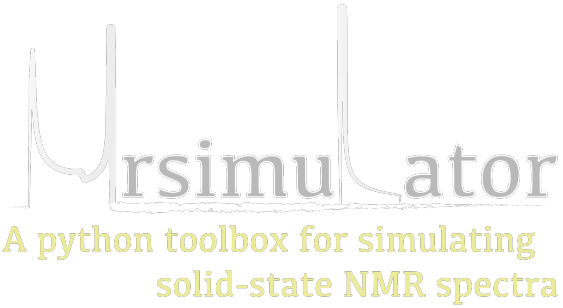


mrsimulator Documentation

Release 0.5.1

Mrsimulator Developers

Dec 10, 2020

GETTING STARTED

About

`mrsimulator` is an open-source python package for fast simulation and analysis of multi-dimensional solid-state magnetic resonance (NMR) spectra of crystalline materials, bio macro-molecules, and even amorphous materials. Simulate the NMR spectrum of macro-molecules or amorphous in just a few seconds.

Why use mrsimulator?

- It is open-source and free.
- It is a fast and versatile multi-dimensional solid-state NMR spectra simulator including, MAS and VAS spectra of nuclei experiencing chemical shift (nuclear shielding) and quadrupolar coupling interactions.
- Future release will include simulations of weakly coupled nuclei experiencing J and dipolar couplings, and multi-dimensional NMR spectra.
- It is fully documented with a stable and simple API and is easily incorporated into your python scripts and web apps.
- It is compatible with modern python packages, such as scikit-learn, Keras, etc.
- Packages using mrsimulator -
  - mrinversion

Features

The `mrsimulator` package currently offers the following

- **Fast simulation** of one-dimensional solid-state NMR spectra. See our *Benchmark* (page 140) results.
- Simulation of uncoupled spin system
  - for spin $I = \frac{1}{2}$, and quadrupole $I \geq \frac{1}{2}$ nuclei,
  - at arbitrary macroscopic magnetic flux density,
  - at arbitrary rotor angles, and
  - at arbitrary spinning frequency.
- The library includes the following NMR methods,
  - 1D Bloch decay spectrum, and
  - 1D Bloch decay central transition spectrum.
  - 2D Multi-quantum Variable Angle Spinning (MQ-VAS),
  - 2D Satellite-transition Variable Angle Spinning (ST-VAS), and
  - 2D Dynamic Angle Spinning (DAS),
  - 2D isotropic/anisotropic sideband correlation spectrum (e.g. PASS and MAT), and
  - 2D Magic Angle Flipping (MAF).
- Models for tensor parameter distribution in amorphous materials.
  - Czjzek
  - Extended Czjzek

Goals for the near future

Our current objectives are the following

- Include spectral simulation of coupled spin systems.

Warning: The package is currently under development. We advice using with caution. Bug report are greatly appreciated.

CHAPTER

ONE

GETTING STARTED

1.1 Installation

1.1.1 Requirements

`mrsimulator` has the following strict requirements:

- Python 3.6 or later
- Numpy 1.17 or later

See *Package dependencies* (page 9) for a full list of requirements.

Make sure you have the required version of python by typing the following in the terminal,

Tip: You may also click the copy-button located at the top-right corner of the code cell area in the HTML docs, to copy the code lines without the prompts and then paste it as usual. Thanks to Sphinx-copybutton)

```
$ python --version
```

For *Mac* users, python version 3 is installed under the name *python3*. You may replace *python* for *python3* in the above command and all subsequent python statements.

For *Windows* users, we recommend the Anaconda or miniconda distribution of python>3.6. Anaconda distribution for python comes with popular python packages that are frequently used in scientific computing. Miniconda is a minimal installer for conda. It is a smaller version of Anaconda that includes conda, Python, and the packages they depend on, along with other useful packages such as pip.

See also:

If you do not have python or have an older version of python, you may visit the Python downloads or Anaconda websites and follow their instructions on how to install python.

1.1.2 Installing `mrsimulator`

On Google Colab Notebook

Colaboratory is a Google research project. It is a Jupyter notebook environment that runs entirely in the cloud. Launch a new notebook on Colab. To install the mrsimulator package, type

```
!pip install mrsimulator
```

in the first cell, and execute. All done! You may now proceed to the next section and start using the library.

On Local machine (Using pip)

PIP is a package manager for Python packages and is included with python version 3.4 and higher. PIP is the easiest way to install python packages.

Linux

For Linux users, we provide the binary distributions of the mrsimulator package for python versions 3.6-3.8. Install the package using pip as follows,

```
$ pip install mrsimulator
```

Mac OSX

For *Mac* users, we provide the binary distributions of the mrsimulator package for python versions 3.6-3.8. Install the package using pip as follows,

```
$ pip install mrsimulator
```

If the above statement didn't work, you are probably using mac OS system python, in which case, use the following,

```
$ python3 -m pip install mrsimulator --user
```

Windows

Note: We currently do not provide binary distributions for windows. You'll need to compile and build the mrsimulator library from source. The following instructions are one-time installation only. If you are upgrading the package, see the *Upgrading to a newer version* (page 5) sub-section.

Install conda

Skip this step if you already have miniconda or anaconda for python>=3.6 installed on your system. Download the latest version of conda on your operating system from either miniconda or Anaconda websites. Make sure you download conda for python 3. Double click the downloaded .exe file and follow the installation steps.

OpenBLAS and FFTW libraries

Launch the `Anaconda prompt` (it should be located under the start menu). Within the anaconda prompt, type the following to install the package dependencies.

```
$ conda install -c conda-forge openblas fftw
```

Install a C/C++ compiler

Because the core of the mrsimulator package is written in C, you will require a C-compiler to build and install the package. Download and install the Microsoft Visual C++ compiler from Build Tools for Visual Studio 2019.

Install the package.

From within the `Anaconda Prompt`, build and install the mrsimulator package using pip.

```
$ pip install mrsimulator
```

If you get a `PermissionError`, it usually means that you do not have the required administrative access to install new packages to your Python installation. In this case, you may consider adding the `--user` option, at the end of the statement, to install the package into your home directory. You can read more about how to do this in the pip documentation.

Upgrading to a newer version

If you are upgrading to a newer version of `mrsimulator`, you have all the prerequisites installed on your system. In this case, type the following in the terminal/Prompt

```
$ pip install mrsimulator -U
```

1.1.3 Building from the source

Prerequisites

You will need a C-compiler suite and the development headers for the BLAS and FFTW libraries, along with development headers from Python and Numpy, to build the `mrsimulator` library from source. The mrsimulator package utilizes the BLAS and FFTW routines for numerical computation. To leverage the best performance, we recommend installing the BLAS and FFTW libraries, which are optimized and tuned for your system. In the following, we list recommendations on how to install the c-compiler (if applicable), BLAS, FFTW, and building the mrsimulator libraries.

Obtaining the Source Packages

Stable packages

The latest stable source package for `mrsimulator` is available on PyPI.

OS-dependent prerequisites

Note: Installing OS-dependent prerequisites is a one-time process. If you are upgrading to a newer version of mrsimulator, skip to *Building and Installing* (page 7) section.

Linux

OpenBLAS and FFTW libraries

On Linux, the package manager for your distribution is usually the easiest route to ensure you have the prerequisites to building the mrsimulator library. To build from source, you will need the OpenBLAS and FFTW development headers for your Linux distribution. Type the following command in the terminal, based on your Linux distribution.

For (Debian/Ubuntu):

```
$ sudo apt-get install libopenblas-dev libfftw3-dev
```

For (Fedora/RHEL):

```
$ sudo yum install openblas-devel fftw-devel
```

Install a C/C++ compiler

The C-compiler comes with your Linux distribution. No further action is required.

Mac OSX

OpenBLAS/Accelerate and FFTW libraries

You will require the `brew` package manager to install the development headers for the OpenBLAS (if applicable) and FFTW libraries. Read more on installing brew from homebrew.

Step-1 Install the FFTW library using the homebrew formulae.

```
$ brew install fftw
```

Step-2 By default, the mrsimulator package links to the openblas library for BLAS operations. Mac users may opt to choose the in-build Apple's Accelerate library. If you opt for Apple's Accelerate library, skip to *Step-3*. If you wish to link the mrsimulator package to the OpenBLAS library, type the following in the terminal,

```
$ brew install openblas
```

Step-3 If you choose to link the mrsimulator package to the OpenBLAS library, skip to the next section, *Building and Installing* (page 7).

(a) You will need to install the BLAS development header for Apple's Accelerate library. The easiest way is to install the Xcode Command Line Tools. Note, this is a one-time installation. If you have previously installed the Xcode Command Line Tools, you may skip this sub-step. Type the following in the terminal,

```
$ xcode-select --install
```

(b) The next step is to let the mrsimulator setup know your preference. Open the `settings.py` file, located at the root level of the mrsimulator source code folder, in a text editor. You should see

```
# -*- coding: utf-8 -*-
# BLAS library
use_openblas = True
# mac-os only
use_accelerate = False
```

To link the mrsimulator package to the Apple's Accelerate library, change the fields to

```
# -*- coding: utf-8 -*-
# BLAS library
use_openblas = False
# mac-os only
use_accelerate = True
```

Install a C/C++ compiler

The C-compiler installs with the Xcode Command Line Tools. No further action is required.

Windows

Install conda

Skip this step if you already have miniconda or anaconda for python>=3.6 installed on your system. Download the latest version of conda on your operating system from either miniconda or Anaconda websites. Make sure you download conda for python 3. Double click the downloaded .exe file and follow the installation steps.

OpenBLAS and FFTW libraries

Launch the `Anaconda prompt` (it should be located under the start menu). Within the anaconda prompt, type the following to install the package dependencies.

```
$ conda install -c conda-forge openblas fftw
```

Install a C/C++ compiler

Because the core of the mrsimulator package is written in C, you will require a C-compiler to build and install the package. Download and install the Microsoft Visual C++ compiler from Build Tools for Visual Studio 2019.

Building and Installing

Use the terminal/Prompt to navigate into the directory containing the package (usually, the folder is named mrsimulator),

```
$ cd mrsimulator
```

From within the source code folder, type the following in the terminal to install the library.

```
$ pip install .
```

If you get an error that you don't have the permission to install the package into the default `site-packages` directory, you may try installing with the `--user` options as,

```
$ pip install . --user
```

1.1.4 Test your build

If the installation is successful, you should be able to run the following test file in your terminal. Download the test file here.

```
$ python test_file.py
```

The above statement should produce the following figure.

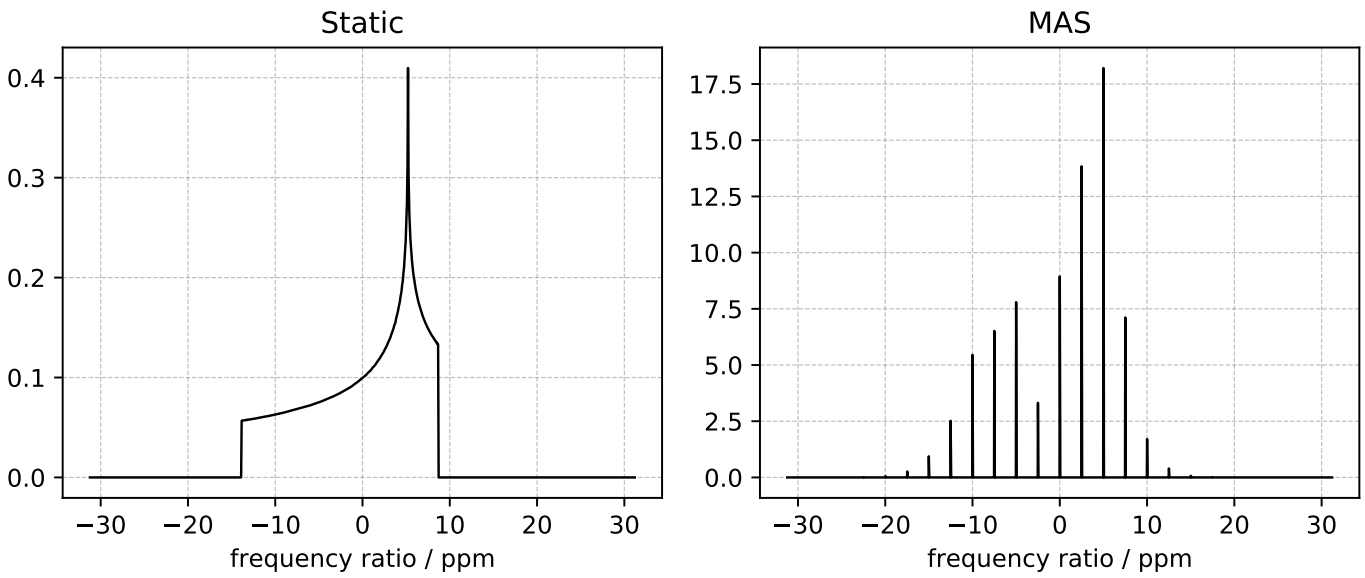


Figure 1.1: A test example simulation of solid-state NMR spectrum.

1.1.5 Setup for developers and contributors

A GitHub account is required for developers and contributors. Make sure you have git installed on your system.

Step-A (Optional) Create a virtual environment. It is a good practice to create separate virtual python environments for packages when in developer mode. The following is an example of a Conda environment.

```
$ conda create -n mrsimulator-dev python=3.7
$ conda activate mrsimulator-dev
```

Step-B Clone the mrsimulator repository using git and navigate into the package folder.

```
$ git clone git://github.com/DeepanshS/mrsimulator.git
$ cd mrsimulator
```

Step-C Follow the instruction under *OS-dependent prerequisites* (page 5) from *Building from the source* (page 5) section. For developers and contributors using mac OSX, please run the setup by binding to the openblas libraries.

Step-D You will need cython for development build.

```
$ pip install cython
```

Step-E Build and install the package in the development (editable) mode using pip.

```
$ pip install -e .
```

Step-F: Install the required packages for developers using pip.

```
$ pip install -r requirements-dev.txt
```

As always, if you get an error that you don't have the permission to install the package into the default site-packages directory, you may try installing by adding the `--user` options at the end of the statements in steps D-F.

Note for the developers and contributors

Running tests: For unit tests, we use the pytest module. At the root directory of the mrsimulator package folder, type

```
$ pytest
```

which will run a series of tests.

Building docs: We use the sphinx python documentation generator for building docs. Navigate to the `docs` folder within the mrsimulator package folder, and type,

```
$ make html
```

The above command will build the documentation and store the build at `mrsimulator/docs/_build/html`. Double click the *index.html* file within this folder to view the offline documentation.

1.2 Package dependencies

`mrsimulator` depends on the following packages:

Required packages

- NumPy>=1.17
- openblas
- cython>=0.29.14
- typing-extensions>=3.7
- matplotlib>=3.1 for figures and visualization,
- monty>=2.0.4
- csdmpy>=0.3.4
- pydantic>=1.0
- monty>=2.0.4

Other packages

- pytest>=4.5.0 for unit tests.
- pre-commit for code formatting
- sphinx>=2.0 for generating the documentation
- sphinxjp.themes.basicstrap for documentation.
- breathe>4.19 for generating C documentation
- sphinx-copybutton

1.3 Introduction to Spin Systems

At the heart of any `mrsimulator` calculation is the definition of a **SpinSystem** object describing the sites and couplings within a spin system. We begin by examining the definition of a **Site** object.

1.3.1 Site

Consider the example below of the JSON serialization of a **Site** object for a deuterium nucleus.

Listing 1.1: An example 2H site in JSON representation.

```
{
       "isotope": "2H",
       "isotropic_chemical_shift": "4.1 ppm",
       "shielding_symmetric": {
               "zeta": "12.12 ppm",
               "eta": 0.82
               },
       "quadrupolar": {
               "Cq": "1.47 MHz",
               "eta": 0.27,
               "alpha": "0.212 rad",
```

(continues on next page)

(continued from previous page)

```
            "beta": "1.231 rad",
            "gamma": "3.1415 rad"
            }
}
```

The value of the *isotope* key holds the spin isotope, here given a value of *2H*. The value of the *isotropic_chemical_shift* is the optional ^{2}H isotropic chemical shift, here given as *4.1 ppm*. We have additionally defined an optional *shielding_symmetric*, whose value holds a dictionary with the components of the second-rank traceless symmetric nuclear shielding tensor. We parameterize this tensor using the Haeberlen convention with parameters *zeta* and *eta*, defined as the strength of the anisotropy and asymmetry, respectively. Since deuterium is a quadrupolar nucleus, $I > 1/2$, there also can be a quadrupolar coupling interaction between the nuclear quadrupole moment and the surrounding electric field gradient (EFG) tensor, defined in a dictionary held in the optional key *quadrupolar*. An EFG tensor is a second-rank traceless symmetric tensor, and we describe the quadrupolar coupling with the parameters *Cq* and *eta*, i.e., the quadrupolar coupling constant and asymmetry parameter, respectively. Additionally, we see the Euler angle orientations, *alpha*, *beta*, and *gamma*, which are the relative orientation of the EFG tensor from the nuclear shielding tensor.

See Table 1.1 and Table 1.2 for further information on the **Site** and **SymmetricTensor** objects and their attributes, respectively.

Table of Site Class Attributes

Table 1.1: The attributes of a Site object.

Attribute name	Type	Description
`isotope`	String	A *required* isotope string given as the atomic number followed by the isotope symbol, for example, `13C`, `29Si`, `27Al`, and so on.
`isotropic_chemical_shift`	ScalarQuantity	An *optional* physical quantity describing the isotropic chemical shift of the site. The value is given in dimensionless frequency ratio, for example, `10 ppm` or `10 µHz/Hz`. The default value is `0 ppm`.
`shielding_symmetric`	symmetric_tensor	An *optional* object describing the second-rank traceless symmetric nuclear shielding tensor following the Haeberlen convention. The default is a `NULL` object. See the description for the symmetric_tensor object.
`quadrupolar`	symmetric_tensor	An *optional* object describing the second-rank traceless electric quadrupole tensor. The default is a `NULL` object. See the description for the symmetric_tensor object.

Table 1.2: The attributes of a SymmetricTensor object.

Attribute name	Type	Description

`zeta`
or
`Cq`
	ScalarQuantity	A *required* quantity.
		Nuclear shielding: The strength of the anisotropy, `zeta`, calculated using the Haeberlen convention. The value is a physical quantity given in dimensionless frequency ratio, for example, `10 ppm` or `10 µHz/Hz`.
		Electric quadrupole: The quadrupole coupling constant, `Cq`. The value is a physical quantity given in units of frequency, for example, `3.1 MHz`.

`eta`	Float	A *required* asymmetry parameter calculated using the Haeberlen convention, for example, `0.75`.
`alpha`	ScalarQuantity	An *optional* Euler angle, α. For example, `2.1 rad`. The default value is `0 rad`.
`beta`	ScalarQuantity	An *optional* Euler angle, β. For example, `90°`. The default value is `0 rad`.
`gamma`	ScalarQuantity	An *optional* Euler angle, γ. For example, `0.5 rad`. The default value is `0 rad`.

1.3.2 SpinSystem

As mentioned earlier, the **SpinSystem** object, used in the `mrsimulator` package, describes the sites and couplings within a spin system.

Uncoupled spin systems

Using the previous **Site** object example, we construct a simple single site **SpinSystem** object shown below.

Listing 1.2: An example 2H spin system in JSON representation.

```
{
    "name": "2H spin system",
    "description": "An optional description on the spin system",
    "sites": [
        {
            "isotope": "2H",
            "isotropic_chemical_shift": "4.1 ppm",
            "shielding_symmetric": {
                "zeta": "12.12 ppm",
                "eta": 0.82
            },
            "quadrupolar": {
                "Cq": "1.47 MHz",
                "eta": 0.27,
                "alpha": "0.212 rad",
                "beta": "1.231 rad",
                "gamma": "3.1415 rad"
            }
        }
    ],
    "couplings": [],
    "abundance": "0.148%"
}
```

At the root level of the **SpinSystem** object, we find four keywords, **name**, **description**, **sites**, and **abundance**. The value of the *name* key is the name of the spin system, here given a value of *2H spin system*. The value of the description key is an optional string describing the spin system. The value of the *sites* key is a list of **Site** objects. Here, this list comprises of a single **Site** object (lines 5-19). The value of the *abundance* key is the abundance of the spin system, here given a value of *0.148%*. The value of the *couplings* key is a list of **Coupling** objects. In this example, there are no couplings, and hence the value of this attribute is an empty list. See Table 1.3 for further description of the **SpinSystem** class and its attributes.

Table 1.3: The attributes of a SpinSystem object.

Attributes	Type	Description
`name`	String	An *optional* attribute with a name for the spin system. Naming is a good practice as it improves the readability, especially when multiple spin systems are present. The default value is an empty string.
`description`	String	An *optional* attribute describing the spin system. The default value is an empty string.
`sites`	List	An *options* list of site objects. The default value is an empty list.
`couplings`	List	An *optional* list of coupling objects. The default value is an empty list. Not yet implemented.
`abundance`	String	An *optional* quantity representing the abundance of the spin system. The abundance is given as percentage, for example, `25.4 %`. This value is useful when multiple spin systems are present. The default value is `100 %`.

Coupled spin systems

Note: The current version of the `mrsimulator` package does not include coupled spin systems. The SpinSystem model for the couplings will be made available when we include the coupled spin systems to the package. The `mrsimulator` package will eventually handle coupled spin systems, but only in the weak coupling limit.

1.4 Getting started with `mrsimulator`: The basics

We have put together a set of guidelines for using the `mrsimulator` package. We encourage our users to follow these guidelines for consistency. In `mrsimulator`, the solid-state nuclear magnetic resonance (ssNMR) spectrum is calculated through an instance of the *Simulator* (page 151) class.

Import the *Simulator* (page 151) class using

```
>>> from mrsimulator import Simulator
```

and create an instance as follows,

```
>>> sim = Simulator()
```

Here, the variable `sim` is an instance of the *Simulator* (page 151) class. The two attributes of this class that you will frequently use are the *spin_systems* (page 151) and *methods* (page 151), whose values are a list of *SpinSystem* (page 158) and *Method* (page 170) objects, respectively. The default value of these attributes is an empty list.

```
>>> sim.spin_systems
[]
>>> sim.methods
[]
```

Before you can start simulating the NMR spectrum, you need to understand the role of the SpinSystem and Method objects. The following provides a brief description of the respective objects.

1.4.1 Setting up the SpinSystem objects

An NMR spin system is an isolated system of sites (spins) and couplings. You may construct a spin system with as many sites and couplings, as necessary, for this example, we stick to a single-site spin system. Let's start by first building a site.

A site object is a collection of attributes that describe site-specific interactions. In NMR, these spin interactions are described by a second-rank tensor. Site-specific interactions include the interaction between the magnetic dipole moment of the nucleus and the surrounding magnetic field and the interaction between the electric quadrupole moment of the nucleus with the surrounding electric field gradient. The latter is zero for sites with the spin quantum number, $I = 1/2$.

Let's start with a spin-1/2 isotope, ^{29}Si, and create a site.

```
>>> the_site = {
...     "isotope": "29Si",
...     "isotropic_chemical_shift": "-101.1 ppm",
...     "shielding_symmetric": {"zeta": "70.5 ppm", "eta": 0.5},
... }
```

In the above code, `the_site` is a simplified python dictionary representation of a *Site* (page 163) object. This site describes a ^{29}Si isotope with a -101.1 ppm isotropic chemical shift along with the symmetric part of the nuclear shielding anisotropy tensor, described here with the parameters *zeta* and *eta* using the Haeberlen convention.

That's it! Now that we have a site, we can create a single-site spin system following,

```
>>> the_spin_system = {
...     "name": "site A",
...     "description": "A test 29Si site",
...     "sites": [the_site],  # from the above code
...     "abundance": "80%",
... }
```

As mentioned before, a spin system is a collection of sites and couplings. In the above example, we have created a spin system with a single site and no couplings. Here, the attribute *sites* hold a list of sites. The attributes *name*, *description*, and *abundance* are optional.

Until now, we have only created a python dictionary representation of a spin system. To run the simulation, you need to create an instance of the *SpinSystem* (page 158) class. Import the SpinSystem class and use it's *parse_dict_with_units()* (page 161) method to parse the python dictionary and create an instance of the spin system class, as follows,

```
>>> from mrsimulator import SpinSystem
>>> system_object_1 = SpinSystem.parse_dict_with_units(the_spin_system)
```

Note: We provide the *parse_dict_with_units()* (page 161) method because it allows the user to create spin systems, where the attribute value is a physical quantity, represented as a string with a value and a unit. Physical quantities remove the ambiguity in the units, which is otherwise a source of general confusion within many scientific applications. With this said, parsing physical quantities can add significant overhead when used in an iterative algorithm, such as the least-squares minimization. In such cases, we recommend defining objects directly. See the *Getting started with mrsimulator: Using objects* (page 16) for details.

We have successfully created a spin system object. To create more spin system objects, repeat the above set of instructions. In this example, we stick with a single spin system object. Once all spin system objects are ready, add these objects to the instance of the Simulator class, as follows

```
>>> sim.spin_systems += [system_object_1] # add all spin system objects.
```

1.4.2 Setting up the Method objects

A *Method* (page 170) object is a collection of attributes that describe an NMR method. In `mrsimulator`, all methods are described through five keywords -

Keywords	Description
channels	A list of isotope symbols over which the given method applies.
magnetic_flux_density	The macroscopic magnetic flux density of the applied external magnetic field.
rotor_angle	The angle between the sample rotation axis and the applied external magnetic field.
rotor_frequency	The sample rotation frequency.
spectral_dimensions	A list of spectral dimensions. The coordinates along each spectral dimension is described with the keywords, *count* (N), *spectral_width* (ν_{sw}), and *reference_offset* (ν_0). The coordinates are given as, $$\left([0, 1, 2, ...N-1] - \frac{T}{2}\right)\frac{\nu_{sw}}{N} + \nu_0 \quad (1.1)$$ where $T = N$ when N is even else $T = N - 1$.

Let's start with the simplest method, the *BlochDecaySpectrum()* (page 177). The following is a python dictionary representation of the BlochDecaySpectrum method.

```
>>> method_dict = {
...     "channels": ["29Si"],
...     "magnetic_flux_density": "9.4 T",
...     "rotor_angle": "54.735 deg",
...     "rotor_frequency": "0 Hz",
...     "spectral_dimensions": [{
...         "count": 2048,
...         "spectral_width": "25 kHz",
...         "reference_offset": "-8 kHz",
...         "label": r"$^{29}$Si resonances",
...     }]
... }
```

Here, the key *channels* is a list of isotope symbols over which the method is applied. A Bloch Decay method only has a single channel. In this example, it is given a value of `29Si`, which implies that the simulated spectrum from this method will comprise frequency components arising from the ^{29}Si resonances. The keys *magnetic_flux_density*, *rotor_angle*, and *rotor_frequency* collectively describe the spin environment under which the resonance frequency is evaluated. The key *spectral_dimensions* is a list of spectral dimensions. A Bloch Decay method only has one spectral dimension. In this example, the spectral dimension defines a frequency dimension with 2048 points, spanning for 25 kHz with a reference offset of -8 kHz.

Like before, you may parse the above `method_dict` using the `parse_dict_with_units()` function of the method. Import the BlochDecaySpectrum class and create an instance of the method, following,

```
>>> from mrsimulator.methods import BlochDecaySpectrum
>>> method_object = BlochDecaySpectrum.parse_dict_with_units(method_dict)
```

Here, `method_object`, is an instance of the *Method* (page 170) class.

Likewise, you may create multiple method objects. In this example, we stick with a single method. Finally, add all the method objects, in this case, `method_object`, to the instance of the Simulator class, `sim`, as follows,

```
>>> sim.methods += [method_object] # add all methods.
```

1.4.3 Running simulation

To simulate the spectrum, run the simulator with the *run()* (page 156) method, as follows,

```
>>> sim.run()
```

Note: In `mrsimulator`, all resonant frequencies are calculated assuming the weakly-coupled (Zeeman) basis for the spin system.

The simulator object, `sim`, will process every method over all the spin systems and store the result in the *simulation* (page 171) attribute of the respective Method object. In this example, we have a single method. You may access the simulation data for this method as,

```
>>> data_0 = sim.methods[0].simulation
>>> # data_n = sim.method[n].simulation # when there are multiple methods.
```

Here, `data_0` is a CSDM object holding the simulation data from the method at index 0 of the *methods* (page 151) attribute from the `sim` object.

See also:

The core scientific dataset model (CSDM)[1] is a lightweight and portable file format model for multi-dimensional scientific datasets and is supported by numerous NMR software—DMFIT, SIMPSON, jsNMR, and RMN. We also provide a python package csdmpy.

1.4.4 Visualizing the dataset

At this point, you may continue with additional post-simulation processing. We end this example with a plot of the data from the simulation. Figure 1.2 depicts the plot of the simulated spectrum.

For a quick plot of the csdm data, you may use the csdmpy library. The *csdmpy* package uses the matplotlib library to produce basic plots. You may optionally customize the plot using matplotlib methods.

```
>>> import matplotlib.pyplot as plt
>>> plt.figure(figsize=(6, 3.5)) # set the figure size
>>> ax = plt.subplot(projection='csdm')
>>> ax.plot(data_0)
>>> ax.invert_xaxis() # reverse x-axis
>>> plt.tight_layout(pad=0.1)
>>> plt.show()
```

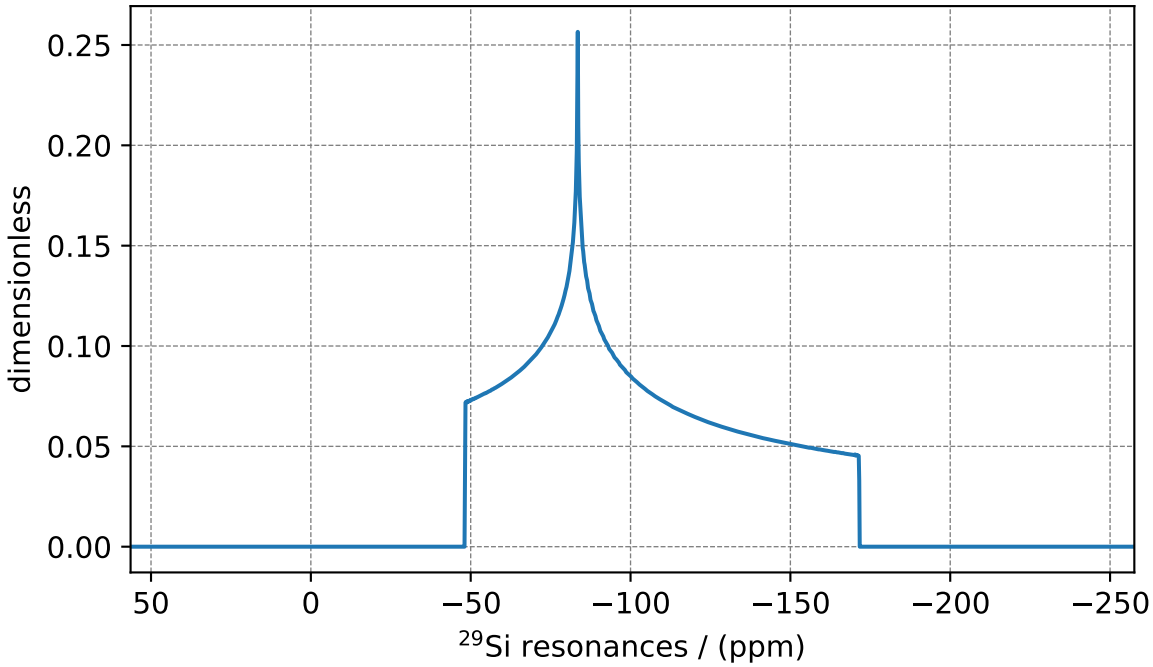


Figure 1.2: An example of solid-state static NMR spectrum simulation.

[1] Srivastava, D. J., Vosegaard, T., Massiot, D., Grandinetti, P. J. Core Scientific Dataset Model: A lightweight and portable model and file format for multi-dimensional scientific data. PLOS ONE, 2020, 15, 1. DOI 10.1371/e0225953

1.5 Getting started with `mrsimulator`: Using objects

In the previous section on getting started, we show an example where we parse the python dictionaries to create instances of the *Spin-System* (page 158) and *Method* (page 170) objects. In this section, we'll illustrate how we can achieve the same result using the core `mrsimulator` objects.

Note: Unlike python dictionary objects from our last example, when using `mrsimulator` objects, the attribute value is given as a number rather than a string with a number and a unit. We assume default units for the class attributes. To learn more about the default units, please refer to the documentation of the respective class. For the convenience of our users, we have added an attribute, `property_units`, to every class that holds the default unit of the respective class attributes.

Let's start by importing the classes.

```
>>> from mrsimulator import Simulator, SpinSystem, Site
>>> from mrsimulator.methods import BlochDecaySpectrum
```

The following code is used to produce the figures in this section.

```
>>> import matplotlib.pyplot as plt
>>> import matplotlib as mpl
>>> mpl.rcParams["figure.figsize"] = (6, 3.5)
>>> mpl.rcParams["font.size"] = 11
...
>>> # function to render figures.
>>> def plot(csdm_object):
...     # set matplotlib axes projection='csdm' to directly plot CSDM objects.
...     ax = plt.subplot(projection='csdm')
...     ax.plot(csdm_object, linewidth=1.5)
...     ax.invert_xaxis()
...     plt.tight_layout()
...     plt.show()
```

1.5.1 Site object

As the name suggests, a *Site* (page 163) object is used in creating sites. For example,

```
>>> C13A = Site(isotope='13C')
```

The above code creates a site with a ^{13}C isotope. Because, no further information is delivered to the site object, other attributes such as the isotropic chemical shift assume their default value.

```
>>> C13A.isotropic_chemical_shift # value is given in ppm
0
```

Here, the isotropic chemical shift is given in ppm. This information is also present in the `property_units` attribute of the instance. For example,

```
>>> C13A.property_units
{'isotropic_chemical_shift': 'ppm'}
```

Let's create a few more sites.

```
>>> C13B = Site(isotope='13C', isotropic_chemical_shift=-10)
>>> H1 = Site(isotope='1H', shielding_symmetric=dict(zeta=5.1, eta=0.1))
>>> O17 = Site(isotope='17O', isotropic_chemical_shift=41.7, quadrupolar=dict(Cq=5.15e6, eta=0.
↪21))
```

The site, `C13B`, is the second ^{13}C site with an isotropic chemical shift of -10 ppm.

In creating the site, `H1`, we use the dictionary object to describe a traceless symmetric second-rank irreducible nuclear shielding tensor, using the attributes *zeta* and *eta*, respectively. The parameter *zeta* and *eta* are defined as per the Haeberlen convention and describes the anisotropy and asymmetry parameter of the tensor, respectively. The default unit of the attributes from the *shielding_symmetric* is found with the `property_units` attribute, such as

```
>>> H1.shielding_symmetric.property_units
{'zeta': 'ppm', 'alpha': 'rad', 'beta': 'rad', 'gamma': 'rad'}
```

For site, `O17`, we once again make use of the dictionary object, only this time to describe a traceless symmetric second-rank irreducible electric quadrupole tensor, using the attributes *Cq* and *eta*, respectively. The parameter *Cq* is the quadrupole coupling constant, and *eta* is the asymmetry parameters of the quadrupole tensor, respectively. The default unit of these attributes is once again found with the `property_units` attribute,

```
>>> O17.quadrupolar.property_units
{'Cq': 'Hz', 'alpha': 'rad', 'beta': 'rad', 'gamma': 'rad'}
```

1.5.2 SpinSystem object

A SpinSystem object contains sites and couplings along with the abundance of the respective spin system. In this version, we focus on the spin systems with a single site, and therefore the couplings are irrelevant.

Let's use the sites we have already created to set up four spin systems.

```
>>> system_1 = SpinSystem(name='C13A', sites=[C13A], abundance=20)
>>> system_2 = SpinSystem(name='C13B', sites=[C13B], abundance=56)
>>> system_3 = SpinSystem(name='H1', sites=[H1], abundance=100)
>>> system_4 = SpinSystem(name='O17', sites=[O17], abundance=1)
```

1.5.3 Method object

Likewise, we can create a *BlochDecaySpectrum* (page 177) object following,

```
>>> from mrsimulator.methods import BlochDecaySpectrum
>>> method_1 = BlochDecaySpectrum(
...     channels=["13C"],
...     spectral_dimensions = [dict(
...         count=2048,
...         spectral_width=25000, # in Hz.
...         label=r"$^{13}$C resonances",
...     )]
... )
```

The above method, `method_1`, is defined to record ^{13}C resonances over 25 kHz spectral width using 2048 points. The unspecified attributes, such as *rotor_frequency*, *rotor_angle*, *magnetic_flux_density*, assume their default value. The default units of these attributes is once again found with the `property_units` attribute,

```
>>> method_1.property_units
{'magnetic_flux_density': 'T', 'rotor_angle': 'rad', 'rotor_frequency': 'Hz'}
```

1.5.4 Simulator object

The use of the simulator object is the same as described in the previous section.

```
>>> sim = Simulator()
>>> sim.spin_systems += [system_1, system_2, system_3, system_4] # add the spin systems
>>> sim.methods += [method_1] # add the method
```

1.5.5 Running simulation

Let's run the simulator and observe the spectrum.

```
>>> sim.run()
>>> plot(sim.methods[0].simulation)
```

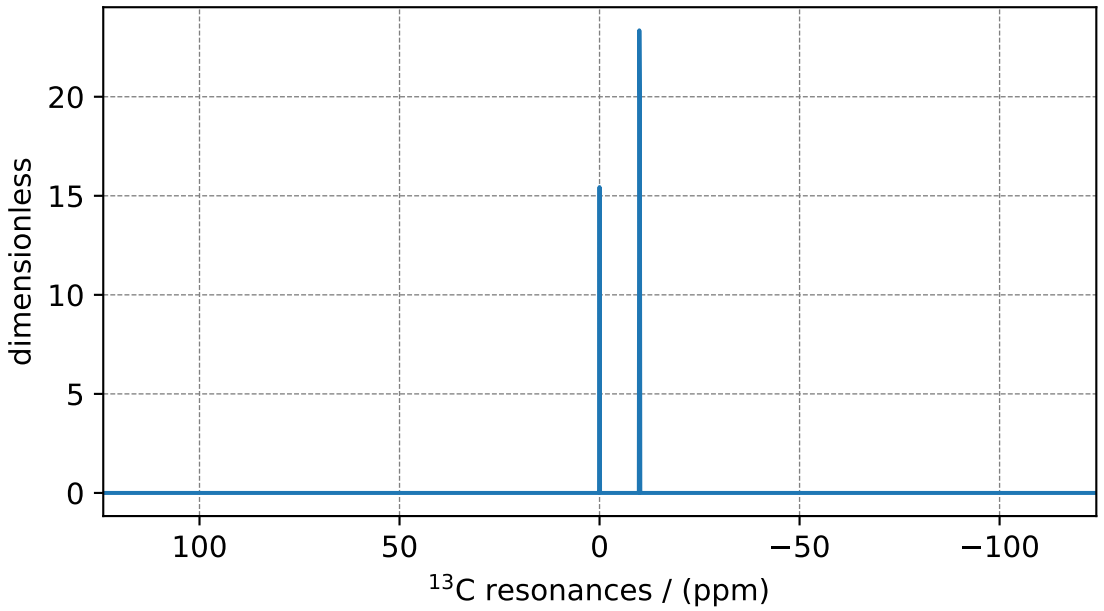


Figure 1.3: An example solid-state NMR simulation of ^{13}C isotropic spectrum.

Notice, we have four single-site spin systems within the `sim` object, two with ^{13}C sites, one with ^{1}H site, and one with an ^{17}O site, along with a BlochDecaySpectrum method which is tuned to record the resonances from the ^{13}C channel. When you run this simulation, only ^{13}C resonances are recorded, as seen from Figure 1.3, where just the two ^{13}C isotropic chemical shifts resonances are observed.

Modifying the site attributes

Let's modify the `C13A` and `C13B` sites by adding the shielding tensors information.

```
>>> sim.spin_systems[0].sites[0].shielding_symmetric = dict(zeta=80, eta=0.5) # site C13A
>>> sim.spin_systems[1].sites[0].shielding_symmetric = dict(zeta=-100, eta=0.25) # site C13B
```

Running the simulation with the previously defined method will produce two overlapping CSA patterns, see Figure 1.4.

```
>>> sim.run()
>>> plot(sim.methods[0].simulation)
```

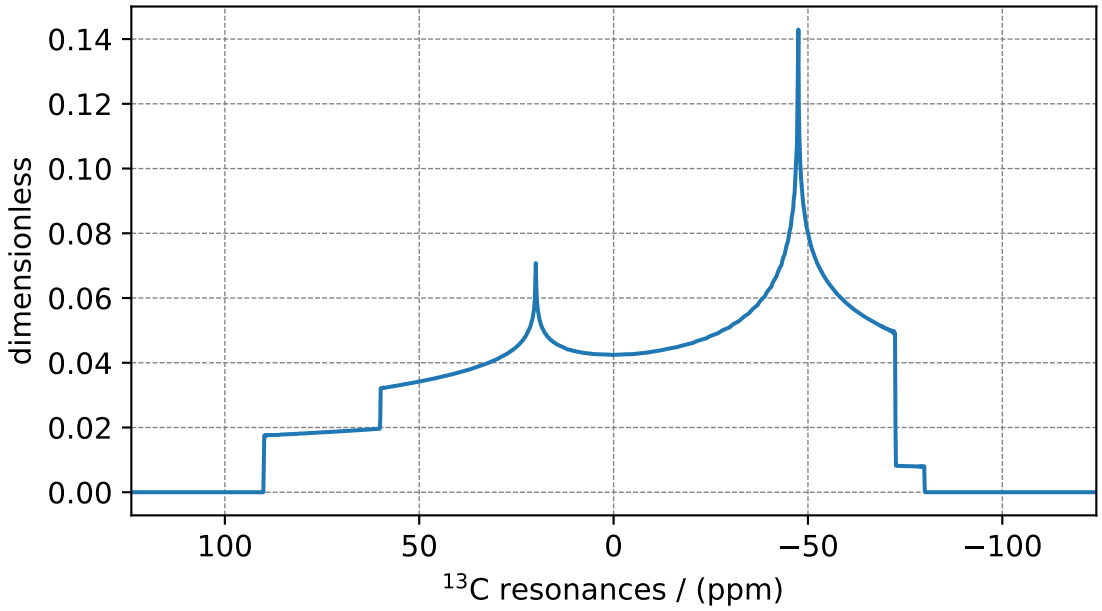


Figure 1.4: An example state-solid NMR simulation of static ^{13}C CSA spectrum.

Modifying the rotor frequency of the method

Let's turn up the rotor frequency from 0 Hz (default) to 1 kHz. Note, that we do not add another method to the `sim` object, but update the existing method at index 0 with a new method. Figure 1.5 depicts the simulation from this method.

```
>>> # Update the method object at index 0.
>>> sim.methods[0] = BlochDecaySpectrum(
...     channels=["13C"],
...     rotor_frequency=1000, # in Hz.  <------------ updated entry
...     spectral_dimensions=[dict(
...         count=2048,
...         spectral_width=25000, # in Hz.
...         label=r"$^{13}$C resonances",
...     )]
... )
```

```
>>> sim.run()
>>> plot(sim.methods[0].simulation)
```

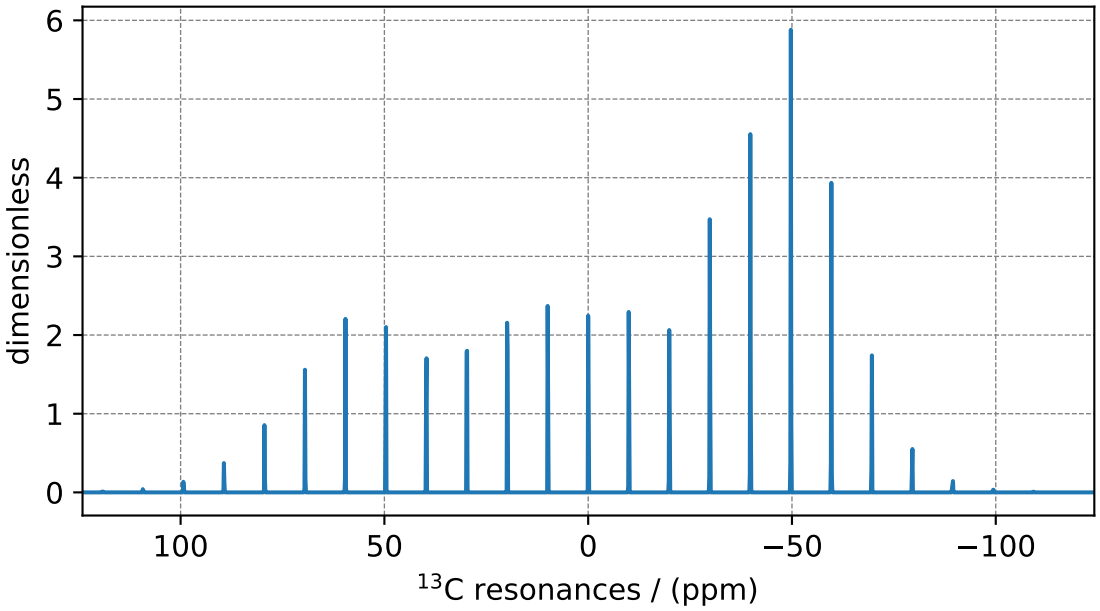


Figure 1.5: An example of the solid-state ^{13}C MAS sideband simulation.

Modifying the rotor angle of the method

Let's also set the rotor angle from magic angle (default) to 90 degrees. Again, we update the method at index 0. Figure 1.6 depicts the simulation from this method.

```
>>> # Update the method object at index 0.
>>> sim.methods[0] = BlochDecaySpectrum(
...     channels=["13C"],
...     rotor_frequency=1000, # in Hz.
...     rotor_angle=90*3.1415926/180, # 90 degree in radians.  <------------ updated entry
...     spectral_dimensions=[dict(
...         count=2048,
...         spectral_width=25000, # in Hz.
...         label=r"$^{13}$C resonances",
...     )]
... )
```

```
>>> sim.run()
>>> plot(sim.methods[0].simulation)
```

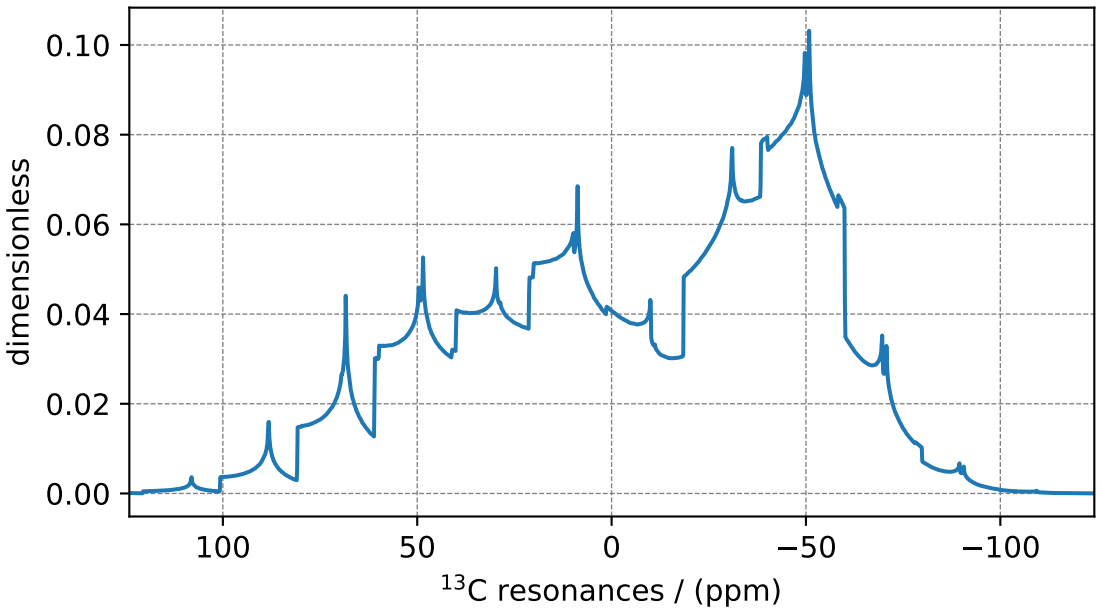


Figure 1.6: An example of the solid-state ^{13}C VAS sideband simulation.

Switching the detection channels of the method

To switch to another channels, update the value of the *channels* attribute of the method. Here, we update the method to *1H* channel.

```
>>> # Update the method object at index 0.
>>> sim.methods[0] = BlochDecaySpectrum(
...     channels=["1H"], # <------------ updated entry
...     rotor_frequency=1000, # in Hz.
...     rotor_angle=90*3.1415926/180, # 90 degree in radians.
...     spectral_dimensions=[dict(
...         count=2048,
...         spectral_width=25000, # in Hz.
...         label=r"$^1$H resonances",
...     )]
... )
```

```
>>> sim.run()
>>> plot(sim.methods[0].simulation)
```

In Figure 1.7, we see a 90° spinning sideband ^{1}H-spectrum, whose frequency contributions arise from `system_3` because `system_3` is the only spin system with ^{1}H site.

Note, although you are free to assign any channel to the *channels* (page 170) attribute of the BlochDecaySpectrum method, only channels whose isotopes are also a member of the spin systems will produce a spectrum. For example, the following method

```
>>> # Update the method object at index 0.
>>> sim.methods[0] = BlochDecaySpectrum(
...     channels=["23Na"], # <------------ updated entry
...     rotor_frequency=1000, # in Hz.
...     rotor_angle=90*3.1415926/180, # 90 degree in radians.
```

(continues on next page)

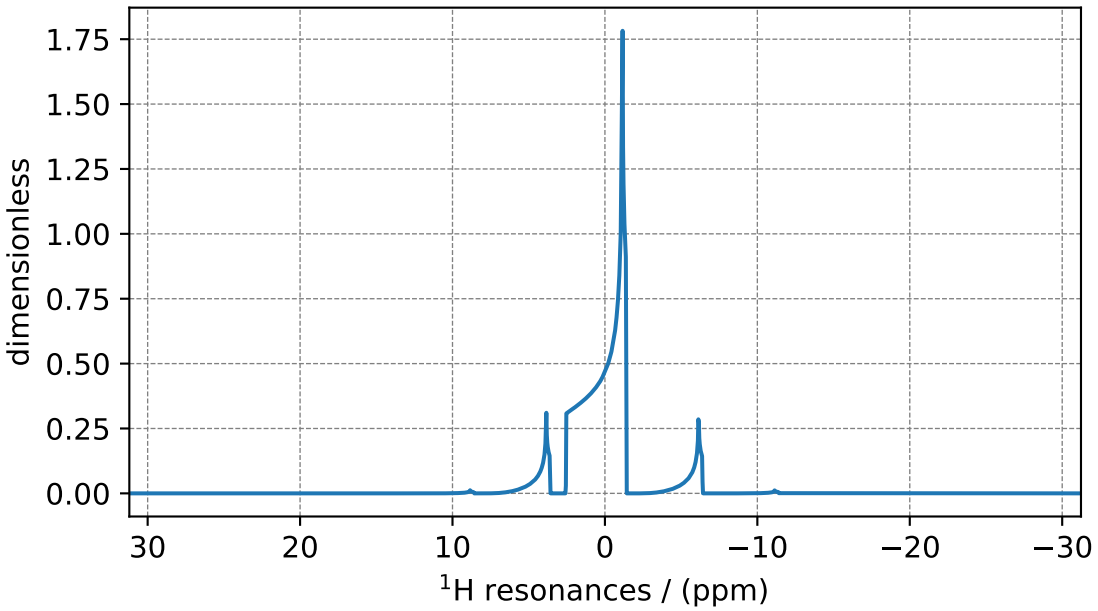


Figure 1.7: An example of solid-state ^{1}H VAS sideband simulation.

(continued from previous page)

```
...     spectral_dimensions=[dict(
...         count=2048,
...         spectral_width=25000, # in Hz.
...         label=r"$^{23}$Na resonances",
...     )]
... )
```

is defined to collect the resonances from ^{23}Na isotope. As you may have noticed, we do not have any ^{23}Na site in the spin systems. Simulating the spectrum from this method will result in a zero amplitude spectrum, see Figure 1.8.

```
>>> sim.run()
>>> plot(sim.methods[0].simulation)
```

Switching the channel to 17O

Likewise, update the value of the *channels* attribute to *17O*.

```
>>> sim.methods[0] = BlochDecaySpectrum(
...     channels=["17O"],
...     rotor_frequency= 15000, # in Hz.
...     rotor_angle = 0.9553166, # magic angle is rad.
...     spectral_dimensions=[dict(
...         count=2048,
...         spectral_width=25000, # in Hz.
...         label=r"$^{17}$O resonances",
...     )]
... )
```

(continues on next page)

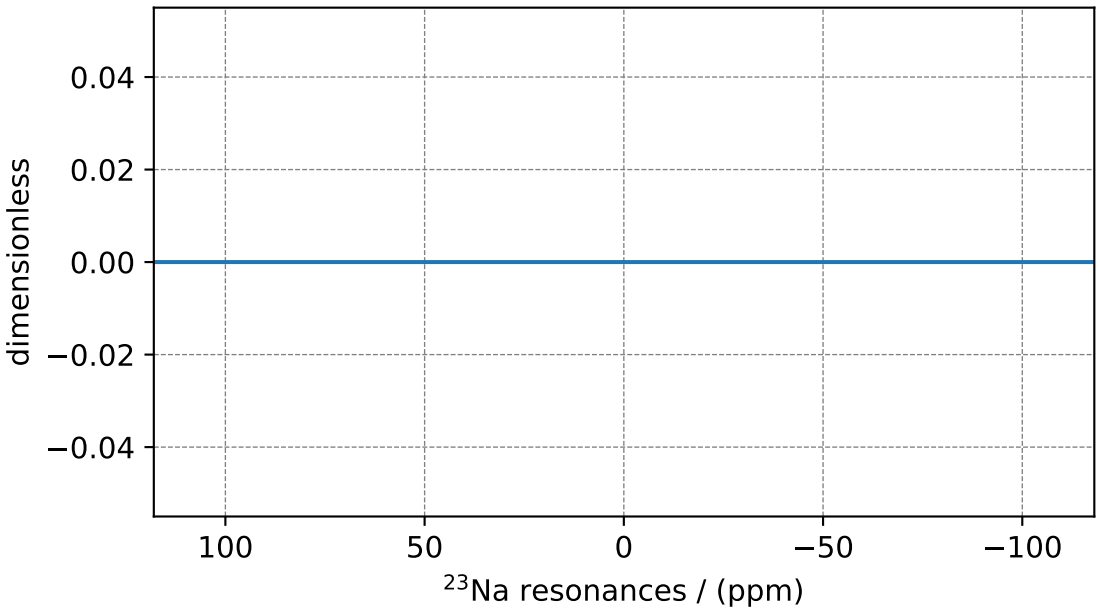


Figure 1.8: An example of a simulation where the isotope from the methods channel attribute does not exist within the spin systems.

(continued from previous page)

```
>>> sim.run()
>>> plot(sim.methods[0].simulation)
```

If you are familiar with the quadrupolar line-shapes, you may immediately associate the spectrum in Figure 1.9 to a second-order quadrupolar line-shape of the central transition. You may also notice some unexpected resonances around 50 ppm and -220 ppm. These unexpected resonances are the spinning sidebands of the satellite transitions. Note, the BlochDecaySpectrum method computes resonances from all transitions with $p = \Delta m = -1$.

Let's see what transition pathways are used in our simulation. Use the *get_transition_pathways()* (page 172) function of the Method instance to see the list of transition pathways, for example,

```
>>> from pprint import pprint
>>> pprint(sim.methods[0].get_transition_pathways(system_4)) # 17O
[TransitionPathway(|-2.5⟩⟨-1.5|),
 TransitionPathway(|-1.5⟩⟨-0.5|),
 TransitionPathway(|-0.5⟩⟨0.5|),
 TransitionPathway(|0.5⟩⟨1.5|),
 TransitionPathway(|1.5⟩⟨2.5|)]
```

Notice, there are five transition pathways for the ^{17}O site, one associated with the central-transition, two with the inner-satellites, and two with the outer-satellites. For central transition selective simulation, use the *BlochDecayCentralTransitionSpectrum* (page 178) method.

```
>>> from mrsimulator.methods import BlochDecayCentralTransitionSpectrum
>>> sim.methods[0] = BlochDecayCentralTransitionSpectrum(
...     channels=["17O"],
...     rotor_frequency= 15000, # in Hz.
```

(continues on next page)

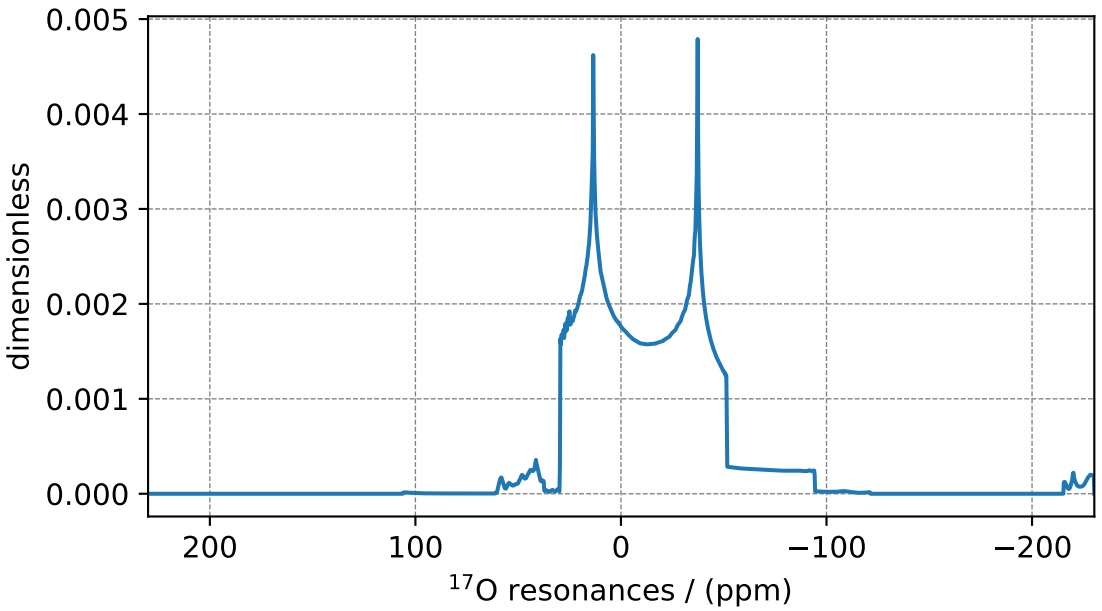


Figure 1.9: An example of the solid-state ^{17}O BlochDecaySpectrum simulation.

(continued from previous page)

```
...     rotor_angle = 0.9553166, # magic angle is rad.
...     spectral_dimensions=[dict(
...         count=2048,
...         spectral_width=25000, # in Hz.
...         label=r"$^{17}$O resonances",
...     )]
... )
>>> # the transition pathways
>>> print(sim.methods[0].get_transition_pathways(system_4)) # 17O
[TransitionPathway(|-0.5⟩⟨0.5|)]
```

Now, you may simulate the central transition selective spectrum. Figure 1.10 depicts a central transition selective spectrum.

```
>>> sim.run()
>>> plot(sim.methods[0].simulation)
```

1.6 Configuring Simulator object

The following code is used to produce the figures in this section.

```
>>> import matplotlib.pyplot as plt
>>> import matplotlib as mpl
>>> mpl.rcParams["figure.figsize"] = (6, 3.5)
>>> mpl.rcParams["font.size"] = 11
...
>>> # function to render figures.
```

(continues on next page)

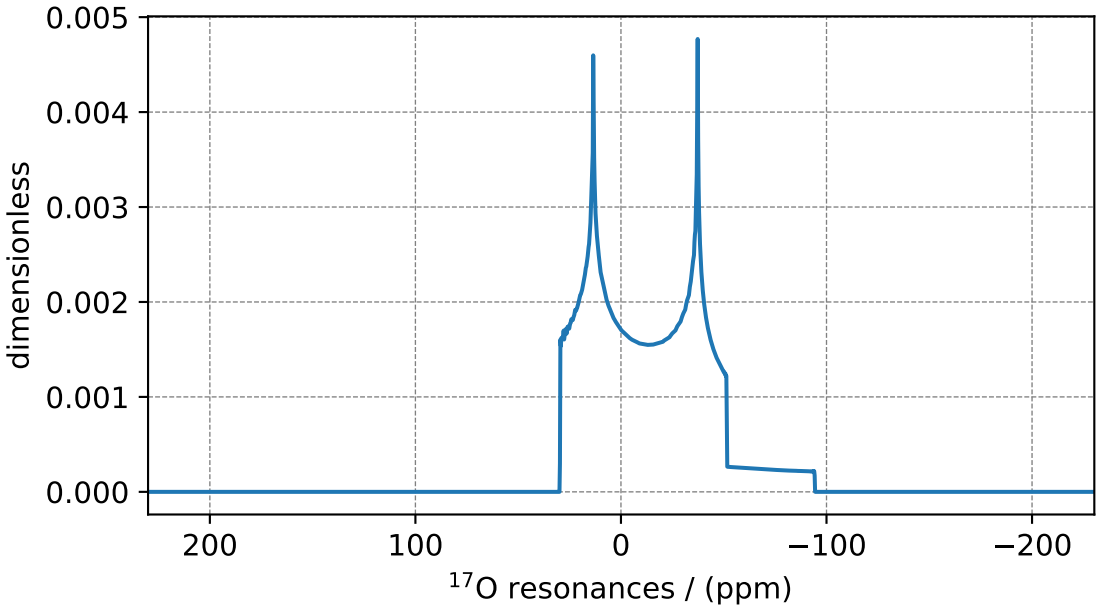


Figure 1.10: An example of the solid-state ^{17}O BlochDecayCentralTransitionSpectrum simulation.

(continued from previous page)

```
>>> def plot(csdm_object):
...     # set matplotlib axes projection='csdm' to directly plot CSDM objects.
...     ax = plt.subplot(projection='csdm')
...     ax.plot(csdm_object, linewidth=1.5)
...     ax.invert_xaxis()
...     plt.tight_layout()
...     plt.show()
```

Up until now, we have been using the simulator object with the default setting. In `mrsimulator`, we choose the default settings such that it applies to a wide range of simulations including, static, magic angle spinning (MAS), and variable angle spinning (VAS) spectra. In certain situations, however, the default settings are not sufficient to accurately represent the spectrum. In such cases, the user is advised to modify these settings as required. In the following section, we briefly describe the configuration settings.

The *Simulator* (page 151) class is configured using the *config* (page 152) attribute. The default value of the config attributes is as follows,

```
>>> from mrsimulator import Simulator, SpinSystem, Site
>>> from mrsimulator.methods import BlochDecaySpectrum
...
>>> sim = Simulator()
>>> sim.config
ConfigSimulator(number_of_sidebands=64, integration_volume='octant', integration_density=70,␣
↪decompose_spectrum='none')
```

Here, the configurable attributes are `number_of_sidebands`, `integration_volume`, `integration_density`, and `decompose_spectrum`.

1.6.1 Number of sidebands

The value of this attribute is the number of sidebands requested in evaluating the spectrum. The default value is 64 and is sufficient for most cases, as seen from our previous examples. In certain circumstances, especially when the anisotropy is large or the rotor spin frequency is low, 64 sidebands might not be sufficient. In such cases, the user will need to increase the value of this attribute as required. Conversely, 64 sidebands might be redundant for other problems, in which case the user may want to reduce the value of this attribute. Note, reducing the number of sidebands will significantly improve computation performance, which might save computation time when used in iterative algorithms, such as least-squares minimization.

The following is an example of when the number of sidebands is insufficient.

```
>>> sim = Simulator()
...
>>> # create a site with a large anisotropy, 100 ppm.
>>> Si29site = Site(isotope='29Si', shielding_symmetric={'zeta': 100, 'eta': 0.2})
...
>>> # create a method. Set a low rotor frequency, 200 Hz.
>>> method = BlochDecaySpectrum(
...     channels=['29Si'],
...     rotor_frequency=200, # in Hz.
...     spectral_dimensions=[dict(count=1024, spectral_width=25000)]
... )
...
>>> sim.spin_systems += [SpinSystem(sites=[Si29site])]
>>> sim.methods += [method]
...
>>> # simulate and plot
>>> sim.run()
>>> plot(sim.methods[0].simulation)
```

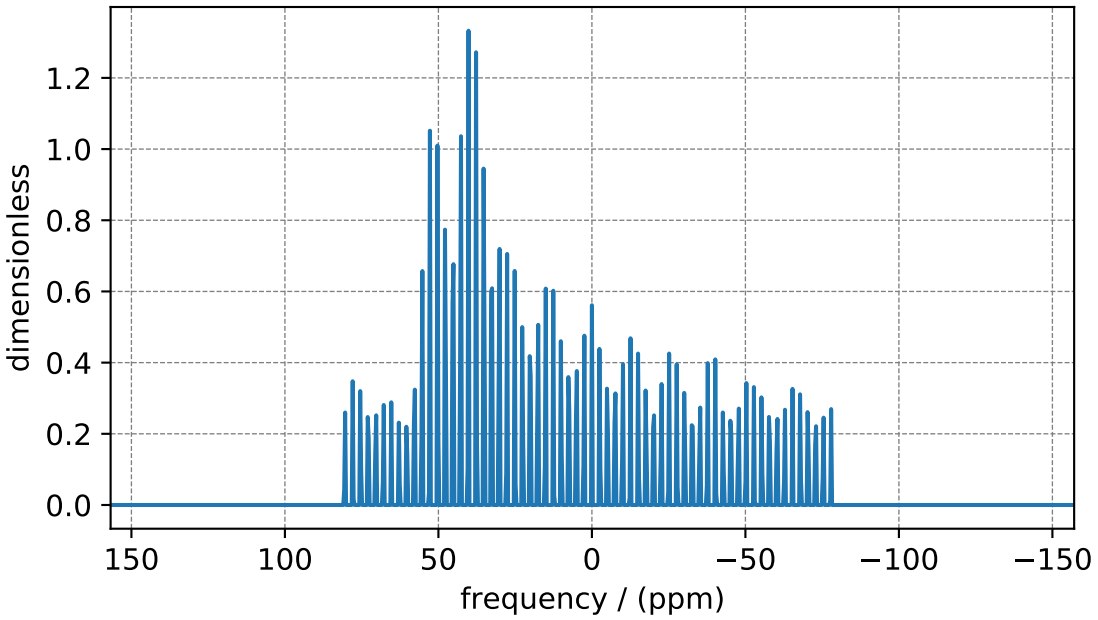


Figure 1.11: Inaccurate spinning sidebands simulation resulting from computing a relatively low number of sidebands.

If you are familiar with the NMR spinning sideband patterns, you may notice that the sideband simulation spectrum in Figure 1.11 is

inaccurate, as evident from the abrupt termination of the sideband amplitudes at the edges. As mentioned earlier, this inaccuracy arises from evaluating a small number of sidebands relative to the given anisotropy. Let's increase the number of sidebands to 90 and observe. Figure 1.12 depicts an accurate spinning sideband simulation.

```
>>> # set the number of sidebands to 90.
>>> sim.config.number_of_sidebands = 90
>>> sim.run()
>>> plot(sim.methods[0].simulation)
```

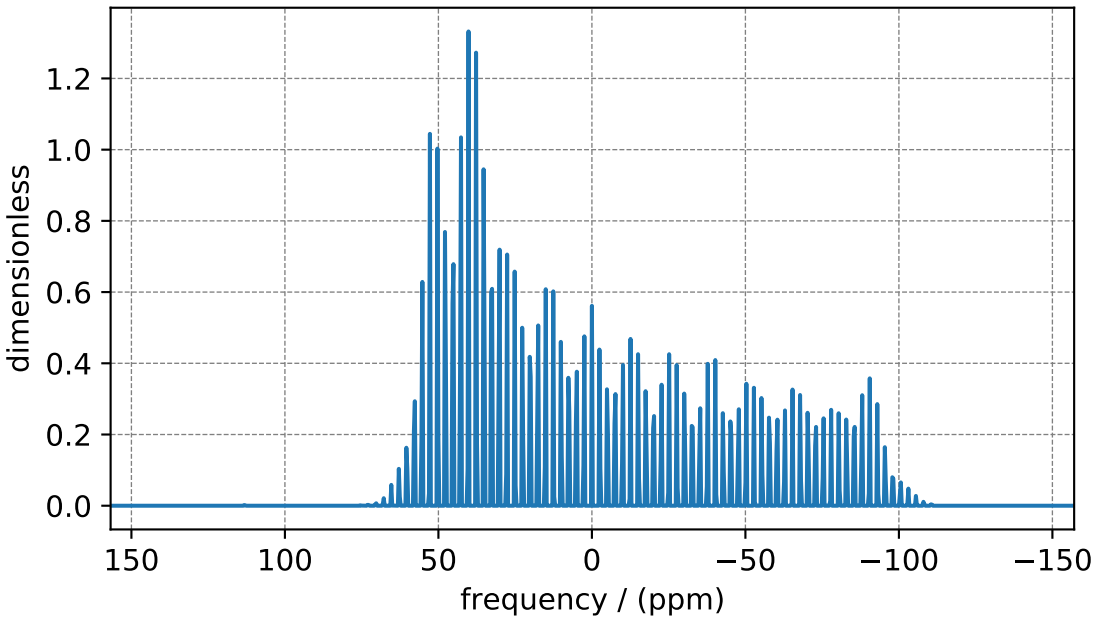


Figure 1.12: Accurate spinning sideband simulation when using a large number of sidebands.

1.6.2 Integration volume

The attribute *integration_volume* is an enumeration with two literals, `octant` and `hemisphere`. The integration volume refers to the volume of the sphere over which the NMR frequencies are integrated. The default value is *octant*, i.e., the spectrum comprises of integrated frequencies arising from the positive octant of the sphere. The `mrsimulator` package enables the user to exploit the orientational symmetry of the problem, and thus optimize the simulation by performing a partial integration —*octant* or *hemisphere*. To learn more about the orientational symmetries, please refer to Eden et. al.[1]

Consider the ^{29}Si site, `Si29site`, from the previous example. This site has a symmetric shielding tensor with *zeta* and *eta* as 100 ppm and 0.2, respectively. With only *zeta* and *eta*, we can exploit the symmetry of the problem, and evaluate the frequency integral over the octant, which is equivalent to the integration over the sphere. By adding the Euler angles to this tensor, we break the symmetry, and the integration over the octant is no longer accurate. Consider the following examples.

```
>>> # add Euler angles to the shielding tensor.
>>> Si29site.shielding_symmetric.alpha = 1.563 # in rad
>>> Si29site.shielding_symmetric.beta = 1.2131 # in rad
```

(continues on next page)

[1] Edén, M. and Levitt, M. H. Computation of orientational averages in solid-state nmr by gaussian spherical quadrature. J. Mag. Res., 132, 2, 220239, 1998. doi:10.1006/jmre.1998.1427.

(continued from previous page)

```
>>> Si29site.shielding_symmetric.gamma = 2.132 # in rad
...
>>> # Let's observe the static spectrum which is more intuitive.
>>> sim.methods[0] = BlochDecaySpectrum(
...     channels=['29Si'],
...     rotor_frequency=0, # in Hz.
...     spectral_dimensions=[dict(count=1024, spectral_width=25000)]
... )
...
>>> # simulate and plot
>>> sim.run()
>>> plot(sim.methods[0].simulation)
```

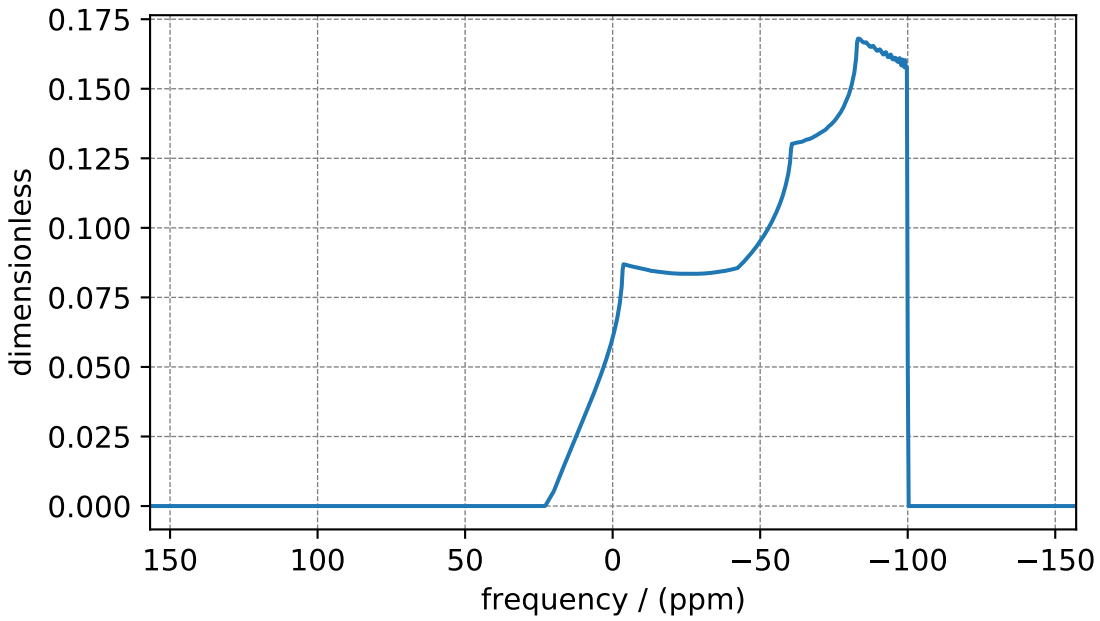


Figure 1.13: An example of an incomplete spectral averaging, where the simulation comprises of frequency contributions evaluated over the positive octant.

The spectrum in Figure 1.13 is incorrect. To fix this, set the integration volume to *hemisphere* and re-simulate. Figure 1.14 depicts the accurate simulation of the CSA tensor.

```
>>> # set integration volume to `hemisphere`.
>>> sim.config.integration_volume = 'hemisphere'
...
>>> # simulate and plot
>>> sim.run()
>>> plot(sim.methods[0].simulation)
```

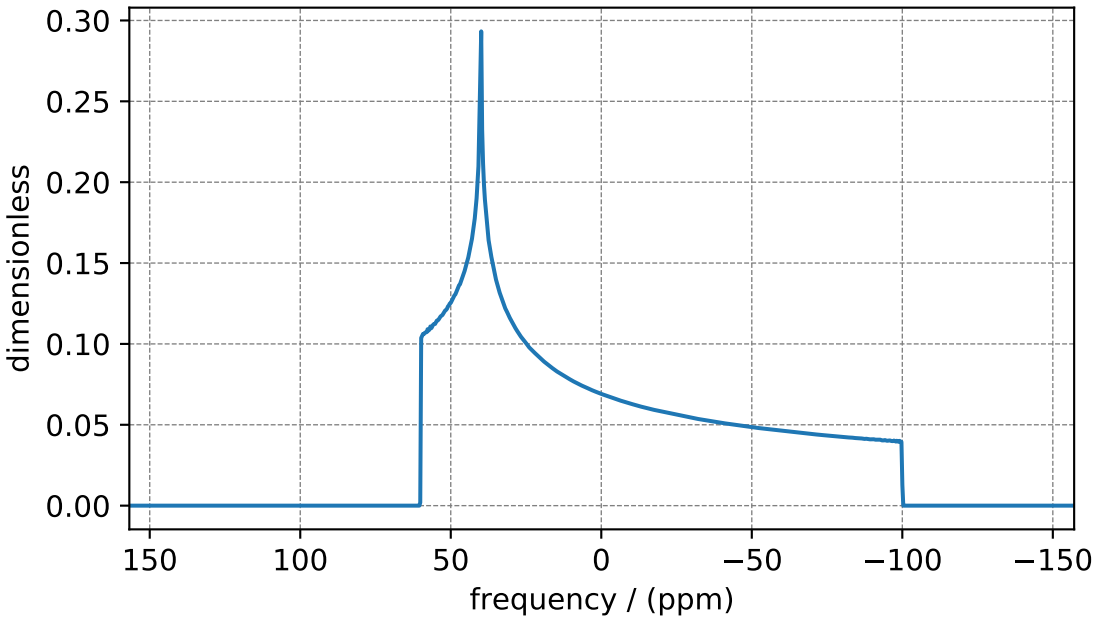


Figure 1.14: The spectrum resulting from the frequency contributions evaluated over the top hemisphere.

1.6.3 Integration density

Integration density controls the number of orientational points sampled over the given volume. The resulting spectrum is an integration of the NMR resonance frequency evaluated at these orientations. The total number of orientations, Θ_{count}, is given as

$$\Theta_{\text{count}} = M(n+1)(n+2)/2, \tag{1.2}$$

where M is the number of octants and n is value of this attribute. The number of octants is deciphered form the value of the *integration_volume* attribute. The default value of this attribute, 70, produces 2556 orientations at which the NMR frequency contribution is evaluated. The user may increase or decrease the value of this attribute as required by the problem.

Consider the following example.

```
>>> sim = Simulator()
>>> sim.config.integration_density
70
>>> sim.config.get_orientations_count() # 1 * 71 * 72 / 2
2556
>>> sim.config.integration_density = 100
>>> sim.config.get_orientations_count() # 1 * 101 * 102 / 2
5151
```

1.6.4 Decompose spectrum

The attribute *decompose_spectrum* is an enumeration with two literals, `none`, and `spin_system`. The value of this attribute lets us know how the user intends the simulation to be stored.

none

If the value is `none` (default), the result of the simulation is a single spectrum where the frequency contributions from all the spin systems are co-added. Consider the following example.

```
>>> # Create two sites
>>> site_A = Site(isotope='1H', shielding_symmetric={'zeta': 5, 'eta': 0.1})
>>> site_B = Site(isotope='1H', shielding_symmetric={'zeta': -2, 'eta': 0.83})
...
>>> # Create two spin systems, each with single site.
>>> system_A = SpinSystem(sites=[site_A], name='System-A')
>>> system_B = SpinSystem(sites=[site_B], name='System-B')
...
>>> # Create a method object.
>>> method = BlochDecaySpectrum(
...     channels=['1H'],
...     spectral_dimensions=[dict(count=1024, spectral_width=10000)]
... )
...
>>> # Create simulator object.
>>> sim = Simulator()
>>> sim.spin_systems += [system_A,  system_B] # add the spin systems
>>> sim.methods += [method] # add the method
...
>>> # simulate and plot.
>>> sim.run()
>>> plot(sim.methods[0].simulation)
```

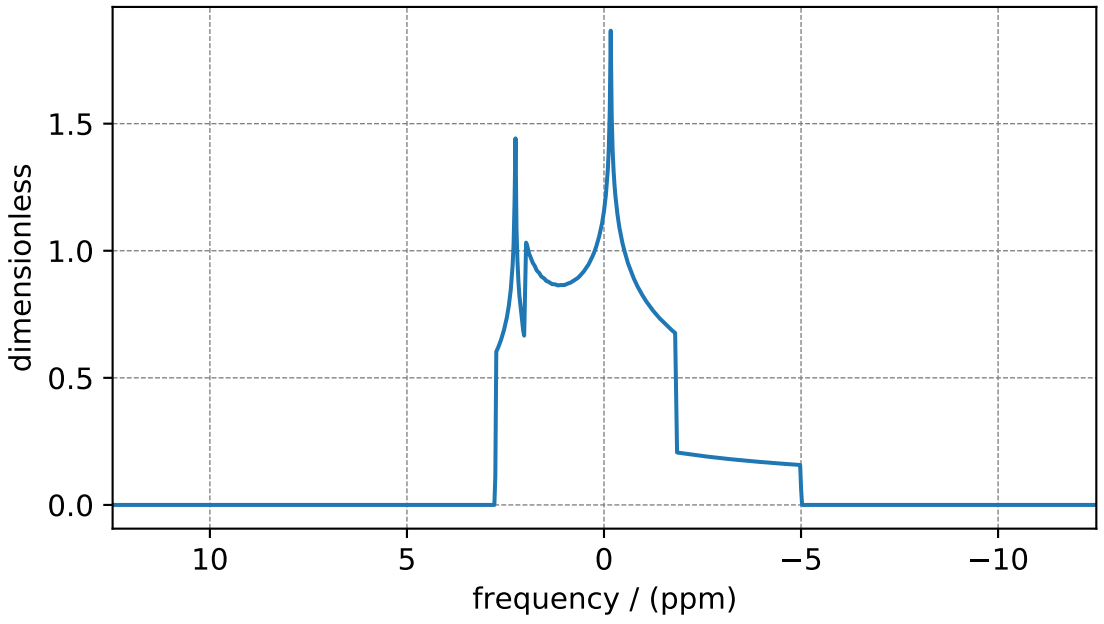


Figure 1.15: The spectrum is an integration of the spectra from individual spin systems when the value of decompose_spectrum is none.

Figure 1.15 depicts the simulation of the spectrum from two spin systems where the contributions from individual spin systems are co-added.

spin_system

When the value of this attribute is spin_system, the resulting simulation is a series of spectra, each arising from a spin system. In this case, the number of spectra is the same as the number of spin system objects. Try setting the value of the decompose_spectrum attribute to *spin_system* and observe the simulation.

```
>>> # set decompose_spectrum to true.
>>> sim.config.decompose_spectrum = "spin_system"
...
>>> # simulate.
>>> sim.run()
...
>>> # plot of the two spectrum
>>> plot(sim.methods[0].simulation)
```

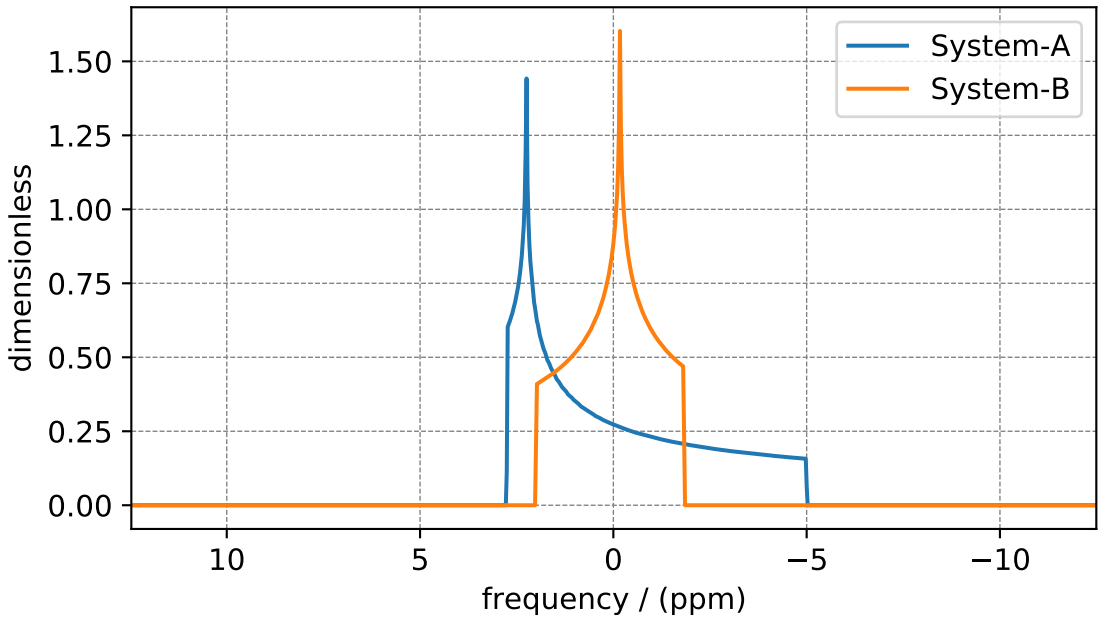


Figure 1.16: Spectrum from individual spin systems when the value of the decompose_spectrum config is spin_system.

1.7 mrsimulator I/O

1.7.1 Simulator object

Export simulator object to a JSON file

To serialize a *Simulator* (page 151) object to a JSON-compliant file, use the *save()* (page 156) method of the object.

```
>>> sim_coesite.save('sample.mrsim')
```

where sim_coesite is a *Simulator* (page 151) object. By default, the attribute values are serialized as physical quantities, represented as a string with a value and a unit.

Load simulator object from a JSON file

To load a JSON-compliant *Simulator* (page 151) serialized file, use the *load()* (page 156) method of the class. By default, the load method parses the file for units.

```
>>> from mrsimulator import Simulator
>>> sim_load = Simulator.load('sample.mrsim')
>>> sim_coesite == sim_load
True
```

1.7.2 Spin systems objects from Simulator class

Export spin systems to a JSON file

You may also serialize the spin system objects from the *Simulator* (page 151) object to a JSON-compliant file using the *export_spin_systems()* (page 155) method as

```
>>> sim_coesite.export_spin_systems('coesite_spin_systems.mrsys')
```

Import spin systems from a JSON file

Similarly, a list of spin systems can be directly imported from a JSON serialized file. To import the spin systems, use the *load_spin_systems()* (page 155) method of the *Simulator* (page 151) class as

```
>>> sim.load_spin_systems('coesite_spin_systems.mrsys')
```

Importing spin systems from URL

```
>>> from mrsimulator import Simulator
>>> sim = Simulator()
>>> filename = 'https://raw.githubusercontent.com/DeepanshS/mrsimulator-examples/master/spin_
↪systems.json'
>>> sim.load_spin_systems(filename)
>>> # The seven spin systems from the file are added to the sim object.
>>> len(sim.spin_systems)
7
```

CHAPTER

TWO

SIGNAL PROCESSING (`mrsimulator.signalprocessor`)

2.1 Signal Processing

2.1.1 Introduction

After running a simulation, you may need to apply some post-simulation signal processing. For example, you may want to scale the intensities to match the experiment or convolve the spectrum with a Lorentzian, Gaussian, or sinc line-broadening functions. There are many signal-processing libraries, such as Numpy and Scipy, that you may use to accomplish this. Although, in NMR, certain operations like convolutions, Fourier transform, and apodizations are so regularly used that it soon becomes inconvenient to have to write your own set of code. For this reason, the `mrsimulator` package offers some frequently used NMR signal processing tools.

Note: The simulation object in *mrsimulator* is a CSDM object. A CSDM object is the python support for the core scientific dataset model (CSDM)[1], which is a new open-source universal file format for multi-dimensional datasets. Since CSDM objects hold a generic multi-dimensional scientific dataset, the following signal processing operation can be applied to any CSDM dataset, *i.e.*, NMR, EPR, FTIR, GC, etc.

In the following section, we demonstrate the use of the *SignalProcessor* (page 189) class in applying various operations to a generic CSDM object. But before we start explaining signal processing with CSDM objects, it seems necessary to first describe the construct of CSDM objects. Each CSDM object has two main attributes, *dimensions* and *dependent_variables*. The *dimensions* attribute holds a list of Dimension objects, which collectively form a multi-dimensional Cartesian coordinates grid system. A Dimension object can represent both physical and non-physical dimensions. The *dependent_variables* attribute holds the responses of the multi-dimensional grid points. You may have as many dependent variables as you like, as long as all dependent variables share the same coordinates grid, *i.e.*, dimensions.

2.1.2 `SignalProcessor` class

Signal processing is a series of operations that are applied to the dataset. In this workflow, the result from the previous operation becomes the input for the next operation.

In the `mrsimulator` library, all signal processing operations are accessed through the *signal_processing* module. Within the module is the *apodization* sub-module. An apodization is a point-wise multiplication operation of the input signal with the apodizing vector. See *Operations* (page 190) documentation for a complete list of operations.

Import the module and sub-module as

```
>>> import mrsimulator.signal_processing as sp
>>> import mrsimulator.signal_processing.apodization as apo
```

[1] Srivastava, D. J., Vosegaard, T., Massiot, D., Grandinetti, P. J., Core Scientific Dataset Model: A lightweight and portable model and file format for multi-dimensional scientific data, PLOS ONE, 15, 1-38, (2020). DOI:10.1371/journal.pone.0225953

2.1.3 Convolution

The convolution theorem states that under suitable conditions, the Fourier transform of a convolution of two signals is the pointwise product (apodization) of their Fourier transforms. In the following example, we employ this theorem to demonstrate how to apply a Gaussian convoluting to a dataset.

```
>>> processor = sp.SignalProcessor(
...     operations=[
...         sp.IFFT(), apo.Gaussian(FWHM='0.1 km'), sp.FFT()
...     ]
... )
```

Here, the *processor* is an instance of the *SignalProcessor* (page 189) class. The required attribute of this class, *operations*, is a list of operations. In the above example, we employ the convolution theorem by sandwiching the Gaussian apodization function between two Fourier transformations.

In this scheme, first, an inverse Fourier transform is applied to the datasets. On the resulting dataset, a Gaussian apodization, equivalent to a full width at half maximum of 0.1 km in the reciprocal dimension, is applied. The unit that you use for the FWHM attribute depends on the dimensionality of the dataset dimension. By choosing the unit as km, we imply that the corresponding dimension of the CSDM object undergoing the above series of operations has a dimensionality of length. Finally, a forward Fourier transform is applied to the apodized dataset.

Let's create a CSDM object and then apply the above signal processing operations.

```
>>> import csdmpy as cp
>>> import numpy as np
...
>>> # Creating a test CSDM object.
>>> test_data = np.zeros(500)
>>> test_data[250] = 1
>>> csdm_object = cp.CSDM(
...     dependent_variables=[cp.as_dependent_variable(test_data)],
...     dimensions=[cp.LinearDimension(count=500, increment='1 m')]
... )
```

Note: See csdmpy for a detailed description of generating CSDM objects. In `mrsimulator`, the simulation data is already stored as a CSDM object.

To apply the previously defined signal processing operations to the above CSDM object, use the *apply_operations()* (page 190) method of the `SignalProcessor` instance as follows,

```
>>> processed_data = processor.apply_operations(data=csdm_object)
```

The *data* is the required argument of the *apply_operations* method, whose value is a CSDM object holding the dataset. The variable *processed_data* holds the output, that is, the processed data as a CSDM object. The plot of the original and the processed data is shown below.

```
>>> import matplotlib.pyplot as plt
>>> _, ax = plt.subplots(1, 2, figsize=(8, 3), subplot_kw={"projection":"csdm"})
>>> ax[0].plot(csdm_object, color="black", linewidth=1)
>>> ax[0].set_title('Before')
>>> ax[1].plot(processed_data.real, color="black", linewidth=1)
>>> ax[1].set_title('After')
>>> plt.tight_layout()
>>> plt.show()
```

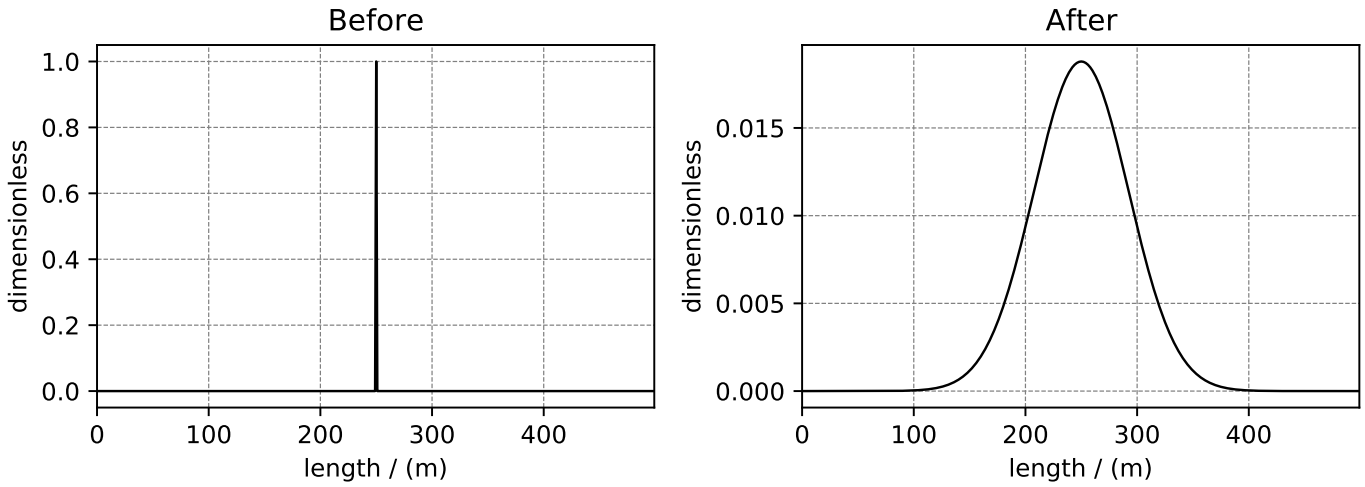


Figure 2.1: The figure depicts an application of Gaussian convolution on a CSDM object.

Multiple convolutions

As mentioned before, a CSDM object may hold multiple dependent variables. When using the list of the operations, you may selectively apply a given operation to a specific dependent-variable by specifying the index of the corresponding dependent-variable as an argument to the operation class. Consider the following list of operations.

```
>>> processor = sp.SignalProcessor(
...     operations=[
...         sp.IFFT(),
...         apo.Gaussian(FWHM='0.1 km', dv_index=0),
...         apo.Exponential(FWHM='50 m', dv_index=1),
...         sp.FFT(),
...     ]
... )
```

The above signal processing operations first applies an inverse Fourier transform, followed by a Gaussian apodization on the dependent variable at index 0, followed by an Exponential apodization on the dependent variable at index 1, and finally a forward Fourier transform. Note, the FFT and IFFT operations apply on all dependent-variables.

Let's add another dependent variable to the previously created CSDM object.

```
>>> # Add a dependent variable to the test CSDM object.
>>> test_data = np.zeros(500)
>>> test_data[150] = 1
>>> csdm_object.add_dependent_variable(cp.as_dependent_variable(test_data))
```

As before, apply the operations with the *apply_operations()* (page 190) method.

```
>>> processed_data = processor.apply_operations(data=csdm_object)
```

The plot of the dataset before and after signal processing is shown below.

```
>>> _, ax = plt.subplots(1, 2, figsize=(8, 3), subplot_kw={"projection":"csdm"})
>>> ax[0].plot(csdm_object, linewidth=1)
>>> ax[0].set_title('Before')
```

(continues on next page)

(continued from previous page)

```
>>> ax[1].plot(processed_data.real, linewidth=1)
>>> ax[1].set_title('After')
>>> plt.tight_layout()
>>> plt.show()
```

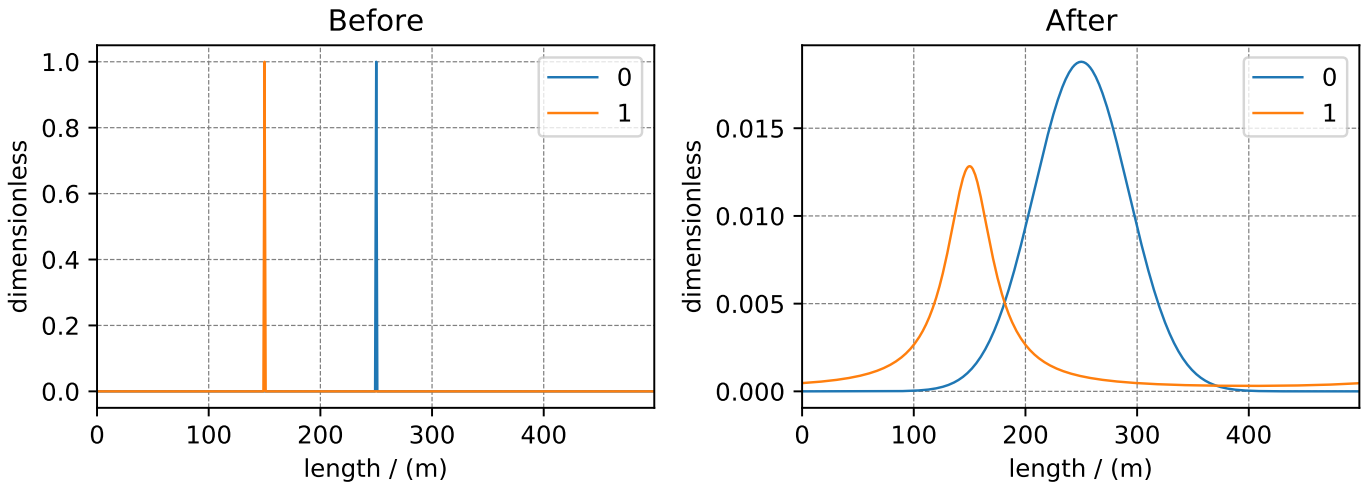


Figure 2.2: Gaussian and Lorentzian convolution applied to two different dependent variables of the CSDM object.

Convolution along multiple dimensions

In the case of multi-dimensional datasets, besides the dependent-variable index, you may also specify a dimension index along which a particular operation will apply. For example, consider the following 2D datasets as a CSDM object,

```
>>> # Create a two-dimensional CSDM object.
>>> test_data = np.zeros(600).reshape(30, 20)
>>> test_data[15, 10] = 1
>>> dv = cp.as_dependent_variable(test_data)
>>> dim1 = cp.LinearDimension(count=20, increment='0.1 ms', coordinates_offset='-1 ms', label='t1
↪')
>>> dim2 = cp.LinearDimension(count=30, increment='1 cm/s', label='s1')
>>> csdm_data = cp.CSDM(dependent_variables=[dv], dimensions=[dim1, dim2])
```

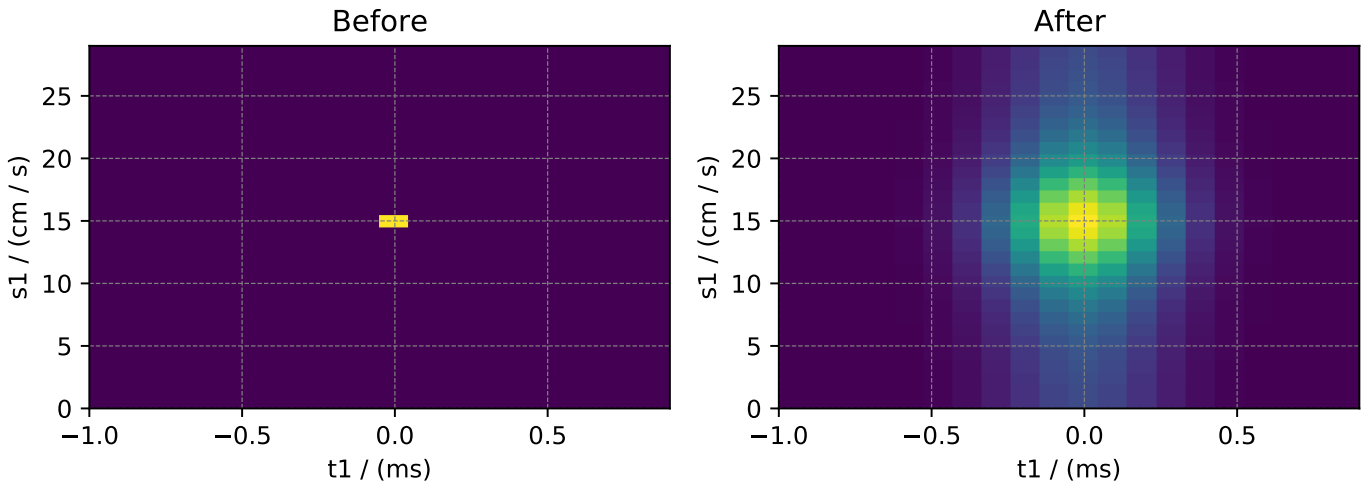
where `csdm_data` is a two-dimensional dataset. Now consider the following signal processing operations

```
>>> processor = sp.SignalProcessor(
...     operations=[
...         sp.IFFT(dim_index=(0, 1)),
...         apo.Gaussian(FWHM='0.5 ms', dim_index=0),
...         apo.Exponential(FWHM='10 cm/s', dim_index=1),
...         sp.FFT(dim_index=(0, 1)),
...     ]
... )
>>> processed_data = processor.apply_operations(data=csdm_data)
```

The above set of operations first performs an inverse FFT on the dataset along the dimension index 0 and 1. The second and third

operations apply a Gaussian and Lorentzian apodization along dimensions 0 and 1, respectively. The last operation is a forward Fourier transform. The before and after plots of the datasets are shown below.

```
>>> _, ax = plt.subplots(1, 2, figsize=(8, 3), subplot_kw={"projection":"csdm"})
>>> ax[0].imshow(csdm_data, aspect='auto')
>>> ax[0].set_title('Before')
>>> ax[1].imshow(processed_data.real, aspect='auto')
>>> ax[1].set_title('After')
>>> plt.tight_layout()
>>> plt.show()
```



2.1.4 Serializing the operations list

You may also serialize the operations list using the *json()* (page 190) method, as follows

```
>>> from pprint import pprint
>>> pprint(processor.json())
{'operations': [{'dim_index': [0, 1], 'function': 'IFFT'},
                {'FWHM': '0.5 ms',
                 'dim_index': 0,
                 'function': 'apodization',
                 'type': 'Gaussian'},
                {'FWHM': '10.0 cm / s',
                 'dim_index': 1,
                 'function': 'apodization',
                 'type': 'Exponential'},
                {'dim_index': [0, 1], 'function': 'FFT'}]}
```

See also:

Simulation Examples (page 45) for application of signal processing on NMR simulations.

CHAPTER

THREE

MODELS

3.1 Czjzek distribution

A Czjzek distribution model is a random distribution of the second-rank traceless symmetric tensors about a zero tensor. See *Czjzek distribution* (page 147) and references within for a brief description of the model.

3.1.1 Czjzek distribution of symmetric shielding tensors

To generate a Czjzek distribution, use the *CzjzekDistribution* (page 192) class as follows.

```
>>> from mrsimulator.models import CzjzekDistribution
>>> cz_model = CzjzekDistribution(sigma=0.8)
```

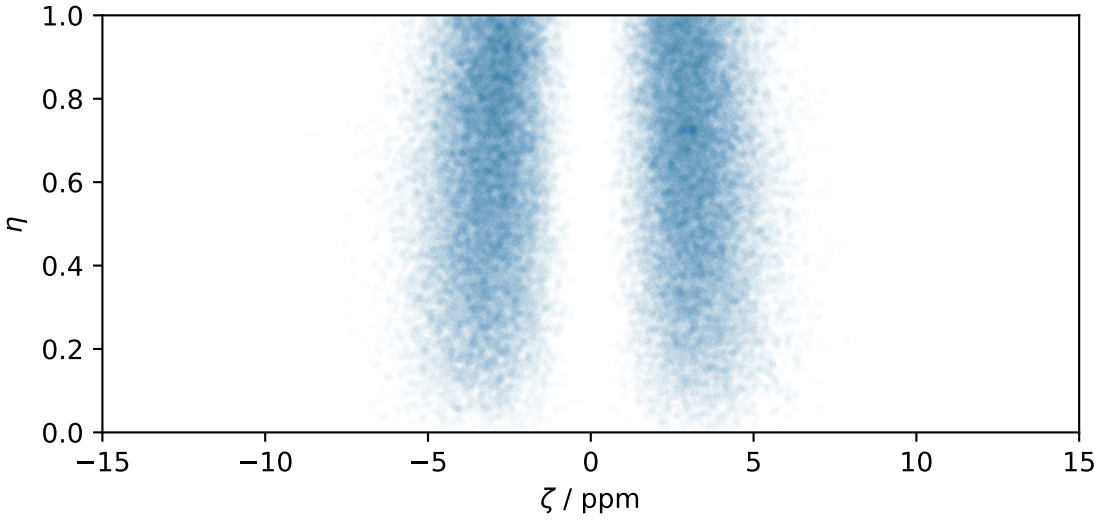
The *CzjzekDistribution* class accepts a single argument, *sigma*, which is the standard deviation of the second-rank traceless symmetric tensor parameters. In the above example, we create `cz_model` as an instance of the CzjzekDistribution class with $\sigma = 0.8$.

Note, *cz_model* is only a class instance of the Czjzek distribution. You can either draw random points from this distribution or generate a probability distribution function. Let's first draw points from this distribution, using the *rvs()* (page 192) method of the instance.

```
>>> zeta_dist, eta_dist = cz_model.rvs(size=50000)
```

In the above example, we draw *size=50000* random points of the distribution. The output `zeta_dist` and `eta_dist` hold the tensor parameter coordinates of the points, defined in the Haeberlen convention. The scatter plot of these coordinates is shown below.

```
>>> import matplotlib.pyplot as plt
>>> plt.scatter(zeta_dist, eta_dist, s=4, alpha=0.02)
>>> plt.xlabel('$\zeta$ / ppm')
>>> plt.ylabel('$\eta$')
>>> plt.xlim(-15, 15)
>>> plt.ylim(0, 1)
>>> plt.tight_layout()
>>> plt.show()
```



3.1.2 Czjzek distribution of symmetric quadrupolar tensors

The Czjzek distribution of symmetric quadrupolar tensors follows a similar setup as the Czjzek distribution of symmetric shielding tensors, except we assign the outputs to Cq and η_q. In the following example, we generate the probability distribution function using the *pdf()* (page 193) method.

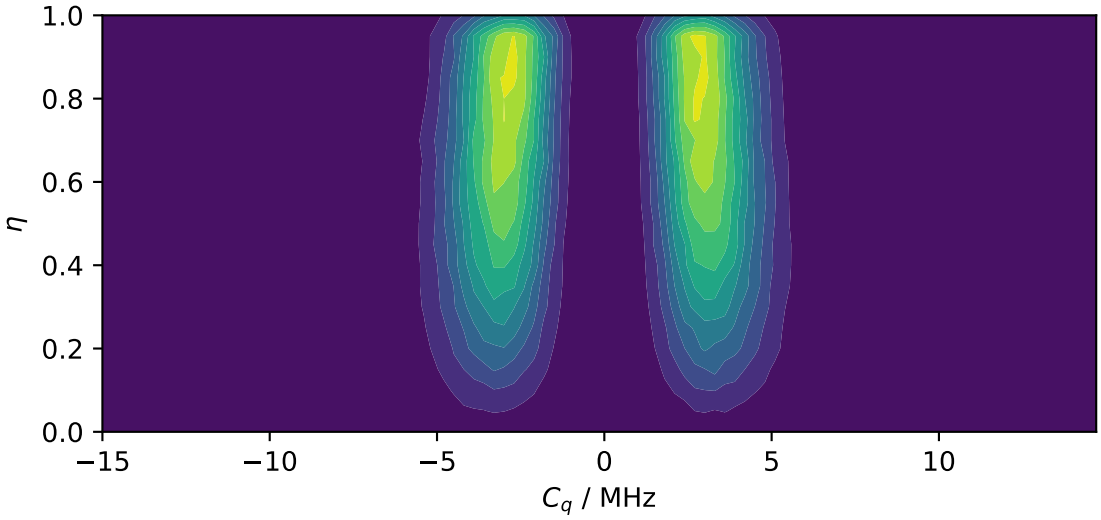
```
>>> import numpy as np
>>> Cq_range = np.arange(100)*0.3 - 15 # pre-defined Cq range in MHz.
>>> eta_range = np.arange(21)/20  # pre-defined eta range.
>>> Cq, eta, amp = cz_model.pdf(pos=[Cq_range, eta_range])
```

To generate a probability distribution, we need to define a grid system over which the distribution probabilities will be evaluated. We do so by defining the range of coordinates along the two dimensions. In the above example, `Cq_range` and `eta_range` are the range of Cq and η_q coordinates, which is then given as the argument to the *pdf()* (page 193) method. The output `Cq`, `eta`, and `amp` hold the two coordinates and amplitude, respectively.

The plot of the Czjzek probability distribution is shown below.

```
>>> import matplotlib.pyplot as plt
>>> plt.contourf(Cq, eta, amp, levels=10)
>>> plt.xlabel('$C_q$ / MHz')
>>> plt.ylabel('$\eta$')
>>> plt.tight_layout()
>>> plt.show()
```

Note: The `pdf` method of the instance generates the probability distribution function by first drawing random points from the distribution and then binning it onto a pre-defined grid.



3.1.3 Mini-gallery using the Czjzek distributions

- *Czjzek distribution (Shielding and Quadrupolar)* (page 69)
- *Czjzek distribution, 27Al (I=5/2) 3QMAS* (page 108)

3.2 Extended Czjzek distribution

An Extended Czjzek distribution model is a random perturbation of the second-rank traceless symmetric tensors about a non-zero tensor. See *Extended Czjzek distribution* (page 149) and references within for a brief description of the model.

3.2.1 Extended Czjzek distribution of symmetric shielding tensors

To generate an extended Czjzek distribution, use the *ExtCzjzekDistribution* (page 193) class as follows.

```
>>> from mrsimulator.models import ExtCzjzekDistribution
>>> shielding_tensor = {'zeta': 80, 'eta': 0.4}
>>> shielding_model = ExtCzjzekDistribution(shielding_tensor, eps=0.1)
```

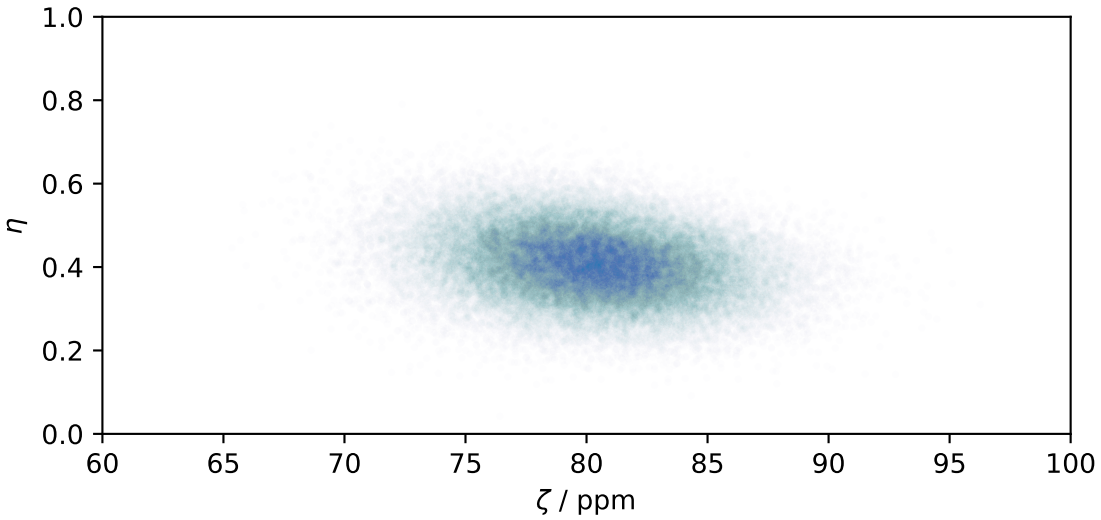
The *ExtCzjzekDistribution* class accepts two arguments. The first argument is the dominant tensor about which the perturbation applies, and the second parameter, *eps*, is the perturbation fraction. The minimum value of the *eps* parameter is 0, which means the distribution is the same as the dominant tensor. As the value of this parameter increases, the distribution gets broader. At values greater than 1, the extended Czjzek distribution approaches a Czjzek distribution. In the above example, we create an extended Czjzek distribution about a second-rank traceless symmetric shielding tensor described by anisotropy of 80 ppm and an asymmetry parameter of 0.4. The perturbation fraction is 0.1.

As before, you may either draw random samples from this distribution or generate a probability distribution function. Let's first draw points from this distribution, using the *rvs()* (page 194) method of the instance.

```
>>> zeta_dist, eta_dist = shielding_model.rvs(size=50000)
```

In the above example, we draw *size=50000* random points of the distribution. The output `zeta_dist` and `eta_dist` hold the tensor parameter coordinates of the points, defined in the Haeberlen convention. The scatter plot of these coordinates is shown below.

```
>>> import matplotlib.pyplot as plt
>>> plt.scatter(zeta_dist, eta_dist, s=4, alpha=0.01)
>>> plt.xlabel('$\zeta$ / ppm')
>>> plt.ylabel('$\eta$')
>>> plt.xlim(60, 100)
>>> plt.ylim(0, 1)
>>> plt.tight_layout()
>>> plt.show()
```



3.2.2 Extended Czjzek distribution of symmetric quadrupolar tensors

The extended Czjzek distribution of symmetric quadrupolar tensors follows a similar setup as the extended Czjzek distribution of symmetric shielding tensors, shown above. In the following example, we generate the probability distribution function using the *pdf()* (page 194) method.

```
>>> import numpy as np
>>> Cq_range = np.arange(100)*0.04 + 2 # pre-defined Cq range in MHz.
>>> eta_range = np.arange(21)/20  # pre-defined eta range.
...
>>> quad_tensor = {'Cq': 3.5, 'eta': 0.23} # Cq assumed in MHz
>>> model_quad = ExtCzjzekDistribution(quad_tensor, eps=0.2)
>>> Cq, eta, amp = model_quad.pdf(pos=[Cq_range, eta_range])
```

As with the case with Czjzek distribution, to generate a probability distribution of the extended Czjzek distribution, we need to define a grid system over which the distribution probabilities will be evaluated. We do so by defining the range of coordinates along the two dimensions. In the above example, `Cq_range` and `eta_range` are the range of Cq and η_q coordinates, which is then given as the argument to the *pdf()* (page 194) method. The output `Cq`, `eta`, and `amp` hold the two coordinates and amplitude, respectively.

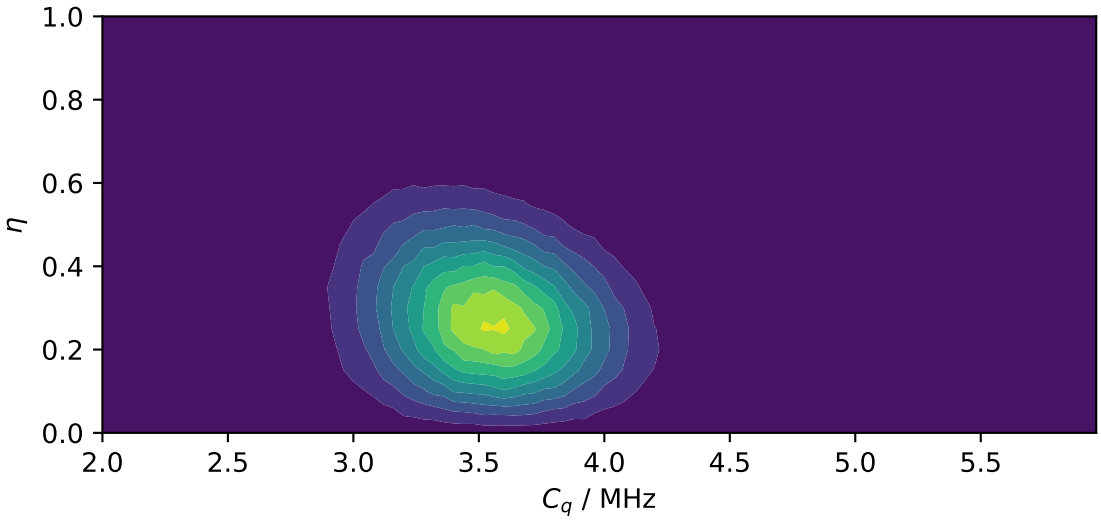
The plot of the extended Czjzek probability distribution is shown below.

```
>>> import matplotlib.pyplot as plt
>>> plt.contourf(Cq, eta, amp, levels=10)
>>> plt.xlabel('$C_q$ / MHz')
```

(continues on next page)

(continued from previous page)

```
>>> plt.ylabel('$\eta$')
>>> plt.tight_layout()
>>> plt.show()
```



Note: The `pdf` method of the instance generates the probability distribution function by first drawing random points from the distribution and then binning it onto a pre-defined grid.

3.2.3 Mini-gallery using the extended Czjzek distributions

- *Extended Czjzek distribution (Shielding and Quadrupolar)* (page 73)
- *Simulating site disorder (crystalline)* (page 105)

CHAPTER

FOUR

EXAMPLES AND BENCHMARKS

4.1 Simulation Examples

In this section, we use the `mrsimulator` tools to create spin systems and simulate spectrum with practical/experimental applications. The examples illustrate

- building spin systems,
- building NMR methods,
- simulating spectrum, and
- processing spectrum (e.g. adding line-broadening).

For mrsimulator applications related to least-squares fitting, see the *Fitting Examples (Least Squares)* (page 111).

4.1.1 1D NMR simulation (crystalline solids)

The following examples are the NMR spectrum simulation of crystalline solids for the following methods:

- Bloch decay method (*BlochDecaySpectrum* (page 177)),
- Central transition selective Bloch decay method (*BlochDecayCentralTransitionSpectrum* (page 178)).
- Generic one-dimensional method (*Method1D* (page 176)).

Wollastonite, 29Si (I=1/2)

29Si (I=1/2) spinning sideband simulation.

Wollastonite is a high-temperature calcium-silicate, β–$Ca_3Si_3O_9$, with three distinct ^{29}Si sites. The ^{29}Si tensor parameters were obtained from Hansen *et. al.*[1]

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import mrsimulator.signal_processing as sp
import mrsimulator.signal_processing.apodization as apo
from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.methods import BlochDecaySpectrum

# global plot configuration
mpl.rcParams["figure.figsize"] = [4.5, 3.0]
```

[1] Hansen, M. R., Jakobsen, H. J., Skibsted, J., ^{29}Si Chemical Shift Anisotropies in Calcium Silicates from High-Field ^{29}Si MAS NMR Spectroscopy, Inorg. Chem. 2003, 42, 7, 2368-2377. DOI: 10.1021/ic020647f

Step 1: Create the sites.

```
S29_1 = Site(
    isotope="29Si",
    isotropic_chemical_shift=-89.0,  # in ppm
    shielding_symmetric={"zeta": 59.8, "eta": 0.62},  # zeta in ppm
)
S29_2 = Site(
    isotope="29Si",
    isotropic_chemical_shift=-89.5,  # in ppm
    shielding_symmetric={"zeta": 52.1, "eta": 0.68},  # zeta in ppm
)
S29_3 = Site(
    isotope="29Si",
    isotropic_chemical_shift=-87.8,  # in ppm
    shielding_symmetric={"zeta": 69.4, "eta": 0.60},  # zeta in ppm
)

sites = [S29_1, S29_2, S29_3]  # all sites
```

Step 2: Create the spin systems from these sites. Again, we create three single-site spin systems for better performance.

```
spin_systems = [SpinSystem(sites=[s]) for s in sites]
```

Step 3: Create a Bloch decay spectrum method.

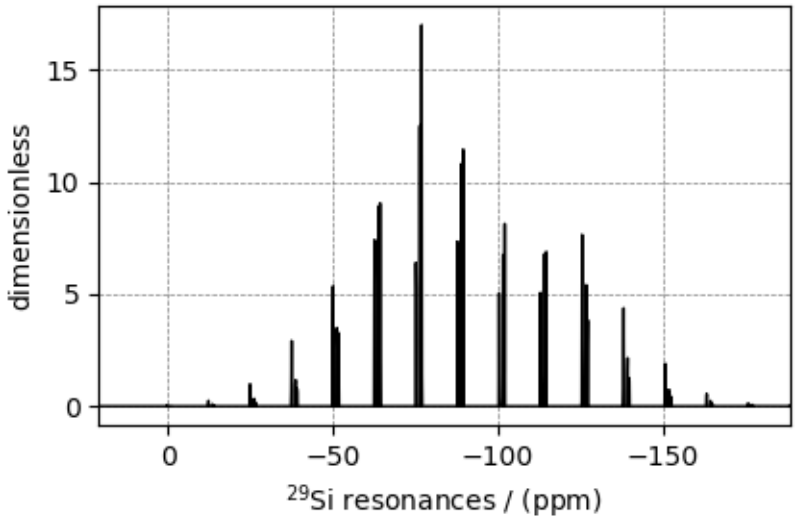
```
method = BlochDecaySpectrum(
    channels=["29Si"],
    magnetic_flux_density=14.1,  # in T
    rotor_frequency=1500,  # in Hz
    spectral_dimensions=[
        {
            "count": 2048,
            "spectral_width": 25000,  # in Hz
            "reference_offset": -10000,  # in Hz
            "label": r"$^{29}$Si resonances",
        }
    ],
)
```

Step 4: Create the Simulator object and add the method and spin system objects.

```
sim = Simulator()
sim.spin_systems += spin_systems  # add the spin systems
sim.methods += [method]  # add the method
```

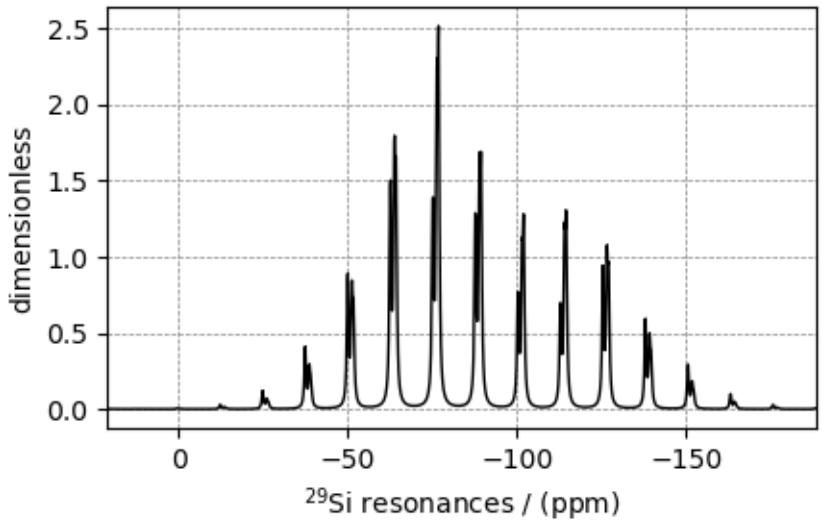
Step 5: Simulate the spectrum.

```
sim.run()

# The plot of the simulation before signal processing.
ax = plt.subplot(projection="csdm")
ax.plot(sim.methods[0].simulation.real, color="black", linewidth=1)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```



Step 6: Add post-simulation signal processing.

```
processor = sp.SignalProcessor(
    operations=[sp.IFFT(), apo.Exponential(FWHM="70 Hz"), sp.FFT()]
)
processed_data = processor.apply_operations(data=sim.methods[0].simulation)

# The plot of the simulation after signal processing.
ax = plt.subplot(projection="csdm")
ax.plot(processed_data.real, color="black", linewidth=1)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 1.045 seconds)

Potassium Sulfate, 33S (I=3/2)

33S (I=3/2) quadrupolar spectrum simulation.

The following example is the ^{33}S NMR spectrum simulation of potassium sulfate (K_2SO_4). The quadrupole tensor parameters for ^{33}S is obtained from Moudrakovski *et. al.*[1]

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import mrsimulator.signal_processing as sp
import mrsimulator.signal_processing.apodization as apo
from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.methods import BlochDecayCentralTransitionSpectrum

# global plot configuration
mpl.rcParams["figure.figsize"] = [4.5, 3.0]
```

Step 1: Create the spin system

```
site = Site(
    name="33S",
    isotope="33S",
    isotropic_chemical_shift=335.7,  # in ppm
    quadrupolar={"Cq": 0.959e6, "eta": 0.42},  # Cq is in Hz
)
spin_system = SpinSystem(sites=[site])
```

Step 2: Create a central transition selective Bloch decay spectrum method.

```
method = BlochDecayCentralTransitionSpectrum(
    channels=["33S"],
    magnetic_flux_density=21.14,  # in T
    rotor_frequency=14000,  # in Hz
    spectral_dimensions=[
        {
            "count": 2048,
            "spectral_width": 5000,  # in Hz
            "reference_offset": 22500,  # in Hz
            "label": r"$^{33}$S resonances",
        }
    ],
)
```

Step 3: Create the Simulator object and add method and spin system objects.

```
sim = Simulator()
sim.spin_systems += [spin_system]  # add the spin system
sim.methods += [method]  # add the method
```

Step 4: Simulate the spectrum.

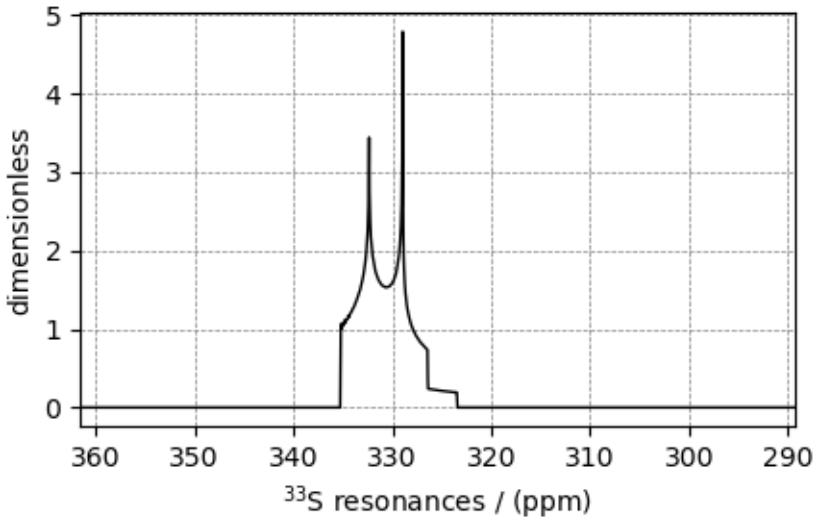
```
sim.run()

# The plot of the simulation before signal processing.
ax = plt.subplot(projection="csdm")
ax.plot(sim.methods[0].simulation.real, color="black", linewidth=1)
```

(continues on next page)

[1] Moudrakovski, I., Lang, S., Patchkovskii, S., and Ripmeester, J. High field ^{33}S solid state NMR and first-principles calculations in potassium sulfates. J. Phys. Chem. A, 2010, 114, 1, 309316. DOI: 10.1021/jp908206c

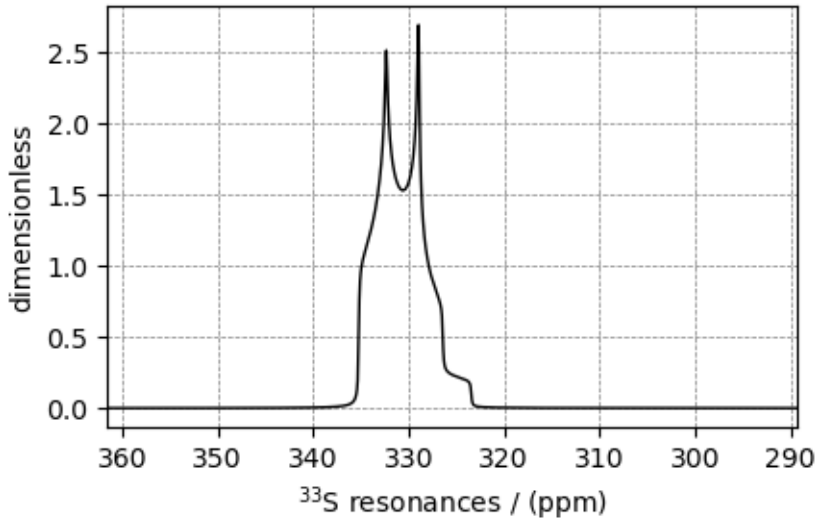
(continued from previous page)

```
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```



Step 5: Add post-simulation signal processing.

```
processor = sp.SignalProcessor(
    operations=[sp.IFFT(), apo.Exponential(FWHM="10 Hz"), sp.FFT()]
)
processed_data = processor.apply_operations(data=sim.methods[0].simulation)

# The plot of the simulation after signal processing.
ax = plt.subplot(projection="csdm")
ax.plot(processed_data.real, color="black", linewidth=1)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.409 seconds)

Coesite, 17O (I=5/2)

17O (I=5/2) quadrupolar spectrum simulation.

Coesite is a high-pressure (2-3 GPa) and high-temperature (700°C) polymorph of silicon dioxide SiO_2. Coesite has five crystallographic ^{17}O sites. In the following, we use the ^{17}O EFG tensor information from Grandinetti *et. al.*[1]

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import mrsimulator.signal_processing as sp
import mrsimulator.signal_processing.apodization as apo
from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.methods import BlochDecayCentralTransitionSpectrum

# global plot configuration
mpl.rcParams["figure.figsize"] = [4.5, 3.0]
```

Step 1: Create the sites.

```
# default unit of isotropic_chemical_shift is ppm and Cq is Hz.
O17_1 = Site(
    isotope="17O", isotropic_chemical_shift=29, quadrupolar={"Cq": 6.05e6, "eta": 0.000}
)
O17_2 = Site(
    isotope="17O", isotropic_chemical_shift=41, quadrupolar={"Cq": 5.43e6, "eta": 0.166}
)
O17_3 = Site(
    isotope="17O", isotropic_chemical_shift=57, quadrupolar={"Cq": 5.45e6, "eta": 0.168}
)
O17_4 = Site(
    isotope="17O", isotropic_chemical_shift=53, quadrupolar={"Cq": 5.52e6, "eta": 0.169}
)
```

(continues on next page)

[1] Grandinetti, P. J., Baltisberger, J. H., Farnan, I., Stebbins, J. F., Werner, U. and Pines, A. Solid-State ^{17}O Magic-Angle and Dynamic-Angle Spinning NMR Study of the SiO_2 Polymorph Coesite, J. Phys. Chem. 1995, 99, 32, 12341-12348. DOI: 10.1021/j100032a045

(continued from previous page)

```
O17_5 = Site(
    isotope="17O", isotropic_chemical_shift=58, quadrupolar={"Cq": 5.16e6, "eta": 0.292}
)

# all five sites.
sites = [O17_1, O17_2, O17_3, O17_4, O17_5]
```

Step 2: Create the spin systems from these sites. For optimum performance, we create five single-site spin systems instead of a single five-site spin system. The abundance of each spin system is taken from above reference.

```
abundance = [0.83, 1.05, 2.16, 2.05, 1.90]
spin_systems = [SpinSystem(sites=[s], abundance=a) for s, a in zip(sites, abundance)]
```

Step 3: Create a central transition selective Bloch decay spectrum method.

```
method = BlochDecayCentralTransitionSpectrum(
    channels=["17O"],
    rotor_frequency=14000,  # in Hz
    spectral_dimensions=[
        {
            "count": 2048,
            "spectral_width": 50000,  # in Hz
            "label": r"$^{17}$O resonances",
        }
    ],
)
```

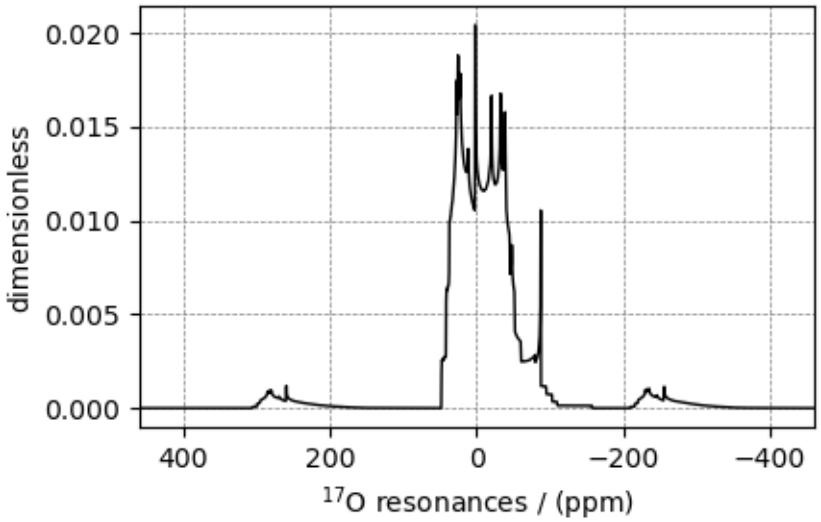
The above method is set up to record the ^{17}O resonances at the magic angle, spinning at 14 kHz and 9.4 T (default, if the value is not provided) external magnetic flux density. The resonances are recorded over 50 kHz spectral width using 2048 points.

Step 4: Create the Simulator object and add the method and spin system objects.

```
sim = Simulator()
sim.spin_systems = spin_systems  # add the spin systems
sim.methods = [method]  # add the method
```

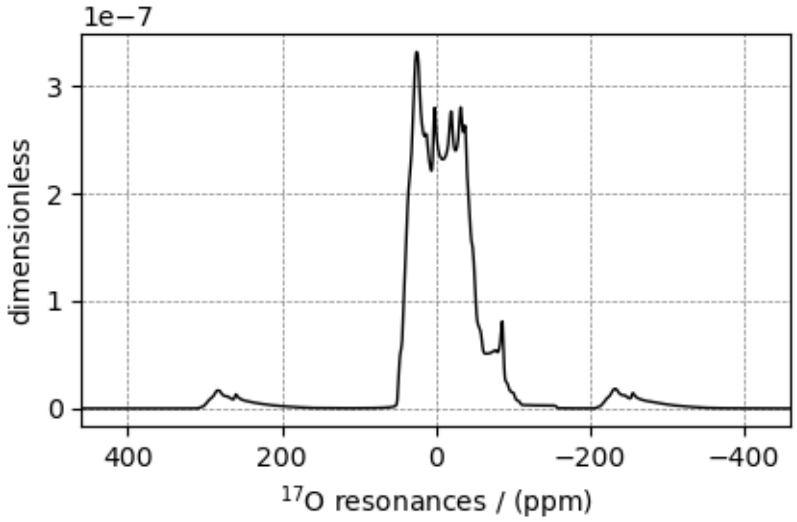
Step 5: Simulate the spectrum.

```
sim.run()

# The plot of the simulation before signal processing.
ax = plt.subplot(projection="csdm")
ax.plot(sim.methods[0].simulation.real, color="black", linewidth=1)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```



Step 6: Add post-simulation signal processing.

```
processor = sp.SignalProcessor(
    operations=[
        sp.IFFT(),
        apo.Exponential(FWHM="30 Hz"),
        apo.Gaussian(FWHM="145 Hz"),
        sp.FFT(),
    ]
)
processed_data = processor.apply_operations(data=sim.methods[0].simulation)

# The plot of the simulation after signal processing.
ax = plt.subplot(projection="csdm")
ax.plot(processed_data.real, color="black", linewidth=1)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.446 seconds)

Non-coincidental Quad and CSA, 17O (I=5/2)

17O (I=5/2) quadrupolar static spectrum simulation.

The following example illustrates the simulation of NMR spectra arising from non-coincidental quadrupolar and shielding tensors. The tensor parameter values for the simulation are obtained from Yamada *et. al.*[1], for the ^{17}O site in benzanilide.

Warning: The Euler angles representation using by Yamada *et. al* is different from the representation used in mrsimulator. The resulting simulation might not resemble the published spectrum.

```
import matplotlib as mpl
import matplotlib.pyplot as plt
from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.methods import BlochDecayCentralTransitionSpectrum
import numpy as np

# global plot configuration
mpl.rcParams["figure.figsize"] = [4.5, 3.0]
```

Step 1: Create the spin system.

```
site = Site(
    isotope="17O",
    isotropic_chemical_shift=320,  # in ppm
    shielding_symmetric={"zeta": 376.667, "eta": 0.345},
    quadrupolar={
        "Cq": 8.97e6,  # in Hz
        "eta": 0.15,
        "alpha": 5 * np.pi / 180,
        "beta": np.pi / 2,
        "gamma": 70 * np.pi / 180,
    },
)
spin_system = SpinSystem(sites=[site])
```

Step 2: Create a central transition selective Bloch decay spectrum method.

```
method = BlochDecayCentralTransitionSpectrum(
    channels=["17O"],
    magnetic_flux_density=11.74,  # in T
    rotor_frequency=0,  # in Hz
    spectral_dimensions=[
        {
            "count": 1024,
            "spectral_width": 1e5,  # in Hz
            "reference_offset": 22500,  # in Hz
            "label": r"$^{17}$O resonances",
        }
    ],
)
```

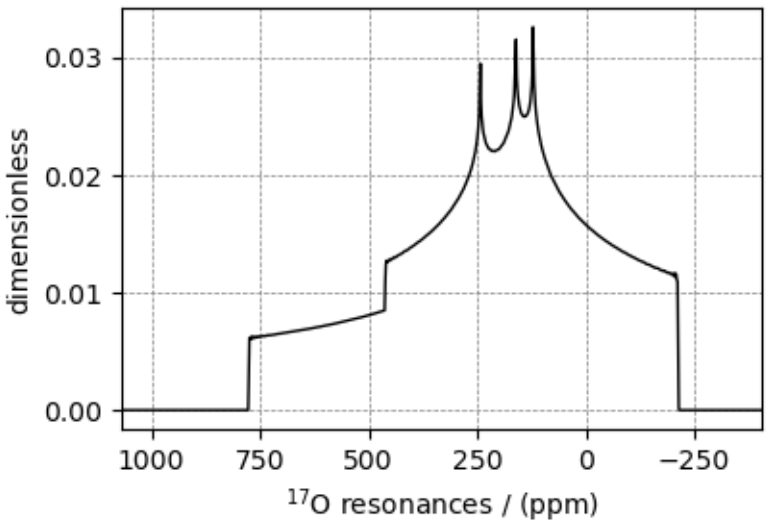
[1] Yamada, K., Dong, S., Wu, G., Solid-State 17O NMR Investigation of the Carbonyl Oxygen Electric-Field-Gradient Tensor and Chemical Shielding Tensor in Amides, J. Am. Chem. Soc. 2000, 122, 11602-11609. DOI: 10.1021/ja0008315

Step 3: Create the Simulator object and add method and spin system objects.

```
sim = Simulator()
sim.spin_systems = [spin_system]  # add the spin system
sim.methods = [method]  # add the method

# Since the spin system have non-zero Euler angles, set the integration_volume to
# hemisphere.
sim.config.integration_volume = "hemisphere"
```

Step 4: Simulate the spectrum.

```
sim.run()

# The plot of the simulation before signal processing.
ax = plt.subplot(projection="csdm")
ax.plot(sim.methods[0].simulation.real, color="black", linewidth=1)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.221 seconds)

Simulate arbitrary transitions (single-quantum)

27Al (I=5/2) quadrupolar spectrum simulation.

The mrsimulator built-in one-dimensional methods, BlochDecaySpectrum and BlochDecayCentralTransitionSpectrum, are designed to simulate spectrum from all single quantum transitions or central transition selective transition, respectively. In this example, we show how you can simulate any arbitrary transition using the generic Method1D method.

```
import matplotlib as mpl
import matplotlib.pyplot as plt
from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.methods import Method1D
```

(continues on next page)

(continued from previous page)

```
# global plot configuration
mpl.rcParams["figure.figsize"] = [4.5, 3.0]
```

Create a single-site arbitrary spin system.

```
site = Site(
    name="27Al",
    isotope="27Al",
    isotropic_chemical_shift=35.7,  # in ppm
    quadrupolar={"Cq": 5.959e6, "eta": 0.32},  # Cq is in Hz
)
spin_system = SpinSystem(sites=[site])
```

Selecting spin transitions for simulation

The arguments of the Method1D object are the same as the arguments of the BlochDecaySpectrum method; however, unlike a BlochDecaySpectrum method, the *SpectralDimension* (page 173) object in Method1D contains additional argument—*events*.

The *Event* (page 174) object is a collection of attributes, which are local to the event. It is here where we define a *transition_query* to select one or more transitions for simulating the spectrum. The two attributes of the *transition_query* are *P* and *D*, which are given as,

$$\begin{aligned} P &= m_f - m_i \\ D &= m_f^2 - m_i^2, \end{aligned} \tag{4.1}$$

where m_f and m_i are the spin quantum numbers for the final and initial energy states. Based on the query, the method selects all transitions from the spin system that satisfy the query selection criterion. For example, to simulate a spectrum for the satellite transition, $|-1/2\rangle \rightarrow |-3/2\rangle$, set the value of

$$\begin{aligned} P &= (-3/2) - (-1/2) = -1 \\ D &= (9/4) - (1/4) = 2. \end{aligned} \tag{4.2}$$

For illustrative purposes, let's look at the infinite speed spectrum from this satellite transition.

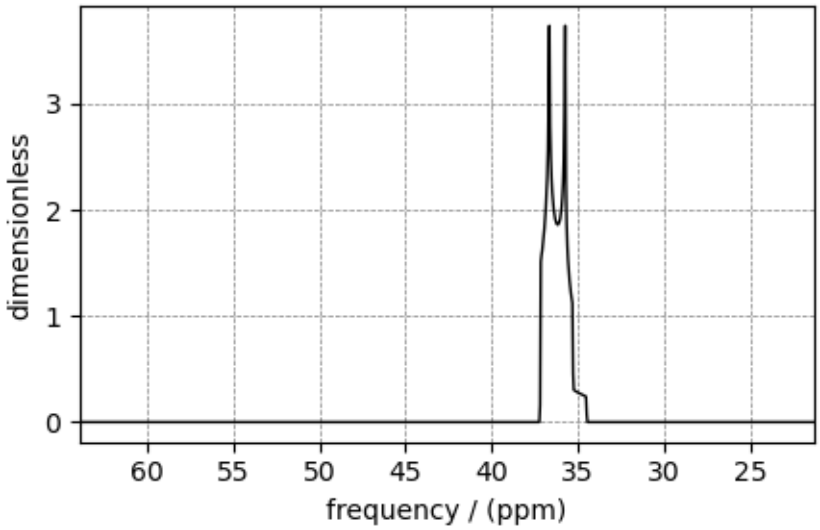
```
method = Method1D(
    channels=["27Al"],
    magnetic_flux_density=21.14,  # in T
    rotor_frequency=1e9,  # in Hz
    spectral_dimensions=[
        {
            "count": 1024,
            "spectral_width": 1e4,  # in Hz
            "reference_offset": 1e4,  # in Hz
            "events": [
                {"transition_query": {"P": [-1], "D": [2]}}  # <-- select transitions
            ],
        }
    ],
)
```

Create the Simulator object and add the method and the spin system object.

```
sim = Simulator()
sim.spin_systems += [spin_system]  # add the spin system
sim.methods += [method]  # add the method
```

Simulate the spectrum.

```
sim.run()

# The plot of the simulation before signal processing.
ax = plt.subplot(projection="csdm")
ax.plot(sim.methods[0].simulation.real, color="black", linewidth=1)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```



Selecting both inner and outer-satellite transitions

You may use the same transition query selection criterion to select multiple transitions. Consider the following transitions with respective P and D values.

- $|-1/2\rangle \rightarrow |-3/2\rangle$ $(P = -1, D = 2)$
- $|-3/2\rangle \rightarrow |-5/2\rangle$ $(P = -1, D = 4)$

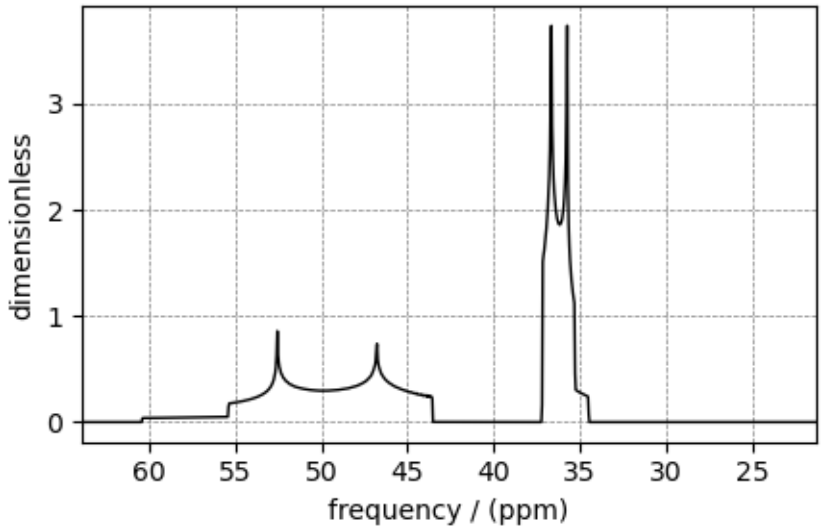
```
method2 = Method1D(
    channels=["27Al"],
    magnetic_flux_density=21.14,  # in T
    rotor_frequency=1e9,  # in Hz
    spectral_dimensions=[
        {
            "count": 1024,
            "spectral_width": 1e4,  # in Hz
            "reference_offset": 1e4,  # in Hz
            "events": [
                {"transition_query": {"P": [-1], "D": [2, 4]}}  # <-- select transitions
            ],
        }
    ],
)
```

Update the method object in the Simulator object.

```
sim.methods[0] = method2  # add the method
```

Simulate the spectrum.

```
sim.run()

# The plot of the simulation before signal processing.
ax = plt.subplot(projection="csdm")
ax.plot(sim.methods[0].simulation.real, color="black", linewidth=1)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.417 seconds)

Simulate arbitrary transitions (multi-quantum)

33S (I=5/2) quadrupolar spectrum simulation.

Simulate a triple quantum spectrum.

```
import matplotlib as mpl
import matplotlib.pyplot as plt
from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.methods import Method1D

# global plot configuration
mpl.rcParams["figure.figsize"] = [4.5, 3.0]
```

Create a single-site arbitrary spin system.

```
site = Site(
    name="27Al",
    isotope="27Al",
    isotropic_chemical_shift=35.7,  # in ppm
    quadrupolar={"Cq": 2.959e6, "eta": 0.98},  # Cq is in Hz
```

(continues on next page)

(continued from previous page)

```
)
spin_system = SpinSystem(sites=[site])
```

Selecting the triple-quantum transition

For spin-site spin-5/2 spin system, there are three triple-quantum transition

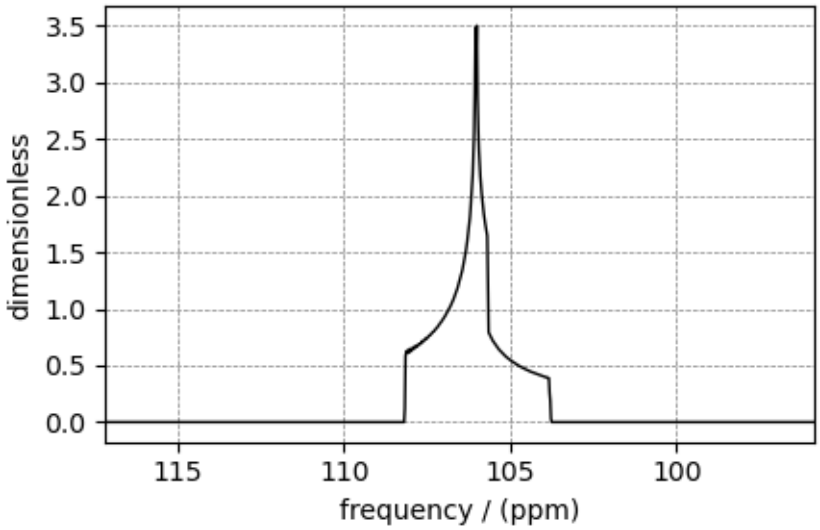
- $|1/2\rangle \rightarrow |-5/2\rangle$ ($P = -3, D = 6$)
- $|3/2\rangle \rightarrow |-3/2\rangle$ ($P = -3, D = 0$)
- $|5/2\rangle \rightarrow |-1/2\rangle$ ($P = -3, D = -6$)

To select one or more triple-quantum transitions, assign the respective value of P and D to the *transition_query*.

```
method = Method1D(
    channels=["27Al"],
    magnetic_flux_density=21.14,  # in T
    rotor_frequency=1e9,  # in Hz
    spectral_dimensions=[
        {
            "count": 1024,
            "spectral_width": 5e3,  # in Hz
            "reference_offset": 2.5e4,  # in Hz
            "events": [
                {  # symmetric triple quantum transitions
                    "transition_query": {"P": [-3], "D": [0]}
                }
            ],
        }
    ],
)
```

Create the Simulator object and add the method and the spin system object.

```
sim = Simulator()
sim.spin_systems += [spin_system]  # add the spin system
sim.methods += [method]  # add the method
sim.run()

# The plot of the simulation before signal processing.
ax = plt.subplot(projection="csdm")
ax.plot(sim.methods[0].simulation.real, color="black", linewidth=1)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.205 seconds)

4.1.2 1D NMR simulation (macromolecules/amorphous solids)

The following examples are the NMR spectrum simulation of macromolecules and amorphous materials for the following methods:

- Bloch decay method (*BlochDecaySpectrum* (page 177)),
- Central transition selective Bloch decay method (*BlochDecayCentralTransitionSpectrum* (page 178)).

For NMR simulation of amorphous solids, we also show examples of simulating spectrum using user-deined model or using commonly accepted models such as Czjzek or extended Czjzek distribution.

Protein GB1, 13C and 15N (I=1/2)

13C/15N (I=1/2) spinning sideband simulation.

The following is the spinning sideband simulation of a macromolecule, protein GB1. The ^{13}C and ^{15}N CSA tensor parameters were obtained from Hung *et. al.*[1], which consists of 42 ^{13}Cα, 44 ^{13}CO, and 44 ^{15}NH tensors. In the following example, instead of creating 130 spin systems, we download the spin systems from a remote file and load it directly to the Simulator object.

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import mrsimulator.signal_processing as sp
import mrsimulator.signal_processing.apodization as apo
from mrsimulator import Simulator
from mrsimulator.methods import BlochDecaySpectrum

# global plot configuration
mpl.rcParams["figure.figsize"] = [9, 4]
```

Create the Simulator object and load the spin systems from an external file.

[1] Hung I., Ge Y., Liu X., Liu M., Li C., Gan Z., Measuring ^{13}C/^{15}N chemical shift anisotropy in [^{13}C, ^{15}N] uniformly enriched proteins using CSA amplification, Solid State Nuclear Magnetic Resonance. 2015, 72, 96-103. DOI: 10.1016/j.ssnmr.2015.09.002

```
sim = Simulator()

file_ = "https://sandbox.zenodo.org/record/687656/files/protein_GB1_15N_13CA_13CO.mrsys"
sim.load_spin_systems(file_)  # load the spin systems.
print(f"number of spin systems = {len(sim.spin_systems)}")
```

Out:

```
number of spin systems = 130
```

Create a ^{13}C Bloch decay spectrum method.

```
method_13C = BlochDecaySpectrum(
    channels=["13C"],
    magnetic_flux_density=11.7,  # in T
    rotor_frequency=3000,  # in Hz
    spectral_dimensions=[
        {
            "count": 8192,
            "spectral_width": 5e4,  # in Hz
            "reference_offset": 2e4,  # in Hz
            "label": r"$^{13}$C resonances",
        }
    ],
)
```

Since the spin systems contain both ^{13}C and ^{15}N sites, let's also create a ^{15}N Bloch decay spectrum method.

```
method_15N = BlochDecaySpectrum(
    channels=["15N"],
    magnetic_flux_density=11.7,  # in T
    rotor_frequency=3000,  # in Hz
    spectral_dimensions=[
        {
            "count": 8192,
            "spectral_width": 4e4,  # in Hz
            "reference_offset": 7e3,  # in Hz
            "label": r"$^{15}$N resonances",
        }
    ],
)
```

Add the methods to the Simulator object and run the simulation

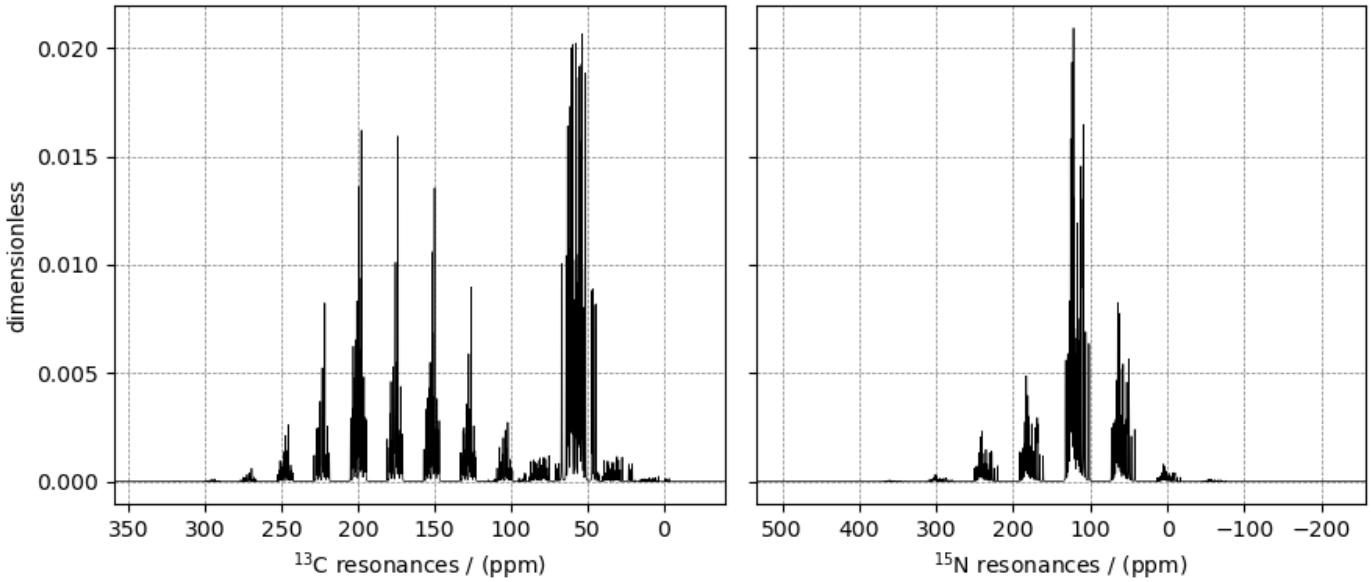
```
# Add the methods.
sim.methods = [method_13C, method_15N]

# Run the simulation.
sim.run()

# Get the simulation data from the respective methods.
data_13C = sim.methods[0].simulation  # method at index 0 is 13C Bloch decay method.
data_15N = sim.methods[1].simulation  # method at index 1 is 15N Bloch decay method.
```

Add post-simulation signal processing.

```
processor = sp.SignalProcessor(
    operations=[sp.IFFT(), apo.Exponential(FWHM="10 Hz"), sp.FFT()]
)
# apply post-simulation processing to data_13C
processed_data_13C = processor.apply_operations(data=data_13C).real

# apply post-simulation processing to data_15N
processed_data_15N = processor.apply_operations(data=data_15N).real
```

The plot of the simulation after signal processing.

```
fig, ax = plt.subplots(1, 2, subplot_kw={"projection": "csdm"}, sharey=True)

ax[0].plot(processed_data_13C, color="black", linewidth=0.5)
ax[0].invert_xaxis()

ax[1].plot(processed_data_15N, color="black", linewidth=0.5)
ax[1].set_ylabel(None)
ax[1].invert_xaxis()

plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 3.302 seconds)

Amorphous material, 29Si (I=1/2)

29Si (I=1/2) simulation of amorphous material.

One of the advantages of the `mrsimulator` package is that it is a fast NMR spectrum simulation library. We can exploit this feature to simulate bulk spectra and eventually model amorphous materials. In this section, we illustrate how the `mrsimulator` library may be used in simulating the NMR spectrum of amorphous materials.

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
from mrsimulator import Simulator
from mrsimulator.methods import BlochDecaySpectrum
from mrsimulator.utils.collection import single_site_system_generator
from scipy.stats import multivariate_normal

# global plot configuration
mpl.rcParams["figure.figsize"] = [4.5, 3.0]
```

Generating tensor parameter distribution

We model the amorphous material by assuming a distribution of interaction tensors. For example, a tri-variate normal distribution of the shielding tensor parameters, *i.e.*, the isotropic chemical shift, the anisotropy parameter, ζ, and the asymmetry parameter, η. In the following, we use pure NumPy and SciPy methods to generate the three-dimensional distribution, as follows,

```
mean = [-100, 50, 0.15]  # given as [isotropic chemical shift in ppm, zeta in ppm, eta].
covariance = [[3.25, 0, 0], [0, 26.2, 0], [0, 0, 0.002]]  # same order as the mean.

# range of coordinates along the three dimensions
iso_range = np.arange(100) * 0.3055 - 115  # in ppm
zeta_range = np.arange(30) * 2.5 + 10  # in ppm
eta_range = np.arange(21) / 20

# The coordinates grid
iso, zeta, eta = np.meshgrid(iso_range, zeta_range, eta_range, indexing="ij")
pos = np.asarray([iso, zeta, eta]).T

# Three-dimensional probability distribution function.
pdf = multivariate_normal(mean=mean, cov=covariance).pdf(pos).T
```

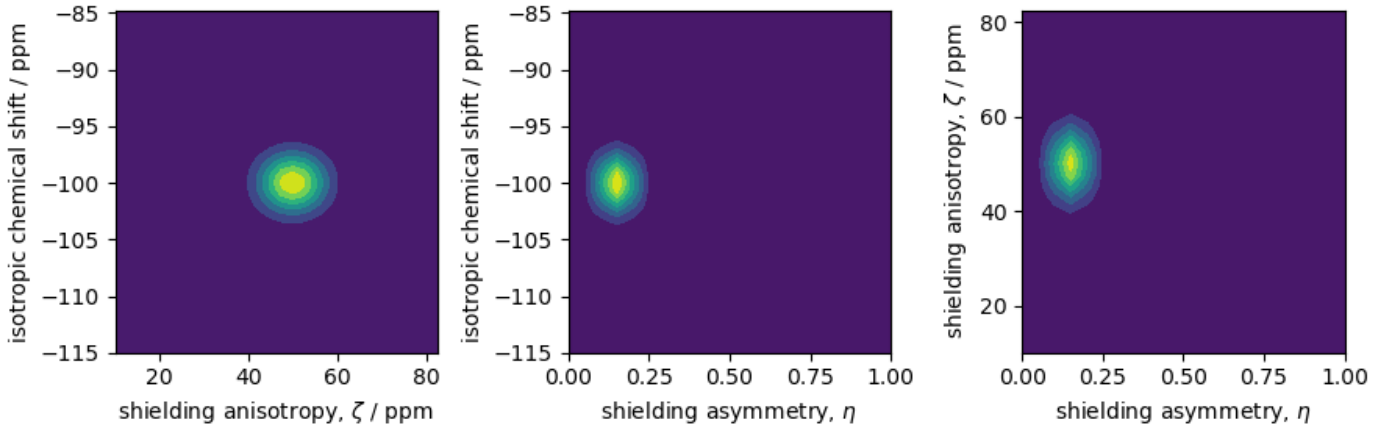
Here, `iso`, `zeta`, and `eta` are the isotropic chemical shift, nuclear shielding anisotropy, and nuclear shielding asymmetry coordinates of the 3D-grid system over which the multivariate normal probability distribution is evaluated. The mean of the distribution is given by the variable `mean` and holds a value of -100 ppm, 50 ppm, and 0.15 for the isotropic chemical shift, nuclear shielding anisotropy, and nuclear shielding asymmetry parameter, respectively. Similarly, the variable `covariance` holds the covariance matrix of the multivariate normal distribution. The two-dimensional projections from this three-dimensional distribution are shown below.

```
_, ax = plt.subplots(1, 3, figsize=(9, 3))

# isotropic shift v.s. shielding anisotropy
ax[0].contourf(zeta_range, iso_range, pdf.sum(axis=2))
ax[0].set_xlabel(r"shielding anisotropy, $\zeta$ / ppm")
ax[0].set_ylabel("isotropic chemical shift / ppm")

# isotropic shift v.s. shielding asymmetry
ax[1].contourf(eta_range, iso_range, pdf.sum(axis=1))
ax[1].set_xlabel(r"shielding asymmetry, $\eta$")
```

(continues on next page)

(continued from previous page)

```
ax[1].set_ylabel("isotropic chemical shift / ppm")

# shielding anisotropy v.s. shielding asymmetry
ax[2].contourf(eta_range, zeta_range, pdf.sum(axis=0))
ax[2].set_xlabel(r"shielding asymmetry, $\eta$")
ax[2].set_ylabel(r"shielding anisotropy, $\zeta$ / ppm")

plt.tight_layout()
plt.show()
```



Create the Simulator object

Spin system: Let's create the sites and single-site spin system objects from these parameters. Use the `single_site_system_generator()` utility function to generate single-site spin systems.

```
spin_systems = single_site_system_generator(
    isotopes="29Si",
    isotropic_chemical_shifts=iso,
    shielding_symmetric={"zeta": zeta, "eta": eta},
    abundance=pdf,
)
```

Here, `iso`, `zeta`, and `eta` are the array of tensor parameter coordinates, and `pdf` is the array of corresponding amplitudes.

Method: Let's also create the Bloch decay spectrum method.

```
method = BlochDecaySpectrum(
    channels=["29Si"],
    spectral_dimensions=[
        {"spectral_width": 25000, "reference_offset": -7000}  # values in Hz
    ],
)
```

The above method simulates a static ^{29}Si spectrum at 9.4 T field (default value).

Simulator: Now, that we have the spin systems and the method, create the simulator object and add the respective objects.

```
sim = Simulator()
sim.spin_systems = spin_systems  # add the spin systems
sim.methods += [method]  # add the method
```

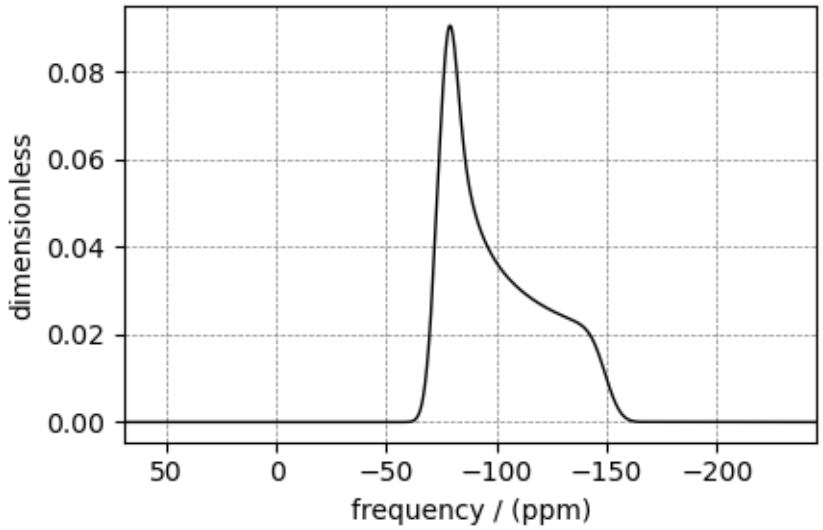
Static spectrum

Observe the static ^{29}Si NMR spectrum simulation.

```
sim.run()
```

The plot of the simulation.

```
ax = plt.subplot(projection="csdm")
ax.plot(sim.methods[0].simulation, color="black", linewidth=1)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```



Note: The broad spectrum seen in the above figure is a result of spectral averaging of spectra arising from a distribution of shielding tensors. There is no line-broadening filter applied to the spectrum.

Spinning sideband simulation at 90°

Here is an example of a sideband simulation, spinning at a 90-degree angle.

```
sim.methods[0] = BlochDecaySpectrum(
    channels=["29Si"],
    rotor_frequency=5000,  # in Hz
    rotor_angle=1.57079,  # in rads, equivalent to 90 deg.
    spectral_dimensions=[
        {"spectral_width": 25000, "reference_offset": -7000}  # values in Hz
```

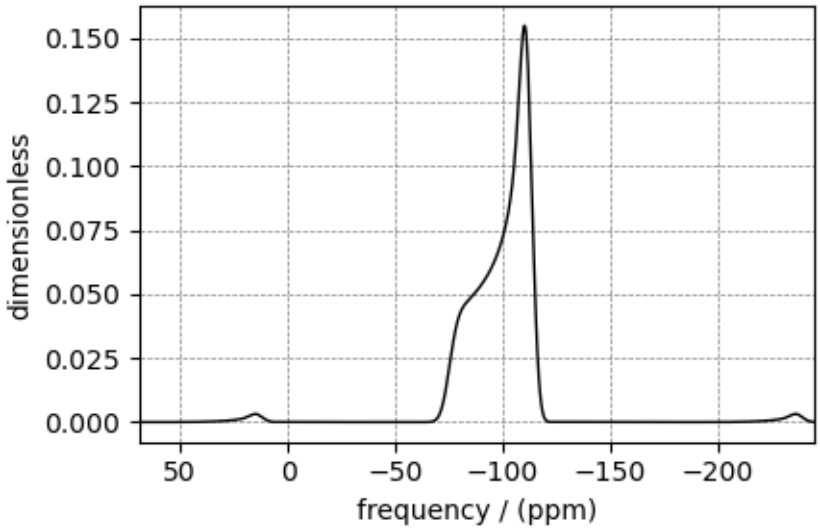
(continues on next page)

(continued from previous page)

```
    ],
)
sim.config.number_of_sidebands = 8  # eight sidebands are sufficient for this example
sim.run()
```

The plot of the simulation.

```
ax = plt.subplot(projection="csdm")
ax.plot(sim.methods[0].simulation, color="black", linewidth=1)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```



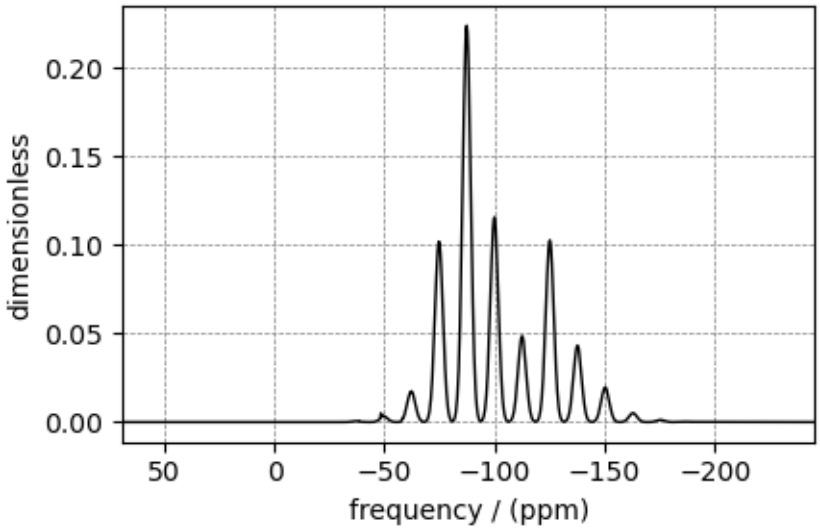
Spinning sideband simulation at the magic angle

Here is another example of a sideband simulation at the magic angle.

```
sim.methods[0] = BlochDecaySpectrum(
    channels=["29Si"],
    rotor_frequency=1000,  # in Hz
    rotor_angle=54.735 * np.pi / 180.0,  # in rads
    spectral_dimensions=[
        {"spectral_width": 25000, "reference_offset": -7000}  # values in Hz
    ],
)
sim.config.number_of_sidebands = 16  # sixteen sidebands are sufficient for this example
sim.run()
```

The plot of the simulation.

```
ax = plt.subplot(projection="csdm")
ax.plot(sim.methods[0].simulation, color="black", linewidth=1)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 22.762 seconds)

Amorphous material, 27Al (I=5/2)

27Al (I=5/2) simulation of amorphous material.

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
from mrsimulator import Simulator
from mrsimulator.methods import BlochDecayCentralTransitionSpectrum
from mrsimulator.utils.collection import single_site_system_generator
from scipy.stats import multivariate_normal

# global plot configuration
mpl.rcParams["figure.figsize"] = [4.5, 3.0]
```

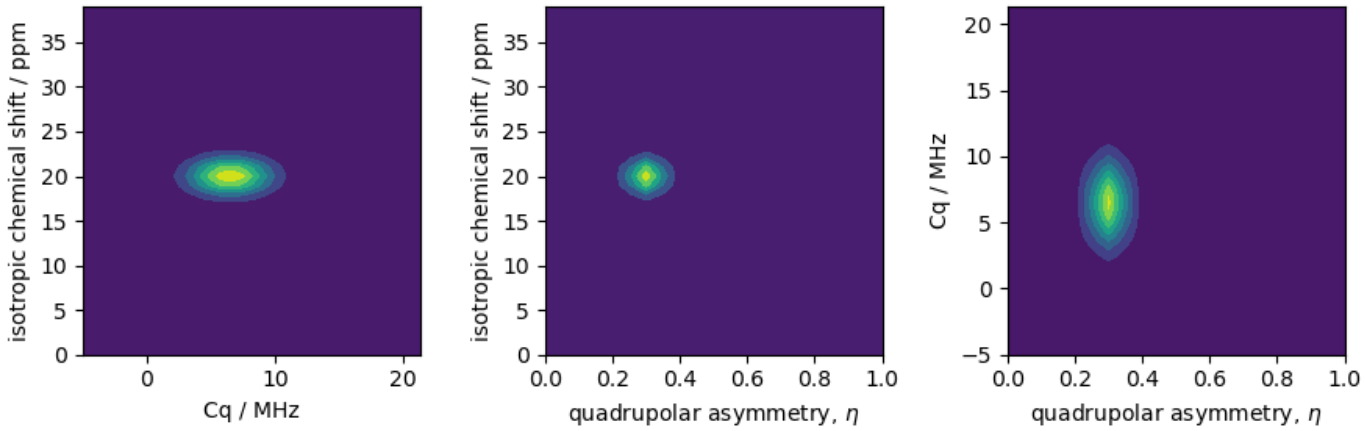
In this section, we illustrate the simulation of a quadrupolar spectrum arising from a distribution of the electric field gradient (EFG) tensors from an amorphous material. We proceed by assuming a multi-variate normal distribution, as follows,

```
mean = [20, 6.5, 0.3]  # given as [isotropic chemical shift in ppm, Cq in MHz, eta].
covariance = [[1.98, 0, 0], [0, 4.9, 0], [0, 0, 0.0016]]  # same order as the mean.

# range of coordinates along the three dimensions
iso_range = np.arange(40)  # in ppm
Cq_range = np.arange(80) / 3 - 5  # in MHz
eta_range = np.arange(21) / 20

# The coordinates grid
iso, Cq, eta = np.meshgrid(iso_range, Cq_range, eta_range, indexing="ij")
pos = np.asarray([iso, Cq, eta]).T

# Three-dimensional probability distribution function.
pdf = multivariate_normal(mean=mean, cov=covariance).pdf(pos).T
```

Here, `iso`, `Cq`, and `eta` are the isotropic chemical shift, the quadrupolar coupling constant, and quadrupolar asymmetry coordinates of the 3D-grid system over which the multivariate normal probability distribution is evaluated. The mean of the distribution is given by the

variable `mean` and holds a value of 20 ppm, 6.5 MHz, and 0.3 for the isotropic chemical shift, the quadrupolar coupling constant, and quadrupolar asymmetry parameter, respectively. Similarly, the variable `covariance` holds the covariance matrix of the multivariate normal distribution. The two-dimensional projections from this three-dimensional distribution are shown below.

```
_, ax = plt.subplots(1, 3, figsize=(9, 3))

# isotropic shift v.s. quadrupolar coupling constant
ax[0].contourf(Cq_range, iso_range, pdf.sum(axis=2))
ax[0].set_xlabel("Cq / MHz")
ax[0].set_ylabel("isotropic chemical shift / ppm")

# isotropic shift v.s. quadrupolar asymmetry
ax[1].contourf(eta_range, iso_range, pdf.sum(axis=1))
ax[1].set_xlabel(r"quadrupolar asymmetry, $\eta$")
ax[1].set_ylabel("isotropic chemical shift / ppm")

# quadrupolar coupling constant v.s. quadrupolar asymmetry
ax[2].contourf(eta_range, Cq_range, pdf.sum(axis=0))
ax[2].set_xlabel(r"quadrupolar asymmetry, $\eta$")
ax[2].set_ylabel("Cq / MHz")

plt.tight_layout()
plt.show()
```



Let's create the site and spin system objects from these parameters. Note, we create single-site spin systems for optimum performance. Use the `single_site_system_generator()` utility function to generate single-site spin systems.

```
spin_systems = single_site_system_generator(
    isotopes="27Al",
    isotropic_chemical_shifts=iso,
    quadrupolar={"Cq": Cq * 1e6, "eta": eta},  # Cq in Hz
    abundance=pdf,
)
```

Static spectrum

Observe the static ^{27}Al NMR spectrum simulation. First, create a central transition selective Bloch decay spectrum method.

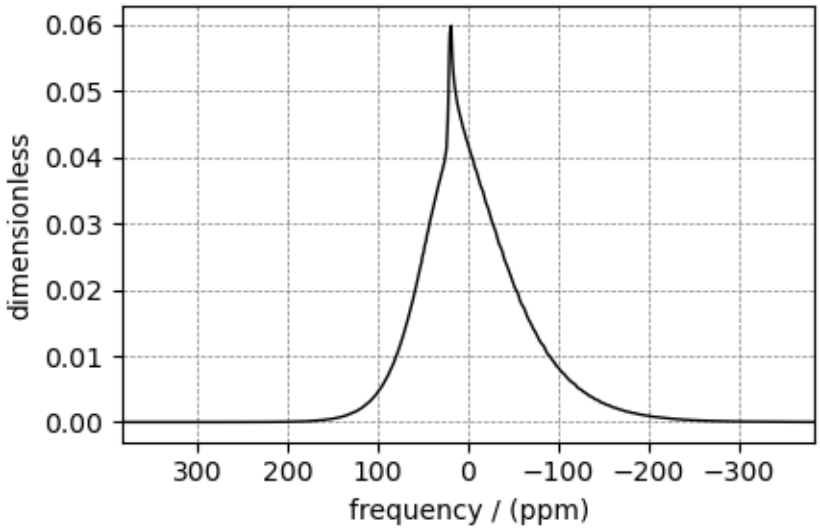
```
static_method = BlochDecayCentralTransitionSpectrum(
    channels=["27Al"], spectral_dimensions=[{"spectral_width": 80000}]
)
```

Create the simulator object and add the spin systems and method.

```
sim = Simulator()
sim.spin_systems = spin_systems  # add the spin systems
sim.methods = [static_method]  # add the method
sim.run()
```

The plot of the corresponding spectrum.

```
ax = plt.subplot(projection="csdm")
ax.plot(sim.methods[0].simulation, color="black", linewidth=1)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```



Spinning sideband simulation at the magic angle

Simulation of the same spin systems at the magic angle and spinning at 25 kHz.

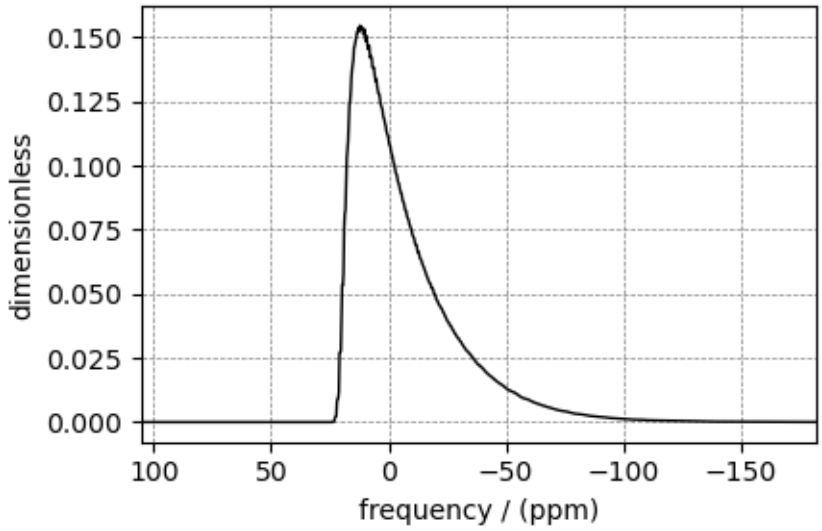
```
MAS_method = BlochDecayCentralTransitionSpectrum(
    channels=["27Al"],
    rotor_frequency=25000,  # in Hz
    rotor_angle=54.735 * np.pi / 180.0,  # in rads
    spectral_dimensions=[
        {"spectral_width": 30000, "reference_offset": -4000}  # values in Hz
    ],
)
sim.methods[0] = MAS_method
```

Configure the sim object to calculate up to 4 sidebands, and run the simulation.

```
sim.config.number_of_sidebands = 4
sim.run()
```

and the corresponding plot.

```
ax = plt.subplot(projection="csdm")
ax.plot(sim.methods[0].simulation, color="black", linewidth=1)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 5.601 seconds)

Czjzek distribution (Shielding and Quadrupolar)

In this example, we illustrate the simulation of spectrum originating from a Czjzek distribution of traceless symmetric tensors. We show two cases, the Czjzek distribution of the shielding and quadrupolar tensor parameters, respectively.

Import the required modules.

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
from mrsimulator import Simulator
from mrsimulator.methods import BlochDecaySpectrum, BlochDecayCentralTransitionSpectrum
from mrsimulator.models import CzjzekDistribution
from mrsimulator.utils.collection import single_site_system_generator

# pre config the figures
mpl.rcParams["figure.figsize"] = [4.25, 3.0]
```

Symmetric shielding tensor

Create the Czjzek distribution

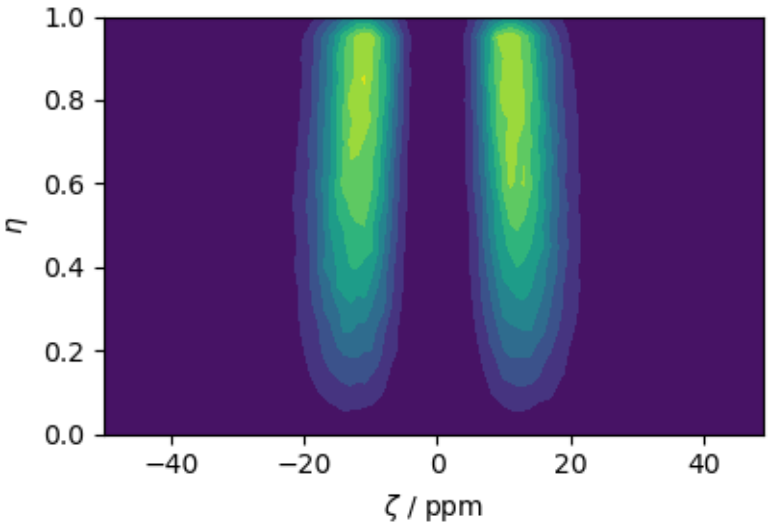
First, create a distribution of the zeta and eta parameters of the shielding tensors using the *Czjzek distribution* (page 39) model as follows.

```
# The range of zeta and eta coordinates over which the distribution is sampled.
z_range = np.arange(100) - 50  # in ppm
e_range = np.arange(21) / 20
z_dist, e_dist, amp = CzjzekDistribution(sigma=3.1415).pdf(pos=[z_range, e_range])
```

Here `z_range` and `e_range` are the coordinates along the ζ and η dimensions that form a two-dimensional ζ-η grid. The argument *sigma* of the CzjzekDistribution class is the standard deviation of the second-rank tensor parameters used in generating the distribution, and *pos* hold the one-dimensional arrays of ζ and η coordinates, respectively.

The following is the contour plot of the Czjzek distribution.

```
plt.contourf(z_dist, e_dist, amp, levels=10)
plt.xlabel(r"$\zeta$ / ppm")
plt.ylabel(r"$\eta$")
plt.tight_layout()
plt.show()
```



Simulate the spectrum

To quickly generate single-site spin systems from the above ζ and η parameters, use the `single_site_system_generator()` utility function.

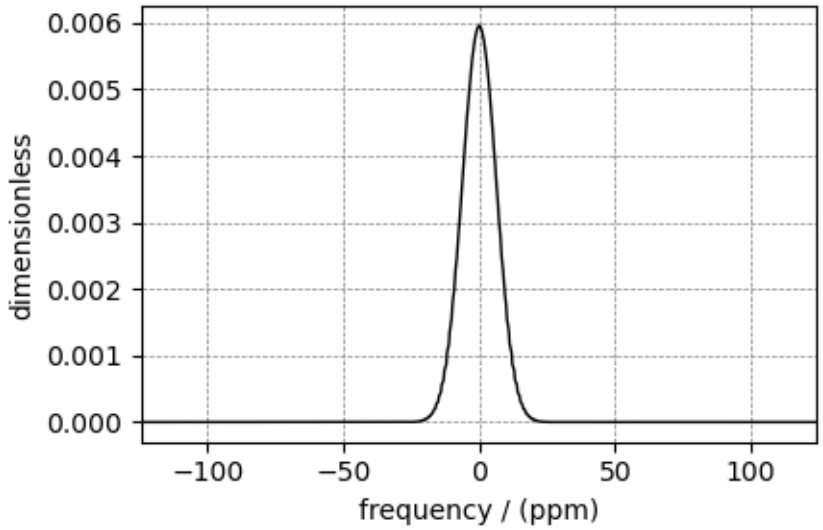
```
systems = single_site_system_generator(
    isotopes="13C", shielding_symmetric={"zeta": z_dist, "eta": e_dist}, abundance=amp
)
```

Here, the variable `systems` hold an array of single-site spin systems. Next, create a simulator object and add the above system and a method.

```
sim = Simulator()
sim.spin_systems = systems  # add the systems
sim.methods = [BlochDecaySpectrum(channels=["13C"])]  # add the method
sim.run()
```

The following is the static spectrum arising from a Czjzek distribution of the second-rank traceless shielding tensors.

```
plt.figure(figsize=(4.5, 3.0))
ax = plt.gca(projection="csdm")
ax.plot(sim.methods[0].simulation, color="black", linewidth=1)
plt.tight_layout()
plt.show()
```

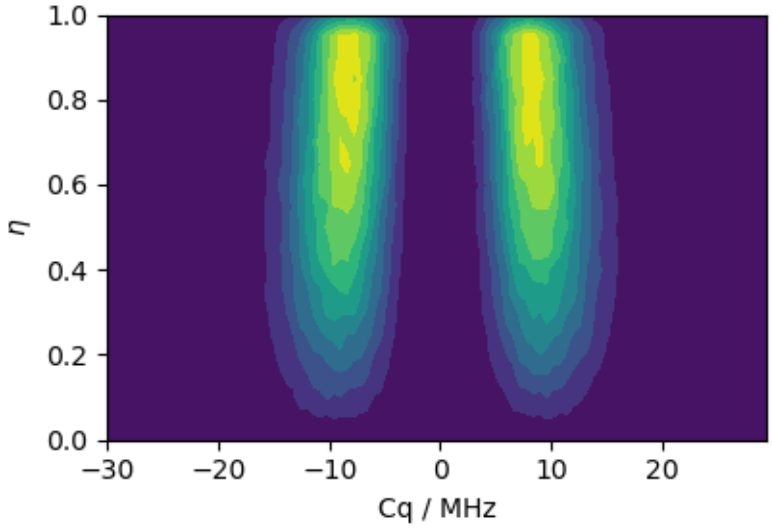


Quadrupolar tensor

Create the Czjzek distribution

Similarly, you may also create a Czjzek distribution of the electric field gradient (EFG) tensor parameters.

```
# The range of Cq and eta coordinates over which the distribution is sampled.
cq_range = np.arange(100) * 0.6 - 30  # in MHz
e_range = np.arange(21) / 20
cq_dist, e_dist, amp = CzjzekDistribution(sigma=2.3).pdf(pos=[cq_range, e_range])

# The following is the contour plot of the Czjzek distribution.
plt.contourf(cq_dist, e_dist, amp, levels=10)
plt.xlabel(r"Cq / MHz")
plt.ylabel(r"$\eta$")
plt.tight_layout()
plt.show()
```



Simulate the spectrum

Create the spin systems.

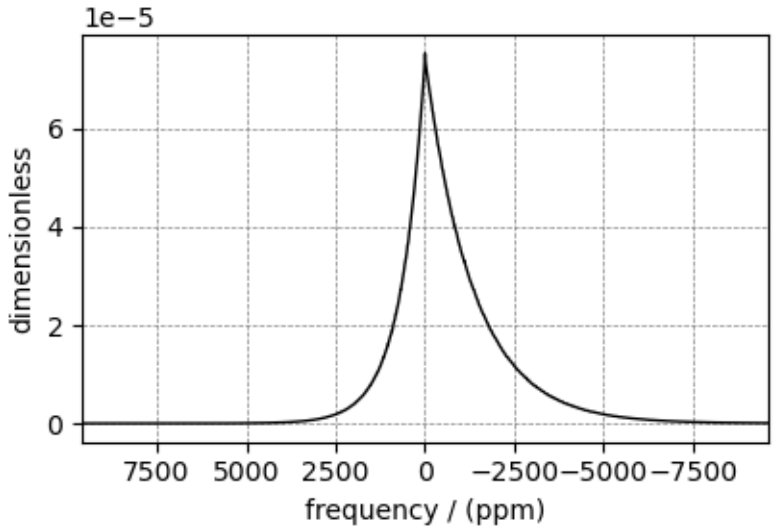
```
systems = single_site_system_generator(
    isotopes="71Ga", quadrupolar={"Cq": cq_dist * 1e6, "eta": e_dist}, abundance=amp
)
```

Create a simulator object and add the above system.

```
sim = Simulator()
sim.spin_systems = systems  # add the systems
sim.methods = [
    BlochDecayCentralTransitionSpectrum(
        channels=["71Ga"],
        magnetic_flux_density=4.8,  # in T
        spectral_dimensions=[{"count": 2048, "spectral_width": 1.2e6}],
    )
]  # add the method
sim.run()
```

The following is the static spectrum arising from a Czjzek distribution of the second-rank traceless EFG tensors.

```
plt.figure(figsize=(4.25, 3.0))
ax = plt.gca(projection="csdm")
ax.plot(sim.methods[0].simulation, color="black", linewidth=1)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 3.926 seconds)

Extended Czjzek distribution (Shielding and Quadrupolar)

In this example, we illustrate the simulation of spectrum originating from an extended Czjzek distribution of traceless symmetric tensors. We show two cases, an extended Czjzek distribution of the shielding and quadrupolar tensor parameters, respectively.

Import the required modules.

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
from mrsimulator import Simulator
from mrsimulator.methods import BlochDecaySpectrum, BlochDecayCentralTransitionSpectrum
from mrsimulator.models import ExtCzjzekDistribution
from mrsimulator.utils.collection import single_site_system_generator

# pre config the figures
mpl.rcParams["figure.figsize"] = [4.25, 3.0]
```

Symmetric shielding tensor

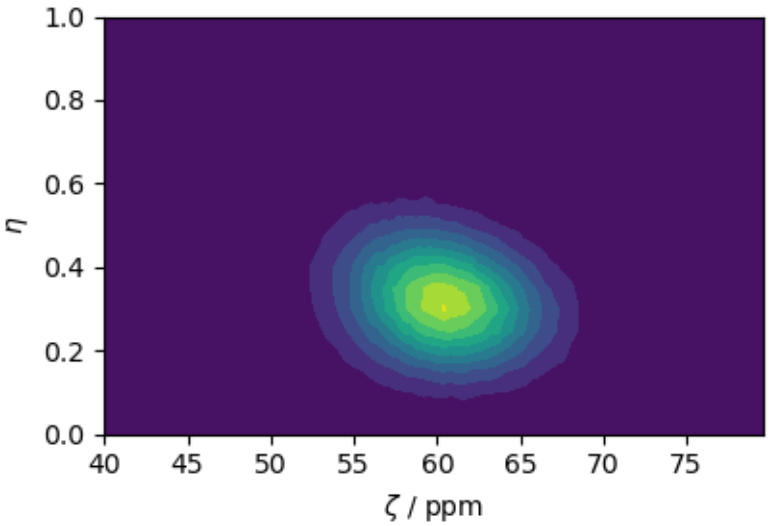
Create the extended Czjzek distribution

First, create a distribution of the zeta and eta parameters of the shielding tensors using the *Extended Czjzek distribution* (page 41) model as follows,

```
# The range of zeta and eta coordinates over which the distribution is sampled.
z_lim = np.arange(100) * 0.4 + 40  # in ppm
e_lim = np.arange(21) / 20

dominant = {"zeta": 60, "eta": 0.3}
z_dist, e_dist, amp = ExtCzjzekDistribution(dominant, eps=0.14).pdf(pos=[z_lim, e_lim])
```

The following is the plot of the extended Czjzek distribution.

```
plt.contourf(z_dist, e_dist, amp, levels=10)
plt.xlabel(r"$\zeta$ / ppm")
plt.ylabel(r"$\eta$")
plt.tight_layout()
plt.show()
```



Simulate the spectrum

Create the spin systems from the above ζ and η parameters.

```
systems = single_site_system_generator(
    isotopes="13C", shielding_symmetric={"zeta": z_dist, "eta": e_dist}, abundance=amp
)
print(len(systems))
```

Out:

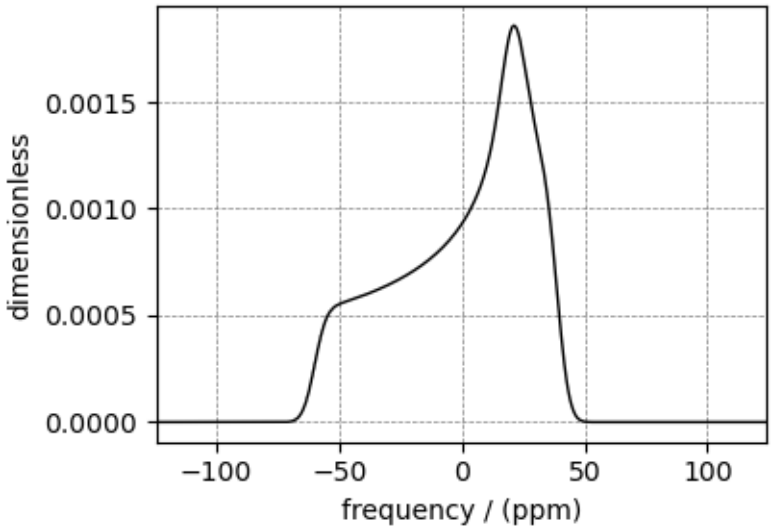
```
838
```

Create a simulator object and add the above system.

```
sim = Simulator()
sim.spin_systems = systems  # add the systems
sim.methods = [BlochDecaySpectrum(channels=["13C"])]  # add the method
sim.run()
```

The following is the static spectrum arising from a Czjzek distribution of the second-rank traceless shielding tensors.

```
plt.figure(figsize=(4.25, 3.0))
ax = plt.gca(projection="csdm")
ax.plot(sim.methods[0].simulation, color="black", linewidth=1)
plt.tight_layout()
plt.show()
```



Quadrupolar tensor

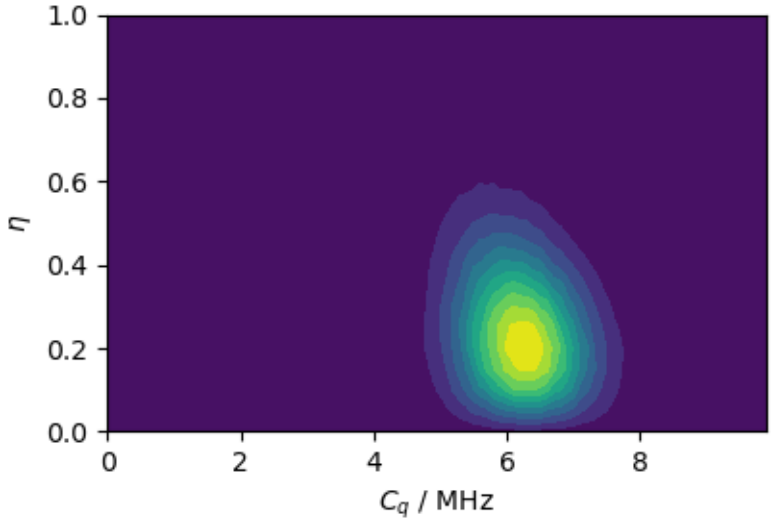
Create the extended Czjzek distribution

Similarly, you may also create an extended Czjzek distribution of the electric field gradient (EFG) tensor parameters.

```
# The range of Cq and eta coordinates over which the distribution is sampled.
cq_lim = np.arange(100) * 0.1  # assumed in MHz
e_lim = np.arange(21) / 20

dominant = {"Cq": 6.1, "eta": 0.1}
cq_dist, e_dist, amp = ExtCzjzekDistribution(dominant, eps=0.25).pdf(
    pos=[cq_lim, e_lim]
)
```

The following is the plot of the extended Czjzek distribution.

```
plt.contourf(cq_dist, e_dist, amp, levels=10)
plt.xlabel(r"$C_q$ / MHz")
plt.ylabel(r"$\eta$")
plt.tight_layout()
plt.show()
```



Simulate the spectrum

Static spectrum Create the spin systems.

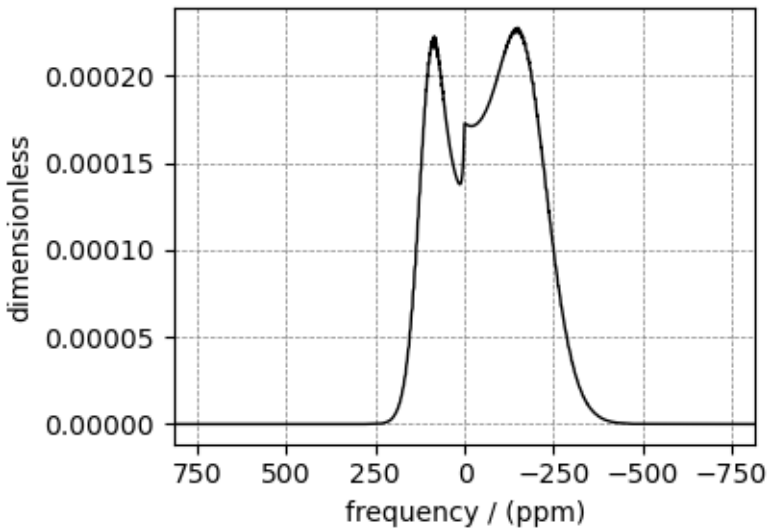
```
systems = single_site_system_generator(
    isotopes="71Ga", quadrupolar={"Cq": cq_dist * 1e6, "eta": e_dist}, abundance=amp
)
```

Create a simulator object and add the above system.

```
sim = Simulator()
sim.spin_systems = systems  # add the systems
sim.methods = [
    BlochDecayCentralTransitionSpectrum(
        channels=["71Ga"],
        magnetic_flux_density=9.4,  # in T
        spectral_dimensions=[{"count": 2048, "spectral_width": 2e5}],
    )
]  # add the method
sim.run()
```

The following is a static spectrum arising from an extended Czjzek distribution of the second-rank traceless EFG tensors.

```
plt.figure(figsize=(4.25, 3.0))
ax = plt.gca(projection="csdm")
ax.plot(sim.methods[0].simulation, color="black", linewidth=1)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```

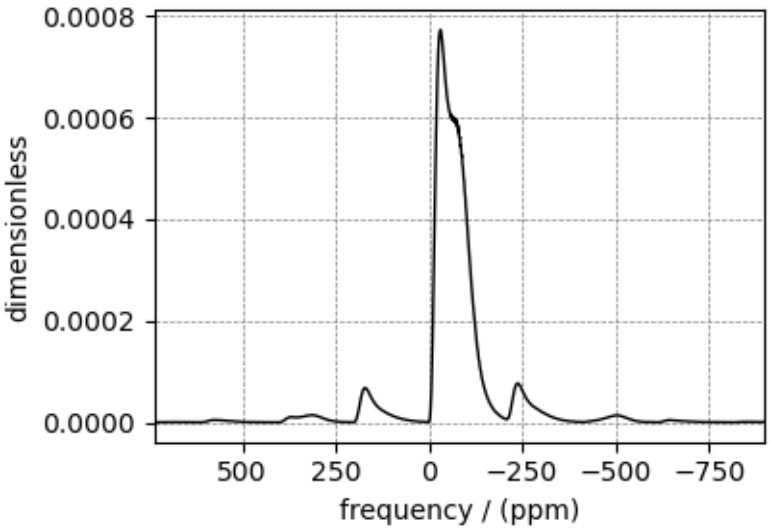


MAS spectrum

```
sim.methods = [
    BlochDecayCentralTransitionSpectrum(
        channels=["71Ga"],
        magnetic_flux_density=9.4,  # in T
        rotor_frequency=25000,  # in Hz
        spectral_dimensions=[
            {"count": 2048, "spectral_width": 2e5, "reference_offset": -1e4}
        ],
    )
]  # add the method
sim.config.number_of_sidebands = 16
sim.run()
```

The following is the MAS spectrum arising from an extended Czjzek distribution of the second-rank traceless EFG tensors.

```
plt.figure(figsize=(4.25, 3.0))
ax = plt.gca(projection="csdm")
ax.plot(sim.methods[0].simulation, color="black", linewidth=1)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 8.447 seconds)

4.1.3 2D NMR simulation (Crystalline solids)

The following examples are the NMR spectrum simulation for crystalline solids. The examples include the illustrations for the following methods:

- Triple-quantum variable-angle spinning (3Q-MAS) using the specialized *ThreeQ_VAS()* (page 182) method.
- Satellite-transition variable-angle spinning (ST-MAS) using the specialized *ST1_VAS()* (page 186) method.
- Switched angle spinning (SAS) using the generic *Method2D()* (page 180) method.
- Double hop Dynamic angle spinning (DAS) using the generic *Method2D()* (page 180) method.
- Correlation of anisotropies separated through echo refocusing (COASTER) using the generic *Method2D()* (page 180) method.
- Phase adjusted spinning sidebands (PASS) and Quadrupolar magic-angle turning (QMAT) using the specialized *SSB2D()* (page 188) method.

RbNO3, 87Rb (I=3/2) 3QMAS

87Rb (I=3/2) triple-quantum magic-angle spinning (3Q-MAS) simulation.

The following is an example of the 3QMAS simulation of $RbNO_3$, which has three distinct ^{87}Rb sites. The ^{87}Rb tensor parameters were obtained from Massiot *et. al.*[1]. In this simulation, a Gaussian broadening is applied to the spectrum as a post-simulation step.

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import mrsimulator.signal_processing as sp
import mrsimulator.signal_processing.apodization as apo
from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.methods import ThreeQ_VAS

# global plot configuration
```

(continues on next page)

[1] Massiot, D., Touzoa, B., Trumeaua, D., Coutures, J.P., Virlet, J., Florian, P., Grandinetti, P.J. Two-dimensional magic-angle spinning isotropic reconstruction sequences for quadrupolar nuclei, ssnmr, (1996), 6, 1, 73-83. DOI: 10.1016/0926-2040(95)01210-9

(continued from previous page)

```
font = {"size": 9}
mpl.rc("font", **font)
mpl.rcParams["figure.figsize"] = [4.25, 3.0]
```

Generate the site and spin system objects.

```
Rb87_1 = Site(
    isotope="87Rb",
    isotropic_chemical_shift=-27.4,  # in ppm
    quadrupolar={"Cq": 1.68e6, "eta": 0.2},  # Cq is in Hz
)
Rb87_2 = Site(
    isotope="87Rb",
    isotropic_chemical_shift=-28.5,  # in ppm
    quadrupolar={"Cq": 1.94e6, "eta": 1.0},  # Cq is in Hz
)
Rb87_3 = Site(
    isotope="87Rb",
    isotropic_chemical_shift=-31.3,  # in ppm
    quadrupolar={"Cq": 1.72e6, "eta": 0.5},  # Cq is in Hz
)

sites = [Rb87_1, Rb87_2, Rb87_3]  # all sites
spin_systems = [SpinSystem(sites=[s]) for s in sites]
```

Select a Triple Quantum variable-angle spinning method. You may optionally provide a *rotor_angle* to the method. The default *rotor_angle* is the magic-angle.

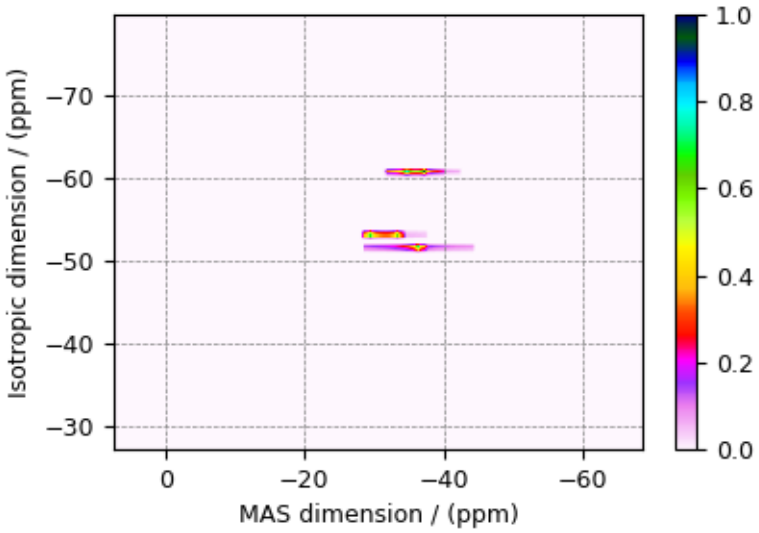
```
method = ThreeQ_VAS(
    channels=["87Rb"],
    magnetic_flux_density=9.4,  # in T
    spectral_dimensions=[
        {
            "count": 128,
            "spectral_width": 7e3,  # in Hz
            "reference_offset": -7e3,  # in Hz
            "label": "Isotropic dimension",
        },
        {
            "count": 256,
            "spectral_width": 1e4,  # in Hz
            "reference_offset": -4e3,  # in Hz
            "label": "MAS dimension",
        },
    ],
)
```

Create the Simulator object, add the method and spin system objects, and run the simulation.

```
sim = Simulator()
sim.spin_systems = spin_systems  # add the spin systems
sim.methods = [method]  # add the method.
sim.run()
```

The plot of the simulation.

```
data = sim.methods[0].simulation
ax = plt.gca(projection="csdm")
cb = ax.imshow(data / data.max(), aspect="auto", cmap="gist_ncar_r")
plt.colorbar(cb)
ax.invert_xaxis()
ax.invert_yaxis()
plt.tight_layout()
plt.show()
```

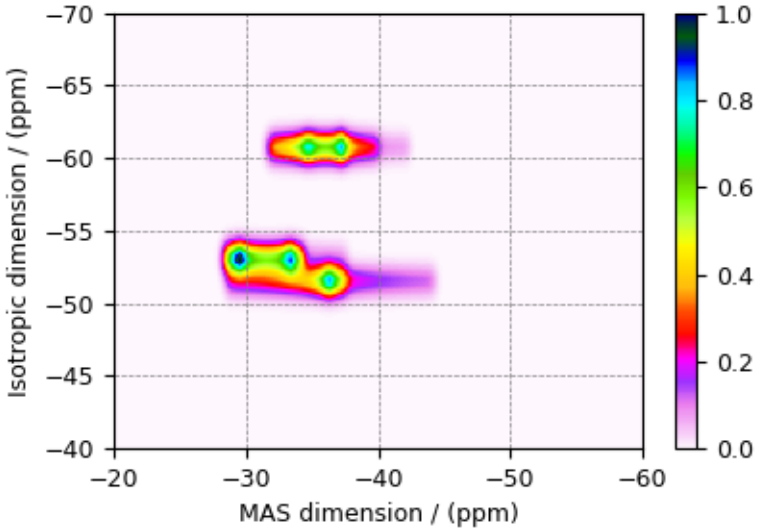


Add post-simulation signal processing.

```
processor = sp.SignalProcessor(
    operations=[
        # Gaussian convolution along both dimensions.
        sp.IFFT(dim_index=(0, 1)),
        apo.Gaussian(FWHM="0.08 kHz", dim_index=0),
        apo.Gaussian(FWHM="0.22 kHz", dim_index=1),
        sp.FFT(dim_index=(0, 1)),
    ]
)
processed_data = processor.apply_operations(data=sim.methods[0].simulation)
processed_data /= processed_data.max()
```

The plot of the simulation after signal processing.

```
ax = plt.subplot(projection="csdm")
cb = ax.imshow(processed_data.real, cmap="gist_ncar_r", aspect="auto")
plt.colorbar(cb)
ax.set_ylim(-40, -70)
ax.set_xlim(-20, -60)
plt.tight_layout()
plt.show()
```



See also:

Simulating site disorder (crystalline) (page 105) for RbNO3.

Total running time of the script: (0 minutes 0.522 seconds)

Albite, 27Al (I=5/2) 3QMAS

27Al (I=5/2) triple-quantum magic-angle spinning (3Q-MAS) simulation.

The following is an example of ^{27}Al 3QMAS simulation of albite $NaSi_3AlO_8$. The ^{27}Al tensor parameters were obtained from Massiot *et. al.*[1].

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import mrsimulator.signal_processing as sp
import mrsimulator.signal_processing.apodization as apo
from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.methods import ThreeQ_VAS

# global plot configuration
font = {"size": 9}
mpl.rc("font", **font)
mpl.rcParams["figure.figsize"] = [4.25, 3.0]
```

Generate the site and spin system objects.

```
site = Site(
    isotope="27Al",
    isotropic_chemical_shift=64.7,  # in ppm
    quadrupolar={"Cq": 3.25e6, "eta": 0.68},  # Cq is in Hz
)

spin_systems = [SpinSystem(sites=[site])]
```

[1] Massiot, D., Touzoa, B., Trumeaua, D., Coutures, J.P., Virlet, J., Florian, P., Grandinetti, P.J. Two-dimensional magic-angle spinning isotropic reconstruction sequences for quadrupolar nuclei, ssnmr, (1996), 6, 1, 73-83. DOI: 10.1016/0926-2040(95)01210-9

Select a Triple Quantum variable-angle spinning method. You may optionally provide a *rotor_angle* to the method. The default *rotor_angle* is the magic-angle.

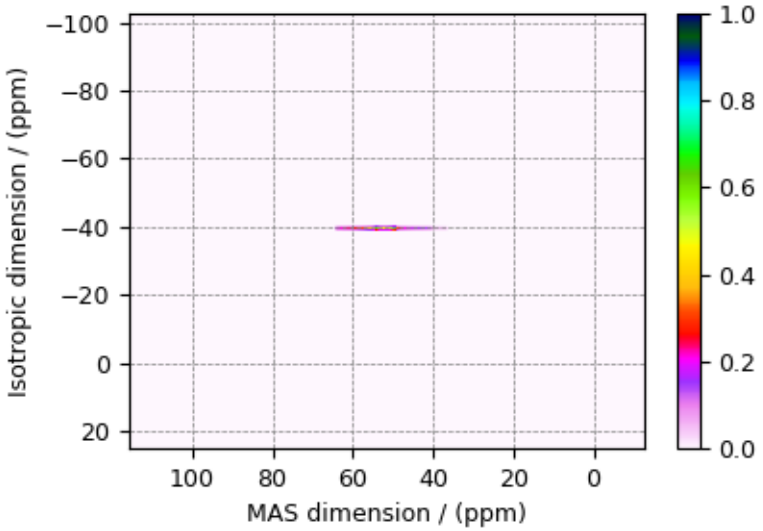
```
method = ThreeQ_VAS(
    channels=["27Al"],
    magnetic_flux_density=7,  # in T
    spectral_dimensions=[
        {
            "count": 256,
            "spectral_width": 1e4,  # in Hz
            "reference_offset": -3e3,  # in Hz
            "label": "Isotropic dimension",
        },
        {
            "count": 512,
            "spectral_width": 1e4,  # in Hz
            "reference_offset": 4e3,  # in Hz
            "label": "MAS dimension",
        },
    ],
)
```

Create the Simulator object, add the method and spin system objects, and run the simulation.

```
sim = Simulator()
sim.spin_systems = spin_systems  # add the spin systems
sim.methods = [method]  # add the method.
sim.run()
```

The plot of the simulation.

```
data = sim.methods[0].simulation
ax = plt.subplot(projection="csdm")
cb = ax.imshow(data / data.max(), aspect="auto", cmap="gist_ncar_r")
plt.colorbar(cb)
ax.invert_xaxis()
ax.invert_yaxis()
plt.tight_layout()
plt.show()
```

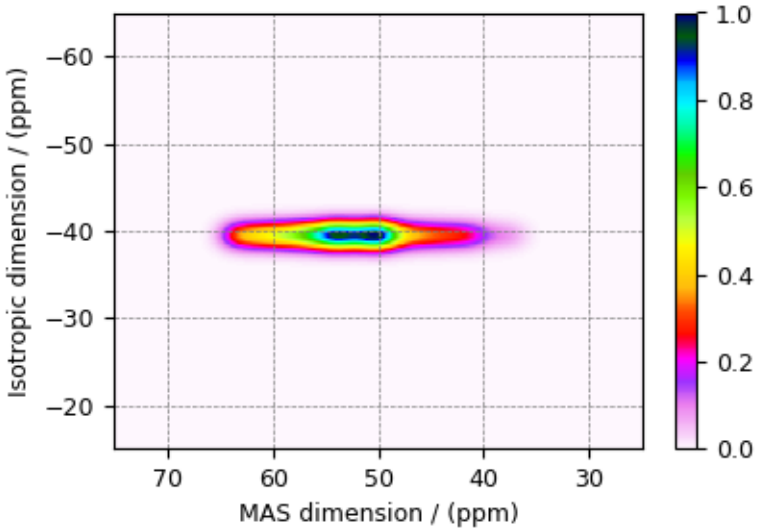


Add post-simulation signal processing.

```
processor = sp.SignalProcessor(
    operations=[
        # Gaussian convolution along both dimensions.
        sp.IFFT(dim_index=(0, 1)),
        apo.Gaussian(FWHM="0.2 kHz", dim_index=0),
        apo.Gaussian(FWHM="0.2 kHz", dim_index=1),
        sp.FFT(dim_index=(0, 1)),
    ]
)
processed_data = processor.apply_operations(data=sim.methods[0].simulation)
processed_data /= processed_data.max()
```

The plot of the simulation after signal processing.

```
ax = plt.subplot(projection="csdm")
cb = ax.imshow(processed_data.real, cmap="gist_ncar_r", aspect="auto")
plt.colorbar(cb)
ax.set_xlim(75, 25)
ax.set_ylim(-15, -65)
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.535 seconds)

RbNO3, 87Rb (I=3/2) STMAS

87Rb (I=3/2) staellite-transition off magic-angle spinning simulation.

The following is an example of the STMAS simulation of $RbNO_3$. The ^{87}Rb tensor parameters were obtained from Massiot *et. al.*[1].

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import mrsimulator.signal_processing as sp
import mrsimulator.signal_processing.apodization as apo
from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.methods import ST1_VAS

# global plot configuration
font = {"size": 9}
mpl.rc("font", **font)
mpl.rcParams["figure.figsize"] = [4.25, 3.0]
```

Generate the site and spin system objects.

```
Rb87_1 = Site(
    isotope="87Rb",
    isotropic_chemical_shift=-27.4,  # in ppm
    quadrupolar={"Cq": 1.68e6, "eta": 0.2},  # Cq is in Hz
)
Rb87_2 = Site(
    isotope="87Rb",
    isotropic_chemical_shift=-28.5,  # in ppm
    quadrupolar={"Cq": 1.94e6, "eta": 1.0},  # Cq is in Hz
)
Rb87_3 = Site(
    isotope="87Rb",
```

(continues on next page)

[1] Massiot, D., Touzoa, B., Trumeaua, D., Coutures, J.P., Virlet, J., Florian, P., Grandinetti, P.J. Two-dimensional magic-angle spinning isotropic reconstruction sequences for quadrupolar nuclei, ssnmr, (1996), 6, 1, 73-83. DOI: 10.1016/0926-2040(95)01210-9

(continued from previous page)

```
    isotropic_chemical_shift=-31.3,  # in ppm
    quadrupolar={"Cq": 1.72e6, "eta": 0.5},  # Cq is in Hz
)

sites = [Rb87_1, Rb87_2, Rb87_3]  # all sites
spin_systems = [SpinSystem(sites=[s]) for s in sites]
```

Step 2: Select a satellite-transition variable-angle spinning method. The following *ST1_VAS* method correlates the frequencies from the two inner-satellite transitions to the central transition. Note, STMAS measurements are highly suspectable to rotor angle mismatch. In the following, we show two methods, first set to magic-angle and the second deliberately miss-sets by approximately 0.0059 degrees.

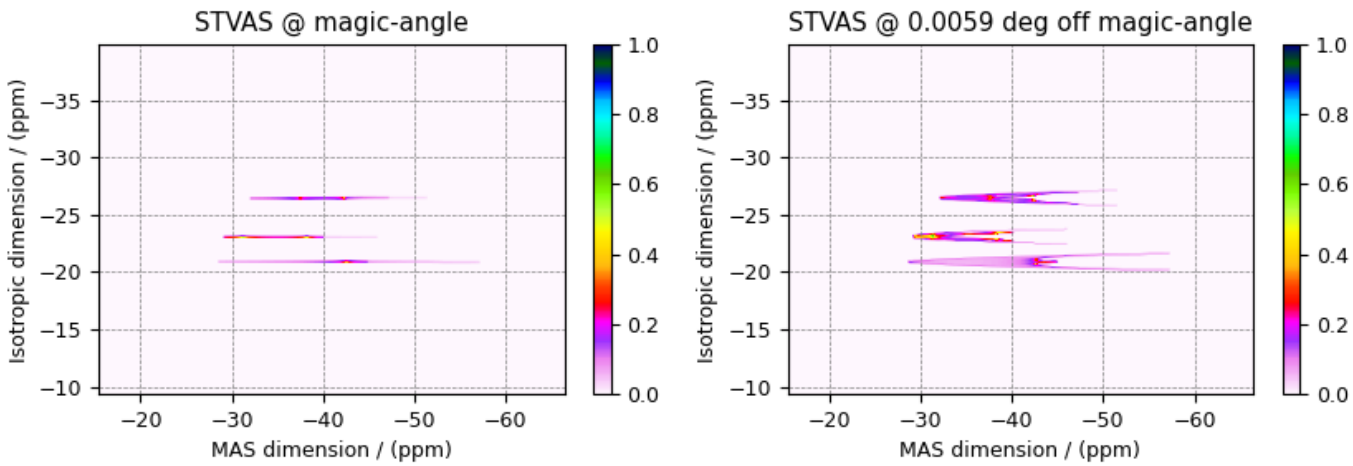
```
angles = [54.7359, 54.73]
method = []
for angle in angles:
    method.append(
        ST1_VAS(
            channels=["87Rb"],
            magnetic_flux_density=7,  # in T
            rotor_angle=angle * 3.14159 / 180,  # in rad (magic angle)
            spectral_dimensions=[
                {
                    "count": 256,
                    "spectral_width": 3e3,  # in Hz
                    "reference_offset": -2.4e3,  # in Hz
                    "label": "Isotropic dimension",
                },
                {
                    "count": 512,
                    "spectral_width": 5e3,  # in Hz
                    "reference_offset": -4e3,  # in Hz
                    "label": "MAS dimension",
                },
            ],
        )
    )
```

Create the Simulator object, add the method and spin system objects, and run the simulation.

```
sim = Simulator()
sim.spin_systems = spin_systems  # add the spin systems
sim.methods = method  # add the methods.
sim.run()
```

The plot of the simulation.

```
data = [sim.methods[0].simulation, sim.methods[1].simulation]
fig, ax = plt.subplots(1, 2, figsize=(8.5, 3), subplot_kw={"projection": "csdm"})

titles = ["STVAS @ magic-angle", "STVAS @ 0.0059 deg off magic-angle"]
for i, item in enumerate(data):
    cb1 = ax[i].imshow(item / item.max(), aspect="auto", cmap="gist_ncar_r")
    ax[i].set_title(titles[i])
    plt.colorbar(cb1, ax=ax[i])
    ax[i].invert_xaxis()
    ax[i].invert_yaxis()
plt.tight_layout()
plt.show()
```

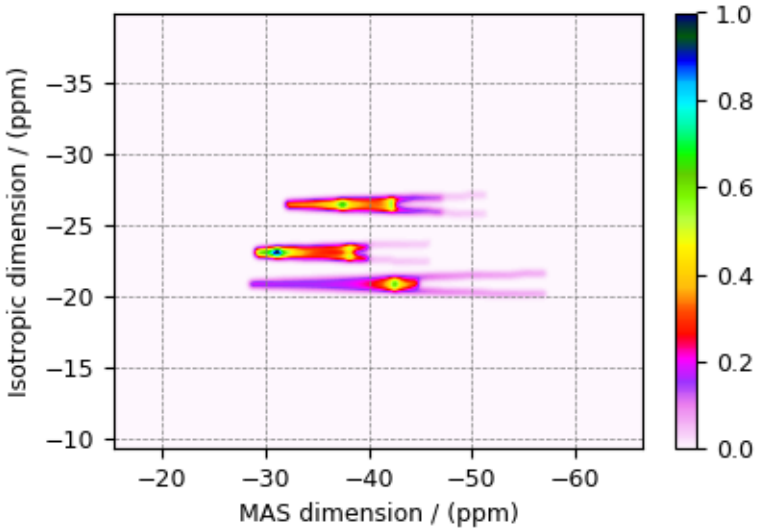


Add post-simulation signal processing.

```
processor = sp.SignalProcessor(
    operations=[
        # Gaussian convolution along both dimensions.
        sp.IFFT(dim_index=(0, 1)),
        apo.Gaussian(FWHM="50 Hz", dim_index=0),
        apo.Gaussian(FWHM="50 Hz", dim_index=1),
        sp.FFT(dim_index=(0, 1)),
    ]
)
processed_data = []
for item in data:
    processed_data.append(processor.apply_operations(data=item))
    processed_data[-1] /= processed_data[-1].max()
```

The plot of the simulation after signal processing.

```
ax = plt.subplot(projection="csdm")
cb = ax.imshow(processed_data[1].real, cmap="gist_ncar_r", aspect="auto")
plt.colorbar(cb)
ax.invert_xaxis()
ax.invert_yaxis()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.913 seconds)

Rb2SO4, 87Rb (I=3/2) SAS

87Rb (I=3/2) Switched-angle spinning (SAS) simulation.

The following is an example of switched-angle spinning (SAS) simulation of Rb_2SO_4, which has two distinct rubidium sites. The NMR tensor parameters for these sites are taken from Shore *et. al.*[1].

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import mrsimulator.signal_processing as sp
import mrsimulator.signal_processing.apodization as apo
from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.methods import Method2D

# global plot configuration
font = {"size": 9}
mpl.rc("font", **font)
mpl.rcParams["figure.figsize"] = [4.25, 3.0]
```

Generate the site and spin system objects.

```
sites = [
    Site(
        isotope="87Rb",
        isotropic_chemical_shift=16,  # in ppm
        quadrupolar={"Cq": 5.3e6, "eta": 0.1},  # Cq in Hz
    ),
    Site(
        isotope="87Rb",
        isotropic_chemical_shift=40,  # in ppm
        quadrupolar={"Cq": 2.6e6, "eta": 1.0},  # Cq in Hz
    ),
```

(continues on next page)

[1] Shore, J.S., Wang, S.H., Taylor, R.E., Bell, A.T., Pines, A. Determination of quadrupolar and chemical shielding tensors using solid-state two-dimensional NMR spectroscopy, J. Chem. Phys. (1996) 105 21, 9412. DOI: 10.1063/1.472776

(continued from previous page)

```
]
spin_systems = [SpinSystem(sites=[s]) for s in sites]
```

Use the generic 2D method, *Method2D*, to simulate a SAS spectrum by customizing the method parameters, as shown below. Note, the Method2D method simulates an infinite spinning speed spectrum.

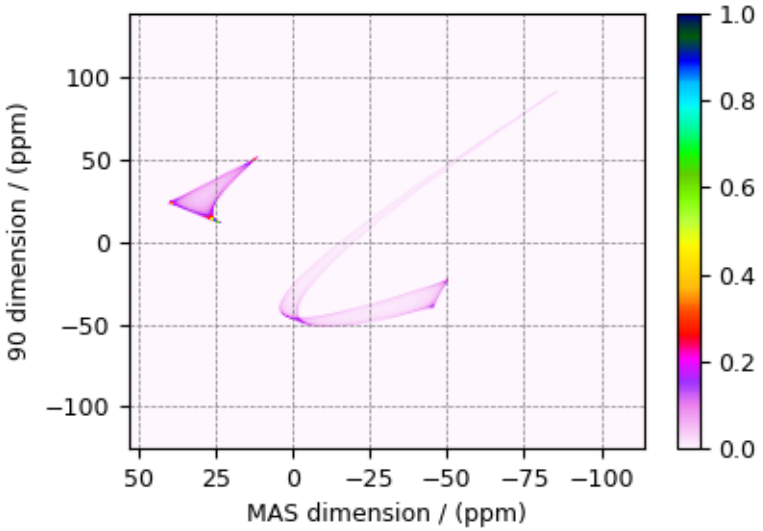
```
sas = Method2D(
    channels=["87Rb"],
    magnetic_flux_density=9.4,  # in T
    spectral_dimensions=[
        {
            "count": 256,
            "spectral_width": 3.5e4,  # in Hz
            "reference_offset": 1e3,  # in Hz
            "label": "90 dimension",
            "events": [{"rotor_angle": 90 * 3.14159 / 180}],  # in radians
        },
        {
            "count": 256,
            "spectral_width": 22e3,  # in Hz
            "reference_offset": -4e3,  # in Hz
            "label": "MAS dimension",
            "events": [{"rotor_angle": 54.74 * 3.14159 / 180}],  # in radians
        },
    ],
)
```

Create the Simulator object, add the method and spin system objects, and run the simulation.

```
sim = Simulator()
sim.spin_systems = spin_systems  # add the spin systems
sim.methods = [sas]  # add the method.
sim.run()
```

The plot of the simulation.

```
data = sim.methods[0].simulation
ax = plt.subplot(projection="csdm")
cb = ax.imshow(data / data.max(), aspect="auto", cmap="gist_ncar_r")
plt.colorbar(cb)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```

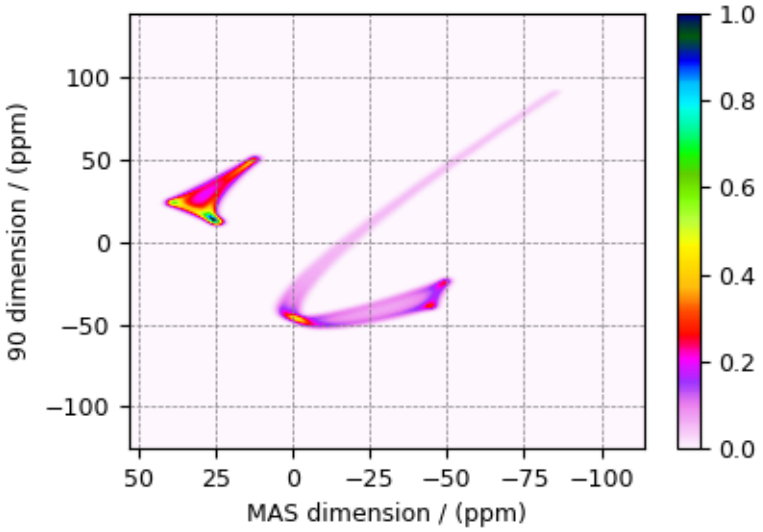


Add post-simulation signal processing.

```
processor = sp.SignalProcessor(
    operations=[
        # Gaussian convolution along both dimensions.
        sp.IFFT(dim_index=(0, 1)),
        apo.Gaussian(FWHM="0.4 kHz", dim_index=0),
        apo.Gaussian(FWHM="0.4 kHz", dim_index=1),
        sp.FFT(dim_index=(0, 1)),
    ]
)
processed_data = processor.apply_operations(data=data)
processed_data /= processed_data.max()
```

The plot of the simulation after signal processing.

```
ax = plt.subplot(projection="csdm")
cb = ax.imshow(processed_data.real, cmap="gist_ncar_r", aspect="auto")
plt.colorbar(cb)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.511 seconds)

Rb2CrO4, 87Rb (I=3/2) SAS

87Rb (I=3/2) Switched-angle spinning (SAS) simulation.

The following is a switched-angle spinning (SAS) simulation of Rb_2CrO_4. While Rb_2CrO_4 has two rubidium sites, the site with the smaller quadrupolar interaction was selectively observed and reported by Shore *et. al.*[1]. The following is the simulation based on the published tensor parameters.

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import mrsimulator.signal_processing as sp
import mrsimulator.signal_processing.apodization as apo
from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.methods import Method2D

# global plot configuration
font = {"size": 9}
mpl.rc("font", **font)
mpl.rcParams["figure.figsize"] = [4.25, 3.0]
```

Generate the site and spin system objects.

```
site = Site(
    isotope="87Rb",
    isotropic_chemical_shift=-7,  # in ppm
    shielding_symmetric={"zeta": 110, "eta": 0},
    quadrupolar={
        "Cq": 3.5e6,  # in Hz
        "eta": 0.3,
        "alpha": 0,  # in rads
        "beta": 70 * 3.14159 / 180,  # in rads
        "gamma": 0,  # in rads
```

(continues on next page)

[1] Shore, J.S., Wang, S.H., Taylor, R.E., Bell, A.T., Pines, A. Determination of quadrupolar and chemical shielding tensors using solid-state two-dimensional NMR spectroscopy, J. Chem. Phys. (1996) 105 21, 9412. DOI: 10.1063/1.472776

(continued from previous page)

```
    },
)
spin_system = SpinSystem(sites=[site])
```

Use the generic 2D method, *Method2D*, to simulate a SAS spectrum by customizing the method parameters, as shown below. Note, the Method2D method simulates an infinite spinning speed spectrum.

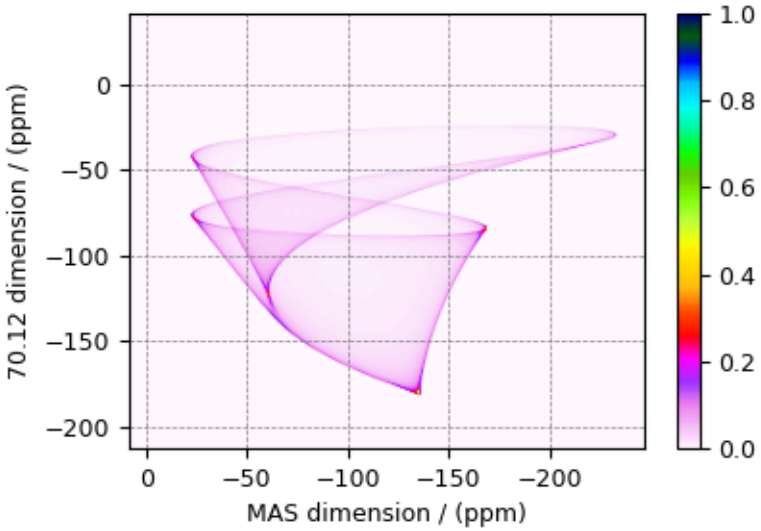
```
sas = Method2D(
    channels=["87Rb"],
    magnetic_flux_density=4.2,  # in T
    spectral_dimensions=[
        {
            "count": 256,
            "spectral_width": 1.5e4,  # in Hz
            "reference_offset": -5e3,  # in Hz
            "label": "70.12 dimension",
            "events": [{"rotor_angle": 70.12 * 3.14159 / 180}],  # in radians
        },
        {
            "count": 512,
            "spectral_width": 15e3,  # in Hz
            "reference_offset": -7e3,  # in Hz
            "label": "MAS dimension",
            "events": [{"rotor_angle": 54.74 * 3.14159 / 180}],  # in radians
        },
    ],
)
```

Create the Simulator object, add the method and spin system objects, and run the simulation.

```
sim = Simulator()
sim.spin_systems = [spin_system]  # add the spin systems
sim.methods = [sas]  # add the method.

# Configure the simulator object. For non-coincidental tensors, set the value of the
# `integration_volume` attribute to `hemisphere`.
sim.config.integration_volume = "hemisphere"
sim.run()
```

The plot of the simulation.

```
data = sim.methods[0].simulation
ax = plt.subplot(projection="csdm")
cb = ax.imshow(data / data.max(), aspect="auto", cmap="gist_ncar_r")
plt.colorbar(cb)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```

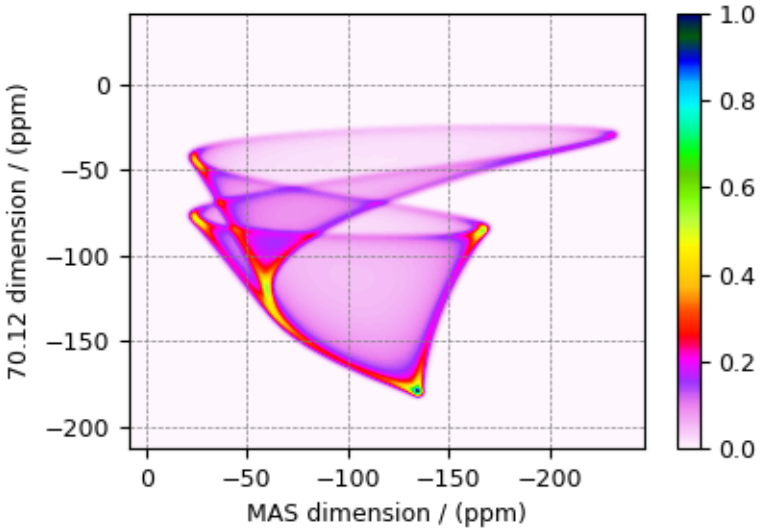


Add post-simulation signal processing.

```
processor = sp.SignalProcessor(
    operations=[
        # Gaussian convolution along both dimensions.
        sp.IFFT(dim_index=(0, 1)),
        apo.Gaussian(FWHM="0.2 kHz", dim_index=0),
        apo.Gaussian(FWHM="0.2 kHz", dim_index=1),
        sp.FFT(dim_index=(0, 1)),
    ]
)
processed_data = processor.apply_operations(data=data)
processed_data /= processed_data.max()
```

The plot of the simulation after signal processing.

```
ax = plt.subplot(projection="csdm")
cb = ax.imshow(processed_data.real, cmap="gist_ncar_r", aspect="auto")
plt.colorbar(cb)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.585 seconds)

Coesite, 17O (I=5/2) DAS

17O (I=5/2) Dynamic-angle spinning (DAS) simulation.

The following is a dynamic angle spinning (DAS) simulation of Coesite. Coesite has five crystallographic ^{17}O sites. In the following, we use the ^{17}O EFG tensor information from Grandinetti *et. al.*[1]

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import mrsimulator.signal_processing as sp
import mrsimulator.signal_processing.apodization as apo
from mrsimulator import Simulator
from mrsimulator.methods import Method2D

# global plot configuration
font = {"size": 9}
mpl.rc("font", **font)
mpl.rcParams["figure.figsize"] = [4.25, 3.0]
```

Create the Simulator object and load the spin systems database or url address.

```
sim = Simulator()

# load the spin systems from url.
filename = "https://sandbox.zenodo.org/record/687656/files/coesite.mrsys"
sim.load_spin_systems(filename)
```

Use the generic 2D method, *Method2D*, to simulate a DAS spectrum by customizing the method parameters, as shown below. Note, the Method2D method simulates an infinite spinning speed spectrum.

```
das = Method2D(
    channels=["17O"],
```

(continues on next page)

[1] Grandinetti, P. J., Baltisberger, J. H., Farnan, I., Stebbins, J. F., Werner, U. and Pines, A. Solid-State ^{17}O Magic-Angle and Dynamic-Angle Spinning NMR Study of the SiO_2 Polymorph Coesite, J. Phys. Chem. 1995, 99, 32, 12341-12348. DOI: 10.1021/j100032a045

(continued from previous page)

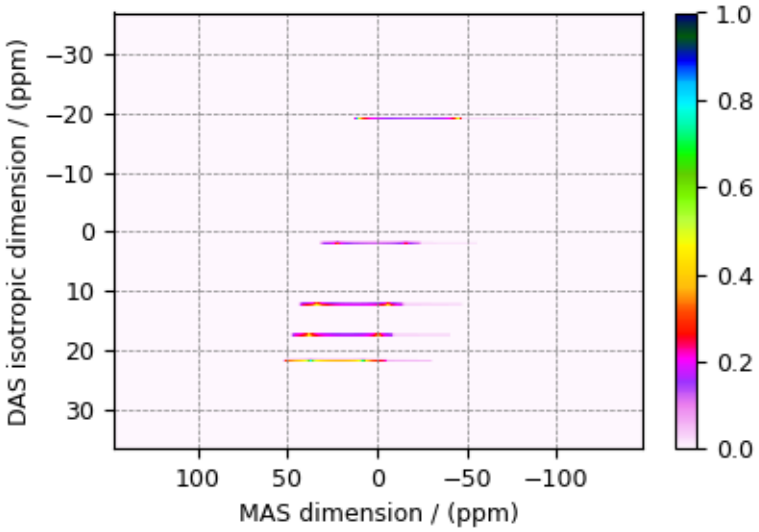
```
    magnetic_flux_density=11.7,  # in T
    spectral_dimensions=[
        {
            "count": 256,
            "spectral_width": 5e3,  # in Hz
            "reference_offset": 0,  # in Hz
            "label": "DAS isotropic dimension",
            "events": [
                {"fraction": 0.5, "rotor_angle": 37.38 * 3.14159 / 180},
                {"fraction": 0.5, "rotor_angle": 79.19 * 3.14159 / 180},
            ],
        },
        # The last spectral dimension block is the direct-dimension
        {
            "count": 256,
            "spectral_width": 2e4,  # in Hz
            "reference_offset": 0,  # in Hz
            "label": "MAS dimension",
            "events": [{"rotor_angle": 54.735 * 3.14159 / 180}],
        },
    ],
)
sim.methods = [das]  # add the method.
```

Run the simulation

```
sim.run()
```

The plot of the simulation.

```
data = sim.methods[0].simulation
ax = plt.subplot(projection="csdm")
cb = ax.imshow(data / data.max(), aspect="auto", cmap="gist_ncar_r")
plt.colorbar(cb)
ax.invert_xaxis()
ax.invert_yaxis()
plt.tight_layout()
plt.show()
```

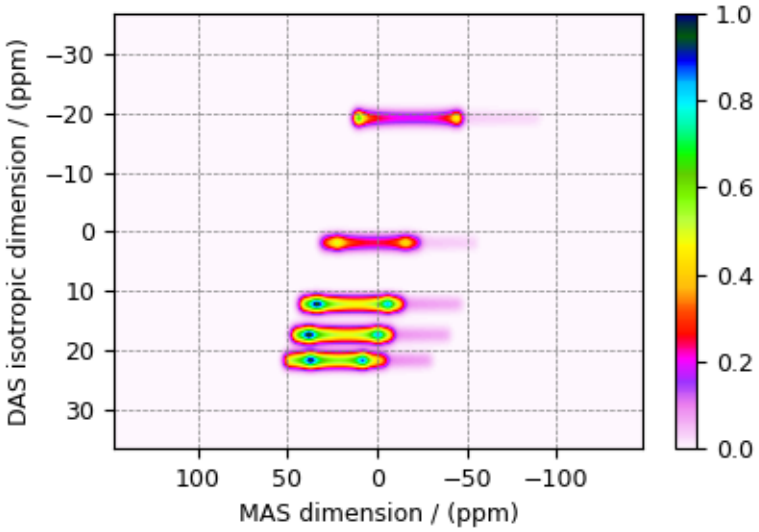


Add post-simulation signal processing.

```
processor = sp.SignalProcessor(
    operations=[
        # Gaussian convolution along both dimensions.
        sp.IFFT(dim_index=(0, 1)),
        apo.Gaussian(FWHM="0.3 kHz", dim_index=0),
        apo.Gaussian(FWHM="0.15 kHz", dim_index=1),
        sp.FFT(dim_index=(0, 1)),
    ]
)
processed_data = processor.apply_operations(data=data)
processed_data /= processed_data.max()
```

The plot of the simulation after signal processing.

```
ax = plt.subplot(projection="csdm")
cb = ax.imshow(processed_data.real, cmap="gist_ncar_r", aspect="auto")
plt.colorbar(cb)
ax.invert_xaxis()
ax.invert_yaxis()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 1.269 seconds)

Rb2CrO4, 87Rb (I=3/2) COASTER

87Rb (I=3/2) Correlation of anisotropies separated through echo refocusing (COASTER) simulation.

The following is a correlation of anisotropies separated through echo refocusing (COASTER) simulation of Rb_2CrO_4. The Rb site with the smaller quadrupolar interaction is selectively observed and reported by Ash *et. al.*[1]. The following is the simulation based on the published tensor parameters.

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import mrsimulator.signal_processing as sp
import mrsimulator.signal_processing.apodization as apo
from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.methods import Method2D

# global plot configuration
font = {"size": 9}
mpl.rc("font", **font)
mpl.rcParams["figure.figsize"] = [4.25, 3.0]
```

Generate the site and spin system objects.

```
site = Site(
    isotope="87Rb",
    isotropic_chemical_shift=-9,  # in ppm
    shielding_symmetric={"zeta": 110, "eta": 0},
    quadrupolar={
        "Cq": 3.5e6,  # in Hz
        "eta": 0.36,
        "alpha": 0,  # in rads
        "beta": 70 * 3.14159 / 180,  # in rads
        "gamma": 0,  # in rads
```

(continues on next page)

[1] Jason T. Ash, Nicole M. Trease, and Philip J. Grandinetti. Separating Chemical Shift and Quadrupolar Anisotropies via Multiple-Quantum NMR Spectroscopy, J. Am. Chem. Soc. (2008) 130, 10858-10859. DOI: 10.1021/ja802865x

(continued from previous page)

```
    },
)
spin_system = SpinSystem(sites=[site])
```

Use the generic 2D method, *Method2D*, to simulate a COASTER spectrum by customizing the method parameters, as shown below. Note, the Method2D method simulates an infinite spinning speed spectrum.

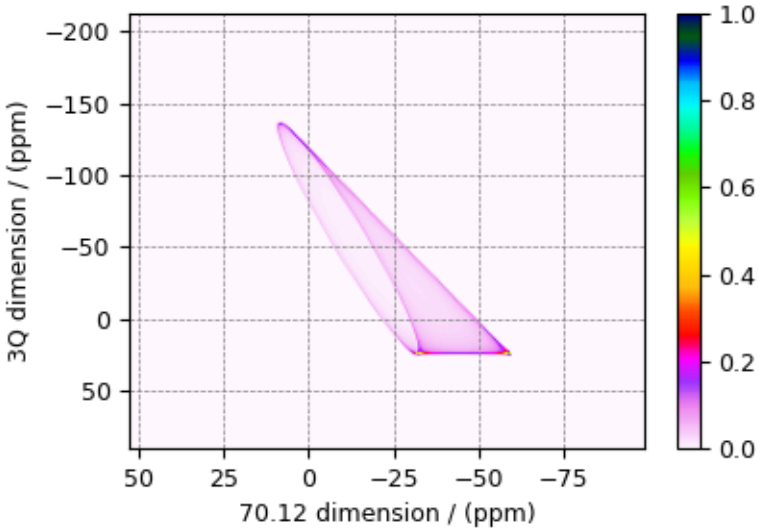
```
coaster = Method2D(
    channels=["87Rb"],
    magnetic_flux_density=9.4,  # in T
    rotor_angle=70.12 * 3.14159 / 180,  # in rads
    spectral_dimensions=[
        {
            "count": 256,
            "spectral_width": 4e4,  # in Hz
            "reference_offset": -8e3,  # in Hz
            "label": "3Q dimension",
            "events": [{"transition_query": {"P": [3], "D": [0]}}],
        },
        # The last spectral dimension block is the direct-dimension
        {
            "count": 256,
            "spectral_width": 2e4,  # in Hz
            "reference_offset": -3e3,  # in Hz
            "label": "70.12 dimension",
            "events": [{"transition_query": {"P": [-1], "D": [0]}}],
        },
    ],
)
```

Create the Simulator object, add the method and spin system objects, and run the simulation.

```
sim = Simulator()
sim.spin_systems = [spin_system]  # add the spin systems
sim.methods = [coaster]  # add the method.

# configure the simulator object. For non-coincidental tensors, set the value of the
# `integration_volume` attribute to `hemisphere`.
sim.config.integration_volume = "hemisphere"
sim.run()
```

The plot of the simulation.

```
data = sim.methods[0].simulation
ax = plt.subplot(projection="csdm")
cb = ax.imshow(data / data.max(), aspect="auto", cmap="gist_ncar_r")
plt.colorbar(cb)
ax.invert_xaxis()
ax.invert_yaxis()
plt.tight_layout()
plt.show()
```

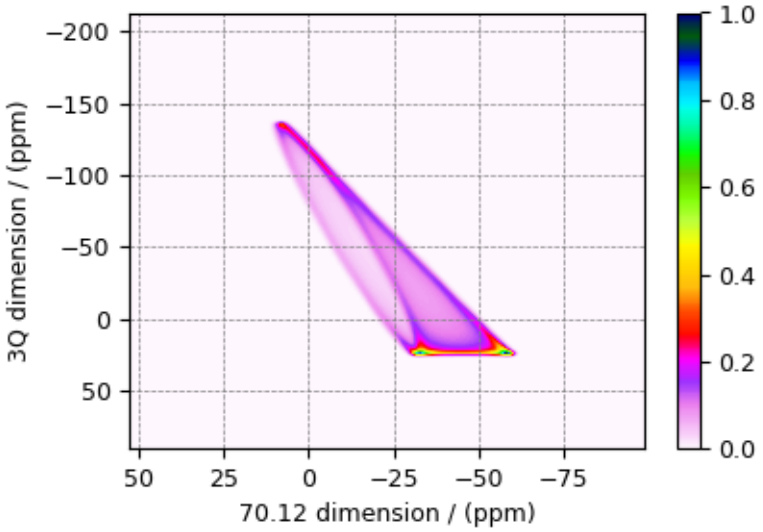


Add post-simulation signal processing.

```
processor = sp.SignalProcessor(
    operations=[
        # Gaussian convolution along both dimensions.
        sp.IFFT(dim_index=(0, 1)),
        apo.Gaussian(FWHM="0.3 kHz", dim_index=0),
        apo.Gaussian(FWHM="0.3 kHz", dim_index=1),
        sp.FFT(dim_index=(0, 1)),
    ]
)
processed_data = processor.apply_operations(data=data)
processed_data /= processed_data.max()
```

The plot of the simulation after signal processing.

```
ax = plt.subplot(projection="csdm")
cb = ax.imshow(processed_data.real, cmap="gist_ncar_r", aspect="auto")
plt.colorbar(cb)
ax.invert_xaxis()
ax.invert_yaxis()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.595 seconds)

Itraconazole, 13C (I=1/2) PASS

13C (I=1/2) 2D Phase-adjusted spinning sideband (PASS) simulation.

The following is a simulation of a 2D PASS spectrum of itraconazole, a triazole containing drug prescribed for the prevention and treatment of fungal infection. The 2D PASS spectrum is a correlation of finite speed MAS to an infinite speed MAS spectrum. The parameters for the simulation are obtained from Dey *et. al.*[1].

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import mrsimulator.signal_processing as sp
import mrsimulator.signal_processing.apodization as apo
from mrsimulator import Simulator
from mrsimulator.methods import SSB2D

# global plot configuration
font = {"size": 9}
mpl.rc("font", **font)
mpl.rcParams["figure.figsize"] = [4.5, 3.0]
```

There are 41 ^{13}C single-site spin systems partially describing the NMR parameters of itraconazole. We will import the directly import the spin systems to the Simulator object using the *load_spin_systems* method.

```
sim = Simulator()

filename = "https://sandbox.zenodo.org/record/687656/files/itraconazole_13C.mrsys"
sim.load_spin_systems(filename)
```

Use the `SSB2D` method to simulate a PASS, MAT, QPASS, QMAT, or any equivalent sideband separation spectrum. Here, we use the method to generate a PASS spectrum.

[1] Dey, K .K, Gayen, S., Ghosh, M., Investigation of the Detailed Internal Structure and Dynamics of Itraconazole by Solid-State NMR Measurements, ACS Omega (2019) 4, 21627. DOI:10.1021/acsomega.9b03558

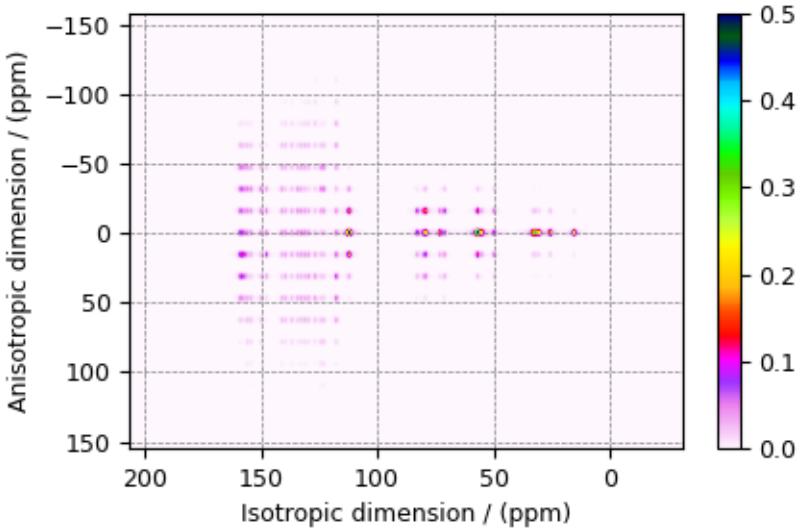
```
PASS = SSB2D(
    channels=["13C"],
    magnetic_flux_density=11.74,
    rotor_frequency=2000,
    spectral_dimensions=[
        {
            "count": 20 * 4,
            "spectral_width": 2000 * 20,  # value in Hz
            "label": "Anisotropic dimension",
        },
        {
            "count": 1024,
            "spectral_width": 3e4,  # value in Hz
            "reference_offset": 1.1e4,  # value in Hz
            "label": "Isotropic dimension",
        },
    ],
)
sim.methods = [PASS]  # add the method.

# For 2D spinning sideband simulation, set the number of spinning sidebands in the
# Simulator.config object to `spectral_width/rotor_frequency` along the sideband
# dimension.
sim.config.number_of_sidebands = 20

# run the simulation.
sim.run()
```

Apply post-simulation processing. Here, we apply a Lorentzian line broadening to the isotropic dimension.

```
data = sim.methods[0].simulation
processor = sp.SignalProcessor(
    operations=[
        sp.IFFT(dim_index=0),
        apo.Exponential(FWHM="100 Hz", dim_index=0),
        sp.FFT(dim_index=0),
    ]
)
processed_data = processor.apply_operations(data=data).real
processed_data /= processed_data.max()
```

The plot of the simulation.

```
ax = plt.subplot(projection="csdm")
cb = ax.imshow(processed_data, aspect="auto", cmap="gist_ncar_r", vmax=0.5)
plt.colorbar(cb)
ax.invert_xaxis()
ax.invert_yaxis()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 1.452 seconds)

Rb2SO4, 87Rb (I=3/2) QMAT

87Rb (I=3/2) Quadrupolar Magic-angle turning (QMAT) simulation.

The following is a simulation of the QMAT spectrum of Rb_2SiO_4. The 2D QMAT spectrum is a correlation of finite speed MAS to an infinite speed MAS spectrum. The parameters for the simulation are obtained from Walder *et. al.*[1].

```
import matplotlib as mpl
import matplotlib.pyplot as plt
from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.methods import SSB2D

# global plot configuration
font = {"size": 9}
mpl.rc("font", **font)
mpl.rcParams["figure.figsize"] = [4.5, 3.0]
```

Generate the site and spin system objects.

```
sites = [
    Site(
        isotope="87Rb",
        isotropic_chemical_shift=16,  # in ppm
        quadrupolar={"Cq": 5.3e6, "eta": 0.1},  # Cq in Hz
    ),
    Site(
        isotope="87Rb",
        isotropic_chemical_shift=40,  # in ppm
        quadrupolar={"Cq": 2.6e6, "eta": 1.0},  # Cq in Hz
    ),
]
spin_systems = [SpinSystem(sites=[s]) for s in sites]
```

[1] Walder, B. J., Dey, K .K, Kaseman, D. C., Baltisberger, J. H., and Philip J. Grandinetti. Sideband separation experiments in NMR with phase incremented echo train acquisition, J. Chem. Phys. (2013) 138, 174203. DOI:10.1063/1.4803142

Use the `SSB2D` method to simulate a PASS, MAT, QPASS, QMAT, or any equivalent sideband separation spectrum. Here, we use the method to generate a QMAT spectrum. The QMAT method is created from the `SSB2D` method in the same as a PASS or MAT method. The difference is that the observed channel is a half-integer quadrupolar spin instead of a spin I=1/2.

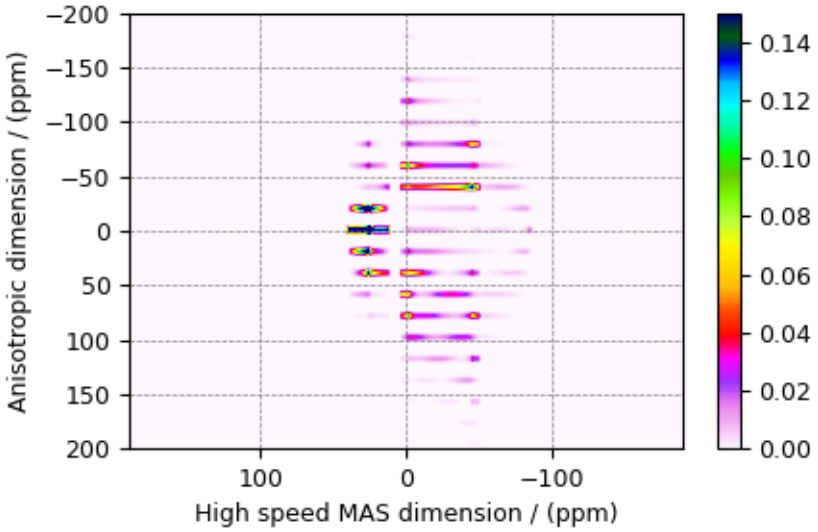
```
qmat = SSB2D(
    channels=["87Rb"],
    magnetic_flux_density=9.4,
    rotor_frequency=2604,
    spectral_dimensions=[
        {
            "count": 32 * 4,
            "spectral_width": 2604 * 32,  # in Hz
            "label": "Anisotropic dimension",
        },
        {
            "count": 512,
            "spectral_width": 50000,  # in Hz
            "label": "High speed MAS dimension",
        },
    ],
)
```

Create the Simulator object, add the method and spin system objects, and run the simulation.

```
sim = Simulator()
sim.spin_systems = spin_systems  # add the spin systems
sim.methods = [qmat]  # add the method.

# For 2D spinning sideband simulation, set the number of spinning sidebands in the
# Simulator.config object to `spectral_width/rotor_frequency` along the sideband
# dimension.
sim.config.number_of_sidebands = 32
sim.run()
```

The plot of the simulation.

```
data = sim.methods[0].simulation
ax = plt.subplot(projection="csdm")
cb = ax.imshow(data / data.max(), aspect="auto", cmap="gist_ncar_r", vmax=0.15)
plt.colorbar(cb)
ax.invert_xaxis()
ax.set_ylim(200, -200)
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.326 seconds)

Wollastonite, 29Si (I=1/2), MAF

29Si (I=1/2) magic angle flipping.

Wollastonite is a high-temperature calcium-silicate, β–$Ca_3Si_3O_9$, with three distinct ^{29}Si sites. The ^{29}Si tensor parameters were obtained from Hansen *et. al.*[1]

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import mrsimulator.signal_processing as sp
import mrsimulator.signal_processing.apodization as apo
from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.methods import Method2D

# global plot configuration
mpl.rcParams["figure.figsize"] = [4.5, 3.0]
```

Create the sites and spin systems

```
sites = [
    Site(
        isotope="29Si",
        isotropic_chemical_shift=-89.0,  # in ppm
        shielding_symmetric={"zeta": 59.8, "eta": 0.62},  # zeta in ppm
    ),
    Site(
        isotope="29Si",
        isotropic_chemical_shift=-89.5,  # in ppm
        shielding_symmetric={"zeta": 52.1, "eta": 0.68},  # zeta in ppm
    ),
    Site(
        isotope="29Si",
```

(continues on next page)

[1] Hansen, M. R., Jakobsen, H. J., Skibsted, J., ^{29}Si Chemical Shift Anisotropies in Calcium Silicates from High-Field ^{29}Si MAS NMR Spectroscopy, Inorg. Chem. 2003, 42, 7, 2368-2377. DOI: 10.1021/ic020647f

(continued from previous page)

```
        isotropic_chemical_shift=-87.8,  # in ppm
        shielding_symmetric={"zeta": 69.4, "eta": 0.60},  # zeta in ppm
    ),
]

spin_systems = [SpinSystem(sites=[s]) for s in sites]
```

Use the generic 2D method, *Method2D*, to simulate a MAF spectrum by customizing the method parameters, as shown below. Note, the Method2D method simulates an infinite spinning speed spectrum.

```
maf = Method2D(
    channels=["29Si"],
    magnetic_flux_density=14.1,  # in T
    spectral_dimensions=[
        {
            "count": 128,
            "spectral_width": 2e4,  # in Hz
            "label": "Anisotropic dimension",
            "events": [{"rotor_angle": 90 * 3.14159 / 180}],
        },
        {
            "count": 128,
            "spectral_width": 3e3,  # in Hz
            "reference_offset": -1.05e4,  # in Hz
            "label": "Isotropic dimension",
            "events": [{"rotor_angle": 54.735 * 3.14159 / 180}],
        },
    ],
    affine_matrix=[[1, -1], [0, 1]],
)
```

Create the Simulator object, add the method and spin system objects, and run the simulation.

```
sim = Simulator()
sim.spin_systems = spin_systems  # add the spin systems
sim.methods = [maf]  # add the method
sim.run()
```

Add post-simulation signal processing.

```
csdm_data = sim.methods[0].simulation
processor = sp.SignalProcessor(
    operations=[
        sp.IFFT(dim_index=(0, 1)),
        apo.Gaussian(FWHM="50 Hz", dim_index=0),
        apo.Gaussian(FWHM="50 Hz", dim_index=1),
        sp.FFT(dim_index=(0, 1)),
    ]
)
processed_data = processor.apply_operations(data=csdm_data).real
processed_data /= processed_data.max()
```

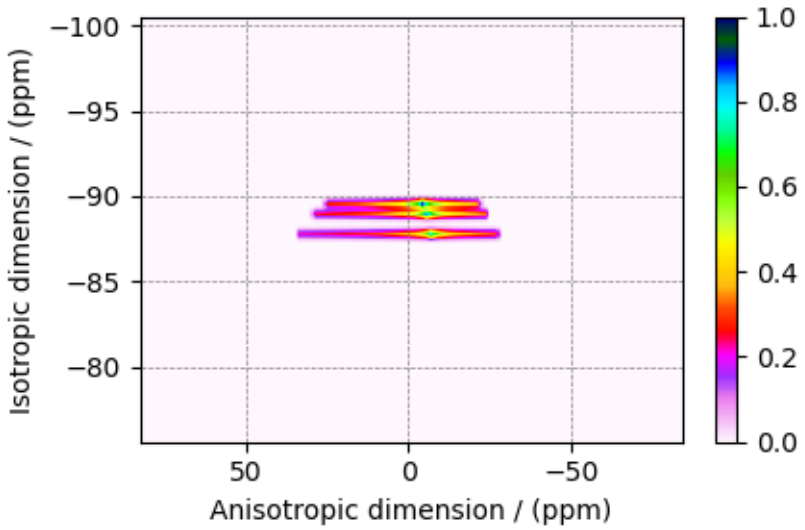
The plot of the simulation after signal processing.

```
ax = plt.subplot(projection="csdm")
cb = ax.imshow(processed_data.T, aspect="auto", cmap="gist_ncar_r")
plt.colorbar(cb)
```

(continues on next page)

(continued from previous page)

```
ax.invert_xaxis()
ax.invert_yaxis()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.289 seconds)

4.1.4 2D NMR simulation (Disordered/Amorphous solids)

The following examples are the NMR spectrum simulation for amorphous solids. The examples include the illustrations for the following methods:

- Triple-quantum variable-angle spinning (*ThreeQ_VAS()* (page 182))

Simulating site disorder (crystalline)

87Rb (I=3/2) 3QMAS simulation with site disorder.

The following example illustrates an NMR simulation of a crystalline solid with site disorders. We model such disorders with Extended Czjzek distribution. The following case study shows an ^{87}Rb 3QMAS simulation of RbNO3.

```
import matplotlib as mpl
import numpy as np
from mrsimulator import Simulator
from mrsimulator.methods import ThreeQ_VAS
import matplotlib.pyplot as plt

from mrsimulator.models import ExtCzjzekDistribution
from mrsimulator.utils.collection import single_site_system_generator
from scipy.stats import multivariate_normal

# global plot configuration
mpl.rcParams["figure.figsize"] = [4.5, 3.0]
```

Generate probability distribution

Create three extended Czjzek distributions for the three sites in RbNO3 about their respective mean tensors.

```
# The range of isotropic chemical shifts, the quadrupolar coupling constant, and
# asymmetry parameters used in generating a 3D grid.
iso_r = np.arange(101) / 6.5 - 35  # in ppm
Cq_r = np.arange(100) / 100 + 1.25  # in MHz
eta_r = np.arange(11) / 10

# The 3D mesh grid over which the distribution amplitudes are evaluated.
iso, Cq, eta = np.meshgrid(iso_r, Cq_r, eta_r, indexing="ij")


def get_prob_dist(iso, Cq, eta, eps, cov):
    pdf = 0
    for i in range(len(iso)):
        # The 2D amplitudes for Cq and eta is sampled from the extended Czjzek model.
        avg_tensor = {"Cq": Cq[i], "eta": eta[i]}
        _, _, amp = ExtCzjzekDistribution(avg_tensor, eps=eps[i]).pdf(pos=[Cq_r, eta_r])

        # The 1D amplitudes for isotropic chemical shifts is sampled as a Gaussian.
        iso_amp = multivariate_normal(mean=iso[i], cov=[cov[i]]).pdf(iso_r)

        # The 3D amplitude grid is generated as an uncorrelated distribution of the
        # above two distribution, which is the product of the two distributions.
        pdf_t = np.repeat(amp, iso_r.size).reshape(eta_r.size, Cq_r.size, iso_r.size)
        pdf_t *= iso_amp
        pdf += pdf_t
    return pdf


iso_0 = [-27.4, -28.5, -31.3]  # isotropic chemical shifts for the three sites in ppm
Cq_0 = [1.68, 1.94, 1.72]  # Cq in MHz for the three sites
eta_0 = [0.2, 1, 0.5]  # eta for the three sites
eps_0 = [0.02, 0.02, 0.02]  # perturbation fractions for extended Czjzek distribution.
var_0 = [0.1, 0.1, 0.1]  # variance for the isotropic chemical shifts in ppm^2.

pdf = get_prob_dist(iso_0, Cq_0, eta_0, eps_0, var_0).T
```

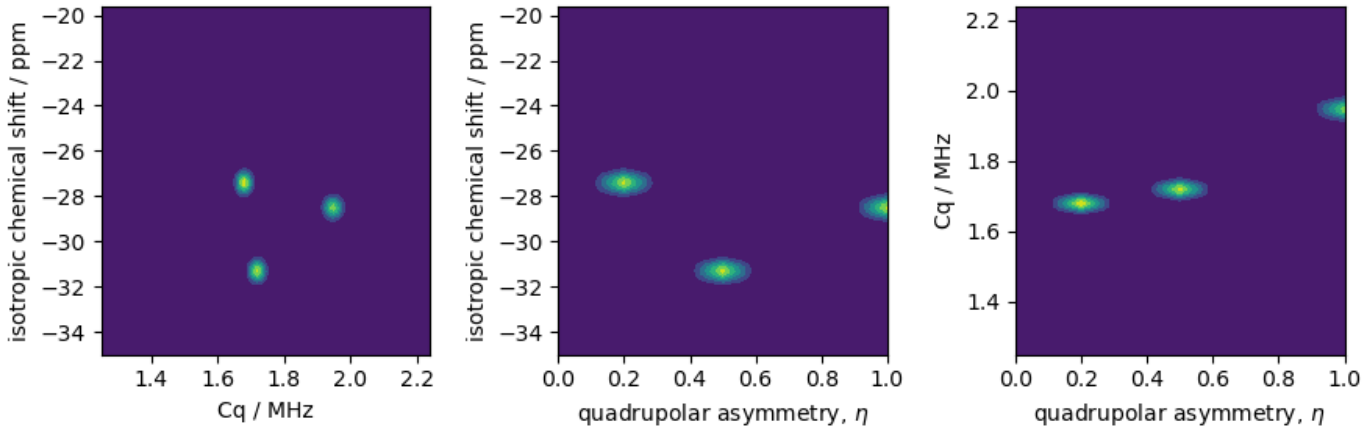
The two-dimensional projections from this three-dimensional distribution are shown below.

```
_, ax = plt.subplots(1, 3, figsize=(9, 3))

# isotropic shift v.s. quadrupolar coupling constant
ax[0].contourf(Cq_r, iso_r, pdf.sum(axis=2))
ax[0].set_xlabel("Cq / MHz")
ax[0].set_ylabel("isotropic chemical shift / ppm")

# isotropic shift v.s. quadrupolar asymmetry
ax[1].contourf(eta_r, iso_r, pdf.sum(axis=1))
ax[1].set_xlabel(r"quadrupolar asymmetry, $\eta$")
ax[1].set_ylabel("isotropic chemical shift / ppm")

# quadrupolar coupling constant v.s. quadrupolar asymmetry
ax[2].contourf(eta_r, Cq_r, pdf.sum(axis=0))
ax[2].set_xlabel(r"quadrupolar asymmetry, $\eta$")
ax[2].set_ylabel("Cq / MHz")
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout()
plt.show()
```



Simulation setup

Generate spin systems from the above probability distribution.

```
spin_systems = single_site_system_generator(
    isotopes="87Rb",
    isotropic_chemical_shifts=iso,
    quadrupolar={"Cq": Cq * 1e6, "eta": eta},  # Cq in Hz
    abundance=pdf,
)
len(spin_systems)
```

Out:

```
510
```

Simulate a ^{27}Al 3Q-MAS spectrum by using the *ThreeQ_MAS* method.

```
method = ThreeQ_VAS(
    channels=["87Rb"],
    magnetic_flux_density=9.4,  # in T
    rotor_angle=54.735 * np.pi / 180,
    spectral_dimensions=[
        {
            "count": 96,
            "spectral_width": 7e3,  # in Hz
            "reference_offset": -7e3,  # in Hz
            "label": "Isotropic dimension",
        },
        {
            "count": 256,
            "spectral_width": 1e4,  # in Hz
            "reference_offset": -4e3,  # in Hz
            "label": "MAS dimension",
        },
```

(continues on next page)

(continued from previous page)

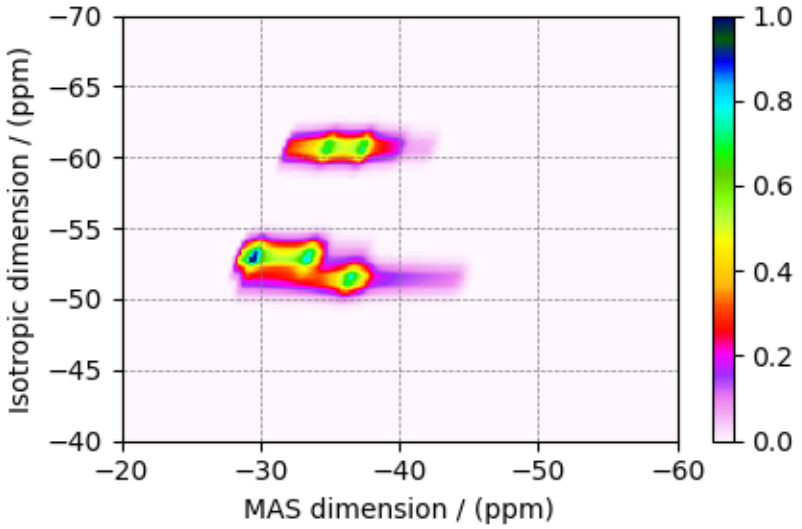
```
    ],
)
```

Create the simulator object, add the spin systems and method, and run the simulation.

```
sim = Simulator()
sim.spin_systems = spin_systems  # add the spin systems
sim.methods = [method]  # add the method
sim.config.number_of_sidebands = 1
sim.run()

data = sim.methods[0].simulation
```

The plot of the corresponding spectrum.

```
ax = plt.subplot(projection="csdm")
cb = ax.imshow(data / data.max(), cmap="gist_ncar_r", aspect="auto")
ax.set_ylim(-40, -70)
ax.set_xlim(-20, -60)
plt.colorbar(cb)
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 3.338 seconds)

Czjzek distribution, 27Al (I=5/2) 3QMAS

27Al (I=5/2) 3QMAS simulation of amorphous material.

In this section, we illustrate the simulation of a quadrupolar MQMAS spectrum arising from a distribution of the electric field gradient (EFG) tensors from amorphous material. We proceed by employing the Czjzek distribution model.

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
```

(continues on next page)

(continued from previous page)

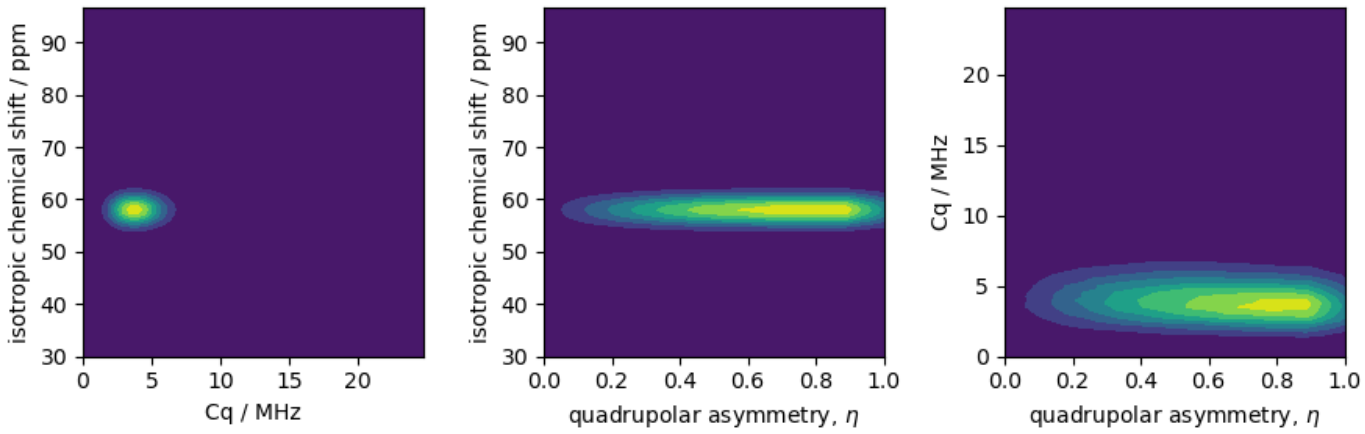
```
from mrsimulator import Simulator
from mrsimulator.methods import ThreeQ_VAS
from mrsimulator.models import CzjzekDistribution
from mrsimulator.utils.collection import single_site_system_generator
from scipy.stats import multivariate_normal

# global plot configuration
mpl.rcParams["figure.figsize"] = [4.5, 3.0]
```

Generate probability distribution

```
# The range of isotropic chemical shifts, the quadrupolar coupling constant, and
# asymmetry parameters used in generating a 3D grid.
iso_r = np.arange(101) / 1.5 + 30  # in ppm
Cq_r = np.arange(100) / 4  # in MHz
eta_r = np.arange(10) / 9

# The 3D mesh grid over which the distribution amplitudes are evaluated.
iso, Cq, eta = np.meshgrid(iso_r, Cq_r, eta_r, indexing="ij")

# The 2D amplitude grid of Cq and eta is sampled from the Czjzek distribution model.
Cq_dist, e_dist, amp = CzjzekDistribution(sigma=1).pdf(pos=[Cq_r, eta_r])

# The 1D amplitude grid of isotropic chemical shifts is sampled from a Gaussian model.
iso_amp = multivariate_normal(mean=58, cov=[4]).pdf(iso_r)

# The 3D amplitude grid is generated as an uncorrelated distribution of the above two
# distribution, which is the product of the two distributions.
pdf = np.repeat(amp, iso_r.size).reshape(eta_r.size, Cq_r.size, iso_r.size)
pdf *= iso_amp
pdf = pdf.T
```

The two-dimensional projections from this three-dimensional distribution are shown below.

```
_, ax = plt.subplots(1, 3, figsize=(9, 3))

# isotropic shift v.s. quadrupolar coupling constant
ax[0].contourf(Cq_r, iso_r, pdf.sum(axis=2))
ax[0].set_xlabel("Cq / MHz")
ax[0].set_ylabel("isotropic chemical shift / ppm")

# isotropic shift v.s. quadrupolar asymmetry
ax[1].contourf(eta_r, iso_r, pdf.sum(axis=1))
ax[1].set_xlabel(r"quadrupolar asymmetry, $\eta$")
ax[1].set_ylabel("isotropic chemical shift / ppm")

# quadrupolar coupling constant v.s. quadrupolar asymmetry
ax[2].contourf(eta_r, Cq_r, pdf.sum(axis=0))
ax[2].set_xlabel(r"quadrupolar asymmetry, $\eta$")
ax[2].set_ylabel("Cq / MHz")

plt.tight_layout()
plt.show()
```



Simulation setup

Let's create the site and spin system objects from these parameters. Use the `single_site_system_generator()` utility function to generate single-site spin systems.

```
spin_systems = single_site_system_generator(
    isotopes="27Al",
    isotropic_chemical_shifts=iso,
    quadrupolar={"Cq": Cq * 1e6, "eta": eta},  # Cq in Hz
    abundance=pdf,
)
len(spin_systems)
```

Out:

```
5770
```

Simulate a ^{27}Al 3Q-MAS spectrum by using the *ThreeQ_MAS* method.

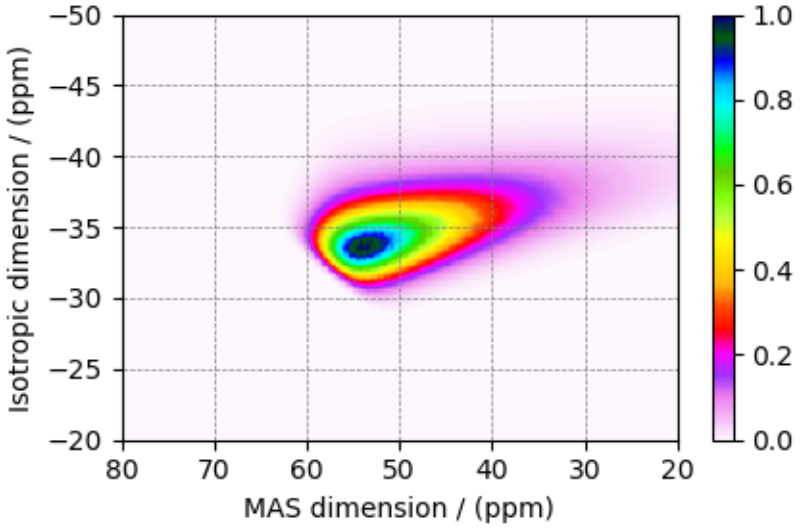
```
mqvas = ThreeQ_VAS(
    channels=["27Al"],
    spectral_dimensions=[
        {
            "count": 512,
            "spectral_width": 26718.475776,  # in Hz
            "reference_offset": -4174.76184,  # in Hz
            "label": "Isotropic dimension",
        },
        {
            "count": 512,
            "spectral_width": 2e4,  # in Hz
            "reference_offset": 2e3,  # in Hz
            "label": "MAS dimension",
        },
    ],
)
```

Create the simulator object, add the spin systems and method, and run the simulation.

```
sim = Simulator()
sim.spin_systems = spin_systems  # add the spin systems
sim.methods = [mqvas]  # add the method
sim.config.number_of_sidebands = 1
sim.run()

data = sim.methods[0].simulation
```

The plot of the corresponding spectrum.

```
ax = plt.subplot(projection="csdm")
cb = ax.imshow(data / data.max(), cmap="gist_ncar_r", aspect="auto")
plt.colorbar(cb)
ax.set_ylim(-20, -50)
ax.set_xlim(80, 20)
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 14.612 seconds)

4.2 Fitting Examples (Least Squares)

The `mrsimulator` library is easily integrable with other python-based libraries. In the following examples, we illustrate the use of LMFIT non-linear least-squares minimization python package to fit a simulation object to experimental data.

4.2.1 1D Data Fitting

Fitting Cusipidine

After acquiring an NMR spectrum, we often require a least-squares analysis to determine site populations and nuclear spin interaction parameters. Generally, this comprises of two steps:

- create a fitting model, and
- determine the model parameters that give the best fit to the spectrum.

Here, we will use the mrsimulator objects to create a fitting model, and use the LMFIT library for performing the least-squares fitting optimization. In this example, we use a synthetic ^{29}Si NMR spectrum of cuspidine, generated from the tensor parameters reported by Hansen *et. al.*[1], to demonstrate a simple fitting procedure.

We will begin by importing relevant modules and establishing figure size.

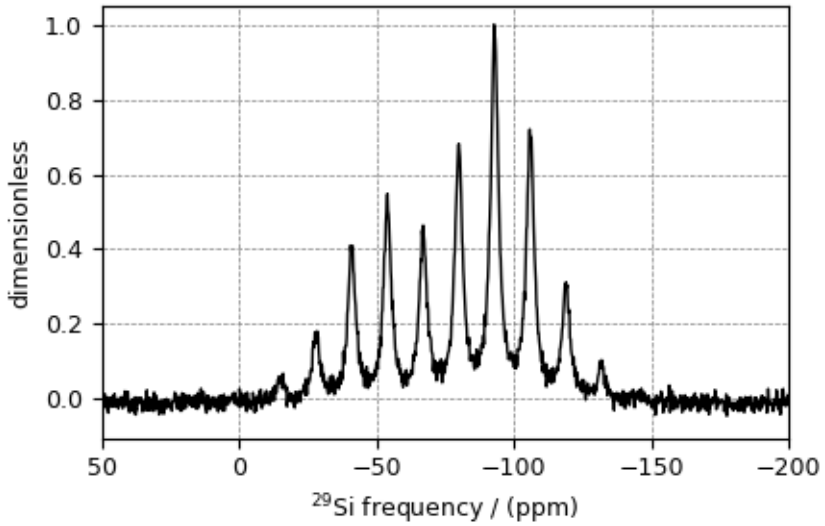
```
import csdmpy as cp
import matplotlib as mpl
import matplotlib.pyplot as plt
import mrsimulator.signal_processing as sp
import mrsimulator.signal_processing.apodization as apo
from mrsimulator import Simulator, SpinSystem
from mrsimulator.methods import BlochDecaySpectrum
from lmfit import Minimizer, Parameters, fit_report

font = {"size": 9}
mpl.rc("font", **font)
mpl.rcParams["figure.figsize"] = [4.5, 3.0]
```

Import the dataset

Use the csdmpy module to load the synthetic dataset as a CSDM object.

```
file_ = "https://sandbox.zenodo.org/record/687656/files/synthetic_cuspidine_test.csdf"
synthetic_experiment = cp.load(file_)

# convert the dimension coordinates from Hz to ppm
synthetic_experiment.dimensions[0].to("ppm", "nmr_frequency_ratio")

# Normalize the spectrum
synthetic_experiment /= synthetic_experiment.max()

# Plot of the synthetic dataset.
ax = plt.subplot(projection="csdm")
ax.plot(synthetic_experiment, color="black", linewidth=1)
ax.set_xlim(-200, 50)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```

[1] Hansen, M. R., Jakobsen, H. J., Skibsted, J., ^{29}Si Chemical Shift Anisotropies in Calcium Silicates from High-Field ^{29}Si MAS NMR Spectroscopy, Inorg. Chem. 2003, 42, 7, 2368-2377. DOI: 10.1021/ic020647f



Create a fitting model

Before you can fit a simulation to an experiment, in this case, the synthetic dataset, you will first need to create a fitting model. We will use the `mrsimulator` objects as tools in creating a model for the least-squares fitting.

Step 1: Create initial guess sites and spin systems. The initial guess is often based on some prior knowledge about the system under investigation. For the current example, we know that Cuspidine is a crystalline silica polymorph with one crystallographic Si site. Therefore, our initial guess model is a single ^{29}Si site spin system. For non-linear fitting algorithms, as a general recommendation, the initial guess model parameters should be a good starting point for the algorithms to converge.

```
# the guess model comprising of a single site spin system
site = dict(
    isotope="29Si",
    isotropic_chemical_shift=-82.0,  # in ppm,
    shielding_symmetric={"zeta": -63, "eta": 0.4},  # zeta in ppm
)

system_object = SpinSystem(
    name="Si Site",
    description="A 29Si site in cuspidine",
    sites=[site],  # from the above code
    abundance=100,
)
```

Step 2: Create the method object. The method should be the same as the one used in the measurement. In this example, we use the *BlochDecaySpectrum* method. Note, when creating the method object, the value of the method parameters must match the respective values used in the experiment.

```
method = BlochDecaySpectrum(
    channels=["29Si"],
    magnetic_flux_density=7.1,  # in T
    rotor_frequency=780,  # in Hz
    spectral_dimensions=[
        {
            "count": 2048,
            "spectral_width": 25000,  # in Hz
```

(continues on next page)

(continued from previous page)

```
            "reference_offset": -5000,  # in Hz
        }
    ],
)
```

Step 3: Create the Simulator object and add the method and spin system objects.

```
sim = Simulator()
sim.spin_systems = [system_object]
sim.methods = [method]

sim.methods[0].experiment = synthetic_experiment
```

Step 5: Simulate the spectrum.

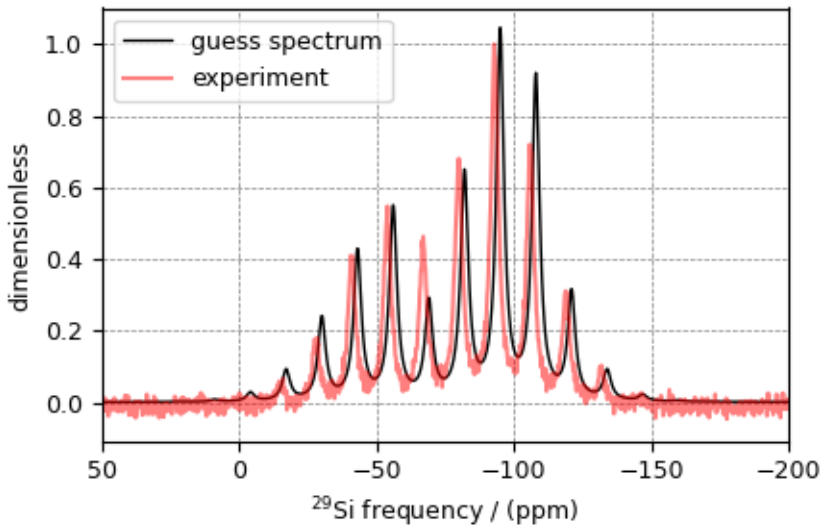
```
sim.run()
```

Step 6: Create a SignalProcessor class and apply post simulation processing.

```
processor = sp.SignalProcessor(
    operations=[
        sp.IFFT(),
        apo.Exponential(FWHM="200 Hz"),
        sp.FFT(),
        sp.Scale(factor=1.5),
    ]
)
processed_data = processor.apply_operations(data=sim.methods[0].simulation)
```

Step 7: The plot the spectrum. We also plot the synthetic dataset for comparison.

```
ax = plt.subplot(projection="csdm")
ax.plot(processed_data.real, c="k", linewidth=1, label="guess spectrum")
ax.plot(synthetic_experiment.real, c="r", linewidth=1.5, alpha=0.5, label="experiment")
ax.set_xlim(-200, 50)
ax.invert_xaxis()
plt.legend()
plt.tight_layout()
plt.show()
```



Setup a Least-squares minimization

Now that our model is ready, the next step is to set up a least-squares minimization. You may use any optimization package of choice, here we show an application using LMFIT. You may read more on the LMFIT documentation page.

Create fitting parameters

Next, you will need a list of parameters that will be used in the fit. The *LMFIT* library provides a Parameters class to create a list of parameters.

```
site1 = system_object.sites[0]
params = Parameters()

params.add(name="iso", value=site1.isotropic_chemical_shift)
params.add(name="eta", value=site1.shielding_symmetric.eta, min=0, max=1)
params.add(name="zeta", value=site1.shielding_symmetric.zeta)
params.add(name="FWHM", value=processor.operations[1].FWHM)
params.add(name="factor", value=processor.operations[3].factor)
```

Create a minimization function

Note, the above set of parameters does not know about the model. You will need to set up a function that will

- update the parameters of the *Simulator* and *SignalProcessor* object based on the LMFIT parameter updates,
- re-simulate the spectrum based on the updated values, and
- return the difference between the experiment and simulation.

```
def minimization_function(params, sim, processor):
    values = params.valuesdict()

    # the experiment data as a Numpy array
```

(continues on next page)

(continued from previous page)

```
    intensity = sim.methods[0].experiment.dependent_variables[0].components[0].real

    # Here, we update simulation parameters iso, eta, and zeta for the site object
    site = sim.spin_systems[0].sites[0]
    site.isotropic_chemical_shift = values["iso"]
    site.shielding_symmetric.eta = values["eta"]
    site.shielding_symmetric.zeta = values["zeta"]

    # run the simulation
    sim.run()

    # update the SignalProcessor parameter and apply line broadening.
    # update the scaling factor parameter at index 3 of operations list.
    processor.operations[3].factor = values["factor"]
    # update the exponential apodization FWHM parameter at index 1 of operations list.
    processor.operations[1].FWHM = values["FWHM"]

    # apply signal processing
    processed_data = processor.apply_operations(sim.methods[0].simulation)

    # return the difference vector.
    return intensity - processed_data.dependent_variables[0].components[0].real
```

Note: To automate the fitting process, we provide a function to parse the `Simulator` and `SignalProcessor` objects for parameters and construct an *LMFIT* `Parameters` object. Similarly, a minimization function, analogous to the above *minimization_function*, is also included in the *mrsimulator* library. See the next example for usage instructions.

Perform the least-squares minimization

With the synthetic dataset, simulation, and the initial guess parameters, we are ready to perform the fit. To fit, we use the *LMFIT* Minimizer class.

```
minner = Minimizer(minimization_function, params, fcn_args=(sim, processor))
result = minner.minimize()
print(fit_report(result))
```

Out:

```
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 43
    # data points      = 2048
    # variables        = 5
    chi-square         = 0.64652669
    reduced chi-square = 3.1646e-04
    Akaike info crit   = -16498.4360
    Bayesian info crit = -16470.3129
[[Variables]]
    iso:    -79.9380013 +/- 0.00711831 (0.01%) (init = -82)
    eta:     0.59954040 +/- 0.00454110 (0.76%) (init = 0.4)
    zeta:   -57.8375715 +/- 0.14157932 (0.24%) (init = -63)
    FWHM:    190.132207 +/- 1.05980445 (0.56%) (init = 200)
    factor:  1.36785444 +/- 0.00530840 (0.39%) (init = 1.5)
```

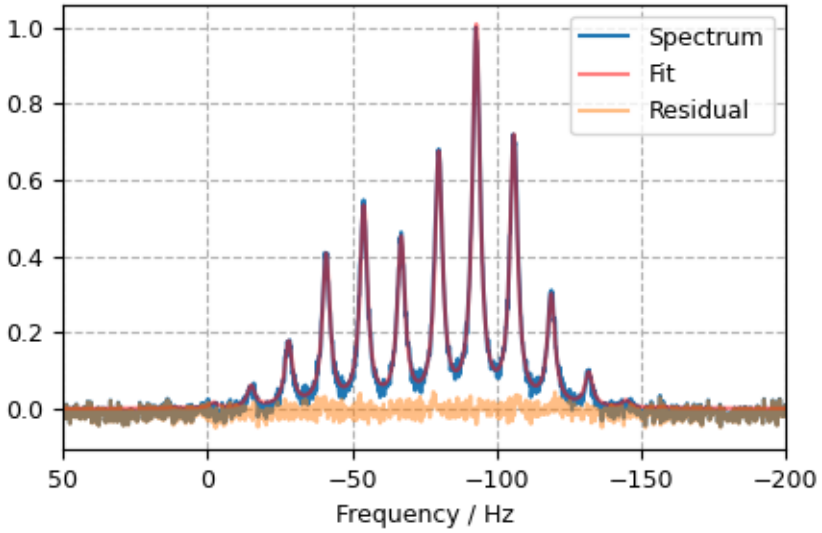
(continues on next page)

(continued from previous page)

```
[[Correlations]] (unreported correlations are < 0.100)
    C(FWHM, factor) =  0.531
    C(eta, zeta)    =  0.321
    C(zeta, factor) = -0.228
    C(eta, factor)  =  0.165
```

The plot of the fit, measurement and the residuals is shown below.

```
plt.figsize = (4, 3)
x, y_data = synthetic_experiment.to_list()
residual = result.residual
plt.plot(x, y_data, label="Spectrum")
plt.plot(x, y_data - residual, "r", alpha=0.5, label="Fit")
plt.plot(x, residual, alpha=0.5, label="Residual")

plt.xlabel("Frequency / Hz")
plt.xlim(-200, 50)
plt.gca().invert_xaxis()
plt.grid(which="major", axis="both", linestyle="--")
plt.legend()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 2.765 seconds)

Fitting 17O MAS NMR of crystalline Na2SiO3

In this example, we illustrate the use of the mrsimulator objects to

- create a spin system fitting model,
- use the fitting model to perform a least-squares fit on the experimental, and
- extract the tensor parameters of the spin system model.

We will be using the LMFIT methods to establish fitting parameters and fit the spectrum. The following example illustrates the least-squares fitting on a ^{17}O measurement of Na_2SiO_3[1].

We will begin by importing relevant modules and presetting figure style and layout.

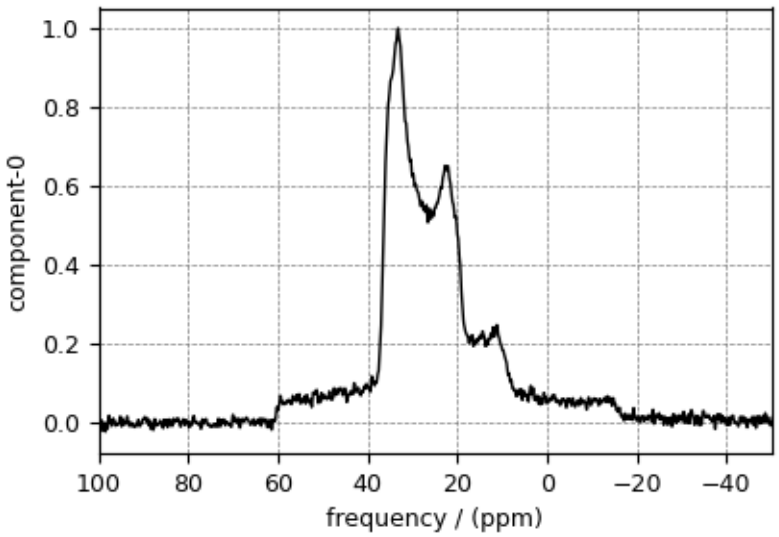
```
import csdmpy as cp
import matplotlib as mpl
import matplotlib.pyplot as plt
import mrsimulator.signal_processing as sp
import mrsimulator.signal_processing.apodization as apo
from lmfit import Minimizer, report_fit
from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.methods import BlochDecayCentralTransitionSpectrum
from mrsimulator.utils import get_spectral_dimensions
from mrsimulator.utils.spectral_fitting import LMFIT_min_function, make_LMFIT_params

font = {"size": 9}
mpl.rc("font", **font)
mpl.rcParams["figure.figsize"] = [4.25, 3.0]
```

Import the dataset

Import the experimental data. In this example, we will import the dataset file serialized with the CSDM file-format, using the csdmpy module.

```
filename = "https://sandbox.zenodo.org/record/687656/files/Na2SiO3_O17.csdf"
oxygen_experiment = cp.load(filename)

# For spectral fitting, we only focus on the real part of the complex dataset
oxygen_experiment = oxygen_experiment.real

# Convert the dimension coordinates from Hz to ppm.
oxygen_experiment.dimensions[0].to("ppm", "nmr_frequency_ratio")

# Normalize the spectrum
oxygen_experiment /= oxygen_experiment.max()

# plot of the dataset.
ax = plt.subplot(projection="csdm")
ax.plot(oxygen_experiment, color="black", linewidth=1)
ax.set_xlim(-50, 100)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```

[1] T. M. Clark, P. Florian, J. F. Stebbins, and P. J. Grandinetti, An ^{17}O NMR Investigation of Crystalline Sodium Metasilicate: Implications for the Determination of Local Structure in Alkali Silicates, J. Phys. Chem. B. 2001, 105, 12257-12265. DOI: 10.1021/jp011289p



Create a fitting model

Next, we will create a `simulator` object that we use to fit the spectrum. We will start by creating the guess `SpinSystem` objects.

Step 1: Create initial guess sites and spin systems

```
O17_1 = Site(
    isotope="17O",
    isotropic_chemical_shift=60.0,  # in ppm,
    quadrupolar={"Cq": 4.2e6, "eta": 0.5},  # Cq in Hz
)

O17_2 = Site(
    isotope="17O",
    isotropic_chemical_shift=40.0,  # in ppm,
    quadrupolar={"Cq": 2.4e6, "eta": 0},  # Cq in Hz
)

system_object = [SpinSystem(sites=[s], abundance=50) for s in [O17_1, O17_2]]
```

Step 2: Create the method object. Note, when performing the least-squares fit, you must create an appropriate method object which matches the method used in acquiring the experimental data. The attribute values of this method must match the exact conditions under which the experiment was acquired. This including the acquisition channels, the magnetic flux density, rotor angle, rotor frequency, and the spectral/spectroscopic dimension. In the following example, we set up a central transition selective Bloch decay spectrum method, where we obtain the spectral/spectroscopic information from the metadata of the CSDM dimension. Use the *get_spectral_dimensions()* (page 192) utility function for quick extraction of the spectroscopic information, *i.e.*, count, spectral_width, and reference_offset from the CSDM object. The remaining attribute values are set to the experimental conditions.

```
# get the count, spectral_width, and reference_offset information from the experiment.
spectral_dims = get_spectral_dimensions(oxygen_experiment)

method = BlochDecayCentralTransitionSpectrum(
    channels=["17O"],
    magnetic_flux_density=9.4,  # in T
    rotor_frequency=14000,  # in Hz
```

(continues on next page)

(continued from previous page)

```
    spectral_dimensions=spectral_dims,
)
```

Assign the experimental dataset to the `experiment` attribute of the above method.

```
method.experiment = oxygen_experiment
```

Step 3: Create the Simulator object and add the method and spin system objects.

```
sim = Simulator()
sim.spin_systems = system_object
sim.methods = [method]
```

Step 4: Simulate the spectrum.

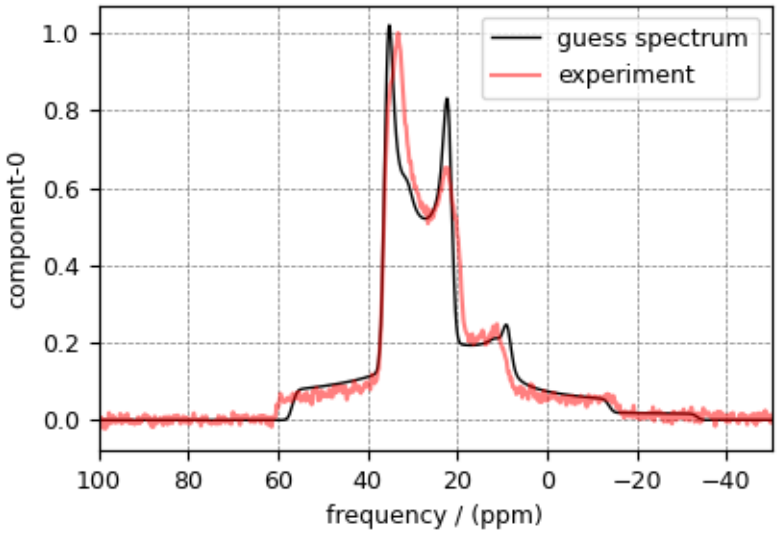
```
for iso in sim.spin_systems:
    # A method object queries every spin system for a list of transition pathways that
    # are relevant for the given method. Since the method and the number of spin systems
    # remain the same during the least-squares fit, a one-time query is sufficient. To
    # avoid querying for the transition pathways at every iteration in a least-squares
    # fitting, evaluate the transition pathways once and store it as follows
    iso.transition_pathways = method.get_transition_pathways(iso)

# Now simulate as usual.
sim.run()
```

Step 5: Create the SignalProcessor class object and apply the post-simulation signal processing operations.

```
processor = sp.SignalProcessor(
    operations=[
        sp.IFFT(),
        apo.Gaussian(FWHM="100 Hz"),
        sp.FFT(),
        sp.Scale(factor=20000.0),
    ]
)
processed_data = processor.apply_operations(data=sim.methods[0].simulation)
```

Step 6: The plot of initial guess simulation (black) along with the experiment (red) is shown below.

```
ax = plt.subplot(projection="csdm")
ax.plot(processed_data.real, color="black", linewidth=1, label="guess spectrum")
ax.plot(oxygen_experiment, c="r", linewidth=1.5, alpha=0.5, label="experiment")
ax.set_xlim(-50, 100)
ax.invert_xaxis()
plt.legend()
plt.tight_layout()
plt.show()
```



Least-squares minimization with LMFIT

Once you have a fitting model, you need to create the list of parameters to use in the least-squares fitting. For this, you may use the Parameters class from *LMFIT*, as described in the previous example. Here, we make use of a utility function, *make_LMFIT_params()* (page 191), that considerably simplifies the LMFIT parameters generation process.

Step 7: Create a list of parameters.

```
params = make_LMFIT_params(sim, processor)
```

The *make_LMFIT_params* parses the instances of the `Simulator` and the `PostSimulator` objects for parameters and returns an LMFIT *Parameters* object.

Customize the Parameters: You may customize the parameters list, `params`, as desired. Here, we remove the abundance of the two spin systems and constrain it to the initial value of 50% each.

```
params.pop("sys_0_abundance")
params.pop("sys_1_abundance")
params.pretty_print()
```

Out:

```
Name                                 Value       Min       Max    Stderr     Vary     Expr␣
↪Brute_Step
operation_1_Gaussian_FWHM              100      -inf       inf      None     True     None␣
↪None
operation_3_Scale_factor              2e+04      -inf       inf      None     True     None␣
↪None
sys_0_site_0_isotropic_chemical_shift    60      -inf       inf      None     True     None␣
↪None
sys_0_site_0_quadrupolar_Cq          4.2e+06      -inf       inf      None     True     None␣
↪None
sys_0_site_0_quadrupolar_eta             0.5         0         1      None     True     None␣
↪None
sys_1_site_0_isotropic_chemical_shift    40      -inf       inf      None     True     None␣
↪None
```

(continues on next page)

(continued from previous page)

```
sys_1_site_0_quadrupolar_Cq             2.4e+06     -inf      inf     None      True      None     ␣
↪None
sys_1_site_0_quadrupolar_eta                  0        0        1     None      True      None     ␣
↪None
```

Step 8: Perform least-squares minimization. For the user's convenience, we also provide a utility function, *LMFIT_min_function()* (page 191), for evaluating the difference vector between the simulation and experiment, based on the parameters update. You may use this function directly as the argument of the LMFIT Minimizer class, as follows,

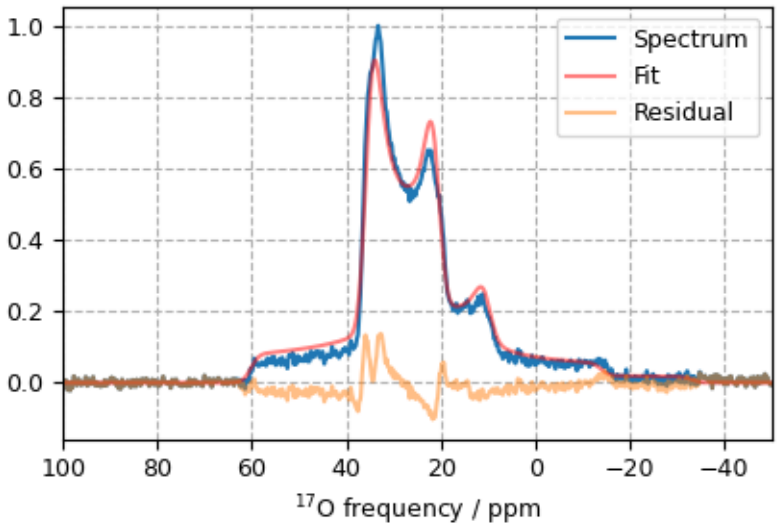
```
minner = Minimizer(LMFIT_min_function, params, fcn_args=(sim, processor))
result = minner.minimize()
report_fit(result)
```

Out:

```
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 75
    # data points      = 4096
    # variables        = 8
    chi-square         = 1.47285018
    reduced chi-square = 3.6029e-04
    Akaike info crit   = -32467.6014
    Bayesian info crit = -32417.0593
[[Variables]]
    sys_0_site_0_isotropic_chemical_shift:  63.1644543 +/- 0.15646429 (0.25%) (init = 60)
    sys_0_site_0_quadrupolar_Cq:            4255845.06 +/- 7374.39734 (0.17%) (init = 4200000)
    sys_0_site_0_quadrupolar_eta:           0.52688815 +/- 0.00416758 (0.79%) (init = 0.5)
    sys_1_site_0_isotropic_chemical_shift:  39.3214919 +/- 0.06149227 (0.16%) (init = 40)
    sys_1_site_0_quadrupolar_Cq:            2399432.05 +/- 6060.12351 (0.25%) (init = 2400000)
    sys_1_site_0_quadrupolar_eta:           0.00460343 +/- 0.06747584 (1465.77%) (init = 0)
    operation_1_Gaussian_FWHM:              176.197353 +/- 1.70616109 (0.97%) (init = 100)
    operation_3_Scale_factor:               21407.0797 +/- 47.5123039 (0.22%) (init = 20000)
[[Correlations]] (unreported correlations are < 0.100)
    C(sys_1_site_0_isotropic_chemical_shift, sys_1_site_0_quadrupolar_Cq)           =  0.978
    C(sys_1_site_0_quadrupolar_Cq, sys_1_site_0_quadrupolar_eta)                    =  0.938
    C(sys_1_site_0_isotropic_chemical_shift, sys_1_site_0_quadrupolar_eta)          =  0.931
    C(sys_0_site_0_isotropic_chemical_shift, sys_0_site_0_quadrupolar_Cq)           =  0.903
    C(sys_0_site_0_quadrupolar_eta, sys_1_site_0_isotropic_chemical_shift)          =  0.438
    C(sys_0_site_0_quadrupolar_eta, sys_1_site_0_quadrupolar_Cq)                    =  0.375
    C(sys_0_site_0_quadrupolar_Cq, operation_3_Scale_factor)                        =  0.363
    C(sys_0_site_0_isotropic_chemical_shift, operation_3_Scale_factor)              =  0.337
    C(sys_0_site_0_quadrupolar_Cq, sys_0_site_0_quadrupolar_eta)                    = -0.307
    C(sys_0_site_0_quadrupolar_eta, operation_1_Gaussian_FWHM)                      = -0.298
    C(operation_1_Gaussian_FWHM, operation_3_Scale_factor)                          =  0.286
    C(sys_0_site_0_quadrupolar_Cq, sys_1_site_0_isotropic_chemical_shift)           = -0.277
    C(sys_1_site_0_quadrupolar_eta, operation_1_Gaussian_FWHM)                      =  0.263
    C(sys_0_site_0_quadrupolar_eta, sys_1_site_0_quadrupolar_eta)                   =  0.256
    C(sys_0_site_0_quadrupolar_Cq, sys_1_site_0_quadrupolar_Cq)                     = -0.243
    C(sys_0_site_0_isotropic_chemical_shift, operation_1_Gaussian_FWHM)             =  0.235
    C(sys_0_site_0_quadrupolar_Cq, operation_1_Gaussian_FWHM)                       =  0.201
    C(sys_0_site_0_isotropic_chemical_shift, sys_1_site_0_isotropic_chemical_shift) = -0.197
    C(sys_0_site_0_quadrupolar_Cq, sys_1_site_0_quadrupolar_eta)                    = -0.188
    C(sys_0_site_0_isotropic_chemical_shift, sys_0_site_0_quadrupolar_eta)          = -0.175
    C(sys_0_site_0_isotropic_chemical_shift, sys_1_site_0_quadrupolar_Cq)           = -0.159
    C(sys_1_site_0_quadrupolar_Cq, operation_1_Gaussian_FWHM)                       =  0.156
    C(sys_1_site_0_isotropic_chemical_shift, operation_1_Gaussian_FWHM)             =  0.122
```

Step 9: The plot of the fit, measurement and the residuals is shown below.

```
plt.figsize = (4, 3)
x, y_data = oxygen_experiment.to_list()
residual = result.residual
plt.plot(x, y_data, label="Spectrum")
plt.plot(x, y_data - residual, "r", alpha=0.5, label="Fit")
plt.plot(x, residual, alpha=0.5, label="Residual")

plt.xlabel("$^{17}$O frequency / ppm")
plt.xlim(-50, 100)
plt.gca().invert_xaxis()
plt.grid(which="major", axis="both", linestyle="--")
plt.legend()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 4.966 seconds)

Fitting PASS/MAT cross-sections

This example illustrates the use of mrsimulator and LMFIT modules in fitting the sideband intensity profile across the isotropic chemical shift cross-section from a PASS/MAT dataset.

```
import numpy as np
import csdmpy as cp
import matplotlib as mpl
import matplotlib.pyplot as plt
import mrsimulator.signal_processing as sp
from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.methods import BlochDecaySpectrum
from mrsimulator.utils import get_spectral_dimensions
from mrsimulator.utils.spectral_fitting import LMFIT_min_function, make_LMFIT_params
from lmfit import Minimizer, report_fit
```

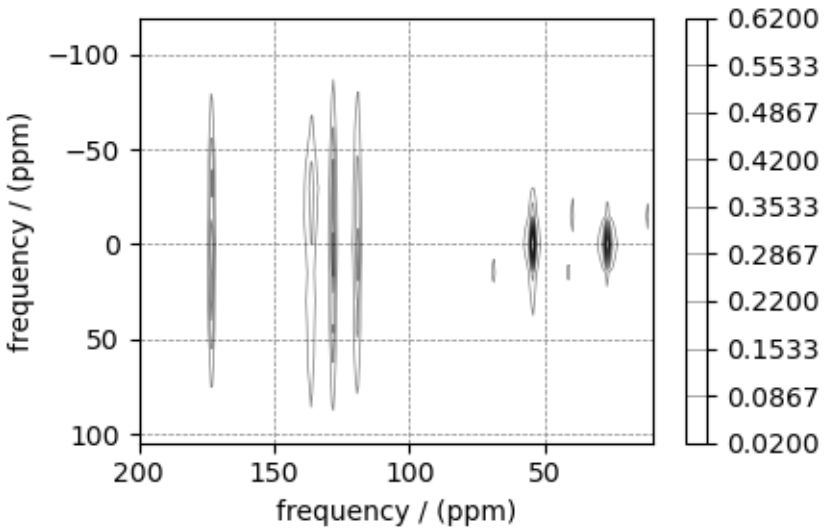
(continues on next page)

(continued from previous page)

```
# global plot configuration
mpl.rcParams["figure.figsize"] = [4.5, 3.0]
```

Import the dataset

```
filename = "https://sandbox.zenodo.org/record/687656/files/1H13C_CPPASS_LHistidine.csdf"
pass_data = cp.load(filename)

# For the spectral fitting, we only focus on the real part of the complex dataset.
# The script assumes that the dimension at index 0 is the isotropic dimension.
# Transpose the dataset as required.
pass_data = pass_data.real.T

# Convert the coordinates along each dimension from Hz to ppm.
_ = [item.to("ppm", "nmr_frequency_ratio") for item in pass_data.dimensions]

# Normalize the spectrum.
pass_data /= pass_data.max()

# The plot of the dataset.
levels = (np.arange(10) + 0.3) / 15  # contours are drawn at these levels.
ax = plt.subplot(projection="csdm")
cb = ax.contour(pass_data, colors="k", levels=levels, alpha=0.5, linewidths=0.5)
plt.colorbar(cb)
ax.set_xlim(200, 10)
ax.invert_yaxis()
plt.tight_layout()
plt.show()
```



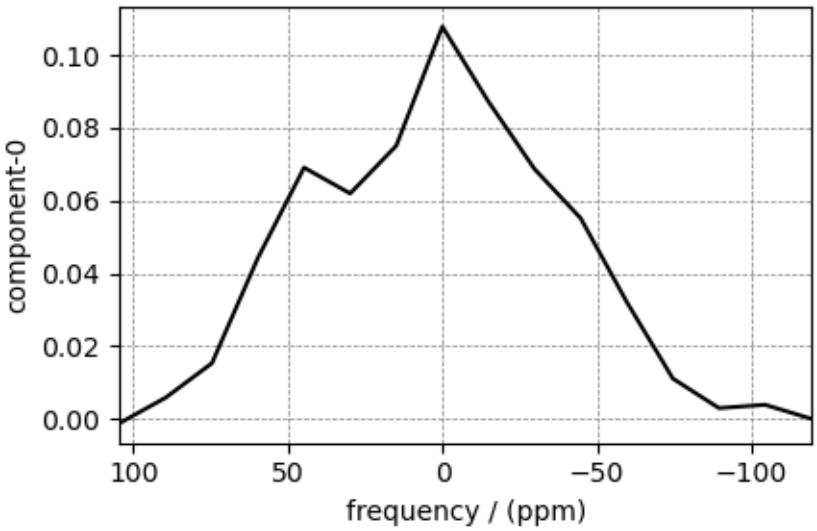
Extract a 1D sideband intensity cross-section from the 2D dataset using the array indexing.

```
data1D = pass_data[1100]  # sideband dataset

# The plot of the cross-section.
```

(continues on next page)

(continued from previous page)

```
ax = plt.subplot(projection="csdm")
ax.plot(data1D, color="k")
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```



The isotropic chemical shift coordinate of the cross-section is

```
isotropic_shift = pass_data.x[0].coords[1100]
print(isotropic_shift)
```

Out:

```
119.8940272861969 ppm
```

Create a fitting model

The fitting model includes the Simulator and SignalProcessor objects. First, create the Simulator object.

```
# Create the guess site and spin system for the 1D cross-section.
zeta = -70  # in ppm
eta = 0.8

site = Site(
    isotope="13C",
    isotropic_chemical_shift=0,
    shielding_symmetric={"zeta": zeta, "eta": eta},
)
spin_systems = [SpinSystem(sites=[site])]
```

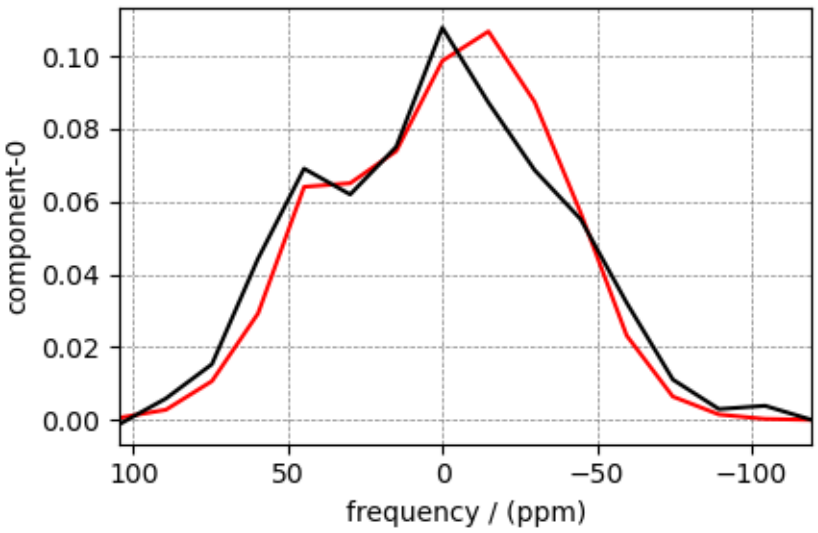
For the sideband only cross-section, use the BlochDecaySpectrum method.

```
# Get the dimension information from the experiment. Note, the following function
# returns an array of two spectral dimensions corresponding to the 2D PASS dimensions.
```

(continues on next page)

(continued from previous page)

```
# Use the spectral dimension that is along the anisotropic dimensions for the
# BlochDecaySpectrum method.
spectral_dims = get_spectral_dimensions(pass_data)
method = BlochDecaySpectrum(
    channels=["13C"],
    magnetic_flux_density=9.4,  # in T
    rotor_frequency=1500,  # in Hz
    spectral_dimensions=[spectral_dims[0]],
    experiment=data1D,  # also add the measurement to the method.
)

# Optimize the script by pre-setting the transition pathways for each spin system from
# the method.
for sys in spin_systems:
    sys.transition_pathways = method.get_transition_pathways(sys)

# Create the Simulator object and add the method and spin system objects.
sim = Simulator()
sim.spin_systems = spin_systems  # add the spin systems
sim.methods = [method]  # add the method
sim.run()

# Add and apply Post simulation processing.
processor = sp.SignalProcessor(operations=[sp.Scale(factor=1)])
processed_data = processor.apply_operations(data=sim.methods[0].simulation).real

# The plot of the simulation from the guess model and experiment cross-section.
ax = plt.subplot(projection="csdm")
ax.plot(processed_data, color="r", label="guess")
ax.plot(data1D, color="k", label="experiment")
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```



Least-squares minimization with LMFIT

First, create the fitting parameters. Use the *make_LMFIT_params()* (page 191) for a quick setup.

```
params = make_LMFIT_params(sim, processor)

# Fix the value of the isotropic chemical shift to zero for pure anisotropic sideband
# amplitude simulation.
params["sys_0_site_0_isotropic_chemical_shift"].vary = False
params.pretty_print()
```

Out:

```
Name                                    Value      Min      Max    Stderr     Vary     Expr␣
↪Brute_Step
operation_0_Scale_factor                    1     -inf      inf      None     True     None␣
↪None
sys_0_abundance                           100        0      100      None    False      100␣
↪None
sys_0_site_0_isotropic_chemical_shift       0     -inf      inf      None    False     None␣
↪None
sys_0_site_0_shielding_symmetric_eta      0.8        0        1      None     True     None␣
↪None
sys_0_site_0_shielding_symmetric_zeta     -70     -inf      inf      None     True     None␣
↪None
```

Run the minimization using LMFIT

```
minner = Minimizer(LMFIT_min_function, params, fcn_args=(sim, processor))
result = minner.minimize()
report_fit(result)
```

Out:

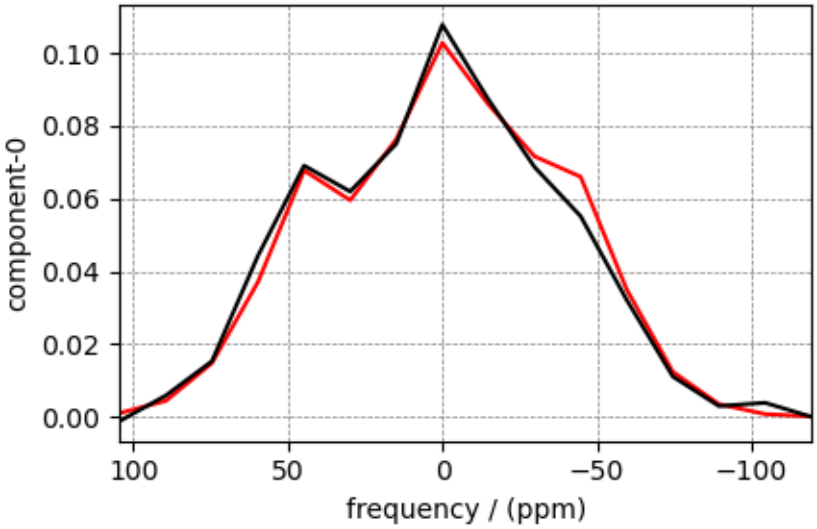
```
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 25
    # data points      = 16
    # variables        = 3
    chi-square         = 2.4041e-04
    reduced chi-square = 1.8493e-05
    Akaike info crit   = -171.692083
    Bayesian info crit = -169.374317
[[Variables]]
    sys_0_site_0_isotropic_chemical_shift:  0 (fixed)
    sys_0_site_0_shielding_symmetric_zeta: -74.8435735 +/- 1.40429904 (1.88%) (init = -70)
    sys_0_site_0_shielding_symmetric_eta:   0.92016512 +/- 0.02992412 (3.25%) (init = 0.8)
    sys_0_abundance:                        100.000000 +/- 0.00000000 (0.00%) == '100'
    operation_0_Scale_factor:               1.01870585 +/- 0.02213807 (2.17%) (init = 1)
[[Correlations]] (unreported correlations are < 0.100)
    C(sys_0_site_0_shielding_symmetric_zeta, sys_0_site_0_shielding_symmetric_eta) =  0.449
    C(sys_0_site_0_shielding_symmetric_zeta, operation_0_Scale_factor)             = -0.303
```

Simulate the spectrum corresponding to the optimum parameters

```
sim.run()
processed_data = processor.apply_operations(data=sim.methods[0].simulation).real
```

Plot the spectrum

```
ax = plt.subplot(projection="csdm")
ax.plot(processed_data, color="r", label="fit")
ax.plot(data1D, color="k", label="experiment")
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 2.436 seconds)

4.2.2 2D Data Fitting

13C 2D PASS NMR of LHistidine

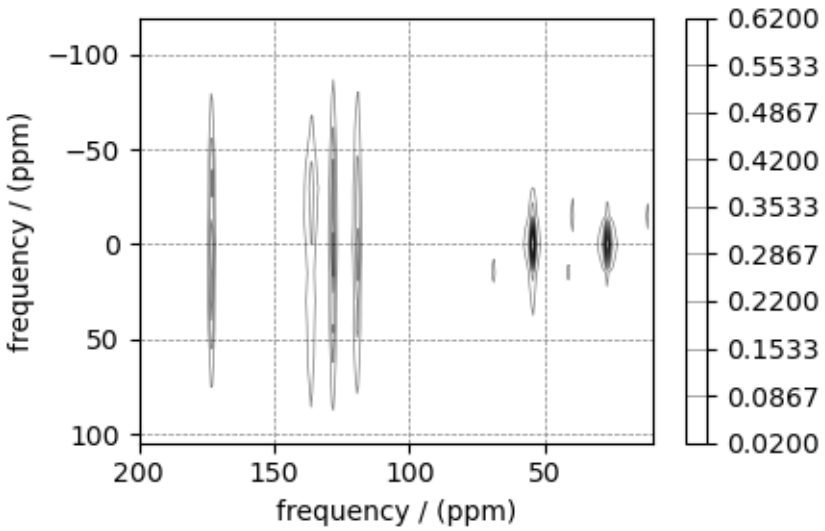
Coesite is a high-pressure (2-3 GPa) and high-temperature (700°C) polymorph of silicon dioxide SiO_2. Coesite has five crystallographic ^{17}O sites. The experimental dataset used in this example is published in Grandinetti *et. al.*[1]

```
import numpy as np
import csdmpy as cp
import matplotlib as mpl
import matplotlib.pyplot as plt
import mrsimulator.signal_processing as sp
import mrsimulator.signal_processing.apodization as apo
from mrsimulator import Simulator
from mrsimulator.methods import SSB2D
from mrsimulator.utils import get_spectral_dimensions
from mrsimulator.utils.spectral_fitting import LMFIT_min_function, make_LMFIT_params
from lmfit import Minimizer, report_fit
from mrsimulator.utils.collection import single_site_system_generator

# global plot configuration
mpl.rcParams["figure.figsize"] = [4.5, 3.0]
```

[1] Grandinetti, P. J., Baltisberger, J. H., Farnan, I., Stebbins, J. F., Werner, U. and Pines, A. Solid-State ^{17}O Magic-Angle and Dynamic-Angle Spinning NMR Study of the SiO_2 Polymorph Coesite, J. Phys. Chem. 1995, 99, 32, 12341-12348. DOI: 10.1021/j100032a045

Import the dataset

```
filename = "https://sandbox.zenodo.org/record/687656/files/1H13C_CPPASS_LHistidine.csdf"
pass_data = cp.load(filename)

# For the spectral fitting, we only focus on the real part of the complex dataset.
# The script assumes that the dimension at index 0 is the isotropic dimension.
# Transpose the dataset as required.
pass_data = pass_data.real.T

# Convert the coordinates along each dimension from Hz to ppm.
_ = [item.to("ppm", "nmr_frequency_ratio") for item in pass_data.dimensions]

# Normalize the spectrum
pass_data /= pass_data.max()

# plot of the dataset.
levels = (np.arange(10) + 0.3) / 15  # contours are drawn at these levels.
ax = plt.subplot(projection="csdm")
cb = ax.contour(pass_data, colors="k", levels=levels, alpha=0.5, linewidths=0.5)
plt.colorbar(cb)
ax.set_xlim(200, 10)
ax.invert_yaxis()
plt.tight_layout()
plt.show()
```



Create a fitting model

The fitting model includes the Simulator and the SignalProcessor objects. First create the Simulator object.

```
# Create the guess sites and spin systems.
# default unit of isotropic_chemical_shift is ppm and Cq is Hz.
shifts = [120, 128, 135, 175, 55, 25]  # in ppm
zeta = [-70, -65, -60, -60, -10, -10]  # in  Hz
eta = [0.8, 0.4, 0.9, 0.3, 0.0, 0.0]

spin_systems = single_site_system_generator(
    isotopes="13C",
    isotropic_chemical_shifts=shifts,
    shielding_symmetric={"zeta": zeta, "eta": eta},
    abundance=100 / 6,
)

# Create the DAS method.
# Get the spectral dimension paramters from the experiment.
spectral_dims = get_spectral_dimensions(pass_data)
```

```
ssb = SSB2D(
    channels=["13C"],
    magnetic_flux_density=9.4,  # in T
    rotor_frequency=1500,  # in Hz
    spectral_dimensions=spectral_dims,
    experiment=pass_data,  # also add the measurement to the method.
)

# Optimize the script by pre-setting the transition pathways for each spin system from
# the das method.
for sys in spin_systems:
    sys.transition_pathways = ssb.get_transition_pathways(sys)
```

```
# Create the Simulator object and add the method and spin system objects.
sim = Simulator()
sim.spin_systems = spin_systems  # add the spin systems
sim.methods = [ssb]  # add the method
sim.run()
```

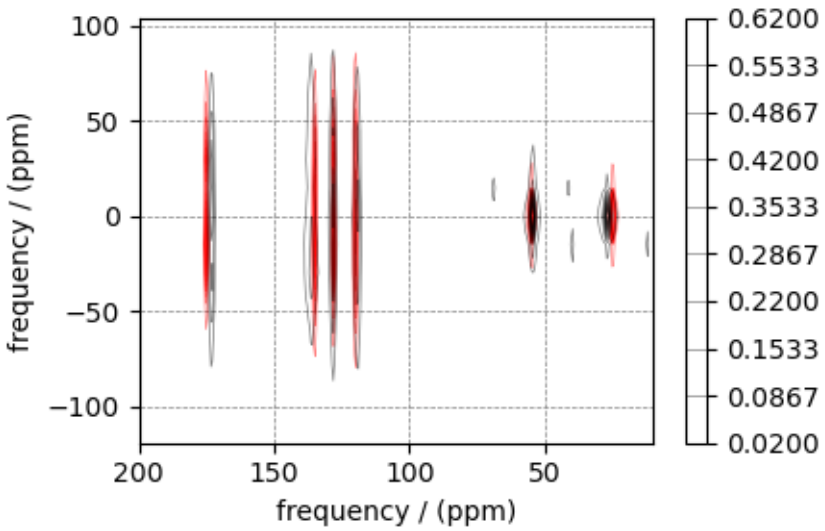
```
# Add Post simulation processing
processor = sp.SignalProcessor(
    operations=[
        # Gaussian convolution along the isotropic dimensions.
        sp.FFT(axis=0),
        apo.Exponential(FWHM="20 Hz"),
        sp.IFFT(axis=0),
        sp.Scale(factor=0.6),
    ]
)
# Apply post simulation operations
processed_data = processor.apply_operations(data=sim.methods[0].simulation).real
```

```
# The plot of the simulation after signal processing.
ax = plt.subplot(projection="csdm")
ax.contour(processed_data, colors="r", levels=levels, alpha=0.5, linewidths=0.5)
```

(continues on next page)

(continued from previous page)

```
cb = ax.contour(pass_data, colors="k", levels=levels, alpha=0.5, linewidths=0.5)
plt.colorbar(cb)
ax.set_xlim(200, 10)
plt.tight_layout()
plt.show()
```



Least-squares minimization with LMFIT

First create the fitting parameters. Use the *make_LMFIT_params()* (page 191) for a quick setup.

```
params = make_LMFIT_params(sim, processor)
print(params.pretty_print())
```

Out:

```
Name                                 Value       Min       Max    Stderr     Vary     Expr␣
↪Brute_Step
operation_1_Exponential_FWHM            20      -inf       inf      None     True     None␣
↪None
operation_3_Scale_factor               0.6      -inf       inf      None     True     None␣
↪None
sys_0_abundance                      16.67         0       100      None     True     None␣
↪None
sys_0_site_0_isotropic_chemical_shift  120      -inf       inf      None     True     None␣
↪None
sys_0_site_0_shielding_symmetric_eta   0.8         0         1      None     True     None␣
↪None
sys_0_site_0_shielding_symmetric_zeta  -70      -inf       inf      None     True     None␣
↪None
sys_1_abundance                      16.67         0       100      None     True     None␣
↪None
sys_1_site_0_isotropic_chemical_shift  128      -inf       inf      None     True     None␣
↪None
sys_1_site_0_shielding_symmetric_eta   0.4         0         1      None     True     None␣
↪None
```

(continues on next page)

(continued from previous page)

```
sys_1_site_0_shielding_symmetric_zeta       -65     -inf      inf     None     True     None  ␣
↪None
sys_2_abundance                           16.67        0      100     None     True     None  ␣
↪None
sys_2_site_0_isotropic_chemical_shift       135     -inf      inf     None     True     None  ␣
↪None
sys_2_site_0_shielding_symmetric_eta        0.9        0        1     None     True     None  ␣
↪None
sys_2_site_0_shielding_symmetric_zeta       -60     -inf      inf     None     True     None  ␣
↪None
sys_3_abundance                           16.67        0      100     None     True     None  ␣
↪None
sys_3_site_0_isotropic_chemical_shift       175     -inf      inf     None     True     None  ␣
↪None
sys_3_site_0_shielding_symmetric_eta        0.3        0        1     None     True     None  ␣
↪None
sys_3_site_0_shielding_symmetric_zeta       -60     -inf      inf     None     True     None  ␣
↪None
sys_4_abundance                           16.67        0      100     None     True     None  ␣
↪None
sys_4_site_0_isotropic_chemical_shift        55     -inf      inf     None     True     None  ␣
↪None
sys_4_site_0_shielding_symmetric_eta          0        0        1     None     True     None  ␣
↪None
sys_4_site_0_shielding_symmetric_zeta       -10     -inf      inf     None     True     None  ␣
↪None
sys_5_abundance                           16.67        0      100     None    False 100-sys_0_
↪abundance-sys_1_abundance-sys_2_abundance-sys_3_abundance-sys_4_abundance     None
sys_5_site_0_isotropic_chemical_shift       25     -inf      inf     None     True     None  ␣
↪None
sys_5_site_0_shielding_symmetric_eta          0        0        1     None     True     None  ␣
↪None
sys_5_site_0_shielding_symmetric_zeta       -10     -inf      inf     None     True     None  ␣
↪None
None
```

Run the minimization using LMFIT

```
minner = Minimizer(LMFIT_min_function, params, fcn_args=(sim, processor))
result = minner.minimize()
report_fit(result)
```

Out:

```
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 288
    # data points      = 32768
    # variables        = 25
    chi-square         = 0.24900550
    reduced chi-square = 7.6048e-06
    Akaike info crit   = -386202.407
    Bayesian info crit = -385992.477
[[Variables]]
    sys_0_site_0_isotropic_chemical_shift:  119.106046 +/- 0.00370472 (0.00%) (init = 120)
    sys_0_site_0_shielding_symmetric_zeta: -72.0665767 +/- 0.32561995 (0.45%) (init = -70)
    sys_0_site_0_shielding_symmetric_eta:   0.98532915 +/- 0.00760810 (0.77%) (init = 0.8)
```

(continues on next page)

(continued from previous page)

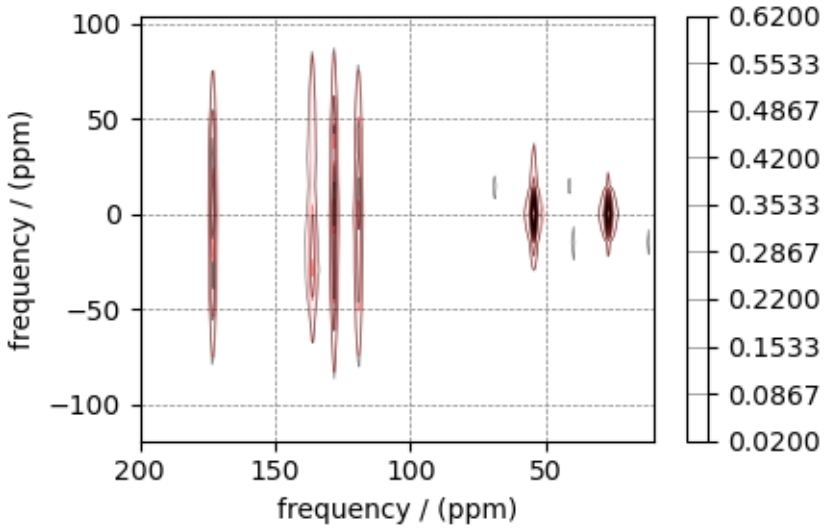
```
    sys_0_abundance:                          16.2131303 +/- 0.07746184 (0.48%) (init = 16.66667)
    sys_1_site_0_isotropic_chemical_shift:  128.128413 +/- 0.00311088 (0.00%) (init = 128)
    sys_1_site_0_shielding_symmetric_zeta: -75.5968193 +/- 0.27328472 (0.36%) (init = -65)
    sys_1_site_0_shielding_symmetric_eta:   0.94580299 +/- 0.00579597 (0.61%) (init = 0.4)
    sys_1_abundance:                          20.4664007 +/- 0.07798771 (0.38%) (init = 16.66667)
    sys_2_site_0_isotropic_chemical_shift:  136.122892 +/- 0.00473325 (0.00%) (init = 135)
    sys_2_site_0_shielding_symmetric_zeta: -86.2835196 +/- 0.38552276 (0.45%) (init = -60)
    sys_2_site_0_shielding_symmetric_eta:   0.42623625 +/- 0.00813561 (1.91%) (init = 0.9)
    sys_2_abundance:                          12.3406889 +/- 0.07816727 (0.63%) (init = 16.66667)
    sys_3_site_0_isotropic_chemical_shift:  172.906078 +/- 0.00306329 (0.00%) (init = 175)
    sys_3_site_0_shielding_symmetric_zeta: -69.4823878 +/- 0.25495175 (0.37%) (init = -60)
    sys_3_site_0_shielding_symmetric_eta:   0.99764477 +/- 0.00633951 (0.64%) (init = 0.3)
    sys_3_abundance:                          19.3309855 +/- 0.07589593 (0.39%) (init = 16.66667)
    sys_4_site_0_isotropic_chemical_shift:  54.4880968 +/- 0.00146751 (0.00%) (init = 55)
    sys_4_site_0_shielding_symmetric_zeta: -20.0861214 +/- 0.12734617 (0.63%) (init = -10)
    sys_4_site_0_shielding_symmetric_eta:   0.41793437 +/- 0.03977290 (9.52%) (init = 0)
    sys_4_abundance:                          18.1291926 +/- 0.05433628 (0.30%) (init = 16.66667)
    sys_5_site_0_isotropic_chemical_shift:  26.9768352 +/- 0.00161380 (0.01%) (init = 25)
    sys_5_site_0_shielding_symmetric_zeta: -10.3529547 +/- 0.53266920 (5.15%) (init = -10)
    sys_5_site_0_shielding_symmetric_eta:   0.71438965 +/- 0.24213064 (33.89%) (init = 0)
    sys_5_abundance:                          13.5196021 +/- 0.05051756 (0.37%) == '100-sys_0_
↪abundance-sys_1_abundance-sys_2_abundance-sys_3_abundance-sys_4_abundance'
    operation_1_Exponential_FWHM:             98.6457994 +/- 0.31250846 (0.32%) (init = 20)
    operation_3_Scale_factor:                 0.51492602 +/- 0.00116100 (0.23%) (init = 0.6)
[[Correlations]] (unreported correlations are < 0.100)
    C(sys_5_site_0_shielding_symmetric_zeta, sys_5_site_0_shielding_symmetric_eta) =  0.929
    C(sys_4_site_0_shielding_symmetric_zeta, sys_4_site_0_shielding_symmetric_eta) =  0.719
    C(operation_1_Exponential_FWHM, operation_3_Scale_factor)                      =  0.563
    C(sys_1_site_0_shielding_symmetric_zeta, sys_1_site_0_shielding_symmetric_eta) =  0.438
    C(sys_3_site_0_shielding_symmetric_zeta, sys_3_site_0_shielding_symmetric_eta) =  0.433
    C(sys_0_site_0_shielding_symmetric_zeta, sys_0_site_0_shielding_symmetric_eta) =  0.430
    C(sys_2_site_0_shielding_symmetric_zeta, sys_2_site_0_shielding_symmetric_eta) =  0.340
    C(sys_4_site_0_shielding_symmetric_zeta, sys_4_abundance)                      = -0.291
    C(sys_0_site_0_shielding_symmetric_zeta, sys_0_abundance)                      = -0.291
    C(sys_3_site_0_shielding_symmetric_zeta, sys_3_abundance)                      = -0.284
    C(sys_4_abundance, operation_3_Scale_factor)                                 = -0.277
    C(sys_0_abundance, sys_1_abundance)                                  = -0.274
    C(sys_1_site_0_shielding_symmetric_zeta, sys_1_abundance)                      = -0.270
    C(sys_1_abundance, sys_2_abundance)                                  = -0.269
    C(sys_1_abundance, sys_3_abundance)                                  = -0.263
    C(sys_0_abundance, sys_3_abundance)                                  = -0.257
    C(sys_2_abundance, sys_3_abundance)                                  = -0.247
    C(sys_0_abundance, sys_2_abundance)                                  = -0.232
    C(sys_2_site_0_shielding_symmetric_eta, sys_2_abundance)                       =  0.223
    C(sys_2_site_0_shielding_symmetric_zeta, sys_2_abundance)                      = -0.218
    C(sys_2_abundance, sys_4_abundance)                                  = -0.207
    C(sys_1_site_0_isotropic_chemical_shift, sys_1_abundance)                      =  0.199
    C(sys_0_abundance, sys_4_abundance)                                  = -0.183
    C(sys_4_site_0_isotropic_chemical_shift, operation_1_Exponential_FWHM)         = -0.162
    C(sys_2_abundance, operation_3_Scale_factor)                                 =  0.161
    C(sys_1_site_0_isotropic_chemical_shift, operation_1_Exponential_FWHM)         = -0.160
    C(sys_4_site_0_shielding_symmetric_eta, sys_4_abundance)                       =  0.157
    C(sys_3_abundance, sys_4_abundance)                                  = -0.157
    C(sys_1_abundance, sys_4_abundance)                                  = -0.152
    C(sys_3_site_0_shielding_symmetric_zeta, operation_3_Scale_factor)             = -0.118
    C(sys_0_site_0_shielding_symmetric_zeta, operation_3_Scale_factor)             = -0.116
    C(sys_1_site_0_shielding_symmetric_zeta, operation_3_Scale_factor)             = -0.114
```

Simulate the spectrum corresponding to the optimum parameters

```
sim.run()
processed_data = processor.apply_operations(data=sim.methods[0].simulation).real
```

Plot the spectrum

```
ax = plt.subplot(projection="csdm")
ax.contour(processed_data, colors="r", levels=levels, alpha=0.5, linewidths=0.5)
cb = ax.contour(pass_data, colors="k", levels=levels, alpha=0.5, linewidths=0.5)
plt.colorbar(cb)
ax.set_xlim(200, 10)
plt.tight_layout()
plt.show()
```



Total running time of the script: (1 minutes 6.494 seconds)

17O DAS NMR of Coesite

Coesite is a high-pressure (2-3 GPa) and high-temperature (700°C) polymorph of silicon dioxide SiO_2. Coesite has five crystallographic ^{17}O sites. The experimental dataset used in this example is published in Grandinetti *et. al.*[1]

```
import numpy as np
import csdmpy as cp
import matplotlib as mpl
import matplotlib.pyplot as plt
import mrsimulator.signal_processing as sp
import mrsimulator.signal_processing.apodization as apo
from mrsimulator import Simulator
from mrsimulator.methods import Method2D
from mrsimulator.utils import get_spectral_dimensions
from mrsimulator.utils.collection import single_site_system_generator
from mrsimulator.utils.spectral_fitting import LMFIT_min_function, make_LMFIT_params
from lmfit import Minimizer, report_fit
```

(continues on next page)

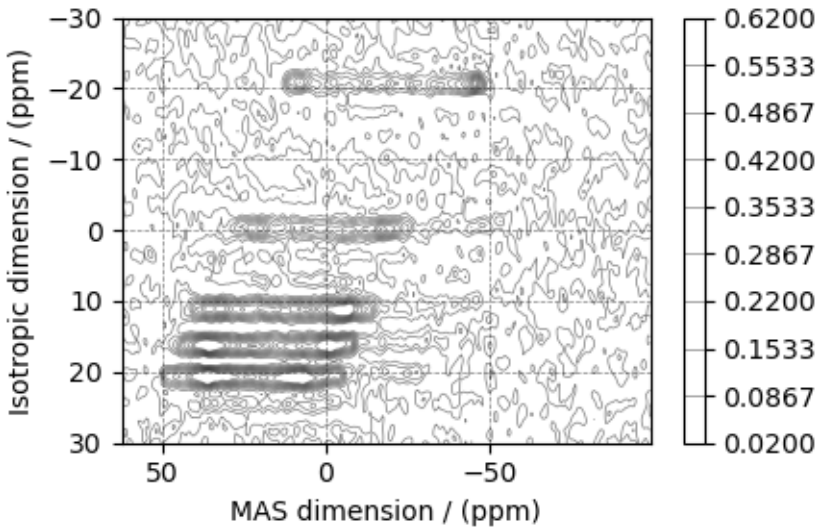
[1] Grandinetti, P. J., Baltisberger, J. H., Farnan, I., Stebbins, J. F., Werner, U. and Pines, A. Solid-State ^{17}O Magic-Angle and Dynamic-Angle Spinning NMR Study of the SiO_2 Polymorph Coesite, J. Phys. Chem. 1995, 99, 32, 12341-12348. DOI: 10.1021/j100032a045

(continued from previous page)

```
# global plot configuration
mpl.rcParams["figure.figsize"] = [4.5, 3.0]
```

Import the dataset

```
filename = "https://sandbox.zenodo.org/record/687656/files/DASCoesite.csdf"
experiment = cp.load(filename)

# For spectral fitting, we only focus on the real part of the complex dataset
experiment = experiment.real

# Convert the coordinates along each dimension from Hz to ppm.
_ = [item.to("ppm", "nmr_frequency_ratio") for item in experiment.dimensions]

# Normalize the spectrum
experiment /= experiment.max()

# plot of the dataset.
levels = (np.arange(10) + 0.3) / 15  # contours are drawn at these levels.
ax = plt.subplot(projection="csdm")
cb = ax.contour(experiment, colors="k", levels=levels, alpha=0.5, linewidths=0.5)
plt.colorbar(cb)
ax.invert_xaxis()
ax.set_ylim(30, -30)
plt.tight_layout()
plt.show()
```



Create a fitting model

The fitting model includes the Simulator and SignalProcessor objects. First, create the Simulator object.

```
# Create the guess sites and spin systems.
# default unit of isotropic_chemical_shift is ppm and Cq is Hz.
shifts = [29, 41, 57, 53, 58]  # in ppm
Cq = [6.1e6, 5.4e6, 5.5e6, 5.5e6, 5.1e6]  # in  Hz
eta = [0.1, 0.2, 0.1, 0.1, 0.3]
abundance = [1, 1, 2, 2, 2]

spin_systems = single_site_system_generator(
    isotopes="17O",
    isotropic_chemical_shifts=shifts,
    quadrupolar={"Cq": Cq, "eta": eta},
    abundance=abundance,
)

# Create the DAS method.
# Get the spectral dimension paramters from the experiment.
spectral_dims = get_spectral_dimensions(experiment)
```

```
das = Method2D(
    channels=["17O"],
    magnetic_flux_density=11.7,  # in T
    spectral_dimensions=[
        {
            **spectral_dims[0],
            "events": [
                {"fraction": 0.5, "rotor_angle": 37.38 * 3.14159 / 180},
                {"fraction": 0.5, "rotor_angle": 79.19 * 3.14159 / 180},
            ],
        },
        # The last spectral dimension block is the direct-dimension
        {**spectral_dims[1], "events": [{"rotor_angle": 54.735 * 3.14159 / 180}]},
    ],
    experiment=experiment,  # also add the measurement to the method.
)

# Optimize the script by pre-setting the transition pathways for each spin system from
# the das method.
for sys in spin_systems:
    sys.transition_pathways = das.get_transition_pathways(sys)
```

```
# Create the Simulator object and add the method and spin system objects.
sim = Simulator()
sim.spin_systems = spin_systems  # add the spin systems
sim.methods = [das]  # add the method
sim.run()
```

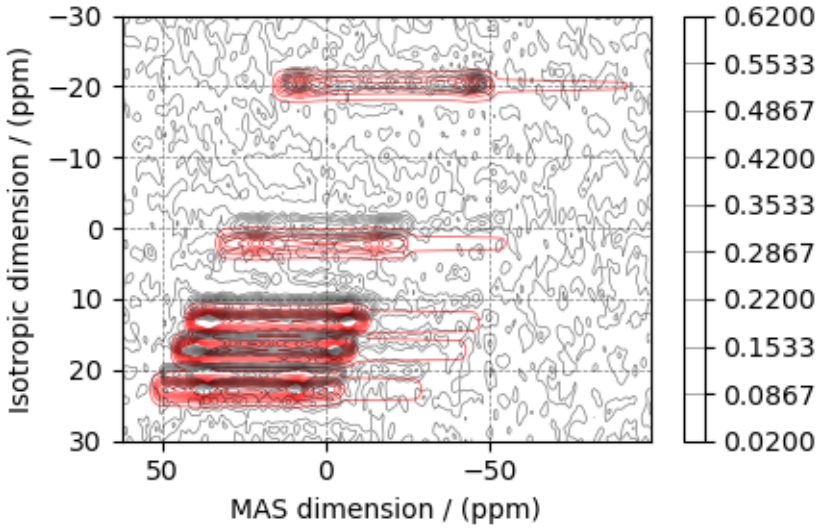
```
# Add Post simulation processing.
processor = sp.SignalProcessor(
    operations=[
        # Gaussian convolution along both dimensions.
        sp.IFFT(dim_index=(0, 1)),
        apo.Gaussian(FWHM="0.15 kHz", dim_index=0),
        apo.Gaussian(FWHM="0.15 kHz", dim_index=1),
```

(continues on next page)

(continued from previous page)

```
        sp.FFT(dim_index=(0, 1)),
        sp.Scale(factor=1 / 8),
    ]
)
# Apply post simulation operations.
processed_data = processor.apply_operations(data=sim.methods[0].simulation).real
```

```
# The plot of the simulation after signal processing.
ax = plt.subplot(projection="csdm")
ax.contour(processed_data, colors="r", levels=levels, alpha=0.75, linewidths=0.5)
cb = ax.contour(experiment, colors="k", levels=levels, alpha=0.5, linewidths=0.5)
plt.colorbar(cb)
ax.invert_xaxis()
ax.set_ylim(30, -30)
plt.tight_layout()
plt.show()
```



Least-squares minimization with LMFIT

First, create the fitting parameters. Use the *make_LMFIT_params()* (page 191) for a quick setup.

```
params = make_LMFIT_params(sim, processor)

# Here, we fix the abundance parameters to their initial value.
for i in range(5):
    params[f"sys_{i}_abundance"].vary = False

params.pretty_print()
```

Out:

```
Name                                    Value      Min      Max    Stderr     Vary     Expr␣
↪Brute_Step
operation_1_Gaussian_FWHM                0.15     -inf      inf      None     True     None    ␣
↪None
```

(continues on next page)

(continued from previous page)

```
operation_2_Gaussian_FWHM                    0.15      -inf       inf      None      True      None  ␣
↪None
operation_4_Scale_factor                    0.125      -inf       inf      None      True      None  ␣
↪None
sys_0_abundance                              12.5         0       100      None     False      None  ␣
↪None
sys_0_site_0_isotropic_chemical_shift          29      -inf       inf      None      True      None  ␣
↪None
sys_0_site_0_quadrupolar_Cq               6.1e+06      -inf       inf      None      True      None  ␣
↪None
sys_0_site_0_quadrupolar_eta                  0.1         0         1      None      True      None  ␣
↪None
sys_1_abundance                              12.5         0       100      None     False      None  ␣
↪None
sys_1_site_0_isotropic_chemical_shift          41      -inf       inf      None      True      None  ␣
↪None
sys_1_site_0_quadrupolar_Cq               5.4e+06      -inf       inf      None      True      None  ␣
↪None
sys_1_site_0_quadrupolar_eta                  0.2         0         1      None      True      None  ␣
↪None
sys_2_abundance                              25          0       100      None     False      None  ␣
↪None
sys_2_site_0_isotropic_chemical_shift          57      -inf       inf      None      True      None  ␣
↪None
sys_2_site_0_quadrupolar_Cq               5.5e+06      -inf       inf      None      True      None  ␣
↪None
sys_2_site_0_quadrupolar_eta                  0.1         0         1      None      True      None  ␣
↪None
sys_3_abundance                              25          0       100      None     False      None  ␣
↪None
sys_3_site_0_isotropic_chemical_shift          53      -inf       inf      None      True      None  ␣
↪None
sys_3_site_0_quadrupolar_Cq               5.5e+06      -inf       inf      None      True      None  ␣
↪None
sys_3_site_0_quadrupolar_eta                  0.1         0         1      None      True      None  ␣
↪None
sys_4_abundance                              25          0       100      None     False 100-sys_0_
↪abundance-sys_1_abundance-sys_2_abundance-sys_3_abundance      None
sys_4_site_0_isotropic_chemical_shift          58      -inf       inf      None      True      None  ␣
↪None
sys_4_site_0_quadrupolar_Cq               5.1e+06      -inf       inf      None      True      None  ␣
↪None
sys_4_site_0_quadrupolar_eta                  0.3         0         1      None      True      None  ␣
↪None
```

Run the minimization using LMFIT

```
minner = Minimizer(LMFIT_min_function, params, fcn_args=(sim, processor))
result = minner.minimize()
report_fit(result)
```

Out:

```
[[Fit Statistics]]
    # fitting method   = leastsq
    # function evals   = 371
    # data points      = 131072
```

(continues on next page)

(continued from previous page)

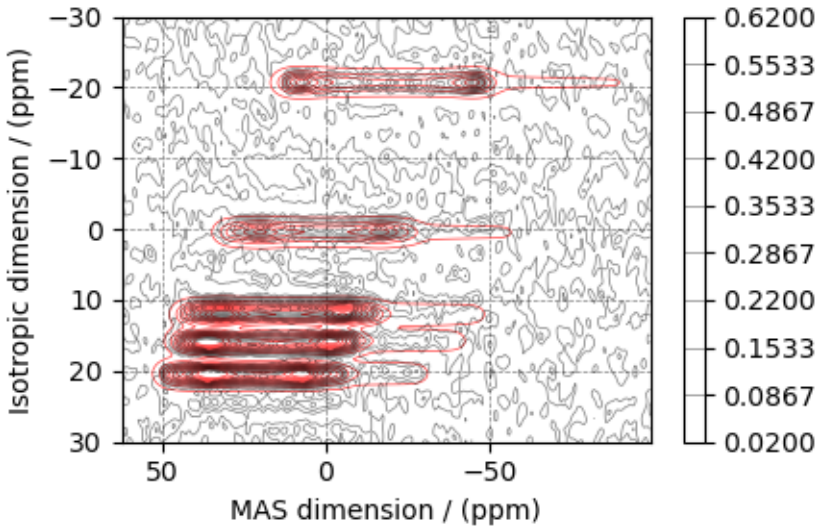
```
    # variables        = 18
    chi-square         = 363.301226
    reduced chi-square = 0.00277215
    Akaike info crit   = -771751.293
    Bayesian info crit = -771575.190
[[Variables]]
    sys_0_site_0_isotropic_chemical_shift:  27.4394744 +/- 0.16037405 (0.58%) (init = 29)
    sys_0_site_0_quadrupolar_Cq:            6044709.32 +/- 10092.8744 (0.17%) (init = 6100000)
    sys_0_site_0_quadrupolar_eta:           0.09058695 +/- 0.00541829 (5.98%) (init = 0.1)
    sys_0_abundance:                        12.5 (fixed)
    sys_1_site_0_isotropic_chemical_shift:  40.2138886 +/- 0.20780233 (0.52%) (init = 41)
    sys_1_site_0_quadrupolar_Cq:            5453223.41 +/- 14094.8807 (0.26%) (init = 5400000)
    sys_1_site_0_quadrupolar_eta:           0.20537807 +/- 0.00544236 (2.65%) (init = 0.2)
    sys_1_abundance:                        12.5 (fixed)
    sys_2_site_0_isotropic_chemical_shift:  54.3501005 +/- 0.09034982 (0.17%) (init = 57)
    sys_2_site_0_quadrupolar_Cq:            5394012.50 +/- 6386.20846 (0.12%) (init = 5500000)
    sys_2_site_0_quadrupolar_eta:           0.17076714 +/- 0.00278735 (1.63%) (init = 0.1)
    sys_2_abundance:                        25 (fixed)
    sys_3_site_0_isotropic_chemical_shift:  52.3588929 +/- 0.10898489 (0.21%) (init = 53)
    sys_3_site_0_quadrupolar_Cq:            5497997.60 +/- 7105.04765 (0.13%) (init = 5500000)
    sys_3_site_0_quadrupolar_eta:           0.21373888 +/- 0.00275690 (1.29%) (init = 0.1)
    sys_3_abundance:                        25 (fixed)
    sys_4_site_0_isotropic_chemical_shift:  54.7343082 +/- 0.10652839 (0.19%) (init = 58)
    sys_4_site_0_quadrupolar_Cq:            5042385.28 +/- 7655.24974 (0.15%) (init = 5100000)
    sys_4_site_0_quadrupolar_eta:           0.29135745 +/- 0.00309323 (1.06%) (init = 0.3)
    sys_4_abundance:                        25.0000000 +/- 0.00000000 (0.00%) == '100-sys_0_
↪abundance-sys_1_abundance-sys_2_abundance-sys_3_abundance'
    operation_1_Gaussian_FWHM:              0.39458618 +/- 0.00896349 (2.27%) (init = 0.15)
    operation_2_Gaussian_FWHM:              0.15185217 +/- 4.4454e-04 (0.29%) (init = 0.15)
    operation_4_Scale_factor:               0.00977120 +/- 2.7355e-05 (0.28%) (init = 0.125)
[[Correlations]] (unreported correlations are < 0.100)
    C(sys_3_site_0_isotropic_chemical_shift, sys_3_site_0_quadrupolar_Cq)  =  0.810
    C(sys_0_site_0_isotropic_chemical_shift, sys_0_site_0_quadrupolar_Cq)  =  0.801
    C(sys_1_site_0_isotropic_chemical_shift, sys_1_site_0_quadrupolar_Cq)  =  0.792
    C(sys_4_site_0_isotropic_chemical_shift, sys_4_site_0_quadrupolar_Cq)  =  0.792
    C(sys_2_site_0_isotropic_chemical_shift, sys_2_site_0_quadrupolar_Cq)  =  0.789
    C(operation_2_Gaussian_FWHM, operation_4_Scale_factor)                 =  0.467
    C(sys_2_site_0_quadrupolar_eta, operation_1_Gaussian_FWHM)             = -0.362
    C(sys_0_site_0_quadrupolar_eta, operation_1_Gaussian_FWHM)             = -0.347
    C(sys_3_site_0_quadrupolar_eta, operation_1_Gaussian_FWHM)             = -0.191
    C(sys_0_site_0_isotropic_chemical_shift, sys_0_site_0_quadrupolar_eta) =  0.147
    C(sys_4_site_0_quadrupolar_Cq, operation_4_Scale_factor)               =  0.144
    C(operation_1_Gaussian_FWHM, operation_4_Scale_factor)                 =  0.144
    C(sys_2_site_0_isotropic_chemical_shift, sys_2_site_0_quadrupolar_eta) =  0.136
    C(sys_0_site_0_quadrupolar_eta, sys_2_site_0_quadrupolar_eta)          =  0.133
    C(sys_4_site_0_isotropic_chemical_shift, operation_4_Scale_factor)     =  0.126
    C(sys_4_site_0_quadrupolar_eta, operation_1_Gaussian_FWHM)             = -0.126
    C(sys_3_site_0_quadrupolar_Cq, operation_4_Scale_factor)               =  0.119
    C(sys_1_site_0_quadrupolar_eta, operation_1_Gaussian_FWHM)             = -0.115
    C(sys_2_site_0_quadrupolar_Cq, operation_4_Scale_factor)               =  0.109
    C(sys_2_site_0_isotropic_chemical_shift, operation_1_Gaussian_FWHM)    = -0.103
```

Simulate the spectrum corresponding to the optimum parameters

```
sim.run()
processed_data = processor.apply_operations(data=sim.methods[0].simulation).real
```

Plot the spectrum

```
ax = plt.subplot(projection="csdm")
ax.contour(processed_data, colors="r", levels=levels, alpha=0.75, linewidths=0.5)
cb = ax.contour(experiment, colors="k", levels=levels, alpha=0.5, linewidths=0.5)
plt.colorbar(cb)
ax.invert_xaxis()
ax.set_ylim(30, -30)
plt.tight_layout()
plt.show()
```



Total running time of the script: (1 minutes 50.062 seconds)

4.3 Benchmark

One of the objectives in the design of the `mrsimulator` library is to enable fast NMR spectrum simulation. For this, we have put a considerable effort into the optimization of the library. The following benchmark shows the performance of the library in computing the solid-state NMR spectra from single-site spin systems for the shift and quadrupolar tensor interactions at static and MAS conditions.

Benchmark specs

The benchmarks were performed on a 2.3 GHz Quad-Core Intel Core i5 Laptop using 8 GB 2133 MHz LPDDR3 memory. For consistent benchmarking, 1000 single-site spin systems were constructed, where the tensor parameters of the sites (*zeta* and *eta* for the shielding tensor, and *Cq* and *eta* for the quadrupolar tensor) were randomly populated. The execution time for this setup was recorded. and the process repeated 70 times. The reported value is the mean and the standard deviation.

All calculations were performed using the default Simulator *config* (page 152) attribute values.

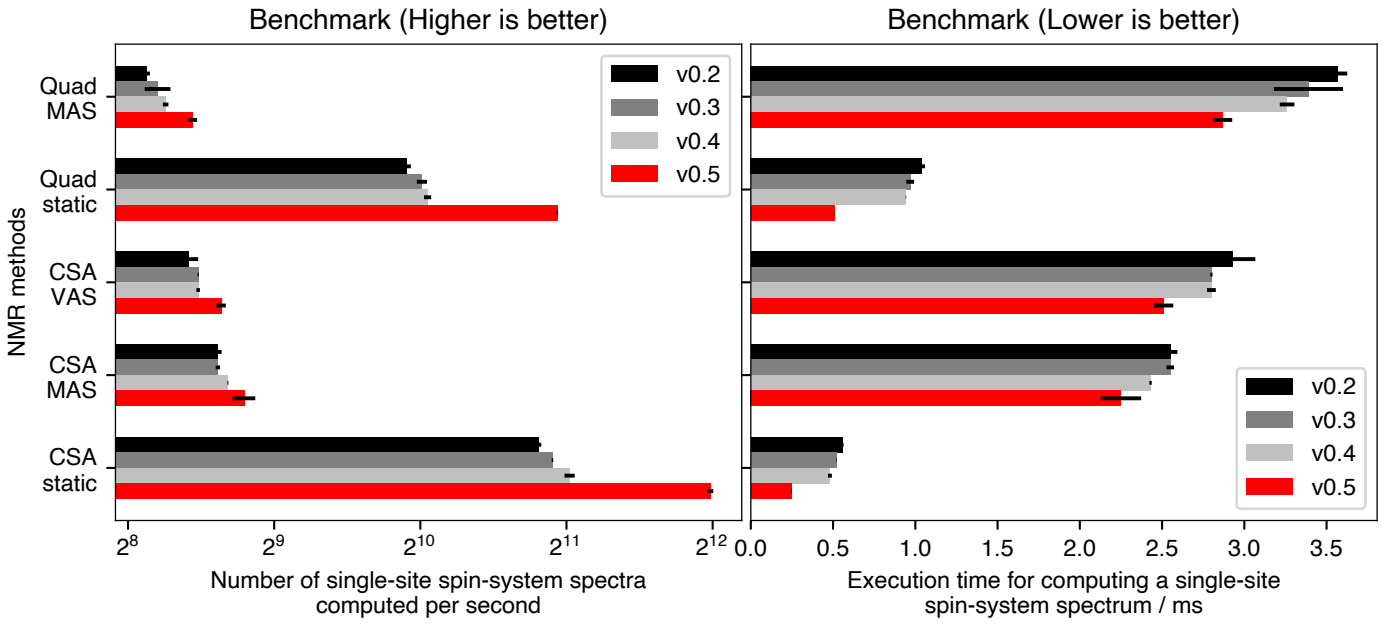


Figure 4.1: (Left) The number of single-site spin systems computer per seconds. (Right) The execution time (in ms) in computing spectrum from a single-site spin system.

CHAPTER

FIVE

THEORY

5.1 How does `mrsimulator` work?

The NMR spectral simulation in `mrsimulator` is based on *Symmetry Pathways in Solid-State NMR* by Grandinetti *et. al.*[1]

5.1.1 Introduction to NMR frequency components

The nuclear magnetic resonance (NMR) frequency, $\Omega(\Theta, i, j)$, for the $|i\rangle \rightarrow |j\rangle$ transition, where $|i\rangle$ and $|j\rangle$ are the eigenstates of the stationary-state semi-classical Hamiltonian, can be written as a sum of frequency components,

$$\Omega(\Theta, i, j) = \sum_k \Omega_k(\Theta, i, j), \tag{5.1}$$

where Θ is the sample's lattice spatial orientation described with the Euler angles $\Theta = (\alpha, \beta, \gamma)$, and Ω_k is the frequency component from the k^{th} interaction of the stationary-state semi-classical Hamiltonian.

Each frequency component, $\Omega_k(\Theta, i, j)$, is separated into three parts,

$$\Omega_k(\Theta, i, j) = \omega_k \, \Xi_L^{(k)}(\Theta) \, \xi_L^{(k)}(i, j), \tag{5.2}$$

where ω_k is the size of the k^{th} frequency component, and $\Xi_L^{(k)}(\Theta)$ and $\xi_L^{(k)}(i, j)$ are the sample's spatial orientation and quantized NMR transition functions corresponding to the L^{th} rank spatial and spin irreducible spherical tensors, respectively.

The spatial orientation function, $\Xi_L^{(k)}(\Theta)$, in Eq. (5.2), is defined in the laboratory frame, where the z-axis is the direction of the external magnetic field. This function is the spatial contribution to the observed frequency component arising from the rotation of the L^{th}-rank irreducible tensor, $\varrho_{L,n}^{(k)}$, from the principal axis system, to the lab frame via Wigner rotation which follows,

$$\Xi_L^{(k)}(\Theta) = \sum_{n_0=-L}^{L} D_{n_0,0}^L(\Theta_0) \sum_{n_1=-L}^{L} D_{n_1,n_0}^L(\Theta_1) \; ... \sum_{n_i=-L}^{L} D_{n_i,n}^L(\Theta_i) \;\; \varrho_{L,n}^{(k)}. \tag{5.3}$$

Here, the term $D_{n_i,n_j}^L(\Theta)$ is the Wigner rotation matrix element, generically denoted as,

$$D_{n_i,n_j}^L(\Theta) = e^{-in_i\alpha} d_{n_i,n_j}^L(\beta) e^{-in_j\gamma}, \tag{5.4}$$

where $d_{n_i,n_j}^L(\beta)$ is Wigner small d element.

---

[1] Grandinetti, P. J., Ash, J. T., Trease, N. M. Symmetry pathways in solid-state NMR, PNMRS 2011 59, 2, 121-196. DOI: 10.1016/j.pnmrs.2010.11.003

In the case of the single interaction Hamiltonian, that is, in the absence of cross-terms, `mrsimulator` further defines the product of the size of the k^{th} frequency component, ω_k, and the L^{th}-rank irreducible tensor components, $\varrho^{(k)}_{L,n}$, in the principal axis system of the interaction tensor, $\boldsymbol{\rho}^{(\lambda)}$, as the scaled spatial orientation tensor (sSOT) components,

$$\varsigma^{(k)}_{L,n} = \omega_k \varrho^{(k)}_{L,n}, \tag{5.5}$$

of rank L, also defined in the principal axis system of the interaction tensor, $\boldsymbol{\rho}^{(\lambda)}$. Using Eqs. (5.3) and (5.5), we re-express Eq. (5.2) as

$$\Omega_k(\Theta, i, j) = \sum_{n_0=-L}^{L} D^L_{n_0,0}(\Theta_0) \sum_{n_1=-L}^{L} D^L_{n_1,n_0}(\Theta_1) \ldots \sum_{n_i=-L}^{L} D^L_{n_i,n}(\Theta_i)\ \varpi^{(k)}_{L,n}(i,j), \tag{5.6}$$

where

$$\varpi^{(k)}_{L,n}(i,j) = \varsigma^{(k)}_{L,n}\ \xi^{(k)}_L(i,j) \tag{5.7}$$

is the frequency tensor components (FT) of rank L, defined in the principal axis system of the interaction tensor and corresponds to the $|i\rangle \rightarrow |j\rangle$ spin transition.

5.1.2 Scaled spatial orientation tensor (sSOT) components in PAS, $\varsigma^{(k)}_{L,n}$

Single nucleus scaled spatial orientation tensor components

Nuclear shielding interaction

The nuclear shielding tensor, $\boldsymbol{\rho}^{(\sigma)}$, is a second rank reducible tensor which can be decomposed into a sum of the zeroth-rank isotropic, first-rank anti-symmetric and second-rank traceless symmetric irreducible spherical tensors. In the principal axis system, the zeroth-rank, $\rho^{(\sigma)}_{0,0}$ and the second-rank, $\rho^{(\sigma)}_{2,n}$, irreducible tensors follow,

$$\rho^{(\sigma)}_{0,0} = -\sqrt{3}\sigma_{\text{iso}}, \quad \rho^{(\sigma)}_{2,0} = \sqrt{\tfrac{3}{2}}\zeta_\sigma, \quad \rho^{(\sigma)}_{2,\pm1} = 0, \quad \rho^{(\sigma)}_{2,\pm2} = -\tfrac{1}{2}\eta_\sigma\zeta_\sigma, \tag{5.8}$$

where $\sigma_{\text{iso}}, \zeta_\sigma$, and η_σ are the isotropic nuclear shielding, shielding anisotropy, and shielding asymmetry of the site, respectively. The shielding anisotropy, and asymmetry are defined using Haeberlen notation.

First-order perturbation

The size of the frequency component, ω_k, from the first-order perturbation expansion of Nuclear shielding Hamiltonian is $\omega_0 = -\gamma B_0$, where ω_0 is the Larmor angular frequency of the nucleus, and γ, B_0 are the gyromagnetic ratio of the nucleus and the macroscopic magnetic flux density of the applied external magnetic field, respectively. The relation between $\varrho^{(\sigma)}_{L,n}$ and $\rho^{(\sigma)}_{L,n}$ follows,

$$\begin{aligned} \varrho^{(\sigma)}_{0,0} &= -\frac{1}{\sqrt{3}}\rho^{(\sigma)}_{0,0} \\ \varrho^{(\sigma)}_{2,n} &= \sqrt{\frac{2}{3}}\rho^{(\sigma)}_{2,n} \end{aligned} \tag{5.9}$$

Table 5.1: A list of scaled spatial orientation tensors in the principal axis system of the nuclear shielding tensor, $\varsigma^{(k)}_{L,n}$, from Eq. (5.5) of rank L resulting from the Mth order perturbation expansion of the Nuclear shielding Hamiltonian is presented.

Order, M	Rank, L	$\varsigma^{(k)}_{L,n} = \omega_k \varrho^{(k)}_{L,n}$
1	0	$\varsigma^{(\sigma)}_{0,0} = -\omega_0 \sigma_{\text{iso}}$
1	2	$\varsigma^{(\sigma)}_{2,0} = -\omega_0 \zeta_\sigma$, $\varsigma^{(\sigma)}_{2,\pm1} = 0$, $\varsigma^{(\sigma)}_{2,\pm2} = \frac{1}{\sqrt{6}} \omega_0 \eta_\sigma \zeta_\sigma$

Electric quadrupole interaction

The electric field gradient (efg) tensor, $\boldsymbol{\rho}^{(q)}$, is also a second-rank tensor, however, unlike the nuclear shielding tensor, the efg tensor is always a symmetric second-rank irreducible tensor. In the principal axis system, this tensor is given as,

$$\rho^{(q)}_{2,0} = \sqrt{\tfrac{3}{2}} \zeta_q, \quad \rho^{(q)}_{2,\pm1} = 0, \quad \rho^{(q)}_{2,\pm2} = -\tfrac{1}{2} \eta_q \zeta_q, \tag{5.10}$$

where ζ_q, and η_q are the efg tensor anisotropy, and asymmetry of the site, respectively. The efg anisotropy, and asymmetry are defined using Haeberlen convention.

First-order perturbation

The size of the frequency component from the first-order perturbation expansion of Electric quadrupole Hamiltonian is $\omega_k = \omega_q$, where $\omega_q = \frac{6\pi C_q}{2I(2I-1)}$ is the quadrupole splitting angular frequency. Here, C_q is the quadrupole coupling constant, and I is the spin quantum number of the quadrupole nucleus. The relation between $\varrho^{(q)}_{L,n}$ and $\rho^{(q)}_{L,n}$ follows,

$$\varrho^{(q)}_{2,n} = \frac{1}{3\zeta_q} \rho^{(q)}_{2,n}. \tag{5.11}$$

Second-order perturbation

The size of the frequency component from the second-order perturbation expansion of Electric quadrupole Hamiltonian is $\omega_k = \frac{\omega_q^2}{\omega_0}$, where ω_0 is the Larmor angular frequency of the quadrupole nucleus. The relation between $\varrho^{(qq)}_{L,n}$ and $\rho^{(q)}_{L,n}$ follows,

$$\varrho^{(qq)}_{L,n} = \frac{1}{9\zeta_q^2} \sum_{m=-2}^{2} \langle L\ n \mid 2\ 2\ m\ n-m \rangle\ \rho^{(q)}_{2,m}\ \rho^{(q)}_{2,n-m}, \tag{5.12}$$

where $\langle L\ M \mid l_1\ l_2\ m_1\ m_2 \rangle$ is the Clebsch Gordan coefficient.

Table 5.2: A list of scaled spatial orientation tensors in the principal axis system of the efg tensor, $\varsigma^{(k)}_{L,n}$, from Eq. (5.5) of rank L resulting from the Mth order perturbation expansion of the Electric Quadrupole Hamiltonian is presented.

Order, M	Rank, L	$\varsigma^{(k)}_{L,n} = \omega_k \varrho^{(k)}_{L,n}$
1	2	$\varsigma^{(q)}_{2,0} = \frac{1}{\sqrt{6}}\omega_q$, $\varsigma^{(q)}_{2,\pm1} = 0$, $\varsigma^{(q)}_{2,\pm2} = -\frac{1}{6}\eta_q\omega_q$
2	0	$\varsigma^{(qq)}_{0,0} = \frac{\omega_q^2}{\omega_0}\frac{1}{6\sqrt{5}}\left(\frac{\eta_q^2}{3}+1\right)$
2	2	$\varsigma^{(qq)}_{2,0} = \frac{\omega_q^2}{\omega_0}\frac{\sqrt{2}}{6\sqrt{7}}\left(\frac{\eta_q^2}{3}-1\right)$, $\varsigma^{(qq)}_{2,\pm1} = 0$, $\varsigma^{(qq)}_{2,\pm2} = -\frac{\omega_q^2}{\omega_0}\frac{1}{3\sqrt{21}}\eta_q$
2	4	$\varsigma^{(qq)}_{4,0} = \frac{\omega_q^2}{\omega_0}\frac{1}{\sqrt{70}}\left(\frac{\eta_q^2}{18}+1\right)$, $\varsigma^{(qq)}_{4,\pm1} = 0$, $\varsigma^{(qq)}_{4,\pm2} = -\frac{\omega_q^2}{\omega_0}\frac{1}{6\sqrt{7}}\eta_q$, $\varsigma^{(qq)}_{4,\pm3} = 0$, $\varsigma^{(qq)}_{4,\pm4} = \frac{\omega_q^2}{\omega_0}\frac{1}{36}\eta_q^2$

5.1.3 Spin transition functions, $\xi^{(k)}_L(i,j)$

The spin transition function is typically manipulated via the coupling of the nuclear magnetic dipole moment with the oscillating external magnetic field from the applied radio-frequency pulse. Considering the strength of the external magnetic rf field is orders of magnitude larger than the internal spin-couplings, the manipulation of spin transition functions are described using the orthogonal rotation subgroups.

Single nucleus spin transition functions

Table 5.3: A list of single nucleus spin transition functions, $\xi^{(k)}_L(i,j)$.

$\xi^{(k)}_L(i,j)$	Rank, L	Value	Description
$\mathbb{s}(i,j)$	0	0	$\left\langle j\vert\hat{T}_{00}\vert j\right\rangle - \left\langle i\vert\hat{T}_{00}\vert i\right\rangle$
$\mathbb{p}(i,j)$	1	$j-i$	$\left\langle j\vert\hat{T}_{10}\vert j\right\rangle - \left\langle i\vert\hat{T}_{10}\vert i\right\rangle$
$\mathbb{d}(i,j)$	2	$\sqrt{\frac{3}{2}}\left(j^2-i^2\right)$	$\left\langle j\vert\hat{T}_{20}\vert j\right\rangle - \left\langle i\vert\hat{T}_{20}\vert i\right\rangle$
$\mathbb{f}(i,j)$	3	$\frac{1}{\sqrt{10}}[5(j^3-i^3)+(1-3I(I+1))(j-i)]$	$\left\langle j\vert\hat{T}_{30}\vert j\right\rangle - \left\langle i\vert\hat{T}_{30}\vert i\right\rangle$

Here, $\hat{T}_{L,k}(\mathbf{I})$ are the irreducible spherical tensor operators of rank L, and $k \in [-L, L]$. In terms of the tensor product of the Cartesian operators, the above spherical tensors are expressed as follows,

Spherical tensor operator	Representation in Cartesian operators
$\hat{T}_{0,0}(\mathbf{I})$	$\hat{1}$
$\hat{T}_{1,0}(\mathbf{I})$	$\hat{I}_z$
$\hat{T}_{2,0}(\mathbf{I})$	$\frac{1}{\sqrt{6}}\left[3\hat{I}_z^2 - I(I+1)\hat{1}\right]$
$\hat{T}_{3,0}(\mathbf{I})$	$\frac{1}{\sqrt{10}}\left[5\hat{I}_z^3 + (1-3I(I+1))\,\hat{I}_z\right]$

where I is the spin quantum number of the nucleus and $\hat{\mathbf{1}}$ is the identity operator.

Table 5.4: A list of composite single nucleus spin transition functions, $\xi_L^{(k)}(i,j)$. Here, *I* is the spin quantum number of the nucleus.

$\xi_L^{(k)}(i,j)$	Value
$\mathbb{c}_0(i,j)$	$\frac{4}{\sqrt{125}}\left[I(I+1)-\frac{3}{4}\right]\mathbb{p}(i,j)+\sqrt{\frac{18}{25}}\mathbb{f}(i,j)$
$\mathbb{c}_2(i,j)$	$\sqrt{\frac{2}{175}}\left[I(I+1)-\frac{3}{4}\right]\mathbb{p}(i,j)-\frac{6}{\sqrt{35}}\mathbb{f}(i,j)$
$\mathbb{c}_4(i,j)$	$-\sqrt{\frac{18}{875}}\left[I(I+1)-\frac{3}{4}\right]\mathbb{p}(i,j)-\frac{17}{\sqrt{175}}\mathbb{f}(i,j)$

5.1.4 Frequency tensor components (FT) in PAS, $\varpi_{L,n}^{(k)}(i,j)$

Single nucleus frequency tensor components

Table 5.5: The table presents a list of frequency tensors defined in the principal axis system of the respective interaction tensor from Eq. (5.7), $\varpi_{L,n}^{(k)}(i,j)$, of rank L resulting from the Mth order perturbation expansion of the interaction Hamiltonians supported in `mrsimulator`.

Interaction	Order, M	Rank, L	$\varpi_{L,n}^{(k)}(i,j)$
Nuclear shielding	1	0	$\varpi_{0,0}^{(\sigma)}(i,j)=\varsigma_{0,0}^{(\sigma)}\ \mathbb{p}(i,j)$
Nuclear shielding	1	2	$\varpi_{2,n}^{(\sigma)}(i,j)=\varsigma_{2,n}^{(\sigma)}\ \mathbb{p}(i,j)$
Electric Quadrupole	1	2	$\varpi_{2,n}^{(q)}(i,j)=\varsigma_{2,n}^{(q)}\ \mathbb{d}(i,j)$
Electric Quadrupole	2	0	$\varpi_{0,0}^{(qq)}(i,j)=\varsigma_{0,0}^{(qq)}\ \mathbb{c}_0(i,j)$
Electric Quadrupole	2	2	$\varpi_{2,n}^{(qq)}(i,j)=\varsigma_{2,n}^{(qq)}\ \mathbb{c}_2(i,j)$
Electric Quadrupole	2	4	$\varpi_{4,n}^{(qq)}(i,j)=\varsigma_{4,n}^{(qq)}\ \mathbb{c}_4(i,j)$

5.2 Models

5.2.1 Czjzek distribution

A Czjzek distribution model[1] is a random distribution of the second-rank traceless symmetric tensors about a zero tensor. An explicit form of a traceless symmetric second-rank tensor, **S**, in Cartesian basis, follows,

$$\mathbf{S}=\begin{bmatrix} S_{xx} & S_{xy} & S_{xz} \\ S_{xy} & S_{yy} & S_{yz} \\ S_{xz} & S_{yz} & S_{zz} \end{bmatrix}, \tag{5.13}$$

[1] Czjzek, G., Fink, J., Götz, F., Schmidt, H., Coey, J. M. D., Atomic coordination and the distribution of electric field gradients in amorphous solids Phys. Rev. B (1981) 23 2513-30. DOI: 10.1103/PhysRevB.23.2513

where $S_{xx}+S_{yy}+S_{zz}=0$. The elements of the above Cartesian tensor, S_{ij}, can be decomposed into second-rank irreducible spherical tensor components[3], $R_{2,k}$, following

$$
\begin{aligned}
S_{xx} &= \frac{1}{2}(R_{2,2} + R_{2,-2}) - \frac{1}{\sqrt{6}}R_{2,0}, \\
S_{xy} = S_{yx} &= -\frac{i}{2}(R_{2,2} - R_{2,-2}), \\
S_{yy} &= -\frac{1}{2}(R_{2,2} + R_{2,-2}) - \frac{1}{\sqrt{6}}R_{2,0}, \\
S_{xz} = S_{zx} &= -\frac{1}{2}(R_{2,1} - R_{2,-1}), \\
S_{zz} &= \sqrt{\frac{2}{3}}R_{2,0}, \\
S_{yz} = S_{zy} &= \frac{i}{2}(R_{2,1} + R_{2,-1})
\end{aligned}
\tag{5.14}
$$

In the Czjzek model, the distribution of the second-rank traceless symmetric tensor is based on the assumption of a random distribution of the five irreducible spherical tensor components, $R_{2,k}$, drawn from an uncorrelated five-dimensional multivariate normal distribution. Since $R_{2,k}$ components are complex, random sampling is performed on the equivalent real tensor components, which are a linear combination of $R_{2,k}$, and are given as

$$
\begin{aligned}
U_1 &= \frac{1}{\sqrt{6}}R_{2,0}, \\
U_2 &= -\frac{1}{\sqrt{12}}(R_{2,1} - R_{2,-1}), \\
U_3 &= \frac{i}{\sqrt{12}}(R_{2,1} + R_{2,-1}), \\
U_4 &= -\frac{i}{\sqrt{12}}(R_{2,2} - R_{2,-2}), \\
U_5 &= \frac{1}{\sqrt{12}}(R_{2,2} + R_{2,-2})
\end{aligned}
\tag{5.14}
$$

where U_i forms an ortho-normal basis. The components, U_i, are drawn from a five-dimensional uncorrelated multivariate normal distribution with zero mean and covariance matrix, $\Lambda = \sigma^2\mathbf{I}_5$, where $\mathbf{I}_5$ is a 5×5 identity matrix and σ is the standard deviation.

In terms of U_i, the traceless second-rank symmetric Cartesian tensor elements, S_{ij}, follows

$$
\begin{aligned}
S_{xx} &= \sqrt{3}U_5 - U_1, \\
S_{xy} = S_{yx} &= \sqrt{3}U_4, \\
S_{yy} &= -\sqrt{3}U_5 - U_1, \\
S_{xz} = S_{zx} &= \sqrt{3}U_2, \\
S_{zz} &= 2U_1, \\
S_{yz} = S_{zy} &= \sqrt{3}U_3
\end{aligned}
\tag{5.14}
$$

and the explicit matrix form of $\mathbf{S}$ is

$$
\mathbf{S} = \begin{bmatrix} \sqrt{3}U_5 - U_1 & \sqrt{3}U_4 & \sqrt{3}U_2 \\ \sqrt{3}U_4 & -\sqrt{3}U_5 - U_1 & \sqrt{3}U_3 \\ \sqrt{3}U_2 & \sqrt{3}U_3 & 2U_1 \end{bmatrix}. \tag{5.14}
$$

[3] Grandinetti, P. J., Ash, J. T., Trease, N. M. Symmetry pathways in solid-state NMR, PNMRS 2011 59, 2, 121-196. DOI: 10.1016/j.pnmrs.2010.11.003

In a shorthand notation, we denote a Czjzek distribution of second-rank traceless symmetric tensor as $S_C(\sigma)$.

5.2.2 Extended Czjzek distribution

An Extended Czjzek distribution model[2] is a random perturbation of the second-rank traceless symmetric tensors about a non-zero tensor, which is given as

$$S_T = S(0) + \rho S_C(\sigma = 1), \tag{5.15}$$

where S_T is the total tensor, $S(0)$ is the non-zero dominant second-rank tensor, $S_C(\sigma = 1)$ is the Czjzek random model attributing to the random perturbation of the tensor about the dominant tensor, $S(0)$, and ρ is the size of the perturbation. Note, in the above equation, the σ parameter from the Czjzek random model, S_C, has no meaning and is set to one. The factor, ρ, is defined as

$$\rho = \frac{||S(0)||\epsilon}{\sqrt{30}}, \tag{5.16}$$

where $\|S(0)\|$ is the 2-norm of the dominant tensor, and ϵ is a fraction.

[2] Caër, G.L., Bureau, B., Massiot, D., An extension of the Czjzek model for the distributions of electric field gradients in disordered solids and an application to NMR spectra of 71Ga in chalcogenide glasses. Journal of Physics: Condensed Matter, (2010), 22. DOI: 10.1088/0953-8984/22/6/065402

CHAPTER

SIX

API AND REFERENCES

6.1 Simulation API

6.1.1 Simulator

class mrsimulator.**Simulator**(*, *name: str = None*, *label: str = None*, *description: str = None*, *spin_systems: List[mrsimulator.spin_system.SpinSystem] = []*, *methods: List[mrsimulator.method.Method] = []*, *config: mrsimulator.simulator.config.ConfigSimulator = ConfigSimulator(number_of_sidebands=64, integration_volume='octant', integration_density=70, decompose_spectrum='none')*, *indexes: list = []*)

Bases: pydantic.main.BaseModel

The simulator class.

spin_systems

The value is a list of NMR spin systems present within the sample, where each spin system is an isolated system. The default value is an empty list.

Example

```
>>> sim = Simulator()
>>> sim.spin_systems = [
...     SpinSystem(sites=[Site(isotope='17O')], abundance=0.015),
...     SpinSystem(sites=[Site(isotope='1H')], abundance=1),
... ]
>>> # or equivalently
>>> sim.spin_systems = [
...     {'sites': [{'isotope': '17O'}], 'abundance': 0.015},
...     {'sites': [{'isotope': '1H'}], 'abundance': 1},
... ]
```

Type A list of *SpinSystem* (page 158) or equivalent dict objects (optional).

methods

The value is a list of NMR methods. The default value is an empty list.

Example

```
>>> from mrsimulator.methods import BlochDecaySpectrum
>>> from mrsimulator.methods import BlochDecayCentralTransitionSpectrum
>>> sim.methods = [
...     BlochDecaySpectrum(channels=['17O'], spectral_width=50000),
...     BlochDecayCentralTransitionSpectrum(channels=['17O'])
... ]
```

Type A list of *Method* (page 170) (optional).

config

The *ConfigSimulator* (page 157) object is used to configure the simulation. The valid attributes of the ConfigSimulator object are

- `number_of_sidebands`,
- `integration_density`,
- `integration_volume`, and
- `decompose_spectrum`

Example

```
>>> from mrsimulator.simulator.config import ConfigSimulator
>>> sim.config = ConfigSimulator(
...     number_of_sidebands=32,
...     integration_density=64,
...     integration_volume='hemisphere',
...     decompose_spectrum='spin_system',
... )
>>> # or equivalently
>>> sim.config = {
...     'number_of_sidebands': 32,
...     'integration_density': 64,
...     'integration_volume': 'hemisphere',
...     'decompose_spectrum': 'spin_system',
... }
```

See *Configuring Simulator object* (page 24) for details.

Type *ConfigSimulator* (page 157) object or equivalent dict object (optional).

name

The value is the name or id of the simulation or sample. The default value is None.

Example

```
>>> sim.name = '1H-17O'
>>> sim.name
'1H-17O'
```

Type str (optional)

label
The value is a label for the simulation or sample. The default value is None.

Example

```
>>> sim.label = 'Test simulator'
>>> sim.label
'Test simulator'
```

Type str (optional)

description
The value is a description of the simulation or sample. The default value is None.

Example

```
>>> sim.description = 'Simulation for sample 1'
>>> sim.description
'Simulation for sample 1'
```

Type str (optional)

Method Documentation

get_isotopes(*spin_I=None*) → set
Set of unique isotopes from the sites within the list of the spin systems corresponding to spin quantum number *I*. If *I* is None, a set of all unique isotopes is returned instead.

Parameters **spin_I** (*float*) – An optional spin quantum number. The valid input are the multiples of 0.5.

Returns A Set.

Example

```
>>> sim.get_isotopes()
{'1H', '27Al', '13C'}
>>> sim.get_isotopes(spin_I=0.5)
{'1H', '13C'}
>>> sim.get_isotopes(spin_I=1.5)
set()
>>> sim.get_isotopes(spin_I=2.5)
{'27Al'}
```

json (*include_methods: bool = False*, *include_version: bool = False*)

Serialize the Simulator object to a JSON compliant python dictionary object where physical quantities are represented as string with a value and a unit.

Parameters

- **include_methods** (*bool*) – If True, the output dictionary will include the serialized method objects. The default value is False.
- **include_version** (*bool*) – If True, add a version key-value pair to the serialized output dictionary. The default is False.

Returns A Dict object.

Example

```
>>> pprint(sim.json())
{'config': {'decompose_spectrum': 'none',
            'integration_density': 70,
            'integration_volume': 'octant',
            'number_of_sidebands': 64},
 'spin_systems': [{'abundance': '100 %',
                   'sites': [{'isotope': '13C',
                              'isotropic_chemical_shift': '20.0 ppm',
                              'shielding_symmetric': {'eta': 0.5,
                                                      'zeta': '10.0 ppm'}}]},
                  {'abundance': '100 %',
                   'sites': [{'isotope': '1H',
                              'isotropic_chemical_shift': '-4.0 ppm',
                              'shielding_symmetric': {'eta': 0.1,
                                                      'zeta': '2.1 ppm'}}]},
                  {'abundance': '100 %',
                   'sites': [{'isotope': '27Al',
                              'isotropic_chemical_shift': '120.0 ppm',
                              'shielding_symmetric': {'eta': 0.1,
                                                      'zeta': '2.1 ppm'}}]}]}
```

classmethod parse_dict_with_units (*py_dict*)

Parse the physical quantity from a dictionary representation of the Simulator object, where the physical quantity is expressed as a string with a number and a unit.

Parameters **py_dict** (*dict*) – A required python dict object.

Returns A *Simulator* (page 151) object.

Example

```
>>> sim_py_dict = {
...     'config': {
...         'decompose_spectrum': 'none',
...         'integration_density': 70,
...         'integration_volume': 'octant',
...         'number_of_sidebands': 64
...     },
...     'spin_systems': [
...         {
...             'abundance': '100 %',
```

(continues on next page)

(continued from previous page)

```
...                 'sites': [{
...                     'isotope': '13C',
...                     'isotropic_chemical_shift': '20.0 ppm',
...                     'shielding_symmetric': {'eta': 0.5, 'zeta': '10.0 ppm'}
...                 }]
...             },
...             {
...                 'abundance': '100 %',
...                 'sites': [{
...                     'isotope': '1H',
...                         'isotropic_chemical_shift': '-4.0 ppm',
...                         'shielding_symmetric': {'eta': 0.1, 'zeta': '2.1 ppm'}
...                 }]
...             },
...             {
...                 'abundance': '100 %',
...                 'sites': [{
...                     'isotope': '27Al',
...                     'isotropic_chemical_shift': '120.0 ppm',
...                     'shielding_symmetric': {'eta': 0.1, 'zeta': '2.1 ppm'}
...                 }]
...             }
...         ]
... }
>>> sim = Simulator.parse_dict_with_units(sim_py_dict)
>>> len(sim.spin_systems)
3
```

load_spin_systems (*filename: str*)

Load a list of spin systems from the given JSON serialized file.

See an example of a JSON serialized file. For details, refer to the *mrsimulator I/O* (page 32) section of this documentation.

Parameters **filename** (*str*) – A local or remote address to a JSON serialized file.

Example

```
>>> sim.load_spin_systems(filename)
```

export_spin_systems (*filename: str*)

Export a list of spin systems to a JSON serialized file.

See an example of a JSON serialized file. For details, refer to the *mrsimulator I/O* (page 32) section.

Parameters **filename** (*str*) – A filename of the serialized file.

Example

```
>>> sim.export_spin_systems(filename)
```

run (*method_index=None*, *pack_as_csdm=True*, ***kwargs*)
Run the simulation and compute spectrum.

Parameters

- **method_index** – An integer or a list of integers. If provided, only the simulations corresponding to the methods at the given index/indexes will be computed. The default is None, *i.e.*, the simulation for every method will be computed.
- **pack_as_csdm** (*bool*) – If true, the simulation results are stored as a CSDM object, otherwise, as a ndarray object. The simulations are stored as the value of the *simulation* (page 171) attribute of the corresponding method.

Example

```
>>> sim.run()
```

save (*filename: str*, *with_units=True*)
Serialize the simulator object to a JSON file.

Parameters

- **with_units** (*bool*) – If true, the attribute values are serialized as physical quantities expressed as a string with a value and a unit. If false, the attribute values are serialized as floats.
- **filename** (*str*) – The filename of the serialized file.

Example

```
>>> sim.save('filename')
```

classmethod load (*filename: str*, *parse_units=True*)
Load the *Simulator* (page 151) object from a JSON file by parsing.

Parameters

- **parse_units** (*bool*) – If true, parse the attribute values from the serialized file for physical quantities, expressed as a string with a value and a unit.
- **filename** (*str*) – The filename of a JSON serialized mrsimulator file.

Returns A *Simulator* (page 151) object.

Example

```
>>> sim_1 = sim.load('filename')
```

See also:

mrsimulator I/O (page 32)

6.1.2 ConfigSimulator

class mrsimulator.simulator.**ConfigSimulator**(*, *number_of_sidebands: mrsimulator.simulator.config.ConstrainedIntValue = 64*, *integration_volume: Literal[octant, hemisphere] = 'octant'*, *integration_density: mrsimulator.simulator.config.ConstrainedIntValue = 70*, *decompose_spectrum: Literal[none, spin_system] = 'none'*)

Bases: pydantic.main.BaseModel

The configurable attributes for the Simulator class used in simulation.

number_of_sidebands
The value is the requested number of sidebands that will be computed in the simulation. The value cannot be zero or negative. The default value is 64.

Type int (optional)

integration_volume
The value is the volume over which the solid-state spectral frequency integration is performed. The valid literals of this enumeration are

- `octant` (default), and
- `hemisphere`

Type enum (optional)

integration_density
The value represents the integration density or equivalently the number of orientations over which the frequency integration is performed within a given volume. If n is the integration_density, then the total number of orientation is given as

$$n_{\text{octants}} \frac{(n+1)(n+2)}{2}, \tag{6.1}$$

where n_{octants} is the number of octants in the given volume. The default value is 70.

Type int (optional)

decompose_spectrum
The value specifies how a simulation result is decomposed into an array of spectra. The valid literals of this enumeration are

- `none` (default): When the value is *none*, the resulting simulation is a single spectrum, which is an integration of the spectra over all spin systems.
- `spin_system`: When the value is *spin_system*, the resulting simulation is an array of spectra, where each spectrum arises from a spin system within the Simulator object.

Type enum (optional)

Example

```
>>> a = Simulator()
>>> a.config.number_of_sidebands = 128
>>> a.config.integration_density = 96
>>> a.config.integration_volume = 'hemisphere'
>>> a.config.decompose_spectrum = 'spin_system'
```

Method Documentation

get_orientations_count()
Return the total number of orientations.

Example

```
>>> a = Simulator()
>>> a.config.integration_density = 20
>>> a.config.integration_volume = 'hemisphere'
>>> a.config.get_orientations_count() # (4 * 21 * 22 / 2) = 924
924
```

6.1.3 SpinSystem

class mrsimulator.**SpinSystem**(*, *property_units: Dict = {'abundance': 'pct'}*, *name: str = None*, *label: str = None*, *description: str = None*, *sites: List[mrsimulator.spin_system.site.Site] = []*, *abundance: mrsimulator.spin_system.ConstrainedFloatValue = 100*, *transition_pathways: List = None*)
Bases: mrsimulator.utils.parseable.Parseable

Base class representing an isolated spin system containing multiple sites and couplings amongst them.

Attribute Documentation

sites
The value is a list of sites within the spin system, where each site represents a single-site nuclear spin interaction tensor parameters. The default value is an empty list.

Example

```
>>> sys1 = SpinSystem()
>>> sys1.sites = [Site(isotope='17O'), Site(isotope='1H')]
>>> # or equivalently
>>> sys1.sites = [{'isotope': '17O'}, {'isotope': '1H'}]
```

Type A list of *Site* (page 163) objects or equivalent dict objects (optional).

abundance
The abundance of the spin system in units of %. The default value is 100. The value of this attribute is useful when multiple spin systems are present.

Example

```
>>> sys1.abundance = 10
```

Type float (optional)

name
The value is the name or id of the spin system. The default value is None.

Example

```
>>> sys1.name = '1H-17O-0'
>>> sys1.name
'1H-17O-0'
```

Type str (optional)

label
The value is a label for the spin system. The default value is None.

Example

```
>>> sys1.label = 'Heteronuclear spin system'
>>> sys1.label
'Heteronuclear spin system'
```

Type str (optional)

description
The value is a description of the spin system. The default value is None.

Example

```
>>> sys1.description = 'A test for the spin system'
>>> sys1.description
'A test for the spin system'
```

Type str (optional)

transition_pathways
The value is a list of lists, where the inner list represents a transition pathway, and the outer list is the number of transition pathways. Each transition pathways is a list of Transition objects. The resulting spectrum is a sum of the resonances arising from individual transition pathways. The default value is None.

Example

```
>>> sys1.transition_pathways = [
...     [
...         {'initial': [-2.5, 0.5], 'final': [2.5, 0.5]},
...         {'initial': [0.5, 0.5], 'final': [-0.5, 0.5]}
...     ]
... ]
```

Note: From any given spin system, the list of relevant transition pathways is determined by the applied NMR method. For example, consider a single site I=3/2 spin system. For this system, a Bloch decay spectrum method will select three transition pathways, one corresponding to the central and two to the satellite transitions. On the other hand, a Bloch decay central transition selective method will only select one transition pathway, corresponding to the central transition.

Since the spin system is independent of the NMR method, the value of this attribute is, therefore, transient. You may use this attribute to override the default transition pathway query selection criterion of the NMR method objects.

Only use this attribute if you know what you are doing.

At times, this attribute may provide a significant improvement in the performance, especially in iterative algorithms, such as the least-squares algorithm, where a one-time transition pathway query is sufficient. Repeated queries for the transition pathways will add significant overhead to the computation.

See also:

Fitting example

Type list (optional)

Method Documentation

get_isotopes(*spin_I=None*) → list

An ordered list of isotopes from sites within the spin system corresponding to the given value of spin quantum number *I*. If *I* is None, a list of all isotopes is returned instead.

Parameters **spin_I** (*float*) – An optional spin quantum number. The valid inputs are the multiples of 0.5.

Returns A list of isotopes.

Example

```
>>> spin_systems.get_isotopes() # three spin systems
['13C', '1H', '27Al']
>>> spin_systems.get_isotopes(spin_I=0.5) # isotopes with I=0.5
['13C', '1H']
>>> spin_systems.get_isotopes(spin_I=1.5) # isotopes with I=1.5
[]
>>> spin_systems.get_isotopes(spin_I=2.5) # isotopes with I=2.5
['27Al']
```

zeeman_energy_states() → list

Return a list of all Zeeman energy states of the spin system, where the energy states are represented by a list of quantum numbers,

$$|\Psi\rangle = [m_1, m_2, ..m_n], \tag{6.2}$$

where m_i is the quantum number associated with the i^{th} site within the spin system, and Ψ is the energy state.

Example

```
>>> spin_system_1H_13C.get_isotopes() # two site (spin-1/2) spin systems
['13C', '1H']
>>> spin_system_1H_13C.zeeman_energy_states()  # four energy level system.
[|-0.5, -0.5⟩, |-0.5, 0.5⟩, |0.5, -0.5⟩, |0.5, 0.5⟩]
```

Returns A list of *ZeemanState* (page 170) objects.

all_transitions() → mrsimulator.transition.transition_list.TransitionList
Returns a list of all possible spin transitions in the given spin system.

Example

```
>>> spin_system_1H_13C.get_isotopes()  # two site (spin-1/2) spin system
['13C', '1H']
>>> spin_system_1H_13C.all_transitions()  # 16 two energy level transitions
[|-0.5, -0.5⟩⟨-0.5, -0.5|,
|-0.5, 0.5⟩⟨-0.5, -0.5|,
|0.5, -0.5⟩⟨-0.5, -0.5|,
|0.5, 0.5⟩⟨-0.5, -0.5|,
|-0.5, -0.5⟩⟨-0.5, 0.5|,
|-0.5, 0.5⟩⟨-0.5, 0.5|,
|0.5, -0.5⟩⟨-0.5, 0.5|,
|0.5, 0.5⟩⟨-0.5, 0.5|,
|-0.5, -0.5⟩⟨0.5, -0.5|,
|-0.5, 0.5⟩⟨0.5, -0.5|,
|0.5, -0.5⟩⟨0.5, -0.5|,
|0.5, 0.5⟩⟨0.5, -0.5|,
|-0.5, -0.5⟩⟨0.5, 0.5|,
|-0.5, 0.5⟩⟨0.5, 0.5|,
|0.5, -0.5⟩⟨0.5, 0.5|,
|0.5, 0.5⟩⟨0.5, 0.5|]
```

classmethod parse_dict_with_units(*py_dict: dict*) → dict
Parse the physical quantity from a dictionary representation of the SpinSystem object, where the physical quantity is expressed as a string with a number and a unit.

Parameters **py_dict** (*dict*) – A required python dict object.

Returns *SpinSystem* (page 158) object.

Example

```
>>> spin_system_dict = {
...     "sites": [{
...         "isotope":"13C",
...         "isotropic_chemical_shift": "20 ppm",
...         "shielding_symmetric": {
...             "zeta": "10 ppm",
...             "eta": 0.5
...         }
...     }]
... }
>>> spin_system_1 = SpinSystem.parse_dict_with_units(spin_system_dict)
```

to_freq_dict(*B0: float*) → dict

Serialize the SpinSystem object to a JSON compliant python dictionary object, where the attribute value is a numbers expressed in the attribute's default unit. The default unit for the attributes with respective dimensionalities are:

- frequency: *Hz*
- angle: *rad*

Parameters **B0** (*float*) – A required macroscopic magnetic flux density in units of T.

Returns A python dict

Example

```
>>> pprint(spin_system_1.to_freq_dict(B0=9.4))
{'abundance': 100,
 'description': None,
 'label': None,
 'name': None,
 'sites': [{'description': None,
            'isotope': '13C',
            'isotropic_chemical_shift': -2013.1791999999998,
            'label': None,
            'name': None,
            'quadrupolar': None,
            'shielding_antisymmetric': None,
            'shielding_symmetric': {'alpha': None,
                                    'beta': None,
                                    'eta': 0.5,
                                    'gamma': None,
                                    'zeta': -1006.5895999999999}}],
 'transition_pathways': None}
```

json() → dict

Parse the class object to a JSON compliant python dictionary object where the attribute value with physical quantity is expressed as a string with a number and a unit.

```
>>> pprint(spin_system_1.json())
{'abundance': '100 %',
 'sites': [{'isotope': '13C',
            'isotropic_chemical_shift': '20.0 ppm',
            'shielding_symmetric': {'eta': 0.5, 'zeta': '10.0 ppm'}}]}
```

6.1.4 Site

class mrsimulator.**Site**(*, *property_units: Dict = {'isotropic_chemical_shift': 'ppm'}*, *name: str = None*, *label: str = None*, *description: str = None*, *isotope: str = '1H'*, *isotropic_chemical_shift: float = 0*, *shielding_symmetric:* mrsimulator.spin_system.tensors.SymmetricTensor *(page 167) = None*, *shielding_antisymmetric:* mrsimulator.spin_system.tensors.AntisymmetricTensor *(page 169) = None*, *quadrupolar:* mrsimulator.spin_system.tensors.SymmetricTensor *(page 167) = None*)

Bases: mrsimulator.utils.parseable.Parseable

Base class representing a single-site nuclear spin interaction tensor parameters. The single-site nuclear spin interaction tensors include the nuclear shielding and the electric quadrupolar tensor.

Attribute Documentation

isotope

A string expressed as an atomic number followed by an isotope symbol, eg., *'13C'*, *'17O'*. The default value is *'1H'*.

Example

```
>>> site = Site(isotope='2H')
```

Type str (optional)

isotropic_chemical_shift

The value is the isotropic chemical shift of the site in the unit of ppm. The default value is 0.

Example

```
>>> site.isotropic_chemical_shift = 43.3
```

Type float (optional)

shielding_symmetric

The value of this attribute represents the irreducible second-rank traceless symmetric part of the nuclear shielding tensor. The default value is None.

The allowed attributes of the *SymmetricTensor* (page 167) class for *shielding_symmetric* are `zeta`, `eta`, `alpha`, `beta`, and `gamma`, where `zeta` is the shielding anisotropy, in ppm, and `eta` is the shielding asymmetry parameter defined using the Haeberlen convention. The Euler angles `alpha`, `beta`, and `gamma` are in radians.

Example

```
>>> site.shielding_symmetric = {'zeta': 10, 'eta': 0.5}
>>> # or equivalently
>>> site.shielding_symmetric = SymmetricTensor(zeta=10, eta=0.5)
```

Type *SymmetricTensor* (page 167) or equivalent dict object (optional).

shielding_antisymmetric

The value of this attribute represents the irreducible first-rank antisymmetric part of the nuclear shielding tensor. The default value is None.

The allowed attributes of the *AntisymmetricTensor* (page 169) class for *shielding_antisymmetric* are `zeta`, `alpha`, and `beta`, where `zeta` is the anisotropy parameter of the anti-symmetric first-rank tensor given in ppm. The angles `alpha` and `beta` are in radians.

Example

```
>>> site.shielding_antisymmetric = {'zeta': 20}
>>> # or equivalently
>>> site.shielding_antisymmetric = AntisymmetricTensor(zeta=20)
```

Type *AntisymmetricTensor* (page 169) or equivalent dict object (optional).

quadrupolar

The value of this attribute represents the irreducible second-rank traceless symmetric part of the electric-field gradient tensor. The default value is None.

The allowed attributes of the *SymmetricTensor* (page 167) class for *quadrupolar* are `Cq`, `eta`, `alpha`, `beta`, and `gamma`, where `Cq` is the quadrupolar coupling constant, in Hz, and `eta` is the quadrupolar asymmetry parameter. The Euler angles `alpha`, `beta`, and `gamma` are in radians.

Example

```
>>> site.quadrupolar = {'Cq': 3.2e6, 'eta': 0.52}
>>> # or equivalently
>>> site.quadrupolar = SymmetricTensor(Cq=3.2e6, eta=0.52)
```

Type *SymmetricTensor* (page 167) or equivalent dict object (optional).

name

The value is the name or id of the site. The default value is None.

Example

```
>>> site.name = '2H-0'
>>> site.name
'2H-0'
```

Type str (optional)

label

The value is a label for the site. The default value is None.

Example

```
>>> site.label = 'Quad site'
>>> site.label
'Quad site'
```

Type str (optional)

description
The value is a description of the site. The default value is None.

Example

```
>>> site.description = 'An example Quadrupolar site.'
>>> site.description
'An example Quadrupolar site.'
```

Type str (optional)

Example

The following are a few examples of setting the site object.

```
>>> site1 = Site(
...     isotope='33S',
...     isotropic_chemical_shift=20, # in ppm
...     shielding_symmetric={
...         "zeta": 10, # in ppm
...         "eta": 0.5
...     },
...     quadrupolar={
...         "Cq": 5.1e6, # in Hz
...         "eta": 0.5
...     }
... )
```

Using SymmetricTensor objects.

```
>>> site1 = Site(
...     isotope='13C',
...     isotropic_chemical_shift=20, # in ppm
...     shielding_symmetric=SymmetricTensor(zeta=10, eta=0.5),
... )
```

Method Documentation

classmethod parse_dict_with_units(*py_dict*)

Parse the physical quantity from a dictionary representation of the Site object, where the physical quantity is expressed as a string with a number and a unit.

Parameters **py_dict** (*dict*) – A required python dict object.

Returns *Site* (page 163) object.

Example

```
>>> site_dict = {
...     "isotope": "13C",
...     "isotropic_chemical_shift": "20 ppm",
...     "shielding_symmetric": {"zeta": "10 ppm", "eta":0.5}
... }
>>> site1 = Site.parse_dict_with_units(site_dict)
```

to_freq_dict(*B0*)

Serialize the Site object to a JSON compliant python dictionary object, where the attribute value is a number expressed in the attribute's default unit. The default unit for the attributes with respective dimensionalities is:

- frequency: `Hz`
- angle: `rad`

Parameters **B0** (*float*) – A required macroscopic magnetic flux density in units of T.

Returns Python dict object.

Example

```
>>> pprint(site1.to_freq_dict(B0=9.4))
{'description': None,
 'isotope': '13C',
 'isotropic_chemical_shift': -2013.1791999999998,
 'label': None,
 'name': None,
 'quadrupolar': None,
 'shielding_antisymmetric': None,
 'shielding_symmetric': {'alpha': None,
                         'beta': None,
                         'eta': 0.5,
                         'gamma': None,
                         'zeta': -1006.5895999999999}}
```

json() → dict

Parse the class object to a JSON compliant python dictionary object where the attribute value with physical quantity is expressed as a string with a number and a unit.

```
>>> pprint(site1.json())
{'isotope': '13C',
 'isotropic_chemical_shift': '20.0 ppm',
 'shielding_symmetric': {'eta': 0.5, 'zeta': '10.0 ppm'}}
```

6.1.5 Other Objects

SymmetricTensor

class mrsimulator.spin_system.tensors.**SymmetricTensor**(*, *property_units: Dict = {'Cq': 'Hz', 'alpha': 'rad', 'beta': 'rad', 'gamma': 'rad', 'zeta': 'ppm'}*, *zeta: float = None*, *Cq: float = None*, *eta: mrsimulator.spin_system.tensors.ConstrainedFloatValue = None*, *alpha: float = None*, *beta: float = None*, *gamma: float = None*)

Bases: mrsimulator.utils.parseable.Parseable

Base SymmetricTensor class representing the traceless symmetric part of an irreducible second-rank tensor.

zeta
The anisotropy parameter of the nuclear shielding tensor, in ppm, expressed using the Haeberlen convention. The default value is None.

Example

```
>>> shielding = SymmetricTensor()
>>> shielding.zeta = 10
```

Type float (optional)

Cq
The quadrupolar coupling constant, in Hz, derived from the electric field gradient tensor. The default value is None.

Example

```
>>> efg = SymmetricTensor()
>>> efg.Cq = 10e6
```

Type float (optional)

eta
The asymmetry parameter of the SymmetricTensor expressed using the Haeberlen convention. The default value is None.

Example

```
>>> shielding.eta = 0.1
>>> efg.eta = 0.5
```

Type float (optional)

alpha
Euler angle, α, in radians. The default value is None.

Example

```
>>> shielding.alpha = 0.15
>>> efg.alpha = 1.5
```

Type float (optional)

beta
Euler angle, β, in radians. The default value is None.

Example

```
>>> shielding.beta = 3.1415
>>> efg.beta = 1.1451
```

Type float (optional)

gamma
Euler angle, γ, in radians. The default value is None.

Example

```
>>> shielding.gamma = 2.1
>>> efg.gamma = 0
```

Type float (optional)

Example

```
>>> shielding = SymmetricTensor(zeta=10, eta=0.1, alpha=0.15, beta=3.14, gamma=2.1)
>>> efg = SymmetricTensor(Cq=10e6, eta=0.5, alpha=1.5, beta=1.1451, gamma=0)
```

Method Documentation

to_freq_dict(*larmor_frequency: float*) → dict
Serialize the SymmetricTensor object to a JSON compliant python dictionary where the attribute values are numbers expressed in default units. The default unit for attributes with respective dimensionalities are: - frequency: *Hz* - angle: *rad*

Parameters **larmor_frequency** (*float*) – The larmor frequency in MHz.

Returns A python dict

json() → dict
Parse the class object to a JSON compliant python dictionary object where the attribute value with physical quantity is expressed as a string with a number and a unit.

AntisymmetricTensor

class mrsimulator.spin_system.tensors.**AntisymmetricTensor**(*, *property_units: Dict = {'alpha': 'rad', 'beta': 'rad', 'zeta': 'ppm'}*, *zeta: float = None*, *alpha: float = None*, *beta: float = None*)

Bases: mrsimulator.utils.parseable.Parseable

Base SymmetricTensor class representing the traceless symmetric part of an irreducible second-rank tensor.

zeta
The anisotropy parameter of the AntiSymmetricTensor expressed using the Haeberlen convention. The default value is None.

alpha
Euler angle, alpha, given in radian. The default value is None.

beta
Euler angle, beta, given in radian. The default value is None.

Method Documentation

to_freq_dict(*larmor_frequency: float*) → dict
Serialize the AntisymmetricTensor object to a JSON compliant python dictionary where the attribute values are numbers expressed in default units. The default unit for attributes with respective dimensionalities are: - frequency: *Hz* - angle: *rad*

Parameters **larmor_frequency** (*float*) – The larmor frequency in MHz.

Returns Python dict

json() → dict
Parse the class object to a JSON compliant python dictionary object where the attribute value with physical quantity is expressed as a string with a number and a unit.

Isotope

class mrsimulator.spin_system.isotope.**Isotope**(*, *symbol: str*)

Bases: pydantic.main.BaseModel

The Isotope class.

symbol
The isotope symbol given as the atomic number followed by the atomic symbol.

Type str (required)

Example

```
>>> # 13C isotope information
>>> carbon = Isotope(symbol='13C')
>>> carbon.spin
0.5
>>> carbon.natural_abundance # in %
1.11
>>> carbon.gyromagnetic_ratio # in MHz/T
10.7084
>>> carbon.atomic_number
6
```

(continues on next page)

(continued from previous page)

```
>>> carbon.quadrupole_moment # in eB
0.0
```

Attribute Description

spin	Spin quantum number, I, of the isotope.
natural_abundance	Natural abundance of the isotope in units of %.
gyromagnetic_ratio	Reduced gyromagnetic ratio of the nucleus given in units of MHz/T.
atomic_number	Atomic number of the isotope.
quadrupole_moment	Quadrupole moment of the nucleus given in units of eB (electron-barn).

Method Documentation

json() → dict

Parse the class object to a JSON compliant python dictionary object where the attribute value with physical quantity is expressed as a string with a value and a unit.

ZeemanState

class mrsimulator.spin_system.zeeman_state.**ZeemanState**(*n_sites*, **args*)

Bases: object

Zeeman energy state class.

Method Documentation

tolist()

6.1.6 Method

class mrsimulator.**Method**(*, *property_units: Dict = {'magnetic_flux_density': 'T', 'rotor_angle': 'rad', 'rotor_frequency': 'Hz'}*, *name: str = None*, *label: str = None*, *description: str = None*, *channels: List[str] = []*, *spectral_dimensions: List[mrsimulator.method.spectral_dimension.SpectralDimension] = [SpectralDimension(property_units={'spectral_width': 'Hz', 'reference_offset': 'Hz', 'origin_offset': 'Hz'}, count=1024, spectral_width=25000.0, reference_offset=0.0, origin_offset=None, label=None, description=None, events=[])]*, *affine_matrix: Union[numpy.ndarray, List] = None*, *simulation: Union[csdmpy.csdm.CSDM, numpy.ndarray] = None*, *experiment: Union[csdmpy.csdm.CSDM, numpy.ndarray] = None*)

Bases: mrsimulator.utils.parseable.Parseable

Base Method class. A method class represents the NMR method.

channels

The value is a list of isotope symbols over which the given method applies. An isotope symbol is given as a string with the atomic number followed by its atomic symbol, for example, '1H', '13C', and '33S'. The default is an empty list. The number

of isotopes in a *channel* depends on the method. For example, a *BlochDecaySpectrum* method is a single channel method, in which case, the value of this attribute is a list with a single isotope symbol, ['13C'].

Example

```
>>> bloch = Method()
>>> bloch.channels = ['1H']
```

Type list (optional)

spectral_dimensions

The number of spectral dimensions depends on the given method. For example, a *BlochDecaySpectrum* method is a one-dimensional method and thus requires a single spectral dimension. The default is a single default *SpectralDimension* (page 173) object.

Example

```
>>> bloch = Method()
>>> bloch.spectral_dimensions = [SpectralDimension(count=8, spectral_width=50)]
>>> # or equivalently
>>> bloch.spectral_dimensions = [{'count': 8, 'spectral_width': 50}]
```

Type list of *SpectralDimension* (page 173) or dict objects (optional).

simulation

An object holding the result of the simulation. The initial value of this attribute is None. A value is assigned to this attribute when you run the simulation using the *run()* (page 156) method.

Type CSDM or ndarray (N/A)

experiment

An object holding the experimental measurement for the given method, if available. The default value is None.

Example

```
>>> bloch.experiment = my_data
```

Type CSDM or ndarray (optional)

name

The value is the name or id of the method. The default value is None.

Example

```
>>> bloch.name = 'BlochDecaySpectrum'
>>> bloch.name
'BlochDecaySpectrum'
```

Type str (optional)

label
The value is a label for the method. The default value is None.

Example

```
>>> bloch.label = 'One pulse acquired spectrum'
>>> bloch.label
'One pulse acquired spectrum'
```

Type str (optional)

description
The value is a description of the method. The default value is None.

Example

```
>>> bloch.description = 'Huh!'
>>> bloch.description
'Huh!'
```

Type str (optional)

Method Documentation

classmethod parse_dict_with_units(*py_dict*)
Parse the physical quantity from a dictionary representation of the Method object, where the physical quantity is expressed as a string with a number and a unit.

Parameters **py_dict** (*dict*) – A python dict representation of the Method object.

Returns A *Method* (page 170) object.

json()
Parse the class object to a JSON compliant python dictionary object where the attribute value with physical quantity is expressed as a string with a value and a unit.

Returns A python dict object.

update_spectral_dimension_attributes_from_experiment()
Update the spectral dimension attributes of the method to match the attributes of the experiment from the *experiment* (page 171) attribute.

get_transition_pathways(*spin_system*) → list
Return a list of transition pathways from the given spin system that satisfy the query selection criterion of the method.

Parameters **spin_system** (SpinSystem (page 158)) – A SpinSystem object.

Returns An array of TransitionPathway objects. Each TransitionPathway object is an ordered collection of Transition objects.

SpectralDimension

class mrsimulator.**SpectralDimension**(*, *property_units: Dict = {'origin_offset': 'Hz', 'reference_offset': 'Hz', 'spectral_width': 'Hz'}*, *count: mrsimulator.method.spectral_dimension.ConstrainedIntValue = 1024*, *spectral_width: mrsimulator.method.spectral_dimension.ConstrainedFloatValue = 25000.0*, *reference_offset: float = 0.0*, *origin_offset: float = None*, *label: str = None*, *description: str = None*, *events: List[mrsimulator.method.event.Event] = []*)

Bases: mrsimulator.utils.parseable.Parseable

Base SpectralDimension class defines a spectroscopic dimension of the method.

count
The number of points, N, along the spectroscopic dimension. The default value is 1024.

Type int (optional)

spectral_width
The spectral width, Δx, of the spectroscopic dimension in units of Hz. The default value is 25000.

Type float (optional)

reference_offset
The reference offset, x_0, of the spectroscopic dimension in units of Hz. The default value is 0.

Type float (optional)

origin_offset
The origin offset (Larmor frequency) along the spectroscopic dimension in units of Hz. The default value is None. When the value is None, the origin offset is set to the Larmor frequency of the isotope from the *channels* (page 170) attribute of the method.

Type float (optional)

label
The value is a label of the spectroscopic dimension. The default value is None.

Type str (optional)

description
The value is a description of the spectroscopic dimension. The default value is None.

Type str (optional)

events
The value describes a series of events along the spectroscopic dimension.

Type A list of *Event* (page 174) or equivalent dict objects (optional).

Method Documentation

classmethod parse_dict_with_units(*py_dict: dict*)
Parse the physical quantities of a SpectralDimension object from a python dictionary object.

Parameters **py_dict** (*dict*) – Dict object

coordinates_Hz() → numpy.ndarray
The grid coordinates along the dimension in units of Hz, evaluated as

$$x_{\text{Hz}} = ([0, 1, ...N-1] - T)\,\frac{\Delta x}{N} + x_0 \tag{6.3}$$

where $T = N/2$ and $T = (N-1)/2$ for even and odd values of N, respectively.

coordinates_ppm() → numpy.ndarray
The grid coordinates along the dimension as dimension frequency ratio in units of ppm. The coordinates are evaluated as

$$x_{\text{ppm}} = \frac{x_{\text{Hz}}}{x_0 + \omega_0} \tag{6.4}$$

where ω_0 is the Larmor frequency.

json() → dict
Parse the class object to a JSON compliant python dictionary object where the attribute value with physical quantity is expressed as a string with a number and a unit.

to_csdm_dimension() → csdmpy.dimensions.Dimension
Return the spectral dimension as a CSDM dimension object.

Event

class mrsimulator.**Event**(**, property_units: Dict = {'magnetic_flux_density': 'T', 'rotor_angle': 'rad', 'rotor_frequency': 'Hz'}, fraction: float = 1.0, magnetic_flux_density: mrsimulator.method.event.ConstrainedFloatValue = 9.4, rotor_frequency: mrsimulator.method.event.ConstrainedFloatValue = 0.0, rotor_angle: mrsimulator.method.event.ConstrainedFloatValue = 0.955316618, freq_contrib: List[mrsimulator.method.frequency_contrib.FrequencyEnum] = [<FrequencyEnum.Shielding1_0: 'Shielding1_0'>, <FrequencyEnum.Shielding1_2: 'Shielding1_2'>, <FrequencyEnum.Quad1_2: 'Quad1_2'>, <FrequencyEnum.Quad2_0: 'Quad2_0'>, <FrequencyEnum.Quad2_2: 'Quad2_2'>, <FrequencyEnum.Quad2_4: 'Quad2_4'>], transition_query: mrsimulator.method.transition_query.TransitionQuery = TransitionQuery(P={'channel-1': [[-1.0]]}, D=None, f=None, transitions=None)*)
Bases: mrsimulator.utils.parseable.Parseable

Base Event class defines the spin environment and the transition query for a segment of the transition pathway.

fraction
A *required* float containing the weight of the frequency contribution from the event.

magnetic_flux_density
An *optional* float containing the macroscopic magnetic flux density, H_0, of the applied external magnetic field during the event in units of T. The default value is `9.4`.

rotor_frequency
An *optional* float containing the sample spinning frequency ν_r, during the event in units of Hz. The default value is `0`.

rotor_angle
An *optional* float containing the angle between the sample rotation axis and the applied external magnetic field, θ, during the event in units of rad. The default value is `0.9553166`, i.e. the magic angle.

transition_query
An *optional* TransitionQuery object or an equivalent dict object listing the queries used in selecting the active transitions during the event. Only the active transitions from this query contribute to the frequency.

Method Documentation

classmethod parse_dict_with_units(*py_dict: dict*)
Parse the physical quantities of an Event object from a python dictionary object.

Parameters **py_dict** (*dict*) – Dict object

json()
Parse the class object to a JSON compliant python dictionary object where the attribute value with physical quantity is expressed as a string with a number and a unit.

6.1.7 Methods

The following are the list of methods currently supported by `mrsimulator` as a part of the `mrsimulator.methods` module. To import a method, for example the *BlochDecaySpectrum*, used

```
>>> from mrsimulator.methods import BlochDecaySpectrum
```

All methods categorize into two groups, generic and specialized methods. A generic method is general and is based on the number of spectral dimensions. At present, there are two generic methods, `Method1D` and `Method2D`. All specialized methods are derived from their respective generic method objects. The purpose of the specialized methods is to facilitate user ease when setting up some commonly used methods, such as the MQMAS, STMAS, PASS, MAT, etc.

Summary

Generic methods

Method1D (page 176)([spectral_dimensions])	A generic one-dimensional spectrum method.
Method2D (page 180)([spectral_dimensions])	A generic two-dimensional correlation spectrum method.

Specialized methods

BlochDecaySpectrum (page 177)([spectral_dimensions])	A one-dimensional Bloch decay spectrum method.
BlochDecayCentralTransitionSpectrum (page 178)([…])	A one-dimensional central transition selective Bloch decay spectrum method.
ThreeQ_VAS (page 182)(**kwargs)	Simulate a sheared and scaled 3Q 2D variable-angle spinning spectrum.
FiveQ_VAS (page 183)(**kwargs)	Simulate a sheared and scaled 5Q variable-angle spinning spectrum.
SevenQ_VAS (page 184)(**kwargs)	Simulate a sheared and scaled 7Q variable-angle spinning spectrum.
ST1_VAS (page 186)(**kwargs)	Simulate a sheared and scaled inner satellite and central transition correlation spectrum.
ST2_VAS (page 187)(**kwargs)	Simulate a sheared and scaled second to inner satellite and central transition correlation spectrum.
SSB2D (page 188)(**kwargs)	A specialized method for simulating 2D finite speed to infinite speed MAS correlation spectum.

Table of contents

Generic one-dimensional method

mrsimulator.methods.**Method1D**(*spectral_dimensions=[{}]*, ***kwargs*)

A generic one-dimensional spectrum method.

Parameters

- **name** (*str (optional)*) – The value is the name or id of the method. The default value is None.
- **label** (*str (optional)*) – The value is a label for the method. The default value is None.
- **description** (*str (optional)*) – The value is a description of the method. The default value is None.
- **experiment** (*CSDM or ndarray (optional)*) – An object holding the experimental measurement for the given method, if available. The default value is None.
- **channels** (*list (optional)*) – The value is a list of isotope symbols over which the given method applies. An isotope symbol is given as a string with the atomic number followed by its atomic symbol, for example, '1H', '13C', and '33S'. The default is an empty list. The number of isotopes in a *channel* depends on the method. For example, a *BlochDecaySpectrum* method is a single channel method, in which case, the value of this attribute is a list with a single isotope symbol, ['13C'].
- **spectral_dimensions** (List of *SpectralDimension* (page 173) or dict objects (optional).) – The number of spectral dimensions depends on the given method. For example, a *BlochDecaySpectrum* method is a one-dimensional method and thus requires a single spectral dimension. The default is a single default *SpectralDimension* (page 173) object.
- **magetic_flux_density** (*float (optional)*) – A global value for the macroscopic magnetic flux density, H_0, of the applied external magnetic field in units of T. The default is `9.4`.
- **rotor_angle** (*float (optional)*) – A global value for the angle between the sample rotation axis and the applied external magnetic field, θ, in units of rad. The default value is `0.9553166`, i.e. the magic angle.
- **rotor_frequency** (*float (optional)*) – A global value for the sample spinning frequency, ν_r, in units of Hz. The default value is `0`.

Returns

Return type A *Method* (page 170) instance.

Note: If any parameter is defined outside of the *spectral_dimensions* list, the value of those parameters is considered global. In a multi-event method, you may also assign parameter values to individual events.

Example

Example method for simulating triple-quantum 1D spectrum.

```
>>> from mrsimulator.methods import Method1D
>>> method1 =  Method1D(
...     channels=["87Rb"],
...     magnetic_flux_density=7,  # in T
...     rotor_angle=54.735*np.pi/180,
...     rotor_frequency=1e9,
...     spectral_dimensions=[
...         {
```

(continues on next page)

(continued from previous page)

```
...             "count": 1024,
...             "spectral_width": 1e4,  # in Hz
...             "reference_offset": -4e3,  # in Hz
...             "label": "quad only",
...             "events":[
...                 {"transition_query": {"P": [-3], "D": [0]}},
...             ]
...         }
...     ]
... )
```

Bloch Decay Spectrum method

mrsimulator.methods.**BlochDecaySpectrum**(*spectral_dimensions=[{}]*, ***kwargs*)

A one-dimensional Bloch decay spectrum method.

Parameters

- **name** (*str (optional)*) – The value is the name or id of the method. The default value is None.
- **label** (*str (optional)*) – The value is a label for the method. The default value is None.
- **description** (*str (optional)*) – The value is a description of the method. The default value is None.
- **experiment** (*CSDM or ndarray (optional)*) – An object holding the experimental measurement for the given method, if available. The default value is None.
- **channels** (*list (optional)*) – The value is a list of isotope symbols over which the given method applies. An isotope symbol is given as a string with the atomic number followed by its atomic symbol, for example, '1H', '13C', and '33S'. The default is an empty list. The number of isotopes in a *channel* depends on the method. For example, a *BlochDecaySpectrum* method is a single channel method, in which case, the value of this attribute is a list with a single isotope symbol, ['13C'].
- **spectral_dimensions** (List of *SpectralDimension* (page 173) or dict objects (optional).) – The number of spectral dimensions depends on the given method. For example, a *BlochDecaySpectrum* method is a one-dimensional method and thus requires a single spectral dimension. The default is a single default *SpectralDimension* (page 173) object.
- **magetic_flux_density** (*float (optional)*) – A global value for the macroscopic magnetic flux density, H_0, of the applied external magnetic field in units of T. The default is `9.4`.
- **rotor_angle** (*float (optional)*) – A global value for the angle between the sample rotation axis and the applied external magnetic field, θ, in units of rad. The default value is `0.9553166`, i.e. the magic angle.
- **rotor_frequency** (*float (optional)*) – A global value for the sample spinning frequency, ν_r, in units of Hz. The default value is `0`.

Returns

Return type A *Method* (page 170) instance.

Example

```
>>> from mrsimulator.methods import BlochDecaySpectrum
>>> Bloch_method = BlochDecaySpectrum(
...     channels=['1H'],
```

(continues on next page)

(continued from previous page)

```
...     rotor_frequency=5000, # in Hz
...     rotor_angle=0.95531, # in rad
...     magnetic_flux_density=9.4, # in T
...     spectral_dimensions=[dict(
...         count=1024,
...         spectral_width=50000, # in Hz
...         reference_offset=-8000, # in Hz
...     )]
... )
```

Bloch decay method is a special case of *Method1D* (page 176), given as

```
>>> from mrsimulator.methods import Method1D
>>> Blochdecay = Method1D(
...     channels=['1H'],
...     rotor_frequency=5000, # in Hz
...     rotor_angle=0.95531, # in rad
...     magnetic_flux_density=9.4, # in T
...     spectral_dimensions=[
...         {
...             "count": 1024,
...             "spectral_width": 50000, # in Hz
...             "reference_offset": -8000, # in Hz
...             "events": [{
...                 "transition_query": {"P": [-1]}
...             }]
...         }
...     ]
... )
```

Bloch Decay Central Transition Spectrum method

mrsimulator.methods.**BlochDecayCentralTransitionSpectrum**(*spectral_dimensions=[{}]*, ***kwargs*)

A one-dimensional central transition selective Bloch decay spectrum method.

Parameters

- **name** (*str (optional)*) – The value is the name or id of the method. The default value is None.
- **label** (*str (optional)*) – The value is a label for the method. The default value is None.
- **description** (*str (optional)*) – The value is a description of the method. The default value is None.
- **experiment** (*CSDM or ndarray (optional)*) – An object holding the experimental measurement for the given method, if available. The default value is None.
- **channels** (*list (optional)*) – The value is a list of isotope symbols over which the given method applies. An isotope symbol is given as a string with the atomic number followed by its atomic symbol, for example, '1H', '13C', and '33S'. The default is an empty list. The number of isotopes in a *channel* depends on the method. For example, a *BlochDecaySpectrum* method is a single channel method, in which case, the value of this attribute is a list with a single isotope symbol, ['13C'].
- **spectral_dimensions** (List of *SpectralDimension* (page 173) or dict objects (optional).) – The number of spectral dimensions depends on the given method. For example, a *BlochDecaySpectrum* method is a one-dimensional method and thus requires a single spectral dimension. The default is a single default *SpectralDimension* (page 173) object.

- **magetic_flux_density** (*float (optional)*) – A global value for the macroscopic magnetic flux density, H_0, of the applied external magnetic field in units of T. The default is `9.4`.
- **rotor_angle** (*float (optional)*) – A global value for the angle between the sample rotation axis and the applied external magnetic field, θ, in units of rad. The default value is `0.9553166`, i.e. the magic angle.
- **rotor_frequency** (*float (optional)*) – A global value for the sample spinning frequency, ν_r, in units of Hz. The default value is `0`.

Returns

Return type A *Method* (page 170) instance.

Example

```
>>> from mrsimulator.methods import BlochDecayCentralTransitionSpectrum
>>> Bloch_CT_method = BlochDecayCentralTransitionSpectrum(
...     channels=['1H'],
...     rotor_frequency=5000, # in Hz
...     rotor_angle=0.95531, # in rad
...     magnetic_flux_density=9.4, # in T
...     spectral_dimensions=[dict(
...         count=1024,
...         spectral_width=50000, # in Hz
...         reference_offset=-8000, # in Hz
...     )]
... )
```

Bloch decay central transition selective method is a special case of *Method1D* (page 176), given as

```
>>> from mrsimulator.methods import Method1D
>>> BlochdecayCT = BlochDecayCentralTransitionSpectrum(
...     channels=['1H'],
...     rotor_frequency=5000, # in Hz
...     rotor_angle=0.95531, # in rad
...     magnetic_flux_density=9.4, # in T
...     spectral_dimensions=[
...         {
...             "count": 1024,
...             "spectral_width": 50000, # in Hz
...             "reference_offset": -8000, # in Hz
...             "events": [{
...                 "transition_query": {"P": [-1], "D": [0]}
...             }]
...         }
...     ]
... )
```

Generic two-dimensional correlation method

mrsimulator.methods.**Method2D**(*spectral_dimensions=[{}]*, ***kwargs*)

A generic two-dimensional correlation spectrum method.

Parameters

- **name** (*str (optional)*) – The value is the name or id of the method. The default value is None.
- **label** (*str (optional)*) – The value is a label for the method. The default value is None.
- **description** (*str (optional)*) – The value is a description of the method. The default value is None.
- **experiment** (*CSDM or ndarray (optional)*) – An object holding the experimental measurement for the given method, if available. The default value is None.
- **channels** (*list (optional)*) – The value is a list of isotope symbols over which the given method applies. An isotope symbol is given as a string with the atomic number followed by its atomic symbol, for example, '1H', '13C', and '33S'. The default is an empty list. The number of isotopes in a *channel* depends on the method. For example, a *BlochDecaySpectrum* method is a single channel method, in which case, the value of this attribute is a list with a single isotope symbol, ['13C'].
- **spectral_dimensions** (List of *SpectralDimension* (page 173) or dict objects (optional).) – The number of spectral dimensions depends on the given method. For example, a *BlochDecaySpectrum* method is a one-dimensional method and thus requires a single spectral dimension. The default is a single default *SpectralDimension* (page 173) object.
- **magetic_flux_density** (*float (optional)*) – A global value for the macroscopic magnetic flux density, H_0, of the applied external magnetic field in units of T. The default is `9.4`.
- **rotor_angle** (*float (optional)*) – A global value for the angle between the sample rotation axis and the applied external magnetic field, θ, in units of rad. The default value is `0.9553166`, i.e. the magic angle.
- **affine_matrix** (*np.ndarray or list (optional)*) – An affine transformation square matrix, $\mathbf{A} \in \mathbb{R}^{n \times n}$, where n is the number of spectral dimensions. The affine operation follows $\mathbf{V}' = \mathbf{A} \cdot \mathbf{V}$, where $\mathbf{V} \in \mathbb{R}^n$ and $\mathbf{V}' \in \mathbb{R}^n$ are the initial and transformed frequency coordinates.

Returns

Return type A *Method* (page 170) instance.

Note: If any parameter is defined outside of the *spectral_dimensions* list, the value of those parameters is considered global. In a multi-event method, you may also assign parameter values to individual events.

Example

```
>>> from mrsimulator.methods import Method2D
>>> method = Method2D(
...     channels=["87Rb"],
...     magnetic_flux_density=7,  # in T. Global value for `magnetic_flux_density`.
...     rotor_angle=0.95531, # in rads. Global value for the `rotor_angle`.
...     spectral_dimensions=[
...         {
...             "count": 256,
...             "spectral_width": 4e3,  # in Hz
...             "reference_offset": -5e3,  # in Hz
...             "event": [
```

(continues on next page)

(continued from previous page)

```
...                     { # Global value for the `magnetic_flux_density` and `rotor_angle`
...                       # is used during this event.
...                         "transition_query": {"P": [-3], "D": [0]}
...                     }
...                 ]
...             },
...             {
...                 "count": 512,
...                 "spectral_width": 1e4,  # in Hz
...                 "reference_offset": -4e3,  # in Hz
...                 "event": [
...                     { # Global value for the `magnetic_flux_density` is used during this
...                       # event. User defined local value for `rotor_angle` is used here.
...                         "rotor_angle": 1.2238, # in rads
...                         "transition_query": {"P": [-1], "D": [0]}
...                     }
...                 ]
...             },
...         ],
...         affine_matrix=[[1, -1], [0, 1]],
... )
```

Multi-quantum variable-angle spinning

The following classes are used in simulating multi-quantum variable-angle spinning spectrum correlating the frequencies from the symmetric multiple-quantum transition to the central transition frequencies. The p and d pathways for the MQVAS methods are

$$\begin{aligned} p &: 0 \rightarrow M \rightarrow -1 \\ d &: 0 \rightarrow 0 \end{aligned} \quad , \tag{6.5}$$

where M is the multiple-quantum number. The value of M depends on the spin quantum number, I, and is listed in Table 6.4.

Affine mapping

The resulting spectrum is sheared and scaled, such that the frequencies along the indirect dimension are given as

$$\langle \Omega \rangle_{\text{MQ-VAS}} = \frac{1}{1+\kappa}\Omega_{m,-m} + \frac{\kappa}{1+\kappa}\Omega_{1/2,-1/2}. \tag{6.5}$$

Here, $\langle \Omega \rangle_{\text{MQ-VAS}}$ is the average frequency along the indirect dimension, $\Omega_{m,-m}$ and $\Omega_{1/2,-1/2}$ are the frequency contributions from the $|m\rangle \rightarrow |-m\rangle$ symmetric multiple-quantum transition and the central transition, respectively, and κ is the shear factor. The values of the shear factor for various transitions are listed in Table 6.4.

Table 6.4: The table lists the multi-quantum transition associated with the spin I, and the corresponding shear factor, κ, used in affine mapping of the MQ-VAS methods.

Spin	Symmetric multi-quantum transition	M	κ
3/2	$\left(\frac{3}{2} \rightarrow -\frac{3}{2}\right)$	-3	21/27
5/2	$\left(-\frac{3}{2} \rightarrow \frac{3}{2}\right)$	3	114/72
5/2	$\left(\frac{5}{2} \rightarrow -\frac{5}{2}\right)$	-5	150/72
7/2	$\left(-\frac{3}{2} \rightarrow \frac{3}{2}\right)$	3	303/135
7/2	$\left(-\frac{5}{2} \rightarrow \frac{5}{2}\right)$	5	165/135
7/2	$\left(\frac{7}{2} \rightarrow -\frac{7}{2}\right)$	-7	483/135
9/2	$\left(-\frac{3}{2} \rightarrow \frac{3}{2}\right)$	3	546/216
9/2	$\left(-\frac{5}{2} \rightarrow \frac{5}{2}\right)$	5	570/216
9/2	$\left(-\frac{7}{2} \rightarrow \frac{7}{2}\right)$	5	84/216

Triple-quantum variable-angle spinning method

class mrsimulator.methods.**ThreeQ_VAS**(***kwargs*)

Simulate a sheared and scaled 3Q 2D variable-angle spinning spectrum.

Parameters

- **channels** – A list of isotope symbols over which the method will be applied.
- **spectral_dimensions** – A list of python dict. Each dict is contains keywords that describe the coordinates along a spectral dimension. The keywords along with its definition are:
  - count: An optional integer with the number of points, N, along the dimension. The default value is 1024.
  - spectral_width: An *optional* float with the spectral width, Δx, along the dimension in units of Hz. The default is 25 kHz.
  - reference_offset: An *optional* float with the reference offset, x_0 along the dimension in units of Hz. The default value is 0 Hz.
  - origin_offset: An *optional* float with the origin offset (Larmor frequency) along the dimension in units of Hz. The default value is None.
- **magetic_flux_density** – An *optional* float containing the macroscopic magnetic flux density, H_0, of the applied external magnetic field in units of T. The default value is 9.4.
- **rotor_angle** – An *optional* float containing the angle between the sample rotation axis and the applied external magnetic field, θ, in units of rad. The default value is 0.9553166, i.e. the magic angle.

Note: The attribute *rotor_frequency* cannot be modified for this method and is set to simulate an infinite speed spectrum.

Returns A *Method* (page 170) instance.

Example

```
>>> method = ThreeQ_VAS(
...     channels=["87Rb"],
...     magnetic_flux_density=7,  # in T
...     spectral_dimensions=[
...         {
...             "count": 256,
...             "spectral_width": 4e3,  # in Hz
...             "reference_offset": -5e3,  # in Hz
...             "label": "Isotropic dimension",
...         },
...         {
...             "count": 512,
...             "spectral_width": 1e4,  # in Hz
...             "reference_offset": -4e3,  # in Hz
...             "label": "MAS dimension",
...         },
...     ],
... )
>>> sys = SpinSystem(sites=[Site(isotope='87Rb')])
>>> method.get_transition_pathways(sys)
[TransitionPathway(|-1.5⟩⟨1.5|, |-0.5⟩⟨0.5|)]
```

Five-quantum variable-angle spinning method

class mrsimulator.methods.**FiveQ_VAS**(***kwargs*)

Simulate a sheared and scaled 5Q variable-angle spinning spectrum.

Parameters

- **channels** – A list of isotope symbols over which the method will be applied.
- **spectral_dimensions** – A list of python dict. Each dict is contains keywords that describe the coordinates along a spectral dimension. The keywords along with its definition are:
  - count: An optional integer with the number of points, N, along the dimension. The default value is 1024.
  - spectral_width: An *optional* float with the spectral width, Δx, along the dimension in units of Hz. The default is 25 kHz.
  - reference_offset: An *optional* float with the reference offset, x_0 along the dimension in units of Hz. The default value is 0 Hz.
  - origin_offset: An *optional* float with the origin offset (Larmor frequency) along the dimension in units of Hz. The default value is None.
- **magetic_flux_density** – An *optional* float containing the macroscopic magnetic flux density, H_0, of the applied external magnetic field in units of T. The default value is 9.4.
- **rotor_angle** – An *optional* float containing the angle between the sample rotation axis and the applied external magnetic field, θ, in units of rad. The default value is 0.9553166, i.e. the magic angle.

Note: The attribute *rotor_frequency* cannot be modified for this method and is set to simulate an infinite speed spectrum.

Returns A *Method* (page 170) instance.

Example

```
>>> method = FiveQ_VAS(
...     channels=["17O"],
...     magnetic_flux_density=9.4,  # in T
...     spectral_dimensions=[
...         {
...             "count": 256,
...             "spectral_width": 4e3,  # in Hz
...             "reference_offset": -5e3,  # in Hz
...             "label": "Isotropic dimension",
...         },
...         {
...             "count": 512,
...             "spectral_width": 1e4,  # in Hz
...             "reference_offset": -4e3,  # in Hz
...             "label": "MAS dimension",
...         },
...     ],
... )
>>> sys = SpinSystem(sites=[Site(isotope='17O')])
>>> method.get_transition_pathways(sys)
[TransitionPathway(|-2.5⟩⟨2.5|, |-0.5⟩⟨0.5|)]
```

Seven-quantum variable-angle spinning method

class mrsimulator.methods.**SevenQ_VAS**(***kwargs*)

Simulate a sheared and scaled 7Q variable-angle spinning spectrum.

Parameters

- **channels** – A list of isotope symbols over which the method will be applied.
- **spectral_dimensions** – A list of python dict. Each dict is contains keywords that describe the coordinates along a spectral dimension. The keywords along with its definition are:
  - count: An optional integer with the number of points, N, along the dimension. The default value is 1024.
  - spectral_width: An *optional* float with the spectral width, Δx, along the dimension in units of Hz. The default is 25 kHz.
  - reference_offset: An *optional* float with the reference offset, x_0 along the dimension in units of Hz. The default value is 0 Hz.
  - origin_offset: An *optional* float with the origin offset (Larmor frequency) along the dimension in units of Hz. The default value is None.
- **magetic_flux_density** – An *optional* float containing the macroscopic magnetic flux density, H_0, of the applied external magnetic field in units of T. The default value is 9.4.
- **rotor_angle** – An *optional* float containing the angle between the sample rotation axis and the applied external magnetic field, θ, in units of rad. The default value is 0.9553166, i.e. the magic angle.

Note: The attribute *rotor_frequency* cannot be modified for this method and is set to simulate an infinite speed spectrum.

Returns A *Method* (page 170) instance.

Example

```
>>> method = SevenQ_VAS(
...     channels=["51V"],
...     magnetic_flux_density=9.4,  # in T
...     spectral_dimensions=[
...         {
...             "count": 256,
...             "spectral_width": 4e3,  # in Hz
...             "reference_offset": -5e3,  # in Hz
...             "label": "Isotropic dimension",
...         },
...         {
...             "count": 512,
...             "spectral_width": 1e4,  # in Hz
...             "reference_offset": -4e3,  # in Hz
...             "label": "MAS dimension",
...         },
...     ],
... )
>>> sys = SpinSystem(sites=[Site(isotope='51V')])
>>> method.get_transition_pathways(sys)
[TransitionPathway(|-3.5⟩⟨3.5|, |-0.5⟩⟨0.5|)]
```

Satellite-transition variable-angle spinning (ST-VAS)

The following classes are used in simulating satellite-transition variable-angle spinning spectrum correlating the frequencies from the satellite transitions to the central transition frequencies. The p and d pathways for the ST-VAS methods are

$$\begin{aligned} p &: 0 \rightarrow -1 \rightarrow -1 \\ d &: 0 \rightarrow \pm d_0 \end{aligned} \tag{6.6}$$

where $d_0 = m_f^2 - m_i^2$ for transition $|m_i\rangle \rightarrow |m_f\rangle$. The value of n depends on the spin quantum number, I, and is listed in Table 6.5.

Affine mapping

The resulting spectrum is sheared and scaled, such that the frequencies along indirect dimension are given as

$$\langle\Omega\rangle_{\text{ST-VAS}} = \frac{1}{1+\kappa}\Omega_{m,m-1} + \frac{\kappa}{1+\kappa}\Omega_{1/2,-1/2}. \tag{6.6}$$

Here, $\langle\Omega\rangle_{\text{ST-VAS}}$ is the average frequency along the indirect dimension, $\Omega_{m,m-1}$ and $\Omega_{1/2,-1/2}$ are the frequency contributions from the $|m\rangle \rightarrow |m-1\rangle$ satellite transition and the central transition, respectively, and κ is the shear factor. The values of the shear factor for various satellite transitions are listed in Table 6.5.

Table 6.5: The table lists the satellite transitions associated with the spin I, and the corresponding shear factor, κ, used in affine mapping of the ST-VAS methods.

Spin	Satellite transitions	d_0	κ
3/2	$\left(\frac{3}{2} \rightarrow \frac{1}{2}\right), \left(-\frac{1}{2} \rightarrow -\frac{3}{2}\right)$	2	24/27
5/2	$\left(-\frac{3}{2} \rightarrow -\frac{1}{2}\right), \left(\frac{1}{2} \rightarrow \frac{3}{2}\right)$	2	21/72
5/2	$\left(\frac{5}{2} \rightarrow \frac{3}{2}\right), \left(-\frac{3}{2} \rightarrow -\frac{5}{2}\right)$	4	132/72
7/2	$\left(-\frac{3}{2} \rightarrow -\frac{1}{2}\right), \left(\frac{1}{2} \rightarrow \frac{3}{2}\right)$	2	84/135
7/2	$\left(-\frac{5}{2} \rightarrow -\frac{3}{2}\right), \left(\frac{3}{2} \rightarrow \frac{5}{2}\right)$	4	69/135
9/2	$\left(-\frac{3}{2} \rightarrow -\frac{1}{2}\right), \left(\frac{1}{2} \rightarrow \frac{3}{2}\right)$	2	165/216
9/2	$\left(-\frac{5}{2} \rightarrow -\frac{3}{2}\right), \left(\frac{3}{2} \rightarrow \frac{5}{2}\right)$	4	12/216

Inner satellite variable-angle spinning method

class mrsimulator.methods.**ST1_VAS**(***kwargs*)

Simulate a sheared and scaled inner satellite and central transition correlation spectrum.

Parameters

- **channels** – A list of isotope symbols over which the method will be applied.
- **spectral_dimensions** – A list of python dict. Each dict is contains keywords that describe the coordinates along a spectral dimension. The keywords along with its definition are:
  - count: An optional integer with the number of points, N, along the dimension. The default value is 1024.
  - spectral_width: An *optional* float with the spectral width, Δx, along the dimension in units of Hz. The default is 25 kHz.
  - reference_offset: An *optional* float with the reference offset, x_0 along the dimension in units of Hz. The default value is 0 Hz.
  - origin_offset: An *optional* float with the origin offset (Larmor frequency) along the dimension in units of Hz. The default value is None.
- **magetic_flux_density** – An *optional* float containing the macroscopic magnetic flux density, H_0, of the applied external magnetic field in units of T. The default value is 9.4.
- **rotor_angle** – An *optional* float containing the angle between the sample rotation axis and the applied external magnetic field, θ, in units of rad. The default value is 0.9553166, i.e. the magic angle.

Note: The attribute *rotor_frequency* cannot be modified for this method and is set to simulate an infinite speed spectrum.

Returns A *Method* (page 170) instance.

Example

```
>>> method = ST1_VAS(
...     channels=["87Rb"],
...     magnetic_flux_density=9.4,  # in T
...     spectral_dimensions=[
...         {
...             "count": 256,
...             "spectral_width": 4e3,  # in Hz
...             "reference_offset": -5e3,  # in Hz
...             "label": "Isotropic dimension",
...         },
...         {
...             "count": 512,
...             "spectral_width": 1e4,  # in Hz
...             "reference_offset": -4e3,  # in Hz
...             "label": "MAS dimension",
...         },
...     ],
... )
>>> sys = SpinSystem(sites=[Site(isotope='87Rb')])
>>> pprint(method.get_transition_pathways(sys))
[TransitionPathway(|-1.5>&lt;-0.5|, |-0.5>&lt;0.5|),
 TransitionPathway(|0.5>&lt;1.5|, |-0.5>&lt;0.5|)]
```

Second to inner satellite variable-angle spinning method

class mrsimulator.methods.**ST2_VAS**(***kwargs*)

Simulate a sheared and scaled second to inner satellite and central transition correlation spectrum.

Parameters

- **channels** – A list of isotope symbols over which the method will be applied.
- **spectral_dimensions** – A list of python dict. Each dict is contains keywords that describe the coordinates along a spectral dimension. The keywords along with its definition are:
  - count: An optional integer with the number of points, N, along the dimension. The default value is 1024.
  - spectral_width: An *optional* float with the spectral width, Δx, along the dimension in units of Hz. The default is 25 kHz.
  - reference_offset: An *optional* float with the reference offset, x_0 along the dimension in units of Hz. The default value is 0 Hz.
  - origin_offset: An *optional* float with the origin offset (Larmor frequency) along the dimension in units of Hz. The default value is None.
- **magetic_flux_density** – An *optional* float containing the macroscopic magnetic flux density, H_0, of the applied external magnetic field in units of T. The default value is 9.4.
- **rotor_angle** – An *optional* float containing the angle between the sample rotation axis and the applied external magnetic field, θ, in units of rad. The default value is 0.9553166, i.e. the magic angle.

Note: The attribute *rotor_frequency* cannot be modified for this method and is set to simulate an infinite speed spectrum.

Returns A *Method* (page 170) instance.

Example

```
>>> method = ST2_VAS(
...     channels=["17O"],
...     magnetic_flux_density=9.4,  # in T
...     spectral_dimensions=[
...         {
...             "count": 256,
...             "spectral_width": 4e3,  # in Hz
...             "reference_offset": -5e3,  # in Hz
...             "label": "Isotropic dimension",
...         },
...         {
...             "count": 512,
...             "spectral_width": 1e4,  # in Hz
...             "reference_offset": -4e3,  # in Hz
...             "label": "MAS dimension",
...         },
...     ],
... )
>>> sys = SpinSystem(sites=[Site(isotope='17O')])
>>> pprint(method.get_transition_pathways(sys))
[TransitionPathway(|-2.5>⟨-1.5|, |-0.5>⟨0.5|),
 TransitionPathway(|1.5>⟨2.5|, |-0.5>⟨0.5|)]
```

Spinning sideband correlation method

class mrsimulator.methods.**SSB2D**(***kwargs*)

A specialized method for simulating 2D finite speed to infinite speed MAS correlation spectum. For spin I=1/2, the infinite speed MAS is the isotropic dimension. The resulting spectrum is sheared.

Parameters

- **channels** – A list of isotope symbols over which the method will be applied.
- **spectral_dimensions** – A list of python dict. Each dict is contains keywords that describe the coordinates along a spectral dimension. The keywords along with its definition are:
  - count: An optional integer with the number of points, N, along the dimension. The default value is 1024.
  - spectral_width: An *optional* float with the spectral width, Δx, along the dimension in units of Hz. The default is 25 kHz.
  - reference_offset: An *optional* float with the reference offset, x_0 along the dimension in units of Hz. The default value is 0 Hz.
  - origin_offset: An *optional* float with the origin offset (Larmor frequency) along the dimension in units of Hz. The default value is None.
- **rotor_frequency** – An *optional* float containing the sample spinning frequency ν_r, in units of Hz. The default value is 0.
- **magetic_flux_density** – An *optional* float containing the macroscopic magnetic flux density, H_0, of the applied external magnetic field in units of T. The default value is 9.4.
- **rotor_angle** – An *optional* float containing the angle between the sample rotation axis and the applied external magnetic field, θ, in units of rad. The default value is 0.9553166, i.e. the magic angle.

Returns A *Method* (page 170) instance.

Example

```
>>> method = SSB2D(
...     channels=["13C"],
...     magnetic_flux_density=7,  # in T
...     rotor_frequency=1500, # in Hz
...     spectral_dimensions=[
...         {
...             "count": 16,
...             "spectral_width": 16*1500,  # in Hz (= count * rotor_frequency)
...             "reference_offset": -5e3,  # in Hz
...             "label": "Sideband dimension",
...         },
...         {
...             "count": 512,
...             "spectral_width": 1e4,  # in Hz
...             "reference_offset": -4e3,  # in Hz
...             "label": "Isotropic dimension",
...         },
...     ],
... )
>>> sys = SpinSystem(sites=[Site(isotope='13C')])
>>> method.get_transition_pathways(sys)
[TransitionPathway(|-0.5⟩⟨0.5|, |-0.5⟩⟨0.5|)]
```

6.2 Signal-processing API

6.2.1 Signal Processing

class mrsimulator.signal_processing.**SignalProcessor**(*, *processed_data: csdmpy.csdm.CSDM = None*, *operations: List[mrsimulator.signal_processing._base.AbstractOperation] = []*)

Bases: pydantic.main.BaseModel

Signal processing class to apply a series of operations to the dependent variables of the simulation dataset.

operations
A list of operations.

Type List

Examples

```
>>> post_sim = SignalProcessor(operations=[o1, o2])
```

Method Documentation

classmethod parse_dict_with_units(*py_dict*)
Parse a list of operations dictionary to a SignalProcessor class object.

Parameters **pt_dict** – A python dict object.

json()
Serialize the SignalProcessor object to a JSON compliant python dictionary object, where physical quantities are represented as string with a value and a unit.

Returns A Dict object.

apply_operations(*data*, ***kwargs*)
Function to apply all the operation functions in the operations member of a SignalProcessor object. Operations applied sequentially over the data member.

Returns A copy of the data member with the operations applied to it.

Return type CSDM object

6.2.2 Operations

Generic operations

Import the module as

```
>>> import mrsimulator.signal_processing as sp
```

Operation Summary

The following list of operations applies to **all dependent variables** within the CSDM object.

Scale	Scale the amplitudes of all dependent variables from a CSDM object.
IFFT	Apply an inverse Fourier transform on all dependent variables of the CSDM object.
FFT	Apply a forward Fourier transform on all dependent variables of the CSDM object.

Apodization

Import the module as

```
>>> import mrsimulator.signal_processing.apodization as apo
```

Operation Summary

The following list of operations applies to **selected dependent variables** within the CSDM object.

`Gaussian`	Apodize a dependent variable of the CSDM object with a Gaussian function.
`Exponential`	Apodize a dependent variable of the CSDM object by an exponential function.

See also:

Signal Processing (page 33) for a details.

Affine Transformation

Import the module as

```
>>> import mrsimulator.signal_processing.affine as af
```

Operation Summary

The following list of operations applies to **selected dependent variables** within the CSDM object.

`Shear`	Apply a shear parallel to dimension at index parallel and normal to dimension at index dim_index.
`Scale`	Scale the dimension along the specified dimension index.

See also:

Signal Processing (page 33) for a details.

6.2.3 Utility Functions

`mrsimulator.utils.spectral_fitting.`**`make_LMFIT_params`**(*sim*, *post_sim=None*, *exclude_key=None*)

Parses the Simulator and PostSimulator objects for a list of LMFIT parameters. The parameter name is generated using the following syntax:

`sys_i_site_j_attribute1_attribute2`

for spin system attribute with signature sys[i].sites[j].attribute1.attribute2

Parameters

- **`sim`** – a Simulator object.
- **`post_sim`** – a SignalProcessor object

Returns LMFIT Parameters object.

`mrsimulator.utils.spectral_fitting.`**`LMFIT_min_function`**(*params*, *sim*, *post_sim=None*)

The simulation routine to calculate the vector difference between simulation and experiment based on the parameters update.

Parameters

- **`params`** – Parameters object containing parameters to vary during minimization.

- **sim** – Simulator object used in the simulation. Initialized with guess fitting parameters.
- **post_sim** – PostSimulator object used in the simulation. Initialized with guess fitting parameters.

Returns Array of the differences between the simulation and the experimental data.

mrsimulator.utils.**get_spectral_dimensions**(*csdm_object*)

Extract the count, spectral_width, and reference_offset parameters, associated with the spectral dimensions of the method, from the CSDM dimension objects.

Parameters **csdm_object** – A CSDM object holding the measurement dataset.

Returns A list of dict objects, where each dict containts the count, spectral_width, and reference_offset.

6.3 Models API

6.3.1 Czjzek distribution Model

class mrsimulator.models.**CzjzekDistribution**(*sigma: float*, *polar=False*)

A Czjzek distribution model class.

The Czjzek distribution model is a random sampling of second-rank traceless symmetric tensors whose explicit matrix form follows

$$\mathbf{S} = \begin{bmatrix} \sqrt{3}U_5 - U_1 & \sqrt{3}U_4 & \sqrt{3}U_2 \\ \sqrt{3}U_4 & -\sqrt{3}U_5 - U_1 & \sqrt{3}U_3 \\ \sqrt{3}U_2 & \sqrt{3}U_3 & 2U_1 \end{bmatrix}, \tag{6.7}$$

where the components, U_i, are randomly drawn from a five-dimensional multivariate normal distribution. Each component, U_i, is a dimension of the five-dimensional uncorrelated multivariate normal distribution with the mean of $< U_i >= 0$ and the variance $< U_i U_i >= \sigma^2$.

$$S_T = S_C(\sigma), \tag{6.8}$$

Parameters **sigma** (*float*) – The Gaussian standard deviation.

Note: In the original Czjzek paper, the parameter σ is given as two times the standard deviation of the multi-variate normal distribution used here.

Example

```
>>> from mrsimulator.models import CzjzekDistribution
>>> cz_model = CzjzekDistribution(0.5)
```

rvs(*size: int*)

Draw random variates of length *size* from the distribution.

Parameters **size** – The number of random points to draw.

Returns A list of two NumPy array, where the first and the second array are the anisotropic/quadrupolar coupling constant and asymmetry parameter, respectively.

Example

```
>>> Cq_dist, eta_dist = cz_model.rvs(size=1000000)
```

pdf (*pos*, *size: int = 400000*)

Generates a probability distribution function by binning the random variates of length size onto the given grid system.

Parameters

- **pos** – A list of coordinates along the two dimensions given as NumPy arrays.
- **size** – The number of random variates drawn in generating the pdf. The default is 400000.

Returns A list of x and y coordinates and the corresponding amplitudes.

Example

```
>>> import numpy as np
>>> cq = np.arange(50) - 25
>>> eta = np.arange(21)/20
>>> Cq_dist, eta_dist, amp = cz_model.pdf(pos=[cq, eta])
```

Mini-gallery using czjzek distributions

- *Czjzek distribution (Shielding and Quadrupolar)* (page 69)
- *Czjzek distribution, 27Al (I=5/2) 3QMAS* (page 108)

6.3.2 Extended Czjzek distribution Model

class mrsimulator.models.**ExtCzjzekDistribution**(*symmetric_tensor:* mrsimulator.spin_system.tensors.SymmetricTensor *(page 167)*, *eps: float*, *polar=False*)

An extended czjzek distribution distribution model.

The extended Czjzek random distribution[1] model is an extension of the Czjzek model, given as

$$S_T = S(0) + \rho S_C(\sigma = 1), \tag{6.9}$$

where S_T is the total tensor, $S(0)$ is the dominant tensor, $S_C(\sigma = 1)$ is the Czjzek random model attributing to the random perturbation of the tensor about the dominant tensor, $S(0)$, and ρ is the size of the perturbation. Note, in the above equation, the σ parameter from the Czjzek random model, S_C, has no meaning and is set to one. The factor, ρ, is defined as

$$\rho = \frac{||S(0)||\epsilon}{\sqrt{30}}, \tag{6.10}$$

where $\|S(0)\|$ is the 2-norm of the dominant tensor, and ϵ is a fraction.

Parameters

- **symmetric_tensor** (SymmetricTensor (page 167)) – A shielding or quadrupolar symmetric tensor or equivalent dict object.

[1] Gérard Le Caër, Bruno Bureau, and Dominique Massiot, An extension of the Czjzek model for the distributions of electric field gradients in disordered solids and an application to NMR spectra of 71Ga in chalcogenide glasses. Journal of Physics: Condensed Matter, 2010, 22, 065402. DOI: 10.1088/0953-8984/22/6/065402

- **eps** (*float*) – A fraction determining the extent of perturbation.

Example

```
>>> from mrsimulator.models import ExtCzjzekDistribution
>>> S0 = {"Cq": 1e6, "eta": 0.3}
>>> ext_cz_model = ExtCzjzekDistribution(S0, eps=0.35)
```

rvs (*size: int*)

Draw random variates of length *size* from the distribution.

Parameters **size** – The number of random points to draw.

Returns A list of two NumPy array, where the first and the second array are the anisotropic/quadrupolar coupling constant and asymmetry parameter, respectively.

Example

```
>>> Cq_dist, eta_dist = ext_cz_model.rvs(size=1000000)
```

pdf (*pos*, *size: int = 400000*)

Generates a probability distribution function by binning the random variates of length size onto the given grid system.

Parameters

- **pos** – A list of coordinates along the two dimensions given as NumPy arrays.
- **size** – The number of random variates drawn in generating the pdf. The default is 400000.

Returns A list of x and y coordinates and the corresponding amplitudes.

Example

```
>>> import numpy as np
>>> cq = np.arange(50) - 25
>>> eta = np.arange(21)/20
>>> Cq_dist, eta_dist, amp = cz_model.pdf(pos=[cq, eta])
```

Mini-gallery using extended czjzek distributions

- *Extended Czjzek distribution (Shielding and Quadrupolar)* (page 73)
- *Simulating site disorder (crystalline)* (page 105)

6.4 C-API References

6.4.1 Spin transition functions (STF), $\xi_L^{(k)}(i,j)$

See also:

Spin transition functions (page 146)

Single nucleus spin transition functions

The single spin transition functions for $|m_i\rangle \rightarrow |m_f\rangle$ transition.

double **STF_p**(**const** double *mf*, **const** double *mi*)
Single nucleus spin transition function from irreducible tensor of rank $L = 1$, given as

$$\begin{aligned} \mathbb{p}(m_f, m_i) &= \left\langle m_f|\hat{T}_{10}|m_f\right\rangle - \left\langle m_i|\hat{T}_{10}|m_i\right\rangle \\ &= m_f - m_i, \end{aligned} \tag{6.11}$$

where $\hat{T}_{10}$ is the irreducible 1st-rank spherical tensor operator in the rotating tilted frame.

Return The spin transition function $\mathbb{p}$.

Parameters

- `mi`: The quantum number associated with the quantized initial energy level.
- `mf`: The quantum number associated with the quantized final energy level.

double **STF_d**(**const** double *mf*, **const** double *mi*)
Single nucleus spin transition function from irreducible tensor of rank $L = 2$, given as

$$\begin{aligned} \mathbb{d}(m_f, m_i) &= \left\langle m_f|\hat{T}_{20}|m_f\right\rangle - \left\langle m_i|\hat{T}_{20}|m_i\right\rangle \\ &= \sqrt{\frac{3}{2}}\left(m_f^2 - m_i^2\right), \end{aligned} \tag{6.12}$$

where $\hat{T}_{20}$ is the irreducible 2nd-rank spherical tensor operator in the rotating tilted frame.

Return The spin transition function $\mathbb{d}$.

Parameters

- `mi`: The quantum number associated with the quantized initial energy level.
- `mf`: The quantum number associated with the quantized final energy level.

double **STF_f**(**const** double *mf*, **const** double *mi*, **const** double *spin*)
Single nucleus spin transition function from irreducible tensor of rank $L = 3$, given as

$$\begin{aligned} \mathbb{f}(m_f, m_i) &= \left\langle m_f|\hat{T}_{30}|m_f\right\rangle - \left\langle m_i|\hat{T}_{30}|m_i\right\rangle \\ &= \frac{1}{\sqrt{10}}[5(m_f^3 - m_i^3) + (1 - 3I(I+1))(m_f - m_i)], \end{aligned} \tag{6.13}$$

where $\hat{T}_{30}$ is the irreducible 3rd-rank spherical tensor operator in the rotating tilted frame.

Return The spin transition function $\mathbb{f}$.

Parameters

- `mi`: The quantum number associated with the quantized initial energy level.
- `mf`: The quantum number associated with the quantized final energy level.

Composite single nucleus spin transition functions

void **`STF_cL`**(double ***restrict** *cl_value*, **const** double *mf*, **const** double *mi*, **const** double *spin*)

Single nucleus composite spin transition functions corresponding to rank $L = [0, 2, 4]$ irreducible tensors resulting from the second-order corrections to the quadrupole frequency. The functions are defined as

$$\begin{aligned}
\mathbb{c}_0(m_f, m_i) &= \frac{4}{\sqrt{125}}\left[I(I+1) - \frac{3}{4}\right]\mathbb{p}(m_f, m_i) + \sqrt{\frac{18}{25}}\mathbb{f}(m_f, m_i),\\
\mathbb{c}_2(m_f, m_i) &= \sqrt{\frac{2}{175}}\left[I(I+1) - \frac{3}{4}\right]\mathbb{p}(m_f, m_i) - \frac{6}{\sqrt{35}}\mathbb{f}(m_f, m_i),\\
\mathbb{c}_4(m_f, m_i) &= -\sqrt{\frac{18}{875}}\left[I(I+1) - \frac{3}{4}\right]\mathbb{p}(m_f, m_i) - \frac{17}{\sqrt{175}}\mathbb{f}(m_f, m_i),
\end{aligned} \tag{6.14}$$

where $\mathbb{p}(m_f, m_i)$ and $\mathbb{f}(m_f, m_i)$ are single nucleus spin transition functions described before, and I is the spin quantum number.

Parameters

- `mi`: The quantum number associated with the quantized initial energy level.
- `mf`: The quantum number associated with the quantized final energy level.
- `spin`: The spin quantum number, I.
- `cl_value`: A pointer to an array of size 3 where the spin transition functions, $\mathbb{c}_L$, will be stored ordered according to $L = [0, 2, 4]$.

6.4.2 Scaled spatial orientation tensors (sSOT), $\varsigma^{(k)}_{L,n}$

See also:

Scaled spatial orientation functions in PAS (page 144)

Single nucleus spatial orientation tensors

First order Nuclear shielding

void **`sSOT_1st_order_nuclear_shielding_tensor_components`**(double ***restrict** *R_0*, void ***restrict** *R_2*, **const** double *omega_0_delta_iso_in_Hz*, **const** double *omega_0_zeta_sigma_in_Hz*, **const** double *eta*, **const** double **Theta*)

The scaled spatial orientation tensors (sSOT) from the first-order perturbation expansion of the nuclear shielding Hamiltonian, in the principal axis system (PAS), includes the zeroth and second-rank irreducible tensors which follows,

$$\left.\varsigma^{(\sigma)}_{0,0} = \omega_0\delta_{\text{iso}}\right\} \text{Rank-0}, \tag{6.15}$$

$$\left.\begin{aligned} \varsigma^{(\sigma)}_{2,0} &= -\omega_0\zeta_\sigma, \\ \varsigma^{(\sigma)}_{2,\pm1} &= 0, \\ \varsigma^{(\sigma)}_{2,\pm2} &= \frac{1}{\sqrt{6}}\omega_0\eta_\sigma\zeta_\sigma, \end{aligned}\right\} \text{Rank-2}, \tag{6.16}$$

where σ_{iso} is the isotropic nuclear shielding, and, ζ_σ, η_σ are the shielding anisotropy and asymmetry parameters from the symmetric second-rank irreducible nuclear shielding tensor defined using Haeberlen convention. Here, $\omega_0 = -\gamma_I B_0$ is the Larmor frequency where, γ_I and B_0 are the gyromagnetic ratio of the nucleus and the magnetic flux density of the external magnetic field, respectively.

For non-zero Euler angles, $\Theta = [\alpha, \beta, \gamma]$, Wigner rotation of $\varsigma^{(\sigma)}_{2,n}$ is performed following,

$$\mathcal{R}'^{(\sigma)}_{2,n}(\Theta) = \sum_{m=-2}^{2} D^2_{m,n}(\Theta)\varsigma^{(\sigma)}_{2,n}, \tag{6.17}$$

where $\mathcal{R}'^{(\sigma)}_{2,n}(\Theta)$ are the tensors in the frame defined by the Euler angles Θ.

Note

- The method accepts a frequency physical quantity, that is, $\omega_0\sigma_{\text{iso}}/2\pi$ and $\omega_0\zeta_\sigma/2\pi$, as the isotropic nuclear shielding and nuclear shielding anisotropy, respectively.
- When $\Theta = [0, 0, 0]$, $\mathcal{R}'^{(\sigma)}_{2,n}(\Theta) = \varsigma^{(\sigma)}_{2,n}$ where $n \in [-2, 2]$.
- $\mathcal{R}'^{(\sigma)}_{0,0}(\Theta) = \varsigma^{(\sigma)}_{0,0} \quad \forall\, \Theta$.
- The method returns $\mathcal{R}'^{(\sigma)}_{0,0}(\Theta)/2\pi$ and $\mathcal{R}'^{(\sigma)}_{2,n}(\Theta)/2\pi$, that is, in **units of frequency**.

Parameters

- `R_0`: A pointer to an array of length 1 where the zeroth-rank irreducible tensor, $\mathcal{R}'^{(\sigma)}_{0,0}(\Theta)/2\pi$, will be stored.
- `R_2`: A pointer to a complex array of length 5 where the second-rank irreducible tensor, $\mathcal{R}'^{(\sigma)}_{2,n}(\Theta)/2\pi$, will be stored ordered according to $\left[\mathcal{R}'^{(\sigma)}_{2,n}(\Theta)/2\pi\right]^2_{n=-2}$.
- `omega_0_delta_iso_in_Hz`: The quantity, $\omega_0\sigma_{\text{iso}}/2\pi$, given in Hz.
- `omega_0_zeta_sigma_in_Hz`: The quantity, $\omega_0\zeta_\sigma/2\pi$, given in Hz.
- `eta`: The nuclear shielding asymmetry, $\eta_\sigma \in [0, 1]$.
- `Theta`: A pointer to an array of length 3 where Euler angles, ordered as $[\alpha, \beta, \gamma]$, are stored in radians.

First order Electric Quadrupole

void **sSOT_1st_order_electric_quadrupole_tensor_components**(void *__restrict__ *R_2*, __const__ double *spin*, __const__ double *Cq_in_Hz*, __const__ double *eta*, __const__ double **Theta*)

The scaled spatial orientation tensors (sSOT) from the first-order perturbation expansion of the electric quadrupole Hamiltonian, in the principal axis system (PAS), includes the second-rank irreducible tensor which follows,

$$\left.\begin{aligned} \varsigma^{(q)}_{2,0} &= \frac{1}{\sqrt{6}}\omega_q, \\ \varsigma^{(q)}_{2,\pm1} &= 0, \\ \varsigma^{(q)}_{2,\pm2} &= -\frac{1}{6}\eta_q\omega_q, \end{aligned}\right\} \text{Rank-2}, \tag{6.18}$$

where $\omega_q = \frac{6\pi C_q}{2I(2I-1)}$ is the quadrupole splitting frequency, and η_q is the quadrupole asymmetry parameter. Here, I is the spin quantum number of the quadrupole nucleus, and C_q is the quadrupole coupling constant.

As before, for non-zero Euler angles, $\Theta = [\alpha, \beta, \gamma]$, a Wigner rotation of $\varsigma^{(q)}_{2,n}$ is performed following,

$$\mathcal{R}'^{(q)}_{2,n}(\Theta) = \sum_{m=-2}^{2} D^2_{m,n}(\Theta)\varsigma^{(q)}_{2,n}. \tag{6.19}$$

where $\mathcal{R}'^{(q)}_{2,n}(\Theta)$ are the tensors in the frame defined by the Euler angles Θ.

Note

- When $\Theta = [0, 0, 0]$, $\mathcal{R}'^{(q)}_{2,n}(\Theta) = \varsigma^{(q)}_{2,n}$ where $n \in [-2, 2]$.
- The method returns $\mathcal{R}'^{(q)}_{2,0}(\Theta)/2\pi$, that is, in **units of frequency**.

Parameters

- `R_2`: A pointer to a complex array of length 5 where the the second-rank irreducible tensor, $\mathcal{R}'^{(q)}_{2,n}(\Theta)/2\pi$, will be stored ordered according to $\left[\mathcal{R}'^{(q)}_{2,n}(\Theta)/2\pi\right]^2_{n=-2}$.
- `spin`: The spin quantum number, I.
- `Cq_in_Hz`: The quadrupole coupling constant, C_q, in Hz.
- `eta`: The quadrupole asymmetry parameter, $\eta_q \in [0, 1]$.
- `Theta`: A pointer to an array of length 3 where Euler angles, ordered as $[\alpha, \beta, \gamma]$, are stored in radians.

Second order Electric Quadrupole

void **sSOT_2nd_order_electric_quadrupole_tensor_components**(double *__restrict__ *R_0*, void *__restrict__ *R_2*, void *__restrict__ *R_4*, __const__ double *spin*, __const__ double *v0_in_Hz*, __const__ double *Cq_in_Hz*, __const__ double *eta*, __const__ double *Theta*)

The scaled spatial orientation tensors (sSOT) from the second-order perturbation expansion of the electric quadrupole Hamiltonian, in the principal axis system (PAS), includes the zeroth, second and fourth-rank irreducible tensors which follows,

$$\left.\varsigma^{(qq)}_{0,0} = \frac{\omega_q^2}{\omega_0}\frac{1}{6\sqrt{5}}\left(\frac{\eta_q^2}{3}+1\right)\right\} \text{Rank-0}, \tag{6.20}$$

$$\left.\begin{aligned}\varsigma^{(qq)}_{2,0} &= \frac{\omega_q^2}{\omega_0}\frac{\sqrt{2}}{6\sqrt{7}}\left(\frac{\eta_q^2}{3}-1\right),\\ \varsigma^{(qq)}_{2,\pm1} &= 0,\\ \varsigma^{(qq)}_{2,\pm2} &= -\frac{\omega_q^2}{\omega_0}\frac{1}{3\sqrt{21}}\eta_q,\end{aligned}\right\} \text{Rank-2}, \tag{6.21}$$

$$\left.\begin{aligned}
\varsigma^{(qq)}_{4,0} &= \frac{\omega_q^2}{\omega_0}\frac{1}{\sqrt{70}}\left(\frac{\eta_q^2}{18}+1\right),\\
\varsigma^{(qq)}_{4,\pm1} &= 0,\\
\varsigma^{(qq)}_{4,\pm2} &= -\frac{\omega_q^2}{\omega_0}\frac{1}{6\sqrt{7}}\eta_q,\\
\varsigma^{(qq)}_{4,\pm3} &= 0,\\
\varsigma^{(qq)}_{4,\pm4} &= \frac{\omega_q^2}{\omega_0}\frac{1}{36}\eta_q^2,
\end{aligned}\right\}\text{Rank-4,} \tag{6.22}$$

where $\omega_q = \frac{6\pi C_q}{2I(2I-1)}$ is the quadrupole splitting frequency, ω_0 is the Larmor angular frequency, and η_q is the quadrupole asymmetry parameter. Here, I is the spin quantum number, and C_q is the quadrupole coupling constant.

For non-zero Euler angles, $\Theta = [\alpha, \beta, \gamma]$, Wigner rotation of $\varsigma^{(qq)}_{2,n}$ and $\varsigma^{(qq)}_{4,n}$ are performed following,

$$\begin{aligned}
\mathcal{R}'^{(qq)}_{2,n}(\Theta) &= \sum_{m=-2}^{2} D^2_{m,n}(\Theta)\varsigma^{(qq)}_{2,n},\\
\mathcal{R}'^{(qq)}_{4,n}(\Theta) &= \sum_{m=-4}^{4} D^4_{m,n}(\Theta)\varsigma^{(qq)}_{4,n}.
\end{aligned} \tag{6.23}$$

where $\mathcal{R}'^{(qq)}_{2,n}(\Theta)$ and $\mathcal{R}'^{(qq)}_{4,n}(\Theta)$ are the tensors in the frame defined by the Euler angles Θ.

Note

- When $\Theta = [0, 0, 0]$, $\mathcal{R}'^{(qq)}_{2,n}(\Theta) = \varsigma^{(qq)}_{2,n}$ where $n \in [-2, 2]$.
- When $\Theta = [0, 0, 0]$, $\mathcal{R}'^{(qq)}_{4,n}(\Theta) = \varsigma^{(qq)}_{4,n}$ where $n \in [-4, 4]$.
- $\mathcal{R}'^{(qq)}_{0,0}(\Theta) = \varsigma^{(qq)}_{0,0} \quad \forall\, \Theta$.
- The method returns $\mathcal{R}'^{(qq)}_{0,0}(\Theta)/2\pi$, $\mathcal{R}'^{(qq)}_{2,n}(\Theta)/2\pi$, and $\mathcal{R}'^{(qq)}_{4,n}(\Theta)/2\pi$, that is, in **units of frequency**.

Parameters

- `R_0`: A pointer to an array of length 1 where the zeroth-rank irreducible tensor, $\mathcal{R}'^{(qq)}_{0,0}(\Theta)/2\pi$, will be stored.
- `R_2`: A pointer to a complex array of length 5 where the second-rank irreducible tensor, $\mathcal{R}'^{(qq)}_{2,n}(\Theta)/2\pi$, will be stored ordered according to $\left[\mathcal{R}'^{(qq)}_{2,n}(\Theta)/2\pi\right]^2_{n=-2}$.
- `R_4`: A pointer to a complex array of length 9 where the fourth-rank irreducible tensor, $\mathcal{R}'^{(qq)}_{4,n}(\Theta)/2\pi$, will be stored ordered according to $\left[\mathcal{R}'^{(qq)}_{4,n}(\Theta)/2\pi\right]^4_{n=-4}$.
- `spin`: The spin quantum number, I.
- `v0_in_Hz`: The Larmor frequency, $\omega_0/2\pi$, in Hz.
- `Cq_in_Hz`: The quadrupole coupling constant, C_q, in Hz.
- `eta`: The quadrupole asymmetry parameter, $\eta_q \in [0, 1]$.
- `Theta`: A pointer to an array of length 3 where Euler angles, ordered as $[\alpha, \beta, \gamma]$, are stored in radians.

6.4.3 Frequency Tensors (FT), $\Lambda^{(k)}_{L,n}(i,j)$

See also:

Frequency component functions in PAS (page 147), *Spatial orientation functions (SOF)* (page 196), *Spin transition functions (STF)* (page 195)

Single nucleus frequency tensors

First order Nuclear shielding

void **FCF_1st_order_nuclear_shielding_tensor_components**(double *__restrict__ *Lambda_0*, void *__restrict__ *Lambda_2*, const double *omega_0_delta_iso_in_Hz*, const double *omega_0_zeta_sigma_in_Hz*, const double *eta*, const double *Theta*, const float *mf*, const float *mi*)

The frequency tensors (FT) from the first-order perturbation expansion of the nuclear shielding Hamiltonian, in a given frame, $\mathcal{F}$, described by the Euler angles $\Theta = [\alpha, \beta, \gamma]$ are

$$\begin{aligned} \Lambda'^{(\sigma)}_{0,0}(\Theta, i, j) &= \mathcal{R}'^{(\sigma)}_{0,0}(\Theta)\ \mathbb{p}(i,j), \text{ and} \\ \Lambda'^{(\sigma)}_{2,n}(\Theta, i, j) &= \mathcal{R}'^{(\sigma)}_{2,n}(\Theta)\ \mathbb{p}(i,j), \end{aligned} \tag{6.24}$$

where $\mathcal{R}'^{(\sigma)}_{0,0}(\Theta)$ and $\mathcal{R}'^{(\sigma)}_{2,n}(\Theta)$ are the spatial orientation functions in frame $\mathcal{F}$, and $\mathbb{p}(i,j)$ is the spin transition function for $|i\rangle \rightarrow |j\rangle$ transition.

Parameters

- `Lambda_0`: A pointer to an array of length 1 where the frequency components from $\Lambda'^{(\sigma)}_{0,0}(\Theta, i, j)$ will be stored.
- `Lambda_2`: A pointer to a complex array of length 5 where the frequency components from $\Lambda'^{(\sigma)}_{2,n}(\Theta, i, j)$ will be stored ordered according to $\left[\Lambda'^{(\sigma)}_{2,n}(\Theta, i, j)\right]^{2}_{n=-2}$.
- `omega_0_delta_iso_in_Hz`: The quantity, $2\pi\omega_0\delta_{\text{iso}}$, given in Hz.
- `omega_0_zeta_sigma_in_Hz`: The quantity, $2\pi\omega_0\zeta_s igma$, representing the strength of the nuclear shielding anisotropy, given in Hz, defined using Haeberlen convention.
- `eta`: The nuclear shielding asymmetry parameter, $\eta_\sigma \in [-1, 1]$, defined using Haeberlen convention.
- `Theta`: A pointer to an array of length 3 where Euler angles, ordered as $[\alpha, \beta, \gamma]$, are stored.
- `mf`: A float containing the spin quantum number of the final energy state.
- `mi`: A float containing the spin quantum number of the initial energy state.

First order Electric Quadrupole

void **FCF_1st_order_electric_quadrupole_tensor_components**(void *__restrict__ *Lambda_2*, const double *spin*, const double *Cq_in_Hz*, const double *eta*, const double *Theta*, const float *mf*, const float *mi*)

The frequency component function (FCF) from the first-order electric quadrupole Hamiltonian, in a given frame, $\mathcal{F}$, described by the Euler angles $\Theta = [\alpha, \beta, \gamma]$, is

$$\Lambda'^{(q)}_{2,n}(\Theta, i, j) = \mathcal{R}'^{(q)}_{2,n}(\Theta)\ \ \mathbb{d}(i, j), \tag{6.25}$$

where $\mathcal{R}^{(q)}_{2,n}(\Theta)$ are the spatial orientation functions in frame $\mathcal{F}$, and $\mathbb{d}(i, j)$ is the spin transition function for $|i\rangle \rightarrow |j\rangle$ transition.

Parameters

- `Lambda_2`: A pointer to a complex array of length 5 where the frequency components from $\Lambda'^{(q)}_{2,n}(\Theta, i, j)$ will be stored ordered according to $\left[\Lambda'^{(q)}_{2,n}(\Theta, i, j)\right]^{2}_{n=-2}$.
- `spin`: The spin quantum number, I.
- `Cq_in_Hz`: The quadrupole coupling constant, C_q, in Hz.
- `eta`: The quadrupole asymmetry parameter, $\eta_q \in [0, 1]$.
- `Theta`: A pointer to an array of length 3 where Euler angles, ordered as $[\alpha, \beta, \gamma]$, are stored.
- `mf`: A float containing the spin quantum number of the final energy state.
- `mi`: A float containing the spin quantum number of the initial energy state.

Second order Electric Quadrupole

void **FCF_2nd_order_electric_quadrupole_tensor_components**(double *__restrict__ *Lambda_0*, void *__restrict__ *Lambda_2*, void *__restrict__ *Lambda_4*, const double *spin*, const double *v0_in_Hz*, const double *Cq_in_Hz*, const double *eta*, const double *Theta*, const float *mf*, const float *mi*)

The frequency component functions (FCF) from the second-order electric quadrupole Hamiltonian, in a given frame, $\mathcal{F}$, described by the Euler angles $\Theta = [\alpha, \beta, \gamma]$, are

$$\begin{aligned}
\Lambda'^{(qq)}_{0,0}(\Theta, i, j) &= \mathcal{R}'^{(qq)}_{0,0}(\Theta)\ \ \mathbb{c}_0(i, j), \\
\Lambda'^{(qq)}_{2,n}(\Theta, i, j) &= \mathcal{R}'^{(qq)}_{2,n}(\Theta)\ \ \mathbb{c}_2(i, j), \text{ and} \\
\Lambda'^{(qq)}_{4,n}(\Theta, i, j) &= \mathcal{R}'^{(qq)}_{4,n}(\Theta)\ \ \mathbb{c}_4(i, j),
\end{aligned} \tag{6.26}$$

where $\mathcal{R}'^{(qq)}_{0,0}(\Theta)$, $\mathcal{R}'^{(qq)}_{2,n}(\Theta)$, and, $\mathcal{R}'^{(qq)}_{4,n}(\Theta)$ are the spatial orientation functions in frame $\mathcal{F}$, and $\mathbb{c}_i(i, j)$ are the composite spin transition functions for $|i\rangle \rightarrow |j\rangle$ transition.

Parameters

- `Lambda_0`: A pointer to an array of length 1 where the frequency component from $\Lambda'^{(qq)}_{0,0}(\Theta, i, j)$ will be stored.

- `Lambda_2`: A pointer to a complex array of length 5 where the frequency components from $\Lambda^{(qq)}_{2,n}(\Theta, i, j)$ will be stored ordered according to $\left[\Lambda'^{(qq)}_{2,n}(\Theta, i, j)\right]^{2}_{n=-2}$.
- `Lambda_4`: A pointer to a complex array of length 5 where the frequency components from $\Lambda'^{(qq)}_{4,n}(\Theta, i, j)$ will be stored ordered according to $\left[\Lambda'^{(qq)}_{4,n}(\Theta, i, j)\right]^{4}_{n=-4}$.
- `spin`: The spin quantum number, I.
- `Cq_in_Hz`: The quadrupole coupling constant, C_q, in Hz.
- `eta`: The quadrupole asymmetry parameter, $\eta_q \in [0, 1]$.
- `v0_in_Hz`: The Larmor frequency, ν_0, in Hz.
- `Theta`: A pointer to an array of length 3 where Euler angles, ordered as $[\alpha, \beta, \gamma]$, are stored.
- `mf`: A float containing the spin quantum number of the final energy state.
- `mi`: A float containing the spin quantum number of the initial energy state.

6.4.4 Angular Momentum Method Documentation

Generic methods

double **`wigner_d_element`**(**`const`** int *l*, **`const`** int *m1*, **`const`** int *m2*, **`const`** double *beta*)

Evaluates $d^l_{m_1,m_2}(\beta)$ wigner-d element of the given angle β.

Return The wigner-d element, $d^l_{m_1,m_2}(\beta)$.

Parameters

- `l`: The rank of the wigner-d matrix element.
- `m1`: The quantum number m_1.
- `m2`: The quantum number m_2.
- `beta`: The angle β given in radian.

void **`wigner_d_matrices`**(**`const`** int *l*, **`const`** int *n*, **`const`** double **beta*, double **wigner*)

Evaluates n wigner-d matrices of rank l at n angles given in radians.

At a given angle, β, the wigner-d matrix, $d^l\ (m_1, m_2|\beta)$, is a $(2l+1) \times (2l+1)$ square matrix where m_1 and m_2 range from $-l$ to l. The wigner-d elements, $d^l_{m_1,m_2}(\beta)$, are ordered with m_1 as the leading dimension. For example, when $l = 2$, the wigner-d elements are ordered according to

$$\left[d^2_{-2,-2}(\beta),\ d^2_{-1,-2}(\beta),\ d^2_{0,-2}(\beta),\ \ ...\ \ d^2_{1,2}(\beta),\ d^2_{2,2}(\beta)\right]. \tag{6.27}$$

The n matrices are stored such that the wigner-d matrix from angle at index i is stacked after the wigner-d matrix from angle at index $i-1$, that is, the wigner-d matrix corresponding to `angle[i]` starts at the index `i*(2*l+1)*(2*l+1)`.

Parameters

- `l`: The rank of the wigner-d matrix.
- `n`: The number of wigner-d matrix to evaluate. This is also the length of angle array, `beta`.
- `beta`: A pointer to an array of length n where the angles β are stored in radians.

- `wigner`: A pointer to an array of length `n*(2*l+1)*(2*l+1)`.

CHAPTER

SEVEN

PROJECT DETAILS

7.1 Changelog

7.1.1 v0.5.1

Bug fixes

- Fixed a bug that was causing incorrect spectral binning when the frequency contribution is pure isotropic.

Other changes

- The `to_dict_with_units()` method is deprecated and is replaced with `json()`
- The `json()` function returns a python dictionary object with minimal required keywords, where the event keys are globally serialized at the root method object. In the case where the event key value is different from the global value, the respective key is serialized within the event object.
- The `json()` function will no longer serialize the *transition_query* objects for the named objects.

7.1.2 v0.5.0

What's new

- ⭐ Improved simulation performance. ⭐ See our *Benchmark* (page 140).

The update introduces various two-dimensional methods for simulating NMR spectrum.

- Introduces a generic one-dimensional method, *Method1D* (page 176).
- Introduces a generic two-dimensional method, *Method2D* (page 180).
- Specialized two-dimensional methods for multi-quantum variable-angle spinning with build-in affine transformations.
  - *ThreeQ_VAS* (page 182),
  - *FiveQ_VAS* (page 183),
  - *SevenQ_VAS* (page 184)
- Specialized two-dimensional methods for satellite-transition variable-angle spinning with build-in affine transformations.
  - *ST1_VAS* (page 186),
  - *ST2_VAS* (page 187),

- Specialized two-dimensional isotropic/anisotropic sideband correlation method, *SSB2D* (page 188).

Other changes

- The *get_transition_pathways()* (page 172) method no longer return a numpy array, instead a python list.

7.1.3 v0.4.0

What's new!

- ⋆ Improved simulation performance. ⋆ See our *Benchmark* (page 140).
- New *CzjzekDistribution* (page 192) and *ExtCzjzekDistribution* (page 193) classes for generating Czjzek and extended Czjzek second-rank symmetric tensor distribution models for use in simulating amorphous materials.
- New utility function, `single_site_system_generator()`, for generating a list of single-site spin systems from a 1D list/array of respective tensor parameters.

7.1.4 v0.3.0

What's new!

- ⋆ Improved simulation performance. ⋆ See our *Benchmark* (page 140).
- Removed the `Dimension` class and added a new `Method` class instead.
- New methods for simulating the NMR spectrum:
  - `BlochDecaySpectrum` and
  - `BlochDecayCentralTransitionSpectrum`.

  The Bloch decay spectrum method simulates all p=Δm=-1 transition pathways, while the Bloch decay central transition selective spectrum method simulates all transition pathways with p=Δm=-1 and d=0.
- New `Isotope`, `Transition`, and `ZeemanState` classes.
- Every class now includes a `reduced_dict()` method. The reduced_dict method returns a dictionary with minimal key-value pairs required to simulate the spectrum. Note, this may cause metadata loss, if any.
- Added a `label` and `description` attributes to the `Site` class.
- Added a new `label` attribute to the `SpinSystem` class.
- New `SignalProcessor` class for post-simulation signal processing.
- Improved usage of least-squares minimization using python LMFIT package.
- Added a new `get_spectral_dimensions` utility function to extract the spectral dimensions information from the CSDM object.

Bug fixes

- Fixed bug resulting from the rotation of the fourth rank tensor with non-zero euler angles.
- Fixed bug causing a change in the spectral area as the sampling points change. Now the area is constant.
- Fixed bug resulting in an incorrect spectrum when non-coincidental quad and shielding tensors are given.
- Fixed bug causing incorrect generation of transition pathways when multiple events are present.

Other changes

- Renamed the `decompose` attribute from the `ConfigSimulator` class to `decompose_spectrum`. The attribute is an enumeration with the following literals:
  - `none`: Computes a spectrum which is an integration of the spectra from all spin systems.
  - `spin_system`: Computes a series of spectra each corresponding to a single spin system.
- Renamed `Isotopomer` class to `SpinSystem`.
- Renamed `isotopomers` attribute from `Simulator` class to `spin_systems`.
- Renamed `dimensions` attribute from `Simulator` class to `methods`.
- Changed the default value of `name` and `description` attribute from the `SpinSystem` class from `""` to `None`.

7.1.5 v0.2.x

What's new!

- Added more isotopes to the simulator. Source NMR Tables (https://apps.apple.com/bn/app/nmr-tables/id1030899609?mt=12).
- Added two new keywords: *atomic_number* and *quadrupole_moment*.
- Added documentation for every class.
- Added examples for simulating NMR quadrupolar spectrum.

Bug fixes

- Fixed amplitude normalization. The spectral amplitude no longer change when the *integration_density*, integration_volume', or the *number_of_sidebands* attributes change.

Other changes

- Removed plotly-dash app to its own repository.
- Renamed the class `Spectrum` to `Dimension`

7.1.6 v0.1.3

- Fixed missing files from source tar.

7.1.7 v0.1.2

- Initial release on pypi.

7.1.8 v0.1.1

- Added solid state quadrupolar spectrum simulation.
- Added mrsimulator plotly-dash app.

7.1.9 v0.1.0

- Solid state chemical shift anisotropy spectrum simulation.

7.2 Authors and Credits

- Deepansh Srivastava
- Maxwell C. Venetos
- Philip J. Grandinetti
- Shyam Dwaraknath

7.3 License

7.3.1 Mrsimulator License

Mrsimulator is licensed under BSD 3-Clause License

Copyright (c) 2019-2020, Mrsimulator Developors,

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

7.4 Acknowledgment

The development of the mrsimulator project was supported in part by the US National Science Foundation under Grant No. DIBBS OAC 1640899 and Grant No. CHE 1807922.

CHAPTER

EIGHT

REPORTING BUGS

The preferred location for submitting feature requests and bug reports is the Github issue tracker. Reports are also welcomed on the mrsimulator mailing list or by directly contacting Deepansh Srivastava.

INDEX

T

U

W

Z