NMR-EsPy

Simon Hulse & Mohammadali Foroozandeh

Jul 06, 2021

CONTENTS

Simon Hulse and Mohammadali Foroozandeh, University of Oxford

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 the majority of packages that do a lot of number crunching in Python, NMR-EsPy relies heavily on NumPy as well as SciPy routines to function efficiently.

The NMR-EsPy package is a product of work carried out during my Ph.D within the group of Dr. Mohammadali Foroozandeh, in the University of Oxford's Chemistry Department. To find out more about the group, take a look at our website.

The following publications are directly linked to NMR-EsPy and provide an extensive background on the underlying theory and examples of results obtained by using it:

No publications yet...

For more information:

https://github.com/foroozandehgroup/NMR-EsPy

simon.hulse@chem.ox.ac.uk

CHAPTER

ONE

INSTALLATION

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

```
$ python3 -m pip install nmrespy
```

You need to be using Python 3.8 or above.

1.1 Python Packages

Package	Version	Details
Numpy	1.20+	Ubiquitous
Scipy	1.6+	Ubiquitous
Matplotlib	3.3+	Required for result plotting, and for manual phase-correction
Colorama	0.4+	Required on Windows only. Enables ANSI escape character sequences to work, allowing coloured terminal output.

1.2 LaTeX

NMR-EsPy provides functionality to save result files to PDF format using LaTeX. Of course, if you wish to use this feature, you will need to have a LaTeX installed on your machine. The easiest way to get LaTeX is probably to install TexLive.

Essentially, in order for generation of PDFs to work, the command `pdflatex` should exist. To check this, open a terminal/command prompt.

Note: Windows users

Press `Win` + `R`, and then type `cmd` into the window that pops up. Finally, press `<Return>`.

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.

There are a few LaTeX packages required to generate result PDFs. These are outlined in the *Notes* section of the documentation of *nmrespy.write.write_result()*.

CHAPTER

TWO

THEORY OVERVIEW

Note: This page is work in progress.

On this page, I provide a brief overview of the goal of NMR-EsPy and how it attempts to achieve this. If you simply wish to use NMR-EsPy without worrying about the underlying theory, feel free to skip this.

The signal produced as a result of a typical NMR experiment (FID) can be thought of as a summation of a number of complex exponentials, in the presence of experimental noise. For the general case of a signal from a D-dimensional NMR experiment, we expect the functional form of the signal at any time during acquisition to be:

$$y(t_1, \dots, t_D) = \sum_{m=1}^{M} \left\{ a_m \exp\left(\mathrm{i}\phi_m\right) \prod_{d=1}^{D} \exp\left[\left(2\pi \mathrm{i} f_{d,m} - \eta_{d,m}\right) t_d\right] \right\} + w(t_1, \dots, t_D),$$

where

- t_d is the time cosidered in the d-th dimension
- M is the number of complex exponentials (oscillators) contributing to the FID.
- a_m is the amplitude of oscillator m
- ϕ_m is the phase of oscillator m
- $f_{d,m}$ is the frequency of oscillator m in the d-th dimension
- $\eta_{d,m}$ is the damping factor of oscillator m in the d-th dimension
- $w(t_1, \dots, t_D)$ is the contribution from experimental noise

Of course, NMR signals are digitised, and so the raw output of an experiment will be a D-dimensional array Y of a finite shape:

$$Y \in \mathbb{C}^{N_1 \times \dots \times N_D},$$

where N_d is the number of points sampled in the d-th dimension. Assuming that the signal is uniformly sampled in each dimesnion, the time at which any point is sampled is given by

$$\tau_n^{(d)} = n_d \Delta t_d, \quad n_d \in \{0, 1, \dots, N_d - 1\},$$

where Δt_d is the sampling rate (the time between successive samples) in dimension d. The discrete fid Y therefore has elements $Y\left[n_1, \dots, n_D\right]$ of the form

$$Y\left[n_1, \dots, n_D\right] = \sum_{m=1}^{M} \left\{ a_m \exp\left(\mathrm{i}\phi_m\right) \prod_{d=1}^{D} \exp\left[\left(2\pi \mathrm{i} f_{d,m} - \eta_{d,m}\right) \tau_n^{(d)}\right] \right\} + W(n_1, \dots, n_D)$$

Writing this in a more succinct notation:

$$Y = X\left(\theta\right) + W$$

where the vector $\theta \in \mathbb{R}^{(2+2D)M}$ cotains all the deterministic parameters which describe the signal

$$\theta = \left[a_1, a_2, \dots, a_M, \phi_1, \dots, \phi_M, f_{1,1}, \dots, f_{1,M}, \dots, f_{D,1}, \dots, f_{D,M}, \eta_{1,1}, \dots \eta_{D,M}\right]^{\mathrm{T}}$$

The goal of NMR-EsPy is to estimate θ for a given signal. This is rather challenging, especially for signals with many resonances, as one doesn't even know how many oscilltors are contained within the signal in general (M).

To estimate θ, we apply *Newton's Method*, an iterative procedure which attempts to locate extrema of functions. In the case of NMR-EsPy, we wish to minimise the following:

$$\mathcal{F}(\theta) = \|\tilde{Y} - X(\theta)\|_2^2$$

where $\tilde{Y} = Y/\|Y\|$. This is a very commonly encountered function in the context of optimisation, called the residual sum of squares. There are numerous variants of Newton's method, but the general idea is to approximate the neighbourhood of $\mathcal{F}(\theta)$ about the current value of θ as quadratic, and to determine a step with a certain direction and size such that $\mathcal{F}(\theta)$ is reduced. This procedure is iterated until the routine converges to a minimum in the function.

CHAPTER

THREE

EXAMPLE WALKTHROUGH

This page provides an outline of basic NMR-EsPy usage through writing Python code. If you wish to use the graphical user interface instead, *go to Chapter 4*.

As an illustration of the typical steps involved in using NMR-EsPy, we will consider an example dataset that ships with TopSpin 4. Assuming you installed TopSpin in the default path, this should be present in the path:

- Linux: `/opt/topspin4.x.y/examdata/exam1d_1H/1/pdata/1`
- Windows: `C:\Bruker\TopSpin4.x.y\examdata\exam1d_1H\1\pdata\1`

In what follows, as I am using TopSpin 4.0.8, I shall be replacing `topspin4.x.y` with `topspin4.0.8`.

I recommend that you follow this walkthrough using a Python interpreter to ensure everything runs smoothly on your system.

3.1 Generating an Estimator instance

To get started, it is necessary to import the *`nmrespy.core.Estimator`* class. A new instance of this class is initialised using the static *`new_bruker()`* method:

```
>>> from nmrespy.core import Estimator
>>> # Specify the path to the directory containing 1r
>>> path = "/opt/topspin4.0.8/examdata/exam1d_1H/1/pdata/1"
>>> estimator = Estimator.new_bruker(path)
>>> type(estimator)
<class 'nmrespy.core.Estimator'>
```

Note: If you are using Windows, you should also import and initialise `colorama` to ensure that coloured output is possible. If you do not, you may see a fair amount of gobbledygook:

```
>>> import colorama
>>> colorama.init()
```

Information about the estimator can be seen by printing it:

```
>>> print(estimator)
<nmrespy.core.Estimator at 0x7f50e8ad8fa0>
source : bruker_pdata
data : [ 8237241.76470947      +0.j          1834272.48552552-9941412.67849912j
 -7908307.89165751+1371281.69800517j ...
        0.              +0.j                0.              +0.j
        0.              +0.j        ]
```

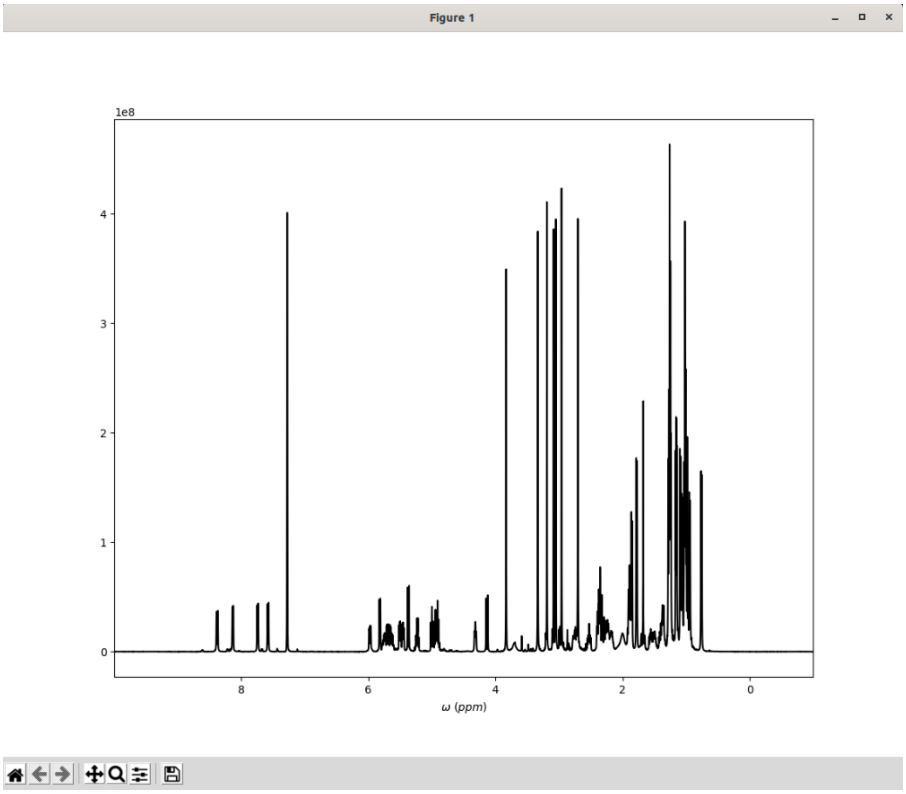
(continues on next page)

(continued from previous page)

```
dim : 1
n : [32768]
path : /opt/topspin4.0.8/examdata/exam1d_1H/1/pdata/1
sw : [5494.50549450549]
offset : [2249.20599998768]
sfo : [500.132249206]
nuc : ['1H']
fmt : <i4
filter_info : None
result : None
errors : None
```

An interactive plot of the data, in the frequency domain, can be seen using the *view_data()* method:

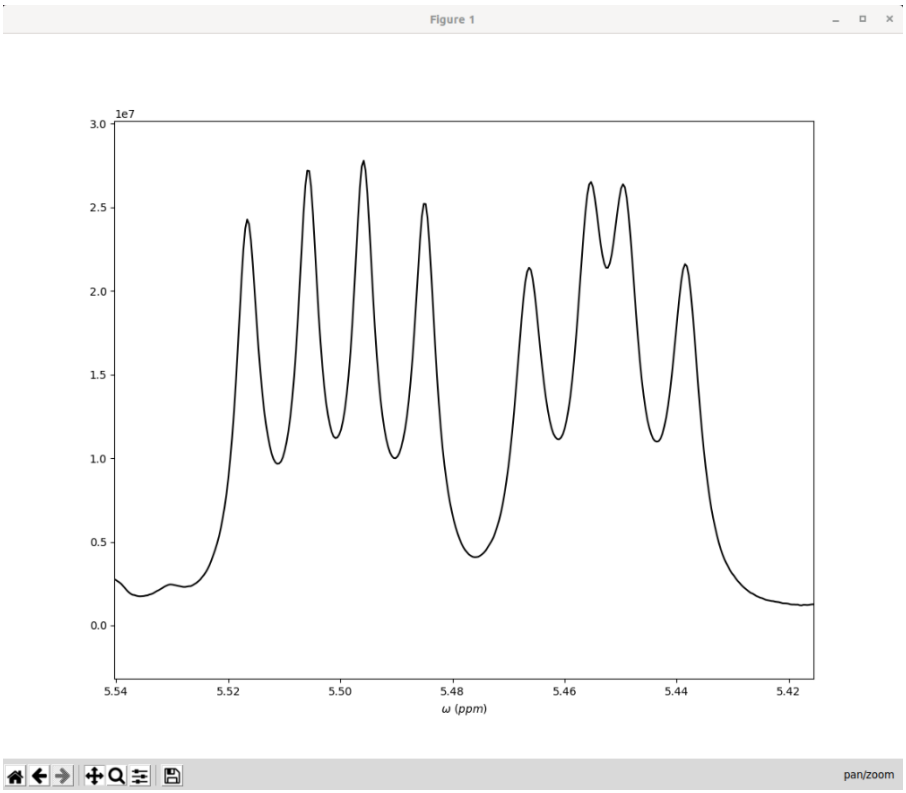
```
>>> estimator.view_data()
```



3.2 Frequency Filtration

For complex NMR signals, it is typically necessary to consider a subset of the frequency space at any time, otherwise the computational burden would be too large. To overcome this, it is possible to derive a time-domain signal which has been generated via frequency-filtration.

In this example, I am going to focus on the spectral region between 5.54-5.42ppm. The region looks like this:



To generate a frequency-filtered signal from the imported data, the *frequency_filter()* method is used. All well as specifying the region of interest, it is also necessary to provide a region that appears to contain no signals (this is used to gain an insight into the data's noise variance). In this example, I will set this region to be -0.15 to -0.3ppm.

```
>>> estimator.frequency_filter([[5.54, 5.42]], [[-0.15, -0.3]])
```

Note: Be aware of the use of two sets of parentheses around the regions specified. This may seem odd, but a nested list is required to ensure compatibility with 2D data as well.

3.3 Estimating the Signal Parameters

3.3.1 Matrix Pencil Method

Now that a frequency filtered signal has been generated, we can begin the estimation routine. Before estimating the signal parameters using nonlinear programming (NLP), an initial guess of the parameters is required. We can derive this guess using *matrix_pencil()*:

```
>>> estimator.matrix_pencil()
============================
MATRIX PENCIL METHOD STARTED
============================
--> Pencil Parameter: 358
--> Hankel data matrix constructed:
Size:   718 x 359
Memory: 3.9331MiB
```

(continues on next page)

(continued from previous page)

```
--> Performing Singular Value Decomposition...
--> Determining number of oscillators...
    Number of oscillators will be estimated using MDL
    Number of oscillations: 12
--> Determining signal poles...
--> Determining complex amplitudes...
--> Checking for oscillators with negative damping...
    None found
 ============================
 MATRIX PENCIL METHOD COMPLETE
 ============================
 Time elapsed: 0 mins, 0 secs, 388 msecs
```

The result of the estimation is stored within the `result` attribute, which can be accessed using *`get_result()`*.

3.3.2 Nonlinear Programming

The `result` attribute is next subjected to a NLP routine using the *`nonlinear_programming()`* method. As the frequency-filtered data was derived from well-phased spectral data, the optional `phase_variance` argument is set to `True`. The optimisation routine will then ensure that the estimate's phases are similar to each other (and hopefully very close to 0), and will often remove excessive oscillators from the Matrix Pencil result (note that our initial guess in this example contains 12 oscillators).

```
>>> estimator.nonlinear_programming(phase_variance=True)
============================
NONLINEAR PROGRAMMING STARTED
============================
| niter |f evals|CG iter|  obj func   |tr radius |   opt    |  c viol  | penalty  |CG stop|
|-------|-------|-------|-------------|----------|----------|----------|----------|-------|
|   1   |   1   |   0   | +1.6287e-01 | 1.00e+00 | 9.30e-02 | 0.00e+00 | 1.00e+00 |   0   |
|   2   |   2   |   1   | +9.0652e-02 | 7.00e+00 | 6.92e-01 | 0.00e+00 | 1.00e+00 |   2   |
|   3   |   3   |   9   | +9.0652e-02 | 7.00e-01 | 6.92e-01 | 0.00e+00 | 1.00e+00 |   3   |


--snip--

|  100  |  100  |  966  | +6.4830e-04 | 1.27e-01 | 2.56e-03 | 0.00e+00 | 1.00e+00 |   2   |

--snip--

Negative amplitudes detected. These oscillators will be removed
Updated number of oscillators: 9
| niter |f evals|CG iter|  obj func   |tr radius |   opt    |  c viol  | penalty  |CG stop|
|-------|-------|-------|-------------|----------|----------|----------|----------|-------|
|   1   |   1   |   0   | +1.2497e-03 | 1.00e+00 | 1.08e-01 | 0.00e+00 | 1.00e+00 |   0   |

--snip--

|  100  |  100  | 2228  | +8.5451e-04 | 9.95e+00 | 2.47e-05 | 0.00e+00 | 1.00e+00 |   1   |

--snip--

============================
NONLINEAR PROGRAMMING COMPLETE
============================
Time elapsed: 0 mins, 3 secs, 186 msecs
```

The `result` attribute has now been updated with the result obtained using NLP. The routine also computes the errors associated with each parameter, which can be accessed with *`get_errors()`*.

3.4 Saving Results

3.4.1 Writing Results to a Text File/PDF/CSV

The estimation result can be written to `.txt`, `.pdf` and `.csv` formats, using the *`write_result()`* method.

```
>>> msg = "Example estimation result for NMR-EsPy docs."
>>> for fmt in ["txt", "pdf", "csv"]:
...     estimator.write_result(path="example", description=msg, fmt=fmt)
...
Saved result to /<pwd>/example.txt
Result successfully output to:
/<pwd>/example.pdf
If you wish to customise the document, the TeX file can be found at:
/<pwd>/example.tex
Saved result to /<pwd>/example.csv
```

Note: In order to generate PDF files, you will need a LaTeX installation on your system. See the documentation for the *`nmrespy.write.write_result()`* function for more details.

3.4.2 Generating Result Figures

To generate a figure of the result, you can use the *`plot_result()`* method, which utilises matplotlib. There is wide scope for customising the plot, which is described in detail in *Figure Customisation*. See *Summary* below for an example of some basic plot customisation.

```
>>> plot = estimator.plot_result()
>>> plot.fig.savefig("plot_example.png")
```

3.4.3 Pickling Estimator Instances

The estimator instance can be serialised, and saved to a byte stream using Python's pickle module, with *`to_pickle()`*:

```
>>> estimator.to_pickle(path="pickle_example")
Saved instance of Estimator to /<pwd>/pickle_example.pkl
```

The estimator can subsequently be recovered using *`from_pickle()`*:

```
>>> estimator_cp = Estimator.from_pickle(path="pickle_example")
>>> type(estimator_cp)
<class 'nmrespy.core.Estimator'>
```

3.4.4 Saving a Logfile

A summary of the methods applied to the estimator can be saved using the *save_logfile()* method:

```
>>> estimator.save_logfile(path="logfile_example")
Log file successfully saved to /<pwd>/logfile_example.log
```

3.5 Summary

A script which performs the entire procedure described above is as follows. Note that further customisation has been applied to the plot to give it an "aesthetic upgrade".

```
from nmrespy.core import Estimator

# Path to data. You'll need to change the 4.0.8 bit if you are using a
# different TopSpin version.

# --- UNIX users ---
path = "/opt/topspin4.0.8/examdata/exam1d_1H/1/pdata/1"

# --- Windows users ---
# path = "C:/Bruker/TopSpin4.0.8/examdata/exam1d_1H/1/pdata/1"

estimator = Estimator.new_bruker(path)

# --- Frequency filter & estimate ---
estimator.frequency_filter([[5.54, 5.42]], [[-0.15, -0.3]])
estimator.matrix_pencil()
estimator.nonlinear_programming(phase_variance=True)

# --- Write result files ---
msg = "Example estimation result for NMR-EsPy docs."
for fmt in ["txt", "pdf", "csv"]:
    estimator.write_result(path="example", description=msg, fmt=fmt)

# --- Plot result ---
# Set oscillator colours using the viridis colourmap
plot = estimator.plot_result(oscillator_colors='viridis')
# Shift oscillator labels
# Move the labels associated with oscillators 1, 2, 5, and 6
# to the right and up.
plot.displace_labels([1,2,5,6], (0.02, 0.01))
# Move the labels associated with oscillators 3, 4, 7, and 8
# to the left and up.
plot.displace_labels([3,4,7,8], (-0.04, 0.01))
# Move oscillators 9's label to the right
plot.displace_labels([9], (0.02, 0.0))

# Save figure as a PNG
plot.fig.savefig("plot_example_edited.png")

# Save the estimator to a binary file.
estimator.to_pickle(path="pickle_example")
```

(continues on next page)

(continued from previous page)

```
# Save a logfile of method calls
estimator.save_logfile(path="logfile_example")
```

More features are provided by the *Estimator* beyond what is described on this page, but this gives an overview of the primary functionality.

CHAPTER

FOUR

THE NMR-ESPY GUI

A Graphical User Interface (GUI) is available for using NMR-EsPy without the requirement of writing Python scripts. Find out more through the following links:

4.1 Integrating the GUI into TopSpin

It is possible to directly load the NMR-EsPy GUI from within TopSpin. The GUI can be installed to TopSpin either at the point of installing NMR-EsPy using `pip install`, or at any subsequent point.

Note: In this section, `<pyexe>` denotes the symbolic link/path to the Python executable you are using.

4.1.1 Automatic installation

After installing NMR-EsPy using `pip install`, you can set up the TopSpin GUI loader by entering the following into a terminal:

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

The script searches for directories matching the following glob pattern in 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/topspin3.6.3
    [2] /opt/topspin4.0.8
For each installation that 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, if you wanted to install the GUI loader to both TopSpin 3.6.3 and TopSpin 4.0.8, you would enter `1 2` or simply press `<Return>`. If you only wanted to install to TopSpin 4.0.8, you would enter `2`. To cancel the install, enter `0`.

For each specified path to install to, the script will try to copy the GUI loader to the path `/.../topspin<x.y.z>/exp/stan/nmr/py/user/nmrespy.py`, where `<x.y.z>` is the TopSpin version number.

The result of the attempted install will be printed to the terminal. Here is an example where I try to install to both TopSpin 4.0.8 and TopSpin 3.6.3:

```
SUCCESS:
    /opt/topspin3.6.3/exp/stan/nmr/py/user/nmrespy.py

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

4.1.2 Manual Installation

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

Copying the loader script

Open a terminal/command prompt and enter the following to determine where the GUI loading script is located:

```
$ <pyexe> -c "import nmrespy; print(nmrespy.TOPSPINPATH)"
/home/simon/.local/lib/python3.9/site-packages/nmrespy/app/_topspin.py
```

Now you simply need to copy this file to your TopSpin installation. You should rename the copied file as `nmrespy.py`:

- UNIX:

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

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

- Windows:

  ```
  > copy C:\Users\simon\AppData\Roaming\Python\Python38\site-packages\nmrespy\app\_topspin.py ^
  More? C:\path\to\...\TopSpinx.y.z\exp\stan\nmr\py\user\nmrespy.py
  ```

Note: In the UNIX example, `\` followed by pressing `<Return>` allows a single long command to span multiple lines. Similarly, `^`, followed by `<Return>` achieves the same thing in Windows cmd.

Editing the loader script

Now you need to open the newly created file:

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

- **Specifying the Python executable path**

  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 interpreter or write a script with 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\\AppData\\Local\\Programs\\Python\\Python38\\python.exe"
```

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

```
py_exe = "C:\\Users\\simon\\AppData\\Local\\Programs\\Python\\Python38\\python.exe"
```

- **(Optional) Specifying the pdflatex path**

  If you have `pdflatex` on your system (see the *LaTeX* section in *Installation*), 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, load a Python interpreter/ write a Python script with the following lines:

  – *UNIX*

  ```
  >>> from subprocess import check_output as co
  >>> exe = check_output("which pdflatex", shell=True)
  >>> 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"
```

With the Python path and (optionally) the `pdflatex` path set, the script should now work.

4.2 Using The GUI

4.2.1 Loading the GUI

Note: In this section, <pyexe> denotes the symbolic link/path to the Python executable you are using.

The GUI can be loaded both from a terminal/command prompt, or from within TopSpin provided the GUI loader has been installed (see *Integrating the GUI into TopSpin*).

From a terminal

To set-up an estimation routine from the terminal/command prompt, enter the following command:

```
$ <pyexe> -m nmrespy --estimate <path_to_bruker_data>
```

Note: The shorthand flag `-e` can be used in place of `--estimate`.

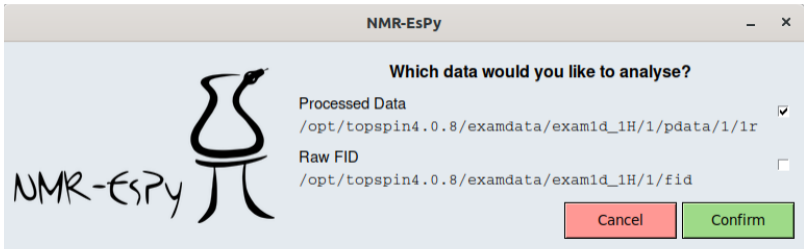
`<path_to_bruker_data>` should be one of the following:

- The path to the parent directory of the raw time-domain data (`fid`).
- The path to the parent directory of the processed data (`1r`).

From TopSpin

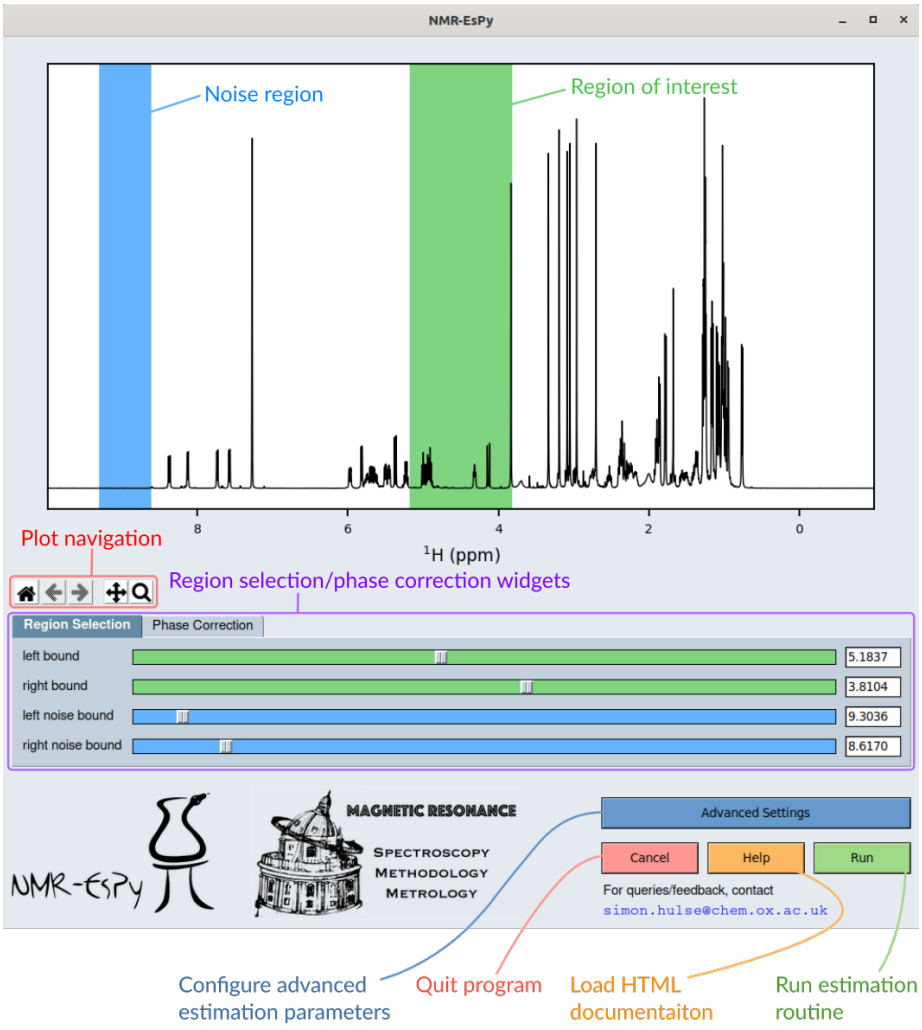
To load the GUI from TopSpin, simply select the data you wish to look at, and then enter the command `nmrespy` into the prompt in the bottom left corner.

You will be asked to select the data you wish to consider (either the raw time-domain data, or the processed data):



4.2.2 Estimation Set-up

The following is a screenshot of the NMR-EsPy GUI calculation set-up window. Key features of the window are annotated:



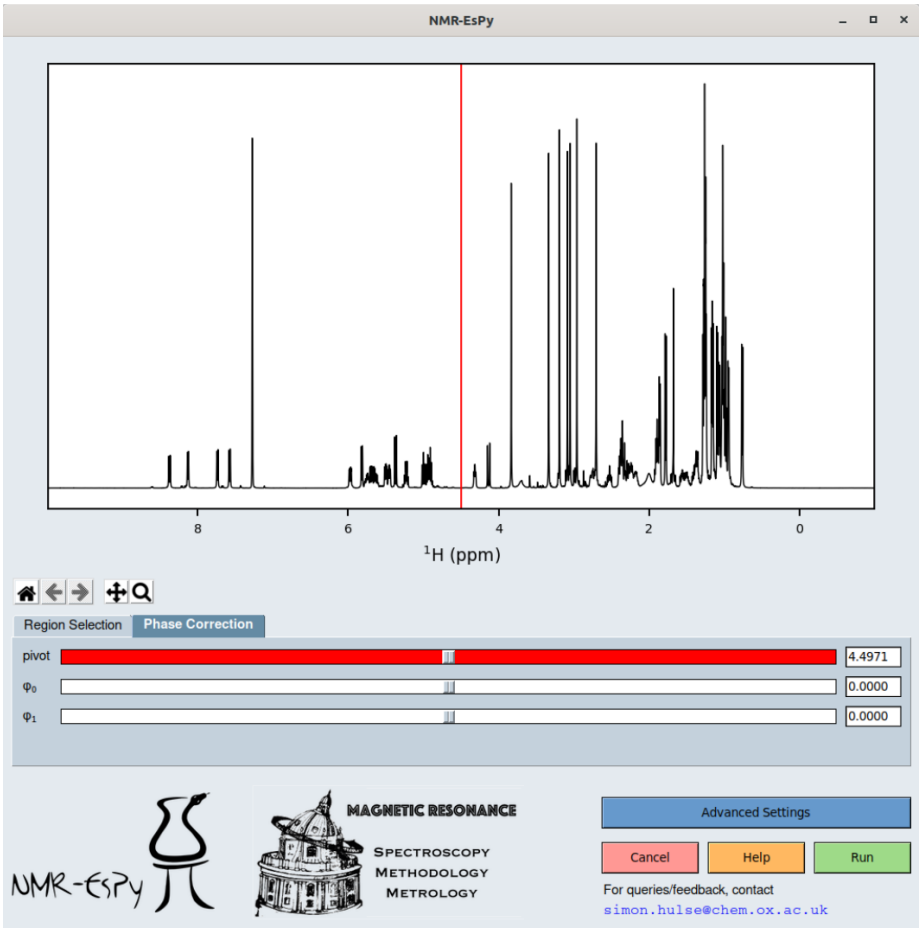
Plot Navigation

The Plot navigation toolbar enables you to change the view of the data. It is an edited version of matplotlib's toolbar, with the following available buttons:

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.

Phase Correction

The GUI has the following appearance when the *Phase Correction* tab is selected:



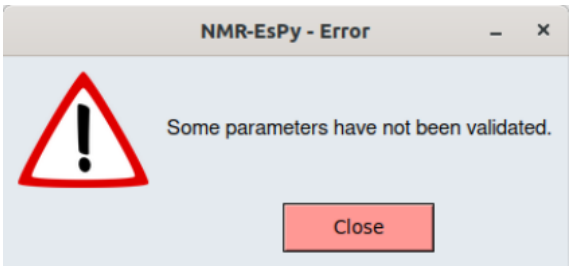
Phase correction can be carried out by editing the pivot (red line in the above figure), zero-order phase and first-order phase. This is unlikely to be necessary if you are considering processed data, however you will probably need to do this if you are considering the raw time-domain data.

The values may be changed either by adjusting the scale widgets, or by manually inputting desired values into the adjacent entry boxes.

Note: Validating entry box inputs

For the majority of entry boxes in the GUI, you will notice that the box turns red after you manually change its contents. This indicates that the input adhere to certain criteria (i.e. it must be a number, a valid path on your computer etc.), and it has not been validated. After you have changed the value in an entry box, press `<Return>`. The entry box will then turn back to its original colour. If the value you provided is valid for the given parameter, the value will be kept. If the value provided is invalid, the entry box will revert back to the previous valid value.

Note that if you try to run the estimation routine while at least one entry box has not be validated, you will be prevented from doing so:



Region Selection

For typical NMR signals, the estimation routine used in NMR-EsPy is too expensive to analyse the entire signal. For this reason, it is typically necessary to generate a signal which has been frequency-filtered, drastically reducing the computation time, and increasing the accuracy of the estimation for the region chosen. As a rule of thumb, try to choose a region with fewer than 30 peaks. Any more than this, and the routine may take too long for you to bear.

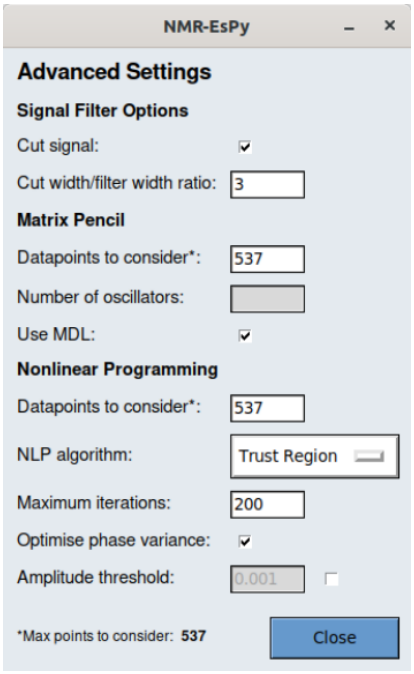
To filter the signal, two regions of the spectrum need to be indicated:

- The region to estimate, highlighted in green.
- A region which appears to contain no signals (i.e. is just experimental noise), highlighted in blue.

These regions can be adjusted by editing the scale widgets and entry boxes in the *Region Selection* tab.

Advanced Estimation Settings

Clicking the *Advanced Settings* button will load a window enabling various aspects of the estimation routine to be tweaked:

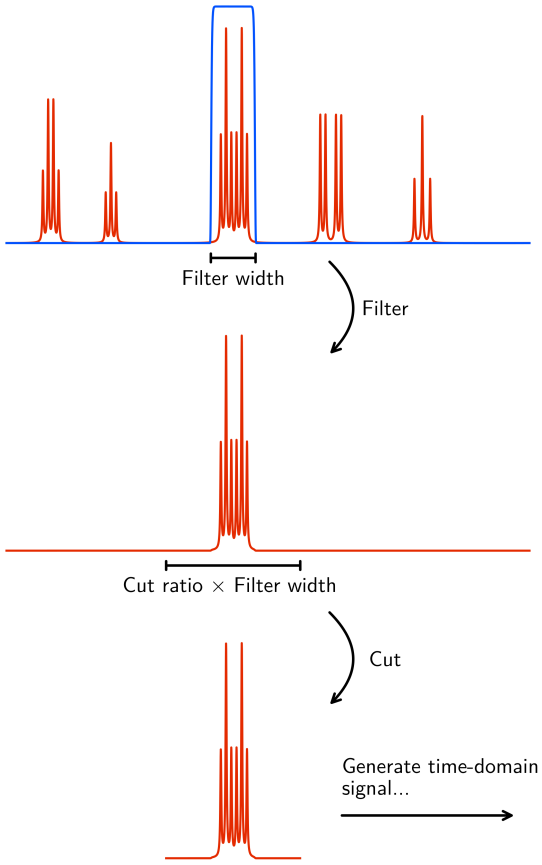


Below is a summary of the meaning of all of these parameters.

Note: For the majority of cases, you should find that the default parameters provided will be suitable.

Signal Filter Options

The basic idea behind frequency-filtering the data is to apply a band-pass filter to the spectral data, and then to convert the spectrum back to the time domain. By applying this filter, a substantial amount of the spectrum becomes redundant, and so it can be appropriate the "cut" off regions that are not of interest. The basic idea is illustrated in this figure:



- *Cut signal* - Specifies whether or not to perform cutting of the spectrum. By default, this is selected.
- *Cut width/filter width ratio* - Specifies how many points the cut signal will be composed of relative to the number of points the filter spans. This is set to 3 by default.

Matrix Pencil Method Options

The Matrix Pencil Method (MPM) is a singular-value decomposition-based approach for estimating signal parameters. It is used in NMR-EsPy to generate an initial guess for numerical optimisation. It is possible to either manually choose how many oscillators to generate using the MPM, or to estimate the number of oscillators using the Minimum Description Length (MDL).

- *Datapoints to consider* - Specifies how many points in the filtered signal to consider. The fewer datapoints, the faster the MPM will be. However, if too few datapoints are used, the result may be unreliable. If the signal contains fewer than 4096 (2^{12}) points, the full signal will be considered by default. Otherwise, the first 4096 points will be considered.
- *Use MDL* - Whether or not to use the Minimum Description Length. By default, the MDL will be used.
- *Number of Oscillators* - The number of oscillators used in the MPM. This can only be specified if *Use MDL* is unticked.

Nonlinear Programming Options

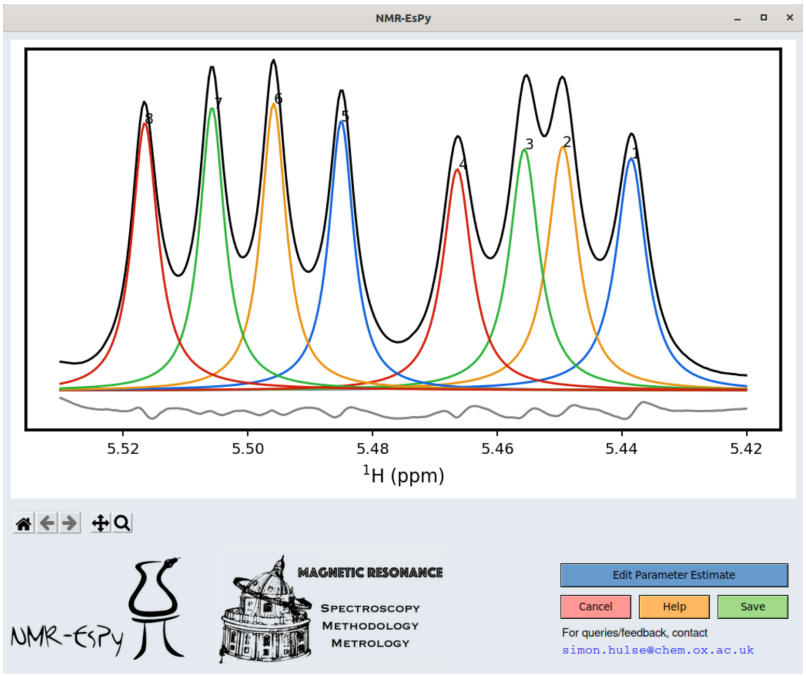
The result of the Matrix Pencil Method is fed into a nonlinear programming (NLP) routine to determine the final signal parameter estimate.

- *Datapoints to consider* - Analogous to the parameter with the same name under **Matrix Pencil**. The cost of running NLP is less susceptible to increases in the number of datapoints, so the full signal will be analysed by default if it comprises 8192 (2^{13}) points or fewer. Otherwise, the signal's first 8192 points will be considered by default.
- *NLP algorithm* - The optimisation routine. This can be either *Trust Region* or *L-BFGS*. By default, Trust-Region is used. The primary difference between these methods is that for Trust-Region, the Hessian matrix (a matrix of second order derivatives) is computed explicitly. In L-BFGS, the Hessian is approximated. The upshot of this is that the Trust-Region routine tends to lead to convergence in fewer iterations, however each iteration takes longer to compute.
- *Maximum iterations* - The largest number of iterations to perform before terminating an returning the optimiser. The default value is dependent on the NLP algorithm used (200 if Trust-Region selected, 500 if L-BFGS selected).
- *Optimise phase variance* - Specifies whether to consider the variance of oscillator phases during the estimation routine. If your data is derived from a well-phased spectrum, it is advised you have this selected.
- *Amplitude threshold* - Sets a threshold, such that any oscillator in the final result with an amplitude below the threshold will be removed. The threshold is defined as $a_{\text{thold}}\|a\|_2$ where $\|a\|_2$ is the Euclidian norm of the oscillator amplitudes, and a_{thold} is the specified threshold. By default, no such threshold will be applied to the data.

Once you are happy with the calculation setup, simply click the *Run* button. You will find that details of the routine are output to the terminal as it runs.

4.2.3 Estimation Result

Once the routine is complete, a new window will load with the following appearance:



Featured in the result plot are:

- The data selected (black).
- Individual peaks that comprise the estimation result (multi-coloured). Each of these is given a numerical label.
- The residual between the data and the model (grey).

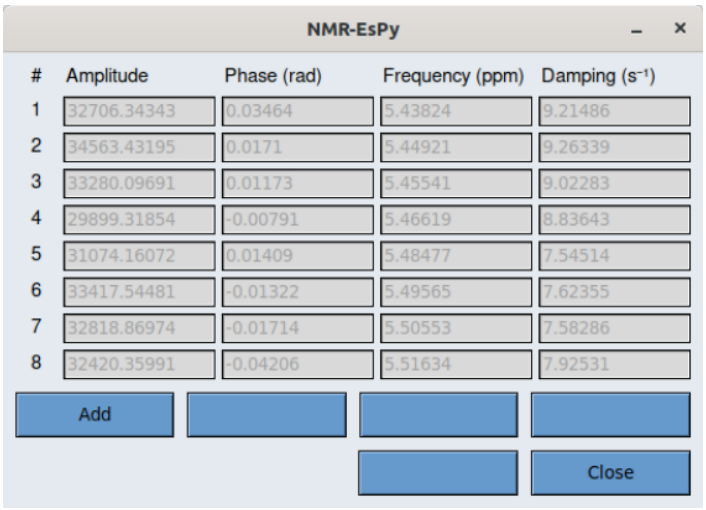
Tweaking and Re-optimising the result

There may be circumstances where you feel that the estimation result has not succeeded in describing certain aspects of the data well. For example, it may fit two very closely-separated resonances with a single oscillator.

The *Edit Parameter Estimate* button provides functionality to manually change the estimation result. Upon making such changes, the optimiser should be re-run to achieve a minimisation of the cost function of interest.

> **Warning:** This feature is present to fix "glaring" incorrect features of the estimation result. It is not intended for liberal "fudging" of the result.

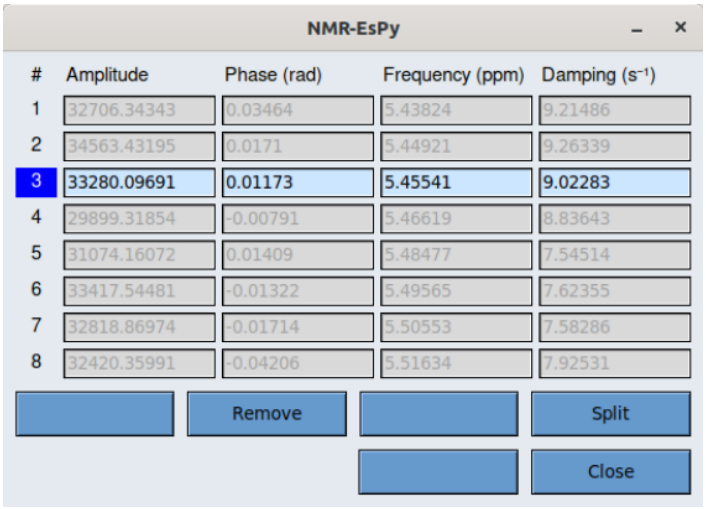
A window with the following appearance will appear after you click *Edit Parameter Estimate*:



Each oscillator is listed with its associated parameters. The numerical values assigned to each oscillator match those in the result figure.

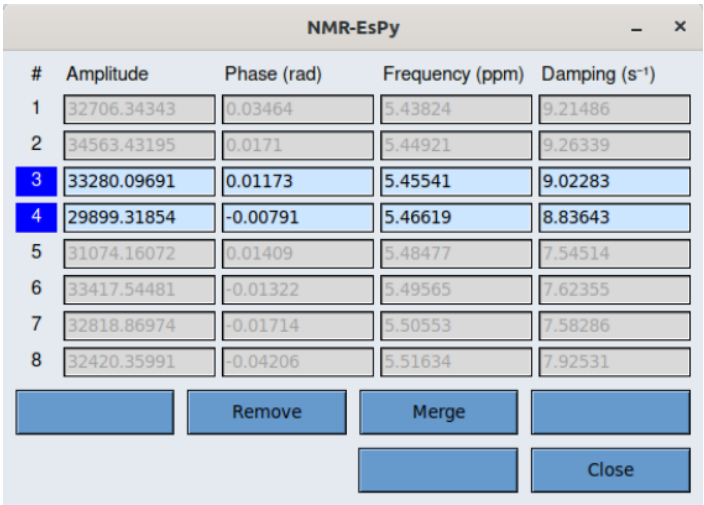
Initially, no oscillators are selected. When this is the case, two buttons are active: *Add*, which allows you to add extra oscillators, and *Close*, which closes the window.

To select and oscillator, left-click the associated numerical label. This will highlight the oscillator. Here is an example after oscillator 3 is clicked:



To de-select the oscillator, simply left-click the numerical label again. Two new buttons are activated when one oscillator is selected: *Remove*, which will purge the selected oscillator from the result, and *Split*, which allows you to create "child oscillators" with the same cumulative amplitude as the parent.

To select multiple oscillators at a single time, left-click on each oscillator label whilst holding `<Shift>`:



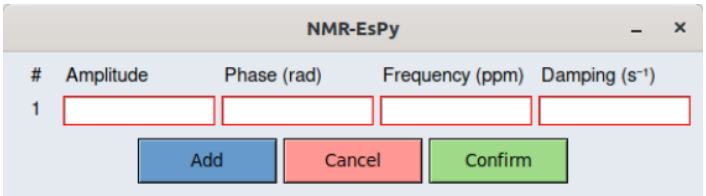
When more than one oscillator is selected, the *Merge* button is activated, along with the Remove button.

The Add Button

The Add button allows you to add an oscillator with arbitrary parameters to the estimation result. The main circumstance that this may be useful is when there is a low-intensity oscillator in the data which the estimation routine has failed to identify. Extensive use of this button is not advised.

Note: The add button provides exactly the same functionality as *`add_oscillators()`*.

Clicking on the Add button when no oscillators are selected will load the following window:



You need to input the desired parameters that make up the oscillator to be added. Each entry box needs to be validated by pressing `<Return>` after inputting the desired value:

- Amplitudes must be positive.
- Phases may be any numerical value. The value you provide will be wrapped to be in the range $(-\pi, \pi]$.
- Frequencies must be within the spectral range of the data considered.
- Damping factors must be positive.

If you wish to include more than one extra oscillator, click the *Add* button. This will append an extra row to the table. When you have a parameter table with all entry boxes validated (i.e. none of them are red), click *Confirm* to append the changes to the result. If you want to quit the window without making any changes, press *Cancel*.

The Remove Button

If one or more oscillators are selected, the Remove button will purge these from the result.

Note: The add button provides exactly the same functionality as *remove_oscillators()*.

The Merge Button

With more than one oscillator selected, the merge button will remove all selected oscillators, and create a single oscillator with parameters that reflect the selected oscillators. The new oscillator's amplitude will be the sum of the selected oscillators, and the other parameters will be the mean of the selected oscillators.

The main use of this is to merge a "superfluous" set of oscillators which are modelling a single resonance in the data.

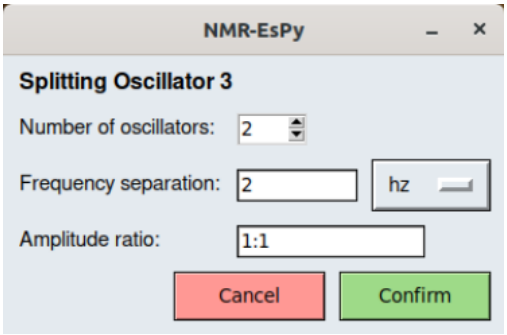
Note: The add button provides exactly the same functionality as *merge_oscillators()*.

The Split Button

With one oscillator selected, the Split button will purge the oscillator and in its place create a series of "child" oscillators.

Note: The add button provides exactly the same functionality as *split_oscillator()*.

The following window loads when you click *Split*:



- The *Number of oscillators* box specifies how many child oscillators to generate.
- The *Frequency separation* box specifies how far apart adjacent child oscillators will be. You can choose the units to be in Hz or ppm. By default, it is set at 2Hz
- The *Amplitude ratio* box specifies the relative amplitudes of the oscillators. A valid input for this box takes the form of *n* integers, each separated by a colon, where *n* is the value in the *Number of oscillators* box. By default, this will be set to be *n* 1s separated by colons, such that all child oscillators will have the same amplitude.

  Note: If you are familiar with regular expressions, the value in the Amplitude ratio box should match `^d+(:d+){n}$`.

To enact the splitting, click *Confirm*.

Re-running the Optimiser

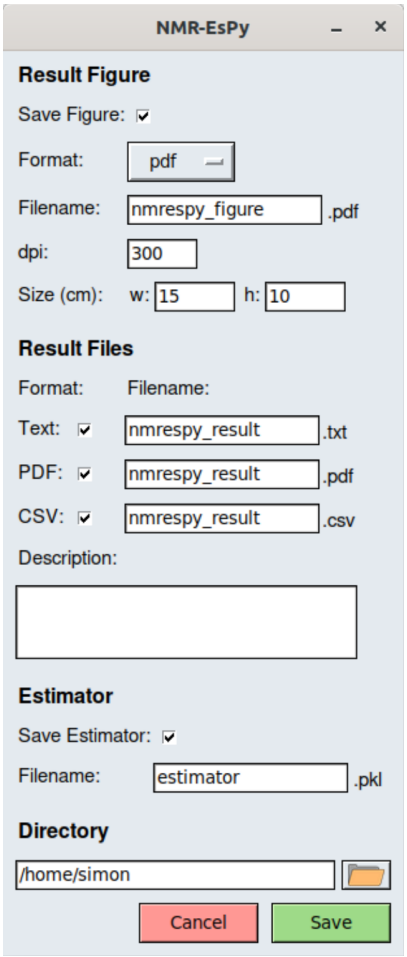
If you have made any changes to the estimation result, you will notice that the bottom right button has changed from *Close* to *Re-run optimiser*. As well as this, the *Reset* button to the left has been activated. NMR-EsPy does not allow you to save an estimation result for which the last step of the process was manual editing of the result. As such, if you wish to enact the changes made, you have to re-run nonlinear programming with the current result as the initial guess.

Undoing changes

If you decide that you want to undo all the changes made in the *Edit Parameter Estimate* window, simply click the Reset button.

Saving the result

Clicking the *Save* button loads the following window:



Result Figure

This section is used for specifying whether to save a result figure, and for customising some simple figure settings.

- *Save Figure* - Whether to save a figure or not.
- *Format* - The figure's file format. Valid options are `eps`, `png`, `pdf`, `jpg`, `ps` and `svg`.
- *Filename* - The name of the file to save the Figure to.
- *dpi* - Dots per inch.
- *Size (cm)* - The width and height of the figure, in centimeters.

Note: The most up-voted answer to this Stack Overflow question provides a good description of the relationship between figure size and dpi.

Note: Beyond specifying the dpi and size of the figure, the GUI does not provide any means of customising the appearance of the figure in this version. I intend to provide support of for in a future version. At the moment, the only means of customising the figure is to do it by writing a Python script. I provide an outline of how you can achieve certain customisations *here*

Result Files

Used for saving a table of result parameters to various file formats. For each of the valid formats (`txt`, `pdf`, and `csv`), the associated tick-boxes are used for specifying whether or not to generate a file of that format. Adjacent to each tick-box is an entry box for specifying the name of the result file.

Finally, the *Description* box can be used to enter a description relating to the estimation, which will be added to the result file(s).

Estimator

Used for saving ("pickling") the *`nmrespy.core.Estimator`* class instance, associated with the estimation result.

- *Save Estimator* - Specifies whether or not to save the estimator to a binary file.
- *Filename* - The filename to save the estimator to.

Directory

The entry box is used to specify the path to the directory to save **all** specified files to. The full path can either be typed out manually, or selected, by loading the file navigation window, by pressing the button with a folder icon.

Clicking *Save* will result in all the specified files to be saved to the desired paths. The application will also be closed.

CHAPTER

FIVE

MISCELLANEOUS TOPICS

Here you will find information about various topics which don't quite fall neatly into the other sections of the documentation.

5.1 Figure Customisation

On this page, you can find out how to customise result figures to suit your preferences. It will help to have some experience with matplotlib.

Editing result figures is the only major feature that the NMR-EsPy GUI does not provide support for, so if you use the GUI and want to make changes to the figure you'll have to do this manually with a Python script I'm afraid. I hope to provide support for figure customisation within the GUI in a later version.

Note: The next two sections are intended for GUI users. If you manually write scripts for the full estimation process, you can jump to *Generate the Estimation Figure*

5.1.1 Getting Started

If you are using the GUI, when saving the result of an estimation routine, make sure you "pickle" the estimator (see the section *Saving the Result* on *this page*).

You will now have to create a Python script, which will perform the following:

- Load the estimator (an instance of the `Estimator` class).
- Generate a figure of the estimator using the `plot_result` method.
- Customise the properties of the figure.
- Save the figure.

At the top of the script, place the following:

```
#!/usr/bin/python3
from nmrespy.core import Estimator
```

5.1.2 Load the Estimator

To load the estimator, include the following line:

```
estimator = Estimator.from_pickle("path/to/estimator")
```

where `path\to\estimator` is the full or relative path to the saved estimator file. **DO NOT INCLUDE THE `.pkl` EXTENSION**.

5.1.3 Generate the Estimation Figure

To generate the estimator figure, add the following line:

```
plot = estimator.plot_result()
```

Giving the *`plot_result()`* method no arguments will produce a figure which is identical to the one produced by using the GUI. There are some things that you can customise by providing arguments to the method. Notable things that can be tweaked are:

- The frequency unit of the spectrum (available options are ppm (default) or Hz).
- Whether or not to include a plot of the model (sum of individual oscillators) and/or a plot of the residual (difference between the data and the model).
- The colours of the data, residual, model, and individual oscillators.
- The vertical positioning of the residual and the model.
- Whether or not to include oscillator labels
- If you are familiar with matplotlib and the use of stylesheets, you can also specify a path to a stylesheet. One thing to note with stylesheets is that even if you have a colour cycle specified in the sheet, it will be overwritten, so you must still manually specify the desired oscillator colours if you want these to not be the default colours.

Have a look at the *`nmrespy.plot.plot_result()`* function for a description of the acceptable arguments. Only the following arguments should be used (the others are automatically determined internally by the *`nmrespy.core.Estimator.plot_result()`* method):

- *shifts_unit*
- *plot_residual*
- *plot_model*
- *residual_shift*
- *model_shift*
- *data_color*
- *oscillator_colors*
- *residual_color*
- *model_color*
- *labels*
- *stylesheet*

5.1.4 Further Customisation

If there are further features you would like to customise, this can be done by editing `plot`.

General Guidance

`nmrespy.core.Estimator.plot_result()` produces an instance of the *`nmrespy.plot.NmrespyPlot`* class. This possesses four notable attributes:

- *fig* (matplotlib.figure.Figure) the overall figure.
- *ax* (matplotlib.axes._subplots.AxesSubplot) the figure axes.
- *lines* (A dictionary of matplotlib.lines.Line2D objects).
- *labels* (A dictionary of matplotlib.text.Text objects).

The *lines* dictionary will possess the following keys:

- *'data'*
- *'residual'* (if the residual was chosen to be plotted)
- *'model'* (if the model was chose to be plotted)
- For each oscillator, the corresponding line is given a numerical value as its key (i.e. keys will be `1`, `2`, to `<M>` where `<M>` is the number of oscillators in the result.)

As an example, to access the line object corresponding to oscillator 6, and change its line-width to 2 px, you should use `plot.lines[6].set_linewidth(2)`.

The *labels* dictionary will possess numerical keys from `1` to `<M>` (same as oscillator keys in the *lines* dictionary).

Specific Things

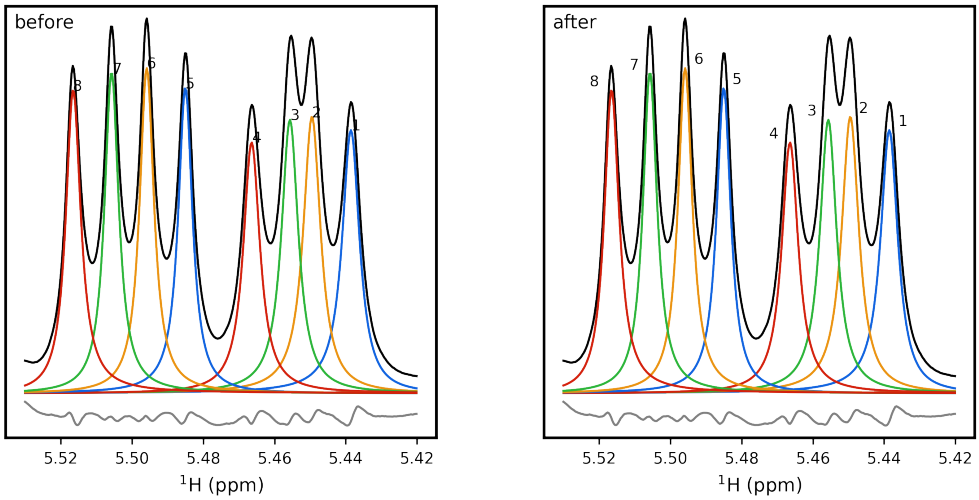
I have written some "convenience methods" to achieve certain things that I anticipate users will frequently want to carry out.

- **Re-positioning oscillator labels** Often, the automatically-assigned positions of the numerical labels that are given to each oscillator can overlap with other items in the figure, which is not ideal. To re-position the oscillator labels, use the *`displace_labels()`* method:

```
# Load the estimation result and create the plot (see above)
path = "/path/to/estimator"
estimator = Estimator.from_pickle(path)
plot = estimator.plot_result()

# Shift the labels for oscillators 1, 2, 5 and 6
# to the right and up
plot.displace_labels([1,2,5,6], (0.02, 0.01))

# Shift the labels for oscillators 3, 4, 7 and 8
# to the left and up
plot.displace_labels([3,4,7,8], (-0.05, 0.01))
```

Note: The size of displacement is given using the axes co-ordinate system

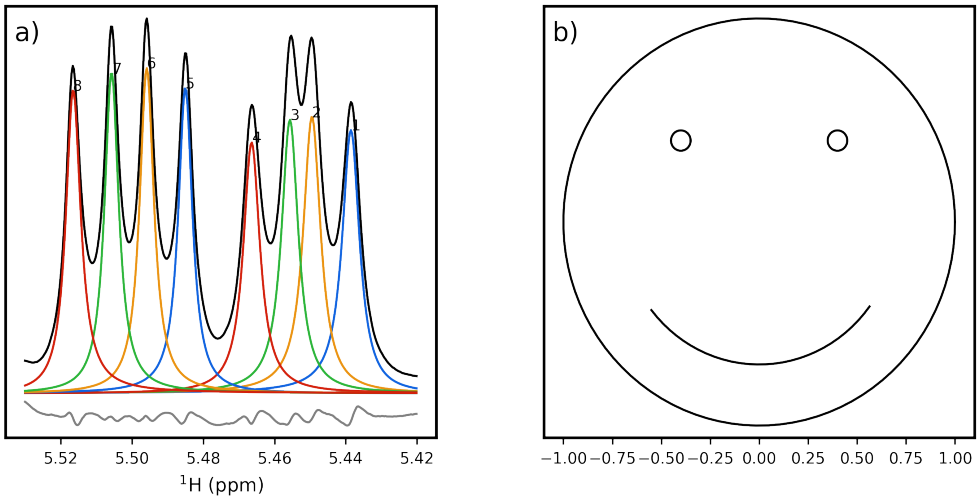
The plot before and after shifting the label positions are as follows:



- **Appending a result plot to a subplot** You may wish to set the result plot as a subplot amongst other plots that make up a figure. The *transfer_to_axes()* method enables this to be achieved very easily. Here is a simple example:

```
from nmrespy.core import Estimator
import numpy as np
import matplotlib.pyplot as plt

# Load the estimation result and create the plot (see above)
path = "estimator"
estimator = Estimator.from_pickle(path)
plot = estimator.plot_result()

# Create a simple figure with two subplots (side-by-side)
fig = plt.figure(figsize=(8, 4))
ax0 = fig.add_axes([0.05, 0.15, 0.4, 0.8])
ax1 = fig.add_axes([0.55, 0.15, 0.4, 0.8])

# Add some random stuff to ax1
theta = np.linspace(0, 2 * np.pi, 100)
x = np.cos(theta)
y = np.sin(theta)
ax1.plot(x, y, color='k')
ax1.plot(0.7 * x[60:90], 0.7 * y[60:90], color='k')
ax1.plot(0.05 * x - 0.4, 0.05 * y + 0.4, color='k')
ax1.plot(0.05 * x + 0.4, 0.05 * y + 0.4, color='k')

# Transfer contents of the result plot to ax0
plot.transfer_to_axes(ax0)

# Add label to each axes
# N.B. This has been done after the call to `transfer_to_axes`
# If it had been done beforehand, the a) label would have been cleared.
for txt, ax in zip(('a)', 'b)'), (ax0, ax1)):
    ax.text(0.02, 0.92, txt, fontsize=14, transform=ax.transAxes)

plt.show()
```

The resulting figure is as follows:



If there are other features that you would like to see added to the *NmrespyPlot()* class to increase the ease of generating figures, feel free to make a pull request, or get in touch.

5.1.5 Save the Figure

Once you have completed all the desired customisation, the `fig` object can be saved using matplotlib.savefig:

```
plot.fig.savefig("<path/to/figure>", ...)
```

CHAPTER

SIX

REFERENCE

6.1 nmrespy.core

class nmrespy.core.**Estimator**(*source*, *data*, *path*, *sw*, *off*, *sfo*, *nuc*, *fmt*, *_origin=None*)

Estimation class

Note: The methods *new_bruker()*, new_synthetic_from_data() and new_synthetic_from_parameters() generate instances of the class. The method *from_pickle()* loads an estimator instance that was previously saved using *to_pickle()*. While you can manually input the listed parameters as arguments to initialise the class, it is more straightforward to use one of these.

Parameters

- **source** (*{'bruker', 'synthetic'}*) – The type of data imported.
- **data** (*numpy.ndarray*) – The data associated with the binary file in *path*.
- **path** (*pathlib.Path or None*) – The path to the directory contaioning the NMR data.
- **sw** (*[float] or [float, float]*) – The experiment sweep width in each dimension (Hz).
- **offset** (*[float] or [float, float]*) – The transmitter's offset frequency in each dimension (Hz).
- **sfo** (*[float] or [float, float] or None*) – The transmitter frequency in each dimension (MHz)
- **nuc** (*[str] or [str, str] or None*) – The nucleus in each dimension. Elements will be of the form *'<mass><element>'*, where *'<mass>'* is the mass number of the isotope and *'<element>'* is the chemical symbol of the element.
- **fmt** (*str or None*) – The format of the binary file from which the data was obtained. Of the form *'<endian><unitsize>'*, where *'<endian>'* is either *'<'* (little endian) or *'>'* (big endian), and *'<unitsize>'* is either *'i4'* (32-bit integer) or *'f8'* (64-bit float).
- **_origin** (*dict or None, default None*) – For internal use. Specifies how the instance was initalised. If *None*, implies that the instance was initialised manually, rather than using one of *new_bruker()*, new_synthetic_from_data() and new_synthetic_from_parameters().

_check_if_none(*name*, *kill*, *method=None*)

Retrieve attributes that may be assigned the value *None*. Return None/raise error depending on the value of kill

Parameters

- **name** (*str*) – The name of the attribute requested.

- **kill** (*bool*) – Whether or not to raise an error if the desired attribute is *None*.
- **method** (*str or None, default: None*) – The name of the method that needs to be run to obtain the desired attribute. If *None*, it implies that the attribute requested was never given to the class in the first place.

Returns **attribute** – The attribute requested.

Return type any

_get_array(*name*, *kill*, *freq_unit*)
Returns an array (result or errors), wioth frequencies in either Hz or ppm

_get_data_sw_offset()
Retrieve data, sweep width and offset, based on whether frequency filtration have been applied.

Returns

- **data** (*numpy.ndarray*)
- **sw** (*[float] or [float, float]*) – Sweep width (Hz).
- **offset** (*[float] or [float, float]*) – Transmitter offset (Hz).

Notes

- If *self.filter_info* is equal to *None*, *self.data* will be analysed
- If *self.filter_info* is an instance of *nmrespy.freqfilter.FrequencyFilter*, *self.filter_info.filtered_signal* will be analysed.

add_oscillators(*oscillators*)
Adds new oscillators an estimation result.

Parameters **oscillators** (*numpy.ndarray*) – An array of the new oscillator(s) to add to the array. *NB oscillators* should always be a two-dimensional array, even if only one oscillator is being added:

```
>>> oscillators = np.array([[a, φ, f, η]]) # 1D
>>> oscillators = np.array([[a, φ, f1, f2, η1, η2]]) # 2D
>>> # or, equivalently:
>>> oscillators = np.insert_axis(
...     np.array([a, φ, f, η]), axis=1
... ) # 1D
>>> oscillators = np.insert_axis(
...     np.array([a, φ, f1, f2, η1, η2]), axis=1
... ) # 2D
```

frequency_filter(*region*, *noise_region*, *cut=True*, *cut_ratio=3.0*, *region_unit='ppm'*)
Generates frequency-filtered data from *self.data*.

Parameters

- **region** (*[[int, int]], [[int, int], [int, int]], [[float, float]] or [[float, float], [float, float]]*) – Cut-off points of the spectral region to consider. If the signal is 1D, this should be of the form *[[a,b]]* where *a* and *b* are the boundaries. If the signal is 2D, this should be of the form *[[a,b], [c,d]]* where *a* and *b* are the boundaries in dimension 1, and *c* and *d* are the boundaries in dimension 2. The ordering of the bounds in each dimension is not important.

- **noise_region** (*[[int, int]], [[int, int], [int, int]], [[float, float]] or [[float, float], [float, float]]*) – Cut-off points of the spectral region to extract the spectrum's noise variance. This should have the same structure as *region*.
- **cut** (*bool, default: True*) – If *False*, the filtered signal will comprise the same number of data points as the original data. If *True*, prior to inverse FT, the data will be sliced, with points not in the region specified by *cut_ratio* being removed.
- **cut_ratio** (*float, default: 2.5*) – If cut is *True*, defines the ratio between the cut signal's sweep width, and the region width, in each dimesnion. It is reccommended that this is comfortably larger than *1.0*. *2.0* or higher should be appropriate.
- **region_unit** (*'ppm', 'hz' or 'idx', default: 'ppm'*) – The unit the elements of *region* and *noise_region* are expressed in.

Notes

This method assigns the attribute *filter_info* to an instance of *nmrespy.freqfilter.FrequencyFilter*. To obtain information on the filtration, use *get_filter_info()*.

classmethod from_pickle(*path*)

Loads an intance of *Estimator*, which was saved previously using *to_pickle()*.

Parameters **path** (*str*) – The path to the pickle file. **DO NOT INCLUDE THE FILE EXTENSION.**

Returns **estimator**

Return type *Estimator*

Notes

> **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 instance of *Estimator*, an error will be raised.

get_bf(*kill=True*)

Return the transmitter's basic frequency for each channel (MHz).

Parameters **kill** (*bool, default: True*) – If the path is *None*, *kill* specifies how the method will act:

- If *True*, an AttributeIsNoneError is raised.
- If *False*, *None* is returned.

Returns **bf**

Return type [float] or [float, float]

get_data()

Return the original data.

Returns **data**

Return type numpy.ndarray

get_datapath(*type_='Path'*, *kill=True*)
Return path of the data directory.

Parameters

- **type** (*'Path' or 'str', default: 'Path'*) – The type of the returned path. If *'Path'*, the returned object is an instance of pathlib.Path. If *'str'*, the returned object is an instance of str.
- **kill** (*bool, default: True*) – If the path is *None*, *kill* specifies how the method will act:
  - If *True*, an AttributeIsNoneError is raised.
  - If *False*, *None* is returned.

Returns **path**

Return type str or pathlib.Path

get_dim()
Return the data dimension.

Returns **dim**

Return type 1 or 2

get_errors(*kill=True*, *freq_unit='hz'*)
Returns the errors of the estimation result derived from *nonlinear_programming()*

Parameters

- **kill** (*bool, default: True*) – If *self.errors* is *None*, *kill* specifies how the method will act:
  - If *True*, an AttributeIsNoneError is raised.
  - If *False*, *None* is returned.
- **freq_unit** (*'hz' or 'ppm', default: 'hz'*) –

get_filter_info(*kill=True*)
Returns information relating to frequency filtration.

Parameters **kill** (*bool, default: True*) – If *filter_info* is *None*, and *kill* is *True*, an error will be raised. If *kill* is False, *None* will be returned.

Returns **filter_info**

Return type *nmrespy.freqfilter.FrequencyFilter*

Notes

There are numerous methods associated with *filter_info* for obtaining relavent infomation about the filtration. See *nmrespy.freqfilter.FrequencyFilter* for details.

get_n()
Return the number of datapoints in each dimension

Returns **n**

Return type [int] or [int, int]

get_nucleus(*kill=True*)
Return the target nucleus of each channel.

Parameters **kill** (*bool, default: True*) – If the path is *None*, *kill* specifies how the method will act:

- If *True*, an AttributeIsNoneError is raised.
- If *False*, *None* is returned.

Returns **nuc**

Return type [str] or [str, str]

get_offset(*unit='hz'*, *kill=True*)

Return the transmitter's offset frequency in each dimesnion.

Parameters **unit** (*'hz' or 'ppm', default: 'hz'*) –

Returns

- **offset** (*[float] or [float, float]*)
- **kill** (*bool, default: True*) – If *unit* is *'ppm'*, but *self.sfo* is *None*, *kill* specifies how the method will act:
  - If *True*, an AttributeIsNoneError is raised.
  - If *False*, *None* is returned.

Raises ***InvalidUnitError*** – If *unit* is not *'hz'* or *'ppm'*

Notes

If *unit* is set to *'ppm'* and *self.sfo* is not specified (*None*), there is no way of retreiving the offset in ppm. *None* will be returned.

get_result(*kill=True*, *freq_unit='hz'*)

Returns the estimation result

Parameters

- **kill** (*bool, default: True*) – If *self.result* is *None*, *kill* specifies how the method will act:
  - If *True*, an AttributeIsNoneError is raised.
  - If *False*, *None* is returned.
- **freq_unit** (*'hz' or 'ppm', default: 'hz'*) –

get_sfo(*kill=True*)

Return transmitter frequency for each channel (MHz).

Parameters **kill** (*bool, default: True*) – If the path is *None*, *kill* specifies how the method will act:

- If *True*, an AttributeIsNoneError is raised.
- If *False*, *None* is returned.

Returns **sfo**

Return type [float] or [float, float]

get_shifts(*unit='hz'*, *meshgrid=False*, *kill=True*)

Return the sampled frequencies consistent with experiment's parameters (sweep width, transmitter offset, number of points).

Parameters

- **unit** (*'ppm' or 'hz', default: 'ppm'*) – The unit of the value(s).
- **meshgrid** (*bool*) – Only appicable for 2D data. If set to *True*, the shifts in each dimension will be fed into numpy.meshgrid
- **kill** (*bool*) – If *self.sfo* (need to get shifts in ppm) is *None*, *kill* specifies how the method will act:
  - If *True*, an AttributeIsNoneError is raised.
  - If *False*, *None* is returned.

Returns **shifts** – The frequencies sampled along each dimension.

Return type [numpy.ndarray] or [numpy.ndarray, numpy.ndarray]

Raises ***InvalidUnitError*** – If *unit* is not *'hz'* or *'ppm'*

Notes

The shifts are returned in ascending order.

get_sw(*unit='hz'*, *kill=True*)

Return the experiment sweep width in each dimension.

Parameters

- **unit** (*'hz' or 'ppm', default: 'hz'*) –
- **kill** (*bool, default: True*) – If *unit* is *'ppm'*, but *self.sfo* is *None*, *kill* specifies how the method will act:
  - If *True*, an AttributeIsNoneError is raised.
  - If *False*, *None* is returned.

Returns **sw**

Return type [float] or [float, float]

Raises ***InvalidUnitError*** – If *unit* is not *'hz'* or *'ppm'*

Notes

If *unit* is set to *'ppm'* and *self.sfo* is not specified (*None*), there is no way of retreiving the sweep width in ppm. *None* will be returned.

get_timepoints(*meshgrid=False*)

Return the sampled times consistent with experiment's parameters (sweep width, number of points).

Parameters **meshgrid** (*bool*) – Only appicable for 2D data. If set to *True*, the time-points in each dimension will be fed into numpy.meshgrid

Returns **tp** – The times sampled along each dimension (seconds).

Return type [numpy.ndarray] or [numpy.ndarray, numpy.ndarray]

logger()

Decorator for logging *Estimator* method calls

make_fid(*n=None*, *oscillators=None*, *kill=True*)

Constructs a synthetic FID using a parameter estimate and experiment parameters.

Parameters

- **n** (*[int], or [int, int], or None default: None*) – The number of points to construct the FID with in each dimesnion. If *None*, *get_n()* will be used, meaning the signal will have the same number of points as the original data.
- **oscillators** (*None or list, default: None*) – Which oscillators to include in result. If *None*, all oscillators will be included. If a list of ints, the subset of oscillators corresponding to these indices will be used. Note that all elements should be in `range(self.result.shape[0])`.
- **kill** (*bool, default: True*) – If *self.result* is *None*, *kill* specifies how the method will act:
  - If *True*, an AttributeIsNoneError is raised.
  - If *False*, *None* is returned.

Returns

- **fid** (*numpy.ndarray*) – The generated FID.
- **tp** (*[numpy.ndarray] or [numpy.ndarray, numpy.ndarray]*) – The time-points at which the signal is sampled, in each dimension.

See also:

nmrespy.sig.make_fid()

manual_phase_data(*max_p1=None*)

Perform manual phase correction of *self.data*.

Zero- and first-order phase pharameters are determined via interaction with a Tkinter- and matplotlib-based graphical user interface.

Parameters **max_p1** (*float or None, default: None*) – Specifies the range of first-order phases permitted. For each dimension, the user will be allowed to choose a value of *p1* within [*-max_p1*, *max_p1*]. By default, *max_p1* will be `10 * numpy.pi`.

matrix_pencil(*M=0*, *trim=None*, *fprint=True*)

Implementation of the 1D Matrix Pencil Method[1][2] or 2D Modified Matrix Enchancement and Matrix Pencil (MMEMP) method[3][4] with the option of Model Order Selection using the Minumum Descrition Length (MDL)[5].

Parameters

- **M** (*int, default: 0*) – The number of oscillators to use in generating a parameter estimate. If *M* is set to *0*, the number of oscillators will be estimated using the MDL.
- **trim** (*[int], [int, int], or None, default: None*) – If *trim* is a list, the analysed data will be sliced such that its shape matches *trim*, with the initial points in the signal being retained. If *trim* is *None*, the data will not be sliced. Consider using this in cases where the full signal is large, such that the method takes a very long time, or your PC has insufficient memory to process it.

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

- **fprint** (*bool, default: True*) – If *True* (default), the method provides information on progress to the terminal as it runs. If *False*, the method will run silently.

Notes

The data analysed will be the following:

- If *self.filter_info* is equal to *None*, *self.data* will be analysed
- If *self.filter_info* is an instance of *nmrespy.freqfilter.FrequencyFilter*, *self.filter_info.filtered_signal* will be analysed.

For developers: See *_get_data_sw_offset()*

Upon successful completion is this method, *self.mpm_info* will be updated with an instance of *nmrespy.mpm.MatrixPencil*.

References

merge_oscillators(*indices*)

Merges the oscillators corresponding to *indices*.

Removes the osccilators specified, and constructs a single new oscillator with a cumulative amplitude, and averaged phase, frequency and damping.

Parameters **indices** (*list, tuple or numpy.ndarray*) – A list of indices corresponding to the oscillators to be merged. The elements of *indices* should be ints that are in `range(result.shape[0])`, where *result* is the current estimation result.

Notes

Assuming that an estimation result contains a subset of oscillators denoted by indices $\{m_1, m_2, \cdots, m_J\}$, where $J \leq M$, the new oscillator formed by the merging of the oscillator subset will possess the following parameters:

- $a_{\text{new}} = \sum_{i=1}^{J} a_{m_i}$
- $\phi_{\text{new}} = \frac{1}{J} \sum_{i=1}^{J} \phi_{m_i}$
- $f_{\text{new}} = \frac{1}{J} \sum_{i=1}^{J} f_{m_i}$
- $\eta_{\text{new}} = \frac{1}{J} \sum_{i=1}^{J} \eta_{m_i}$

classmethod new_bruker(*dir*, *ask_convdta=True*)

Generate an instance of *Estimator* from a Bruker-formatted data directory.

Parameters

- **dir** (*str*) – The path to the data containing the data of interest.
- **ask_convdta** (*bool*) – See `nmrespy.load_bruker()`

Returns **estimator**

Return type *Estimator*

Notes

For a more detailed specification of the directory requirements, see `nmrespy.load_bruker()`.

nonlinear_programming(*trim=None*, ***kwargs*)
Estimation of signal parameters using nonlinear programming, given an inital guess.

Parameters

- **trim** (*None, [int], or [int, int], default: None*) – If *trim* is a list, the analysed data will be sliced such that its shape matches *trim*, with the initial points in the signal being retained. If *trim* is *None*, the data will not be sliced. Consider using this in cases where the full signal is large, such that the method takes a very long time, or your PC has insufficient memory to process it.
- ****kwargs** – Properties of *nmrespy.nlp.nlp.NonlinearProgramming*. Valid arguments:
  - *phase_variance*
  - *method*
  - *bound*
  - *max_iterations*
  - *amp_thold*
  - *freq_thold*
  - *negative_amps*
  - *fprint*

  Other keyword arguments that are valid in *nmrespy.nlp.nlp.NonlinearProgramming()* will be ignored (these are generated internally by the class instance).

Raises ***PhaseVarianceAmbiguityError*** – Raised when `phase_variance` is set to `True`, but the user has specified that they do not wish to optimise phases using the `mode` argument.

Notes

The data analysed will be the following:

- If *self.filter_info* is equal to *None*, *self.data* will be analysed
- If *self.filter_info* is an instance of *nmrespy.freqfilter.FrequencyFilter*, *self.filter_info.filtered_signal* will be analysed.

Upon successful completion is this method, *self.result* and *self.errors* will be updated.

See also:

nmrespy.nlp.nlp.NonlinearProgramming

phase_data(*p0=None*, *p1=None*)
Phase *self.data*

Parameters

- **p0** (*[float], [float, float], or None default: None*) – Zero-order phase correction in each dimension in radians. If *None*, the phase will be set to *0.0* in each dimension.
- **p1** (*[float], [float, float], or None default: None*) – First-order phase correction in each dimension in radians. If *None*, the phase will be set to *0.0* in each dimension.

plot_result(***kwargs*)
Produces a figure of an estimation result.

The figure consists of the original data, in the Fourier domain, along with each oscillator.

Parameters **kwargs** (Properties of *nmrespy.write.write_result()*.) – Valid arguments are:

- *shifts_unit*
- *plot_residual*
- *plot_model*
- *residual_shift*
- *model_shift*
- *data_color*
- *oscillator_colors*
- *residual_color*
- *model_color*
- *labels*
- *stylesheet*

Other keyword arguments that are valid in *nmrespy.plot.plot_result()* will be ignored (these are generated internally by the class instance).

Raises ***AttributeIsNoneError*** – If no parameter estimate derived from nonlinear programming is found (see *nonlinear_programming()*).

See also:

nmrespy.plot.plot_result()

remove_oscillators(*indices*)
Removes the oscillators corresponding to `indices`.

Parameters **indices** (*list*) – A list of indices corresponding to the oscillators to be removed. The elements of *indices* should be ints that are in `range(result.shape[0])`, where *result* is the current estimation result.

save_logfile(*path='./nmrespy_log'*, *force_overwrite=False*)
Saves log file of class instance usage to a specified path.

Parameters

- **path** (*str, default: './nmrespy_log'*) – The path to save the file to. DO NOT INCLUDE A FILE EXTENSION. *.log* will be added automatically.
- **force_overwrite** (*bool. default: False*) – Defines behaviour if `f'{path}.log'` already exists:
  - If *force_overwrite* is set to *False*, the user will be prompted if they are happy overwriting the current file.
  - If *force_overwrite* is set to *True*, the current file will be overwritten without prompt.

split_oscillator(*index*, *separation_frequency=None*, *unit='hz'*, *split_number=2*, *amp_ratio=None*)
Splits the oscillator corresponding to *index*.

Removes an oscillator, and incorporates two or more oscillators whose cumulative amplitudes match that of the removed oscillator.

Parameters

- **index** (*int*) – Array index of the oscilator to be split.
- **separation_frequency** (*float, or None default: None*) – The frequency separation given to adjacent oscillators formed from the splitting. If *None*, the splitting will be set to `sw / n` where *sw* is the sweep width and *n* is the number of points in the data.
- **unit** (*'hz' or 'ppm', default: 'hz'*) – The unit of *separation_frequency*.
- **split_number** (*int, default: 2*) – The number of peaks to split the oscillator into.
- **amp_ratio** (*list or None, default: None*) – The ratio of amplitudes to be fulfilled by the newly formed peaks. If a list, `len(amp_ratio) == split_number` must be satisfied. The first element will relate to the highest frequency oscillator constructed, and the last element will relate to the lowest frequency oscillator constructed. If *None*, all oscillators will be given equal amplitudes.

to_pickle(*path='./estimator'*, *force_overwrite=False*, *fprint=True*)

Converts the class instance to a byte stream using Python's "Pickling" protocol, and saves it to a .pkl file.

Parameters

- **path** (*str, default: './estimator'*) – Path of file to save the byte stream to. **DO NOT INCLUDE A '.pkl" EXTENSION!** *'.pkl'* is added to the end of the path automatically.
- **force_overwrite** (*bool, default: False*) – Defines behaviour if `f'{path}.pkl'` already exists:
  - If *force_overwrite* is set to *False*, the user will be prompted if they are happy overwriting the current file.
  - If *force_overwrite* is set to *True*, the current file will be overwritten without prompt.
- **fprint** (*bool, default: True*) – Specifies whether or not to print infomation to the terminal.

Notes

This method complements *from_pickle()*, in that an instance saved using *to_pickle()* can be recovered by `pickle_load()`.

view_data(*domain='frequency'*, *freq_xunit='ppm'*, *component='real'*)

Generate a simple, interactive plot of the data using matplotlib.

Parameters

- **domain** (*'frequency' or 'time', default: 'frequency'*) – The domain of the sig.
- **freq_xunit** (*'ppm' or 'hz', default: 'ppm'*) – The unit of the x-axis, if *domain* is set as *'frequency'*. If *domain* is set as *'time'*, the x-axis unit will the seconds.
- **component** (*'real', 'imag' or 'both', default: 'real'*) – The component of the data to display. *'both'* displays both the real and imaginary components

write_result(***kwargs*)

Saves an estimation result to a file in a human-readable format (text, PDF, CSV).

Parameters **kwargs** (Properties of *nmrespy.write.write_result()*.) – Valid arguments are:

- *path*
- *description*

- *sig_figs*
- *sci_lims*
- *fmt*
- *force_overwrite*
- *fprint*

Other keyword arguments that are valid in `nmrespy.write.write_result()` will be ignored (these are generated internally by the class instance).

Raises `AttributeIsNoneError` – If no parameter estimate derived from nonlinear programming is found (see `nonlinear_programming()`).

See also:

`nmrespy.write.write_result()`

6.2 nmrespy.load

6.3 nmrespy.freqfilter

Frequecy filtration of NMR data using super-Gaussian band-pass filters

class nmrespy.freqfilter.**FrequencyFilter**(*data*, *region*, *noise_region*, *sw*, *offset*, *sfo*, *region_unit='idx'*, *cut=True*, *cut_ratio=3.0*)

Fequency filter class.

Parameters

- **data** (*numpy.ndarray*) – The time-domain signal.
- **region** (*[[int, int]], [[float, float]], [[int, int], [int, int]] or [[float, float], [float, float]]*) – Boundaries specifying the region to apply the filter to.
- **noise_region** ((Same type as *region*)) – Boundaries specifying a region which does not contain any noticable signals (i.e. just containing experiemntal noise).
- **region_unit** (*'idx', 'ppm' or 'hz', default: 'idx'*) – The units which the boundaries in *region* and *noise_region* are given in.
- **sw** (*[float], [float, float] or None, default: None*) – The sweep width of the signal in each dimension. Required as float list if *region_unit* is *'ppm'* or *'hz'*.
- **offset** (*[float], [float, float] or None, default: None*) – The transmitter offset in each dimension. Required as float list if *region_unit* is *'ppm'* or *'hz'*.
- **sfo** (*[float], [float, float] or None, default: None*) – The tansmitter frequency in each dimnesion (MHz). Required as float list if *region_unit* is *'ppm'* or *'hz'*.
- **p0** (*[float], [float, float], or None default: None*) – Zero-order phase correction in each dimension in radians. If *None*, the phase will be set to *0.0* in each dimension.
- **p1** (*[float] or [float, float], default: [0.0, 0.0]*) – First-order phase correction in each dimension in radians. If *None*, the phase will be set to *0.0* in each dimension.

- **cut** (*bool, default: True*) – If *True*, the filtered frequency-domain data will be trancated prior to inverse Fourier Transformation, reducing the number of signal points. If False, the data is not truncated after FT.
- **cut_ratio** (*float, default: 3.0*) – If *cut* is set to *True*, this gives the ratio of the cut signal's bandwidth and the filter bandwidth. This should be greater than 1.0.

Notes

Todo: Write me!

_get_region(*name*, *unit*)
Return either *region* or *noise_region*, based on *name*

Parameters

- **name** (*'region' or 'noise_region'*) – Name of attribute to obtain.
- **unit** (*'idx', 'hz', 'ppm'*) – Unit to express the region bounds in.

Returns region

Return type [[int, int]], [[int, int], [int, int]], [[float, float]], or [[float, float], [float, float]]

get_fid(*cut=True*)
Returns frequency-filtered time domain data.

Parameters **cut** (*bool, default: True*) – If *True*, and *cut* was set to *True* when the class was initialised, the FID derived from the cut, filtered spectrum is returned. Otherwise, the FID of the uncut spectrum is returned.

Returns fid

Return type numpy.ndarray

get_filtered_spectrum(*cut=True*)
Returns frequency-filtered spectral data.

Parameters **cut** (*bool, default: True*) – If *True*, and *cut* was set to *True* when the class was initialised, the cut, filtered spectrum is returned. Otherwise, the uncut spectrum is returned.

Returns filtered_spectrum

Return type numpy.ndarray

get_fs(*cut=True*)
Shorthand for *get_filtered_spectrum()*.

get_noise_region(*unit='idx'*)
Returns the spectral noise region selected

Parameters **unit** (*'idx', 'hz', 'ppm', default: 'idx'*) – Unit to express the region bounds in.

Returns noise_region

Return type [[int, int]], [[int, int], [int, int]], [[float, float]], or [[float, float], [float, float]]

get_offset(*unit='hz'*, *cut=True*)
Returns the offset of the cut signal

Parameters

- **unit** (*{'hz', 'ppm'}, default: 'hz'*) – Unit to express the sweep width in.
- **cut** (If *True*, and *cut* was set to *True* when the class was) – initialised, the offset of the cut, filtered signal is returned. Otherwise, the offset of the uncut signal is returned.

get_region(*unit='idx'*)

Returns the spectral region selected

Parameters **unit** (*'idx', 'hz', 'ppm', default: 'idx'*) – Unit to express the region bounds in.

Returns **region**

Return type [[int, int]], [[int, int], [int, int]], [[float, float]], or [[float, float], [float, float]]

get_sg()

Shorthand for *get_super_gaussian()*.

get_super_gaussian()

Returns the super-Gaussian filter used.

Returns **super_gaussian**

Return type numpy.ndarray

get_sw(*unit='hz', cut=True*)

Returns the sweep width of the cut signal

Parameters

- **unit** (*{'hz', 'ppm'}, default: 'hz'*) – Unit to express the sweep width in.
- **cut** (If *True*, and *cut* was set to *True* when the class was) – initialised, the sweep width of the cut, filtered signal is returned. Otherwise, the sweep width of the uncut signal is returned.

nmrespy.freqfilter.**super_gaussian**(*region*, *shape*, *p=40.0*)

Generates a super-Gaussian for filtration of frequency-domian data.

$$g\left[n_1, \cdots, n_D\right] = \exp\left[\sum_{d=1}^{D} -2^{p+1}\left(\frac{n_d - c_d}{b_d}\right)^p\right]$$

Parameters

- **region** (*[[int, int]] or [[int, int], [int, int]]*) – The region for the filter to span. For each dimension, a list of 2 entries should exist, with the first element specifying the low boundary of the region, and the second element specifying the high boundary of the region (in array indices). Note that for a given dimension d,
- **shape** (*[int] or [int, int]*) – The number of elements along each axis.
- **p** (*float, default: 40.0*) – Power of the super-Gaussian. The greater the value, the more box-like the filter.

Returns

- **sg** (*numpy.ndarray*) – Super-Gaussian filter.
- **center** (*[int] or [int, int]*) – Index of the center of the filter in each dimension.
- **bw** (*[int] or [int, int]*) – Bandwidth of the filter in each dimension, in terms of the number of points spanned.

6.4 nmrespy.mpm

Computation of signal estimates using the Matrix Pencil Method.

class nmrespy.mpm.**MatrixPencil**(*data*, *sw*, *offset=None*, *sfo=None*, *M=0*, *fprint=True*)

Class for performing the Matrix Pencil Method with the option of model order selection using the Minimum Description Length (MDL)[1]. Supports analysis of one-dimensional[2][3] or two-dimensional data[4][5]

Parameters

- **data** (*numpy.ndarray*) – Signal to be considered (unnormalised).
- **sw** (*[float] or [float, float]*) – The experiment sweep width in each dimension in Hz.
- **offset** (*[float], [float, float] or None, default: None*) – The experiment transmitter offset frequency in Hz.
- **sfo** (*[float], [float, float] or None, default: None*) – The experiment transmitter frequency in each dimension in MHz. This is not necessary, however if it set it *None*, no conversion from Hz to ppm will be possible!
- **M** (*int, default: 0*) – The number of oscillators. If *0*, the number of oscilators will be estimated using the MDL.
- **fprint** (*bool, default: True*) – Flag specifiying whether to print infomation to the terminal as the method runs.

References

__init__(*data*, *sw*, *offset=None*, *sfo=None*, *M=0*, *fprint=True*)

Checks validity of inputs, and if valid, calls `_mpm()`

_generate_params(*alpha*, *poles*)

Converts complex amplitude and signal pole arrays into a parameter array.

Parameters **alpha** (*numpy.ndarray*) – Complex amplitude array, of shape"(self.M,)"

poles: numpy.ndarray Signal pole array, of shape `(self.dim, self.M)`

Returns **result** – Parameter array, of shape `(self.M, 2 * self.dim + 2)`

Return type numpy.ndarray

_mpm_1d()

Performs 1-dimensional Matrix Pencil Method

_mpm_2d()

Performs 2-dimensional Modified Matrix Enhanced Pencil Method.

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

[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.

_remove_negative_damping(*params*)
Determines whether any oscillators possess negative amplitudes, and remove these from the parameter array.

Parameters **params** (*numpy.ndarray*) – Parameter array, with shape `(self.M, 2 * self.dim + 2)`

Returns

- **ud_params** (*numpy.ndarray*) – Updated parameter array, with negative damping oscillators removed, with shape `(M_new, 2 * self.dim + 2)`, where `M_new <= self.M`.
- **M** (*int*) – Number of oscillators after removal of negative damping oscillators.

get_result(*freq_unit='hz'*)
Obtain the result of the MPM.

Parameters **freq_unit** (*'hz' or 'ppm', default: 'hz'*) – The unit of the oscillator frequencies (corresponding to `result[:, 2]`)

Returns **result**

Return type numpy.ndarray

6.5 nmrespy.nlp

Nonlinear programming.

6.5.1 nmrespy.nlp.nlp

Nonlinear programming for generating NMR parameter estiamtes

class nmrespy.nlp.nlp.**NonlinearProgramming**(*data*, *theta0*, *sw*, *sfo=None*, *offset=None*, *start_point=0*, *phase_variance=True*, *method='trust_region'*, *bound=False*, *max_iterations=None*, *amp_thold=None*, *freq_thold=None*, *negative_amps='remove'*, *fprint=True*, *mode='apfd'*)
Class for nonlinear programming for determination of spectral parameter estimates.

Parameters

- **data** (*numpy.ndarray*) – Signal to be considered (unnormalised).
- **theta0** (*numpy.ndarray*) – Initial parameter guess in the following form:
  - **1-dimensional data:**

```
theta0 = 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:**

```
theta0 = 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],
    ...,
```

(continues on next page)

(continued from previous page)

```
    [a_m, φ_m, f1_m, f2_m, η1_m, η2_m],
])
```

- **sw** (*[float] or [float, float]*) – The experiment sweep width in each dimension in Hz.
- **offset** (*[float] or [float, float] or None, default: None*) – The experiment transmitter offset frequency in Hz. If *None*, *offset* will be set as `data.ndim * [0.0]`.
- **sfo** (*[float], [float, float] or None, default: None*) – The experiment transmitter frequency in each dimension in MHz. This is not necessary, however if it set it to *None*, no conversion of frequencies from Hz to ppm will be possible!
- **start_point** (*int, default: 0*) – The first timepoint sampled, in units of $\Delta t = 1/f_{\text{sw}}$
- **phase_variance** (*bool, default: True*) – Specifies whether or not to include the variance of oscillator phases into the NLP routine. The fiedlity (cost function) is given by:
  - *phase_variance* set to *False*:

$$\mathcal{F}(\theta) = \|Y - X\|_2^2$$

  - *phase_variance* set to *True*:

$$\mathcal{F}(\theta) = \|Y - X\|_2^2 + \text{Var}(\phi)$$

- **method** (*'trust_region' or 'lbfgs', default: 'trust_region'*) – Optimisation algorithm to use. These utilise scipy.optimise.minimise, with the method either being trust-constr, or L-BFGS-B.
- **bound** (*bool, default: False*) – Specifies whether or not to bound the parameters during optimisation. Bounds are given by:
  - $0 \leq a_m \leq \infty$
  - $-\pi < \phi_m \leq \pi$
  - $-f_{\text{sw}}/2 + f_{\text{off}} \leq f_m \leq f_{\text{sw}}/2 + f_{\text{off}}$
  - $0 \leq \eta_m \leq \infty$

  $(\forall m \in \{1, \cdots, M\})$
- **max_iterations** (*int or None, default: None*) – A value specifiying the number of iterations the routine may run through before it is terminated. If *None*, the default number of maximum iterations is set (*100* if *method* is *'trust_region'*, and *500* if *method* is *'lbfgs'*).
- **amp_thold** (*float or None, default: None*) – 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.
- **freq_thold** (*float or None*) – If *None*, does nothing. If a float, oscillator pairs with frequencies satisfying $|f_m - f_p| < f_{\text{thold}}$ will be removed from the parameter array. A new oscillator will be included in the array, with parameters:
  - amplitude: $a = a_m + a_p$
  - phase: $\phi = \left(\phi_m + \phi_p\right)/2$

  - frequency: $f = \left(f_m + f_p\right)/2$
  - damping: $\eta = \left(\eta_m + \eta_p\right)/2$

> **Warning:** NOT IMPLEMENTED YET

- **negative_amps** (*'remove' or 'flip_phase', default: 'remove'*) – 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 to these oscillators.
- **fprint** (*bool, default: True*) – If *True*, the method provides information on progress to the terminal as it runs. If *False*, the method will run silently.

Notes

The two optimisation algorithms (specified by *method*) primarily differ in how they treat the calculation of the matrix of cost function second derivatives (called the Hessian). *'trust_region'* will calculate the Hessian explicitly at every iteration, whilst *'lbfgs'* uses an update formula based on gradient information to estimate the Hessian. The upshot of this is that the convergence rate (the number of iterations needed to reach convergence) is typically better for *'trust_region'*, though each iteration typically takes longer to generate. By default, it is advised to use *'trust_region'*, however if your guess has a large number of signals, you may find *'lbfgs'* performs more effectively.

__init__(*data*, *theta0*, *sw*, *sfo=None*, *offset=None*, *start_point=0*, *phase_variance=True*, *method='trust_region'*, *bound=False*, *max_iterations=None*, *amp_thold=None*, *freq_thold=None*, *negative_amps='remove'*, *fprint=True*, *mode='apfd'*)

Initialise the class instance. Checks that all arguments are valid

_check_negative_amps()

Determines which oscillators (if any) have negative amplitudes, and removes them, or recasts them with positive amplitude and a 180° phase shift.

Returns term – Used by **_optimise()** to decide whether to terminate or re-run the optimisation routine.

Return type bool

_get_active_passive_indices()

Determine the indices of blocks of the parameter vector that contain the active and passive parameters

_get_bounds()

Constructs a list of bounding constraints to set for each parameter. The bounds are as follows:

- amplitudes: $0 < a < \infty$
- phases: $-\pi < \phi < \pi$
- frequencies: offset - sw/2 < f < offset + sw/2
- damping: $0 < \eta < \infty$

_get_errors()

Determine the errors of the estimation result

_get_slice(*idx*, *osc_idx=None*)

Parameters

- **idx** (*list*) – Parameter types to be targeted. Valid ints are *0* to *3* (included) for a 1D signal, and *0* to *5* for a 2D signal
- **osc_idx** (*list or None default: None*) – Oscillators to be targeted. Can be either *None*, where all oscillators are indexed, or a list of ints, in order to select a subset of oscillators. Valid ints are *0* to *self.m - 1* (included).

Returns **slice** – Array slice.

Return type numpy.ndarray

_merge_active_passive()
Given the active and passive parameters in vector form, merge to form the complete parameter vector

Parameters

- **active_vec** (*numpy.ndarray*) – Active vector.
- **passive_vec** (*numpy,ndarray*) – Passive vector.

Returns **merged_vec** – Merged (complete) vector.

Return type numpy.ndarray

static _pi_flip(*arr*)
flip array of phases by π raidnas, ensuring the phases remain in the range ($-\pi$, π]

_run_nlp()
Runs nonlinear programming

_shift_offset(*params*, *direction*)
Shifts frequencies to center to or displace from 0

Parameters

- **params** (*numpy.ndarray*) – Full parameter array
- **direction** (*'center' or 'displace'*) – *'center'* shifts frerquencies such that the central frequency is set to zero. *'displace'* moves frequencies away from zero, to be reflected by offset.

_split_active_passive(*merged_vec*)
Given a full vector of parameters, split to form vectors of active and passive parameters.

Parameters **merged_vec** (*numpy.ndarray*) – Full parameter vector

Returns

- **active_vec** (*numpy.ndarray*) – Active vector.
- **passive_vec** (*numpy,ndarray*) – Passive vector.

get_errors(*freq_unit='hz'*)
Obtain errors of parameters estimates.

Parameters **freq_unit** (*'hz' or 'ppm', default: 'hz'*) – The unit of the oscillator frequencies (corresponding to `result[:, 2]`)

Returns **result**

Return type numpy.ndarray

get_result(*freq_unit='hz'*)
Obtain the result of nonlinear programming.

Parameters **freq_unit** (*'hz' or 'ppm', default: 'hz'*) – The unit of the oscillator frequencies (corresponding to `result[:, 2]`)

Returns **result**

Return type numpy.ndarray

6.5.2 nmrespy.nlp._funcs

Definitions of fidelities, gradients, and Hessians.

nmrespy.nlp._funcs.**_construct_parameters**(*active*, *passive*, *m*, *idx*)
Constructs the full parameter vector from active and passive sub-vectors.

Parameters

- **active** (*numpy.ndarray*) – Active parameter vector.
- **passive** (*numpy.ndarray*) – Passive parameter vector.
- **m** (*int*) – Number of oscillators.
- **idx** (*list*) – Indicates the columns (axis 1) for active parameters.

Returns Full parameter vector with correct ordering

Return type parameters - numpy.ndarray

nmrespy.nlp._funcs.**_diagonal_indices**(*arr*, *k=0*)
Returns the indices of an array's kth diagonal. A generalisation of numpy.diag_indices_from(), which can only be used to obtain the indices along the main diagonal of an array.

Parameters

- **arr** (*numpy.ndarray*) – Square array (Hessian matrix)
- **k** (*int*) – Displacement from the main diagonal

Returns

- **rows** (*numpy.ndarray*) – 0-axis coordinates of indices.
- **cols** (*numpy.ndarray*) – 1-axis coordinates of indices.

nmrespy.nlp._funcs.**_generate_diagonal_indices**(*p*, *m*)
Determines all array indicies that correspond to positions in which non-zero second derivatives reside, in the top right half of the Hessian.

Parameters

- **p** (*int*) – The number of parameter 'groups'. For an N-dimensional signal this will be 2N + 2
- **m** (*int*) – Number of oscillators in parameter estimate

Returns

- **idx_0** (*numpy.ndarray*) – 0-axis coordinates of indices.
- **idx_1** (*numpy.ndarray*) – 1-axis coordinates of indices.

nmrespy.nlp._funcs.**f_1d**(*active*, **args*)
Cost function for 1D data

Parameters

- **active** (*numpy.ndarray*) – Array of active parameters (parameters to be optimised).
- **args** (*list_iterator*) – Contains elements in the following order:
  - **data:** *numpy.ndarray.* Array of the original FID data.
  - **tp:** *numpy.ndarray.* The time-points the signal was sampled at
  - **m:** *int.* Number of oscillators
  - **passive:** *numpy.ndarray.* Passive parameters (not to be optimised).
  - **idx:** *list.* Indicates the types of parameters that are active: `0` - amplitudes, `1` - phases, `2` - frequencies, `3` - damping factors.
  - **phase_variance:** *bool.* If *True*, include the oscillator phase variance to the cost function.

Returns **func** – Value of the cost function

Return type float

nmrespy.nlp._funcs.**f_2d**(*active*, **args*)
Cost function for 2D data.

Parameters

- **para_act** (*numpy.ndarray*) – Array of active parameters (parameters to be optimised).
- **args** (*list_iterator*) – Contains elements in the following order:
  - **data:** *numpy.ndarray.* Array of the original FID data.
  - **tp:** *(numpy.ndarray, numpy.ndarray).* The time-points the signal was sampled, in both dimensions.
  - **M:** *int.* Number of oscillators.
  - **passive:** *numpy.ndarray.* Passive parameters (not to be optimised).
  - **idx:** *list.* Indicates the types of parameters that are active: `0` - amplitudes, `1` - phases, `2` & `3` - frequencies, `4` & `5` - damping factors.
  - **phasevar:** *bool.* If *True*, include the oscillator phase variance to the cost function.

Returns **func** – Value of the cost function.

Return type float

nmrespy.nlp._funcs.**g_1d**(*active*, **args*)
Gradient of cost function for 1D data.

Parameters

- **active** (*numpy.ndarray*) – Array of active parameters (parameters to be optimised).
- **args** (*list_iterator*) – Contains elements in the following order:
  - **data:** *numpy.ndarray.* Array of the original FID data.
  - **tp:** *numpy.ndarray.* The time-points the signal was sampled at
  - **m:** *int.* Number of oscillators
  - **passive:** *numpy.ndarray.* Passive parameters (not to be optimised).
  - **idx:** *list.* Indicates the types of parameters are present in the active oscillators (para_act): `0` - amplitudes, `1` - phases, `2` - frequencies, `3` - damping factors.

– **phase_variance:** *Bool.* If True, include the oscillator phase variance to the cost function.

Returns **grad** – Gradient of cost function, with `grad.shape = (4 * m,)`.

Return type numpy.ndarray

`nmrespy.nlp._funcs.g_2d(active, *args)`
Gradient of cost function for 2D data.

Parameters

- **active** (*numpy.ndarray*) – Array of active parameters (parameters to be optimised).
- **args** (*list_iterator*) – Contains elements in the following order:
  - **data:** *numpy.ndarray.* Array of the original FID data.
  - **tp:** *(numpy.ndarray, numpy.ndarray).* The time-points the signal was sampled, in both dimensions.
  - **m:** *int.* Number of oscillators.
  - **passive:** *numpy.ndarray.* Passive parameters (not to be optimised).
  - **idx:** *list.* Indicates the types of parameters that are active: `0` - amplitudes, `1` - phases, `2` & `3` - frequencies, `4` & `5` - damping factors.
  - **phasevar:** *bool.* If *True*, include the oscillator phase variance to the cost function.

Returns **grad** – Gradient of cost function, with `grad.shape = (6*M,)`.

Return type numpy.ndarray

`nmrespy.nlp._funcs.h_1d(active, *args)`
Hessian of cost function for 1D data

Parameters

- **active** (*numpy.ndarray*) – Array of active parameters (parameters to be optimised).
- **args** (*list_iterator*) – Contains elements in the following order:
  - **data:** *numpy.ndarray.* Array of the original FID data.
  - **tp:** *numpy.ndarray.* The time-points the signal was sampled at
  - **m:** *int.* Number of oscillators
  - **passive:** *numpy.ndarray.* Passive parameters (not to be optimised).
  - **idx:** *list.* Indicates the types of parameters are present in the active oscillators (para_act): `0` - amplitudes, `1` - phases, `2` - frequencies, `3` - damping factors.
  - **phase_variance:** *Bool.* If True, include the oscillator phase variance to the cost function.

Returns **hess** – Hessian of cost function, with `hess.shape = (4 * m,4 * m)`.

Return type numpy.ndarray

`nmrespy.nlp._funcs.h_2d(active, *args)`

Hessian of cost function for 2D data

active [numpy.ndarray] Array of active parameters (parameters to be optimised).

args [list_iterator] Contains elements in the following order:

- **data:** *numpy.ndarray.* Array of the original FID data.
- **tp:** *(numpy.ndarray, numpy.ndarray).* The time-points the signal was sampled at
- **m:** *int.* Number of oscillators
- **passive:** *numpy.ndarray.* Passive parameters (not to be optimised).
- **idx:** *list.* Indicates the types of parameters that are active: `0` - amplitudes, `1` - phases, `2` & `3` - frequencies, `4` & `5` - damping factors.
- **phasevar:** *bool.* If *True*, include the oscillator phase variance to the cost function.

hess [numpy.ndarray] Hessian of cost function, with `hess.shape = (6*M,6*M)`.

6.6 nmrespy.plot

Support for plotting estimation results

class nmrespy.plot.**NmrespyPlot**(*fig*, *ax*, *lines*, *labels*)
Plot result class

Note: The class is very minimal at the moment. I plan to expand its functionality in later versions.

Parameters

- **fig** (matplotlib.figure.Figure) – Figure.
- **ax** (matplotlib.axes._subplots.AxesSubplot) – Axes.
- **lines** (*dict*) – Lines dictionary.
- **labels** (*dict*) – Labels dictionary.

Notes

To save the figure, simply access the *fig* attribute and use the savefig method:

```
>>> plot.fig.savefig(f'example.{ext}', ...)
```

displace_labels(*values*, *displacement*)
Displace labels with values given by *values*

Parameters

- **values** (*list*) – The value(s) of the label(s) to displace. All elements should be ints.
- **displacement** (*(float, float)*) – The amount to displace the labels relative to their current positions. The displacement uses the axes co-ordinate system. Both values provided should be less than *1.0*.

hide_labels()
Make the oscillator labels visible

show_labels()
Make the oscillator labels visible

transfer_to_axes(*ax*)

Reproduces the plot in *self.ax* in another axes object.

Parameters **ax** (matplotlib.axes.Axes) – The axes object to construct the result plot onto.

> **Warning:** Everything present in *ax* before calling the method will be removed. If you want to add further things to *ax*, do it after calling this method.

nmrespy.plot.**plot_result**(*data*, *result*, *sw*, *offset*, *plot_residual=True*, *plot_model=False*, *residual_shift=None*, *model_shift=None*, *sfo=None*, *shifts_unit='ppm'*, *nucleus=None*, *region=None*, *data_color='#000000'*, *residual_color='#808080'*, *model_color='#808080'*, *oscillator_colors=None*, *labels=True*, *stylesheet=None*)

Produces a figure of an estimation result.

The figure consists of the original data, in the Fourier domain, along with each oscillator.

Parameters

- **data** (*numpy.ndarray*) – Data of interest (in the time-domain).
- **result** (*numpy.ndarray*) – Parameter estimate, of form:
  - **1-dimensional data:**

```
parameters = 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:**

```
parameters = 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],
])
```

- **sw** (*[float] or [float, float]*) – Sweep width in each dimension (Hz).
- **offset** (*[float] or [float, float]*) – Transmitter offset in each dimension (Hz).
- **sfo** (*[float, [float, float] or None, default: None*) – Transmitter frequency in each dimnesion (MHz). Needed to plot the chemical shift axis in ppm. If *None*, chemical shifts will be plotted in Hz.
- **nucleus** (*[str], [str, str] or None, default: None*) – The nucleus in each dimension.
- **region** (*[[int, int]], [[float, float]], [[int, int], [int, int]] or [[float, float], [float, float]]*) – Boundaries specifying the region to show. See also *nmrespy.freqfilter.FrequencyFilter*.
- **plot_residual** (*bool, default: True*) – If *True*, plot a difference between the FT of *data* and the FT of the model generated using *result*. NB the residual is plotted regardless of *plot_residual*. *plot_residual* specifies the alpha transparency of the plot line (1 for *True*, 0 for *False*)
- **residual_shift** (*float or None, default: None*) – Specifies a translation of the residual plot along the y-axis. If *None*, the default shift will be applied.

- **plot_model** (*bool, default: False*) – If *True*, plot the FT of the model generated using *result*. NB the residual is plotted regardless of *plot_model*. *plot_model* specifies the alpha transparency of the plot line (1 for *True*, 0 for *False*)
- **model_shift** (*float or None, default: None*) – Specifies a translation of the residual plot along the y-axis. If *None*, the default shift will be applied.
- **data_color** (*matplotlib color, default: '#000000'*) – The colour used to plot the original data. Any value that is recognised by matplotlib as a color is permitted. See herehttps://matplotlib.org/3.1.0/tutorials/colors/colors.html for a full description of valid values.
- **residual_color** (*matplotlib color, default: '#808080'*) – The colour used to plot the residual.
- **model_color** (*matplotlib color, default: '#808080'*) – The colour used to plot the model.
- **oscillator_colors** (*{matplotlib color, matplotlib colormap name, list, numpy.ndarray, None}, default: None*) – Describes how to color individual oscillators. The following is a complete list of options:
  - If a valid matplotlib color is given, all oscillators will be given this color.
  - If a string corresponding to a matplotlib colormap is given, the oscillators will be consecutively shaded by linear increments of this colormap. For all valid colormaps, see herehttps://matplotlib.org/stable/tutorials/colors/colormaps.html
  - If a list or NumPy array containing valid matplotlib colors is given, these colors will be cycled. For example, if `oscillator_colors = ['r', 'g', 'b']`:
    * Oscillators 1, 4, 7, ... would be red (#FF0000)
    * Oscillators 2, 5, 8, ... would be green (#008000)
    * Oscillators 3, 6, 9, ... would be blue (#0000FF)
  - If *None*:
    * If a stylesheet is specified, with the attribute `axes.prop_cycle` provided, this colour cycle will be used.
    * Otherwise, the default colouring method will be applied, which involves cycling through the following colors:
      · #1063E0
      · #EB9310
      · #2BB539
      · #D4200C
- **labels** (*Bool, default: True*) – If *True*, each oscillator will be given a numerical label in the plot, if *False*, no labels will be produced.
- **stylesheet** (*str or None, default: None*) – The name of/path to a matplotlib stylesheet for further customaisation of the plot. See herehttps://matplotlib.org/stable/tutorials/introductory/customizing.html for more information on stylesheets.

Returns

plot – A class instance with the following attributes:

- **fig** [matplotlib.figure.Figure] The resulting figure.
- **ax** [matplotlib.axes.Axes] The resulting set of axes.

- **lines** [dict] A dictionary containing a series of matplotlib.lines.Line2D instances. The data plot is given the key *0*, and the individual oscillator plots are given the keys *1*, *2*, *3*, ..., <*M*> where <*M*> is the number of oscillators in the parameter estimate.
- **labels** [dict] A dictionary containing a series of of matplotlib.text.Text instances, with the keys *1*, *2*, etc. The Boolean argument *labels* affects the alpha transparency of the labels:
  - *True* sets alpha to 1 (making the labels visible)
  - *False* sets alpha to 0 (making the labels invisible)

Return type *NmrespyPlot*

6.7 nmrespy.sig

Constructing and processing NMR signals

class nmrespy.sig.**PhaseApp**(*spectrum*, *max_p1*)
Tkinter application for manual phase correction.

See also:

manual_phase_spectrum()

nmrespy.sig.**ft**(*fid*, *flip=True*)
Performs Fourier transformation and (optionally) flips the resulting spectrum to satisfy NMR convention.

Parameters

- **fid** (*numpy.ndarray*) – Time-domain data.
- **flip** (*bool, default: True*) – Whether or not to flip the Fourier Trnasform of *fid* in each dimension.

Returns **spectrum** – Fourier transform of the data, flipped in each dimension.

Return type numpy.ndarray

nmrespy.sig.**generate_random_signal**(*m*, *n*, *sw*, *offset=None*, *snr=None*)
A convienince function to generate a synthetic FID with random parameters for testing purposes.

Parameters

- **m** (*int*) – Number of oscillators
- **n** (*[int] or [int, int]*) – Number of points in each dimension
- **sw** (*[float] or [float, float]*) – Sweep width in each dimension
- **offset** (*[float], [float, float] or None, deafult: None*) – Transmitter offset in each dimension
- **snr** (*float or None, default: None*) – Signal-to-noise ratio (dB)
- **fid** (*numpy.ndarray*) – The synthetic FID.
- **tp** (*[numpy.ndarray], [numpy.ndarray, numpy.ndarray]*) – The time points the FID is sampled at in each dimension.
- **parameters** (*numpy.ndarray*) – Parameters used to construct the signal

nmrespy.sig.**get_shifts**(*n*, *sw*, *offset=None*, *flip=True*)
Generates the frequencies that the FT of the FID is sampled at, given its sweep-width, the transmitter offset, and the number of points.

Parameters

- **n** (*[int] or [int, int]*) – The number of points in each dimension.
- **sw** (*[float] or [float, float]*) – The sweep width in each dimension.
- **offset** (*[float], [float, float], or None, default: None*) – The transmitter offset in each dimension. If *None*, the offset will be set to zero in each dimension.
- **flip** (*bool, default: True*) – If *True*, the shifts will be returned in descending order, as is conventional in NMR. If *False*, the shifts will be in ascending order.

Returns **shifts** – The chemical shift values sampled in each dimension.

Return type [numpy.ndarray] or [numpy.ndarray, numpy.ndarray]

nmrespy.sig.**get_timepoints**(*n*, *sw*)

Generates the timepoints at which an FID is sampled at, given its sweep-width, and the number of points.

Parameters

- **n** (*[int] or [int, int]*) – The number of points in each dimension.
- **sw** (*[float] or [float, float]*) – THe sweep width in each dimension (Hz).

Returns **tp** – The time points sampled in each dimension

Return type [numpy.ndarray] or [numpy.ndarray, numpy.ndarray]

nmrespy.sig.**ift**(*spectrum*, *flip=True*)

Flips spectral data in each dimension, and then inverse Fourier transforms.

Parameters

- **spectrum** (*numpy.ndarray*) – Spectrum
- **flip** (*bool, default: True*) – Whether or not to flip *spectrum* in each dimension prior to Inverse Fourier Transform.

Returns **fid** – Inverse Fourier transform of the spectrum.

Return type numpy.ndarray

nmrespy.sig.**make_fid**(*parameters*, *n*, *sw*, *offset=None*, *snr=None*, *decibels=True*, *modulation='none'*)

Constructs a discrete time-domain signal (FID), as a summation of exponentially damped complex sinusoids.

Parameters

- **parameters** (*numpy.ndarray*) – Parameter array with the following structure:
  - **1-dimensional data:**

```
parameters = 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:**

```
parameters = 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],
   ...,
```

(continues on next page)

(continued from previous page)

```
    [a_m, φ_m, f1_m, f2_m, η1_m, η2_m],
])
```

- **n** (*[int], [int, int]*) – Number of points to construct signal from in each dimension.
- **sw** (*[float], [float, float]*) – Sweep width in each dimension, in Hz.
- **offset** (*[float], [float, float], or None, default: None*) – Transmitter offset frequency in each dimension, in Hz. If set to *None*, the offset frequency will be set to 0Hz in each dimension.
- **snr** (*float or None, default: None*) – The signal-to-noise ratio. If *None* then no noise will be added to the FID.
- **decibels** (*bool, default: True*) – 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.
- **modulation** (*{'none', 'amp', 'phase'}, default: 'none'*) – The type of modulation present in the indirect dimension, if the data is 2D. *In the expressions below, a it is assumed a single oscillator has been provided for simplicity*.
  - *'none'*: Returns a single signal of the form:

$$y(t_1, t_2) = a \exp(\mathrm{i}\phi) \exp\left[\left(2\pi \mathrm{i} f_1 - \eta_1\right) t_1\right] \exp\left[\left(2\pi \mathrm{i} f_2 - \eta_2\right) t_2\right]$$

  - *'amp'*: Returns an amplitude-modulated pair of signals of the form:

$$y_{\cos}(t_1, t_2) = a \exp(\mathrm{i}\phi) \cos\left(2\pi f_1 t_1\right) \exp\left(-\eta_1 t_1\right) \exp\left[\left(2\pi \mathrm{i} f_2 - \eta_2\right) t_2\right]$$
$$y_{\sin}(t_1, t_2) = a \exp(\mathrm{i}\phi) \sin\left(2\pi f_1 t_1\right) \exp\left(-\eta_1 t_1\right) \exp\left[\left(2\pi \mathrm{i} f_2 - \eta_2\right) t_2\right]$$

  - *'phase'*: Returns an phase-modulated pair of signals of the form:

$$y_{\mathrm{P}}(t_1, t_2) = a \exp(\mathrm{i}\phi) \exp\left[\left(2\pi \mathrm{i} f_1 - \eta_1\right) t_1\right] \exp\left[\left(2\pi \mathrm{i} f_2 - \eta_2\right) t_2\right]$$
$$y_{\mathrm{N}}(t_1, t_2) = a \exp(\mathrm{i}\phi) \exp\left[\left(-2\pi \mathrm{i} f_1 - \eta_1\right) t_1\right] \exp\left[\left(2\pi \mathrm{i} f_2 - \eta_2\right) t_2\right]$$

Returns

- **fid** (*numpy.ndarray, [numpy.ndarray, numpy.ndarray]*) – The synthetic signal generated.
  - If the data to be constructed is 1D or 2D with *modulation* set to *'none'*, the result will be a NumPy array.
  - If the data is 2D with *modulation* set to *'amp'*, or *'phase'* the result will be a length-2 list with signals of the forms indicated above (See *modulation*).
- **tp** (*[numpy.ndarray], [numpy.ndarray, numpy.ndarray]*) – The time points the FID is sampled at in each dimension.

Notes

The resulting *fid* is given by

$$y[n_1, \cdots, n_D] = \sum_{m=1}^{M} a_m \exp\left(\mathrm{i}\phi_m\right) \prod_{d=1}^{D} \exp\left[\left(2\pi \mathrm{i} f_m - \eta_m\right) n_d \Delta t_d\right]$$

where d is either 1 or 2, M is the number of oscillators, and $\Delta t_d = 1/f_{\mathrm{sw},d}$.

nmrespy.sig.**make_noise**(*fid*, *snr*, *decibels=True*)

Given a synthetic FID, generate an array of normally distributed complex noise with zero mean and a variance that abides by the desired SNR.

Parameters

- **fid** (*numpy.ndarray*) – Noiseless FID.
- **snr** (*float*) – The signal-to-noise ratio.
- **decibels** (*bool, default: True*) – 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.

Returns **noise**

Return type numpy.ndarray

nmrespy.sig.**make_virtual_echo**(*data*, *modulation='amp'*)

Given the time-domain signal *data*, generates the corresponding virtual echo[1], a signal with a purely real Fourier-Tranform and absorption mode line shape if the data is phased.

Parameters

- **data** (*[numpy.ndarray] or [numpy.ndarray, numpy.ndarray]*) – The data to construct the virtual echo from. This should be a list of NumPy arrays, with `len(data) == d` where `d` is the dimension of the signal.
- **modulation** (*{'amp' or 'phase'}, default: 'amp'*) – If the data is 2D, this parameter specifies the type of modulation present in the indirect dimension of the dataset.
  - If set to *'amp'*, the two signals in the *data* should be an amplitude modulated pair.
  - If set to *'phase'*, the two signals in the *data* should be a phase modulated pair.

  See the docs for *make_fid()* for more info on *modulation*.

Returns **virtual_echo** – The virtual echo signal assocaited with *data*.

Return type numpy.ndarray

References

nmrespy.sig.**manual_phase_spectrum**(*spectrum*, *max_p1=None*)

Generates a GUI, enabling manual phase correction, with the zero- and first-order phases returned.

> **Warning:** Only 1D spectral data is currently supported.

Parameters

- **spectrum** (*numpy.ndarray*) – Spectral data of interest.
- **max_p1** (*float or None, default: None*) – Specifies the range of first-order phases permitted. For each dimension, the user will be allowed to choose a value of *p1* within [*-max_p1*, *max_p1*]. By default, *max_p1* will be `10 * numpy.pi`.

Returns

- **p0** (*[float] or None*) – Zero-order phase correction in each dimension, in radians. If the user chooses to cancel rather than save, this is set to *None*.

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

- **p1** (*[float] or None*) – 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.**oscillator_integral**(*parameters*, *n*, *sw*, *offset=None*)

Determines the (absolute) integral of the Fourier transform of an oscillator.

Parameters

- **parameters** (*numpy.ndarray*) – Oscillator parameters of the following form:
  - **1-dimensional data:**

```
parameters = numpy.array([a, φ, f, η])
```

  - **2-dimensional data:**

```
parameters = numpy.array([a, φ, f1, f2, η1, η2])
```

- **n** (*[int], [int, int]*) – Number of points to construct signal from in each dimension.
- **sw** (*[float], [float, float]*) – Sweep width in each dimension, in Hz.
- **offset** (*[float], [float, float], or None, default: None*) – Transmitter offset frequency in each dimension, in Hz. If set to *None*, the offset frequency will be set to 0Hz in each dimension.

Returns

Return type integral

Notes

The integration is performed using the composite Simpsons rule, provided by scipy.integrate.simpson

Spacing of points along the frequency axes is set a *1* (i.e. $dx = 1$).

nmrespy.sig.**phase**(*data*, *p0*, *p1*, *pivot=None*)

Applies a linear phase correction to *data*.

Parameters

- **data** (*numpy.ndarray*) – Data to be phased.
- **p0** (*[float] or [float, float]*) – Zero-order phase correction in each dimension, in radians.
- **p1** (*[float] or [float, float]*) – First-order phase correction in each dimension, in radians.
- **pivot** (*[int], [int, int] or None*) – Index of the pivot in each dimension. If None, the pivot will be *0* in each dimension.

Returns **phased_data**

Return type numpy.ndarray

nmrespy.sig.**proc_amp_modulated**(*data*)

Takes a pair of 2D amplitude-modulated signals, and generates the frequency-discriminated spectrum.

Parameters **data** (*[numpy.ndarray, numpy.ndarray]*) – cos-modulated signal and sin-modulated signal

Returns **spectrum** – Dictionary of four elements: `rr`, `ri`, `ir`, and `ii`.

Return type dict

nmrespy.sig.**proc_phase_modulated**(*data*)

Takes a pair of 2D phase-modulated signals, and generates the set of spectra corresponding to the processing protocol outlined in[2].

Parameters **data** (*[numpy.ndarray, numpy.ndarray]*) – P-type signal and N-type signal

Returns **spectra** – Dictionary of four elements: `rr`, `ri`, `ir`, and `ii`.

Return type dict

References

6.8 nmrespy.write

Writing estimation results to .txt, .pdf and .csv files

nmrespy.write.**_construct_paramtable**(*parameters*, *errors*, *integrals*, *sfo*, *sig_figs*, *sci_lims*, *fmt*)

Creates a nested list of values to input to parameter table, with desired formatting.

Parameters

- **parameters** (*numpy.ndarray*) – Parameter array.
- **errors** (*numpy.ndarray or None*) – Parameter errors.
- **integrals** (*list*) – Oscillator peak integrals.
- **sfo** (*[float], or [float, float] or None*) – Transmitter offset frequency (MHz) in each dimension.
- **sig_figs** (*int or None*) – Desired nuber of significant figures.
- **sci_lims** (*(int, int) or None*) – Bounds defining thresholds for using scientific notation.

Returns

- **titles** (*list*) – Titles of parameter table
- **table** (*list*) – Values of parameter table.

nmrespy.write.**_latex_tabular**(*rows*)

Creates a string of text that denotes a tabular entity in LaTeX

Parameters **rows** (*list*) – Nested list, with each sublist containing elements of a single row of the table.

Returns **table** – LaTeX -formated table

Return type str

[2] 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.

Example

```
>>> from nmrespy.write import _latex_tabular
>>> rows = [['A1', 'A2', 'A3'], ['B1', 'B2', 'B3']]
>>> print(_latex_tabular(rows))
A1 & A2 & A3 \\
B1 & B2 & B3 \\
```

nmrespy.write.**_map_to_latex_titles**(*titles*)

Given a list of titles produced by *_construct_paramtable()*, Generate equivalent titles for LaTeX.

Uses a simple literal matching approach.

nmrespy.write.**_scientific_notation**(*value*, *sci_lims*, *fmt*)

Converts value to scientific notation

Parameters

- **value** (*float*) – Value to process
- **sci_lims** (*(int, int)*) – See description in *write_result()*
- **fmt** (*'txt', 'pdf', or 'csv'*) – File format.

nmrespy.write.**_strval**(*value*, *sig_figs*, *sci_lims*, *fmt*)

Convert float to formatted string.

Parameters

- **- float** (*value*) – Value to convert.
- **- int or None** (*sig_figs*) – Number of significant figures.
- **- (int** (*sci_lims*) – Bounds defining thresholds for using scientific notation.
- **or None** (*int)*) – Bounds defining thresholds for using scientific notation.

Returns Formatted value.

Return type strval - str

nmrespy.write.**_timestamp**()

Constructs a string with time/date information.

Returns

timestamp – Of the form:

```
hh:mm:ss
dd-mm-yy
```

Return type str

nmrespy.write.**_txt_tabular**(*columns*, *titles=None*, *separator=''*)

Tabularises a list of lists, with the option of including titles. Used in textfile outputs.

Parameters

- **columns** (*list*) – A list of lists, with each sublist representing the columns of the table. Each list must be of the same length.
- **titles** (*None or list, default: None*) – Titles for the table. If desired, *titles* should be of the same length as all of the sublists in *columns*.

- **separator** (*str, default: ''*) – Column separator. By default, an empty string is used. (See first example above).

Returns **table** – A string with the contents of *titles* (opt.) and *columns* tabularised.

Return type str

Examples

A simple example with *titles* specified and the default *separator*:

```
>>> from nmrespy.write import _txt_tabular
>>> columns = [['A1', 'B1'], ['A2', 'B2'], ['A3', 'B3']]
>>> titles = ['title 1', 'title 2', 'title 3']
>>> print(_txt_tabular(columns, titles=titles))
title 1  title 2  title 3
─────────┼────────┼────────
A1       A2       A3
B1       B2       B3
```

You may want to set *separator* to something like `'|'` in order for a nicer layout when you have titles:

```
>>> # Same as before...
>>> print(_txt_tabular(columns, titles=titles, separator='|'))
title 1 |title 2 |title 3
────────┼────────┼────────
A1      |A2      |A3
B1      |B2      |B3
```

`nmrespy.write._write_csv`(*path*, *description*, *info_headings*, *info*, *param_titles*, *param_table*, *fprint*)

Writes parameter estimate to a CSV.

Parameters

- **path** (*pathlib.Path*) – File path
- **description** (*str or None, default: None*) – A descriptive statement.
- **info_headings** (*list or None, default: None*) – Headings for experiment information.
- **info** (*list or None, default: None*) – Information that corresponds to each heading in *info_headings*.
- **param_titles** (*list*) – Titles for parameter array table.
- **param_table** (*list*) – Array of contents to append to the result table.
- **fprint** (*bool*) – Specifies whether or not to print output to terminal.

`nmrespy.write._write_pdf`(*path*, *description*, *info_headings*, *info*, *param_titles*, *param_table*, *pdflatex_exe*, *fprint*)

Writes parameter estimate to a PDF using `pdflatex`.

Parameters

- **path** (*pathlib.Path*) – File path
- **description** (*str or None, default: None*) – A descriptive statement.
- **info_headings** (*list or None, default: None*) – Headings for experiment information.
- **info** (*list or None, default: None*) – Information that corresponds to each heading in *info_headings*.

- **param_titles** (*list*) – Titles for parameter array table.
- **param_table** (*list*) – Array of contents to append to the result table.
- **fprint** (*bool*) – Specifies whether or not to print output to terminal.

nmrespy.write.**_write_txt**(*path*, *description*, *info_headings*, *info*, *param_titles*, *param_table*, *fprint*)

Writes parameter estimate to a textfile.

Parameters

- **path** (*pathlib.Path*) – File path
- **description** (*str or None, default: None*) – A descriptive statement.
- **info_headings** (*list or None, default: None*) – Headings for experiment information.
- **info** (*list or None, default: None*) – Information that corresponds to each heading in *info_headings*.
- **param_titles** (*list*) – Titles for parameter array table.
- **param_table** (*list*) – Array of contents to append to the result table.
- **fprint** (*bool*) – Specifies whether or not to print output to terminal.

nmrespy.write.**write_result**(*parameters*, *errors=None*, *path='./nmrespy_result'*, *sfo=None*, *integrals=None*, *description=None*, *info_headings=None*, *info=None*, *sig_figs=5*, *sci_lims=(- 2, 3)*, *fmt='txt'*, *force_overwrite=False*, *pdflatex_exe=None*, *fprint=True*)

Writes an estimation result to a .txt, .pdf or .csv file.

Parameters

- **parameters** (*numpy.ndarray*) – The estimated parameter array.
- **errors** (*numpy.ndarray or None, default: None*) – The errors associated with the parameters. If not *None*, the shape of *errors* should match that of *parameters*.
- **path** (*str, default: './nmrespy_result'*) – The path to save the file to. DO NOT INCLUDE A FILE EXTENSION. This will be added based on the value of *fmt*. For example, if you set *path* to *.../result.txt* and *fmt* is *'txt'*, the resulting file will be *.../result.txt.txt*
- **sfo**(*[float], [float, float] or None:*) – The transmitter offset in each dimension. This is required to express frequencies in ppm as well as Hz. If *None*, frequencies will only be expressed in Hz.
- **integrals** (*list or None*) – Peak integrals of each oscillator. Note that `parameters.shape[0] == len(integrals)` should be satisfied.
- **fmt** (*'txt', 'pdf' or 'csv', default: 'txt'*) – File format. See notes for details on system requirements for PDF generation.
- **force_overwrite** (*bool. default: False*) – Defines behaviour if `f'{path}.{fmt}'` already exists:
  - If *force_overwrite* is set to *False*, the user will be prompted if they are happy overwriting the current file.
  - If *force_overwrite* is set to *True*, the current file will be overwritten without prompt.
- **description** (*str or None, default: None*) – A descriptive statement.
- **info_headings** (*list or None, default: None*) – Headings for experiment information. Could include items like *'Sweep width (Hz)'*, *'Transmitter offset (Hz)'*, etc. N.B. All the elements in *info_headings* should be strings!

- **info** (*list or None, default: None*) – Information that corresponds to each heading in *info_headings*. N.B. All the elements in *info* should be strings!
- **sig_figs** (*int or None default: 5*) – The number of significant figures to give to parameter values. If *None*, the full value will be used.
- **sci_lims** (*(int, int) or None, default: (-2, 3)*) – Given a value `(-x, y)`, for positive *x* and *y*, any parameter *p* value which satisfies `p < 10 ** -x` or `p >= 10 ** y` will be expressed in scientific notation, rather than explicit notation. If *None*, all values will be expressed explicitely.
- **pdflatex_exe** (*str or None, default: None*) – 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.

- **fprint** (*bool, default: True*) – Specifies whether or not to print information to the terminal.

Raises ***LaTeXFailedError*** – With *fmt* set to *'pdf'*, this will be raised if an error was encountered in trying to run `pdflatex`.

Notes

To generate PDF result files, it is necessary to have a LaTeX installation set up on your system. For a simple to set up implementation that is supported on all major operating systems, consider TexLive. To ensure that you have a functioning LaTeX installation, open a command prompt/terminal and type:

```
$ pdflatex -version
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.
There is NO warranty.  Redistribution of this software is
covered by the terms of both the pdfTeX copyright and
the Lesser GNU General Public License.
For more information about these matters, see the file
named COPYING and the pdfTeX source.
Primary author of pdfTeX: Han The Thanh (pdfTeX) et al.
Compiled with libpng 1.6.37; using libpng 1.6.37
Compiled with zlib 1.2.11; using zlib 1.2.11
Compiled with xpdf version 4.01
```

The following is a full list of packages that your LaTeX installation will need to successfully compile the generated .tex file:

- amsmath
- array
- booktabs
- cmbright
- geometry
- hyperref
- longtable

- 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.

6.9 nmrespy._cols

Coloured terminal output

6.10 nmrespy._errors

nmrespy-specific errors

exception nmrespy._errors.**AttributeIsNoneError**(*attribute*, *method*)
Raise when the user calls a *get_<attr>* method, but the attribute is None

exception nmrespy._errors.**InvalidDirectoryError**(*dir*)
Raise when the a dictionary does not have the requisite files

exception nmrespy._errors.**InvalidUnitError**(**args*)
Raise when the specified unit is invalid

exception nmrespy._errors.**LaTeXFailedError**(*texpath*)
Raise when the user calls write_result(), with format set to 'pdf', but compiling the TeX file failed when pdflatex was called.

exception nmrespy._errors.**MoreThanTwoDimError**
Raise when user tries importing data that is >2D

exception nmrespy._errors.**NoParameterEstimateError**
Raise when instance does not possess a valid parameter array

exception nmrespy._errors.**ParameterNotFoundError**(*param_name*, *path*)
Raise when a desired parameter is not present in an acqus/procs file

exception nmrespy._errors.**PhaseVarianceAmbiguityError**(*mode*)
Raise when phase_variance is True, but 'p' is not specified in mode

exception nmrespy._errors.**TwoDimUnsupportedError**
Raise when user tries running a method that doesn't support 2D data yet

6.11 nmrespy._misc

Various miscellaneous functions/classes for internal nmrespy use.

class nmrespy._misc.**ArgumentChecker**(*components*, *dim=None*, *n=None*)

Checks that user-given arguments are of an appropriate type.

Parameters

- **components** (*list of 3-tuples*) – Each tuple should contain the following elements:
  - The object to check.
  - A string to identify the object in any error messages‘
  - A string specifying whaty type the object should be. Valid options are:
    - *'ndarray'*
    - *'parameter'*
    - *'int_list'*
    - *'float_list'*
    - *'str_list'*
    - *'array_list'*
    - *'region_int'*
    - *'region_float'*
    - *'bool'*
    - *'int'*
    - *'float'*
    - *'str'*
    - *'list'*
    - *'positive_int'*
    - *'positive_int_or_zero'*
    - *'positive_float'*
    - *'optimiser_mode'*
    - *'optimiser_algorithm'*
    - *'zero_to_one'*
    - *'greater_than_one'*
    - *'negative_amplidue'*
    - *'file_fmt'*
    - *'pos_neg_tuple'*
    - *'mpl_color'*
    - *'osc_cols'*
    - *'displacement'*

- **dim** (*1, 2 or None, default: None*) – Dimension of the data. Only needs to be specified as *1* or *2* if one or more of the arguments to check have a structure that depends on the data dimension.

static check_displacement(*obj*)
Check for a valid mpl label displacement tuple.

static check_mpl_color(*obj*)
Check for a valid matplotlib color.

static check_optimiser_mode(*obj*)
Ensures that the optimisation mode is valid. This should be a string containing only the characters 'a', 'p', 'f', and 'd', without any repetition.

check_oscillator_colors(*obj*)
Check for valid oscillator colorcycle

static check_pos_neg_tuple(*obj*)
Check for object of the form `(-x, y)` where *x* and *y* are positive ints.

class nmrespy._misc.**FrequencyConverter**(*n*, *sw*, *offset*, *sfo=None*)
Handles converting objects with frequency values between units

Parameters

- **n** (*[int] or [int, int]*) – Number of points in each dimension.
- **sw** (*[float] or [float, float]*) – Experiment sweep width in each dimension (Hz)
- **offset** (*[float] or [float, float]*) – Transmitter offset in each dimension (Hz)
- **sfo** (*[float] or [float, float] or None, default: None*) – Transmitter frequency in each dimension (MHz). If set to *None*, only conversion between Hz and array indices will be possible.

_check_valid_conversion(*conversion*)
Check that conversion is a valid value

convert(*lst*, *conversion*)
Convert quantities contained within a list

Parameters

- **lst** (*list*) – A list of numerical values, with the same length as `len(self)`.
- **conversion** (*str*) – A string denoting the coversion to be applied. The form of *conversion* should be `'<from>-><to>'`, where `<from>` and `<to>` are not matching, and are each one of the following:
  - *'idx'*: array index
  - *'hz'*: Hertz
  - *'ppm'*: parts per million

Returns **converted_lst** – A list of the same dimensions as *lst*, with converted values.

Return type lst

class nmrespy._misc.**PathManager**(*fname*, *dir*)
Class for performing checks on paths.

Parameters

- **fname** (*str*) – Filename.
- **dir** (*str*) – Directory.

ask_overwrite()
Asks the user if the are happy to overwrite the file given by the path *self.path*

check_file(*force_overwrite=False*)
Performs checks on the path file dir/fname

Parameters **force_overwrite** (*bool, default: False*) – Specifies whether to ask the user if they are happy for the file *self.path* to be overwritten if it already exists in their filesystem.

Returns **return_code** – See notes for details

Return type int

Notes

This method first checks whether dir exists. If it does, it checks whether the file *dir/fname* exists. If it does, the user is asked for permission to overwrite, if *force_overwrite* is *False*. The following codes can be returned:

- `0` *dir/fname* doesn't exist/can be overwritten, and *dir* exists.
- `1` *dir/fname* already exists and the user does not give permission to overwrite.
- `2` *dir* does not exist.

nmrespy._misc.**get_yes_no**(*prompt*)
Ask user to input 'yes' or 'no' (Y/y or N/n). Repeatedly does this until a valid response is recieved

nmrespy._misc.**latex_nucleus**(*nucleus*)
Creates a isotope symbol string for processing by LaTeX.

Parameters **nucleus** (*str*) – Of the form *'<mass><sym>'*, where *'<mass>'* is the nuceleus' mass number and *'<sym>'* is its chemical symbol. I.e. for lead-207, *nucleus* would be *'207Pb'*.

Returns **latex_nucleus** – Of the form `$^{<mass>}$<sym>` i.e. given *'207Pb'*, the return value would be `$^{207}$Pb`

Return type str

Raises **ValueError** – If *nucleus* does not match the regex `^[0-9]+[a-zA-Z]+$`

nmrespy._misc.**significant_figures**(*value*, *s*)
Rounds *value* to *s* significant figures.

nmrespy._misc.**start_end_wrapper**(*start_text*, *end_text*)
Decorator which prints a message prior to and after a method.

Messages are sandwiched between double-lines.

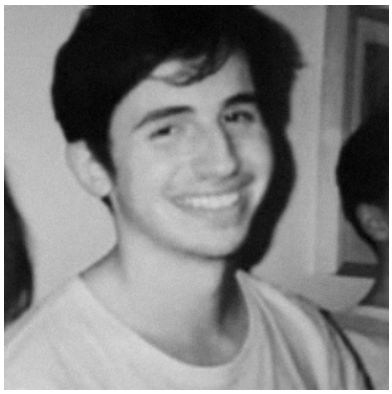
CHAPTER

SEVEN

CONTRIBUTORS

Simon Hulse
Ph.D. Student 2019-
Project's main developer

Mohammadali Foroozandeh
Simon's Ph.D. supervisor



Thomas Moss
MChem Student Jan 2020 - June 2020
Helped in extending NMR-EsPy to support 2D data