where the signal being considered is derived from well-phased data.
- **mpm_trim** – Specifies the maximal size allowed for the filtered signal when undergoing the Matrix Pencil. If `None`, no trimming is applied to the signal. If an int, and the filtered signal has a size greater than `mpm_trim`, this signal will be set as `signal[:mpm_trim]`.
- **nlp_trim** – Specifies the maximal size allowed for the filtered signal when undergoing nonlinear programming. By default (`None`), no trimming is applied to the signal. If an int, and the filtered signal has a size greater than `nlp_trim`, this signal will be set as `signal[:nlp_trim]`.
- **max_iterations** – A value specifiying the number of iterations the routine may run through before it is terminated. If `None`, a default number of maximum iterations is set, based on the the data dimension and the value of `hessian`.
- **negative_amps** – Indicates how to treat oscillators which have gained negative amplitudes during the optimisation.
  - `"remove"` will result in such oscillators being purged from the parameter estimate. The optimisation routine will the be re-run recursively until no oscillators have a negative amplitude.
  - `"flip_phase"` will retain oscillators with negative amplitudes, but the the amplitudes will be multiplied by -1, and a π radians phase shift will be applied.
  - `"ignore"` will do nothing (negative amplitude oscillators will remain).
- **output_mode** – Dictates what information is sent to stdout.
  - If `None`, nothing will be sent.
  - If `0`, only a message on the outcome of the optimisation will be sent.
  - If a positive int `k`, information on the cost function, gradient norm, and trust region radius is sent every kth iteration.
- **save_trajectory** – If `True`, a list of parameters at each iteration will be saved, and accessible via the `trajectory` attribute.

> **Warning:** Not implemented yet!

- **epsilon** – Sets the convergence criterion. Convergence will occur when $\|g_k\|_2 < \epsilon$.

- **eta** – Criterion for accepting an update. An update will be accepted if the ratio of the actual reduction and the predicted reduction is greater than `eta`:

$$\rho_k = \frac{f(x_k) - f(x_k - p_k)}{m_k(0) - m_k(p_k)} > \eta$$

- **initial_trust_radius** – The initial value of the radius of the trust region.
- **max_trust_radius** – The largest permitted radius for the trust region.
- **check_neg_amps_every** – For every iteration that is a multiple of this, negative amplitudes will be checked for and dealt with if found.
- **_log** – Ignore this!

References

exp_apodisation(*k: Iterable[float]*)

Apply an exponential window function to the direct dimnsion of the data.

Parameters

k – Line-broadening factor for each dimension.

property expinfo_direct: nmrespy.expinfo.ExpInfo

Generate a 1D `ExpInfo()` object with parameters related to the direct dimension.

property fn_mode: str | None

Get acquisiton mode in indirect dimensions. If `self.dim == 1`, returns `None`.

classmethod from_pickle(*path: str | pathlib.Path*) → nmrespy.estimators.Estimator

Load a pickled estimator instance.

> **Warning:** *From the Python docs:*
>
> *"The pickle module is not secure. Only unpickle data you trust. It is possible to construct malicious pickle data which will execute arbitrary code during unpickling. Never unpickle data that could have come from an untrusted source, or that could have been tampered with."*
>
> You should only use `from_pickle` on files that you are 100% certain were generated using `to_pickle()`. If you load pickled data from a .pkl file, and the resulting output is not an estimator object, an error will be raised.

Parameters

path – The path to the pickle file. Do not include the `.pkl` suffix.

get_errors(*indices: Iterable[int] | None = None*, *merge: bool = True*, *funit: str = 'hz'*, *sort_by: str = 'f-1'*) → Iterable[numpy.ndarray] | numpy.ndarray | None

Return estimation result errors.

Parameters

- **indices** – see *indices*
- **merge** –
  - If `True`, a single array of all parameters will be returned.

  - If `False`, an iterable of each individual estimation result's parameters will be returned.

- **funit** – The unit to express frequencies in. Must be one of `"hz"` and `"ppm"`.
- **sort_by** – Specifies the parameter by which the oscillators are ordered by.

  Note the errors are re-ordered such that they would agree with the parameters from `get_params()` when given the same `sort_by` argument.

  Should be one of

  - `"a"` for amplitude
  - `"p"` for phase
  - `"f<n>"` for frequency in the `<n>`-th dimension
  - `"d<n>"` for the damping factor in the `<n>`-th dimension.

  By setting `<n>` to `-1`, the final (direct) dimension will be used. For 1D data, `"f"` and `"d"` can be used to specify the frequency or damping factor.

get_log( ) → str

Get the log for the estimator instance.

get_params(*indices: Iterable[int] | None = None*, *merge: bool = True*, *funit: str = 'hz'*, *sort_by: str = 'f-1'*) → Iterable[numpy.ndarray] | numpy.ndarray | None

Return estimation result parameters.

Parameters

- **indices** – see *indices*
- **merge** –
  - If `True`, a single array of all parameters will be returned.
  - If `False`, an iterable of each individual estimation result's parameters will be returned.
- **funit** – The unit to express frequencies in. Must be one of `"hz"` and `"ppm"`.
- **sort_by** – Specifies the parameter by which the oscillators are ordered by.

  Should be one of

  - `"a"` for amplitude
  - `"p"` for phase
  - `"f<n>"` for frequency in the `<n>`-th dimension
  - `"d<n>"` for the damping factor in the `<n>`-th dimension.

  By setting `<n>` to `-1`, the final (direct) dimension will be used. For 1D data, `"f"` and `"d"` can be used to specify the frequency or damping factor.

get_results(*indices: Iterable[int] | None = None*) → Iterable[nmrespy.estimators.Result]

Obtain a subset of the estimation results obtained.

By default, all results are returned, in the order in which they are obtained.

Parameters

indices – see *indices*

get_shifts(*pts: Iterable[int] | None = None*, *unit: str = 'hz'*, *flip: bool = True*, *meshgrid: bool = True*) → Iterable[numpy.ndarray]

Construct chemical shifts which reflect the experiment parameters.

Parameters

- **pts** – The number of points to construct the shifts with in each dimesnion. If `None`, and `self.default_pts` is a tuple of ints, it will be used.
- **unit** – Must be one of `"hz"` or `"ppm"`.
- **flip** – If `True`, the shifts will be returned in descending order, as is conventional in NMR. If `False`, the shifts will be in ascending order.
- **meshgrid** – If time-points are being derived for a N-dimensional signal (N > 1), setting this argument to `True` will return N-dimensional arrays corresponding to all combinations of points in each dimension. If `False`, an iterable of 1D arrays will be returned.

get_timepoints(*pts: Iterable[int] | None = None*, *start_time: Iterable[float | str] | None = None*, *meshgrid: bool = True*) → Iterable[numpy.ndarray]

Construct time-points which reflect the experiment parameters.

Parameters

- **pts** – The number of points to construct the time-points with in each dimesnion. If `None`, and `self.default_pts` is a tuple of ints, it will be used.
- **start_time** – The start time in each dimension. If set to `None`, the initial point in each dimension will be `0.0`. To set non-zero start times, a list of floats or strings can be used.
  - If floats are used, they specify the first value in each dimension in seconds.
  - Strings of the form `f'{N}dt'`, where `N` is an integer, may be used, which indicates a cetain multiple of the dwell time.
- **meshgrid** – If time-points are being derived for a N-dimensional signal (N > 1), setting this argument to `True` will return N-dimensional arrays corresponding to all combinations of points in each dimension. If `False`, an iterable of 1D arrays will be returned.

property latex_nuclei: Iterable[str | None] | None

Get the nuclei associated with each channel with for use in LaTeX.

Examples:

- `"1H"` → `"\\textsuperscript{1}H"`
- `"195Pt"` → `"\\textsuperscript{195}Pt"`

make_fid(*params: numpy.ndarray*, *pts: Iterable[int] | None = None*, *snr: float | None = None*, *decibels: bool = True*, *indirect_modulation: str | None = None*) → numpy.ndarray

Construct an FID from an array of oscillator parameters.

Parameters

- **params** – Parameter array with the following structure:
  - **1-dimensional data:**

```
params = numpy.array([
   [a_1, φ_1, f_1, η_1],
   [a_2, φ_2, f_2, η_2],
   ...,
   [a_m, φ_m, f_m, η_m],
])
```

– **2-dimensional data:**

```
params = numpy.array([
   [a_1, φ_1, f1_1, f2_1, η1_1, η2_1],
   [a_2, φ_2, f1_2, f2_2, η1_2, η2_2],
   ...,
   [a_m, φ_m, f1_m, f2_m, η1_m, η2_m],
])
```

- **pts** – The number of points to construct the time-points with in each dimesnion. If `None`, and `self.default_pts` is a tuple of ints, it will be used.
- **snr** – The signal-to-noise ratio. If *None* then no noise will be added to the FID.
- **decibels** – If `True`, the snr is taken to be in units of decibels. If *False*, it is taken to be simply the ratio of the singal power over the noise power.
- **indirect_modulation** – Acquisition mode in the indirect dimension if the data is 2D. If the data is 1D, this argument is ignored.
  - `None` - hypercomplex dataset:

$$y\left(t_1, t_2\right) = \sum_m a_m e^{\mathrm{i}\phi_m} e^{\left(2\pi \mathrm{i} f_{1,m} - \eta_{1,m}\right)t_1} e^{\left(2\pi \mathrm{i} f_{2,m} - \eta_{2,m}\right)t_2}$$

  - `"amp"` - amplitude modulated pair:

$$y_{\cos}\left(t_1, t_2\right) = \sum_m a_m e^{\mathrm{i}\phi_m} \cos\left(\left(2\pi \mathrm{i} f_{1,m} - \eta_{1,m}\right)t_1\right) e^{\left(2\pi \mathrm{i} f_{2,m} - \eta_{2,m}\right)t_2}$$

$$y_{\sin}\left(t_1, t_2\right) = \sum_m a_m e^{\mathrm{i}\phi_m} \sin\left(\left(2\pi \mathrm{i} f_{1,m} - \eta_{1,m}\right)t_1\right) e^{\left(2\pi \mathrm{i} f_{2,m} - \eta_{2,m}\right)t_2}$$

  - `"phase"` - phase-modulated pair:

$$y_{\mathrm{P}}\left(t_1, t_2\right) = \sum_m a_m e^{\mathrm{i}\phi_m} e^{\left(2\pi \mathrm{i} f_{1,m} - \eta_{1,m}\right)t_1} e^{\left(2\pi \mathrm{i} f_{2,m} - \eta_{2,m}\right)t_2}$$

$$y_{\mathrm{N}}\left(t_1, t_2\right) = \sum_m a_m e^{\mathrm{i}\phi_m} e^{\left(-2\pi \mathrm{i} f_{1,m} - \eta_{1,m}\right)t_1} e^{\left(2\pi \mathrm{i} f_{2,m} - \eta_{2,m}\right)t_2}$$

  `None` will lead to an array of shape `(n1, n2)`. `amp` and `phase` will lead to an array of shape `(2, n1, n2)`, with `fid[0]` and `fid[1]` being the two components of the pair.

See also:

- For converting amplitude-modulated data to spectral data, see `nmrespy.sig.proc_amp_modulated()`

- For converting phase-modulated data to spectral data, see `nmrespy.sig.proc_phase_modulated()`

manual_phase_data(*max_p1: float = 31.41592653589793*) → Tuple[float, float]

Manually phase the data using a Graphical User Interface.

Parameters

max_p1 – The largest permitted first order correction (rad). Set this to a larger value than the default (10π) if you anticipate having to apply a very large first order correction.

Returns

- *p0* – Zero order phase (rad)
- *p1* – First prder phase (rad)

See also:

`phase_data()`

classmethod new_bruker(*directory: str | pathlib.Path*, *convdta: bool = True*) → *nmrespy.estimators.onedim.Estimator1D*

Create a new instance from Bruker-formatted data.

Parameters

- **directory** – Absolute path to data directory.
- **convdta** – If `True` and the data is derived from an `fid` file, removal of the FID's digital filter will be carried out.

Notes

There are certain file paths expected to be found relative to `directory` which contain the data and parameter files. Here is an extensive list of the paths expected to exist, for different data types:

- Raw FID
  - `directory/fid`
  - `directory/acqus`
- Processed data
  - `directory/1r`
  - `directory/../../acqus`
  - `directory/procs`

classmethod new_from_parameters(*params: numpy.ndarray*, *pts: int*, *sw: float*, *offset: float*, *sfo: float = 500.0*, *nucleus: str = '1H'*, *snr: float | None = 20.0*) → *nmrespy.estimators.onedim.Estimator1D*

Generate an estimator instance with sythetic data created from an array of oscillator parameters.

Parameters

- **params** – Parameter array with the following structure:

```
params = numpy.array([
   [a_1, φ_1, f_1, η_1],
   [a_2, φ_2, f_2, η_2],
   ...,
   [a_m, φ_m, f_m, η_m],
])
```

- **pts** – The number of points the signal comprises.
- **sw** – The sweep width of the signal (Hz).
- **offset** – The transmitter offset (Hz).
- **sfo** – The transmitter frequency (MHz).
- **nucleus** – The identity of the nucleus. Should be of the form `"<mass><sym>"` where `<mass>` is the atomic mass and `<sym>` is the element symbol. Examples: `"1H"`, `"13C"`, `"195Pt"`
- **snr** – The signal-to-noise ratio (dB). If `None` then no noise will be added to the FID.

classmethod **new_spinach**(*shifts: Iterable[float]*, *couplings: Optional[Iterable[Tuple(int, int, float)]]*, *pts: int*, *sw: float*, *offset: float = 0.0*, *field: float = 11.74*, *nucleus: str = '1H'*, *snr: float | None = 20.0*, *lb: float = 6.91*) → *Estimator1D*

Create a new instance from a pulse-acquire Spinach simulation.

See *MATLAB and Spinach (Optional)* for requirments to use this method.

Parameters

- **shifts** – A list of tuple of chemical shift values for each spin.
- **couplings** – The scalar couplings present in the spin system. Given `shifts` is of length `n`, couplings should be an iterable with entries of the form `(i1, i2, coupling)`, where `1 <= i1, i2 <= n` are the indices of the two spins involved in the coupling, and `coupling` is the value of the scalar coupling in Hz. `None` will set all spins to be uncoupled.
- **pts** – The number of points the signal comprises.
- **sw** – The sweep width of the signal (Hz).
- **offset** – The transmitter offset (Hz).
- **sfo** – The magnetic field strength (T).
- **nucleus** – The identity of the nucleus. Should be of the form `"<mass><sym>"` where `<mass>` is the atomic mass and `<sym>` is the element symbol. Examples:
  - `"1H"`
  - `"13C"`
  - `"195Pt"`
- **snr** – The signal-to-noise ratio of the resulting signal, in decibels. `None` produces a noiseless signal.
- **lb** – Line broadening (exponential damping) to apply to the signal. The first point will be unaffected by damping, and the final point will be multiplied by `np.exp(-lb)`. The default results in the final point being decreased in value by a factor of roughly 1000.

property nuclei: Iterable[str | None] | None

Get the nuclei associated with each channel.

offset(*unit: str = 'hz'*) → Iterable[float]

Get the transmitter offset frequency.

Parameters

unit – Must be `"hz"` or `"ppm"`.

phase_data(*p0: float = 0.0*, *p1: float = 0.0*, *pivot: int = 0*) → None

Apply a first-order phase correction in the direct dimension.

Parameters

- **p0** – Zero-order phase correction, in radians.
- **p1** – First-order phase correction, in radians.
- **pivot** – Index of the pivot. `0` corresponds to the leftmost point in the spectrum.

See also:

`manual_phase_data()`

plot_result(*indices: Iterable[int] | None = None*, *high_resolution_pts: int | None = None*, *axes_left: float = 0.07*, *axes_right: float = 0.96*, *axes_bottom: float = 0.08*, *axes_top: float = 0.96*, *axes_region_separation: float = 0.05*, *xaxis_unit: str = 'hz'*, *xaxis_label_height: float = 0.02*, *xaxis_ticks: Iterable[Tuple[int, Iterable[float]]] | None = None*, *oscillator_colors: Any = None*, *plot_model: bool = True*, *plot_residual: bool = True*, *model_shift: float | None = None*, *residual_shift: float | None = None*, *label_peaks: bool = True*, *denote_regions: bool = False*, *spectrum_line_kwargs: Dict | None = None*, *oscillator_line_kwargs: Dict | None = None*, *residual_line_kwargs: Dict | None = None*, *model_line_kwargs: Dict | None = None*, *label_kwargs: Dict | None = None*, ***kwargs*) → Tuple[matplotlib.figure.Figure, numpy.ndarray[matplotlib.axes._axes.Axes]]

Generate a figure of the estimation result.

Parameters

- **indices** – See *indices*.
- **high_resolution_pts** – Indicates the number of points used to generate the oscillators and model. Should be greater than or equal to `self.default_pts[0]`. If `None`, `self.default_pts[0]` will be used.
- **axes_left** – The position of the left edge of the axes, in figure coordinates. Should be between `0.` and `1.`.
- **axes_right** – The position of the right edge of the axes, in figure coordinates. Should be between `0.` and `1.`.
- **axes_top** – The position of the top edge of the axes, in figure coordinates. Should be between `0.` and `1.`.
- **axes_bottom** – The position of the bottom edge of the axes, in figure coordinates. Should be between `0.` and `1.`.
- **axes_region_separation** – The extent by which adjacent regions are separated in the figure, in figure coordinates.
- **xaxis_unit** – The unit to express chemical shifts in. Should be `"hz"` or `"ppm"`.

- **xaxis_label_height** – The vertical location of the x-axis label, in figure coordinates. Should be between `0.` and `1.`, though you are likely to want this to be only slightly larger than `0.`.
- **xaxis_ticks** – Specifies custom x-axis ticks for each region, overwriting the default ticks. Should be of the form: `[(i, (a, b, ...)), (j, (c, d, ...)), ...]` where `i` and `j` are ints indicating the region under consideration, and `a-d` are floats indicating the tick values.
- **oscillator_colors** – Describes how to color individual oscillators. See *color cycle* for details.
- **plot_model** –

  Todo: Add description

- **plot_residual** –

  Todo: Add description

- **model_shift** – The vertical displacement of the model relative to the data.
- **residual_shift** – The vertical displacement of the residaul relative to the data.
- **label_peaks** – If True, label peaks according to their index. The parameters of a peak denoted with the label `i` in the figure can be accessed with `self.get_results(indices)[i]`.
- **denote_regions** – If `True`, and there are regions which share a boundary, a vertical line will be plotted to show the boundary.
- **spectrum_line_kwargs** – Keyword arguments for the spectrum line. All keys should be valid arguments for matplotlib.axes.Axes.plot.
- **oscillator_line_kwargs** – Keyword arguments for the oscillator lines. All keys should be valid arguments for matplotlib.axes.Axes.plot. If `"color"` is included, it is ignored (colors are processed based on the `oscillator_colors` argument.
- **residual_line_kwargs** – Keyword arguments for the residual line (if included). All keys should be valid arguments for matplotlib.axes.Axes.plot.
- **model_line_kwargs** – Keyword arguments for the model line (if included). All keys should be valid arguments for matplotlib.axes.Axes.plot.
- **label_kwargs** – Keyword arguments for oscillator labels. All keys should be valid arguments for matplotlib.text.Text If `"color"` is included, it is ignored (colors are procecessed based on the `oscillator_colors` argument. `"horizontalalignment"`, `"ha"`, `"verticalalignment"`, and `"va"` are also ignored, as these are determined internally.
- **kwargs** – Keyword arguments provided to matplotlib.pyplot.figure.

Returns

- *fig* – The result figure. This can be saved to various formats using the savefig method.
- *axs* – A `(1, N)` NumPy array of the axes generated.

save_log(*path: str | pathlib.Path = './espy_logfile'*, *force_overwrite: bool = False*, *fprint: bool = True*) → None

Save the estimator's log.

Parameters

- **path** – The path to save the log to.
- **force_overwrite** – If `path` already exists and `force_overwrite` is set to `False`, the user will be asked to confirm whether they are happy to overwrite the file. If `True`, the file will be overwritten without prompt.
- **fprint** – Specifies whether or not to print infomation to the terminal.

property sfo: Iterable[float | None] | None

Get the transmitter frequency (MHz).

property spectrum: numpy.ndarray

Return the spectrum associated with the estimator.

property spectrum_direct: numpy.ndarray

Generate a 1D spectrum of the first signal in the direct dimension.

subband_estimate(*noise_region: Tuple[float, float]*, *noise_region_unit: str = 'hz'*, *nsubbands: int | None = None*, ***estimate_kwargs*) → None

Perform estiamtion on the entire signal via estimation of frequency-filtered sub-bands.

This method splits the signal up into `nsubbands` equally-sized region and extracts parameters from each region before finally concatenating all the results together.

> **Warning:** This method is a work-in-progress. It is unlikely to produce decent results at the moment! I aim to improve the way that regions are created in the future.

Parameters

- **noise_region** – Specifies a frequency range where no noticeable signals reside, i.e. only noise exists.
- **noise_region_unit** – One of `"hz"` or `"ppm"`. Specifies the units that `noise_region` have been given in.
- **nsubbands** – The number of sub-bands to break the signal into. If `None`, the number will be set as the nearest integer to the data size divided by 500.
- **estimate_kwargs** – Keyword arguments to give to `estimate()`. Note that `region` and `initial_guess` will be ignored.

sw(*unit: str = 'hz'*) → Iterable[float]

Get the sweep width.

Parameters

unit – Must be `"hz"` or `"ppm"`.

to_pickle(*path: str | pathlib.Path | None = None*, *force_overwrite: bool = False*, *fprint: bool = True*) → None

Save the estimator to a byte stream using Python's pickling protocol.

Parameters

- **path** – Path of file to save the byte stream to. Do not include the `'".pkl"` suffix. If `None`, `./estimator_<x>.pkl` will be used, where `<x>` is the first number that doesn't cause a clash with an already existent file.
- **force_overwrite** – Defines behaviour if the specified path already exists:
  - If `False`, the user will be prompted if they are happy overwriting the current file.
  - If `True`, the current file will be overwritten without prompt.
- **fprint** – Specifies whether or not to print infomation to the terminal.

See also:

`from_pickle()`

property unicode_nuclei: Iterable[str | None] | None

Get the nuclei associated with each channel with superscript numbers.

Examples: `"1H"` → `"¹H"`, `"15N"` → `"¹⁵N"`.

view_data(*domain: str = 'freq'*, *components: str = 'real'*, *freq_unit: str = 'hz'*) → None

View the data (FID or spectrum) with an interactive matplotlib plot.

Parameters

- **domain** – Must be `"freq"` or `"time"`.
- **components** – Must be `"real"`, `"imag"`, or `"both"`.
- **freq_unit** – Must be `"hz"` or `"ppm"`. If `domain` is `freq`, this determines which unit to set chemical shifts to.

write_result(*path: pathlib.Path | str = './nmrespy_result'*, *indices: Iterable[int] | None = None*, *fmt: str = 'txt'*, *description: str | None = None*, *sig_figs: int | None = 5*, *sci_lims: Tuple[int, int] | None = (-2, 3)*, *integral_mode: str = 'relative'*, *force_overwrite: bool = False*, *fprint: bool = True*, *pdflatex_exe: str | pathlib.Path | None = None*) → None

Write estimation result tables to a text/PDF file.

Parameters

- **path** – Path to save the result file to.
- **indices** – see *indices*
- **fmt** – Must be one of `"txt"` or `"pdf"`. If you wish to generate a PDF, you must have a LaTeX installation. See *LaTeX (Optional)*.
- **description** – Descriptive text to add to the top of the file.
- **sig_figs** – The number of significant figures to give to parameters. If `None`, the full value will be used. By default this is set to `5`.
- **sci_lims** – Given a value `(-x, y)` with ints `x` and `y`, any parameter `p` with a value which satisfies `p < 10 ** -x` or `p >= 10 ** y` will be expressed in scientific notation. If `None`, scientific notation will never be used.
- **integral_mode** – One of `"relative"` or `"absolute"`.
  - If `"relative"`, the smallest integral will be set to `1`, and all other integrals will be scaled accordingly.
  - If `"absolute"`, the absolute integral will be computed. This should be used if you wish to directly compare different datasets.

- **force_overwrite** – Defines behaviour if the specified path already exists:
  - If `False`, the user will be prompted if they are happy overwriting the current file.
  - If `True`, the current file will be overwritten without prompt.
- **fprint** – Specifies whether or not to print information to the terminal.
- **pdflatex_exe** – The path to the system's `pdflatex` executable.

Note: You are unlikely to need to set this manually. It is primarily present to specify the path to `pdflatex.exe` on Windows when the NMR-EsPy GUI has been loaded from TopSpin.

---

write_to_bruker(*path: str | pathlib.Path*, *indices: Iterable[int] | None = None*, *pts: Iterable[int] | None = None*, *expno: int | None = None*, *procno: int | None = None*, *force_overwrite: bool = False*) → None

Write a signal generated with estimated parameters to Bruker format.

- `<path>/<expno>/` will contain the time-domain data and information (`fid`, `acqus`, ...)
- `<path>/<expno>/pdata/1/` will contain the processed data and information (`pdata`, `procs`, ...)

Note: There is a known problem that the spectral data has timepoints along the x-axis rather than chemical shifts. I will try to figure out why and fix this in due course!

Parameters

- **path** – The path to the root directory to store the data in.
- **indices** – See *indices*.
- **pts** – The number of points to construct the signal from.
- **expno** – The experiment number. If `None`, the smallest int `x` for which the directory `<path>/<x>/` doesn't exist will be used.
- **force_overwrite** – If `False` and the directory `<path>/<expno>/` already exists, the user will be prompted to confirm whether they are happy to overwrite it. If `True`, said directory will be overwritten.

4.2 Estimator2DJ

class nmrespy.**Estimator2DJ**(*data: numpy.ndarray*, *expinfo: nmrespy.expinfo.ExpInfo*, *datapath: pathlib.Path | None = None*)

Estimator class for J-Resolved (2DJ) datasets, enabling use of our CUPID method for Pure Shift spectra. For a tutorial on the basic functionailty this provides, see *Using Estimator2DJ*.

Note: To create an instance of `Estimator2DJ`, you are advised to use one of the following methods if any are appropriate:

- `new_bruker()`

- `increment=i, new_spinach()`
- `from_pickle()` (re-loads a previously saved estimator).

Parameters

- **data** – The data associated with the binary file in *path*.
- **datapath** – The path to the directory containing the NMR data.
- **expinfo** – Experiment information.

baseline_correction(*min_length: int = 50*) → None

Apply baseline correction to the estimator's data.

The algorithm applied is desribed in[1]. This uses an implementation provided by pybaselines.

Parameters

min_length – *From the pybaseline docs:* Any region of consecutive baseline points less than `min_length` is considered to be a false positive and all points in the region are converted to peak points. A higher `min_length` ensures less points are falsely assigned as baseline points.

References

property bf: Iterable[float | None] | None

Get the basic frequency (MHz).

For each dimension where `sfo()` is not `None`, this is equivalent to `self.sfo[i] - self.offset()[i]`

property bruker_params: dict | None

Return a dictionary of Bruker parameters.

If the class instance was generated by `new_bruker()`, a dictionary of experiment parameters will be returned. Otherwise, `None` will be returned.

construct_multiplet_fids(*indices: Iterable[int] | None = None*, *pts: int | None = None*, *thold: float | None = None*, *freq_unit: str = 'hz'*) → Iterable[numpy.ndarray]

Generate a list of FIDs corresponding to each multiplet structure.

Parameters

- **indices** – See *indices*.
- **pts** – The number of points to construct the mutliplets from.
- **thold** – Frequency threshold for multiplet prediction. All oscillators that make up a multiplet are assumed to obey the following expression:

$$f_c - f_t < f^{(2)} - f^{(1)} < f_c + f_t$$

where f_c is the central frequency of the multiplet, and *f_t* is `thold`

- **freq_unit** – Must be `"hz"` or `"ppm"`.

[1] Cobas, J., et al. A new general-purpose fully automatic baseline-correction procedure for 1D and 2D NMR data. Journal of Magnetic Resonance, 2006, 183(1), 145-151.

Return type
List of numpy arrays with each FID.

cupid_signal(*indices: Iterable[int] | None = None*, *pts: int | None = None*, *_log: bool = True*) → numpy.ndarray

Generate the signal $y_{-45^{\circ}}(t)$, where $t \geq 0$:

$$y_{-45^{\circ}}(t) = \sum_{m=1}^{M} a_m \exp\left(i\phi_m\right) \exp\left(\left(2i\pi\left(f_m^{(2)} - f_m^{(1)}\right) - \eta_m^{(2)}\right)t\right)$$

Producing this signal from parameters derived from estimation of a 2DJ dataset should generate an absorption-mode 1D homodecoupled spectrum.

Parameters

- **indices** – See *indices*.
- **pts** – The number of points to construct the signal from. If `None`, `self.default_pts` will be used.

cupid_spectrum(*indices: Iterable[int] | None = None*, *pts: int | None = None*, *_log: bool = True*) → numpy.ndarray

Generate a homodecoupled spectrum according to the CUPID method.

This generates an FID using `cupid_signal()`, halves the first point, and applies FT.

Parameters

- **indices** – See *indices*.
- **pts** – The number of points to construct the signal from. If `None`, `self.default_pts` will be used.
- **_log** – Ignore this!

property data: numpy.ndarray

Return the data associated with the estimator.

property data_direct: numpy.ndarray

Generate a 1D FID of the first signal in the direct dimension.

property default_multiplet_thold: float

The default margin for error when determining oscillators which belong to the same multiplet.

Given by $f_{sw}^{(1)}/2N^{(1)}$ (i.e. half the spetral resolution in the indirect dimension).

property default_pts: Iterable[int]

Get default points associated with each dimension.

edit_result(*index: int = -1*, *add_oscs: numpy.ndarray | None = None*, *rm_oscs: Iterable[int] | None = None*, *merge_oscs: Iterable[Iterable[int]] | None = None*, *split_oscs: Dict[int, Dict | None] | None = None*, *mirror_oscs: Iterable[int] | None = None*, ***estimate_kwargs*) → None

Manipulate an estimation result. After the result has been changed, it is subjected to optimisation.

There are five types of edit that you can make:

- *Add* new oscillators with defined parameters.
- *Remove* oscillators.

- *Merge* multiple oscillators into a single oscillator.
- *Split* an oscillator into many oscillators.
- **Unique to 2DJ**: *Mirror* an oscillator. This allows you add a new oscillator with the same parameters as an osciallator in the result, except with the following frequencies:

$$f^{(1)}_{new} = -f^{(1)}_{old}$$

$$f^{(2)}_{new} = f^{(2)}_{old} - f^{(1)}_{old}$$

Parameters

- **index** – See *index*.
- **add_oscs** – The parameters of new oscillators to be added. Should be of shape `(n, 2 * (1 + self.dim))`, where `n` is the number of new oscillators to add. Even when one oscillator is being added this should be a 2D array, i.e.
  - 1D data:

    ```
    params = np.array([[a, φ, f, η]])
    ```

  - 2D data:

    ```
    params = np.array([[a, φ, f₁, f₂, η₁, η₂]])
    ```

- **rm_oscs** – An iterable of ints for the indices of oscillators to remove from the result.
- **merge_oscs** – An iterable of iterables. Each sub-iterable denotes the indices of oscillators to merge together. For example, `[[0, 2], [6, 7]]` would mean that oscillators 0 and 2 are merged, and oscillators 6 and 7 are merged. A merge involves removing all the oscillators, and creating a new oscillator with the sum of amplitudes, and the average of phases, freqeuncies and damping factors.
- **split_oscs** – A dictionary with ints as keys, denoting the oscillators to split. The values should themselves be dicts, with the following permitted key/value pairs:
  - `"separation"` - An list of length equal to `self.dim`. Indicates the frequency separation of the split oscillators in Hz. If not specified, this will be the spectral resolution in each dimension.
  - `"number"` - An int indicating how many oscillators to split into. If not specified, this will be `2`.
  - `"amp_ratio"` A list of floats with length equal to the number of oscillators to be split into (see `"number"`). Specifies the relative amplitudes of the oscillators. If not specified, the amplitudes will be equal.

  As an example for a 1D estimator:

```
split_oscs = {
    2: {
        "separation": 1.,  # if 1D, don't need a list
    },
    5: {
```

(continues on next page)

(continued from previous page)

```
        "number": 3,
        "amp_ratio": [1., 2., 1.],
    },
}
```

Here, 2 oscillators will be split.

- Oscillator 2 will be split into 2 (default) oscillators with equal amplitude (default). These will be separated by 1Hz.
- Oscillator 5 will be split into 3 oscillators with relative amplitudes 1:2:1. These will be separated by `self.sw()[0] / self.default_pts()[0]` Hz (default).

- **`mirror_oscs`** – An interable of oscillators to mirror (see the description above).
- **`estimate_kwargs`** – Keyword arguments to provide to the call to `estimate()`. Note that `"initial_guess"` and `"region_unit"` are set internally and will be ignored if given.

`estimate`(*region: Iterable[Tuple[float, float]] | None = None*, *noise_region: Iterable[Tuple[float, float]] | None = None*, *region_unit: str = 'hz'*, *initial_guess: numpy.ndarray | int | None = None*, *mode: str = 'apfd'*, *amp_thold: float | None = None*, *phase_variance: bool = True*, *cut_ratio: float | None = 1.1*, *mpm_trim: Iterable[int] | None = None*, *nlp_trim: Iterable[int] | None = None*, *hessian: str = 'gauss-newton'*, *max_iterations: int | None = None*, *negative_amps: str = 'remove'*, *output_mode: int | None = 10*, *save_trajectory: bool = False*, *epsilon: float = 1e-08*, *eta: float = 0.15*, *initial_trust_radius: float = 1.0*, *max_trust_radius: float = 4.0*, *check_neg_amps_every: int = 10*, *_log: bool = True*, ***optimiser_kwargs*)

Estimate a specified region of the signal.

The basic steps that this method carries out are:

- (Optional, but highly advised) Generate a frequency-filtered "sub-FID" corresponding to a specified region of interest.
- (Optional) Generate an initial guess using the Minimum Description Length (MDL)[2] and Matrix Pencil Method (MPM)[3][4][5][6]
- Apply numerical optimisation to determine a final estimate of the signal parameters. The optimisation routine employed is the Trust Newton Conjugate Gradient (NCG) algorithm ([7], Algorithm 7.2).

Parameters

- **`region`** – The frequency range of interest. Should be of the form `[left, right]` where `left` and `right` are the left and right bounds of the region of interest in Hz or ppm (see `region_unit`). If `None`, the full signal will be considered,

[2] Yingbo Hua and Tapan K Sarkar. "Matrix pencil method for estimating parameters of exponentially damped/undamped sinusoids in noise". In: IEEE Trans. Acoust., Speech, Signal Process. 38.5 (1990), pp. 814–824.

[3] Yung-Ya Lin et al. "A novel detection–estimation scheme for noisy NMR signals: applications to delayed acquisition data". In: J. Magn. Reson. 128.1 (1997), pp. 30–41.

[4] Yingbo Hua. "Estimating two-dimensional frequencies by matrix enhancement and matrix pencil". In: [Proceedings] ICASSP 91: 1991 International Conference on Acoustics, Speech, and Signal Processing. IEEE. 1991, pp. 3073–3076.

[5] Fang-Jiong Chen et al. "Estimation of two-dimensional frequencies using modified matrix pencil method". In: IEEE Trans. Signal Process. 55.2 (2007), pp. 718–724.

[6] M. Wax, T. Kailath, Detection of signals by information theoretic criteria, IEEE Transactions on Acoustics, Speech, and Signal Processing 33 (2) (1985) 387–392.

[7] Jorge Nocedal and Stephen J. Wright. Numerical optimization. 2nd ed. Springer series in operations research. New York: Springer, 2006.

though for sufficently large and complex signals it is probable that poor and slow performance will be realised.

- **noise_region** – If `region` is not `None`, this must be of the form `[left, right]` too. This should specify a frequency range where no noticeable signals reside, i.e. only noise exists.
- **region_unit** – One of `"hz"` or `"ppm"` Specifies the units that `region` and `noise_region` have been given as.
- **initial_guess** –
  - If `None`, an initial guess will be generated using the MPM with the MDL being used to estimate the number of oscillators present.
  - If an int, the MPM will be used to compute the initial guess with the value given being the number of oscillators.
  - If a NumPy array, this array will be used as the initial guess.
- **hessian** – Specifies how to construct the Hessian matrix.
  - If `"exact"`, the exact Hessian will be used.
  - If `"gauss-newton"`, the Hessian will be approximated as is done with the Gauss-Newton method. See the *"Derivation from Newton's method"* section of this article.
- **mode** – A string containing a subset of the characters `"a"` (amplitudes), `"p"` (phases), `"f"` (frequencies), and `"d"` (damping factors). Specifies which types of parameters should be considered for optimisation. In most scenarios, you are likely to want the default value, `"apfd"`.
- **amp_thold** – A value that imposes a threshold for deleting oscillators of negligible ampltiude.
  - If `None`, does nothing.
  - If a float, oscillators with amplitudes satisfying $a_m < a_{\text{thold}} \|a\|_2$ will be removed from the parameter array, where $\|a\|_2$ is the Euclidian norm of the vector of all the oscillator amplitudes. It is advised to set `amp_thold` at least a couple of orders of magnitude below 1.
- **phase_variance** – Whether or not to include the variance of oscillator phases in the cost function. This should be set to `True` in cases where the signal being considered is derived from well-phased data.
- **mpm_trim** – Specifies the maximal size allowed for the filtered signal when undergoing the Matrix Pencil. If `None`, no trimming is applied to the signal. If an int, and the filtered signal has a size greater than `mpm_trim`, this signal will be set as `signal[:mpm_trim]`.
- **nlp_trim** – Specifies the maximal size allowed for the filtered signal when undergoing nonlinear programming. By default (`None`), no trimming is applied to the signal. If an int, and the filtered signal has a size greater than `nlp_trim`, this signal will be set as `signal[:nlp_trim]`.
- **max_iterations** – A value specifiying the number of iterations the routine may run through before it is terminated. If `None`, a default number of maximum iterations is set, based on the the data dimension and the value of `hessian`.
- **negative_amps** – Indicates how to treat oscillators which have gained negative amplitudes during the optimisation.

  - `"remove"` will result in such oscillators being purged from the parameter estimate. The optimisation routine will the be re-run recursively until no oscillators have a negative amplitude.
  - `"flip_phase"` will retain oscillators with negative amplitudes, but the the amplitudes will be multiplied by -1, and a π radians phase shift will be applied.
  - `"ignore"` will do nothing (negative amplitude oscillators will remain).
- **`output_mode`** – Dictates what information is sent to stdout.
  - If `None`, nothing will be sent.
  - If `0`, only a message on the outcome of the optimisation will be sent.
  - If a positive int `k`, information on the cost function, gradient norm, and trust region radius is sent every kth iteration.
- **`save_trajectory`** – If `True`, a list of parameters at each iteration will be saved, and accessible via the `trajectory` attribute.

  Warning: Not implemented yet!

- **`epsilon`** – Sets the convergence criterion. Convergence will occur when $\|g_k\|_2 < \epsilon$.
- **`eta`** – Criterion for accepting an update. An update will be accepted if the ratio of the actual reduction and the predicted reduction is greater than `eta`:

$$\rho_k = \frac{f(x_k) - f(x_k - p_k)}{m_k(0) - m_k(p_k)} > \eta$$

- **`initial_trust_radius`** – The initial value of the radius of the trust region.
- **`max_trust_radius`** – The largest permitted radius for the trust region.
- **`check_neg_amps_every`** – For every iteration that is a multiple of this, negative amplitudes will be checked for and dealt with if found.
- **`_log`** – Ignore this!

References

`exp_apodisation`(*k: Iterable[float]*)

Apply an exponential window function to the direct dimnsion of the data.

Parameters

k – Line-broadening factor for each dimension.

`property expinfo_direct: nmrespy.expinfo.ExpInfo`

Generate a 1D `ExpInfo()` object with parameters related to the direct dimension.

`property fn_mode: str | None`

Get acquisiton mode in indirect dimensions. If `self.dim == 1`, returns `None`.

classmethod from_pickle(*path: str | pathlib.Path*) → nmrespy.estimators.Estimator

Load a pickled estimator instance.

> **Warning:** *From the Python docs:*
>
> *"The pickle module is not secure. Only unpickle data you trust. It is possible to construct malicious pickle data which will execute arbitrary code during unpickling. Never unpickle data that could have come from an untrusted source, or that could have been tampered with."*
>
> You should only use `from_pickle` on files that you are 100% certain were generated using `to_pickle()`. If you load pickled data from a .pkl file, and the resulting output is not an estimator object, an error will be raised.

Parameters

path – The path to the pickle file. Do not include the `.pkl` suffix.

get_errors(*indices: Iterable[int] | None = None*, *merge: bool = True*, *funit: str = 'hz'*, *sort_by: str = 'f-1'*) → Iterable[numpy.ndarray] | numpy.ndarray | None

Return estimation result errors.

Parameters

- **indices** – see *indices*
- **merge** –
  - If `True`, a single array of all parameters will be returned.
  - If `False`, an iterable of each individual estimation result's parameters will be returned.
- **funit** – The unit to express frequencies in. Must be one of `"hz"` and `"ppm"`.
- **sort_by** – Specifies the parameter by which the oscillators are ordered by.

  Note the errors are re-ordered such that they would agree with the parameters from `get_params()` when given the same `sort_by` argument.

  Should be one of
  - `"a"` for amplitude
  - `"p"` for phase
  - `"f<n>"` for frequency in the `<n>`-th dimension
  - `"d<n>"` for the damping factor in the `<n>`-th dimension.

  By setting `<n>` to `-1`, the final (direct) dimension will be used. For 1D data, `"f"` and `"d"` can be used to specify the frequency or damping factor.

get_log() → str

Get the log for the estimator instance.

get_multiplet_integrals(*scale: bool = True*, ***kwargs*) → Dict[float, float]

Get integrals of multiplets assigned using `predict_multiplets()`.

Parameters

- **scale** – If `True`, the integrals are scaled so that the smallest integral is 1.

- **kwargs** – Keyword arguments for `predict_multiplet_integrals()`.

get_params(*indices: Iterable[int] | None = None*, *merge: bool = True*, *funit: str = 'hz'*, *sort_by: str = 'f-1'*) → Iterable[numpy.ndarray] | numpy.ndarray | None

Return estimation result parameters.

Parameters

- **indices** – see *indices*
- **merge** –
  - If `True`, a single array of all parameters will be returned.
  - If `False`, an iterable of each individual estimation result's parameters will be returned.
- **funit** – The unit to express frequencies in. Must be one of `"hz"` and `"ppm"`.
- **sort_by** – Specifies the parameter by which the oscillators are ordered by.

  Should be one of
  - `"a"` for amplitude
  - `"p"` for phase
  - `"f<n>"` for frequency in the `<n>`-th dimension
  - `"d<n>"` for the damping factor in the `<n>`-th dimension.

  By setting `<n>` to `-1`, the final (direct) dimension will be used. For 1D data, `"f"` and `"d"` can be used to specify the frequency or damping factor.

get_results(*indices: Iterable[int] | None = None*) → Iterable[nmrespy.estimators.Result]

Obtain a subset of the estimation results obtained.

By default, all results are returned, in the order in which they are obtained.

Parameters

indices – see *indices*

get_shifts(*pts: Iterable[int] | None = None*, *unit: str = 'hz'*, *flip: bool = True*, *meshgrid: bool = True*) → Iterable[numpy.ndarray]

Construct chemical shifts which reflect the experiment parameters.

Parameters

- **pts** – The number of points to construct the shifts with in each dimesnion. If `None`, and `self.default_pts` is a tuple of ints, it will be used.
- **unit** – Must be one of `"hz"` or `"ppm"`.
- **flip** – If `True`, the shifts will be returned in descending order, as is conventional in NMR. If `False`, the shifts will be in ascending order.
- **meshgrid** – If time-points are being derived for a N-dimensional signal (N > 1), setting this argument to `True` will return N-dimensional arrays corresponding to all combinations of points in each dimension. If `False`, an iterable of 1D arrays will be returned.

get_timepoints(*pts: Iterable[int] | None = None*, *start_time: Iterable[float | str] | None = None*, *meshgrid: bool = True*) → Iterable[numpy.ndarray]

Construct time-points which reflect the experiment parameters.

Parameters

- **pts** – The number of points to construct the time-points with in each dimesnion. If `None`, and `self.default_pts` is a tuple of ints, it will be used.
- **start_time** – The start time in each dimension. If set to `None`, the initial point in each dimension will be `0.0`. To set non-zero start times, a list of floats or strings can be used.
  - If floats are used, they specify the first value in each dimension in seconds.
  - Strings of the form `f'{N}dt'`, where `N` is an integer, may be used, which indicates a cetain multiple of the dwell time.
- **meshgrid** – If time-points are being derived for a N-dimensional signal (N > 1), setting this argument to `True` will return N-dimensional arrays corresponding to all combinations of points in each dimension. If `False`, an iterable of 1D arrays will be returned.

property latex_nuclei: Iterable[str | None] | None

Get the nuclei associated with each channel with for use in LaTeX.

Examples:

- `"1H"` → `"\\textsuperscript{1}H"`
- `"195Pt"` → `"\\textsuperscript{195}Pt"`

make_fid(*params: numpy.ndarray*, *pts: Iterable[int] | None = None*, *snr: float | None = None*, *decibels: bool = True*, *indirect_modulation: str | None = None*) → numpy.ndarray

Construct an FID from an array of oscillator parameters.

Parameters

- **params** – Parameter array with the following structure:
  - **1-dimensional data:**

```
params = numpy.array([
   [a_1, φ_1, f_1, η_1],
   [a_2, φ_2, f_2, η_2],
   ...,
   [a_m, φ_m, f_m, η_m],
])
```

  - **2-dimensional data:**

```
params = numpy.array([
   [a_1, φ_1, f1_1, f2_1, η1_1, η2_1],
   [a_2, φ_2, f1_2, f2_2, η1_2, η2_2],
   ...,
   [a_m, φ_m, f1_m, f2_m, η1_m, η2_m],
])
```

- **pts** – The number of points to construct the time-points with in each dimesnion. If `None`, and `self.default_pts` is a tuple of ints, it will be used.
- **snr** – The signal-to-noise ratio. If *None* then no noise will be added to the FID.
- **decibels** – If `True`, the snr is taken to be in units of decibels. If *False*, it is taken to be simply the ratio of the singal power over the noise power.

- **indirect_modulation** – Acquisition mode in the indirect dimension if the data is 2D. If the data is 1D, this argument is ignored.
  - `None` - hypercomplex dataset:

$$y(t_1, t_2) = \sum_m a_m e^{i\phi_m} e^{\left(2\pi i f_{1,m} - \eta_{1,m}\right)t_1} e^{\left(2\pi i f_{2,m} - \eta_{2,m}\right)t_2}$$

  - `"amp"` - amplitude modulated pair:

$$y_{\cos}(t_1, t_2) = \sum_m a_m e^{i\phi_m} \cos\left(\left(2\pi i f_{1,m} - \eta_{1,m}\right)t_1\right) e^{\left(2\pi i f_{2,m} - \eta_{2,m}\right)t_2}$$

$$y_{\sin}(t_1, t_2) = \sum_m a_m e^{i\phi_m} \sin\left(\left(2\pi i f_{1,m} - \eta_{1,m}\right)t_1\right) e^{\left(2\pi i f_{2,m} - \eta_{2,m}\right)t_2}$$

  - `"phase"` - phase-modulated pair:

$$y_{\mathrm{P}}(t_1, t_2) = \sum_m a_m e^{i\phi_m} e^{\left(2\pi i f_{1,m} - \eta_{1,m}\right)t_1} e^{\left(2\pi i f_{2,m} - \eta_{2,m}\right)t_2}$$

$$y_{\mathrm{N}}(t_1, t_2) = \sum_m a_m e^{i\phi_m} e^{\left(-2\pi i f_{1,m} - \eta_{1,m}\right)t_1} e^{\left(2\pi i f_{2,m} - \eta_{2,m}\right)t_2}$$

  `None` will lead to an array of shape `(n1, n2)`. `amp` and `phase` will lead to an array of shape `(2, n1, n2)`, with `fid[0]` and `fid[1]` being the two components of the pair.

See also:

- For converting amplitude-modulated data to spectral data, see `nmrespy.sig.proc_amp_modulated()`
- For converting phase-modulated data to spectral data, see `nmrespy.sig.proc_phase_modulated()`

manual_phase_data(*max_p1: float = 31.41592653589793*) → Tuple[float, float]

Manually phase the data using a Graphical User Interface.

Parameters

max_p1 – The largest permitted first order correction (rad). Set this to a larger value than the default (10π) if you anticipate having to apply a very large first order correction.

Returns

- *p0* – Zero order phase (rad)
- *p1* – First prder phase (rad)

See also:

`phase_data()`

classmethod new_bruker(*directory: str | pathlib.Path*, *convdta: bool = True*) → *nmrespy.estimators.jres.Estimator2DJ*

Create a new instance from Bruker-formatted data.

Parameters

- **directory** – Absolute path to data directory.
- **convdta** – If `True`, removal of the FID's digital filter will be carried out, using the `GRPDLY` parameter.

Notes

There are certain file paths expected to be found relative to `directory` which contain the data and parameter files:

- `directory/ser`
- `directory/acqus`
- `directory/acqu2s`

See also:

`nmrespy.sig.convdta()`

classmethod new_spinach(*shifts: Iterable[float]*, *couplings: Iterable[Tuple(int, int, float)]*, *pts: Tuple[int, int]*, *sw: Tuple[float, float]*, *offset: float*, *field: float = 11.74*, *nucleus: str = '1H'*, *snr: float | None = 20.0*, *lb: Tuple[float, float] | None = (6.91, 6.91)*) → None

Create a new instance from a 2DJ Spinach simulation.

Parameters

- **shifts** – A list of tuple of chemical shift values for each spin.
- **couplings** – The scalar couplings present in the spin system. Given `shifts` is of length `n`, couplings should be an iterable with entries of the form `(i1, i2, coupling)`, where `1 <= i1, i2 <= n` are the indices of the two spins involved in the coupling, and `coupling` is the value of the scalar coupling in Hz. `None` will set all spins to be uncoupled.
- **pts** – The number of points the signal comprises.
- **sw** – The sweep width of the signal (Hz).
- **offset** – The transmitter offset (Hz).
- **field** – The magnetic field strength (T).
- **nucleus** – The identity of the nucleus targeted in the pulse sequence.
- **snr** – The signal-to-noise ratio of the resulting signal, in decibels. `None` produces a noiseless signal.
- **lb** – Line broadening (exponential damping) to apply to the signal. The first point will be unaffected by damping, and the final point will be multiplied by `np.exp(-lb)`. The default results in the final point being decreased in value by a factor of roughly 1000.

property nuclei: Iterable[str | None] | None

Get the nuclei associated with each channel.

offset(*unit: str = 'hz'*) → Iterable[float]

Get the transmitter offset frequency.

Parameters

unit – Must be `"hz"` or `"ppm"`.

phase_data(*p0: float = 0.0*, *p1: float = 0.0*, *pivot: int = 0*) → None

Apply a first-order phase correction in the direct dimension.

Parameters

- **p0** – Zero-order phase correction, in radians.
- **p1** – First-order phase correction, in radians.
- **pivot** – Index of the pivot. `0` corresponds to the leftmost point in the spectrum.

See also:

`manual_phase_data()`

plot_result(*indices: Iterable[int] | None = None*, *multiplet_thold: float | None = None*, *high_resolution_pts: int | None = None*, *ratio_1d_2d: Tuple[float, float] = (2.0, 1.0)*, *region_unit: str = 'hz'*, *axes_left: float = 0.07*, *axes_right: float = 0.96*, *axes_bottom: float = 0.08*, *axes_top: float = 0.96*, *axes_region_separation: float = 0.05*, *xaxis_label_height: float = 0.02*, *xaxis_ticks: Iterable[Tuple[int, Iterable[float]]] | None = None*, *contour_base: float | None = None*, *contour_nlevels: int | None = None*, *contour_factor: float | None = None*, *contour_lw: float = 0.5*, *contour_color: Any = 'k'*, *jres_sinebell: bool = True*, *multiplet_colors: Any = None*, *multiplet_lw: float = 1.0*, *multiplet_vertical_shift: float = 0.0*, *multiplet_show_center_freq: bool = True*, *multiplet_show_45: bool = True*, *marker_size: float = 3.0*, *marker_shape: str = 'o'*, *label_peaks: bool = False*, *denote_regions: bool = False*, ***kwargs*) → Tuple[matplotlib.figure.Figure, numpy.ndarray[matplotlib.axes._axes.Axes]]

Generate a figure of the estimation result.

The figure includes a contour plot of the 2DJ spectrum, a 1D plot of the first slice through the indirect dimension, plots of estimated multiplets, and a plot of `cupid_spectrum()`.

Parameters

- **indices** – See *indices*.
- **multiplet_thold** – Frequency threshold for multiplet prediction. All oscillators that make up a multiplet are assumed to obey the following expression:

$$f_c - f_t < f^{(2)} - f^{(1)} < f_c + f_t$$

  where f_c is the central frequency of the multiplet, and *f_t* is the threshold.
- **high_resolution_pts** – Indicates the number of points used to generate the multiplet structures and `cupid_spectrum()`. Should be greater than or equal to `self.default_pts[1]`.
- **ratio_1d_2d** – The relative heights of the regions containing the 1D spectra and the 2DJ spectrum.
- **axes_left** – The position of the left edge of the axes, in figure coordinates. Should be between `0.` and `1.`.
- **axes_right** – The position of the right edge of the axes, in figure coordinates. Should be between `0.` and `1.`.
- **axes_top** – The position of the top edge of the axes, in figure coordinates. Should be between `0.` and `1.`.
- **axes_bottom** – The position of the bottom edge of the axes, in figure coordinates. Should be between `0.` and `1.`.

- **axes_region_separation** – The extent by which adjacent regions are separated in the figure.
- **xaxis_label_height** – The vertical location of the x-axis label, in figure coordinates. Should be between `0.` and `1.`, though you are likely to want this to be only slightly larger than `0.`.
- **xaxis_ticks** – See *xaxis_ticks*.
- **contour_base** – The lowest level for the contour levels in the 2DJ spectrum plot.
- **contour_nlevels** – The number of contour levels in the 2DJ spectrum plot.
- **contour_factor** – The geometric scaling factor for adjacent contours in the 2DJ spectrum plot.
- **contour_lw** – The linewidth of contours in the 2DJ spectrum plot.
- **contour_color** – The color of the 2DJ spectrum plot.
- **jres_sinebell** – If `True`, applies sine-bell apodisation to the 2DJ spectrum.
- **multiplet_colors** – Describes how to color multiplets. See *color cycle* for options.
- **multiplet_lw** – Line width of multiplet spectra
- **multiplet_vertical_shift** – The vertical displacement of adjacent mutliplets, as a multiple of `mutliplet_lw`. Set to `0.` if you want all mutliplets to lie on the same line.
- **multiplet_show_center_freq** – If `True`, lines are plotted on the 2DJ spectrum indicating the central frequency of each mutliplet.
- **multiplet_show_45** – If `True`, lines are plotted on the 2DJ spectrum indicating the 45° line along which peaks lie in each multiplet.
- **marker_size** – The size of markers indicating positions of peaks on the 2DJ contour plot.
- **marker_shape** – The shape of markers indicating positions of peaks on the 2DJ contour plot.
- **kwargs** – Keyword arguments provided to matplotlib.pyplot.figure. Allowed arguments include `figsize`, `facecolor`, `edgecolor`, etc.

Returns

- *fig* – The result figure. This can be saved to various formats using the savefig method.
- *axs* – A `(2, N)` NumPy array of the axes used for plotting. The first row of axes contain the 1D plots. The second row contain the 2DJ contour plots.

predict_multiplets(*indices: Iterable[int] | None = None*, *thold: float | None = None*, *freq_unit: str = 'hz'*, *rm_spurious: bool = False*, *_log: bool = True*, ***estimate_kwargs*) → Dict[float, Iterable[int]]

Predict the estimated oscillators which correspond to each multiplet in the signal.

Parameters

- **indices** – See *indices*.

- **thold** – Frequency threshold for multiplet prediction. All oscillators that make up a multiplet are assumed to obey the following expression:

$$f_c - f_t < f^{(2)} - f^{(1)} < f_c + f_t$$

  where f_c is the central frequency of the multiplet, and *f_t* is the threshold.

- **freq_unit** – Must be `"hz"` or `"ppm"`.
- **rm_spurious** – If set to `True`, all oscillators which fall into the following criteria will be purged:
  - The oscillator is the only member in a multiplet set.
  - The oscillator's frequency in F1 has a magnitude greater than `thold` (i.e. the indirect-dimension frequency is sufficiently far from 0Hz)
- **_log** – Ignore me!
- **estimate_kwargs** – If `rm_suprious` is `True`, and oscillators are purged, optimisation isrun. Kwargs are given to :py:meth:estimate:.

Returns
A dictionary with keys as the multiplet's central frequency, and values as a list of oscillator indices which make up the multiplet.

Return type
Dict[float, Iterable[int]]

save_log(*path: str | pathlib.Path = './espy_logfile'*, *force_overwrite: bool = False*, *fprint: bool = True*) → None

Save the estimator's log.

Parameters

- **path** – The path to save the log to.
- **force_overwrite** – If `path` already exists and `force_overwrite` is set to `False`, the user will be asked to confirm whether they are happy to overwrite the file. If `True`, the file will be overwritten without prompt.
- **fprint** – Specifies whether or not to print infomation to the terminal.

property sfo: Iterable[float | None] | None

Get the transmitter frequency (MHz).

sheared_signal(*indices: Iterable[int] | None = None*, *pts: Tuple[int, int] | None = None*, *indirect_modulation: str | None = None*) → numpy.ndarray

Return an FID where direct dimension frequencies are perturbed such that:

$$f_m^{(2)} = f_m^{(2)} - f_m^{(1)}$$

This should yeild a signal where all components in a multiplet are centered at the spin's chemical shift in the direct dimenion, akin to performing a 45° tilt.

Parameters

- **indices** – See *indices*.
- **pts** – The number of points to construct the signal from. If `None`, `self.default_pts` will be used.
- **indirect_modulation** – Acquisition mode in the indirect dimension.

- `None` - hypercomplex dataset:

$$y(t_1, t_2) = \sum_m a_m e^{i\phi_m} e^{\left(2\pi i f_{1,m} - \eta_{1,m}\right)t_1} e^{\left(2\pi i f_{2,m} - \eta_{2,m}\right)t_2}$$

- `"amp"` - amplitude modulated pair:

$$y_{\cos}(t_1, t_2) = \sum_m a_m e^{i\phi_m} \cos\left(\left(2\pi i f_{1,m} - \eta_{1,m}\right)t_1\right) e^{\left(2\pi i f_{2,m} - \eta_{2,m}\right)t_2}$$

$$y_{\sin}(t_1, t_2) = \sum_m a_m e^{i\phi_m} \sin\left(\left(2\pi i f_{1,m} - \eta_{1,m}\right)t_1\right) e^{\left(2\pi i f_{2,m} - \eta_{2,m}\right)t_2}$$

- `"phase"` - phase-modulated pair:

$$y_{\mathrm{P}}(t_1, t_2) = \sum_m a_m e^{i\phi_m} e^{\left(2\pi i f_{1,m} - \eta_{1,m}\right)t_1} e^{\left(2\pi i f_{2,m} - \eta_{2,m}\right)t_2}$$

$$y_{\mathrm{N}}(t_1, t_2) = \sum_m a_m e^{i\phi_m} e^{\left(-2\pi i f_{1,m} - \eta_{1,m}\right)t_1} e^{\left(2\pi i f_{2,m} - \eta_{2,m}\right)t_2}$$

`None` will lead to an array of shape `(n1, n2)`. `amp` and `phase` will lead to an array of shape `(2, n1, n2)`, with `fid[0]` and `fid[1]` being the two components of the pair.

See also:

- For converting amplitude-modulated data to spectral data, see `nmrespy.sig.proc_amp_modulated()`
- For converting phase-modulated data to spectral data, see `nmrespy.sig.proc_phase_modulated()`

property spectrum: numpy.ndarray

Return the spectrum associated with the estimator.

property spectrum_direct: numpy.ndarray

Generate a 1D spectrum of the first signal in the direct dimension.

property spectrum_first_direct: numpy.ndarray

Generate a 1D spectrum of the first signal in the direct dimension.

Generated by taking the first direct-dimension slice (`self.data[0]`), halving the initial point, and applying FT.

property spectrum_sinebell: numpy.ndarray

Spectrum with sine-bell apodisation.

Generated applying sine-bell apodisation to the FID, and applying FT.

property spectrum_tilt: numpy.ndarray

Generate the spectrum of the data with a 45° tilt.

subband_estimate(*noise_region: Tuple[float, float]*, *noise_region_unit: str = 'hz'*, *nsubbands: int | None = None*, ***estimate_kwargs*) → None

Perform estiamtion on the entire signal via estimation of frequency-filtered sub-bands.

This method splits the signal up into `nsubbands` equally-sized region and extracts parameters from each region before finally concatenating all the results together.

> **Warning:** This method is a work-in-progress. It is unlikely to produce decent results at the moment! I aim to improve the way that regions are created in the future.

Parameters

- **noise_region** – Specifies a frequency range where no noticeable signals reside, i.e. only noise exists.
- **noise_region_unit** – One of `"hz"` or `"ppm"`. Specifies the units that `noise_region` have been given in.
- **nsubbands** – The number of sub-bands to break the signal into. If `None`, the number will be set as the nearest integer to the data size divided by 500.
- **estimate_kwargs** – Keyword arguments to give to `estimate()`. Note that `region` and `initial_guess` will be ignored.

sw(*unit: str = 'hz'*) → Iterable[float]

Get the sweep width.

Parameters

unit – Must be `"hz"` or `"ppm"`.

to_pickle(*path: str | pathlib.Path | None = None*, *force_overwrite: bool = False*, *fprint: bool = True*) → None

Save the estimator to a byte stream using Python's pickling protocol.

Parameters

- **path** – Path of file to save the byte stream to. Do not include the `'".pkl"` suffix. If `None`, `./estimator_<x>.pkl` will be used, where `<x>` is the first number that doesn't cause a clash with an already existent file.
- **force_overwrite** – Defines behaviour if the specified path already exists:
  - If `False`, the user will be prompted if they are happy overwriting the current file.
  - If `True`, the current file will be overwritten without prompt.
- **fprint** – Specifies whether or not to print infomation to the terminal.

See also:

`from_pickle()`

property unicode_nuclei: Iterable[str | None] | None

Get the nuclei associated with each channel with superscript numbers.

Examples: `"1H"` → `"¹H"`, `"15N"` → `"¹⁵N"`.

view_data(*domain: str = 'freq'*, *abs_: bool = True*) → None

View the data FID or the spectral data with an interactive matplotlib figure.

Parameters

- **domain** – Must be `"freq"` or `"time"`.
- **abs_** – Whether or not to display frequency-domain data in absolute-value mode, as is conventional with 2DJ data.

write_cupid_to_bruker(*path: pathlib.Path | str*, *expno: int | None = None*, *indices: Iterable[int] | None = None*, *pts: int | None = None*, *force_overwrite: bool = False*) → None

Write the signal generated by `cupid_signal()` to a Bruker dataset.

The dataset is saved to a directory of the form `<path>/<expno>`

Parameters

- **path** – The path to the root directory to store the data in. This must already exist.
- **expno** – The experiment number. If `None`, the first directory number `<x>` for which `<path>/<x>/` isn;t currently a directory will be used.
- **indices** – See *indices*.
- **pts** – The number of points to construct the dataset from.
- **force_overwite** – If `False`, and `<path>/<expno>/` already exists, the user will be asked if they are happy to overwrite. If `True`, overwriting will take place without asking.

write_multiplets_to_bruker(*path: pathlib.Path | str*, *expno_prefix: int | None = None*, *indices: Iterable[int] | None = None*, *pts: int | None = None*, *thold: float | None = None*, *force_overwrite: bool = False*) → None

Write each individual multiplet structure to a Bruker data directory.

Each multiplet is saved to a directory of the form `<path>/<expno_prefix><x>/pdata/1` where `<x>` is iterated from 1 onwards.

Parameters

- **path** – The path to the root directory to store the data in.
- **expinfo_prefix** – Prefix to the experiment numbers for storing the multiplets to. If `None`, experiments will be numbered `1`, `2`, etc.
- **indices** – See *indices*.
- **pts** – The number of points to construct the mutliplets from.
- **thold** – Frequency threshold for multiplet prediction. All oscillators that make up a multiplet are assumed to obey the following expression:

$$f_c - f_t < f^{(2)} - f^{(1)} < f_c + f_t$$

  where f_c is the central frequency of the multiplet, and *f_t* is `thold`
- **force_overwite** – If `False`, if any directories that will be written to already exist, you will be promted if you are happy to overwrite. If `True`, overwriting will take place without asking.

write_result(*path: pathlib.Path | str = './nmrespy_result'*, *indices: Iterable[int] | None = None*, *fmt: str = 'txt'*, *description: str | None = None*, *sig_figs: int | None = 5*, *sci_lims: Tuple[int, int] | None = (-2, 3)*, *integral_mode: str = 'relative'*, *force_overwrite: bool = False*, *fprint: bool = True*, *pdflatex_exe: str | pathlib.Path | None = None*) → None

Write estimation result tables to a text/PDF file.

Parameters

- **path** – Path to save the result file to.
- **indices** – see *indices*
- **fmt** – Must be one of `"txt"` or `"pdf"`. If you wish to generate a PDF, you must have a LaTeX installation. See *LaTeX (Optional)*.
- **description** – Descriptive text to add to the top of the file.
- **sig_figs** – The number of significant figures to give to parameters. If `None`, the full value will be used. By default this is set to `5`.
- **sci_lims** – Given a value `(-x, y)` with ints `x` and `y`, any parameter `p` with a value which satisfies `p < 10 ** -x` or `p >= 10 ** y` will be expressed in scientific notation. If `None`, scientific notation will never be used.
- **integral_mode** – One of `"relative"` or `"absolute"`.
  - If `"relative"`, the smallest integral will be set to `1`, and all other integrals will be scaled accordingly.
  - If `"absolute"`, the absolute integral will be computed. This should be used if you wish to directly compare different datasets.
- **force_overwrite** – Defines behaviour if the specified path already exists:
  - If `False`, the user will be prompted if they are happy overwriting the current file.
  - If `True`, the current file will be overwritten without prompt.
- **fprint** – Specifies whether or not to print information to the terminal.
- **pdflatex_exe** – The path to the system's `pdflatex` executable.

Note: You are unlikely to need to set this manually. It is primarily present to specify the path to `pdflatex.exe` on Windows when the NMR-EsPy GUI has been loaded from TopSpin.

4.3 sig

A module for manipulating and processing NMR signals.

nmrespy.sig.**add_noise**(*fid: numpy.ndarray*, *snr: float*, *decibels: bool = True*) → numpy.ndarray

Add Gaussian white noise noise to an FID.

Parameters

- **fid** – Noiseless FID.
- **snr** – The signal-to-noise ratio. The smaller this value, the greater the variance of the noise.

- **decibels** – If *True*, the snr is taken to be in units of decibels. If *False*, it is taken to be simply the ratio of the singal power and noise power.

See also:

`make_noise()`

nmrespy.sig.**baseline_correction**(*spectrum: numpy.ndarray*, *mask: numpy.ndarray | None = None*, *min_length: int = 50*) → Tuple[numpy.ndarray, dict]

Apply baseline correction to a 1D dataset.

The algorithm applied is desribed in[1]. This uses an implementation provided by pybaselines.

Parameters

- **spectrum** – Spectrum to apply baseline correction to.
- **mask** – Should be either:
  - `None`: the points which comprise noise are predicted automatically
  - A boolean array with the same size as `spectrum`.
    - `True` indicates that a particular point comprises baseline.
    - `False` indicates that a point comprises a peak.
- **min_length** – *from pybaselines:* Any region of consecutive baseline points less than `min_length` is considered to be a false positive and all points in the region are converted to peak points. A higher *min_length* ensures less points are falsely assigned as baseline points.

Returns

- *fixed_spectrum* – The baseline-corrected spectrum.
- *params* – A dictionary with the items:
  - `"mask"` A boolean array designating baseline points as `True` and peak points as `False`.
  - `"baseline"` The computed baseline. Note that `fixed_spectrum` is computed via `spectrum - baseline`.

References

nmrespy.sig.**convdta**(*data: numpy.ndarray*, *grpdly: float*) → numpy.ndarray

Remove the digital filter from time-domain Bruker data.

This function is inspired by nmrglue.fileio.bruker.rm_dig_filter.

Parameters

- **data** – Time-domain data to process.
- **grpdly** – Group delay.

[1] Cobas, J., et al. A new general-purpose fully automatic baseline-correction procedure for 1D and 2D NMR data. Journal of Magnetic Resonance, 2006, 183(1), 145-151.

nmrespy.sig.**exp_apodisation**(*fid: numpy.ndarray*, *k: float*, *axes: Iterable[int] | None = None*) → numpy.ndarray

Apply exponential apodisation to an FID.

The FID is multiplied by `np.exp(-k * np.linspace(0, 1, n))` in each dimension specified by `axes`, where `n` is the size of each dimension.

Parameters

- **fid** – FID to process.
- **k** – Line-broadening factor.
- **axes** – The axes to apply the apodiisation over. If `None`, all axes are apodised.

nmrespy.sig.**ft**(*fid: numpy.ndarray*, *axes: Iterable[int] | int | None = None*, *flip: bool = True*) → numpy.ndarray

Fourier transformation with optional spectrum flipping.

It is conventional in NMR to plot spectra from high to low going left to right/down to up. This function utilises the numpy.fft module.

Parameters

- **fid** – Time-domain data.
- **axes** – The axes to apply Fourier Transformation to. By default (`None`), FT is applied to all axes. If an int, FT will only be applied to the relevant axis. If a list of ints, FT will be applied to this subset of axes.
- **flip** – Whether or not to flip the Fourier Transform of *fid* in each dimension.

nmrespy.sig.**ift**(*spectrum: numpy.ndarray*, *axes: Iterable[int] | int | None = None*, *flip: bool = True*) → numpy.ndarray

Inverse Fourier Transform a spectrum.

This function utilises the numpy.fft module.

Parameters

- **spectrum** (`numpy.ndarray`) – Spectrum
- **axes** – The axes to apply IFT to. By default (`None`), IFT is applied to all axes. If an int, IFT will only be applied to the relevant axis. If a list of ints, IFT will be applied to this subset of axes.
- **flip** (`bool, default: True`) – Whether or not to flip `spectrum` in each dimension prior to Inverse Fourier Transform.

nmrespy.sig.**make_noise**(*fid: numpy.ndarray*, *snr: float*, *decibels: bool = True*) → numpy.ndarray

Generate an array of white Guassian complex noise.

The noise will be created with zero mean and a variance that abides by the desired SNR, in accordance with the FID provided.

Parameters

- **fid** – Noiseless FID.
- **snr** – The signal-to-noise ratio.
- **decibels** – If *True*, the snr is taken to be in units of decibels. If *False*, it is taken to be simply the ratio of the singal power and noise power.

Return type
noise

Notes

Noise variance is given by:

$$\rho = \frac{\sum_{n=0}^{N-1} \left(x_n - \mu_x\right)^2}{N \cdot 20 \log_1 0 \left(\mathrm{SNR}_{\mathrm{dB}}\right)}$$

See also:

`add_noise()`

nmrespy.sig.**make_virtual_echo**(*data: numpy.ndarray*, *twodim_dtype: str | None = None*) → numpy.ndarray

Generate a virtual echo[2] from a time-domain signal.

A vitrual echo is a signal with a purely real Fourier-Tranform.

Parameters

- **data** – The data to construct the virtual echo from. If the data comprises a pair of amplitude/phase modulated signals, these should be stored in a single 3D array with `shape[0] == 2`, such that `data[0]` is the cos/p signal, and `data[1]` is the sin/n signal.
- **twodim_dtype** – If the data is 2D, this parameter specifies the way to process the data. Allowed options are:
  - **"hyper": The data is hypercomplex. Virtual echo is constructed** along the second axis.
  - `"amp"`: The data comprises an amplitude-modulated pair.
  - `"phase"`: The data comprises a phase-modulated pair.

References

nmrespy.sig.**manual_phase_data**(*spectrum: numpy.ndarray*, *max_p1: Iterable[float] | None = None*) → Tuple[Iterable[float] | None, Iterable[float] | None]

Manual phase correction using a Graphical User Interface.

Note: Only 1D spectral data is currently supported.

Parameters

- **spectrum** – Spectral data of interest.
- **max_p1** – Specifies the range of first-order phases permitted. Bounds are set as `[-max_p1, max_p1]`.

Returns

[2] M. Mayzel, K. Kazimierczuk, V. Y. Orekhov, The causality principle in the reconstruction of sparse nmr spectra, Chem. Commun. 50 (64) (2014) 8947–8950.

- *p0* – Zero-order phase correction in each dimension, in radians. If the user chooses to cancel rather than save, this is set to `None`.
- *p1* – First-order phase correction in each dimension, in radians. If the user chooses to cancel rather than save, this is set to `None`.

`nmrespy.sig.`**`phase`**(*data: numpy.ndarray*, *p0: Iterable[float]*, *p1: Iterable[float]*, *pivot: Iterable[float | int] | None = None*) → numpy.ndarray

Apply a linear phase correction to a signal.

Parameters

- **`data`** – Data to be phased.
- **`p0`** – Zero-order phase correction in each dimension, in radians.
- **`p1`** – First-order phase correction in each dimension, in radians.
- **`pivot`** – Index of the pivot in each dimension. If `None`, the pivot will be `0` in each dimension.

`nmrespy.sig.`**`proc_amp_modulated`**(*data: numpy.ndarray*) → numpy.ndarray

Generate a frequency-discrimiated signal from amplitude-modulated 2D FIDs.

Parameters
`data` – cos-modulated signal and sin-modulated signal, stored in a 3D numpy array, such that `data[0]` is the the cos signal and `data[1]` is the sin signal.

Returns
spectrum – Frequency-discrimiated spectrum.

Return type
np.ndarray

`nmrespy.sig.`**`proc_phase_modulated`**(*data: numpy.ndarray*) → numpy.ndarray

Process phase modulated 2D FIDs.

This function generates the set of spectra corresponding to the processing protocol outlined in[3].

Parameters
`data` – P-type signal and N-type signal, stored in a 3D numpy array, such that `data[0]` is the the P signal and `data[1]` is the N signal.

Returns

3D array with `spectra.shape[0] == 4`. The sub-arrays in axis 0 correspond to the following signals:

- `spectra[0]`: RR
- `spectra[1]`: RI
- `spectra[2]`: IR
- `spectra[3]`: II

Return type
spectra

[3] A. L. Davis, J. Keeler, E. D. Laue, and D. Moskau, "Experiments for recording pure-absorption heteronuclear correlation spectra using pulsed field gradients," Journal of Magnetic Resonance (1969), vol. 98, no. 1, pp. 207–216, 1992.

References

nmrespy.sig.**sinebell_apodisation**(*fid: numpy.ndarray*, *axes: Iterable[int] | None = None*) → numpy.ndarray

Apply sine-bell apodisation to an FID.

The FID is multiplied by `np.exp(-k * np.linspace(0, 1, n))` in each dimension specified by `axes`, where `n` is the size of each dimension.

Warning: This is not intended for manipulating the FID prior to estimation.

Parameters

- **fid** – FID to process.
- **axes** – The axes to apply the apodiisation over. If `None`, all axes are apodised.

nmrespy.sig.**zf**(*data: numpy.ndarray*) → numpy.ndarray

Zero-fill data to the next power of 2 in each dimension.

Parameters

data – Signal to zero-fill.

4.4 Common Arguments

Listed here are descriptions of arguments which commonly appear in methods of the various estimator classes.

4.4.1 indices

A list of the indices of results to consider. Index `0` corresponds to the first result obtained using the estimator, `1` corresponds to the next, etc. You can also you negative ints for backward-indexing. For example `-1` corresponds to the last result acquired. If `None`, all results will be considered.

Suppose you have called the estimator's `estimate` method 3 times:

```
estimator.estimate(region=(5., 4.5), ...)
estimator.estimate(region=(3., 2.5), ...)
estimator.estimate(region=(1., 0.5), ...)
```

With `indices=[0]`, only the result corresponding to the 5-4.5 region will be considered. With `indices=[1, 2]`, both the results corresponding to the 3-2.5 and 1-0.5 regions will be considered. With `indices=[-1]`, only the result corresponding to 1-0.5 will be considered.

4.4.2 index

An integer denoting the estimation result to consider. See *indices* for more details.

4.4.3 color cycle

The following is a complete list of options:

- If a valid matplotlib colour is given, all multiplets will be given this color.
- If a string corresponding to a matplotlib colormap is given, the multiplets will be consecutively shaded by linear increments of this colormap.
- If an iterable object containing valid matplotlib colors is given, these colors will be cycled. For example, if `oscillator_colors = ['r', 'g', 'b']`:
  - Multiplets 0, 3, 6, ... would be red (#FF0000)
  - Multiplets 1, 4, 7, ... would be green (#008000)
  - Multiplets 2, 5, 8, ... would be blue (#0000FF)
- If `None`, the default colouring method will be applied, which involves cycling through the following colors:
  - #1063E0
  - #EB9310
  - #2BB539
  - #D4200C

4.4.4 xaxis_ticks

Specifies custom x-axis ticks for each region, overwriting the default ticks. Should be of the form: `[(i, (a, b, ...)), (j, (c, d, ...)), ...]` where `i` and `j` are ints indicating the region under consideration, and `a-d` are floats indicating the tick values.