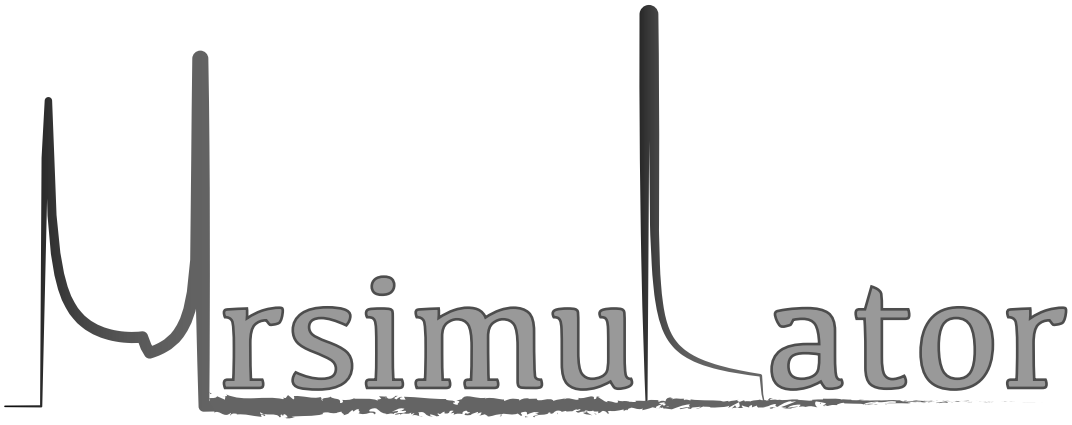


Mrsimulator Documentation

Release 0.8.0rc

The Mrsimulator Developers

2024

INTRODUCTION

LIST OF FIGURES

LIST OF TABLES

Part I

About

mrsimulator is an open-source Python package for fast computation/analysis of nuclear magnetic resonance (NMR) spectra in fluid and solid phases.

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.
- It includes simulation of weakly coupled nuclei experiencing J and dipolar couplings.
- It is fully documented with a stable and simple API and is easily incorporated into Python scripts and web apps.
- It is compatible with modern Python packages, such as Scikit-learn, Keras, etc.
- Packages using **mrsimulator** -
  - mrinversion

A brief example

```
from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.method.lib import BlochDecaySpectrum
import matplotlib.pyplot as plt

# Make Site and SpinSystem objects
H_site = Site(isotope="1H", shielding_symmetric={"zeta": 13.89, "eta": 0.25})
spin_system = SpinSystem(sites=[H_site])

# Make static and MAS one-pulse acquire Method objects
static = BlochDecaySpectrum(channels=["1H"])
mas = BlochDecaySpectrum(channels=["1H"], rotor_frequency=1000)  # in Hz

# Setup and run the Simulation object
sim = Simulator(spin_systems=[spin_system], methods=[static, mas])
sim.run()

# Plot the spectra
fig, ax = plt.subplots(1, 2, figsize=(6, 3), subplot_kw={"projection": "csdm"})
ax[0].plot(sim.methods[0].simulation)
ax[0].set_title("Static")
ax[1].plot(sim.methods[1].simulation)
ax[1].set_title("MAS")
plt.tight_layout()
plt.show()
```

Note: Throughout the web version of this documentation, you can copy code blocks into your clipboard by hovering over the top right corner of each gray code block and clicking the copy-to-clipboard icon. This is useful for copying code examples into your Python scripts and Jupyter notebooks.

Features

The **mrsimulator** package offers the following

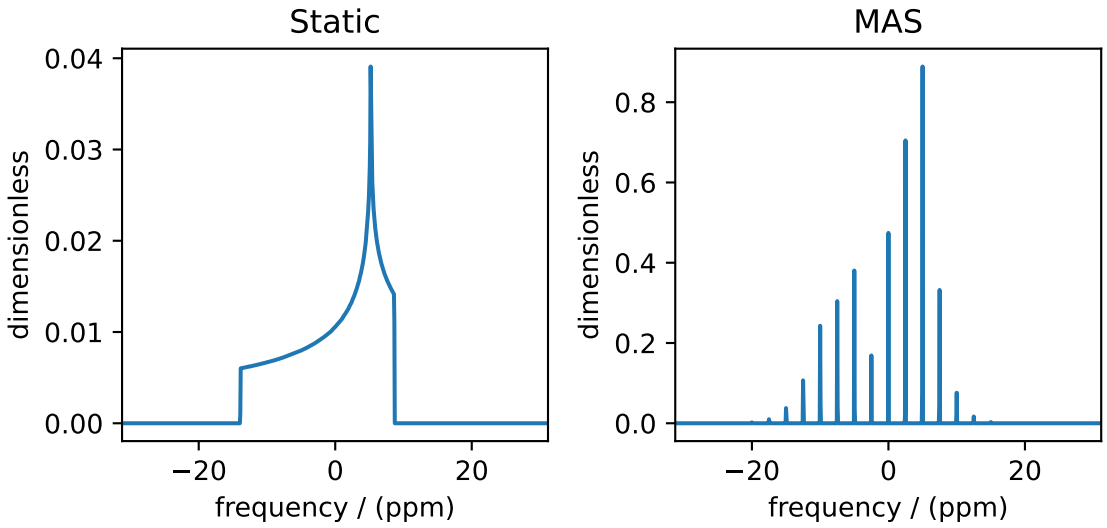


Figure 1: Simulation of static and MAS solid-state NMR spectra

- **Fast simulation** of one and two-dimensional solid-state NMR spectra.
- **Simulation of coupled and 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
  - at arbitrary spinning frequency
- **A library of pre-built NMR methods,**
  - 1D Bloch decay spectrum
  - 1D Bloch decay central transition spectrum
  - 2D Multi-Quantum Variable Angle Spinning (MQ-VAS)
  - 2D Satellite-Transition Variable Angle Spinning (ST-VAS)
  - 2D isotropic/anisotropic sideband correlation spectrum (e.g. PASS and MAT)
  - 2D Magic-Angle Flipping (MAF)
  - 2D Dynamic-Angle Spinning (DAS)
  - Custom user-defined methods (Method)
- **Models for tensor parameter distribution in amorphous materials.**
  - Czjzek
  - Extended Czjzek
  - Custom user-defined models

Part II

Introduction

CHAPTER

ONE

INSTALLATION

1.1 For the users

Note: If you encounter an issue during installation, see our troubleshooting section. If that doesn't resolve your issue, please create a bug report on our Github issue tracker.

1.1.1 Strict Requirements

mrsimulator has the following strict requirements:

- Python 3.8 or later
- Numpy 1.17 or later

See *Package dependencies* (page 21) for a full list of requirements.

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

```
$ python --version
```

For *MacOS* 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, Python is not usually installed by default. See Python.org for a list of official Python downloads and Windows installation instructions.

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 installing Python.

1.1.2 Installing mrsimulator

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. We recommend going through the *Welcome to Colab!* tutorial if you are new to Notebooks.

To install the **mrsimulator** package, type

```
!pip install mrsimulator
```

in a new cell, execute, and finally restart the runtime. All done! You may now start using the library, or proceed to *Getting Started* (page 23) to continue the tutorial.

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. Install the package using pip as follows,

```
$ pip install mrsimulator
```

For *Mac* users, if the above statement didn't work, you are probably using MacOS system python, in which case, use the following,

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

If you get a `PermissionError`, it usually means that you do not have the 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.

From 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 installing the C-compiler (if applicable), BLAS, FFTW, and building the **mrsimulator** libraries.

Obtaining the Source Packages

The latest stable source package for **mrsimulator** is available on PyPI and Github release. Download and extract the *.tar.gz* file.

OS-dependent prerequisites

Note: Installing OS-dependent prerequisites is a one-time process. If upgrading to a newer version of mrsimulator, skip to the next 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.

(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 >= 3.8 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.

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
```

1.1.3 Updating mrsimulator

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

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

All done! You may now start using the library or proceed to *Getting Started* (page 23) to continue the tutorial.

1.1.4 Testing your build

Note: For Windows users using anaconda Python 3.8 and higher, you need to set the following environment variable in the `Anaconda Prompt` before running mrsimulator scripts.

```
$ set CONDA_DLL_SEARCH_MODIFICATION_ENABLE='1'
```

If the installation is successful, you should be able to run the following test file in your terminal. Download the test file here or copy and paste the following code into a Python file and run the code.

```
from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.method.lib import BlochDecaySpectrum
import matplotlib.pyplot as plt

# Make Site and SpinSystem objects
```

(continues on next page)

(continued from previous page)

```
H_site = Site(isotope="1H", shielding_symmetric={"zeta": 13.89, "eta": 0.25})
spin_system = SpinSystem(sites=[H_site])

# Make static and MAS one-pulse acquire Method objects
static = BlochDecaySpectrum(channels=["1H"])
mas = BlochDecaySpectrum(channels=["1H"], rotor_frequency=1000)  # in Hz

# Setup and run the Simulation object
sim = Simulator(spin_systems=[spin_system], methods=[static, mas])
sim.run()

# Plot the spectra
fig, ax = plt.subplots(1, 2, figsize=(6, 3), subplot_kw={"projection": "csdm"})
ax[0].plot(sim.methods[0].simulation.real)
ax[0].set_title("Static")
ax[1].plot(sim.methods[1].simulation.real)
ax[1].set_title("MAS")
plt.tight_layout()
plt.show()
```

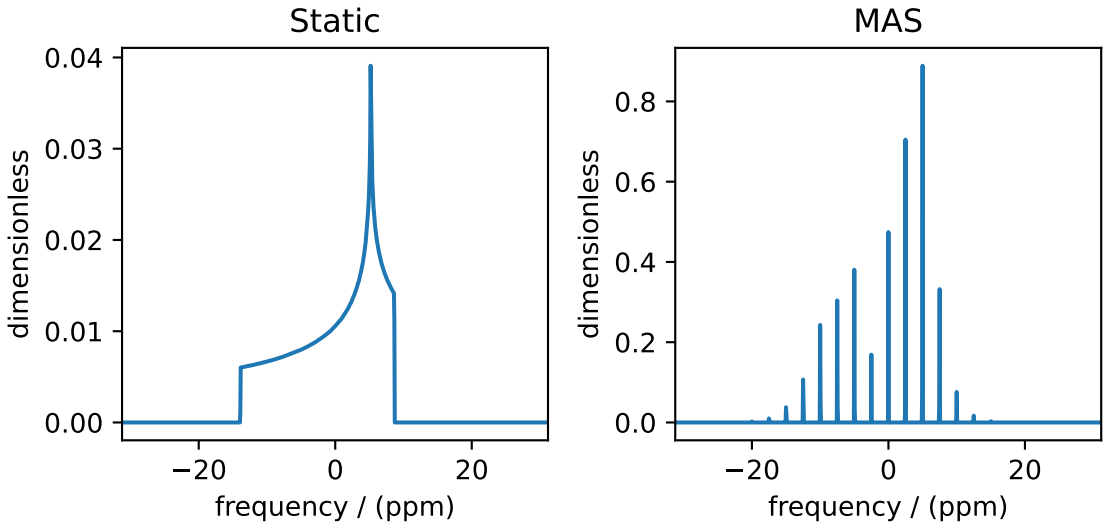


Figure 1.1: Simulation of static and MAS solid-state NMR spectra

Note: If you encounter the following error

```
ValueError: numpy.ndarray size changed, may indicate binary incompatibility.
Expected 88 from C header, got 80 from PyObject
```

update numpy by running

```
$ pip install -U numpy
```

1.2 For developers and contributors

1.2.1 Setting up a dedicated code editor

Using a code editor or IDE is useful when contributing to a codebase. Many products are available; use what is most familiar. For new developers, we recommend VS Code since it is lightweight, free, and has a breadth of community extensions.

1.2.2 Make your copy of mrsimulator on GitHub

Making a copy of someone's code on GitHub is the same as making a *fork*. A fork is a complete copy of the code and its revision history.

1. Log in to a GitHub account.
2. Go to the mrsimulator Github home page.
3. Click on the *fork* button.

You will see a short animation of Octocat scanning a book on a flatbed scanner. After that, you should find yourself on the home page for your forked copy of mrsimulator.

1.2.3 Create a development environment

It is good practice to create separate virtual environments when developing packages. There are many environment managers available; however, we recommend using Anaconda or Miniconda.

Note: For Mac users with Apple Silicon, Anaconda and Miniconda are natively supported on M1 as of release 2022.05. See the downloads page for compatible versions.

If your Python is built for Apple Silicon, the following command should display similar output.

```
$ file `which python`
/some/path/to/python: Mach-O 64-bit executable arm64
```

The following is an example of creating a Conda environment.

```
$ conda create -n mrsimulator-dev python=3.9
```

The above command will create a new environment named *mrsimulator-dev* using Python 3.9. To activate the environment, use

```
$ conda activate mrsimulator-dev
```

1.2.4 Make sure git is installed on your computer

Git is a source code management system. It keeps track of the changes made to the code and manages contributions from several individuals. You may notice that much of its terminology comes from river and tree metaphors, i.e., source, fork, branch, upstream, etc. You may read about git at the Git Basics.

If you are using anaconda/miniconda, you probably have git pre-installed. To check, type in terminal

```
$ git --version
# if git is installed, you will get something like git version 2.30.2
```

If git is not installed, install it before continuing.

Basic git configuration:

Follow the instructions at Set Up Git at GitHub to configure:

- your user name and email in your copy of git.
- authentication, so you don't have to type your GitHub password every time you

You'll need to access GitHub from the command line.

1.2.5 Copy your fork of mrsimulator from GitHub to your computer

Unless you plan on always editing the code using the online Github editor, you may need to copy the fork of **mrsimulator** from your GitHub account to your computer. Make a complete copy of the fork with

```
$ git clone --recursive https://github.com/your-user-name/mrsimulator.git
```

Insert *your-user-name* with your GitHub account username. If there is an error at this stage, it is probably an error in setting up authentication.

You now have a copy of the **mrsimulator** fork from your GitHub account to your local computer into a **mrsimulator** folder.

1.2.6 Understanding *Remotes*

In git, the name for another location of the same repository is *remote*. The repository that contains the latest "official" development version is traditionally called the *upstream* remote. You can read more about remotes on Git Basics.

At this point, your local copy of **mrsimulator** doesn't know where the *upstream* development version of **mrsimulator** is. To let git know, change into the **mrsimulator** folder you created in the previous step, and add a remote:

```
cd mrsimulator
git remote add mrsimulator git://github.com/deepanshs/mrsimulator.git
```

You can check that everything is set up correctly so far by asking git to show you all of the remotes it knows about for your local repository of **mrsimulator** with `git remote -v`, which should display

```
upstream git://github.com/deepanshs/mrsimulator.git (fetch)
upstream git://github.com/deepanshs/mrsimulator.git (push)
origin git@github.com:your-user-name/mrsimulator.git (fetch)
origin git@github.com:your-user-name/mrsimulator.git (push)
```

1.2.7 Build the development version of mrsimulator

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 the next 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.

(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 >= 3.8 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.

Build and install

Before building the development version of mrsimulator, install the development requirement packages with pip. In the directory where your copy of **mrsimulator** is, type:

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

As before, 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` option.

Note: If you are using a Mac with Apple Silicon and unable to install **mrsimulator**, open an issue on the GitHub page.

1.2.8 Note for the developers and contributors

Before commits: **mrsimulator** follows Python community standards for writing code and documentation. To help guide the developers and contributors toward these standards, we have created a *.pre-commit-config.yaml* file that, when used with `pre-commit`, will inspect the code and document for issues. To set up `pre-commit`, type the following one-time install statement in the terminals,

```
$ pre-commit install
```

Once set up, navigate to the root level of the **mrsimulator** folder and type

```
$ pre-commit run
```

The above statement auto-fixes some issues and lists others for you to fix. Review the changes and address the listed issues before a git commit.

Note: The pre-commit command ignores unstaged changes. Before running `pre-commit run`, make sure to stage files for a commit.

Running tests: We use the pytest module for unit tests. At the root level of the **mrsimulator** folder, type

```
$ pytest
```

which will run a series of tests alerting you to any unit tests that fail.

Checking test coverage: To check which lines in the codebase are covered when running a test, use the following command.

```
$ pytest --cov-report=html
```

To view the unit test coverage report, open the *mrsimulator/htmlcov/index.html* file in a web browser.

Building docs: We use the sphinx Python documentation generator for building docs. Navigate to the *docs* directory within the **mrsimulator** folder, and type,

```
$ make html
```

The above command will build the documentation and store the build at *mrsimulator/docs/_build/html*. Open the *index.html* file in a web browser within this folder to view the locally-built documentation.

1.3 Troubleshooting

We've compiled solutions to some common issues encountered when installing and using mrsimulator. However, this list is by no means comprehensive and may change as mrsimulator is continuously updated.

If the following sections don't resolve your issue, we ask that you open an issue on the GitHub issue tracker for problems related to installing and using mrsimulator. For other issues, such as installing Python or a code editor, we ask you to open a discussion post on the GitHub discussion page.

1.3.1 Installing Python

Mrsimulator requires Python or a hosted Notebook service to run. If you are using Google Colab, see specific instructions in the *For the users* (page 7) section.

Checking the version of Python

Windows

MacOS

Linux

To check if Python is installed on a Windows machine, first open the Command Prompt application. Next, type

```
python -V
```

and press enter. `Python 2.7.x` or `Python 3.x.x` should be printed to the console. However, if Python is not installed, an error message like the following will appear:

```
'python' is not recognized as an internal or external command, operable program or batch file.
```

To install Python, visit python.org to download a version of Python 3. During the installation process, check **add Python 3.x to PATH**. If this isn't selected, Python may not be accessible across your computer and will cause errors.

If you are certain Python is installed on your system but continue to receive errors, see adding Python to your PATH variable.

Nearly all recent versions of MacOS come with Python pre-installed. If you're unsure if Python is installed, follow these steps.

To check if Python is installed on MacOS, open the Terminal application. Next, type

```
python -V
```

and press enter. `Python 2.7.x` or `Python 3.x.x` should be printed to the console. However, if Python is not installed, an error message like the following will appear:

```
zsh: command not found: python
```

To install Python, visit python.org to download a version of Python 3 for your system.

To check if Python is installed on Linux, open a terminal. Next, type

```
python -V
```

and press enter. `Python 2.7.x` or `Python 3.x.x` should be printed to the console. However, if Python is not installed, an error message like the following will appear:

```
bash: python: command not found
```

To install Python, visit python.org to download a version of Python 3 for your system.

Updating Python

If Python is already installed on your system but is out of date, we recommend installing Anaconda to manage Python versions. Anaconda is versatile and allows multiple versions of Python to run on one computer without interfering with each-other.

However, if Anaconda can't be used, newer versions of Python can be installed from python.org. Python 3.8 or greater is required to run mrsimulator, but we encourage using the latest compatible version of Python.

1.3.2 Google Colab and Jupyter Notebooks

If you are new to Google Colab or just need a refresher, we suggest going through Google's introduction to Colaboratory.

Google Colab has the same functionality as a Jupyter Notebooks. The main difference is that Jupyter Notebooks are run on your local machine. The Jupyter Notebook documentation details installation and use.

Updating Numpy

Google Colab is versatile and platform independent, but some of the pre-installed libraries may be old version. When running **mrsimulator** on Google Colab, an incompatible version of `numpy` will cause the following error

```
ValueError: numpy.ndarray size changed, may indicate binary incompatibility. Expected 88 from C␣
↪header, got 80 from PyObject
```

To update `numpy` in Google Colab, execute the following command in a new cell.

```
!pip install -U numpy
```

This step may take a few seconds. After updating, a warning should pop up saying

```
WARNING: The following packages were previously imported in this runtime: [numpy]    You must␣
↪restart the runtime in order to use newly installed versions.
```

Press the restart runtime button and `numpy` should be up-to-date. Remember to re-run all code cells since all previous outputs are cleared after restarting the runtime.

Order of Cell Execution

For Jupyter Notebooks and Google Colab, the order of cell execution is important. If a variable `foo` is referenced before assignment, Python will throw `NameError: name 'foo' is not defined`.

If you get this error when working in a notebook, first check the cell where your variable was defined has been executed. A cell can be executed by pressing `shift + enter` while the text cursor is in that cell, or by pressing the run button near the top-left of the cell.

Similarly, if you've reassigned a variable but the code isn't reflecting that reassignment, check to make sure the cell where the variable was reassigned has been executed. Value changes are only recognized after cell execution.

1.3.3 Virtual Environments

Creating a Python environment using Anaconda

Since different Python packages have different dependencies, installing multiple packages on the same machine can cause issues. For example, **mrsimulator** requires at least `numpy v1.17` but `some-other-library` might require exactly `numpy v1.15`. These two libraries would likely throw errors when run in the same environment.

For this reason, we recommend using an environment manager, like `venv` or `anaconda`. We will look at anaconda for its simple commands. Installation instructions can be found on the anaconda documentation page.

Note: Anaconda is a robust package and environment management program, but it does require a significant amount of space on disk (>400mb). If you need a lightweight environment manager and are confident with Python, we recommend looking at Python's venv documentation.

Once Anaconda is installed, create a new environment by running

```
$ conda create -n <name> python=3.9
```

where `<name>` is the desired name of your environment. Each environment can have a Python version specified after `python=`. We recommend using `python=3.9`. Next activate the environment by running

```
$ conda activate <name>
```

The current environment name is reflected in the leftmost portion of a line. On MacOS, an environment named `mrsimulator-0.7` should look like

```
(mrsimulator-0.7) nmruser@machine $
```

If you are using a code editor or IDE, the current environment should be displayed somewhere on the window. For VS Code, the environment name and Python version are shown in the bottom-left corner.

To install **mrsimulator** in this new environment, follow the *installation* (page 7) instructions. **Mrsimulator** and any other libraries will only be installed in the active environment. This way different projects can run in separate environments.

To exit the environment run

```
$ conda deactivate
```

To start using **mrsimulator** again, simply activate the environment in which it was installed.

Packages installed in an environment remain installed between sessions and won't interfere with packages in other environments.

1.3.4 Common Python Syntax Errors

Python syntax is slightly different than other languages, which can cause some confusion. A dedicated code editor is the easiest way to find and prevent syntax errors. We recommend using VS Code on your local machine or Google Colab, which runs everything online. These programs check for syntax errors as you write code. The following are some typical syntax errors encountered and how to solve them.

IndentationError

If you encounter an `IndentationError`, you have an extra/missing whitespace in your code. Code editors make finding troublesome whitespace easier, but the error should also show the code snippet which threw the error.

- `IndentationError: expected an indented block` means some code is missing an indent after a class/method/loop deceleration.
- `IndentationError: unindent does not match any outer indentation level` means the code didn't return to a previous indentation level.
- `IndentationError: unexpected indent` means Python encountered unexpected whitespace.

Code blocks in Python rely on indentation levels (1 level = 4 spaces), so whitespace can't be placed randomly. Code blocks are preceded by a `:`, and all code in one block has the same indentation. To get out of a code block, remove an indentation level.

As an example of indentation, here is some code that adds the numbers 0 to 9:

```
# Add numbers 0 through 9
total = 0
for i in range(10):
    # New code block (4 spaces)
    total += i
# Exit loop code block (0 spaces)
print(total)
```

Mismatched Brackets and Square Brackets

Nesting many lists and dictionaries inside each other become hard to read. If you have mismatched or missing brackets, Python will throw `SyntaxError: invalid syntax`. Code editors can automatically format large nestings and highlight which openings and closings go together, making the code easier to understand.

Make sure all brackets are balanced and that opening and closing brackets match. Python uses three types of brackets:

- `()` is used when creating a tuple or when creating/calling method signatures.
- `[]` is used when creating a list or when indexing an item in a list or tuple.
- `{}` is used when creating a dict or set.

TypeError: object is not callable

The most common reason `TypeError: object is not callable` is when `()` is used instead of `[]`. Parentheses are used to call functions. For example

```
def foo(n):
    print("I received", n)


foo(1)
# I received 1
```

But parentheses aren't valid for indexing a subscriptable object (list, tuple, etc.). For example, the following code will throw a TypeError

```
bar = [1, 2, 3, 4]
bar(1)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'list' object is not callable
```

but the following code is valid

```
bar = [1, 2, 3, 4]
print(bar[1])
# 2
```

The same applies to dictionaries, but instead of indexing with an integer, you would index with a keyword. For example

```
spam = {"ham": "Hello World!", "eggs": 54.73}
print(spam["ham"])  # prints Hello World!
print(spam("ham"))  # throws error
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'dict' object is not callable
```

TypeError: object is not subscriptable

`TypeError: object is not subscriptable` is thrown when indexing a non-subscriptable object. For example

```
some_num = 42
some_num[3]
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not subscriptable
```

Also, there is a limit to how times you can index a subscriptable object. A 1D list can only be indexed once, 2D twice, and so on. If you are using nested lists/dicts, make sure you aren't exceeding the number of indexes possible.

1.4 Package dependencies

Mrsimulator works with Python versions > 3.8 and is compatible with the following operating systems:

- MacOS 10.15 or later
- Windows 7 or later
- Most releases of Linux

---

mrsimulator depends on the following packages:

Required packages

- numpy>=1.20
- matplotlib>=3.3.4 for figures and visualization
- lmfit>=1.0.3 for least-squares fitting
- pandas>=1.1.3
- csdmpy>=0.6

- pydantic<2
- nmrglue>=0.9
- monty>=2.0.4
- typing-extensions>=3.7
- numexpr==2.8.4
- psutil>=5.4.8
- joblib>=1.0.0

Required libraries for local build

- openblas
- fftw

For `mrsimualtor` developers, the following packages are required:

For building C libraries

- cython>=0.29.14

For unit tests - sympy - pytest<8.0 - pytest-cov - sybil>=3.0.0

For formatting

- black
- pre-commit>=2.11.1

For building documentation

- sphinxjp.themes.basicstrap
- sphinx<=6.0
- sphinx-gallery>=0.10
- pillow>=7.1.2
- breathe==4.34.0
- sphinx_copybutton>=0.3.0
- sphinx-tabs>=1.1.13
- recommonmark
- sphinx-version-warning

CHAPTER

TWO

GETTING STARTED

In mrsimulator, the user initializes objects from mrsimulator classes; The three main classes we will use in this example are: *Spin System* (page 51), *Method* (page 79), and *Simulator* (page 121).

SpinSystem defines the spin system and its tensor parameters used to generate a particular subspectrum, and Method defines the behavior and parameters for the particular NMR measurement to be simulated. A list of Method and SpinSystem objects are used to initialize a Simulator object, which is then used to generate the corresponding NMR spectra—returned as a CSDM object in each Method object. For more information on the CSDM (Core Scientific Dataset Model), see the csdmpy documentation. There is an additional class, *Signal Processor* (page 133), for applying various post-simulation signal processing operations to CSDM dataset objects.

All objects in **mrsimulator** can be serialized. We adopt the Javascript Object Notation (JSON) as the file-serialization format for the model because it is human-readable if properly organized and easily integrable with numerous programming languages and related software packages. It is also the preferred serialization for data exchange in web-based applications.

Here, we have put together a tutorial which introduces the key objects in a typical **mrsimulator** workflow. See the User Documentation section for more detailed documentation on the usage of **mrsimulator** classes. Also, check out our *Simulation Gallery* (page 145) and *Fitting (Least Squares) Gallery* (page 255).

2.1 SpinSystem

An NMR spin system is an isolated system of sites (spins) and couplings. Spin systems can include as many sites and couplings as necessary to model a sample. For this introductory example, you will create a coupled ^{1}H - ^{13}C spin system. Use the code below to construct two *Site* (page 52) objects for the ^{1}H and ^{13}C sites.

```
# Import the Site and SymmetricTensor classes
from mrsimulator import Site
from mrsimulator.spin_system.tensors import SymmetricTensor

# Create the Site objects
H_site = Site(isotope="1H")
C_site = Site(
    isotope="13C",
    isotropic_chemical_shift=100.0,  # in ppm
    shielding_symmetric=SymmetricTensor(
        zeta=70.0,  # in ppm
        eta=0.5,
    ),
)
my_sites = [H_site, C_site]
```

Note that isotopes in **mrsimulator** are specified with a string that starts with the isotope's mass number followed by its element symbol.

In the code above, you created two Site objects in the variables `H_site` and `C_site`. The `H_site` variable represents a proton site with a (default) chemical shift of zero. The `C_site` variable represents a carbon-13 site with a chemical shift of 100 ppm and a shielding component represented by a *SymmetricTensor* (page 475) object. We parametrize tensors using the Haeberlen convention. All spin interaction parameters, e.g., isotropic chemical shift and other coupling parameters, are initialized to zero by default. Additionally, the default Site isotope is `1H`.

At the end of the code above, you placed `H_site` and `C_site` into a Python list named `my_sites`. The order of Sites in this list is important, as the indexes of Sites in this list are used when specifying couplings between sites. Note that indexes in Python start at zero.

Using the code below, define a dipolar coupling between `H_site` and `C_site` by creating a *Coupling* (page 53) object.

```
# Import the Coupling class
from mrsimulator import Coupling

# Create the Coupling object
coupling = Coupling(
    site_index=[0, 1],
    dipolar=SymmetricTensor(D=-2e4),  # in Hz
)
```

The two sites involved in the Coupling are identified by their indexes in the list variable `site_index`.

Now you have all the pieces needed to create the spin system using the code below.

```
# Import the SpinSystem class
from mrsimulator import SpinSystem

# Create the SpinSystem object
spin_system = SpinSystem(
    sites = my_sites,
    couplings=[coupling],
)
```

That's it! You have created a spin system whose spectrum is ready to be simulated. If you had wanted to create an uncoupled spin system, simply omit the `couplings` attribute.

2.2 Method

A Method object in **mrsimulator** describes an NMR method. For this introduction, you can use the pre-defined method *BlochDecaySpectrum* (page 425). This method simulations the spectrum obtained from the Fourier transform of a Bloch decay signal, i.e., one-pulse and acquire. You can use the code below to create the Method object initialized with attributes whose names should be relatively familiar to an NMR spectroscopist.

```
# Import the BlochDecaySpectrum class
from mrsimulator.method.lib import BlochDecaySpectrum
from mrsimulator.method import SpectralDimension

# Create a BlochDecaySpectrum object
method = BlochDecaySpectrum(
    channels=["13C"],
```

(continues on next page)

(continued from previous page)

```
    magnetic_flux_density=9.4,  # in T
    rotor_angle=54.735 * 3.14159 / 180,  # in rad (magic angle)
    rotor_frequency=3000,  # in Hz
    spectral_dimensions=[
        SpectralDimension(
            count=2048,
            spectral_width=80e3,  # in Hz
            reference_offset=6e3,  # in Hz
            label=r"$^{13}$C resonances",
        )
    ],
)
```

The `channel` attribute holds a list of isotope strings. In the BlochDecaySpectrum method, however, only the first isotope in the list, i.e., ^{13}C, is used to simulate the spectrum. The BlochDecaySpectrum method has one spectral dimension. In this example, that spectral dimension has 2048 points, spanning 80 kHz with a reference offset of 6 kHz.

Next, you will bring the SpinSystem and Method objects together and create a Simulator object that will simulate the spectrum.

2.3 Simulator

At the heart of **mrsimulator** is the Simulator object, which calculates the NMR spectrum. **Mrsimulator** performs all calculations in the frequency domain, and all resonance frequencies are calculated in the weakly-coupled (Zeeman) basis for the spin system.

In the code below, you create a Simulator object, initialized with your previously defined spin system and method, and then call *run()* (page 385) on your Simulator object.

```
# Import the Simulator class
from mrsimulator import Simulator

# Create a Simulator object
sim = Simulator(spin_systems=[spin_system], methods=[method])
sim.run()
```

The simulated spectrum is stored as a CSDM object in the Method object at `sim.methods[0].simulation`. To match an experimental MAS spectrum, however, you still need to add some line broadening to the simulated spectrum. For this, you can use the *Signal Processor* (page 133) object described in the next section.

2.4 SignalProcessor

A *Signal Processor* (page 489) object holds a list of operations applied sequentially to a dataset. For a comprehensive list of operations and further details on using the SignalProcessor object, consult the *Signal Processor* (page 133) documentation.

Use the code below to create a SignalProcessor object that performs a convolution of the simulated spectrum with a Lorentzian distribution having a full-width-half-maximum of 200 Hz. This is done with three operations: the first operation applies an inverse fast Fourier transform of the spectrum into the time domain, the second operation applies a time-domain apodization with an exponential decay, and the third operation applies a fast Fourier transform back into the frequency domain.

```
from mrsimulator import signal_processor as sp

# Create the SignalProcessor object
processor = sp.SignalProcessor(
    operations=[
        sp.IFFT(),
        sp.apodization.Exponential(FWHM="200 Hz"),
        sp.FFT(),
    ]
)

# Apply the processor to the simulation dataset
processed_simulation = processor.apply_operations(dataset=sim.methods[0].simulation)
```

2.5 PyPlot

You can use Matplotlib's PyPlot module to plot your simulations. To aid in plotting CSDM objects with PyPlot, csdmpy provides a custom CSDM dataset plot axes. To use it, simply pass `projection="csdm"` when instantiating an Axes instance. Below is code using the PyPlot module which will generate a plot and a pdf file of the simulated spectrum:

Note: To use the custom CSDM axes with `projection="csdm"`, the csdmpy library needs imported.

```
import matplotlib.pyplot as plt

plt.rcParams['pdf.fonttype'] = 42   # For using plots in Illustrator
plt.figure(figsize=(5, 3))  # set the figure size
ax = plt.subplot(projection="csdm")
ax.plot(processed_simulation.real)
ax.invert_xaxis()  # reverse x-axis
plt.tight_layout()
plt.savefig("spectrum.pdf")
plt.show()
```

The `plt.savefig("spectrum.pdf")` line creates a pdf file that can be edited in a vector graphics editor such as Adobe Illustrator. We encourage you to work through the PyPlot basic usage tutorial to understand its methods and learn how to further customize your plots.

2.6 CSDM

Mrsimulator is designed to be part of a larger data workflow involving other software packages. For this larger context, **mrsimulator** uses the Core Scientific Dataset Model (CSDM) for importing and exporting your datasets. CSDM is a lightweight, portable, human-readable, and versatile standard for intra- and interdisciplinary exchange of scientific datasets. The model supports multi-dimensional datasets with a multi-component dependent variable discretely sampled at unique points in a multi-dimensional independent variable space. It can also hold correlated datasets assuming the different physical quantities (dependent variables) are sampled on the same orthogonal grid of independent variables. It can even handle datasets with non-uniform sampling on a grid. The CSDM can also serve as a re-usable building block in developing more sophisticated portable scientific dataset file standards.

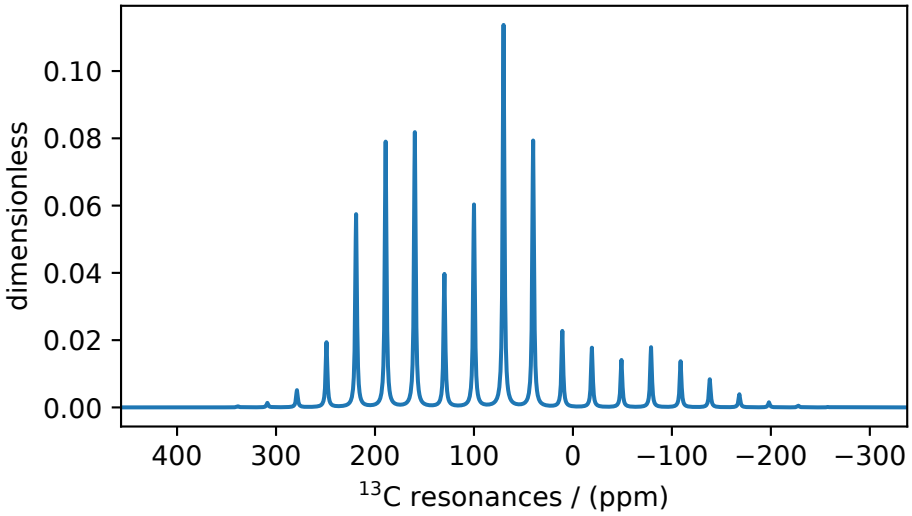


Figure 2.1: A simulated ^{13}C MAS spectrum.

Mrsimulator also uses CSDM internally as its object model for simulated and experimental datasets. Any CSDM object in **mrsimulator** can be serialized as a JavaScript Object Notation (JSON) file using its `save()` method. For example, the simulation after the signal processing step above is saved as a csdf file as shown below.

```
processed_simulation.save("processed_simulation.csdf")
```

For more information on the CSDM file formats, see the csdmpy documentation.

CHAPTER

THREE

ISOTOPOMERS EXAMPLE

Here you will work through an example that should be familiar to nearly all practitioners of NMR spectroscopy, i.e., the simulation of the ^{1}H and ^{13}C liquid-state NMR spectra of ethanol with its various isotopomers. The ^{1}H spectrum will include the characteristic ^{13}C satellite peaks which arise from couplings between ^{1}H and ^{13}C in low-abundance isotopomers.

3.1 Spin Systems

The molecules in a sample of ethanol, CH_3CH_2OH, can be formed with any of the naturally abundant isotopes of hydrogen, carbon, and oxygen present. Of the most abundant isotopes, ^{1}H (99.985%), ^{12}C (98.93%), and ^{16}O (99.762%), only ^{1}H is NMR active. The most abundant NMR active isotopes of carbon and oxygen are ^{13}C (1.11%) and ^{17}O (0.038%). Additionally, the ^{2}H (0.015%) isotope will be present. For our purposes, we will ignore the effects of the lower abundant ^{17}O and ^{2}H isotopes, and focus solely on the spectra of the isotopomers formed from ^{1}H, ^{12}C , and ^{13}C. This leaves us with the three most abundant isotopomers of ethanol shown below.

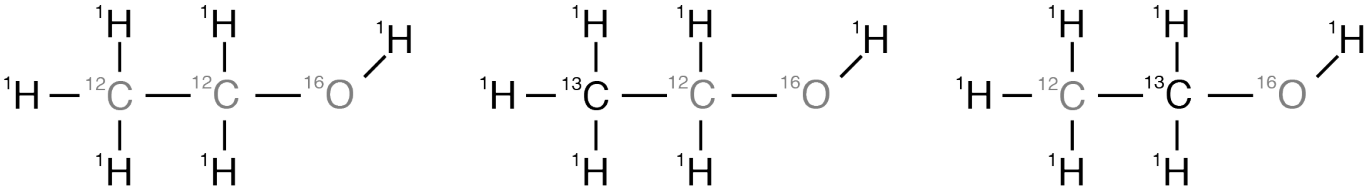


Figure 3.1: The three most abundant isotopomers of ethanol.

The most abundant isotopomer, on the left, has a probability of $(0.99985)^6 \times (0.9893)^2 \times (0.99762) = 0.9625$, while the last two have identical probabilities of $(0.99985)^6 \times (0.0111)(0.9893) \times (0.99762) = 0.0108$

Before you can construct spin systems for each of these isotopomers, you'll need to create sites for each of the three magnetically inequivalent ^{1}H and two magnetically inequivalent ^{13}C sites, as shown in the code below.

```
from mrsimulator import Simulator, Site, SpinSystem, Coupling

# All shifts in ppm
# methyl proton site
H_CH3 = Site(isotope="1H", isotropic_chemical_shift=1.226)
# methylene proton site
H_CH2 = Site(isotope="1H", isotropic_chemical_shift=2.61)
# hydroxyl proton site
H_OH = Site(isotope="1H", isotropic_chemical_shift=3.687)
```

(continues on next page)

(continued from previous page)

```
# methyl carbon site
C_CH3 = Site(isotope="13C", isotropic_chemical_shift=18)
# methylene carbon site
C_CH2 = Site(isotope="13C", isotropic_chemical_shift=58)
```

These sites will be used, along with *Coupling* (page 53) objects described below, to create each of the isotopomers.

3.1.1 Isotopomer 1

To create the SpinSystem object for the most abundant isotopomer, start by creating a list of sites present in this isotopomer.

```
# Put sites into list
iso1_sites = [H_CH3, H_CH3, H_CH3, H_CH2, H_CH2, H_OH]
```

Each site in the isotopomer is identified by its index in the `iso1_sites` ordered list, which are numbered from 0 to 5. Remember that the two Sites involved in a Coupling are identified by their indexes in this list.

Next, create the *Coupling* (page 53) objects between the sites and place the Coupling objects in a list.

```
# All isotropic_j coupling in Hz
HH_coupling_1 = Coupling(site_index=[0, 3], isotropic_j=7)
HH_coupling_2 = Coupling(site_index=[0, 4], isotropic_j=7)
HH_coupling_3 = Coupling(site_index=[1, 3], isotropic_j=7)
HH_coupling_4 = Coupling(site_index=[1, 4], isotropic_j=7)
HH_coupling_5 = Coupling(site_index=[2, 3], isotropic_j=7)
HH_coupling_6 = Coupling(site_index=[2, 4], isotropic_j=7)

# Put couplings into list
iso1_couplings = [
    HH_coupling_1,
    HH_coupling_2,
    HH_coupling_3,
    HH_coupling_4,
    HH_coupling_5,
    HH_coupling_6,
]
```

Finally, create the SpinSystem object for this isotopomer along with its abundance.

```
isotopomer1 = SpinSystem(sites=iso1_sites, couplings=iso1_couplings, abundance=96.25)
```

3.1.2 Isotopomer 2

Replacing the methyl carbon with a ^{13}C isotope gives the second isotopomer. To create its SpinSystem object, follow the code below, where (1) you create the list of sites to include the `C_CH3` site, (2) you create three Coupling objects for its J coupling to the three attached protons, (3) you create the list of couplings, and, finally, (4) you create the SpinSystem object for the isotopomer using the lists of sites and couplings along with the isotopomer's abundance of 1.08%.

```
# Put sites into list
iso2_sites = [H_CH3, H_CH3, H_CH3, H_CH2, H_CH2, H_OH, C_CH3]

# Define methyl 13C - 1H couplings
CH3_coupling_1 = Coupling(site_index=[0, 6], isotropic_j=125)
CH3_coupling_2 = Coupling(site_index=[1, 6], isotropic_j=125)
CH3_coupling_3 = Coupling(site_index=[2, 6], isotropic_j=125)

# Add new couplings to existing 1H - 1H couplings
iso2_couplings = iso1_couplings + [CH3_coupling_1, CH3_coupling_2, CH3_coupling_3]

isotopomer2 = SpinSystem(sites=iso2_sites, couplings=iso2_couplings, abundance=1.08)
```

3.1.3 Isotopomer 3

Lastly, build the sites, couplings, and spin system for the isotopomer with the methylene carbon replaced with a ^{13}C isotope.

```
# Put sites into list
iso3_sites = [H_CH3, H_CH3, H_CH3, H_CH2, H_CH2, H_OH, C_CH2]

# Define methylene 13C - 1H couplings
CH2_coupling_1 = Coupling(site_index=[3, 6], isotropic_j=141)
CH2_coupling_2 = Coupling(site_index=[4, 6], isotropic_j=141)

# Add new couplings to existing 1H - 1H couplings
iso3_couplings = iso1_couplings + [CH2_coupling_1, CH2_coupling_2]

isotopomer3 = SpinSystem(sites=iso3_sites, couplings=iso3_couplings, abundance=1.08)
```

3.2 Methods

For this example, create two `BlochDecaySpectrum` methods for ^{1}H and ^{13}C. Recall that this method simulates the spectrum for the first isotope in the `channels` attribute list.

```
from mrsimulator.method.lib import BlochDecaySpectrum
from mrsimulator.method import SpectralDimension

method_H = BlochDecaySpectrum(
    channels=["1H"],
    magnetic_flux_density=9.4,  # in T
    spectral_dimensions=[
        SpectralDimension(
            count=16000,
            spectral_width=1.5e3,  # in Hz
            reference_offset=950,  # in Hz
            label="$^{1}$H frequency",
        )
    ],
```

(continues on next page)

(continued from previous page)

```
)

method_C = BlochDecaySpectrum(
    channels=["13C"],
    magnetic_flux_density=9.4,  # in T
    spectral_dimensions=[
        SpectralDimension(
            count=32000,
            spectral_width=8e3,  # in Hz
            reference_offset=4e3,  # in Hz
            label="$^{13}$C frequency",
        )
    ],
)
```

3.3 Simulations

Next, create an instance of the simulator object with the list of your three spin systems and the list of your two methods, and run the simulations.

```
sim = Simulator(
    spin_systems = [isotopomer1, isotopomer2, isotopomer3],
    methods = [method_H, method_C]
)
sim.run()
```

Note that the Simulator object runs six simulations in this example, i.e., three `method_H` simulations are run for each of the three isotopomers before being added together to create the final `method_H` simulation. Similarly three simulations are run to create the final `method_C` simulation.

3.4 Signal Processors

Before plotting the spectra, add some line broadening to the resonances. For this, create SignalProcessor objects initialized with a list of operations that give a convolution with a Lorentzian line shape. For the ^{1}H spectrum, create a SignalProcessor object with an exponential apodization that gives a full-width-half-maximum (FWHM) of 1 Hz, while for the ^{13}C spectrum, create an otherwise identical SignalProcessor object that gives an FWHM of 20 Hz.

```
from mrsimulator import signal_processor as sp

# Get the simulation datasets
H_spectrum = sim.methods[0].simulation
C_spectrum = sim.methods[1].simulation

# Create the signal processors
processor_1H = sp.SignalProcessor(
    operations=[
        sp.IFFT(),
        sp.apodization.Exponential(FWHM="1 Hz"),
```

(continues on next page)

(continued from previous page)

```
        sp.FFT(),
    ]
)

processor_13C = sp.SignalProcessor(
    operations=[
        sp.IFFT(),
        sp.apodization.Exponential(FWHM="20 Hz"),
        sp.FFT(),
    ]
)

# apply the signal processors
processed_H_spectrum=processor_1H.apply_operations(dataset=H_spectrum)
processed_C_spectrum=processor_13C.apply_operations(dataset=C_spectrum)
```

3.4.1 Plotting the Dataset

Finally, after applying the convolution with a Lorentzian line shape, you can plot the two spectra using the code below. Additionally, you can save the plot as a pdf file in this example.

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(
    nrows=1, ncols=2, subplot_kw={"projection": "csdm"}, figsize=[9, 4]
)

ax[0].plot(processed_H_spectrum.real)
ax[0].invert_xaxis()
ax[0].set_title("$^1$H")

ax[1].plot(processed_C_spectrum.real)
ax[1].invert_xaxis()
ax[1].set_title("$^{13}$C")

plt.tight_layout()
plt.savefig("spectra.pdf")
plt.show()
```

3.4.2 Saving your Work

Saving the Spectra

You can save the spectra in csdf format using the code below.

```
processed_H_spectrum.save("processed_H_spectrum.csdf")
processed_C_spectrum.save("processed_C_spectrum.csdf")
```

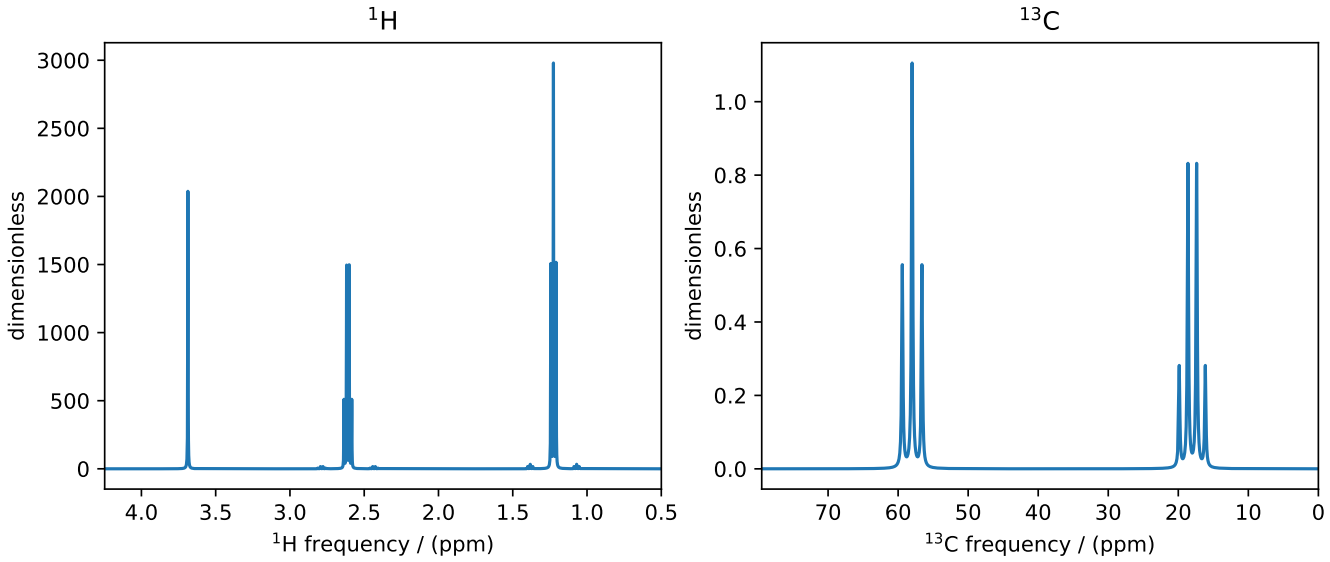


Figure 3.2: ^{1}H and ^{13}C spectrum of ethanol. Note, the ^{13}C satellites seen on either side of the peaks near 1.2 ppm and 2.6 ppm in the ^{1}H spectrum.

Saving the SpinSystems

If you want to save the spin systems for use in a different project, you can ask the Simulator object to export the list of SpinSystem objects to a JSON file with the code below.

```
sim.export_spin_systems("ethanol.mrsys")
```

The file `ethanol.mrsys` holds a JSON representation of the SpinSystem objects. We encourage the convention of using `.mrsys` extension for this JSON file.

The list of SpinSystem objects can be reloaded back into a Simulator object by calling `load_spin_systems()` with the file name of the saved SpinSystem objects, as shown below.

```
new_sim = Simulator()
new_sim.load_spin_systems("ethanol.mrsys")
```

Saving the Methods

Similarly, if you want to save the methods for use in a another project, you can ask the Simulator object to export the list of Method objects to a JSON file.

```
sim.export_methods("H1C13Methods.mrmtd")
```

As before, the file `H1C13Methods.mrmtd` holds a JSON representation of the method objects. We encourage the convention of using `.mrmtd` extension for this JSON file.

The list of Method objects can also be reloaded back into a Simulator object by calling `load_methods()` with the file name of the saved Method objects, as shown below.

```
new_sim = Simulator()
new_sim.load_methods("H1C13Methods.mrmtd")
```

Saving the full Simulation

The Simulation and SignalProcessor objects can also be serialized into JSON files. At some point, however, saving the Python script or Jupyter notebook with your code will be just as convenient. Nonetheless, you can find additional details on JSON serialization of **mrsimulator** objects in the *mrsimulator I/O* (page 137) section.

CHAPTER

FOUR

LEAST-SQUARES FITTING EXAMPLE

Mrsimulator can interact with various Python data science packages. One such package, LMFIT, can be used to perform non-linear least-squares analysis of experimental NMR spectra.

Here, we illustrate the use of the **mrsimulator** objects to

- import and prepare an experimental dataset for the least-squares analysis,
- create a fitting model using Simulator and SignalProcessor objects,
- use the fitting model to perform a least-squares analysis,
- extract the model parameters with uncertainties, and
- plot the experimental spectrum along with the best-fit simulation and residuals.

4.1 Import Experimental Dataset

In this example, you will apply the least-squares fitting procedure to a ^{27}Al magic-angle spinning spectrum of $Al(acac)_3$ measured with whole echo acquisition.

You will begin by importing an experimental dataset measured on a 9.4 T Bruker AVANCE III HD NMR spectrometer into the script. Bruker datasets are saved in folders with a number as the folder name. In this case, that folder has been transferred from the spectrometer and renamed "Al_acac".

4.1.1 Download experimental dataset

For our purposes, the folder was also compressed into a zip archive and uploaded to an internet-accessible server. You can use the code block below to download the zip archive from the server and unzip it into the originally named folder.

```
import requests
import zipfile
from io import BytesIO

file_ = "https://ssnmr.org/sites/default/files/mrsimulator/Al_acac3_0.zip"
request = requests.get(file_)
z = zipfile.ZipFile(BytesIO(request.content))
z.extractall("Al_acac")
```

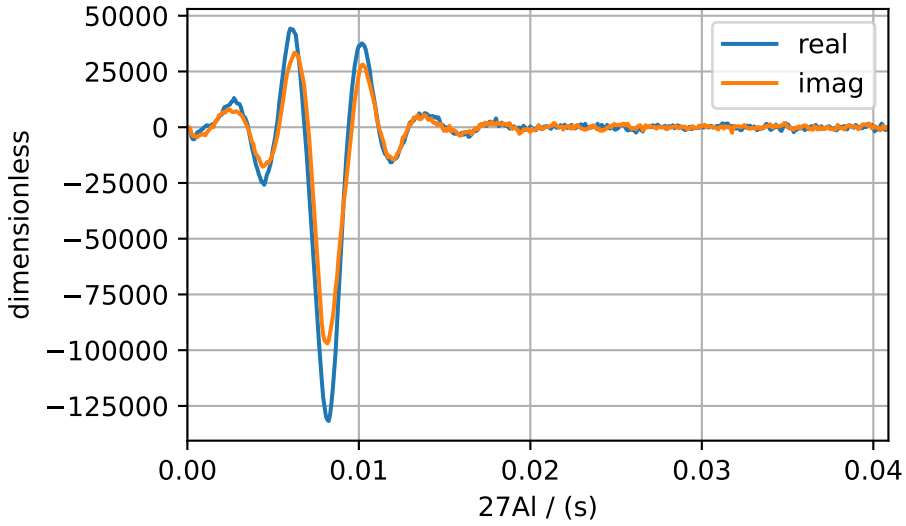
4.1.2 Convert experimental dataset to CSDM

Now that the Bruker dataset folder is accessible to your Python code, you can use the Python package nmrglue to convert the dataset into a CSDM object.

```
import nmrglue as ng

# initialize nmrglue converter object
converter = ng.convert.converter()

# read in the bruker dataset file
dic, data = ng.bruker.read("Al_acac")
converter.from_bruker(dic, data)

# convert to CSDM format
csdm_ds = converter.to_csdm()
```

With the dataset converted into a CSDM object, plot the dataset to make sure that you imported it correctly.

```
import matplotlib.pyplot as plt

plt.figure(figsize=(5, 3))  # set the figure size
ax = plt.subplot(projection="csdm")
ax.plot(csdm_ds.real, label="real")
ax.plot(csdm_ds.imag, label="imag")
plt.tight_layout()
plt.grid()
plt.legend()
plt.show()
```



This is the raw time-domain dataset, acquired using whole-echo acquisition. The blue and orange lines are the real and imaginary parts of the complex time-domain signal. If you're working with your own experimental dataset and have already processed it into the frequency domain, then you can skip the next few steps and proceed to the *Measure Noise* (page 40) section.

4.2 Process Experimental Dataset

Proceeding from here, you'll need to transform this dataset into the frequency domain for the least-squares analysis. Before applying the Fourier transform, however, two things need to be adjusted.

First, you need to adjust the `coordinates_offset` to place the time origin at the top of the echo. You can find this time offset among the pulse sequence parameters. If you acquired the signal with a simple Hahn-echo sequence, i.e., $\pi/2 - \tau - \pi - t$, then the `coordinates_offset` should be the time between the centers of the two pulses. However, there are often some additional receiver delays before the signal acquisition begins, and those times need to be subtracted from the interpulse spacing. In this measurement, we determined the echo top position to be 0.00816 s. The `coordinates_offset`, the time associated with the first point in the signal, will need to be set to –0.00816 s. When correctly set, the time origin should coincide with the maximum magnitude of the complex signal.

Second, you need to phase correct the time domain so that the maximum echo amplitude is in the real part of the signal. For this operation, you can use numpy abs () to take the absolute value of each complex signal amplitude, and numpy argmax () to find the time index where the absolute value of the signal is at a maximum. Then use the signal phase at that time index to place the maximum amplitude into the real part of the time domain signal.

Both these steps are performed by the code below.

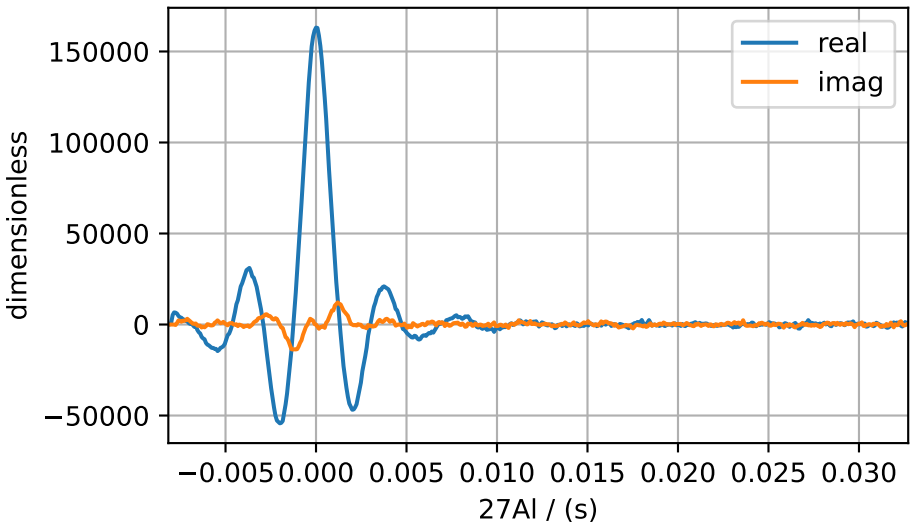
```
import numpy as np

# set time origin to echo top
csdm_ds.dimensions[0].coordinates_offset = "-0.00816 s"

# Phase echo top, putting maximum amplitude into real part
index = np.argmax(np.abs(csdm_ds.dependent_variables[0].components[0]))
angle = np.angle(csdm_ds.dependent_variables[0].components[0][index])
phased_ds = csdm_ds * np.exp(-1j * angle)

plt.figure(figsize=(5, 3))  # set the figure size
ax = plt.subplot(projection="csdm")
ax.plot(phased_ds.real, label="real")
ax.plot(phased_ds.imag, label="imag")
plt.tight_layout()
plt.grid()
plt.legend()
plt.show()
```

Here, you see that the echo top has been phased so that the maximum amplitude is in the real (blue) part and that the echo top occurs at the time origin. Notice that the echo has a slight asymmetry about the time origin after it has been phased. The first half of the echo has a slightly stronger amplitude than the last half. This asymmetry is due to an additional dephasing caused by homonuclear dipolar couplings among the ^{27}Al nuclei. It may have been possible to remove or minimize the effects of these dipolar couplings using a higher MAS rate. Nonetheless, you can still proceed in this analysis and, as you will see later, can model this additional decay with an ad-hoc Gaussian convolution of the spectrum.

Next, create a SignalProcessor object to apply the Fourier transform operation to the CSDM object `exp_spectrum`. Note that with a correctly set time origin, the `FFT()` operation automatically applies the appropriate first-order phase correction to the spectrum after performing the fast Fourier transform. After performing the Fourier transform, convert the coordinate units of the CSDM dimension from frequency to a frequency ratio using the to() method of the Dimension object.

```
from mrsimulator import signal_processor as sp
```

(continues on next page)



(continued from previous page)

```
ft = sp.SignalProcessor(operations=[sp.FFT()])
exp_spectrum = ft.apply_operations(dataset=phased_ds)
exp_spectrum.dimensions[0].to("ppm", "nmr_frequency_ratio")

fig, ax = plt.subplots(1, 2, figsize=(9, 3.5), subplot_kw={"projection": "csdm"})
ax[0].plot(exp_spectrum.real)
ax[0].plot(exp_spectrum.imag)
ax[0].set_title("Full Spectrum")
ax[0].grid()
ax[1].plot(exp_spectrum.real, label="real")
ax[1].plot(exp_spectrum.imag, label="imag")
ax[1].set_title("Zoomed Spectrum")
ax[1].set_xlim(-15, 15)
ax[1].grid()
plt.tight_layout()
plt.legend()
plt.show()
```

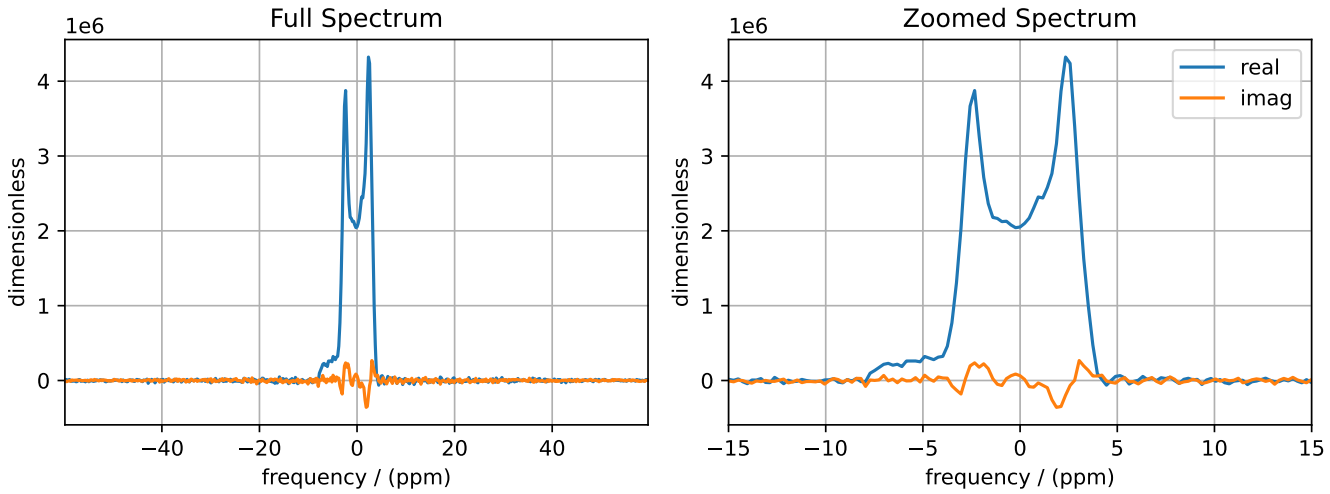
4.3 Measure Noise

Now that you have an adequately phased frequency domain dataset, you'll need to take the real part of the spectrum for the rest of the analysis, i.e., remove the imaginary part.

The least-squares analysis also needs the standard deviation of the noise in the spectrum. We can obtain that from the spectrum regions below -20 ppm or above 20 ppm, where there is no signal amplitude. To accomplish this, you can use numpy where(). It evaluates a condition for each item in the list and returns the indexes for those items where the condition is true. With the indexes returned by where(), you can calculate the standard deviation of the noise region with numpy std().

```
# Use only the real part of the spectrum
exp_spectrum = exp_spectrum.real
```

(continues on next page)



(continued from previous page)

```
# Use region below -20 ppm to calculate the noise standard deviation
loc = np.where(exp_spectrum.dimensions[0].coordinates < -20e-6)
sigma = exp_spectrum[loc].std()
```

You can now move to the next step and create the fitting model.

4.4 Create Fitting Model

To create a proper fitting model, you'll need more information about the nuclei being observed, the material's phase, and some idea about the local structure around the atoms holding the observed nuclei. In this example, you know that you are working with ^{27}Al, a quadrupolar nucleus with a half-integer spin of 5/2, and that the material, $Al(acac)_3$, is a solid polycrystalline sample. The symmetry of the first-coordination sphere around aluminum is likely low enough to generate a large electric field gradient, and hence a sizeable quadrupolar coupling constant for ^{27}Al. These details are usually sorted out before the NMR measurement and used to choose the appropriate NMR methods for the sample. In this example, the measurement was performed under magic-angle spinning at a rotation rate of 12.5 kHz. Due to the expected large quadrupolar coupling, relatively low power rf pulses were used to excite only the central $m = \frac{1}{2} \rightarrow -\frac{1}{2}$ transition of ^{27}Al. The central transition is much narrower and more easily detected than the other transitions. Armed with this understanding of the sample and method, you can proceed to create the fitting model.

Start by creating the Method object to model the experimental method used to acquire the spectrum. Choose the *BlochDecayCTSpectrum()* (page 431) method since the measurement is designed to excite only the central transition of the ^{27}Al nuclei. From the CSDM object holding the experimental spectrum, i.e., `exp_spectrum`, you can extract the relevant parameters for the `spectral_dimensions` attribute of the BlochDecayCTSpectrum method using the fitting utility function *get_spectral_dimensions()* (page 487). The experimental measurement parameters associated with the method attributes `magnetic_flux_density` and `rotor_frequency` are also used in creating this BlochDecayCT-Spectrum method. Finally, every Method object has the `experiment` attribute used to hold the experimental spectrum that is to be modeled with the Method object.

```
from mrsimulator.method.lib import BlochDecayCTSpectrum
from mrsimulator.utils import get_spectral_dimensions
```

(continues on next page)

(continued from previous page)

```
spectral_dims = get_spectral_dimensions(exp_spectrum)
MAS = BlochDecayCTSpectrum(
    channels=["27Al"],
    magnetic_flux_density=9.4,  # in T
    rotor_frequency=12500,  # in Hz
    spectral_dimensions=spectral_dims,
    experiment=exp_spectrum,  # add the measurement to the method.
)
```

To build a spin system, you need to know how many magnetically inequivalent nuclei are in the sample and if there are couplings between them. Inspection of the spectrum reveals an anisotropic lineshape that appears to be characteristic of the second-order MAS lineshape of a single site. Knowing this requires that you are already familiar with such lineshapes (**mrsimulator** can help with that!). One might also hypothesize that there may be other sites with lower intensity present in the spectrum, or perhaps the spectrum, as noted earlier, is from a distribution of ^{27}Al sites with very similar efg tensor parameters and dipolar couplings among them. These are all valid hypotheses and could be used to create more elaborate and perhaps even more realistic spin system models. For now, you can choose the simplest spin system model with a single ^{27}Al site, as shown in the code below.

```
from mrsimulator import Site, SpinSystem, Simulator

site = Site(
    isotope="27Al",
    isotropic_chemical_shift=5,
    quadrupolar={"Cq": 3e6, "eta": 0.0},
)
sys = SpinSystem(sites=[site])
```

The tensor parameters above are an educated guess for the tensor parameters, which can be iteratively refined using the code that follows.

Create the simulator object initialized with the SpinSystem and Method objects and run.

```
sim = Simulator(spin_systems=[sys], methods=[MAS])
sim.run()
```

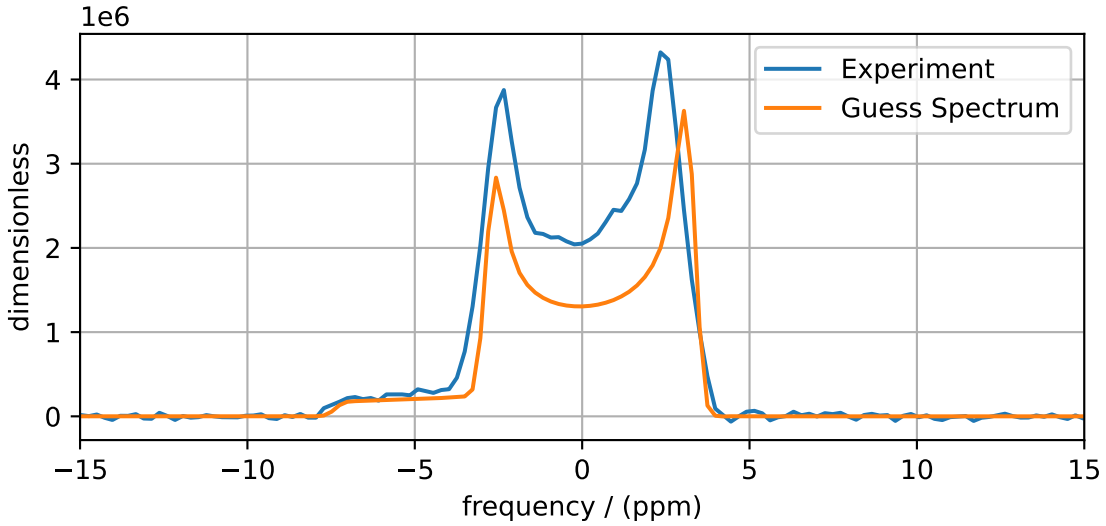
Before comparing the simulation to the experimental spectrum, you need to add the Gaussian line broadening to the simulation. Setup a SignalProcessor object to do a Gaussian lineshape convolution with an FWHM of 50 Hz.

Additionally, you must scale the simulation in intensity to match the experimental spectrum. You may have noticed in earlier plots that the vertical axis of the experimental spectrum plot was on the order of 1e6. Use numpy max() to get the highest amplitude, set that as the factor as a Scale operation in the SignalProcessor.

```
# Post Simulation Processing
# --------------------------
relative_intensity_factor = exp_spectrum.max() / sim.methods[0].simulation.max()
processor = sp.SignalProcessor(
    operations=[
        sp.IFFT(),
        sp.apodization.Gaussian(FWHM="50 Hz"),
        sp.FFT(),
        sp.Scale(factor=relative_intensity_factor)
    ]
)
processed_dataset = processor.apply_operations(dataset=sim.methods[0].simulation).real
```

You now have set up and simulated the first guess in modeling the experimental spectrum. Plot it and see how it compares to the experimental spectrum.

```
# Plot of the guess spectrum
# --------------------------
plt.figure(figsize=(6, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(exp_spectrum, label="Experiment")
ax.plot(processed_dataset, label="Guess Spectrum")
ax.set_xlim(-15, 15)
plt.legend()
plt.grid()
plt.tight_layout()
plt.show()
```



The fit parameters are the spin system tensor and signal processor parameters. If your initial guess is not so good, you could iteratively change the fit parameters until your simulation is closer to the experimental spectrum. This will ensure faster convergence to the best-fit parameters and could prevent the least-squares analysis from falling into false minima on the chi-squared surface. For this example, however, the above initial guess should be good enough

4.5 Perform Least-Squares Analysis

Up to this point in the discussion, you've done little more than what you've learned earlier in setting up a simulation with mrsimulator. Except now, you're ready to leverage the power of LMFIT to obtain the best-fit parameters.

4.5.1 Define the fit parameters

Begin by using an **mrsimulator** utility function *make_LMFIT_params()* (page 497) to extract a list of LMFIT parameters from the Simulator and SignalProcessor objects.

```
from mrsimulator.utils import spectral_fitting as sf
fit_parameters = sf.make_LMFIT_params(sim, processor)
print(fit_parameters.pretty_print(columns=["value", "min", "max", "vary", "expr"]))
```

```
Name                                    Value      Min      Max     Vary     Expr
SP_0_operation_1_Gaussian_FWHM             50     -inf      inf     True     None
SP_0_operation_3_Scale_factor         4.322e+06     -inf      inf     True     None
sys_0_abundance                             100        0      100    False      100
sys_0_site_0_isotropic_chemical_shift         5     -inf      inf     True     None
sys_0_site_0_quadrupolar_Cq                3e+06     -inf      inf     True     None
sys_0_site_0_quadrupolar_eta                  0        0        1     True     None
None
```

The output of the `print()` statement, shown above, gives the table of the LMFIT parameters created by *make_LMFIT_params()* (page 497). The returned `fit_parameters` is a dictionary with each fit parameter object identified by a string. LMFIT does not allow special characters such as `[`, `]` or `.` in the parameter string identifiers. Therefore, when the *make_LMFIT_params()* (page 497) function creates the LMFIT parameters dictionary, it flattens the variable namespace into a string with these special characters replaced by a `_`. For example,

"sim.spin_systems[0].sites[1].quadrupolar.Cq" → "sys_0_site_1_quadrupolar_Cq"

or

"sp[0].operation[3].factor" → "SP_0_operation_3_Scale_factor".

Using these parameter string names, you can access and change any of its LMFIT parameter attributes, i.e., `value`, `min`, `max`, `vary`, `expr`. For example, using the code below, you can set the quadrupolar asymmetry parameter value to be zero and request that it be held constant during the fit.

```
fit_parameters["sys_0_site_0_quadrupolar_eta"].value = 0
fit_parameters["sys_0_site_0_quadrupolar_eta"].vary = False
```

Warning: First-principles DFT calculations based on structural hypotheses can sometimes help determine the initial guess for some parameters; however, they are rarely accurate enough, even when using the correct structure, to be used as fixed parameters in a least-squares analysis of an experimental spectrum.

4.5.2 Define and minimize the chi-squared function

To perform a least-squares analysis, LMFIT needs a chi-squared function. LMFIT expects this function to return a list of residuals (difference between model and data) divided by the experimental noise standard deviation. Mrsimulator comes with a pre-built chi-squared function *LMFIT_min_function()* (page 497) which takes the Simulator, SignalProcessor, and the experimental noise standard deviation as function arguments.

4.5.3 Perform the chi-squared minimization

The least-squares analysis is performed by creating an LMFIT Minimizer object initialized with a chi-squared function and the fit parameters (`fit_parameters`). Any additional objects needed to evaluate the chi-squared function are placed in `fcn_args`. For *LMFIT_min_function()* (page 497), `fcn_args` needs to hold the Simulator, SignalProcessor, and the experimental noise standard deviation.

After the minimize() function of the Minimizer object exits, the parameters in the Simulator and SignalProcessor are updated with the best-fit parameters and the results of the least-squares analysis is returned as an MinimizerResult object containing the optimized parameters and several goodness-of-fit statistics.

Use the code below to create and initialize the `Minimizer` object, run the minimization, and print the MinimizerResult.

```
from lmfit import Minimizer
# Optimize fitting by pre-computing transition pathways
opt = sim.optimize()

minner = Minimizer(
    sf.LMFIT_min_function,
    fit_parameters,
    fcn_args=(sim, processor, sigma),
    fcn_kws={"opt": opt}
)
result = minner.minimize()
result
```

Fit Statistics

fitting method	leastsq
# function evals	41
# data points	512
# variables	4
chi-square	7354.91136
reduced chi-square	14.4781720
Akaike info crit.	1372.37707
Bayesian info crit.	1389.33036

Variables

name	value	standard error	relative error	initial value	min	max	vary	expression
sys_0_site_0_isotropic_chemical_shift	4.50079234	0.00853716	(0.19%)	5.0	-inf	inf	True	
sys_0_site_0_quadrupolar_Cq	2915460.30	2739.93180	(0.09%)	3000000.0	-inf	inf	True	
sys_0_site_0_quadrupolar_eta	0.00000000	0.00000000		0	0.00000000	1.00000000	False	
sys_0_abundance	100.000000	0.00000000	(0.00%)	100	0.00000000	100.000000	False	100
SP_0_operation_1_Gaussian_FWHM	105.693397	1.13517327	(1.07%)	50.0	-inf	inf	True	
SP_0_operation_3_Scale_factor	2180272.12	7279.97788	(0.33%)	4322355.161244868	-inf	inf	True	

Correlations (unreported correlations are < 0.100)

sys_0_site_0_isotropic_chemical_shift	sys_0_site_0_quadrupolar_Cq	0.8123
SP_0_operation_1_Gaussian_FWHM	SP_0_operation_3_Scale_factor	0.3462
sys_0_site_0_isotropic_chemical_shift	SP_0_operation_1_Gaussian_FWHM	0.2189
sys_0_site_0_quadrupolar_Cq	SP_0_operation_1_Gaussian_FWHM	0.2063
sys_0_site_0_quadrupolar_Cq	SP_0_operation_3_Scale_factor	0.1901
sys_0_site_0_isotropic_chemical_shift	SP_0_operation_3_Scale_factor	0.1484

From the printout of the MinimizerResult above, you can find the best-fit parameters and their associated uncertainties

from least-squares analysis.

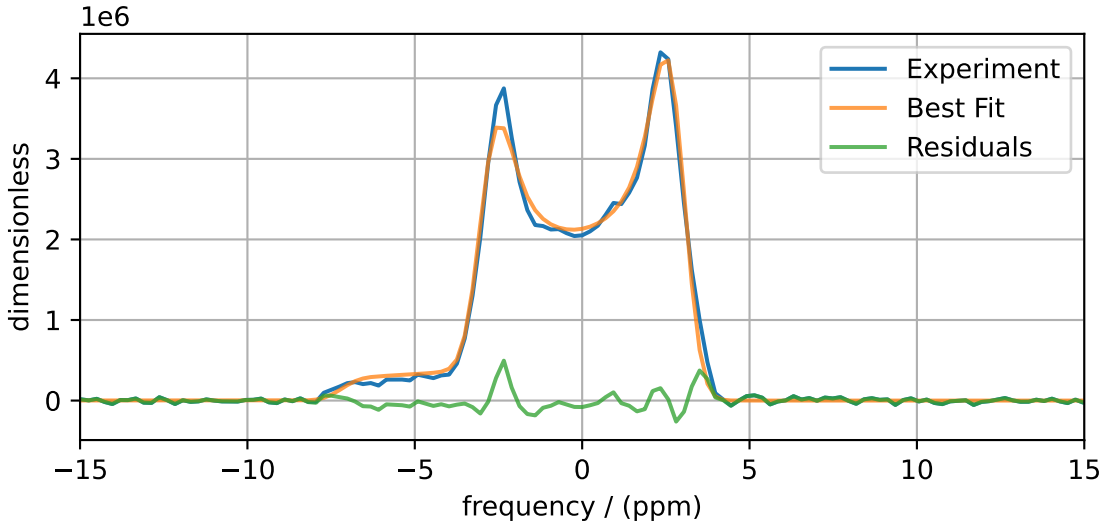
Warning: A word of caution about best-fit parameter uncertainties: If the model is accurate, then you expect the residuals to be pure noise, i.e., a histogram of the residuals should arise from a Gaussian parent distribution with a mean of zero. Therefore, at the very least, you should inspect a plot of the residuals and, even better, check that a histogram of the residuals is consistent with a Gaussian parent distribution.

If this is not true, then the parameter uncertainties from the least-squares analysis will be underestimated. Such discrepancies between the experimental and simulated spectra can often arise from measurement artifacts, e.g., receiver deadtimes, non-uniform excitation, etc. They can also arise from an inadequate model (spin systems and method) for the spectrum.

4.5.4 Compare experimental and best-fit spectra with residuals

You can now plot the experimental and best-fit simulated spectra along with the residuals. Use the **mrsimulator** utility function *bestfit()* (page 498) and *residuals()* (page 498) to extract the best-fit simulation and the residuals as CSDM objects.

```
best_fit = sf.bestfit(sim, processor)[0].real
residuals = sf.residuals(sim, processor)[0].real

# Plot the spectrum
plt.figure(figsize=(6, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(exp_spectrum, label="Experiment")
ax.plot(best_fit, alpha=0.75, label="Best Fit")
ax.plot(residuals, alpha=0.75, label="Residuals")
ax.set_xlim(-15, 15)
plt.legend()
plt.grid()
plt.tight_layout()
plt.show()
```



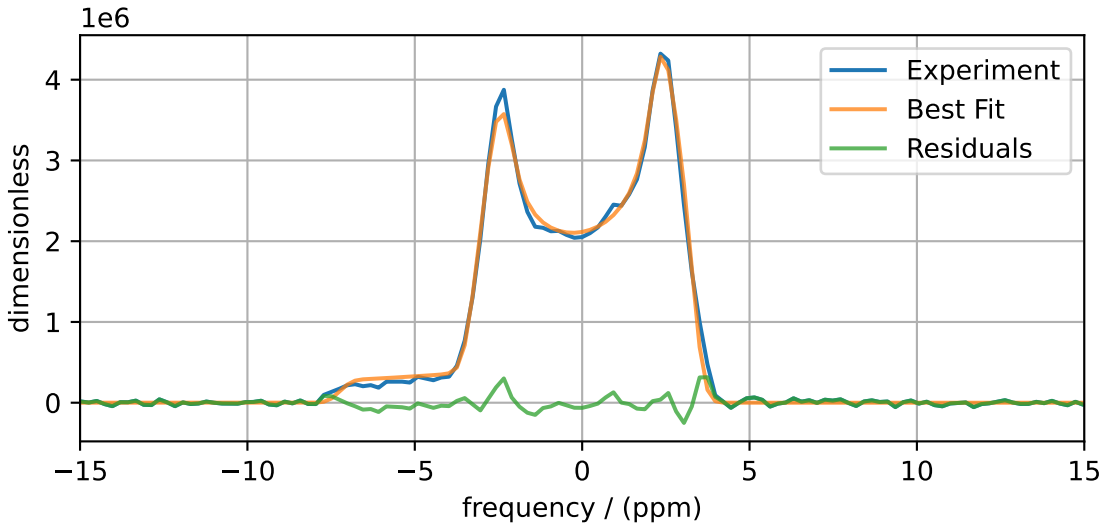
The Minimizer will improve the fit parameters even if the initial parameters are far from the best-fit values. However, if the initial parameters are too far away, the Minimizer may not reach the best-fit parameters in a single run. If you think that may be the case, you can re-extract a new initial guess from the Simulator and SignalProcessor objects using *make_LMFIT_params()* (page 497), create and initialize a new Minimizer object as before, and run again, i.e., rerun the code starting at the beginning of this section. You may see that the fit improves and gives a lower chi-squared value.

In the least-square analysis above, you had locked the quadrupolar asymmetry parameter to a value of zero, which is reasonably close to the true value. At such low values, the quadrupolar asymmetry parameter is correlated to the Gaussian line broadening FWHM in the fit. Set the quadrupolar asymmetry parameter to be a fit parameter, and rerun the analysis.

```
fit_parameters["sys_0_site_0_quadrupolar_eta"].value = 0.05
fit_parameters["sys_0_site_0_quadrupolar_eta"].vary = True

minner = Minimizer(
    sf.LMFIT_min_function,
    fit_parameters,
    fcn_args=(sim, processor, sigma),
    fcn_kws={"opt": opt}
)
result = minner.minimize()

best_fit = sf.bestfit(sim, processor)[0].real
residuals = sf.residuals(sim, processor)[0].real

# Plot the spectrum
plt.figure(figsize=(6, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(exp_spectrum, label="Experiment")
ax.plot(best_fit, alpha=0.75, label="Best Fit")
ax.plot(residuals, alpha=0.75, label="Residuals")
ax.set_xlim(-15, 15)
plt.legend()
plt.grid()
plt.tight_layout()
plt.show()
```

You see a slight improvement in the fit, with the asymmetry parameter increasing from 0 to 0.136, and the Gaussian FWHM decreased from 106 to 63 Hz. The MinimizerResult printout also shows a correlation of -0.74 between these two parameters.

We close this section by noting that a compelling feature of mrsimulator and LMFit is that you can perform a simultaneous spectra fit from different methods for a single set of spin system parameters. Check out all the examples in the *Fitting (Least Squares) Gallery* (page 255), notably the *^{13}C MAS NMR of Glycine (CSA) multi-spectra fit* (page 271) example for fitting one set of spin systems to multiple spectra.



Fit Statistics

fitting method	leastsq
# function evals	75
# data points	512
# variables	5
chi-square	4906.49854
reduced chi-square	9.67751191
Akaike info crit.	1167.11550
Bayesian info crit.	1188.30712

Variables

name	value	standard error	relative error	initial value	min	max	vary	expression
sys_0_site_0_isotropic_chemical_shift	4.57005915	0.00584454	(0.13%)	5.0	-inf	inf	True	
sys_0_site_0_quadrupolar_Cq	2924359.61	1585.45848	(0.05%)	3000000.0	-inf	inf	True	
sys_0_site_0_quadrupolar_eta	0.13633994	0.00217406	(1.59%)	0	0.00000000	1.00000000	True	
sys_0_abundance	100.000000	0.00000000	(0.00%)	100	0.00000000	100.000000	False	100
SP_0_operation_1_Gaussian_FWHM	63.6511732	2.09254749	(3.29%)	50.0	-inf	inf	True	
SP_0_operation_3_Scale_factor	2175046.98	5906.38217	(0.27%)	4322355.161244868	-inf	inf	True	

Correlations (unreported correlations are < 0.100)

sys_0_site_0_isotropic_chemical_shift	sys_0_site_0_quadrupolar_Cq	0.8650
sys_0_site_0_quadrupolar_eta	SP_0_operation_1_Gaussian_FWHM	-0.7394
sys_0_site_0_quadrupolar_Cq	sys_0_site_0_quadrupolar_eta	0.2895
sys_0_site_0_quadrupolar_Cq	SP_0_operation_1_Gaussian_FWHM	-0.2542
sys_0_site_0_isotropic_chemical_shift	SP_0_operation_1_Gaussian_FWHM	-0.1781
sys_0_site_0_isotropic_chemical_shift	sys_0_site_0_quadrupolar_eta	0.1756
SP_0_operation_1_Gaussian_FWHM	SP_0_operation_3_Scale_factor	0.1746
sys_0_site_0_quadrupolar_Cq	SP_0_operation_3_Scale_factor	0.1639
sys_0_site_0_isotropic_chemical_shift	SP_0_operation_3_Scale_factor	0.1303

Part III

User Guide

CHAPTER

FIVE

SPIN SYSTEM

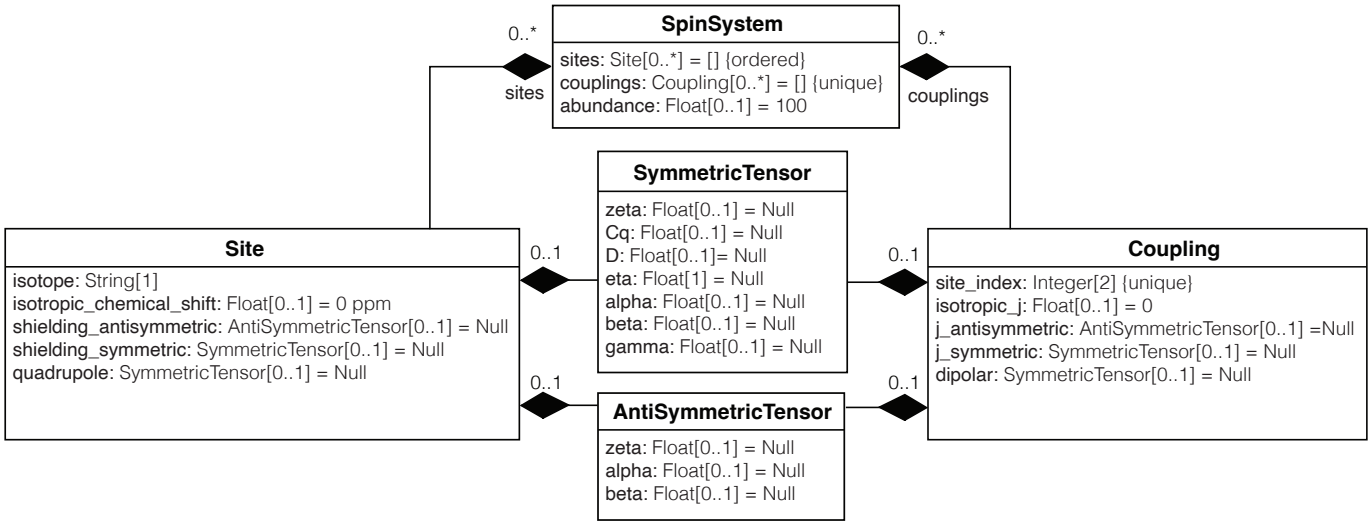
5.1 Overview

At the heart of any **mrsimulator** calculation is the definition of a *SpinSystem* (page 388) object describing the sites and couplings within a spin system. Each *Simulator* (page 379) object holds a list of *SpinSystem* (page 388) objects which are used to calculate frequency contributions.

mrsimulator faces the same limitation faced by all other NMR simulation codes: the computational cost increases exponentially with the number of couplings between sites in a spin system. In liquids, where isotropic molecular motion averages away intermolecular anisotropic couplings, the situation is more tractable as only the intramolecular isotropic J couplings remain.

In solids, where no such isotropic motion exists, the situation is more problematic. In solids that are dilute in NMR-active nuclei is often possible to build a set of SpinSystem objects that can accurately model a spectrum. In solids that are not dilute in NMR-active nuclei, there are still situations where one can build approximately accurate spin systems models. One such case is when the individual anisotropic spin interactions, such as the shielding (shift) anisotropy or the quadrupolar couplings, dominant the spectrum, i.e., they are significantly larger than any dipolar couplings. This can happen for spin 1/2 nuclei in static samples or samples spinning away from the magic-angle. In the case of half-integer quadrupolar nuclei, this can also happen for a central transition spectrum that is significantly broadened by second-order quadrupolar effects. Another case is when an experimental method can successfully decouple the effects of dipolar couplings from the spectrum, rendering it similar to that of a dilute spin system. This can be achieved through rapid sample rotation, a pulse sequence, or some clever combination of the two. In all such cases, any effects of residual dipolar couplings on the spectrum are usually modeled as an ad-hoc Gaussian lineshape convolution.

A *SpinSystem* (page 388) object is organized according to the UML diagram below.



Note: In UML (Unified Modeling Language) diagrams, each class is represented with a box that contains two compartments. The top compartment contains the name of the class, and the bottom compartment contains the attributes of the class. Default attribute values are shown as assignments. A composition is depicted as a binary association decorated with a filled black diamond. Inheritance is shown as a line with a hollow triangle as an arrowhead.

5.2 Site

A site object holds single-site NMR interaction parameters, which include the nuclear shielding and quadrupolar interaction parameters. Consider the example below of a *Site* (page 394) object for a deuterium nucleus created in Python.

```
# Import objects for the Site
from mrsimulator import Site
from mrsimulator.spin_system.tensors import SymmetricTensor

# Create the site object
H2_site = Site(
    isotope="2H",
    isotropic_chemical_shift=4.1,  # in ppm
    shielding_symmetric=SymmetricTensor(
        zeta=12.12,  # in ppm
        eta=0.82,
        alpha=5.45,  # in radians
        beta=4.82,  # in radians
        gamma=0.5,  # in radians
    ),
    quadrupolar=SymmetricTensor(
        Cq=1.47e6,  # in Hz
        eta=0.27,
        alpha=0.212,  # in radians
        beta=1.231,  # in radians
        gamma=3.1415,  # in radians
    ),
)
```

The `isotope` key holds the spin isotope, here given a value of `"2H"`. The `isotropic_chemical_shift` is the isotropic chemical shift of the site isotope, ^{2}H, here given as *4.1 ppm*. We have additionally defined an optional `shielding_symmetric` key, whose value is a second-rank traceless symmetric nuclear shielding tensor represented by a *SymmetricTensor* (page 475) object.

Note: We parameterize a SymmetricTensor using the Haeberlen convention with parameters `zeta` and `eta`, defined as the shielding anisotropy and asymmetry, respectively. The Euler angle orientations, `alpha`, `beta`, and `gamma` are the relative orientation of the nuclear shielding tensor from a common reference frame.

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 the optional `quadrupolar` key. An EFG tensor is a second-rank traceless symmetric tensor, and we describe its coupling to a quadrupolar nucleus with `Cq` and `eta`, i.e., the quadrupolar coupling constant and asymmetry parameter, respectively. Additionally, we use the Euler angle orientations, `alpha`, `beta`, and `gamma`, which are the relative orientation of the EFG tensor from a common reference frame.

See Table 5.2 and Table 5.4 for further information on the *Site* (page 394) and *SymmetricTensor* (page 475) objects and their attributes, respectively.

Also, all objects in **mrsimulator** have the attribute `property_units` which provides the units for all class properties.

```
print(Site().property_units)
# {'isotropic_chemical_shift': 'ppm'}

print(SymmetricTensor().property_units)
# {'zeta': 'ppm', 'Cq': 'Hz', 'D': 'Hz', 'alpha': 'rad', 'beta': 'rad', 'gamma': 'rad'}
```

5.3 Coupling

A coupling object holds two site NMR interaction parameters, which can include the J-coupling and the dipolar coupling interaction parameters. Consider the example below of a *Coupling* (page 399) object between two sites

```
# Import the Coupling object
from mrsimulator import Coupling

coupling = Coupling(
    site_index=[0, 1],
    isotropic_j=15,  # in Hz
    j_symmetric=SymmetricTensor(
        zeta=12.12,  # in Hz
        eta=0.82,
        alpha=2.45,  # in radians
        beta=1.75,  # in radians
        gamma=0.15,  # in radians
    ),
    dipolar=SymmetricTensor(
        D=1.7e3,  # in Hz
        alpha=0.12,  # in radians
        beta=0.231,  # in radians
        gamma=1.1415,  # in radians
    ),
)
```

The `site_index` key holds a list of two integers corresponding to the index of the two coupled sites in the ordered list `sites` within the SpinSystem object. The ordering of the integers in `site_index` is irrelevant.

The value of the `isotropic_j` is the isotropic J-coupling, here given as `15 Hz`. We have additionally defined an optional `j_symmetric` key, whose value holds a SymmetricTensor object representing the traceless 2nd-rank symmetric J-coupling tensor.

Additionally, the dipolar coupling interaction between the coupled nuclei is defined with an optional `dipolar` key. A dipolar tensor is a second-rank traceless symmetric tensor, and we describe the dipolar coupling constant with the parameter `D`. The Euler angle orientations, `alpha`, `beta`, and `gamma` are the relative orientation of the dipolar tensor from a common reference frame.

Note: All frequency contributions from spin-spin couplings are calculated in the weak-coupling limit.

See Table 5.3 and Table 5.4 for further information on the *Site* (page 394) and *SymmetricTensor* (page 475) objects and their attributes, respectively.

5.4 SpinSystem

The *SpinSystem* (page 388) object is a collection of sites and couplings. Below are examples of different spin systems along with discussion on each attribute.

5.4.1 Single Site Spin System

Here we create a relatively unexciting single site proton spin system

```
# Import the SpinSystem object
from mrsimulator import SpinSystem

H1_site = Site(isotope="1H")

single_site_sys = SpinSystem(
    name="1H spin system",
    description="A single site proton spin system",
    sites=[H1_site],
    abundance=80,  # percentage
)
```

We find four keywords at the root level of our SpinSystem object definition: `name`, `description`, `sites`, and `abundance`. The value of the `name` key is the optional name of the spin system. Likewise, 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* (page 394) objects. Here, this list is simply the single object, *H1_site*. The value of the `abundance` key is the abundance of the spin system, here given a value of *80%*. If the abundance key is omitted, the abundance defaults to *100%*.

See Table 5.1 for further description of the *SpinSystem* (page 388) class and its attributes.

5.4.2 Multi Site Spin System

To create a spin system with more than one site, we simply add more site objects to the sites list. Here we create a ^{13}C site and add it along with the previous proton site to a new spin system.

```
# Create the new Site object
C13_site = Site(
    isotope="13C",
    isotropic_chemical_shift=-53.2,  # in ppm
    shielding_symmetric=SymmetricTensor(
        zeta=90.5,  # in ppm
        eta=0.64,
    ),
)

# Create a new SpinSystem object with both Sites
multi_site_sys = SpinSystem(
    name="Multi site spin system",
    description="A spin system with multiple sites",
    sites=[H1_site, C13_site],
    abundance=0.148,  # percentage
)
```

Again we see the optional `name` and `description` attributes. The `sites` attribute is now a list of two *Site* (page 394) objects, the previous ^{1}H site and the new ^{13}C site. We have also set the `abundance` of this spin system to *0.148%*. By leveraging the abundance attribute, multiple spin systems with varying abundances can be simulated together. See our *Isotopomers Example* (page 29) where isotopomers of varying abundance are simulated in tandem.

5.4.3 Coupled Spin System

To create couplings between sites, we simply need to add a list of *Coupling* (page 399) objects to a spin system. Below we create a ^{2}H and ^{13}C site as well as a coupling between them.

```
# Create site objects
H2_site = Site(
    isotope="2H",
    isotropic_chemical_shift=4.1,  # in ppm
    shielding_symmetric=SymmetricTensor(
        zeta=12.12,  # in ppm
        eta=0.82,
        alpha=5.45,  # in radians
        beta=4.82,  # in radians
        gamma=0.5,  # in radians
    ),
    quadrupolar=SymmetricTensor(
        Cq=1.47e6,  # in Hz
        eta=0.27,
        alpha=0.212,  # in radians
        beta=1.231,  # in radians
        gamma=3.1415,  # in radians
    ),
)
C13_site = Site(
    isotope="13C",
    isotropic_chemical_shift=-53.2,  # in ppm
    shielding_symmetric=SymmetricTensor(
        zeta=90.5,  # in ppm
        eta=0.64,
    ),
)

# Create coupling object
H2_C13_coupling = Coupling(
    site_index=[0, 1],
    isotropic_j=15,  # in Hz
    j_symmetric=SymmetricTensor(
        zeta=12.12,  # in Hz
        eta=0.82,
        alpha=2.45,  # in radians
        beta=1.75,  # in radians
        gamma=0.15,  # in radians
    ),
    dipolar=SymmetricTensor(
        D=1.7e3,  # in Hz
        alpha=0.12,  # in radians
```

(continues on next page)

(continued from previous page)

```
        beta=0.231,  # in radians
        gamma=1.1415,  # in radians
    ),
)
```

We now have the site objects and the coupling object to make a coupled spin system. We now construct such a spin system.

```
coupled_spin_system = SpinSystem(sites=[H2_site, C13_site], couplings=[H2_C13_coupling])
```

In contrast to the previous examples, we have omitted the optional `name`, `description`, and `abundance` keywords. The name and description for `coupled_spin_system` will both be `None` and the abundance will be *100%*.

A list of *Coupling* (page 399) objects passed to the `couplings` keywords. The `site_index` attribute of `H2_C13_coupling` correspond to the index of `H2_site` and `C13_site` in the sites list. If we were to add more sites, `site_index` might need to be updated to reflect the index *H2_site`* and `C13_site` in the sites list. Again, our *Isotopomers Example* (page 29) has good usage cases for multiple couplings in a spin system.

5.5 Attribute Summaries

Table 5.1: 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.
`label`	String	An *optional* attribute giving a label to the spin system. Like `name`, it has no effect on a simulation and is purely for readability.
`description`	String	An *optional* attribute describing the spin system. The default value is an empty string.
`sites`	List	An *optional* list of *Site* (page 394) objects. The default value is an empty list.
`couplings`	List	An *optional* list of coupling objects. The default value is an empty list.
`abundance`	String	An *optional* quantity representing the abundance of the spin system. The abundance is given as percentage, for example, `25.4` for 25.4%. This value is useful when multiple spin systems are present. The default value is `100`.

Table 5.2: The attributes of a Site object.

Attribute name	Type	Description
`name`, `label`, and `description`	String	All three are *optional* attributes giving context to a Site object. The default value for all three is an empty string.
`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 ppm, for example, `10` for 10 ppm. The default value is `0`.
`shielding_symmetric`	*SymmetricTensor* (page 475)	An *optional* object describing the second-rank traceless symmetric nuclear shielding tensor following the Haeberlen convention. The default is `None`. See the description for the *SymmetricTensor* (page 475) object.
`quadrupolar`	*SymmetricTensor* (page 475)	An *optional* object describing the second-rank traceless electric quadrupole tensor. The default is `None`. See the description for the *SymmetricTensor* (page 475) object.

Table 5.3: The attributes of a Coupling object.

Attribute name	Type	Description
`site_index`	List of two integers	A *required* list with integers corresponding to the site index of the coupled sites, for example, [0, 1], [2, 1]. The order of the integers is irrelevant.
`isotropic_j`	ScalarQuantity	An *optional* physical quantity describing the isotropic J-coupling in Hz. The default value is `0`.
`j_symmetric`	*SymmetricTensor* (page 475)	An *optional* object describing the second-rank traceless symmetric J-coupling tensor following the Haeberlen convention. The default is `None`. See the description for the *SymmetricTensor* (page 475) object.
`dipolar`	*SymmetricTensor* (page 475)	An *optional* object describing the second-rank traceless dipolar tensor. The default is `None`. See the description for the *SymmetricTensor* (page 475) object.

Table 5.4: The attributes of a SymmetricTensor object.

Attribute name	Type	Description

`zeta`
or
`Cq`
or
`D`
	ScalarQuantity	A *required* quantity.
		Nuclear shielding: The shielding anisotropy, `zeta`, calculated using the Haeberlen convention. The value is a physical quantity given in ppm, for example, `10`
		Electric quadrupole: The quadrupole coupling constant, `Cq`. The value is a physical quantity given in units of Hz, for example, `3.1e6` for 3.1 MHz.
		J-coupling: The J-coupling anisotropy, `zeta`, calculated using the Haeberlen convention. The value is a physical quantity given in Hz, for example, `10` for 10 Hz.
		Dipolar-coupling: The dipolar-coupling constant, `D`. The value is a physical quantity given in Hz, for example, `9e6` for 9 kHz.

`eta`	Float	A *required* asymmetry parameter calculated using the Haeberlen convention, for example, `0.75`. The parameter is set to zero for the dipolar tensor.
`alpha`	ScalarQuantity	An *optional* Euler angle, α. For example, `2.1` for 2.1 radians. The default value is `0`.
`beta`	ScalarQuantity	An *optional* Euler angle, β. For example, `1.5708` for 90 degrees. The default value is `0`.
`gamma`	ScalarQuantity	An *optional* Euler angle, γ. For example, `0.5` for 0.5 radians. The default value is `0`.

CHAPTER

SIX

SPIN SYSTEM DISTRIBUTIONS

6.1 Library distributions

6.1.1 Czjzek distribution

The Czjzek distribution models random variations of second-rank traceless symmetric tensors about zero, i.e., a tensor with zeta of zero. An analytical expression for the Czjzek distribution exists (cite) which follows

$$f(\zeta, \eta, \sigma) = \eta \left(1 - \frac{\eta^2}{9}\right) \frac{\zeta^4}{32\sigma^5\sqrt{2\pi}} \times \exp\left(-\frac{\zeta^2}{8\sigma^2}\left(1 + \frac{\eta^2}{3}\right)\right), \tag{6.1}$$

where ζ and η are the Haberlen components of the tensor and σ is the Czjzek width parameter. See *Czjzek distribution* (page 373) for a further mathematical description of the model.

The remainder of this page quickly describes how to generate Czjzek distributions and generate *SpinSystem* (page 388) objects from these distributions. Also, look at the gallery examples using the Czjzek distribution listed at the bottom of this page.

Creating and sampling a Czjzek distribution

To generate a Czjzek distribution, use the *CzjzekDistribution* (page 493) class as follows.

```
from mrsimulator.models import CzjzekDistribution

cz_model = CzjzekDistribution(sigma=0.8)
```

The **CzjzekDistribution** class accepts the 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 494) method of the instance.

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

In the above example, we draw *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. It is further assumed that the points in `zeta_dist` are in units of `ppm` while `eta_dist` has values since η is dimensionless. 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()
```

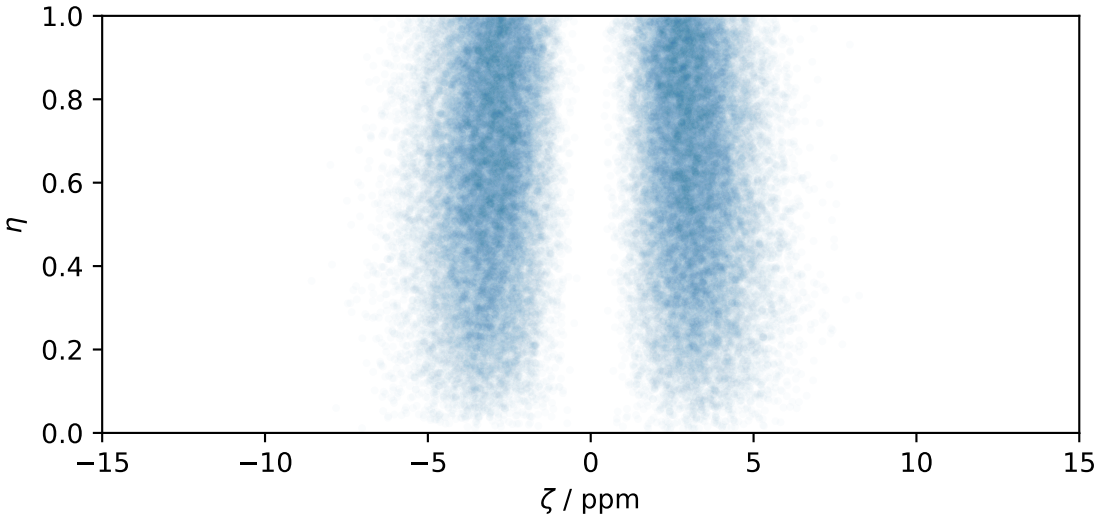


Figure 6.1: Random sampling of Czjzek distribution of shielding tensors.

Creating and sampling a Czjzek distribution in polar coordinates

The *CzjzekDistribution* (page 493) class also supports sampling tensors in polar coordinates. The logic behind transforming from a ζ-η Cartesian grid is further described in mrinversion (cite), and the following definitions are used

$$r_\zeta = |\zeta| \quad \text{and} \quad \theta = \begin{cases} \frac{\pi}{4}\eta & : \zeta \leq 0, \\ \frac{\pi}{2}\left(1 - \frac{\eta}{2}\right) & : \zeta > 0. \end{cases} \tag{6.2}$$

Because Cartesian grids are more manageable in computation, the above polar piece-wise grid is re-express as the x-y Cartesian grid following,

$$x = r_\zeta \cos\theta \quad \text{and} \quad y = r_\zeta \sin\theta. \tag{6.3}$$

Below, we create another instance of the *CzjzekDistribution* (page 493) class with the same value of $sigma = 0.8$, but we now also include the argument `polar=True` which means the *rvs()* (page 494) will sample x and y points.

```
cz_model_polar = CzjzekDistribution(sigma=0.8, polar=True)

# Sample (x, y) points
```

(continues on next page)

(continued from previous page)

```
x_dist, y_dist = cz_model_polar.rvs(size=50000)

# Plot the distribution
plt.figure(figsize=(4, 4))
plt.scatter(x_dist, y_dist, s=4, alpha=0.02)
plt.xlabel("$x$ / ppm")
plt.ylabel("$y$ / ppm")
plt.xlim(0, 8)
plt.ylim(0, 8)
plt.tight_layout()
plt.show()
```

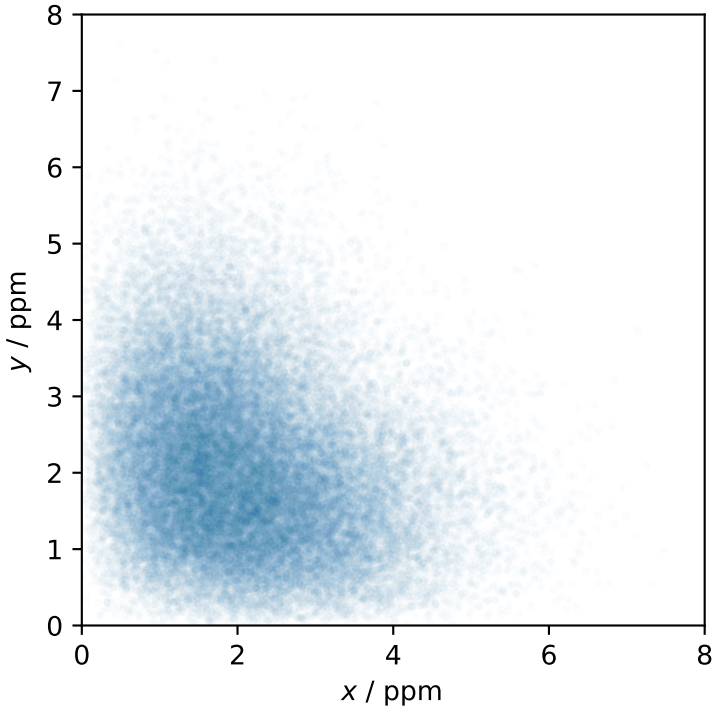


Figure 6.2: Random sampling of Czjzek distribution of shielding tensors in polar coordinates.

Generating probability distribution functions from a Czjzek model

The *pdf()* (page 494) instance method will generate a probability distribution function on the supplied grid using the analytical function defined above. The provided grid – passed to the `pos` keyword argument – needs to be defined in either Cartesian or polar coordinates depending on if the `polar` attribute is `True` or `False`.

Below, we generate and plot a probability distribution on a ζ-η Cartesian grid where `zeta_range` and `eta_range` define the desired coordinates in each dimension of the grid system.

```
import numpy as np

cz_model = CzjzekDistribution(sigma=1.2, polar=False)  # sample in (zeta, eta)

zeta_range = np.linspace(-12, 12, num=200)  # pre-defined zeta range in units of ppm
eta_range = np.linspace(0, 1, num=50)       # pre-defined eta range.
zeta_grid, eta_grid, amp = cz_model.pdf(pos=[zeta_range, eta_range])
```

Here, `zeta_grid` and `eta_grid` are numpy arrays defining a set of pair-wise points on the grid system, and `amp` is another numpy array holding the probability density at each point on the grid. Below, the distribution is plotted

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

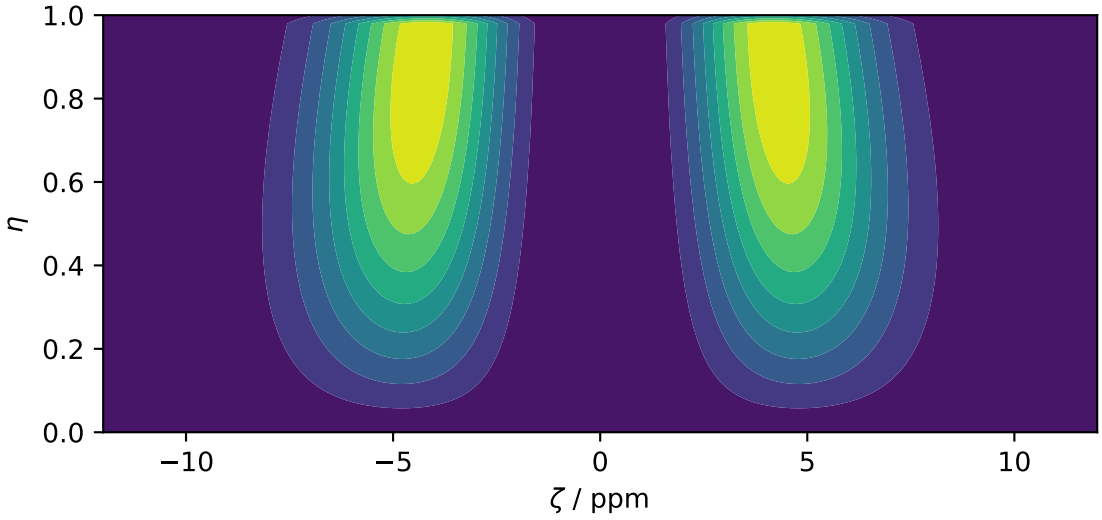


Figure 6.3: Czjzek Distribution of shielding tensors.

—

The probability distribution function can also be generated in polar coordinates. The workflow is the same, except we now define an (x, y) grid system using the variables `x_range` and `y_range`. The code to generate and plot the polar Czjzek distribution is shown below.

```
cz_model_polar = CzjzekDistribution(sigma=1.2, polar=True)  # sample in (x, y)

x_range = np.linspace(0, 10, num=150)
y_range = np.linspace(0, 10, num=150)
x_grid, y_grid, amp = cz_model_polar.pdf(pos=[x_range, y_range])

plt.figure(figsize=(4, 4))
plt.contourf(x_grid, y_grid, amp, levels=10)
plt.xlabel("$x$ / ppm")
```

(continues on next page)

(continued from previous page)

```
plt.ylabel("$y$ / ppm")
plt.tight_layout()
plt.show()
```

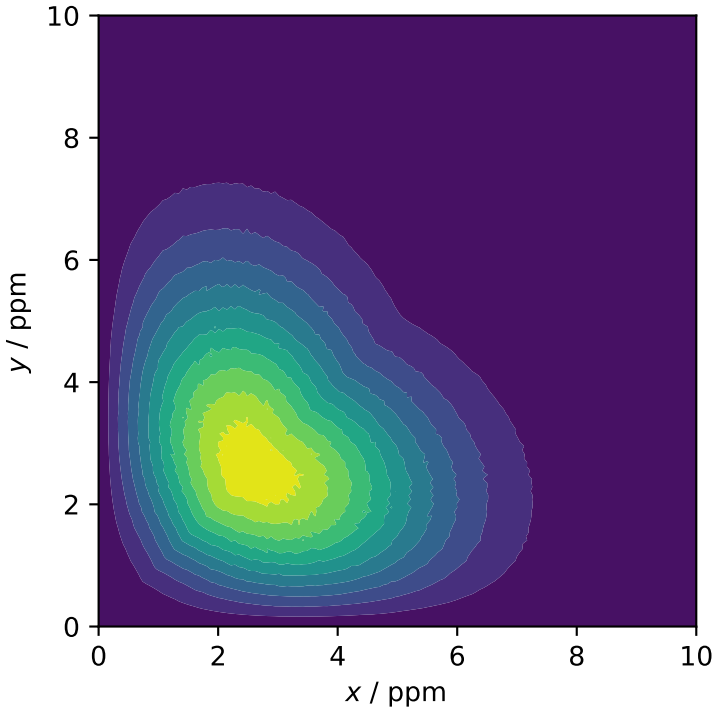


Figure 6.4: Czjzek Distribution of shielding tensors in polar coordinates.

Distributions of shielding and quadrupolar tensors and a note on units

The *CzjzekDistribution* (page 493) class can be used to generate distributions for both symmetric chemical shielding tensors and electric field gradient tensors. It is important to note the Czjzek model is only aware of the Haberlen components of the tensors and not the units of the tensor. In the above code cells, we generated distributions for symmetric shielding tensors and assumed all units for ζ were in ppm.

Quadrupolar tensors, defined using values of C_q in MHz and unitless η, can also be drawn from the Czjzek distribution in the same manner; however, the dimensions are assumed to be in units of MHz. The following code draws a distribution of quadrupolar tensor parameters.

```
Cq_range = np.linspace(-12, 12, num=200)  # pre-defined Cq range in units of MHz
eta_range = np.linspace(0, 1, num=50)     # pre-defined eta range.
Cq_grid, eta_grid, amp = cz_model.pdf(pos=[Cq_range, eta_range])
```

the units for `Cq_range` and `Cq_grid` are assumed in MHz. Similarly, x and y are assumed to be in units of MHz when sampling quadrupolar tensors in polar coordinates.

```
x_range = np.linspace(0, 10, num=150)  # pre-defined x grid in units of MHz
y_range = np.linspace(0, 10, num=150)  # pre-defined y grid in units of MHz
x_grid, y_grid, amp = cz_model_polar.pdf(pos=[x_range, y_range])
```

Generating a list of SpinSystem objects from a Czjzek model

The utility function *single_site_system_generator()* (page 484), further described in *Single Site System Generator* (page 68), can be used in conjunction with the *CzjzekDistribution* (page 493) class to generate a set of spin systems whose tensor parameters follow the Czjzek distribution.

```
from mrsimulator.utils.collection import single_site_system_generator

# Distribution of quadrupolar tensors
cz_model = CzjzekDistribution(sigma=0.7)
Cq_range = np.linspace(0, 10, num=100)
eta_range = np.linspace(0, 1, num=50)

# Create (Cq, eta) grid points and amplitude
Cq_grid, eta_grid = np.meshgrid(Cq_range, eta_range)
_, _, amp = cz_model.pdf(pos=[Cq_range, eta_range])

sys = single_site_system_generator(
    isotope="27Al",
    quadrupolar={"Cq": Cq_grid * 1e6, "eta": eta_grid},  # Cq argument in units of Hz
    abundance=amp,
)
```

A spin system will be generated for each point on the ζ-η grid, and the abundance of each spin system matches the amplitude of the Czjzek distribution. When working in polar coordinates, the set of (x, y) coordinates needs to be transformed into a set of (ζ, η) coordinates before being passed to the *single_site_system_generator()* (page 484) function. The utility function `x_y_to_zeta_eta()` performs this transformation, as shown below.

```
from mrsimulator.models.utils import x_y_to_zeta_eta

# Sample distribution of shielding tensors in polar coords
cz_model_polar = CzjzekDistribution(sigma=0.7, polar=True)
x_range = np.linspace(0, 10, num=150)
y_range = np.linspace(0, 10, num=150)

# Create (x, y) grid points and amplitude
x_grid, y_grid, amp = cz_model_polar.pdf(pos=[x_range, y_range])

# To transformation (x, y) -> (zeta, eta)
zeta_grid, eta_grid = x_y_to_zeta_eta(x_grid, y_grid)
```

—

Mini-gallery using the Czjzek distributions

- *Czjzek distribution (Shielding and Quadrupolar)* (page 190)
- *Czjzek distribution, ^{27}Al (I=5/2) 3QMAS* (page 251)
- *Fitting a Czjzek Model* (page 315)

6.1.2 Extended Czjzek distribution

The Extended Czjzek distribution models random variations of second-rank traceless symmetric tensors about a non-zero tensor. Unlike the Czjzek distribution, the Extended Czjzek model has no known analytical function for the probability distribution. Therefore, mrsimulator relies on random sampling to approximate the probability distribution function. See *Extended Czjzek distribution* (page 374) and references within for a further description of the model.

Extended Czjzek distribution of symmetric shielding tensors

To generate an extended Czjzek distribution, use the *ExtCzjzekDistribution* (page 495) 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 a delta function at the dominant tensor parameters. As the value of `eps` 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 496) 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()
```

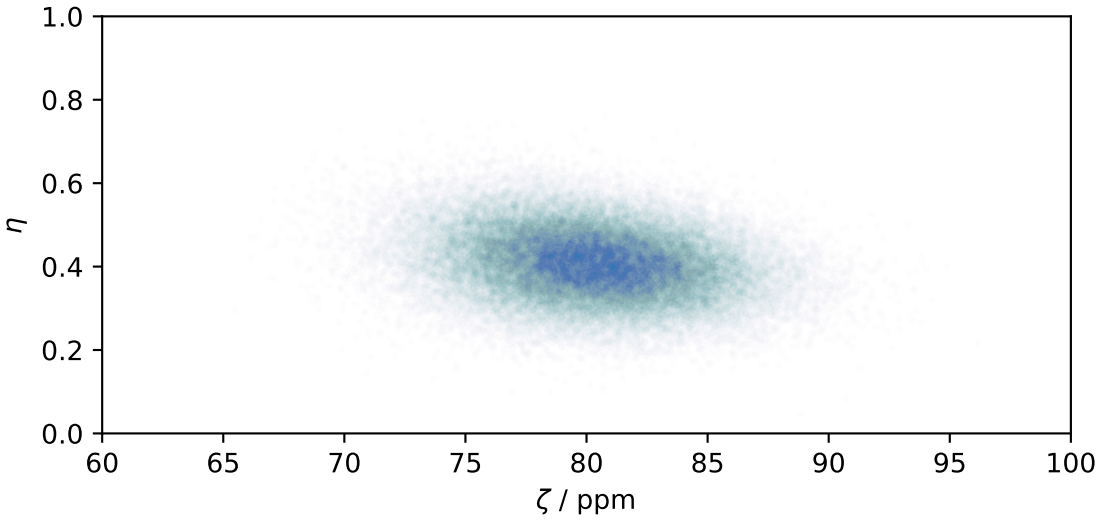


Figure 6.5: Extended Czjzek Distribution of shielding tensors.

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 495) method.

```
import numpy as np

Cq_range = np.linspace(2, 6, num=100)  # pre-defined Cq range in MHz.
eta_range = np.linspace(0, 1, num=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_grid, eta_grid, amp = model_quad.pdf(pos=[Cq_range, eta_range], size=400000)
```

As with the case of the 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 495) method. The pdf method also accepts the keyword argument `size` which defines the number of random samples used to approximate the probability distribution. A larger number will create better approximations, although this increased quality comes at the expense of computation time. The output `Cq_grid`, `eta_grid`, 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_grid, eta_grid, amp, levels=10)
plt.xlabel("$C_q$ / MHz")
plt.ylabel("$\eta$")
plt.tight_layout()
plt.show()
```

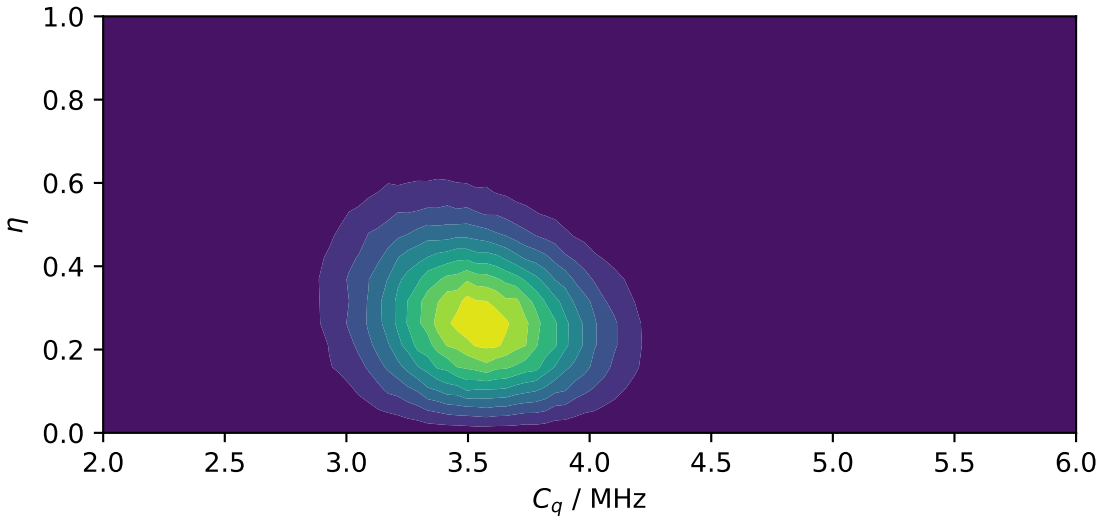


Figure 6.6: Extended Czjzek Distribution of EFG tensors.

Extended Czjzek distribution in polar coordinates

As with the Czjzek distribution, we can sample an Extended Czjzek distribution on a polar (x, y) grid. Below, we construct two equivalent *ExtCzjzekDistribution* (page 495) objects, except one is defined in polar coordinates.

```
quad_tensor = {"Cq": 4.2, "eta": 0.15}  # Cq assumed in MHz
ext_cz_model = ExtCzjzekDistribution(quad_tensor, eps=0.4)
ext_cz_model_polar = ExtCzjzekDistribution(quad_tensor, eps=0.4, polar=True)

# Distribution in cartesian (zeta, eta) coordinates
Cq_range = np.linspace(2, 8, num=50)
eta_range = np.linspace(0, 1, num=20)
Cq_grid, eta_grid, amp = ext_cz_model.pdf(pos=[Cq_range, eta_range], size=2000000)

# Distribution in polar coordinates
x_range = np.linspace(0, 6, num=36)
y_range = np.linspace(0, 6, num=36)
x_grid, y_grid, amp_polar = ext_cz_model_polar.pdf(pos=[x_range, y_range], size=2000000)

# Plot the distributions
fig, ax = plt.subplots(1, 2, figsize=(9, 4), gridspec_kw={"width_ratios": (5, 4)})
ax[0].contourf(Cq_grid, eta_grid, amp, levels=10)
ax[0].set_xlabel("$C_q$ / MHz")
ax[0].set_ylabel("$\eta$")
ax[0].set_title("Cartesian coordinates")
ax[1].contourf(x_grid, y_grid, amp_polar, levels=10)
ax[1].set_xlabel("x / MHz")
ax[1].set_ylabel("y / MHz")
ax[1].set_title("Polar coordinates")

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

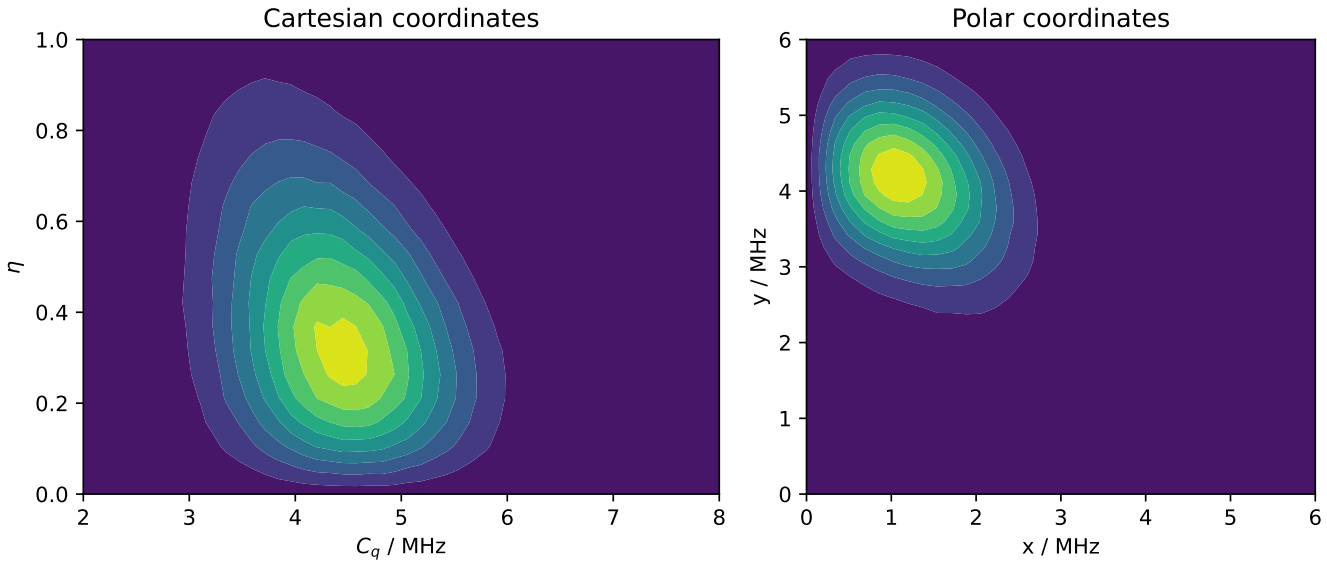


Figure 6.7: Two equivalent Extended Czjzek distributions in Cartesian (ζ, η) coordinates (left) and in polar (x, y) coordinates (right).

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.

Mini-gallery using the extended Czjzek distributions

- *Extended Czjzek distribution (Shielding and Quadrupolar)* (page 194)
- *Simulating site disorder (crystalline)* (page 248)
- *Extended Czjzek fitting of ^{139}La MAS NMR of $La_{0.2}Y_{1.8}Si_2O_7$* (page 307)

6.2 User-defined distributions

User-defined distributions of site parameters can be passed to the *`single_site_system_generator()`* (page 484) method to create custom models of materials.

6.2.1 Single Site System Generator

Custom Site and SpinSystem parameters can be passed to the *`single_site_system_generator()`* (page 484) method to create a list of uncoupled spin systems. Each *SpinSystem* (page 388) in the returned list holds a single *Site* (page 394) object since the backend simulation is more efficient for single site spin systems. Import the method as below

```
from mrsimulator.utils.collection import single_site_system_generator
```

The arguments passed to the function, defined in Table 6.1, can either be a scalar quantity (`float` or `str`, where applicable) or a `list`/`np.array` of those quantities. All lists passed must have the same length, otherwise an error will be thrown. For example,

```
single_site_system_generator(
    isotope=["1H", "1H", "13C", "17O"],
    isotropic_chemical_shift=[1.3, 3.7, 65.0],
)
```

Out:

```
Traceback (most recent call last):
...
ValueError: An array or list was either too short or too long. All arguments must be the
same size. If one attribute is a type list of length n, then all attributes with list types
must also be of length n, and all remaining attributes must be scalar (singular float, int,
or str).
```

The attributes of each returned spin system at a certain index correspond to the attribute passed at that index. For example,

```
single_site_system_generator(
    isotope=["1H", "1H", "13C"],
    isotropic_chemical_shift=[1.3, 3.7, 65.0],
)
```

returns a list of 3 spin systems. The first two spin systems represent proton sites with isotropic chemical shifts of 1.3 ppm and 3.7 ppm, respectively. The third spin system is a ^{13}C site with a chemical shift of 65.0 ppm.

Broadcasting Length of List

Arguments passed as a single value will be broadcast to a list of that value with the same length as other lists passed. For example

```
single_site_system_generator(
    isotope=["1H", "1H", "1H"],
    isotropic_chemical_shift=2.0,
)
```

is equivalent to calling

```
single_site_system_generator(
    isotope=["1H", "1H", "1H"],
    isotropic_chemical_shift=[2.0, 2.0, 2.0],
)
```

Passing lists of Tensor Parameters

Tensor parameters for sites are passed as dictionaries where the keywords represent the tensor attribute and the values are single values or a `list`/`np.array` of values. Again, these lists must have the same length of all other lists passed. Single values will be broadcast to a list of that value with the same length as other lists passed. For example

```
single_site_system_generator(
    isotope="13C",
    shielding_symmetric={
        "zeta": [5, 10, 15, 20, 25],
```

(continues on next page)

(continued from previous page)

```
        "eta": 0.3,
    },
)
```

returns a list of five ^{13}C spin systems with different `shielding_symmetric.zeta` values but the same `shielding_symmetric.eta` value.

If you need to intermix sites with and without tensor parameters, simply put `None` at the index of the site without the tensor parameter.

```
single_site_system_generator(
    isotope=["1H", "17O"],
    quadrupolar={
        "Cq": [None, 3.2e6],
        "eta": [None, 0.5],
    },
)
```

Examples using `single_site_system_generator()`

- *Amorphous material, ^{29}Si (I=1/2)* (page 181)
- *Amorphous material, ^{27}Al (I=5/2)* (page 186)
- *Czjzek distribution (Shielding and Quadrupolar)* (page 190)
- *Extended Czjzek distribution (Shielding and Quadrupolar)* (page 194)
- *Simulating site disorder (crystalline)* (page 248)
- *Czjzek distribution, ^{27}Al (I=5/2) 3QMAS* (page 251)
- *^{13}C 2D MAT NMR of L-Histidine* (page 321)
- *^{17}O 2D DAS NMR of Coesite* (page 332)
- *^{87}Rb 2D 3QMAS NMR of $RbNO_3$* (page 338)

Table 6.1: Arguments for `single_site_system_generator`

Name	Type	Description
`isotope`	`str` or list of `str`	A *required* string or list of strings representing the label of the `isotope` attribute of the *Site* (page 394) (e.g. `"1H"` or `["29Si", "17O"]`).
`isotropic_chemical_shift`	`float`, list of `float`, or numpy array	An *optional* number or list of numbers representing the `isotropic_chemical_shift` attribute of the *Site* (page 394) (e.g. `17.3` or `[2.4, 19.5]`) in ppm. The default value is `0`.
`shielding_symmetric`	`dict`	An *optional* dictionary representing the `shielding_symmetric` attribute of the *Site* (page 394) where the keys are valid *SymmetricTensor* (page 475) attributes and the values are floats or lists/numpy arrays of floats. The default is `None`.
`shielding_antisymmetric`	`dict`	An *optional* dictionary representing the `shielding_antisymmetric` attribute of the *Site* (page 394) where the keys are valid *AntisymmetricTensor* (page 478) attributes and the values are floats or lists/numpy arrays of floats. The default is `None`.
`quadrupolar`	`dict`	An *optional* dictionary representing the `quadrupolar` attribute of the *Site* (page 394) where the keys are valid *SymmetricTensor* (page 475) attributes and the values are floats or lists/numpy arrays of floats. The default is `None`.
`abundance`	`float`, list of `float`, or numpy array	An *optional* number or list of numbers representing the `abundance` attribute of the SpinSystem (e.g. `0.182` or `[85, 7.3]`. By default, the abundance of each spin system will be set to `1 / n_sys` where `n_sys` is the number of spin systems generated.
`site_name`	`str` or list of `str`	An *optional* string or list of strings representing the `name` attribute of each *Site* (page 394). By default, each *Site* (page 394) will take the default name of `None`
`site_label`	`str` or list of `str`	An *optional* string or list of strings representing the `label` attribute of each *Site* (page 394). By default, each *Site* (page 394) will take the default label of `None`
`site_description`	`str` or list of `str`	An *optional* string or list of strings representing the `description` attribute of each *Site* (page 394). By default, each *Site* (page 394) will take the default description of `None`

CHAPTER

SEVEN

METHODS LIBRARY

For convenience, **mrsimulator** offers the following pre-built methods -

- *Bloch Decay Spectrum* (page 73)
- *Bloch Decay Central Transition* (page 74)
- *Multi-Quantum VAS* (page 75)
- *Satellite-Transition VAS* (page 76)
- *SSB2D* (page 77)

An example of the syntax that all library methods follows is shown below.

```
from mrsimulator.method import SpectralDimension
from mrsimulator.method.lib import BlochDecaySpectrum

lib_method = BlochDecaySpectrum(
    channels=["29Si"],  # list of isotopes
    magnetic_flux_density=4.7,  # T
    rotor_angle=57.735 * 3.1415 / 180,  # rad
    rotor_frequency=10000,  # Hz
    spectral_dimensions=[
        SpectralDimension(count=512, spectral_width=5e4, reference_offset=10),
    ],
)
```

where *BlochDecaySpectrum* can be replaced with another library method class. Each method has the `channels` attribute, which is a list of isotopes probed by the method as well as the `magnetic_flux_density`, `rotor_angle`, and `rotor_frequency` attributes which define the global experiment parameters. See Table 7.1 for more details.

The method object also has the `spectral_dimensions` attribute, which contains a list of SpectralDimension objects defining the spectral grid. A 2D method will have two spectral dimensions in this list, whereas a 1D method will only have one. See Table 7.2 for the attributes of a SpectralDimension object.

7.1 Bloch Decay Spectrum

The *BlochDecaySpectrum* (page 425) class simulates the Bloch decay spectrum.

```
from mrsimulator.method.lib import BlochDecaySpectrum
from mrsimulator.method import SpectralDimension

method = BlochDecaySpectrum(
    channels=["1H"],
```

(continues on next page)

(continued from previous page)

```
    rotor_frequency=12500,  # in Hz
    rotor_angle=54.735 * 3.14159 / 180,  # in rad
    magnetic_flux_density=9.4,  # in tesla
    spectral_dimensions=[
        SpectralDimension(
            count=1024,
            spectral_width=25e3,  # in Hz
            reference_offset=-4e3,  # in Hz
        )
    ],
)
```

Examples

- *Wollastonite, ^{29}Si (I=1/2)* (page 146)
- *Selective Excitations using Custom Isotopes* (page 151)
- *Influence of 14N on 13C NMR MAS spectra of glycine* (page 155)
- *Coupled spin-1/2 (Static dipolar spectrum)* (page 170)
- *Coupled spin-1/2 (CSA + heteronuclear dipolar + J-couplings)* (page 172)
- *Protein GB1, ^{13}C and ^{15}N (I=1/2)* (page 179)
- *Amorphous material, ^{29}Si (I=1/2)* (page 181)
- *Czjzek distribution (Shielding and Quadrupolar)* (page 190)
- *Extended Czjzek distribution (Shielding and Quadrupolar)* (page 194)
- *^{31}P MAS NMR of crystalline Na2PO4 (CSA)* (page 255)
- *^{31}P static NMR of crystalline Na2PO4 (CSA)* (page 261)
- *^{13}C MAS NMR of Glycine (CSA) [960 Hz]* (page 265)
- *^{13}C MAS NMR of Glycine (CSA) multi-spectra fit* (page 271)
- *1D PASS/MAT sideband order cross-section* (page 277)
- *^{27}Al MAS NMR of YAG (1st and 2nd order Quad)* (page 291)
- *^{2}H MAS NMR of Methionine* (page 297)
- *^{119}Sn MAS NMR of SnO* (page 301)

7.2 Bloch Decay Central Transition

The *BlochDecayCTSpectrum* (page 431) class simulates the Bloch decay central transition selective spectrum.

```
from mrsimulator.method.lib import BlochDecayCTSpectrum
from mrsimulator.method import SpectralDimension

method = BlochDecayCTSpectrum(
    channels=["87Rb"],
    rotor_frequency=12500,  # in Hz
    rotor_angle=54.735 * 3.14159 / 180,  # in rad
```

(continues on next page)

(continued from previous page)

```
    magnetic_flux_density=9.4,  # in tesla
    spectral_dimensions=[
        SpectralDimension(
            count=1024,
            spectral_width=25e3,  # in Hz
            reference_offset=-4e3,  # in Hz
        )
    ],
)
```

Examples

- *Using Custom Isotopes* (page 149)
- *Potassium Sulfate, ^{33}S (I=3/2)* (page 157)
- *Coesite, ^{17}O (I=5/2)* (page 160)
- *Non-coincidental Quad and CSA, ^{17}O (I=5/2)* (page 162)
- *Amorphous material, ^{27}Al (I=5/2)* (page 186)
- *Czjzek distribution (Shielding and Quadrupolar)* (page 190)
- *Extended Czjzek distribution (Shielding and Quadrupolar)* (page 194)
- *^{17}O MAS NMR of crystalline Na_2SiO_3 (2nd order quad)* (page 281)
- *^{11}B MAS NMR of Lithium orthoborate crystal* (page 287)
- *Extended Czjzek fitting of ^{139}La MAS NMR of $La_{0.2}Y_{1.8}Si_2O_7$* (page 307)
- *Fitting a Czjzek Model* (page 315)

7.3 Multi-Quantum VAS

The ***ThreeQ_VAS*** (page 438), ***FiveQ_VAS*** (page 444), and ***SevenQ_VAS*** (page 450) classes all simulate a multiple quantum VAS spectrum. The spinning speed for all three methods is fixed at infinite speed. The spectrum is also sheared such that the correlated dimensions are the isotropic dimension and the VAS dimension.

```
from mrsimulator.method.lib import ThreeQ_VAS
from mrsimulator.method import SpectralDimension

method = ThreeQ_VAS(
    channels=["87Rb"],
    magnetic_flux_density=7,  # in T
    spectral_dimensions=[
        SpectralDimension(
            count=128,
            spectral_width=3e3,  # in Hz
            reference_offset=-2e3,  # in Hz
            label="Isotropic dimension",
        ),
        SpectralDimension(
            count=512,
            spectral_width=1e4,  # in Hz
```

(continues on next page)

(continued from previous page)

```
            reference_offset=-5e3,  # in Hz
            label="MAS dimension",
        ),
    ],
)
```

The other methods, employing five and seven quantum transitions, can be imported as follows:

```
from mrsimulator.method.lib import FiveQ_VAS
from mrsimulator.method.lib import SevenQ_VAS
```

Examples

- *$RbNO_3$, ^{87}Rb (I=3/2) 3QMAS* (page 199)
- *Albite, ^{27}Al (I=5/2) 3QMAS* (page 203)
- *Coesite, ^{17}O (I=5/2) 3QMAS* (page 221)
- *Simulating site disorder (crystalline)* (page 248)
- *Czjzek distribution, ^{27}Al (I=5/2) 3QMAS* (page 251)
- *^{87}Rb 2D 3QMAS NMR of $RbNO_3$* (page 338)

7.4 Satellite-Transition VAS

The *ST1_VAS* (page 457) and *ST2_VAS* (page 463) classes simulate a sheared and scaled satellite and central transition correlation spectrum. The spinning speed for these methods is fixed at infinite speed.

```
from mrsimulator.method.lib import ST1_VAS
from mrsimulator.method import SpectralDimension

method = ST1_VAS(
    channels=["87Rb"],
    magnetic_flux_density=9.4,  # in T
    spectral_dimensions=[
        SpectralDimension(
            count=128,
            spectral_width=1e3,  # in Hz
            reference_offset=-5e3,  # in Hz
            label="Isotropic dimension",
        ),
        SpectralDimension(
            count=256,
            spectral_width=1e4,  # in Hz
            reference_offset=-3e3,  # in Hz
            label="MAS dimension",
        ),
    ],
)
```

Examples

- *$RbNO_3$, ^{87}Rb (I=3/2) STMAS* (page 205)

- *Co59 (I=7/2) STMAS* (page 212)
- *Co59 (I=7/2) STMAS* (page 214)

7.5 SSB2D

The *SSB2D* (page 469) class simulates a sheared 2D finite to infinite speed MAS correlation spectrum. The spinning speed for the second spectral dimension is fixed at infinite spinning speed

```
from mrsimulator.method.lib import SSB2D
from mrsimulator.method import SpectralDimension

method = SSB2D(
    channels=["13C"],
    magnetic_flux_density=7,  # in T
    rotor_frequency=1500,  # in Hz
    spectral_dimensions=[
        SpectralDimension(
            count=16,
            spectral_width=16 * 1500,  # in Hz (= count * rotor_frequency)
            reference_offset=-5e3,  # in Hz
            label="Sideband dimension",
        ),
        SpectralDimension(
            count=512,
            spectral_width=1e4,  # in Hz
            reference_offset=-4e3,  # in Hz
            label="Isotropic dimension",
        ),
    ],
)
```

Examples

- *Itraconazole, ^{13}C (I=1/2) PASS* (page 230)
- *Rb_2SO_4, ^{87}Rb (I=3/2) QMAT* (page 232)
- *^{13}C 2D MAT NMR of L-Histidine* (page 321)
- *^{87}Rb 2D QMAT NMR of Rb_2SO_4* (page 327)

7.6 Attribute Summaries

Table 7.1: Attribute description for generic library methods.

Attribute Name	Type	Description
channels	`list`	A *required* list of isotopes given as strings over which the given method applies. For example, `["1H"]`.
magnetic_flux_density	`float`	An *optional* float describing the macroscopic magnetic flux density of the applied external magnetic field in tesla. For example, `18.8` tesla. The default value is `9.4` tesla.
rotor_frequency	`float`	An *optional* float describing the sample rotation frequency in Hz. For example, `2000` Hz. The default value is `0` Hz.
rotor_angle	`float`	An *optional* float describing the angle between the sample rotation axis and the external magnetic field in radians. The default value is the magic angle, `54.735 * 3.14159 / 180 = 0.955305` radians.
spectral_dimensions	`list`	A list of *SpectralDimension* (page 410) objects describing the spectral dimensions for the method.
simulation	CSDM object	A CSDM object representing the spectrum simulated by the method. By default, the value is `None`. A value is assigned to this attribute when you run the simulation using the *`run()`* (page 385) method.
experiment	CSDM object	An *optional* CSDM object holding an experimental measurement of the method. The default value is `None`

Table 7.2: Spectral dimension attributes for use with library methods.

Attribute Name	Type	Description
count	`int`	An *optional* integer representing the number of points, N, along the spectroscopic dimension. For example, `4096`. The default value is `1024`.
spectral_width	`float`	An *optional* float representing the width, Δx, of the spectroscopic dimension in Hz. For example, `10e3` for 10 kHz. The default value is `25000` Hz.
reference_offset	`float`	An *optional* float representing the reference offset, x_0, of the spectroscopic dimension in Hz. For example, `-8000` Hz. The default value is `0`.
origin_offset	`float`	An *optional* float representing the origin offset, or Larmor frequency, along the spectroscopic dimension in units of Hz. The default value is `None` and the origin offset is set to the Larmor frequency of isotope from the *`channels`* (page 403) attribute of the method containing the spectral dimension.

CHAPTER

EIGHT

METHOD

While **mrsimulator**'s organization of the *SpinSystem* (page 388) object and its composite objects, *Site* (page 394) and *Coupling* (page 399), are easily understood by anyone familiar with the underlying physical concepts, the organization of the *Method* (page 403) object in **mrsimulator** and its related composite objects require a more detailed explanation of their design. This section assumes that you are already familiar with the topics covered in the Introduction sections *Getting Started* (page 23), *Isotopomers Example* (page 29), and *Least-Squares Fitting Example* (page 37).

Note: Before writing your own custom Method, check if any of our pre-built methods in the *Methods Library* (page 73) can serve your needs.

8.1 Overview

An experimental NMR method involves a sequence of rf pulses, free evolution periods, and sample motion. The Method object in **mrsimulator** models the spectrum from an NMR pulse sequence. The Method object is designed to be versatile in its ability to model spectra from various multi-pulse NMR methods using concepts from the symmetry pathway approach where a pulse sequence is understood in terms of a set of desired (and undesired) *transition pathways*. Each transition pathway is associated with a single resonance in a multi-dimensional NMR spectrum. The transition pathway signal encodes information about the spin system interactions in its amplitude and correlated frequencies. Consider the illustration of a 2D pulse sequence shown below, where a desired signal for the method is associated with a particular transition pathway, $\hat{A} \rightarrow \hat{B} \rightarrow \hat{C} \rightarrow \hat{D} \rightarrow \hat{E} \rightarrow \hat{F}$.

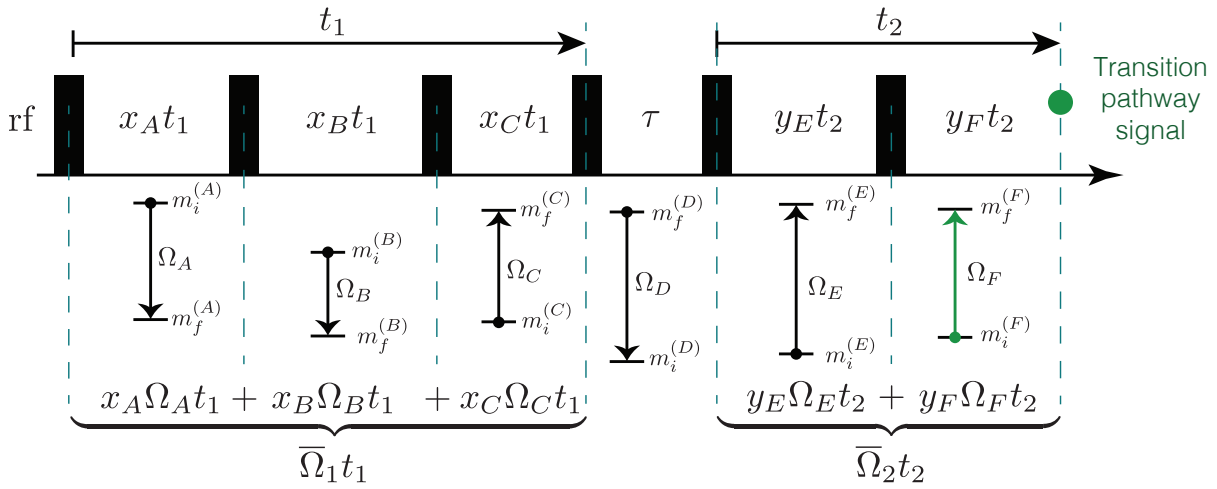


Figure 8.1: An illustration of a two-dimensional NMR pulse sequence leading up to the acquisition of the signal from a transition pathway

Here, the first spectral dimension, i.e., the Fourier transform of the transition pathway signal as a function of t_1, derives its *average frequency*, $\overline{\Omega}_1$, from a weighted average of the $\hat{A}$, $\hat{B}$, and $\hat{C}$ transition frequencies. The second spectral dimension, i.e., the FT with respect to t_2, derives its average frequency, $\overline{\Omega}_2$, from a weighted average of the $\hat{E}$, and $\hat{F}$ transition frequencies. Much of the experimental design and implementation of an NMR method is in identifying the desired transition pathways and finding ways to acquire their signals while eliminating all undesired transition pathway signals.

While NMR measurements occur in the time domain, **mrsimulator** simulates the corresponding multi-dimensional spectra directly in the frequency domain. The Method object in **mrsimulator** needs only a few details of the NMR pulse sequence to generate the spectrum. It mimics the result of the pulse sequence given the desired transition pathways and their complex amplitudes and average frequencies in each spectroscopic dimension of the dataset. To this end, a Method object is organized according to the UML diagram below.

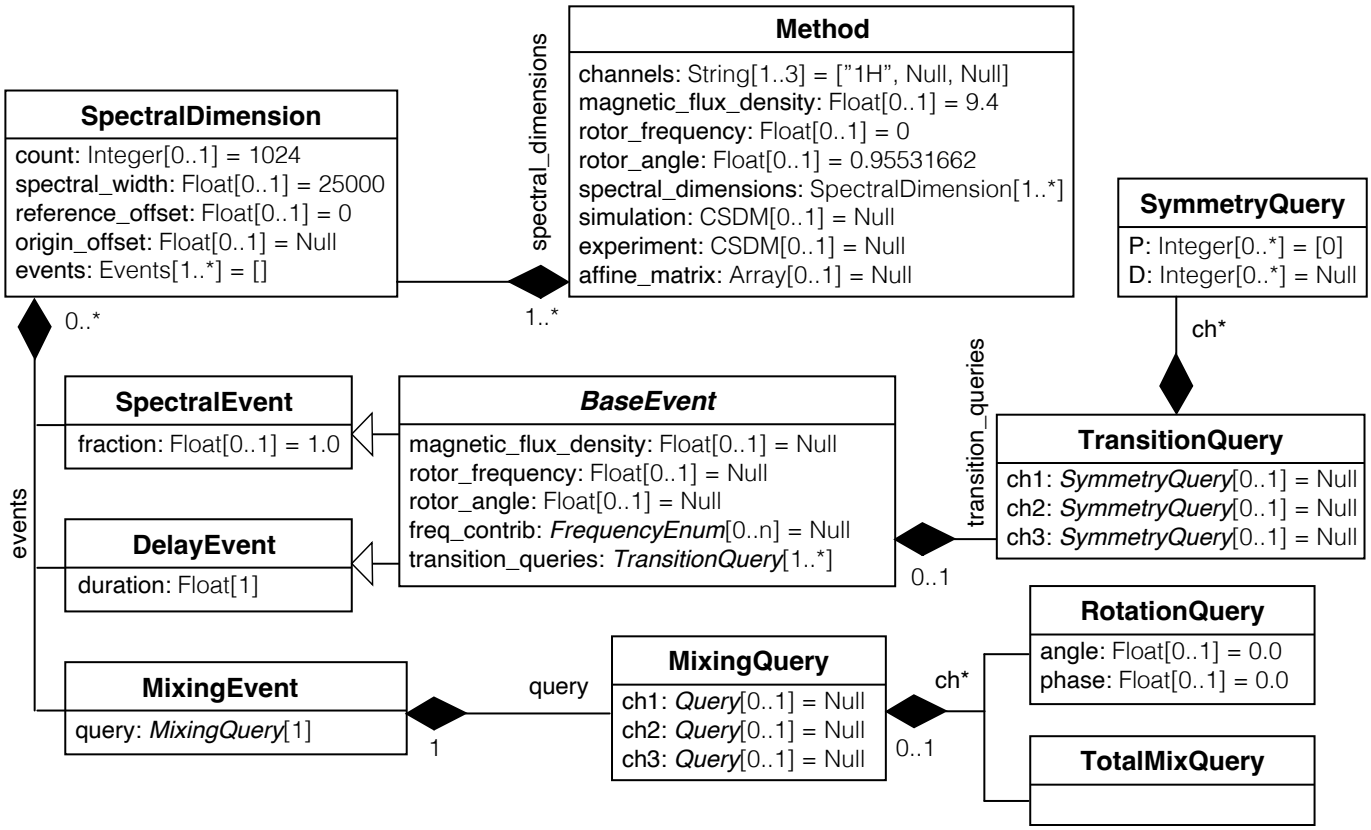


Figure 8.2: Unified Modeling Language class diagram of the Method object in mrsimulator.

Note: In UML (Unified Modeling Language) diagrams, each class is represented with a box that contains two compartments. The top compartment has the class's name, and the bottom compartment contains the class's attributes. Default attribute values are shown as assignments. A composition is depicted as a binary association decorated with a filled black diamond. Inheritance is shown as a line with a hollow triangle as an arrowhead.

At the heart of a Method object, assigned to its attribute `spectral_dimensions`, is an ordered list of *SpectralDimension* (page 410) objects in the same order as the time evolution dimensions of the experimental NMR sequence. In each SpectralDimension object is an ordered list of *Events* (page 412) objects assigned to the attribute `events`; Event objects are divided into three types: (1) *`SpectralEvent()`* (page 412), (2) *`DelayEvent()`* (page 414), and (3) *`MixingEvent()`* (page 416). This ordered list of Event objects is used to select the desired transition pathways and determine their average frequency and complex amplitude in the SpectralDimension.

Warning: DelayEvent objects are not available in version 0.7 of **mrsimulator**.

SpectralEvent and DelayEvent objects define which transitions are observed during the event and under which transition-dependent frequency contributions they evolve. No coherence transfer among transitions or populations occurs in a spectral or delay event. The transition-dependent frequency contributions during an Event are selected from a list of *enumeration literals* (page 416) and placed in the `freq_contrib` attribute of the event. Frequency contributions can be individually excluded by placing an exclamation mark in front of the string representing the enumeration literal. If `freq_contrib` is left unspecified, i.e., the value of `freq_contrib` is set to `None`, a default list holding the enumeration literals for *all* contributions is generated for the event.

Note: All frequency contributions from direct and indirect spin-spin couplings are calculated in the weak-coupling limit in **mrsimulator**.

Additionally, the user can affect transition frequencies during a spectral or delay event by changing other measurement attributes: `rotor_frequency`, `rotor_angle`, and `magnetic_flux_density`. If left unspecified, these attributes default to the values of the identically named global attributes in the Method object. SpectralEvent objects use the `fraction` attribute to calculate the weighted average frequency during the spectral dimension for each selected transition pathway.

Inside SpectralEvent and DelayEvent objects, is a list of *`TransitionQuery()`* (page 419) objects (*vide infra*) which determine which transitions are observed during the event. Method objects in **mrsimulator** are general-purpose because they are designed for an arbitrary spin system, i.e., a method does not know the spin system in advance. When designing a Method object, you cannot identify and select a transition through its initial and final eigenstate quantum numbers. Transition selection is done through TransitionQuery and *`SymmetryQuery()`* (page 420) objects during individual spectral or delay events. TransitionQuery objects can hold a SymmetryQuery object in the attributes `ch1`, `ch2`, or `ch3`, which act on specific isotopes defined by the `channels` attribute in Method. It is only during a simulation that the Method object uses its TransitionQuery objects to determine the selected transition pathways for a given SpinSystem object by the initial and final eigenstate quantum numbers of each transition.

Between adjacent SpectralEvent or DelayEvent objects, **mrsimulator** defaults to *total mixing*, i.e., connecting all selected transitions in the two adjacent spectral or delay events. This default behavior can be overridden by placing an explicit MixingEvent object between such events. Inside MixingEvent objects is a *`MixingQuery()`* (page 422) object, which determines the coherence transfer amplitude between transitions. A MixingQuery object holds *`RotationQuery()`* (page 423) objects acting on specific isotopes in the spin system. As before, the isotope upon which the RotationQuery objects act is determined by the `channels` attribute in the Method object.

In this guide to designing custom Method objects, we begin with a brief review of the relevant *Symmetry Pathway* concepts employed in **mrsimulator**. This review is necessary for understanding (1) how transitions are selected during spectral and delay events and (2) how average signal frequencies and amplitudes in each spectral dimension are determined. We outline the procedures for designing and creating TransitionQuery and MixingQuery objects for single- and multi-spin transitions and how to use them to select the transition pathways with the desired frequency and amplitudes in each spectral dimension of your custom Method object. In multi-dimensional spectra, we illustrate how the desired frequency correlation can sometimes be achieved by using an appropriate affine transformation. We also examine how changing the frequency contributions in SpectralEvent of DelayEvent objects can be used to obtain the desired frequency and amplitude behavior. The ability to select *frequency contributions* (page 416) can often reduce the number of events needed in the design of your custom Method object.

8.2 Theoretical Background

Before giving details on how to create a custom Method object, we review a few key concepts about spin transitions and *transition symmetry functions*.

The number of quantized energy eigenstates for N coupled nuclei is

$$\Upsilon_{\{I_1,I_2,...,I_N\}} = \prod_{u=1}^{N}(2I_u + 1), \tag{8.1}$$

where I_u is the total spin angular momentum quantum number of the uth nucleus. The transition from quantized energy level E_i to E_j is one of

$$\mathcal{N}_{\{I_1,I_2,...,I_N\}} = \frac{\Upsilon_{\{I_1,I_2,...,I_N\}}!}{(\Upsilon_{\{I_1,I_2,...,I_N\}} - 2)!} \tag{8.2}$$

possible transitions between the Υ levels. Here we count $i \rightarrow j$ and $j \rightarrow i$ as different transitions. For example, a single spin with angular momentum quantum number $I = 3/2$ will have $\Upsilon_{\{3/2\}} = 2I + 1 = 4$ energy levels and $\mathcal{N}_{\{3/2\}} = 2I(2I+1) = 12$ possible NMR transitions. A two spin system, with quantum numbers $I = 1/2$ and $S = 1/2$, will have

$$\Upsilon_{\{1/2,1/2\}} = (2I + 1) \cdot (2S + 1) = 4 \tag{8.3}$$

energy levels and

$$\mathcal{N}_{\{1/2,1/2\}} = \frac{[(2I + 1) \cdot (2S + 1)]!}{((2I + 1) \cdot (2S + 1) - 2)!} = \frac{[2 \cdot 2]!}{(2 \cdot 0)!} = 12 \tag{8.4}$$

possible NMR transitions. We write a transition (coherence) from state i to j using the outer product notation $|j\rangle\langle i|$. In **mrsimulator**, all simulations are performed in the high-field limit and further, assume that all spin-spin couplings are in the weak limit.

To write a custom Method in **mrsimulator**, you'll need to determine the desired transition pathways and select the desired transitions during each SpectralEvent or DelayEvent. Keep in mind, however, that Method objects are designed without any details of the spin systems upon which they will act. For example, in the density matrix of a spin system ensemble, one could easily identify a transition by its row and column indexes. However, those indexes depend on the spin system and how the spins and their eigenstates have been assigned to those indexes. Instead, we need a spin-system agnostic approach for selecting transitions.

8.2.1 Spin Transition Symmetry Functions

One way you can select a subset of single-spin transitions if you don't know the energy eigenstate quantum numbers is to request all transitions whose single-spin transition symmetry function, p_I is -1, i.e.,

$$\mathrm{p}_I(m_f, m_i) = m_f - m_i = -1. \tag{8.5}$$

The p_I single-spin transition symmetry function is also known as the single-spin coherence order of the transition.

Note: In the high field limit, only single-spin transitions with $\mathrm{p}_I = \pm 1$ are directly observed. Since the evolution frequencies of the $|j\rangle\langle i|$ and $|i\rangle\langle j|$ transitions are equal in magnitude but opposite in sign, the convention is to only present the $\mathrm{p}_I = -1$ transition resonances in single-quantum spectra.

By selecting only single-spin transitions with $\mathrm{p}_I = -1$, you get all the "observed" transitions from the set of all possible

transitions. Similarly, you can use p_I to select any subset of single-spin transitions, such as double-quantum ($\mathrm{p}_I = \pm 2$) transitions, triple-quantum ($\mathrm{p}_I = \pm 3$) transitions, etc.

While specifying p_I alone is not enough to select an individual single-spin transition, any individual single-spin transition can be identified by a combination of p_I and the single-spin transition symmetry function d_I, given by

$$\mathrm{d}_I(m_i, m_j) = \; m_j^2 - m_i^2. \tag{8.6}$$

You can verify this from the values of p_I and d_I for all single-spin transitions for $I = 1$, $I = 3/2$ and $I = 5/2$ shown below. Note that $\mathrm{d}_I = 0$ for all transitions in a $I = 1/2$ nucleus.

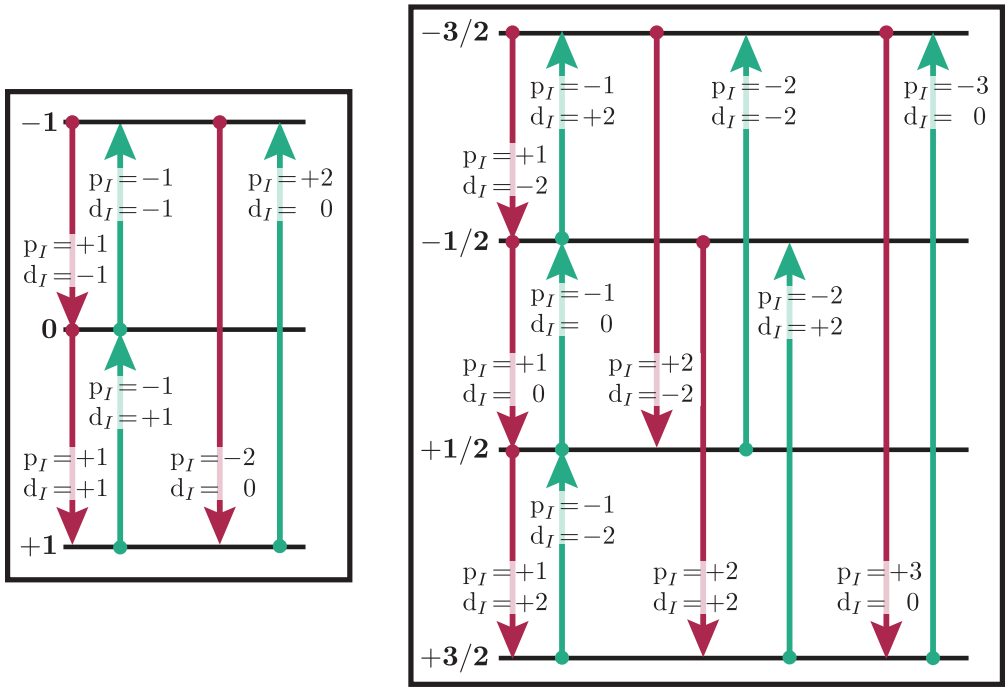


Figure 8.3: Energy level diagrams of a spin $I = 1$ nucleus (left) and spin $I = 3/2$ nucleus (right). Arrows beginning at the initial state and end at the final state represent transitions. Transitions are labeled with their corresponding p_I and d_I transition symmetry function values.

A summary on spin transition symmetry functions in NMR.

Note: In the symmetry pathway approach, the idea of coherence order is extended to form a complete set of spin transition symmetry functions, $\xi_\ell(i, j)$, given by

$$\xi_\ell(i, j) = \langle j|\hat{T}_{\ell,0}|j\rangle - \langle i|\hat{T}_{\ell,0}|i\rangle, \tag{8.7}$$

where the $\hat{T}_{\ell,0}$ are irreducible tensor operators. The function symbol $\xi_\ell(i, j)$ is replaced with the lower-case symbols $\mathfrak{p}(i, j)$, $\mathfrak{d}(i, j)$, $\mathfrak{f}(i, j)$, ..., i.e., we follow the spectroscopic sub-shell letter designations:

$$\begin{array}{lcccccccccccccc} \ell = & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & \leftarrow \text{numerical value} \\ \xi_\ell \equiv & \mathfrak{s} & \mathfrak{p} & \mathfrak{d} & \mathfrak{f} & \mathfrak{g} & \mathfrak{h} & \mathfrak{i} & \mathfrak{k} & \mathfrak{l} & \mathfrak{m} & \mathfrak{o} & \mathfrak{q} & \mathfrak{r} & \mathfrak{t} & \leftarrow \text{symbol} \end{array} \tag{8.8}$$

To simplify usage in figures and discussions, we scale the transition symmetry functions to integers values according

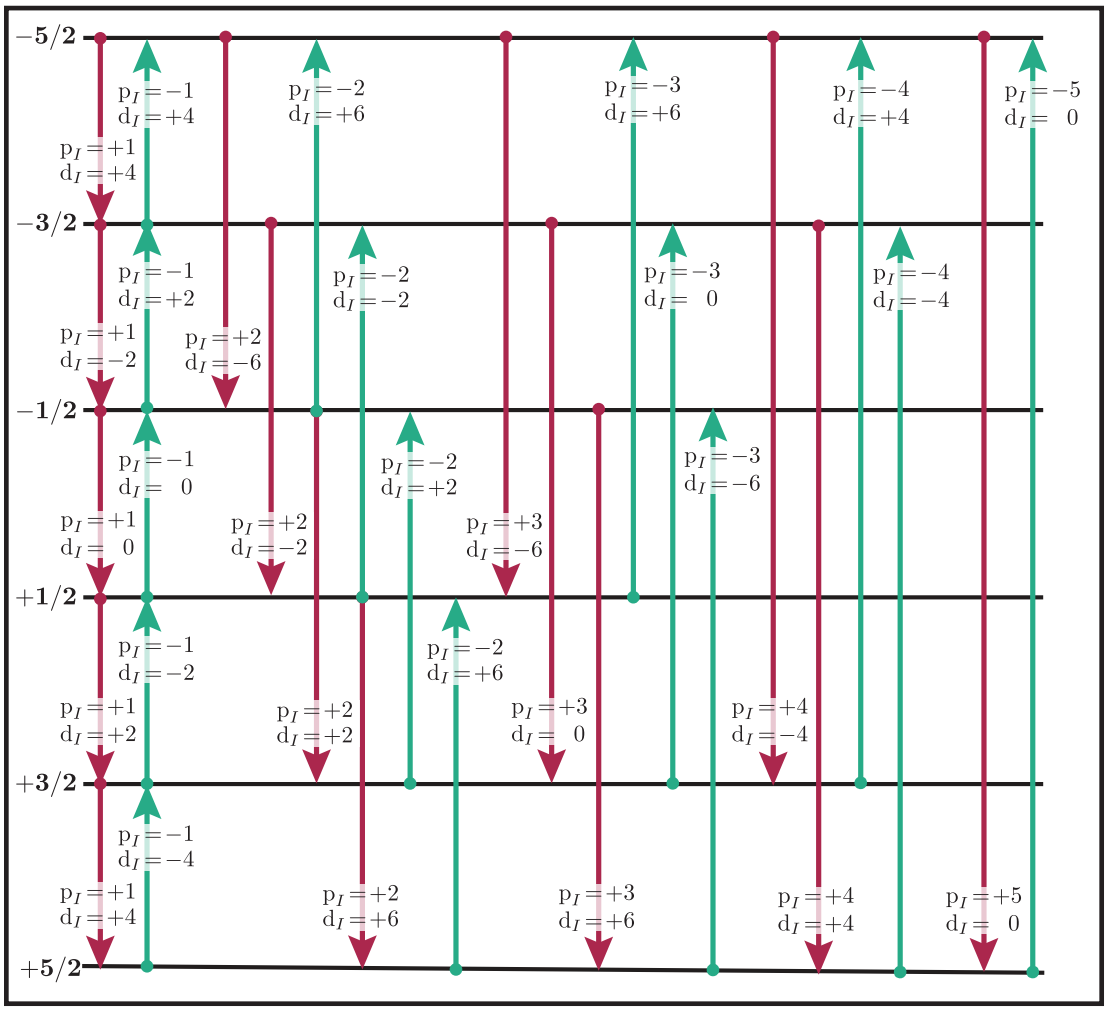


Figure 8.4: Energy level diagram of a spin $I = 5/2$ nucleus. Arrows beginning at the initial state and end at the final state represent transitions. Transitions are labeled with their corresponding p_I and d_I spin transition symmetry function values.

to

$$\mathrm{p}(i,j) = \mathbb{p}(i,j), \qquad \mathrm{d}(i,j) = \sqrt{\frac{2}{3}}\,\mathbb{d}(i,j), \qquad \mathrm{f}(i,j) = \sqrt{\frac{10}{9}}\,\mathbb{f}(i,j), \qquad \cdots \tag{8.9}$$

The $\ell = 0$ function is dropped as it always evaluates to zero. For a single spin, I, a complete set of functions is defined up to $\ell = 2I$.

For weakly coupled nuclei, we define the transition symmetry functions

$$\xi_{\ell_1,\ell_2,\ldots,\ell_n}(i,j) = \langle j|\,\hat{T}_{\ell_1,0}(\mathbf{I}_1)\hat{T}_{\ell_2,0}(\mathbf{I}_2)\ldots\hat{T}_{\ell_n,0}(\mathbf{I}_n)\,|j\rangle - \langle i|\,\hat{T}_{\ell_1,0}(\mathbf{I}_1)\hat{T}_{\ell_2,0}(\mathbf{I}_2)\ldots\hat{T}_{\ell_n,0}(\mathbf{I}_n)\,|i\rangle \tag{8.10}$$

Replacing the symmetry function symbol using sub-shell letter designations becomes more cumbersome in this case. When the ℓ are zero on all nuclei except one, we identify these functions as

$$\begin{aligned}
&\mathbb{p}_1 = \xi_{1,0,\ldots,0}(i,j), \quad \mathbb{p}_2 = \xi_{0,1,\ldots,0}(i,j), \quad \ldots, \quad \mathbb{p}_n = \xi_{0,0,\ldots,1}(i,j),\\
&\mathbb{d}_1 = \xi_{2,0,\ldots,0}(i,j), \quad \mathbb{d}_2 = \xi_{0,2,\ldots,0}(i,j), \quad \ldots, \quad \mathbb{d}_n = \xi_{0,0,\ldots,2}(i,j),\\
&\mathbb{f}_1 = \xi_{3,0,\ldots,0}(i,j), \quad \mathbb{f}_2 = \xi_{0,3,\ldots,0}(i,j), \quad \ldots, \quad \mathbb{f}_n = \xi_{0,0,\ldots,3}(i,j),\\
&\vdots \qquad\qquad \vdots \qquad\qquad \vdots
\end{aligned} \tag{8.11}$$

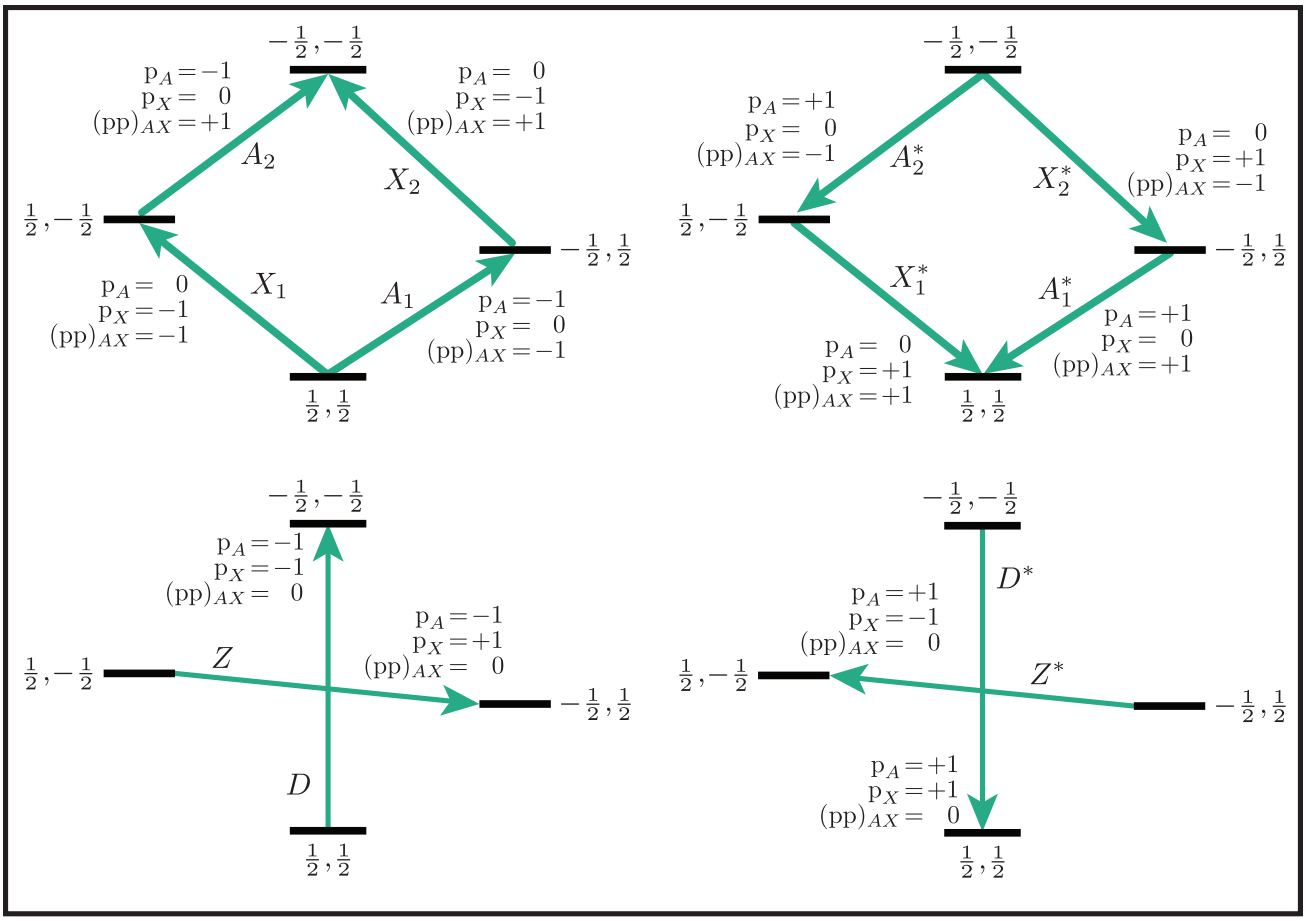
For weakly coupled homonuclear spins it is also convenient to define

$$\begin{aligned}
&\mathbb{p}_{1,2,\ldots,n} = \mathbb{p}_1 + \mathbb{p}_2 + \cdots \mathbb{p}_n,\\
&\mathbb{d}_{1,2,\ldots,n} = \mathbb{d}_1 + \mathbb{d}_2 + \cdots \mathbb{d}_n,\\
&\mathbb{f}_{1,2,\ldots,n} = \mathbb{f}_1 + \mathbb{f}_2 + \cdots \mathbb{f}_n,\\
&\vdots
\end{aligned} \tag{8.12}$$

When the ℓ are zero on all nuclei except two, then we identify these functions using a concatenation of sub-shell letter designations, e.g.,

$$\begin{aligned}
&(\mathbb{pp})_{1,2} = \xi_{1,1,0,\ldots,0}(i,j), \quad (\mathbb{pp})_{1,3} = \xi_{1,0,1,\ldots,0}(i,j), \quad \ldots, \quad (\mathbb{pp})_{1,n} = \xi_{1,0,0,\ldots,1}(i,j),\\
&(\mathbb{pd})_{1,2} = \xi_{1,2,0,\ldots,0}(i,j), \quad (\mathbb{pd})_{1,3} = \xi_{1,0,2\ldots,0}(i,j), \quad \ldots, \quad (\mathbb{pd})_{1,n} = \xi_{1,0,\ldots,2}(i,j),\\
&(\mathbb{dp})_{1,2} = \xi_{2,1,0,\ldots,0}(i,j), \quad (\mathbb{dp})_{1,3} = \xi_{2,0,1\ldots,0}(i,j), \quad \ldots, \quad (\mathbb{dp})_{1,n} = \xi_{2,0,\ldots,1}(i,j),\\
&\vdots \qquad\qquad \vdots \qquad\qquad \vdots
\end{aligned} \tag{8.13}$$

Below is an energy level diagram of two coupled spin $I = 1/2$ nuclei with transition labeled according to their transition symmetry function values. Note that each transition has a unique set of transition symmetry function values.



8.3 Single-Spin Queries

Based on the review above, we now know for the spin $I = 1$, the transition $|-1\rangle\langle 0|$ can be selected with $(p_I, d_I) = (-1, 1)$. In **mrsimulator**, this transition is selected during a SpectralEvent using the SymmetryQuery and TransitionQuery objects, as defined in the code below.

```
from mrsimulator.method.query import SymmetryQuery, TransitionQuery
from mrsimulator.method import SpectralEvent

symm_query = SymmetryQuery(P=[-1], D=[1])
trans_query = TransitionQuery(ch1=symm_query)
spec_event = SpectralEvent(transition_queries=[trans_query])
```

Note: Python dictionaries can also be used to create and initialize **mrsimulator** objects. To do this, the dictionary must use the object's attribute names as the key strings and be passed to a higher level object. Since a SpectralEvent object holds a list of TransitionQuery objects, the above code could have been written as

```
# Both SymmetryQuery and TransitionQuery object as dict
symm_query_dict = {"P": [-1], "D": [1]}
```

(continues on next page)

(continued from previous page)

```
trans_query_dict = {"ch1": symm_query_dict}

# Dictionary of TransitionQuery passed to SpectralEvent
spec_event = SpectralEvent(transition_queries=[trans_query_dict])
```

In the example above, the SymmetryQuery object is created and assigned to the TransitionQuery attribute `ch1`, i.e., it acts on the isotope in the "first channel". Recall that the `channels` attribute of the Method object holds an ordered list of isotope strings. This list's first, second, and third isotopes are associated with `ch1`, `ch2`, and `ch3`, respectively. Currently, **mrsimulator** only supports up to three channels, although this may be increased in future versions.

The TransitionQuery object goes into a list in the `transition_queries` attribute of a SpectralEvent object. The SpectralEvent object, in turn, is added to an ordered list in the `events` attribute of a SpectralDimension object. All this is illustrated in the code below.

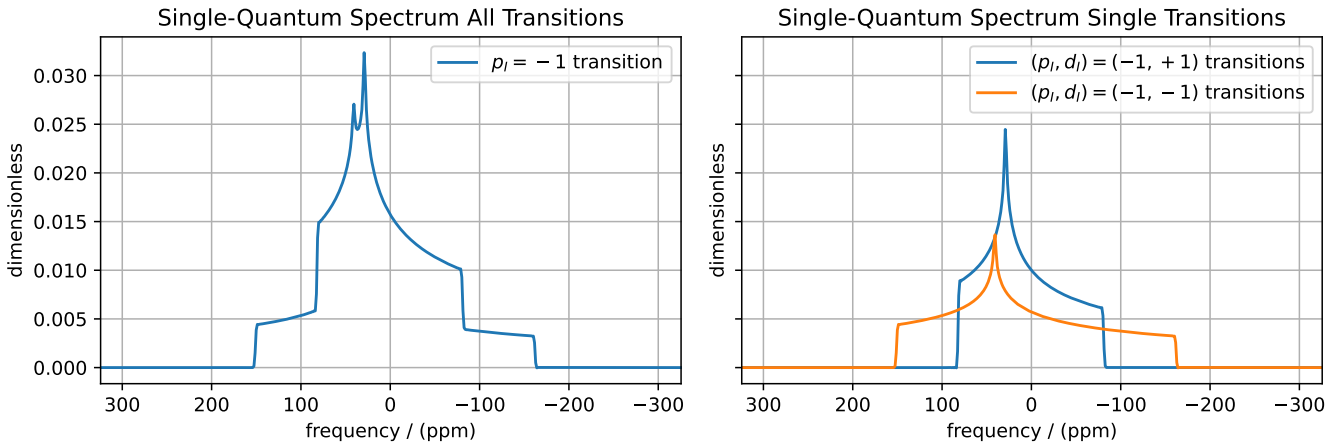
```
from mrsimulator import Site, Coupling, SpinSystem, Simulator
from mrsimulator import Method, SpectralDimension
from mrsimulator import signal_processor as sp
import matplotlib.pyplot as plt
import numpy as np

# Create single Site and Spin System
deuterium = Site(
    isotope="2H",
    isotropic_chemical_shift=10,  # in ppm
    shielding_symmetric={"zeta": -80, "eta": 0.25},  # zeta in ppm
    quadrupolar={"Cq": 10e3, "eta": 0.0, "alpha": 0, "beta": np.pi / 2, "gamma": 0},
)
deuterium_system = SpinSystem(sites=[deuterium])

# This method selects all observable (p_I = -1) transitions
method_both_transitions = Method(
    channels=["2H"],
    magnetic_flux_density=9.4,  # in T
    spectral_dimensions=[
        SpectralDimension(
            count=512,
            spectral_width=40000,  # in Hz
            events=[SpectralEvent(transition_queries=[{"ch1": {"P": [-1]}}])],
        )
    ],
)

# This method selects observable (p_I = -1) transitions with d_I = 1
method_transition1 = Method(
    channels=["2H"],
    magnetic_flux_density=9.4,  # in T
    spectral_dimensions=[
        SpectralDimension(
            count=512,
            spectral_width=40000,  # in Hz
            events=[SpectralEvent(transition_queries=[{"ch1": {"P": [-1], "D": [1]}}])],
```

(continues on next page)

(continued from previous page)

```
        )
    ],
)

# This method selects observable (p_I = -1) transitions with d_I = -1
method_transition2 = Method(
    channels=["2H"],
    magnetic_flux_density=9.4,  # in T
    spectral_dimensions=[
        SpectralDimension(
            count=512,
            spectral_width=40000,  # in Hz
            events=[SpectralEvent(transition_queries=[{"ch1": {"P": [-1], "D": [-1]}}])],
        )
    ],
)
# Simulate spectra for all three method with spin system
sim = Simulator(
    spin_systems=[deuterium_system],
    methods=[method_both_transitions, method_transition1, method_transition2],
)
sim.run()

# Create SignalProcessor for Gaussian Convolution
processor = sp.SignalProcessor(
    operations=[sp.IFFT(), sp.apodization.Gaussian(FWHM="100 Hz"), sp.FFT()]
)
```

```
# Plot spectra from all three methods
fig, ax = plt.subplots(1, 2, figsize=(10, 3.5), subplot_kw={"projection": "csdm"}, sharey=True)
ax[0].plot(
    processor.apply_operations(dataset=sim.methods[0].simulation).real,
    label="$p_I = -1$ transition",
)
ax[0].set_title("Single-Quantum Spectrum All Transitions")
ax[0].legend()
ax[0].grid()
ax[0].invert_xaxis()  # reverse x-axis
ax[1].plot(
    processor.apply_operations(dataset=sim.methods[1].simulation).real,
    label="$(p_I,d_I) = (-1,+1)$ transitions",
)
ax[1].plot(
    processor.apply_operations(dataset=sim.methods[2].simulation).real,
    label="$(p_I,d_I) = (-1,-1)$ transitions",
)
ax[1].set_title("Single-Quantum Spectrum Single Transitions")
ax[1].legend()
ax[1].grid()
ax[1].invert_xaxis()  # reverse x-axis
plt.tight_layout()
plt.show()
```



Note: Whenever the `D` attribute is omitted, the SymmetryQuery allows transitions with all values of d_I. On the other hand, whenever the `P` attribute is omitted, it defaults to `P=[0]`, i.e., no selected transitions on the assigned channel.

8.3.1 Selecting Symmetric Single-Spin Transitions

A notable case, particularly useful for half-integer quadrupolar nuclei, is that $\mathrm{d}_I = 0$ for all symmetric ($m \rightarrow -m$) transitions, as these transitions are unaffected by the first-order quadrupolar coupling frequency contribution. The MQ-MAS experiment involves a 2D correlation of the two symmetric ($\mathrm{d}_I = 0$) transitions, $\left|-\frac{1}{2}\right\rangle\left\langle\frac{1}{2}\right|$, the so-called "central transition," and $\left|-\frac{3}{2}\right\rangle\left\langle\frac{3}{2}\right|$, the symmetric triple quantum transition. The code below is an example of a custom 2D method using two SpectralDimension objects, each holding a single SpectralEvent. The TransitionQuery objects select each transition in their respective SpectralDimension objects.

```
my_mqmas = Method(
    channels=["87Rb"],
    magnetic_flux_density=9.4,
    rotor_frequency=np.inf,  # in Hz (here, set to infinity)
    spectral_dimensions=[
        SpectralDimension(
            count=128,
            spectral_width=6e3,  # in Hz
            reference_offset=-9e3,  # in Hz
            label="Symmetric 3Q Frequency",
            events=[SpectralEvent(transition_queries=[{"ch1": {"P": [-3], "D": [0]}}])],
        ),
        SpectralDimension(
            count=256,
            spectral_width=6e3,  # in Hz
            reference_offset=-5e3,  # in Hz
            label="Central Transition Frequency",
            events=[SpectralEvent(transition_queries=[{"ch1": {"P": [-1], "D": [0]}}])],
        ),
    ],
)
```

(continues on next page)

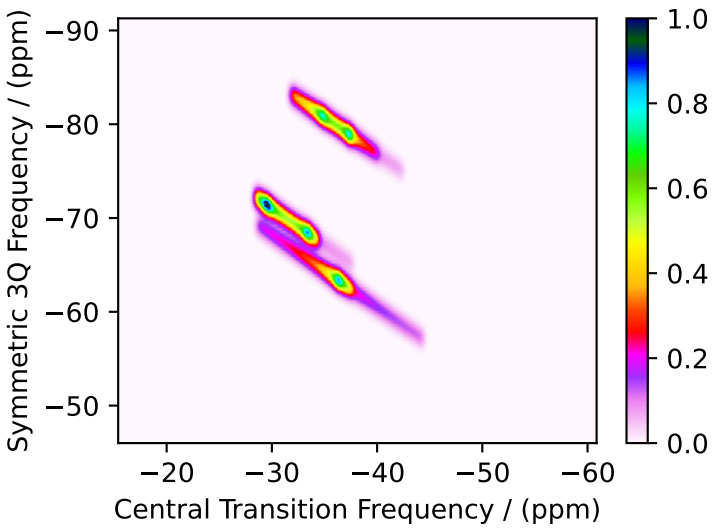
(continued from previous page)

```
# Create three sites in RbNO3
site1 = Site(
    isotope="87Rb",
    isotropic_chemical_shift=-27.4,  # ppm
    quadrupolar={"Cq": 1.68e6, "eta": 0.2},  # Cq in Hz
)
site2 = Site(
    isotope="87Rb",
    isotropic_chemical_shift=-28.5,  # ppm
    quadrupolar={"Cq": 1.94e6, "eta": 1},  # Cq in Hz
)
site3 = Site(
    isotope="87Rb",
    isotropic_chemical_shift=-31.3,  # ppm
    quadrupolar={"Cq": 1.72e6, "eta": 0.5},  # Cq in Hz
)

# No Couplings, so create a separate SpinSystem for each site.
sites = [site1, site2, site3]
RbNO3_spin_systems = [SpinSystem(sites=[s]) for s in sites]

sim = Simulator(spin_systems=RbNO3_spin_systems, methods=[my_mqmas])
sim.run()

# Apply Gaussian line broadening along both dimensions via convolution
gauss_convolve = sp.SignalProcessor(
    operations=[
        sp.IFFT(dim_index=(0, 1)),
        sp.apodization.Gaussian(FWHM="0.08 kHz", dim_index=0),
        sp.apodization.Gaussian(FWHM="0.22 kHz", dim_index=1),
        sp.FFT(dim_index=(0, 1)),
    ]
)
dataset = gauss_convolve.apply_operations(dataset=sim.methods[0].simulation)
```

In the code below, we use the imshow() from the matplotlib.pyplot module to return an image of the dataset on a 2D regular raster. We also use "gist_ncar_r" from matpltolib's included colormaps to map the dataset amplitude to colors; the colorbar() function provides the visualization of the dataset mapping to color to the right of the plot.

```
plt.figure(figsize=(4, 3))
ax = plt.subplot(projection="csdm")
cb = ax.imshow(dataset.real / dataset.real.max(), aspect="auto", cmap="gist_ncar_r")
plt.colorbar(cb)
ax.invert_xaxis()
ax.invert_yaxis()
plt.tight_layout()
plt.show()
```

Warning: This custom method, as well as the built-in Multi-Quantum VAS methods, assumes uniform excitation and mixing of the multiple-quantum transition. In an experimental MQ-MAS measurement, excitation and mixing



efficiencies depend on the ratio of the quadrupolar coupling constant to the rf field strength. Therefore, the relative integrated intensities of this simulation may not agree with the experiment.

8.3.2 Inspecting Transition and Symmetry Pathways

You can view the symmetry pathways that will be selected by your custom method in a given spin system using the function *get_symmetry_pathways()* (page 405) as shown below.

```
from pprint import pprint
pprint(my_mqmas.get_symmetry_pathways("P"))
pprint(my_mqmas.get_symmetry_pathways("D"))
```

Out:

```
[SymmetryPathway(
    ch1(87Rb): [-3] ⟶ [-1]
    total: -3.0 ⟶ -1.0
)]
[SymmetryPathway(
    ch1(87Rb): [0] ⟶ [0]
    total: 0.0 ⟶ 0.0
)]
```

Similarly, you can view the transition pathway that will be selected by your custom method in a given spin system using the function *get_transition_pathways()* (page 407) as shown below.

```
from pprint import pprint
pprint(my_mqmas.get_transition_pathways(SpinSystem(sites=[site1])))
```

Out:

```
[|-1.5⟩⟨1.5| ⟶ |-0.5⟩⟨0.5|, weight=(1+0j)]
```

8.4 Multi-Spin Queries

When there is more than one site in a spin system, things get a little more complicated with the SymmetryQuery objects. Here we review some important concepts associated with transition symmetry functions in coupled spin systems, and see how SymmetryQuery objects are designed to work in such cases.

8.4.1 Single-Spin Single-Quantum Transitions

Consider the case of three weakly coupled proton sites. Here, the selection rule for observable transitions is

$$\left.\begin{aligned} \mathrm{p}_A = -1 \text{ while } \mathrm{p}_M = 0, \mathrm{p}_X = 0 \\ \mathrm{p}_M = -1 \text{ while } \mathrm{p}_A = 0, \mathrm{p}_X = 0 \\ \mathrm{p}_X = -1 \text{ while } \mathrm{p}_A = 0, \mathrm{p}_M = 0 \end{aligned}\right\} \text{ Detection Selection Rules.} \tag{8.14}$$

These corresponds to the *single-spin single-quantum transitions* labeled $\hat{A}_1$, $\hat{A}_2$, $\hat{A}_3$, $\hat{A}_4$, $\hat{M}_1$, $\hat{M}_2$, $\hat{M}_3$, $\hat{M}_4$, $\hat{X}_1$, $\hat{X}_2$, $\hat{X}_3$, and $\hat{X}_4$ in the energy level diagram below.

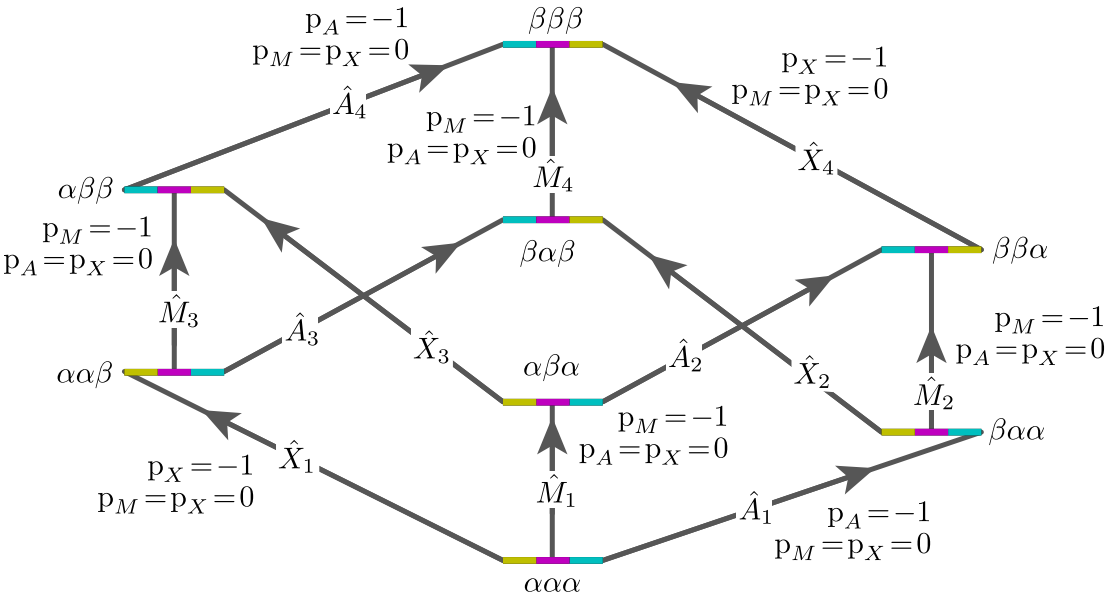


Figure 8.5: Energy level diagram for three coupled spin $I = 1/2$ nuclei. Arrows beginning at the initial state and end at the final state represent the single-spin single-quantum transitions. Transitions are labeled with their corresponding single-spin p_i transition symmetry function values.

Keep in mind that the Method object does not know, in advance, the number of sites in a spin system.

The TransitionQuery for selecting these 12 *single-spin single-quantum* transitions is given in the code below.

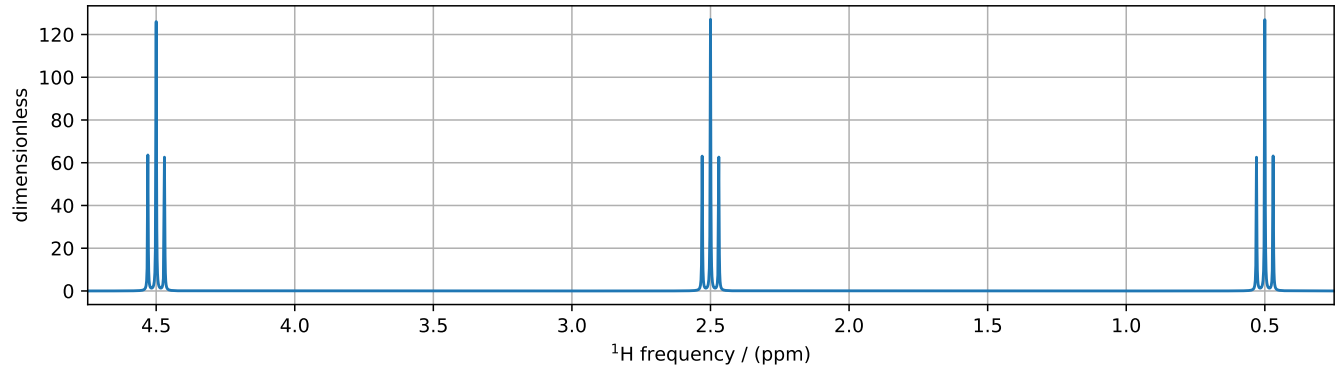
```
# Create Site, Coupling and SpinSystem objects
site_A = Site(isotope="1H", isotropic_chemical_shift=0.5)
site_M = Site(isotope="1H", isotropic_chemical_shift=2.5)
site_X = Site(isotope="1H", isotropic_chemical_shift=4.5)
sites = [site_A, site_M, site_X]
coupling_AM = Coupling(site_index=[0, 1], isotropic_j=12)
```

(continues on next page)

(continued from previous page)

```
coupling_AX = Coupling(site_index=[0, 2], isotropic_j=12)
coupling_MX = Coupling(site_index=[1, 2], isotropic_j=12)
couplings = [coupling_AM, coupling_AX, coupling_MX]
proton_system = SpinSystem(sites=sites, couplings=couplings)

# Custom method object emulating a one-pulse acquire experiment
method = Method(
    channels=["1H"],
    magnetic_flux_density=9.4,  # in T
    spectral_dimensions=[
        SpectralDimension(
            count=16000,
            spectral_width=1800,  # in Hz
            reference_offset=1000,  # in Hz
            label="$^{1}$H frequency",
            events=[SpectralEvent(transition_queries=[{"ch1": {"P": [-1]}}])],
        )
    ],
)

sim = Simulator(spin_systems=[proton_system], methods=[method])
sim.run()

# Add line broadening
processor = sp.SignalProcessor(
    operations=[sp.IFFT(), sp.apodization.Exponential(FWHM="1 Hz"), sp.FFT()]
)
```

```
plt.figure(figsize=(10, 3))  # set the figure size
ax = plt.subplot(projection="csdm")
ax.plot(processor.apply_operations(dataset=sim.methods[0].simulation).real)
ax.invert_xaxis()  # reverse x-axis
plt.tight_layout()
plt.grid()
plt.show()
```



The assignment of transitions in the spectrum above are, from left to right, are $\hat{X}_4, (\hat{X}_3, \hat{X}_2)$, and $\hat{X}_1$ centered at 4.5 ppm, $\hat{M}_4, (\hat{M}_3, \hat{M}_2)$, and $\hat{M}_1$ centered at 2.5 ppm, and $\hat{A}_4, (\hat{A}_3, \hat{A}_2)$, and $\hat{A}_1$ centered at 0.5 ppm.

It is essential to realize that all sites having the same isotope are "indistinguishable" to a TransitionQuery object. Recall that `ch1` is associated with the first isotope in the list of isotope strings assigned to the Method attribute `channels`. When the TransitionQuery above is combined with the SpinSystem object with three ^{1}H Sites, it must first expand its SymmetryQuery into an intermediate set of spin-system-specific symmetry queries, illustrated by each row in the table below.

Transitions	p_A	p_M	p_X
$\hat{A}_1, \hat{A}_2, \hat{A}_3, \hat{A}_4$	–1	0	0
$\hat{M}_1, \hat{M}_2, \hat{M}_3, \hat{M}_4$	0	–1	0
$\hat{X}_1, \hat{X}_2, \hat{X}_3, \hat{X}_4$	0	0	–1

The intermediate spin-system-specific symmetry query in each row selects a subset of transitions from the complete set of transitions. The final set of selected transitions is obtained from the union of transition subsets from each spin-system-specific symmetry query.

The *get_transition_pathways()* (page 407) function will allow you to inspect the transitions selected by the Method in terms of the initial and final Zeeman eigenstate quantum numbers.

```
from pprint import pprint
pprint(method.get_transition_pathways(proton_system))
```

Out:

```
[|-0.5, -0.5, -0.5⟩⟨-0.5, -0.5, 0.5|, weight=(1+0j),
|-0.5, -0.5, -0.5⟩⟨-0.5, 0.5, -0.5|, weight=(1+0j),
|-0.5, -0.5, 0.5⟩⟨-0.5, 0.5, 0.5|, weight=(1+0j),
|-0.5, 0.5, -0.5⟩⟨-0.5, 0.5, 0.5|, weight=(1+0j),
|-0.5, -0.5, -0.5⟩⟨0.5, -0.5, -0.5|, weight=(1+0j),
|-0.5, -0.5, 0.5⟩⟨0.5, -0.5, 0.5|, weight=(1+0j),
|0.5, -0.5, -0.5⟩⟨0.5, -0.5, 0.5|, weight=(1+0j),
|-0.5, 0.5, -0.5⟩⟨0.5, 0.5, -0.5|, weight=(1+0j),
|0.5, -0.5, -0.5⟩⟨0.5, 0.5, -0.5|, weight=(1+0j),
|-0.5, 0.5, 0.5⟩⟨0.5, 0.5, 0.5|, weight=(1+0j),
|0.5, -0.5, 0.5⟩⟨0.5, 0.5, 0.5|, weight=(1+0j),
|0.5, 0.5, -0.5⟩⟨0.5, 0.5, 0.5|, weight=(1+0j)]
```

To further illustrate how the TransitionQuery and SymmetryQuery objects work in a multi-site spin system, let's examine a few more examples in the case of three weakly coupled proton sites.

8.4.2 Two-Spin Double-Quantum Transitions

In this spin system there are six *two-spin double-quantum transitions* where $p_{AMX} = p_A + p_M + p_X = -2$ and another six *two-spin double-quantum transitions* where $p_{AMX} = p_A + p_M + p_X = +2$. The $p_{AMX} = -2$ transitions are illustrated in the energy-level diagram below.

The code below will select the six *two-spin double-quantum transitions* where $p_{AMX} = -2$.

```
method = Method(
    channels=["1H"],
    magnetic_flux_density=9.4,  # in T
```

(continues on next page)

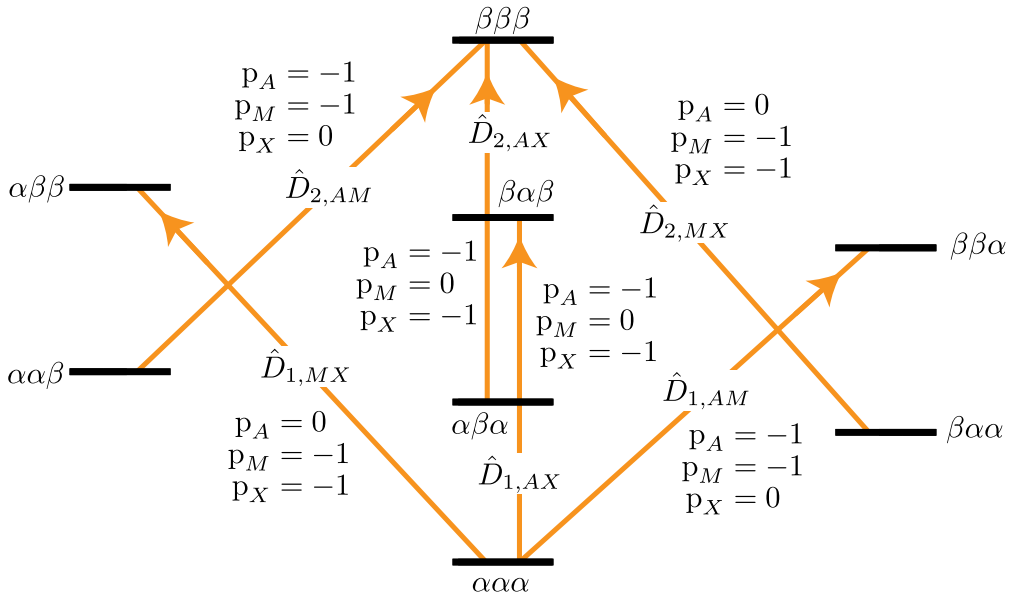


Figure 8.6: Energy level diagram for three coupled spin $I = 1/2$ nuclei. Arrows beginning at the initial state and end at the final state represent the two-spin double-quantum transitions. Transitions are labeled with their corresponding single-spin p_i transition symmetry function values.

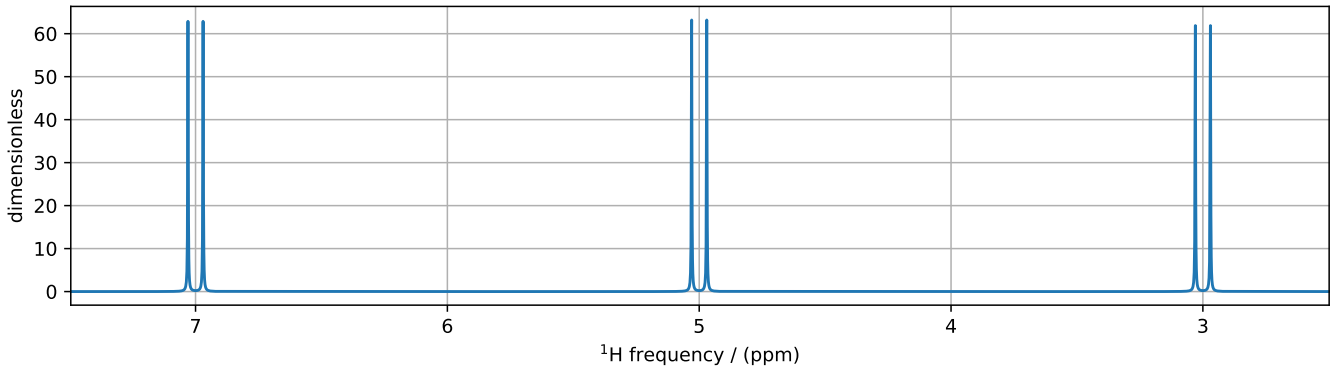
(continued from previous page)

```
    spectral_dimensions=[
        SpectralDimension(
            count=16000,
            spectral_width=2000,  # in Hz
            reference_offset=2000,  # in Hz
            label="$^{1}$H frequency",
            events=[SpectralEvent(transition_queries=[{"ch1": {"P": [-1, -1]}}])],
        )
    ],
)

sim = Simulator(spin_systems=[proton_system], methods=[method])
sim.run()
```

```
plt.figure(figsize=(10, 3))  # set the figure size
ax = plt.subplot(projection="csdm")
ax.plot(processor.apply_operations(dataset=sim.methods[0].simulation).real)
ax.invert_xaxis()  # reverse x-axis
plt.tight_layout()
plt.grid()
plt.show()
```

The assignment of transitions in the spectrum above are, from left to right, are $\hat{D}_{2,MX}$, $\hat{D}_{1,MX}$, $\hat{D}_{2,AX}$, $\hat{D}_{1,AX}$, $\hat{D}_{2,AM}$, and $\hat{D}_{1,AM}$,

As before, when this generic TransitionQuery is combined with the three-site SpinSystem object, the SymmetryQuery is expanded into an intermediate set of spin-system-specific symmetry queries illustrated in the table below.



Transitions	p_A	p_M	p_X
$\hat{D}_{1,AM}, \hat{D}_{2,AM}$	–1	–1	0
$\hat{D}_{1,MX}, \hat{D}_{2,MX}$	0	–1	–1
$\hat{D}_{1,AX}, \hat{D}_{2,AX}$	–1	0	–1

Again, each row's intermediate spin-system-specific symmetry query is used to select a subset of transitions from the complete set of transitions. The final set of selected transitions is obtained from the union of transition subsets from each spin-system-specific symmetry query.

```
from pprint import pprint
pprint(method.get_transition_pathways(proton_system))
```

Out:

```
[|-0.5, -0.5, -0.5⟩⟨-0.5, 0.5, 0.5|, weight=(1+0j),
|-0.5, -0.5, -0.5⟩⟨0.5, -0.5, 0.5|, weight=(1+0j),
|-0.5, -0.5, -0.5⟩⟨0.5, 0.5, -0.5|, weight=(1+0j),
|-0.5, -0.5, 0.5⟩⟨0.5, 0.5, 0.5|, weight=(1+0j),
|-0.5, 0.5, -0.5⟩⟨0.5, 0.5, 0.5|, weight=(1+0j),
|0.5, -0.5, -0.5⟩⟨0.5, 0.5, 0.5|, weight=(1+0j)]
```

8.4.3 Three-Spin Single-Quantum Transitions

Another interesting example in this spin system with three weakly coupled proton sites are the three *three-spin single-quantum transitions* having $p_{AMX} = p_A + p_M + p_X = -1$ and the three *three-spin single-quantum transitions* having $p_{AMX} = p_A + p_M + p_X = +1$

The three *three-spin single-quantum transitions* having $p_{AMX} = -1$ are illustrated in the energy level diagram below.

The code below will select these *three-spin single-quantum transitions*.

```
method = Method(
    channels=["1H"],
    magnetic_flux_density=9.4,  # in T
    spectral_dimensions=[
```

(continues on next page)

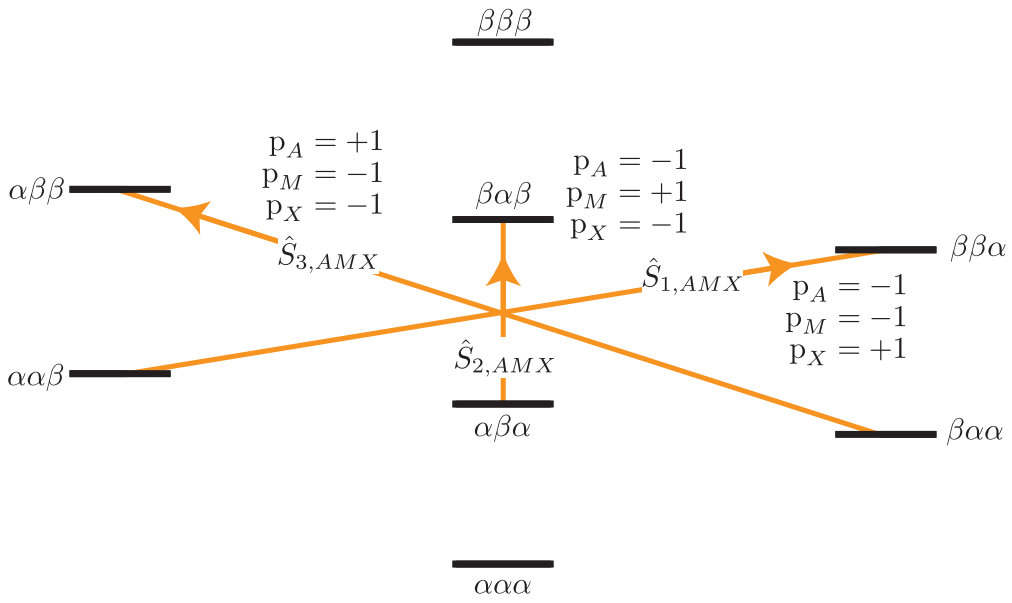


Figure 8.7: Energy level diagram for three coupled spin $I = 1/2$ nuclei. Arrows beginning at the initial state and end at the final state represent the three spin single-quantum transitions. Transitions are labeled with their corresponding single-spin p_i transition symmetry function values.

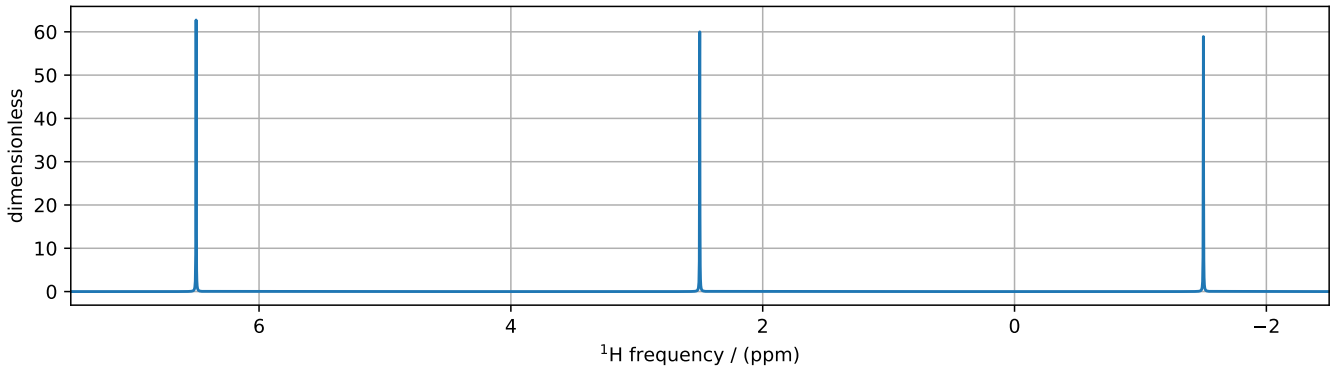
(continued from previous page)

```
        SpectralDimension(
            count=16000,
            spectral_width=4000,  # in Hz
            reference_offset=1000,  # in Hz
            label="$^{1}$H frequency",
            events=[SpectralEvent(transition_queries=[{"ch1": {"P": [-1, -1, +1]}}])]
        )
    ],
)

sim = Simulator(spin_systems=[proton_system], methods=[method])
sim.run()
```

```
plt.figure(figsize=(10, 3))  # set the figure size
ax = plt.subplot(projection="csdm")
ax.plot(processor.apply_operations(dataset=sim.methods[0].simulation).real)
ax.invert_xaxis()  # reverse x-axis
plt.tight_layout()
plt.grid()
plt.show()
```

The assignment of transitions in the spectrum above are, from left to right, are $\hat{S}_{3,AMX}$, $\hat{S}_{2,AMX}$, and $\hat{S}_{1,AMX}$

Again, combined with the three-site SpinSystem object, the SymmetryQuery is expanded into the set of spin-system-specific symmetry queries illustrated in the table below.



Transitions	p_A	p_M	p_X
$\hat{S}_{1,AMX}$	−1	+1	−1
$\hat{S}_{2,AMX}$	−1	−1	+1
$\hat{S}_{3,AMX}$	+1	−1	−1

```
from pprint import pprint
pprint(method.get_transition_pathways(proton_system))
```

Out:

```
[|0.5, -0.5, -0.5⟩⟨-0.5, 0.5, 0.5|, weight=(1+0j),
|-0.5, 0.5, -0.5⟩⟨0.5, -0.5, 0.5|, weight=(1+0j),
|-0.5, -0.5, 0.5⟩⟨0.5, 0.5, -0.5|, weight=(1+0j)]
```

As you can surmise from the examples, the attributes of SymmetryQuery, `P` and `D`, hold a list of single-spin transition symmetry function values, and the length of the list is the desired number of spins that are involved in the transition.

8.4.4 Heteronuclear multiple-spin transitions

How does `D` fit into the multi-site SymmetryQuery story? Consider the case of two coupled hydrogens, except we replace one of the ^1H with ^2H. Let's focus on the single-spin single-quantum transitions, shown below as $\hat{A}_{1\pm}$ and $\hat{A}_{2\pm}$ on the left, and the two-spin triple-quantum transition, shown below as $\hat{T}_{AX}$ on the right.

```
from mrsimulator.spin_system.tensors import SymmetricTensor
import numpy as np

site_A = Site(
    isotope="2H",
    isotropic_chemical_shift=0.5,
    quadrupolar=SymmetricTensor(
        Cq=100000,  # in Hz
        eta=0.2,
        alpha=5 * np.pi / 180,
        beta=np.pi / 2,
```

(continues on next page)

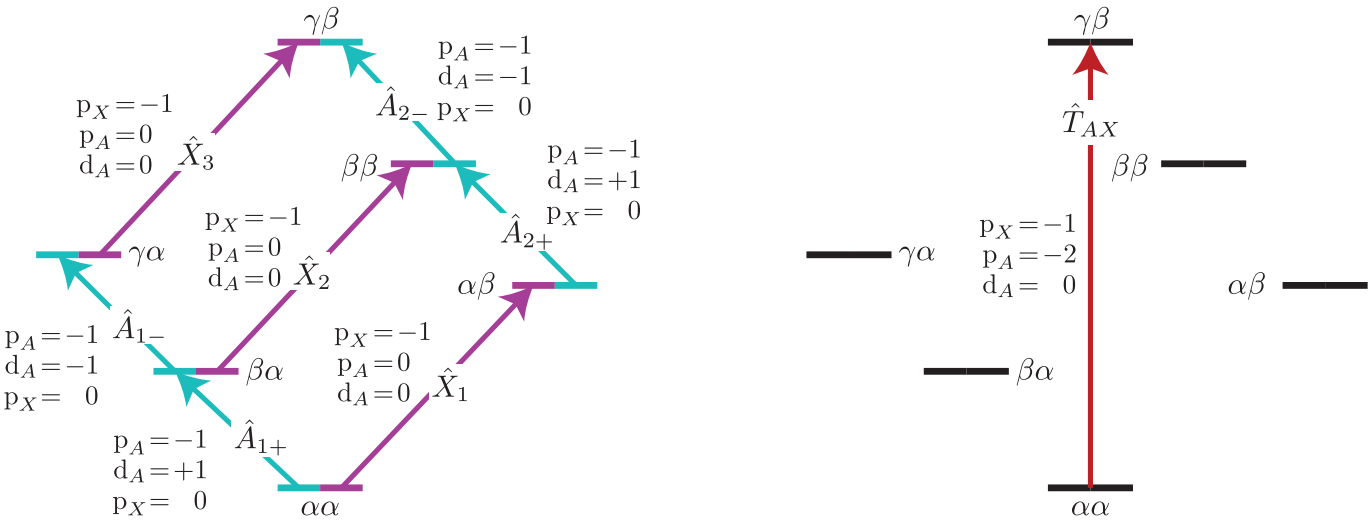


Figure 8.8: Energy level diagram for two coupled nuclei with spins $I = 1/2$ and $I = 1$. Arrows beginning at the initial state and end at the final state represent the single spin single-quantum transitions (left) and the three-spin triple-quantum transition. Transitions are labeled with their corresponding single-spin p_i transition symmetry function values.

(continued from previous page)

```
        gamma=70 * np.pi / 180,
    ),
)
site_X = Site(isotope="1H", isotropic_chemical_shift=4.5)
sites = [site_A, site_X]
coupling_AX = Coupling(site_index=[0, 1], dipolar={"D": -20000})
couplings = [coupling_AX]
system_AX = SpinSystem(sites=sites, couplings=couplings)

methodAll1Q = Method(
    channels=["2H", "1H"],
    magnetic_flux_density=9.4,  # in T
    spectral_dimensions=[
        SpectralDimension(
            count=16000,
            spectral_width=200000,  # in Hz
            reference_offset=0,  # in Hz
            label="$^{2}$H frequency",
            events=[SpectralEvent(transition_queries=[{"ch1": {"P": [-1]}}])],
        )
    ],
)

methodHalf1Q = Method(
    channels=["2H", "1H"],
    magnetic_flux_density=9.4,  # in T
    spectral_dimensions=[
        SpectralDimension(
            count=16000,
```

(continues on next page)

(continued from previous page)

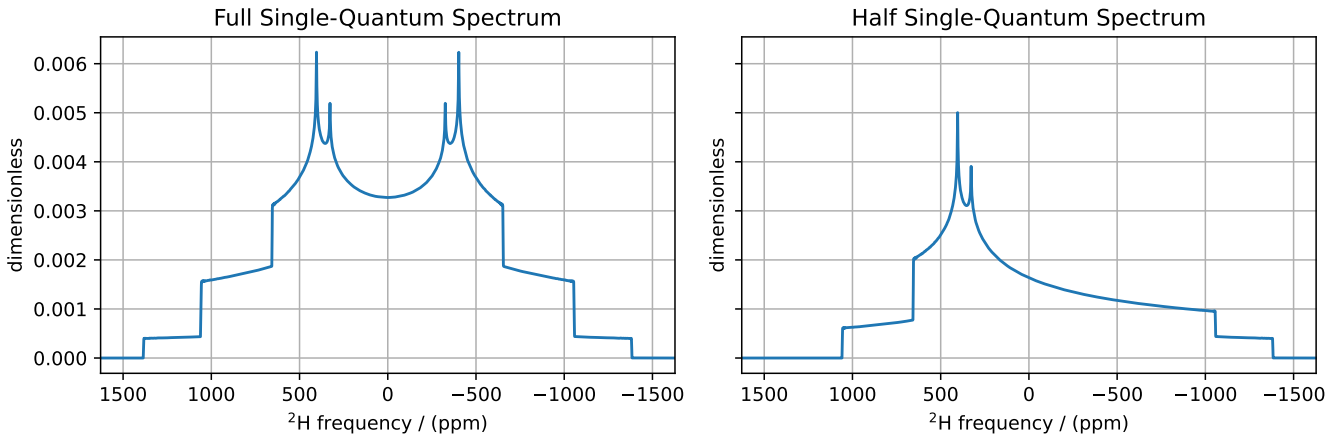
```
            spectral_width=200000,  # in Hz
            reference_offset=0,  # in Hz
            label="$^{2}$H frequency",
            events=[SpectralEvent(transition_queries=[{"ch1": {"P": [-1], "D": [-1]}}])],
        )
    ],
)

method3Q = Method(
    channels=["2H", "1H"],
    magnetic_flux_density=9.4,  # in T
    spectral_dimensions=[
        SpectralDimension(
            count=16000,
            spectral_width=10000,  # in Hz
            reference_offset=5000,  # in Hz
            label="$^{2}$H frequency",
            events=[
                SpectralEvent(
                    transition_queries=[{"ch1": {"P": [-2]}, "ch2": {"P": [-1]}}]
                )
            ],
        )
    ],
)
processor = sp.SignalProcessor(
    operations=[sp.IFFT(), sp.apodization.Gaussian(FWHM="100 Hz"), sp.FFT()]
)

sim = Simulator(spin_systems=[system_AX], methods=[methodAll1Q, methodHalf1Q, method3Q])
sim.config.integration_volume = "hemisphere"
sim.run()
```

```
fig, ax = plt.subplots(1, 2, figsize=(10, 3.5), subplot_kw={"projection": "csdm"}, sharey=True)
ax[0].plot(processor.apply_operations(dataset=sim.methods[0].simulation).real)
ax[0].set_title("Full Single-Quantum Spectrum")
ax[0].grid()
ax[0].invert_xaxis()  # reverse x-axis
ax[1].plot(processor.apply_operations(dataset=sim.methods[1].simulation).real)
ax[1].set_title("Half Single-Quantum Spectrum")
ax[1].grid()
ax[1].invert_xaxis()  # reverse x-axis
plt.tight_layout()
plt.show()
```

The deuterium spectrum of a static-polycrystalline sample is shown on the left for all single-spin single-quantum transitions on deuterium, $\hat{A}_{1\pm}$ and $\hat{A}_{2\pm}$. The spectrum on the right is for half of the single-spin single-quantum transitions on deuterium: $\hat{A}_{1-}$ and $\hat{A}_{2-}$.

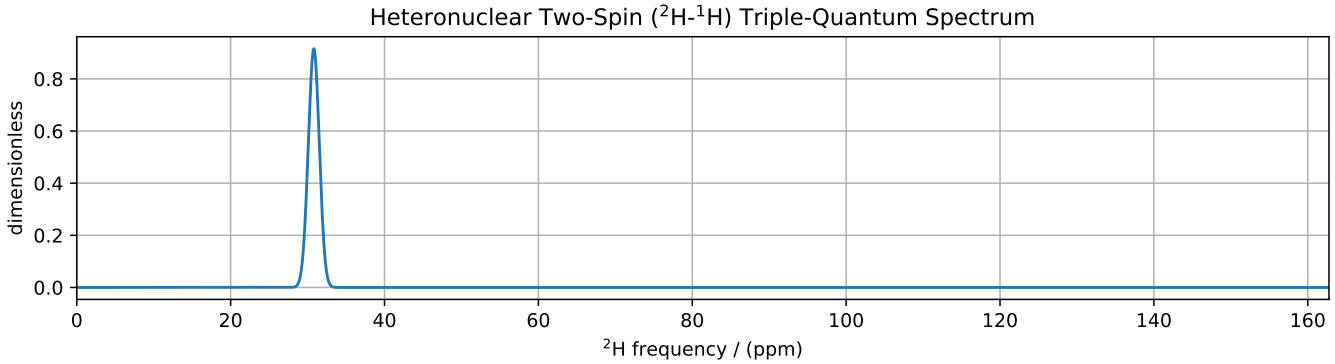
```
plt.figure(figsize=(10, 3))  # set the figure size
ax = plt.subplot(projection="csdm")
ax.set_title("Heteronuclear Two-Spin ($^2$H-$^1$H) Triple-Quantum Spectrum")
```

(continues on next page)



(continued from previous page)

```
ax.plot(processor.apply_operations(dataset=sim.methods[2].simulation).real)
plt.tight_layout()
plt.grid()
plt.show()
```



Transitions	p_A	d_A	p_X
$\hat{T}_{AX}$	–2	0	–1

The single transition in the heteronuclear two-spin (^{2}H-^{1}H) triple-quantum spectrum is unaffected by the dipolar and quadrupolar frequency anisotropies.

8.5 Frequency Contributions

The NMR frequency, $\Omega(\Theta, i, j)$, of an $i \rightarrow j$ transition between the eigenstates of the stationary-state semi-classical Hamiltonian in a sample with a lattice spatial orientation, Θ, can be written as a sum of components,

$$\Omega(\Theta, i, j) = \sum_k \Omega_k(\Theta, i, j) \tag{8.15}$$

with each component, $\Omega_k(\Theta, i, j)$, separated into three parts:

$$\Omega_k(\Theta, i, j) = \omega_k \, \Xi_L^{(k)}(\Theta) \, \xi_\ell^{(k)}(i, j), \tag{8.16}$$

where $\xi_\ell^{(k)}(i, j)$ are the spin transition symmetry functions described earlier, $\Xi_L^{(k)}(\Theta)$ are the spatial symmetry functions, and ω_k gives the size of the kth frequency component. The experimentalist indirectly influences a frequency component Ω_k by direct manipulation of the quantum transition, $i \rightarrow j$, and the spatial orientation, Θ, of the sample.

The function symbol $\Xi_\ell(\Theta)$ is replaced with the upper-case symbols $\mathbb{S}$, $\mathbb{P}(\Theta)$, $\mathbb{D}(\Theta)$, $\mathbb{F}(\Theta)$, $\mathbb{G}(\Theta)$, ..., i.e., following the spectroscopic sub-shell letter designations for L. Consult the Symmetry Pathways paper for more details on the form of the spatial symmetry functions. In short, the $\mathbb{S}$ function is independent of sample orientation, i.e., it will appear in all isotropic frequency contributions. The $\mathbb{D}(\Theta)$ function has a second-rank dependence on sample orientation, and can be averaged away with fast magic-angle spinning, i.e., spinning about an angle, θ_R, that is the root of the second-rank Legendre polynomial $P_2(\cos\theta_R)$. The other spatial symmetry functions are removed by spinning the sample about the corresponding root of the L^{th}-rank Legendre polynomial $P_L(\cos\theta_R)$

Note: For 2nd-order quadrupolar coupling contributions, it is convenient to define "hybrid" spin transition functions as linear combinations of the spin transition functions

$$\mathbb{c}_0 = \frac{4}{\sqrt{125}}\left[I(I+1) - 3/4\right]\mathbb{p}_I + \sqrt{\frac{18}{25}}\,\mathbb{f}_I \tag{8.17}$$

$$\mathbb{c}_2 = \frac{2}{\sqrt{175}}\left[I(I+1) - 3/4\right]\mathbb{p}_I - \frac{6}{\sqrt{35}}\,\mathbb{f}_I \tag{8.18}$$

$$\mathbb{c}_4 = -\frac{184}{\sqrt{875}}\left[I(I+1) - 3/4\right]\mathbb{p}_I - \frac{17}{\sqrt{175}}\,\mathbb{f}_I \tag{8.19}$$

These transition symmetry functions play an essential role in evaluating the individual frequency contributions to the overall transition frequency, given in the table below and in *FrequencyEnum()* (page 416). They also aid in pulse sequence design by identifying how different frequency contributions refocus through the transition pathways.

A summary on echo symmetry classification in NMR.

Note: The well-known Hahn-echo can occur whenever the p_I values of transitions in a transition pathway change sign. This is because the changing sign of p_I leads to a sign change for every p_I-dependent transition frequency contribution. Thus, a Hahn echo forms whenever

$$\overline{\mathrm{p}_I} = \frac{1}{t}\int_0^t \mathrm{p}_I(t')\,dt' = 0, \tag{8.20}$$

assuming a frequency contribution's spatial symmetry function, Ξ, remains constant during this period. As seen in the table in the *Frequency Contributions* (page 104) table, sign changes in other symmetry functions can also lead to corresponding sign changes for dependent frequency contributions. Thus, a problem with showing only the p_I symmetry pathway for an NMR method is that it does not explain the formation of other classes of echoes that result when other symmetry functions change sign in a transition pathway. To fully understand when and which frequency contributions

refocus into echoes, we must follow *all* relevant spatial, transition, or spatial-transition product symmetries through an NMR experiment. Thus, we generally classify echoes that refocus during a time interval as a *transition symmetry echo* (at constant Ξ_k) when

$$\overline{\xi_k} = \frac{1}{t}\int_0^t \xi_k(t')\,dt' = 0, \tag{8.21}$$

and as a *spatial symmetry echo* (at constant ξ_k) when

$$\overline{\Xi_k} = \frac{1}{t}\int_0^t \Xi_k(t')\,dt' = 0, \tag{8.22}$$

and as a *spatial-transition symmetry product* echo when

$$\overline{\Xi_k\xi_k} = \frac{1}{t}\int_0^t \Xi_k(t')\,\xi_k(t')\,dt' = 0. \tag{8.23}$$

Within the class of transition echoes we find subclasses such as p echoes, which include the Hahn echo and the stimulated echo; d echoes, which include the solid echo and Solomon echoes, c_4 echoes, used in MQ-MAS and Satellite-Transition Magic-Angle Spinning (ST-MAS); c_2 echoes, used in Correlation Of Anisotropies Separated Through Echo Refocusing (COASTER); and c_0 echoes, used in Multiple-Quantum DOuble Rotation (MQ-DOR).

Within the class of spatial echoes we find subclasses such as $\mathbb{D}$ rotary echoes, which occur during sample rotation, and $\mathbb{D}_0$ and $\mathbb{G}_0$ echoes, which are designed to occur simultaneously during the Dynamic-Angle Spinning (DAS) experiment.

Table 8.1: Frequency Contributions

Interactions	perturbation order	anisotropy rank	freq_contrib	Expression
shielding	1st	0th	Shielding1_0	$-\omega_0\sigma_{\text{iso}} \cdot \mathbb{p}_I$
shielding	1st	2nd	Shielding1_2	$-\omega_0\zeta_\sigma \cdot \mathbb{D}^{\{\sigma\}} \cdot \mathbb{p}_I$
weak J	1st	0th	J1_0	$2\pi J_{\text{iso}} \, (\mathbb{pp})_{IS}$
weak J	1st	2nd	J1_2	$2\pi\zeta_J \cdot \mathbb{D}^{\{d_{IS}\}} \cdot (\mathbb{pp})_{IS}$
weak dipolar	1st	2nd	D1_2	$\omega_d \cdot \mathbb{D}^{\{d_{IS}\}} \cdot (\mathbb{pp})_{IS}$
quadrupolar	1st	2nd	Quad1_2	$\omega_q \cdot \mathbb{D}^{\{q\}} \cdot \mathbb{d}_I$
quadrupolar	2nd	0th	Quad2_0	$\frac{\omega_q^2}{\omega_0} \cdot \mathbb{S}^{\{qq\}} \cdot \mathbb{c}_0$
quadrupolar	2nd	2nd	Quad2_2	$\frac{\omega_q^2}{\omega_0} \cdot \mathbb{D}^{\{qq\}} \cdot \mathbb{c}_2$
quadrupolar	2nd	4th	Quad2_4	$\frac{\omega_q^2}{\omega_0} \cdot \mathbb{G}^{\{qq\}} \cdot \mathbb{c}_4$
quadrupolar-shielding	2nd	0th	Quad_Shielding_cross_0	$-\zeta_\sigma\omega_q \cdot \mathbb{S}^{\{\sigma q\}} \cdot \mathbb{d}_I$
quadrupolar-shielding	2nd	2nd	Quad_Shielding_cross_2	$-\zeta_\sigma\omega_q \cdot \mathbb{D}^{\{\sigma q\}} \cdot \mathbb{d}_I$
quadrupolar-shielding	2nd	4th	Quad_Shielding_cross_4	$-\zeta_\sigma\omega_q \cdot \mathbb{G}^{\{\sigma q\}} \cdot \mathbb{d}_I$
quadrupolar-weak dipole	2nd	0th	Quad_Dipolar_cross_0	$\frac{\omega_d\,\omega_q^{\{I\}}}{\omega_0^{\{I\}}} \cdot \mathbb{S}^{\{dq_I\}} \cdot (\mathbb{dp})_{IS}$
quadrupolar-weak dipole	2nd	2nd	Quad_Dipolar_cross_2	$\frac{\omega_d\,\omega_q^{\{I\}}}{\omega_0^{\{I\}}} \cdot \mathbb{D}^{\{dq_I\}} \cdot (\mathbb{dp})_{IS}$
quadrupolar-weak dipole	2nd	4th	Quad_Dipolar_cross_4	$\frac{\omega_d\,\omega_q^{\{I\}}}{\omega_0^{\{I\}}} \cdot \mathbb{G}^{\{dq_I\}}(\Theta) \cdot (\mathbb{dp})_{IS}$
quadrupolar-weak dipole	2nd	0th	Quad_Dipolar_cross_0	$\frac{\omega_d\,\omega_q^{\{S\}}}{\omega_0^{\{S\}}} \cdot \mathbb{S}^{\{dq_S\}} \cdot (\mathbb{pd})_{IS}$
quadrupolar-weak dipole	2nd	2nd	Quad_Dipolar_cross_2	$\frac{\omega_d\,\omega_q^{\{S\}}}{\omega_0^{\{S\}}} \cdot \mathbb{D}^{\{dq_S\}} \cdot (\mathbb{pd})_{IS}$
quadrupolar-weak dipole	2nd	4th	Quad_Dipolar_cross_4	$\frac{\omega_d\,\omega_q^{\{S\}}}{\omega_0^{\{S\}}} \cdot \mathbb{G}^{\{dq_S\}}(\Theta) \cdot (\mathbb{pd})_{IS}$
quadrupolar-weak J	2nd	0th	Quad_J_cross_0	$\frac{2\pi\zeta_J\omega_q^{\{I\}}}{\omega_0^{\{I\}}} \cdot \mathbb{S}^{\{Jq_I\}} \cdot (\mathbb{dp})_{IS}$
quadrupolar-weak J	2nd	2nd	Quad_J_cross_2	$\frac{2\pi\zeta_J\omega_q^{\{I\}}}{\omega_0^{\{I\}}} \cdot \mathbb{D}^{\{Jq_I\}} \cdot (\mathbb{dp})_{IS}$
quadrupolar-weak J	2nd	4th	Quad_J_cross_4	$\frac{2\pi\zeta_J\omega_q^{\{I\}}}{\omega_0^{\{I\}}} \cdot \mathbb{G}^{\{Jq_I\}} \cdot$

8.6 Affine Transformations

The ability to refocus different spatial and transition symmetries into echoes with different paths in time-resolved NMR experiments creates opportunities for generating multi-dimensional spectra that correlate different interactions. These spectra can be made easier to interpret through similarity transformations. Most similarity transformations in NMR are affine transformations, as they preserve the colinearity of points and ratios of distances. Essential in any similarity transformation is whether to implement the transformation actively or passively. Active transformations change the appearance of the signal while leaving the coordinate system unchanged, whereas passive transformations leave the appearance of the signal unchanged while changing the coordinate system. Both active and passive transformations are used extensively in NMR.

The general form of the affine transformation of a n-dimensional spectrum is

$$\mathbf{\Omega}' = \mathcal{A}\mathbf{\Omega} \tag{8.24}$$

In the two-dimensional case, this is given by

$$\begin{bmatrix} \Omega'^{[1]} \\ \Omega'^{[2]} \end{bmatrix} = \underbrace{\begin{bmatrix} a & b \\ c & d \end{bmatrix}}_{\mathcal{A}} \begin{bmatrix} \Omega^{[1]} \\ \Omega^{[2]} \end{bmatrix} \tag{8.25}$$

Note: For the multiple-quantum MAS experiment, a shear and scale transformation is often applied to the spectrum to create a 2D spectrum correlating the MQ-MAS isotropic frequency to the anisotropic central transition frequency. This correlation can be achieved by adding an affine matrix to the method.

For 3Q-MAS on a spin $I = 3/2$ nucleus, where the shear factor is $\kappa^{(\omega_2)} = 21/27$, the affine matrix giving the appropriate shear and scale transformation is given by

$$\mathcal{A}_2 = \begin{bmatrix} \frac{1}{1+|\kappa^{(\omega_2)}|} & \frac{\kappa^{(\omega_2)}}{1+|\kappa^{(\omega_2)}|} \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 9/16 & 7/16 \\ 0 & 1 \end{bmatrix} \tag{8.26}$$

After the affine transformation, the position of the resonance in the isotropic projection is a weighted average of the multiple quantum and central transition isotropic frequencies given by

$$\langle \Omega_{\text{iso}} \rangle_{\text{MQ-MAS}} = \frac{1}{1+|\kappa^{(\omega_1)}|}\, \Omega_{\text{iso}}(m, -m) + \frac{\kappa^{(\omega_1)}}{1+|\kappa^{(\omega_1)}|}\, \Omega_{\text{iso}}\left(\tfrac{1}{2}, -\tfrac{1}{2}\right). \tag{8.27}$$

If the spectrum is to be referenced to a frequency other than the rf carrier frequency (i.e. zero is not defined in the middle of the spectrum), then the reference offset used in the single-quantum dimension must be multiplied by a factor of $\left(\mathrm{p}_I^{[1]}/\mathrm{p}_I^{[2]} + |\kappa^{(\omega_1)}|\right)/(1+|\kappa^{(\omega_1)}|)$ when used in the isotropic dimension.

See the "Symmetry Pathways in Solid-State NMR" paper for a more detailed discussion on affine transformations in NMR.

In the code below, the 3Q-MAS method described earlier is modified to include an affine matrix to perform this shear transformation.

```
my_sheared_mqmas = Method(
    channels=["87Rb"],
    magnetic_flux_density=9.4,
```

(continues on next page)

(continued from previous page)

```
    rotor_frequency=np.inf,  # in Hz (here, set to infinity)
    spectral_dimensions=[
        SpectralDimension(
            count=128,
            spectral_width=6e3,  # in Hz
            reference_offset=-9e3,  # in Hz
            label="3Q-MAS isotropic dimension",
            events=[
                SpectralEvent(transition_queries=[{"ch1": {"P": [-3], "D": [0]}}])
            ],
        ),
        SpectralDimension(
            count=256,
            spectral_width=6e3,  # in Hz
            reference_offset=-5e3,  # in Hz
            label="Central Transition Frequency",
            events=[
                SpectralEvent(transition_queries=[{"ch1": {"P": [-1], "D": [0]}}])
            ],
        ),
    ],
    affine_matrix=[[9 / 16, 7 / 16], [0, 1]],
)

sim = Simulator(spin_systems=RbNO3_spin_systems, methods=[my_sheared_mqmas])
sim.run()

dataset = gauss_convolve.apply_operations(dataset=sim.methods[0].simulation)
```

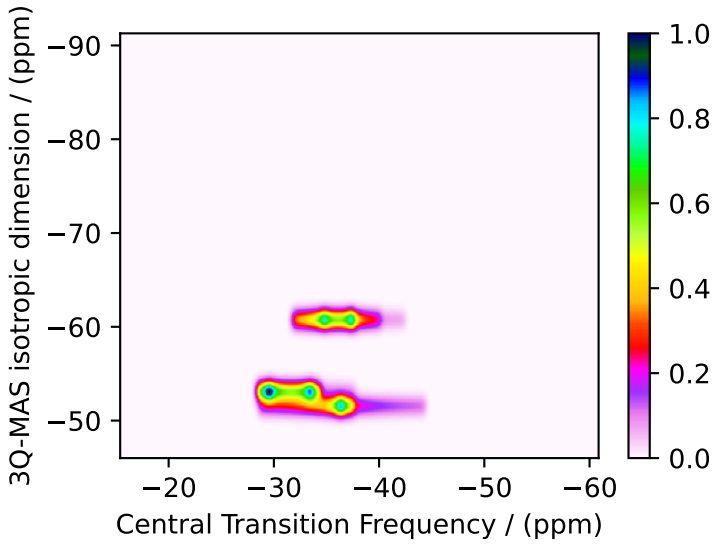
```
plt.figure(figsize=(4, 3))
ax = plt.subplot(projection="csdm")
cb = ax.imshow(dataset.real / dataset.real.max(), aspect="auto", cmap="gist_ncar_r")
plt.colorbar(cb)
ax.invert_xaxis()
ax.invert_yaxis()
plt.tight_layout()
plt.show()
```

Note: For MQ-MAS, a second shear and scale can be applied to remove isotropic chemical shift component along the $\Omega^{[2]''}$ axis. For a spin $I = 3/2$ nucleus, with a second shear factor of $\kappa^{(\omega_1)} = -8/17$, the affine matrix is given by

$$\mathcal{A}_1 = \begin{bmatrix} 1 & 0 \\ \frac{\kappa^{(\omega_1)}}{1+|\kappa^{(\omega_1)}|} & \frac{1}{1+|\kappa^{(\omega_1)}|} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -8/25 & 17/25 \end{bmatrix}, \tag{8.28}$$

and the product of the two affine transformations is

$$\mathcal{A}_T = \mathcal{A}_1\mathcal{A}_2 = \begin{bmatrix} 1 & 0 \\ -8/25 & 17/25 \end{bmatrix} \begin{bmatrix} 9/16 & 7/16 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 9/16 & 7/16 \\ -9/50 & 27/50 \end{bmatrix}. \tag{8.29}$$



Below is the code for simulating a 3Q-MAS spectrum with a double shear transformation.

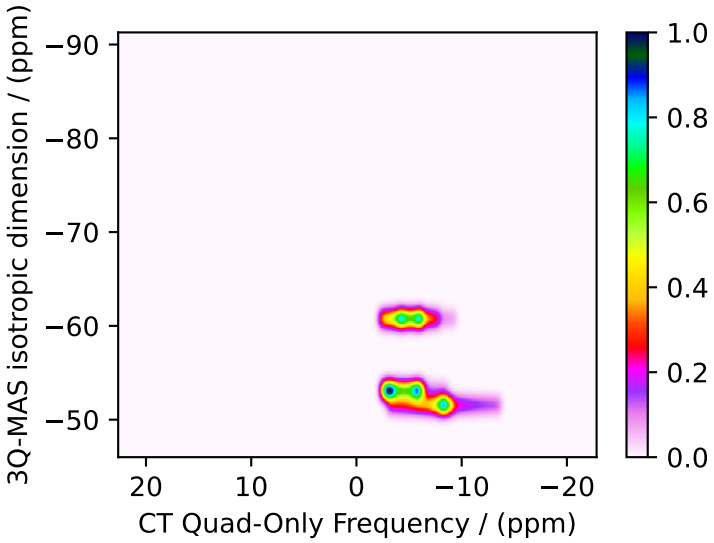
```
my_twice_sheared_mqmas = Method(
    channels=["87Rb"],
    magnetic_flux_density=9.4,
    rotor_frequency=np.inf,  # in Hz (here, set to infinity)
    spectral_dimensions=[
        SpectralDimension(
            count=128,
            spectral_width=6e3,  # in Hz
            reference_offset=-9e3,  # in Hz
            label="3Q-MAS isotropic dimension",
            events=[
                SpectralEvent(transition_queries=[{"ch1": {"P": [-3], "D": [0]}}])
            ],
        ),
        SpectralDimension(
            count=256,
            spectral_width=6e3,  # in Hz
            reference_offset=0,  # in Hz
            label="CT Quad-Only Frequency",
            events=[
                SpectralEvent(transition_queries=[{"ch1": {"P": [-1], "D": [0]}}])
            ],
        ),
    ],
    affine_matrix=[[9 / 16, 7 / 16], [-9 / 50, 27 / 50]],
)

sim = Simulator(spin_systems=RbNO3_spin_systems, methods=[my_twice_sheared_mqmas])
sim.run()
```

(continues on next page)

(continued from previous page)

```
dataset = gauss_convolve.apply_operations(dataset=sim.methods[0].simulation)
```

```
plt.figure(figsize=(4, 3))
ax = plt.subplot(projection="csdm")
cb = ax.imshow(dataset.real / dataset.real.max(), aspect="auto", cmap="gist_ncar_r")
plt.colorbar(cb)
ax.invert_xaxis()
ax.invert_yaxis()
plt.tight_layout()
plt.show()
```



8.7 Average Frequency & Multiple Events

To illustrate the versatility of the Method object, we can also design an MQ-MAS method that correlates the isotropic MQ-MAS frequency to the central transition without the need for an affine transformation. Recall that the 3Q-MAS isotropic frequency on spin $I = 3/2$ is given by

$$\Omega_{\text{iso}} = \frac{9}{16}\Omega_{3Q} + \frac{7}{16}\Omega_{\text{CT}}. \tag{8.30}$$

As we saw at the beginning of this section, the first spectral dimension derives its *average frequency*, $\overline{\Omega}_1$, from a weighted average of multiple transition frequencies. Thus, this weighted average frequency can be obtained through the use of multiple SpectralEvent objects in the SpectralDimension associated with the isotropic dimension, as shown in the code below.

```
my_three_event_mqmas = Method(
    channels=["87Rb"],
    magnetic_flux_density=9.4,
    rotor_frequency=np.inf,  # in Hz (here, set to infinity)
    spectral_dimensions=[
        SpectralDimension(
```

(continues on next page)

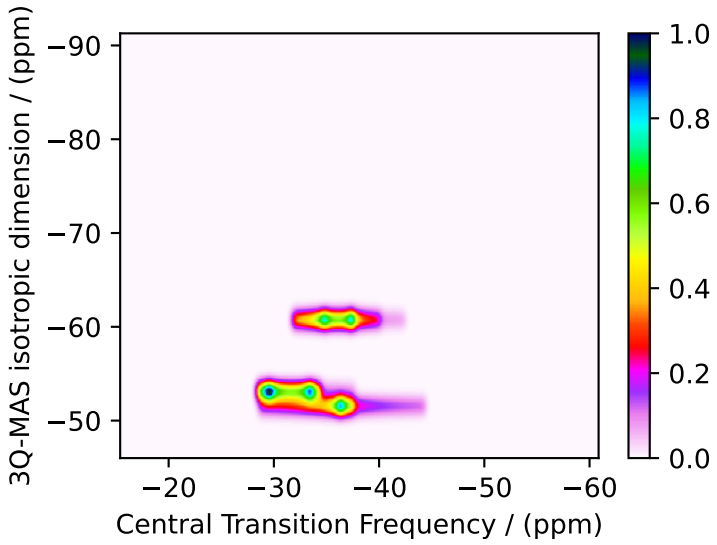
(continued from previous page)

```
            count=128,
            spectral_width=6e3,  # in Hz
            reference_offset=-9e3,  # in Hz
            label="3Q-MAS isotropic dimension",
            events=[
                SpectralEvent(
                    fraction=9 / 16, transition_queries=[{"ch1": {"P": [-3], "D": [0]}}]
                ),
                SpectralEvent(
                    fraction=7 / 16, transition_queries=[{"ch1": {"P": [-1], "D": [0]}}]
                ),
            ],
        ),
        SpectralDimension(
            count=256,
            spectral_width=6e3,  # in Hz
            reference_offset=-5e3,  # in Hz
            label="Central Transition Frequency",
            events=[
                SpectralEvent(transition_queries=[{"ch1": {"P": [-1], "D": [0]}}])
            ],
        ),
    ],
)

sim = Simulator(spin_systems=RbNO3_spin_systems, methods=[my_three_event_mqmas])
sim.run()

dataset = gauss_convolve.apply_operations(dataset=sim.methods[0].simulation)
```

```
plt.figure(figsize=(4, 3))
ax = plt.subplot(projection="csdm")
cb = ax.imshow(dataset.real / dataset.real.max(), aspect="auto", cmap="gist_ncar_r")
plt.colorbar(cb)
ax.invert_xaxis()
ax.invert_yaxis()
plt.tight_layout()
plt.show()
```

We could apply an affine transformation to remove the isotropic chemical shift from the central transition (horizontal) dimension. If you go back to the previous discussion, you will find that the required value for the `affine_matrix` in the Method object to do this shear is given by

```
affine_matrix=[[1,0],[-8/25, 17/25]]
```



8.8 Mixing Queries

The amplitude of a transition pathway signal derives from the product of mixing amplitudes associated with each transfer between transitions in a transition pathway, e.g.,

$$(a_{0A})\,\hat{t}_A \rightarrow (a_{0A}a_{AB})\,\hat{t}_B \rightarrow (a_{0A}a_{AB}a_{BC})\,\hat{t}_C \rightarrow \cdots \tag{8.31}$$

Here, a_{0A} is the amplitude of the initial $\hat{t}_A$ transition, a_{AB} is the mixing amplitude for the transfer from $\hat{t}_A$ to $\hat{t}_B$, a_{BC} is the mixing amplitude for the transfer from $\hat{t}_B$ to $\hat{t}_C$, and so on. The growing product $(a_{0A}a_{AB}a_{BC}\cdots)$ is the transition pathway amplitude. Eliminating a transition with a TransitionQuery in a spectral or delay event sets the eliminated transition's pathway amplitude to zero, i.e., it prunes that transition pathway branch.

8.8.1 Default Total Mixing between Adjacent Spectral or Delay Events

In previous discussions, we did not mention the efficiency of transfer between selected transitions in adjacent SpectralEvent objects. This is because, as default behavior, **mrsimulator** does a *total mixing*, i.e., connects all selected transitions in the two adjacent spectral or delay events. In other words, if the first of two adjacent SpectralEvent objects has three selected transitions, and the second has two selected transitions, then **mrsimulator** will make $3 \times 2 = 6$ connections, i.e., six transition pathways passing from the first to second SpectralEvent objects.

Additionally, this *total mixing* assumes that every connection has a mixing amplitude of 1. This is unrealistic, but if used correctly gives a significant speed-up in the simulation by avoiding the need to calculate mixing amplitudes.

Warning: The use of total mixing, i.e., the default mixing, can complicate the comparison of integrated intensities between different methods, depending on the selected transition pathways.

If this default mixing behavior had been explicitly shown in the previous example, the events list in the first SpectralDimension would have looked like the code below.

```
from mrsimulator.method import MixingEvent
```

(continues on next page)

(continued from previous page)

```
events = [
    SpectralEvent(fraction=9 / 16, transition_queries=[{"ch1": {"P": [-3], "D": [0]}}]),
    MixingEvent(query="TotalMixing"),
    SpectralEvent(fraction=7 / 16, transition_queries=[{"ch1": {"P": [-1], "D": [0]}}]),
    MixingEvent(query="TotalMixing"),
]
```

Since only one transition was selected in each SpectralEvent, the expected (and default) behavior is that there is a mixing (transfer) of coherence between the symmetric triple-quantum and central transitions, forming the desired transition pathway.

However, when multiple transition pathways are present in a method, you may need more accurate mixing amplitudes when connecting selected transitions of adjacent events. You may also need to prevent the undesired mixing of specific transitions between two adjacent events. As described below, you can avoid a `"TotalMixing"` event by inserting MixingEvent object with a certain rotation query.

8.8.2 Rotation Query

A rotation of θ about an axis defined by ϕ in the x-y plane on a selected transition, $|I, m_f\rangle\langle I, m_i|$, in a spectral or delay event transfers it to all selected transitions, $\left|I, m'_f\right\rangle\langle I, m'_i|$ in the next spectral or delay event, according to

$$|I, m_f\rangle\langle I, m_i| \xrightarrow{\theta_\phi} \sum_{m'_f}\sum_{m'_i} d^{(I)}_{m'_f,m_f}(\theta) d^{(I)}_{m'_i,m_i}(\theta) e^{-i\Delta p\phi}(i)^{\Delta p}\left|I, m'_f\right\rangle\langle I, m'_i|, \tag{8.32}$$

where $\Delta p_I = p'_I - p_I$. From this expression, we obtain the complex mixing amplitude from $|I, m_f\rangle\langle I, m_i|$ to $\left|I, m'_f\right\rangle\langle I, m'_i|$ due to a rotation to be

$$a(\theta, \phi) = d^{(I)}_{m'_f,m_f}(\theta) d^{(I)}_{m'_i,m_i}(\theta) e^{-i\Delta p\phi}(i)^{\Delta p}. \tag{8.33}$$

From this expression, we note a few interesting and useful cases. One is the coherence transfer under a π rotation, given by

$$|I, m_f\rangle\langle I, m_i| \xrightarrow{\pi_\phi} |I, -m_f\rangle\langle I, -m_i| e^{-i\Delta p\phi}(i)^{\Delta p}. \tag{8.34}$$

That is, a π rotation can make only one connection between transitions in adjacent events. It is also a special connection because the p_I transition symmetry value for the two transitions are equal but opposite in sign. Additionally, the d_I transition symmetry remains unchanged ($\Delta \mathrm{d}_I = 0$) for the two transitions. (In fact, this behavior under a π rotation is generally true for odd $(\mathrm{p}_I, \mathrm{f}_I, \ldots)$ and even $(\mathrm{d}_I, \mathrm{g}_I, \ldots)$ rank spin transition symmetry functions.)

Another interesting result is that, while a rotation can transfer a transition into many other transitions, the d_I transition symmetry value cannot remain unchanged ($\Delta \mathrm{d}_I \neq 0$) between two connected transitions under a $\pi/2$ rotation.

Finally, another useful result is

$$|I, m_f\rangle\langle I, m_i| \xrightarrow{(0)_\phi} |I, m_f\rangle\langle I, m_i|. \tag{8.35}$$

While it's not surprising that a rotation through an angle of zero does nothing to the transition, this turns out to help act as the opposite of a total mixing event, i.e., a `"NoMixing"` event. As a convenience, this is defined as a `"NoMixing"` query and can be implemented with the code below.

```
MixingEvent(query="NoMixing")
```

The MixingEvent object holds the rotation details in a MixingQuery object as a RotationQuery object associated with a `channels` attribute. This is illustrated in the sample code below.

```
import numpy as np
from mrsimulator.method.query import RotationQuery
rot_query_90 = RotationQuery(angle=np.pi/2, phase=0)
rot_query_180 = RotationQuery(angle=np.pi, phase=0)
rot_mixing = MixingEvent(query={
        "ch1": rot_query_90,
        "ch2": rot_query_180
    }
)
```

8.8.3 p and d Echoes on Deuterium

Here, we examine two examples in a deuterium spin system that illustrate the importance of echo classification in understanding how transition-frequency contributions can be eliminated or separated based on their dependence on different transition symmetry functions.

First, we implement two Method objects that follow the design of two experimental pulse sequences. In this effort, we use RotationQuery objects to select the desired transition pathways and obtain spectra with the desired average frequencies. Then, we implement two simpler Method objects that produce identical spectra and illustrate how *frequency contributions* (page 416) can be used to reduce the number of events needed in a custom method.

Consider the Hahn and Solid Echo pulse sequences on the left and right, respectively.

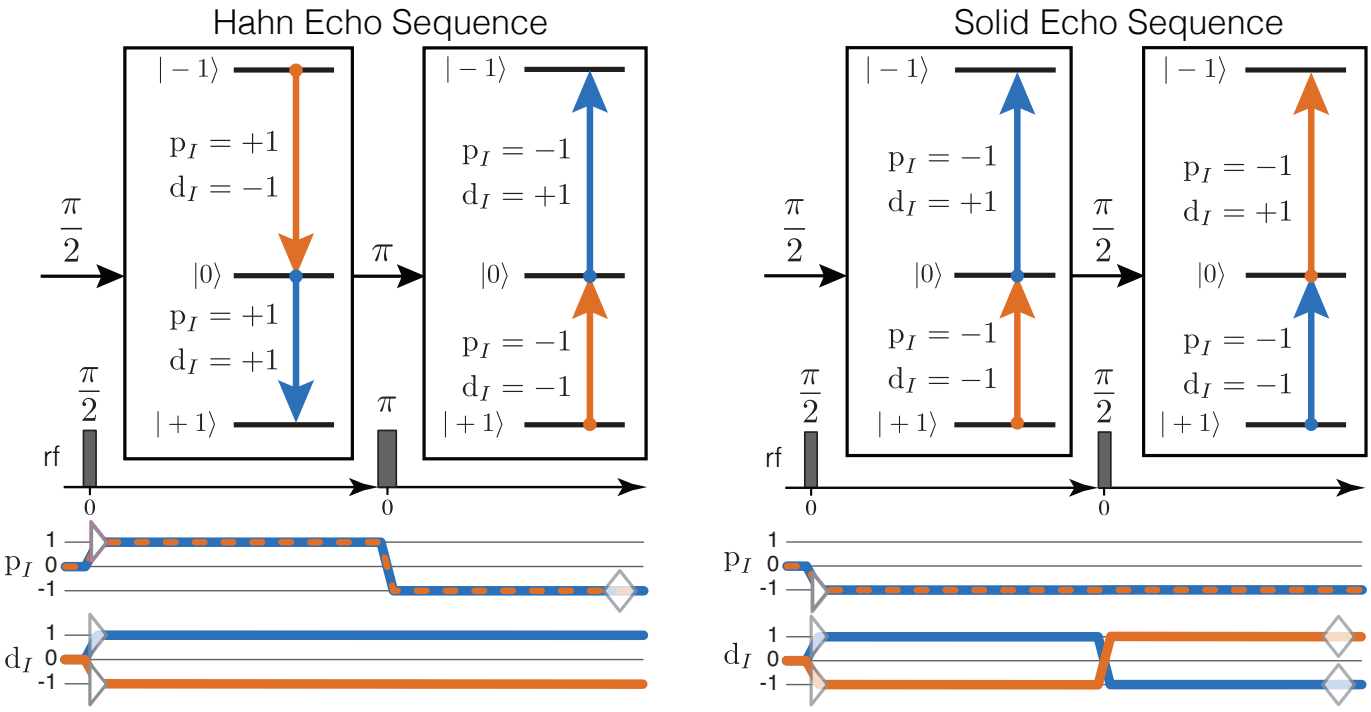


Figure 8.9: Hahn Echo (left) and Solid-Echo (right) pulse sequences. Above each sequence, on the energy level diagram, are the corresponding two transition pathways indicated with blue and orange arrows. Transitions are labeled with their corresponding p_I and d_I transition symmetry function values. Below each sequence are the corresponding transition symmetry pathways, also in blue and orange.

The Hahn Echo sequence, with $\pi/2 - \tau - \pi - t \rightarrow$, leads to the formation of a p_I echo at $t = \tau$. The two transition pathways created by this experiment on a deuterium nucleus are illustrated beneath the sequence. Remember that a π rotation is a special because it connects transitions with equal but opposite signs of p_I while d_I remains invariant.

The Solid Echo sequence, with $\pi/2 - \tau - \pi/2 - t \rightarrow$, leads to the formation of a d_I echo at $t = \tau$. The two transition pathways created by this experiment on a deuterium nucleus are illustrated beneath the sequence. Here, also recall that the d_I transition symmetry value cannot remain unchanged ($\Delta d_I \neq 0$) between two connected transitions under a $\pi/2$ rotation.

Below are two custom Method objects for simulating the Hahn and Solid Echo experiments. There is only one SpectralDimension object in each method, and the average frequency during each spectral dimension is derived from equal fractions of two SpectralEvent objects. Between these two SpectralEvent objects is a MixingEvent with a RotationQuery object. The RotationQuery object is created with a π rotation in the Hahn Echo method, and a $\pi/2$ rotation in the Solid Echo method.

Note: The `transition_queries` attribute of SpectralEvent holds a list of TransitionQuery objects. Each TransitionQuery in the list applies to the full set of transitions in the spin system. The union of these transition subsets becomes the final set of selected transitions during the SpectralEvent.

We use the deuterium Site defined earlier in this document.

```
from mrsimulator.method import MixingEvent

deuterium = Site(
    isotope="2H",
    isotropic_chemical_shift=10,  # in ppm
    shielding_symmetric={"zeta": -80, "eta": 0.25},  # zeta in ppm
    quadrupolar={"Cq": 10e3, "eta": 0.0, "alpha": 0, "beta": np.pi / 2, "gamma": 0},
)
deuterium_system = SpinSystem(sites=[deuterium])

hahn_echo = Method(
    channels=["2H"],
    magnetic_flux_density=9.4,  # in T
    rotor_angle=0,  # in rads
    spectral_dimensions=[
        SpectralDimension(
            count=512,
            spectral_width=2e4,  # in Hz
            events=[
                SpectralEvent(
                    fraction=0.5,
                    transition_queries=[
                        {"ch1": {"P": [1], "D": [1]}},
                        {"ch1": {"P": [1], "D": [-1]}},
                    ],
                ),
                MixingEvent(query={"ch1": {"angle": 3.141592, "phase": 0}}),
                SpectralEvent(
                    fraction=0.5,
                    transition_queries=[
                        {"ch1": {"P": [-1], "D": [1]}},
                        {"ch1": {"P": [-1], "D": [-1]}},
```

(continues on next page)

(continued from previous page)

```
                    ],
                ),
            ],
        )
    ],
)

solid_echo = Method(
    channels=["2H"],
    magnetic_flux_density=9.4,  # in T
    rotor_angle=0,  # in rads
    spectral_dimensions=[
        SpectralDimension(
            count=512,
            spectral_width=2e4,  # in Hz
            events=[
                SpectralEvent(
                    fraction=0.5,
                    transition_queries=[
                        {"ch1": {"P": [-1], "D": [1]}},
                        {"ch1": {"P": [-1], "D": [-1]}},
                    ],
                ),
                MixingEvent(query={"ch1": {"angle": 3.141592 / 2, "phase": 0}}),
                SpectralEvent(
                    fraction=0.5,
                    transition_queries=[
                        {"ch1": {"P": [-1], "D": [1]}},
                        {"ch1": {"P": [-1], "D": [-1]}},
                    ],
                ),
            ],
        )
    ],
)
```

We can check the resulting transition pathways using these TransitionQuery objects with the code below for the `hahn_echo` method,

```
from pprint import pprint
pprint(hahn_echo.get_transition_pathways(deuterium_system))
```

Out:

```
[|1.0⟩⟨0.0| ⟶ |-1.0⟩⟨0.0|, weight=(1+0j)
 |0.0⟩⟨-1.0| ⟶ |0.0⟩⟨1.0|, weight=(1+0j)]
```

and for the `solid_echo` method with the code below.

```
pprint(solid_echo.get_transition_pathways(deuterium_system))
```

Out:

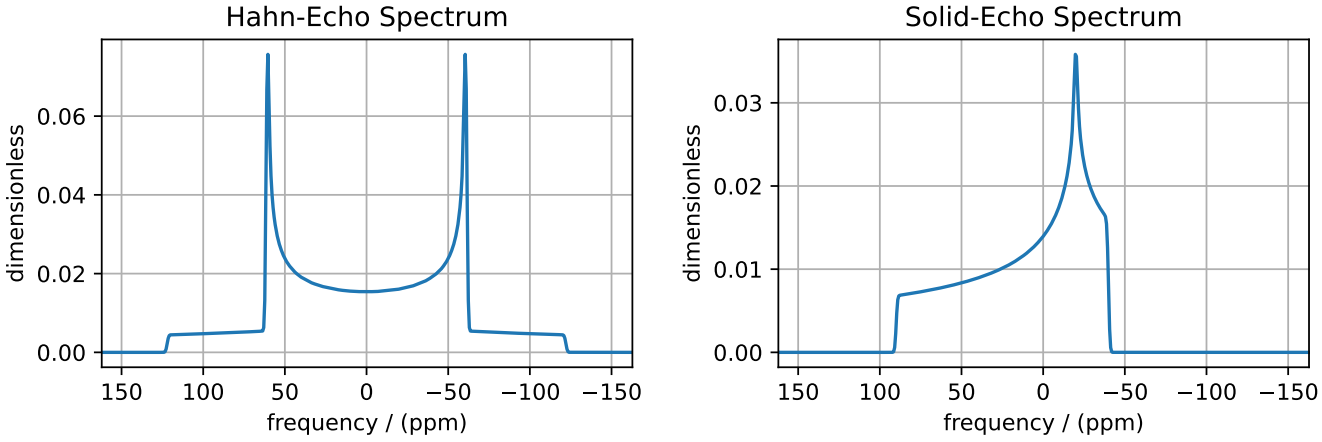
```
[|-1.0⟩⟨0.0| ⟶ |0.0⟩⟨1.0|, weight=(0.5+0j)
 |0.0⟩⟨1.0| ⟶ |-1.0⟩⟨0.0|, weight=(0.5+0j)]
```

Notice that the weights of the transition pathways in the solid-echo method are half of those in the Hahn-echo method. This is because the π pulse in the Hahn-echo method gives perfect transfer between the two transitions in the adjacent spectral events. In contrast, while the $\pi/2$ pulse in the solid-echo method prevents the undesired transition pathways with $\Delta \mathrm{d}_I = 0$, it also connects the selected transitions during the first spectral event to undesired transitions in the second spectral event, which are eliminated by its symmetry query.

Next, we simulate both methods, and perform a Gaussian line shape convolution on each output spectrum, and plot the datasets.

```
sim = Simulator(spin_systems=[deuterium_system], methods=[hahn_echo, solid_echo])
sim.run()

processor = sp.SignalProcessor(
    operations=[
        sp.IFFT(),
        sp.apodization.Gaussian(FWHM="100 Hz"),
        sp.FFT(),
    ]
)
hahn_dataset = processor.apply_operations(dataset=sim.methods[0].simulation)
solid_dataset = processor.apply_operations(dataset=sim.methods[1].simulation)
```

```
fig, ax = plt.subplots(1, 2, subplot_kw={"projection": "csdm"}, figsize=[8.5, 3])
ax[0].set_title("Hahn-Echo Spectrum")
ax[0].plot(hahn_dataset.real)
ax[0].invert_xaxis()
ax[0].grid()
ax[1].set_title("Solid-Echo Spectrum")
ax[1].plot(solid_dataset.real)
ax[1].invert_xaxis()
ax[1].grid()
plt.tight_layout()
plt.show()
```



In the Hahn-echo spectrum, the p_I-dependent frequency contributions (i.e., the shielding contributions) were aver-

aged to zero, leaving only the d_I-dependent frequency contributions (i.e., the first-order quadrupolar contribution). Conversely, in the solid-echo spectrum, the d_I-dependent frequency contributions (i.e., the first-order quadrupolar contribution) were averaged to zero, leaving only the p_I-dependent frequency contributions (i.e., the shielding contributions).

While these two examples nicely illustrate numerous important concepts for building custom methods, it should also be noted that identical spectra could have been obtained with a simpler custom method that used the `freq_contrib` to only select the desired frequency contributions. The code for these two methods is illustrated below.

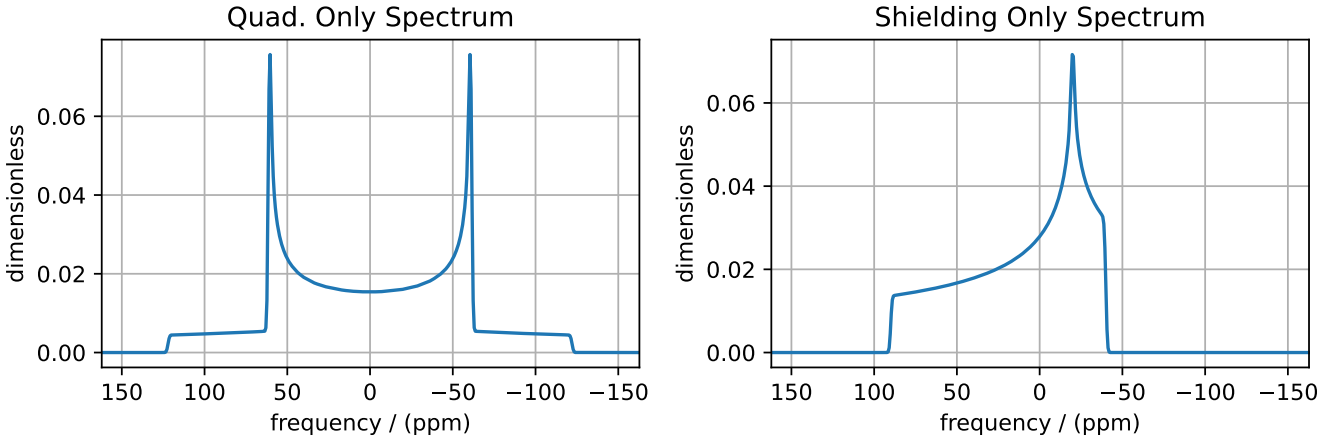
```
quad_only = Method(
    channels=["2H"],
    magnetic_flux_density=9.4,  # in T
    spectral_dimensions=[
        SpectralDimension(
            count=512,
            spectral_width=2e4,  # in Hz
            events=[
                SpectralEvent(
                    transition_queries=[{"ch1": {"P": [-1]}}],
                    freq_contrib=["Quad1_2"]
                )
            ],
        )
    ],
)

shielding_only = Method(
    channels=["2H"],
    magnetic_flux_density=9.4,  # in T
    spectral_dimensions=[
        SpectralDimension(
            count=512,
            spectral_width=2e4,  # in Hz
            events=[
                SpectralEvent(
                    transition_queries=[{"ch1": {"P": [-1]}}],
                    freq_contrib=["Shielding1_0", "Shielding1_2"],
                )
            ],
        )
    ],
)

sim = Simulator()
sim.spin_systems = [SpinSystem(sites=[deuterium])]
sim.methods = [quad_only, shielding_only]
sim.run()

processor = sp.SignalProcessor(
    operations=[
        sp.IFFT(),
        sp.apodization.Gaussian(FWHM="100 Hz"),
        sp.FFT(),
    ]
```

(continues on next page)

(continued from previous page)

```
)
quad_only_dataset = processor.apply_operations(dataset=sim.methods[0].simulation)
shielding_only_dataset = processor.apply_operations(dataset=sim.methods[1].simulation)
```

```
fig, ax = plt.subplots(1, 2, subplot_kw={"projection": "csdm"}, figsize=[8.5, 3])
ax[0].set_title("Quad. Only Spectrum")
ax[0].plot(quad_only_dataset.real)
ax[0].invert_xaxis()
ax[0].grid()
ax[1].set_title("Shielding Only Spectrum")
ax[1].plot(shielding_only_dataset.real)
ax[1].invert_xaxis()
ax[1].grid()
plt.tight_layout()
plt.show()
```



Note: mrsimulator also includes shortcuts for addressing groups of frequency contributions together. For example, the `shielding_only` method could have selected all shielding contributions by using `freq_contrib=["Shielding"]` which expands to zeroth- and second-rank shielding. A complete list of shortcuts are listed in *FrequencyEnum* (page 416).

8.9 Origin and Reference Offset

SpectralDimension() (page 410) has additional attributes that have already been discussed in earlier sections of the documentation. Notably, `origin_offset` and `reference_offset` are important for converting the frequency coordinate into a dimensionless frequency ratio coordinate. For spectra where all the spectral dimensions are associated with single-quantum transitions on a single isotope, the convention for defining `origin_offset` and `reference_offset` is well established; the `origin_offset`, o_k, is interpreted as the NMR spectrometer frequency and the `reference_offset`, b_k, as the reference frequency. Given the frequency coordinate, X, the corresponding dimensionless-frequency ratio follows,

$$X^{\text{ratio}} = \frac{X}{o_k - b_k}. \tag{8.36}$$

In the case of multiple quantum dimensions, however, there appear to be no formal conventions for defining `origin_offset` and `reference_offset`.

8.10 Attribute Summaries

Table 8.2: The attributes of a Method object

Attribute Name	Type	Description
channels	`list`	A *required* list of isotopes given as strings over which the given method applies. For example, `["1H"]`.
magnetic_flux_density	`float`	An *optional* float describing the macroscopic magnetic flux density of the applied external magnetic field in tesla. For example, `18.8` tesla. The default value is `9.4` tesla.
rotor_frequency	`float`	An *optional* float describing the sample rotation frequency in Hz. For example, `2000` Hz. The default value is `0` Hz.
rotor_angle	`float`	An *optional* float describing the angle between the sample rotation axis and the external magnetic field in radians. The default value is the magic angle, `54.735 * 3.14159 / 180 = 0.955305` radians.
spectral_dimensions	`list`	A list of *SpectralDimension* (page 410) objects describing the spectral dimensions for the method.
affine_matrix	`list` or `np.ndarray`	An *optional* (`n` x `n`) affine transformation matrix represented by a numpy array where `n` is the number of spectral dimensions. If provided, the transformation is applied after running a simulation. The default value is `None` and no transformation is applied.
simulation	CSDM object	A CSDM object representing the spectrum simulated by the method. By default, the value is `None`. A value is assigned to this attribute when you run the simulation using the *run()* (page 385) method.
experiment	CSDM object	An *optional* CSDM object holding an experimental measurement of the method. The default value is `None`

Table 8.3: The attributes of a SpectralDimension object

Attribute Name	Type	Description
count	`int`	An *optional* integer representing the number of points, N, along the spectroscopic dimension. For example, `4096`. The default value is `1024`.
spectral_width	`float`	An *optional* float representing the width, Δx, of the spectroscopic dimension in Hz. For example, `10e3` for 10 kHz. The default value is `25000` Hz.
reference_offset	`float`	An *optional* float representing the reference offset, x_0, of the spectroscopic dimension in Hz. For example, `-8000` Hz. The default value is `0`.
origin_offset	`float`	An *optional* float representing the origin offset, or Larmor frequency, along the spectroscopic dimension in units of Hz. The default value is `None` and the origin offset is set to the Larmor frequency of isotope from the *`channels`* (page 403) attribute of the method containing the spectral dimension.
events	`List`	An *optional* list of *Events* (page 412) used to emulate an experiment. The default value is an empty list.

Table 8.4: The attributes of a SpectralEvent object

Attribute Name	Type	Description
magnetic_flux_density	`float`	An *optional* float describing the macroscopic magnetic flux density of the applied external magnetic field in tesla. For example, `18.8` tesla. The default value is `None` and takes the global magnetic flux density defined by the method's `magnetic_flux_density` attribute.
rotor_angle	`float`	An *optional* float describing the angle between the sample rotation axis and the external magnetic field in radians. The default is `None` and takes the global rotor angle defined by the method's `rotor_angle` attribue.
rotor_frequency	`float`	An *optional* float describing the sample rotation frequency in Hz. For example, `2000` Hz. The default value is `None` and takes the global rotor frequency defined by the method's `rotor_frequency` attribute.
freq_contrib	`List`	An *optional* list of *FrequencyEnum* (page 416) (list of allowed strings) selecting which contributions to include when calculating a transition frequency. For example, `["Shielding1_0", "Shielding1_2"]`. String shortcuts encapsulating multiple contributions can also be passed, for example `["Shielding"]` selects all shielding interactions. By default, the list is all frequency enumerations and all frequency contributions are calculated.
transition_queries	`list`	An *optional* `list` of *Query objects* (page 419) objects, or their `dict` representations selecting transitions active during the event. Only these selected transitions will contribute to the net frequency. The default is one TransitionQuery with *P=[0]* on `ch1` and `None` on all other channels

Table 8.5: The attributes of a MixingEvent object

Attribute Name	Type	Description
query	`dict` or `MixingQuery`	A `MixingQuery` object, or its `dict` representation, determines the complex amplitude of mixing between transitions in adjacent spectral or delay events.

CHAPTER

NINE

SIMULATOR

9.1 Overview

The *Simulator* (page 379) is the top-level object in **mrsimulator**. The two main attributes of a Simulator object are *spin_systems* and *methods*, which hold a list of *SpinSystem* (page 388) and *Method* (page 403) objects, respectively. In addition, a simulator object also contains a *config* attribute, which holds a *ConfigSimulator* (page 386) object. The ConfigSimulator object configures the simulation properties, which may be useful in optimizing simulations.

In this section, you will learn about the ConfigSimulator attributes. For simplicity, the following code pre-defines the plot function to use further in this document.

```
import matplotlib.pyplot as plt

# function to render figures.
def plot(csdm_object):
    plt.figure(figsize=(5, 3))
    ax = plt.subplot(projection="csdm")
    ax.plot(csdm_object.real, linewidth=1.5)
    ax.invert_xaxis()
    plt.tight_layout()
    plt.show()
```

9.2 ConfigSimulator

In mrsimulator, the default configuration settings apply 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 insufficient to represent the spectrum accurately. In this section, we use the simulator setup code below to illustrate some of these issues.

```
from mrsimulator import Site, Simulator, SpinSystem
from mrsimulator.spin_system.tensors import SymmetricTensor
from mrsimulator.method import SpectralDimension
from mrsimulator.method.lib import BlochDecaySpectrum

# Setup the spin system and method objects
Si29_site = Site(
    isotope="29Si",
    shielding_symmetric=SymmetricTensor(
        zeta=100,  # in ppm
```

(continues on next page)

(continued from previous page)

```
        eta=0.2,
        alpha=1.563,  # in rads
        beta=1.2131,  # in rads
        gamma=2.132,  # in rads
    )
)
system = SpinSystem(sites=[Si29_site])

method = BlochDecaySpectrum(
    channels=["29Si"],
    rotor_frequency=0,  # in Hz
    spectral_dimensions=[SpectralDimension(count=1024, spectral_width=25000)]
)

# Create the Simulator object
sim = Simulator(spin_systems=[system], methods=[method])
```

Here, `sim` is a *Simulator* (page 379) object that holds one spin system and one method. See *Spin System* (page 51) and *Method* (page 79) documentation for more information on the respective classes.

9.2.1 Integration Volume

The attribute *integration_volume* (page 386) is an enumeration of string literals, `octant`, `hemisphere`, and `sphere`. The integration volume refers to the volume of a unit sphere over which the integrated NMR frequencies are evaluated. The default value is `octant`, i.e., the spectrum comprises integrated frequencies from the positive octant of a unit sphere. **mrsimulator** can exploit the problem's orientational symmetry, thus optimizing the simulation by performing a partial integration.

To learn more about the orientational symmetries, refer to Eden et al.[1]

Consider the ^{29}Si site, `Si29_site`, from the above setup. This site has a symmetric shielding tensor with `zeta` and `eta` as 100 ppm and 0.2, respectively. With only `zeta` and `eta` (and zero Euler angles), we could exploit the symmetry of the problem and evaluate the frequency integral over the octant, equivalent to integration over a sphere. The non-zero Euler angles for this tensor break the symmetry, and integration over the octant will no longer be accurate.

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

To fix this inaccuracy, set the integration volume to `hemisphere` and re-simulate.

```
sim.config.integration_volume = "hemisphere"
sim.run()
plot(sim.methods[0].simulation)
```

[1] Edén, M. and Levitt, M. H. Computation of orientational averages in solid-state nmr by gaussian spherical quadrature. J. Mag. Res., **132**, *2*, 220-239, 1998. doi:10.1006/jmre.1998.1427.

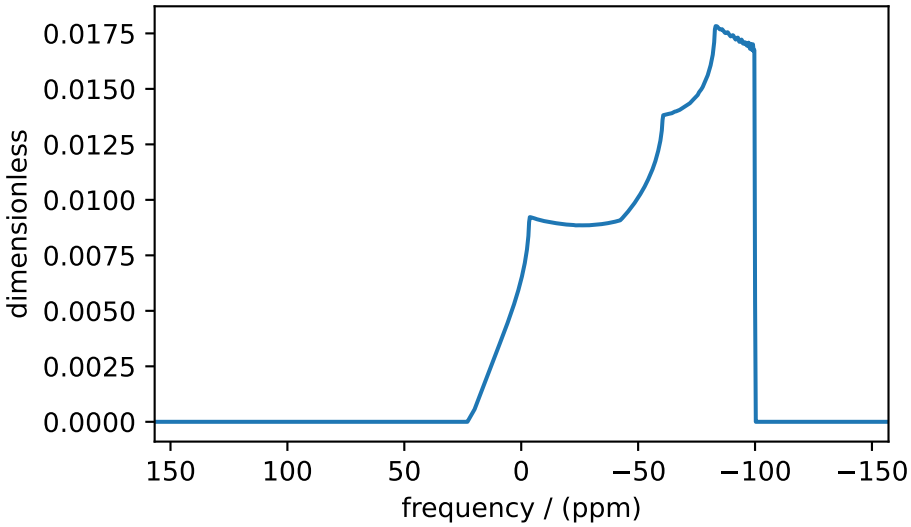


Figure 9.1: Inaccurate simulation resulting from integrating over an octant when the spin system contains non-zero Euler angles.

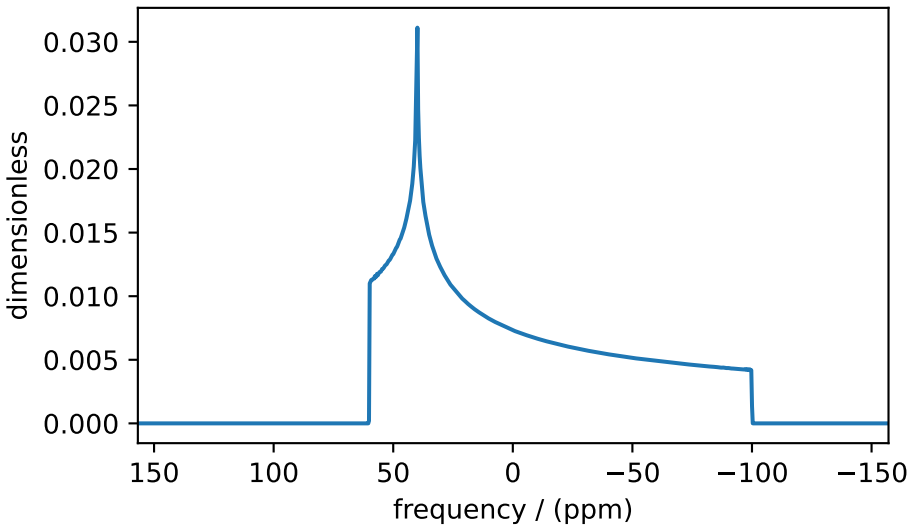


Figure 9.2: Accurate CSA spectrum resulting from the frequency contributions evaluated over the top hemisphere.

9.2.2 Integration Density

The attribute *integration_density* (page 386) controls the number of orientations sampled over the given volume. The resulting spectrum is the integrated NMR resonance frequency evaluated over these orientations. The total number of orientations, Θ_{count}, is

$$\Theta_{\text{count}} = M(n+1)(n+2)/2 \tag{9.1}$$

where M is the number of octants and n is the value of this attribute. The number of octants is the value from the `integration_volume` attribute. The default value of this attribute, 70, produces 2556 orientations at which the NMR frequency contributions are evaluated.

```
sim.config.integration_density = 10
sim.run()
plot(sim.methods[0].simulation)
```

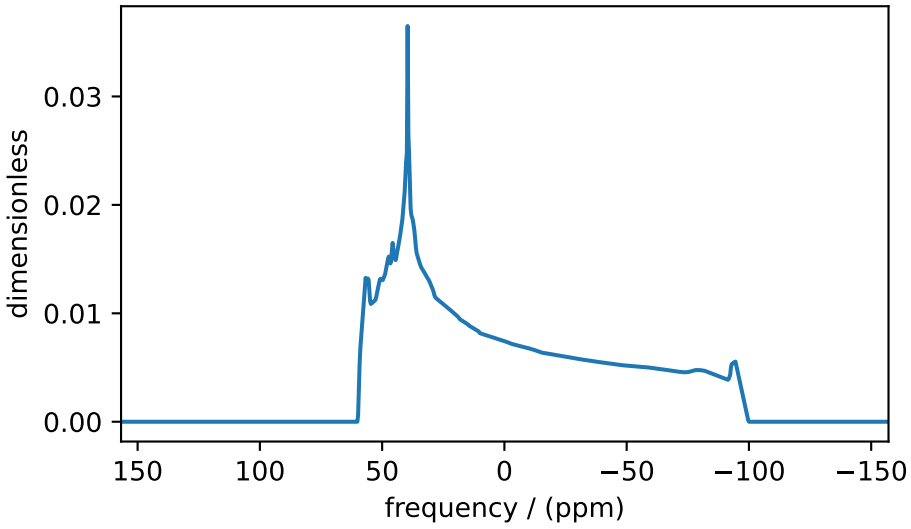


Figure 9.3: Low-quality simulation from reduced integration density (=10).

```
sim.config.integration_density = 100
sim.run()
plot(sim.methods[0].simulation)
```

Decreasing the integration density may decrease the simulation time for computationally intensive simulations but at the cost of spectrum quality. Generally, use a higher integration density for a high-resolution spectrum (*i.e.*, a high-resolution sampling grid). For a low-resolution sampling grid, the spectrum may converge with a lower integration density.

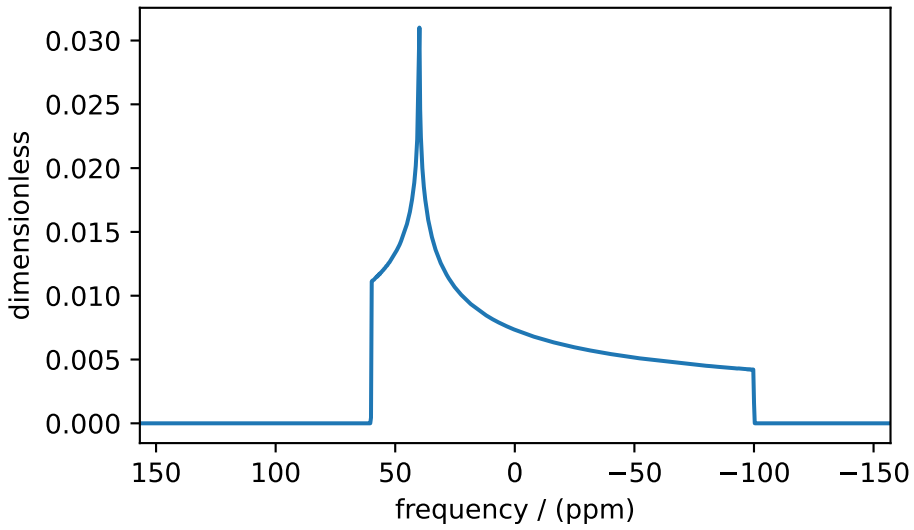


Figure 9.4: High-quality simulation from increased integration density (=100).

9.2.3 Number of Sidebands

The *number_of_sidebands* (page 386) attribute determines the number of sidebands evaluated in the simulation. The default value is 64 which is sufficient for most cases.

In certain circumstances, especially when the anisotropy is large or the rotor spin frequency is low, 64 sidebands might not be sufficient.

```
sim.methods[0] = BlochDecaySpectrum(
    channels=["29Si"],
    rotor_frequency=200,
    spectral_dimensions=[SpectralDimension(count=1024, spectral_width=25000)],
)
sim.run()
plot(sim.methods[0].simulation)
```

Looking at the spinning sideband patterns, you see an abrupt termination of the sideband amplitudes at the edges. This inaccuracy arises from evaluating a small number of sidebands relative to the size of anisotropy. Increasing the number of sidebands will resolve this issue.

```
sim.config.number_of_sidebands = 90
sim.run()
plot(sim.methods[0].simulation)
```

Conversely, 64 sidebands might be excessive, in which case reducing the number of sidebands may significantly improve simulation performance, especially in iterative algorithms, such as the least-squares minimization.

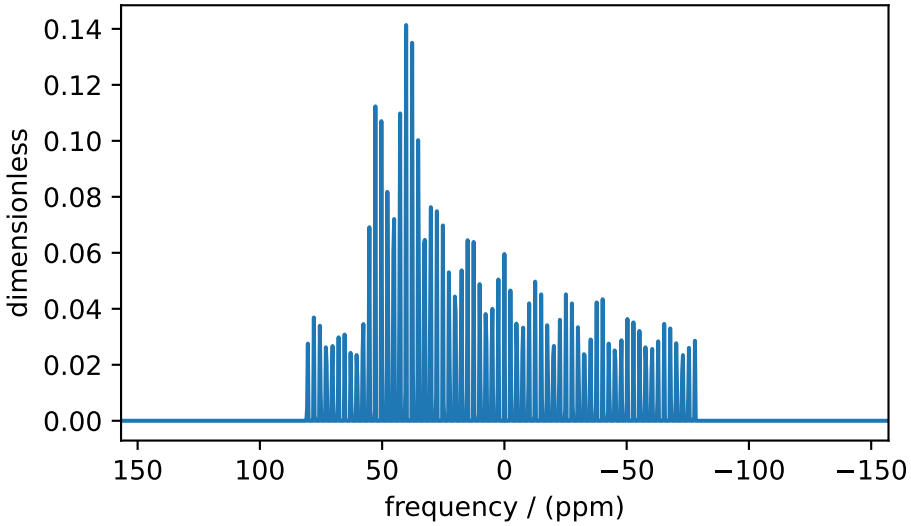


Figure 9.5: Inaccurate sideband simulation resulting from computing a low number of sidebands.

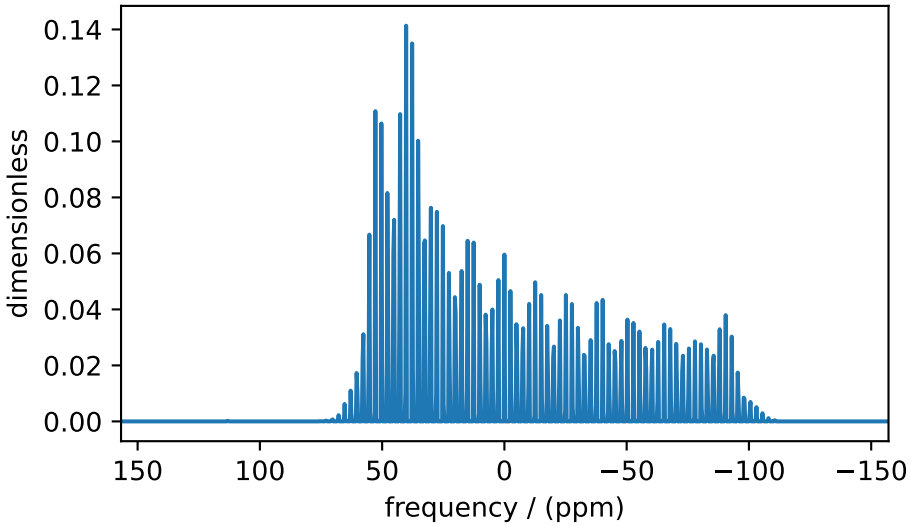


Figure 9.6: Accurate sideband simulation after increasing the number of sidebands.

9.2.4 Number of gamma angles

The *number_of_gamma_angles* (page 386) attribute determines the extent of gamma averaging in the simulation. The gamma angles range from 0 to 2π. The default value is 1, corresponding to $\gamma = 0$.

In most static powder simulations, you can get by with one gamma angle (default) by appropriately setting the *rotor_angle=0*. When evaluating a static powder simulation for a non-zero rotor_angle, use a large number of gamma angles for the simulation to converge.

```
from mrsimulator.method import Method
from mrsimulator.method.event import SpectralEvent, MixingEvent

site = Site(isotope="29Si", shielding_symmetric={"zeta": 100, "eta": 0.2})
spin_system = SpinSystem(sites=[site])

solid_echo = Method(
    channels=["29Si"],
    rotor_frequency=0,  # in Hz
    rotor_angle=54.734 * np.pi / 180,  # in rads
    spectral_dimensions=[
        SpectralDimension(
            count=1024,
            spectral_width=25000,
            events=[
                SpectralEvent(fraction=0.5, transition_queries=[{"ch1": {"P": [-1]}}]),
                MixingEvent(query={"ch1": {"angle": np.pi / 2}}),
                SpectralEvent(fraction=0.5, transition_queries=[{"ch1": {"P": [-1]}}]),
            ]
    )],
)

sim = Simulator(spin_systems=[spin_system], methods=[solid_echo])
sim.run()
plot(sim.methods[0].simulation)
```

To resolve this, increase the number of gamma angles.

```
sim.config.number_of_gamma_angles=1000
sim.run()
plot(sim.methods[0].simulation)
```

9.2.5 Decompose Spectrum

The attribute *decompose_spectrum* (page 387) is an enumeration with two string literals, `None` and `spin_system`. The default value is `None`.

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

```
# Create two distinct sites
site_A = Site(
    isotope="1H",
    shielding_symmetric=SymmetricTensor(zeta=5, eta=0.1),
```

(continues on next page)

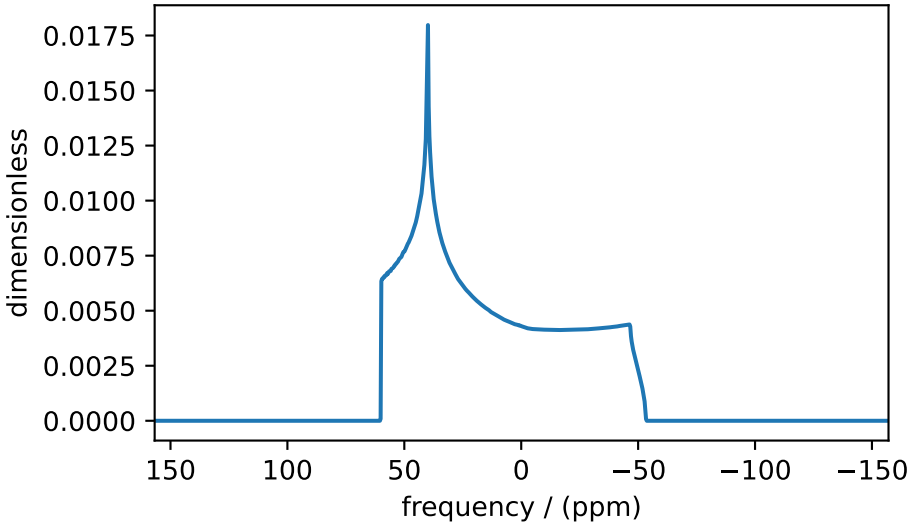


Figure 9.7: Incorrect simulation from an insufficient number of gamma angle averaging.

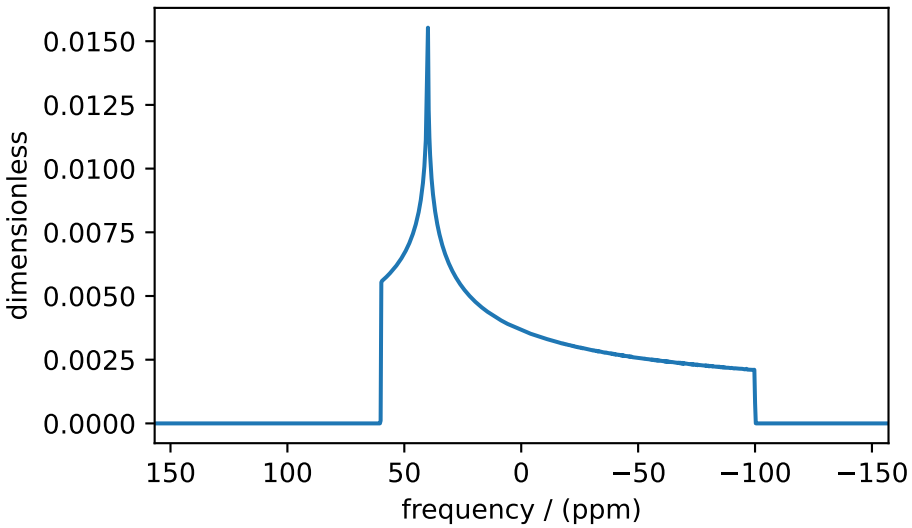


Figure 9.8: Accurate simulation from a sufficiently large number of gamma angle averaging.

(continued from previous page)

```
)
site_B = Site(
    isotope="1H",
    shielding_symmetric=SymmetricTensor(zeta=-2, eta=0.83),
)

# Create two single site spin systems
sys_A = SpinSystem(sites=[site_A], name="System A")
sys_B = SpinSystem(sites=[site_B], name="System B")

# Create a method representing a simple 1-pulse acquire experiment
method = BlochDecaySpectrum(
    channels=["1H"], spectral_dimensions=[SpectralDimension(count=1024, spectral_width=10000)]
)

# Create simulator object, simulate, and plot
sim = Simulator(spin_systems=[sys_A, sys_B], methods=[method])
sim.run()
plot(sim.methods[0].simulation)
```

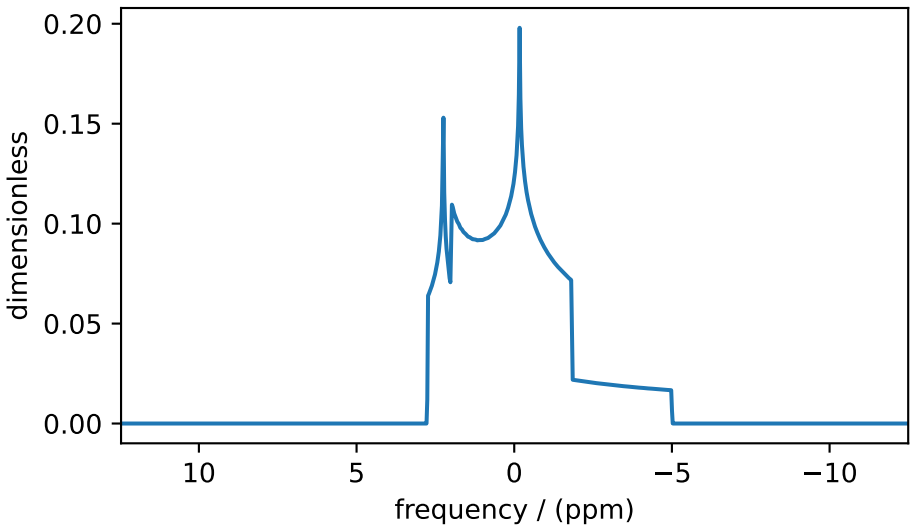


Figure 9.9: The frequency contributions from individual spin systems are combined into one spectrum.

When the value of *decompose_spectrum* (page 387) is `spin_system`, the resulting simulation is a series of subspectra corresponding to individual spin systems. The number of subspectra equals the number of spin systems within the simulator object. Consider the same system as above, now run with decompose_spectrum as `spin_system`.

```
# sim already has the two spin systems and method; no need to reconstruct
sim.config.decompose_spectrum = "spin_system"
sim.run()
plot(sim.methods[0].simulation)
```

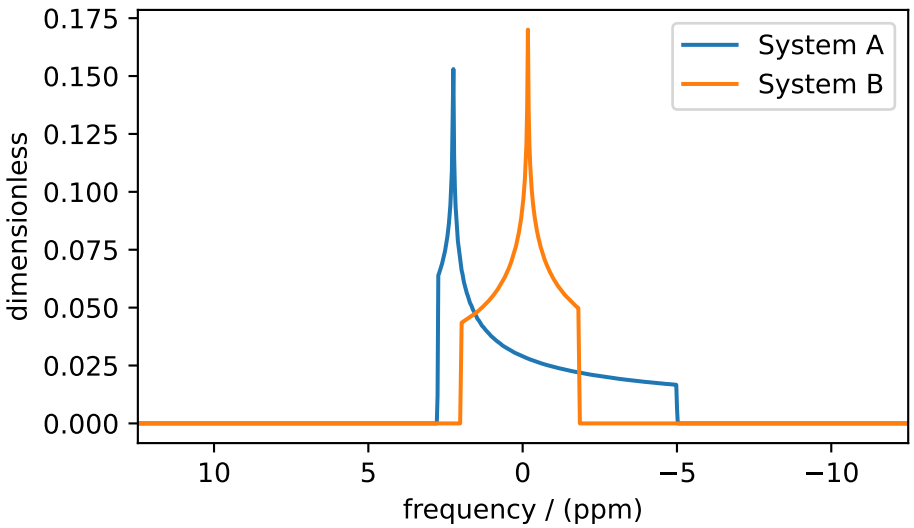


Figure 9.10: Each spin system's frequency contributions are held in separate spectra.

9.2.6 Isotropic interpolation

The attribute *isotropic_interpolation* (page 387) is an enumeration with two string literals, `linear` and `gaussian`. The default value is `linear`.

The value specifies the interpolation scheme used in binning purely isotropic spectrum.

9.3 Attribute Summaries

Table 9.1: The attributes of a Simulator object

Attribute Name	Type	Description
spin_systems	`list`	An *optional* list of *SpinSystem* (page 388) objects.
methods	`list`	An *optional* list of *Method* (page 403) objectss.
config	`dict` or *ConfigSimulator* (page 386)	An *optional* ConfigSimulator object, or its dictionary representation.

Table 9.2: The attributes of a Simulator object

Attribute Name	Type	Description
number_of_sidebands	`int`	An *optional* integer greater than zero specifying the number of sidebands to simulate. The default is `64` sidebands.
integration_volume	`str`	An *optional* string representing the fraction of a unit sphere used in the integrated NMR frequency spectra. The allowed strings are `octant`, `hemisphere`, and `sphere`. The default is `octant`.
integration_density	`int`	An *optional* integer greater than zero specifying the number of orientations sampled over the given volume according to the equation $\Theta_{\text{count}} = M(n+1)(n+2)/2$, where M is the number of octants. The default value is `70`.
decompose_spectrum	`str`	An *optional* string specifying the spectral decomposition type. The allowed strings are `none` and `spin_system`. The value of `none` produces one spectrum averaged over all spin systems, while `spin_system` produces a series of subspectra corresponding to individual spin systems. The default is `none`.
isotropic_interpolation	`str`	An *optional* string specifying the interpolation scheme used in binning purely isotropic subspectra. The allowed strings are `linear` and `gaussian`. The default is `linear`.

CHAPTER

TEN

SIGNAL PROCESSOR

After running a simulation, you may need to apply some post-simulation signal processing. For example, you may need to scale the simulated spectrum to match experimental intensities, or you may want to convolve the spectrum with a Lorentzian, Gaussian, or other line-broadening function. For this reason, **mrsimulator** offers some frequently used NMR signal processing tools within the `mrsimulator.signal_processor` module.

See also:

Signal Processing Gallery (page 353) for notebooks using common processing functions.

10.1 CSDM object

The simulated spectrum is held in a CSDM[1] object, which supports multi-dimensional scientific datasets (NMR, EPR, FTIR, GC, etc.). For more information, see the csdmpy documentation.

10.2 `SignalProcessor` class

Signal processing is a series of operations sequentially applied to the dataset. In mrsimulator, the *`SignalProcessor`* (page 489) object is used to apply operations. Here we create a new SignalProcessor object

```
# Import the signal_processor module
from mrsimulator import signal_processor as sp

# Create a new SignalProcessor object
processor = sp.SignalProcessor()
```

Each signal processor object holds a list of operations under the *operations* attribute. Below we add operations to apply Gaussian line broadening and a scale factor.

```
processor.operations = [
    sp.IFFT(),
    sp.apodization.Gaussian(FWHM="50 Hz"),
    sp.FFT(),
    sp.Scale(factor=120),
]
```

[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 dataset, PLOS ONE, **15**, 1-38, (2020). DOI:10.1371/journal.pone.0225953

First, an inverse Fourier transform is applied to the dataset, converting it to the time domain. Then, a Gaussian apodization, parameterized using a full-width-at-half-maximum (FWHM) of 50 Hz, is applied. Note, the dimensionality of the `FWHM` attribute has the inverse dimensionality of the dataset domain. Finally, a forward Fourier transform is applied to the apodized dataset, and all points are scaled up by 120 times.

Note: Convolutions in **mrsimulator** are performed using the Convolution Theorem. A spectrum is Fourier transformed, and apodizations are performed in the time domain before being transformed back into the frequency domain.

Let's create a CSDM object and then apply the operations to visualize the results.

```
import csdmpy as cp
import numpy as np

# Create a CSDM object with delta function at 200 Hz
test_data = np.zeros(500)
test_data[200] = 1
csdm_object = cp.CSDM(
    dependent_variables=[cp.as_dependent_variable(test_data)],
    dimensions=[cp.LinearDimension(count=500, increment="1 Hz")],
)
```

To apply the previously defined signal processing operations to the above CSDM object, use the *apply_operations()* (page 489) method of the `SignalProcessor` instance as follows

```
processed_dataset = processor.apply_operations(dataset = csdm_object)
```

The variable `processed_dataset` is another CSDM object holding the dataset after the list of operations has been applied to `csdm_object`. Below is a plot comparing the unprocessed and processed dataset

```
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("Unprocessed")
ax[1].plot(processed_dataset.real, color="black", linewidth=1)
ax[1].set_title("Processed")
plt.tight_layout()
plt.show()
```

10.3 Applying Operations along a Dimension

Multi-dimensional NMR simulations may need different operations applied along different dimensions. Each operation has the attribute `dim_index`, which is used to apply operations along a certain dimension.

By default, `dim_index` is `None` and is applied along the 1st dimension. An integer or list of integers can be passed to `dim_index`, specifying the dimensions. Below are examples of specifying the dimensions

```
# Gaussian apodization along the first dimension (default)
sp.apodization.Gaussian(FWHM="10 Hz")

# Constant offset along the second dimension
sp.baseline.ConstantOffset(offset=10, dim_index=1)
```

(continues on next page)

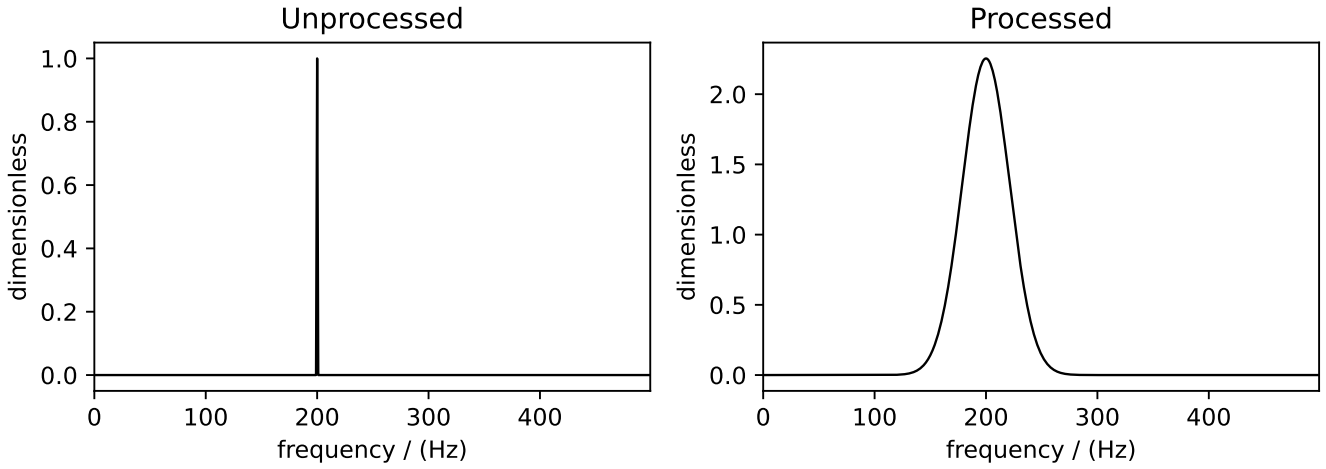


Figure 10.1: The unprocessed dataset (left) and processed dataset (right) with a Gaussian convolution and scale factor.

(continued from previous page)

```
# Exponential apodization along the first and third dimensions
sp.apodization.Exponential(FWHM="10 Hz", dim_index=[0, 2])
```

10.4 Applying Apodizations to specific Dependent Variables

Each dimension in a simulated spectrum can hold multiple dependent variables (a.k.a. contributions from multiple spin systems). Each spin system may need different convolutions applied to match an experimental spectrum. The `Apodization` sub-classes have the `dv_index` attribute, specifying which dependent variable (spin system) to apply the operation on. By default, `dv_index` is `None` and will apply the convolution to all dependent variables in a dimension.

Note: The index of a dependent variable (spin system) corresponds to the order of spin systems in the *spin_systems* (page 379) list.

```
processor = sp.SignalProcessor(
    operations=[
        sp.IFFT(),
        sp.apodization.Gaussian(FWHM="25 Hz", dv_index=0),
        sp.apodization.Gaussian(FWHM="70 Hz", dv_index=1),
        sp.IFFT(),
    ]
)
```

The above list of operations will apply 25 and 70 Hz of Gaussian line broadening to dependent variables at index 0 and 1, respectively.

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

```
test_data = np.zeros(500)
test_data[300] = 1
csdm_object.add_dependent_variable(cp.as_dependent_variable(test_data))
```

Now, we again apply the operations with the *apply_operations()* (page 489) method. The comparison of the unprocessed and processed dataset is also shown below.

```
processed_dataset = processor.apply_operations(dataset = csdm_object)
```

Below is a plot of the dataset before and after applying the operations

```
_, ax = plt.subplots(1, 2, figsize=(8, 3), subplot_kw={"projection":"csdm"})
ax[0].plot(csdm_object, linewidth=1)
ax[0].set_title("Unprocessed")
ax[1].plot(processed_dataset.real, linewidth=1)
ax[1].set_title("Processed")
plt.tight_layout()
plt.show()
```

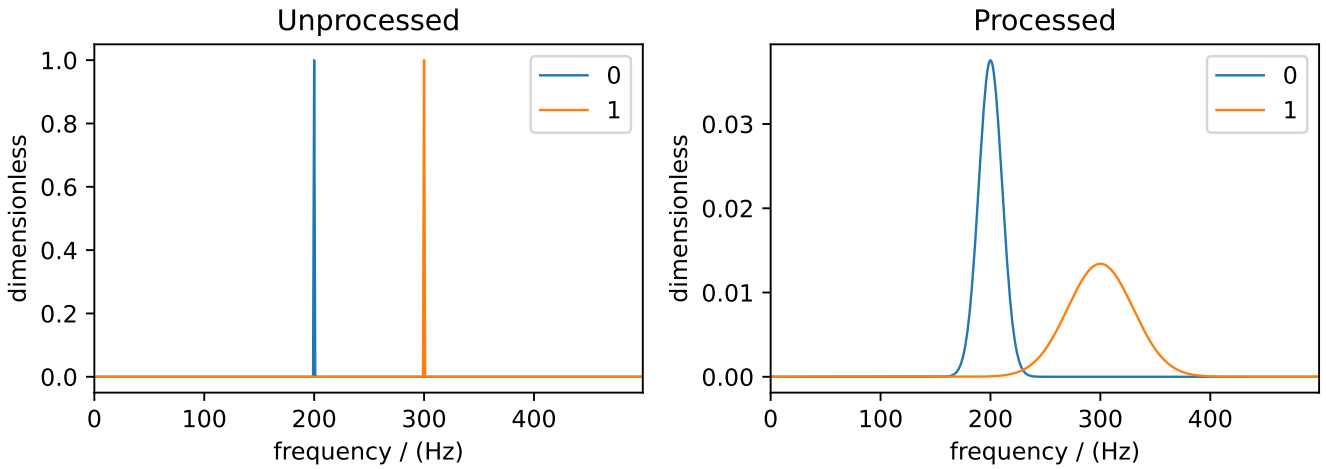


Figure 10.2: The unprocessed dataset (left) and the processed dataset (right) with convolutions applied to different dependent variables.

CHAPTER

ELEVEN

MRSIMULATOR I/O

We offer a range of serialization options based on a JSON structure demonstrated below.

11.1 Dictionary Representation of Objects

All **mrsimulator** objects can be serialized into a JSON format. Calling the `json()` method on an object will return a Python dictionary representing the object in JSON format. Below we call the *json()* (page 397) method of the *Site* (page 394) class.

```
from mrsimulator import Site, SpinSystem
from mrsimulator.spin_system.tensors import SymmetricTensor

Si29_site = Site(
    isotope="29Si",
    isotropic_chemical_shift=-89.0,
    shielding_symmetric=SymmetricTensor(
        zeta=59.8,
        eta=0.62,
    ),
)

py_dict = Si29_site.json()
print(py_dict)
# {
#     'isotope': '29Si',
#     'isotropic_chemical_shift': '-89.0 ppm',
#     'shielding_symmetric': {'zeta': '59.8 ppm', 'eta': 0.62}
# }
```

All values are serialized with units when applicable, but you may call `json(units=False)` if you wish to serialize values without units.

Similarly, all **mrsimulator** objects can be loaded from a dictionary representation. Here we construct the same site as a dictionary and call *parse_dict_with_units()* (page 398) to create a *Site* (page 394) object from a dictionary.

```
site_dict = {
    "isotope": "29Si",
    "isotropic_chemical_shift": "-89.0 ppm",
    "shielding_symmetric": {
        "zeta": "59.8 ppm",
```

(continues on next page)

(continued from previous page)

```
        "eta": 0.62,
    },
}

Si29_site_from_dict = Site().parse_dict_with_units(site_dict)
print(Si29_site_from_dict == Si29_site)
# True
```

We see that both these sites are equivalent. Values in dictionaries can be given as a number and a unit in a string. However, passing values with units increases overhead and will throw errors if the units cannot be converted into the expected units for a field. For this reason, we recommend instantiating objects directly from classes.

11.2 Saving and Loading Spin Systems from a File

A list of spin systems in a *Simulator* (page 379) object can be serialized to a file. Here we create a simulator with three distinct ^{29}Si spin systems and serialize these spin systems to a file by calling *export_spin_systems()* (page 381).

```
from mrsimulator import Site, SpinSystem, Simulator
from mrsimulator.spin_system.tensors import SymmetricTensor

# Create the spin systems
Si29_1 = SpinSystem(
    sites=[
        Site(
            isotope="29Si",
            isotropic_chemical_shift=-89.0,
            shielding_symmetric=SymmetricTensor(zeta=59.8, eta=0.62),
        )
    ]
)
Si29_2 = SpinSystem(
    sites=[
        Site(
            isotope="29Si",
            isotropic_chemical_shift=-89.5,
            shielding_symmetric=SymmetricTensor(zeta=52.1, eta=0.68),
        )
    ]
)
Si29_3 = SpinSystem(
    sites=[
        Site(
            isotope="29Si",
            isotropic_chemical_shift=-87.8,
            shielding_symmetric=SymmetricTensor(zeta=69.4, eta=0.60),
        )
    ]
)

# Create the Simulator object
```

(continues on next page)

(continued from previous page)

```
sim = Simulator(spin_systems=[Si29_1, Si29_2, Si29_3])

# Save spin systems to file
sim.export_spin_systems("example.mrsys")
```

Now the file `example.mrsys` holds a JSON representation of the spin system objects. We encourage the convention of using .mrsys extension for this JSON file.

Just as spin systems can be saved to a file, spin systems can be loaded from a file. Loading spin systems is useful when working with a large number of spin systems over multiple Python scripts. Here we load the spin system file, `example.mrsys`, into a new simulator using the method *load_spin_systems()* (page 383).

```
new_sim = Simulator()
new_sim.load_spin_systems("example.mrsys")
print(len(new_sim.spin_systems))
# 3
```

11.3 Saving and Loading Methods from a File

A list of methods in a *Simulator* (page 379) object can be serialized to a file. Here we create a custom DAS method and serialize it to a file using the method *export_methods()* (page 381).

```
from mrsimulator import Simulator
from mrsimulator.method import Method
from mrsimulator.method import SpectralDimension, SpectralEvent

# Create DAS method
das = Method(
    name="DAS of 17O",
    channels=["17O"],
    magnetic_flux_density=11.744,
    spectral_dimensions=[
        SpectralDimension(
            count=512,
            spectral_width=10000,
            reference_offset=-1220.9,
            origin_offset=67793215,
            label="Isotropic dimension",
            events=[
                SpectralEvent(
                    fraction=0.5,
                    rotor_angle=37.38 * 3.14159 / 180,
                    transition_queries=[{"ch1": {"P": [-1], "D": [0]}}],
                ),
                SpectralEvent(
                    fraction=0.5,
                    rotor_angle=79.19 * 3.14159 / 180,
                    transition_queries=[{"ch1": {"P": [-1], "D": [0]}}],
                ),
            ],
```

(continues on next page)

(continued from previous page)

```
        ),
        # The last spectral dimension block is the direct-dimension
        SpectralDimension(
            count=256,
            spectral_width=11001,
            reference_offset=-1228,
            origin_offset=67793215,
            label="MAS dimension",
            events=[
                SpectralEvent(
                    rotor_angle=54.735 * 3.14159 / 180,
                    transition_queries=[{"ch1": {"P": [-1], "D": [0]}}],
                )
            ],
        ),
    ],
)

# Create simulator with das method
sim = Simulator(methods=[das])

# Save methods to file
sim.export_methods("example.mrmtd")
```

Now the file `example.mrmtd` holds a JSON representation of the method object. If multiple methods are present, e.g., at different spinning speeds, they will also be serialized. We encourage the convention of using .mrmtd extension for this JSON file.

Just like spin systems, methods can also be loaded from a file. Here we load the DAS method into a new simulator object by calling the method *load_methods()* (page 383).

```
new_sim = Simulator()
new_sim.load_methods("example.mrmtd")
print(new_sim.methods[0].name)
# DAS of 17O
```

Loading complex methods from a file, like the DAS example above, can reduce complex code. Methods representing actual experiments can be saved to a file to later be loaded into a script as needed.

11.4 Serializing a Simulator Object

The entire *Simulator* (page 379) object may be serialized to a JSON-compliant file using the *save()* (page 385) Python method. By default, the attribute values are serialized as physical quantities represented as a string with a value and a unit.

```
sim = Simulator()
# ... Setup Simulator object
sim.save("sample.mrsim")
```

Now the file `sample.mrsim` holds the JSON representation of `sim`, a *Simulator* (page 379) object. To load a simulator from a file, call the class method *load()* (page 382). By default, the load method parses the file for units.

```
new_sim = Simulator.load("sample.mrsim")
```

11.5 Serialize simulation from a Method to a CSDM Compliant File

The simulated spectrum may be exported to a CSDM-compliant JSON file using the following code:

```
sim_coesite.methods[0].simulation.save("coesite_simulation.csdf")
```

For more information on the CSDM format, see the csdmpy documentation.

11.6 Serialize Simulator and SignalProcessor object

The *Simulator* (page 379) object and a list of *Signal Processor* (page 489) objects can both be serialized within the same file by calling the *save()* (page 487) method.

```
from mrsimulator import save
from mrsimulator import Simulator
from mrsimulator import signal_processor as sp

sim = Simulator()
processor1 = sp.SignalProcessor()
processor2 = sp.SignalProcessor()

save(
    filename="example.mrsim",
    simulator=sim,
    signal_processors=[processor1, processor2],
)
```

By default, all attribute values are serialized with units. You can serialize attributes without units, assuming the default unit of the attribute, by passing `with_units=False` to the method. Recall that all objects in **mrsimulator** have the attribute `property_units` which provides the default units for all class properties. Additionally, a metadata dictionary can be passed using the `application` keyword.

To load a simulator and signal processors from a file, call the *load()* (page 487) method. This method will return an ordered list of a *Simulator* (page 379) object, a list of *Signal Processor* (page 489) objects, and a metadata dictionary

```
from mrsimulator import load

sim, processors, application = load("example.mrsim")
```

Note: The serialization structure has been updated in **mrsimulator** v0.7. Any *.mrsim* files from v0.6 and earlier will not work. See *Changelog* (page 501) for breaking changes.

Part IV

Examples

CHAPTER

TWELVE

SIMULATION GALLERY

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

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

For applications related to least-squares fitting, see the *Fitting (Least Squares) Gallery* (page 255).

12.1 1D NMR simulation (small molecules/crystalline solids)

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

- Bloch decay method (*BlochDecaySpectrum* (page 425)),
- Central transition selective Bloch decay method (*BlochDecayCTSpectrum* (page 431)).
- Generic one-dimensional method (`Method1D`).

12.2 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 425)),
- Central transition selective Bloch decay method (*BlochDecayCTSpectrum* (page 431)).

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

12.3 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 (i.e., 3Q-MAS) using the specialized *ThreeQ_VAS()* (page 438) method.
- Satellite-transition variable-angle spinning (i.e., ST-MAS) using the specialized *ST1_VAS()* (page 457) method.
- Switched Angle Spinning (SAS) using the generic *Method()* (page 403) object.
- MAS-detected Dynamic Angle Spinning (DAS) using the generic *Method()* (page 403) object.
- Correlation of Anisotropies Separated Through Echo Refocusing (COASTER) using the generic *Method()* (page 403) object.
- Phase Adjusted Spinning Sidebands (PASS and QPASS) and Magic-Angle Turning (MAT and QMAT) using the specialized *SSB2D()* (page 469) method.

12.4 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 438))

12.4.1 1D NMR simulation (small molecules/crystalline solids)

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

- Bloch decay method (*BlochDecaySpectrum* (page 425)),
- Central transition selective Bloch decay method (*BlochDecayCTSpectrum* (page 431)).
- Generic one-dimensional method (`Method1D`).

Wollastonite, ^{29}Si (I=1/2)

^{29}Si (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.pyplot as plt

from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator import signal_processor as sp
from mrsimulator.method.lib import BlochDecaySpectrum
from mrsimulator.spin_system.tensors import SymmetricTensor
from mrsimulator.method import SpectralDimension
```

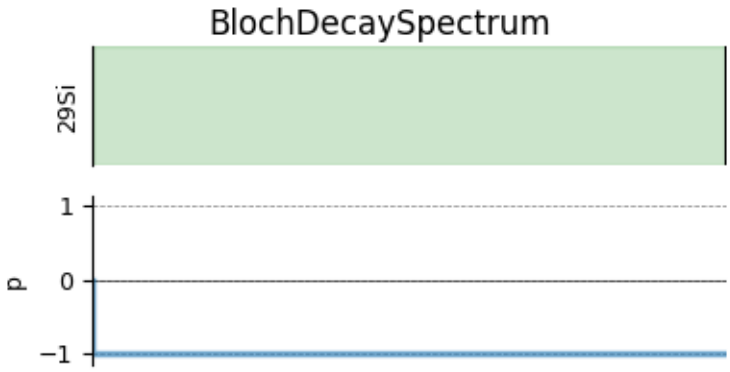
Create sites and spin systems. We create three single-site spin systems for better performance.

[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

```
Si29_1 = Site(
    isotope="29Si",
    isotropic_chemical_shift=-89.0,  # in ppm
    shielding_symmetric=SymmetricTensor(zeta=59.8, eta=0.62),  # zeta in ppm
)
Si29_2 = Site(
    isotope="29Si",
    isotropic_chemical_shift=-89.5,  # in ppm
    shielding_symmetric=SymmetricTensor(zeta=52.1, eta=0.68),  # zeta in ppm
)
Si29_3 = Site(
    isotope="29Si",
    isotropic_chemical_shift=-87.8,  # in ppm
    shielding_symmetric=SymmetricTensor(zeta=69.4, eta=0.60),  # zeta in ppm
)

spin_systems = [
    SpinSystem(sites=[Si29_1]),
    SpinSystem(sites=[Si29_2]),
    SpinSystem(sites=[Si29_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=[
        SpectralDimension(
            count=2048,
            spectral_width=25000,  # in Hz
            reference_offset=-10000,  # in Hz
            label=r"$^{29}$Si resonances",
        )
    ],
)

# A graphical representation of the method object.
plt.figure(figsize=(4, 2))
method.plot()
plt.show()
```

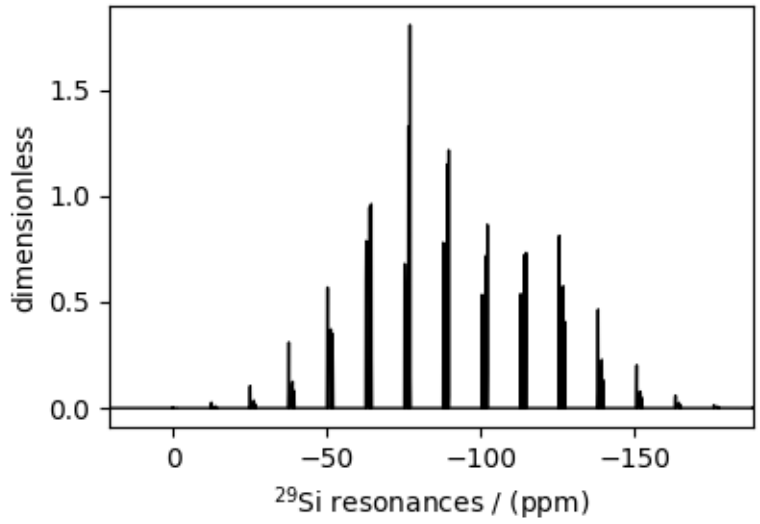


Create the Simulator object and add method and spin system objects, and run.

```
sim = Simulator(spin_systems=spin_systems, methods=[method])
```

Simulate the spectrum.

```
sim.run()

# The plot of the simulation before signal processing.
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(sim.methods[0].simulation.real, color="black", linewidth=1)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```



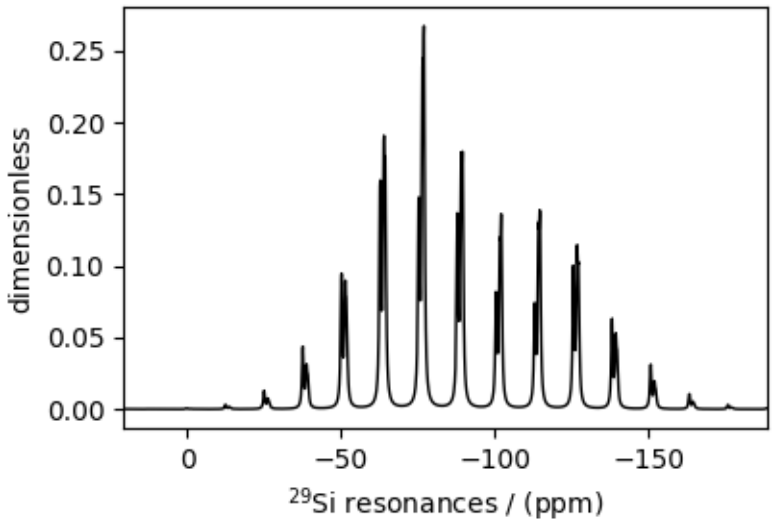
Add post-simulation signal processing.

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

(continues on next page)

(continued from previous page)

```
# The plot of the simulation after signal processing.
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(processed_dataset.real, color="black", linewidth=1)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 1.137 seconds)

Using Custom Isotopes

Simulating spectra of custom isotopes.

Mrsimulator includes the capability for simulating isotopes with user-defined attributes. In this example, we show how to define a custom isotopes and simulate spectra using custom isotopes.

```
from mrsimulator.spin_system.isotope import Isotope
from mrsimulator import Site, SpinSystem, Simulator
from mrsimulator.method import SpectralDimension
from mrsimulator.method.lib import BlochDecayCTSpectrum

import matplotlib.pyplot as plt
```

First, we register a new isotope symbol with custom attributes using the *register()* (page 480) method. The new symbol cannot match any real isotope symbols, and re-calling register on a custom symbol will update the stored attributes of that isotope.

```
Isotope.register(
    symbol="custom",
    spin_multiplicity=4,  # Spin of I=3/2
    gyromagnetic_ratio=12.3,
```

(continues on next page)

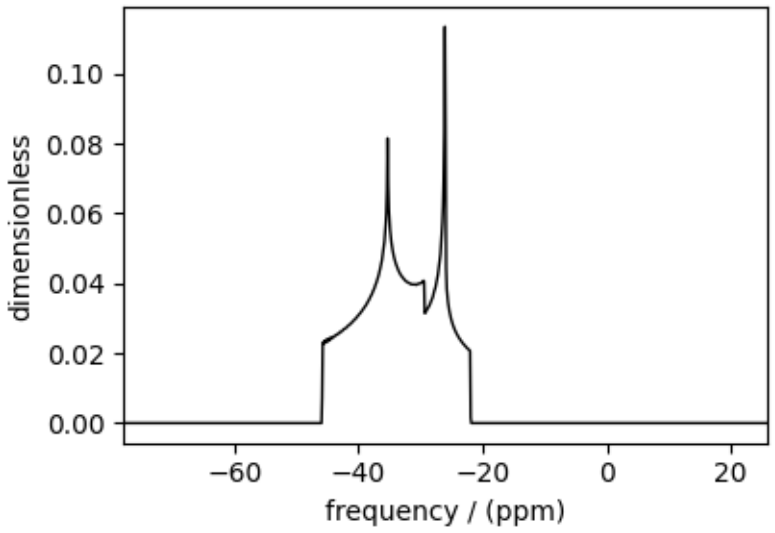
(continued from previous page)

```
    quadrupole_moment=0.045,
)
```

Create `Site` and *`Method`* (page 403) objects using the new isotope symbol. The syntax is the same as any other isotope.

```
site = Site(
    isotope="custom",
    isotropic_chemical_shift=-30,
    quadrupolar={"Cq": 1.3e6, "eta": 0.5},
)
sys = SpinSystem(sites=[site])

mth = BlochDecayCTSpectrum(
    channels=["custom"],
    spectral_dimensions=[
        SpectralDimension(spectral_width=12e3, reference_offset=-3000)
    ],
)
```

```
sim = Simulator(spin_systems=[sys], methods=[mth])
sim.run()

plt.figure(figsize=(4.25, 3))
plt.subplot(projection="csdm")
plt.plot(sim.methods[0].simulation.real, color="black", linewidth=1)
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.250 seconds)

Selective Excitations using Custom Isotopes

Simulating spectra of custom isotopes

The isotope register method has the `copy_from` argument which can be used to create individually addressable channels which the same isotope attributes. This functionality can be used to emulate selective-excitation experiments by querying specific transitions on different sites

```
from mrsimulator.spin_system.isotope import Isotope
from mrsimulator import Site, SpinSystem, Coupling, Simulator, Method
from mrsimulator.method import SpectralDimension, SpectralEvent
from mrsimulator.method.lib import BlochDecaySpectrum
from mrsimulator.method.query import TransitionQuery
import mrsimulator.signal_processor as sp

from pprint import pprint
import matplotlib.pyplot as plt
```

Below, we show how custom isotopes can be copied from existing isotopes, and how these custom isotopes can be used to simulate a selective-excitation spectrum of protons. First, two new isotopes – `"1H-a"` and `"1H-b"` – are registered from the known proton isotope by using the `copy_from` keyword argument.

```
Isotope.register("1H-a", copy_from="1H")
Isotope.register("1H-b", copy_from="1H")
```

Next, we create two site objects using the previously registered `"1H-a"` and `"1H-b"` isotope symbols. The equivalent proton system (without custom isotopes) is also constructed as a comparison.

```
site_a = Site(isotope="1H", isotropic_chemical_shift=-0.5)
site_b = Site(isotope="1H", isotropic_chemical_shift=-2.0)
coupling_ab = Coupling(site_index=[0, 1], isotropic_j=48)
sys = SpinSystem(sites=[site_a, site_b], couplings=[coupling_ab])  # proton system

# Create 1D BlochDecaySpectrum for proton
spec_dim = SpectralDimension(count=10000, spectral_width=2000, reference_offset=-500)
mth = BlochDecaySpectrum(channels=["1H"], spectral_dimensions=[spec_dim])

# Create Simulator object for coupled proton system
sim = Simulator(spin_systems=[sys], methods=[mth])


# Create couple proton system from custom isotopes
site_a_custom = Site(isotope="1H-a", isotropic_chemical_shift=-0.5)
site_b_custom = Site(isotope="1H-b", isotropic_chemical_shift=-2.0)
sys_custom = SpinSystem(sites=[site_a_custom, site_b_custom], couplings=[coupling_ab])

# Create two BlochDecaySpectrum methods with custom isotope channels
mth_a = BlochDecaySpectrum(channels=["1H-a"], spectral_dimensions=[spec_dim])
mth_b = BlochDecaySpectrum(channels=["1H-b"], spectral_dimensions=[spec_dim])
```

Create and run two simulator objects for the simulating the regular and selective excitation proton spectra.

```
sim = Simulator(spin_systems=[sys], methods=[mth])
sim_custom = Simulator(spin_systems=[sys_custom], methods=[mth_a, mth_b])
```

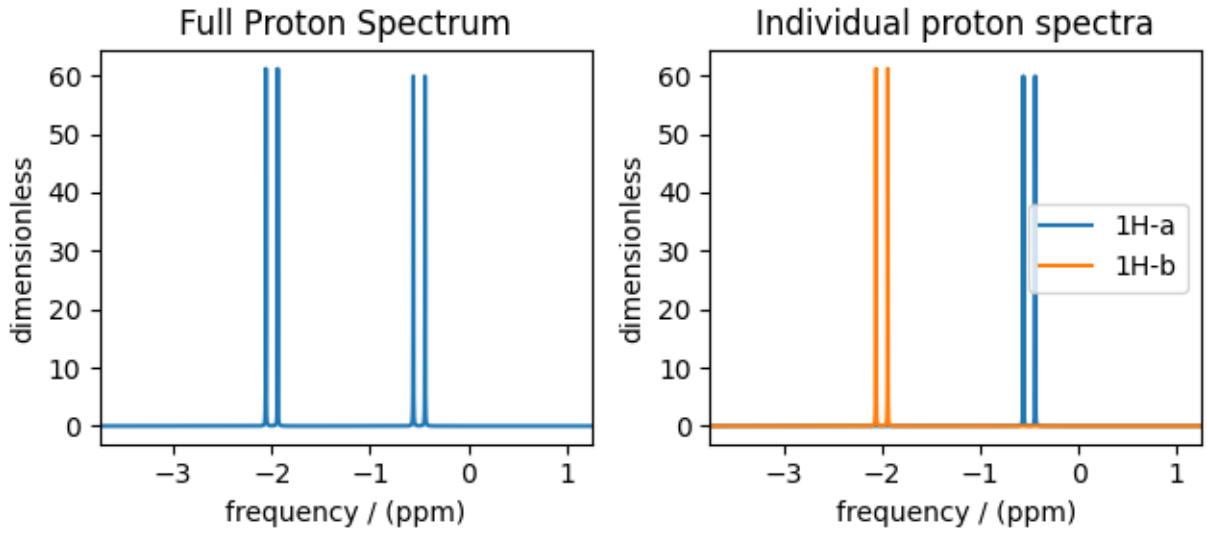
(continues on next page)

(continued from previous page)

```
# Run both the simulations
sim.run()
sim_custom.run()
```

Apply post-simulation signal processing and plot the spectra. Notice how the full proton spectrum (right) is the summation of the two individual proton spectra (left), and how the J-couplings are still present in the selective excitation spectra. Make processor for adding line broadening to spectra

```
processor = sp.SignalProcessor(
    operations=[sp.FFT(), sp.apodization.Exponential(FWHM="1 Hz"), sp.IFFT()]
)
spec_proton = processor.apply_operations(sim.methods[0].simulation).real
spec_a = processor.apply_operations(sim_custom.methods[0].simulation).real
spec_b = processor.apply_operations(sim_custom.methods[1].simulation).real

fig, ax = plt.subplots(1, 2, figsize=(6.5, 3), subplot_kw={"projection": "csdm"})

ax[0].plot(spec_proton)
ax[0].set_title("Full Proton Spectrum")
ax[1].plot(spec_a, label="1H-a")
ax[1].plot(spec_b, label="1H-b")
ax[1].legend()
ax[1].set_title("Individual proton spectra")

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



Custom isotopes can be used in conjunction with custom method objects to query different transitions on custom isotopes. Below, we create a custom Method which simulates a coupled 1H-13C spectrum where the carbon site simultaneous undergoes a transitions with only one of the proton sites.

```
proton_a_custom = Site(isotope="1H-a", isotropic_chemical_shift=0.0)
proton_b_custom = Site(isotope="1H-b", isotropic_chemical_shift=20)
```

(continues on next page)

(continued from previous page)

```
carbon = Site(isotope="13C", isotropic_chemical_shift=-40)

# J- and dipolar-couplings for the system
coupling_ab = Coupling(site_index=[0, 1], isotropic_j=48)  # 1H-a, 1H-b
coupling_ac = Coupling(site_index=[0, 2], dipolar={"D": 2000})  # 1H-a, 13C
coupling_bc = Coupling(site_index=[1, 2], dipolar={"D": 1000})  # 1H-b, 13C

sys_custom = SpinSystem(
    sites=[proton_a_custom, proton_b_custom, carbon],
    couplings=[coupling_ab, coupling_ac, coupling_bc],
)
```

Next, we create the Method object which has three different channels; the first channel is the observed nucleus, here 13C, and the other two are for the custom proton isotopes.

```
mth_custom = Method(
    channels=["13C", "1H-a", "1H-b"],
    spectral_dimensions=[
        SpectralDimension(
            reference_offset=-9000,
            events=[
                SpectralEvent(
                    transition_queries=[
                        TransitionQuery(
                            ch1={"P": [-1]}, ch2={"P": [-1]}, ch3={"P": [0]}
                        )
                    ]
                )
            ],
        )
    ],
)
pprint(mth_custom.get_transition_pathways(sys_custom))
```

```
[|-0.5, -0.5, -0.5⟩⟨0.5, -0.5, 0.5|, weight=(1+0j),
 |-0.5, 0.5, -0.5⟩⟨0.5, 0.5, 0.5|, weight=(1+0j)]
```

The equivalent SpinSystem and Method without custom isotopes or selective excitations as a comparison.

```
proton_a = Site(isotope="1H", isotropic_chemical_shift=0.0)
proton_b = Site(isotope="1H", isotropic_chemical_shift=-20)
carbon = Site(isotope="13C", isotropic_chemical_shift=-40)

# J- and dipolar-couplings for the system
coupling_ab = Coupling(site_index=[0, 1], isotropic_j=48)  # 1H-a, 1H-b
coupling_ac = Coupling(site_index=[0, 2], dipolar={"D": 2000})  # 1H-a, 13C
coupling_bc = Coupling(site_index=[1, 2], dipolar={"D": 1000})  # 1H-b, 13C

sys = SpinSystem(
    sites=[proton_a, proton_b, carbon],
    couplings=[coupling_ab, coupling_ac, coupling_bc],
)
```

(continues on next page)

(continued from previous page)

```
mth = Method(
    channels=["13C", "1H"],
    spectral_dimensions=[
        SpectralDimension(
            reference_offset=-9000,
            events=[
                SpectralEvent(
                    transition_queries=[
                        TransitionQuery(
                            ch1={"P": [-1]},
                            ch2={"P": [-1]},
                        )
                    ]
                )
            ],
        )
    ],
)
pprint(mth.get_transition_pathways(sys))
```

```
[|-0.5, -0.5, -0.5⟩⟨-0.5, 0.5, 0.5|, weight=(1+0j),
 |-0.5, -0.5, -0.5⟩⟨0.5, -0.5, 0.5|, weight=(1+0j),
 |-0.5, 0.5, -0.5⟩⟨0.5, 0.5, 0.5|, weight=(1+0j),
 |0.5, -0.5, -0.5⟩⟨0.5, 0.5, 0.5|, weight=(1+0j)]
```

Create and run Simulator objects for the selective and non-selective spectra.

```
sim_custom = Simulator(spin_systems=[sys_custom], methods=[mth_custom])
sim = Simulator(spin_systems=[sys], methods=[mth])

sim_custom.run()
sim.run()
```

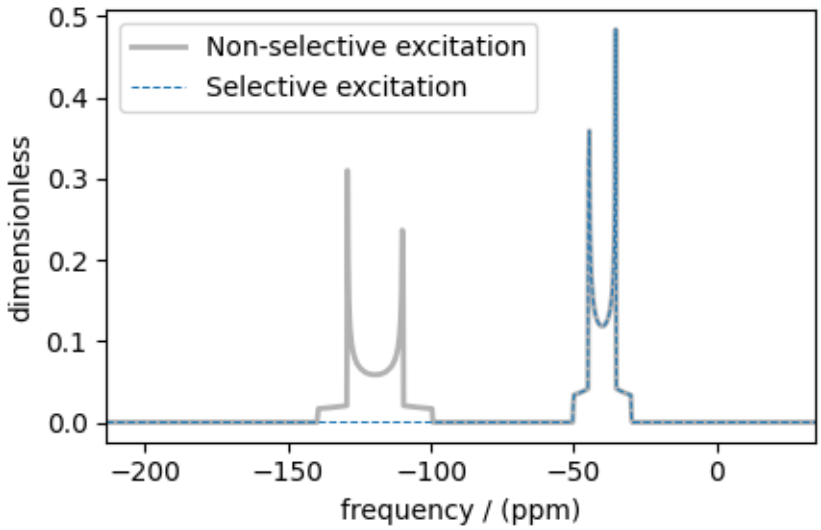
Plot the simulated spectra

```
plt.figure(figsize=(4.5, 3))
plt.subplot(projection="csdm")
plt.plot(
    sim.methods[0].simulation.real,
    color="black",
    alpha=0.3,
    linewidth=2,
    label="Non-selective excitation",
)
plt.plot(
    sim_custom.methods[0].simulation.real,
    linestyle="--",
    alpha=1.0,
    linewidth=0.75,
    label="Selective excitation",
)
```

(continues on next page)

(continued from previous page)

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



Total running time of the script: (0 minutes 1.015 seconds)

Influence of 14N on 13C NMR MAS spectra of glycine

The alpha-carbon resonance of glycine, 13C (I=1/2), attached to 14N (I=1). The 14N quadrupolar tensor parameters were obtained from Hexem *et al.*[1]

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

from mrsimulator import Simulator, SpinSystem, Site, Coupling
from mrsimulator.method.lib import BlochDecaySpectrum
from mrsimulator import signal_processor as sp
from mrsimulator.spin_system.tensors import SymmetricTensor
from mrsimulator.method import SpectralDimension
```

Create a 13C-14N coupled spin system.

```
spin_system = SpinSystem(
    sites=[
        Site(isotope="13C", isotropic_chemical_shift=0.0),
        Site(
            isotope="14N",
            isotropic_chemical_shift=0,  # in ppm
            quadrupolar=SymmetricTensor(
                Cq=1.18e6,  # in Hz
```

(continues on next page)

[1] Hexem, J. G., Frey, M. H., and Opella, S. J., Influence of ^{14}N on ^{13}C NMR Spectra of Solids, J. Am. Chem. Soc., 1981, **103**, 224-226. DOI: 10.1021/ja00391a057

(continued from previous page)

```
                eta=0.54,
                alpha=0,
                beta=5 * np.pi / 180,
                gamma=0,
            ),
        ),
    ],
    couplings=[Coupling(site_index=[0, 1], dipolar=SymmetricTensor(D=-660.2))],
)
```

Create a BlochDecaySpectrum method.

```
method = BlochDecaySpectrum(
    channels=["13C"],
    magnetic_flux_density=3.5338,  # in T
    rotor_frequency=12000,  # in Hz
    spectral_dimensions=[SpectralDimension(count=2048, spectral_width=200)],
)
```

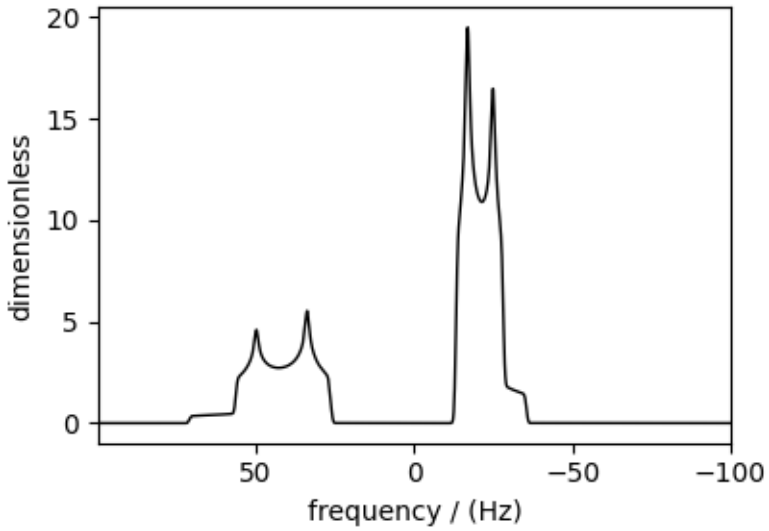
Create the Simulator object and add the method and the spin system object.

```
sim = Simulator(spin_systems=[spin_system], methods=[method])
sim.config.integration_volume = "hemisphere"
sim.run()
```

Add post-simulation signal processing.

```
processor = sp.SignalProcessor(
    operations=[
        sp.IFFT(),
        sp.apodization.Gaussian(FWHM="1 Hz"),
        sp.FFT(),
    ]
)
processed_dataset = processor.apply_operations(dataset=sim.methods[0].simulation)
processed_dataset.dimensions[0].to("Hz")
```

```
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(processed_dataset.real, color="black", linewidth=1)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.309 seconds)

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

^{33}S (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.pyplot as plt

from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator import signal_processor as sp
from mrsimulator.method.lib import BlochDecayCTSpectrum
from mrsimulator.spin_system.tensors import SymmetricTensor
from mrsimulator.method import SpectralDimension
```

Create the spin system

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

Create a central transition selective Bloch decay spectrum method.

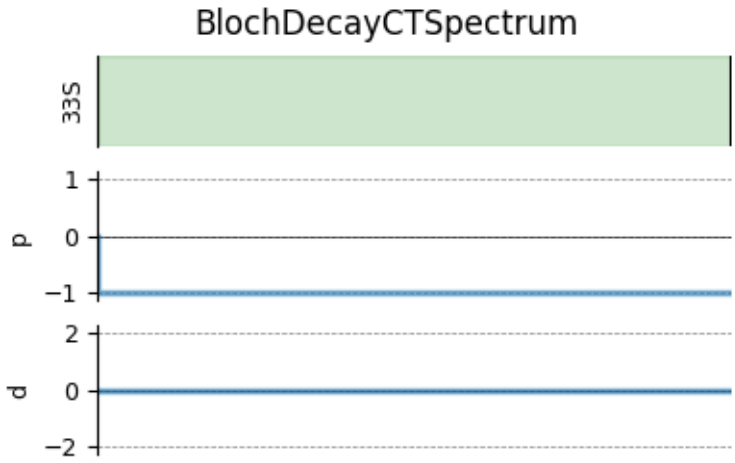
```
method = BlochDecayCTSpectrum(
    channels=["33S"],
    magnetic_flux_density=21.14,  # in T
```

(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*, 309–316. DOI: 10.1021/jp908206c

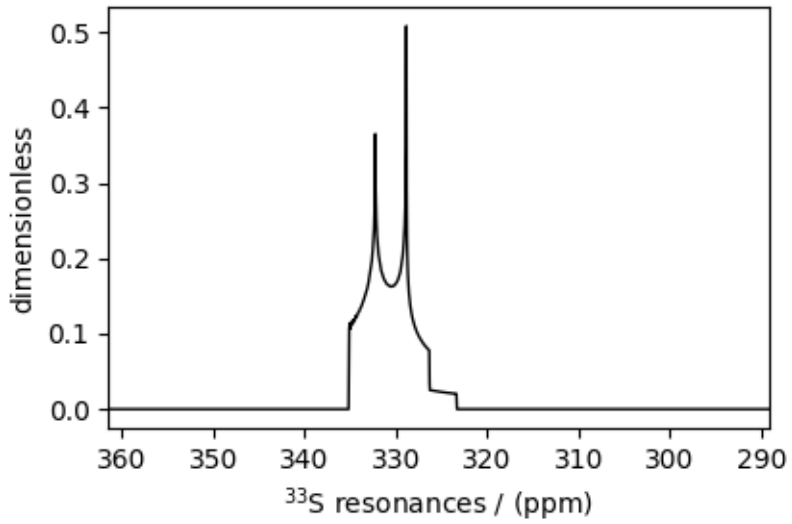
(continued from previous page)

```
    rotor_frequency=14000,  # in Hz
    spectral_dimensions=[
        SpectralDimension(
            count=2048,
            spectral_width=5000,  # in Hz
            reference_offset=22500,  # in Hz
            label=r"$^{33}$S resonances",
        )
    ],
)

# A graphical representation of the method object.
plt.figure(figsize=(4, 2.5))
method.plot()
plt.show()
```



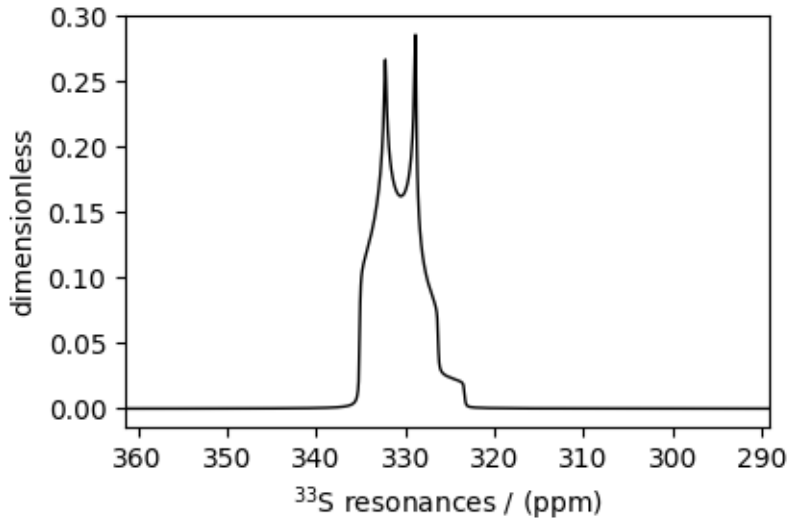
Create the Simulator object and add method and spin system objects.

```
sim = Simulator(spin_systems=[spin_system], methods=[method])
sim.run()

# The plot of the simulation before signal processing.
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(sim.methods[0].simulation.real, color="black", linewidth=1)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```



Add post-simulation signal processing.

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

# The plot of the simulation after signal processing.
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(processed_dataset.real, color="black", linewidth=1)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.696 seconds)

Coesite, ^{17}O (I=5/2)

^{17}O (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.pyplot as plt

from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator import signal_processor as sp
from mrsimulator.method.lib import BlochDecayCTSpectrum
from mrsimulator.spin_system.tensors import SymmetricTensor
from mrsimulator.method import SpectralDimension
```

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=SymmetricTensor(Cq=6.05e6, eta=0.000),
)
O17_2 = Site(
    isotope="17O",
    isotropic_chemical_shift=41,
    quadrupolar=SymmetricTensor(Cq=5.43e6, eta=0.166),
)
O17_3 = Site(
    isotope="17O",
    isotropic_chemical_shift=57,
    quadrupolar=SymmetricTensor(Cq=5.45e6, eta=0.168),
)
O17_4 = Site(
    isotope="17O",
    isotropic_chemical_shift=53,
    quadrupolar=SymmetricTensor(Cq=5.52e6, eta=0.169),
)
O17_5 = Site(
    isotope="17O",
    isotropic_chemical_shift=58,
    quadrupolar=SymmetricTensor(Cq=5.16e6, eta=0.292),
)

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

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. Here we are iterating over both the *sites* and *abundance* list concurrently using a list comprehension to construct a list of SpinSystems

[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

```
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)]
```

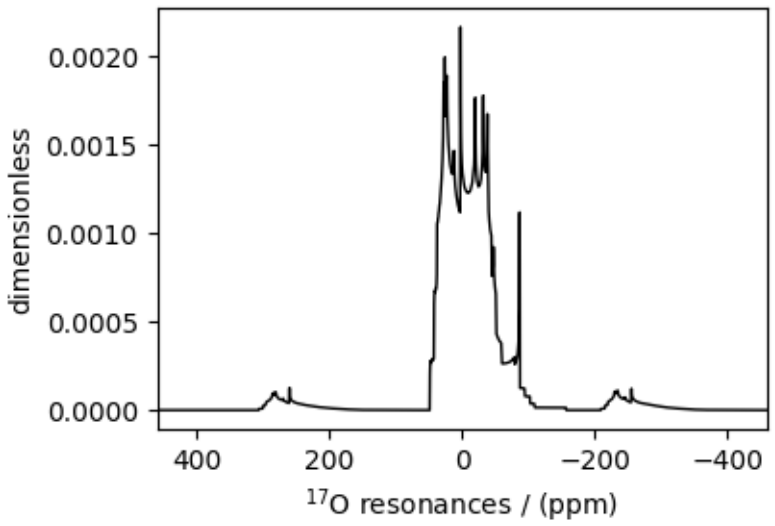
Create a central transition selective Bloch decay spectrum method.

```
method = BlochDecayCTSpectrum(
    channels=["17O"],
    rotor_frequency=14000,  # in Hz
    spectral_dimensions=[
        SpectralDimension(
            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.

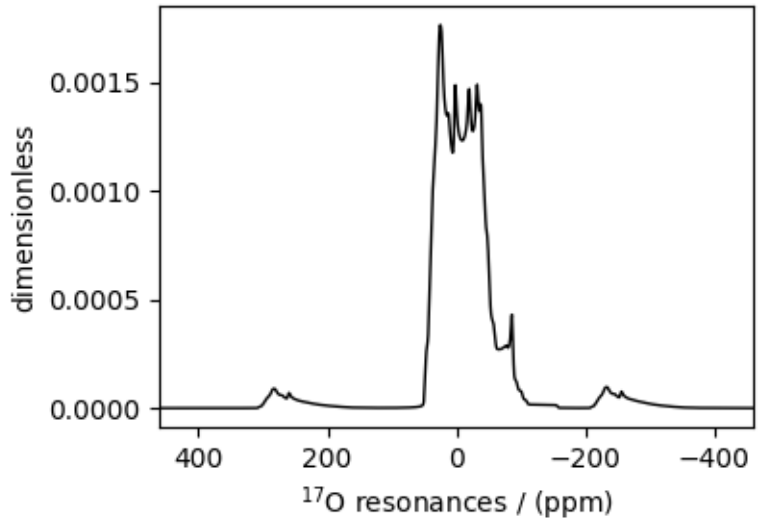
Create the Simulator object and add method and spin system objects.

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

# The plot of the simulation before signal processing.
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(sim.methods[0].simulation.real, color="black", linewidth=1)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```



Add post-simulation signal processing.

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

# The plot of the simulation after signal processing.
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(processed_dataset.real, color="black", linewidth=1)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.547 seconds)

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

^{17}O (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 used by Yamada *et al* is different from the representation used in mrsimulator. The resulting simulation might not resemble the published spectrum.

[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

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

from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.method.lib import BlochDecayCTSpectrum
from mrsimulator.spin_system.tensors import SymmetricTensor
from mrsimulator.method import SpectralDimension
```

Create the spin system.

```
site = Site(
    isotope="17O",
    isotropic_chemical_shift=320,  # in ppm
    shielding_symmetric=SymmetricTensor(zeta=376.667, eta=0.345),
    quadrupolar=SymmetricTensor(
        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])
```

Create a central transition selective Bloch decay spectrum method.

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

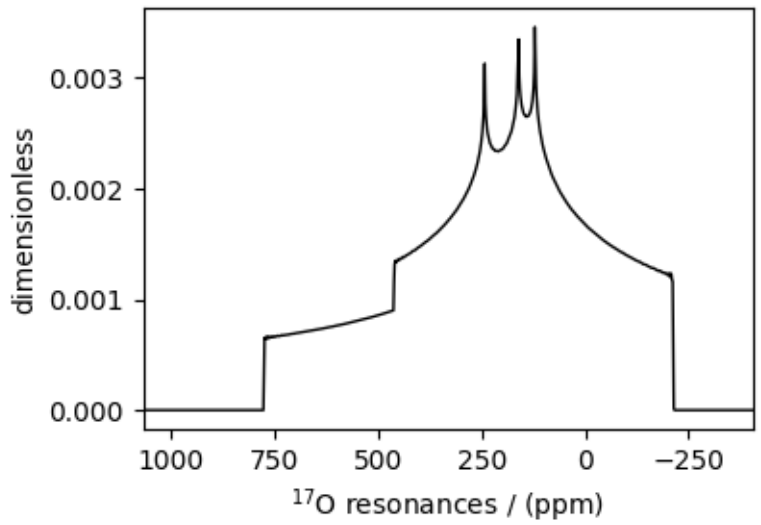
Create the Simulator object and add method and spin system objects.

```
sim = Simulator(spin_systems=[spin_system], methods=[method])

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

# The plot of the simulation before signal processing.
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
```

(continues on next page)

(continued from previous page)

```
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 0.271 seconds)

Arbitrary spin transition (single-quantum)

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

The mrsimulator built-in library methods, BlochDecaySpectrum and BlochDecayCTSpectrum, simulate spectrum from single quantum transitions or central transition selective transition, respectively. In this example, we show how you can simulate any arbitrary transition using the generic Method object.

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

from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.method import Method, SpectralDimension, SpectralEvent
from mrsimulator.spin_system.tensors import SymmetricTensor
```

Create a single-site arbitrary spin system.

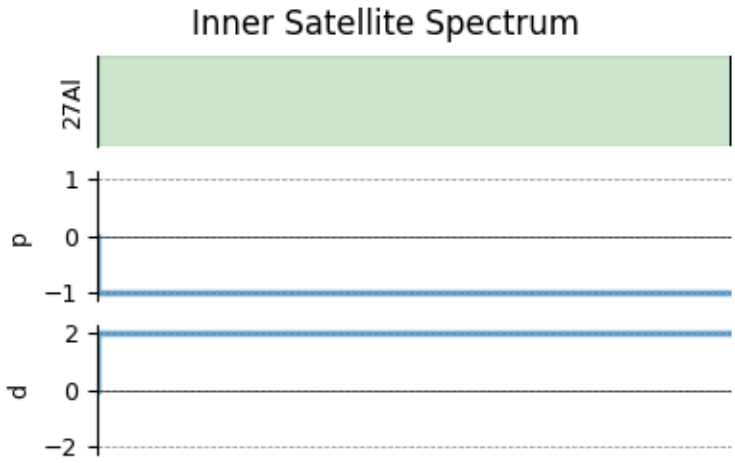
```
site = Site(
    name="27Al",
    isotope="27Al",
    isotropic_chemical_shift=35.7,  # in ppm
    quadrupolar=SymmetricTensor(Cq=5.959e6, eta=0.32),  # Cq is in Hz
)
spin_system = SpinSystem(sites=[site])
```

Selecting spin transitions for simulation

The arguments of the Method object are the same as the arguments of the BlochDecaySpectrum method; however, unlike a BlochDecaySpectrum method, the *SpectralDimension* (page 410) object in Method contains an additional argument—*events*.

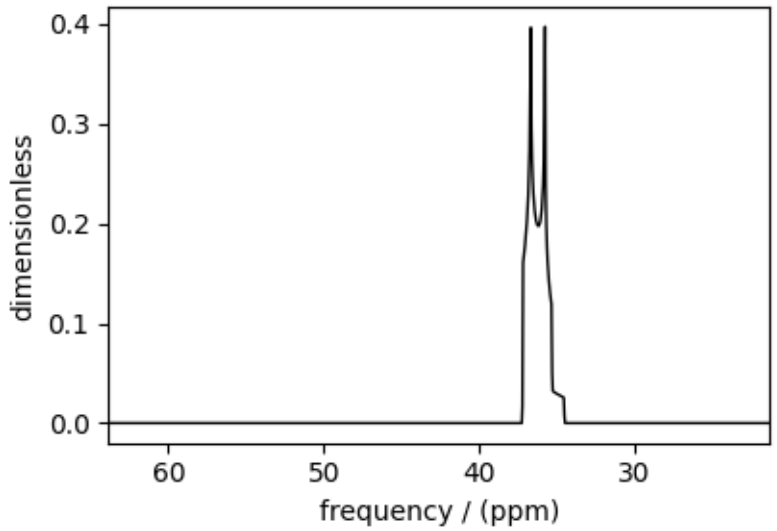
The *Events* (page 412) object is a collection of attributes, which are local to the event. It is here where we define *transition_queries* to select one or more transitions for simulating the spectrum. Recall, a TransitionQuery object holds a channel wise SymmetryQuery. For more information, refer to the *Spin Transition Symmetry Functions* (page 82). In this example, we use the `P=-1` and `D=2` attributes of SymmetryQuery, to select the satellite transition, $|-1/2\rangle \rightarrow |-3/2\rangle$.

```
method = Method(
    name="Inner Satellite Spectrum",
    channels=["27Al"],
    magnetic_flux_density=21.14,  # in T
    rotor_frequency=np.inf,  # in Hz
    rotor_angle=54.7356 * np.pi / 180,  # in rads
    spectral_dimensions=[
        SpectralDimension(
            count=1024,
            spectral_width=1e4,  # in Hz
            reference_offset=1e4,  # in Hz
            events=[
                SpectralEvent(
                    transition_queries=[
                        {"ch1": {"P": [-1], "D": [2]}},  # inner satellite
                    ]
                )
            ],
        )
    ],
)

# A graphical representation of the method object.
plt.figure(figsize=(4, 2.5))
method.plot()
plt.show()
```



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

```
sim = Simulator(spin_systems=[spin_system], methods=[method])
sim.run()

# The plot of the simulation before signal processing.
plt.figure(figsize=(4.25, 3.0))
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

Similarly, you may add another transition query to select to select additional transitions. Consider the following transitions with respective P and D values.

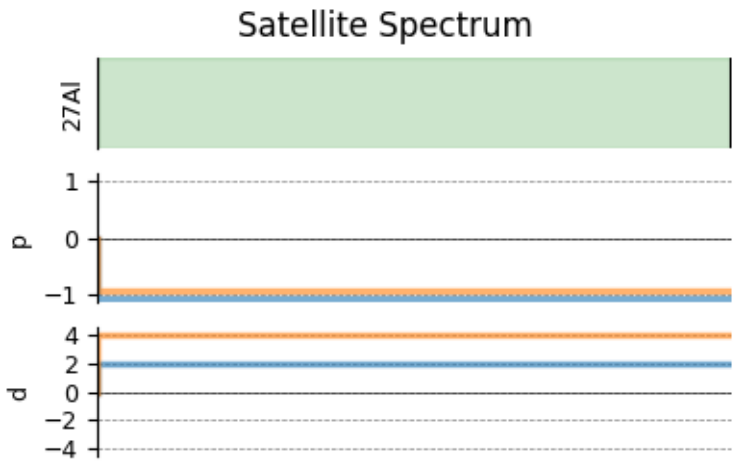
- $|-1/2\rangle \rightarrow |-3/2\rangle \;\; (P=-1, D=2)$
- $|-3/2\rangle \rightarrow |-5/2\rangle \;\; (P=-1, D=4)$

```
method2 = Method(
    name="Satellite Spectrum",
    channels=["27Al"],
    magnetic_flux_density=21.14,  # in T
    rotor_frequency=np.inf,  # in Hz
    rotor_angle=54.7356 * np.pi / 180,  # in rads
    spectral_dimensions=[
        SpectralDimension(
            count=1024,
            spectral_width=1e4,  # in Hz
            reference_offset=1e4,  # in Hz
            events=[
                SpectralEvent(
                    transition_queries=[
```

(continues on next page)

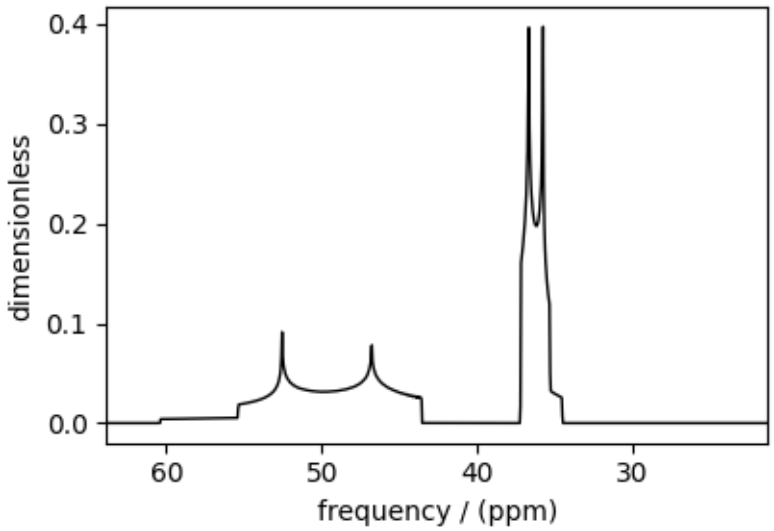
(continued from previous page)

```
                        {"ch1": {"P": [-1], "D": [2]}},  # inter satellite
                        {"ch1": {"P": [-1], "D": [4]}},  # outer satellite
                    ]
                )
            ],
        )
    ],
)

# A graphical representation of the method object.
plt.figure(figsize=(4, 2.5))
method2.plot()
plt.show()
```



Update the method object in the Simulator object and re-simulate

```
sim.methods[0] = method2
sim.run()

# The plot of the simulation before signal processing.
plt.figure(figsize=(4.25, 3.0))
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 0.802 seconds)

Arbitrary spin transition (multi-quantum)

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

Simulate a triple quantum spectrum.

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

from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.method import Method, SpectralDimension, SpectralEvent
from mrsimulator.spin_system.tensors import SymmetricTensor
```

Create a single-site arbitrary spin system.

```
site = Site(
    name="27Al",
    isotope="27Al",
    isotropic_chemical_shift=35.7,  # in ppm
    quadrupolar=SymmetricTensor(Cq=2.959e6, eta=0.98),  # Cq is in Hz
)
spin_system = SpinSystem(sites=[site])
```

Selecting the triple-quantum transition

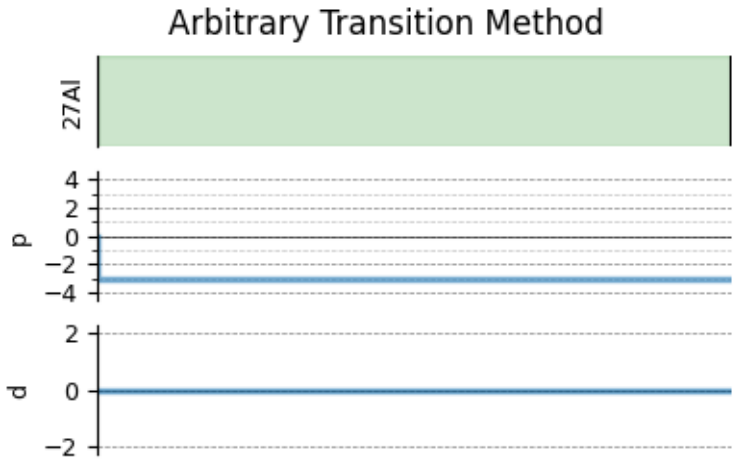
For single-site spin-5/2 spin system, there are three triple-quantum transition

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

To select one or more triple-quantum transitions, assign the respective value of P and D to the symmetry query object of *transition_ queries*. Refer to the *Spin Transition Symmetry Functions* (page 82) for details.

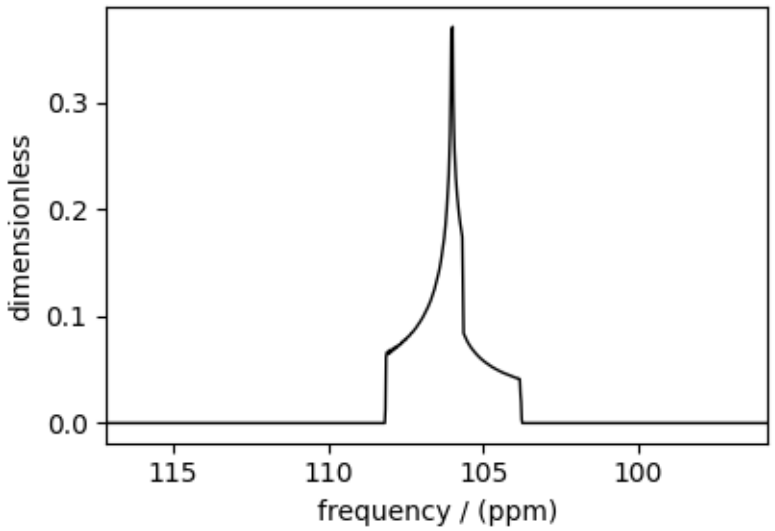
Here, we select the symmetric triple-quantum transition.

```
method = Method(
    name="Arbitrary Transition Method",
    channels=["27Al"],
    magnetic_flux_density=21.14,  # in T
    rotor_frequency=np.inf,  # in Hz
    rotor_angle=54.7356 * np.pi / 180,  # in rads
    spectral_dimensions=[
        SpectralDimension(
            count=1024,
            spectral_width=5e3,  # in Hz
            reference_offset=2.5e4,  # in Hz
            events=[
                SpectralEvent(
                    # symmetric triple quantum transitions
                    transition_queries=[{"ch1": {"P": [-3], "D": [0]}}]
                ),
            ],
        )
    ],
)

# A graphical representation of the method object.
plt.figure(figsize=(4, 2.5))
method.plot()
plt.show()
```



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

```
sim = Simulator(spin_systems=[spin_system], methods=[method])
sim.run()

# The plot of the simulation before signal processing.
plt.figure(figsize=(4.25, 3.0))
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 0.411 seconds)

Coupled spin-1/2 (Static dipolar spectrum)

^{13}C-^{1}H static dipolar coupling simulation.

```
import matplotlib.pyplot as plt

from mrsimulator import Simulator, SpinSystem, Site, Coupling
from mrsimulator.method.lib import BlochDecaySpectrum
from mrsimulator import signal_processor as sp
from mrsimulator.spin_system.tensors import SymmetricTensor
from mrsimulator.method import SpectralDimension
```

Create a 13C-1H coupled spin system.

```
spin_system = SpinSystem(
    sites=[
        Site(isotope="13C", isotropic_chemical_shift=0.0),
        Site(isotope="1H", isotropic_chemical_shift=0.0),
    ],
    couplings=[Coupling(site_index=[0, 1], dipolar=SymmetricTensor(D=-2e4))],
)
```

Create a BlochDecaySpectrum method.

```
method = BlochDecaySpectrum(
    channels=["13C"],
    magnetic_flux_density=9.4,  # in T
    rotor_frequency=0,  # in Hz
    rotor_angle=0,  # in rads
    spectral_dimensions=[SpectralDimension(count=2048, spectral_width=8.0e4)],
)
```

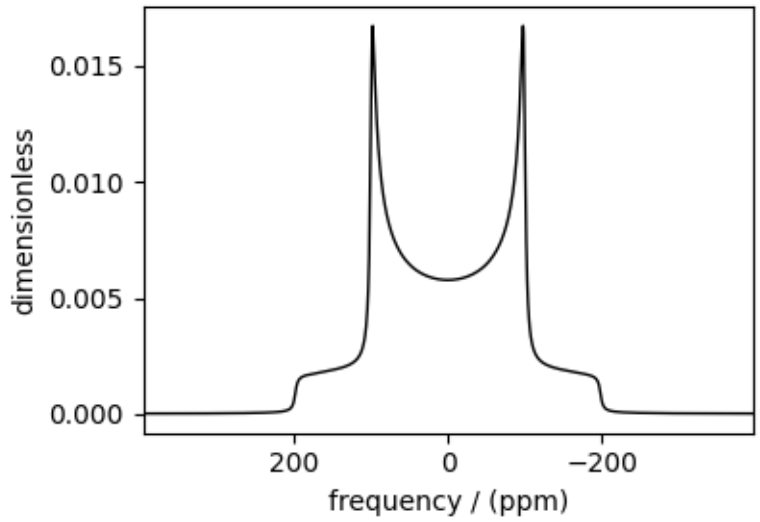
Create the Simulator object and add the method and the spin system object.

```
sim = Simulator(spin_systems=[spin_system], methods=[method])
sim.run()
```

Add post-simulation signal processing.

```
processor = sp.SignalProcessor(
    operations=[
        sp.IFFT(),
        sp.apodization.Exponential(FWHM="500 Hz"),
        sp.FFT(),
    ]
)
processed_dataset = processor.apply_operations(dataset=sim.methods[0].simulation)
```

```
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(processed_dataset.real, color="black", linewidth=1)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.254 seconds)

Coupled spin-1/2 (CSA + heteronuclear dipolar + J-couplings)

^{13}C-^{1}H sideband simulation

The following simulation is an example by Edén[1] from *Computer Simulations in Solid-State NMR.III.Powder Averaging*. The simulation consists of sideband spectra from a 13C-1H coupled spin system computed at various spinning frequencies with different relative tensor orientations between the nuclear shielding and dipolar interaction tensors.

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

from mrsimulator import Simulator, SpinSystem, Site, Coupling
from mrsimulator.method.lib import BlochDecaySpectrum
from mrsimulator import signal_processor as sp
from mrsimulator.spin_system.tensors import SymmetricTensor
from mrsimulator.method import SpectralDimension
```

Here, we create three 13C-1H spin systems with different relative orientations between the shielding and dipolar tensors. The Euler angle orientations $\alpha = \gamma = 0$ and β values are listed below.

```
beta_orientation = [np.pi / 6, 5 * np.pi / 18, np.pi / 2]

# The `variable` spin_systems is a list of three coupled 13C-1H spin systems with
# different relative shielding and dipolar tensor orientation.
spin_systems = [
    SpinSystem(
        sites=[
            Site(
                isotope="13C",
                isotropic_chemical_shift=0.0,  # in ppm
                shielding_symmetric=SymmetricTensor(
                    zeta=18.87562,  # in ppm
                    eta=0.4,
                    beta=beta,
                ),
            ),
            Site(
                isotope="1H",
                isotropic_chemical_shift=0.0,  # in ppm
            ),
        ],
        couplings=[
            Coupling(
                site_index=[0, 1], isotropic_j=200.0, dipolar=SymmetricTensor(D=-2.1e4)
            )
        ],
    )
    for beta in beta_orientation
]
```

Next, we create methods to simulate the sideband manifolds for the above spin systems at four spinning rates: 3 kHz, 5 kHz, 8 kHz, 12 kHz.

[1] Edén, M. Computer Simulations in Solid-State NMR. III. Powder Averaging, Concepts in Magnetic Resonance Part A, Vol. 18A(1) 24–55 (2003). DOI: doi.org/10.1002/cmr.a.10065

```
spin_rates = [3e3, 5e3, 8e3, 12e3]  # in Hz

# The variable `methods` is a list of four BlochDecaySpectrum methods.
methods = [
    BlochDecaySpectrum(
        channels=["13C"],
        magnetic_flux_density=9.4,  # in T
        rotor_frequency=vr,  # in Hz
        spectral_dimensions=[SpectralDimension(count=2048, spectral_width=8.0e4)],
    )
    for vr in spin_rates
]
```

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

```
sim = Simulator(spin_systems=spin_systems, methods=methods)
sim.config.integration_volume = "hemisphere"  # set averaging to hemisphere
# config to decompose spectrum to individual spin systems.
sim.config.decompose_spectrum = "spin_system"
```

The run command will simulate twelve spectra corresponding to the three spin systems evaluated at four different methods (spinning speeds).

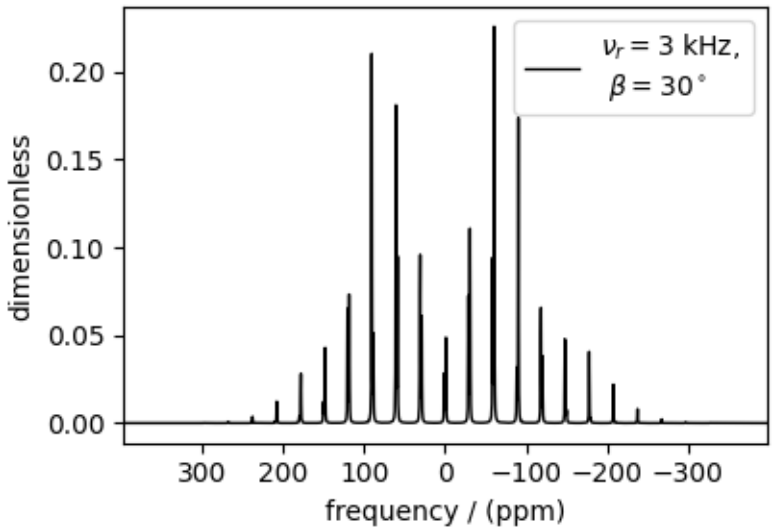
```
sim.run()
```

Add post-simulation signal processing.

```
processor = sp.SignalProcessor(
    operations=[
        sp.IFFT(),
        sp.apodization.Exponential(FWHM="50 Hz"),
        sp.FFT(),
    ]
)
# apply the same post-simulation processing to all the twelve simulations.
processed_dataset = [
    processor.apply_operations(dataset=method.simulation) for method in sim.methods
]
```

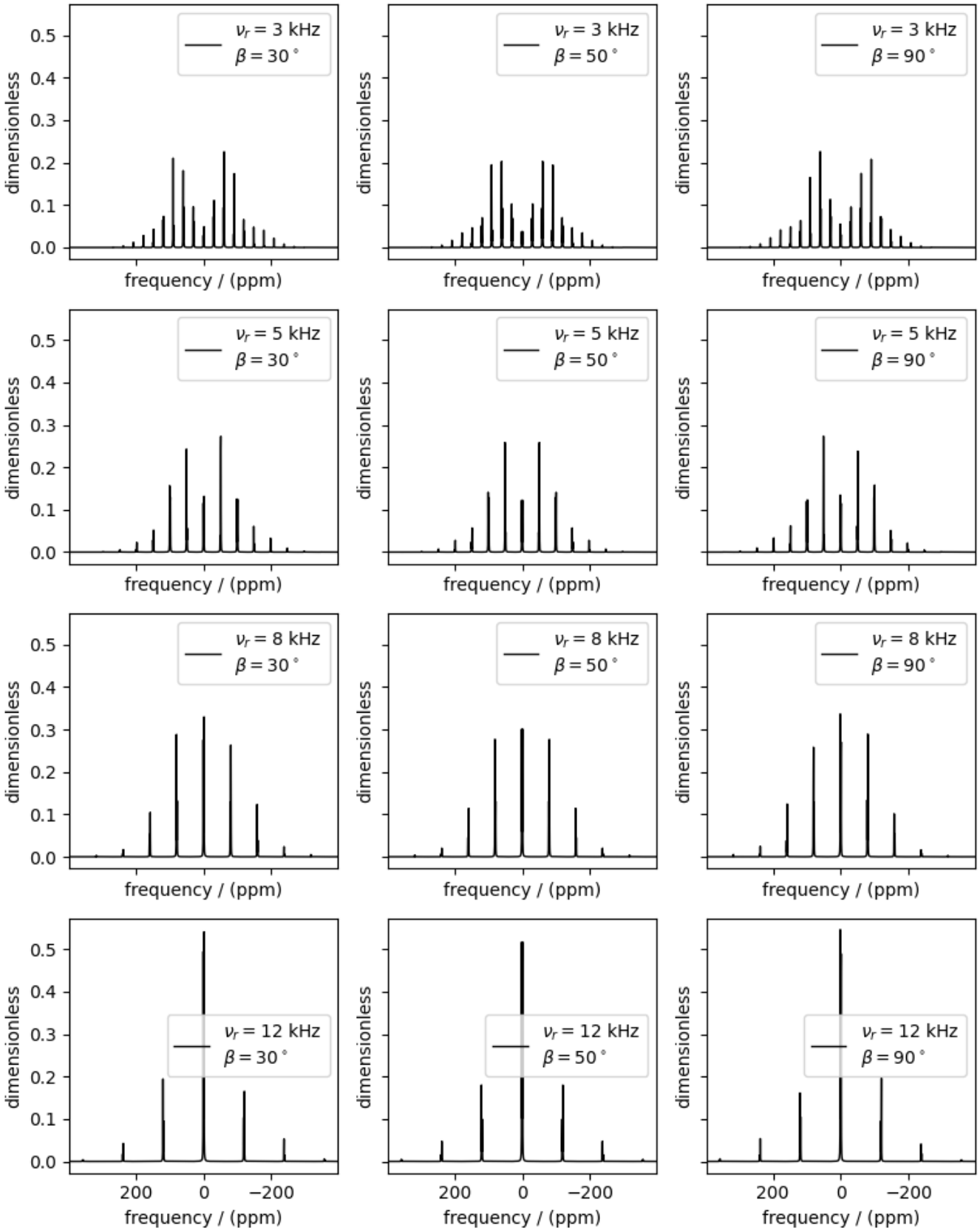
Let's first plot a single simulation, the one corresponding to a relative orientation of $\beta = 30°$ between the shielding and dipolar tensors and a spinning speed of 3 kHz.

```
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(
    processed_dataset[0].split()[0].real,
    color="black",
    linewidth=1,
    label="$\\nu_r=$3 kHz, \n $\\beta=30^\\circ$",
)
ax.legend()
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```



The following is a grid plot showing all twelve simulations. For reference, see Figure 11 from[Page 172, 1].

```
fig, ax = plt.subplots(
    nrows=4,
    ncols=3,
    subplot_kw={"projection": "csdm"},
    sharex=True,
    sharey=True,
    figsize=(8, 10.0),
)
for i, datum in enumerate(processed_dataset):
    datum_spin_sys = datum.split()  # get simulation from the three spin systems.
    for j, item in enumerate(datum_spin_sys):
        ax[i, j].plot(
            item.real,
            color="black",
            linewidth=1,
            label=(
                f"$\\nu_r={spin_rates[i]/1e3: .0f}$ kHz \n"
                f"$\\beta={beta_orientation[j]/np.pi*180: .0f}^\\circ$"
            ),
        )
        ax[i, j].invert_xaxis()
        ax[i, j].legend()
plt.tight_layout()
plt.show()
```


$\nu_r = 3$ kHz
$\beta = 30^\circ$
$\nu_r = 3$ kHz
$\beta = 50^\circ$
$\nu_r = 3$ kHz
$\beta = 90^\circ$
$\nu_r = 5$ kHz
$\beta = 30^\circ$
$\nu_r = 5$ kHz
$\beta = 50^\circ$
$\nu_r = 5$ kHz
$\beta = 90^\circ$
$\nu_r = 8$ kHz
$\beta = 30^\circ$
$\nu_r = 8$ kHz
$\beta = 50^\circ$
$\nu_r = 8$ kHz
$\beta = 90^\circ$
$\nu_r = 12$ kHz
$\beta = 30^\circ$
$\nu_r = 12$ kHz
$\beta = 50^\circ$
$\nu_r = 12$ kHz
$\beta = 90^\circ$
dimensionless
frequency / (ppm)
0.0
0.1
0.2
0.3
0.4
0.5
200
0
−200

Total running time of the script: (0 minutes 2.458 seconds)

Writing Custom methods (HahnEcho)

Writing custom methods using the Event objects.

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

from mrsimulator import Simulator, SpinSystem, Site, Coupling
from mrsimulator.method import Method, SpectralDimension, SpectralEvent, MixingEvent
from mrsimulator.spin_system.tensors import SymmetricTensor
from pprint import pprint
```

For demonstration, we will create two spin systems, one with a single site and other with two spin 1/2 sites.

```
S1 = Site(
    isotope="1H",
    isotropic_chemical_shift=10,  # in ppm
    shielding_symmetric=SymmetricTensor(zeta=-80, eta=0.25),  # zeta in ppm
)
S2 = Site(isotope="1H", isotropic_chemical_shift=-10)
S12 = Coupling(
    site_index=[0, 1], isotropic_j=100, dipolar=SymmetricTensor(D=2000, eta=0, alpha=0)
)

spin_system_1 = SpinSystem(sites=[S1], label="Uncoupled system")
spin_system_2 = SpinSystem(sites=[S1, S2], couplings=[S12], label="Coupled system")
```

Create a custom method

This example is a brief illustrate on how to write a custom method in mrsimulator. For in-depth description, please refer to the *Method* (page 79) documentation.

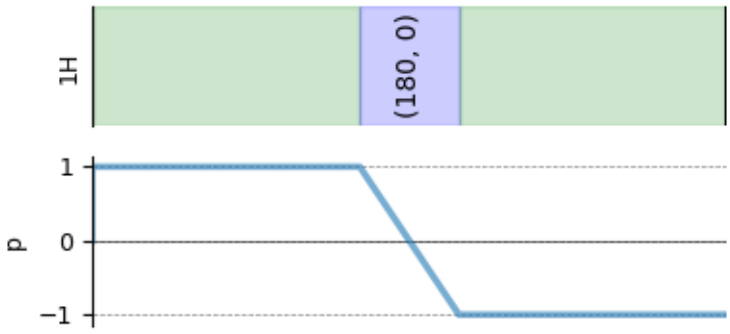
In this example, we use two types of Event objects—SpectralEvent and MixingEvent to create a one-dimensional Hahn echo method.

```
hahn_echo = Method(
    channels=["1H"],
    magnetic_flux_density=9.4,  # in T
    rotor_angle=0,  # in rads
    rotor_frequency=0,  # in Hz
    spectral_dimensions=[
        SpectralDimension(
            count=512,
            spectral_width=2e4,  # in Hz
            events=[
                SpectralEvent(fraction=0.5, transition_queries=[{"ch1": {"P": [1]}}]),
                MixingEvent(query={"ch1": {"angle": np.pi, "phase": 0}}),
                SpectralEvent(fraction=0.5, transition_queries=[{"ch1": {"P": [-1]}}]),
            ],
        )
    ],
)
```

In the above code, we define the two SpectralEvent objects with fraction 0.5 and the transition_queries on channel-1 of P=[1] and P=[-1], respectively. Notice, the value for the `P` attribute is a list. Here, it is a list with a single integer. The list notation, `[1]`, implies that the query selects all transitions where exactly one spin is undergoing a $p = +1$ transition with the remaining spin at $p = 0$. A similar argument holds for `[-1]` query. By implementing query objects, we decouple the method from the spin system, i.e., once a method is defined, it can be used to simulate spectra from any given spin system. We will demonstrate this momentarily by simulating a Hahn echo spectrum from single and two-site spin systems.

Besides the SpectralEvent, you may also notice a MixingEvent sandwiched in-between the two SpectralEvent. A MixingEvent does not directly contribute to the frequencies. As the name suggests, a mixing event is used for the mixing of transitions in a multi-event method such as HahnEcho. In the above code, we define a mixing query on channel-1 by setting the attributes `angle` and `phase` to π and 0, respectively. There two parameters are analogous to the pulse angle and phase.

```
plt.figure(figsize=(4, 2))
hahn_echo.plot()
plt.show()
```



As mentioned before, a method object is decoupled from the spin system object. Notice, when we get the transition pathways from this method for a single-site spin system, we get a single transition pathway.

```
pprint(hahn_echo.get_transition_pathways(spin_system_1))
```

```
[|0.5⟩⟨-0.5| ⟶ |-0.5⟩⟨0.5|, weight=(1+0j)]
```

In the case of a homonuclear two-site spin 1/2 spin system, the same method returns four transition pathways.

```
pprint(hahn_echo.get_transition_pathways(spin_system_2))
```

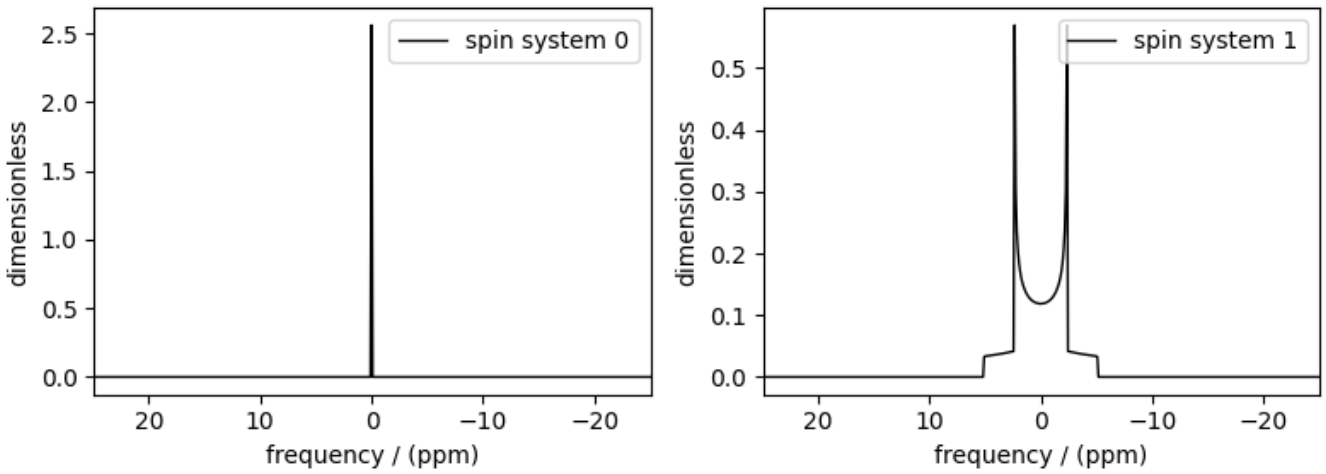
```
[|-0.5, 0.5⟩⟨-0.5, -0.5| ⟶ |0.5, -0.5⟩⟨0.5, 0.5|, weight=(1+0j),
 |0.5, -0.5⟩⟨-0.5, -0.5| ⟶ |-0.5, 0.5⟩⟨0.5, 0.5|, weight=(1+0j),
 |0.5, 0.5⟩⟨-0.5, 0.5| ⟶ |-0.5, -0.5⟩⟨0.5, -0.5|, weight=(1+0j),
 |0.5, 0.5⟩⟨0.5, -0.5| ⟶ |-0.5, -0.5⟩⟨-0.5, 0.5|, weight=(1+0j)]
```

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

```
sim = Simulator(spin_systems=[spin_system_1, spin_system_2], methods=[hahn_echo])
sim.config.decompose_spectrum = "spin_system"
sim.run()
```

The simulation from each spin system is stored as a dependent variable within the CSDM object. Use the *split* function to split the list of the dependent variables into a list of CSDM objects.

```
simulation_results = sim.methods[0].simulation.split()

# The plot of the two simulations.
fig, ax = plt.subplots(1, 2, figsize=(8.0, 3.0), subplot_kw={"projection": "csdm"})
for i in range(2):
    ax[i].plot(simulation_results[i].real, color="black", linewidth=1)
    ax[i].invert_xaxis()
plt.tight_layout()
plt.show()
```



Notice, in the single-site spin system, the hahn echo refocuses the isotropic chemical shits and chemical shift anisotropies. The end result is a resonance at zero frequency. In the case of the two homonuclear spin 1/2 coupled spin system, the Hahn echo refocuses the isotropic chemical shits and chemical shift anisotropies, but not the dipolar and J couplings.

Total running time of the script: (0 minutes 0.542 seconds)

12.4.2 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 425)),
- Central transition selective Bloch decay method (*BlochDecayCTSpectrum* (page 431)).

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

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

^{13}C/^{15}N (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.pyplot as plt

from mrsimulator import Simulator
from mrsimulator.method.lib import BlochDecaySpectrum
from mrsimulator.method import SpectralDimension
from mrsimulator import signal_processor as sp
```

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

```
sim = Simulator()

host = "https://ssnmr.org/sites/default/files/mrsimulator/"
filename = "protein_GB1_15N_13CA_13CO.mrsys"
sim.load_spin_systems(host + filename)  # load the spin systems.
print(f"number of spin systems = {len(sim.spin_systems)}")
```

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

```
all_sites = sim.sites().to_pd()
all_sites.head()
```

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

```
method_13C = BlochDecaySpectrum(
    channels=["13C"],
    magnetic_flux_density=11.74,  # in T
    rotor_frequency=3000,  # in Hz
    spectral_dimensions=[
        SpectralDimension(
            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=11.74,  # in T
```

(continues on next page)

[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

(continued from previous page)

```
    rotor_frequency=3000,  # in Hz
    spectral_dimensions=[
        SpectralDimension(
            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

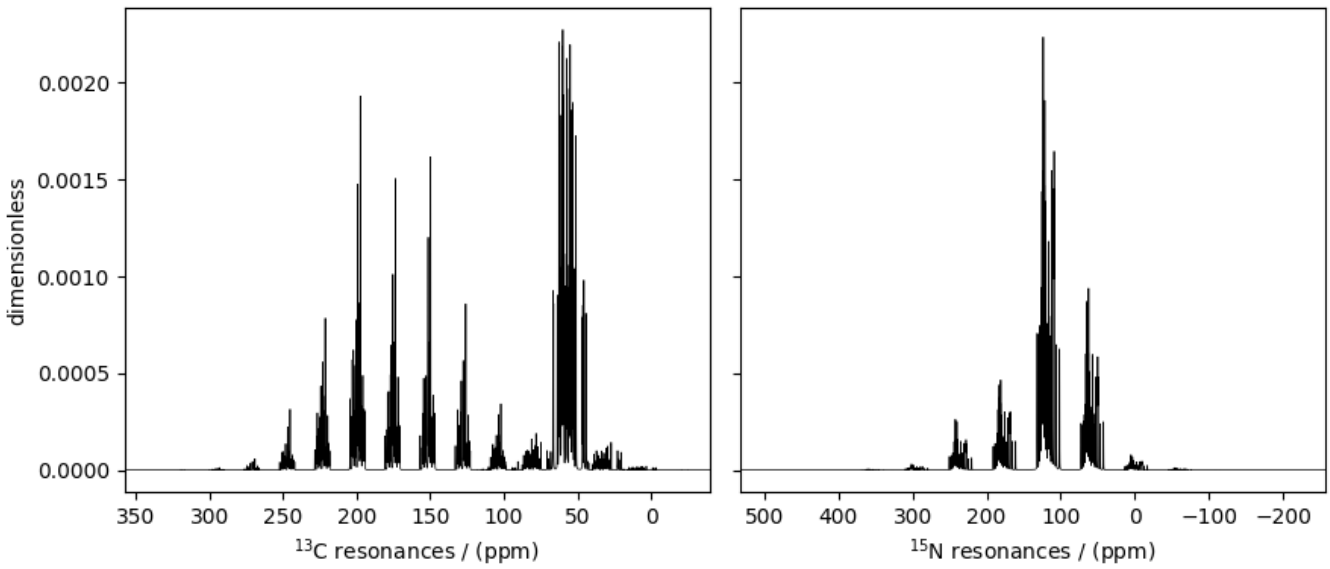
```
# Add the methods.
sim.methods = [method_13C, method_15N]

# Run the simulation.
sim.run()

# Get the simulation dataset from the respective methods.
dataset_13C = sim.methods[0].simulation  # method at index 0 is 13C Bloch decay method.
dataset_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(), sp.apodization.Exponential(FWHM="10 Hz"), sp.FFT()]
)
# apply post-simulation processing to dataset_13C
processed_dataset_13C = processor.apply_operations(dataset=dataset_13C).real

# apply post-simulation processing to dataset_15N
processed_dataset_15N = processor.apply_operations(dataset=dataset_15N).real
```

The plot of the simulation after signal processing.

```
fig, ax = plt.subplots(
    1, 2, subplot_kw={"projection": "csdm"}, sharey=True, figsize=(9, 4)
)

ax[0].plot(processed_dataset_13C, color="black", linewidth=0.5)
ax[0].invert_xaxis()

ax[1].plot(processed_dataset_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 2.776 seconds)

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

^{29}Si (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 numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal

from mrsimulator import Simulator
from mrsimulator.method.lib import BlochDecaySpectrum
from mrsimulator.utils.collection import single_site_system_generator
from mrsimulator.method import SpectralDimension
```

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
```

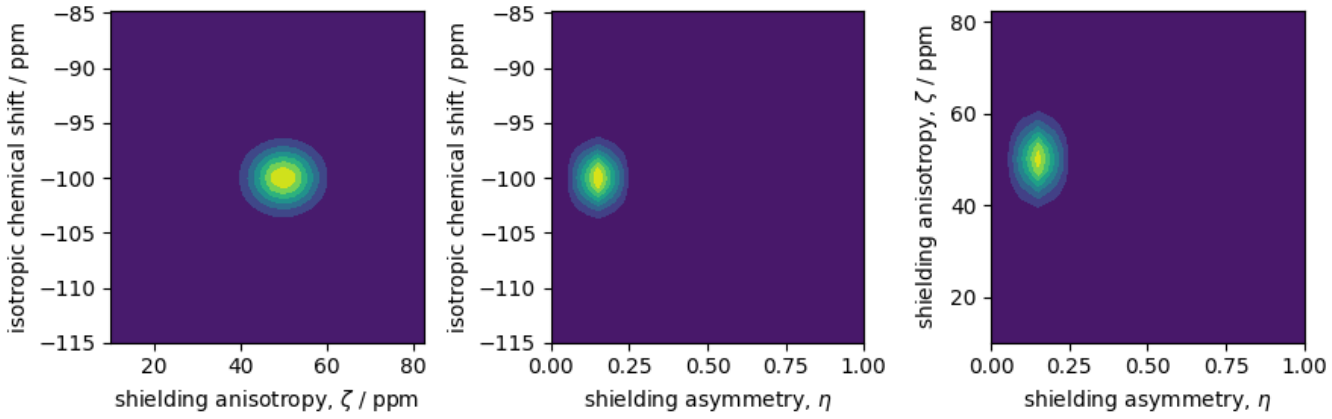
(continues on next page)

(continued from previous page)

```
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, $\eta$")
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()* (page 484) utility function to generate single-site spin systems. # Here, `iso`, `zeta`, and `eta` are the array of tensor parameter coordinates, and `pdf` is the array of the corresponding amplitudes.

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

Method:

Let's also create a Bloch decay spectrum method.

```
method = BlochDecaySpectrum(
    channels=["29Si"],
    rotor_frequency=0,  # in Hz
    rotor_angle=0,  # in rads
    spectral_dimensions=[
        SpectralDimension(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(spin_systems=spin_systems, methods=[method])
```

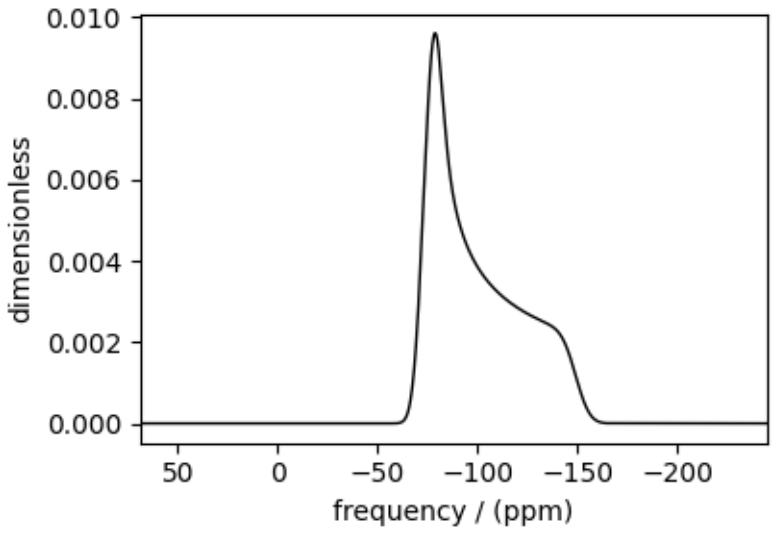
Static spectrum

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

```
sim.run()
```

The plot of the simulation.

```
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(sim.methods[0].simulation.real, 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.

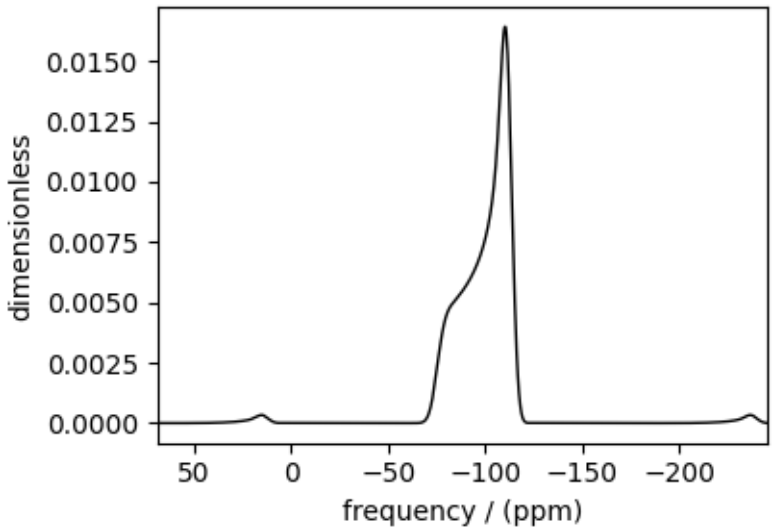
Spinning sideband simulation at 90°

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=[
        SpectralDimension(spectral_width=25000, reference_offset=-7000)  # values in Hz
    ],
)
sim.config.number_of_sidebands = 8  # eight sidebands are sufficient for this example
sim.run()
```

The plot of the simulation.

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



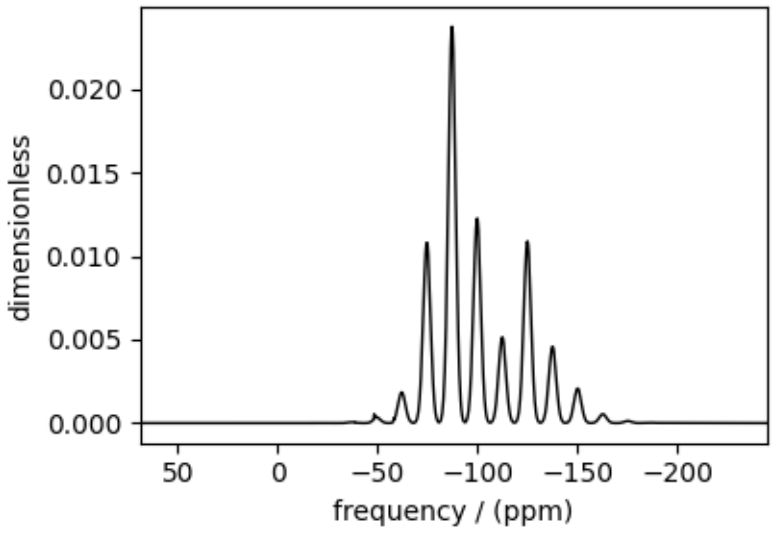
Spinning sideband simulation at the magic angle

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=54.7356 * np.pi / 180.0,  # in rads
    spectral_dimensions=[
        SpectralDimension(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.

```
plt.figure(figsize=(4.25, 3.0))
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 18.725 seconds)

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

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

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal

from mrsimulator import Simulator
from mrsimulator.method.lib import BlochDecayCTSpectrum
from mrsimulator.utils.collection import single_site_system_generator
from mrsimulator.method import SpectralDimension
```

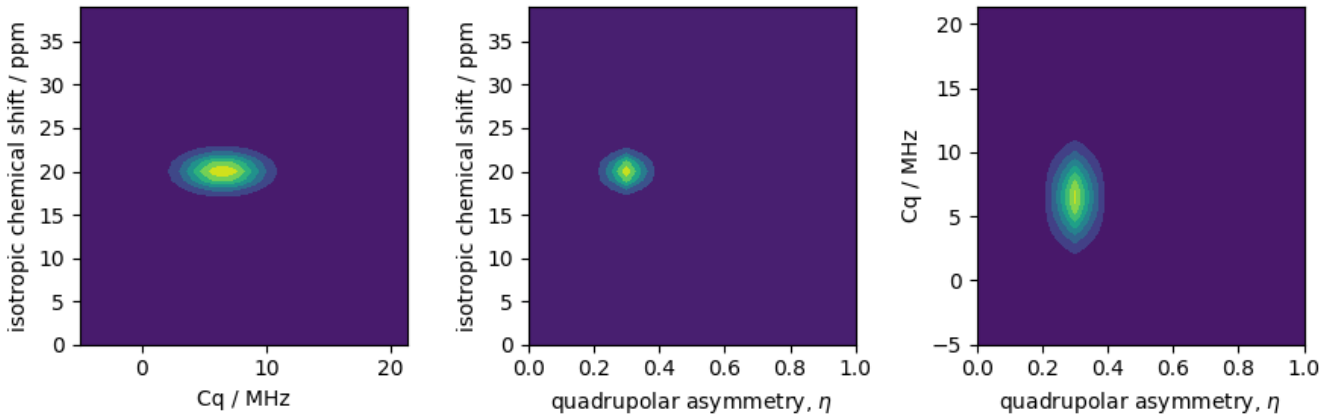
In this section, we illustrate the simulation of a quadrupolar spectrum arising from a distribution of the electric field gradient (EFG) tensors from an amorphous material. We proceed by assuming a multi-variate normal distribution, as follows,

```
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()* (page 484) utility function to generate single-site spin systems.

```
spin_systems = single_site_system_generator(
    isotope="27Al",
    isotropic_chemical_shift=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.

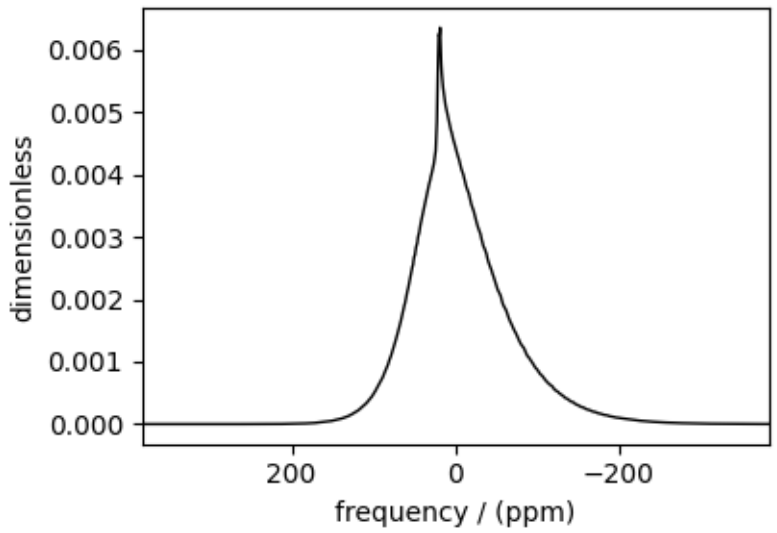
```
static_method = BlochDecayCTSpectrum(
    channels=["27Al"],
    rotor_frequency=0,  # in Hz
    rotor_angle=0,  # in rads
    spectral_dimensions=[SpectralDimension(spectral_width=80000)],
)
```

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

```
sim = Simulator(spin_systems=spin_systems, methods=[static_method])
sim.run()
```

The plot of the corresponding spectrum.

```
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(sim.methods[0].simulation.real, 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.

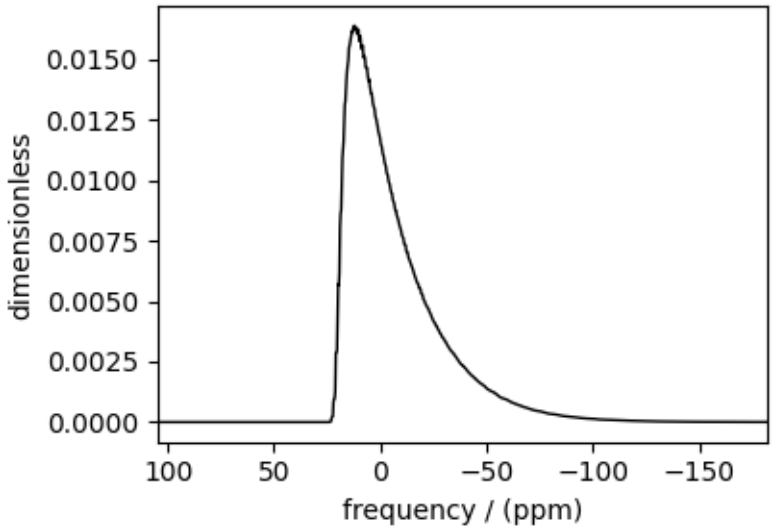
```
MAS_method = BlochDecayCTSpectrum(
    channels=["27Al"],
    rotor_frequency=25000,  # in Hz
    rotor_angle=54.735 * np.pi / 180.0,  # in rads
    spectral_dimensions=[
        SpectralDimension(spectral_width=30000, reference_offset=-4000)  # values in Hz
    ],
)
sim.methods[0] = MAS_method
```

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.

```
plt.figure(figsize=(4.25, 3.0))
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 11.397 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 numpy as np
import matplotlib.pyplot as plt

from mrsimulator import Simulator
from mrsimulator.method.lib import BlochDecaySpectrum, BlochDecayCTSpectrum
from mrsimulator.models import CzjzekDistribution
from mrsimulator.utils.collection import single_site_system_generator
from mrsimulator.method import SpectralDimension
```

Symmetric shielding tensor

Create the Czjzek distribution

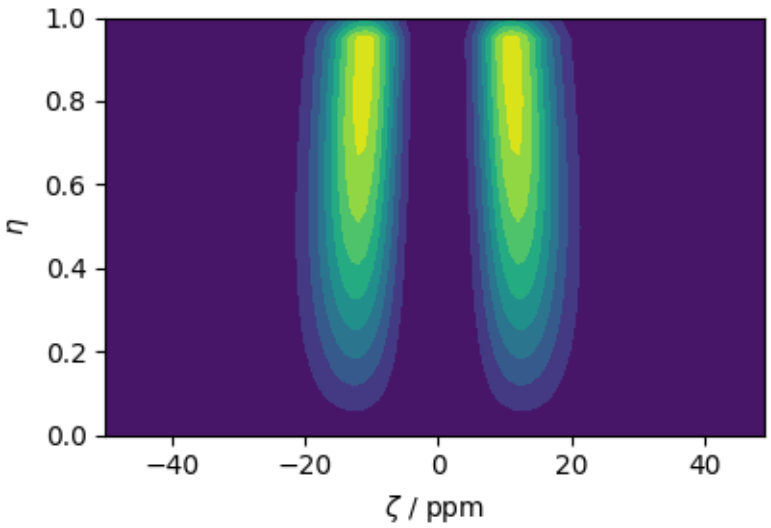
First, create a distribution of the zeta and eta parameters of the shielding tensors using the *Czjzek distribution* (page 59) model as follows.

```
# 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 = np.meshgrid(z_range, e_range)
_, _, 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.figure(figsize=(4.25, 3.0))
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()* (page 484) utility function.

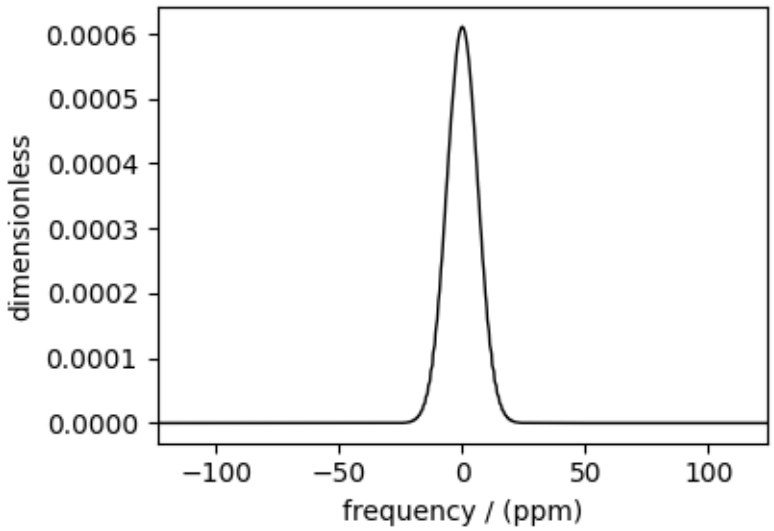
```
systems = single_site_system_generator(
    isotope="13C", shielding_symmetric={"zeta": z_dist, "eta": e_dist}, abundance=amp
)
method = BlochDecaySpectrum(
    channels=["13C"],
    rotor_frequency=0,  # in Hz
    rotor_angle=0,  # in rads
)
```

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

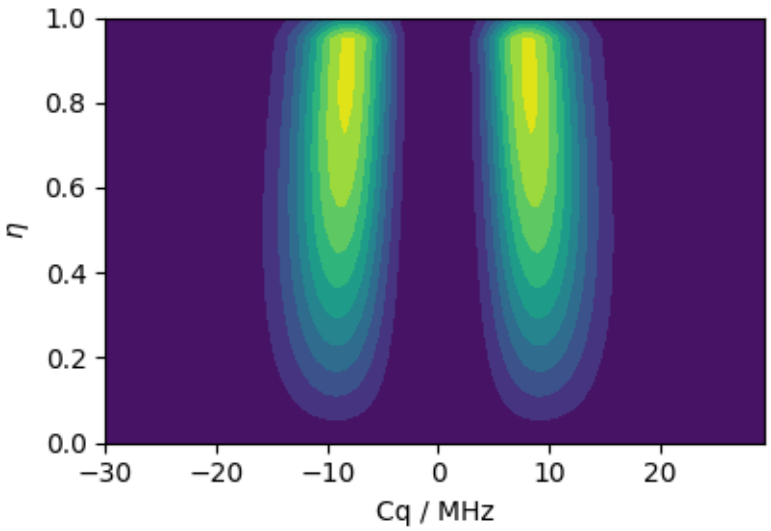


Quadrupolar tensor

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 = np.meshgrid(cq_range, e_range)
_, _, amp = CzjzekDistribution(sigma=2.3).pdf(pos=[cq_range, e_range])

# The following is the contour plot of the Czjzek distribution.
plt.figure(figsize=(4.25, 3.0))
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 and method

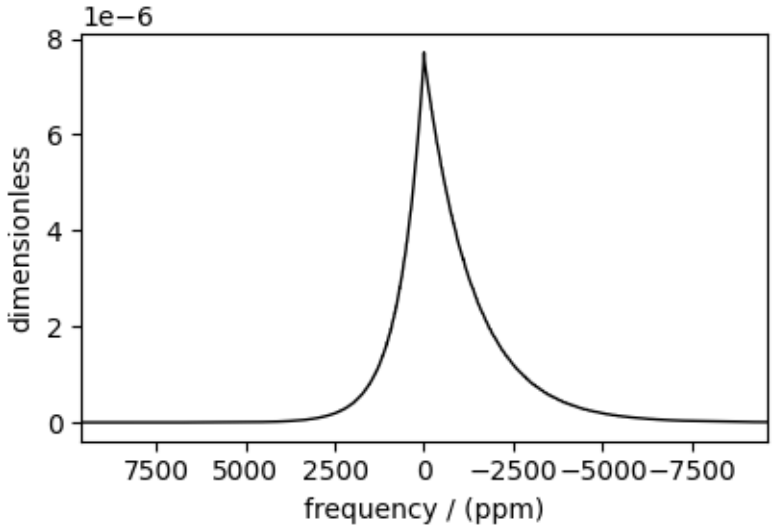
```
systems = single_site_system_generator(
    isotope="71Ga", quadrupolar={"Cq": cq_dist * 1e6, "eta": e_dist}, abundance=amp
)

method = BlochDecayCTSpectrum(
    channels=["71Ga"],
    magnetic_flux_density=4.8,  # in T
    rotor_frequency=0,  # in Hz
    rotor_angle=0,  # in rads
    spectral_dimensions=[SpectralDimension(count=2048, spectral_width=1.2e6)],
)
```

Create a simulator object and add the above system.

```
sim = Simulator(spin_systems=systems, methods=[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.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 4.171 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 numpy as np
import matplotlib.pyplot as plt

from mrsimulator import Simulator
from mrsimulator.method.lib import BlochDecaySpectrum, BlochDecayCTSpectrum
from mrsimulator.models import ExtCzjzekDistribution
from mrsimulator.utils.collection import single_site_system_generator
from mrsimulator.method import SpectralDimension
```

Symmetric shielding tensor

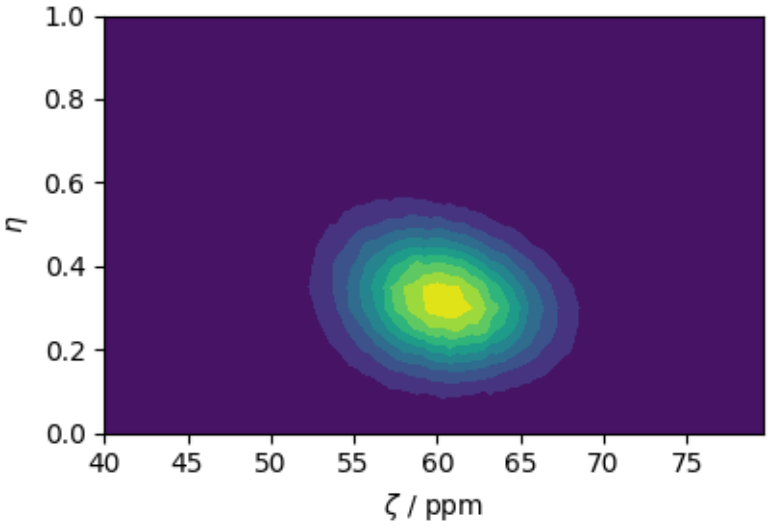
Create the extended Czjzek distribution

First, create a distribution of the zeta and eta parameters of the shielding tensors using the *Extended Czjzek distribution* (page 65) 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 = np.meshgrid(z_lim, e_lim)
_, _, amp = ExtCzjzekDistribution(dominant, eps=0.14).pdf(pos=[z_lim, e_lim])
```

The following is the plot of the extended Czjzek distribution.

```
plt.figure(figsize=(4.25, 3.0))
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(
    isotope="13C", shielding_symmetric={"zeta": z_dist, "eta": e_dist}, abundance=amp
)
print(len(systems))
```

```
845
```

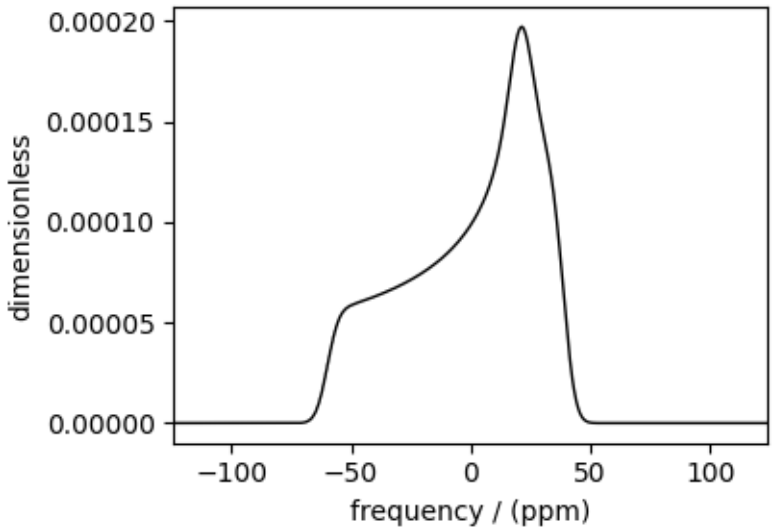
```
method = BlochDecaySpectrum(
    channels=["13C"],
    rotor_frequency=0,  # in Hz
    rotor_angle=0,  # in rads
)
```

Create a simulator object and add the above system.

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



Quadrupolar tensor

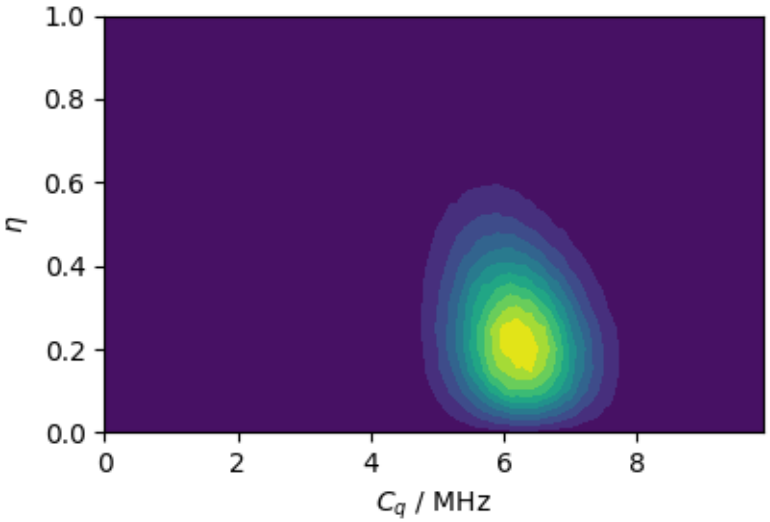
Create the extended Czjzek distribution

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 = np.meshgrid(cq_lim, e_lim)
_, _, amp = ExtCzjzekDistribution(dominant, eps=0.25).pdf(pos=[cq_lim, e_lim])
```

The following is the plot of the extended Czjzek distribution.

```
plt.figure(figsize=(4.25, 3.0))
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.

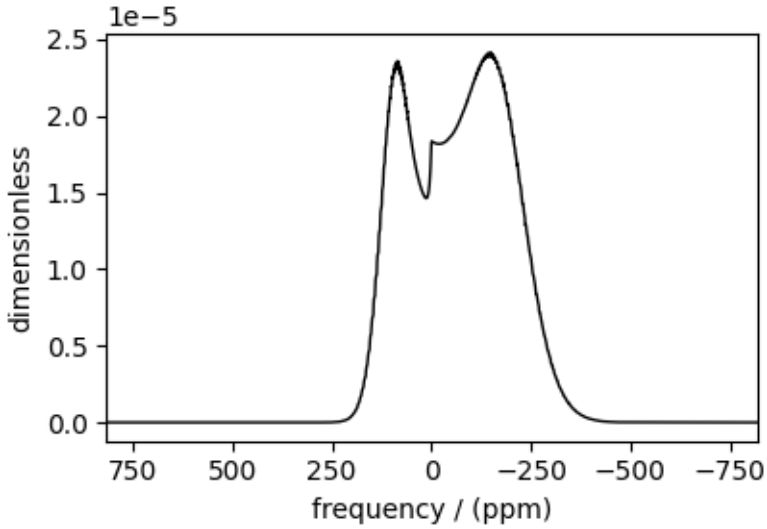
```
systems = single_site_system_generator(
    isotope="71Ga", quadrupolar={"Cq": cq_dist * 1e6, "eta": e_dist}, abundance=amp
)

method = BlochDecayCTSpectrum(
    channels=["71Ga"],
    magnetic_flux_density=9.4,  # in T
    rotor_frequency=0,  # in Hz
    rotor_angle=0,  # in rads
    spectral_dimensions=[SpectralDimension(count=2048, spectral_width=2e5)],
)
```

Create a simulator object and add the above system.

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

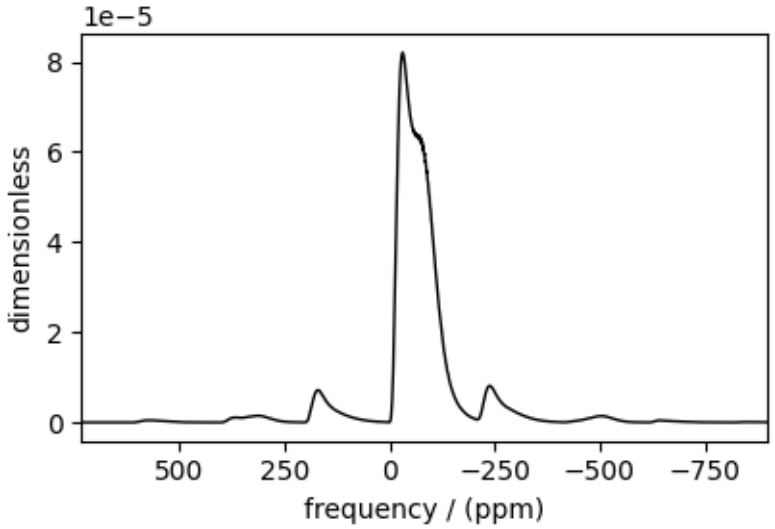


MAS spectrum

```
mas = BlochDecayCTSpectrum(
    channels=["71Ga"],
    magnetic_flux_density=9.4,  # in T
    rotor_frequency=25000,  # in Hz
    rotor_angle=54.7356 * np.pi / 180,  # in rads
    spectral_dimensions=[
        SpectralDimension(count=2048, spectral_width=2e5, reference_offset=-1e4)
    ],
)
sim.methods[0] = mas  # 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.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 6.186 seconds)

12.4.3 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 (i.e., 3Q-MAS) using the specialized *ThreeQ_VAS()* (page 438) method.
- Satellite-transition variable-angle spinning (i.e., ST-MAS) using the specialized *ST1_VAS()* (page 457) method.
- Switched Angle Spinning (SAS) using the generic *Method()* (page 403) object.
- MAS-detected Dynamic Angle Spinning (DAS) using the generic *Method()* (page 403) object.
- Correlation of Anisotropies Separated Through Echo Refocusing (COASTER) using the generic *Method()* (page 403) object.
- Phase Adjusted Spinning Sidebands (PASS and QPASS) and Magic-Angle Turning (MAT and QMAT) using the specialized *SSB2D()* (page 469) method.

$RbNO_3$, ^{87}Rb (I=3/2) 3QMAS

^{87}Rb (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.pyplot as plt

from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.method.lib import ThreeQ_VAS
from mrsimulator import signal_processor as sp
from mrsimulator.spin_system.tensors import SymmetricTensor
from mrsimulator.method import SpectralDimension
```

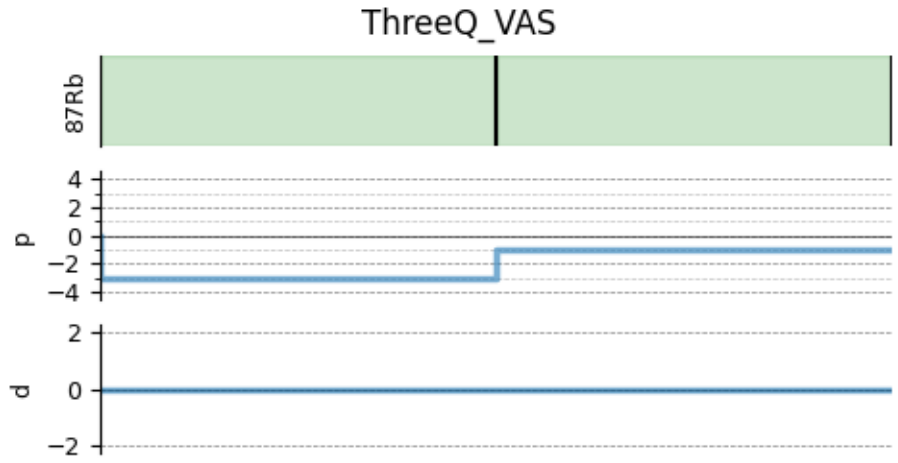
Generate the site and spin system objects.

[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

```
Rb87_1 = Site(
    isotope="87Rb",
    isotropic_chemical_shift=-27.4,  # in ppm
    quadrupolar=SymmetricTensor(Cq=1.68e6, eta=0.2),  # Cq is in Hz
)
Rb87_2 = Site(
    isotope="87Rb",
    isotropic_chemical_shift=-28.5,  # in ppm
    quadrupolar=SymmetricTensor(Cq=1.94e6, eta=1.0),  # Cq is in Hz
)
Rb87_3 = Site(
    isotope="87Rb",
    isotropic_chemical_shift=-31.3,  # in ppm
    quadrupolar=SymmetricTensor(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=[
        SpectralDimension(
            count=128,
            spectral_width=7e3,  # in Hz
            reference_offset=-7e3,  # in Hz
            label="Isotropic dimension",
        ),
        SpectralDimension(
            count=256,
            spectral_width=1e4,  # in Hz
            reference_offset=-4e3,  # in Hz
            label="MAS dimension",
        ),
    ],
)

# A graphical representation of the method object.
plt.figure(figsize=(5, 2.5))
method.plot()
plt.show()
```

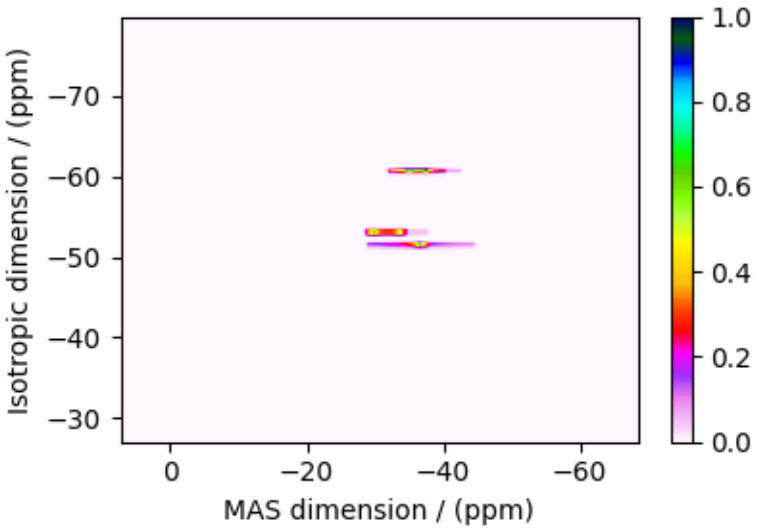


Create the Simulator object, add the method and spin system objects, and run the simulation.

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

The plot of the simulation.

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

plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
cb = ax.imshow(dataset.real / dataset.real.max(), aspect="auto", cmap="gist_ncar_r")
plt.colorbar(cb)
ax.invert_xaxis()
ax.invert_yaxis()
plt.tight_layout()
plt.show()
```

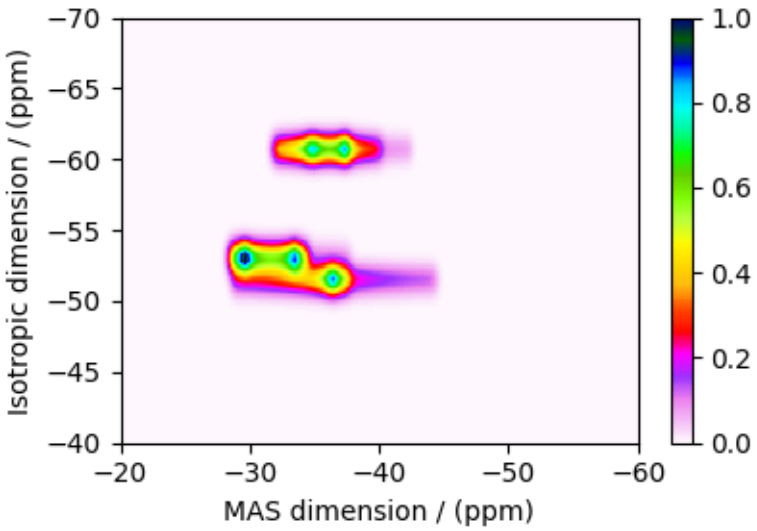


Add post-simulation signal processing.

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

The plot of the simulation after signal processing.

```
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
cb = ax.imshow(processed_dataset.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 248) for RbNO3.

Total running time of the script: (0 minutes 0.969 seconds)

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

^{27}Al (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.pyplot as plt

from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.method.lib import ThreeQ_VAS
from mrsimulator import signal_processor as sp
from mrsimulator.spin_system.tensors import SymmetricTensor
from mrsimulator.method import SpectralDimension
```

Generate the site and spin system objects.

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

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

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=[
        SpectralDimension(
            count=256,
            spectral_width=1e4,  # in Hz
            reference_offset=-3e3,  # in Hz
            label="Isotropic dimension",
        ),
        SpectralDimension(
            count=512,
            spectral_width=1e4,  # in Hz
            reference_offset=4e3,  # in Hz
            label="MAS dimension",
        ),
    ],
)
```

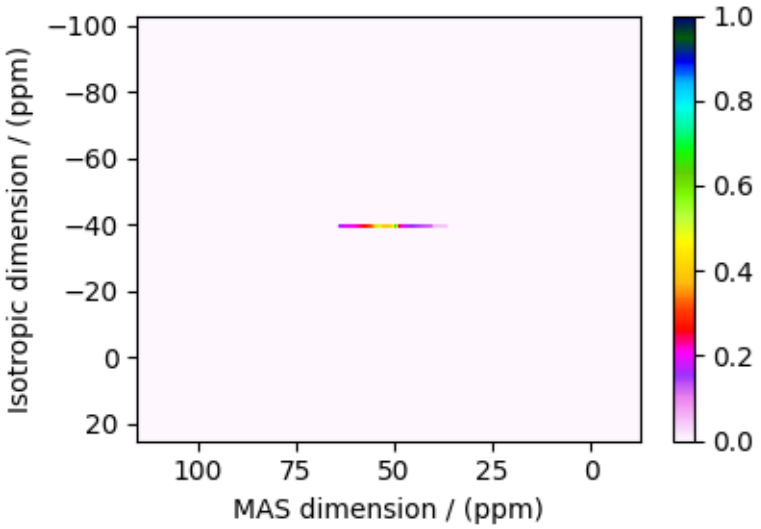
Create the Simulator object, add the method and spin system objects, and run the simulation.

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

The plot of the simulation.

[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

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

plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
cb = ax.imshow(dataset.real / dataset.real.max(), aspect="auto", cmap="gist_ncar_r")
plt.colorbar(cb)
ax.invert_xaxis()
ax.invert_yaxis()
plt.tight_layout()
plt.show()
```

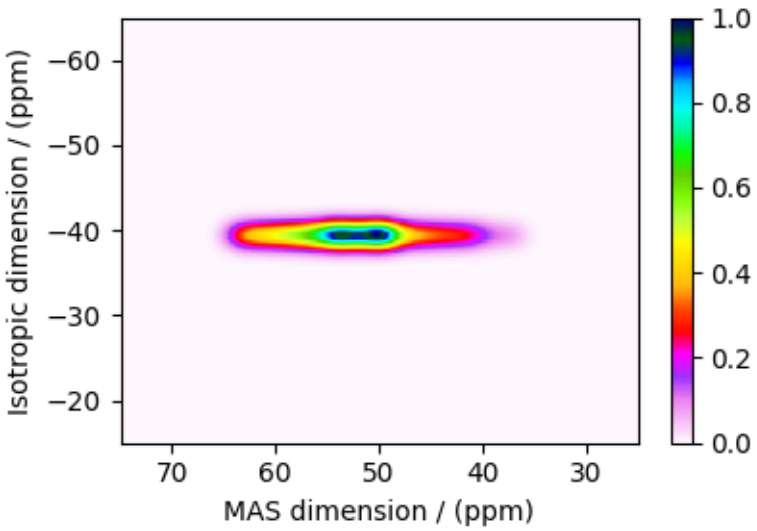


Add post-simulation signal processing.

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

The plot of the simulation after signal processing.

```
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
cb = ax.imshow(processed_dataset.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 0.788 seconds)

$RbNO_3$, ^{87}Rb (I=3/2) STMAS

^{87}Rb (I=3/2) satellite-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.pyplot as plt

from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.method.lib import ST1_VAS
from mrsimulator import signal_processor as sp
from mrsimulator.spin_system.tensors import SymmetricTensor
from mrsimulator.method import SpectralDimension
```

Generate the site and spin system objects.

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

(continues on next page)

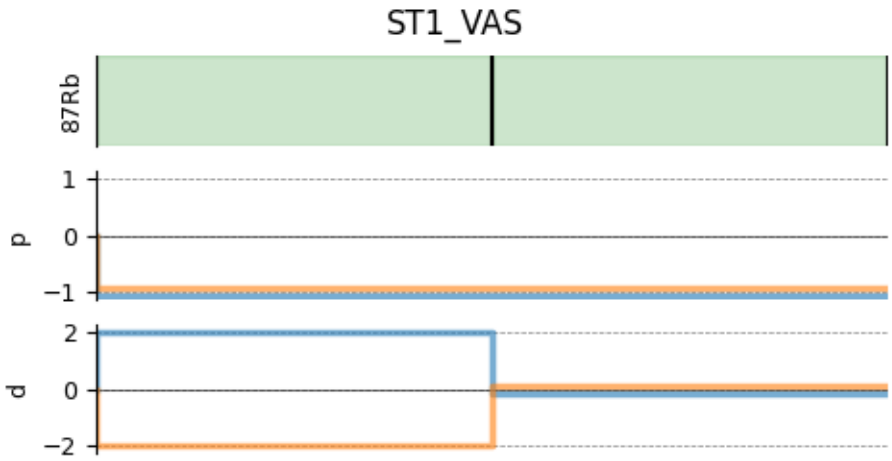
[1] Massiot, D., Touzoa, B., Trumeaua, D., Coutures, J.P., Virlet, J., Florian, P., Grandinetti, P.J. Two-dimensional magic-angle spinning isotropic reconstruction sequences for quadrupolar nuclei, ssnmr, (1996), **6**, *1*, 73-83. DOI: 10.1016/0926-2040(95)01210-9

(continued from previous page)

```
    quadrupolar=SymmetricTensor(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 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, the first at the magic angle and 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=[
                SpectralDimension(
                    count=256,
                    spectral_width=3e3,  # in Hz
                    reference_offset=-2.4e3,  # in Hz
                    label="Isotropic dimension",
                ),
                SpectralDimension(
                    count=512,
                    spectral_width=5e3,  # in Hz
                    reference_offset=-4e3,  # in Hz
                    label="MAS dimension",
                ),
            ],
        )
    )

# A graphical representation of the method object.
plt.figure(figsize=(5, 2.5))
method[0].plot()
plt.show()
```

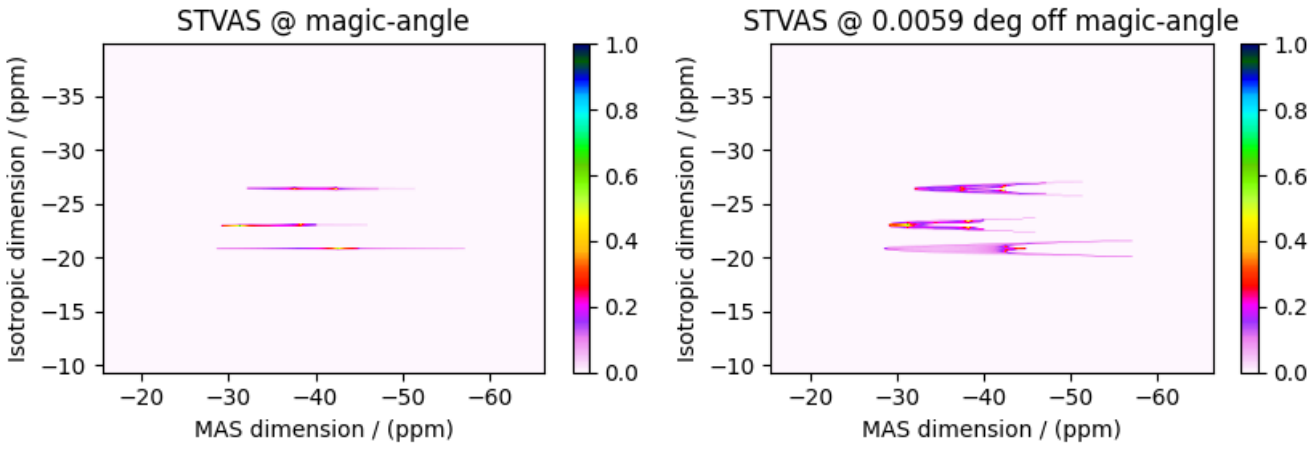


Create the Simulator object, add the method and spin system objects, and run the simulation.

```
sim = Simulator(spin_systems=spin_systems, methods=method)
sim.run()
```

The plot of the simulation.

```
dataset = [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(dataset):
    cb1 = ax[i].imshow(item.real / item.real.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()
```

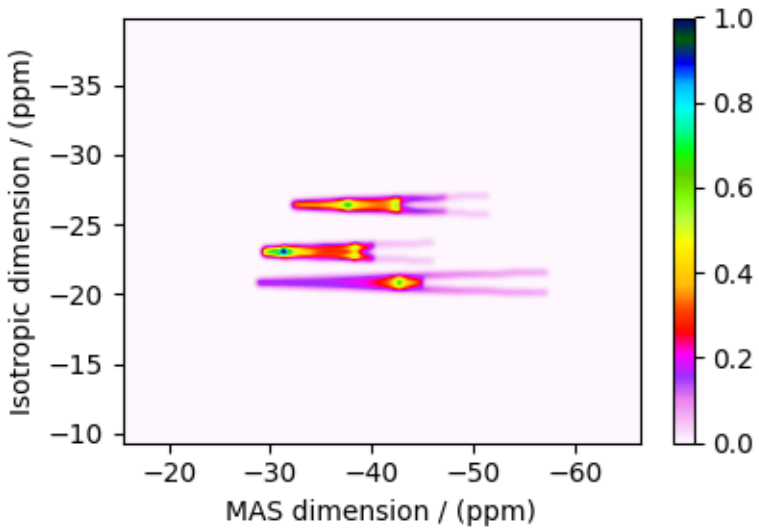


Add post-simulation signal processing.

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

The plot of the simulation after signal processing.

```
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
cb = ax.imshow(processed_dataset[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 1.314 seconds)

Rb_2SO_4, ^{87}Rb (I=3/2) SAS

^{87}Rb (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 numpy as np
import matplotlib.pyplot as plt

from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.method import Method, SpectralDimension, SpectralEvent
from mrsimulator import signal_processor as sp
from mrsimulator.spin_system.tensors import SymmetricTensor
```

Generate the site and spin system objects.

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

Use the generic *Method* class to simulate a 2D SAS spectrum by customizing the method parameters, as shown below.

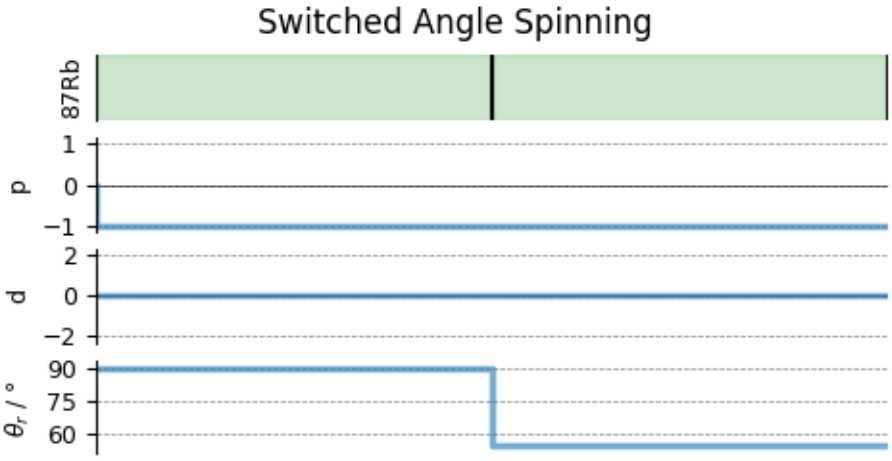
```
sas = Method(
    name="Switched Angle Spinning",
    channels=["87Rb"],
    magnetic_flux_density=9.4,  # in T
    rotor_frequency=np.inf,
    spectral_dimensions=[
        SpectralDimension(
            count=256,
            spectral_width=3.5e4,  # in Hz
            reference_offset=1e3,  # in Hz
            label="90 dimension",
            events=[
                SpectralEvent(
                    rotor_angle=90 * np.pi / 180,  # in radians
                    transition_queries=[{"ch1": {"P": [-1], "D": [0]}}],
                )
            ],
        ),
        SpectralDimension(
```

(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)

```
            count=256,
            spectral_width=22e3,  # in Hz
            reference_offset=-4e3,  # in Hz
            label="MAS dimension",
            events=[
                SpectralEvent(
                    rotor_angle=54.74 * np.pi / 180,  # in radians
                    transition_queries=[{"ch1": {"P": [-1], "D": [0]}}],
                )
            ],
        ),
    ],
)

# A graphical representation of the method object.
plt.figure(figsize=(5, 2.5))
sas.plot()
plt.show()
```

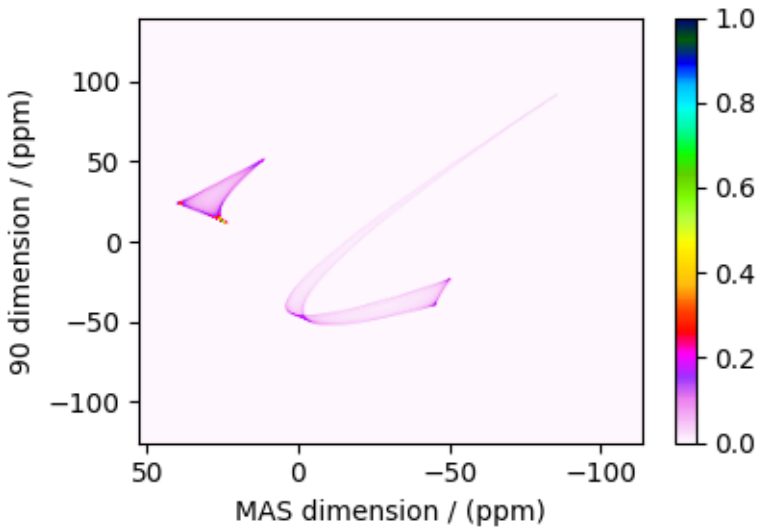


Create the Simulator object, add the method and spin system objects, and run the simulation.

```
sim = Simulator(spin_systems=spin_systems, methods=[sas])
sim.run()
```

The plot of the simulation.

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

plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
cb = ax.imshow(dataset.real / dataset.real.max(), aspect="auto", cmap="gist_ncar_r")
plt.colorbar(cb)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```

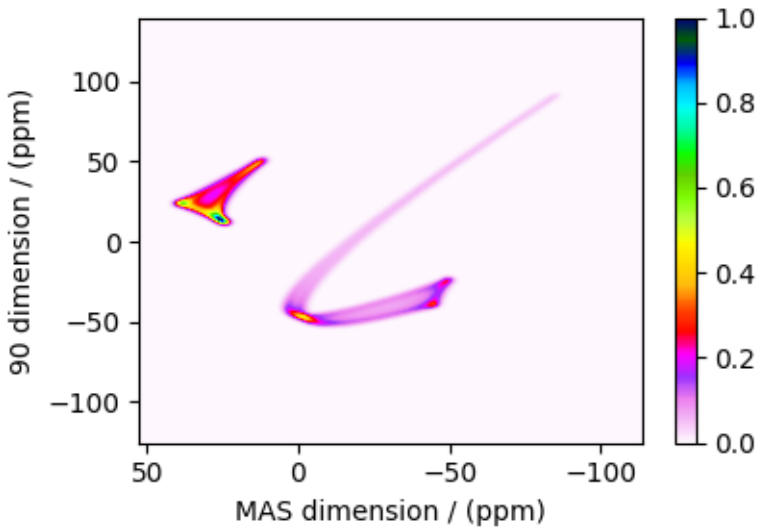


Add post-simulation signal processing.

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

The plot of the simulation after signal processing.

```
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
cb = ax.imshow(processed_dataset.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 0.995 seconds)

Co59 (I=7/2) STMAS

Co59 (I=7/2) satellite-transition magic-angle spinning simulation. (Quad-coupling cross terms)

```
import matplotlib.pyplot as plt

from mrsimulator import Simulator, SpinSystem, Site, Coupling
from mrsimulator.method.lib import ST1_VAS
from mrsimulator import signal_processor as sp
from mrsimulator.spin_system.tensors import SymmetricTensor
from mrsimulator.method import SpectralDimension
```

Generate the site and spin system objects.

```
Co_sites = [
    Site(
        isotope="59Co",  # 59Co
        isotropic_chemical_shift=0,  # in ppm
        # shielding_symmetric=SymmetricTensor(zeta=-1750, eta=0),
        quadrupolar=SymmetricTensor(Cq=3.1e6, eta=0.2),  # Cq is in Hz
    ),
    Site(isotope="1H", isotropic_chemical_shift=0),
]

coupling = [Coupling(site_index=[0, 1], dipolar={"D": 50000})]
spin_systems = [SpinSystem(sites=Co_sites, couplings=coupling)]
```

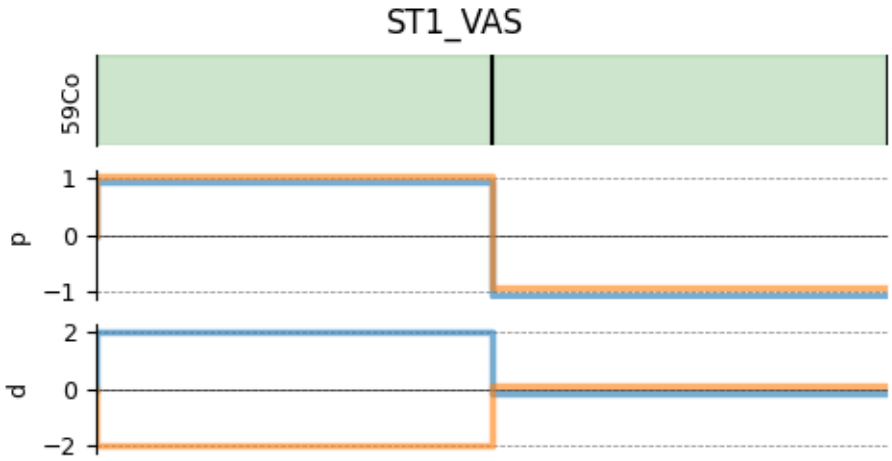
Select a satellite-transition variable-angle spinning method. The following *ST1_ VAS* method correlates the frequencies from the two inner-satellite transitions to the central transition.

```
method = ST1_VAS(
    channels=["59Co"],
```

(continues on next page)

(continued from previous page)

```
    magnetic_flux_density=4.684,  # in T
    rotor_angle=54.7359 * 3.14159 / 180,  # in rad (magic angle)
    spectral_dimensions=[
        SpectralDimension(
            count=256,
            spectral_width=1e3,  # in Hz
            # reference_offset=-1e3,  # in Hz
            label="Isotropic dimension",
        ),
        SpectralDimension(
            count=512,
            spectral_width=3e3,  # in Hz
            reference_offset=-1e3,  # in Hz
            label="MAS dimension",
        ),
    ],
)

# A graphical representation of the method object.
plt.figure(figsize=(5, 2.5))
method.plot()
plt.show()
```



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

```
sim = Simulator(spin_systems=spin_systems, methods=[method])
sim.config.decompose_spectrum = "spin_system"
sim.run()
```

Add post-simulation signal processing.

```
dataset = sim.methods[0].simulation
processor = sp.SignalProcessor(
    operations=[
        # Gaussian convolution along both dimensions.
        sp.IFFT(dim_index=(0, 1)),
        sp.apodization.Gaussian(FWHM="20 Hz", dim_index=0),
```

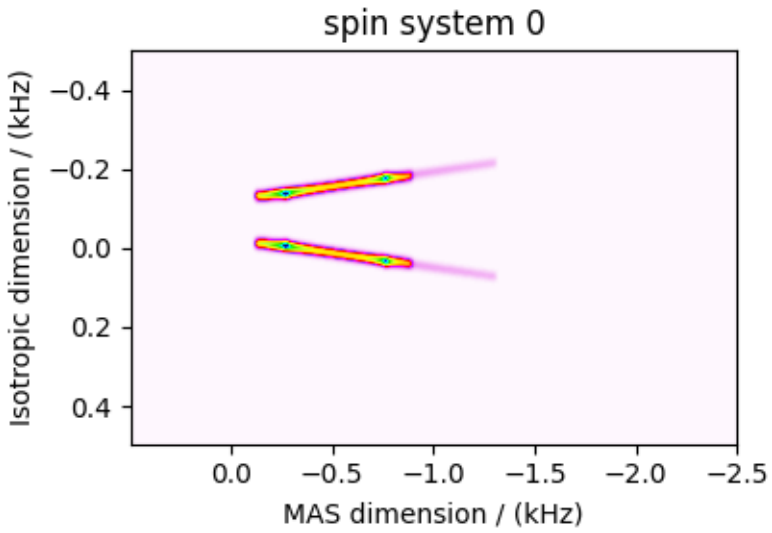
(continues on next page)

(continued from previous page)

```
        sp.apodization.Gaussian(FWHM="20 Hz", dim_index=1),
        sp.FFT(dim_index=(0, 1)),
    ]
)

processed_dataset = processor.apply_operations(dataset=dataset)
```

The plot of the simulation.

```
_ = [item.to("kHz", "nmr_frequency_ratio") for item in processed_dataset.x]

processed_dataset = processed_dataset.split()
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.imshow(processed_dataset[0].real, cmap="gist_ncar_r", aspect="auto")
ax.invert_xaxis()
ax.invert_yaxis()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.553 seconds)

Co59 (I=7/2) STMAS

Co59 (I=7/2) satellite-transition magic-angle spinning simulation. (Quad-csa cross terms)

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

from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.method.lib import ST1_VAS
from mrsimulator import signal_processor as sp
```

(continues on next page)

(continued from previous page)

```
from mrsimulator.spin_system.tensors import SymmetricTensor
from mrsimulator.method import SpectralDimension
```

Generate the site and spin system objects.

```
Co_sites = [
    Site(
        isotope="59Co",  # 59Co
        isotropic_chemical_shift=0,  # in ppm
        shielding_symmetric=SymmetricTensor(zeta=-1750, eta=1),
        quadrupolar=SymmetricTensor(Cq=3.1e6, eta=1),  # Cq is in Hz
        name="$\\alpha=\\beta=\\gamma=0$",
    ),
    Site(
        isotope="59Co",  # 59Co
        isotropic_chemical_shift=0,  # in ppm
        shielding_symmetric=SymmetricTensor(zeta=-1750, eta=1),
        quadrupolar=SymmetricTensor(Cq=3.1e6, eta=1, beta=np.pi / 2),  # Cq is in Hz
        name="$\\beta=90, \\alpha=\\gamma=0$",
    ),
    Site(
        isotope="59Co",  # 59Co
        isotropic_chemical_shift=0,  # in ppm
        shielding_symmetric=SymmetricTensor(zeta=-1750, eta=1),
        quadrupolar=SymmetricTensor(
            Cq=3.1e6, eta=1, alpha=np.pi / 2, beta=np.pi / 2
        ),  # Cq is in Hz
        name="$\\alpha=\\beta=90, \\gamma=0$",
    ),
    Site(
        isotope="59Co",  # 59Co
        isotropic_chemical_shift=0,  # in ppm
        shielding_symmetric=SymmetricTensor(zeta=-1750, eta=1),
        quadrupolar=SymmetricTensor(
            Cq=3.1e6, eta=1, alpha=np.pi / 2, beta=np.pi / 2, gamma=np.pi / 2
        ),  # Cq is in Hz
        name="$\\alpha=\\beta=\\gamma=90$",
    ),
]

spin_systems = [SpinSystem(sites=[site], name=site.name) for site in Co_sites]
```

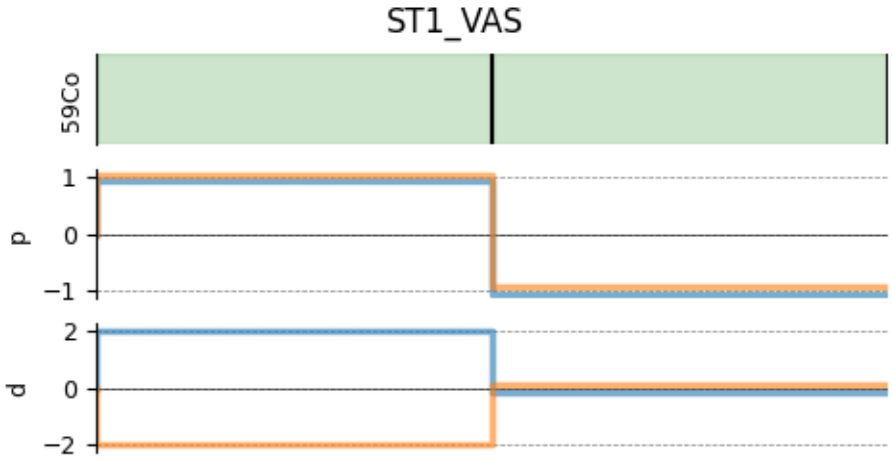
Select a satellite-transition variable-angle spinning method. The following *ST1_ VAS* method correlates the frequencies from the two inner-satellite transitions to the central transition.

```
method = ST1_VAS(
    channels=["59Co"],
    magnetic_flux_density=4.684,  # in T
    rotor_angle=54.7359 * 3.14159 / 180,  # in rad (magic angle)
    spectral_dimensions=[
        SpectralDimension(
            count=256,
```

(continues on next page)

(continued from previous page)

```
            spectral_width=1e3,  # in Hz
            label="Isotropic dimension",
        ),
        SpectralDimension(
            count=512,
            spectral_width=3e3,  # in Hz
            reference_offset=-1e3,  # in Hz
            label="MAS dimension",
        ),
    ],
)

# A graphical representation of the method object.
plt.figure(figsize=(5, 2.5))
method.plot()
plt.show()
```



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

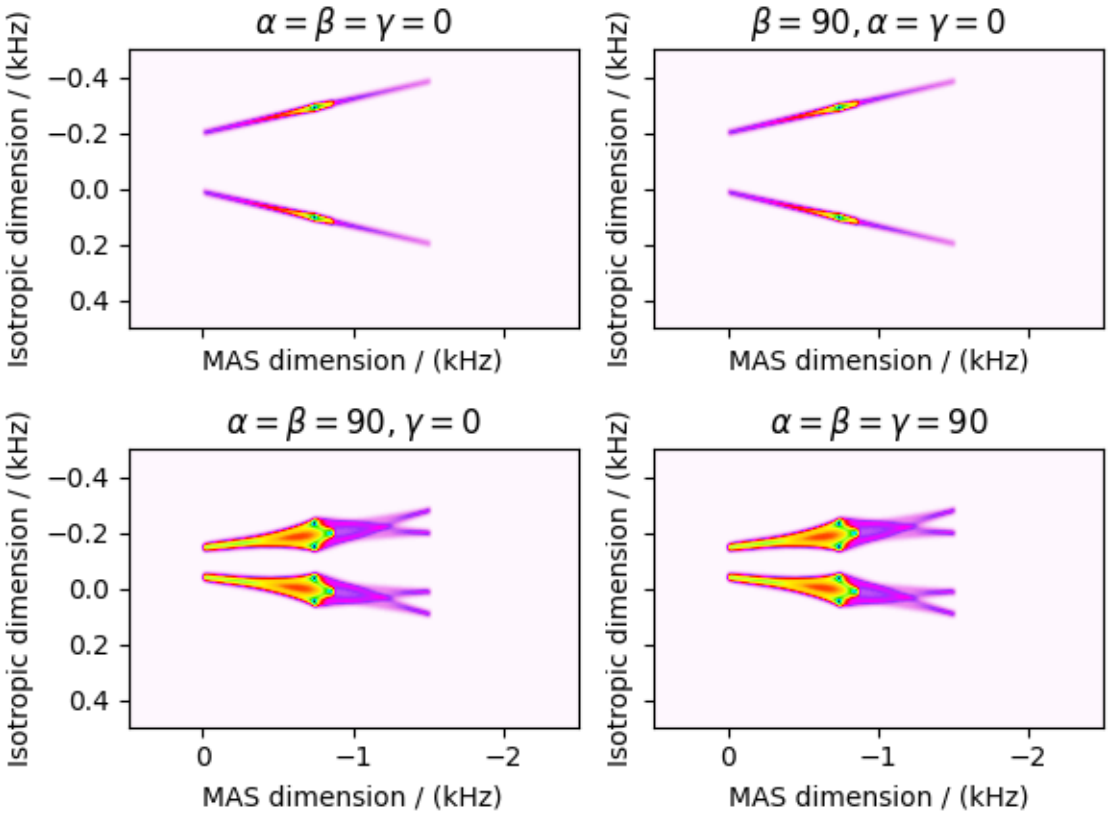
```
sim = Simulator(spin_systems=spin_systems, methods=[method])
sim.config.decompose_spectrum = "spin_system"
sim.run()
```

Add post-simulation signal processing.

```
dataset = sim.methods[0].simulation
processor = sp.SignalProcessor(
    operations=[
        # Gaussian convolution along both dimensions.
        sp.IFFT(dim_index=(0, 1)),
        sp.apodization.Gaussian(FWHM="20 Hz", dim_index=0),
        sp.apodization.Gaussian(FWHM="20 Hz", dim_index=1),
        sp.FFT(dim_index=(0, 1)),
    ]
)

processed_dataset = processor.apply_operations(dataset=dataset)
```

The plot of the simulation.

```
_ = [item.to("kHz", "nmr_frequency_ratio") for item in processed_dataset.x]

processed_dataset = processed_dataset.split()
fig, ax = plt.subplots(
    2, 2, figsize=(6, 4.5), sharex=True, sharey=True, subplot_kw={"projection": "csdm"}
)
ax[0, 0].imshow(processed_dataset[0].real, cmap="gist_ncar_r", aspect="auto")
ax[0, 1].imshow(processed_dataset[1].real, cmap="gist_ncar_r", aspect="auto")
ax[1, 0].imshow(processed_dataset[2].real, cmap="gist_ncar_r", aspect="auto")
ax[1, 1].imshow(processed_dataset[3].real, cmap="gist_ncar_r", aspect="auto")
ax[0, 0].invert_xaxis()
ax[0, 0].invert_yaxis()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 1.065 seconds)

Rb_2CrO_4, ^{87}Rb (I=3/2) SAS

^{87}Rb (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 numpy as np
import matplotlib.pyplot as plt

from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.method import Method, SpectralDimension, SpectralEvent
from mrsimulator import signal_processor as sp
from mrsimulator.spin_system.tensors import SymmetricTensor
```

Generate the site and spin system objects.

```
site = Site(
    isotope="87Rb",
    isotropic_chemical_shift=-7,  # in ppm
    shielding_symmetric=SymmetricTensor(zeta=110, eta=0),
    quadrupolar=SymmetricTensor(
        Cq=3.5e6,  # in Hz
        eta=0.3,
        alpha=0,  # in rads
        beta=70 * 3.14159 / 180,  # in rads
        gamma=0,  # in rads
    ),
)
spin_system = SpinSystem(sites=[site])
```

Use the generic *Method* class to simulate a 2D SAS spectrum by customizing the method parameters, as shown below.

```
sas = Method(
    channels=["87Rb"],
    magnetic_flux_density=4.2,  # in T
    rotor_frequency=np.inf,
    spectral_dimensions=[
        SpectralDimension(
            count=256,
            spectral_width=1.5e4,  # in Hz
            reference_offset=-5e3,  # in Hz
            label="70.12 dimension",
            events=[
                SpectralEvent(
                    rotor_angle=70.12 * np.pi / 180,  # in radians
                    transition_queries=[{"ch1": {"P": [-1], "D": [0]}}],
                )
            ],
        ),
        SpectralDimension(
```

(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)

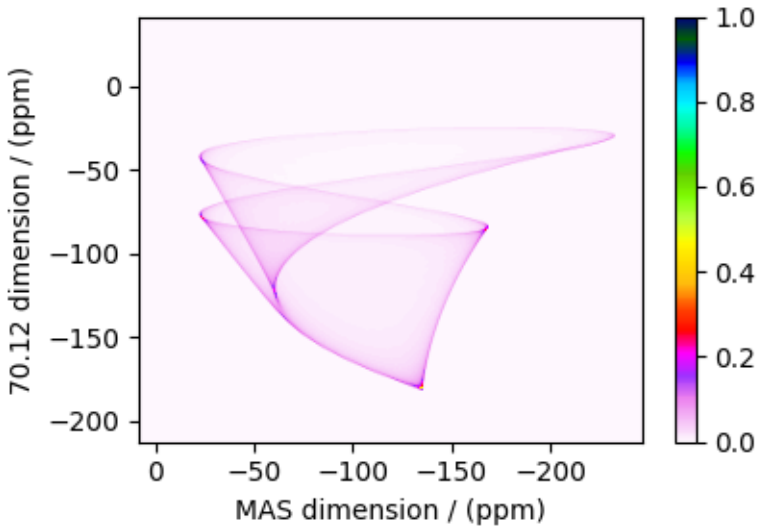
```
            count=512,
            spectral_width=15e3,  # in Hz
            reference_offset=-7e3,  # in Hz
            label="MAS dimension",
            events=[
                SpectralEvent(
                    rotor_angle=54.74 * np.pi / 180,  # in radians
                    transition_queries=[{"ch1": {"P": [-1], "D": [0]}}],
                )
            ],
        ),
    ],
)
```

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

```
sim = Simulator(spin_systems=[spin_system], methods=[sas])

# 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.

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

plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
cb = ax.imshow(dataset.real / dataset.real.max(), aspect="auto", cmap="gist_ncar_r")
plt.colorbar(cb)
ax.invert_xaxis()
plt.tight_layout()
plt.show()
```

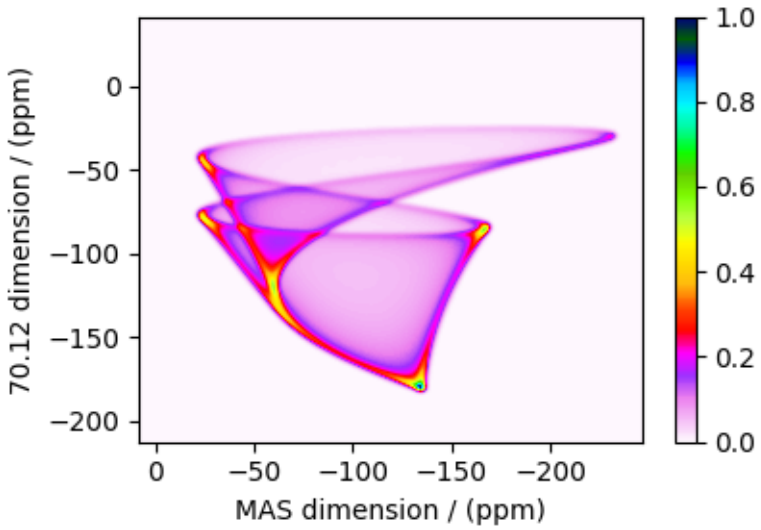


Add post-simulation signal processing.

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

The plot of the simulation after signal processing.

```
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
cb = ax.imshow(processed_dataset.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 0.839 seconds)

Coesite, ^{17}O (I=5/2) 3QMAS

^{17}O (I=5/2) 3QMAS simulation.

The following is a triple quantum magic angle spinning (3QMAS) simulation of Coesite. The NMR EFG tensor parameters for ^{17}O sites in coesite is obtained from Grandinetti *et al.*[1]

```
import matplotlib.pyplot as plt

from mrsimulator import Simulator
from mrsimulator.method.lib import ThreeQ_VAS
from mrsimulator import signal_processor as sp
from mrsimulator.method import SpectralDimension
```

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

```
sim = Simulator()

# load the spin systems from url.
filename = "https://ssnmr.org/sites/default/files/mrsimulator/coesite_0.mrsys"
sim.load_spin_systems(filename)

method = ThreeQ_VAS(
    channels=["17O"],
    magnetic_flux_density=11.74,  # in T
    spectral_dimensions=[
        SpectralDimension(
            count=256,
            spectral_width=5e3,  # in Hz
            reference_offset=-2.5e3,  # in Hz
            label="Isotropic dimension",
        ),
        # The last spectral dimension block is the direct-dimension
        SpectralDimension(
            count=256,
            spectral_width=2e4,  # in Hz
            reference_offset=0,  # in Hz
            label="MAS dimension",
        ),
    ],
)
sim.methods = [method]  # add the method.
sim.run()  # Run the simulation
```

The plot of the simulation.

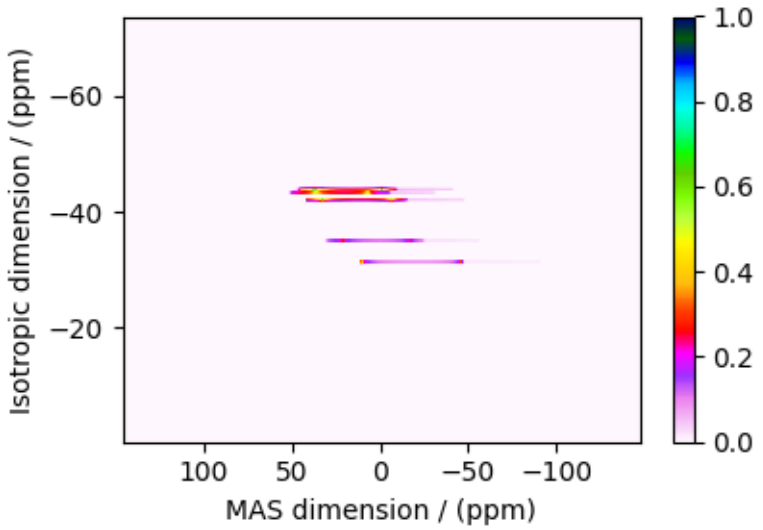
```
dataset = sim.methods[0].simulation

plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
cb = ax.imshow(dataset.real / dataset.real.max(), aspect="auto", cmap="gist_ncar_r")
plt.colorbar(cb)
```

(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)

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

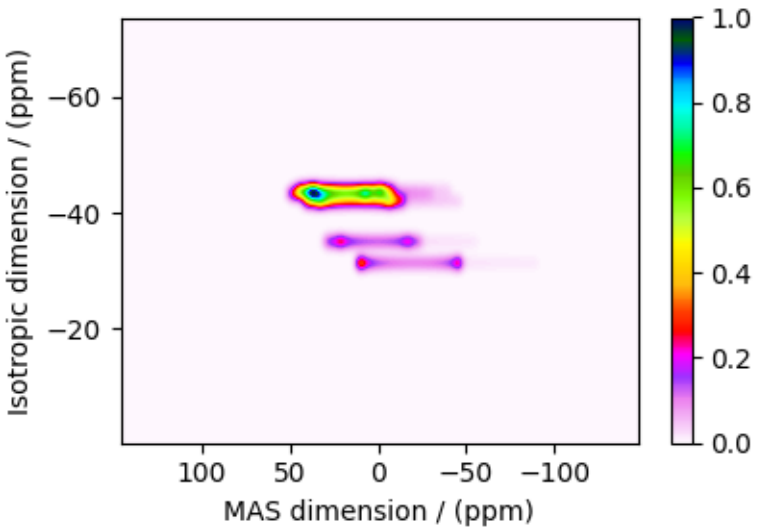


Add post-simulation signal processing.

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

The plot of the simulation after signal processing.

```
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
cb = ax.imshow(processed_dataset.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 0.922 seconds)

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

^{17}O (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 numpy as np
import matplotlib.pyplot as plt

from mrsimulator import Simulator
from mrsimulator import signal_processor as sp
from mrsimulator.method import Method, SpectralDimension, SpectralEvent, MixingEvent
```

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

```
sim = Simulator()

# load the spin systems from url.
filename = "https://ssnmr.org/sites/default/files/mrsimulator/coesite_0.mrsys"
sim.load_spin_systems(filename)
```

Use the generic *Method* class to simulate a 2D DAS spectrum by customizing the method parameters, as shown below.

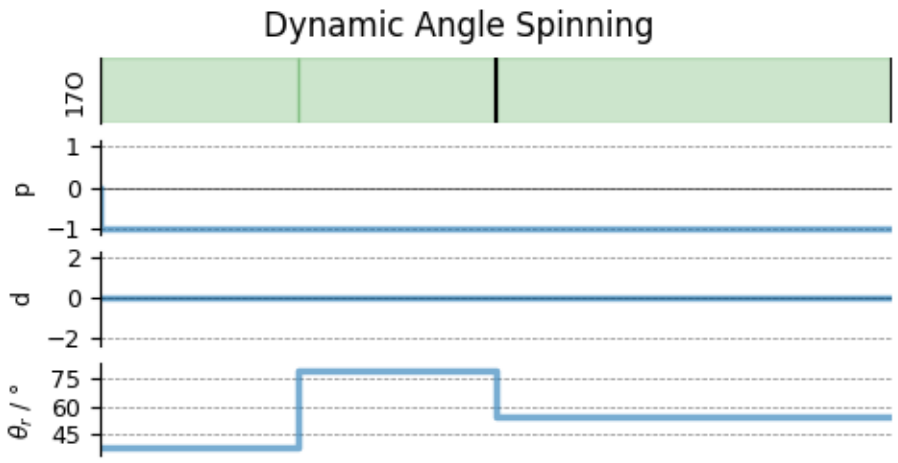
```
das = Method(
    name="Dynamic Angle Spinning",
    channels=["17O"],
    magnetic_flux_density=11.74,  # in T
    rotor_frequency=np.inf,
    spectral_dimensions=[
```

(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)

```
        SpectralDimension(
            count=256,
            spectral_width=5e3,  # in Hz
            reference_offset=0,  # in Hz
            label="DAS isotropic dimension",
            events=[
                SpectralEvent(
                    fraction=0.5,
                    rotor_angle=37.38 * np.pi / 180,  # in rads
                    transition_queries=[{"ch1": {"P": [-1], "D": [0]}}],
                ),
                MixingEvent(query="NoMixing"),
                SpectralEvent(
                    fraction=0.5,
                    rotor_angle=79.19 * np.pi / 180,  # in rads
                    transition_queries=[{"ch1": {"P": [-1], "D": [0]}}],
                ),
                MixingEvent(query="NoMixing"),
            ],
        ),
        # The last spectral dimension block is the direct-dimension
        SpectralDimension(
            count=256,
            spectral_width=2e4,  # in Hz
            reference_offset=0,  # in Hz
            label="MAS dimension",
            events=[
                SpectralEvent(
                    rotor_angle=54.735 * np.pi / 180,  # in rads
                    transition_queries=[{"ch1": {"P": [-1], "D": [0]}}],
                )
            ],
        ),
    ],
)
sim.methods = [das]  # add the method

# A graphical representation of the method object.
plt.figure(figsize=(5, 2.5))
das.plot()
plt.show()
```

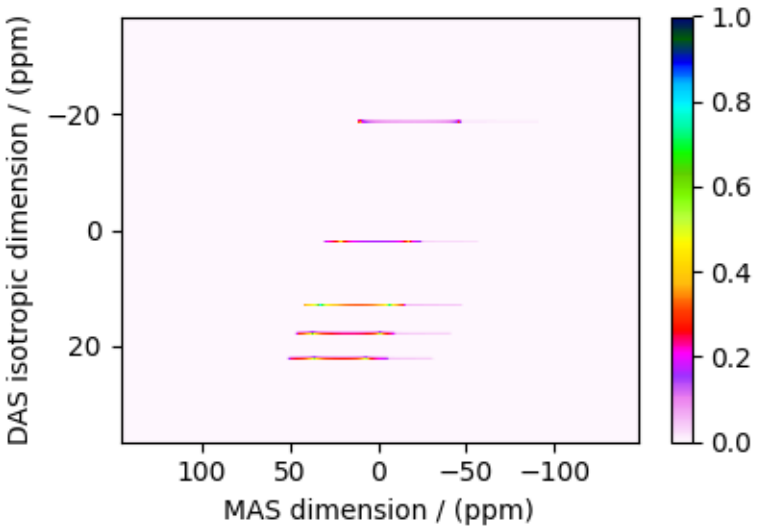


Run the simulation

```
sim.run()
```

The plot of the simulation.

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

plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
cb = ax.imshow(dataset.real / dataset.real.max(), aspect="auto", cmap="gist_ncar_r")
plt.colorbar(cb)
ax.invert_xaxis()
ax.invert_yaxis()
plt.tight_layout()
plt.show()
```

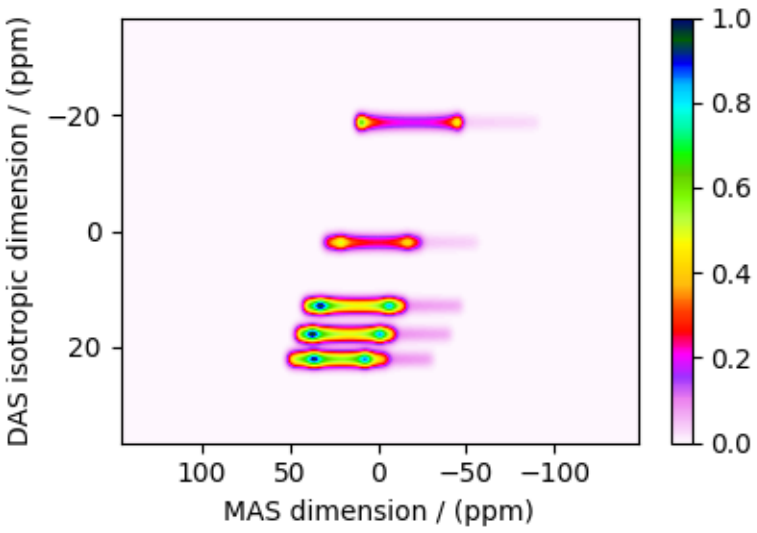


Add post-simulation signal processing.

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

The plot of the simulation after signal processing.

```
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
cb = ax.imshow(processed_dataset.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 1.011 seconds)

Rb_2CrO_4, ^{87}Rb (I=3/2) COASTER

^{87}Rb (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 numpy as np
import matplotlib.pyplot as plt

from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator import signal_processor as sp
from mrsimulator.spin_system.tensors import SymmetricTensor
from mrsimulator.method import Method, SpectralDimension, SpectralEvent, MixingEvent
```

Generate the site and spin system objects.

```
site = Site(
    isotope="87Rb",
    isotropic_chemical_shift=-9,  # in ppm
    shielding_symmetric=SymmetricTensor(zeta=110, eta=0),
    quadrupolar=SymmetricTensor(
        Cq=3.5e6,  # in Hz
        eta=0.36,
        alpha=0,  # in rads
        beta=70 * 3.14159 / 180,  # in rads
        gamma=0,  # in rads
    ),
)
spin_system = SpinSystem(sites=[site])
```

Use the generic *Method* class to simulate a 2D COASTER spectrum by customizing the method parameters, as shown below.

By default, all transitions selected from a *SpectralEvent* connect to all selected transitions from the following *SpectralEvent* if no *MixingEvent* is defined between them. Here, we define a *MixingEvent* with an angle of 109.5 degrees to connect the 3Q to 1Q transitions.

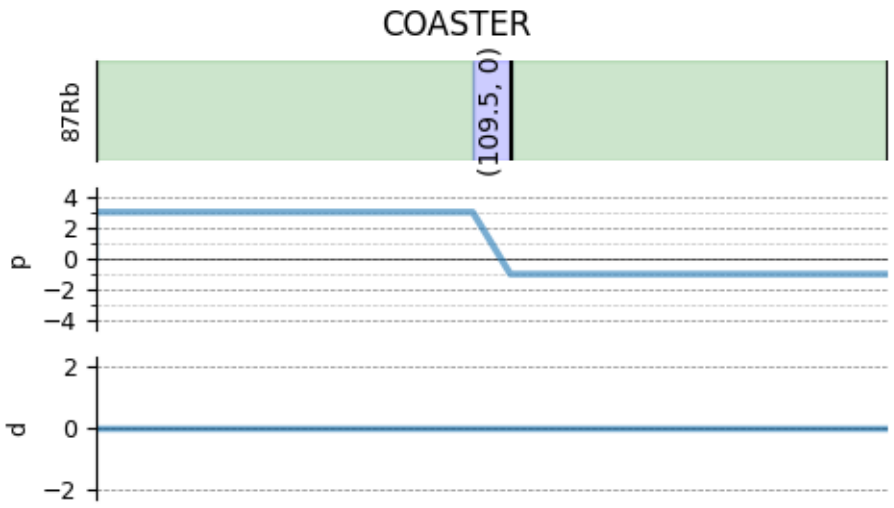
```
coaster = Method(
    name="COASTER",
    channels=["87Rb"],
    magnetic_flux_density=9.4,  # in T
    rotor_angle=70.12 * np.pi / 180,  # in rads
    rotor_frequency=np.inf,
    spectral_dimensions=[
        SpectralDimension(
            count=512,
            spectral_width=4e4,  # in Hz
            reference_offset=-8e3,  # in Hz
            label="$\\omega_1$ (CSA)",
            events=[
```

(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)

```
                SpectralEvent(transition_queries=[{"ch1": {"P": [3], "D": [0]}}]),
                MixingEvent(query={"ch1": {"angle": np.pi * 109.5 / 180, "phase": 0}}),
            ],
        ),
        # The last spectral dimension block is the direct-dimension
        SpectralDimension(
            count=512,
            spectral_width=8e3,  # in Hz
            reference_offset=-4e3,  # in Hz
            label="$\\omega_2$ (Q)",
            events=[SpectralEvent(transition_queries=[{"ch1": {"P": [-1], "D": [0]}}])],
        ),
    ],
    affine_matrix=[[1, 0], [1 / 4, 3 / 4]],
)

# A graphical representation of the method object.
plt.figure(figsize=(5, 2.75))
coaster.plot()
plt.show()
```



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

```
sim = Simulator(spin_systems=[spin_system], methods=[coaster])

# 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()
```

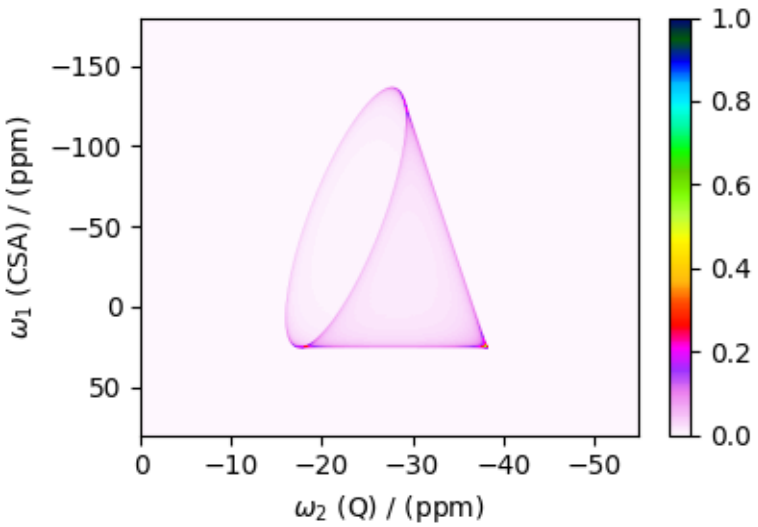
The plot of the simulation.

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

plt.figure(figsize=(4.25, 3.0))
```

(continues on next page)

(continued from previous page)

```
ax = plt.subplot(projection="csdm")
cb = ax.imshow(dataset.real / dataset.real.max(), aspect="auto", cmap="gist_ncar_r")
plt.colorbar(cb)
ax.set_xlim(-0, -55)
ax.set_ylim(80, -180)
plt.tight_layout()
plt.show()
```

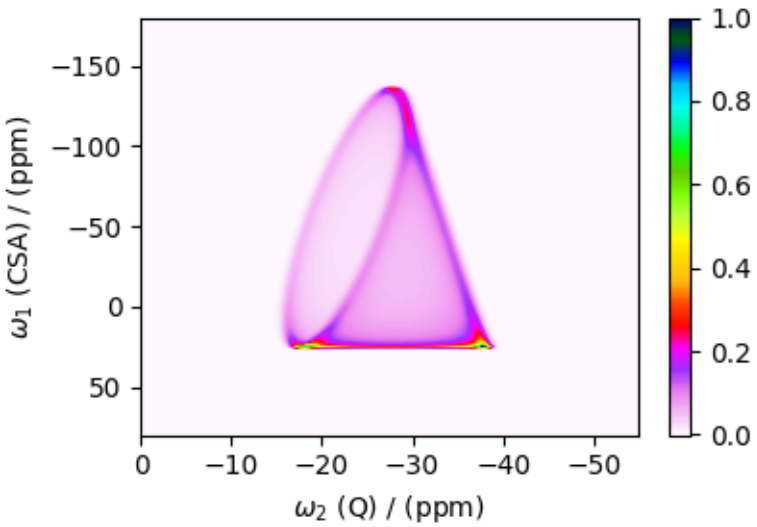


Add post-simulation signal processing.

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

The plot of the simulation after signal processing.

```
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
cb = ax.imshow(processed_dataset.real, cmap="gist_ncar_r", aspect="auto")
plt.colorbar(cb)
ax.set_xlim(-0, -55)
ax.set_ylim(80, -180)
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 1.059 seconds)

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

^{13}C (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.pyplot as plt

from mrsimulator import Simulator
from mrsimulator.method.lib import SSB2D
from mrsimulator import signal_processor as sp
from mrsimulator.method import SpectralDimension
```

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://ssnmr.org/sites/default/files/mrsimulator/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.

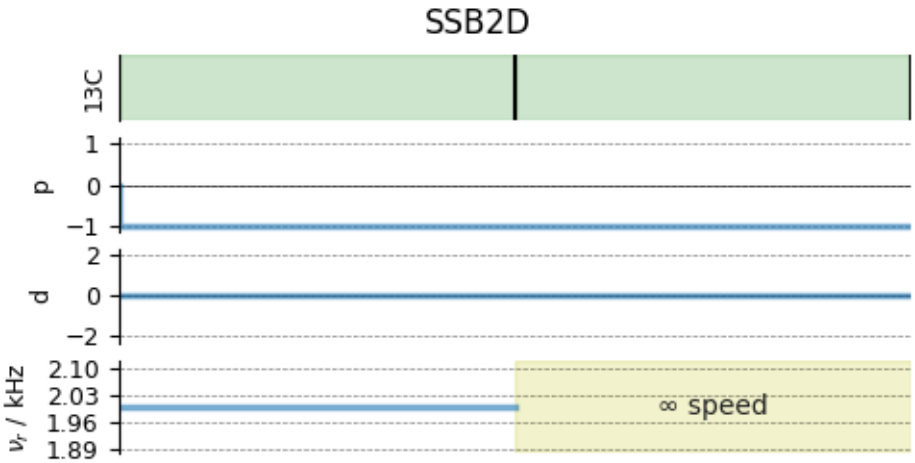
```
PASS = SSB2D(
    channels=["13C"],
    magnetic_flux_density=11.74,
    rotor_frequency=2000,
```

(continues on next page)

[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

(continued from previous page)

```
    spectral_dimensions=[
        SpectralDimension(
            count=20 * 4,
            spectral_width=2000 * 20,  # value in Hz
            label="Anisotropic dimension",
        ),
        SpectralDimension(
            count=1024,
            spectral_width=3e4,  # value in Hz
            reference_offset=1.1e4,  # value in Hz
            label="Isotropic dimension",
        ),
    ],
)
sim.methods = [PASS]  # add the method.

# A graphical representation of the method object.
plt.figure(figsize=(5, 2.5))
PASS.plot()
plt.show()
```



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.

```
dataset = sim.methods[0].simulation
processor = sp.SignalProcessor(
    operations=[
        sp.IFFT(dim_index=0),
        sp.apodization.Exponential(FWHM="100 Hz", dim_index=0),
        sp.FFT(dim_index=0),
```

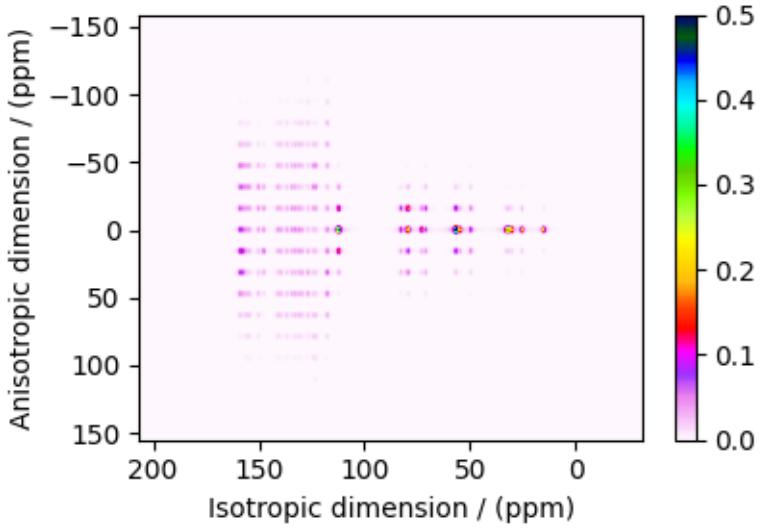
(continues on next page)

(continued from previous page)

```
    ]
)
processed_dataset = processor.apply_operations(dataset=dataset).real
processed_dataset /= processed_dataset.max()
```

The plot of the simulation.

```
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
cb = ax.imshow(processed_dataset, 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 1.045 seconds)

Rb_2SO_4, ^{87}Rb (I=3/2) QMAT

^{87}Rb (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.pyplot as plt

from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.method.lib import SSB2D
from mrsimulator.spin_system.tensors import SymmetricTensor
from mrsimulator.method import SpectralDimension
```

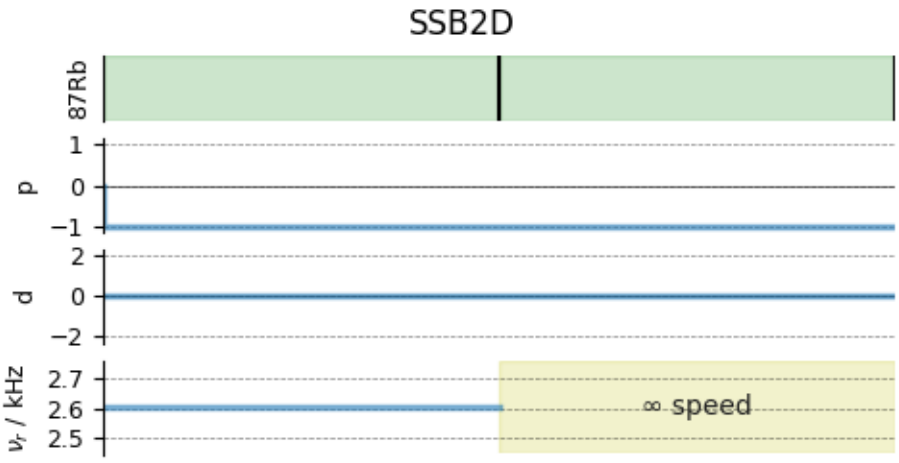
[1] Walder, B. J., Dey, K .K, Kaseman, D. C., Baltisberger, J. H., and Philip J. Grandinetti. Sideband separation experiments in NMR with phase incremented echo train acquisition, J. Chem. Phys. (2013) **138**, 174203. DOI:10.1063/1.4803142

Generate the site and spin system objects.

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

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=[
        SpectralDimension(
            count=32 * 4,
            spectral_width=2604 * 32,  # in Hz
            label="Anisotropic dimension",
        ),
        SpectralDimension(
            count=512,
            spectral_width=50000,  # in Hz
            label="Fast MAS dimension",
        ),
    ],
)

# A graphical representation of the method object.
plt.figure(figsize=(5, 2.5))
qmat.plot()
plt.show()
```

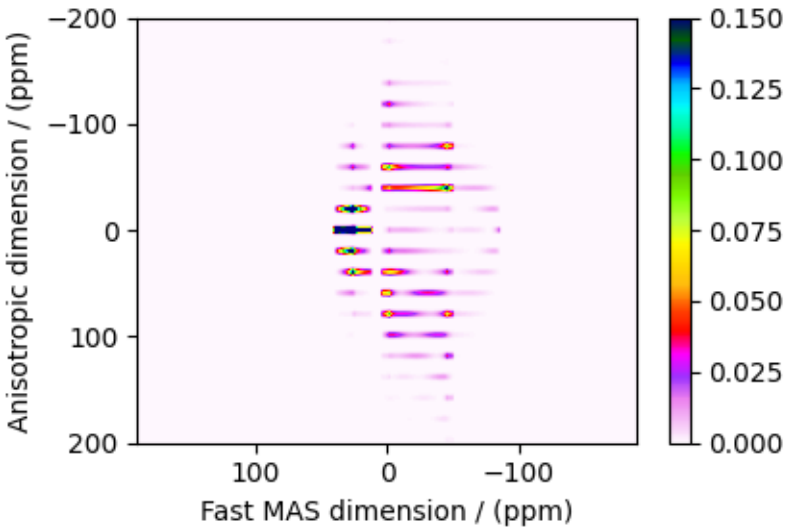


Create the Simulator object, add the method and spin system objects, and run the simulation.

```
sim = Simulator(spin_systems=spin_systems, methods=[qmat])

# 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.

```
plt.figure(figsize=(4.25, 3.0))
dataset = sim.methods[0].simulation.real
ax = plt.subplot(projection="csdm")
cb = ax.imshow(dataset / dataset.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 0.659 seconds)

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

^{29}Si (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 numpy as np
import matplotlib.pyplot as plt

from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator import signal_processor as sp
from mrsimulator.spin_system.tensors import SymmetricTensor
from mrsimulator.method import Method, SpectralDimension, SpectralEvent, MixingEvent
```

Create the sites and spin systems

```
sites = [
    Site(
        isotope="29Si",
        isotropic_chemical_shift=-89.0,  # in ppm
        shielding_symmetric=SymmetricTensor(zeta=59.8, eta=0.62),  # zeta in ppm
    ),
    Site(
        isotope="29Si",
        isotropic_chemical_shift=-89.5,  # in ppm
        shielding_symmetric=SymmetricTensor(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=SymmetricTensor(zeta=69.4, eta=0.60),  # zeta in ppm
    ),
]

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

Use the generic *Method* class to simulate a 2D magic-angle Flipping (MAF) spectrum by customizing the method parameters, as shown below.

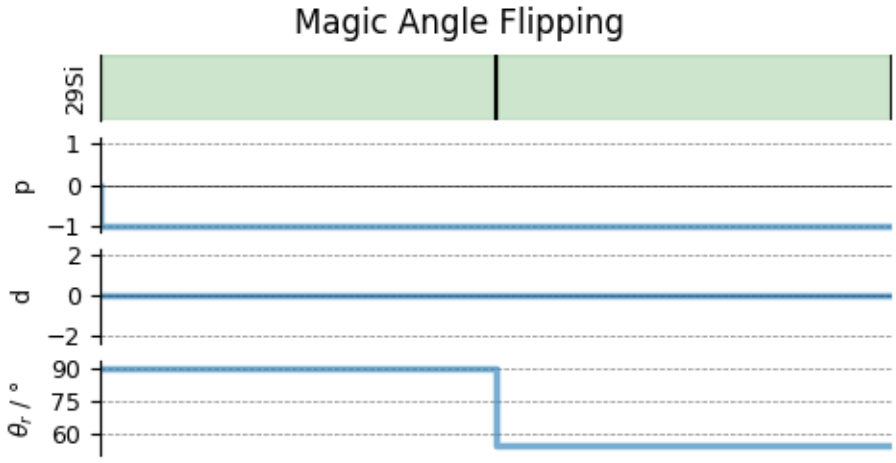
Here, we include a *MixingEvent* with a `NoMixing` query. A no mixing query instructs the MAF method to not mix the transitions from the first and second SpectralEvent. A no mixing query is equivalent to a rotation query where each channel has a zero phase and angle. Since all spin systems in this example have a single site, defining no mixing between the two spectral events is superfluous. We include it such that the method is applicable with multi-site spin systems.

```
maf = Method(
    name="Magic Angle Flipping",
    channels=["29Si"],
    magnetic_flux_density=14.1,  # in T
    rotor_frequency=np.inf,
    spectral_dimensions=[
        SpectralDimension(
            count=128,
            spectral_width=2e4,  # in Hz
            label="Anisotropic dimension",
            events=[
                SpectralEvent(
                    rotor_angle=90 * np.pi / 180,  # in rads
                    transition_queries=[{"ch1": {"P": [-1], "D": [0]}}],
                ),
                MixingEvent(query="NoMixing"),
            ],
        ),
        SpectralDimension(
            count=128,
            spectral_width=3e3,  # in Hz
            reference_offset=-1.05e4,  # in Hz
            label="Isotropic dimension",
            events=[
                SpectralEvent(
                    rotor_angle=54.735 * np.pi / 180,  # in rads
                    transition_queries=[{"ch1": {"P": [-1], "D": [0]}}],
                )
            ],
        ),
    ],
    affine_matrix=[[1, -1], [0, 1]],
)

# A graphical representation of the method object.
plt.figure(figsize=(5, 2.5))
```

(continues on next page)

(continued from previous page)

```
maf.plot()
plt.show()
```



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

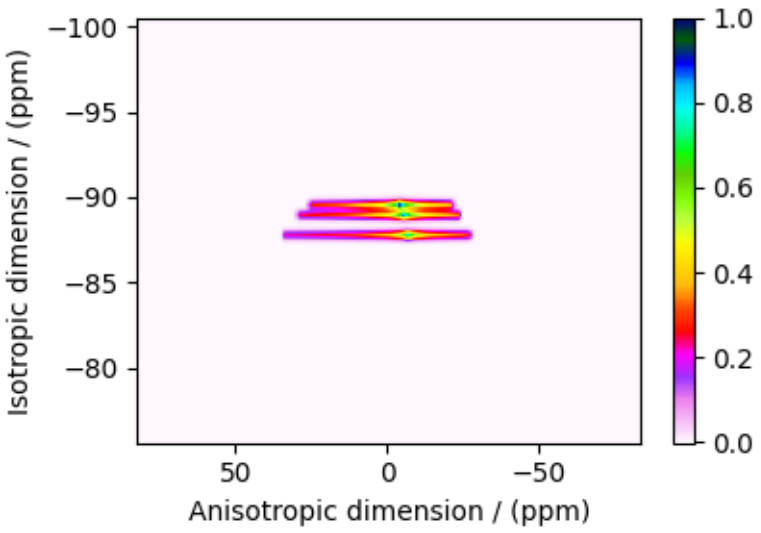
```
sim = Simulator(spin_systems=spin_systems, methods=[maf])
sim.run()
```

Add post-simulation signal processing.

```
csdm_dataset = sim.methods[0].simulation
processor = sp.SignalProcessor(
    operations=[
        sp.IFFT(dim_index=(0, 1)),
        sp.apodization.Gaussian(FWHM="50 Hz", dim_index=0),
        sp.apodization.Gaussian(FWHM="50 Hz", dim_index=1),
        sp.FFT(dim_index=(0, 1)),
    ]
)
processed_dataset = processor.apply_operations(dataset=csdm_dataset).real
processed_dataset /= processed_dataset.max()
```

The plot of the simulation after signal processing.

```
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
cb = ax.imshow(processed_dataset.T, aspect="auto", cmap="gist_ncar_r")
plt.colorbar(cb)
ax.invert_xaxis()
ax.invert_yaxis()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.594 seconds)

$MCl_2.2D_2O$, ^2H (I=1) Shifting-d echo

^2H (I=1) 2D NMR CSA-Quad 1st order correlation spectrum simulation.

The following is a static shifting-d echo NMR correlation simulation of $MCl_2 \cdot 2D_2O$ crystalline solid, where $M \in [Cu, Ni, Co, Fe, Mn]$. The tensor parameters for the simulation and the corresponding spectrum are reported by Walder *et al.*[1].

```
import matplotlib.pyplot as plt

from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator import signal_processor as sp
from mrsimulator.spin_system.tensors import SymmetricTensor
from mrsimulator.method import Method, SpectralDimension, SpectralEvent, MixingEvent
```

Generate the site and spin system objects.

```
site_Ni = Site(
    isotope="2H",
    isotropic_chemical_shift=-97,  # in ppm
    shielding_symmetric=SymmetricTensor(
        zeta=-551,  # in ppm
        eta=0.12,
        alpha=62 * 3.14159 / 180,  # in rads
        beta=114 * 3.14159 / 180,  # in rads
        gamma=171 * 3.14159 / 180,  # in rads
    ),
    quadrupolar=SymmetricTensor(Cq=77.2e3, eta=0.9),  # Cq in Hz
)
```

(continues on next page)

[1] Walder B.J, Patterson A.M., Baltisberger J.H, and Grandinetti P.J Hydrogen motional disorder in crystalline iron group chloride di-hydrates spectroscopy, J. Chem. Phys. (2018) **149**, 084503. DOI: 10.1063/1.5037151

(continued from previous page)

```
site_Cu = Site(
    isotope="2H",
    isotropic_chemical_shift=51,  # in ppm
    shielding_symmetric=SymmetricTensor(
        zeta=146,  # in ppm
        eta=0.84,
        alpha=95 * 3.14159 / 180,  # in rads
        beta=90 * 3.14159 / 180,  # in rads
        gamma=0 * 3.14159 / 180,  # in rads
    ),
    quadrupolar=SymmetricTensor(Cq=118.2e3, eta=0.8),  # Cq in Hz
)

site_Co = Site(
    isotope="2H",
    isotropic_chemical_shift=215,  # in ppm
    shielding_symmetric=SymmetricTensor(
        zeta=-1310,  # in ppm
        eta=0.23,
        alpha=180 * 3.14159 / 180,  # in rads
        beta=90 * 3.14159 / 180,  # in rads
        gamma=90 * 3.14159 / 180,  # in rads
    ),
    quadrupolar=SymmetricTensor(Cq=114.6e3, eta=0.95),  # Cq in Hz
)

site_Fe = Site(
    isotope="2H",
    isotropic_chemical_shift=101,  # in ppm
    shielding_symmetric=SymmetricTensor(
        zeta=-1187,  # in ppm
        eta=0.4,
        alpha=122 * 3.14159 / 180,  # in rads
        beta=90 * 3.14159 / 180,  # in rads
        gamma=90 * 3.14159 / 180,  # in rads
    ),
    quadrupolar=SymmetricTensor(Cq=114.2e3, eta=0.98),  # Cq in Hz
)

site_Mn = Site(
    isotope="2H",
    isotropic_chemical_shift=145,  # in ppm
    shielding_symmetric=SymmetricTensor(
        zeta=-1236,  # in ppm
        eta=0.23,
        alpha=136 * 3.14159 / 180,  # in rads
        beta=90 * 3.14159 / 180,  # in rads
        gamma=90 * 3.14159 / 180,  # in rads
    ),
    quadrupolar=SymmetricTensor(Cq=1.114e5, eta=1.0),  # Cq in Hz
)
```

(continues on next page)

(continued from previous page)

```
spin_systems = [
    SpinSystem(sites=[s], name=f"{n}Cl$_2$.2D$_2$O")
    for s, n in zip(
        [site_Ni, site_Cu, site_Co, site_Fe, site_Mn], ["Ni", "Cu", "Co", "Fe", "Mn"]
    )
]
```

Use the generic *Method* class to generate a 2D shifting-d echo method. The reported shifting-d 2D sequence correlates the shielding frequencies to the first-order quadrupolar frequencies. Here, we create a correlation method using the `freq_contrib` attribute, which acts as a switch for including the frequency contributions from interactions during an event.

In the following method, we assign the `["Quad1_2"]` and `["Shielding1_0", "Shielding1_2"]` as the value to the `freq_contrib` key. The *Quad1_2* is an enumeration for selecting the first-order second-rank quadrupolar frequency contributions. *Shielding1_0* and *Shielding1_2* are enumerations for the first-order shielding with zeroth and second-rank tensor contributions, respectively. See *FrequencyEnum* (page 416) for details.

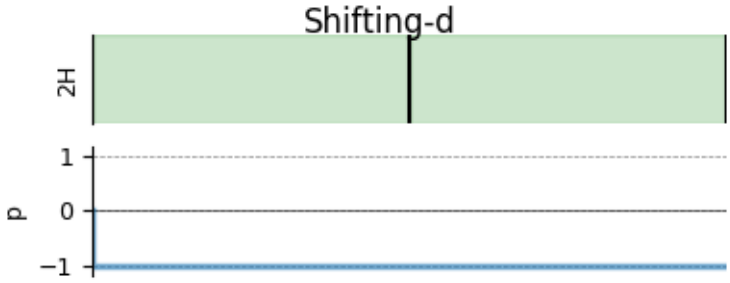
Like the previous example, we stipulate no mixing between the two spectral events using a *MixingEvent* with `NoMixing` as the query. Since all spin systems in this example have a single site, defining no mixing between the two spectral events is superfluous. We include it such that the method is applicable with multi-site spin systems.

```
shifting_d = Method(
    name="Shifting-d",
    channels=["2H"],
    magnetic_flux_density=9.395,  # in T
    rotor_frequency=0,  # in Hz
    rotor_angle=0,  # in Hz
    spectral_dimensions=[
        SpectralDimension(
            count=512,
            spectral_width=2.5e5,  # in Hz
            label="Quadrupolar frequency",
            events=[
                SpectralEvent(
                    transition_queries=[{"ch1": {"P": [-1]}}],
                    freq_contrib=["Quad1_2"],
                ),
                MixingEvent(query="NoMixing"),
            ],
        ),
        SpectralDimension(
            count=256,
            spectral_width=2e5,  # in Hz
            reference_offset=2e4,  # in Hz
            label="Paramagnetic shift",
            events=[
                SpectralEvent(
                    transition_queries=[{"ch1": {"P": [-1]}}],
                    freq_contrib=["Shielding1_0", "Shielding1_2"],
                )
            ],
        ),
    ],
```

(continues on next page)

(continued from previous page)

```
)

# A graphical representation of the method object.
plt.figure(figsize=(4, 1.5))
shifting_d.plot()
plt.show()
```



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

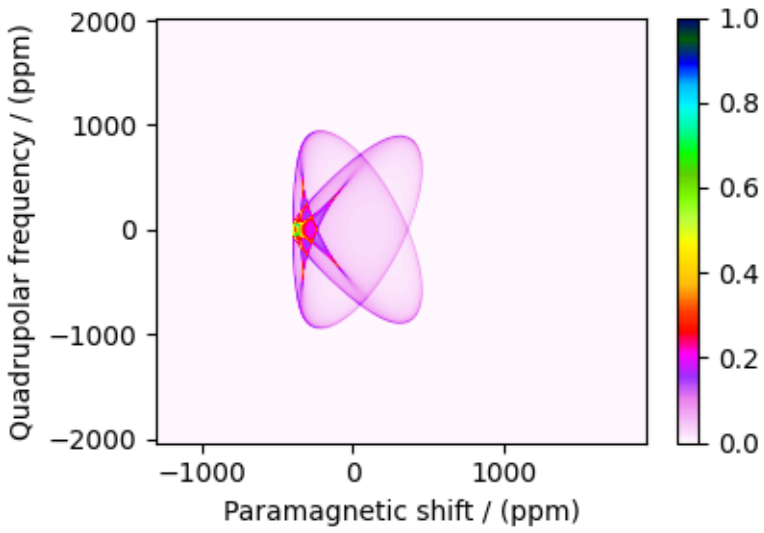
```
sim = Simulator(spin_systems=spin_systems, methods=[shifting_d])
# Configure the simulator object. For non-coincidental tensors, set the value of the
# `integration_volume` attribute to `hemisphere`.
sim.config.integration_volume = "hemisphere"
sim.config.decompose_spectrum = "spin_system"  # simulate spectra per spin system
sim.run()
```

Add post-simulation signal processing.

```
dataset = sim.methods[0].simulation
processor = sp.SignalProcessor(
    operations=[
        # Gaussian convolution along both dimensions.
        sp.IFFT(dim_index=(0, 1)),
        sp.apodization.Gaussian(FWHM="9 kHz", dim_index=0),  # along dimension 0
        sp.apodization.Gaussian(FWHM="9 kHz", dim_index=1),  # along dimension 1
        sp.FFT(dim_index=(0, 1)),
    ]
)
processed_dataset = processor.apply_operations(dataset=dataset)
```

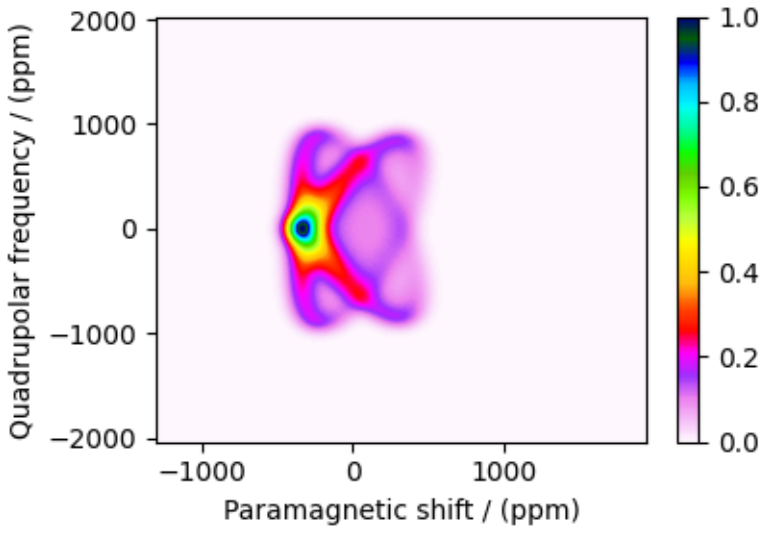
The plot of the simulation. Because we configured the simulator object to simulate spectrum per spin system, the following dataset is a CSDM object containing five simulations (dependent variables). Let's visualize the first dataset corresponding to $NiCl_2 \cdot 2D_2O$.

```
dataset_Ni = dataset.split()[0].real

plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
cb = ax.imshow(dataset_Ni / dataset_Ni.max(), aspect="auto", cmap="gist_ncar_r")
plt.title(None)
plt.colorbar(cb)
plt.tight_layout()
plt.show()
```



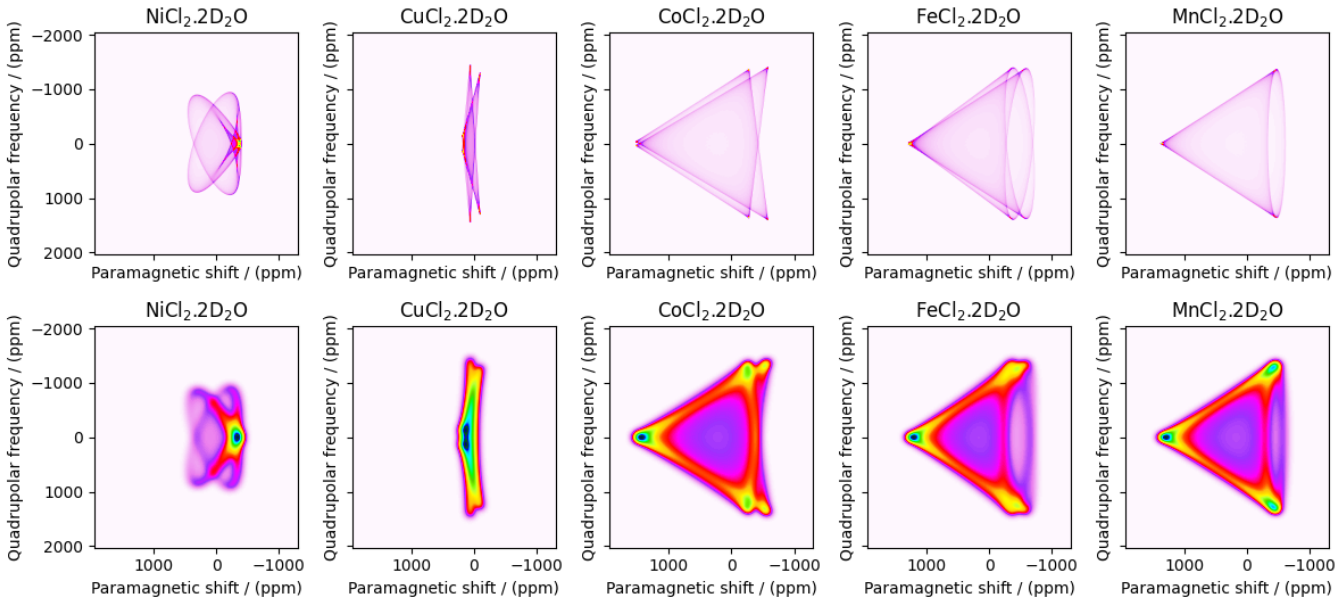
The plot of the simulation after signal processing.

```
proc_dataset_Ni = processed_dataset.split()[0].real

plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
cb = ax.imshow(
    proc_dataset_Ni / proc_dataset_Ni.max(), cmap="gist_ncar_r", aspect="auto"
)
plt.title(None)
plt.colorbar(cb)
plt.tight_layout()
plt.show()
```



Let's plot all the simulated datasets.

```
fig, ax = plt.subplots(
    2, 5, sharex=True, sharey=True, figsize=(12, 5.5), subplot_kw={"projection": "csdm"}
)
for i, dataset_obj in enumerate([dataset, processed_dataset]):
    for j, datum in enumerate(dataset_obj.split()):
        ax[i, j].imshow((datum / datum.max()).real, aspect="auto", cmap="gist_ncar_r")
        ax[i, j].invert_xaxis()
        ax[i, j].invert_yaxis()

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



Total running time of the script: (0 minutes 2.857 seconds)

^{2}H (I=1) 2D sideband-sideband correlation

^{2}H (I=1) 2D NMR CSA-Quad 1st order sideband correlation spectrum simulation.

Sideband-sideband NMR correlation simulation of crystalline solid as reported by Aleksis and Pell[1].

```
import matplotlib.pyplot as plt

from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.spin_system.tensors import SymmetricTensor
from mrsimulator.method import Method, SpectralDimension, SpectralEvent, MixingEvent
from mrsimulator.simulator.sampling_scheme import zcw_averaging
```

Generate the site and spin system objects.

[1] Aleksis R and Pell A.J., Separation of quadrupolar and paramagnetic shift interactions in high-resolution nuclear magnetic resonance of spinning powders, J. Chem. Phys. (2021) **155**, 094202. DOI: 10.1063/5.0061611

```
alpha = [0, 30, 60, 90, 0, 0, 0, 0, 90, 90, 90, 90]
beta = [0, 0, 0, 0, 0, 30, 60, 90, 90, 90, 90, 90]
gamma = [0, 0, 0, 0, 90, 90, 90, 90, 0, 30, 60, 90]

spin_systems = [
    SpinSystem(
        name=f"$\\alpha={al}, \\beta={be}, \\gamma={ga}$",
        sites=[
            Site(
                isotope="2H",
                isotropic_chemical_shift=0,  # in ppm
                shielding_symmetric=SymmetricTensor(
                    zeta=150,  # in ppm
                    eta=0.7,
                    alpha=al * 3.14159 / 180,  # in rads
                    beta=be * 3.14159 / 180,  # in rads
                    gamma=ga * 3.14159 / 180,  # in rads
                ),
                quadrupolar=SymmetricTensor(Cq=50e3, eta=0.9),  # Cq in Hz
            )
        ],
    )
    for al, be, ga in zip(alpha, beta, gamma)
]
```

Create a sideband-sideband correlation method

```
rotor_frequency = 2000

sideband_2d = Method(
    name="2D sideband correlation",
    channels=["2H"],
    magnetic_flux_density=9.395,  # in T
    rotor_frequency=rotor_frequency,  # in Hz
    spectral_dimensions=[
        SpectralDimension(
            count=16,
            spectral_width=rotor_frequency * 16,  # in Hz
            label="Paramagnetic shift",
            events=[
                SpectralEvent(
                    transition_queries=[{"ch1": {"P": [-1]}}],
                    freq_contrib=["Shielding1_2"],
                ),
                MixingEvent(query="NoMixing"),
            ],
        ),
        SpectralDimension(
            count=50,
            spectral_width=50 * rotor_frequency,  # in Hz
            label="Quadrupolar + iso frequency",
            events=[
                SpectralEvent(
```

(continues on next page)

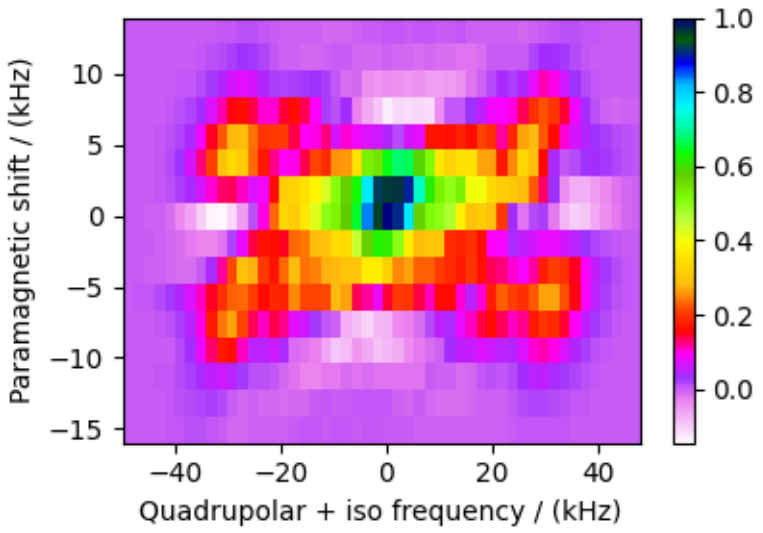
(continued from previous page)

```
                    transition_queries=[{"ch1": {"P": [-1]}}],
                    freq_contrib=["Quad1_2", "Shielding1_0"],
                )
            ],
        ),
    ],
)
```

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

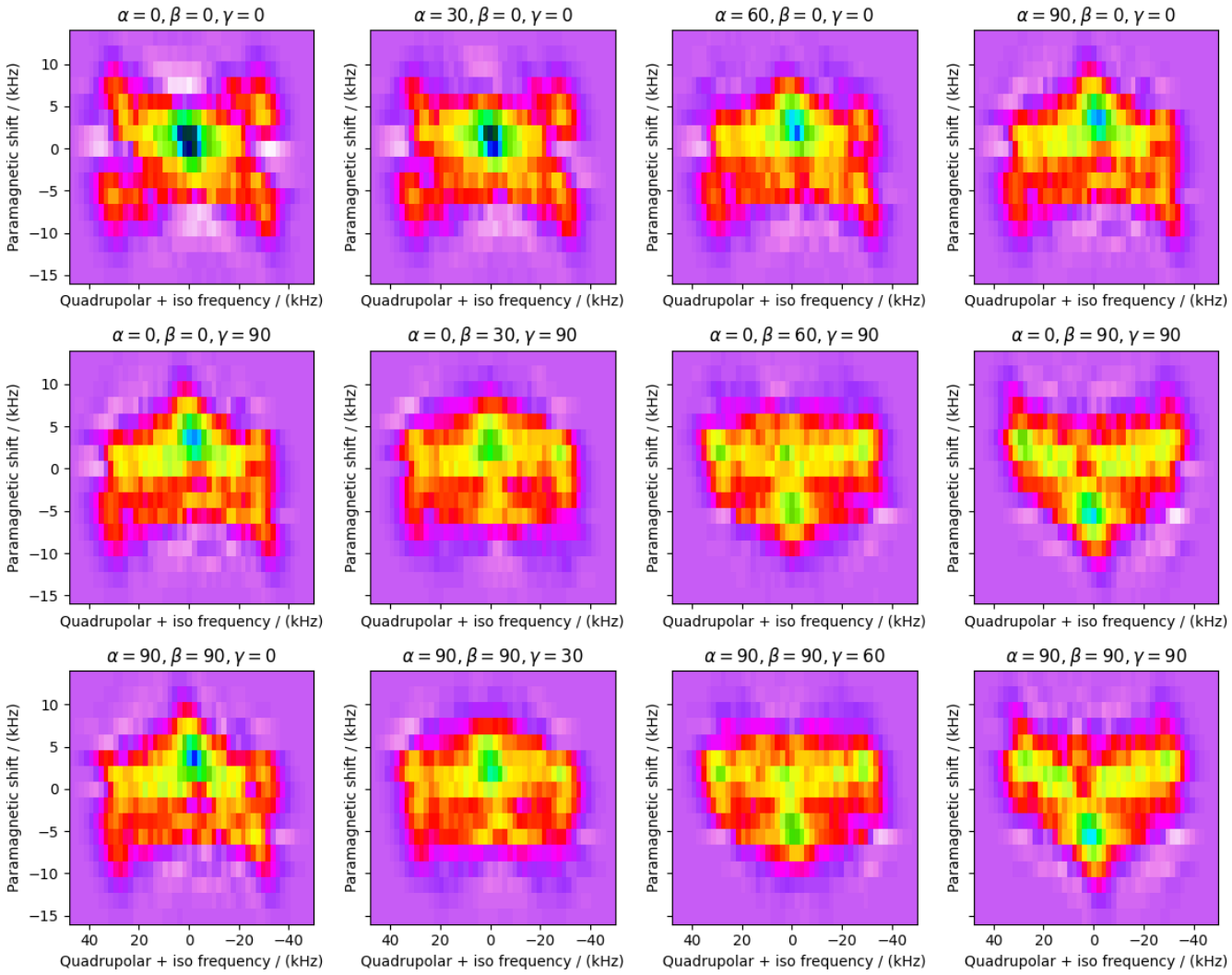
```
sim = Simulator(spin_systems=spin_systems, methods=[sideband_2d])
sim.config.decompose_spectrum = "spin_system"  # simulate spectra per spin system
sim.config.number_of_sidebands = 56

# custom sampling scheme
sim.config.custom_sampling = zcw_averaging(
    M=11, integration_volume="hemisphere", triangle_mesh=False
)

sim.run()
```

```
dataset = sim.methods[0].simulation.real
_ = [dim.to("kHz", "nmr_frequency_ratio") for dim in dataset.dimensions]
datasets = dataset.split()

plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
cb = ax.imshow(
    datasets[0] / datasets[0].max(),
    aspect="auto",
    cmap="gist_ncar_r",
    interpolation="none",
)
plt.title(None)
plt.colorbar(cb)
plt.tight_layout()
plt.show()
```



Let's plot all the simulated datasets.

```
fig, ax = plt.subplots(
    3, 4, sharex=True, sharey=True, figsize=(12, 9.5), subplot_kw={"projection": "csdm"}
)

vmax = max(dataset.max())
vmin = min(dataset.min())
for j, datum in enumerate(datasets):
    row, col = j // 4, j % 4
    ax[row, col].imshow(
        datum,
        aspect="auto",
        cmap="gist_ncar_r",
        interpolation="none",
        vmax=vmax,
        vmin=vmin,
    )
    ax[row, col].invert_xaxis()

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



Total running time of the script: (0 minutes 5.576 seconds)

12.4.4 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 438))

Simulating site disorder (crystalline)

^{87}Rb (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 numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal

from mrsimulator import Simulator
from mrsimulator.method.lib import ThreeQ_VAS
from mrsimulator.models import ExtCzjzekDistribution
from mrsimulator.utils.collection import single_site_system_generator
from mrsimulator.method import SpectralDimension
```

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
```

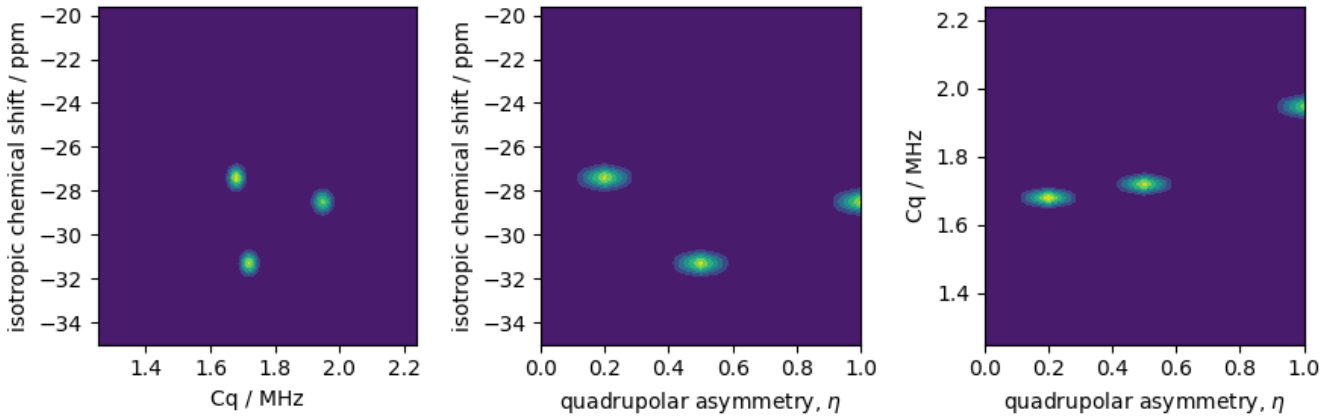
(continues on next page)

(continued from previous page)

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



Simulation setup

Generate spin systems from the above probability distribution.

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

(continues on next page)

(continued from previous page)

```
)
len(spin_systems)
```

```
515
```

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

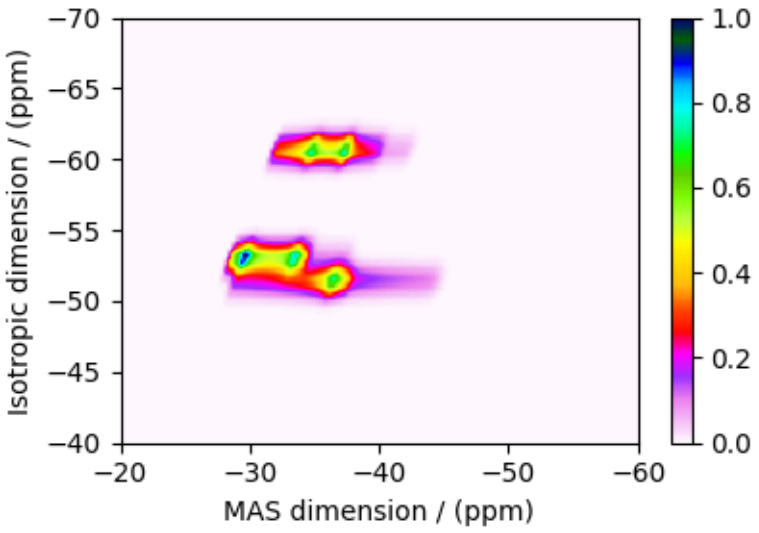
```
method = ThreeQ_VAS(
    channels=["87Rb"],
    magnetic_flux_density=9.4,  # in T
    rotor_angle=54.735 * np.pi / 180,
    spectral_dimensions=[
        SpectralDimension(
            count=96,
            spectral_width=7e3,  # in Hz
            reference_offset=-7e3,  # in Hz
            label="Isotropic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

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

```
sim = Simulator(spin_systems=spin_systems, methods=[method])
sim.config.number_of_sidebands = 1
sim.run()

dataset = sim.methods[0].simulation.real
```

The plot of the corresponding spectrum.

```
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
cb = ax.imshow(dataset / dataset.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 13.767 seconds)

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

^{27}Al (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 numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal

from mrsimulator import Simulator
from mrsimulator.method.lib import ThreeQ_VAS
from mrsimulator.models import CzjzekDistribution
from mrsimulator.utils.collection import single_site_system_generator
from mrsimulator.method import SpectralDimension
```

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])
```

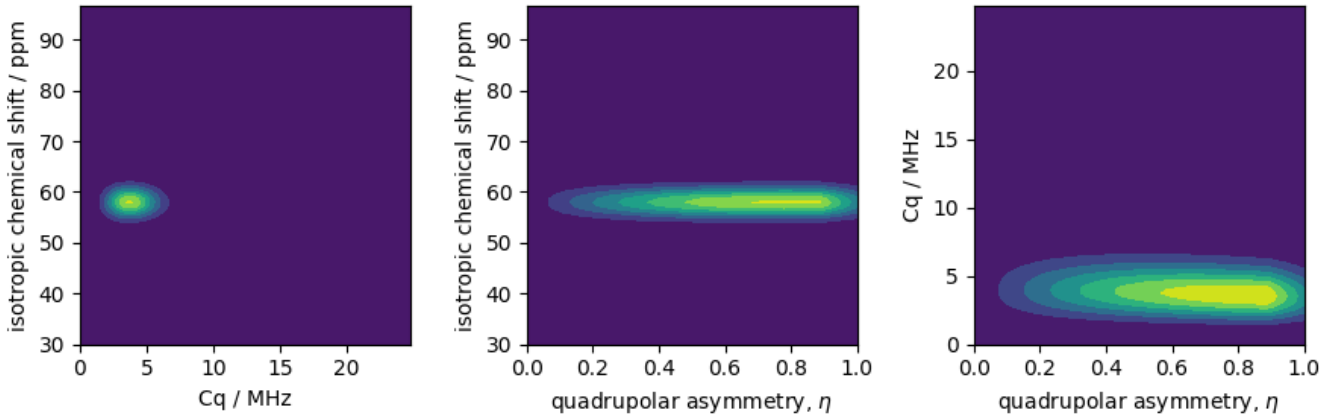
(continues on next page)

(continued from previous page)

```
# 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()* (page 484) utility function to generate single-site spin systems.

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

```
5468
```

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

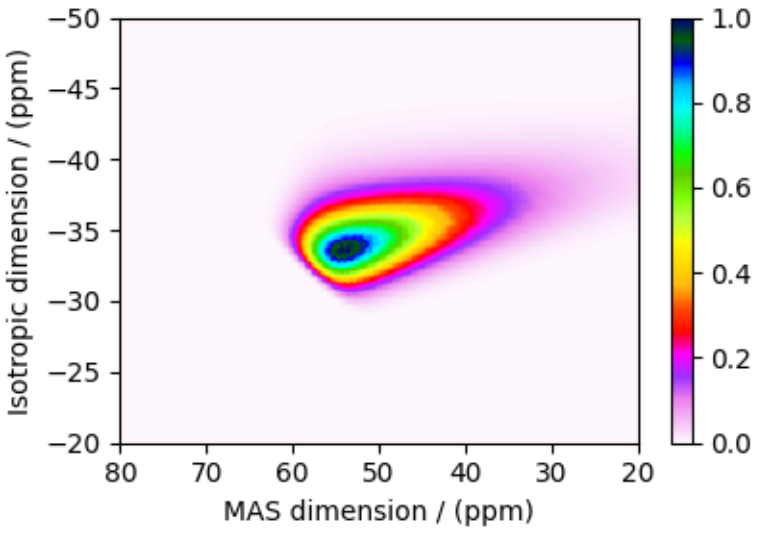
```
mqvas = ThreeQ_VAS(
    channels=["27Al"],
    spectral_dimensions=[
        SpectralDimension(
            count=512,
            spectral_width=26718.475776,  # in Hz
            reference_offset=-4174.76184,  # in Hz
            label="Isotropic dimension",
        ),
        SpectralDimension(
            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(spin_systems=spin_systems, methods=[mqvas])
sim.config.number_of_sidebands = 1
sim.run()

dataset = sim.methods[0].simulation.real
```

The plot of the corresponding spectrum.

```
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
cb = ax.imshow(dataset / dataset.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 25.436 seconds)

CHAPTER

THIRTEEN

FITTING (LEAST SQUARES) GALLERY

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 dataset.

13.1 1D Dataset Fitting

13.2 2D Dataset Fitting

13.2.1 1D Dataset Fitting

^{31}P MAS NMR of crystalline Na2PO4 (CSA)

In this example, we illustrate the use of the mrsimulator objects to

- create a CSA fitting model using Simulator and SignalProcessor objects,
- use the fitting model to perform a least-squares analysis, and
- extract the fitting parameters from the model.

We use the LMFIT library to fit the spectrum. The following example shows the least-squares fitting procedure applied to the ^{31}P MAS NMR spectrum of Na_2PO_4. The following experimental dataset is a part of DMFIT[1] examples. We thank Dr. Dominique Massiot for sharing the dataset.

Start by importing the relevant modules.

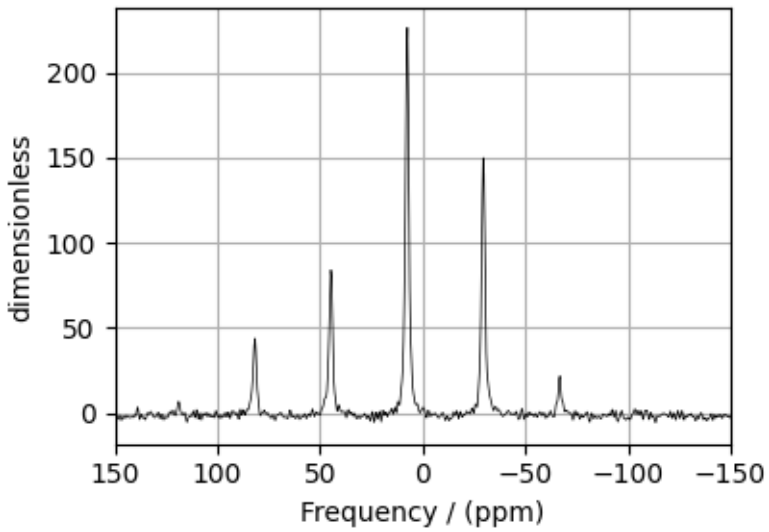
```
import csdmpy as cp
import numpy as np
import matplotlib.pyplot as plt
from lmfit import Minimizer

from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.method.lib import BlochDecaySpectrum
from mrsimulator import signal_processor as sp
from mrsimulator.utils import spectral_fitting as sf
from mrsimulator.utils import get_spectral_dimensions
from mrsimulator.spin_system.tensors import SymmetricTensor
```

[1] D.Massiot, F.Fayon, M.Capron, I.King, S.Le Calvé, B.Alonso, J.O.Durand, B.Bujoli, Z.Gan, G.Hoatson, 'Modelling one and two-dimensional solid-state NMR spectra.', Magn. Reson. Chem. **40** 70-76 (2002) DOI: 10.1002/mrc.984

Import the dataset

Import the experimental data. We use dataset file serialized with the CSDM file-format, using the csdmpy module.

```
host = "https://nmr.cemhti.cnrs-orleans.fr/Dmfit/Help/csdm/"
filename = "31P Phosphate 6kHz.csdf"
experiment = cp.load(host + filename)

# For spectral fitting, we only focus on the real part of the complex dataset
experiment = experiment.real

# Convert the dimension coordinates from Hz to ppm.
experiment.x[0].to("ppm", "nmr_frequency_ratio")

# plot of the dataset.
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(experiment, color="black", linewidth=0.5, label="Experiment")
ax.set_xlim(150, -150)
plt.grid()
plt.tight_layout()
plt.show()
```



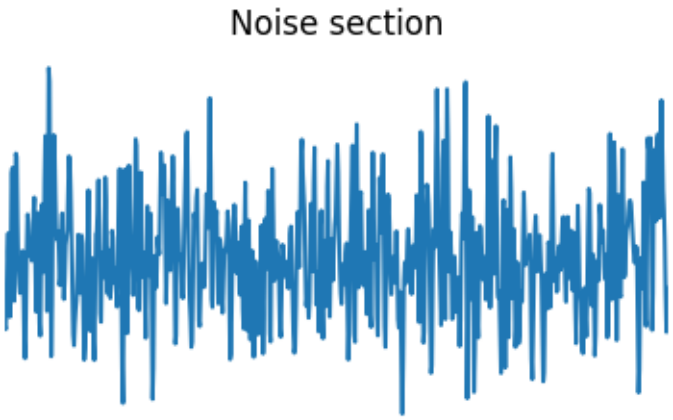
Estimate noise statistics from the dataset

```
coords = experiment.dimensions[0].coordinates
noise_region = np.where(coords < -100e-6)
noise_data = experiment[noise_region]

plt.figure(figsize=(3.75, 2.5))
ax = plt.subplot(projection="csdm")
ax.plot(noise_data, label="noise")
plt.title("Noise section")
plt.axis("off")
plt.tight_layout()
```

(continues on next page)

(continued from previous page)

```
plt.show()

noise_mean, sigma = experiment[noise_region].mean(), experiment[noise_region].std()
noise_mean, sigma
```




```
(<Quantity -1.6514109>, <Quantity 1.5352644>)
```

Create a fitting model

A fitting model is a composite of `Simulator` and `SignalProcessor` objects.

Step 1: Create initial guess sites and spin systems

```
P_31 = Site(
    isotope="31P",
    isotropic_chemical_shift=5.0,  # in ppm,
    shielding_symmetric=SymmetricTensor(zeta=-80, eta=0.5),  # zeta in Hz
)

spin_systems = [SpinSystem(sites=[P_31])]
```

Step 2: Create the method object. Create an appropriate method object that closely resembles the technique used in acquiring the experimental dataset. The attribute values of this method must meet the experimental conditions, 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 Bloch decay spectrum method where the spectral/spectroscopic dimension information, i.e., count, spectral_width, and the reference_offset, is extracted from the CSDM dimension metadata using the *get_spectral_dimensions()* (page 487) utility function. 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(experiment)

MAS = BlochDecaySpectrum(
    channels=["31P"],
```

(continues on next page)

(continued from previous page)

```
    magnetic_flux_density=9.395,  # in T
    rotor_frequency=6000,  # in Hz
    spectral_dimensions=spectral_dims,
    experiment=experiment,  # experimental dataset
)
```

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

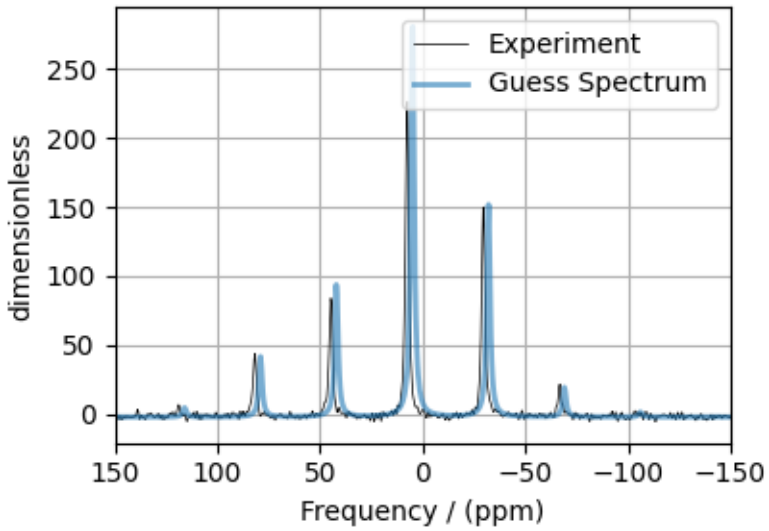
```
sim = Simulator(spin_systems=spin_systems, methods=[MAS])
sim.run()
```

Step 4: Create a SignalProcessor class object and apply the post-simulation signal processing operations.

```
processor = sp.SignalProcessor(
    operations=[
        sp.IFFT(),
        sp.apodization.Exponential(FWHM="0.3 kHz"),
        sp.FFT(),
        sp.Scale(factor=3000),
        sp.baseline.ConstantOffset(offset=-2),
    ]
)
processed_dataset = processor.apply_operations(dataset=sim.methods[0].simulation).real
```

Step 5: The plot of the dataset and the guess spectrum.

```
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(experiment, color="black", linewidth=0.5, label="Experiment")
ax.plot(processed_dataset, linewidth=2, alpha=0.6, label="Guess Spectrum")
ax.set_xlim(150, -150)
plt.legend()
plt.grid()
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 minimization. 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 497), to simplifies the LMFIT parameters generation process. By default, the function only creates parameters from the SpinSystem and SignalProcessor objects. Often, in spectrum with sidebands, spinning speed may not be accurately known; and is, therefore, included as a fitting parameter. To include a keyword from the method object, use the *include* argument of the function, as follows,

Step 6: Create a list of parameters.

```
params = sf.make_LMFIT_params(sim, processor, include={"rotor_frequency"})
```

The *make_ LMFIT_ params* parses the instances of the `Simulator` and the `PostSimulator` objects for parameters and returns a LMFIT *Parameters* object.

Customize the Parameters: You may customize the parameters list, `params`, as desired. Here, we remove the abundance parameter.

```
params.pop("sys_0_abundance")
print(params.pretty_print(columns=["value", "min", "max", "vary", "expr"]))
```

```
Name                                    Value      Min      Max     Vary     Expr
SP_0_operation_1_Exponential_FWHM         0.3     -inf      inf     True     None
SP_0_operation_3_Scale_factor            3000     -inf      inf     True     None
SP_0_operation_4_ConstantOffset_offset     -2     -inf      inf     True     None
mth_0_rotor_frequency                    6000     5900     6100     True     None
sys_0_site_0_isotropic_chemical_shift       5     -inf      inf     True     None
sys_0_site_0_shielding_symmetric_eta      0.5        0        1     True     None
sys_0_site_0_shielding_symmetric_zeta      -80     -inf      inf     True     None
None
```

Step 7: Perform the least-squares minimization. A method object queries every spin system for a list of transition pathways that are relevant to the given method. Since the method and the number of spin systems remains unchanged

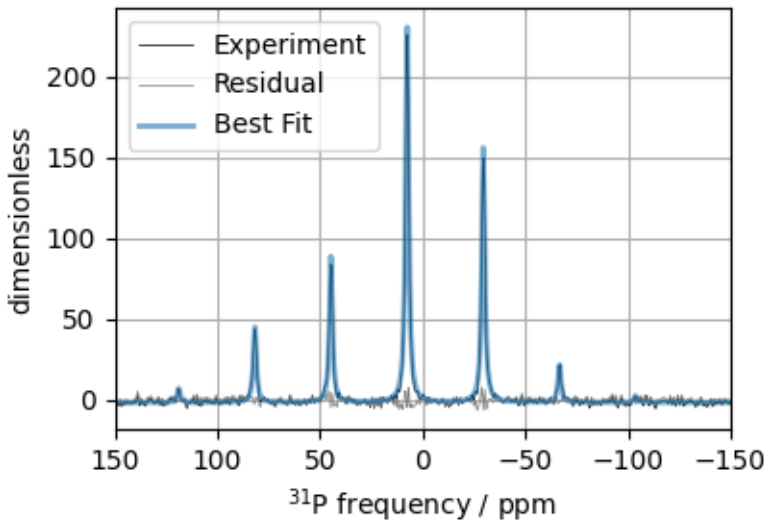
during the least-squares analysis, a one-time query is sufficient. To avoid querying for the transition pathways at every iteration in a least-squares fitting, call the *optimize()* (page 383) method to pre-compute the pathways.

For the user's convenience, we also provide a utility function, *LMFIT_min_function()* (page 497), for evaluating the difference vector between the simulation and experiment, based on the parameters update. You may use this function directly to instantiate the LMFIT Minimizer class where *fcn_args* and *fcn_kws* are arguments passed to the function, as follows,

```
opt = sim.optimize()  # Pre-compute transition pathways
minner = Minimizer(
    sf.LMFIT_min_function,
    params,
    fcn_args=(sim, processor, sigma),
    fcn_kws={"opt": opt},
)
result = minner.minimize()
result
```

Step 8: The plot of the fit, measurement, and residuals.

```
# Best fit spectrum
best_fit = sf.bestfit(sim, processor)[0].real
residuals = sf.residuals(sim, processor)[0].real

plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(experiment, color="black", linewidth=0.5, label="Experiment")
ax.plot(residuals, color="gray", linewidth=0.5, label="Residual")
ax.plot(best_fit, linewidth=2, alpha=0.6, label="Best Fit")
ax.set_xlabel(r"$^{31}$P frequency / ppm")
ax.set_xlim(150, -150)
plt.legend()
plt.grid()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 2.901 seconds)

^{31}P static NMR of crystalline Na2PO4 (CSA)

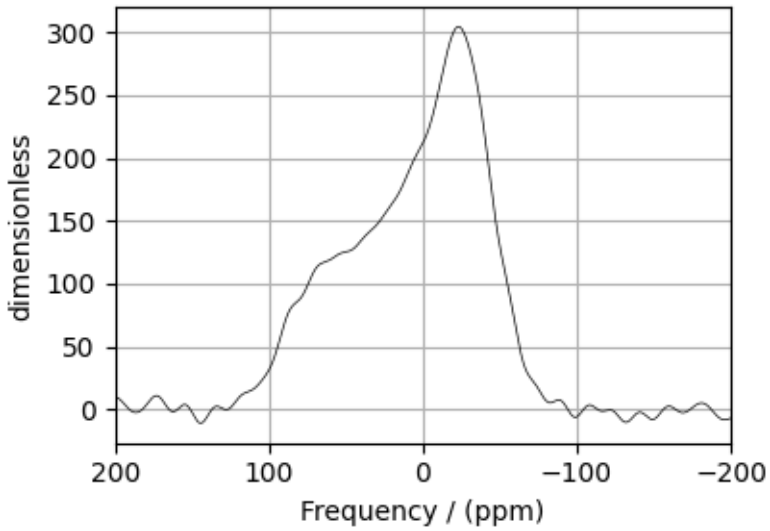
The following is a CSA static least-squares fitting example of a ^{31}P MAS NMR spectrum of Na_2PO_4. The following experimental dataset is a part of DMFIT[1] examples. We thank Dr. Dominique Massiot for sharing the dataset.

```
import csdmpy as cp
import numpy as np
import matplotlib.pyplot as plt
from lmfit import Minimizer

from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.method.lib import BlochDecaySpectrum
from mrsimulator import signal_processor as sp
from mrsimulator.utils import spectral_fitting as sf
from mrsimulator.utils import get_spectral_dimensions
from mrsimulator.spin_system.tensors import SymmetricTensor
```

Import the dataset

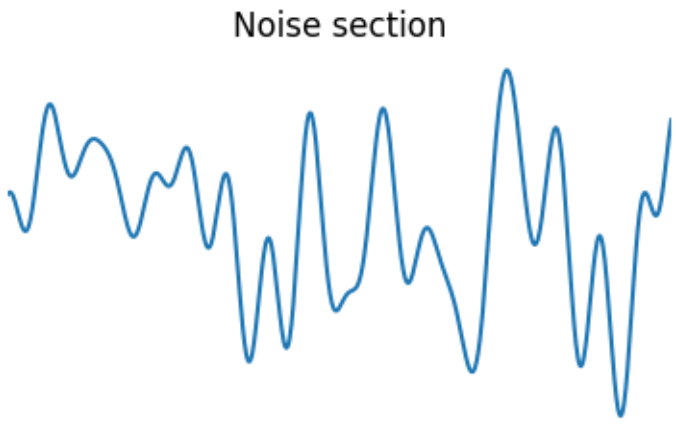
```
host = "https://nmr.cemhti.cnrs-orleans.fr/Dmfit/Help/csdm/"
filename = "31P Phophonate Static.csdf"
experiment = cp.load(host + filename)

# 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]

# plot of the dataset.
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(experiment, color="black", linewidth=0.5, label="Experiment")
ax.set_xlim(200, -200)
plt.grid()
plt.tight_layout()
plt.show()
```

[1] D.Massiot, F.Fayon, M.Capron, I.King, S.Le Calvé, B.Alonso, J.O.Durand, B.Bujoli, Z.Gan, G.Hoatson, 'Modelling one and two-dimensional solid-state NMR spectra.', Magn. Reson. Chem. **40** 70-76 (2002) DOI: 10.1002/mrc.984



Estimate noise statistics from the dataset

```
coords = experiment.dimensions[0].coordinates
noise_region = np.where(coords < -110e-6)
noise_data = experiment[noise_region]

plt.figure(figsize=(3.75, 2.5))
ax = plt.subplot(projection="csdm")
ax.plot(noise_data, label="noise")
plt.title("Noise section")
plt.axis("off")
plt.tight_layout()
plt.show()

noise_mean, sigma = experiment[noise_region].mean(), experiment[noise_region].std()
noise_mean, sigma
```




```
(<Quantity -1.9085335>, <Quantity 3.3310158>)
```

Create a fitting model

Spin System

```
P_31 = Site(
    isotope="31P",
    isotropic_chemical_shift=5.0,  # in ppm,
    shielding_symmetric=SymmetricTensor(zeta=-80, eta=0.5),  # zeta in Hz
)

spin_systems = [SpinSystem(sites=[P_31])]
```

Method

```
# Get the spectral dimension parameters from the experiment.
spectral_dims = get_spectral_dimensions(experiment)

static1D = BlochDecaySpectrum(
    channels=["31P"],
    magnetic_flux_density=9.395,  # in T
    rotor_frequency=0,  # in Hz
    rotor_angle=0,  # in rads
    spectral_dimensions=spectral_dims,
    experiment=experiment,  # experimental dataset
)
```

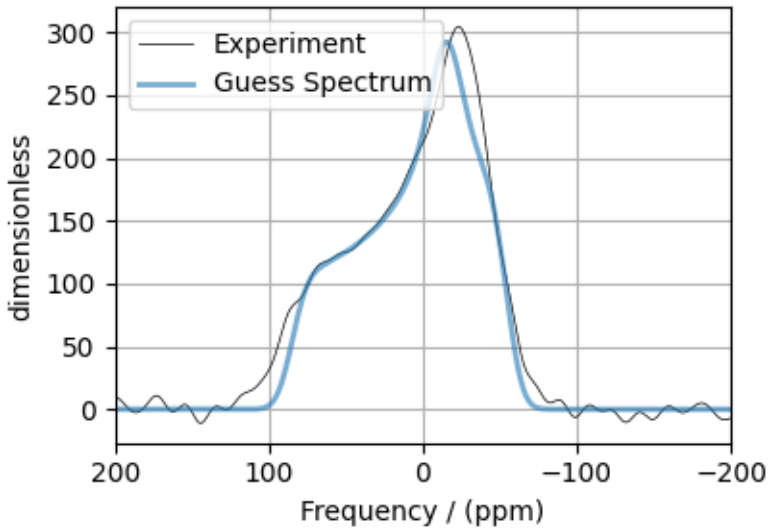
Guess Model Spectrum

```
# Simulation
# ----------
sim = Simulator(spin_systems=spin_systems, methods=[static1D])
sim.run()

# Post Simulation Processing
# --------------------------
processor = sp.SignalProcessor(
    operations=[
        sp.IFFT(),
        sp.apodization.Gaussian(FWHM="3000 Hz"),
        sp.FFT(),
        sp.Scale(factor=40000),
    ]
)
processed_dataset = processor.apply_operations(dataset=sim.methods[0].simulation).real

# Plot of the guess Spectrum
# --------------------------
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(experiment, color="black", linewidth=0.5, label="Experiment")
ax.plot(processed_dataset, linewidth=2, alpha=0.6, label="Guess Spectrum")
ax.set_xlim(200, -200)
plt.grid()
```

(continues on next page)

(continued from previous page)

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



Least-squares minimization with LMFIT

Use the *make_LMFIT_params()* (page 497) for a quick setup of the fitting parameters.

```
params = sf.make_LMFIT_params(sim, processor)
params.pop("sys_0_abundance")
print(params.pretty_print(columns=["value", "min", "max", "vary", "expr"]))
```

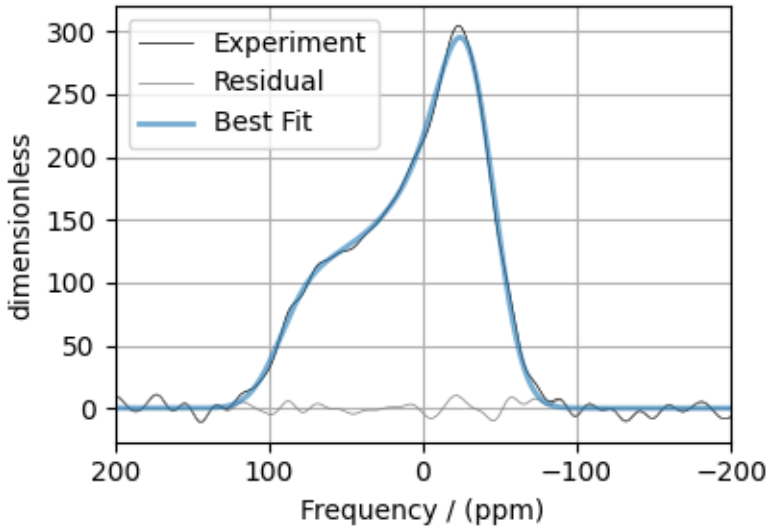
```
Name                                    Value      Min      Max     Vary     Expr
SP_0_operation_1_Gaussian_FWHM           3000     -inf      inf     True     None
SP_0_operation_3_Scale_factor           4e+04     -inf      inf     True     None
sys_0_site_0_isotropic_chemical_shift       5     -inf      inf     True     None
sys_0_site_0_shielding_symmetric_eta      0.5        0        1     True     None
sys_0_site_0_shielding_symmetric_zeta      -80     -inf      inf     True     None
None
```

Solve the minimizer using LMFIT

```
opt = sim.optimize()  # Pre-compute transition pathways
minner = Minimizer(
    sf.LMFIT_min_function,
    params,
    fcn_args=(sim, processor, sigma),
    fcn_kws={"opt": opt},
)
result = minner.minimize()
result
```

The best fit solution

```
best_fit = sf.bestfit(sim, processor)[0].real
residuals = sf.residuals(sim, processor)[0].real

# Plot the spectrum
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(experiment, color="black", linewidth=0.5, label="Experiment")
ax.plot(residuals, color="gray", linewidth=0.5, label="Residual")
ax.plot(best_fit, linewidth=2, alpha=0.6, label="Best Fit")
ax.set_xlim(200, -200)
plt.grid()
plt.legend()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 2.006 seconds)

^{13}C MAS NMR of Glycine (CSA) [960 Hz]

The following is a sideband least-squares fitting example of a ^{13}C MAS NMR spectrum of Glycine spinning at 960 Hz. The following experimental dataset is a part of DMFIT[1] examples. We thank Dr. Dominique Massiot for sharing the dataset.

```
import csdmpy as cp
import numpy as np
import matplotlib.pyplot as plt
from lmfit import Minimizer

from mrsimulator import Simulator, SpinSystem, Site
```

(continues on next page)

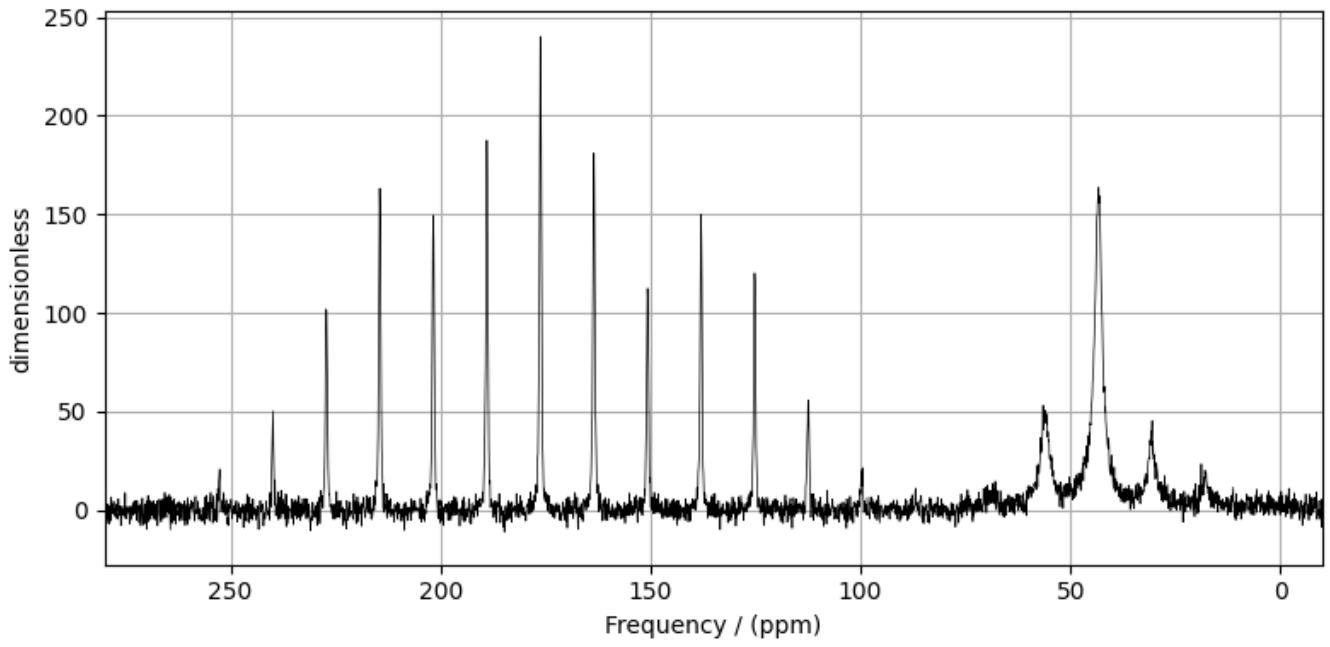
[1] D.Massiot, F.Fayon, M.Capron, I.King, S.Le Calvé, B.Alonso, J.O.Durand, B.Bujoli, Z.Gan, G.Hoatson, 'Modelling one and two-dimensional solid-state NMR spectra.', Magn. Reson. Chem. **40** 70-76 (2002) DOI: 10.1002/mrc.984

(continued from previous page)

```
from mrsimulator.method.lib import BlochDecaySpectrum
from mrsimulator import signal_processor as sp
from mrsimulator.utils import spectral_fitting as sf
from mrsimulator.utils import get_spectral_dimensions
```

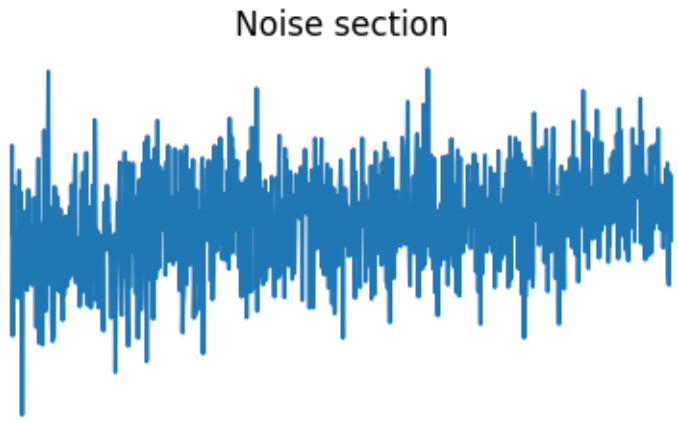
Import the dataset

```
host = "https://nmr.cemhti.cnrs-orleans.fr/Dmfit/Help/csdm/"
filename = "13C MAS 960Hz - Glycine.csdf"
experiment = cp.load(host + 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]

# plot of the dataset.
plt.figure(figsize=(8, 4))
ax = plt.subplot(projection="csdm")
ax.plot(experiment, color="black", linewidth=0.5, label="Experiment")
ax.set_xlim(280, -10)
plt.grid()
plt.tight_layout()
plt.show()
```



Estimate noise statistics from the dataset

```
coords = experiment.dimensions[0].coordinates
noise_region = np.where(coords < 10e-6)
noise_data = experiment[noise_region]

plt.figure(figsize=(3.75, 2.5))
ax = plt.subplot(projection="csdm")
ax.plot(noise_data, label="noise")
plt.title("Noise section")
plt.axis("off")
plt.tight_layout()
plt.show()

noise_mean, sigma = experiment[noise_region].mean(), experiment[noise_region].std()
noise_mean, sigma
```




```
(<Quantity 0.6193483>, <Quantity 3.939941>)
```

Create a fitting model

Spin System

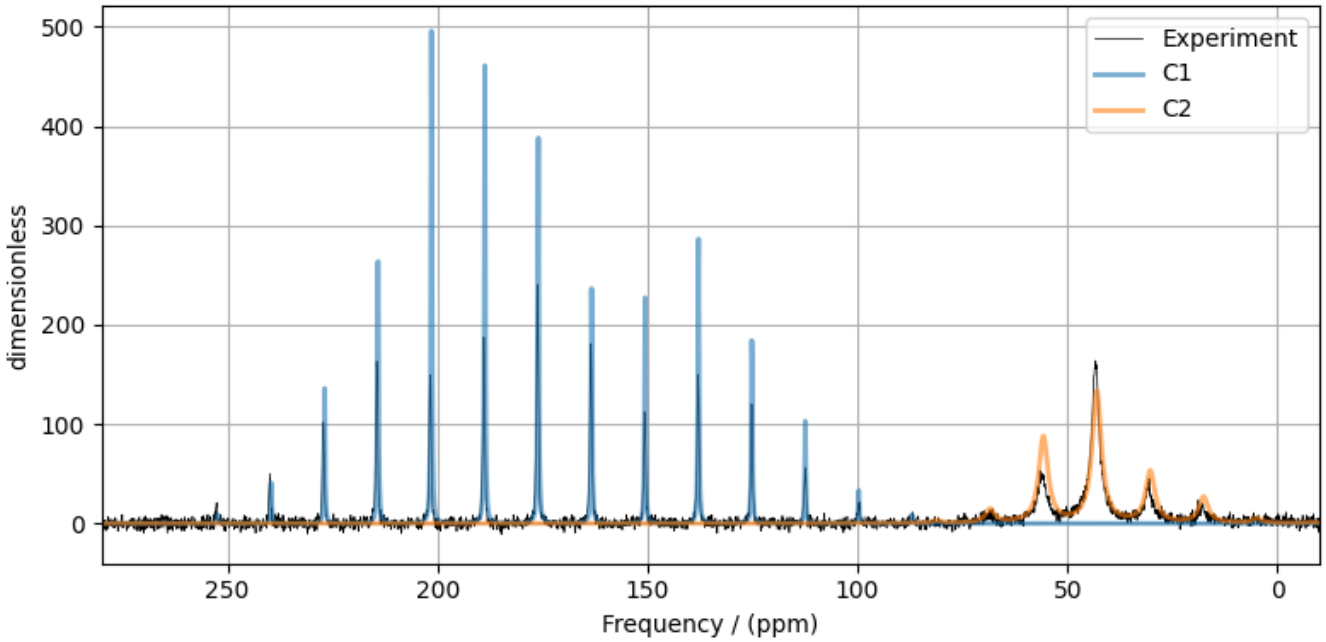
```
C1 = Site(
    isotope="13C",
    isotropic_chemical_shift=176.0,  # in ppm
    shielding_symmetric={"zeta": 70, "eta": 0.6},  # zeta in Hz
)
C2 = Site(
    isotope="13C",
    isotropic_chemical_shift=43.0,  # in ppm
    shielding_symmetric={"zeta": 30, "eta": 0.5},  # zeta in Hz
)

spin_systems = [SpinSystem(sites=[C1], name="C1"), SpinSystem(sites=[C2], name="C2")]
```

Method

```
# Get the spectral dimension parameters from the experiment.
spectral_dims = get_spectral_dimensions(experiment)

MAS = BlochDecaySpectrum(
    channels=["13C"],
    magnetic_flux_density=7.05,  # in T
    rotor_frequency=960,  # in Hz
    spectral_dimensions=spectral_dims,
    experiment=experiment,  # experimental dataset
)
```

Guess Model Spectrum

```
# Simulation
# ----------
sim = Simulator(spin_systems=spin_systems, methods=[MAS])
sim.config.decompose_spectrum = "spin_system"
sim.run()

# Post Simulation Processing
# --------------------------
processor = sp.SignalProcessor(
    operations=[
        sp.IFFT(),
        sp.apodization.Exponential(FWHM="20 Hz", dv_index=0),  # spin system 0
        sp.apodization.Exponential(FWHM="200 Hz", dv_index=1),  # spin system 1
        sp.FFT(),
        sp.Scale(factor=1000),
    ]
)
processed_dataset = processor.apply_operations(dataset=sim.methods[0].simulation).real

# Plot of the guess Spectrum
# --------------------------
plt.figure(figsize=(8, 4))
ax = plt.subplot(projection="csdm")
ax.plot(experiment, color="black", linewidth=0.5, label="Experiment")
ax.plot(processed_dataset, linewidth=2, alpha=0.6)
ax.set_xlim(280, -10)
plt.grid()
plt.legend()
plt.tight_layout()
plt.show()
```



Least-squares minimization with LMFIT

Use the *make_LMFIT_params()* (page 497) for a quick setup of the fitting parameters.

```
params = sf.make_LMFIT_params(sim, processor, include={"rotor_frequency"})
print(params.pretty_print(columns=["value", "min", "max", "vary", "expr"]))
```

```
Name                                    Value      Min      Max     Vary     Expr
SP_0_operation_1_Exponential_FWHM          20     -inf      inf     True     None
SP_0_operation_2_Exponential_FWHM         200     -inf      inf     True     None
SP_0_operation_4_Scale_factor            1000     -inf      inf     True     None
mth_0_rotor_frequency                     960      860     1060     True     None
sys_0_abundance                            50        0      100     True     None
sys_0_site_0_isotropic_chemical_shift     176     -inf      inf     True     None
sys_0_site_0_shielding_symmetric_eta      0.6        0        1     True     None
sys_0_site_0_shielding_symmetric_zeta      70     -inf      inf     True     None
sys_1_abundance                            50        0      100    False 100-sys_0_abundance
sys_1_site_0_isotropic_chemical_shift      43     -inf      inf     True     None
sys_1_site_0_shielding_symmetric_eta      0.5        0        1     True     None
sys_1_site_0_shielding_symmetric_zeta      30     -inf      inf     True     None
None
```

Solve the minimizer using LMFIT

```
opt = sim.optimize()
minner = Minimizer(
    sf.LMFIT_min_function,
    params,
    fcn_args=(sim, processor, sigma),
    fcn_kws={"opt": opt},
```

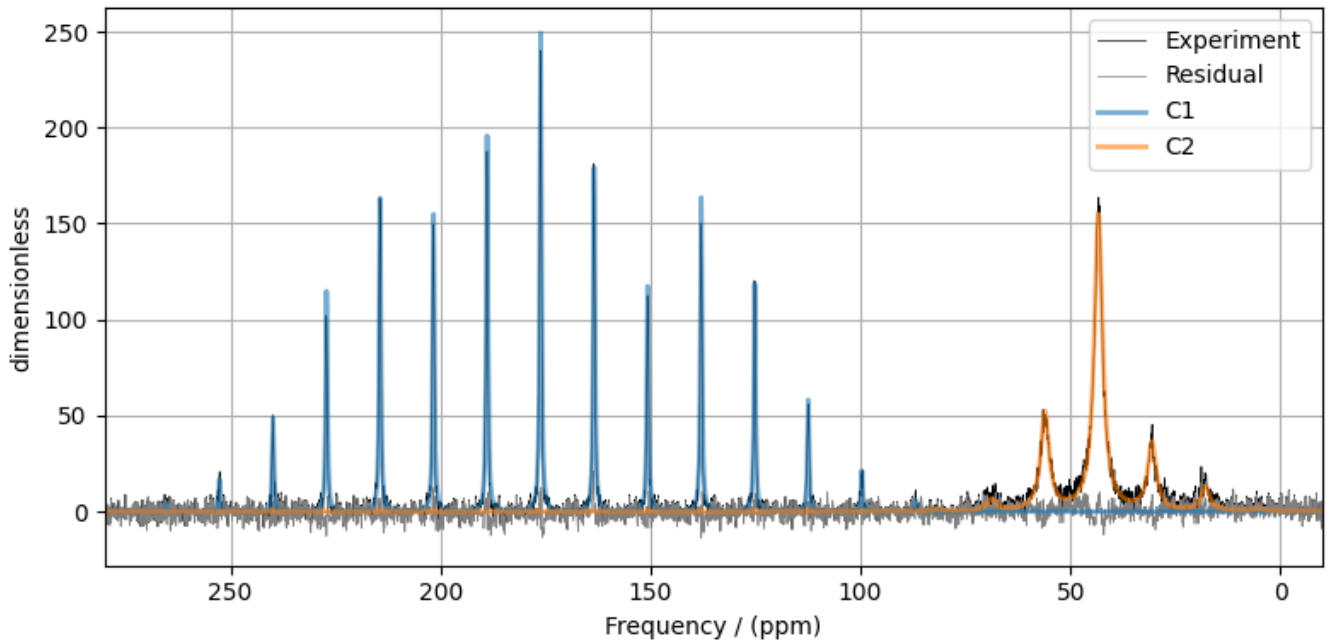
(continues on next page)

(continued from previous page)

```
)
result = minner.minimize()
result
```

The best fit solution

```
best_fit = sf.bestfit(sim, processor)[0].real
residuals = sf.residuals(sim, processor)[0].real

plt.figure(figsize=(8, 4))
ax = plt.subplot(projection="csdm")
ax.plot(experiment, color="black", linewidth=0.5, label="Experiment")
ax.plot(residuals, color="gray", linewidth=0.5, label="Residual")
ax.plot(best_fit, linewidth=2, alpha=0.6)
ax.set_xlim(280, -10)
plt.grid()
plt.legend()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 4.040 seconds)

^{13}C MAS NMR of Glycine (CSA) multi-spectra fit

The following is a multi-dataset least-squares fitting example of ^{13}C MAS NMR spectrum of Glycine spinning at 5 kHz, 1.94 kHz, and 960 Hz. Before trying multi-dataset fitting, we recommend that you first try individual fits. The experimental datasets are part of DMFIT[1] examples.

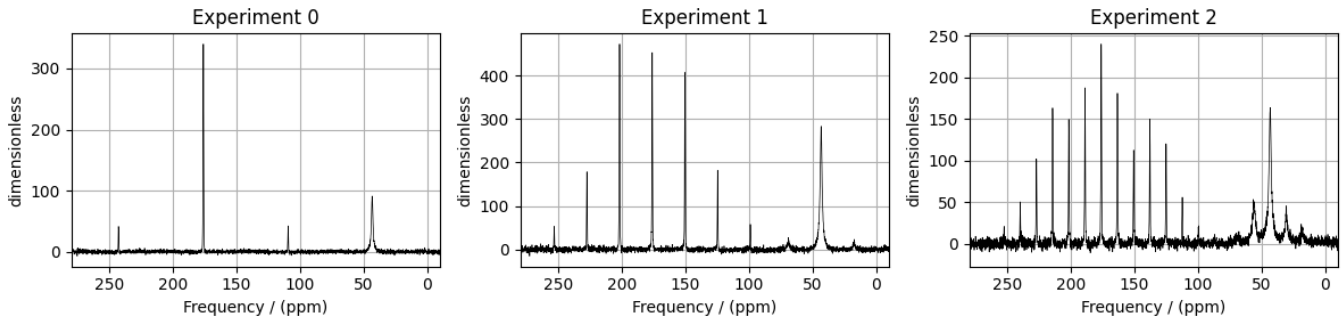
```
import csdmpy as cp
import numpy as np
import matplotlib.pyplot as plt
from lmfit import Minimizer

from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.method.lib import BlochDecaySpectrum
from mrsimulator import signal_processor as sp
from mrsimulator.utils import spectral_fitting as sf
from mrsimulator.utils import get_spectral_dimensions
from mrsimulator.spin_system.tensors import SymmetricTensor
```

Import the datasets

Import the datasets and assign the standard deviation of noise for each dataset. Here, `sigma1`, `sigma2`, and `sigma3` are the noise standard deviation for the dataset acquired at 5 kHz, 1.94 kHz, and 960 Hz spinning speeds, respectively.

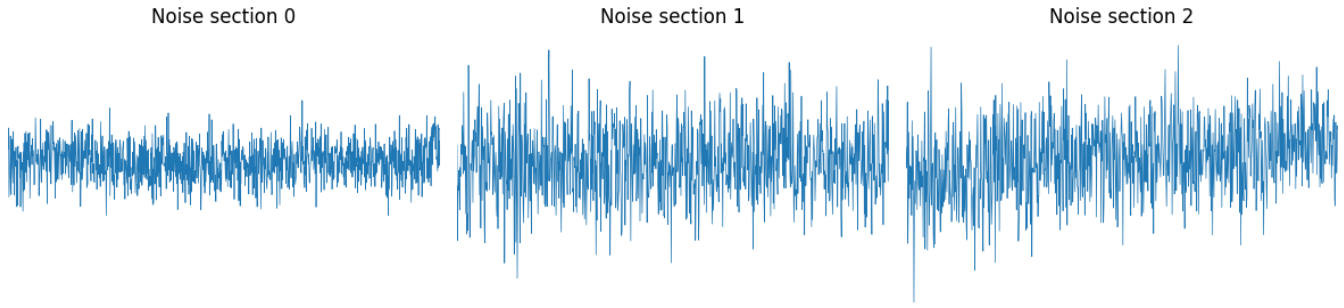
```
host = "https://nmr.cemhti.cnrs-orleans.fr/Dmfit/Help/csdm/"
filename1 = "13C MAS 5000Hz - Glycine.csdf"
filename2 = "13C MAS 1940Hz - Glycine.csdf"
filename3 = "13C MAS 960Hz - Glycine.csdf"

experiment1 = cp.load(host + filename1).real
experiment2 = cp.load(host + filename2).real
experiment3 = cp.load(host + filename3).real
experiments = [experiment1, experiment2, experiment3]

fig, ax = plt.subplots(1, 3, figsize=(12, 3), subplot_kw={"projection": "csdm"})
for i, experiment in enumerate(experiments):
    _ = [item.to("ppm", "nmr_frequency_ratio") for item in experiment.dimensions]

    # plot of the dataset.
    ax[i].plot(experiment, color="black", linewidth=0.5, label="Experiment")
    ax[i].set_title(f"Experiment {i}")
    ax[i].set_xlim(280, -10)
    ax[i].grid()
plt.tight_layout()
plt.show()
```

[1] D.Massiot, F.Fayon, M.Capron, I.King, S.Le Calvé, B.Alonso, J.O.Durand, B.Bujoli, Z.Gan, G.Hoatson, 'Modelling one and two-dimensional solid-state NMR spectra.', Magn. Reson. Chem. **40** 70-76 (2002) DOI: 10.1002/mrc.984



Estimate noise statistics from the dataset

```
noise_data = []
limits = [40e-6, 15e-6, 10e-6]
for measurement, cutoff in zip(experiments, limits):
    coords = measurement.dimensions[0].coordinates
    noise_region = np.where(coords < cutoff)
    noise_data.append(measurement[noise_region])

fig, ax = plt.subplots(
    1, 3, figsize=(12, 3), sharey=True, subplot_kw={"projection": "csdm"}
)
for i, noise in enumerate(noise_data):
    ax[i].plot(noise, linewidth=0.5, label="noise")
    ax[i].set_title(f"Noise section {i}")
    ax[i].axis("off")
plt.tight_layout()
plt.show()

noise_mean = [item.mean() for item in noise_data]
sigma = [item.std() for item in noise_data]
print("mean", noise_mean)
print("standard deviation", sigma)
```




```
mean [<Quantity 0.2857664>, <Quantity 0.2312411>, <Quantity 0.6193483>]
standard deviation [<Quantity 1.950852>, <Quantity 3.784959>, <Quantity 3.939941>]
```

Create a fitting model

Spin System: The objective of a multi-dataset fitting is to optimize the spin system parameters using multiple datasets. In this example, we create two single-site spin systems, which are then shared by three method objects.

```
C1 = Site(
    isotope="13C",
    isotropic_chemical_shift=176.0,  # in ppm
    shielding_symmetric=SymmetricTensor(zeta=60, eta=0.6),  # zeta in Hz
)
C2 = Site(
    isotope="13C",
    isotropic_chemical_shift=43.0,  # in ppm
    shielding_symmetric=SymmetricTensor(zeta=30, eta=0.5),  # zeta in Hz
)

spin_systems = [SpinSystem(sites=[C1], name="C1"), SpinSystem(sites=[C2], name="C2")]
```

Method: Create the three MAS method objects with respective MAS spinning speeds.

```
# Get the spectral dimension parameters from the respective experiment and setup the
# corresponding method.

# Method for dataset 1
spectral_dims1 = get_spectral_dimensions(experiment1)
MAS1 = BlochDecaySpectrum(
    channels=["13C"],
    magnetic_flux_density=7.05,  # in T
    rotor_frequency=5000,  # in Hz
    spectral_dimensions=spectral_dims1,
    experiment=experiment1,  # add experimental dataset 1
)

# Method for dataset 2
spectral_dims2 = get_spectral_dimensions(experiment2)
MAS2 = BlochDecaySpectrum(
    channels=["13C"],
    magnetic_flux_density=7.05,  # in T
    rotor_frequency=1940,  # in Hz
    spectral_dimensions=spectral_dims2,
    experiment=experiment2,  # add experimental dataset 2
)

# Method for dataset 3
spectral_dims3 = get_spectral_dimensions(experiment3)
MAS3 = BlochDecaySpectrum(
    channels=["13C"],
    magnetic_flux_density=7.05,  # in T
    rotor_frequency=960,  # in Hz
    spectral_dimensions=spectral_dims3,
    experiment=experiment3,  # add experimental dataset 3
)
```

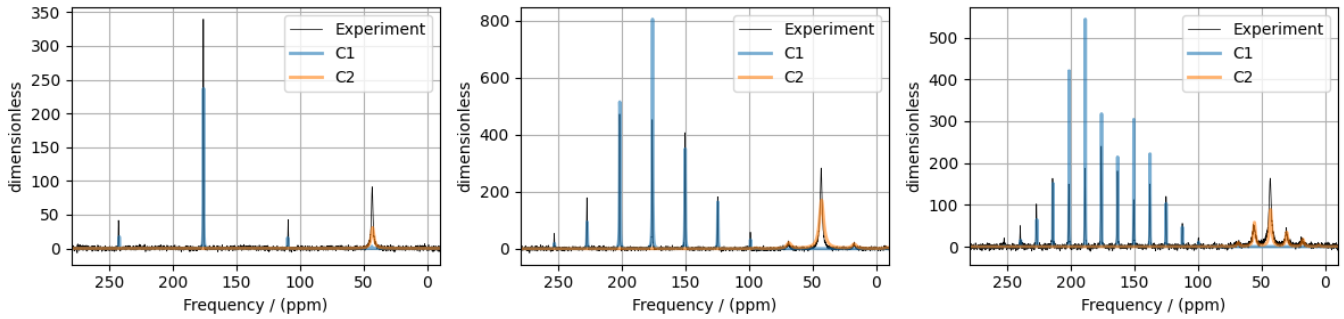
Guess Model Spectrum

```
# Simulation
# ----------
# Add the spin systems and the three methods to the simulator object.
sim = Simulator(spin_systems=spin_systems, methods=[MAS1, MAS2, MAS3])
sim.config.decompose_spectrum = "spin_system"
sim.run()

# Post Simulation Processing
# --------------------------
# Add signal processing to simulation dataset from the three methods.

# Processor for dataset 1
processor1 = sp.SignalProcessor(
    operations=[
        sp.IFFT(),
        sp.apodization.Exponential(FWHM="20 Hz", dv_index=0),  # spin system 0
        sp.apodization.Exponential(FWHM="200 Hz", dv_index=1),  # spin system 1
        sp.FFT(),
        sp.Scale(factor=100),  # dataset is scaled independently using scale factor.
    ]
)

# Processor for dataset 2
processor2 = sp.SignalProcessor(
    operations=[
        sp.IFFT(),
        sp.apodization.Exponential(FWHM="30 Hz", dv_index=0),  # spin system 0
        sp.apodization.Exponential(FWHM="300 Hz", dv_index=1),  # spin system 1
        sp.FFT(),
        sp.Scale(factor=1000),  # dataset is scaled independently using scale factor.
    ]
)

# Processor for dataset 3
processor3 = sp.SignalProcessor(
    operations=[
        sp.IFFT(),
        sp.apodization.Exponential(FWHM="10 Hz", dv_index=0),  # spin system 0
        sp.apodization.Exponential(FWHM="150 Hz", dv_index=1),  # spin system 1
        sp.FFT(),
        sp.Scale(factor=500),  # dataset is scaled independently using scale factor.
    ]
)
processors = [processor1, processor2, processor3]

processed_dataset = []
for i, proc in enumerate(processors):
    processed_dataset.append(
        proc.apply_operations(dataset=sim.methods[i].simulation).real
    )
```

(continues on next page)

(continued from previous page)

```
# Plot of the guess Spectrum
# --------------------------

fig, ax = plt.subplots(1, 3, figsize=(12, 3), subplot_kw={"projection": "csdm"})
for i, exp_dataset in enumerate(experiments):
    ax[i].plot(exp_dataset, color="black", linewidth=0.5, label="Experiment")
    ax[i].plot(processed_dataset[i], linewidth=2, alpha=0.6)
    ax[i].set_xlim(280, -10)
    ax[i].grid()

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



Least-squares minimization with LMFIT

Use the *make_LMFIT_params()* (page 497) for a quick setup of the fitting parameters. Note, the first two arguments of this function is the simulator object and a list of SignalProcessor objects, `processors`. The fitting parameters corresponding to the signal processor objects are generated using `SP_i_operation_j_FunctionName_FunctionArg`, where i is the *ith* signal processor within the list, j is the operation index of the *ith* processor, and *FunctionName* and *FunctionArg* are the operation function name and function argument, respectively.

```
params = sf.make_LMFIT_params(sim, processors, include={"rotor_frequency"})
print(params.pretty_print(columns=["value", "min", "max", "vary", "expr"]))
```

Name	Value	Min	Max	Vary	Expr
SP_0_operation_1_Exponential_FWHM	20	-inf	inf	True	None
SP_0_operation_2_Exponential_FWHM	200	-inf	inf	True	None
SP_0_operation_4_Scale_factor	100	-inf	inf	True	None
SP_1_operation_1_Exponential_FWHM	30	-inf	inf	True	None
SP_1_operation_2_Exponential_FWHM	300	-inf	inf	True	None
SP_1_operation_4_Scale_factor	1000	-inf	inf	True	None
SP_2_operation_1_Exponential_FWHM	10	-inf	inf	True	None
SP_2_operation_2_Exponential_FWHM	150	-inf	inf	True	None
SP_2_operation_4_Scale_factor	500	-inf	inf	True	None
mth_0_rotor_frequency	5000	4900	5100	True	None
mth_1_rotor_frequency	1940	1840	2040	True	None
mth_2_rotor_frequency	960	860	1060	True	None
sys_0_abundance	50	0	100	True	None

(continues on next page)

(continued from previous page)

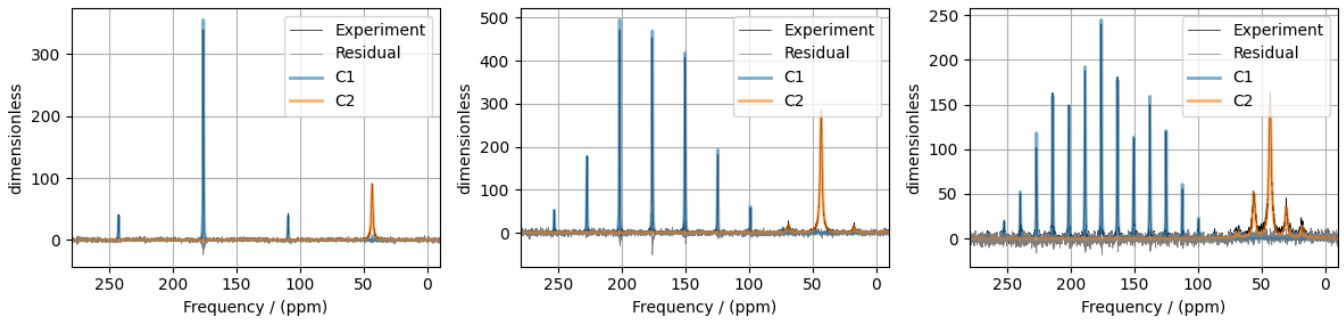
```
sys_0_site_0_isotropic_chemical_shift       176     -inf      inf     True      None
sys_0_site_0_shielding_symmetric_eta        0.6        0        1     True      None
sys_0_site_0_shielding_symmetric_zeta        60     -inf      inf     True      None
sys_1_abundance                              50        0      100    False 100-sys_0_abundance
sys_1_site_0_isotropic_chemical_shift        43     -inf      inf     True      None
sys_1_site_0_shielding_symmetric_eta        0.5        0        1     True      None
sys_1_site_0_shielding_symmetric_zeta        30     -inf      inf     True      None
None
```

Solve the minimizer using LMFIT

```
opt = sim.optimize()  # Pre-compute transition pathways
minner = Minimizer(
    sf.LMFIT_min_function,
    params,
    fcn_args=(sim, processors, sigma),
    fcn_kws={"opt": opt},
)
result = minner.minimize()
result
```

The best fit solution

```
all_best_fit = sf.bestfit(sim, processors)  # a list of best fit simulations
all_residuals = sf.residuals(sim, processors)  # a list of residuals

# Plot the spectrum
fig, ax = plt.subplots(1, 3, figsize=(12, 3), subplot_kw={"projection": "csdm"})
for i, proc in enumerate(processors):
    ax[i].plot(experiments[i], color="black", linewidth=0.5, label="Experiment")
    ax[i].plot(all_residuals[i].real, color="gray", linewidth=0.5, label="Residual")
    ax[i].plot(all_best_fit[i].real, linewidth=2, alpha=0.6)
    ax[i].set_xlim(280, -10)
    ax[i].grid()

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



Total running time of the script: (0 minutes 14.114 seconds)

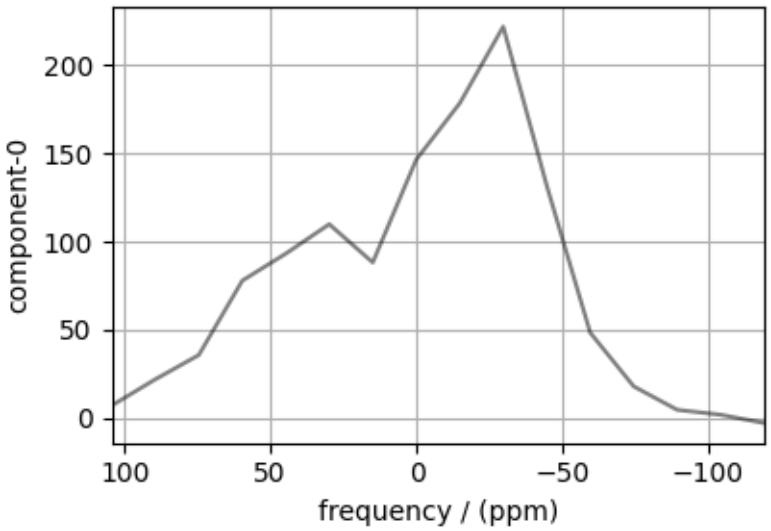
1D PASS/MAT sideband order cross-section

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 csdmpy as cp
import matplotlib.pyplot as plt
from lmfit import Minimizer

from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.method.lib import BlochDecaySpectrum
from mrsimulator import signal_processor as sp
from mrsimulator.utils import spectral_fitting as sf
from mrsimulator.utils import get_spectral_dimensions
from mrsimulator.spin_system.tensors import SymmetricTensor
```

Import the dataset

```
name = "https://ssnmr.org/sites/default/files/mrsimulator/LHistidine_cross_section.csdf"
pass_cross_section = cp.load(name)

# standard deviation of noise from the dataset
sigma = 4.640351

# For the spectral fitting, we only focus on the real part of the complex dataset.
pass_cross_section = pass_cross_section.real

# Convert the coordinates along each dimension from Hz to ppm.
_ = [item.to("ppm", "nmr_frequency_ratio") for item in pass_cross_section.dimensions]


# The plot of the dataset.
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(pass_cross_section, "k", alpha=0.5)
ax.invert_xaxis()
plt.grid()
plt.tight_layout()
plt.show()
```



Create a fitting model

Guess model

Create a guess list of spin systems. For fitting the sideband profile at an isotropic chemical shift cross-section from PASS/MAT datasets, set the isotropic_chemical_shift parameter of the site object as zero.

```
site = Site(
    isotope="13C",
    isotropic_chemical_shift=0,  #
    shielding_symmetric=SymmetricTensor(zeta=-70, eta=0.8),
)
spin_systems = [SpinSystem(sites=[site])]
```

Method

For the sideband-only cross-section, use the BlochDecaySpectrum method.

```
# Get the dimension information from the experiment.
spectral_dims = get_spectral_dimensions(pass_cross_section)

PASS = BlochDecaySpectrum(
    channels=["13C"],
    magnetic_flux_density=9.395,  # in T
    rotor_frequency=1500,  # in Hz
    spectral_dimensions=spectral_dims,
    experiment=pass_cross_section,  # also add the measurement to the method.
)
```

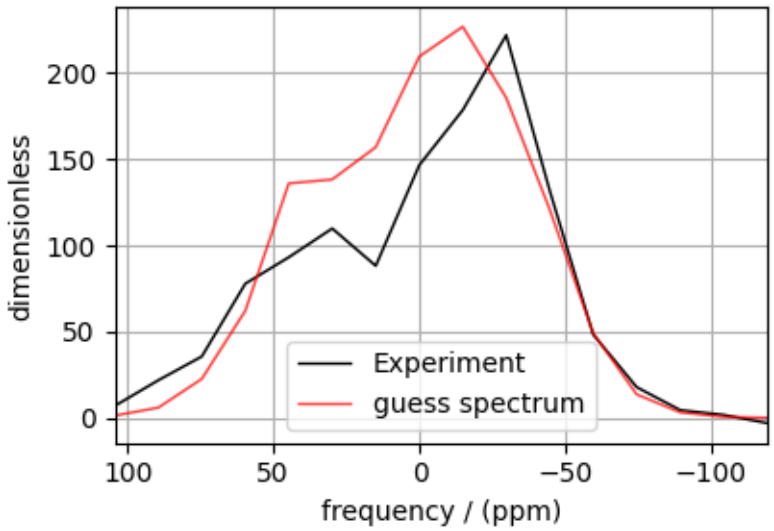
Guess Spectrum

```
# Simulation
# ----------
sim = Simulator(spin_systems=spin_systems, methods=[PASS])
```

(continues on next page)

(continued from previous page)

```
sim.run()

# Post Simulation Processing
# --------------------------
processor = sp.SignalProcessor(operations=[sp.Scale(factor=20000)])
processed_dataset = processor.apply_operations(dataset=sim.methods[0].simulation).real

# Plot of the guess Spectrum
# --------------------------
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(pass_cross_section, color="k", linewidth=1, label="Experiment")
ax.plot(processed_dataset, color="r", alpha=0.75, linewidth=1, label="guess spectrum")
plt.grid()
ax.invert_xaxis()
plt.legend()
plt.tight_layout()
plt.show()
```



Least-squares minimization with LMFIT

First, create the fitting parameters. Use the *make_LMFIT_params()* (page 497) for a quick setup.

```
params = sf.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
print(params.pretty_print(columns=["value", "min", "max", "vary", "expr"]))
```

```
Name                                Value      Min      Max     Vary     Expr
SP_0_operation_0_Scale_factor       2e+04     -inf      inf     True     None
```

(continues on next page)

(continued from previous page)

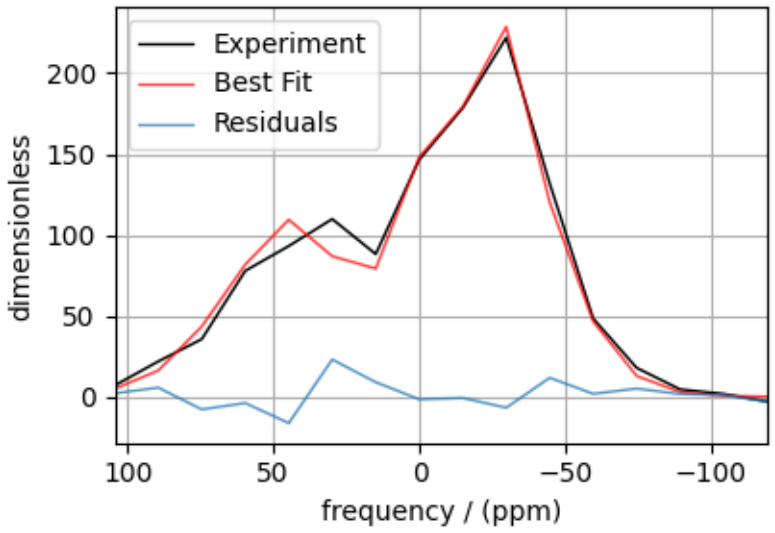
```
sys_0_abundance                          100        0      100    False      100
sys_0_site_0_isotropic_chemical_shift      0     -inf      inf    False     None
sys_0_site_0_shielding_symmetric_eta     0.8        0        1     True     None
sys_0_site_0_shielding_symmetric_zeta    -70     -inf      inf     True     None
None
```

Run the minimization using LMFIT

```
opt = sim.optimize()
minner = Minimizer(
    sf.LMFIT_min_function,
    params,
    fcn_args=(sim, processor, sigma),
    fcn_kws={"opt": opt},
)
result = minner.minimize()
result
```

The best fit solution

```
best_fit = sf.bestfit(sim, processor)[0].real
residuals = sf.residuals(sim, processor)[0].real

# Plot the spectrum
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(pass_cross_section, color="k", linewidth=1, label="Experiment")
ax.plot(best_fit, "r", alpha=0.75, linewidth=1, label="Best Fit")
ax.plot(residuals, alpha=0.75, linewidth=1, label="Residuals")
ax.invert_xaxis()
plt.grid()
plt.legend()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 1.466 seconds)

^{17}O MAS NMR of crystalline Na_2SiO_3 (2nd order quad)

In this example, we illustrate the use of the mrsimulator objects to

- create a quadrupolar fitting model using Simulator and SignalProcessor objects,
- use the fitting model to perform a least-squares analysis, and
- extract the fitting parameters from the model.

We use the LMFIT library to fit the spectrum. The following example shows the least-squares fitting procedure applied to the ^{17}O MAS NMR spectrum of Na_2SiO_3[1].

Start by importing the relevant modules.

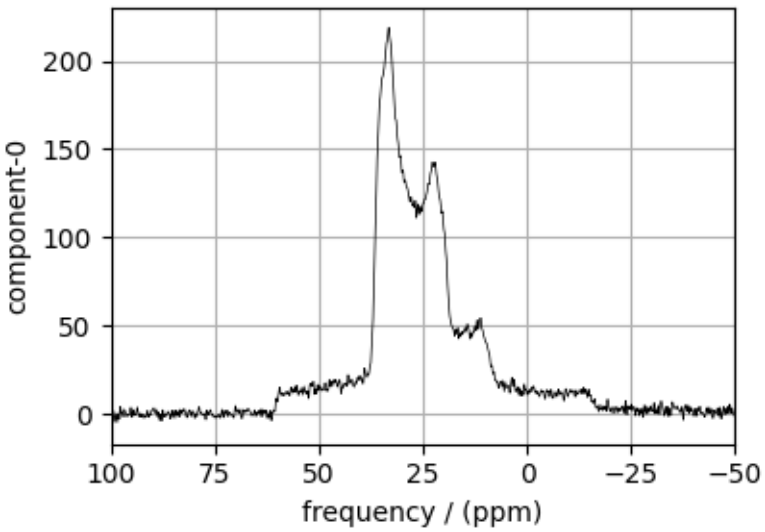
```
import csdmpy as cp
import numpy as np
import matplotlib.pyplot as plt
from lmfit import Minimizer

from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.method.lib import BlochDecayCTSpectrum
from mrsimulator import signal_processor as sp
from mrsimulator.utils import spectral_fitting as sf
from mrsimulator.utils import get_spectral_dimensions
from mrsimulator.spin_system.tensors import SymmetricTensor
```

[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

Import the dataset

Import the experimental dataset. We use dataset file serialized with the CSDM file-format, using the csdmpy module.

```
filename = "https://ssnmr.org/sites/default/files/mrsimulator/Na2SiO3_O17.csdf"
experiment = cp.load(filename)

# For spectral fitting, we only focus on the real part of the complex dataset
experiment = experiment.real

# Convert the dimension coordinates from Hz to ppm.
experiment.x[0].to("ppm", "nmr_frequency_ratio")

# plot of the dataset.
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(experiment, color="black", linewidth=0.5, label="Experiment")
ax.set_xlim(100, -50)
plt.grid()
plt.tight_layout()
plt.show()
```



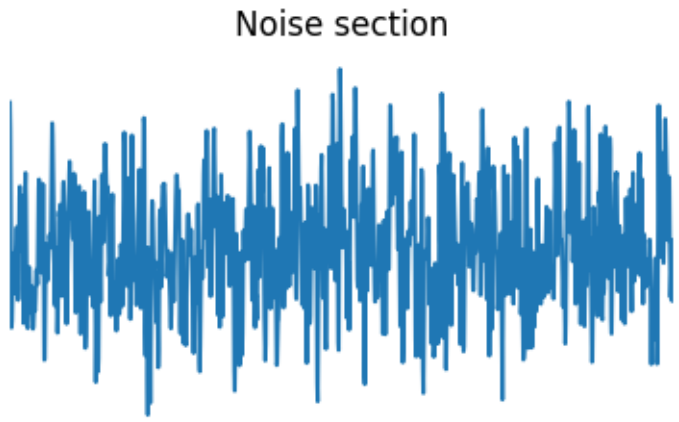
Estimate noise statistics from the dataset

```
coords = experiment.dimensions[0].coordinates
noise_region = np.where(coords > 70e-6)
noise_data = experiment[noise_region]

plt.figure(figsize=(3.75, 2.5))
ax = plt.subplot(projection="csdm")
ax.plot(noise_data, label="noise")
plt.title("Noise section")
plt.axis("off")
plt.tight_layout()
plt.show()
```

(continues on next page)

(continued from previous page)

```
noise_mean, sigma = experiment[noise_region].mean(), experiment[noise_region].std()
noise_mean, sigma
```




```
(<Quantity 0.13096036>, <Quantity 1.710335>)
```

Create a fitting model

A fitting model is a composite of `Simulator` and `SignalProcessor` objects.

Step 1: Create initial guess sites and spin systems

```
O1 = Site(
    isotope="17O",
    isotropic_chemical_shift=60.0,  # in ppm,
    quadrupolar=SymmetricTensor(Cq=4.2e6, eta=0.5),  # Cq in Hz
)

O2 = Site(
    isotope="17O",
    isotropic_chemical_shift=40.0,  # in ppm,
    quadrupolar=SymmetricTensor(Cq=2.4e6, eta=0.0),  # Cq in Hz
)

spin_systems = [
    SpinSystem(sites=[O1], abundance=50, name="O1"),
    SpinSystem(sites=[O2], abundance=50, name="O2"),
]
```

Step 2: Create the method object. Create an appropriate method object that closely resembles the technique used in acquiring the experimental dataset. The attribute values of this method must meet the experimental conditions, 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 the spectral/spectroscopic dimension information, i.e., count, spectral_width, and the reference_offset, is extracted from

the CSDM dimension metadata using the *get_spectral_dimensions()* (page 487) utility function. 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(experiment)

MAS_CT = BlochDecayCTSpectrum(
    channels=["17O"],
    magnetic_flux_density=9.395,  # in T
    rotor_frequency=14000,  # in Hz
    spectral_dimensions=spectral_dims,
    experiment=experiment,  # experimental dataset
)
```

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

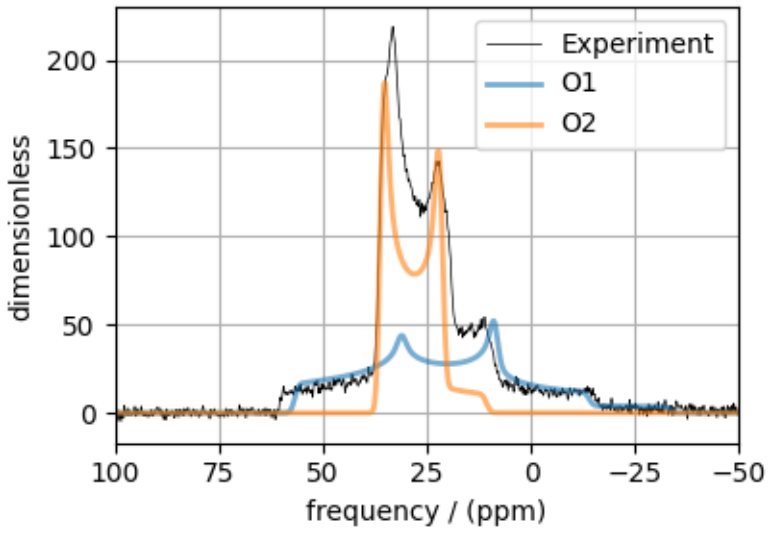
```
sim = Simulator(spin_systems=spin_systems, methods=[MAS_CT])
sim.config.decompose_spectrum = "spin_system"
sim.run()
```

Step 4: Create a SignalProcessor class object and apply the post-simulation signal processor operations.

```
processor = sp.SignalProcessor(
    operations=[
        sp.IFFT(),
        sp.apodization.Gaussian(FWHM="100 Hz"),
        sp.FFT(),
        sp.Scale(factor=2000.0),
    ]
)
processed_dataset = processor.apply_operations(dataset=sim.methods[0].simulation).real
```

Step 5: The plot of the dataset and the guess spectrum.

```
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(experiment, color="black", linewidth=0.5, label="Experiment")
ax.plot(processed_dataset, linewidth=2, alpha=0.6)
ax.set_xlim(100, -50)
plt.legend()
plt.grid()
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 497), that considerably simplifies the LMFIT parameters generation process.

Step 6: Create a list of parameters.

```
params = sf.make_LMFIT_params(sim, processor)
```

The *make_ LMFIT_ params* parses the instances of the `Simulator` and the `PostSimulator` objects for parameters and returns a 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, and constrain *eta=0* for spin system at index 1.

```
params.pop("sys_0_abundance")
params.pop("sys_1_abundance")
params["sys_1_site_0_quadrupolar_eta"].vary = False
print(params.pretty_print(columns=["value", "min", "max", "vary", "expr"]))
```

```
Name                                    Value      Min      Max     Vary     Expr
SP_0_operation_1_Gaussian_FWHM            100     -inf      inf     True     None
SP_0_operation_3_Scale_factor            2000     -inf      inf     True     None
sys_0_site_0_isotropic_chemical_shift      60     -inf      inf     True     None
sys_0_site_0_quadrupolar_Cq           4.2e+06     -inf      inf     True     None
sys_0_site_0_quadrupolar_eta              0.5        0        1     True     None
sys_1_site_0_isotropic_chemical_shift      40     -inf      inf     True     None
sys_1_site_0_quadrupolar_Cq           2.4e+06     -inf      inf     True     None
sys_1_site_0_quadrupolar_eta                0        0        1    False     None
None
```

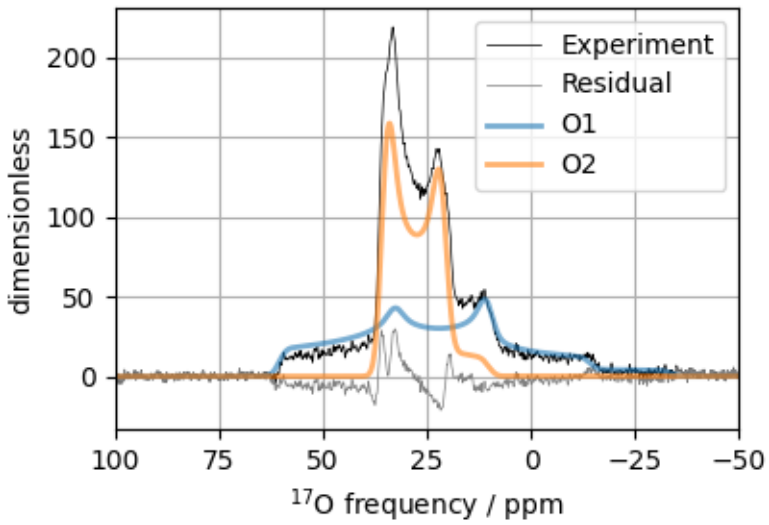
Step 7: Perform least-squares minimization. For the user's convenience, we also provide a utility function, *LMFIT_min_function()* (page 497), for evaluating the difference vector between the simulation and experiment,

based on the parameters update. You may use this function directly to instantiate the LMFIT Minimizer class where *fcn_args* and *fcn_kws* are arguments passed to the function, as follows,

```
opt = sim.optimize()
minner = Minimizer(
    sf.LMFIT_min_function,
    params,
    fcn_args=(sim, processor, sigma),
    fcn_kws={"opt": opt},
)
result = minner.minimize()
result
```

Step 8: The plot of the fit and the measurement dataset.

```
# Best fit spectrum
best_fit = sf.bestfit(sim, processor)[0].real
residuals = sf.residuals(sim, processor)[0].real

plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(experiment, color="black", linewidth=0.5, label="Experiment")
ax.plot(residuals, color="gray", linewidth=0.5, label="Residual")
ax.plot(best_fit, linewidth=2, alpha=0.6)
ax.set_xlabel("$^{17}$O frequency / ppm")
ax.set_xlim(100, -50)
plt.legend()
plt.grid()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 8.046 seconds)

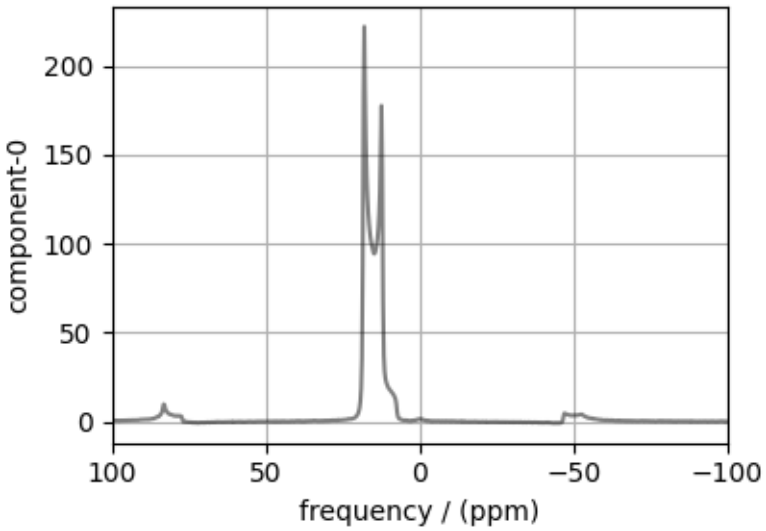
^{11}B MAS NMR of Lithium orthoborate crystal

The following is a quadrupolar lineshape fitting example for the 11B MAS NMR of lithium orthoborate crystal. The dataset was shared by Dr. Nathan Barrow.

```
import csdmpy as cp
import numpy as np
import matplotlib.pyplot as plt
from lmfit import Minimizer

from mrsimulator import Simulator, Site, SpinSystem
from mrsimulator.method.lib import BlochDecayCTSpectrum
from mrsimulator import signal_processor as sp
from mrsimulator.utils import spectral_fitting as sf
from mrsimulator.utils import get_spectral_dimensions
from mrsimulator.spin_system.tensors import SymmetricTensor
```

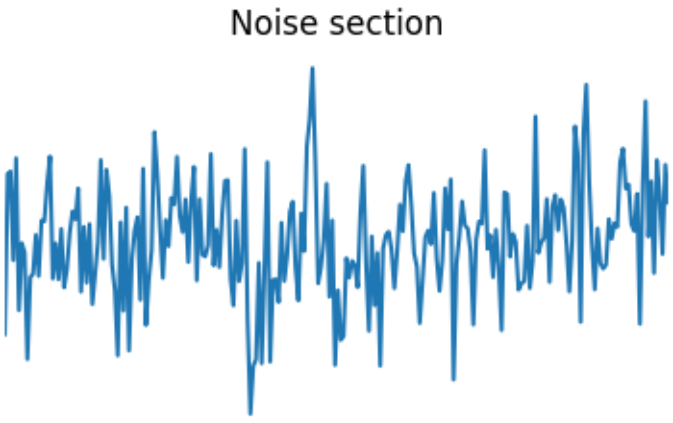
Import the dataset

```
host = "https://ssnmr.org/sites/default/files/mrsimulator/"
filename = "11B_lithum_orthoborate.csdf"
experiment = cp.load(host + 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]

# plot of the dataset.
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(experiment, "k", alpha=0.5)
ax.set_xlim(100, -100)
plt.grid()
plt.tight_layout()
plt.show()
```



Estimate noise statistics from the dataset

```
coords = experiment.dimensions[0].coordinates
noise_region = np.where(np.logical_and(coords < -140e-6, coords > -200e-6))
noise_data = experiment[noise_region]

plt.figure(figsize=(3.75, 2.5))
ax = plt.subplot(projection="csdm")
ax.plot(noise_data, label="noise")
plt.title("Noise section")
plt.axis("off")
plt.tight_layout()
plt.show()

noise_mean, sigma = experiment[noise_region].mean(), experiment[noise_region].std()
noise_mean, sigma
```




```
(<Quantity 0.12421671>, <Quantity 0.07768127>)
```

Create a fitting model

Spin System

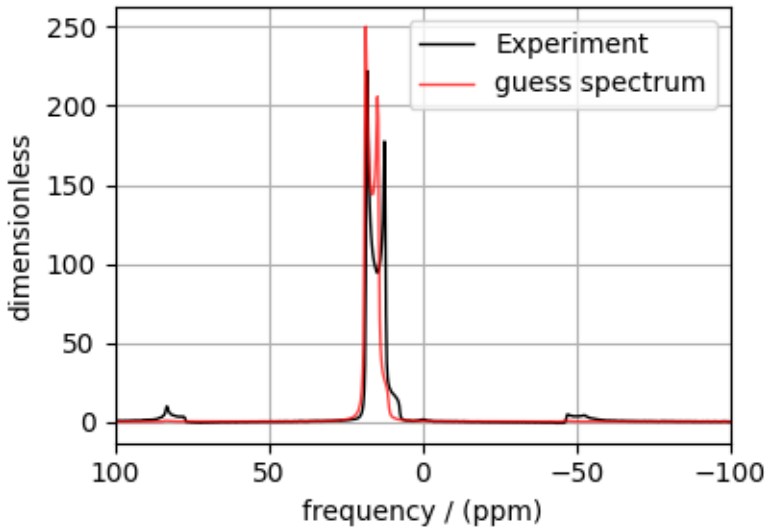
```
B11 = Site(
    isotope="11B",
    isotropic_chemical_shift=20.0,  # in ppm
    quadrupolar=SymmetricTensor(Cq=2.3e6, eta=0.03),  # Cq in Hz
)
spin_systems = [SpinSystem(sites=[B11])]
```

Method

```
# Get the spectral dimension parameters from the experiment.
spectral_dims = get_spectral_dimensions(experiment)

MAS_CT = BlochDecayCTSpectrum(
    channels=["11B"],
    magnetic_flux_density=14.1,  # in T
    rotor_frequency=12500,  # in Hz
    spectral_dimensions=spectral_dims,
    experiment=experiment,  # add the measurement to the method.
)
```

Guess Model Spectrum

```
# Simulation
# ----------
sim = Simulator(spin_systems=spin_systems, methods=[MAS_CT])
sim.run()

# Post Simulation Processing
# --------------------------
processor = sp.SignalProcessor(
    operations=[
        sp.IFFT(),
        sp.apodization.Exponential(FWHM="100 Hz"),
        sp.FFT(),
        sp.Scale(factor=2000),
    ]
)
processed_dataset = processor.apply_operations(dataset=sim.methods[0].simulation).real

# Plot of the guess Spectrum
# --------------------------
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(experiment, "k", linewidth=1, label="Experiment")
ax.plot(processed_dataset, "r", alpha=0.75, linewidth=1, label="guess spectrum")
ax.set_xlim(100, -100)
plt.grid()
plt.legend()
plt.tight_layout()
plt.show()
```



Least-squares minimization with LMFIT

Use the *make_LMFIT_params()* (page 497) for a quick setup of the fitting parameters.

```
params = sf.make_LMFIT_params(sim, processor)
params.pop("sys_0_abundance")
print(params.pretty_print(columns=["value", "min", "max", "vary", "expr"]))
```

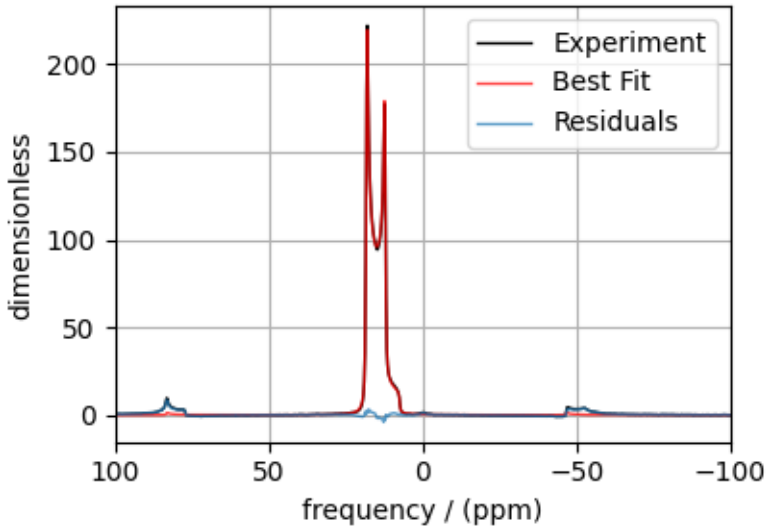
```
Name                                    Value      Min      Max     Vary     Expr
SP_0_operation_1_Exponential_FWHM         100     -inf      inf     True     None
SP_0_operation_3_Scale_factor            2000     -inf      inf     True     None
sys_0_site_0_isotropic_chemical_shift      20     -inf      inf     True     None
sys_0_site_0_quadrupolar_Cq           2.3e+06     -inf      inf     True     None
sys_0_site_0_quadrupolar_eta             0.03        0        1     True     None
None
```

Solve the minimizer using LMFIT

```
opt = sim.optimize()  # Pre-compute transition pathways
minner = Minimizer(
    sf.LMFIT_min_function,
    params,
    fcn_args=(sim, processor, sigma),
    fcn_kws={"opt": opt},
)
result = minner.minimize()
result
```

The best fit solution

```
best_fit = sf.bestfit(sim, processor)[0].real
residuals = sf.residuals(sim, processor)[0].real

# Plot the spectrum
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(experiment, "k", linewidth=1, label="Experiment")
ax.plot(best_fit, "r", alpha=0.75, linewidth=1, label="Best Fit")
ax.plot(residuals, alpha=0.75, linewidth=1, label="Residuals")
ax.set_xlim(100, -100)
plt.grid()
plt.legend()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 2.540 seconds)

^{27}Al MAS NMR of YAG (1st and 2nd order Quad)

The following is a quadrupolar lineshape fitting example for the 27Al MAS NMR of Yttrium aluminum garnet (YAG) crystal. The following experimental dataset is a part of DMFIT[1] examples. We thank Dr. Dominique Massiot for sharing the dataset.

```
import csdmpy as cp
import numpy as np
import matplotlib.pyplot as plt
from lmfit import Minimizer

from mrsimulator import Simulator, Site, SpinSystem
```

(continues on next page)

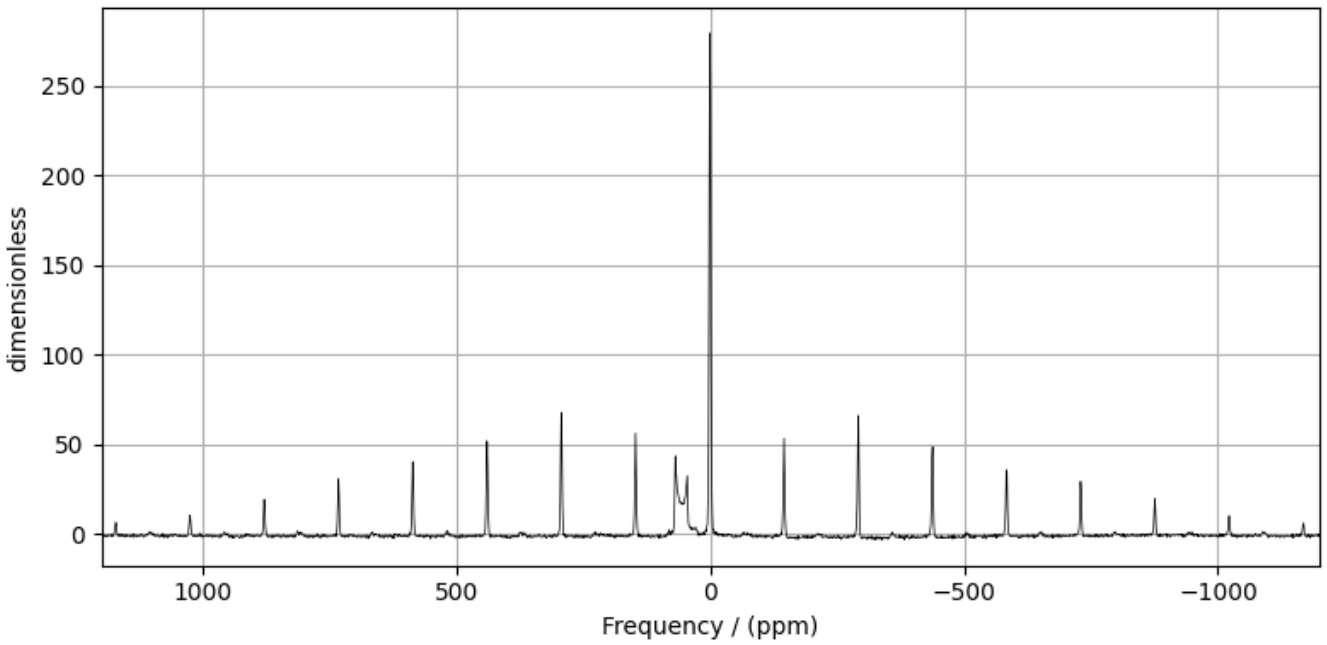
[1] D.Massiot, F.Fayon, M.Capron, I.King, S.Le Calvé, B.Alonso, J.O.Durand, B.Bujoli, Z.Gan, G.Hoatson, 'Modelling one and two-dimensional solid-state NMR spectra.', Magn. Reson. Chem. **40** 70-76 (2002) DOI: 10.1002/mrc.984

(continued from previous page)

```
from mrsimulator.method.lib import BlochDecaySpectrum
from mrsimulator import signal_processor as sp
from mrsimulator.utils import spectral_fitting as sf
from mrsimulator.utils import get_spectral_dimensions
from mrsimulator.spin_system.tensors import SymmetricTensor
```

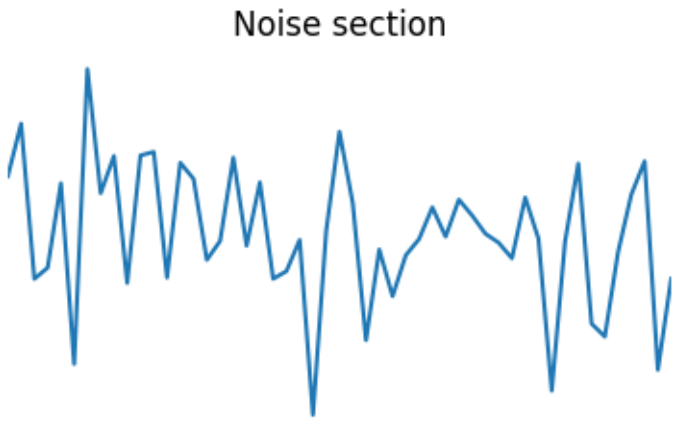
Import the dataset

```
host = "https://nmr.cemhti.cnrs-orleans.fr/Dmfit/Help/csdm/"
filename = "27Al Quad MAS YAG 400MHz.csdf"
experiment = cp.load(host + 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]

# plot of the dataset.
plt.figure(figsize=(8, 4))
ax = plt.subplot(projection="csdm")
ax.plot(experiment, color="black", linewidth=0.5, label="Experiment")
ax.set_xlim(1200, -1200)
plt.grid()
plt.tight_layout()
plt.show()
```



Estimate noise statistics from the dataset

```
coords = experiment.dimensions[0].coordinates
noise_region = np.where(np.logical_and(coords > -570e-6, coords < -510e-6))
noise_data = experiment[noise_region]

plt.figure(figsize=(3.75, 2.5))
ax = plt.subplot(projection="csdm")
ax.plot(noise_data, label="noise")
plt.title("Noise section")
plt.axis("off")
plt.tight_layout()
plt.show()

noise_mean, sigma = experiment[noise_region].mean(), experiment[noise_region].std()
noise_mean, sigma
```




```
(<Quantity -1.3760117>, <Quantity 0.5487051>)
```

Create a fitting model

Guess model

Create a guess list of spin systems.

```
Al_1 = Site(
    isotope="27Al",
    isotropic_chemical_shift=76,  # in ppm
    quadrupolar=SymmetricTensor(Cq=6e6, eta=0.0),  # Cq in Hz
)

Al_2 = Site(
    isotope="27Al",
    isotropic_chemical_shift=1,  # in ppm
    quadrupolar=SymmetricTensor(Cq=5e5, eta=0.3),  # Cq in Hz
)
spin_systems = [
    SpinSystem(sites=[Al_1], name="AlO4"),
```

(continues on next page)

(continued from previous page)

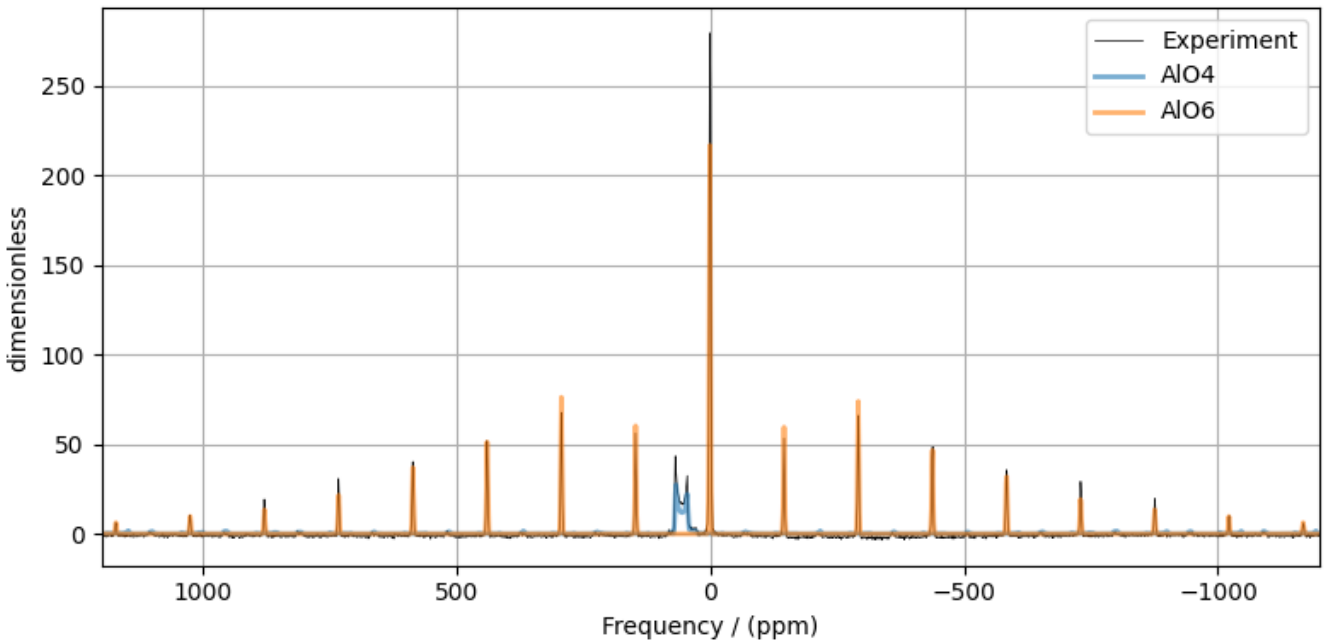
```
    SpinSystem(sites=[Al_2], name="AlO6"),
]
```

Method

```
# Get the spectral dimension parameters from the experiment.
spectral_dims = get_spectral_dimensions(experiment)

MAS = BlochDecaySpectrum(
    channels=["27Al"],
    magnetic_flux_density=9.395,  # in T
    rotor_frequency=15250,  # in Hz
    spectral_dimensions=spectral_dims,
    experiment=experiment,  # add the measurement to the method.
)
```

Guess Spectrum

```
# Simulation
# ----------
sim = Simulator(spin_systems=spin_systems, methods=[MAS])
sim.config.decompose_spectrum = "spin_system"
sim.run()

# Post Simulation Processing
# --------------------------
processor = sp.SignalProcessor(
    operations=[
        sp.IFFT(),
        sp.apodization.Gaussian(FWHM="300 Hz"),
        sp.FFT(),
        sp.Scale(factor=500),
    ]
)
processed_dataset = processor.apply_operations(dataset=sim.methods[0].simulation).real

# Plot of the guess Spectrum
# --------------------------
plt.figure(figsize=(8, 4))
ax = plt.subplot(projection="csdm")
ax.plot(experiment, color="black", linewidth=0.5, label="Experiment")
ax.plot(processed_dataset, linewidth=2, alpha=0.6)
ax.set_xlim(1200, -1200)
plt.grid()
plt.legend()
plt.tight_layout()
plt.show()
```



Least-squares minimization with LMFIT

Use the *make_LMFIT_params()* (page 497) for a quick setup of the fitting parameters.

```
params = sf.make_LMFIT_params(sim, processor, include={"rotor_frequency"})
print(params.pretty_print(columns=["value", "min", "max", "vary", "expr"]))
```

```
Name                                    Value      Min      Max     Vary     Expr
SP_0_operation_1_Gaussian_FWHM            300     -inf      inf     True     None
SP_0_operation_3_Scale_factor             500     -inf      inf     True     None
mth_0_rotor_frequency                1.525e+04 1.515e+04 1.535e+04     True     None
sys_0_abundance                           50        0      100     True     None
sys_0_site_0_isotropic_chemical_shift     76     -inf      inf     True     None
sys_0_site_0_quadrupolar_Cq              6e+06     -inf      inf     True     None
sys_0_site_0_quadrupolar_eta                0        0        1     True     None
sys_1_abundance                            50        0      100    False 100-sys_0_abundance
sys_1_site_0_isotropic_chemical_shift       1     -inf      inf     True     None
sys_1_site_0_quadrupolar_Cq              5e+05     -inf      inf     True     None
sys_1_site_0_quadrupolar_eta               0.3        0        1     True     None
None
```

Solve the minimizer using LMFIT

```
opt = sim.optimize()  # Pre-compute transition pathways
minner = Minimizer(
    sf.LMFIT_min_function,
    params,
    fcn_args=(sim, processor, sigma),
    fcn_kws={"opt": opt},
)
```

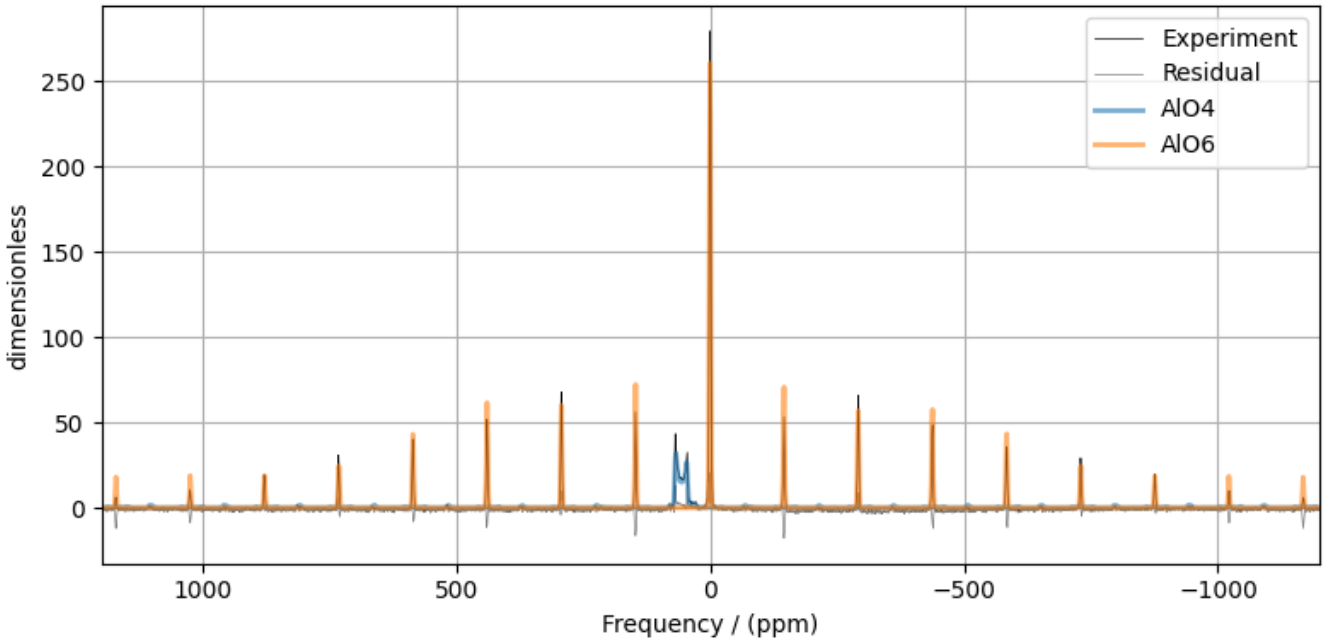
(continues on next page)

(continued from previous page)

```
result = minner.minimize()
result
```

The best fit solution

```
best_fit = sf.bestfit(sim, processor)[0].real
residuals = sf.residuals(sim, processor)[0].real

# Plot the spectrum
plt.figure(figsize=(8, 4))
ax = plt.subplot(projection="csdm")
ax.plot(experiment, color="black", linewidth=0.5, label="Experiment")
ax.plot(residuals, color="gray", linewidth=0.5, label="Residual")
ax.plot(best_fit, linewidth=2, alpha=0.6)
ax.set_xlim(1200, -1200)
plt.grid()
plt.legend()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 10.738 seconds)

^{2}H MAS NMR of Methionine

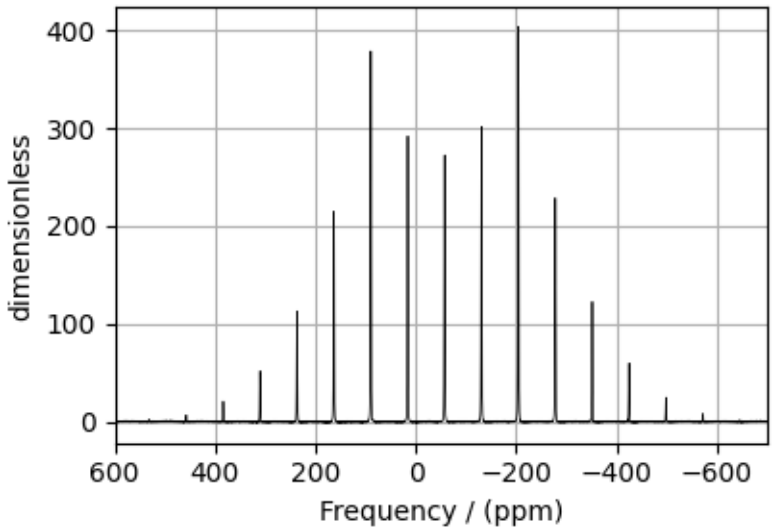
The following is a least-squares fitting example of a ^{2}H MAS NMR spectrum of Methionine. The experimental dataset is a part of DMFIT[1] examples. We thank Dr. Dominique Massiot for sharing the dataset.

```
import csdmpy as cp
import numpy as np
import matplotlib.pyplot as plt
from lmfit import Minimizer

from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.method.lib import BlochDecaySpectrum
from mrsimulator import signal_processor as sp
from mrsimulator.utils import spectral_fitting as sf
from mrsimulator.utils import get_spectral_dimensions
from mrsimulator.spin_system.tensors import SymmetricTensor
```

Import the dataset

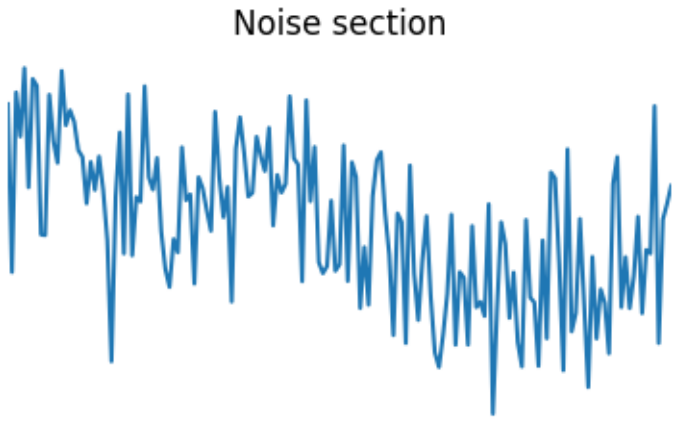
```
host = "https://nmr.cemhti.cnrs-orleans.fr/Dmfit/Help/csdm/"
filename = "2H methiodine MAS.csdf"
experiment = cp.load(host + filename)

# 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]

# plot of the dataset.
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(experiment, color="black", linewidth=0.5, label="Experiment")
ax.set_xlim(600, -700)
plt.grid()
plt.tight_layout()
plt.show()
```

[1] D.Massiot, F.Fayon, M.Capron, I.King, S.Le Calvé, B.Alonso, J.O.Durand, B.Bujoli, Z.Gan, G.Hoatson, 'Modelling one and two-dimensional solid-state NMR spectra.', Magn. Reson. Chem. **40** 70-76 (2002) DOI: 10.1002/mrc.984



Estimate noise statistics from the dataset

```
coords = experiment.dimensions[0].coordinates
noise_region = np.where(np.logical_and(coords > -250e-6, coords < -210e-6))
noise_data = experiment[noise_region]

plt.figure(figsize=(3.75, 2.5))
ax = plt.subplot(projection="csdm")
ax.plot(noise_data, label="noise")
plt.title("Noise section")
plt.axis("off")
plt.tight_layout()
plt.show()

noise_mean, sigma = experiment[noise_region].mean(), experiment[noise_region].std()
noise_mean, sigma
```




```
(<Quantity -0.1204676>, <Quantity 0.35856736>)
```

Create a fitting model

Spin System

```
H_2 = Site(
    isotope="2H",
    isotropic_chemical_shift=-57.12,  # in ppm,
    quadrupolar=SymmetricTensor(Cq=3e4, eta=0.0),  # Cq in Hz
)

spin_systems = [SpinSystem(sites=[H_2])]
```

Method

```
# Get the spectral dimension parameters from the experiment.
spectral_dims = get_spectral_dimensions(experiment)

MAS = BlochDecaySpectrum(
    channels=["2H"],
    magnetic_flux_density=9.395,  # in T
    rotor_frequency=4517.1,  # in Hz
    spectral_dimensions=spectral_dims,
    experiment=experiment,  # experimental dataset
)
```

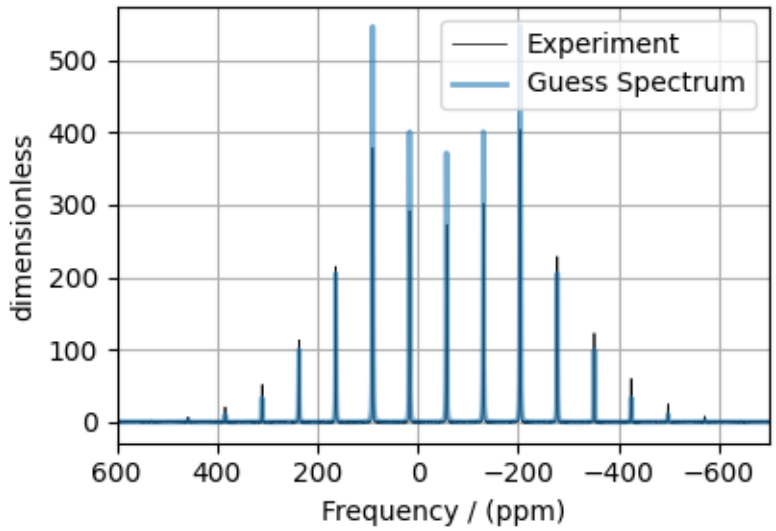
Guess Model Spectrum

```
# Simulation
# ----------
sim = Simulator(spin_systems=spin_systems, methods=[MAS])
sim.run()

# Post Simulation Processing
# --------------------------
processor = sp.SignalProcessor(
    operations=[
        sp.IFFT(),
        sp.apodization.Exponential(FWHM="60 Hz"),
        sp.FFT(),
        sp.Scale(factor=1400),
    ]
)
processed_dataset = processor.apply_operations(dataset=sim.methods[0].simulation).real

# Plot of the guess Spectrum
# --------------------------
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(experiment, color="black", linewidth=0.5, label="Experiment")
ax.plot(processed_dataset, linewidth=2, alpha=0.6, label="Guess Spectrum")
ax.set_xlim(600, -700)
plt.grid()
plt.legend()
```

(continues on next page)

(continued from previous page)

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



Least-squares minimization with LMFIT

Use the *make_LMFIT_params()* (page 497) for a quick setup of the fitting parameters.

```
params = sf.make_LMFIT_params(sim, processor)
params["sys_0_site_0_isotropic_chemical_shift"].vary = False
print(params.pretty_print(columns=["value", "min", "max", "vary", "expr"]))
```

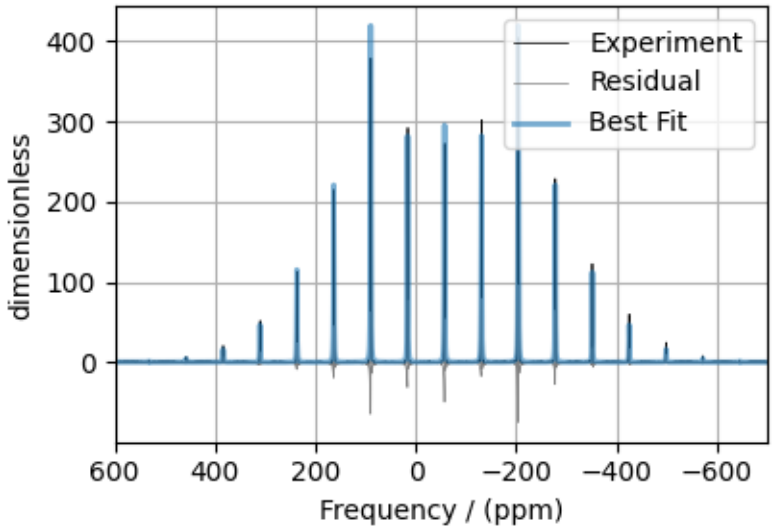
```
Name                                    Value      Min      Max     Vary     Expr
SP_0_operation_1_Exponential_FWHM          60     -inf      inf     True     None
SP_0_operation_3_Scale_factor            1400     -inf      inf     True     None
sys_0_abundance                           100        0      100    False      100
sys_0_site_0_isotropic_chemical_shift  -57.12     -inf      inf    False     None
sys_0_site_0_quadrupolar_Cq             3e+04     -inf      inf     True     None
sys_0_site_0_quadrupolar_eta                0        0        1     True     None
None
```

Solve the minimizer using LMFIT

```
opt = sim.optimize()  # Pre-compute transition pathways
minner = Minimizer(
    sf.LMFIT_min_function,
    params,
    fcn_args=(sim, processor, sigma),
    fcn_kws={"opt": opt},
)
result = minner.minimize()
result
```

The best fit solution

```
best_fit = sf.bestfit(sim, processor)[0].real
residuals = sf.residuals(sim, processor)[0].real

# Plot the spectrum
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(experiment, color="black", linewidth=0.5, label="Experiment")
ax.plot(residuals, color="gray", linewidth=0.5, label="Residual")
ax.plot(best_fit, linewidth=2, alpha=0.6, label="Best Fit")
ax.set_xlim(600, -700)
plt.grid()
plt.legend()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 2.328 seconds)

^{119}Sn MAS NMR of SnO

The following is a spinning sideband manifold fitting example for the 119Sn MAS NMR of SnO. The dataset was acquired and shared by Altenhof *et al.*[1].

```
import csdmpy as cp
import numpy as np
import matplotlib.pyplot as plt
from lmfit import Minimizer

from mrsimulator import Simulator, SpinSystem, Site, Coupling
```

(continues on next page)

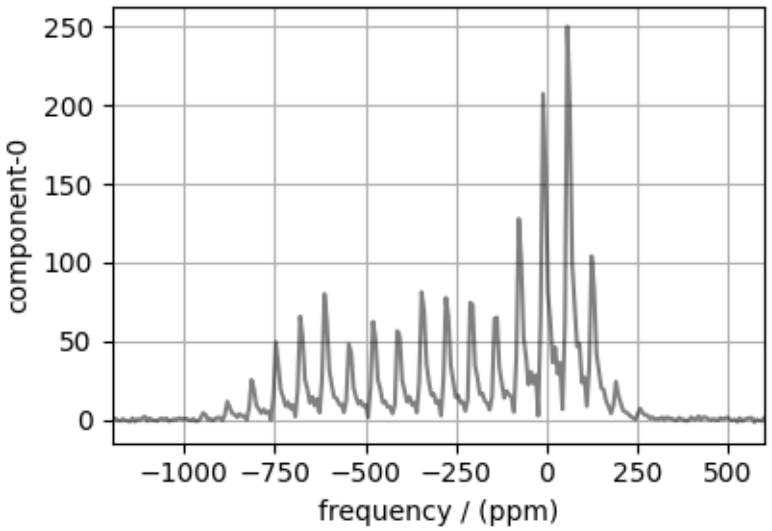
[1] Altenhof A. R., Jaroszewicz M. J., Lindquist A. W., Foster L. D. D., Veinberg S. L., and Schurko R. W. Practical Aspects of Recording Ultra-Wideline NMR Patterns under Magic-Angle Spinning Conditions. J. Phys. Chem. C. 2020, **124**, 27, 14730–14744 DOI: 10.1021/acs.jpcc.0c04510

(continued from previous page)

```
from mrsimulator.method.lib import BlochDecaySpectrum
from mrsimulator import signal_processor as sp
from mrsimulator.utils import spectral_fitting as sf
from mrsimulator.utils import get_spectral_dimensions
from mrsimulator.spin_system.tensors import SymmetricTensor
```

Import the dataset

```
filename = "https://ssnmr.org/sites/default/files/mrsimulator/119Sn_SnO.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]

# plot of the dataset.
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(experiment, "k", alpha=0.5)
ax.set_xlim(-1200, 600)
plt.grid()
plt.tight_layout()
plt.show()
```



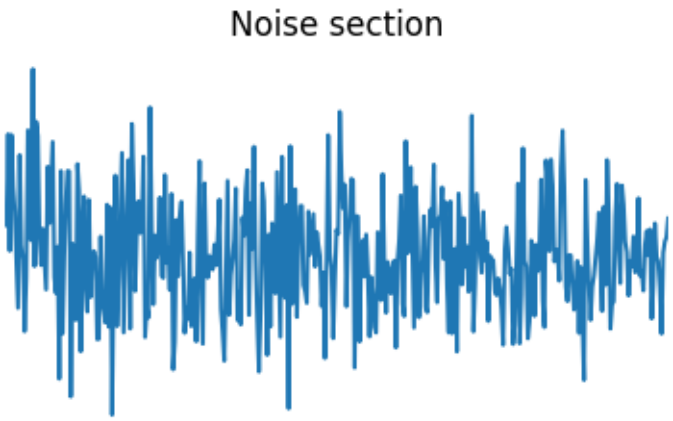
Estimate noise statistics from the dataset

```
coords = experiment.dimensions[0].coordinates
noise_region = np.where(coords > 300e-6)
noise_data = experiment[noise_region]
```

(continues on next page)

(continued from previous page)

```
plt.figure(figsize=(3.75, 2.5))
ax = plt.subplot(projection="csdm")
ax.plot(noise_data, label="noise")
plt.title("Noise section")
plt.axis("off")
plt.tight_layout()
plt.show()

noise_mean, sigma = experiment[noise_region].mean(), experiment[noise_region].std()
noise_mean, sigma
```




```
(<Quantity 0.06896736>, <Quantity 0.67926073>)
```

Create a fitting model

Guess model

Create a guess list of spin systems. There are two spin systems present in this example, - 1) an uncoupled ^{119}Sn and - 2) a coupled ^{119}Sn-^{117}Sn spin systems.

```
sn119 = Site(
    isotope="119Sn",
    isotropic_chemical_shift=-210,
    shielding_symmetric=SymmetricTensor(zeta=700, eta=0.1),
)
sn117 = Site(
    isotope="117Sn",
    isotropic_chemical_shift=0,
)
j_sn = Coupling(
    site_index=[0, 1],
    isotropic_j=8150.0,
)

sn117_abundance = 7.68  # in %
```

(continues on next page)

(continued from previous page)

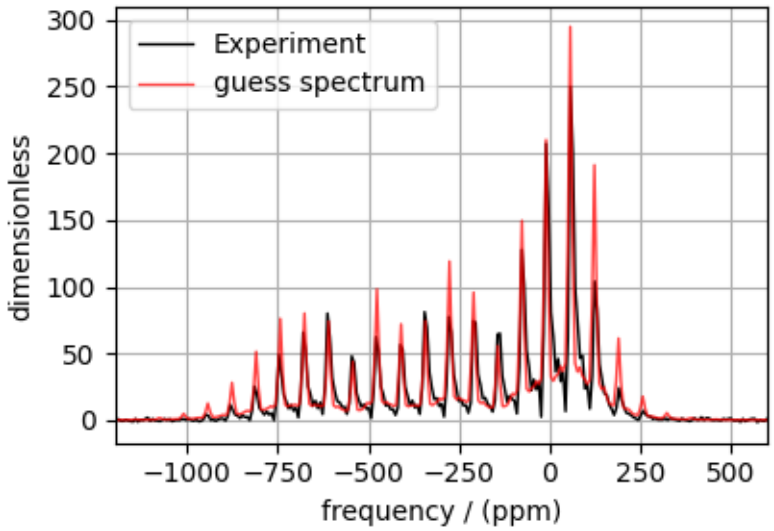
```
spin_systems = [
    # uncoupled spin system
    SpinSystem(sites=[sn119], abundance=100 - sn117_abundance),
    # coupled spin systems
    SpinSystem(sites=[sn119, sn117], couplings=[j_sn], abundance=sn117_abundance),
]
```

Method

```
# Get the spectral dimension parameters from the experiment.
spectral_dims = get_spectral_dimensions(experiment)

MAS = BlochDecaySpectrum(
    channels=["119Sn"],
    magnetic_flux_density=9.395,  # in T
    rotor_frequency=10000,  # in Hz
    spectral_dimensions=spectral_dims,
    experiment=experiment,  # add the measurement to the method.
)
```

Guess Spectrum

```
# Simulation
# ----------
sim = Simulator(spin_systems=spin_systems, methods=[MAS])
sim.run()

# Post Simulation Processing
# --------------------------
processor = sp.SignalProcessor(
    operations=[
        sp.IFFT(),
        sp.apodization.Exponential(FWHM="1500 Hz"),
        sp.FFT(),
        sp.Scale(factor=50000),
    ]
)
processed_dataset = processor.apply_operations(dataset=sim.methods[0].simulation).real

# Plot of the guess Spectrum
# --------------------------
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(experiment, "k", linewidth=1, label="Experiment")
ax.plot(processed_dataset, "r", alpha=0.75, linewidth=1, label="guess spectrum")
ax.set_xlim(-1200, 600)
plt.grid()
plt.legend()
plt.tight_layout()
plt.show()
```



Least-squares minimization with LMFIT

Use the *make_LMFIT_params()* (page 497) for a quick setup of the fitting parameters.

```
params = sf.make_LMFIT_params(sim, processor, include={"rotor_frequency"})

# Remove the abundance parameters from params. Since the measurement detects 119Sn, we
# also remove the isotropic chemical shift parameter of 117Sn site from params. The
# 117Sn is the site at index 1 of the spin system at index 1.
params.pop("sys_0_abundance")
params.pop("sys_1_abundance")
params.pop("sys_1_site_1_isotropic_chemical_shift")

# Since the 119Sn site is shared between the two spin systems, we add constraints to the
# 119Sn site parameters from the spin system at index 1 to be the same as 119Sn site
# parameters from the spin system at index 0.
lst = [
    "isotropic_chemical_shift",
    "shielding_symmetric_zeta",
    "shielding_symmetric_eta",
]
for item in lst:
    params[f"sys_1_site_0_{item}"].expr = f"sys_0_site_0_{item}"

print(params.pretty_print(columns=["value", "min", "max", "vary", "expr"]))
```

```
Name                                    Value      Min      Max     Vary     Expr
SP_0_operation_1_Exponential_FWHM        1500     -inf      inf     True     None
SP_0_operation_3_Scale_factor           5e+04     -inf      inf     True     None
mth_0_rotor_frequency                   1e+04     9900 1.01e+04     True     None
sys_0_site_0_isotropic_chemical_shift    -210     -inf      inf     True     None
sys_0_site_0_shielding_symmetric_eta      0.1        0        1     True     None
sys_0_site_0_shielding_symmetric_zeta     700     -inf      inf     True     None
```

(continues on next page)

(continued from previous page)

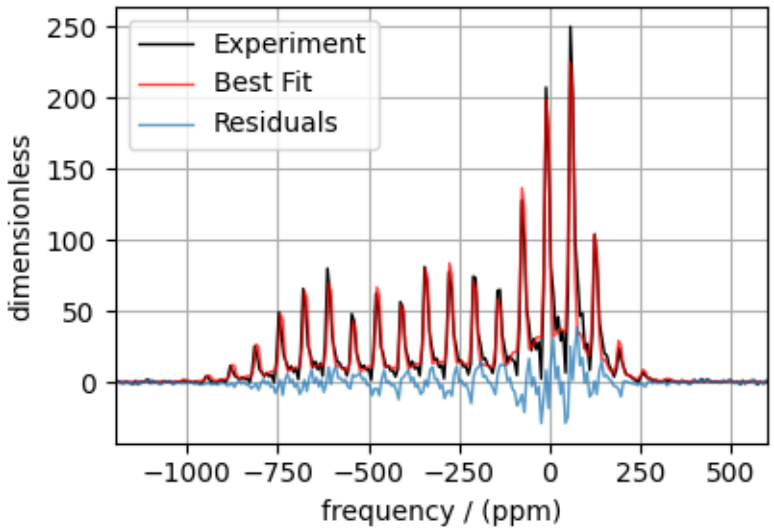
```
sys_1_coupling_0_isotropic_j                8150     -inf      inf     True     None
sys_1_site_0_isotropic_chemical_shift       -210     -inf      inf    False sys_0_site_0_
↪isotropic_chemical_shift
sys_1_site_0_shielding_symmetric_eta         0.1        0        1    False sys_0_site_0_
↪shielding_symmetric_eta
sys_1_site_0_shielding_symmetric_zeta        700     -inf      inf    False sys_0_site_0_
↪shielding_symmetric_zeta
None
```

Solve the minimizer using LMFIT

```
opt = sim.optimize()  # Pre-compute transition pathways
minner = Minimizer(
    sf.LMFIT_min_function,
    params,
    fcn_args=(sim, processor, sigma),
    fcn_kws={"opt": opt},
)
result = minner.minimize()
result
```

The best fit solution

```
best_fit = sf.bestfit(sim, processor)[0].real
residuals = sf.residuals(sim, processor)[0].real

# Plot the spectrum
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.plot(experiment, "k", linewidth=1, label="Experiment")
ax.plot(best_fit, "r", alpha=0.75, linewidth=1, label="Best Fit")
ax.plot(residuals, alpha=0.75, linewidth=1, label="Residuals")
ax.set_xlim(-1200, 600)
plt.grid()
plt.legend()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 2.727 seconds)

Extended Czjzek fitting of ^{139}La MAS NMR of $La_{0.2}Y_{1.8}Si_2O_7$

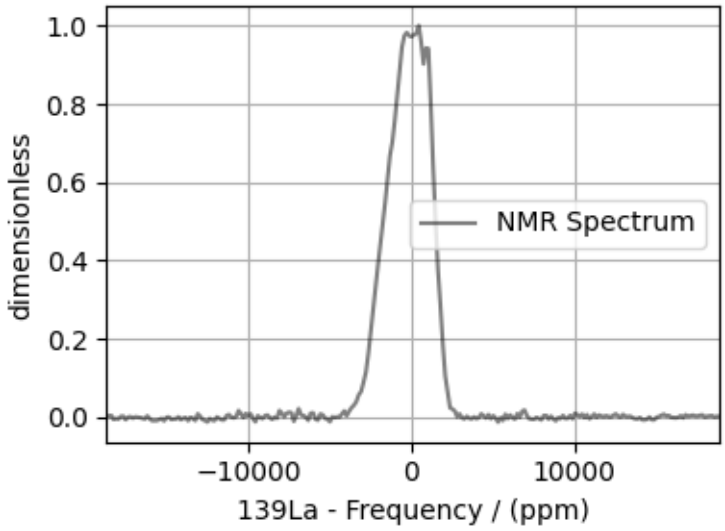
The following is a demonstration on how to fit tensor distributions to an experimental spectrum using a structure-forward approach. The dataset was acquired from a sample of $La_{0.2}Y_{1.8}Si_2O_7$ silicate glass and shared by Fernańdez-Carrioń *et al.*[1].

```
from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.method.lib import BlochDecayCTSpectrum
from mrsimulator.models import ExtCzjzekDistribution
from mrsimulator.simulator import Isotope
from mrsimulator.utils import get_spectral_dimensions
from mrsimulator import signal_processor as sp

import numpy as np
import csdmpy as cp
import lmfit
from lmfit import Minimizer
import matplotlib.pyplot as plt
```

[1] A. J. Fernández-Carrión, M. Allix, P. Florian, M. R. Suchomel, and A. I. Becerro. Revealing Structural Detail in the High Temperature La2Si2O7–Y2Si2O7 Phase Diagram by Synchrotron Powder Diffraction and Nuclear Magnetic Resonance Spectroscopy. The Journal of Physical Chemistry C 2012 **116** (40), 21523-21535 DOI: 10.1021/jp305777m

Import the dataset

```
host = "http://ssnmr.org/sites/default/files/"
filename = "mrsimulator/La bc - LaY90 cpmg.csdf"
experiment = cp.load(host + filename).real
experiment.x[0].to("ppm", "nmr_frequency_ratio")
experiment /= experiment.max()

plt.figure(figsize=(4, 3))
ax = plt.subplot(projection="csdm")
ax.plot(experiment, "k", alpha=0.5)
plt.grid()
plt.tight_layout()
plt.show()
```



Calculate Noise Standard Deviation

```
loc = np.where(experiment.dimensions[0].coordinates < -5000e-6)
sigma = experiment[loc].std()
print(sigma)
```

```
0.006770052015781403
```

Setup Method and Processor

Next setup a method which simulates the experiment, again using the *get_spectral_dimensions()* (page 487) function to get the spectral dimension parameters from the experimental dataset. We also create a signal processor object which will scale the simulated spectrum.

```
spec_dims = get_spectral_dimensions(experiment)
method = BlochDecayCTSpectrum(
    channels=["139La"], magnetic_flux_density=17.6, spectral_dimensions=spec_dims
)

processor = sp.SignalProcessor(operations=[sp.Scale(factor=1000)])
```

Create a Lineshape Kernel

An approach for fitting model values to the experimental dataset could be computing the spectrum from a list of spin system objects with tensor parameters pulled from an Extended Czjzek model during each round of fitting, but this would be inefficient! The grid of Cq and η_q points will not change during the fit; we are interested in the probability distribution across these points.

We therefore define a function which pre-computes a lineshape kernel given a range of Cq and η_q values, an NMR method, and an isotope which, for this example, is `"139La"`.

```
def make_kernel(pos, method, isotope):
    """Pre-computes the kernel to use when fitting the experimental spectrum

    Arguments:
        (np.array) pos: list of Numpy array of cq and eta points
        (Method) method: mrsimulator Method object used to simulate the spectra
        (str) isotope: Isotope of the site

    Returns:
        Lineshape kernel as a numpy array
    """
    Cq, eta = np.meshgrid(pos[0], pos[1], indexing="xy")

    spin_systems = [
        SpinSystem(sites=[Site(isotope=isotope, quadrupolar=dict(Cq=cq_, eta=e_))])
        for cq_, e_ in zip(Cq.ravel(), eta.ravel())
    ]
    sim = Simulator(spin_systems=spin_systems, methods=[method])
    sim.config.number_of_sidebands = 4
    sim.config.decompose_spectrum = "spin_system"
    sim.run(pack_as_csdm=False)  # Will return spectrum as numpy array, not CSDM object

    amp = sim.methods[0].simulation.real
    return amp.T


# Create ranges to construct cq and eta grid points
cq_range = (np.linspace(0, 100, num=100) * 0.8 + 25) * 1e6  # in Hz
eta_range = np.arange(21) / 20
```

(continues on next page)

(continued from previous page)

```
pos = [cq_range, eta_range]
kernel = make_kernel(pos, method, "139La")
```

Make spectrum from kernel and model values

Next define a function which takes a LMFIT Parameters object, the pre-computed kernel and a signal processor object and returns a CSDM object holding the guess spectrum. Here the Parameters object holds attributes from the *ExtCzjzekDistribution* (page 495) class which is used to create the probability density function, $\mathbf{f}$. Then $\mathbf{f}$ is multiplied with the kernel, $\mathbf{K}$, to produce a spectrum, $\mathbf{s}$.

$$\mathbf{s} = \mathbf{K} \cdot \mathbf{f}, \tag{13.1}$$

A constant isotropic chemical shift, whose value is also in the Parameters object, is applied to $\mathbf{s}$ using the Fourier shift relation. Finally, the spectrum is scaled and returned.

```
def make_spectrum_from_parameters(params, kernel, processor, pos, distribution):
    """Makes a spectrum with values given in a parameters object and a pre-computed
    kernel.

    Arguments:
        (Parameters) params: LMFIT Parameters object with values
        (np.array) kernel: Pre-computed spectrum kernel
        (sp.SignalProcessor) processor: SignalProcessor to apply to spectrum

    Returns:
        CSDM object of spectrum
    """
    # Extract values from parameters object
    values = params.valuesdict()
    Cq = values["dist_Cq"]
    eta = values["dist_eta"]
    eps = values["dist_eps"]
    iso_shift = values["dist_iso_shift"]

    # Setup model object and get the amplitude
    distribution.symmetric_tensor.Cq = Cq
    distribution.symmetric_tensor.eta = eta
    distribution.eps = eps
    _, _, amp = distribution.pdf(pos=pos)

    # Create spectra by dotting the amplitude distribution with the kernel
    dist = np.dot(kernel, amp.ravel())

    # Pack numpy array as csdm object and apply signal processing
    guess_dataset = cp.CSDM(
        dimensions=experiment.x,
        dependent_variables=[cp.as_dependent_variable(dist)],
    )

    # Calculate isotropic shift in Hz
    larmor_freq = Isotope(symbol="139La").gyromagnetic_ratio * 17.6
```

(continues on next page)

(continued from previous page)

```
    iso_shift_in_hz = larmor_freq * iso_shift

    # Apply isotropic shift using FFT shift theorem
    guess_dataset = guess_dataset.fft()
    time_coords = guess_dataset.x[0].coordinates.value
    guess_dataset.y[0].components[0] *= np.exp(
        -np.pi * 2j * iso_shift_in_hz * time_coords
    )
    guess_dataset = guess_dataset.fft()

    # Apply signal processor and return
    processor.operations[0].factor = values["sp_scale_factor"]
    guess_dataset = processor.apply_operations(guess_dataset)
    return guess_dataset.real


def residuals(exp_spectra, simulated_spectra):
    """Returns the difference between exp_spectra and simulated_spectra"""
    return exp_spectra - simulated_spectra
```

Make initial guess

Next pack the attributes to be fit into a Parameters object and plot the initial guess spectrum.

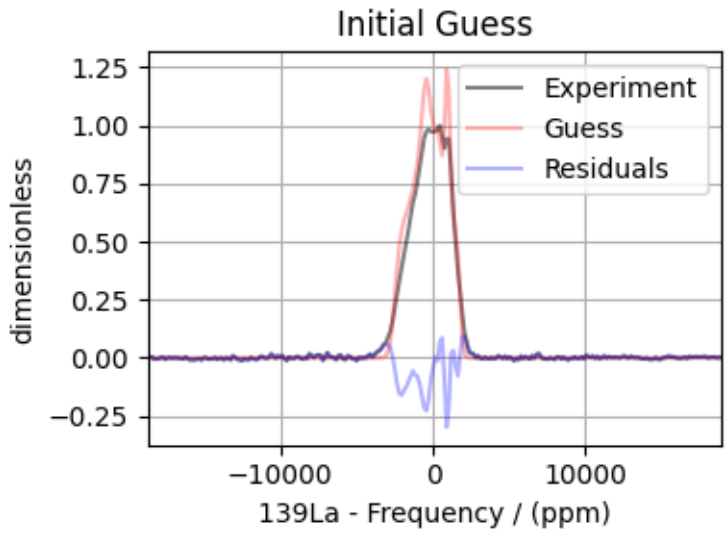
Note: If you adapt this example to your own dataset, make sure the initial guess is decently good, otherwise LMFIT is likely to fall into a local minima.

```
params = lmfit.Parameters()
params.add("dist_Cq", value=49.5e6)  # Hz
params.add("dist_eta", value=0.55, min=0, max=1)
params.add("dist_eps", value=0.1, min=0)
params.add("dist_iso_shift", value=350)  # ppm
params.add("sp_scale_factor", value=3.8e3, min=0)

# Plot the initial guess spectrum along with the experimental data
distribution = ExtCzjzekDistribution(
    symmetric_tensor={"Cq": params["dist_Cq"], "eta": params["dist_eta"]},
    eps=params["dist_eps"],
)
guess_distribution = distribution.pdf(pos, size=400_000, pack_as_csdm=True)

initial_guess_spectrum = make_spectrum_from_parameters(
    params, kernel, processor, pos, distribution
)
residual_spectrum = residuals(experiment, initial_guess_spectrum)

plt.figure(figsize=(4, 3))
ax = plt.subplot(projection="csdm")
ax.plot(experiment.real, "k", alpha=0.5, label="Experiment")
```

(continues on next page)

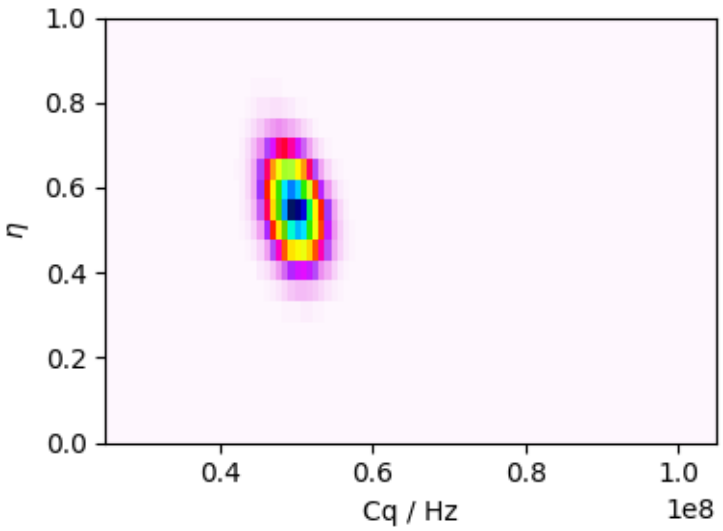
(continued from previous page)

```
ax.plot(initial_guess_spectrum.real, "r", alpha=0.3, label="Guess")
ax.plot(residual_spectrum.real, "b", alpha=0.3, label="Residuals")
plt.legend()
plt.grid()
plt.title("Initial Guess")
plt.tight_layout()
plt.show()
```



Guess distribution

```
plt.figure(figsize=(4, 3))
ax = plt.subplot(projection="csdm")
ax.imshow(guess_distribution, interpolation="none", cmap="gist_ncar_r", aspect="auto")
ax.set_xlabel("Cq / Hz")
ax.set_ylabel(r"$\eta$")
plt.tight_layout()
plt.show()
```



Least-squares minimization with LMFIT

Now define minimization function per the LMFIT specifications which will return the difference between the experimental and guess spectrum scaled by the noise standard deviation which was calculated in a previous cell.

```
def minimization_function(
    params, experiment, processor, kernel, pos, distribution, sigma=sigma
):
    guess_spectrum = make_spectrum_from_parameters(
        params, kernel, processor, pos, distribution
    )
    residual_spectrum = residuals(experiment, guess_spectrum)
    return residual_spectrum.y[0].components[0].real / sigma
```

Sinice the probabilty distribution is generated from a sparsely sampled from a 5D second rank tensor parameter space, we increase the *diff_ step* size from machine precession to avoid approaching local minima from noise.

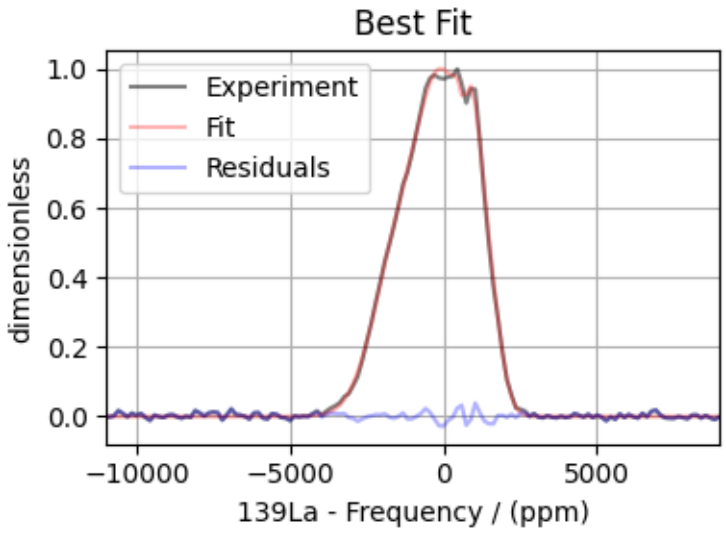
```
scipy_minimization_kwargs = dict(
    diff_step=1e-4,  # Increase step size from machine precession
    gtol=1e-10,  # Decrease global convergence requirement (default 1e-8)
    xtol=1e-10,  # Decrease variable convergence requirement (default 1e-8)
    verbose=2,  # Print minimization info during each step
    loss="linear",
)

minner = Minimizer(
    minimization_function,
    params,
    fcn_args=(experiment, processor, kernel, pos, distribution),
    **scipy_minimization_kwargs,
)
result = minner.minimize(method="least_squares")
best_fit_params = result.params  # Grab the Parameters object from the best fit
result
```

```
   Iteration     Total nfev        Cost      Cost reduction    Step norm     Optimality
       0              1         5.6207e+03                                    5.30e+04
       1              2         5.4536e+02      5.08e+03       2.54e+05       1.18e+04
       2              3         1.8821e+02      3.57e+02       3.37e+05       1.43e+03
       3              4         1.7542e+02      1.28e+01       1.97e+04       5.21e+02
       4              5         1.7361e+02      1.82e+00       7.38e+04       3.07e+02
       5             18         1.7361e+02      0.00e+00       0.00e+00       3.07e+02
`xtol` termination condition is satisfied.
Function evaluations 18, initial cost 5.6207e+03, final cost 1.7361e+02, first-order optimality␣
↪3.07e+02.
```

Plot the best-fit solution

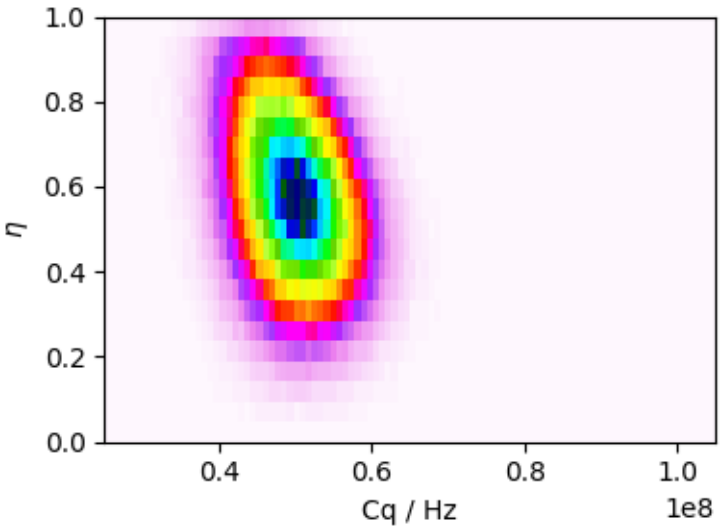
Finally, plot the best fit spectrum and the residuals.

```
final_fit = make_spectrum_from_parameters(
    best_fit_params, kernel, processor, pos, distribution
)
residual_spectrum = residuals(experiment, final_fit)

plt.figure(figsize=(4, 3))
ax = plt.subplot(projection="csdm")
ax.plot(experiment, "k", alpha=0.5, label="Experiment")
ax.plot(final_fit, "r", alpha=0.3, label="Fit")
ax.plot(residual_spectrum, "b", alpha=0.3, label="Residuals")
plt.legend()
ax.set_xlim(-11000, 9000)
plt.grid()
plt.title("Best Fit")
plt.tight_layout()
plt.show()
```



Best fit probability distribution

```
prob = distribution.pdf(pos, pack_as_csdm=True)
plt.figure(figsize=(4, 3))
ax = plt.subplot(projection="csdm")
ax.imshow(prob, interpolation="none", cmap="gist_ncar_r", aspect="auto")
ax.set_xlabel("Cq / Hz")
ax.set_ylabel(r"$\eta$")
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 7.197 seconds)

Fitting a Czjzek Model

In this example, we illustrate an application of mrsimulator in fitting a lineshape from a Czjzek distribution of quadrupolar tensors to an experimental ^{27}Al MAS spectrum of a phospho-aluminosilicate glass. Setting up the least-squares minimization for a distribution of spin systems is slightly different than that of a crystalline solid.

There are 4 steps involved in the processs:

- Importing the experimental dataset,
- Generating a pre-computed line shape kernel of subspectra,
- Creating parameters for the Czjzek distribution model from an initial guess,
- Minimizing and visualizing.

```
import numpy as np
import csdmpy as cp
import matplotlib.pyplot as plt
import mrsimulator.signal_processor as sp
from mrsimulator.simulator.config import ConfigSimulator
import mrsimulator.utils.spectral_fitting as sf
from lmfit import Minimizer

from mrsimulator.method.lib import BlochDecayCTSpectrum
from mrsimulator.utils import get_spectral_dimensions
```

(continues on next page)

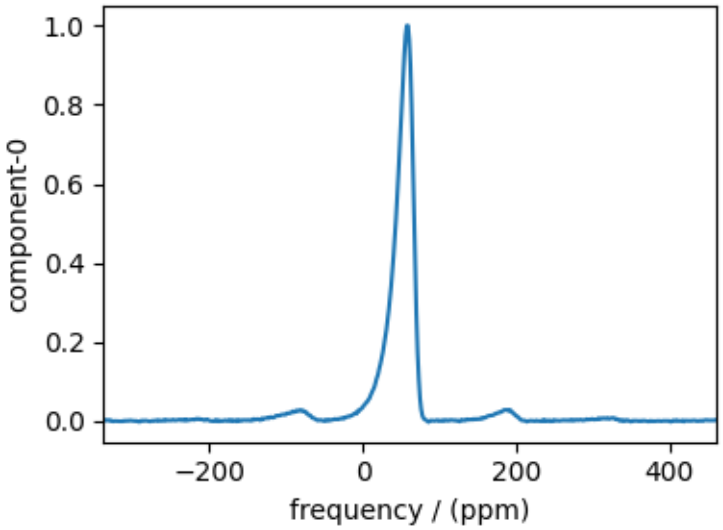
(continued from previous page)

```
from mrsimulator.models.czjzek import CzjzekDistribution
from mrsimulator.models.utils import LineShapeKernel
```

Import the experimental dataset

Below we import and visualize the experimental dataset.

```
host = "http://ssnmr.org/sites/default/files/mrsimulator/"
filename = "20K_20Al_10P_50Si_HahnEcho_27Al.csdf"
exp_data = cp.load(host + filename).real

exp_data.x[0].to("ppm", "nmr_frequency_ratio")
exp_data /= exp_data.max()

plt.figure(figsize=(4, 3))
ax = plt.subplot(projection="csdm")
ax.plot(exp_data)
plt.tight_layout()
plt.show()
```



Generating a line shape kernel

The Czjzek distribution is a statistical model that represents a five-dimensional multivariate normal distribution of second-rank tensor components. These random second-rank tensors are converted into corresponding anisotropy and asymmetry parameters using the Haeberlen convention. The resulting parameters are then binned on a two-dimensional grid, forming the Czjzek probability distribution. The Czjzek spectra is which are the probability weighted sum of lineshapes at each grid point.

However, simulating the spectra of spin systems at each minimization step for the least-squares fitting is computationally expensive. A more efficient way is to pre-define a grid system for the tensor parameters, simulate a library of sub-spectra for each grid point, and only update the probability distribution during each minimization step.

To simulate the spectra of the given experiment, we create a Method object. Using this method, we generate a kernel of lineshape sub-spectra defined on a polar grid by using the LineShapeKernel class. The argument "method" is the mrsimulator method object used for the simulation, "pos" is a tuple of coordinates defining the two-dimensional grid, "polar" defines a polar parameter coordinate space, and "config" is the Simulator config object used in lineshape simulation. Here, we use the "config" argument to set the number of sidebands as 8.

The kernel is generated with the "generate_lineshape()" function, where we pass the "tensor_type='quadrupolar'" argument to specify a quadrupolar parameter grid. A symmetric shielding lineshape kernel can also be generated by specifying

```
# Create a Method object to simulate the spectrum
spectral_dimensions = get_spectral_dimensions(exp_data)
method = BlochDecayCTSpectrum(
    channels=["27Al"],
    rotor_frequency=14.2e3,
    spectral_dimensions=spectral_dimensions,
    experiment=exp_data,
)

# Define a polar grid for the lineshape kernel
x = np.linspace(0, 2e7, num=36)
y = np.linspace(0, 2e7, num=36)
pos = (x, y)

# Generate the kernel
sim_config = ConfigSimulator(number_of_sidebands=8)
quad_kernel = LineShapeKernel(method=method, pos=pos, polar=True, config=sim_config)
quad_kernel.generate_lineshape(tensor_type="quadrupolar")

print("Desired Kernel shape: ", (x.size * y.size, spectral_dimensions[0]["count"]))
print("Actual Kernel shape:  ", quad_kernel.kernel.shape)
```

```
Desired Kernel shape:  (1296, 512)
Actual Kernel shape:   (1296, 512)
```

Create a Parameters object

Next, we create an instance of the *CzjzekDistribution* class with initial guess values along with a *SignalProcessor* object.

```
# Create initial guess CzjzekDistribution
cz_model = CzjzekDistribution(
    mean_isotropic_chemical_shift=60.0, sigma=1.4e6, polar=True
)
all_models = [cz_model]

processor = sp.SignalProcessor(
    operations=[
        sp.IFFT(),
        sp.apodization.Gaussian(FWHM="600 Hz"),
        sp.FFT(),
        sp.Scale(factor=0.3),
```

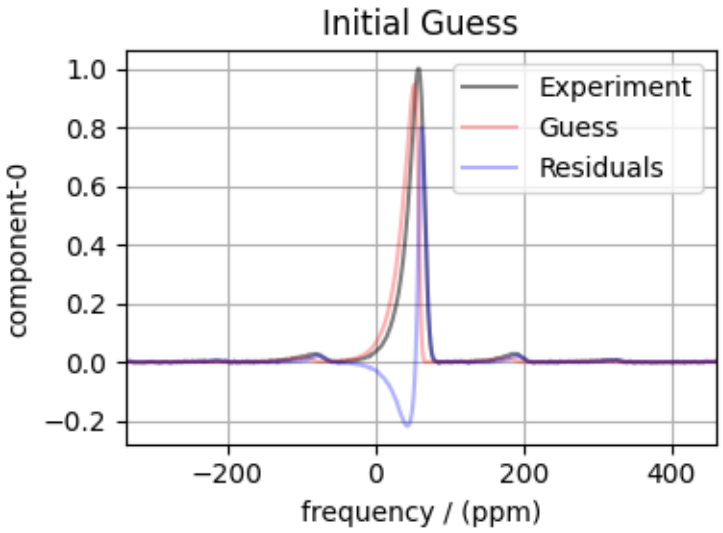
(continues on next page)

(continued from previous page)

```
    ]
)
```

Make the Parameters object and simulate a guess spectrum. Note that the variable *sf_kwargs* holds some additional keyword arguments that many of the spectral fitting function takes in. This dictionary needs to be updated to reflect any changes made in the minimization.

```
params = sf.make_LMFIT_params(spin_system_models=all_models, processors=[processor])

# Additional keyword arguments passed to best-fit and residual functions.
sf_kwargs = dict(
    kernel=quad_kernel,
    spin_system_models=all_models,
    processor=processor,
)

# Make a guess and residuals spectrum from the initial guess
guess = sf.bestfit_dist(params=params, **sf_kwargs)
residuals = sf.residuals_dist(params=params, **sf_kwargs)

plt.figure(figsize=(4, 3))
ax = plt.subplot(projection="csdm")
ax.plot(exp_data, "k", alpha=0.5, label="Experiment")
ax.plot(guess, "r", alpha=0.3, label="Guess")
ax.plot(residuals, "b", alpha=0.3, label="Residuals")
plt.legend()
plt.grid()
plt.title("Initial Guess")
plt.tight_layout()
plt.show()

# Print the Parameters object
params
```



Create and run a minimization

Finally, a *Minimizer* object is created and a minimization run using least-squares. The same arguments defined in the *addtl_sf_kwargs* variable are also passed to the minimizer. Sinice the probabilty distribution is generated from a sparsely sampled from a 5D second rank tensor parameter space, we increase the *diff_step* size from machine precession to avoid approaching local minima from noise.

```
scipy_minimization_kwargs = dict(
    diff_step=1e-4,  # Increase step size from machine precesion.
    gtol=1e-10,  # Decrease global convergence requirement (default 1e-8)
    xtol=1e-10,  # Decrease variable convergence requirement (default 1e-8)
    verbose=2,  # Print minimization info during each step
    loss="linear",
)

minner = Minimizer(
    sf.LMFIT_min_function_dist,
    params,
    fcn_kws=sf_kwargs,
    **scipy_minimization_kwargs,
)
result = minner.minimize(method="least_squares")
result
```

```
   Iteration     Total nfev        Cost      Cost reduction    Step norm     Optimality
       0              1         1.9057e+00                                 4.06e+00
       1              2         4.6019e-01      1.45e+00       1.00e+05       5.95e+00
       2              3         8.4380e-02      3.76e-01       5.94e+04       6.06e-01
       3              4         1.3069e-02      7.13e-02       2.66e+03       4.21e-01
       4              5         6.4617e-03      6.61e-03       4.65e+04       2.81e-02
       5              6         6.4205e-03      4.12e-05       4.78e+03       1.17e-03
       6              7         6.4204e-03      6.98e-08       6.18e+01       1.38e-05
       7              8         6.4204e-03      3.96e-10       8.39e+00       6.97e-07
       8             10         6.4204e-03      4.41e-10       9.13e-02       3.52e-07
       9             12         6.4204e-03      0.00e+00       0.00e+00       3.52e-07
`xtol` termination condition is satisfied.
Function evaluations 12, initial cost 1.9057e+00, final cost 6.4204e-03, first-order optimality␣
↪3.52e-07.
```

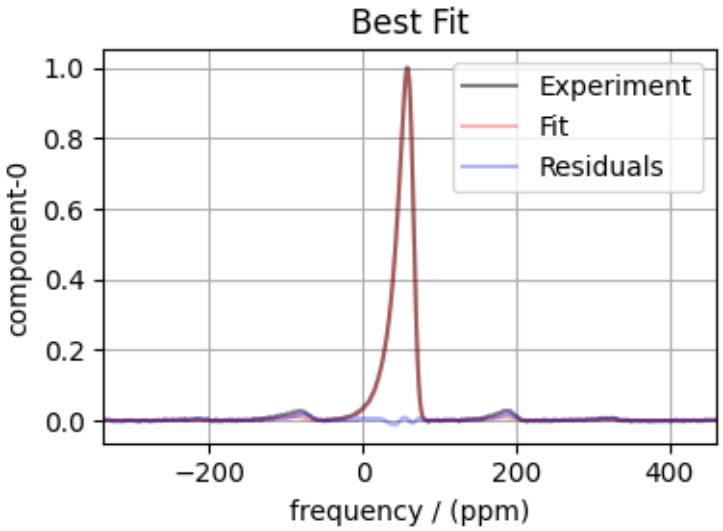
Plot the best-fit spectrum

```
bestfit = sf.bestfit_dist(params=result.params, **sf_kwargs)
residuals = sf.residuals_dist(params=result.params, **sf_kwargs)

plt.figure(figsize=(4, 3))
ax = plt.subplot(projection="csdm")
ax.plot(exp_data, "k", alpha=0.5, label="Experiment")
ax.plot(bestfit, "r", alpha=0.3, label="Fit")
ax.plot(residuals, "b", alpha=0.3, label="Residuals")
plt.legend()
plt.grid()
```

(continues on next page)

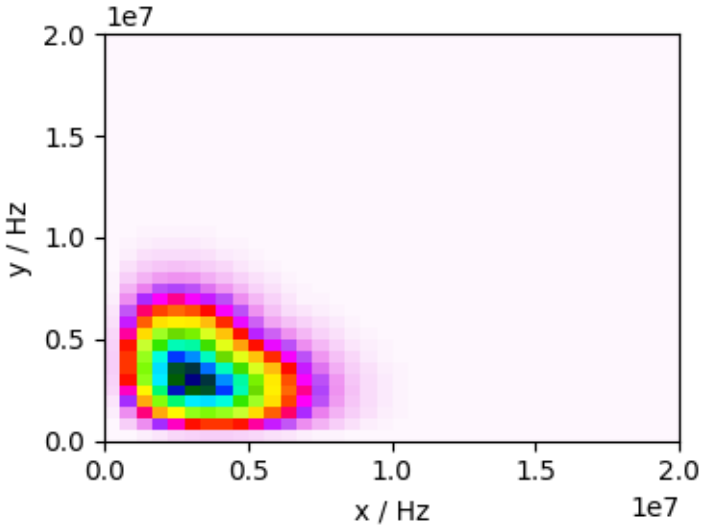
(continued from previous page)

```
plt.title("Best Fit")
plt.tight_layout()
plt.show()
```



Plot the best-fit distribution

```
for i, model in enumerate(all_models):
    model.update_lmfit_params(result.params, i)

amp = cz_model.pdf(pos=pos, pack_as_csdm=True)

plt.figure(figsize=(4, 3))
ax = plt.subplot(projection="csdm")
ax.imshow(amp, cmap="gist_ncar_r", interpolation="none", aspect="auto")
ax.set_xlabel("x / Hz")
ax.set_ylabel("y / Hz")
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 5.877 seconds)

13.2.2 2D Dataset Fitting

^{13}C 2D MAT NMR of L-Histidine

The following is an illustration for fitting 2D MAT/PASS datasets. The example dataset is a ^{13}C 2D MAT spectrum of L-Histidine from Walder *et al.*[1]

```
import numpy as np
import csdmpy as cp
import matplotlib.pyplot as plt
from lmfit import Minimizer

from mrsimulator import Simulator
from mrsimulator.method.lib import SSB2D
from mrsimulator import signal_processor as sp
from mrsimulator.utils import spectral_fitting as sf
from mrsimulator.utils import get_spectral_dimensions
from mrsimulator.utils.collection import single_site_system_generator
```

Import the dataset

```
host = "https://ssnmr.org/sites/default/files/mrsimulator/"
filename = "1H13C_CPPASS_LHistidine.csdf"
mat_dataset = cp.load(host + filename)

# For the spectral fitting, we only focus on the real part of the complex dataset.
mat_dataset = mat_dataset.real
```

(continues on next page)

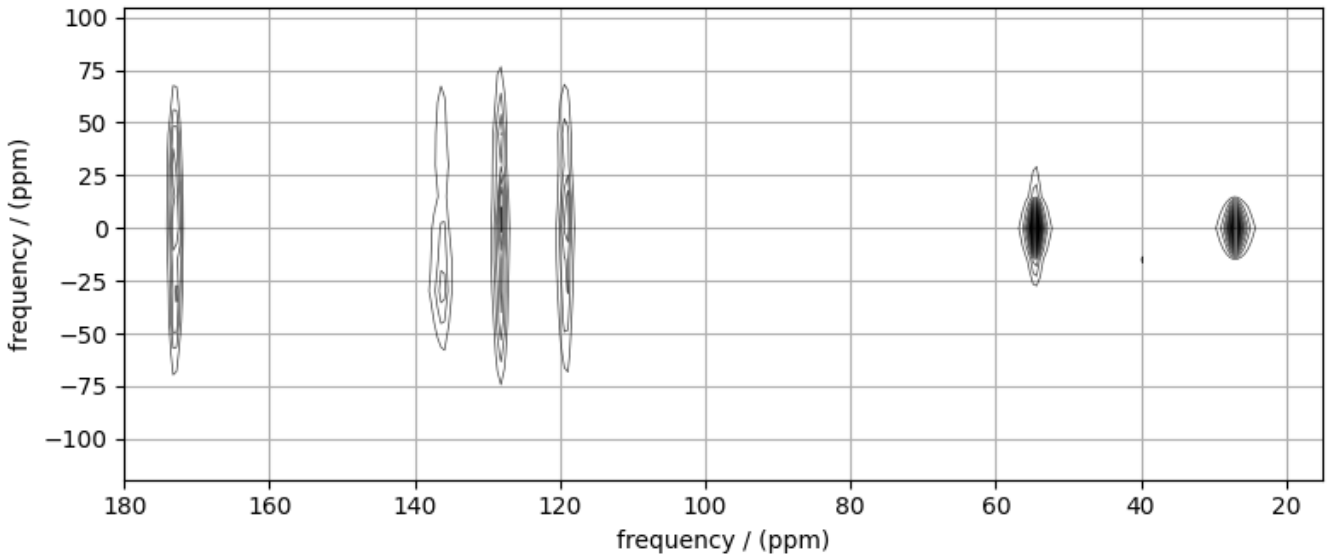
[1] B. J. Walder, K. K. Dey, D. C. Kaseman, J. H. Baltisberger, and P. J. Grandinetti, Sideband separation experiments in NMR with phase incremented echo train acquisition, J. Phys. Chem. 2013, **138**, 174203-1-12. DOI: 10.1063/1.4803142

(continued from previous page)

```
# Convert the coordinates along each dimension from Hz to ppm.
_ = [item.to("ppm", "nmr_frequency_ratio") for item in mat_dataset.dimensions]
```

When using the SSB2D method, ensure the horizontal dimension of the dataset is the isotropic dimension. Here, we apply an appropriate transpose operation to the dataset.

```
mat_dataset = mat_dataset.T  # transpose

# plot of the dataset.
max_amp = mat_dataset.max()
levels = (np.arange(24) + 1) * max_amp / 25  # contours are drawn at these levels.
options = dict(levels=levels, alpha=0.75, linewidths=0.5)  # plot options

plt.figure(figsize=(8, 3.5))
ax = plt.subplot(projection="csdm")
ax.contour(mat_dataset, colors="k", **options)
ax.set_xlim(180, 15)
plt.grid()
plt.tight_layout()
plt.show()
```



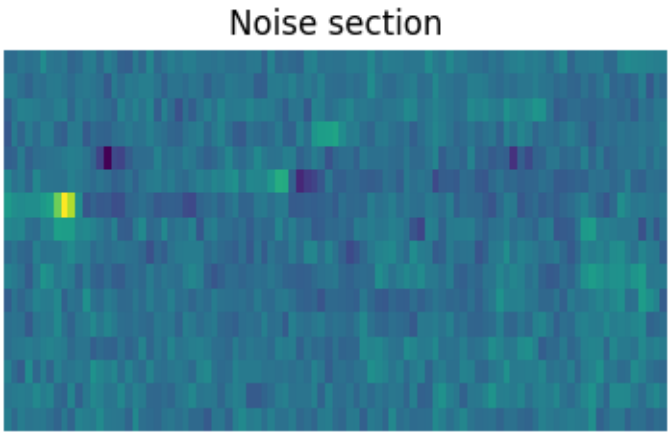
Estimate noise statistics from the dataset

```
coords = mat_dataset.dimensions[0].coordinates
# noise_region = np.where(np.logical_and(coords > 65e-6, coords < 110e-6))
noise_region = np.where(np.logical_and(coords < 110e-6, coords > 65e-6))
noise_data = mat_dataset[noise_region]

plt.figure(figsize=(3.75, 2.5))
ax = plt.subplot(projection="csdm")
ax.imshow(noise_data, aspect="auto", interpolation="none")
plt.title("Noise section")
```

(continues on next page)

(continued from previous page)

```
plt.axis("off")
plt.tight_layout()
plt.show()

noise_mean, sigma = noise_data.mean(), noise_data.std()
noise_mean, sigma
```




```
(<Quantity -0.00846708>, <Quantity 0.5979741>)
```

Create a fitting model

Guess model

Create a guess list of spin systems.

```
shifts = [120, 128, 135, 175, 55, 25]  # in ppm
zeta = [-70, -65, -60, -60, -10, -10]  # in ppm
eta = [0.8, 0.4, 0.9, 0.3, 0.05, 0.05]

spin_systems = single_site_system_generator(
    isotope="13C",
    isotropic_chemical_shift=shifts,
    shielding_symmetric={"zeta": zeta, "eta": eta},
    abundance=100 / 6,
)
```

Method

Create the SSB2D method.

```
# Get the spectral dimension parameters from the experiment.
spectral_dims = get_spectral_dimensions(mat_dataset)

PASS = SSB2D(
    channels=["13C"],
    magnetic_flux_density=9.395,  # in T
```

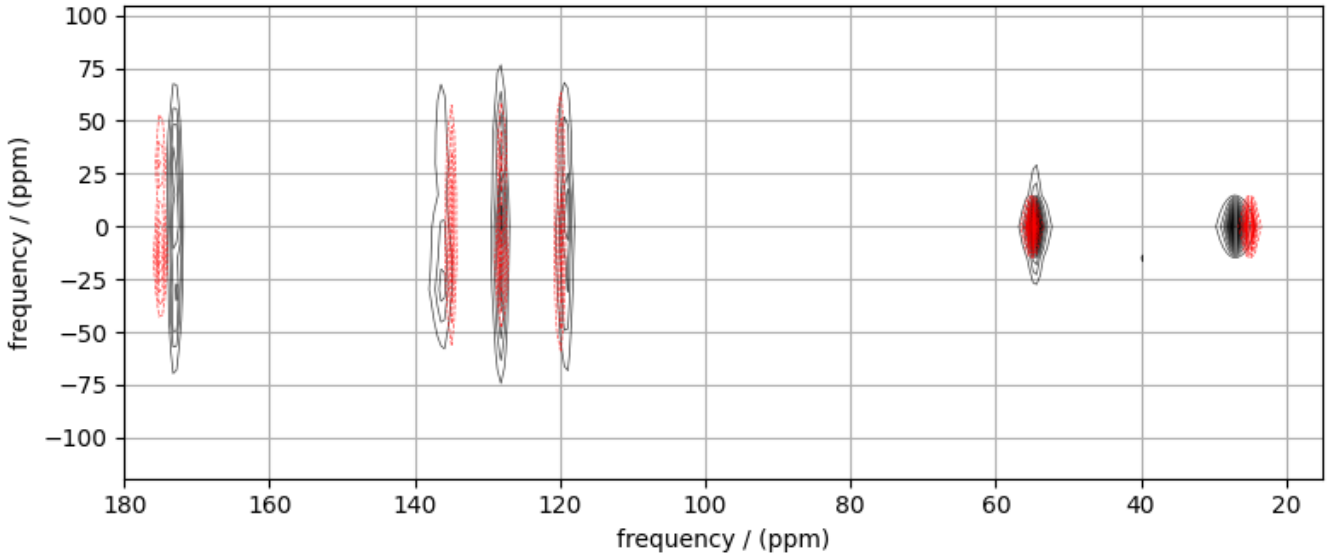
(continues on next page)

(continued from previous page)

```
    rotor_frequency=1500,  # in Hz
    spectral_dimensions=spectral_dims,
    experiment=mat_dataset,  # add the measurement to the method.
)
```

Guess Spectrum

```
# Simulation
# ----------
sim = Simulator(spin_systems=spin_systems, methods=[PASS])
sim.run()

# Post Simulation Processing
# --------------------------
processor = sp.SignalProcessor(
    operations=[
        # Lorentzian convolution along the isotropic dimensions.
        sp.FFT(dim_index=0),
        sp.apodization.Exponential(FWHM="50 Hz"),
        sp.IFFT(dim_index=0),
        sp.Scale(factor=2122600),
    ]
)
processed_dataset = processor.apply_operations(dataset=sim.methods[0].simulation).real

# Plot of the guess Spectrum
# --------------------------
plt.figure(figsize=(8, 3.5))
ax = plt.subplot(projection="csdm")
ax.contour(mat_dataset, colors="k", **options)
ax.contour(processed_dataset, colors="r", linestyles="--", **options)
ax.set_xlim(180, 15)
plt.grid()
plt.tight_layout()
plt.show()
```



Least-squares minimization with LMFIT

Use the *make_LMFIT_params()* (page 497) for a quick setup of the fitting parameters.

```
params = sf.make_LMFIT_params(sim, processor)
print(params.pretty_print(columns=["value", "min", "max", "vary", "expr"]))
```

```
Name                                    Value      Min      Max     Vary     Expr
SP_0_operation_1_Exponential_FWHM          50     -inf      inf     True     None
SP_0_operation_3_Scale_factor         2.123e+06     -inf      inf     True     None
sys_0_abundance                           16.67        0      100     True     None
sys_0_site_0_isotropic_chemical_shift       120     -inf      inf     True     None
sys_0_site_0_shielding_symmetric_eta        0.8        0        1     True     None
sys_0_site_0_shielding_symmetric_zeta       -70     -inf      inf     True     None
sys_1_abundance                           16.67        0      100     True     None
sys_1_site_0_isotropic_chemical_shift       128     -inf      inf     True     None
sys_1_site_0_shielding_symmetric_eta        0.4        0        1     True     None
sys_1_site_0_shielding_symmetric_zeta       -65     -inf      inf     True     None
sys_2_abundance                           16.67        0      100     True     None
sys_2_site_0_isotropic_chemical_shift       135     -inf      inf     True     None
sys_2_site_0_shielding_symmetric_eta        0.9        0        1     True     None
sys_2_site_0_shielding_symmetric_zeta       -60     -inf      inf     True     None
sys_3_abundance                           16.67        0      100     True     None
sys_3_site_0_isotropic_chemical_shift       175     -inf      inf     True     None
sys_3_site_0_shielding_symmetric_eta        0.3        0        1     True     None
sys_3_site_0_shielding_symmetric_zeta       -60     -inf      inf     True     None
sys_4_abundance                           16.67        0      100     True     None
sys_4_site_0_isotropic_chemical_shift        55     -inf      inf     True     None
sys_4_site_0_shielding_symmetric_eta       0.05        0        1     True     None
sys_4_site_0_shielding_symmetric_zeta       -10     -inf      inf     True     None
sys_5_abundance                           16.67        0      100    False 100-sys_0_abundance-
↪sys_1_abundance-sys_2_abundance-sys_3_abundance-sys_4_abundance
```

(continues on next page)

(continued from previous page)

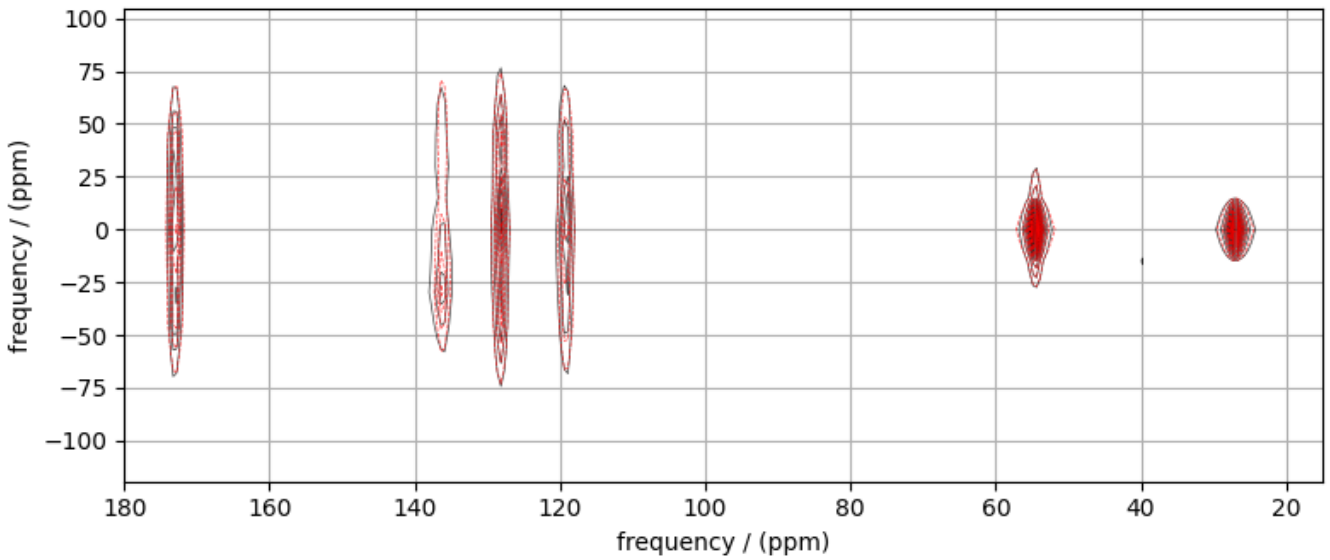
```
sys_5_site_0_isotropic_chemical_shift        25     -inf      inf     True     None
sys_5_site_0_shielding_symmetric_eta       0.05        0        1     True     None
sys_5_site_0_shielding_symmetric_zeta       -10     -inf      inf     True     None
None
```

Solve the minimizer using LMFIT

```
opt = sim.optimize()  # Pre-compute transition pathways
minner = Minimizer(
    sf.LMFIT_min_function,
    params,
    fcn_args=(sim, processor, sigma),
    fcn_kws={"opt": opt},
)
result = minner.minimize()
result
```

The best fit solution

```
best_fit = sf.bestfit(sim, processor)[0].real

# Plot of the best fit solution
plt.figure(figsize=(8, 3.5))
ax = plt.subplot(projection="csdm")
ax.contour(mat_dataset, colors="k", **options)
ax.contour(best_fit, colors="r", linestyles="--", **options)
ax.set_xlim(180, 15)
plt.grid()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 24.190 seconds)

^{87}Rb 2D QMAT NMR of Rb_2SO_4

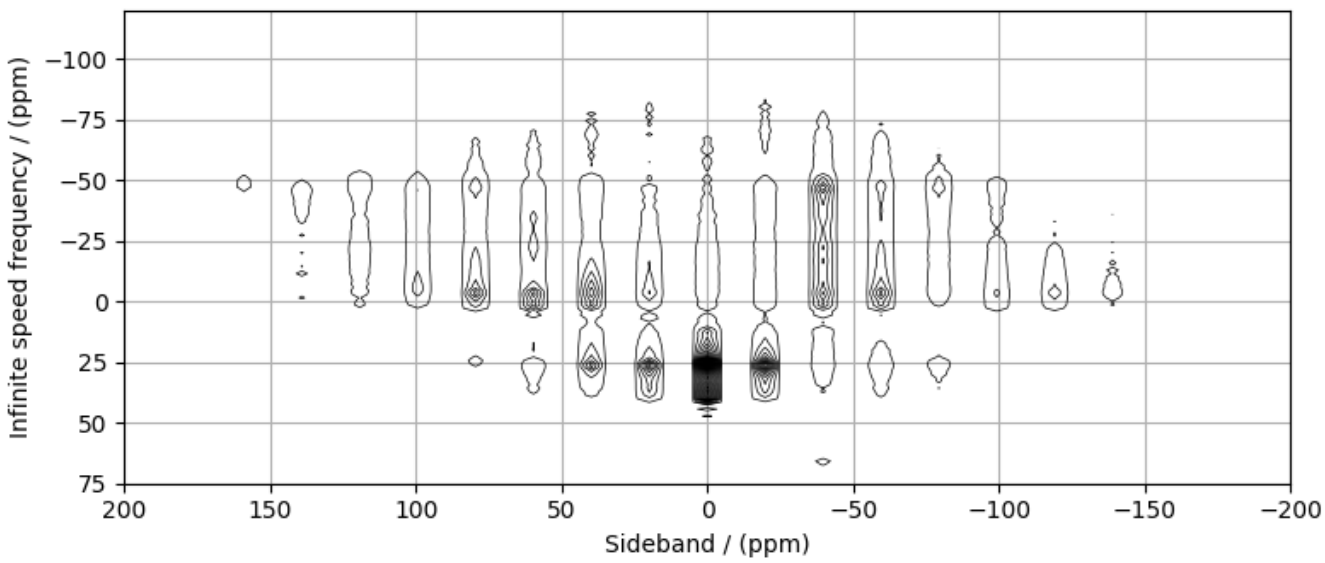
The following is an illustration for fitting 2D QMAT/QPASS datasets. The example dataset is a ^{87}Rb 2D QMAT spectrum of Rb_2SO_4 from Walder *et al.*[1]

```
import numpy as np
import csdmpy as cp
import matplotlib.pyplot as plt
from lmfit import Minimizer

from mrsimulator import Simulator, SpinSystem, Site
from mrsimulator.method.lib import SSB2D
from mrsimulator import signal_processor as sp
from mrsimulator.utils import spectral_fitting as sf
from mrsimulator.utils import get_spectral_dimensions
from mrsimulator.spin_system.tensors import SymmetricTensor
```

Import the dataset

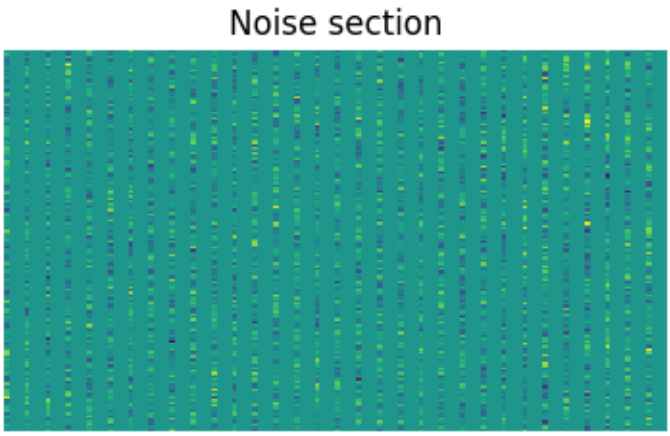
```
filename = "https://ssnmr.org/sites/default/files/mrsimulator/Rb2SO4_QMAT.csdf"
qmat_dataset = cp.load(filename)

# For the spectral fitting, we only focus on the real part of the complex dataset.
qmat_dataset = qmat_dataset.real

# Convert the coordinates along each dimension from Hz to ppm.
_ = [item.to("ppm", "nmr_frequency_ratio") for item in qmat_dataset.dimensions]

# plot of the dataset.
max_amp = qmat_dataset.max()
levels = (np.arange(31) + 0.15) * max_amp / 32  # contours are drawn at these levels.
options = dict(levels=levels, alpha=1, linewidths=0.5)  # plot options

plt.figure(figsize=(8, 3.5))
ax = plt.subplot(projection="csdm")
ax.contour(qmat_dataset.T, colors="k", **options)
ax.set_xlim(200, -200)
ax.set_ylim(75, -120)
plt.grid()
plt.tight_layout()
plt.show()
```

[1] B. J. Walder, K. K. Dey, D. C. Kaseman, J. H. Baltisberger, and P. J. Grandinetti, Sideband separation experiments in NMR with phase incremented echo train acquisition, J. Phys. Chem. 2013, **138**, 174203-1-12. DOI: 10.1063/1.4803142



Estimate noise statistics from the dataset

```
noise_region = np.where(qmat_dataset.dimensions[0].coordinates < -175e-6)
noise_data = qmat_dataset[noise_region]

plt.figure(figsize=(3.75, 2.5))
ax = plt.subplot(projection="csdm")
ax.imshow(noise_data.T, aspect="auto", interpolation="none")
plt.title("Noise section")
plt.axis("off")
plt.tight_layout()
plt.show()

noise_mean, sigma = noise_data.mean(), noise_data.std()
noise_mean, sigma
```




```
(<Quantity 0.02628929>, <Quantity 3.0965025>)
```

Create a fitting model

Guess model

Create a guess list of spin systems.

```
Rb_1 = Site(
    isotope="87Rb",
    isotropic_chemical_shift=16,  # in ppm
    quadrupolar=SymmetricTensor(Cq=5.3e6, eta=0.1),  # Cq in Hz
)
Rb_2 = Site(
    isotope="87Rb",
    isotropic_chemical_shift=40,  # in ppm
    quadrupolar=SymmetricTensor(Cq=2.2e6, eta=0.95),  # Cq in Hz
)

spin_systems = [SpinSystem(sites=[s]) for s in [Rb_1, Rb_2]]
```

Method

Create the SSB2D method.

```
# Get the spectral dimension parameters from the experiment.
spectral_dims = get_spectral_dimensions(qmat_dataset)

PASS = SSB2D(
    channels=["87Rb"],
    magnetic_flux_density=9.395,  # in T
    rotor_frequency=2604,  # in Hz
    spectral_dimensions=spectral_dims,
    experiment=qmat_dataset,  # add the measurement to the method.
)
```

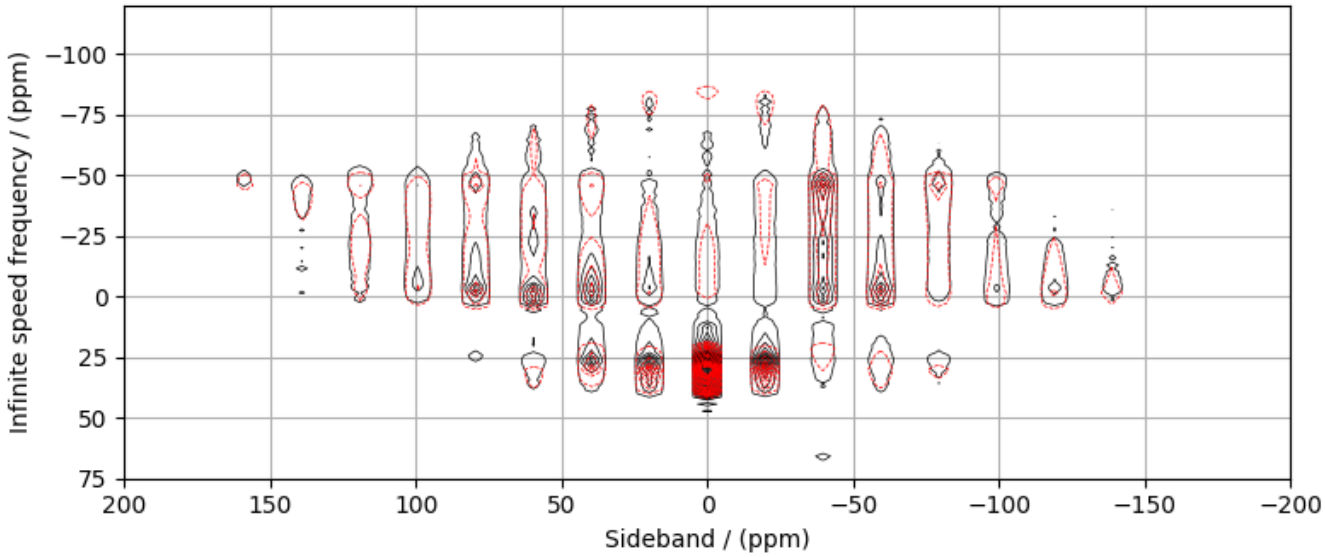
Guess Spectrum

```
# Simulation
# ----------
sim = Simulator(spin_systems=spin_systems, methods=[PASS])
sim.run()

# Post Simulation Processing
# --------------------------
processor = sp.SignalProcessor(
    operations=[
        # Lorentzian convolution along the isotropic dimensions.
        sp.FFT(dim_index=0),
        sp.apodization.Gaussian(FWHM="100 Hz"),
        sp.IFFT(dim_index=0),
        sp.Scale(factor=1e8),
    ]
)
processed_dataset = processor.apply_operations(dataset=sim.methods[0].simulation).real
```

(continues on next page)

(continued from previous page)

```
# Plot of the guess Spectrum
# --------------------------
plt.figure(figsize=(8, 3.5))
ax = plt.subplot(projection="csdm")
ax.contour(qmat_dataset.T, colors="k", **options)
ax.contour(processed_dataset.T, colors="r", linestyles="--", **options)
ax.set_xlim(200, -200)
ax.set_ylim(75, -120)
plt.grid()
plt.tight_layout()
plt.show()
```



Least-squares minimization with LMFIT

Use the *make_LMFIT_params()* (page 497) for a quick setup of the fitting parameters.

```
params = sf.make_LMFIT_params(sim, processor)
params["SP_0_operation_1_Gaussian_FWHM"].min = 0
print(params.pretty_print(columns=["value", "min", "max", "vary", "expr"]))
```

```
Name                                    Value      Min      Max     Vary     Expr
SP_0_operation_1_Gaussian_FWHM            100        0      inf     True     None
SP_0_operation_3_Scale_factor           1e+08     -inf      inf     True     None
sys_0_abundance                            50        0      100     True     None
sys_0_site_0_isotropic_chemical_shift      16     -inf      inf     True     None
sys_0_site_0_quadrupolar_Cq           5.3e+06     -inf      inf     True     None
sys_0_site_0_quadrupolar_eta              0.1        0        1     True     None
sys_1_abundance                            50        0      100    False 100-sys_0_abundance
sys_1_site_0_isotropic_chemical_shift      40     -inf      inf     True     None
sys_1_site_0_quadrupolar_Cq           2.2e+06     -inf      inf     True     None
sys_1_site_0_quadrupolar_eta              0.95        0        1     True     None
```

(continues on next page)

(continued from previous page)

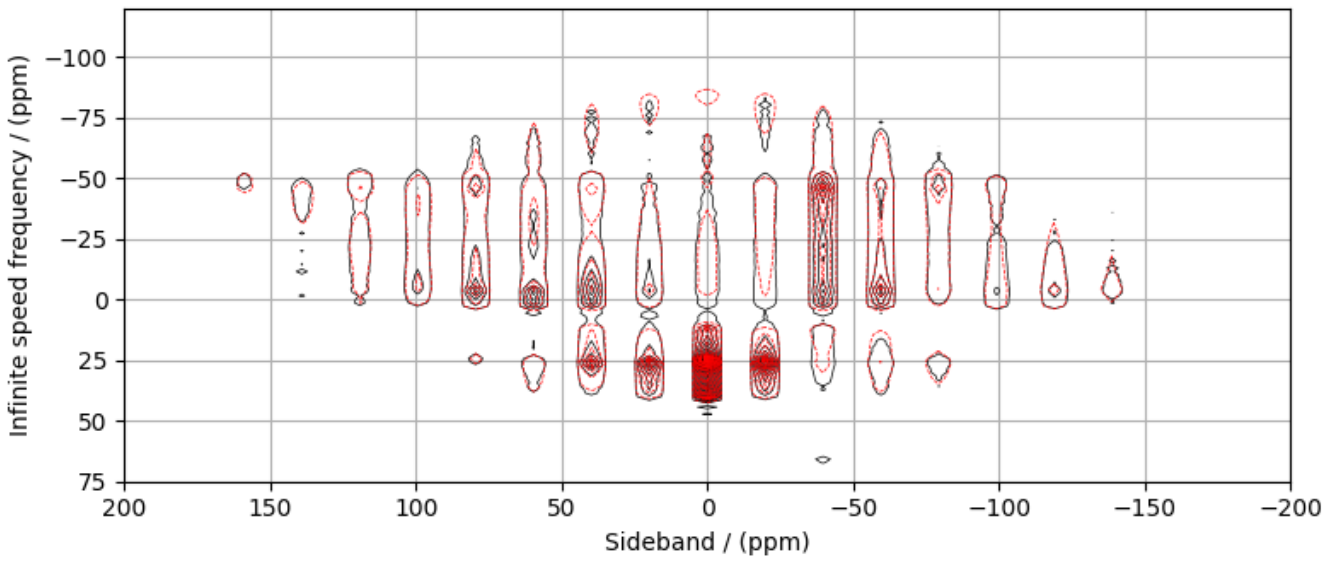
```
None
```

Solve the minimizer using LMFIT

```
opt = sim.optimize()  # Pre-compute transition pathways
minner = Minimizer(
    sf.LMFIT_min_function,
    params,
    fcn_args=(sim, processor, sigma),
    fcn_kws={"opt": opt},
)
result = minner.minimize()
result
```

The best fit solution

```
best_fit = sf.bestfit(sim, processor)[0].real

# Plot of the best fit solution
plt.figure(figsize=(8, 3.5))
ax = plt.subplot(projection="csdm")
ax.contour(qmat_dataset.T, colors="k", **options)
ax.contour(best_fit.T, colors="r", linestyles="--", **options)
ax.set_xlim(200, -200)
ax.set_ylim(75, -120)
plt.grid()
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 11.468 seconds)

^{17}O 2D DAS NMR of Coesite

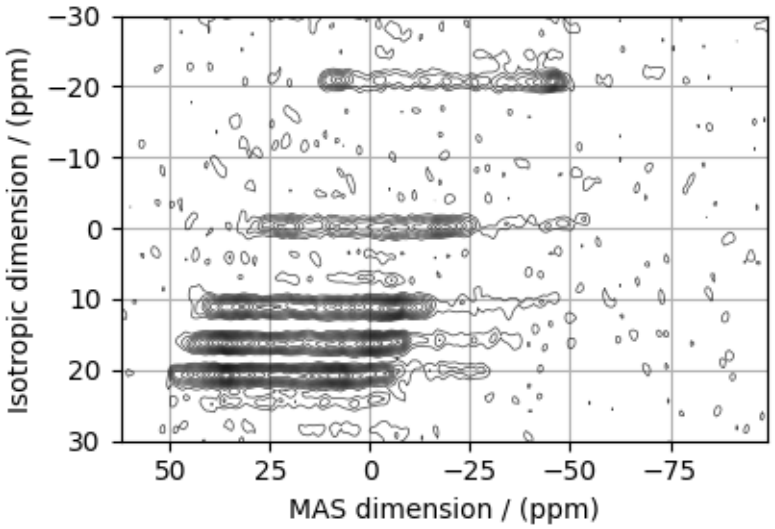
Coesite is a high-pressure (2-3 GPa) and high-temperature (700°C) polymorph of silicon dioxide SiO_2. Coesite has five crystallographic ^{17}O sites. The experimental dataset used in this example is published in Grandinetti *et al.*[1]

```
import numpy as np
import csdmpy as cp
import matplotlib.pyplot as plt
from lmfit import Minimizer

from mrsimulator import Simulator
from mrsimulator import signal_processor as sp
from mrsimulator.utils import spectral_fitting as sf
from mrsimulator.utils import get_spectral_dimensions
from mrsimulator.utils.collection import single_site_system_generator
from mrsimulator.method import Method, SpectralDimension, SpectralEvent, MixingEvent
```

Import the dataset

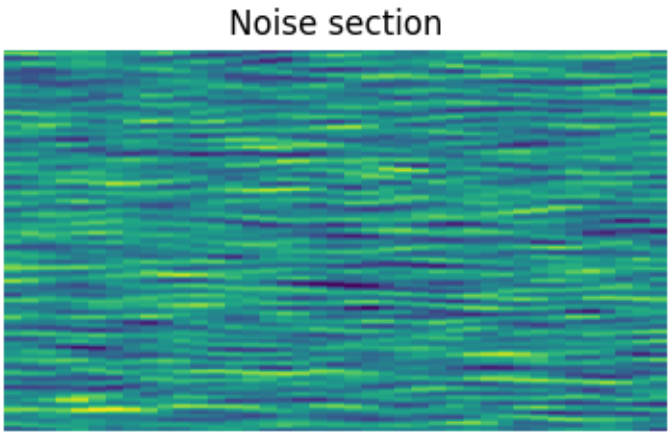
```
filename = "https://ssnmr.org/sites/default/files/mrsimulator/DASCoesite.csdf"
experiment = cp.load(filename)

# 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]

# plot of the dataset.
max_amp = experiment.max()
levels = (np.arange(14) + 1) * max_amp / 15  # contours are drawn at these levels.
options = dict(levels=levels, alpha=0.75, linewidths=0.5)  # plot options

plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.contour(experiment, colors="k", **options)
ax.invert_xaxis()
ax.set_ylim(30, -30)
plt.grid()
plt.tight_layout()
plt.show()
```

[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



Estimate noise statistics from the dataset

```
coords = experiment.dimensions[0].coordinates
noise_region = np.where(coords < -75e-6)
noise_data = experiment[noise_region]

plt.figure(figsize=(3.75, 2.5))
ax = plt.subplot(projection="csdm")
ax.imshow(noise_data, aspect="auto", interpolation="none")
plt.title("Noise section")
plt.axis("off")
plt.tight_layout()
plt.show()

noise_mean, sigma = noise_data.mean(), noise_data.std()
noise_mean, sigma
```




```
(<Quantity -21.397367>, <Quantity 970.9593>)
```

Create a fitting model

Guess model

Create a guess list of spin systems.

```
shifts = [29, 39, 54.8, 51, 56]  # in ppm
Cq = [6.1e6, 5.4e6, 5.5e6, 5.5e6, 5.1e6]  # in  Hz
eta = [0.1, 0.2, 0.15, 0.15, 0.3]
abundance_ratio = [1, 1, 2, 2, 2]
abundance = np.asarray(abundance_ratio) / 8 * 100  # in %

spin_systems = single_site_system_generator(
    isotope="17O",
    isotropic_chemical_shift=shifts,
    quadrupolar={"Cq": Cq, "eta": eta},
    abundance=abundance,
)
```

Method

Create the DAS method.

```
# Get the spectral dimension parameters from the experiment.
spectral_dims = get_spectral_dimensions(experiment)

DAS = Method(
    channels=["17O"],
    magnetic_flux_density=11.744,  # in T
    rotor_frequency=np.inf,
    spectral_dimensions=[
        SpectralDimension(
            **spectral_dims[0],
            events=[
                SpectralEvent(
                    fraction=0.5,
                    rotor_angle=37.38 * np.pi / 180,  # in rads
                    transition_queries=[{"ch1": {"P": [-1], "D": [0]}}],
                ),
                MixingEvent(query="NoMixing"),
                SpectralEvent(
                    fraction=0.5,
                    rotor_angle=79.19 * np.pi / 180,  # in rads
                    transition_queries=[{"ch1": {"P": [-1], "D": [0]}}],
                ),
                MixingEvent(query="NoMixing"),
            ],
        ),
        # The last spectral dimension block is the direct-dimension
        SpectralDimension(
            **spectral_dims[1],
            events=[
                SpectralEvent(
                    rotor_angle=54.735 * np.pi / 180,  # in rads
```

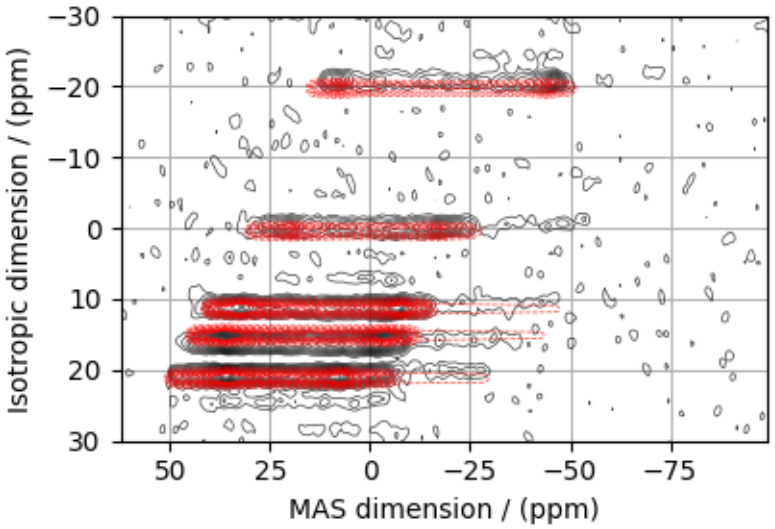
(continues on next page)

(continued from previous page)

```
                    transition_queries=[{"ch1": {"P": [-1], "D": [0]}}],
                )
            ],
        ),
    ],
    experiment=experiment,  # also add the measurement to the method.
)
```

Guess Spectrum

```
# Simulation
# ----------
sim = Simulator(spin_systems=spin_systems, methods=[DAS])
sim.config.number_of_sidebands = 1  # no sidebands are required for this dataset.
sim.run()

# Post Simulation Processing
# --------------------------
processor = sp.SignalProcessor(
    operations=[
        # Gaussian convolution along both dimensions.
        sp.IFFT(dim_index=(0, 1)),
        sp.apodization.Gaussian(FWHM="0.15 kHz", dim_index=0),
        sp.apodization.Gaussian(FWHM="0.1 kHz", dim_index=1),
        sp.FFT(dim_index=(0, 1)),
        sp.Scale(factor=4e8),
    ]
)
processed_dataset = processor.apply_operations(dataset=sim.methods[0].simulation).real

# Plot of the guess Spectrum
# --------------------------
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.contour(experiment, colors="k", **options)
ax.contour(processed_dataset, colors="r", linestyles="--", **options)
ax.invert_xaxis()
ax.set_ylim(30, -30)
plt.grid()
plt.tight_layout()
plt.show()
```



Least-squares minimization with LMFIT

Use the *make_LMFIT_params()* (page 497) for a quick setup of the fitting parameters.

```
params = sf.make_LMFIT_params(sim, processor)
print(params.pretty_print(columns=["value", "min", "max", "vary", "expr"]))
```

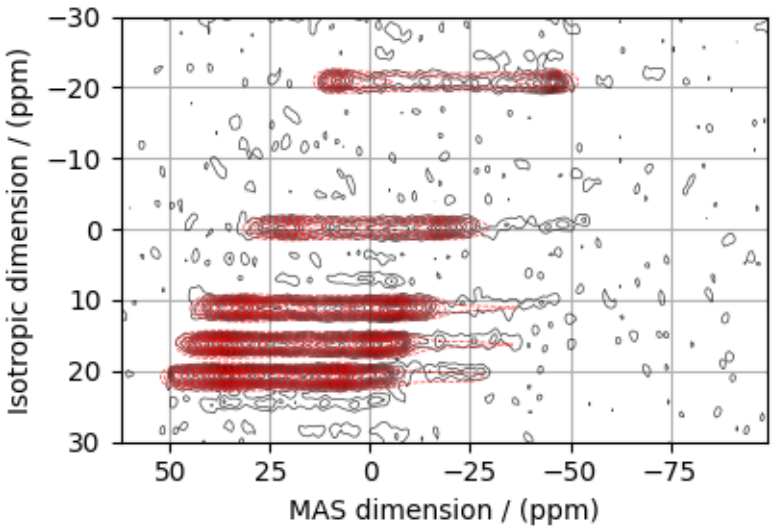
```
Name                                    Value      Min      Max     Vary     Expr
SP_0_operation_1_Gaussian_FWHM           0.15     -inf      inf     True     None
SP_0_operation_2_Gaussian_FWHM            0.1     -inf      inf     True     None
SP_0_operation_4_Scale_factor            4e+08     -inf      inf     True     None
sys_0_abundance                           12.5        0      100     True     None
sys_0_site_0_isotropic_chemical_shift       29     -inf      inf     True     None
sys_0_site_0_quadrupolar_Cq            6.1e+06     -inf      inf     True     None
sys_0_site_0_quadrupolar_eta                0.1        0        1     True     None
sys_1_abundance                           12.5        0      100     True     None
sys_1_site_0_isotropic_chemical_shift       39     -inf      inf     True     None
sys_1_site_0_quadrupolar_Cq            5.4e+06     -inf      inf     True     None
sys_1_site_0_quadrupolar_eta                0.2        0        1     True     None
sys_2_abundance                             25        0      100     True     None
sys_2_site_0_isotropic_chemical_shift     54.8     -inf      inf     True     None
sys_2_site_0_quadrupolar_Cq            5.5e+06     -inf      inf     True     None
sys_2_site_0_quadrupolar_eta               0.15        0        1     True     None
sys_3_abundance                             25        0      100     True     None
sys_3_site_0_isotropic_chemical_shift       51     -inf      inf     True     None
sys_3_site_0_quadrupolar_Cq            5.5e+06     -inf      inf     True     None
sys_3_site_0_quadrupolar_eta               0.15        0        1     True     None
sys_4_abundance                             25        0      100    False 100-sys_0_abundance-
↪sys_1_abundance-sys_2_abundance-sys_3_abundance
sys_4_site_0_isotropic_chemical_shift       56     -inf      inf     True     None
sys_4_site_0_quadrupolar_Cq            5.1e+06     -inf      inf     True     None
sys_4_site_0_quadrupolar_eta                0.3        0        1     True     None
None
```

Solve the minimizer using LMFIT

```
opt = sim.optimize()  # Pre-compute transition pathways
minner = Minimizer(
    sf.LMFIT_min_function,
    params,
    fcn_args=(sim, processor, sigma),
    fcn_kws={"opt": opt},
)
result = minner.minimize(method="powell")
result
```

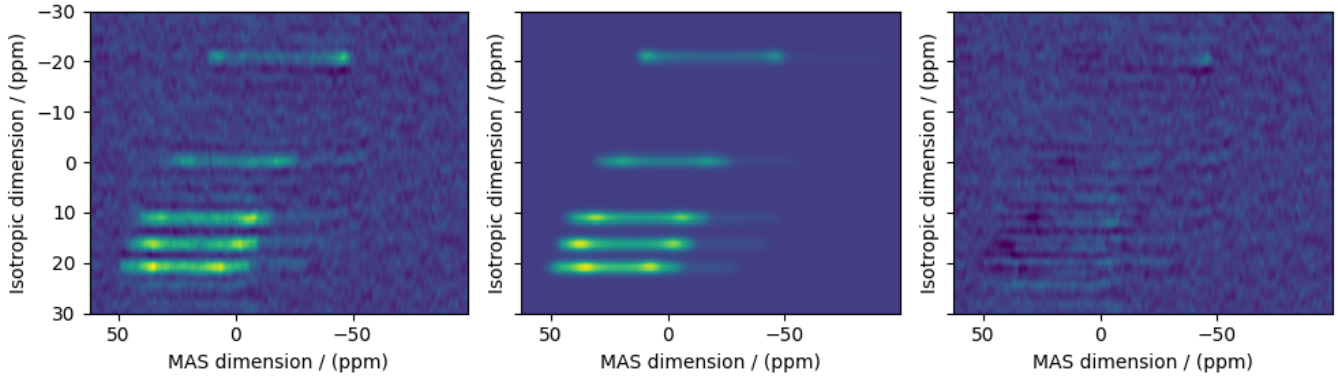
The best fit solution

```
best_fit = sf.bestfit(sim, processor)[0].real

# Plot the spectrum
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.contour(experiment, colors="k", **options)
ax.contour(best_fit, colors="r", linestyles="--", **options)
ax.invert_xaxis()
ax.set_ylim(30, -30)
plt.grid()
plt.tight_layout()
plt.show()
```



The best fit solution

```
residuals = sf.residuals(sim, processor)[0].real

fig, ax = plt.subplots(
    1, 3, sharey=True, figsize=(10, 3.0), subplot_kw={"projection": "csdm"}
)
vmax, vmin = experiment.max(), experiment.min()
for i, dat in enumerate([experiment, best_fit, residuals]):
    ax[i].imshow(dat, aspect="auto", vmax=vmax, vmin=vmin, interpolation="none")
    ax[i].invert_xaxis()
ax[0].set_ylim(30, -30)
plt.tight_layout()
plt.show()
```



Total running time of the script: (1 minutes 16.751 seconds)

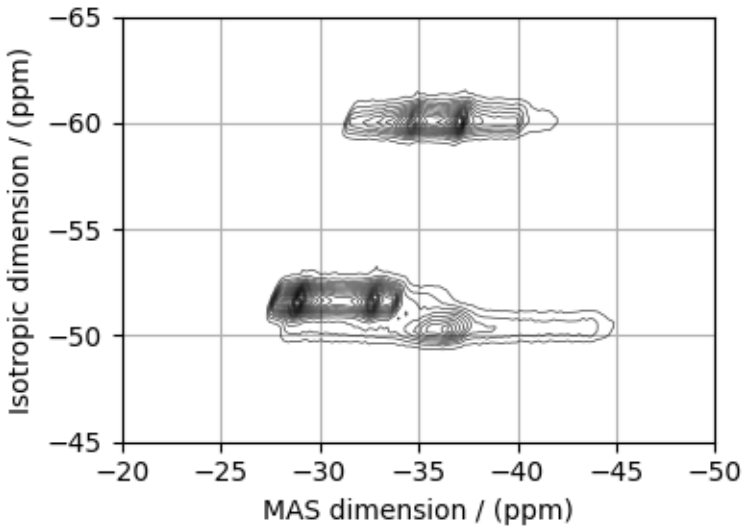
^{87}Rb 2D 3QMAS NMR of $RbNO_3$

The following is a 3QMAS fitting example for $RbNO_3$. The dataset was acquired and shared by Brendan Wilson.

```
import numpy as np
import csdmpy as cp
import matplotlib.pyplot as plt
from lmfit import Minimizer

from mrsimulator import Simulator
from mrsimulator.method.lib import ThreeQ_VAS
from mrsimulator import signal_processor as sp
from mrsimulator.utils import spectral_fitting as sf
from mrsimulator.utils import get_spectral_dimensions
from mrsimulator.utils.collection import single_site_system_generator
```

Import the dataset

```
filename = "https://ssnmr.org/sites/default/files/mrsimulator/RbNO3_MQMAS.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]

# plot of the dataset.
max_amp = experiment.max()
levels = (np.arange(24) + 1) * max_amp / 25  # contours are drawn at these levels.
options = dict(levels=levels, alpha=0.75, linewidths=0.5)  # plot options

plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.contour(experiment, colors="k", **options)
ax.set_xlim(-20, -50)
ax.set_ylim(-45, -65)
plt.grid()
plt.tight_layout()
plt.show()
```



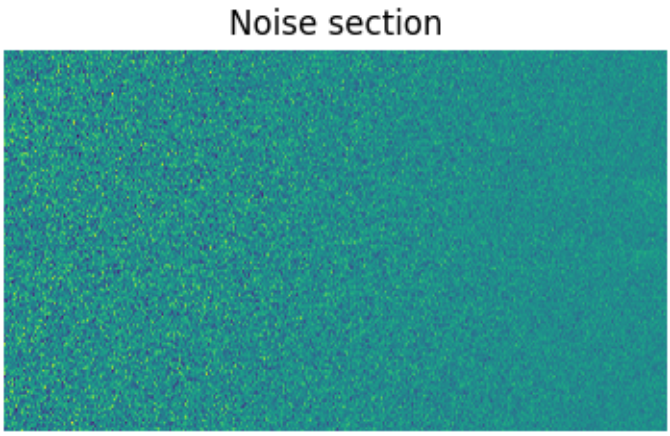
Estimate noise statistics from the dataset.

```
noise_region = np.where(experiment.dimensions[0].coordinates > -25e-6)
noise_data = experiment[noise_region]

plt.figure(figsize=(3.75, 2.5))
ax = plt.subplot(projection="csdm")
ax.imshow(noise_data, aspect="auto", interpolation="none")
plt.title("Noise section")
```

(continues on next page)

(continued from previous page)

```
plt.axis("off")
plt.tight_layout()
plt.show()

noise_mean, sigma = noise_data.mean(), noise_data.std()
noise_mean, sigma
```




```
(<Quantity 1.1995176>, <Quantity 159.38718>)
```

Create a fitting model

Guess model

Create a guess list of spin systems.

```
shifts = [-26.8, -28.4, -31.2]  # in ppm
Cq = [1.7e6, 2.0e6, 1.7e6]  # in  Hz
eta = [0.2, 0.95, 0.6]
abundance = [40.0, 25.0, 35.0]  # in %

spin_systems = single_site_system_generator(
    isotope="87Rb",
    isotropic_chemical_shift=shifts,
    quadrupolar={"Cq": Cq, "eta": eta},
    abundance=abundance,
)
```

Method

Create the 3QMAS method.

```
# Get the spectral dimension parameters from the experiment.
spectral_dims = get_spectral_dimensions(experiment)

MQMAS = ThreeQ_VAS(
    channels=["87Rb"],
```

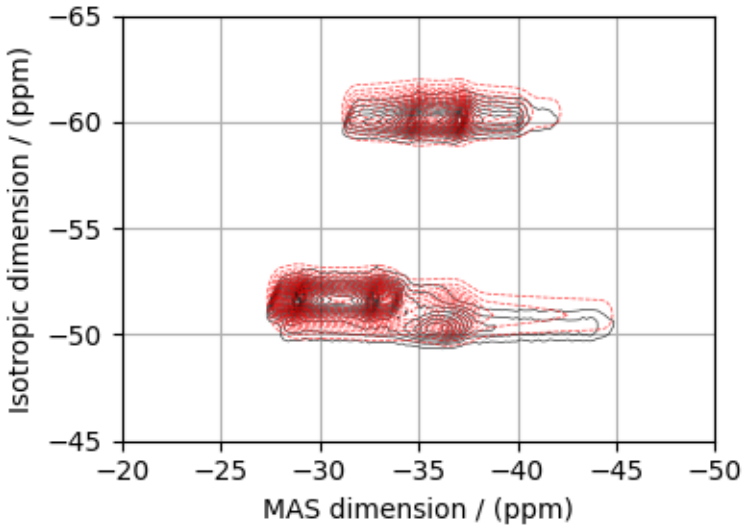
(continues on next page)

(continued from previous page)

```
    magnetic_flux_density=9.395,  # in T
    spectral_dimensions=spectral_dims,
    experiment=experiment,  # add the measurement to the method.
)
```

Guess Spectrum

```
# Simulation
# ----------
sim = Simulator(spin_systems=spin_systems, methods=[MQMAS])
sim.config.number_of_sidebands = 1
sim.run()

# Post Simulation Processing
# --------------------------
processor = sp.SignalProcessor(
    operations=[
        # Gaussian convolution along both dimensions.
        sp.IFFT(dim_index=(0, 1)),
        sp.apodization.Gaussian(FWHM="0.08 kHz", dim_index=0),
        sp.apodization.Gaussian(FWHM="0.2 kHz", dim_index=1),
        sp.FFT(dim_index=(0, 1)),
        sp.Scale(factor=2e8),
    ]
)
processed_dataset = processor.apply_operations(dataset=sim.methods[0].simulation).real

# Plot of the guess Spectrum
# --------------------------
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.contour(experiment, colors="k", **options)
ax.contour(processed_dataset, colors="r", linestyles="--", **options)
ax.set_xlim(-20, -50)
ax.set_ylim(-45, -65)
plt.grid()
plt.tight_layout()
plt.show()
```



Least-squares minimization with LMFIT

Use the *make_LMFIT_params()* (page 497) for a quick setup of the fitting parameters.

```
params = sf.make_LMFIT_params(sim, processor)
print(params.pretty_print(columns=["value", "min", "max", "vary", "expr"]))
```

```
Name                                    Value      Min      Max     Vary     Expr
SP_0_operation_1_Gaussian_FWHM           0.08     -inf      inf     True     None
SP_0_operation_2_Gaussian_FWHM            0.2     -inf      inf     True     None
SP_0_operation_4_Scale_factor            2e+08     -inf      inf     True     None
sys_0_abundance                             40        0      100     True     None
sys_0_site_0_isotropic_chemical_shift    -26.8     -inf      inf     True     None
sys_0_site_0_quadrupolar_Cq            1.7e+06     -inf      inf     True     None
sys_0_site_0_quadrupolar_eta               0.2        0        1     True     None
sys_1_abundance                             25        0      100     True     None
sys_1_site_0_isotropic_chemical_shift    -28.4     -inf      inf     True     None
sys_1_site_0_quadrupolar_Cq              2e+06     -inf      inf     True     None
sys_1_site_0_quadrupolar_eta              0.95        0        1     True     None
sys_2_abundance                             35        0      100    False 100-sys_0_abundance-
↪sys_1_abundance
sys_2_site_0_isotropic_chemical_shift    -31.2     -inf      inf     True     None
sys_2_site_0_quadrupolar_Cq            1.7e+06     -inf      inf     True     None
sys_2_site_0_quadrupolar_eta               0.6        0        1     True     None
None
```

Solve the minimizer using LMFIT

```
opt = sim.optimize()  # Pre-compute transition pathways
minner = Minimizer(
    sf.LMFIT_min_function,
    params,
    fcn_args=(sim, processor, sigma),
```

(continues on next page)

(continued from previous page)

```
    fcn_kws={"opt": opt},
)
result = minner.minimize()
result
```

The best fit solution

```
best_fit = sf.bestfit(sim, processor)[0].real

# Plot the spectrum
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.contour(experiment, colors="k", **options)
ax.contour(best_fit, colors="r", linestyles="--", **options)
ax.set_xlim(-20, -50)
ax.set_ylim(-45, -65)
plt.grid()
plt.tight_layout()
plt.show()
```

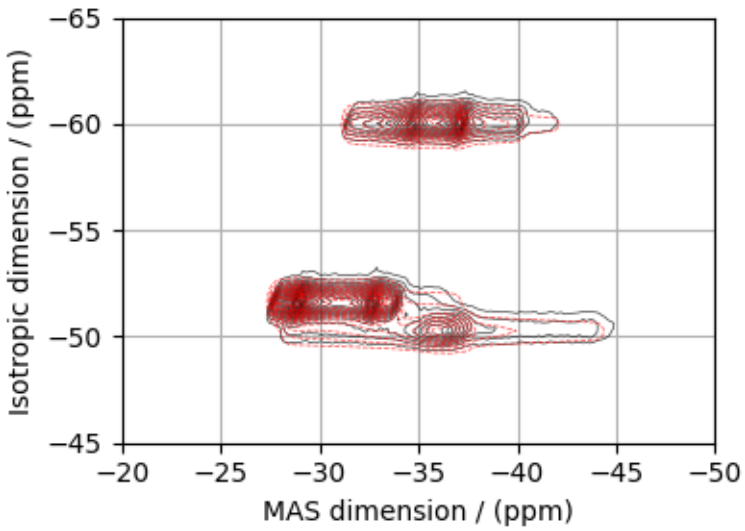


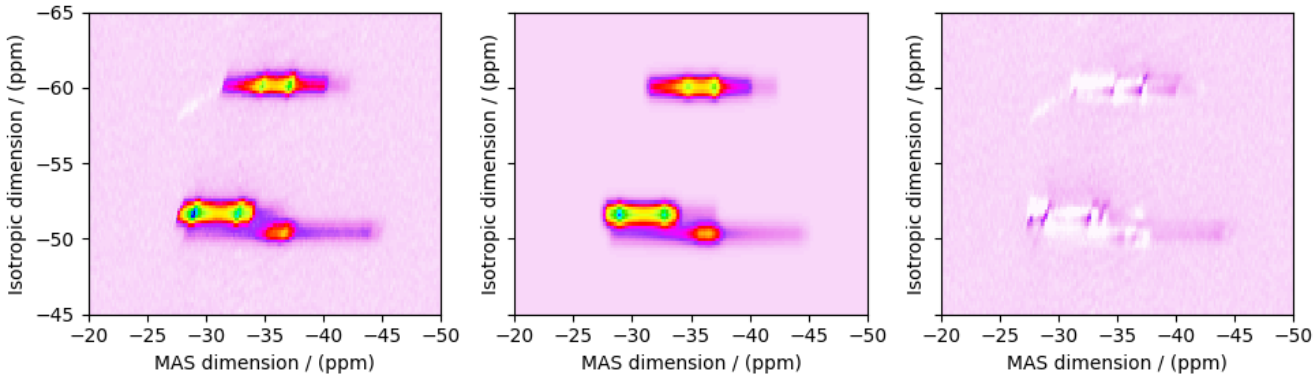
Image plots with residuals

```
residuals = sf.residuals(sim, processor)[0].real

fig, ax = plt.subplots(
    1, 3, sharey=True, figsize=(10, 3.0), subplot_kw={"projection": "csdm"}
)
vmax, vmin = experiment.max(), experiment.min()
for i, dat in enumerate([experiment, best_fit, residuals]):
    ax[i].imshow(
```

(continues on next page)

(continued from previous page)

```
        dat,
        aspect="auto",
        cmap="gist_ncar_r",
        vmax=vmax,
        vmin=vmin,
        interpolation="none",
    )
    ax[i].set_xlim(-20, -50)
ax[0].set_ylim(-45, -65)
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 14.206 seconds)

$NiCl_2.2D_2O$, ^2H (I=1) Shifting-d echo

^2H (I=1) 2D NMR CSA-Quad 1st order correlation spectrum.

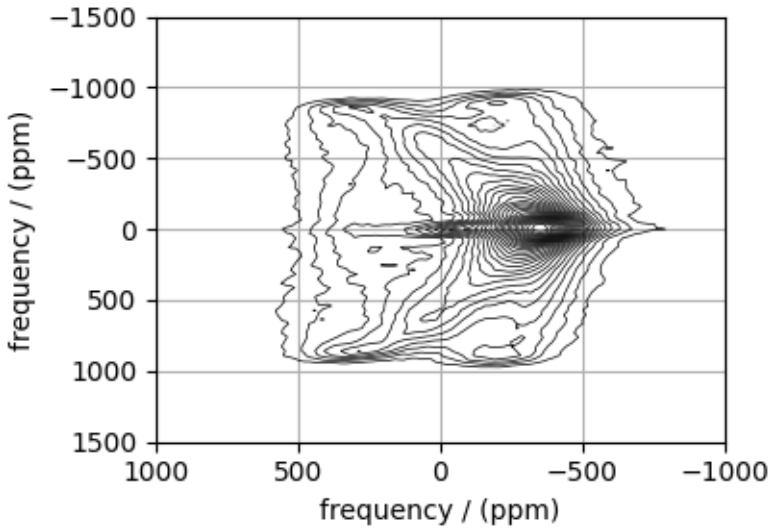
The following is an example of fitting static shifting-d echo NMR correlation spectrum of $NiCl_2 \cdot 2D_2O$ crystalline solid. The spectrum used here is from Walder *et al.*[1].

```
import numpy as np
import csdmpy as cp
import matplotlib.pyplot as plt
from lmfit import Minimizer

from mrsimulator import Simulator, Site, SpinSystem
from mrsimulator import signal_processor as sp
from mrsimulator.utils import spectral_fitting as sf
from mrsimulator.utils import get_spectral_dimensions
from mrsimulator.spin_system.tensors import SymmetricTensor
from mrsimulator.method import Method, SpectralDimension, SpectralEvent, MixingEvent
```

[1] Walder B.J, Patterson A.M., Baltisberger J.H, and Grandinetti P.J Hydrogen motional disorder in crystalline iron group chloride di-hydrates spectroscopy, J. Chem. Phys. (2018) **149**, 084503. DOI: 10.1063/1.5037151

Import the dataset

```
filename = "https://ssnmr.org/sites/default/files/mrsimulator/NiCl2.2D2O.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]

# plot of the dataset.
max_amp = experiment.max()
levels = (np.arange(29) + 1) * max_amp / 30  # contours are drawn at these levels.
options = dict(levels=levels, linewidths=0.5)  # plot options

plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.contour(experiment, colors="k", **options)
ax.set_xlim(1000, -1000)
ax.set_ylim(1500, -1500)
plt.grid()
plt.tight_layout()
plt.show()
```



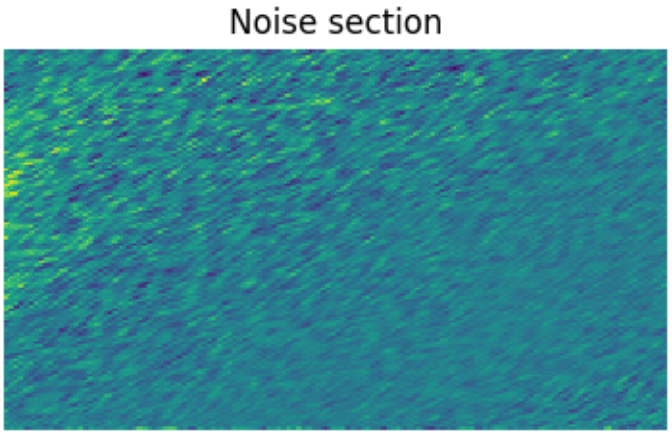
Estimate noise statistics from the dataset

```
coords = experiment.dimensions[0].coordinates
noise_region = np.where(coords > 700e-6)
noise_data = experiment[noise_region]

plt.figure(figsize=(3.75, 2.5))
ax = plt.subplot(projection="csdm")
ax.imshow(noise_data, aspect="auto", interpolation="none")
```

(continues on next page)

(continued from previous page)

```
plt.title("Noise section")
plt.axis("off")
plt.tight_layout()
plt.show()

noise_mean, sigma = experiment[noise_region].mean(), experiment[noise_region].std()
noise_mean, sigma
```




```
(<Quantity 0.64582>, <Quantity 5.835975>)
```

Create a fitting model

Guess model

Create a guess list of spin systems.

```
site = Site(
    isotope="2H",
    isotropic_chemical_shift=-90,  # in ppm
    shielding_symmetric=SymmetricTensor(
        zeta=-610,  # in ppm
        eta=0.15,
        alpha=0.7,  # in rads
        beta=2.0,  # in rads
        gamma=3.0,  # in rads
    ),
    quadrupolar=SymmetricTensor(Cq=75.2e3, eta=0.9),  # Cq in Hz
)

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

Method

Use the generic method, *Method*, to generate a shifting-d echo method. The reported shifting-d 2D sequence is a correlation of the shielding frequencies to the first-order quadrupolar frequencies. Here, we create a correlation method using the `freq_contrib` attribute, which acts as a switch for including the frequency contributions from interaction during the event.

In the following method, we assign the `["Quad1_2"]` and `["Shielding1_0", "Shielding1_2"]` as the value to the `freq_contrib` key. The *Quad1_2* is an enumeration for selecting the first-order second-rank quadrupolar frequency contributions. *Shielding1_0* and *Shielding1_2* are enumerations for the first-order shielding with zeroth and second-rank tensor contributions, respectively. See *FrequencyEnum* (page 416) for details.

```
# Get the spectral dimension parameters from the experiment.
spectral_dims = get_spectral_dimensions(experiment)

shifting_d = Method(
    channels=["2H"],
    magnetic_flux_density=9.395,  # in T
    rotor_frequency=0,  # in Hz
    rotor_angle=0,  # in rads
    spectral_dimensions=[
        SpectralDimension(
            **spectral_dims[0],
            label="Quadrupolar frequency",
            events=[
                SpectralEvent(
                    transition_queries=[{"ch1": {"P": [-1]}}],
                    freq_contrib=["Quad1_2"],
                ),
                MixingEvent(query="NoMixing"),
            ],
        ),
        SpectralDimension(
            **spectral_dims[1],
            label="Paramagnetic shift",
            events=[
                SpectralEvent(
                    transition_queries=[{"ch1": {"P": [-1]}}],
                    freq_contrib=["Shielding1_0", "Shielding1_2"],
                )
            ],
        ),
    ],
    experiment=experiment,  # also add the measurement to the method.
)
```

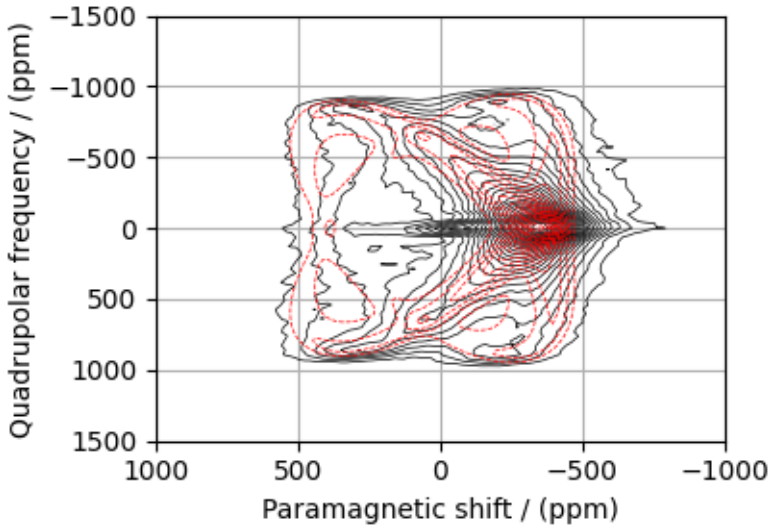
Guess Spectrum

```
# Simulation
# ----------
sim = Simulator(spin_systems=spin_systems, methods=[shifting_d])
sim.config.integration_volume = "hemisphere"
sim.run()

# Post Simulation Processing
# --------------------------
processor = sp.SignalProcessor(
    operations=[
        # Gaussian convolution along both dimensions.
        sp.IFFT(dim_index=(0, 1)),
        sp.apodization.Gaussian(FWHM="5 kHz", dim_index=0),  # along dimension 0
```

(continues on next page)

(continued from previous page)

```
        sp.apodization.Gaussian(FWHM="5 kHz", dim_index=1),  # along dimension 1
        sp.FFT(dim_index=(0, 1)),
        sp.Scale(factor=5e9),
    ]
)
processed_dataset = processor.apply_operations(dataset=sim.methods[0].simulation).real

# Plot of the guess Spectrum
# --------------------------
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.contour(experiment, colors="k", **options)
ax.contour(processed_dataset, colors="r", linestyles="--", **options)
ax.set_xlim(1000, -1000)
ax.set_ylim(1500, -1500)
plt.grid()
plt.tight_layout()
plt.show()
```



Least-squares minimization with LMFIT

Use the *make_LMFIT_params()* (page 497) for a quick setup of the fitting parameters.

```
params = sf.make_LMFIT_params(sim, processor)
print(params.pretty_print(columns=["value", "min", "max", "vary", "expr"]))
```

```
Name                                  Value      Min      Max     Vary     Expr
SP_0_operation_1_Gaussian_FWHM            5     -inf      inf     True     None
SP_0_operation_2_Gaussian_FWHM            5     -inf      inf     True     None
SP_0_operation_4_Scale_factor          5e+09     -inf      inf     True     None
sys_0_abundance                          100        0      100    False      100
sys_0_site_0_isotropic_chemical_shift    -90     -inf      inf     True     None
```

(continues on next page)

(continued from previous page)

```
sys_0_site_0_quadrupolar_Cq             7.52e+04     -inf      inf     True     None
sys_0_site_0_quadrupolar_eta                 0.9        0        1     True     None
sys_0_site_0_shielding_symmetric_alpha       0.7     -inf      inf     True     None
sys_0_site_0_shielding_symmetric_beta          2     -inf      inf     True     None
sys_0_site_0_shielding_symmetric_eta        0.15        0        1     True     None
sys_0_site_0_shielding_symmetric_gamma         3     -inf      inf     True     None
sys_0_site_0_shielding_symmetric_zeta       -610     -inf      inf     True     None
None
```

Solve the minimizer using LMFIT

```
opt = sim.optimize()  # Pre-compute transition pathways
minner = Minimizer(
    sf.LMFIT_min_function,
    params,
    fcn_args=(sim, processor, sigma),
    fcn_kws={"opt": opt},
)
result = minner.minimize()
result
```

The best fit solution

```
best_fit = sf.bestfit(sim, processor)[0].real

# Plot the spectrum
plt.figure(figsize=(4.25, 3.0))
ax = plt.subplot(projection="csdm")
ax.contour(experiment, colors="k", **options)
ax.contour(best_fit, colors="r", linestyles="--", **options)
ax.set_xlim(1000, -1000)
ax.set_ylim(1500, -1500)
plt.grid()
plt.tight_layout()
plt.show()
```

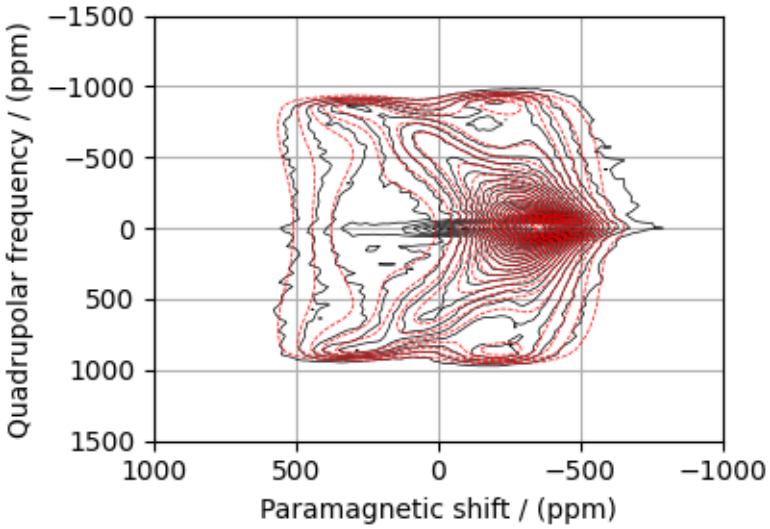
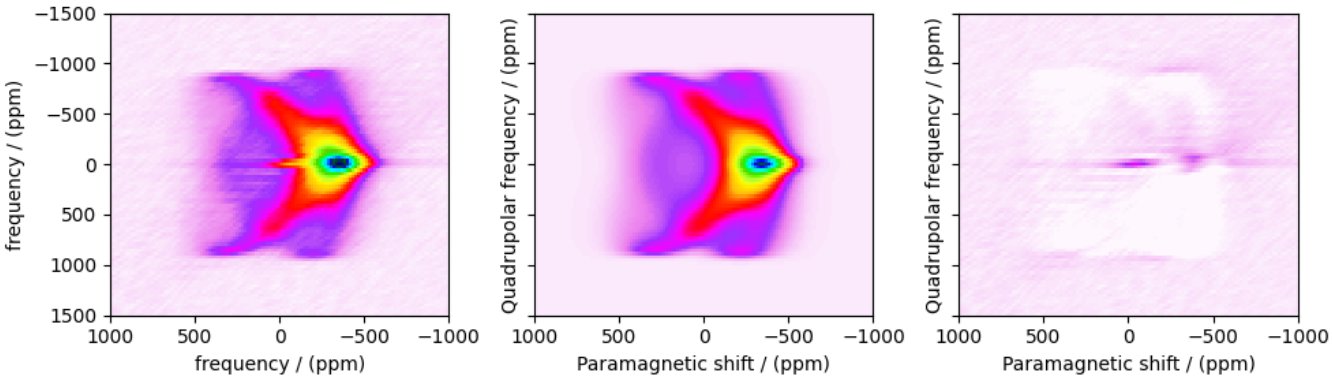


Image plots with residuals

```
residuals = sf.residuals(sim, processor)[0].real

fig, ax = plt.subplots(
    1, 3, sharey=True, figsize=(10, 3.0), subplot_kw={"projection": "csdm"}
)
vmax, vmin = experiment.max(), experiment.min()
for i, dat in enumerate([experiment, best_fit, residuals]):
    ax[i].imshow(
        dat,
        aspect="auto",
        cmap="gist_ncar_r",
        vmax=vmax,
        vmin=vmin,
        interpolation="none",
    )
    ax[i].set_xlim(1000, -1000)
ax[0].set_ylim(1500, -1500)
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 10.746 seconds)

CHAPTER

FOURTEEN

SIGNAL PROCESSING GALLERY

In this gallery, we show examples of processing operations provided in the `mrsimulator.signal_processor` module. For a more in-depth discussion of how signal processing works in mrsimulator, see the *Signal Processor* (page 133) documentation.

14.1 Constant Offset

In this example, we will use the `ConstantOffset` class to offset the baseline of a dataset by a constant.

Below we import the necessary modules

```
import csdmpy as cp
import numpy as np
from mrsimulator import signal_processor as sp
```

First we create `processor`, an instance of the *SignalProcessor* (page 489) class. The required attribute of the SignalProcessor class, *operations*, is a list of operations to which we add a `ConstantOffset` object.

```
processor = sp.SignalProcessor(operations=[sp.baseline.ConstantOffset(offset=0.2)])
```

Next we create a CSDM object with a test dataset which our signal processor will operate on. Here, the dataset spans 500 Hz with a delta function centered at 0 hz

```
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 Hz", complex_fft=True)],
)
```

Now to apply the processor to the CSDM object, use the *apply_operations()* (page 489) method as follows

```
processed_dataset = processor.apply_operations(dataset=csdm_object.copy()).real
```

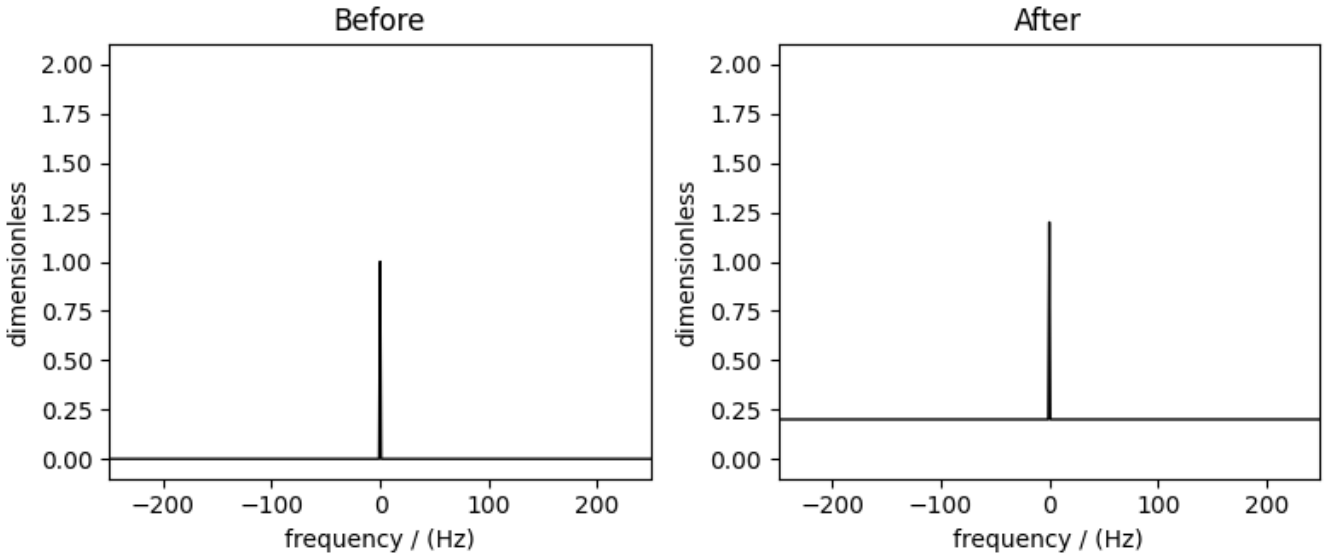
To see the results of the offset, we create a simple plot using the `matplotlib` library.

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(1, 2, figsize=(8, 3.5), subplot_kw={"projection": "csdm"})
ax[0].plot(csdm_object, color="black", linewidth=1)
ax[0].set(ylim=(-0.1, 2.1))
```

(continues on next page)

(continued from previous page)

```
ax[0].set_title("Before")
ax[1].plot(processed_dataset.real, color="black", linewidth=1)
ax[1].set(ylim=(-0.1, 2.1))
ax[1].set_title("After")
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 1.275 seconds)

14.2 Polynomial Offset

In this example, we will use the `Polynomial` class to offset the baseline of a dataset by a polynomial function.

Below we import the necessary modules

```
import csdmpy as cp
import numpy as np
from mrsimulator import signal_processor as sp
```

First we create `processor`, an instance of the *SignalProcessor* (page 489) class. The required attribute of the SignalProcessor class, *operations*, is a list of operations to which we add a `Polynomial` object.

The required argument for the polynomial offset is *polynomial_dictionary* which is a Python dict defining the polynomial coefficients. An arbitrary number of coefficients may be passed.

```
processor = sp.SignalProcessor(
    operations=[
        sp.baseline.Polynomial(polynomial_dictionary={"c0": 0.2, "c2": 0.00001})
    ]
)
```

Here the applied offset will be the following function

$$f(x) = 0.00001 \cdot x^2 + 0.2 \tag{14.1}$$

Next we create a CSDM object with a test dataset which our signal processor will operate on. Here, the dataset spans 500 Hz with a delta function centered at 100 Hz.

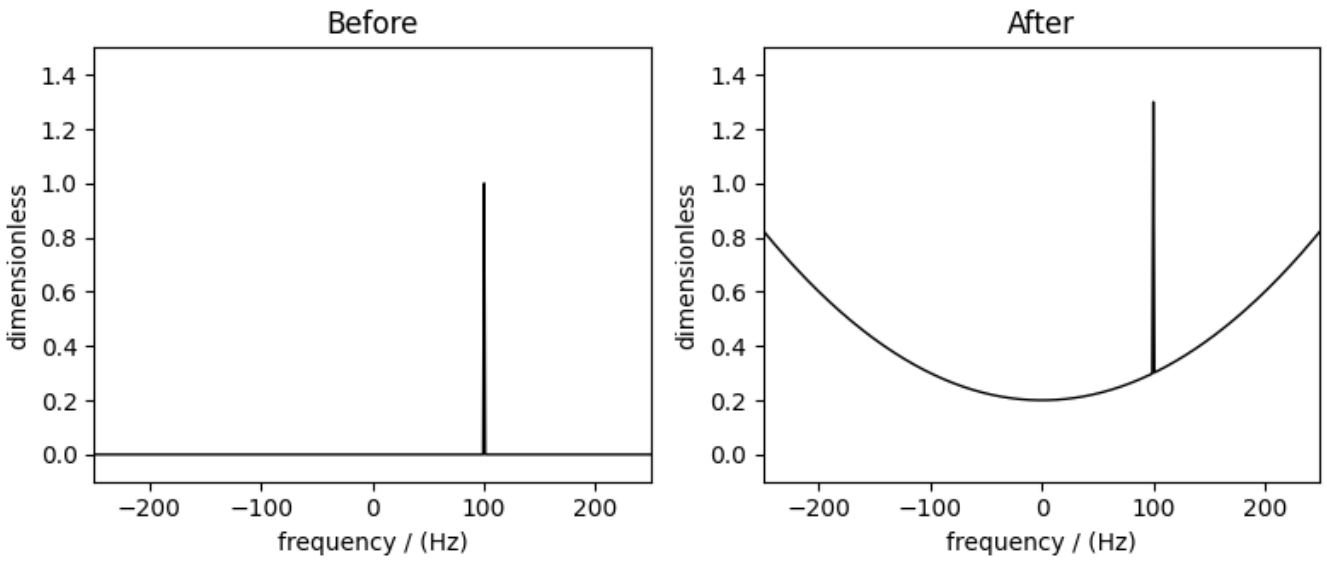
```
test_data = np.zeros(500)
test_data[350] = 1
csdm_object = cp.CSDM(
    dependent_variables=[cp.as_dependent_variable(test_data)],
    dimensions=[cp.LinearDimension(count=500, increment="1 Hz", complex_fft=True)],
)
```

Now to apply the processor to the CSDM object, use the *apply_operations()* (page 489) method as follows

```
processed_dataset = processor.apply_operations(dataset=csdm_object.copy()).real
```

To see the results of the exponential apodization, we create a simple plot using the `matplotlib` library.

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(1, 2, figsize=(8, 3.5), subplot_kw={"projection": "csdm"})
ax[0].plot(csdm_object, color="black", linewidth=1)
ax[0].set(ylim=(-0.1, 1.5))
ax[0].set_title("Before")
ax[1].plot(processed_dataset.real, color="black", linewidth=1)
ax[1].set(ylim=(-0.1, 1.5))
ax[1].set_title("After")
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.444 seconds)

14.3 Exponential Apodization

In this example, we will use an exponential function to perform a Lorentzian convolution to an example dataset. The exponential function used for this apodization is defined as follows

$$f(x) = e^{-\sigma\pi|x|} \tag{14.2}$$

where σ is parametrized by the the full width at half maximum as follows

$$\sigma = \frac{\text{FWHM}}{2\sqrt{2\ln 2}} \tag{14.3}$$

Below we import the necessary modules

```
import csdmpy as cp
import numpy as np
from mrsimulator import signal_processor as sp
```

First we create `processor`, an instance of the *SignalProcessor* (page 489) class. The required attribute of the SignalProcessor class, *operations*, is a list of operations to which we add a `Exponential` object sandwiched between two Fourier transformations.

```
processor = sp.SignalProcessor(
    operations=[
        sp.IFFT(),
        sp.apodization.Exponential(FWHM="75 Hz"),
        sp.FFT(),
    ]
)
```

Next we create a CSDM object with a test dataset which our signal processor will operate on. Here, the dataset spans 500 Hz with a delta function centered at 0 Hz.

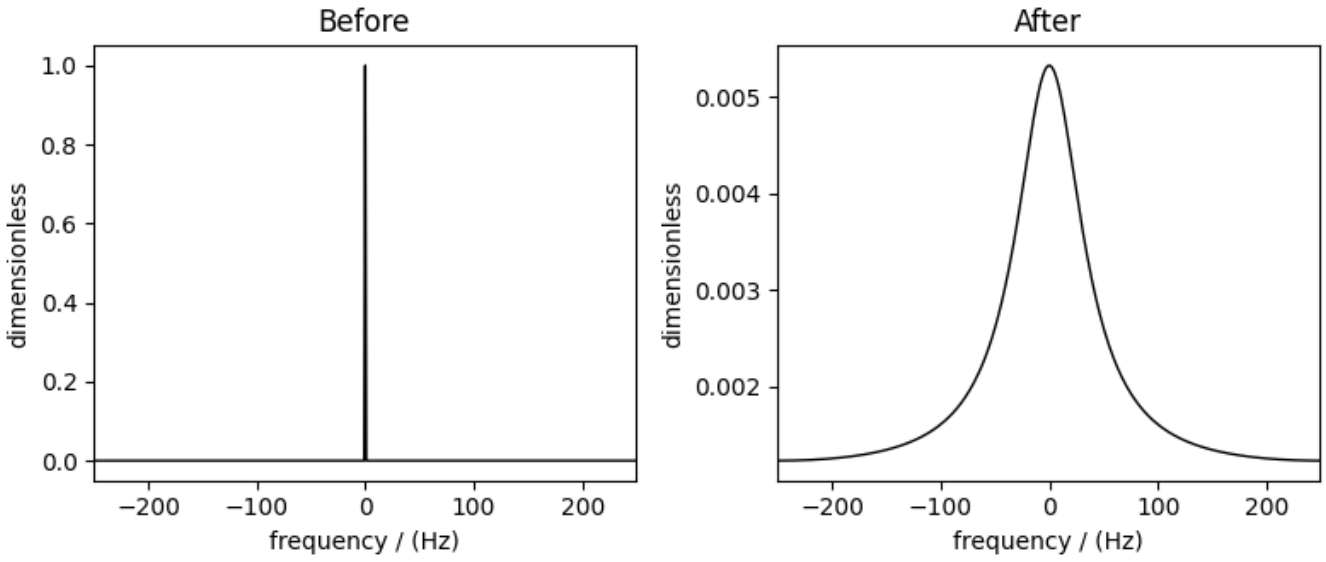
```
test_data = np.zeros(500)
test_data[250] = 1
csdm_object = cp.CSDM(
    dependent_variables=[cp.as_dependent_variable(test_data)],
    dimensions=[cp.LinearDimension(count=500, increment="1 Hz", complex_fft=True)],
)
```

Now to apply the processor to the CSDM object, use the *apply_operations()* (page 489) method as follows

```
processed_dataset = processor.apply_operations(dataset=csdm_object).real
```

To see the results of the exponential apodization, we create a simple plot using the `matplotlib` library.

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(1, 2, figsize=(8, 3.5), subplot_kw={"projection": "csdm"})
ax[0].plot(csdm_object, color="black", linewidth=1)
ax[0].set_title("Before")
ax[1].plot(processed_dataset.real, color="black", linewidth=1)
ax[1].set_title("After")
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.433 seconds)

14.4 Gaussian Apodization

In this example, we will use the `Gaussian` class to perform a Gaussian convolution on an example dataset. The function used for this apodization is defined as follows

$$f(x) = e^{-2\pi^2\sigma^2x^2} \tag{14.4}$$

where σ is the standard deviation of the Gaussian function and is parameterized by the full width as half maximum (FWHM) as

$$\sigma = \frac{\text{FWHM}}{2\sqrt{2\ln 2}} \tag{14.5}$$

Below we import the necessary modules sphinx_gallery_thumbnail_number = 1

```
import csdmpy as cp
import numpy as np
from mrsimulator import signal_processor as sp
```

First we create `processor`, an instance of the *SignalProcessor* (page 489) class. The required attribute of the SignalProcessor class, *operations*, is a list of operations to which we add a `Gaussian` object sandwiched between two Fourier transformations.

```
processor = sp.SignalProcessor(
    operations=[
        sp.IFFT(),
        sp.apodization.Gaussian(FWHM="75 Hz"),
        sp.FFT(),
    ]
)
```

Next we create a CSDM object with a test dataset which our signal processor will operate on. Here, the dataset spans 500 Hz with a delta function centered at 0 Hz.

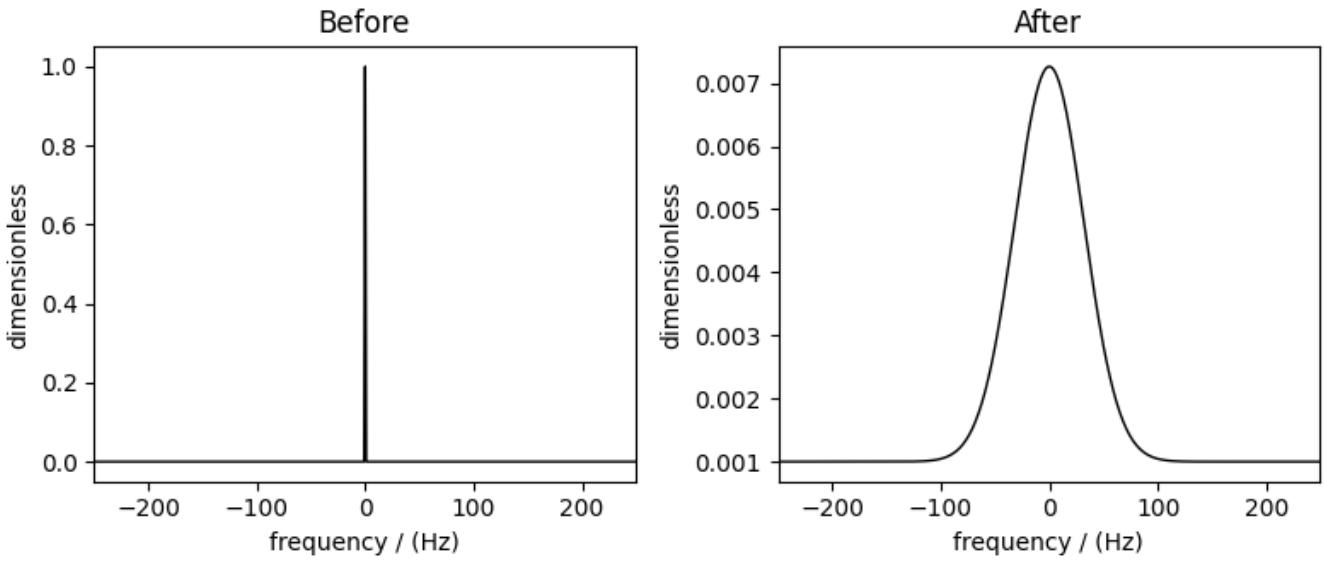
```
test_data = np.zeros(500)
test_data[250] = 1
csdm_object = cp.CSDM(
    dependent_variables=[cp.as_dependent_variable(test_data)],
    dimensions=[cp.LinearDimension(count=500, increment="1 Hz", complex_fft=True)],
)
```

Now to apply the processor to the CSDM object, use the *apply_operations()* (page 489) method as follows

```
processed_dataset = processor.apply_operations(dataset=csdm_object).real
```

To see the results of the Gaussian apodization, we create a simple plot using the `matplotlib` library.

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(1, 2, figsize=(8, 3.5), subplot_kw={"projection": "csdm"})
ax[0].plot(csdm_object, color="black", linewidth=1)
ax[0].set_title("Before")
ax[1].plot(processed_dataset.real, color="black", linewidth=1)
ax[1].set_title("After")
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.447 seconds)

14.5 Top-Hat Apodization

In this example, we will use the `TopHat` class to apply a point-wise top hat apodization on the Fourier transform of an example dataset. The function is defined as follows

$$f(x) = \begin{cases} 1, \texttt{rising\_edge} \leq \texttt{x} \leq \texttt{falling\_edge} \\ 0, \text{otherwise} \end{cases} \tag{14.6}$$

where `rising_edge` is the start of the window and `falling_edge` is the end of the window.

When `falling_edge` is undefined, all points after `rising_edge` will be 1. Similarly, when `rising_edge` is undefined, all points before `falling_edge` are 1.

Below we import the necessary modules

```
import csdmpy as cp
import matplotlib.pyplot as plt
import numpy as np
from mrsimulator import signal_processor as sp
```

First we create `processor`, and instance of the *SignalProcessor* (page 489) class. The required attribute of the SignalProcessor class, *operations*, is a list of operations to which we add a `TopHat` object sandwiched between two Fourier transformations. Here the window is between 1 and 9 seconds.

```
processor = sp.SignalProcessor(
    operations=[
        sp.IFFT(),
        sp.apodization.TopHat(rising_edge="1 s", falling_edge="9 s"),
        sp.FFT(),
    ]
)
```

Next we create a CSDM object with a test dataset which our signal processor will operate on. Here, the dataset is a delta function centered at 0 Hz with a some applied Gaussian line broadening.

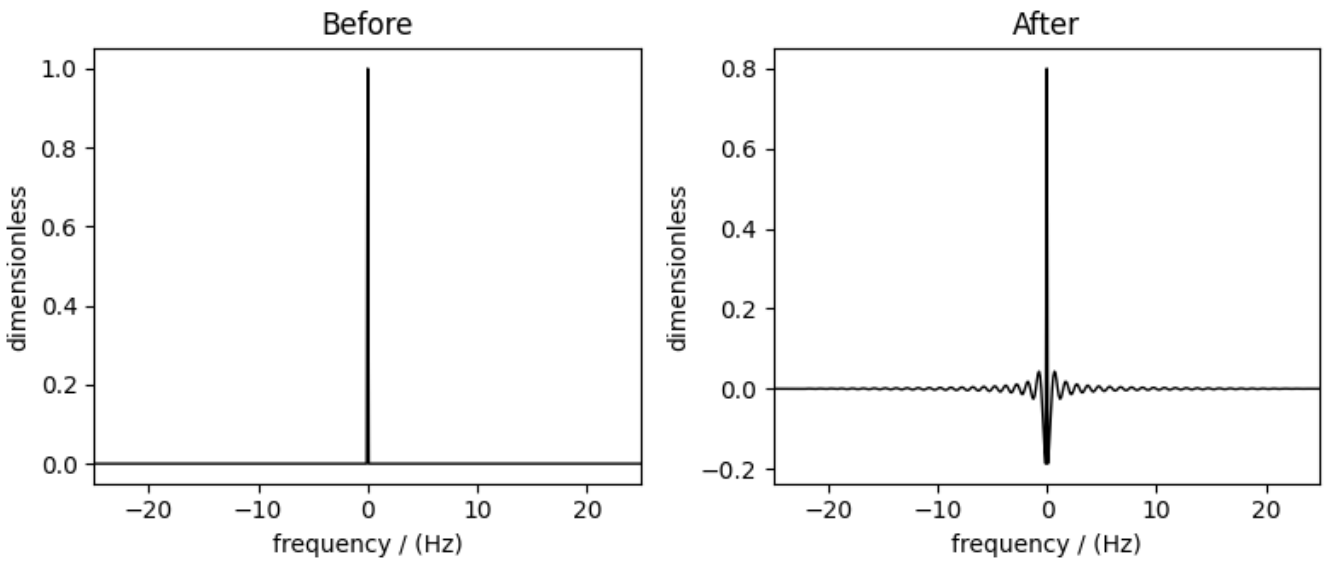
```
test_data = np.zeros(500)
test_data[250] = 1
csdm_object = cp.CSDM(
    dependent_variables=[cp.as_dependent_variable(test_data)],
    dimensions=[cp.LinearDimension(count=500, increment="0.1 Hz", complex_fft=True)],
)
```

To apply the previously defined signal processor, we use the *apply_operations()* (page 489) method as as follows

```
processed_dataset = processor.apply_operations(dataset=csdm_object).real
```

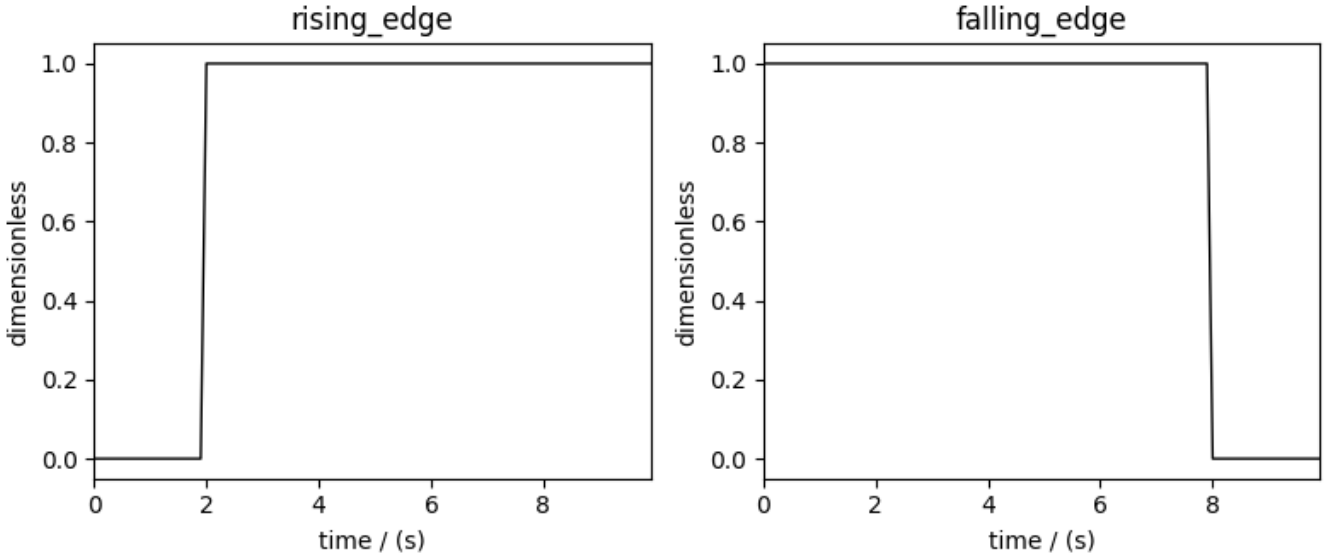
To see the results of the top hat apodization, we create a simple plot using the `matplotlib` library.

```
fig, ax = plt.subplots(1, 2, figsize=(8, 3.5), subplot_kw={"projection": "csdm"})
ax[0].plot(csdm_object, color="black", linewidth=1)
ax[0].set_title("Before")
ax[1].plot(processed_dataset.real, color="black", linewidth=1)
ax[1].set_title("After")
plt.tight_layout()
plt.show()
```



Below are plots showing how the apodization functions when only `rising_edge` or `falling_edge` are defined.

```
rising_edge_processor = sp.SignalProcessor(
    operations=[sp.apodization.TopHat(rising_edge="2 s")]
)
falling_edge_processor = sp.SignalProcessor(
    operations=[sp.apodization.TopHat(falling_edge="8 s")]
)

constant_csdm = cp.CSDM(
    dependent_variables=[cp.as_dependent_variable(np.ones(100))],
    dimensions=[cp.LinearDimension(100, increment="0.1 s")],
)
rising_dataset = rising_edge_processor.apply_operations(
    dataset=constant_csdm.copy()
).real
falling_dataset = falling_edge_processor.apply_operations(
    dataset=constant_csdm.copy()
).real

fig, ax = plt.subplots(1, 2, figsize=(8, 3.5), subplot_kw={"projection": "csdm"})
ax[0].plot(rising_dataset, color="black", linewidth=1)
ax[0].set_title("rising_edge")
ax[1].plot(falling_dataset, color="black", linewidth=1)
ax[1].set_title("falling_edge")
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.827 seconds)

Part V

Theory

CHAPTER

FIFTEEN

TRANSITION FREQUENCY COMPONENTS

The NMR spectral simulation in **mrsimulator** is based on Symmetry Pathways in Solid-State NMR by Grandinetti *et al.*[1]

15.1 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,

$$\Omega(\Theta, i, j) = \sum_k \Omega_k(\Theta, i, j), \tag{15.1}$$

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 written as the product,

$$\Omega_k(\Theta, i, j) = \omega_k \; \Xi_L^{(k)}(\Theta) \; \xi_\ell^{(k)}(i, j), \tag{15.2}$$

where ω_k is the size of the k^{th} frequency component, and $\Xi_L^{(k)}(\Theta)$ and $\xi_\ell^{(k)}(i, j)$ are the sample's spatial orientation and quantized NMR transition functions corresponding to the L^{th} rank spatial and the ℓ^{th} rank spin irreducible spherical tensors, respectively.

The spatial orientation function, $\Xi_L^{(k)}(\Theta)$, in Eq. (15.2), 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,

$$\Xi_L^{(k)}(\Theta) = \sum_{n_0=-L}^{L} D_{n_0,0}^L(\Theta_0) \sum_{n_1=-L}^{L} D_{n_1,n_0}^L(\Theta_1) \; ... \; \sum_{n_i=-L}^{L} D_{n_i,n}^L(\Theta_i) \;\; \varrho_{L,n}^{(k)}. \tag{15.3}$$

Here, the term $D_{n_i,n_j}^L(\Theta)$ is the Wigner rotation matrix element, generically denoted as,

$$D_{n_i,n_j}^L(\Theta) = e^{-in_i\alpha} d_{n_i,n_j}^L(\beta) e^{-in_j\gamma}, \tag{15.4}$$

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 frequency scaled spatial spherical tensor (fsSST) component,

$$\varsigma^{(k)}_{L,n} = \omega_k \varrho^{(k)}_{L,n}, \tag{15.5}$$

of rank L, also defined in the principal axis system of the interaction tensor. Using Eqs. (15.3) and (15.5), we re-express Eq. (15.2) as

$$\Omega_k(\Theta, i, j) = \sum_{n_0=-L}^{L} D^L_{n_0,0}(\Theta_0) \sum_{n_1=-L}^{L} D^L_{n_1,n_0}(\Theta_1) \ ... \ \sum_{n_i=-L}^{L} D^L_{n_i,n}(\Theta_i) \ \ \varpi^{(k)}_{\ell,L,n}, \tag{15.6}$$

where

$$\varpi^{(k)}_{\ell,L,n} = \varsigma^{(k)}_{L,n} \ \ \xi^{(k)}_{\ell}(i,j) \tag{15.7}$$

is the frequency tensor component (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.

15.2 Frequency-scaled spatial spherical tensor (fsSST) components in PAS, $\varsigma^{(k)}_{L,n}$

15.2.1 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,

$$\rho^{(\sigma)}_{0,0} = -\sqrt{3}\sigma_{\text{iso}}, \quad \rho^{(\sigma)}_{2,0} = \sqrt{\tfrac{3}{2}}\zeta_\sigma, \quad \rho^{(\sigma)}_{2,\pm1} = 0, \quad \rho^{(\sigma)}_{2,\pm2} = -\tfrac{1}{2}\eta_\sigma\zeta_\sigma, \tag{15.8}$$

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,

$$\begin{aligned} \varrho^{(\sigma)}_{0,0} &= -\frac{1}{\sqrt{3}}\rho^{(\sigma)}_{0,0} \\ \varrho^{(\sigma)}_{2,n} &= \sqrt{\frac{2}{3}}\rho^{(\sigma)}_{2,n} \end{aligned} \tag{15.9}$$

Table 15.1: A list of scaled spatial orientation tensors in the principal axis system of the nuclear shielding tensor, $\varsigma^{(k)}_{L,n}$ from Eq. (15.5), 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,

$$\rho^{(q)}_{2,0} = \sqrt{\tfrac{3}{2}} \zeta_q, \quad \rho^{(q)}_{2,\pm1} = 0, \quad \rho^{(q)}_{2,\pm2} = -\tfrac{1}{2} \eta_q \zeta_q, \tag{15.10}$$

where ζ_q, and η_q are the efg tensor anisotropy and asymmetry of the site, respectively. The efg anisotropy and asymmetry values are defined using the 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,

$$\varrho^{(q)}_{2,n} = \frac{1}{3\zeta_q} \rho^{(q)}_{2,n}. \tag{15.11}$$

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,

$$\varrho^{(qq)}_{L,n} = \frac{1}{9\zeta_q^2} \sum_{m=-2}^{2} \langle L\ n \mid 2\ 2\ m\ n-m \rangle\, \rho^{(q)}_{2,m}\ \rho^{(q)}_{2,n-m}, \tag{15.12}$$

where $\langle L\ M \mid l_1\ l_2\ m_1\ m_2 \rangle$ is the Clebsch Gordan coefficient.

Table 15.2: A list of scaled spatial orientation tensors in the principal axis system of the efg tensor, $\varsigma^{(k)}_{L,n}$ from Eq. (15.5), 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	$\varsigma^{(qq)}_{4,0} = \frac{\omega_q^2}{\omega_0}\frac{1}{\sqrt{70}}\left(\frac{\eta_q^2}{18}+1\right)$, $\varsigma^{(qq)}_{4,\pm1} = 0$, $\varsigma^{(qq)}_{4,\pm2} = -\frac{\omega_q^2}{\omega_0}\frac{1}{6\sqrt{7}}\eta_q$, $\varsigma^{(qq)}_{4,\pm3} = 0$, $\varsigma^{(qq)}_{4,\pm4} = \frac{\omega_q^2}{\omega_0}\frac{1}{36}\eta_q^2$

15.2.2 Coupled nucleus scaled spatial orientation tensor components

Weak J-coupling interaction

The J-coupling tensor, $\boldsymbol{\rho}^{(J)}$, 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, $\rho^{(J)}_{0,0}$ and the second-rank, $\rho^{(J)}_{2,n}$, irreducible tensors follow,

$$\rho^{(J)}_{0,0} = -\sqrt{3}J_{\text{iso}}, \quad \rho^{(J)}_{2,0} = \sqrt{\tfrac{3}{2}}\zeta_J, \quad \rho^{(J)}_{2,\pm1} = 0, \quad \rho^{(J)}_{2,\pm2} = -\tfrac{1}{2}\eta_J\zeta_J, \tag{15.13}$$

where J_{iso}, ζ_J, and η_J are the isotropic J-coupling, coupling anisotropy and asymmetry parameters, respectively. The J anisotropy and asymmetry are defined using Haeberlen notation.

First-order perturbation

The size of the frequency component from the first-order perturbation expansion of weak J-coupling Hamiltonian is $\omega_k = 2\pi$. The relation between $\varrho^{(J)}_{L,n}$ and $\rho^{(J)}_{L,n}$ follows,

$$\begin{aligned} \varrho^{(J)}_{0,0} &= -\frac{1}{\sqrt{3}}\rho^{(J)}_{0,0} \\ \varrho^{(J)}_{2,n} &= \sqrt{\frac{2}{3}}\rho^{(J)}_{2,n} \end{aligned} \tag{15.14}$$

Table 15.3: A list of scaled spatial orientation tensors in the principal axis system of the J-coupling tensor, $\varsigma^{(k)}_{L,n}$ from Eq. (15.5), of rank L resulting from the Mth order perturbation expansion of the J-coupling Hamiltonian is presented.

Order, M	Rank, L	$\varsigma^{(k)}_{L,n} = \omega_k \varrho^{(k)}_{L,n}$
1	0	$\varsigma^{(J)}_{0,0} = 2\pi J_{\text{iso}}$
1	2	$\varsigma^{(J)}_{2,0} = 2\pi\zeta_J$, $\varsigma^{(J)}_{2,\pm1} = 0$, $\varsigma^{(J)}_{2,\pm2} = -\frac{1}{\sqrt{6}} 2\pi\eta_J\zeta_J$

Weak dipolar-coupling interaction

The dipolar-coupling tensor, $\boldsymbol{\rho}^{(d)}$, is a second rank reducible tensor, which can be decomposed as a second-rank traceless symmetric irreducible spherical tensors. In the principal axis system, the second-rank, $\rho^{(d)}_{2,n}$, irreducible tensors follow,

$$\rho^{(d)}_{2,0} = \sqrt{\tfrac{3}{2}}\zeta_d, \quad \rho^{(d)}_{2,\pm1} = 0, \quad \rho^{(d)}_{2,\pm2} = 0, \tag{15.15}$$

where ζ_d is second-rank symmetric dipolar coupling tensor anisotropy given as

$$\zeta_d = \frac{2}{r^3} \tag{15.16}$$

where r is the distance between two coupled magnetic dipoles. The dipolar splitting is given as

$$\omega_d = -\frac{\mu_0}{4\pi}\frac{\gamma_1\gamma_2\hbar}{r^3} = -\frac{\mu_0}{8\pi}\zeta_d\gamma_1\gamma_2\hbar \tag{15.17}$$

and the dipolar coupling constant, $D = \frac{\omega_d}{2\pi}$.

First-order perturbation

The size of the frequency component from the first-order perturbation expansion of weak J-coupling Hamiltonian is $\omega_k = \frac{2\omega_d}{\zeta_d}$. The relation between $\varrho^{(d)}_{L,n}$ and $\rho^{(d)}_{L,n}$ follows,

$$\varrho^{(d)}_{2,n} = \sqrt{\frac{2}{3}}\rho^{(d)}_{2,n} \tag{15.18}$$

Table 15.4: A list of scaled spatial orientation tensors in the principal axis system of the dipolar-coupling tensor, $\varsigma^{(k)}_{L,n}$ from Eq. (15.5), of rank L resulting from the Mth order perturbation expansion of the dipolar-coupling Hamiltonian is presented.

Order, M	Rank, L	$\varsigma^{(k)}_{L,n} = \omega_k \varrho^{(k)}_{L,n}$
1	2	$\varsigma^{(d)}_{2,0} = 2\omega_d$, $\varsigma^{(d)}_{2,\pm1} = 0$, $\varsigma^{(d)}_{2,\pm2} = 0$

15.3 Spin transition functions, $\xi_\ell^{(k)}(i,j)$

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 is described using the orthogonal rotation subgroups.

15.3.1 Single nucleus spin transition functions

Table 15.5: A list of single nucleus spin transition functions, $\xi_\ell^{(k)}(i,j)$.

$\xi_\ell^{(k)}(i,j)$	Rank, ℓ	Value	Description
$\mathbb{s}(i,j)$	0	0	$\left\langle j\lvert\hat{T}_{00}\rvert j\right\rangle - \left\langle i\lvert\hat{T}_{00}\rvert i\right\rangle$
$\mathbb{p}(i,j)$	1	$j-i$	$\left\langle j\lvert\hat{T}_{10}\rvert j\right\rangle - \left\langle i\lvert\hat{T}_{10}\rvert i\right\rangle$
$\mathbb{d}(i,j)$	2	$\sqrt{\frac{3}{2}}\left(j^2-i^2\right)$	$\left\langle j\lvert\hat{T}_{20}\rvert j\right\rangle - \left\langle i\lvert\hat{T}_{20}\rvert i\right\rangle$
$\mathbb{f}(i,j)$	3	$\frac{1}{\sqrt{10}}[5(j^3-i^3)+(1-3I(I+1))(j-i)]$	$\left\langle j\lvert\hat{T}_{30}\rvert j\right\rangle - \left\langle i\lvert\hat{T}_{30}\rvert i\right\rangle$

Here, $\hat{T}_{\ell,k}(\mathbf{I})$ are the irreducible spherical tensor operators of rank ℓ, $k \in [-\ell,\ell]$, for transition $|i\rangle \rightarrow |j\rangle$. 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 15.6: A list of composite single nucleus spin transition functions, $\xi_\ell^{(k)}(i,j)$. Here, I is the spin quantum number of the nucleus.

$\xi_\ell^{(k)}(i,j)$	Value
$\mathbb{c}_0(i,j)$	$\frac{4}{\sqrt{125}}\left[I(I+1)-\frac{3}{4}\right]\mathbb{p}(i,j) + \sqrt{\frac{18}{25}}\mathbb{f}(i,j)$
$\mathbb{c}_2(i,j)$	$\sqrt{\frac{2}{175}}\left[I(I+1)-\frac{3}{4}\right]\mathbb{p}(i,j) - \frac{6}{\sqrt{35}}\mathbb{f}(i,j)$
$\mathbb{c}_4(i,j)$	$-\sqrt{\frac{18}{875}}\left[I(I+1)-\frac{3}{4}\right]\mathbb{p}(i,j) - \frac{17}{\sqrt{175}}\mathbb{f}(i,j)$

15.3.2 Weakly coupled nucleus spin transition functions

Table 15.7: A list of weakly coupled nucleus spin transition functions, $\xi_\ell^{(k)}(m_{f_I}, m_{f_S}, m_{i_I}, m_{i_S})$.

$\xi_\ell^{(k)}(m_{f_I}, m_{f_S}, m_{i_I}, m_{i_S})$	Value	Description
$(\mathbb{pp})_{IS}(m_{f_I}, m_{f_S}, m_{i_I}, m_{i_S})$	$m_{f_I} m_{f_S} - m_{i_I} m_{i_S}$	$\left\langle m_{f_I} m_{f_S} \lvert \hat{T}_{10}(I)\hat{T}_{10}(S) \rvert m_{f_I} m_{f_S} \right\rangle$ – $\left\langle m_{i_I} m_{i_S} \lvert \hat{T}_{10}(I)\hat{T}_{10}(S) \rvert m_{i_I} m_{i_S} \right\rangle$

Here, $\hat{T}_{\ell,k}(\mathbf{I})$ are the irreducible spherical tensor operators of rank ℓ, $k \in [-\ell, \ell]$, for transition $|m_{i_I} m_{i_S}\rangle \rightarrow |m_{f_I} m_{f_S}\rangle$ in weakly coupled basis.

15.4 Frequency tensor components (FT) in PAS, $\varpi_{\ell,L,n}^{(k)}$

Table 15.8: The table presents a list of frequency tensors defined in the principal axis system of the respective interaction tensor from Eq. (15.7), $\varpi_{\ell,L,n}^{(k)}$, of ranks ℓ and L resulting from the Mth order perturbation expansion of the interaction Hamiltonian supported in mrsimulator.

Interaction	Order, M	Rank, L	$\varpi_{\ell,L,n}^{(k)}$
Nuclear shielding	1	0	$\varpi_{1,0,0}^{(\sigma)} = \varsigma_{0,0}^{(\sigma)} \quad \mathbb{p}(i,j)$
Nuclear shielding	1	2	$\varpi_{1,2,n}^{(\sigma)} = \varsigma_{2,n}^{(\sigma)} \quad \mathbb{p}(i,j)$
Electric Quadrupole	1	2	$\varpi_{2,2,n}^{(q)} = \varsigma_{2,n}^{(q)} \quad \mathbb{d}(i,j)$
Electric Quadrupole	2	0	$\varpi_{c_0,0,0}^{(qq)} = \varsigma_{0,0}^{(qq)} \quad \mathbb{c}_0(i,j)$
Electric Quadrupole	2	2	$\varpi_{c_2,2,n}^{(qq)} = \varsigma_{2,n}^{(qq)} \quad \mathbb{c}_2(i,j)$
Electric Quadrupole	2	4	$\varpi_{c_4,4,n}^{(qq)} = \varsigma_{4,n}^{(qq)} \quad \mathbb{c}_4(i,j)$
Weak J-coupling	1	0	$\varpi_{(1,1),0,0}^{(J)} = \varsigma_{0,0}^{(J)} \quad (\mathbb{pp})_{IS}(m_{f_I}, m_{f_S}, m_{i_I}, m_{i_S})$
Weak J-coupling	1	2	$\varpi_{(1,1),2,n}^{(J)} = \varsigma_{2,n}^{(J)} \quad (\mathbb{pp})_{IS}(m_{f_I}, m_{f_S}, m_{i_I}, m_{i_S})$
Weak dipolar-coupling	1	2	$\varpi_{(1,1),2,n}^{(d)} = \varsigma_{2,n}^{(d)} \quad (\mathbb{pp})_{IS}(m_{f_I}, m_{f_S}, m_{i_I}, m_{i_S})$

References

CHAPTER

SIXTEEN

MODELS

16.1 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, $\mathbf{S}$, in Cartesian basis, follows,

$$\mathbf{S} = \begin{bmatrix} S_{xx} & S_{xy} & S_{xz} \\ S_{xy} & S_{yy} & S_{yz} \\ S_{xz} & S_{yz} & S_{zz} \end{bmatrix}, \tag{16.1}$$

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

$$\begin{aligned} S_{xx} &= \frac{1}{2}(R_{2,2} + R_{2,-2}) - \frac{1}{\sqrt{6}}R_{2,0}, \\ S_{xy} &= S_{yx} = -\frac{i}{2}(R_{2,2} - R_{2,-2}), \\ S_{yy} &= -\frac{1}{2}(R_{2,2} + R_{2,-2}) - \frac{1}{\sqrt{6}}R_{2,0}, \\ S_{xz} &= S_{zx} = -\frac{1}{2}(R_{2,1} - R_{2,-1}), \\ S_{zz} &= \sqrt{\frac{2}{3}}R_{20}, \\ S_{yz} &= S_{zy} = \frac{i}{2}(R_{2,1} + R_{2,-1}). \end{aligned} \tag{16.2}$$

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

[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

[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

the equivalent real tensor components, which are a linear combination of $R_{2,k}$, and are given as

$$\begin{aligned} U_1 &= \frac{1}{\sqrt{6}} R_{2,0}, \\ U_2 &= -\frac{1}{\sqrt{12}} (R_{2,1} - R_{2,-1}), \\ U_3 &= \frac{i}{\sqrt{12}} (R_{2,1} + R_{2,-1}), \\ U_4 &= -\frac{i}{\sqrt{12}} (R_{2,2} - R_{2,-2}), \\ U_5 &= \frac{1}{\sqrt{12}} (R_{2,2} + R_{2,-2}), \end{aligned} \tag{16.3}$$

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

$$\begin{aligned} S_{xx} &= \sqrt{3} U_5 - U_1, \\ S_{xy} &= S_{yx} = \sqrt{3} U_4, \\ S_{yy} &= -\sqrt{3} U_5 - U_1, \\ S_{xz} &= S_{zx} = \sqrt{3} U_2, \\ S_{zz} &= 2 U_1, \\ S_{yz} &= S_{zy} = \sqrt{3} U_3, \end{aligned} \tag{16.4}$$

and the explicit matrix form of $\mathbf{S}$ is

$$\mathbf{S} = \begin{bmatrix} \sqrt{3} U_5 - U_1 & \sqrt{3} U_4 & \sqrt{3} U_2 \\ \sqrt{3} U_4 & -\sqrt{3} U_5 - U_1 & \sqrt{3} U_3 \\ \sqrt{3} U_2 & \sqrt{3} U_3 & 2 U_1 \end{bmatrix}. \tag{16.5}$$

In a shorthand notation, we denote a Czjzek distribution of second-rank traceless symmetric tensor as $S_C(\sigma)$.

16.2 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

$$S_T = S(0) + \rho S_C(\sigma = 1), \tag{16.6}$$

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. In the above equation, the σ parameter from the Czjzek random model, S_C, has no meaning and is set

[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

to one. The factor, ρ, is defined as

$$\rho = \frac{||S(0)||\epsilon}{\sqrt{30}}, \tag{16.7}$$

where $\|S(0)\|$ is the 2-norm of the dominant tensor, and ϵ is a fraction.

Part VI

API and references

CHAPTER

SEVENTEEN

SIMULATION API

17.1 Simulator

class mrsimulator.Simulator(*, name: str = None, description: str = None, label: str = None, property_units: Dict = {}, spin_systems: List[SpinSystem (page 388)] = [], spin_system_models: List[Any] = [], methods: List[Method (page 403)] = [], config: ConfigSimulator (page 386) = ConfigSimulator(name=None, description=None, label=None, property_units={}, number_of_sidebands=64, number_of_gamma_angles=1, integration_volume='octant', integration_density=70, decompose_spectrum='none', isotropic_interpolation='linear', custom_sampling=None))

Bases: Parseable

The simulator class.

spin_systems

A list of *SpinSystem* (page 388) or equivalent dict objects representing a collection of isolated NMR spin systems present within the sample. 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 388) or equivalent dict objects (optional).

methods

A list of *Method* (page 403) or equivalent dict objects representing an NMR methods. The default value is an empty list.

Example

```
>>> from mrsimulator.method.lib import BlochDecaySpectrum
>>> from mrsimulator.method.lib import BlochDecayCTSpectrum
>>> sim.methods = [
...     BlochDecaySpectrum(channels=['17O'], rotor_frequency=15000),
...     BlochDecayCTSpectrum(channels=['17O'])
... ]
```

Type
A list of *Method* (page 403) or equivalent dict objects (optional).

`config`

The *ConfigSimulator* (page 386) 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 *ConfigSimulator* (page 121) for details.

Type
ConfigSimulator (page 386) object or equivalent dict object (optional).

`name`

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 label for the simulation or sample. The default value is None.

Example

```
>>> sim.label = 'Test simulator'
>>> sim.label
'Test simulator'
```

Type
str (optional).

`description`
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).

`export_methods(`*filename: str*`)`
Export a list of methods to a JSON serialized file.

Parameters
`filename` (*str*) – A filename of the serialized file.

Example

```
>>> sim.export_methods(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 *Saving and Loading Spin Systems from a File* (page 138) section.

Parameters

`filename` (*str*) – A filename of the serialized file.

Example

```
>>> sim.export_spin_systems(filename)
```

`get_isotopes`(*spin_I: Optional[float] = None*, *symbol: bool = False*) → list

List of unique isotopes from the sites within the list of the spin systems corresponding to spin quantum number *I*. If *I* is None, a list 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.
- `symbol` (*bool*) – If true, return a list of str with isotope symbols.

Returns

A list of *Isotope* (page 479) objects.

Example

```
>>> sim.get_isotopes()
[Isotope(symbol='13C'), Isotope(symbol='1H'), Isotope(symbol='27Al')]
>>> sim.get_isotopes(symbol=True)
['13C', '1H', '27Al']
```

```
>>> sim.get_isotopes(spin_I=0.5)
[Isotope(symbol='13C'), Isotope(symbol='1H')]
>>> sim.get_isotopes(spin_I=0.5, symbol=True)
['13C', '1H']
```

```
>>> sim.get_isotopes(spin_I=1.5)
[]
```

```
>>> sim.get_isotopes(spin_I=2.5)
[Isotope(symbol='27Al')]
>>> sim.get_isotopes(spin_I=2.5, symbol=True)
['27Al']
```

`json`(*exclude={}*, *units=True*) → dict

Parse the class object to a JSON compliant python dictionary object.

Parameters

- `exclude` – Set of keys that will be excluded from the result.
- `units` – If true, the attribute value is a physical quantity expressed as a string with a number and a unit, else a float.

Returns: dict

`classmethod load`(*filename: str*, *parse_units: bool = True*)

Load the *Simulator* (page 379) 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 Simulator object file.

Returns

A *Simulator* (page 379) object.

Example

```
>>> sim_1 = sim.load('filename')
```

See also:

Saving and Loading Spin Systems from a File (page 138)

`load_methods`(*filename: str*)

Load a list of methods from the given JSON serialized file.

Parameters

`filename` (*str*) – A local or remote address to a JSON serialized file.

Example

```
>>> sim.load_methods(filename)
```

`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 *Saving and Loading Spin Systems from a File* (page 138) section of this documentation.

Parameters

`filename` (*str*) – A local or remote address to a JSON serialized file.

Example

```
>>> sim.load_spin_systems(filename)
```

`optimize()` → None

Pre-computes transition pathways and associated weights for each of the *Method* (page 403) and *SpinSystem* (page 388) objects held by the simulator. This increases the efficiency during least-squared minimization since pathways are not re-computed during every iteration.

Example

```
>>> sim = Simulator()
>>> # Add spin systems and methods
>>> optimization = sim.optimize()
>>> sim.run(opt=optimization)
```

classmethod parse(*py_dict: dict*, *parse_units: bool = True*)

Parse a dictionary for Simulator object.

Parameters

- **py_dict** (*dict*) – Dictionary object.
- **parse_units** (*bool*) – If true, parse quantity from string.

classmethod parse_dict_with_units(*py_dict: 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 379) object.

Example

```
>>> sim_py_dict = {
...     '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'}
...             }]
...         }
...     ]
... }
>>> sim = Simulator.parse_dict_with_units(sim_py_dict)
```

(continues on next page)

(continued from previous page)

```
>>> len(sim.spin_systems)
3
```

reduced_dict(*exclude={}*) → dict

Returns a reduced dictionary representation of the class object by removing all key-value pair corresponding to keys listed in the *exclude* argument, and keys with value as None.

Parameters

`exclude` – A list of keys to exclude from the dictionary.

Return: A dict.

run(*method_index: Optional[list] = None*, *n_jobs: int = 1*, *pack_as_csdm: bool = True*, *opt: Optional[dict] = None*, ***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 all 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 404) attribute of the corresponding method.
- `opt` (*dict*) – An optional optimization dictionary storing pre-computed transition pathways and transition weights for the given methods and spin systems in the simulator. The default is None, that is, the pathways and weights are calculated in the run method.

Example

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

save(*filename: str*, *with_units: bool = 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')
```

sites()

Unique sites within the Simulator object as a list of Site objects.

Returns

A *Sites* (page 474) object.

Example

```
>>> sites = sim.sites()
```

17.2 ConfigSimulator

class mrsimulator.simulator.ConfigSimulator(*, *name: str = None*, *description: str = None*, *label: str = None*, *property_units: Dict = {}*, *number_of_sidebands: ConstrainedIntValue = 64*, *number_of_gamma_angles: ConstrainedIntValue = 1*, *integration_volume: Literal['octant', 'hemisphere', 'sphere'] = 'octant'*, *integration_density: ConstrainedIntValue = 70*, *decompose_spectrum: Literal['none', 'spin_system'] = 'none'*, *isotropic_interpolation: Literal['linear', 'gaussian'] = 'linear'*, *custom_sampling: Optional[CustomSampling] = None*)

Bases: `Parseable`

The configurable attributes for the Simulator class used in simulation.

`number_of_sidebands`

Number of sidebands to evaluate in the simulation. The default value is 64. Value cannot be negative or zero.

Type
int (optional).

`number_of_gamma_angles`

Number of gamma angles averages in the simulation. The default value is 1. Value cannot be negative or zero.

Type
int (optional).

`integration_volume`

The spatial volume over which the spectral frequency integration/averaging is performed. The valid literals of this enumeration are

- `octant` (default),
- `hemisphere`, and
- `sphere`

Type
enum (optional).

`integration_density`

The integration/sampling density or equivalently the number of (alpha, beta) orientations over which the frequency spatial averaging is performed within the given volume. If n is the integration_density, then the total number of orientation is given as

$$n_{\text{octants}} \frac{(n+1)(n+2)}{2} n_\gamma, \tag{17.1}$$

where n_{octants} is the number of octants in the given volume and n_γ is the number of gamma angles. 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).

`isotropic_interpolation`

Interpolation scheme for isotropic binning. The valid literals are

- `linear` (default): linear interpolation.
- `gaussian`: Gaussian interpolation with *sigma=0.25*bin_width*.

Type
enum (optional).

Example

```
>>> a = Simulator()
>>> a.config.number_of_sidebands = 128
>>> a.config.number_of_gamma_angles = 10
>>> a.config.integration_density = 96
>>> a.config.integration_volume = 'hemisphere'
>>> a.config.decompose_spectrum = 'spin_system'
```

`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
```

`json`(*exclude={}*, *units=True*) → dict

Parse the class object to a JSON compliant python dictionary object.

Parameters
- `exclude` – Set of keys that will be excluded from the result.
- `units` – If true, the attribute value is a physical quantity expressed as a string with a number and a unit, else a float.

Returns: dict

classmethod parse_dict_with_units(*json_ dict: dict*)

Parse the physical quantity from a dictionary representation of the class object, where the physical quantity is expressed as a string with a number and a unit.

Parameters

json_dict (*dict*) – A required python dict object.

reduced_dict(*exclude={}*) → dict

Returns a reduced dictionary representation of the class object by removing all key-value pair corresponding to keys listed in the *exclude* argument, and keys with value as None.

Parameters

exclude – A list of keys to exclude from the dictionary.

Return: A dict.

17.3 SpinSystem

class mrsimulator.SpinSystem(**, name: str = None, description: str = None, label: str = None, property_ units: Dict = {'abundance': 'pct'}, sites: Union[List[Site (page 394)], ndarray] = [], couplings: Union[List[Coupling (page 399)], ndarray] = [], abundance: ConstrainedFloatValue = 100.0, transition_ pathways: List = None*)

Bases: Parseable

Base class representing an isolated spin system containing multiple sites and couplings amongst them.

Attribute Documentation

sites

A list of *Site* (page 394) or equivalent dict objects within the spin system. Each site object represents single-site nuclear spin interaction (nuclear shielding and EFG) 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

list of *Site* (page 394) or equivalent dict objects (optional)

couplings

A list of *Coupling* (page 399) or equivalent dict objects within the spin system. Each coupling object represents two-site spin interaction (J-coupling and Dipolar) tensor parameters. The default value is an empty list.

Example

```
>>> sys1 = SpinSystem()
>>> sys1.couplings = [
...     Coupling(site_index=[0, 1], isotropic_j=10.1),
...     Coupling(site_index=[2, 1], dipolar={"D": 1500})
... ]
>>> # or equivalently
>>> sys1.couplings = [
...     {"site_index": [0, 1], "isotropic_j": 10.1},
...     {"site_index": [2, 1], "dipolar": {"D": 1500}}
... ]
```

Type
list of *Coupling* (page 399) 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"
>>> print(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"
>>> print(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"
>>> print(sys1.description)
A test for the spin system
```

Type
str (optional).

`transition_pathways`
A list of *TransitionPathway* (page 482) or equivalent dict objects. Each transition pathway is a list of *Transition* (page 480) 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]}
...     ]
... ]
>>> print(sys1.transition_pathways)
[|2.5, 0.5⟩⟨-2.5, 0.5| ⟶ |-0.5, 0.5⟩⟨0.5, 0.5|, weight=(1+0j)]
```

Note: From any given spin system, the list of relevant transition pathways is determined by the 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 of *TransitionPathway* (page 482) (optional).

`all_transitions()` → TransitionList

Returns a list of all possible spin *Transition* (page 480) objects in the given spin system.

Example

```
>>> 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|]
```

Returns
A list of *Transition* (page 480) objects.

`get_isotopes(`*spin_I: Optional[float] = None*, *symbol: bool = False*`)` → list

An ordered list of *Isotope* (page 479) objects from the sites within the spin system corresponding to the given value of spin quantum number *I*. If *I* is None, a list of all Isotope objects is returned instead.

Parameters
- `spin_I` (*float*) – An optional spin quantum number. The valid inputs are the multiples of 0.5.
- `symbol` (*bool*) – If true, return a list of str with isotope symbols.

Returns
A list of *Isotope* (page 479) objects.

Example

```
>>> spin_systems.get_isotopes() # three spin systems
[Isotope(symbol='13C'), Isotope(symbol='1H'), Isotope(symbol='27Al')]
>>> spin_systems.get_isotopes(symbol=True) # three spin systems
['13C', '1H', '27Al']
```

```
>>> spin_systems.get_isotopes(spin_I=0.5) # isotopes with I=0.5
[Isotope(symbol='13C'), Isotope(symbol='1H')]
>>> spin_systems.get_isotopes(spin_I=0.5, symbol=True) # 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
[Isotope(symbol='27Al')]
>>> spin_systems.get_isotopes(spin_I=2.5, symbol=True) # isotopes with I=2.5
['27Al']
```

`json`(*exclude={}*, *units=True*) → dict

Parse the class object to a JSON compliant python dictionary object.

Parameters

- `exclude` – Set of keys that will be excluded from the result.
- `units` – If true, the attribute value is a physical quantity expressed as a string with a number and a unit, else a float.

Returns: dict

`classmethod parse_dict_with_units`(*py_dict: dict*)

Parse physical quantities 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 388) 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)
```

`reduced_dict(`*exclude={}*`)` → dict

Returns a reduced dictionary representation of the class object by removing all key-value pair corresponding to keys listed in the *exclude* argument, and keys with value as None.

Parameters

`exclude` – A list of keys to exclude from the dictionary.

Return: A dict.

`rotate(`*euler_angles: list*`)` → None

Rotate the spin system coupling tensors and sites by the given list of Euler angle rotations. Euler angles are given as a list of (alpha, beta, gamma) tuples, and rotations happen in the Haeberlen (ZYZ) convention.

Parameters

`euler_angles` (*list*) – An ordered list of angle tuples (alpha, beta, gamma) to rotate through each tensor through.

Example

```
>>> sys = SpinSystem(
...     sites=[Site(isotope="1H"), Site(isotope="13C")],
...     couplings=[Coupling(site_index=[0, 1], dipolar={"D": 3})]
... )
>>> angles = [(3.1415, 0, -3.1415), (1.5701, 1.5701, 1.5701)]
>>> sys.rotate(angles)
```

`simplify()`

Simplifies user-defined spin systems into irreducible spin system objects.

Example

```
>>> sites = [
...     Site(isotope="1H", isotropic_chemical_shift=0, name="A"),
...     Site(isotope="1H", isotropic_chemical_shift=2, name="B"),
...     Site(isotope="1H", isotropic_chemical_shift=4, name="C"),
...     Site(isotope="1H", isotropic_chemical_shift=6, name="D"),
...     Site(isotope="1H", isotropic_chemical_shift=8, name="E"),
...     Site(isotope="1H", isotropic_chemical_shift=10, name="F")
... ]
>>> couplings = [
...     Coupling(site_index=[0, 1], isotropic_j=10, name="AB"),
...     Coupling(site_index=[1, 2], isotropic_j=10, name="BC"),
...     Coupling(site_index=[3, 5], isotropic_j=30, name="DF")
... ]
>>> sys = SpinSystem(sites=sites, couplings=couplings, abundance=30)
>>> simplified_sys = sys.simplify()
>>> simple_sys = [sub_sys.json() for sub_sys in simplified_sys]
>>> pprint(simple_sys)
[{'abundance': '30.0 %',
  'couplings': [{'isotropic_j': '10.0 Hz', 'name': 'AB', 'site_index': [0, 1]},
                {'isotropic_j': '10.0 Hz', 'name': 'BC', 'site_index': [1, 2]}],
  'sites': [{'isotope': '1H',
```

(continues on next page)

(continued from previous page)

```
             'isotropic_chemical_shift': '0.0 ppm',
             'name': 'A'},
            {'isotope': '1H',
             'isotropic_chemical_shift': '2.0 ppm',
             'name': 'B'},
            {'isotope': '1H',
             'isotropic_chemical_shift': '4.0 ppm',
             'name': 'C'}]},
 {'abundance': '30.0 %',
  'couplings': [{'isotropic_j': '30.0 Hz', 'name': 'DF', 'site_index': [0, 1]}],
  'sites': [{'isotope': '1H',
             'isotropic_chemical_shift': '6.0 ppm',
             'name': 'D'},
            {'isotope': '1H',
             'isotropic_chemical_shift': '10.0 ppm',
             'name': 'F'}]},
 {'abundance': '30.0 %',
  'sites': [{'isotope': '1H',
             'isotropic_chemical_shift': '8.0 ppm',
             'name': 'E'}]}]
```

`zeeman_energy_states()` → list

Return a list of all *ZeemanState* (page 475) objects of the spin system, where the energy states are represented by a list of quantum numbers,

$$|\Psi\rangle = [m_1, m_2, ..m_n], \tag{17.2}$$

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.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 475) objects.

17.4 Site

`class mrsimulator.Site(*, name: str = None, description: str = None, label: str = None, property_units: Dict = {'isotropic_chemical_shift': 'ppm'}, isotope: Union[str, dict, Isotope (page 479)] = '1H', isotropic_chemical_shift: float = 0.0, shielding_symmetric: SymmetricTensor (page 475) = None, shielding_antisymmetric: AntisymmetricTensor (page 478) = None, quadrupolar: SymmetricTensor (page 475) = None)`

Bases: `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 isotropic chemical shift of the site in ppm. The default value is 0.

Example

```
>>> site.isotropic_chemical_shift = 43.3
```

Type
float (optional).

`shielding_symmetric`

The attribute represents the parameters of the irreducible second-rank traceless symmetric part of the nuclear shielding tensor. The default value is None.

The allowed attributes of the *SymmetricTensor* (page 475) 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 475) or equivalent dict object (optional).

`shielding_antisymmetric`

The attribute represents the parameters of the irreducible first-rank antisymmetric part of the nuclear shielding tensor. The default value is None.

The allowed attributes of the *AntisymmetricTensor* (page 478) class for *shielding_antisymmetric* are `zeta`, `alpha`, and `beta`, where `zeta` is the anisotropy parameter, in ppm, of the anti-symmetric first-rank tensor. 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 478) or equivalent dict object (optional).

`quadrupolar`

The attribute represents the parameters of the traceless irreducible second-rank symmetric part of the electric-field gradient tensor. The default value is None.

The allowed attributes of the *SymmetricTensor* (page 475) 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 475) or equivalent dict object (optional).

`name`

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 label for the site. The default value is None.

Example

```
>>> site.label = 'Quad site'
>>> site.label
'Quad site'
```

Type
str (optional).

`description`
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 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),
... )
```

`json`(*exclude={}*, *units=True*) → dict
Parse the class object to a JSON compliant python dictionary object.

Parameters
- `exclude` – Set of keys that will be excluded from the result.

- `units` – If true, the attribute value is a physical quantity expressed as a string with a number and a unit, else a float.

Returns: dict

`classmethod parse_dict_with_units(`*py_dict: 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 394) 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)
```

`reduced_dict(`*exclude={}*`)` → dict

Returns a reduced dictionary representation of the class object by removing all key-value pair corresponding to keys listed in the *exclude* argument, and keys with value as None.

Parameters
`exclude` – A list of keys to exclude from the dictionary.

Return: A dict.

`rotate(`*euler_angles: list*`)` → None

Rotate the site tensors (shielding, quadrupolar) by the given list of Euler angle rotations. Euler angles are given as a list of (alpha, beta, gamma) tuples, and rotations happen in the Haeberlen (ZYZ) convention.

Parameters
`euler_angles` (*(list)*) – An ordered list of angle tuples (alpha, beta, gamma) to rotate through each tensor through.

Example

```
>>> site = Site(isotope="13C", shielding_symmetric={"zeta": 5, "eta": 0.2})
>>> angles = [(3.1415, 0, -3.1415), (1.5701, 1.5701, 1.5701)]
>>> site.rotate(angles)
```

17.5 Coupling

class mrsimulator.Coupling(*, *name: str = None*, *description: str = None*, *label: str = None*, *property_units: Dict = {'isotropic_j': 'Hz'}*, *site_index: List[int]*, *isotropic_j: float = 0.0*, *j_symmetric:* SymmetricTensor *(page 475) = None*, *j_antisymmetric:* AntisymmetricTensor *(page 478) = None*, *dipolar:* SymmetricTensor *(page 475) = None*)

Bases: `Parseable`

Base class representing a two-site coupled nuclear spin interaction tensor parameters, which include the J-coupling and dipolar tensor.

Attribute Documentation

`site_index`

A list of two integers, each corresponding to the index of the coupled sites.

Example

```
>>> coupling = Coupling(site_index=[0, 1])
```

Type
list of int (required).

`isotropic_j`

The isotropic j-coupling, in Hz, between the coupled sites. The default is 0.

Example

```
>>> coupling.isotropic_j = 43.3
```

Type
float (optional).

`j_symmetric`

The attribute represents the parameters of the irreducible second-rank traceless symmetric part of the J-coupling tensor. The default value is None.

The allowed attributes of the *SymmetricTensor* (page 475) class for *j_symmetric* are `zeta`, `eta`, `alpha`, `beta`, and `gamma`, where `zeta` is the J anisotropy, in Hz, and `eta` is the J asymmetry parameter defined using the Haeberlen convention. The Euler angles `alpha`, `beta`, and `gamma` are in radians.

Example

```
>>> coupling.j_symmetric = {'zeta': 10, 'eta': 0.5}
```

```
>>> # or equivalently
>>> coupling.j_symmetric = SymmetricTensor(zeta=10, eta=0.5)
```

Type
SymmetricTensor (page 475) or equivalent dict object (optional).

`j_antisymmetric`

The attribute represents the parameters of the irreducible first-rank antisymmetric part of the J tensor. The default value is None.

The allowed attributes of the *AntisymmetricTensor* (page 478) class for *j_ antisymmetric* are `zeta`, `alpha`, and `beta`, where `zeta` is the anisotropy parameter of the anti-symmetric first-rank tensor given in Hz. The angles `alpha` and `beta` are in radians.

Example

```
>>> coupling.j_antisymmetric = {'zeta': 20}
```

```
>>> # or equivalently
>>> coupling.j_antisymmetric = AntisymmetricTensor(zeta=20)
```

Type
AntisymmetricTensor (page 478) or equivalent dict object (optional).

`dipolar`

The attribute represents the parameters of the irreducible second-rank traceless symmetric part of the direct-dipolar coupling tensor. The default value is None.

The allowed attributes of the *SymmetricTensor* (page 475) class for *dipolar* are `D`, `alpha`, `beta`, and `gamma`, where `D` is the dipolar coupling constant, in Hz. The Euler angles `alpha`, `beta`, and `gamma` are in radians.

Example

```
>>> coupling.dipolar = {'D': 320}
```

```
>>> # or equivalently
>>> coupling.dipolar = SymmetricTensor(D=320)
```

Type
SymmetricTensor (page 475) or equivalent dict object (optional).

`name`

The name or id of the coupling. The default value is None.

Example

```
>>> coupling.name = '1H-1H'
>>> coupling.name
'1H-1H'
```

Type
str (optional).

`label`
The label for the coupling. The default value is None.

Example

```
>>> coupling.label = 'Weak coupling'
>>> coupling.label
'Weak coupling'
```

Type
str (optional).

`description`
A description of the coupling. The default value is None.

Example

```
>>> coupling.description = 'An example coupled sites.'
>>> coupling.description
'An example coupled sites.'
```

Type
str (optional).

Example

The following are a few examples of setting the site object.

```
>>> coupling1 = Coupling(
...     site_index=[0, 1],
...     isotropic_j=20, # in Hz
...     j_symmetric={
...         "zeta": 10, # in Hz
...         "eta": 0.5
...     },
...     dipolar={"D": 5.1e3}, # in Hz
... )
```

Using SymmetricTensor objects.

```
>>> coupling1 = Coupling(
...     site_index=[0, 1],
...     isotropic_j=20, # in Hz
...     j_symmetric=SymmetricTensor(zeta=10, eta=0.5),
...     dipolar=SymmetricTensor(D=5.1e3), # in Hz
... )
```

`json`(*exclude={}*, *units=True*) → dict

Parse the class object to a JSON compliant python dictionary object.

Parameters

- `exclude` – Set of keys that will be excluded from the result.
- `units` – If true, the attribute value is a physical quantity expressed as a string with a number and a unit, else a float.

Returns: dict

`classmethod parse_dict_with_units`(*py_dict: dict*)

Parse the physical quantity from a dictionary representation of the Coupling 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 394) object.

Example

```
>>> coupling_dict = {
...     "site_index": [1, 2],
...     "isotropic_j": "20 Hz",
...     "j_symmetric": {"zeta": "10 Hz", "eta":0.5}
... }
>>> coupling1 = Coupling.parse_dict_with_units(coupling_dict)
```

`reduced_dict`(*exclude={}*) → dict

Returns a reduced dictionary representation of the class object by removing all key-value pair corresponding to keys listed in the *exclude* argument, and keys with value as None.

Parameters

`exclude` – A list of keys to exclude from the dictionary.

Return: A dict.

`rotate`(*euler_angles: list*) → None

Rotate the site tensors (shielding, quadrupolar) by the given list of Euler angle rotations. Euler angles are given as a list of (alpha, beta, gamma) tuples, and rotations happen in the Haeberlen (ZYZ) convention.

Parameters

`euler_angles` (*list*) – An ordered list of angle tuples (alpha, beta, gamma) to rotate through each tensor through.

17.6 Method

Method is the root user-level object that may be used in creating custom NMR simulation methods.

17.6.1 Method

class mrsimulator.Method(*, *name: str = None*, *description: str = None*, *label: str = None*, *property_units: Dict = {'magnetic_flux_density': 'T', 'rotor_angle': 'rad', 'rotor_frequency': 'Hz'}*, *channels: List[Union[str, dict, Isotope (page 479)]]*, *spectral_dimensions: List[SpectralDimension (page 410)] = [SpectralDimension(name=None, description=None, label=None, property_units={'spectral_width': 'Hz', 'reference_offset': 'Hz', 'origin_offset': 'Hz'}, count=1024, spectral_width=25000.0, reference_offset=0.0, origin_offset=None, reciprocal=None, events=[])]*, *affine_matrix: List = None*, *simulation: Union[CSDM, ndarray] = None*, *experiment: Union[CSDM, ndarray] = None*, *magnetic_flux_density: ConstrainedFloatValue = 9.4*, *rotor_frequency: ConstrainedFloatValue = 0.0*, *rotor_angle: ConstrainedFloatValue = 0.9553166181245*)

Bases: `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(channels=['1H'], spectral_dimensions=[{}])
>>> bloch.channels = ['13C']  # Change channels
```

Type
List[Union[str, dict, *mrsimulator.spin_system.isotope.Isotope* (page 479)]]

`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.

Example

```
>>> bloch = Method(channels=['1H'], spectral_dimensions=[
...     SpectralDimension(count=8, spectral_width=50)
... ])
>>> # or equivalently
>>> bloch = Method(channels=['1H'], spectral_dimensions=[
...     {"count": 8, "spectral_width": 50}
... ])
```

Type
List[*mrsimulator.method.spectral_dimension.SpectralDimension* (page 410)]

`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 385) method.

Type
Union[csdmpy.csdm.CSDM, numpy.ndarray]

`experiment`

An object holding the experimental measurement for the given method, if available. The default value is None.

Example

```
>>> bloch.experiment = my_dataset
```

Type
Union[csdmpy.csdm.CSDM, numpy.ndarray]

`name`

Name or id of the method. The default value is None.

Example

```
>>> bloch.name = 'BlochDecaySpectrum'
>>> bloch.name
'BlochDecaySpectrum'
```

Type
str

`label`

Label for the method. The default value is None.

Example

```
>>> bloch.label = 'One pulse acquired spectrum'
>>> bloch.label
'One pulse acquired spectrum'
```

Type
str

`description`

A description of the method. The default value is None.

Example

```
>>> bloch.description = 'Huh!'
>>> bloch.description
'Huh!'
```

Type
str

affine_matrix

A (n x n) affine transformation matrix, where n is the number of spectral_dimensions. If provided, the corresponding affine transformation is applied to the computed frequencies. The default is None, i.e., no transformation is applied.

Example

```
>>> method = Method(channels=['1H'], spectral_dimensions=[{}, {}]) # 2D method
>>> method.affine_matrix = [[1, -1], [0, 1]]
>>> print(method.affine_matrix)
[[1, -1], [0, 1]]
```

Type
List

dict(***kwargs*)

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

get_symmetry_pathways(*symmetry_element: str*) → List[*SymmetryPathway* (page 483)]

Return a list of symmetry pathways of the method.

Parameters
symmetry_element (*str*) – The symmetry element, 'P' or 'D'.

Returns
A list of *SymmetryPathway* (page 483) objects.

Single channel example

Example

```
>>> from mrsimulator.method import Method
>>> method = Method(
...     channels=['1H'],
...     spectral_dimensions=[
...         {
...             "events": [
...                 {
...                     "fraction": 0.5,
...                     "transition_queries": [{"ch1": {"P": [1]}}]
...                 },
...                 {
```

(continues on next page)

(continued from previous page)

```
...                     "fraction": 0.5,
...                     "transition_queries": [{"ch1": {"P": [0]}}]
...                 }
...             ],
...         },
...         {
...             "events": [
...                 {"transition_queries": [{"ch1": {"P": [-1]}}]},
...             ],
...         }
...     ]
... )
>>> pprint(method.get_symmetry_pathways("P"))
[SymmetryPathway(
    ch1(1H): [1] ⟶ [0] ⟶ [-1]
    total: 1.0 ⟶ 0.0 ⟶ -1.0
)]
```

Dual channels example

Example

```
>>> from mrsimulator.method import Method
>>> method = Method(
...     channels=['1H', '13C'],
...     spectral_dimensions=[
...         {
...             "events": [{
...                 "fraction": 0.5,
...                 "transition_queries": [
...                     {"ch1": {"P": [1]}},
...                     {"ch1": {"P": [-1]}},
...                 ]
...             },
...             {
...                 "fraction": 0.5,
...                 "transition_queries": [  # selecting double quantum
...                     {"ch1": {"P": [-1]}, "ch2": {"P": [-1]}},
...                     {"ch1": {"P": [1]}, "ch2": {"P": [1]}},
...                 ]
...             }],
...         },
...         {
...             "events": [{
...                 "transition_queries": [ # selecting single quantum
...                     {"ch1": {"P": [-1]}},
...                 ]
...             }],
...         }
...     ]
... )
```

(continues on next page)

(continued from previous page)

```
>>> pprint(method.get_symmetry_pathways("P"))
[SymmetryPathway(
    ch1(1H): [1] ⟶ [-1] ⟶ [-1]
    ch2(13C): None ⟶ [-1] ⟶ None
    total: 1.0 ⟶ -2.0 ⟶ -1.0
),
 SymmetryPathway(
    ch1(1H): [1] ⟶ [1] ⟶ [-1]
    ch2(13C): None ⟶ [1] ⟶ None
    total: 1.0 ⟶ 2.0 ⟶ -1.0
),
 SymmetryPathway(
    ch1(1H): [-1] ⟶ [-1] ⟶ [-1]
    ch2(13C): None ⟶ [-1] ⟶ None
    total: -1.0 ⟶ -2.0 ⟶ -1.0
),
 SymmetryPathway(
    ch1(1H): [-1] ⟶ [1] ⟶ [-1]
    ch2(13C): None ⟶ [1] ⟶ None
    total: -1.0 ⟶ 2.0 ⟶ -1.0
)]
```

`get_transition_pathways`(*spin_system*) → List[*TransitionPathway* (page 482)]

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 388)) – A SpinSystem object.

Returns

A list of *TransitionPathway* (page 482) objects. Each TransitionPathway object is an ordered collection of Transition objects.

Example

```
>>> from mrsimulator import SpinSystem
>>> from mrsimulator.method.lib import ThreeQ_VAS
>>> sys = SpinSystem(sites=[{'isotope': '27Al'}, {'isotope': '29Si'}])
>>> method = ThreeQ_VAS(channels=['27Al'])
>>> pprint(method.get_transition_pathways(sys))
[|1.5, -0.5⟩⟨-1.5, -0.5| ⟶ |-0.5, -0.5⟩⟨0.5, -0.5|, weight=(1+0j),
 |1.5, -0.5⟩⟨-1.5, -0.5| ⟶ |-0.5, 0.5⟩⟨0.5, 0.5|, weight=(1+0j),
 |1.5, 0.5⟩⟨-1.5, 0.5| ⟶ |-0.5, -0.5⟩⟨0.5, -0.5|, weight=(1+0j),
 |1.5, 0.5⟩⟨-1.5, 0.5| ⟶ |-0.5, 0.5⟩⟨0.5, 0.5|, weight=(1+0j)]
```

`json`(*units=True*) → dict

Parse the class object to a JSON compliant python dictionary object.

Parameters

`units` – If true, the attribute value is a physical quantity expressed as a string with a number and a unit, else a float.

Returns: dict

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 403) object.

plot(*df=None*, *include_legend=False*) → figure

Creates a diagram representing the method. By default, only parameters which vary throughout the method are plotted. Figure can be finley adjusted using matplotlib rcParams.

Parameters

- df (*DataFrame*) – DataFrame to plot data from. By default DataFrame is calculated from summary() and will show only parameters which vary throughout the method plus 'p' symmetry pathway and 'd' symmetry pathway if it is not none or defined
- include_legend (*bool*) – Optional argument to include a key for event colors. Default is False and no key will be included in figure

Returns

matplotlib.pyplot.figure

Example

```
>>> from mrsimulator.method.lib import BlochDecaySpectrum
>>> method = BlochDecaySpectrum(channels=["13C"])
>>> fig = method.plot()
```

Adjusting Figure Size rcParams

```
>>> import matplotlib as mpl
>>> from mrsimulator.method.lib import FiveQ_VAS
>>> mpl.rcParams["figure.figsize"] = [14, 10]
>>> mpl.rcParams["font.size"] = 14
>>> method = FiveQ_VAS(channels=["27Al"])
>>> fig = method.plot(include_legend=True)
```

Plotting all Parameters, including Constant

```
>>> from mrsimulator.method.lib import FiveQ_VAS
>>> method = FiveQ_VAS(channels=["27Al"])
>>> df = method.summary(drop_constant_columns=False)
>>> fig = method.plot(df=df)
```

reduced_dict(*exclude={}*) → dict

Returns a reduced dictionary representation of the class object by removing all key-value pair corresponding to keys listed in the *exclude* argument, and keys with value as None.

Parameters

exclude – A list of keys to exclude from the dictionary.

Return: A dict.

shape() → tuple

The shape of the method's spectral dimension array.

Returns
tuple

Example

```
>>> from mrsimulator.method import Method
>>> method = Method(
...     channels=['1H'],
...     spectral_dimensions=[{'count': 40}, {'count': 10}]
... )
>>> method.shape()
(40, 10)
```

summary(*drop_constant_columns=True*) → DataFrame

Returns a DataFrame giving a summary of the Method. A user can specify optional attributes to include which appear as columns in the DataFrame. A user can also ask to leave out attributes which remain constant throughout the method. Invalid attributes for an Event will be replaced with NAN.

Parameters
drop_constant_columns (*bool*) – Removes constant properties if True. Default is True.

Returns
Event number as row and property as column. Invalid properties for an event type are filled with np.nan

Columns
- (str) type: Event type
- (int) spec_dim_index: Index of spectral dimension which event belongs to
- (str) label: Event label
- (float) duration: Duration of the DelayEvent
- (float) fraction: Fraction of the SpectralEvent
- (MixingQuery) query: MixingQuery object of the MixingEvent
- (float) magnetic_flux_density: Magnetic flux density during event in Tesla
- (float) rotor_frequency: Rotor frequency during event in Hz
- (float) rotor_angle: Rotor angle during event converted to Degrees
- (FrequencyEnum) freq_contrib: Frequency

Return type
pd.DataFrame df

Example

All Possible Columns

```
>>> from mrsimulator.method.lib import ThreeQ_VAS
>>> method = ThreeQ_VAS(channels=["17O"])
>>> df = method.summary(drop_constant_columns=False)
>>> pprint(list(df.columns))
['type',
 'spec_dim_index',
 'spec_dim_label',
 'label',
 'duration',
 'fraction',
 'query',
```

(continues on next page)

(continued from previous page)

```
'magnetic_flux_density',
'rotor_frequency',
'rotor_angle',
'freq_contrib',
'p',
'd']
```

17.6.2 SpectralDimension

class mrsimulator.SpectralDimension(*, *name: str = None*, *description: str = None*, *label: str = None*, *property_units: Dict = {'origin_offset': 'Hz', 'reference_offset': 'Hz', 'spectral_width': 'Hz'}*, *count: ConstrainedIntValue = 1024*, *spectral_width: float = 25000.0*, *reference_offset: float = 0.0*, *origin_offset: float = None*, *reciprocal: Reciprocal = None*, *events: List[Union[MixingEvent (page 416), DelayEvent (page 414), SpectralEvent (page 412)]] = []*)

Bases: `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 403) 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 *Events* (page 412) or equivalent dict objects (optional).

`coordinates_Hz()` → ndarray

The grid coordinates along the dimension in units of Hz, evaluated as

$$x_{\text{Hz}} = ([0, 1, ...N-1] - T)\,\frac{\Delta x}{N} + x_0 \tag{17.3}$$

where $T = N/2$ and $T = (N-1)/2$ for even and odd values of N, respectively.

`coordinates_ppm()` → ndarray

The grid coordinates along the dimension as dimension frequency ratio in units of ppm. The coordinates are evaluated as

$$x_{\text{ppm}} = \frac{x_{\text{Hz}}}{x_0 + \omega_0} \tag{17.4}$$

where ω_0 is the Larmor frequency.

`json(`*exclude={}*, *units=True*`)` → dict

Parse the class object to a JSON compliant python dictionary object.

Parameters
- `exclude` – Set of keys that will be excluded from the result.
- `units` – If true, the attribute value is a physical quantity expressed as a string with a number and a unit, else a float.

Returns: dict

`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

`reduced_dict(`*exclude={}*`)` → dict

Returns a reduced dictionary representation of the class object by removing all key-value pair corresponding to keys listed in the *exclude* argument, and keys with value as None.

Parameters
`exclude` – A list of keys to exclude from the dictionary.

Return: A dict.

`to_csdm_dimension()` → Dimension

Return the spectral dimension as a CSDM dimension object.

`classmethod validate_events(`***kwargs*`)`

Ensure at least one spectralEvent and warn is the sum of fraction in SpectralEvents is not 1.

`classmethod validate_spectral_width(`*value*`)`

Spectral width cannot be zero.

17.6.3 Events

class mrsimulator.method.SpectralEvent(*, name: str = None, description: str = None, label: str = None, property_units: ~typing.Dict = {'magnetic_flux_density': 'T', 'rotor_angle': 'rad', 'rotor_frequency': 'Hz'}, magnetic_flux_density: ~mrsimulator.method.event.ConstrainedFloatValue = None, rotor_frequency: ~mrsimulator.method.event.ConstrainedFloatValue = None, rotor_angle: ~mrsimulator.method.event.ConstrainedFloatValue = None, freq_contrib: ~typing.List[~typing.Union[~mrsimulator.method.frequency_contrib.FrequencyEnum, str]] = [<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'>, <FrequencyEnum.J1_0: 'J1_0'>, <FrequencyEnum.J1_2: 'J1_2'>, <FrequencyEnum.D1_2: 'D1_2'>, <FrequencyEnum.Quad_Shielding_cross_0: 'Quad_Shielding_cross_0'>, <FrequencyEnum.Quad_Shielding_cross_2: 'Quad_Shielding_cross_2'>, <FrequencyEnum.Quad_Shielding_cross_4: 'Quad_Shielding_cross_4'>, <FrequencyEnum.Quad_J_cross_0: 'Quad_J_cross_0'>, <FrequencyEnum.Quad_J_cross_2: 'Quad_J_cross_2'>, <FrequencyEnum.Quad_J_cross_4: 'Quad_J_cross_4'>, <FrequencyEnum.Quad_Dipolar_cross_0: 'Quad_Dipolar_cross_0'>, <FrequencyEnum.Quad_Dipolar_cross_2: 'Quad_Dipolar_cross_2'>, <FrequencyEnum.Quad_Dipolar_cross_4: 'Quad_Dipolar_cross_4'>], transition_queries: ~typing.List[~mrsimulator.method.query.TransitionQuery] = [TransitionQuery(name=None, description=None, label=None, property_units={}, ch1=SymmetryQuery(name=None, description=None, label=None, property_units={}, P=[0], D=None, F=None, transitions=None), ch2=None, ch3=None)], fraction: float = 1.0)

Bases: `BaseEvent`

Base SpectralEvent class defines the spin environment and the transition query for a segment of the transition pathway.

`fraction`

The weight of the frequency contribution from the event. The default is 1.

Type
float

`magnetic_flux_density`

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`.

Type
float

`rotor_frequency`

The sample spinning frequency ν_r, during the event in units of Hz. The default value is `0`.

Type
float

`rotor_angle`

The angle between the sample rotation axis and the applied external magnetic field vector, θ, during the event in units of rad. The default value is `0.9553166`, i.e. the magic angle.

Type
float

`freq_contrib`

A list of FrequencyEnum enumeration. The default is all frequency enumerations.

Type
List[Union[*mrsimulator.method.frequency_ contrib.FrequencyEnum* (page 416), str]]

`transition_queries`

A TransitionQuery or an equivalent dict object listing the queries used in selecting the active transitions during the event. Only the active transitions from this query will contribute to the net frequency.

Type
List[*mrsimulator.method.query.TransitionQuery* (page 419)]

`combination(`*isotopes*, *channels*`)`

All possible combinations of the event queries over the given channels and list of isotopes.

Parameters
- `isotopes` (*(list)*) – List of isotopes in the spin system.
- `channels` (*(list)*) – List of method channels.

`dict(`***kwargs*`)` → dict

Return a JSON compliant dictionary of the instance of the event.

`filter_transitions(`*all_ transitions*, *isotopes*, *channels*`)`

Filter transitions based on the transition query.

Parameters
- `all_transitions` (*(list)*) – List of all transitions from the spin system.
- `isotopes` (*(list)*) – List of isotopes in the spin system.
- `channels` (*(list)*) – List of method channels.

`json(`*exclude={}*, *units=True*`)` → dict

Parse the class object to a JSON compliant python dictionary object.

Parameters
- `exclude` – Set of keys that will be excluded from the result.
- `units` – If true, the attribute value is a physical quantity expressed as a string with a number and a unit, else a float.

Returns: dict

`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

reduced_dict(*exclude={}*) → dict

Returns a reduced dictionary representation of the class object by removing all key-value pair corresponding to keys listed in the *exclude* argument, and keys with value as None.

Parameters

exclude – A list of keys to exclude from the dictionary.

Return: A dict.

class mrsimulator.method.DelayEvent(**, name: str = None, description: str = None, label: str = None, property_units: ~typing.Dict = {'duration': 's', 'magnetic_flux_density': 'T', 'rotor_angle': 'rad', 'rotor_frequency': 'Hz'}, magnetic_flux_density: ~mrsimulator.method.event.ConstrainedFloatValue = None, rotor_frequency: ~mrsimulator.method.event.ConstrainedFloatValue = None, rotor_angle: ~mrsimulator.method.event.ConstrainedFloatValue = None, freq_contrib: ~typing.List[~typing.Union[~mrsimulator.method.frequency_contrib.FrequencyEnum, str]] = [<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'>, <FrequencyEnum.J1_0: 'J1_0'>, <FrequencyEnum.J1_2: 'J1_2'>, <FrequencyEnum.D1_2: 'D1_2'>, <FrequencyEnum.Quad_Shielding_cross_0: 'Quad_Shielding_cross_0'>, <FrequencyEnum.Quad_Shielding_cross_2: 'Quad_Shielding_cross_2'>, <FrequencyEnum.Quad_Shielding_cross_4: 'Quad_Shielding_cross_4'>, <FrequencyEnum.Quad_J_cross_0: 'Quad_J_cross_0'>, <FrequencyEnum.Quad_J_cross_2: 'Quad_J_cross_2'>, <FrequencyEnum.Quad_J_cross_4: 'Quad_J_cross_4'>, <FrequencyEnum.Quad_Dipolar_cross_0: 'Quad_Dipolar_cross_0'>, <FrequencyEnum.Quad_Dipolar_cross_2: 'Quad_Dipolar_cross_2'>, <FrequencyEnum.Quad_Dipolar_cross_4: 'Quad_Dipolar_cross_4'>], transition_queries: ~typing.List[~mrsimulator.method.query.TransitionQuery] = [TransitionQuery(name=None, description=None, label=None, property_units={}, ch1=SymmetryQuery(name=None, description=None, label=None, property_units={}, P=[0], D=None, F=None, transitions=None), ch2=None, ch3=None)], duration: float*)

Bases: BaseEvent

Base DelayEvent class defines the spin environment and the transition query for a segment of the transition pathway. The frequency from this event contribute to the spectrum as complex amplitude modulations.

duration

The duration of the event in units of s. The default is 0.

Type

float

magnetic_flux_density

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.

Type
float

`rotor_frequency`

The sample spinning frequency ν_r, during the event in units of Hz. The default value is `0`.

Type
float

`rotor_angle`

The angle between the sample rotation axis and the applied external magnetic field vector, θ, during the event in units of rad. The default value is `0.9553166`, i.e. the magic angle.

Type
float

`freq_contrib`

A list of FrequencyEnum enumeration. The default is all frequency enumerations.

Type
List[Union[*mrsimulator.method.frequency_contrib.FrequencyEnum* (page 416), str]]

`transition_queries`

A TransitionQuery or an equivalent dict object listing the queries used in selecting the active transitions during the event. Only the active transitions from this query will contribute to the net frequency.

Type
List[*mrsimulator.method.query.TransitionQuery* (page 419)]

`combination(`*isotopes*, *channels*`)`

All possible combinations of the event queries over the given channels and list of isotopes.

Parameters
- `isotopes` (*(list)*) – List of isotopes in the spin system.
- `channels` (*(list)*) – List of method channels.

`dict(`***kwargs*`)` → dict

Return a JSON compliant dictionary of the instance of the event.

`filter_transitions(`*all_transitions*, *isotopes*, *channels*`)`

Filter transitions based on the transition query.

Parameters
- `all_transitions` (*(list)*) – List of all transitions from the spin system.
- `isotopes` (*(list)*) – List of isotopes in the spin system.
- `channels` (*(list)*) – List of method channels.

`json(`*exclude={}*, *units=True*`)` → dict

Parse the class object to a JSON compliant python dictionary object.

Parameters
- `exclude` – Set of keys that will be excluded from the result.
- `units` – If true, the attribute value is a physical quantity expressed as a string with a number and a unit, else a float.

Returns: dict

`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

reduced_dict(*exclude={}*) → dict

Returns a reduced dictionary representation of the class object by removing all key-value pair corresponding to keys listed in the *exclude* argument, and keys with value as None.

Parameters
exclude – A list of keys to exclude from the dictionary.

Return: A dict.

class mrsimulator.method.MixingEvent(*, *name: str = None*, *description: str = None*, *label: str = None*, *property_units: Dict = {}*, *query: Union[*MixingQuery *(page 422)*, MixingEnum *(page 424)]*)

Bases: Parseable

Transition mixing class

query

The transition mixing query.

Type
Union[*mrsimulator.method.query.MixingQuery* (page 422), *mrsimulator.method.query.MixingEnum* (page 424)]

classmethod parse_dict_with_units(*py_dict*)

Parse the physical quantity from a dictionary representation of the MixingEvent 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 MixingEvent object.

Returns
A MixingEvent.

classmethod validate_query(*v*, ***kwargs*)

Validator which tries to convert query to a MixingEnum if query is string

17.6.4 FrequencyEnum

class mrsimulator.method.frequency_contrib.FrequencyEnum(*value*)

Bases: str, Enum

Enumeration for selecting specific frequency contributions. The enumerations are:

Shielding1_0

Selects first-order and zeroth-rank nuclear shielding frequency contributions.

Type
str

Shielding1_2

Selects first-order and second-rank nuclear shielding frequency contributions.

Type
str

`Quad1_2`
: Selects first-order and second-rank quadrupolar frequency contributions.

  Type
  : str

`Quad2_0`
: Selects second-order and zeroth-rank quadrupolar frequency contributions.

  Type
  : str

`Quad2_2`
: Selects second-order and second-rank quadrupolar frequency contributions.

  Type
  : str

`Quad2_4`
: Selects second-order and fourth-rank quadrupolar frequency contributions.

  Type
  : str

`J1_0`
: Selects first-order and zeroth-rank weak J-coupling frequency contributions.

  Type
  : str

`J1_2`
: Selects first-order and second-rank weak J-coupling frequency contributions.

  Type
  : str

`D1_2`
: Selects first-order and second-rank weak dipole frequency contributions.

  Type
  : str

`Quad_Shielding_cross_0`
: Selects zeroth-rank quad-shielding cross interaction.

  Type
  : str

`Quad_Shielding_cross_2`
: Selects second-rank quad-shielding cross interaction.

  Type
  : str

`Quad_Shielding_cross_4`
: Selects fourth-rank quad-shielding cross interaction.

  Type
  : str

`Quad_J_cross_0`
: Selects zeroth-rank quad-J-coupling cross interaction.

  Type
  : str

`Quad_J_cross_2`
: Selects second-rank quad-J-coupling cross interaction.

  Type
  : str

`Quad_J_cross_4`
: Selects fourth-rank quad-J-coupling cross interaction.

  Type
  : str

`Quad_Dipolar_cross_0`
: Selects zeroth-rank quad-dipolar coupling cross interaction.

  Type
  : str

`Quad_Dipolar_cross_2`
: Selects second-rank quad-dipolar coupling cross interaction.

  Type
  : str

`Quad_Dipolar_cross_4`
: Selects fourth-rank quad-dipolar coupling cross interaction.

  Type
  : str

There are also shortcuts for including/excluding sets of contributions together. Frequency contributions can be excluded by including an exclamation mark in-front of of the string, for example `"!Shielding"` excludes all shielding interactions. The allowed shortcuts are:

Shortcuts

`"Shielding":`
: Selects all shielding interactions

`"Isotropic":`
: Selects first-order zeroth-rank shielding and first-order zeroth-rank J coupling interactions

`"Quad":`
: Selects all quadrupolar interactions

`"J":`
: Selects all J coupling interactions

`"D":`
: Selects all dipolar interactions

`"cross":`
: Selects all cross-term interactions

`"Quad_Shielding_cross":`
: Selects all quadrupolar-shielding cross terms

`"Quad_J_cross":`
: Selects all quadrupolar-J-coupling cross terms

`"Quad_D_cross"`:
Selects all quadrupolar-dipolar-coupling cross terms
`"First_order"`:
Selects all first-order interactions
`"Second_order"`:
Selects all second-order interactions Selects
`"Zeroth_rank"`:
Selects all zeroth-rank interactions
`"Second_rank"`:
Selects all second-rank interactions
`"Fourth_rank"`:
Selects all fourth-rank interactions

`json(**kwargs)` → str

Parse the class object to a JSON compliant python dictionary object.

17.6.5 Query objects

`class mrsimulator.method.query.TransitionQuery(*, name: str = None, description: str = None, label: str = None, property_units: Dict = {}, ch1: Optional[SymmetryQuery (page 420)] = SymmetryQuery(name=None, description=None, label=None, property_units={}, P=[0], D=None, F=None, transitions=None), ch2: Optional[SymmetryQuery (page 420)] = None, ch3: Optional[SymmetryQuery (page 420)] = None)`

Bases: `Parseable`

TransitionQuery class for querying transition symmetry function.

`ch1`

An optional SymmetryQuery object for querying symmetry functions at channel index 0 of the method's channels array."

Type
Optional[*mrsimulator.method.query.SymmetryQuery* (page 420)]

`ch2`

An optional SymmetryQuery object for querying symmetry functions at channel index 1 of the method's channels array."

Type
Optional[*mrsimulator.method.query.SymmetryQuery* (page 420)]

`ch3`

An optional SymmetryQuery object for querying symmetry functions at channel index 2 of the method's channels array."

Type
Optional[*mrsimulator.method.query.SymmetryQuery* (page 420)]

Example

```
>>> query = TransitionQuery(ch1={'P': [1], 'D': [0]}, ch2={'P': [-1]})
```

static cartesian_product_indexing(*combinations*)

Return Cartesian product of indexes

combination(*isotopes*, *channels*)

Combinations of TransitionQuery based on the number of sites per channel.

Parameters

- **isotopes** (*(list)*) – List of isotope symbols, [‘29Si , ‘13C’, ‘13C’, ‘1H’].
- **channels** (*(int)*) – List of method channels, [‘29Si , ‘13C’].

json(*exclude={}*, *units=True*) → dict

Parse the class object to a JSON compliant python dictionary object.

Parameters

- **exclude** – Set of keys that will be excluded from the result.
- **units** – If true, the attribute value is a physical quantity expressed as a string with a number and a unit, else a float.

Returns: dict

classmethod parse_dict_with_units(*json_dict: dict*)

Parse the physical quantity from a dictionary representation of the class object, where the physical quantity is expressed as a string with a number and a unit.

Parameters

json_dict (*dict*) – A required python dict object.

reduced_dict(*exclude={}*) → dict

Returns a reduced dictionary representation of the class object by removing all key-value pair corresponding to keys listed in the *exclude* argument, and keys with value as None.

Parameters

exclude – A list of keys to exclude from the dictionary.

Return: A dict.

class mrsimulator.method.query.SymmetryQuery(*, *name: str = None*, *description: str = None*, *label: str = None*, *property_units: Dict = {}*, *P: List[int] = [0]*, *D: List[int] = None*, *F: List[float] = None*, *transitions: List[Transition (page 480)] = None*)

Bases: Parseable

Base SymmetryQuery class.

P

A list of p symmetry functions per site. Here p = Δm = $m_f - m_i$ is the difference between the spin quantum numbers of the final and initial states.

Example

```
>>> method = Method(channels=['1H'], spectral_dimensions=[{"events": [
...     {"fraction": 1}
... ]}])
>>> method.spectral_dimensions[0].events[0].transition_queries[0].ch1.P = [-1]
```

Type
List[int]

D

A list of d symmetry functions per site. Here d $= m_f^2 - m_i^2$ is the difference between the square of the spin quantum numbers of the final and initial states.

Example

```
>>> method.spectral_dimensions[0].events[0].transition_queries[0].ch1.D = [0]
```

Type
List[int]

json(*exclude={}*, *units=True*) → dict

Parse the class object to a JSON compliant python dictionary object.

Parameters
- exclude – Set of keys that will be excluded from the result.
- units – If true, the attribute value is a physical quantity expressed as a string with a number and a unit, else a float.

Returns: dict

classmethod parse_dict_with_units(*json_dict: dict*)

Parse the physical quantity from a dictionary representation of the class object, where the physical quantity is expressed as a string with a number and a unit.

Parameters
json_dict (*dict*) – A required python dict object.

query_combination(*symmetry*, *n_site_at_channel_id*)

Combination of symmetry query based on the number of sites in given channel.

Parameters
- symmetry (*str*) – The symmetry element, 'P' or 'D'.
- n_site_at_channel (*int*) – Number of sites for the given channel.

Example

Consider the following

query = {P: [-1], D: [1]} n_isotopes = [3] channels = ['A']

then, 1. P query will expand to [-1, 0, 0], [0, -1, 0], and [0, 0, -1] combinations 2. D query will expand to [1, 0, 0], [0, 1, 0], and [0, 0, 1] combinations

`reduced_dict(`*exclude={}*`)` → dict

Returns a reduced dictionary representation of the class object by removing all key-value pair corresponding to keys listed in the *exclude* argument, and keys with value as None.

Parameters

`exclude` – A list of keys to exclude from the dictionary.

Return: A dict.

`class mrsimulator.method.query.MixingQuery(`**, name: str = None, description: str = None, label: str = None, property_units: Dict = {}, ch1: Optional[RotationQuery (page 423)] = None, ch2: Optional[RotationQuery (page 423)] = None, ch3: Optional[RotationQuery (page 423)] = None*`)`

Bases: `Parseable`

MixingQuery class for querying transition mixing between events.

`ch1`

An optional RotationQuery object for channel at index 0 of method's channels."

Type

Optional[*mrsimulator.method.query.RotationQuery* (page 423)]

`ch2`

An optional RotationQuery object for channel at index 1 of method's channels."

Type

Optional[*mrsimulator.method.query.RotationQuery* (page 423)]

`ch3`

An optional RotationQuery object for channel at index 2 of method's channels."

Type

Optional[*mrsimulator.method.query.RotationQuery* (page 423)]

Example

```
>>> query = MixingQuery(ch1={"angle": 1.570796, "phase": 3.141593})
```

`property channels: List[`*RotationQuery* `(page 423)]`

Returns an ordered list of all channels

`json(`*exclude={}, units=True*`)` → dict

Parse the class object to a JSON compliant python dictionary object.

Parameters

- `exclude` – Set of keys that will be excluded from the result.
- `units` – If true, the attribute value is a physical quantity expressed as a string with a number and a unit, else a float.

Returns: dict

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 403) object.

reduced_dict(*exclude={}*) → dict

Returns a reduced dictionary representation of the class object by removing all key-value pair corresponding to keys listed in the *exclude* argument, and keys with value as None.

Parameters
exclude – A list of keys to exclude from the dictionary.

Return: A dict.

class mrsimulator.method.query.RotationQuery(*, *name: str = None*, *description: str = None*, *label: str = None*, *property_ units: Dict = {'angle': 'rad', 'phase': 'rad'}*, *angle: ConstrainedFloatValue = 0.0*, *phase: float = 0.0*)

Bases: Parseable

Base RotationQuery class.

angle

The rf rotation angle in units of radians.

Type
float

phase

The rf rotation phase in units of radians.

Type
float

json(*units=True*, ***kwargs*)

Parse the class object to a JSON compliant python dictionary object.

Parameters
- exclude – Set of keys that will be excluded from the result.
- units – If true, the attribute value is a physical quantity expressed as a string with a number and a unit, else a float.

Returns: dict

classmethod parse_dict_with_units(*json_ dict: dict*)

Parse the physical quantity from a dictionary representation of the class object, where the physical quantity is expressed as a string with a number and a unit.

Parameters
json_dict (*dict*) – A required python dict object.

reduced_dict(*exclude={}*) → dict

Returns a reduced dictionary representation of the class object by removing all key-value pair corresponding to keys listed in the *exclude* argument, and keys with value as None.

Parameters
exclude – A list of keys to exclude from the dictionary.

Return: A dict.

class mrsimulator.method.query.MixingEnum(*value*)

Bases: Enum

Enumerations for defining common mixing queries. The enumerations are as follows:

TotalMixing

Setting query attribute to TotalMixing causes all transitions in one spectral event to all other transitions. This is the same behavior when no MixingEvent is defined between SpectralEvents.

Type
str

NoMixing

Defines mixing query where no pathways connect

Type
mrsimulator.method.query.MixingQuery (page 422)

Example

The query attribute of the *MixingEvent* (page 416) can be set to the Enum itself or a string representing the Enum.

```
>>> from mrsimulator.method import MixingEvent
>>> from mrsimulator.method.query import MixingEnum
>>> # From Enum object
>>> total_mix = MixingEvent(query=MixingEnum.TotalMixing)
>>> no_mix = MixingEvent(query=MixingEnum.NoMixing)
>>> # From string representing Enum
>>> total_mix = MixingEvent(query="TotalMixing")
>>> no_mix = MixingEvent(query="NoMixing")
```

classmethod allowed_enums()

Returns list of str corresponding to all valid enumerations

json(***kwargs*)

Return a JSON-compliant serialization of the enumeration

17.7 Methods

The following are the list of methods currently supported by **mrsimulator** as a part of the mrsimulator.method.lib module. To import a method, for example the *BlochDecaySpectrum*, used

```
>>> from mrsimulator.method.lib import BlochDecaySpectrum
```

17.7.1 Summary

Specialized 1D methods

BlochDecaySpectrum (page 425)(*[, name, description, ...])	Simulate a Bloch decay spectrum.
BlochDecayCTSpectrum (page 431)(*[, name, description, ...])	Simulate a Bloch decay central transition selective spectrum.

Specialized 2D methods

ThreeQ_VAS (page 438)(*[, name, description, label, ...])	Simulate a sheared and scaled 3Q 2D variable-angle spinning spectrum.
FiveQ_VAS (page 444)(*[, name, description, label, ...])	Simulate a sheared and scaled 5Q variable-angle spinning spectrum.
SevenQ_VAS (page 450)(*[, name, description, label, ...])	Simulate a sheared and scaled 7Q variable-angle spinning spectrum.
ST1_VAS (page 457)(*[, name, description, label, ...])	Simulate a sheared and scaled inner satellite and central transition correlation spectrum.
ST2_VAS (page 463)(*[, name, description, label, ...])	Simulate a sheared and scaled second to inner satellite and central transition correlation spectrum.
SSB2D (page 469)(*[, name, description, label, ...])	Simulating a sheared 2D finite to infinite speed MAS correlation spectrum.

17.7.2 Table of contents

Bloch Decay Spectrum method

`class mrsimulator.method.lib.BlochDecaySpectrum(`**, name: str = 'BlochDecaySpectrum', description: str = 'A one-dimensional Bloch decay spectrum method.', label: str = None, property_units: Dict = {'magnetic_flux_density': 'T', 'rotor_angle': 'rad', 'rotor_frequency': 'Hz'}, channels: List[Union[str, dict, Isotope (page 479)]], spectral_dimensions: List[SpectralDimension (page 410)] = [SpectralDimension(name=None, description=None, label=None, property_units={'spectral_width': 'Hz', 'reference_offset': 'Hz', 'origin_offset': 'Hz'}, count=1024, spectral_width=25000.0, reference_offset=0.0, origin_offset=None, reciprocal=None, events=[])], affine_matrix: List = None, simulation: Union[CSDM, ndarray] = None, experiment: Union[CSDM, ndarray] = None, magnetic_flux_density: ConstrainedFloatValue = 9.4, rotor_frequency: ConstrainedFloatValue = 0.0, rotor_angle: ConstrainedFloatValue = 0.9553166181245*`)`

Bases: `BaseNamedMethod1D`

Simulate a Bloch decay spectrum.

classmethod check_event_objects_for_compatibility(*default_dim*, *obj_dim*, *method_dict*)

Checks Events for compatibility and sets global method attributes

Parameters

- default_dim (*dict*) – Dict representation of SpectralDimension in base method
- obj_dim (SpectralDimension (page 410)) – User-passed SpectralDimension object to check
- method_dict (*dict*) – Dict representation of passed method

classmethod check_method_compatibility(*py_dict*)

Check for events attribute inside the spectral_dimensions. Events are not allowed for NamedMethods.

dict(***kwargs*)

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

get_symmetry_pathways(*symmetry_element: str*) → List[*SymmetryPathway* (page 483)]

Return a list of symmetry pathways of the method.

Parameters

symmetry_element (*str*) – The symmetry element, 'P' or 'D'.

Returns

A list of *SymmetryPathway* (page 483) objects.

Single channel example

Example

```
>>> from mrsimulator.method import Method
>>> method = Method(
...     channels=['1H'],
...     spectral_dimensions=[
...         {
...             "events": [
...                 {
...                     "fraction": 0.5,
...                     "transition_queries": [{"ch1": {"P": [1]}}]
...                 },
...                 {
...                     "fraction": 0.5,
...                     "transition_queries": [{"ch1": {"P": [0]}}]
...                 }
...             ],
...         },
...         {
...             "events": [
...                 {"transition_queries": [{"ch1": {"P": [-1]}}]},
...             ],
...         }
...     ]
... )
>>> pprint(method.get_symmetry_pathways("P"))
[SymmetryPathway(
    ch1(1H): [1] ⟶ [0] ⟶ [-1]
```

(continues on next page)

(continued from previous page)

```
    total: 1.0 ⟶ 0.0 ⟶ -1.0
)]
```

Dual channels example

Example

```
>>> from mrsimulator.method import Method
>>> method = Method(
...     channels=['1H', '13C'],
...     spectral_dimensions=[
...         {
...             "events": [{
...                 "fraction": 0.5,
...                 "transition_queries": [
...                     {"ch1": {"P": [1]}},
...                     {"ch1": {"P": [-1]}},
...                 ]
...             },
...             {
...                 "fraction": 0.5,
...                 "transition_queries": [  # selecting double quantum
...                     {"ch1": {"P": [-1]}, "ch2": {"P": [-1]}},
...                     {"ch1": {"P": [1]}, "ch2": {"P": [1]}},
...                 ]
...             }],
...         },
...         {
...             "events": [{
...                 "transition_queries": [ # selecting single quantum
...                     {"ch1": {"P": [-1]}},
...                 ]
...             }],
...         }
...     ]
... )
>>> pprint(method.get_symmetry_pathways("P"))
[SymmetryPathway(
    ch1(1H): [1] ⟶ [-1] ⟶ [-1]
    ch2(13C): None ⟶ [-1] ⟶ None
    total: 1.0 ⟶ -2.0 ⟶ -1.0
),
 SymmetryPathway(
    ch1(1H): [1] ⟶ [1] ⟶ [-1]
    ch2(13C): None ⟶ [1] ⟶ None
    total: 1.0 ⟶ 2.0 ⟶ -1.0
),
 SymmetryPathway(
    ch1(1H): [-1] ⟶ [-1] ⟶ [-1]
    ch2(13C): None ⟶ [-1] ⟶ None
    total: -1.0 ⟶ -2.0 ⟶ -1.0
```

(continues on next page)

(continued from previous page)

```
),
 SymmetryPathway(
    ch1(1H): [-1] ⟶ [1] ⟶ [-1]
    ch2(13C): None ⟶ [1] ⟶ None
    total: -1.0 ⟶ 2.0 ⟶ -1.0
)]
```

`get_transition_pathways(`*spin_system*`)` → List[*TransitionPathway* (page 482)]

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 388)) – A SpinSystem object.

Returns

A list of *TransitionPathway* (page 482) objects. Each TransitionPathway object is an ordered collection of Transition objects.

Example

```
>>> from mrsimulator import SpinSystem
>>> from mrsimulator.method.lib import ThreeQ_VAS
>>> sys = SpinSystem(sites=[{'isotope': '27Al'}, {'isotope': '29Si'}])
>>> method = ThreeQ_VAS(channels=['27Al'])
>>> pprint(method.get_transition_pathways(sys))
[|1.5, -0.5⟩⟨-1.5, -0.5| ⟶ |-0.5, -0.5⟩⟨0.5, -0.5|, weight=(1+0j),
 |1.5, -0.5⟩⟨-1.5, -0.5| ⟶ |-0.5, 0.5⟩⟨0.5, 0.5|, weight=(1+0j),
 |1.5, 0.5⟩⟨-1.5, 0.5| ⟶ |-0.5, -0.5⟩⟨0.5, -0.5|, weight=(1+0j),
 |1.5, 0.5⟩⟨-1.5, 0.5| ⟶ |-0.5, 0.5⟩⟨0.5, 0.5|, weight=(1+0j)]
```

`json(`*units=True*`)` → dict

Parse the class object to a JSON compliant python dictionary object.

Parameters

`units` – If true, the attribute value is a physical quantity expressed as a string with a number and a unit, else a float.

Returns: dict

`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 403) object.

`plot(`*df=None*, *include_legend=False*`)` → figure

Creates a diagram representing the method. By default, only parameters which vary throughout the method are plotted. Figure can be finley adjusted using matplotlib rcParams.

Parameters

- `df` (*DataFrame*) – DataFrame to plot data from. By default DataFrame is calculated from summary() and will show only parameters which vary throughout the method plus 'p' symmetry pathway and 'd' symmetry pathway if it is not none or defined

- include_legend (*bool*) – Optional argument to include a key for event colors. Default is False and no key will be included in figure

Returns

matplotlib.pyplot.figure

Example

```
>>> from mrsimulator.method.lib import BlochDecaySpectrum
>>> method = BlochDecaySpectrum(channels=["13C"])
>>> fig = method.plot()
```

Adjusting Figure Size rcParams

```
>>> import matplotlib as mpl
>>> from mrsimulator.method.lib import FiveQ_VAS
>>> mpl.rcParams["figure.figsize"] = [14, 10]
>>> mpl.rcParams["font.size"] = 14
>>> method = FiveQ_VAS(channels=["27Al"])
>>> fig = method.plot(include_legend=True)
```

Plotting all Parameters, including Constant

```
>>> from mrsimulator.method.lib import FiveQ_VAS
>>> method = FiveQ_VAS(channels=["27Al"])
>>> df = method.summary(drop_constant_columns=False)
>>> fig = method.plot(df=df)
```

reduced_dict(*exclude={}*) → dict

Returns a reduced dictionary representation of the class object by removing all key-value pair corresponding to keys listed in the *exclude* argument, and keys with value as None.

Parameters

exclude – A list of keys to exclude from the dictionary.

Return: A dict.

shape() → tuple

The shape of the method's spectral dimension array.

Returns

tuple

Example

```
>>> from mrsimulator.method import Method
>>> method = Method(
...     channels=['1H'],
...     spectral_dimensions=[{'count': 40}, {'count': 10}]
... )
>>> method.shape()
(40, 10)
```

summary(*drop_constant_columns=True*) → DataFrame

Returns a DataFrame giving a summary of the Method. A user can specify optional attributes to include which appear as columns in the DataFrame. A user can also ask to leave out attributes which remain constant throughout the method. Invalid attributes for an Event will be replaced with NAN.

Parameters

drop_constant_columns (*bool*) – Removes constant properties if True. Default is True.

Returns

Event number as row and property as column. Invalid properties for an event type are filled with np.nan

Columns

- (str) type: Event type
- (int) spec_dim_index: Index of spectral dimension which event belongs to
- (str) label: Event label
- (float) duration: Duration of the DelayEvent
- (float) fraction: Fraction of the SpectralEvent
- (MixingQuery) query: MixingQuery object of the MixingEvent
- (float) magnetic_flux_density: Magnetic flux density during event in Tesla
- (float) rotor_frequency: Rotor frequency during event in Hz
- (float) rotor_angle: Rotor angle during event converted to Degrees
- (FrequencyEnum) freq_contrib: Frequency

Return type

pd.DataFrame df

Example

All Possible Columns

```
>>> from mrsimulator.method.lib import ThreeQ_VAS
>>> method = ThreeQ_VAS(channels=["17O"])
>>> df = method.summary(drop_constant_columns=False)
>>> pprint(list(df.columns))
['type',
 'spec_dim_index',
 'spec_dim_label',
 'label',
 'duration',
 'fraction',
 'query',
 'magnetic_flux_density',
 'rotor_frequency',
 'rotor_angle',
 'freq_contrib',
 'p',
 'd']
```

Example

```
>>> from mrsimulator.method.lib import BlochDecaySpectrum
>>> from mrsimulator.method import SpectralDimension
>>> Bloch_method = BlochDecaySpectrum(
...     channels=["1H"],
...     rotor_frequency=5000,  # in Hz
```

(continues on next page)

(continued from previous page)

```
...     rotor_angle=54.735 * 3.14159 / 180,  # in rad
...     magnetic_flux_density=9.4,  # in T
...     spectral_dimensions=[
...         SpectralDimension(count=1024, spectral_width=50000, reference_offset=-8000)
...     ],
... )
```

Bloch decay method is is part of the methods library, but can be constructed with a generic method as follows

```
>>> from mrsimulator.method import Method
>>> bloch_decay = Method(
...     channels=["1H"],
...     rotor_frequency=5000,  # in Hz
...     rotor_angle=54.735 * 3.14159 / 180,  # 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_queries": [{"ch1": {"P": [-1]}}]}],
...         }
...     ],
... )
```

Bloch Decay Central Transition Spectrum method

`class mrsimulator.method.lib.BlochDecayCTSpectrum(*, name: str = 'BlochDecayCTSpectrum', description: str = 'A one-dimensional central transition selective Bloch decay spectrum method.', label: str = None, property_units: Dict = {'magnetic_flux_density': 'T', 'rotor_angle': 'rad', 'rotor_frequency': 'Hz'}, channels: List[Union[str, dict, Isotope (page 479)]], spectral_dimensions: List[SpectralDimension (page 410)] = [SpectralDimension(name=None, description=None, label=None, property_units={'spectral_width': 'Hz', 'reference_offset': 'Hz', 'origin_offset': 'Hz'}, count=1024, spectral_width=25000.0, reference_offset=0.0, origin_offset=None, reciprocal=None, events=[])], affine_matrix: List = None, simulation: Union[CSDM, ndarray] = None, experiment: Union[CSDM, ndarray] = None, magnetic_flux_density: ConstrainedFloatValue = 9.4, rotor_frequency: ConstrainedFloatValue = 0.0, rotor_angle: ConstrainedFloatValue = 0.9553166181245)`

Bases: `BaseNamedMethod1D`

Simulate a Bloch decay central transition selective spectrum.

`classmethod check_event_objects_for_compatibility(default_dim, obj_dim, method_dict)`

Checks Events for compatibility and sets global method attributes

Parameters

- `default_dim` (*dict*) – Dict representation of SpectralDimension in base method
- `obj_dim` (SpectralDimension (page 410)) – User-passed SpectralDimension object to check
- `method_dict` (*dict*) – Dict representation of passed method

`classmethod check_method_compatibility(py_dict)`

Check for events attribute inside the spectral_dimensions. Events are not allowed for NamedMethods.

`dict(**kwargs)`

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

`get_symmetry_pathways(symmetry_element: str)` → List[*SymmetryPathway* (page 483)]

Return a list of symmetry pathways of the method.

Parameters

`symmetry_element` (*str*) – The symmetry element, 'P' or 'D'.

Returns

A list of *SymmetryPathway* (page 483) objects.

Single channel example

Example

```
>>> from mrsimulator.method import Method
>>> method = Method(
...     channels=['1H'],
...     spectral_dimensions=[
...         {
...             "events": [
...                 {
...                     "fraction": 0.5,
...                     "transition_queries": [{"ch1": {"P": [1]}}]
...                 },
...                 {
...                     "fraction": 0.5,
...                     "transition_queries": [{"ch1": {"P": [0]}}]
...                 }
...             ],
...         },
...         {
...             "events": [
...                 {"transition_queries": [{"ch1": {"P": [-1]}}]},
...             ],
...         }
...     ]
... )
>>> pprint(method.get_symmetry_pathways("P"))
[SymmetryPathway(
    ch1(1H): [1] ⟶ [0] ⟶ [-1]
    total: 1.0 ⟶ 0.0 ⟶ -1.0
)]
```

Dual channels example

Example

```
>>> from mrsimulator.method import Method
>>> method = Method(
...     channels=['1H', '13C'],
...     spectral_dimensions=[
...         {
...             "events": [{
...                 "fraction": 0.5,
...                 "transition_queries": [
...                     {"ch1": {"P": [1]}},
...                     {"ch1": {"P": [-1]}},
...                 ]
...             },
...             {
...                 "fraction": 0.5,
...                 "transition_queries": [  # selecting double quantum
...                     {"ch1": {"P": [-1]}, "ch2": {"P": [-1]}},
...                     {"ch1": {"P": [1]}, "ch2": {"P": [1]}},
...                 ]
...             }],
...         },
...         {
...             "events": [{
...                 "transition_queries": [ # selecting single quantum
...                     {"ch1": {"P": [-1]}},
...                 ]
...             }],
...         }
...     ]
... )
>>> pprint(method.get_symmetry_pathways("P"))
[SymmetryPathway(
    ch1(1H): [1] ⟶ [-1] ⟶ [-1]
    ch2(13C): None ⟶ [-1] ⟶ None
    total: 1.0 ⟶ -2.0 ⟶ -1.0
),
 SymmetryPathway(
    ch1(1H): [1] ⟶ [1] ⟶ [-1]
    ch2(13C): None ⟶ [1] ⟶ None
    total: 1.0 ⟶ 2.0 ⟶ -1.0
),
 SymmetryPathway(
    ch1(1H): [-1] ⟶ [-1] ⟶ [-1]
    ch2(13C): None ⟶ [-1] ⟶ None
    total: -1.0 ⟶ -2.0 ⟶ -1.0
),
 SymmetryPathway(
    ch1(1H): [-1] ⟶ [1] ⟶ [-1]
    ch2(13C): None ⟶ [1] ⟶ None
    total: -1.0 ⟶ 2.0 ⟶ -1.0
)]
```

`get_transition_pathways(`*spin_system*`)` → List[*TransitionPathway* (page 482)]

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 388)) – A SpinSystem object.

Returns
A list of *TransitionPathway* (page 482) objects. Each TransitionPathway object is an ordered collection of Transition objects.

Example

```
>>> from mrsimulator import SpinSystem
>>> from mrsimulator.method.lib import ThreeQ_VAS
>>> sys = SpinSystem(sites=[{'isotope': '27Al'}, {'isotope': '29Si'}])
>>> method = ThreeQ_VAS(channels=['27Al'])
>>> pprint(method.get_transition_pathways(sys))
[|1.5, -0.5⟩⟨-1.5, -0.5| ⟶ |-0.5, -0.5⟩⟨0.5, -0.5|, weight=(1+0j),
 |1.5, -0.5⟩⟨-1.5, -0.5| ⟶ |-0.5, 0.5⟩⟨0.5, 0.5|, weight=(1+0j),
 |1.5, 0.5⟩⟨-1.5, 0.5| ⟶ |-0.5, -0.5⟩⟨0.5, -0.5|, weight=(1+0j),
 |1.5, 0.5⟩⟨-1.5, 0.5| ⟶ |-0.5, 0.5⟩⟨0.5, 0.5|, weight=(1+0j)]
```

`json(`*units=True*`)` → dict

Parse the class object to a JSON compliant python dictionary object.

Parameters
`units` – If true, the attribute value is a physical quantity expressed as a string with a number and a unit, else a float.

Returns: dict

`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 403) object.

`plot(`*df=None*, *include_legend=False*`)` → figure

Creates a diagram representing the method. By default, only parameters which vary throughout the method are plotted. Figure can be finley adjusted using matplotlib rcParams.

Parameters
- `df` (*DataFrame*) – DataFrame to plot data from. By default DataFrame is calculated from summary() and will show only parameters which vary throughout the method plus 'p' symmetry pathway and 'd' symmetry pathway if it is not none or defined
- `include_legend` (*bool*) – Optional argument to include a key for event colors. Default is False and no key will be included in figure

Returns
matplotlib.pyplot.figure

Example

```
>>> from mrsimulator.method.lib import BlochDecaySpectrum
>>> method = BlochDecaySpectrum(channels=["13C"])
>>> fig = method.plot()
```

Adjusting Figure Size rcParams

```
>>> import matplotlib as mpl
>>> from mrsimulator.method.lib import FiveQ_VAS
>>> mpl.rcParams["figure.figsize"] = [14, 10]
>>> mpl.rcParams["font.size"] = 14
>>> method = FiveQ_VAS(channels=["27Al"])
>>> fig = method.plot(include_legend=True)
```

Plotting all Parameters, including Constant

```
>>> from mrsimulator.method.lib import FiveQ_VAS
>>> method = FiveQ_VAS(channels=["27Al"])
>>> df = method.summary(drop_constant_columns=False)
>>> fig = method.plot(df=df)
```

`reduced_dict`(*exclude={}*) → dict

Returns a reduced dictionary representation of the class object by removing all key-value pair corresponding to keys listed in the *exclude* argument, and keys with value as None.

Parameters

`exclude` – A list of keys to exclude from the dictionary.

Return: A dict.

`shape()` → tuple

The shape of the method's spectral dimension array.

Returns

tuple

Example

```
>>> from mrsimulator.method import Method
>>> method = Method(
...     channels=['1H'],
...     spectral_dimensions=[{'count': 40}, {'count': 10}]
... )
>>> method.shape()
(40, 10)
```

`summary`(*drop_constant_columns=True*) → DataFrame

Returns a DataFrame giving a summary of the Method. A user can specify optional attributes to include which appear as columns in the DataFrame. A user can also ask to leave out attributes which remain constant throughout the method. Invalid attributes for an Event will be replaced with NAN.

Parameters

`drop_constant_columns` (*(bool)*) – Removes constant properties if True. Default is True.

Returns

Event number as row and property as column. Invalid properties for an event type are filled with np.nan

Columns

- (str) type: Event type
- (int) spec_dim_index: Index of spectral dimension which event belongs to
- (str) label: Event label
- (float) duration: Duration of the DelayEvent
- (float) fraction: Fraction of the SpectralEvent
- (MixingQuery) query: MixingQuery object of the MixingEvent
- (float) magnetic_flux_density: Magnetic flux density during event in Tesla
- (float) rotor_frequency: Rotor frequency during event in Hz
- (float) rotor_angle: Rotor angle during event converted to Degrees
- (FrequencyEnum) freq_contrib: Frequency

Return type

pd.DataFrame df

Example

All Possible Columns

```
>>> from mrsimulator.method.lib import ThreeQ_VAS
>>> method = ThreeQ_VAS(channels=["17O"])
>>> df = method.summary(drop_constant_columns=False)
>>> pprint(list(df.columns))
['type',
 'spec_dim_index',
 'spec_dim_label',
 'label',
 'duration',
 'fraction',
 'query',
 'magnetic_flux_density',
 'rotor_frequency',
 'rotor_angle',
 'freq_contrib',
 'p',
 'd']
```

Example

```
>>> from mrsimulator.method.lib import BlochDecayCTSpectrum
>>> from mrsimulator.method import SpectralDimension
>>> bloch_decay_ct = BlochDecayCTSpectrum(
...     channels=["1H"],
...     rotor_frequency=5000,  # in Hz
...     rotor_angle=54.735 * 3.14159 / 180,  # in rad
...     magnetic_flux_density=9.4,  # in T
...     spectral_dimensions=[
...         SpectralDimension(count=1024, spectral_width=50000, reference_offset=-8000)
...     ],
... )
```

Bloch decay central transition selective method is part of the methods library, but can be constructed with a generic method as follows

```
>>> from mrsimulator.method import Method
>>> bloch_decay_ct = Method(
...     channels=["1H"],
...     rotor_frequency=5000,  # in Hz
...     rotor_angle=54.735 * 3.14159 / 180,  # 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_queries": [{"ch1": {"P": [-1], "D": [0]}}]}],
...         }
...     ],
... )
```

Multi-quantum variable-angle spinning

The following classes are used when simulating a multi-quantum variable-angle spinning spectrum which correlates the frequencies from the symmetric multiple-quantum transition to the central transition frequencies. The p and d pathways for the MQVAS methods are

$$\begin{aligned} p &: 0 \rightarrow M \rightarrow -1 \\ d &: 0 \rightarrow 0 \rightarrow 0 \end{aligned} \tag{17.5}$$

where M is the multiple-quantum number. The value of M depends on the spin quantum number, I, and is listed in Table 17.1.

Affine mapping

The resulting spectrum is sheared and scaled, such that the frequencies along the indirect dimension are given as

$$\langle\Omega\rangle_{\text{MQ-VAS}} = \frac{1}{1+\kappa}\Omega_{m,-m} + \frac{\kappa}{1+\kappa}\Omega_{1/2,-1/2}. \tag{17.6}$$

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 17.1.

Table 17.1: 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.method.lib.ThreeQ_VAS(`*, *name: str = None*, *description: str = None*, *label: str = None*, *property_units: Dict = {'magnetic_flux_density': 'T', 'rotor_angle': 'rad', 'rotor_frequency': 'Hz'}*, *channels: List[Union[str, dict,* Isotope *(page 479)]]*, *spectral_dimensions: List[*SpectralDimension *(page 410)] = [SpectralDimension(name=None, description=None, label=None, property_units={'spectral_width': 'Hz', 'reference_offset': 'Hz', 'origin_offset': 'Hz'}, count=1024, spectral_width=25000.0, reference_offset=0.0, origin_offset=None, reciprocal=None, events=[])]*, *affine_matrix: List = None*, *simulation: Union[CSDM, ndarray] = None*, *experiment: Union[CSDM, ndarray] = None*, *magnetic_flux_density: ConstrainedFloatValue = 9.4*, *rotor_frequency: ConstrainedFloatValue = 1000000000000.0*, *rotor_angle: ConstrainedFloatValue = 0.9553166181245*`)`

Bases: `MQ_VAS`

Simulate a sheared and scaled 3Q 2D variable-angle spinning spectrum.

Note: The attribute *rotor_frequency* cannot be modified for this method and is set to simulate an infinite speed spectrum.

Returns

A *Method* (page 403) instance.

Example

```
>>> method = ThreeQ_VAS(
...     channels=["87Rb"],
...     magnetic_flux_density=7,  # in T
...     spectral_dimensions=[
...         {
...             "count": 128,
...             "spectral_width": 3e3,  # in Hz
...             "reference_offset": -2e3,  # in Hz
...             "label": "Isotropic dimension",
...         },
...         {
...             "count": 512,
...             "spectral_width": 1e4,  # in Hz
...             "reference_offset": -5e3,  # in Hz
...             "label": "MAS dimension",
...         },
...     ],
... )
>>> sys = SpinSystem(sites=[Site(isotope='87Rb')])
>>> method.get_transition_pathways(sys)
[|-1.5⟩⟨1.5| ⟶ |-0.5⟩⟨0.5|, weight=(1+0j)]
```

classmethod check_event_objects_for_compatibility(*default_dim*, *obj_dim*, *method_dict*)

Checks Events for compatibility and sets global method attributes

Parameters

- default_dim (*dict*) – Dict representation of SpectralDimension in base method
- obj_dim (SpectralDimension (page 410)) – User-passed SpectralDimension object to check
- method_dict (*dict*) – Dict representation of passed method

classmethod check_method_compatibility(*py_dict*)

Check for events attribute inside the spectral_dimensions. Events are not allowed for NamedMethods.

dict(***kwargs*)

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

get_symmetry_pathways(*symmetry_element: str*) → List[*SymmetryPathway* (page 483)]

Return a list of symmetry pathways of the method.

Parameters

symmetry_element (*str*) – The symmetry element, 'P' or 'D'.

Returns

A list of *SymmetryPathway* (page 483) objects.

Single channel example

Example

```
>>> from mrsimulator.method import Method
>>> method = Method(
...     channels=['1H'],
...     spectral_dimensions=[
...         {
...             "events": [
...                 {
...                     "fraction": 0.5,
...                     "transition_queries": [{"ch1": {"P": [1]}}]
...                 },
...                 {
...                     "fraction": 0.5,
...                     "transition_queries": [{"ch1": {"P": [0]}}]
...                 }
...             ],
...         },
...         {
...             "events": [
...                 {"transition_queries": [{"ch1": {"P": [-1]}}]},
...             ],
...         }
...     ]
... )
>>> pprint(method.get_symmetry_pathways("P"))
[SymmetryPathway(
    ch1(1H): [1] ⟶ [0] ⟶ [-1]
    total: 1.0 ⟶ 0.0 ⟶ -1.0
)]
```

Dual channels example

Example

```
>>> from mrsimulator.method import Method
>>> method = Method(
...     channels=['1H', '13C'],
...     spectral_dimensions=[
...         {
...             "events": [{
...                 "fraction": 0.5,
...                 "transition_queries": [
...                     {"ch1": {"P": [1]}},
...                     {"ch1": {"P": [-1]}},
...                 ]
...             },
...             {
...                 "fraction": 0.5,
...                 "transition_queries": [  # selecting double quantum
...                     {"ch1": {"P": [-1]}, "ch2": {"P": [-1]}},
...                     {"ch1": {"P": [1]}, "ch2": {"P": [1]}},
```

(continues on next page)

(continued from previous page)

```
...                 ]
...             }],
...         },
...         {
...             "events": [{
...                 "transition_queries": [ # selecting single quantum
...                     {"ch1": {"P": [-1]}},
...                 ]
...             }],
...         }
...     ]
... )
>>> pprint(method.get_symmetry_pathways("P"))
[SymmetryPathway(
    ch1(1H): [1] ⟶ [-1] ⟶ [-1]
    ch2(13C): None ⟶ [-1] ⟶ None
    total: 1.0 ⟶ -2.0 ⟶ -1.0
),
 SymmetryPathway(
    ch1(1H): [1] ⟶ [1] ⟶ [-1]
    ch2(13C): None ⟶ [1] ⟶ None
    total: 1.0 ⟶ 2.0 ⟶ -1.0
),
 SymmetryPathway(
    ch1(1H): [-1] ⟶ [-1] ⟶ [-1]
    ch2(13C): None ⟶ [-1] ⟶ None
    total: -1.0 ⟶ -2.0 ⟶ -1.0
),
 SymmetryPathway(
    ch1(1H): [-1] ⟶ [1] ⟶ [-1]
    ch2(13C): None ⟶ [1] ⟶ None
    total: -1.0 ⟶ 2.0 ⟶ -1.0
)]
```

`get_transition_pathways`(*spin_system*) → List[*TransitionPathway* (page 482)]

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 388)) – A SpinSystem object.

Returns

A list of *TransitionPathway* (page 482) objects. Each TransitionPathway object is an ordered collection of Transition objects.

Example

```
>>> from mrsimulator import SpinSystem
>>> from mrsimulator.method.lib import ThreeQ_VAS
>>> sys = SpinSystem(sites=[{'isotope': '27Al'}, {'isotope': '29Si'}])
>>> method = ThreeQ_VAS(channels=['27Al'])
>>> pprint(method.get_transition_pathways(sys))
[|1.5, -0.5⟩⟨-1.5, -0.5| ⟶ |-0.5, -0.5⟩⟨0.5, -0.5|, weight=(1+0j),
 |1.5, -0.5⟩⟨-1.5, -0.5| ⟶ |-0.5, 0.5⟩⟨0.5, 0.5|, weight=(1+0j),
 |1.5, 0.5⟩⟨-1.5, 0.5| ⟶ |-0.5, -0.5⟩⟨0.5, -0.5|, weight=(1+0j),
 |1.5, 0.5⟩⟨-1.5, 0.5| ⟶ |-0.5, 0.5⟩⟨0.5, 0.5|, weight=(1+0j)]
```

`json`(*units=True*) → dict

Parse the class object to a JSON compliant python dictionary object.

Parameters

`units` – If true, the attribute value is a physical quantity expressed as a string with a number and a unit, else a float.

Returns: dict

`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 403) object.

`plot`(*df=None*, *include_legend=False*) → figure

Creates a diagram representing the method. By default, only parameters which vary throughout the method are plotted. Figure can be finley adjusted using matplotlib rcParams.

Parameters

- `df` (*DataFrame*) – DataFrame to plot data from. By default DataFrame is calculated from summary() and will show only parameters which vary throughout the method plus 'p' symmetry pathway and 'd' symmetry pathway if it is not none or defined
- `include_legend` (*bool*) – Optional argument to include a key for event colors. Default is False and no key will be included in figure

Returns

matplotlib.pyplot.figure

Example

```
>>> from mrsimulator.method.lib import BlochDecaySpectrum
>>> method = BlochDecaySpectrum(channels=["13C"])
>>> fig = method.plot()
```

Adjusting Figure Size rcParams

```
>>> import matplotlib as mpl
>>> from mrsimulator.method.lib import FiveQ_VAS
>>> mpl.rcParams["figure.figsize"] = [14, 10]
```

(continues on next page)

(continued from previous page)

```
>>> mpl.rcParams["font.size"] = 14
>>> method = FiveQ_VAS(channels=["27Al"])
>>> fig = method.plot(include_legend=True)
```

Plotting all Parameters, including Constant

```
>>> from mrsimulator.method.lib import FiveQ_VAS
>>> method = FiveQ_VAS(channels=["27Al"])
>>> df = method.summary(drop_constant_columns=False)
>>> fig = method.plot(df=df)
```

`reduced_dict`(*exclude={}*) → dict

Returns a reduced dictionary representation of the class object by removing all key-value pair corresponding to keys listed in the *exclude* argument, and keys with value as None.

Parameters

`exclude` – A list of keys to exclude from the dictionary.

Return: A dict.

`shape`() → tuple

The shape of the method's spectral dimension array.

Returns

tuple

Example

```
>>> from mrsimulator.method import Method
>>> method = Method(
...     channels=['1H'],
...     spectral_dimensions=[{'count': 40}, {'count': 10}]
... )
>>> method.shape()
(40, 10)
```

`summary`(*drop_constant_columns=True*) → DataFrame

Returns a DataFrame giving a summary of the Method. A user can specify optional attributes to include which appear as columns in the DataFrame. A user can also ask to leave out attributes which remain constant throughout the method. Invalid attributes for an Event will be replaced with NAN.

Parameters

`drop_constant_columns` (*(bool)*) – Removes constant properties if True. Default is True.

Returns

Event number as row and property as column. Invalid properties for an event type are filled with np.nan

Columns

- (str) type: Event type
- (int) spec_dim_index: Index of spectral dimension which event belongs to
- (str) label: Event label
- (float) duration: Duration of the DelayEvent
- (float) fraction: Fraction of the SpectralEvent
- (MixingQuery) query: MixingQuery object of the MixingEvent
- (float) magnetic_flux_density: Magnetic flux density during event in Tesla

- (float) rotor_frequency: Rotor frequency during event in Hz
- (float) rotor_angle: Rotor angle during event converted to Degrees
- (FrequencyEnum) freq_contrib: Frequency

Return type
pd.DataFrame df

Example

All Possible Columns

```
>>> from mrsimulator.method.lib import ThreeQ_VAS
>>> method = ThreeQ_VAS(channels=["17O"])
>>> df = method.summary(drop_constant_columns=False)
>>> pprint(list(df.columns))
['type',
 'spec_dim_index',
 'spec_dim_label',
 'label',
 'duration',
 'fraction',
 'query',
 'magnetic_flux_density',
 'rotor_frequency',
 'rotor_angle',
 'freq_contrib',
 'p',
 'd']
```

Five-quantum variable-angle spinning method

class mrsimulator.method.lib.FiveQ_VAS(*, *name: str = None*, *description: str = None*, *label: str = None*, *property_units: Dict = {'magnetic_flux_density': 'T', 'rotor_angle': 'rad', 'rotor_frequency': 'Hz'}*, *channels: List[Union[str, dict, Isotope (page 479)]]*, *spectral_dimensions: List[SpectralDimension (page 410)] = [SpectralDimension(name=None, description=None, label=None, property_units={'spectral_width': 'Hz', 'reference_offset': 'Hz', 'origin_offset': 'Hz'}, count=1024, spectral_width=25000.0, reference_offset=0.0, origin_offset=None, reciprocal=None, events=[])]*, *affine_matrix: List = None*, *simulation: Union[CSDM, ndarray] = None*, *experiment: Union[CSDM, ndarray] = None*, *magnetic_flux_density: ConstrainedFloatValue = 9.4*, *rotor_frequency: ConstrainedFloatValue = 1000000000000.0*, *rotor_angle: ConstrainedFloatValue = 0.95531661812245*)

Bases: MQ_VAS

Simulate a sheared and scaled 5Q variable-angle spinning spectrum.

Note: The attribute *rotor_frequency* cannot be modified for this method and is set to simulate an infinite speed spectrum.

Returns

A *Method* (page 403) instance.

Example

```
>>> method = FiveQ_VAS(
...     channels=["17O"],
...     magnetic_flux_density=11.7,  # in T
...     spectral_dimensions=[
...         {
...             "count": 512,
...             "spectral_width": 5e3,  # in Hz
...             "reference_offset": -3e3,  # in Hz
...             "label": "Isotropic dimension",
...         },
...         {
...             "count": 512,
...             "spectral_width": 2e4,  # in Hz
...             "reference_offset": -2e3,  # in Hz
...             "label": "MAS dimension",
...         },
...     ],
... )
>>> sys = SpinSystem(sites=[Site(isotope='17O')])
>>> method.get_transition_pathways(sys)
[|-2.5⟩⟨2.5| ⟶ |-0.5⟩⟨0.5|, weight=(1+0j)]
```

`classmethod check_event_objects_for_compatibility(`*default_dim*, *obj_dim*, *method_dict*`)`

Checks Events for compatibility and sets global method attributes

Parameters

- `default_dim` (*dict*) – Dict representation of SpectralDimension in base method
- `obj_dim` (`SpectralDimension` (page 410)) – User-passed SpectralDimension object to check
- `method_dict` (*dict*) – Dict representation of passed method

`classmethod check_method_compatibility(`*py_dict*`)`

Check for events attribute inside the spectral_dimensions. Events are not allowed for NamedMethods.

`dict(`***kwargs*`)`

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

`get_symmetry_pathways(`*symmetry_element: str*`)` → List[*SymmetryPathway* (page 483)]

Return a list of symmetry pathways of the method.

Parameters

`symmetry_element` (*str*) – The symmetry element, 'P' or 'D'.

Returns

A list of *SymmetryPathway* (page 483) objects.

Single channel example

Example

```
>>> from mrsimulator.method import Method
>>> method = Method(
...     channels=['1H'],
...     spectral_dimensions=[
...         {
...             "events": [
...                 {
...                     "fraction": 0.5,
...                     "transition_queries": [{"ch1": {"P": [1]}}]
...                 },
...                 {
...                     "fraction": 0.5,
...                     "transition_queries": [{"ch1": {"P": [0]}}]
...                 }
...             ],
...         },
...         {
...             "events": [
...                 {"transition_queries": [{"ch1": {"P": [-1]}}]},
...             ],
...         }
...     ]
... )
>>> pprint(method.get_symmetry_pathways("P"))
[SymmetryPathway(
    ch1(1H): [1] ⟶ [0] ⟶ [-1]
    total: 1.0 ⟶ 0.0 ⟶ -1.0
)]
```

Dual channels example

Example

```
>>> from mrsimulator.method import Method
>>> method = Method(
...     channels=['1H', '13C'],
...     spectral_dimensions=[
...         {
...             "events": [{
...                 "fraction": 0.5,
...                 "transition_queries": [
...                     {"ch1": {"P": [1]}},
...                     {"ch1": {"P": [-1]}},
...                 ]
...             },
...             {
...                 "fraction": 0.5,
...                 "transition_queries": [  # selecting double quantum
...                     {"ch1": {"P": [-1]}, "ch2": {"P": [-1]}},
...                     {"ch1": {"P": [1]}, "ch2": {"P": [1]}},
```

(continues on next page)

(continued from previous page)

```
...                 ]
...             }],
...         },
...         {
...             "events": [{
...                 "transition_queries": [ # selecting single quantum
...                     {"ch1": {"P": [-1]}},
...                 ]
...             }],
...         }
...     ]
... )
>>> pprint(method.get_symmetry_pathways("P"))
[SymmetryPathway(
    ch1(1H): [1] ⟶ [-1] ⟶ [-1]
    ch2(13C): None ⟶ [-1] ⟶ None
    total: 1.0 ⟶ -2.0 ⟶ -1.0
),
 SymmetryPathway(
    ch1(1H): [1] ⟶ [1] ⟶ [-1]
    ch2(13C): None ⟶ [1] ⟶ None
    total: 1.0 ⟶ 2.0 ⟶ -1.0
),
 SymmetryPathway(
    ch1(1H): [-1] ⟶ [-1] ⟶ [-1]
    ch2(13C): None ⟶ [-1] ⟶ None
    total: -1.0 ⟶ -2.0 ⟶ -1.0
),
 SymmetryPathway(
    ch1(1H): [-1] ⟶ [1] ⟶ [-1]
    ch2(13C): None ⟶ [1] ⟶ None
    total: -1.0 ⟶ 2.0 ⟶ -1.0
)]
```

`get_transition_pathways`(*spin_system*) → List[*TransitionPathway* (page 482)]

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 388)) – A SpinSystem object.

Returns

A list of *TransitionPathway* (page 482) objects. Each TransitionPathway object is an ordered collection of Transition objects.

Example

```
>>> from mrsimulator import SpinSystem
>>> from mrsimulator.method.lib import ThreeQ_VAS
>>> sys = SpinSystem(sites=[{'isotope': '27Al'}, {'isotope': '29Si'}])
>>> method = ThreeQ_VAS(channels=['27Al'])
>>> pprint(method.get_transition_pathways(sys))
[|1.5, -0.5⟩⟨-1.5, -0.5| ⟶ |-0.5, -0.5⟩⟨0.5, -0.5|, weight=(1+0j),
 |1.5, -0.5⟩⟨-1.5, -0.5| ⟶ |-0.5, 0.5⟩⟨0.5, 0.5|, weight=(1+0j),
 |1.5, 0.5⟩⟨-1.5, 0.5| ⟶ |-0.5, -0.5⟩⟨0.5, -0.5|, weight=(1+0j),
 |1.5, 0.5⟩⟨-1.5, 0.5| ⟶ |-0.5, 0.5⟩⟨0.5, 0.5|, weight=(1+0j)]
```

`json`(*units=True*) → dict

Parse the class object to a JSON compliant python dictionary object.

Parameters

`units` – If true, the attribute value is a physical quantity expressed as a string with a number and a unit, else a float.

Returns: dict

`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 403) object.

`plot`(*df=None*, *include_legend=False*) → figure

Creates a diagram representing the method. By default, only parameters which vary throughout the method are plotted. Figure can be finley adjusted using matplotlib rcParams.

Parameters

- `df` (*DataFrame*) – DataFrame to plot data from. By default DataFrame is calculated from summary() and will show only parameters which vary throughout the method plus 'p' symmetry pathway and 'd' symmetry pathway if it is not none or defined
- `include_legend` (*bool*) – Optional argument to include a key for event colors. Default is False and no key will be included in figure

Returns

matplotlib.pyplot.figure

Example

```
>>> from mrsimulator.method.lib import BlochDecaySpectrum
>>> method = BlochDecaySpectrum(channels=["13C"])
>>> fig = method.plot()
```

Adjusting Figure Size rcParams

```
>>> import matplotlib as mpl
>>> from mrsimulator.method.lib import FiveQ_VAS
>>> mpl.rcParams["figure.figsize"] = [14, 10]
```

(continues on next page)

(continued from previous page)

```
>>> mpl.rcParams["font.size"] = 14
>>> method = FiveQ_VAS(channels=["27Al"])
>>> fig = method.plot(include_legend=True)
```

Plotting all Parameters, including Constant

```
>>> from mrsimulator.method.lib import FiveQ_VAS
>>> method = FiveQ_VAS(channels=["27Al"])
>>> df = method.summary(drop_constant_columns=False)
>>> fig = method.plot(df=df)
```

`reduced_dict(`*exclude={}*`)` → dict

Returns a reduced dictionary representation of the class object by removing all key-value pair corresponding to keys listed in the *exclude* argument, and keys with value as None.

Parameters

`exclude` – A list of keys to exclude from the dictionary.

Return: A dict.

`shape()` → tuple

The shape of the method's spectral dimension array.

Returns

tuple

Example

```
>>> from mrsimulator.method import Method
>>> method = Method(
...     channels=['1H'],
...     spectral_dimensions=[{'count': 40}, {'count': 10}]
... )
>>> method.shape()
(40, 10)
```

`summary(`*drop_constant_columns=True*`)` → DataFrame

Returns a DataFrame giving a summary of the Method. A user can specify optional attributes to include which appear as columns in the DataFrame. A user can also ask to leave out attributes which remain constant throughout the method. Invalid attributes for an Event will be replaced with NAN.

Parameters

`drop_constant_columns` (*(bool)*) – Removes constant properties if True. Default is True.

Returns

Event number as row and property as column. Invalid properties for an event type are filled with np.nan

Columns

- (str) type: Event type
- (int) spec_dim_index: Index of spectral dimension which event belongs to
- (str) label: Event label
- (float) duration: Duration of the DelayEvent
- (float) fraction: Fraction of the SpectralEvent
- (MixingQuery) query: MixingQuery object of the MixingEvent
- (float) magnetic_flux_density: Magnetic flux density during event in Tesla

- (float) rotor_frequency: Rotor frequency during event in Hz
- (float) rotor_angle: Rotor angle during event converted to Degrees
- (FrequencyEnum) freq_contrib: Frequency

Return type
pd.DataFrame df

Example

All Possible Columns

```
>>> from mrsimulator.method.lib import ThreeQ_VAS
>>> method = ThreeQ_VAS(channels=["17O"])
>>> df = method.summary(drop_constant_columns=False)
>>> pprint(list(df.columns))
['type',
 'spec_dim_index',
 'spec_dim_label',
 'label',
 'duration',
 'fraction',
 'query',
 'magnetic_flux_density',
 'rotor_frequency',
 'rotor_angle',
 'freq_contrib',
 'p',
 'd']
```

Seven-quantum variable-angle spinning method

```
class mrsimulator.method.lib.SevenQ_VAS(*, name: str = None, description: str = None, label: str = None,
                                        property_units: Dict = {'magnetic_flux_density': 'T',
                                        'rotor_angle': 'rad', 'rotor_frequency': 'Hz'}, channels:
                                        List[Union[str, dict, Isotope (page 479)]], spectral_dimensions:
                                        List[SpectralDimension (page 410)] =
                                        [SpectralDimension(name=None, description=None, label=None,
                                        property_units={'spectral_width': 'Hz', 'reference_offset': 'Hz',
                                        'origin_offset': 'Hz'}, count=1024, spectral_width=25000.0,
                                        reference_offset=0.0, origin_offset=None, reciprocal=None,
                                        events=[])], affine_matrix: List = None, simulation: Union[CSDM,
                                        ndarray] = None, experiment: Union[CSDM, ndarray] = None,
                                        magnetic_flux_density: ConstrainedFloatValue = 9.4,
                                        rotor_frequency: ConstrainedFloatValue = 1000000000000.0,
                                        rotor_angle: ConstrainedFloatValue = 0.9553166181245)
```

Bases: `MQ_VAS`

Simulate a sheared and scaled 7Q variable-angle spinning spectrum.

Note: The attribute *rotor_frequency* cannot be modified for this method and is set to simulate an infinite speed spectrum.

Returns

A *Method* (page 403) instance.

Example

```
>>> method = SevenQ_VAS(
...     channels=["51V"],
...     magnetic_flux_density=4.7,  # in T
...     spectral_dimensions=[
...         {
...             "count": 256,
...             "spectral_width": 1e3,  # in Hz
...             "reference_offset": 1e3,  # in Hz
...             "label": "Isotropic dimension",
...         },
...         {
...             "count": 1024,
...             "spectral_width": 1e4,  # in Hz
...             "reference_offset": -2e3,  # in Hz
...             "label": "MAS dimension",
...         },
...     ],
... )
>>> sys = SpinSystem(sites=[Site(isotope='51V')])
>>> method.get_transition_pathways(sys)
[|-3.5⟩⟨3.5| ⟶ |-0.5⟩⟨0.5|, weight=(1+0j)]
```

`classmethod check_event_objects_for_compatibility(`*default_dim*, *obj_dim*, *method_dict*`)`

Checks Events for compatibility and sets global method attributes

Parameters

- `default_dim` (*dict*) – Dict representation of SpectralDimension in base method
- `obj_dim` (`SpectralDimension` (page 410)) – User-passed SpectralDimension object to check
- `method_dict` (*dict*) – Dict representation of passed method

`classmethod check_method_compatibility(`*py_dict*`)`

Check for events attribute inside the spectral_dimensions. Events are not allowed for NamedMethods.

`dict(`***kwargs*`)`

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

`get_symmetry_pathways(`*symmetry_element: str*`)` → List[*SymmetryPathway* (page 483)]

Return a list of symmetry pathways of the method.

Parameters

`symmetry_element` (*str*) – The symmetry element, 'P' or 'D'.

Returns

A list of *SymmetryPathway* (page 483) objects.

Single channel example

Example

```
>>> from mrsimulator.method import Method
>>> method = Method(
...     channels=['1H'],
...     spectral_dimensions=[
...         {
...             "events": [
...                 {
...                     "fraction": 0.5,
...                     "transition_queries": [{"ch1": {"P": [1]}}]
...                 },
...                 {
...                     "fraction": 0.5,
...                     "transition_queries": [{"ch1": {"P": [0]}}]
...                 }
...             ],
...         },
...         {
...             "events": [
...                 {"transition_queries": [{"ch1": {"P": [-1]}}]},
...             ],
...         }
...     ]
... )
>>> pprint(method.get_symmetry_pathways("P"))
[SymmetryPathway(
    ch1(1H): [1] ⟶ [0] ⟶ [-1]
    total: 1.0 ⟶ 0.0 ⟶ -1.0
)]
```

Dual channels example

Example

```
>>> from mrsimulator.method import Method
>>> method = Method(
...     channels=['1H', '13C'],
...     spectral_dimensions=[
...         {
...             "events": [{
...                 "fraction": 0.5,
...                 "transition_queries": [
...                     {"ch1": {"P": [1]}},
...                     {"ch1": {"P": [-1]}},
...                 ]
...             },
...             {
...                 "fraction": 0.5,
...                 "transition_queries": [  # selecting double quantum
...                     {"ch1": {"P": [-1]}, "ch2": {"P": [-1]}},
...                     {"ch1": {"P": [1]}, "ch2": {"P": [1]}},
```

(continues on next page)

(continued from previous page)

```
...                 ]
...             }],
...         },
...         {
...             "events": [{
...                 "transition_queries": [ # selecting single quantum
...                     {"ch1": {"P": [-1]}},
...                 ]
...             }],
...         }
...     ]
... )
>>> pprint(method.get_symmetry_pathways("P"))
[SymmetryPathway(
    ch1(1H): [1] ⟶ [-1] ⟶ [-1]
    ch2(13C): None ⟶ [-1] ⟶ None
    total: 1.0 ⟶ -2.0 ⟶ -1.0
),
 SymmetryPathway(
    ch1(1H): [1] ⟶ [1] ⟶ [-1]
    ch2(13C): None ⟶ [1] ⟶ None
    total: 1.0 ⟶ 2.0 ⟶ -1.0
),
 SymmetryPathway(
    ch1(1H): [-1] ⟶ [-1] ⟶ [-1]
    ch2(13C): None ⟶ [-1] ⟶ None
    total: -1.0 ⟶ -2.0 ⟶ -1.0
),
 SymmetryPathway(
    ch1(1H): [-1] ⟶ [1] ⟶ [-1]
    ch2(13C): None ⟶ [1] ⟶ None
    total: -1.0 ⟶ 2.0 ⟶ -1.0
)]
```

`get_transition_pathways`(*spin_system*) → List[*TransitionPathway* (page 482)]

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 388)) – A SpinSystem object.

Returns

A list of *TransitionPathway* (page 482) objects. Each TransitionPathway object is an ordered collection of Transition objects.

Example

```
>>> from mrsimulator import SpinSystem
>>> from mrsimulator.method.lib import ThreeQ_VAS
>>> sys = SpinSystem(sites=[{'isotope': '27Al'}, {'isotope': '29Si'}])
>>> method = ThreeQ_VAS(channels=['27Al'])
>>> pprint(method.get_transition_pathways(sys))
[|1.5, -0.5⟩⟨-1.5, -0.5| ⟶ |-0.5, -0.5⟩⟨0.5, -0.5|, weight=(1+0j),
 |1.5, -0.5⟩⟨-1.5, -0.5| ⟶ |-0.5, 0.5⟩⟨0.5, 0.5|, weight=(1+0j),
 |1.5, 0.5⟩⟨-1.5, 0.5| ⟶ |-0.5, -0.5⟩⟨0.5, -0.5|, weight=(1+0j),
 |1.5, 0.5⟩⟨-1.5, 0.5| ⟶ |-0.5, 0.5⟩⟨0.5, 0.5|, weight=(1+0j)]
```

`json`(*units=True*) → dict

Parse the class object to a JSON compliant python dictionary object.

Parameters

`units` – If true, the attribute value is a physical quantity expressed as a string with a number and a unit, else a float.

Returns: dict

`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 403) object.

`plot`(*df=None*, *include_legend=False*) → figure

Creates a diagram representing the method. By default, only parameters which vary throughout the method are plotted. Figure can be finley adjusted using matplotlib rcParams.

Parameters

- `df` (*DataFrame*) – DataFrame to plot data from. By default DataFrame is calculated from summary() and will show only parameters which vary throughout the method plus 'p' symmetry pathway and 'd' symmetry pathway if it is not none or defined
- `include_legend` (*bool*) – Optional argument to include a key for event colors. Default is False and no key will be included in figure

Returns

matplotlib.pyplot.figure

Example

```
>>> from mrsimulator.method.lib import BlochDecaySpectrum
>>> method = BlochDecaySpectrum(channels=["13C"])
>>> fig = method.plot()
```

Adjusting Figure Size rcParams

```
>>> import matplotlib as mpl
>>> from mrsimulator.method.lib import FiveQ_VAS
>>> mpl.rcParams["figure.figsize"] = [14, 10]
```

(continues on next page)

(continued from previous page)

```
>>> mpl.rcParams["font.size"] = 14
>>> method = FiveQ_VAS(channels=["27Al"])
>>> fig = method.plot(include_legend=True)
```

Plotting all Parameters, including Constant

```
>>> from mrsimulator.method.lib import FiveQ_VAS
>>> method = FiveQ_VAS(channels=["27Al"])
>>> df = method.summary(drop_constant_columns=False)
>>> fig = method.plot(df=df)
```

`reduced_dict(`*exclude={}*`)` → dict

Returns a reduced dictionary representation of the class object by removing all key-value pair corresponding to keys listed in the *exclude* argument, and keys with value as None.

Parameters

`exclude` – A list of keys to exclude from the dictionary.

Return: A dict.

`shape()` → tuple

The shape of the method's spectral dimension array.

Returns

tuple

Example

```
>>> from mrsimulator.method import Method
>>> method = Method(
...     channels=['1H'],
...     spectral_dimensions=[{'count': 40}, {'count': 10}]
... )
>>> method.shape()
(40, 10)
```

`summary(`*drop_constant_columns=True*`)` → DataFrame

Returns a DataFrame giving a summary of the Method. A user can specify optional attributes to include which appear as columns in the DataFrame. A user can also ask to leave out attributes which remain constant throughout the method. Invalid attributes for an Event will be replaced with NAN.

Parameters

`drop_constant_columns` (*(bool)*) – Removes constant properties if True. Default is True.

Returns

Event number as row and property as column. Invalid properties for an event type are filled with np.nan

Columns

- (str) type: Event type
- (int) spec_dim_index: Index of spectral dimension which event belongs to
- (str) label: Event label
- (float) duration: Duration of the DelayEvent
- (float) fraction: Fraction of the SpectralEvent
- (MixingQuery) query: MixingQuery object of the MixingEvent
- (float) magnetic_flux_density: Magnetic flux density during event in Tesla

- (float) rotor_frequency: Rotor frequency during event in Hz
- (float) rotor_angle: Rotor angle during event converted to Degrees
- (FrequencyEnum) freq_contrib: Frequency

Return type
pd.DataFrame df

Example

All Possible Columns

```
>>> from mrsimulator.method.lib import ThreeQ_VAS
>>> method = ThreeQ_VAS(channels=["17O"])
>>> df = method.summary(drop_constant_columns=False)
>>> pprint(list(df.columns))
['type',
 'spec_dim_index',
 'spec_dim_label',
 'label',
 'duration',
 'fraction',
 'query',
 'magnetic_flux_density',
 'rotor_frequency',
 'rotor_angle',
 'freq_contrib',
 'p',
 'd']
```

Satellite-transition variable-angle spinning (ST-VAS)

The following classes are used when simulating a satellite-transition variable-angle spinning spectrum which correlates the frequencies from the satellite transitions to the central transition frequencies. The p and d pathways for the ST-VAS methods are

$$\begin{aligned} p &: 0 \rightarrow -1 \rightarrow -1 \\ d &: 0 \rightarrow \pm d_0 \rightarrow 0 \end{aligned} \tag{17.7}$$

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 17.2.

Affine mapping

The resulting spectrum is sheared and scaled, such that the frequencies along indirect dimension are given as

$$\langle\Omega\rangle_{\text{ST-VAS}} = \frac{1}{1+\kappa}\Omega_{m,m-1} + \frac{\kappa}{1+\kappa}\Omega_{1/2,-1/2}. \tag{17.8}$$

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 17.2.

Table 17.2: 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	$(\frac{3}{2} \rightarrow \frac{1}{2})$, $(-\frac{1}{2} \rightarrow -\frac{3}{2})$	2	24/27
5/2	$(-\frac{3}{2} \rightarrow -\frac{1}{2})$, $(\frac{1}{2} \rightarrow \frac{3}{2})$	2	21/72
5/2	$(\frac{5}{2} \rightarrow \frac{3}{2})$, $(-\frac{3}{2} \rightarrow -\frac{5}{2})$	4	132/72
7/2	$(-\frac{3}{2} \rightarrow -\frac{1}{2})$, $(\frac{1}{2} \rightarrow \frac{3}{2})$	2	84/135
7/2	$(-\frac{5}{2} \rightarrow -\frac{3}{2})$, $(\frac{3}{2} \rightarrow \frac{5}{2})$	4	69/135
9/2	$(-\frac{3}{2} \rightarrow -\frac{1}{2})$, $(\frac{1}{2} \rightarrow \frac{3}{2})$	2	165/216
9/2	$(-\frac{5}{2} \rightarrow -\frac{3}{2})$, $(\frac{3}{2} \rightarrow \frac{5}{2})$	4	12/216

Inner satellite variable-angle spinning method

class mrsimulator.method.lib.ST1_VAS(*, *name: str = None*, *description: str = None*, *label: str = None*, *property_units: Dict = {'magnetic_flux_density': 'T', 'rotor_angle': 'rad', 'rotor_frequency': 'Hz'}*, *channels: List[Union[str, dict, Isotope (page 479)]]*, *spectral_dimensions: List[SpectralDimension (page 410)] = [SpectralDimension(name=None, description=None, label=None, property_units={'spectral_width': 'Hz', 'reference_offset': 'Hz', 'origin_offset': 'Hz'}, count=1024, spectral_width=25000.0, reference_offset=0.0, origin_offset=None, reciprocal=None, events=[])]*, *affine_matrix: List = None*, *simulation: Union[CSDM, ndarray] = None*, *experiment: Union[CSDM, ndarray] = None*, *magnetic_flux_density: ConstrainedFloatValue = 9.4*, *rotor_frequency: ConstrainedFloatValue = 1000000000000.0*, *rotor_angle: ConstrainedFloatValue = 0.9553166181245*)

Bases: ST_VAS

Simulate a sheared and scaled inner satellite and central transition correlation spectrum.

Note: The attribute *rotor_frequency* cannot be modified for this method and is set to simulate an infinite speed spectrum.

Returns

A *Method* (page 403) instance.

Example

```
>>> method = ST1_VAS(
...     channels=["87Rb"],
...     magnetic_flux_density=9.4,  # in T
...     spectral_dimensions=[
...         {
...             "count": 128,
...             "spectral_width": 1e3,  # in Hz
```

(continues on next page)

(continued from previous page)

```
...             "reference_offset": -5e3,  # in Hz
...             "label": "Isotropic dimension",
...         },
...         {
...             "count": 256,
...             "spectral_width": 1e4,  # in Hz
...             "reference_offset": -3e3,  # in Hz
...             "label": "MAS dimension",
...         },
...     ],
... )
>>> sys = SpinSystem(sites=[Site(isotope='87Rb')])
>>> pprint(method.get_transition_pathways(sys))
[|-1.5⟩⟨-0.5| ⟶ |-0.5⟩⟨0.5|, weight=(1+0j),
 |0.5⟩⟨1.5| ⟶ |-0.5⟩⟨0.5|, weight=(1+0j)]
```

classmethod check_event_objects_for_compatibility(*default_dim*, *obj_dim*, *method_dict*)

Checks Events for compatibility and sets global method attributes

Parameters

- default_dim (*dict*) – Dict representation of SpectralDimension in base method
- obj_dim (SpectralDimension (page 410)) – User-passed SpectralDimension object to check
- method_dict (*dict*) – Dict representation of passed method

classmethod check_method_compatibility(*py_dict*)

Check for events attribute inside the spectral_dimensions. Events are not allowed for NamedMethods.

dict(***kwargs*)

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

get_symmetry_pathways(*symmetry_element: str*) → List[*SymmetryPathway* (page 483)]

Return a list of symmetry pathways of the method.

Parameters

symmetry_element (*str*) – The symmetry element, 'P' or 'D'.

Returns

A list of *SymmetryPathway* (page 483) objects.

Single channel example

Example

```
>>> from mrsimulator.method import Method
>>> method = Method(
...     channels=['1H'],
...     spectral_dimensions=[
...         {
...             "events": [
...                 {
...                     "fraction": 0.5,
...                     "transition_queries": [{"ch1": {"P": [1]}}]
...                 },
...                 {
```

(continues on next page)

(continued from previous page)

```
...                     "fraction": 0.5,
...                     "transition_queries": [{"ch1": {"P": [0]}}]
...                 }
...             ],
...         },
...         {
...             "events": [
...                 {"transition_queries": [{"ch1": {"P": [-1]}}]},
...             ],
...         }
...     ]
... )
>>> pprint(method.get_symmetry_pathways("P"))
[SymmetryPathway(
    ch1(1H): [1] ⟶ [0] ⟶ [-1]
    total: 1.0 ⟶ 0.0 ⟶ -1.0
)]
```

Dual channels example

Example

```
>>> from mrsimulator.method import Method
>>> method = Method(
...     channels=['1H', '13C'],
...     spectral_dimensions=[
...         {
...             "events": [{
...                 "fraction": 0.5,
...                 "transition_queries": [
...                     {"ch1": {"P": [1]}},
...                     {"ch1": {"P": [-1]}},
...                 ]
...             },
...             {
...                 "fraction": 0.5,
...                 "transition_queries": [  # selecting double quantum
...                     {"ch1": {"P": [-1]}, "ch2": {"P": [-1]}},
...                     {"ch1": {"P": [1]}, "ch2": {"P": [1]}},
...                 ]
...             }],
...         },
...         {
...             "events": [{
...                 "transition_queries": [ # selecting single quantum
...                     {"ch1": {"P": [-1]}},
...                 ]
...             }],
...         }
...     ]
... )
```

(continues on next page)

(continued from previous page)

```
>>> pprint(method.get_symmetry_pathways("P"))
[SymmetryPathway(
    ch1(1H): [1] ⟶ [-1] ⟶ [-1]
    ch2(13C): None ⟶ [-1] ⟶ None
    total: 1.0 ⟶ -2.0 ⟶ -1.0
),
 SymmetryPathway(
    ch1(1H): [1] ⟶ [1] ⟶ [-1]
    ch2(13C): None ⟶ [1] ⟶ None
    total: 1.0 ⟶ 2.0 ⟶ -1.0
),
 SymmetryPathway(
    ch1(1H): [-1] ⟶ [-1] ⟶ [-1]
    ch2(13C): None ⟶ [-1] ⟶ None
    total: -1.0 ⟶ -2.0 ⟶ -1.0
),
 SymmetryPathway(
    ch1(1H): [-1] ⟶ [1] ⟶ [-1]
    ch2(13C): None ⟶ [1] ⟶ None
    total: -1.0 ⟶ 2.0 ⟶ -1.0
)]
```

get_transition_pathways(*spin_system*) → List[*TransitionPathway* (page 482)]

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 388)) – A SpinSystem object.

Returns

A list of *TransitionPathway* (page 482) objects. Each TransitionPathway object is an ordered collection of Transition objects.

Example

```
>>> from mrsimulator import SpinSystem
>>> from mrsimulator.method.lib import ThreeQ_VAS
>>> sys = SpinSystem(sites=[{'isotope': '27Al'}, {'isotope': '29Si'}])
>>> method = ThreeQ_VAS(channels=['27Al'])
>>> pprint(method.get_transition_pathways(sys))
[|1.5, -0.5⟩⟨-1.5, -0.5| ⟶ |-0.5, -0.5⟩⟨0.5, -0.5|, weight=(1+0j),
 |1.5, -0.5⟩⟨-1.5, -0.5| ⟶ |-0.5, 0.5⟩⟨0.5, 0.5|, weight=(1+0j),
 |1.5, 0.5⟩⟨-1.5, 0.5| ⟶ |-0.5, -0.5⟩⟨0.5, -0.5|, weight=(1+0j),
 |1.5, 0.5⟩⟨-1.5, 0.5| ⟶ |-0.5, 0.5⟩⟨0.5, 0.5|, weight=(1+0j)]
```

json(*units=True*) → dict

Parse the class object to a JSON compliant python dictionary object.

Parameters

units – If true, the attribute value is a physical quantity expressed as a string with a number and a unit, else a float.

Returns: dict

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 403) object.

plot(*df=None*, *include_legend=False*) → figure

Creates a diagram representing the method. By default, only parameters which vary throughout the method are plotted. Figure can be finley adjusted using matplotlib rcParams.

Parameters

- df (*DataFrame*) – DataFrame to plot data from. By default DataFrame is calculated from summary() and will show only parameters which vary throughout the method plus 'p' symmetry pathway and 'd' symmetry pathway if it is not none or defined
- include_legend (*bool*) – Optional argument to include a key for event colors. Default is False and no key will be included in figure

Returns

matplotlib.pyplot.figure

Example

```
>>> from mrsimulator.method.lib import BlochDecaySpectrum
>>> method = BlochDecaySpectrum(channels=["13C"])
>>> fig = method.plot()
```

Adjusting Figure Size rcParams

```
>>> import matplotlib as mpl
>>> from mrsimulator.method.lib import FiveQ_VAS
>>> mpl.rcParams["figure.figsize"] = [14, 10]
>>> mpl.rcParams["font.size"] = 14
>>> method = FiveQ_VAS(channels=["27Al"])
>>> fig = method.plot(include_legend=True)
```

Plotting all Parameters, including Constant

```
>>> from mrsimulator.method.lib import FiveQ_VAS
>>> method = FiveQ_VAS(channels=["27Al"])
>>> df = method.summary(drop_constant_columns=False)
>>> fig = method.plot(df=df)
```

reduced_dict(*exclude={}*) → dict

Returns a reduced dictionary representation of the class object by removing all key-value pair corresponding to keys listed in the *exclude* argument, and keys with value as None.

Parameters

exclude – A list of keys to exclude from the dictionary.

Return: A dict.

shape() → tuple

The shape of the method's spectral dimension array.

Returns
tuple

Example

```
>>> from mrsimulator.method import Method
>>> method = Method(
...     channels=['1H'],
...     spectral_dimensions=[{'count': 40}, {'count': 10}]
... )
>>> method.shape()
(40, 10)
```

summary(*drop_constant_columns=True*) → DataFrame

Returns a DataFrame giving a summary of the Method. A user can specify optional attributes to include which appear as columns in the DataFrame. A user can also ask to leave out attributes which remain constant throughout the method. Invalid attributes for an Event will be replaced with NAN.

Parameters
drop_constant_columns (*bool*) – Removes constant properties if True. Default is True.

Returns
Event number as row and property as column. Invalid properties for an event type are filled with np.nan

Columns
- (str) type: Event type
- (int) spec_dim_index: Index of spectral dimension which event belongs to
- (str) label: Event label
- (float) duration: Duration of the DelayEvent
- (float) fraction: Fraction of the SpectralEvent
- (MixingQuery) query: MixingQuery object of the MixingEvent
- (float) magnetic_flux_density: Magnetic flux density during event in Tesla
- (float) rotor_frequency: Rotor frequency during event in Hz
- (float) rotor_angle: Rotor angle during event converted to Degrees
- (FrequencyEnum) freq_contrib: Frequency

Return type
pd.DataFrame df

Example

All Possible Columns

```
>>> from mrsimulator.method.lib import ThreeQ_VAS
>>> method = ThreeQ_VAS(channels=["17O"])
>>> df = method.summary(drop_constant_columns=False)
>>> pprint(list(df.columns))
['type',
 'spec_dim_index',
 'spec_dim_label',
 'label',
 'duration',
 'fraction',
 'query',
```

(continues on next page)

(continued from previous page)

```
'magnetic_flux_density',
'rotor_frequency',
'rotor_angle',
'freq_contrib',
'p',
'd']
```

Second to inner satellite variable-angle spinning method

class mrsimulator.method.lib.ST2_VAS(*, *name: str = None*, *description: str = None*, *label: str = None*, *property_units: Dict = {'magnetic_flux_density': 'T', 'rotor_angle': 'rad', 'rotor_frequency': 'Hz'}*, *channels: List[Union[str, dict, Isotope (page 479)]]*, *spectral_dimensions: List[SpectralDimension (page 410)] = [SpectralDimension(name=None, description=None, label=None, property_units={'spectral_width': 'Hz', 'reference_offset': 'Hz', 'origin_offset': 'Hz'}, count=1024, spectral_width=25000.0, reference_offset=0.0, origin_offset=None, reciprocal=None, events=[])]*, *affine_matrix: List = None*, *simulation: Union[CSDM, ndarray] = None*, *experiment: Union[CSDM, ndarray] = None*, *magnetic_flux_density: ConstrainedFloatValue = 9.4*, *rotor_frequency: ConstrainedFloatValue = 1000000000000.0*, *rotor_angle: ConstrainedFloatValue = 0.9553166181245*)

Bases: ST_VAS

Simulate a sheared and scaled second to inner satellite and central transition correlation spectrum.

Note: The attribute *rotor_frequency* cannot be modified for this method and is set to simulate an infinite speed spectrum.

Returns

A *Method* (page 403) 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
```

(continues on next page)

(continued from previous page)

```
...             "label": "MAS dimension",
...         },
...     ],
... )
>>> sys = SpinSystem(sites=[Site(isotope='17O')])
>>> pprint(method.get_transition_pathways(sys))
[|-2.5⟩⟨-1.5| ⟶ |-0.5⟩⟨0.5|, weight=(1+0j),
 |1.5⟩⟨2.5| ⟶ |-0.5⟩⟨0.5|, weight=(1+0j)]
```

classmethod check_event_objects_for_compatibility(*default_dim*, *obj_dim*, *method_dict*)

Checks Events for compatibility and sets global method attributes

Parameters

- default_dim (*dict*) – Dict representation of SpectralDimension in base method
- obj_dim (SpectralDimension (page 410)) – User-passed SpectralDimension object to check
- method_dict (*dict*) – Dict representation of passed method

classmethod check_method_compatibility(*py_dict*)

Check for events attribute inside the spectral_dimensions. Events are not allowed for NamedMethods.

dict(***kwargs*)

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

get_symmetry_pathways(*symmetry_element: str*) → List[*SymmetryPathway* (page 483)]

Return a list of symmetry pathways of the method.

Parameters

symmetry_element (*str*) – The symmetry element, 'P' or 'D'.

Returns

A list of *SymmetryPathway* (page 483) objects.

Single channel example

Example

```
>>> from mrsimulator.method import Method
>>> method = Method(
...     channels=['1H'],
...     spectral_dimensions=[
...         {
...             "events": [
...                 {
...                     "fraction": 0.5,
...                     "transition_queries": [{"ch1": {"P": [1]}}]
...                 },
...                 {
...                     "fraction": 0.5,
...                     "transition_queries": [{"ch1": {"P": [0]}}]
...                 }
...             ],
...         },
...         {
...             "events": [
```

(continues on next page)

(continued from previous page)

```
...                 {"transition_queries": [{"ch1": {"P": [-1]}}]},
...             ],
...         }
...     ]
... )
>>> pprint(method.get_symmetry_pathways("P"))
[SymmetryPathway(
    ch1(1H): [1] ⟶ [0] ⟶ [-1]
    total: 1.0 ⟶ 0.0 ⟶ -1.0
)]
```

Dual channels example

Example

```
>>> from mrsimulator.method import Method
>>> method = Method(
...     channels=['1H', '13C'],
...     spectral_dimensions=[
...         {
...             "events": [{
...                 "fraction": 0.5,
...                 "transition_queries": [
...                     {"ch1": {"P": [1]}},
...                     {"ch1": {"P": [-1]}},
...                 ]
...             },
...             {
...                 "fraction": 0.5,
...                 "transition_queries": [  # selecting double quantum
...                     {"ch1": {"P": [-1]}, "ch2": {"P": [-1]}},
...                     {"ch1": {"P": [1]}, "ch2": {"P": [1]}},
...                 ]
...             }],
...         },
...         {
...             "events": [{
...                 "transition_queries": [ # selecting single quantum
...                     {"ch1": {"P": [-1]}},
...                 ]
...             }],
...         }
...     ]
... )
>>> pprint(method.get_symmetry_pathways("P"))
[SymmetryPathway(
    ch1(1H): [1] ⟶ [-1] ⟶ [-1]
    ch2(13C): None ⟶ [-1] ⟶ None
    total: 1.0 ⟶ -2.0 ⟶ -1.0
),
 SymmetryPathway(
```

(continues on next page)

(continued from previous page)

```
    ch1(1H): [1] ⟶ [1] ⟶ [-1]
    ch2(13C): None ⟶ [1] ⟶ None
    total: 1.0 ⟶ 2.0 ⟶ -1.0
),
 SymmetryPathway(
    ch1(1H): [-1] ⟶ [-1] ⟶ [-1]
    ch2(13C): None ⟶ [-1] ⟶ None
    total: -1.0 ⟶ -2.0 ⟶ -1.0
),
 SymmetryPathway(
    ch1(1H): [-1] ⟶ [1] ⟶ [-1]
    ch2(13C): None ⟶ [1] ⟶ None
    total: -1.0 ⟶ 2.0 ⟶ -1.0
)]
```

`get_transition_pathways`(*spin_system*) → List[*TransitionPathway* (page 482)]

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 388)) – A SpinSystem object.

Returns

A list of *TransitionPathway* (page 482) objects. Each TransitionPathway object is an ordered collection of Transition objects.

Example

```
>>> from mrsimulator import SpinSystem
>>> from mrsimulator.method.lib import ThreeQ_VAS
>>> sys = SpinSystem(sites=[{'isotope': '27Al'}, {'isotope': '29Si'}])
>>> method = ThreeQ_VAS(channels=['27Al'])
>>> pprint(method.get_transition_pathways(sys))
[|1.5, -0.5⟩⟨-1.5, -0.5| ⟶ |-0.5, -0.5⟩⟨0.5, -0.5|, weight=(1+0j),
 |1.5, -0.5⟩⟨-1.5, -0.5| ⟶ |-0.5, 0.5⟩⟨0.5, 0.5|, weight=(1+0j),
 |1.5, 0.5⟩⟨-1.5, 0.5| ⟶ |-0.5, -0.5⟩⟨0.5, -0.5|, weight=(1+0j),
 |1.5, 0.5⟩⟨-1.5, 0.5| ⟶ |-0.5, 0.5⟩⟨0.5, 0.5|, weight=(1+0j)]
```

`json`(*units=True*) → dict

Parse the class object to a JSON compliant python dictionary object.

Parameters

`units` – If true, the attribute value is a physical quantity expressed as a string with a number and a unit, else a float.

Returns: dict

`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 403) object.

`plot`(*df=None*, *include_legend=False*) → figure

Creates a diagram representing the method. By default, only parameters which vary throughout the method are plotted. Figure can be finley adjusted using matplotlib rcParams.

Parameters

- `df` (*DataFrame*) – DataFrame to plot data from. By default DataFrame is calculated from summary() and will show only parameters which vary throughout the method plus 'p' symmetry pathway and 'd' symmetry pathway if it is not none or defined
- `include_legend` (*bool*) – Optional argument to include a key for event colors. Default is False and no key will be included in figure

Returns

matplotlib.pyplot.figure

Example

```
>>> from mrsimulator.method.lib import BlochDecaySpectrum
>>> method = BlochDecaySpectrum(channels=["13C"])
>>> fig = method.plot()
```

Adjusting Figure Size rcParams

```
>>> import matplotlib as mpl
>>> from mrsimulator.method.lib import FiveQ_VAS
>>> mpl.rcParams["figure.figsize"] = [14, 10]
>>> mpl.rcParams["font.size"] = 14
>>> method = FiveQ_VAS(channels=["27Al"])
>>> fig = method.plot(include_legend=True)
```

Plotting all Parameters, including Constant

```
>>> from mrsimulator.method.lib import FiveQ_VAS
>>> method = FiveQ_VAS(channels=["27Al"])
>>> df = method.summary(drop_constant_columns=False)
>>> fig = method.plot(df=df)
```

`reduced_dict`(*exclude={}*) → dict

Returns a reduced dictionary representation of the class object by removing all key-value pair corresponding to keys listed in the *exclude* argument, and keys with value as None.

Parameters

`exclude` – A list of keys to exclude from the dictionary.

Return: A dict.

`shape`() → tuple

The shape of the method's spectral dimension array.

Returns

tuple

Example

```
>>> from mrsimulator.method import Method
>>> method = Method(
...     channels=['1H'],
...     spectral_dimensions=[{'count': 40}, {'count': 10}]
... )
>>> method.shape()
(40, 10)
```

summary(*drop_constant_columns=True*) → DataFrame

Returns a DataFrame giving a summary of the Method. A user can specify optional attributes to include which appear as columns in the DataFrame. A user can also ask to leave out attributes which remain constant throughout the method. Invalid attributes for an Event will be replaced with NAN.

Parameters

drop_constant_columns (*bool*) – Removes constant properties if True. Default is True.

Returns

Event number as row and property as column. Invalid properties for an event type are filled with np.nan

Columns

- (str) type: Event type
- (int) spec_dim_index: Index of spectral dimension which event belongs to
- (str) label: Event label
- (float) duration: Duration of the DelayEvent
- (float) fraction: Fraction of the SpectralEvent
- (MixingQuery) query: MixingQuery object of the MixingEvent
- (float) magnetic_flux_density: Magnetic flux density during event in Tesla
- (float) rotor_frequency: Rotor frequency during event in Hz
- (float) rotor_angle: Rotor angle during event converted to Degrees
- (FrequencyEnum) freq_contrib: Frequency

Return type

pd.DataFrame df

Example

All Possible Columns

```
>>> from mrsimulator.method.lib import ThreeQ_VAS
>>> method = ThreeQ_VAS(channels=["17O"])
>>> df = method.summary(drop_constant_columns=False)
>>> pprint(list(df.columns))
['type',
 'spec_dim_index',
 'spec_dim_label',
 'label',
 'duration',
 'fraction',
 'query',
 'magnetic_flux_density',
 'rotor_frequency',
 'rotor_angle',
 'freq_contrib',
```

(continues on next page)

(continued from previous page)

```
'p',
'd']
```

Spinning sideband correlation method

class mrsimulator.method.lib.SSB2D(*, *name: str = 'SSB2D'*, *description: str = 'Simulate a 2D sideband separation method.'*, *label: str = None*, *property_units: Dict = {'magnetic_flux_density': 'T', 'rotor_angle': 'rad', 'rotor_frequency': 'Hz'}*, *channels: List[Union[str, dict, Isotope (page 479)]]*, *spectral_dimensions: List[SpectralDimension (page 410)] = [SpectralDimension(name=None, description=None, label=None, property_units={'spectral_width': 'Hz', 'reference_offset': 'Hz', 'origin_offset': 'Hz'}, count=1024, spectral_width=25000.0, reference_offset=0.0, origin_offset=None, reciprocal=None, events=[])]*, *affine_matrix: List = [1, -1, 0, 1]*, *simulation: Union[CSDM, ndarray] = None*, *experiment: Union[CSDM, ndarray] = None*, *magnetic_flux_density: ConstrainedFloatValue = 9.4*, *rotor_frequency: ConstrainedFloatValue = 1000000000000.0*, *rotor_angle: ConstrainedFloatValue = 0.9553166181245*)

Bases: BaseNamedMethod2D

Simulating a sheared 2D finite to infinite speed MAS correlation spectrum.

Returns

A *Method* (page 403) 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)
[|-0.5⟩⟨0.5| ⟶ |-0.5⟩⟨0.5|, weight=(1+0j)]
```

classmethod check_event_objects_for_compatibility(*default_dim*, *obj_dim*, *method_dict*)

Checks Events for compatibility and sets global method attributes

Parameters

- default_dim (*dict*) – Dict representation of SpectralDimension in base method
- obj_dim (SpectralDimension (page 410)) – User-passed SpectralDimension object to check
- method_dict (*dict*) – Dict representation of passed method

classmethod check_method_compatibility(*py_dict*)

Check for events attribute inside the spectral_dimensions. Events are not allowed for NamedMethods.

dict(***kwargs*)

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

get_symmetry_pathways(*symmetry_element: str*) → List[*SymmetryPathway* (page 483)]

Return a list of symmetry pathways of the method.

Parameters

symmetry_element (*str*) – The symmetry element, 'P' or 'D'.

Returns

A list of *SymmetryPathway* (page 483) objects.

Single channel example

Example

```
>>> from mrsimulator.method import Method
>>> method = Method(
...     channels=['1H'],
...     spectral_dimensions=[
...         {
...             "events": [
...                 {
...                     "fraction": 0.5,
...                     "transition_queries": [{"ch1": {"P": [1]}}]
...                 },
...                 {
...                     "fraction": 0.5,
...                     "transition_queries": [{"ch1": {"P": [0]}}]
...                 }
...             ],
...         },
...         {
...             "events": [
...                 {"transition_queries": [{"ch1": {"P": [-1]}}]},
...             ],
...         }
...     ]
... )
>>> pprint(method.get_symmetry_pathways("P"))
[SymmetryPathway(
    ch1(1H): [1] ⟶ [0] ⟶ [-1]
    total: 1.0 ⟶ 0.0 ⟶ -1.0
)]
```

Dual channels example

Example

```
>>> from mrsimulator.method import Method
>>> method = Method(
...     channels=['1H', '13C'],
...     spectral_dimensions=[
...         {
...             "events": [{
...                 "fraction": 0.5,
...                 "transition_queries": [
...                     {"ch1": {"P": [1]}},
...                     {"ch1": {"P": [-1]}},
...                 ]
...             },
...             {
...                 "fraction": 0.5,
...                 "transition_queries": [  # selecting double quantum
...                     {"ch1": {"P": [-1]}, "ch2": {"P": [-1]}},
...                     {"ch1": {"P": [1]}, "ch2": {"P": [1]}},
...                 ]
...             }],
...         },
...         {
...             "events": [{
...                 "transition_queries": [ # selecting single quantum
...                     {"ch1": {"P": [-1]}},
...                 ]
...             }],
...         }
...     ]
... )
>>> pprint(method.get_symmetry_pathways("P"))
[SymmetryPathway(
    ch1(1H): [1] ⟶ [-1] ⟶ [-1]
    ch2(13C): None ⟶ [-1] ⟶ None
    total: 1.0 ⟶ -2.0 ⟶ -1.0
),
 SymmetryPathway(
    ch1(1H): [1] ⟶ [1] ⟶ [-1]
    ch2(13C): None ⟶ [1] ⟶ None
    total: 1.0 ⟶ 2.0 ⟶ -1.0
),
 SymmetryPathway(
    ch1(1H): [-1] ⟶ [-1] ⟶ [-1]
    ch2(13C): None ⟶ [-1] ⟶ None
    total: -1.0 ⟶ -2.0 ⟶ -1.0
),
 SymmetryPathway(
    ch1(1H): [-1] ⟶ [1] ⟶ [-1]
    ch2(13C): None ⟶ [1] ⟶ None
```

(continues on next page)

(continued from previous page)

```
    total: -1.0 ⟶ 2.0 ⟶ -1.0
)]
```

get_transition_pathways(*spin_system*) → List[*TransitionPathway* (page 482)]

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 388)) – A SpinSystem object.

Returns

A list of *TransitionPathway* (page 482) objects. Each TransitionPathway object is an ordered collection of Transition objects.

Example

```
>>> from mrsimulator import SpinSystem
>>> from mrsimulator.method.lib import ThreeQ_VAS
>>> sys = SpinSystem(sites=[{'isotope': '27Al'}, {'isotope': '29Si'}])
>>> method = ThreeQ_VAS(channels=['27Al'])
>>> pprint(method.get_transition_pathways(sys))
[|1.5, -0.5⟩⟨-1.5, -0.5| ⟶ |-0.5, -0.5⟩⟨0.5, -0.5|, weight=(1+0j),
 |1.5, -0.5⟩⟨-1.5, -0.5| ⟶ |-0.5, 0.5⟩⟨0.5, 0.5|, weight=(1+0j),
 |1.5, 0.5⟩⟨-1.5, 0.5| ⟶ |-0.5, -0.5⟩⟨0.5, -0.5|, weight=(1+0j),
 |1.5, 0.5⟩⟨-1.5, 0.5| ⟶ |-0.5, 0.5⟩⟨0.5, 0.5|, weight=(1+0j)]
```

json(*units=True*) → dict

Parse the class object to a JSON compliant python dictionary object.

Parameters

units – If true, the attribute value is a physical quantity expressed as a string with a number and a unit, else a float.

Returns: dict

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 403) object.

plot(*df=None*, *include_legend=False*) → figure

Creates a diagram representing the method. By default, only parameters which vary throughout the method are plotted. Figure can be finley adjusted using matplotlib rcParams.

Parameters

- df (*DataFrame*) – DataFrame to plot data from. By default DataFrame is calculated from summary() and will show only parameters which vary throughout the method plus 'p' symmetry pathway and 'd' symmetry pathway if it is not none or defined
- include_legend (*bool*) – Optional argument to include a key for event colors. Default is False and no key will be included in figure

Returns
matplotlib.pyplot.figure

Example

```
>>> from mrsimulator.method.lib import BlochDecaySpectrum
>>> method = BlochDecaySpectrum(channels=["13C"])
>>> fig = method.plot()
```

Adjusting Figure Size rcParams

```
>>> import matplotlib as mpl
>>> from mrsimulator.method.lib import FiveQ_VAS
>>> mpl.rcParams["figure.figsize"] = [14, 10]
>>> mpl.rcParams["font.size"] = 14
>>> method = FiveQ_VAS(channels=["27Al"])
>>> fig = method.plot(include_legend=True)
```

Plotting all Parameters, including Constant

```
>>> from mrsimulator.method.lib import FiveQ_VAS
>>> method = FiveQ_VAS(channels=["27Al"])
>>> df = method.summary(drop_constant_columns=False)
>>> fig = method.plot(df=df)
```

`reduced_dict`(*exclude={}*) → dict

Returns a reduced dictionary representation of the class object by removing all key-value pair corresponding to keys listed in the *exclude* argument, and keys with value as None.

Parameters
`exclude` – A list of keys to exclude from the dictionary.

Return: A dict.

`shape`() → tuple

The shape of the method's spectral dimension array.

Returns
tuple

Example

```
>>> from mrsimulator.method import Method
>>> method = Method(
...     channels=['1H'],
...     spectral_dimensions=[{'count': 40}, {'count': 10}]
... )
>>> method.shape()
(40, 10)
```

`summary`(*drop_constant_columns=True*) → DataFrame

Returns a DataFrame giving a summary of the Method. A user can specify optional attributes to include which appear as columns in the DataFrame. A user can also ask to leave out attributes which remain constant throughout the method. Invalid attributes for an Event will be replaced with NAN.

Parameters
drop_constant_columns (*(bool)*) – Removes constant properties if True. Default is True.

Returns
Event number as row and property as column. Invalid properties for an event type are filled with np.nan

Columns
- (str) type: Event type
- (int) spec_dim_index: Index of spectral dimension which event belongs to
- (str) label: Event label
- (float) duration: Duration of the DelayEvent
- (float) fraction: Fraction of the SpectralEvent
- (MixingQuery) query: MixingQuery object of the MixingEvent
- (float) magnetic_flux_density: Magnetic flux density during event in Tesla
- (float) rotor_frequency: Rotor frequency during event in Hz
- (float) rotor_angle: Rotor angle during event converted to Degrees
- (FrequencyEnum) freq_contrib: Frequency

Return type
pd.DataFrame df

Example

All Possible Columns

```
>>> from mrsimulator.method.lib import ThreeQ_VAS
>>> method = ThreeQ_VAS(channels=["17O"])
>>> df = method.summary(drop_constant_columns=False)
>>> pprint(list(df.columns))
['type',
 'spec_dim_index',
 'spec_dim_label',
 'label',
 'duration',
 'fraction',
 'query',
 'magnetic_flux_density',
 'rotor_frequency',
 'rotor_angle',
 'freq_contrib',
 'p',
 'd']
```

17.8 Other Objects

17.8.1 Sites

class mrsimulator.simulator.Sites(*data=[]*)

Bases: AbstractList

A list of unique *Site* (page 394) objects within a simulator object.

`append(item)`
Append a list item

`clear()` → None -- remove all items from S

`count(value)` → integer -- return number of occurrences of value

`extend(values)`
S.extend(iterable) – extend sequence by appending elements from the iterable

`index(value[, start[, stop]])` → integer -- return first index of value.
Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

`insert(index, item)`
Insert a list item

`pop([index])` → item -- remove and return item at index (default last).
Raise IndexError if list is empty or index is out of range.

`remove(value)`
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

`reverse()`
S.reverse() – reverse *IN PLACE*

`to_pd()`
Return sites as a pandas dataframe.

17.8.2 ZeemanState

`class mrsimulator.spin_system.zeemanstate.ZeemanState(n_sites, *args)`
Bases: `object`

Zeeman energy state class.

17.8.3 SymmetricTensor

`class mrsimulator.spin_system.tensors.SymmetricTensor(*, name: str = None, description: str = None, label: str = None, property_units: Dict = {'Cq': 'Hz', 'D': 'Hz', 'alpha': 'rad', 'beta': 'rad', 'gamma': 'rad', 'zeta': 'ppm'}, zeta: float = None, Cq: float = None, D: float = None, eta: ConstrainedFloatValue = None, alpha: float = None, beta: float = None, gamma: float = None)`
Bases: `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)
```

`json(`*exclude={}*, *units=True*`)` → dict
Parse the class object to a JSON compliant python dictionary object.

Parameters
- `exclude` – Set of keys that will be excluded from the result.
- `units` – If true, the attribute value is a physical quantity expressed as a string with a number and a unit, else a float.

Returns: dict

`classmethod parse_dict_with_units(`*json_ dict: dict*`)`
Parse the physical quantity from a dictionary representation of the class object, where the physical quantity is expressed as a string with a number and a unit.

Parameters
`json_dict` (*dict*) – A required python dict object.

`reduced_dict(`*exclude={}*`)` → dict
Returns a reduced dictionary representation of the class object by removing all key-value pair corresponding to keys listed in the *exclude* argument, and keys with value as None.

Parameters
`exclude` – A list of keys to exclude from the dictionary.

Return: A dict.

`rotate(`*euler_ angles: list*`)` → None
Rotate the tensor by the given list of Euler angle rotations. Euler angles are given as a list of (alpha, beta, gamma) tuples, and rotations happen in the Haeberlen (ZYZ) convention.

Parameters
`euler_angles` (*(list)*) – An ordered list of angle tuples (alpha, beta, gamma) to rotate through.

Example

```
>>> tensor = SymmetricTensor(zeta=10, eta=0.3, alpha=1, beta=1, gamma=1)
>>> angles = [(3.1415, 0, -3.1415), (1.5701, 1.5701, 1.5701)]
>>> tensor.rotate(angles)
```

17.8.4 AntisymmetricTensor

`class mrsimulator.spin_system.tensors.AntisymmetricTensor(*, name: str = None, description: str = None, label: str = None, property_units: Dict = {'alpha': 'rad', 'beta': 'rad', 'zeta': 'ppm'}, zeta: Optional[float] = None, alpha: Optional[float] = None, beta: Optional[float] = None)`

Bases: `Parseable`

Base AntiSymmetricTensor 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.

Type
Optional[float]

`alpha`
Euler angle, alpha, given in radian. The default value is None.

Type
Optional[float]

`beta`
Euler angle, beta, given in radian. The default value is None.

Type
Optional[float]

`json(exclude={}, units=True)` → dict
Parse the class object to a JSON compliant python dictionary object.

Parameters
- `exclude` – Set of keys that will be excluded from the result.
- `units` – If true, the attribute value is a physical quantity expressed as a string with a number and a unit, else a float.

Returns: dict

`classmethod parse_dict_with_units(json_dict: dict)`
Parse the physical quantity from a dictionary representation of the class object, where the physical quantity is expressed as a string with a number and a unit.

Parameters
json_dict (*dict*) – A required python dict object.

reduced_dict(*exclude={}*) → dict

Returns a reduced dictionary representation of the class object by removing all key-value pair corresponding to keys listed in the *exclude* argument, and keys with value as None.

Parameters
exclude – A list of keys to exclude from the dictionary.

Return: A dict.

17.8.5 Isotope

class mrsimulator.spin_system.isotope.Isotope(*, *symbol: str*)

Bases: 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
10.708398861439887
>>> carbon.atomic_number
6
>>> carbon.quadrupole_moment  # in eB
0.0
```

property atomic_number

Atomic number of the isotope.

property gyromagnetic_ratio

Reduced gyromagnetic ratio of the nucleus given in units of MHz/T.

json(***kwargs*) → dict

Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

encoder is an optional function to supply as *default* to json.dumps(), other arguments as per *json.dumps()*.

larmor_freq(*B0=9.4*)

Return the Larmor frequency of the isotope at a magnetic field strength B0.

Parameters
B0 (*float*) – magnetic field strength in T

Returns
larmor frequency in MHz
Return type
float

Example

```
>>> silicon = Isotope(symbol="29Si")
>>> freq = silicon.larmor_freq(B0 = 9.4)
```

property natural_abundance
Natural abundance of the isotope in units of %.

classmethod parse(*item*)
Attempt to parse the provided item into an Isotope instance

property quadrupole_moment
Quadrupole moment of the nucleus given in units of eB (electron-barn).

classmethod register(*symbol: str*, *copy_from: Optional[str] = None*, ***kwargs*)
Register a new isotope symbol with intrinsic attributes given by kwargs. If *copy_from* is not None and matches a known isotope symbol, then the attributes are copied from that symbol.

Parameters
- symbol (*(str)*) – The isotope symbol to register. Must be unique.
- copy_from (*(str)*) – An optional isotope symbol to copy attributes from. If None, then no attributes are copied.

property spin
Spin quantum number, I, of the isotope.

property spin_multiplicity
Spin multiplicity, (2I+1), of the isotope.

17.8.6 Transition

class mrsimulator.transition.Transition(***, *name: str = None*, *description: str = None*, *label: str = None*, *property_units: Dict = {}*, *initial: List[float] = []*, *final: List[float] = []*)

Bases: Parseable

Base Transition class describes a spin transition between two energy states, where the energy states are described using the weakly coupled basis.

$$|m_{i,0}, m_{i,1}, ...m_{i,N}\rangle \rightarrow |m_{f,0}, m_{f,1}, ...m_{f,N}\rangle \tag{17.9}$$

Parameters
- initial (*list*) – The initial Zeeman energy state represented as a list of quantum numbers $m_{i,n}$.
- final (*list*) – The final Zeeman energy state represented as a list of quantum numbers $m_{f,n}$.

Example

```
>>> from mrsimulator.transition import Transition
>>> t1 = Transition(initial = [0.5, 0.5], final = [0.5, -0.5])
>>> t1
|0.5, -0.5⟩⟨0.5, 0.5|
```

property D

Return a list of Δm**2 values of the spin transition for each site in a weakly coupled basis.

Example

```
>>> t1.D
array([0., 0.])
```

property P

Return a list of Δm values of the spin transition for each site in a weakly coupled basis.

Example

```
>>> t1.P
array([ 0., -1.])
```

property delta_m

An alias for p

json(*exclude={}*, *units=True*) → dict

Parse the class object to a JSON compliant python dictionary object.

Parameters

- **exclude** – Set of keys that will be excluded from the result.
- **units** – If true, the attribute value is a physical quantity expressed as a string with a number and a unit, else a float.

Returns: dict

property p

Return the total Δm (m_final-m_initial) value of the spin transition.

Example

```
>>> t1.p
-1.0
```

classmethod parse_dict_with_units(*json_dict: dict*)

Parse the physical quantity from a dictionary representation of the class object, where the physical quantity is expressed as a string with a number and a unit.

Parameters

json_dict (*dict*) – A required python dict object.

`reduced_dict(`*exclude={}*`)` → dict

Returns a reduced dictionary representation of the class object by removing all key-value pair corresponding to keys listed in the *exclude* argument, and keys with value as None.

Parameters

`exclude` – A list of keys to exclude from the dictionary.

Return: A dict.

`tolist()` → list

Convert the transition to a list of quantum numbers where the first N quantum numbers corresponds to the initial energy state, while the last N corresponds to the final energy state, where N is the number of sites.

Example

```
>>> t1.tolist()
[0.5, 0.5, 0.5, -0.5]
```

17.8.7 TransitionPathway

`class mrsimulator.transition.TransitionPathway(`*pathway: list = [], weight=1 + 0j*`)`

Bases: `TransitionList`

Base TransitionPathway class is a list of connected Transitions.

Example

```
>>> from mrsimulator.transition import TransitionPathway, Transition
>>> t1 = Transition(initial = [0.5, 0.5], final = [0.5, -0.5])
>>> t2 = Transition(initial=[0.5, 0.5], final=[-0.5, 0.5])
>>> path = TransitionPathway([t1, t2])
>>> path
|0.5, -0.5⟩⟨0.5, 0.5| ⟶ |-0.5, 0.5⟩⟨0.5, 0.5|, weight=(1+0j)
```

`append(`*item*`)`

Append a list item

`clear()` → None -- remove all items from S

`count(`*value*`)` → integer -- return number of occurrences of value

`extend(`*values*`)`

S.extend(iterable) – extend sequence by appending elements from the iterable

`filter(`*P=None, PP=None, D=None*`)`

Filter a list of transitions to satisfy the filtering criterion.

Parameters

- `P` (*list*) – A list of N (m_final - m_initial) values, where N is the total number of sites within the spin system.
- `D` (*list*) – A list of N (m_final^2 - m_initial^2) values, where N is the total number of sites within the spin system.

`index(`*value*[, *start*[, *stop*]]`)` → integer -- return first index of value.
Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

`insert(`*index*, *item*`)`
Insert a list item

`json(`***kwargs*`)` → dict
Parse the class object to a JSON compliant python dictionary object.

Example

```
>>> pprint(path.json())
{'pathway': [{'final': [0.5, -0.5], 'initial': [0.5, 0.5]},
             {'final': [-0.5, 0.5], 'initial': [0.5, 0.5]}],
 'weight': (1+0j)}
```

`pop(`[*index*]`)` → item -- remove and return item at index (default last).
Raise IndexError if list is empty or index is out of range.

`remove(`*value*`)`
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

`reverse()`
S.reverse() – reverse *IN PLACE*

`tolist()`
Expand TransitionPathway to a Python list.

Example

```
>>> path.tolist()
[0.5, 0.5, 0.5, -0.5, 0.5, 0.5, -0.5, 0.5]
```

17.8.8 SymmetryPathway

`class mrsimulator.transition.SymmetryPathway(`*, *channels: list = []*, *ch1: list = None*, *ch2: list = None*, *ch3: list = None*`)`

Bases: `BaseModel`

Base SymmetryPathway class.

`channels`
The list of channels

Type
list

`ch1`
The symmetry pathway for channel at index 0.

Type
list

`ch2`

The symmetry pathway for channel at index 1.

Type
list

`ch3`

The symmetry pathway for channel at index 2.

Type
list

`total`

The total symmetry pathway.

17.9 Utility functions

`mrsimulator.utils.collection.single_site_system_generator`(*isotope: Union[str, List[str]]*, *isotropic_chemical_shift: Union[float, List[float], ndarray] = 0*, *shielding_symmetric: Optional[Dict] = None*, *shielding_antisymmetric: Optional[Dict] = None*, *quadrupolar: Optional[Dict] = None*, *abundance: Optional[Union[float, List[float], ndarray]] = None*, *site_name: Optional[Union[str, List[str]]] = None*, *site_label: Optional[Union[str, List[str]]] = None*, *site_description: Optional[Union[str, List[str]]] = None*, *rtol: float = 0.001*) → List[*SpinSystem* (page 388)]

Generate and return a list of single-site spin systems from the input parameters

Parameters

- `isotope` – A required string or a list of site isotopes.
- `isotropic_chemical_shift` – A float or a list/ndarray of isotropic chemical shifts per site per spin system. The default is 0.
- `shielding_symmetric` – A shielding symmetric dict object, where the keyword value can either be a float or a list/ndarray of floats. The default value is None. The allowed keywords are `zeta`, `eta`, `alpha`, `beta`, and `gamma`.
- `shielding_antisymmetric` – A shielding antisymmetric dict object, where the keyword value can either be a float or a list/ndarray of floats. The default value is None. The allowed keywords are `zeta`, `alpha`, and `beta`.
- `quadrupolar` – A quadrupolar dict object, where the keyword value can either be a float or a list/ndarray of floats. The default value is None. The allowed keywords are `Cq`, `eta`, `alpha`, `beta`, and `gamma`.
- `abundance` – A float or a list/ndarray of floats describing the abundance of each spin system.
- `site_name` – A string or a list of strings with site names per site per spin system. The default is None.
- `site_label` – A string or a list of strings with site labels per site per spin system. The default is None.
- `site_description` – A string or a list of strings with site descriptions per site per spin system. The default is None.

- `rtol` – The relative tolerance used in determining the cutoff abundance, given as, $\texttt{abundance}_{\texttt{cutoff}} = \texttt{rtol} * \texttt{max}(\texttt{abundance})$. The spin systems with abundance below this threshold are ignored.

Returns

List of *SpinSystem* (page 388) objects with a single *Site* (page 394)

Example

Single spin system:

```
>>> sys1 = single_site_system_generator(
...     isotope=["1H"],
...     isotropic_chemical_shift=10,
...     site_name="Single Proton",
... )
>>> print(len(sys1))
1
```

Multiple spin system:

```
>>> sys2 = single_site_system_generator(
...     isotope="1H",
...     isotropic_chemical_shift=[10] * 5,
...     site_name="5 Protons",
... )
>>> print(len(sys2))
5
```

Multiple spin system with dictionary arguments:

```
>>> Cq = [4.2e6] * 12
>>> sys3 = single_site_system_generator(
...     isotope="17O",
...     isotropic_chemical_shift=60.0,  # in ppm,
...     quadrupolar={"Cq": Cq, "eta": 0.5},  # Cq in Hz
... )
>>> print(len(sys3))
12
```

Note: The parameter value can either be a float or a list/ndarray. If the parameter value is a float, the given value is assigned to the respective parameter in all the spin systems. If the parameter value is a list or ndarray, its *ith* value is assigned to the respective parameter of the *ith* spin system. When multiple parameter values are given as lists/ndarrays, the length of all the lists must be the same.

---

mrsimulator.utils.collection.site_generator(*isotope: Union[str, List[str]], isotropic_chemical_shift: Union[float, List[float], ndarray] = 0, shielding_symmetric: Optional[Dict] = None, shielding_antisymmetric: Optional[Dict] = None, quadrupolar: Optional[Dict] = None, name: Optional[Union[str, List[str]]] = None, label: Optional[Union[str, List[str]]] = None, description: Optional[Union[str, List[str]]] = None*) → List[*Site* (page 394)]

Generate a list of Site objects from lists of site attributes.

Parameters

- `isotope` – A required string or a list of site isotopes.
- `isotropic_chemical_shift` – A float or a list/ndarray of isotropic chemical shifts per site. The default is 0.
- `shielding_symmetric` – A shielding symmetric dict object, where the keyword value can either be a float or a list/ndarray of floats. The default value is None. The allowed keywords are `zeta`, `eta`, `alpha`, `beta`, and `gamma`.
- `shielding_antisymmetric` – A shielding antisymmetric dict object, where the keyword value can either be a float or a list/ndarray of floats. The default value is None. The allowed keywords are `zeta`, `alpha`, and `beta`.
- `quadrupolar` – A quadrupolar dict object, where the keyword value can either be a float or a list/ndarray of floats. The default value is None. The allowed keywords are `Cq`, `eta`, `alpha`, `beta`, and `gamma`.
- `name` – A string or a list of strings with site names per site. The default is None.
- `label` – A string or a list of strings with site labels per site. The default is None.
- `description` – A string or a list of strings with site descriptions per site. The default is None.

Returns

List of *Site* (page 394) objects

Return type

sites

Example

Generating 10 hydrogen sites:

```
>>> sites1 = site_generator(
...     isotope=["1H"] * 10,
...     isotropic_chemical_shift=-15,
...     name="10 Protons",
... )
>>> print(len(sites1))
10
```

Generating 10 hydrogen sites with different shifts:

```
>>> shifts = np.arange(-10, 10, 2)
>>> sites2 = site_generator(
...     isotope=["1H"] * 10,
...     isotropic_chemical_shift=shifts,
...     name="10 Proton",
... )
>>> print(len(sites2))
10
```

Generating multiple sites with dictionary arguments:

```
>>> Cq = [4.2e6] * 12
>>> sys3 = site_generator(
...     isotope="17O",
...     isotropic_chemical_shift=60.0,  # in ppm,
...     quadrupolar={"Cq": Cq, "eta": 0.5},  # Cq in Hz
... )
```

(continues on next page)

(continued from previous page)

```
>>> print(len(sys3))
12
```

mrsimulator.utils.get_spectral_dimensions(*csdm_object*, *units=False*)

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 contains the count, spectral_width, and reference_offset.

17.10 Mrsimulator IO

mrsimulator.save(*filename: str*, *simulator: Simulator (page 379)*, *signal_processors: Optional[List] = None*, *application: Optional[Dict] = None*, *with_units: bool = True*)

Serialize the Simulator, list of SignalProcessor, and an application dict to a file. Creates a Mrsimulator object and calls save.

Parameters

- `filename` (*str*) – The data is serialized to this file.
- `sim` – Simulator object.
- `signal_processors` – A list of PostSimulator objects corresponding to the methods in the Simulator object. Default is None.
- `application` – Dictionary holding metadata to serialize in the file. The dictionary will be held in the application key.
- `with_units` (*bool*) – If true, physical quantities are represented as string with units. The default is True.

mrsimulator.dict(*simulator: Simulator (page 379)*, *signal_processors: Optional[List] = None*, *application: Optional[Dict] = None*, *with_units: bool = True*)

Export the Simulator, list of SignalProcessor, and an application dict to a python dictionary. Creates a Mrsimulator object with given arguments and calls json from the Mrsimulator object.

Parameters

- `sim` – Simulator object.
- `signal_processors` – A list of PostSimulator objects corresponding to the methods in the Simulator object. Default is None.
- `application` – Dictionary holding metadata to serialize in the dict. The dictionary will be held under the application key.
- `with_units` (*bool*) – If true, physical quantities are represented as string with units. The default is True.

Returns

Python dictionary

mrsimulator.load(*filename: str*, *parse_units: bool = True*)

Load Simulator object, list of SignalProcessor objects and metadata from a JSON serialized file of a `Mrsimulator` object.

Parameters

- `filename` (*str*) – The location to the .mrsim file.
- `parse_units` (*bool*) – If true, parse the dictionary for units. The default is True.

Returns
Simulator, List[SignalProcessor], Dict.
Return type
Ordered List

`mrsimulator.parse(`*py_dict*, *parse_units: bool = True*`)`

Parse a dictionary object to the respective Simulator object, list of SignalProcessor objects, and the metadata dictionary. If no signal processors are provided a list of default SignalProcessor objects with length equal to number of methods will be returned.

Parameters
- `py_dict` (*Dict*) – Python dictionary representation of a `Mrsimulator` object.
- `parse_units` (*bool*) – If true, parse the dictionary for units. Default is True.

Returns
Simulator, List[SignalProcessor], Dict.
Return type
Ordered List

CHAPTER

EIGHTEEN

SIGNAL-PROCESSOR API

18.1 Signal Processor

class mrsimulator.signal_processor.SignalProcessor(*, *name: str = None*, *description: str = None*, *label: str = None*, *property_units: Dict = {}*, *processed_dataset: CSDM = None*, *operations: List[Operation] = []*)

Bases: Parseable

SignalProcessor class to apply a series of operations to the dependent variables of the simulation dataset.

Parameters

operations (*List[mrsimulator.signal_processor._base.Operation]*) – A list of operations.

Examples

```
>>> post_sim = SignalProcessor(operations=[o1, o2])
```

apply_operations(*dataset*, ***kwargs*)

Function to apply all the operation functions in the operations member of a SignalProcessor object. Operations applied sequentially over the dataset member.

Returns

A copy of the dataset member with the operations applied to it.

Return type

CSDM object

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.

Returns

A Dict object.

classmethod parse_dict_with_units(*py_dict: dict*)

Parse a list of operations dictionary to a SignalProcessor class object.

Parameters

pt_dict – A python dict object.

reduced_dict(*exclude={}*) → dict

Returns a reduced dictionary representation of the class object by removing all key-value pair corresponding to keys listed in the *exclude* argument, and keys with value as None.

Parameters
`exclude` – A list of keys to exclude from the dictionary.

Return: A dict.

18.2 Operations

18.2.1 Generic operations

Import the module as

```
from mrsimulator import signal_processor 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 (y) from a CSDM object.
`Linear`	Apply linear operation across all dependent variables (y) 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.

18.2.2 Baseline

Access the sub-module as `sp.baseline`

Operation Summary

The following list of operations applies to **selected dependent variables** within the CSDM object.

`Polynomial`	Add a baseline polynomial to all dependent variables (y) in the CSDM object.
`ConstantOffset`	Add an offset to the dependent variables (y) of the CSDM object.

See also:

Signal Processor (page 133) for more details.

Signal Processing Gallery (page 353) for notebooks using these operations.

18.2.3 Apodization

Access the sub-module as `sp.apodization`

Operation Summary

The following list of operations applies to **selected dependent variables** within the CSDM object.

`Gaussian`	Apodize dependent variables of CSDM dataset with Gaussian function.
`Exponential`	Apodize dependent variables of CSDM by exponential function.
`SkewedGaussian`	Apodize dependent variables of CSDM dataset with skewed Gaussian function.
`TopHat`	Apodize dependent variables of CSDM object by top hat function.
`Mask`	Apodize dependent variables of CSDM object by user defined mask.

See also:

Signal Processor (page 133) for more details.

Signal Processing Gallery (page 353) for notebooks using these operations.

18.2.4 Affine Transformation

Access the sub-module as `sp.affine`

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 Processor (page 133) for more details.

CHAPTER

NINETEEN

MODELS API

19.1 Czjzek distribution Model

class mrsimulator.models.CzjzekDistribution(*sigma: float*, *mean_isotropic_chemical_shift: float = 0.0*, *abundance: float = 100.0*, *polar=False*, *cache=True*)

Bases: AbstractDistribution

A Czjzek distribution model class.

The Czjzek distribution model is a random sampling of second-rank traceless symmetric tensors whose explicit matrix form follows

$$\mathbf{S} = \begin{bmatrix} \sqrt{3}U_5 - U_1 & \sqrt{3}U_4 & \sqrt{3}U_2 \\ \sqrt{3}U_4 & -\sqrt{3}U_5 - U_1 & \sqrt{3}U_3 \\ \sqrt{3}U_2 & \sqrt{3}U_3 & 2U_1 \end{bmatrix}, \tag{19.1}$$

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 $< U_i U_i >= \sigma^2$.

$$S_T = S_C(\sigma), \tag{19.2}$$

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)
```

add_lmfit_params(*params*, *i*)

Create lmfit params for index i

`pack_csdm_object(`*pos*, *data*`)`

Pack data and coordinates as csdm objects

`pdf(`*pos*, *size: int = 400000*, *analytical: bool = True*, *pack_as_csdm: bool = False*`)`

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.
- `pack_as_csdm` – If true, returns as csdm object.

Returns

A list of x and y coordinates and the corresponding amplitudes if not packed as csdm object, else a csdm object.

Example

```
>>> import numpy as np
>>> cq = np.arange(50) - 25
>>> eta = np.arange(21)/20
>>> amp = cz_model.pdf(pos=[cq, eta])  # returns amp as a CSDM object.
```

`pdf_numerical(`*pos*, *size: int = 400000*`)`

Generate distribution numerically

`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)
```

`update_lmfit_params(`*params*, *i*`)`

Update lmfit params for index i

19.1.1 Mini-gallery using czjzek distributions

- *Czjzek distribution (Shielding and Quadrupolar)* (page 190)
- *Czjzek distribution, ^{27}Al (I=5/2) 3QMAS* (page 251)
- *Fitting a Czjzek Model* (page 315)

19.2 Extended Czjzek distribution Model

class mrsimulator.models.ExtCzjzekDistribution(*symmetric_tensor: SymmetricTensor (page 475), eps: float, mean_isotropic_chemical_shift: float = 0.0, abundance: float = 100.0, polar=False, cache=True*)

Bases: `AbstractDistribution`

An extended Czjzek distribution distribution model.

The extended Czjzek random distribution[1] model is an extension of the Czjzek model, given as

$$S_T = S(0) + \rho S_C(\sigma = 1), \tag{19.3}$$

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

$$\rho = \frac{||S(0)||\epsilon}{\sqrt{30}}, \tag{19.4}$$

where $\|S(0)\|$ is the 2-norm of the dominant tensor, and ϵ is a fraction.

Parameters

- `symmetric_tensor` (SymmetricTensor (page 475)) – A shielding or quadrupolar symmetric tensor or equivalent dict object.
- `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)
```

add_lmfit_params(*params, i*)

Create lmfit params for index i

pack_csdm_object(*pos, data*)

Pack data and coordinates as csdm objects

pdf(*pos, size: int = 400000, analytical: bool = True, pack_as_csdm: bool = False*)

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.
- `pack_as_csdm` – If true, returns as csdm object.

Returns

A list of x and y coordinates and the corresponding amplitudes if not packed as csdm object, else a csdm 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

Example

```
>>> import numpy as np
>>> cq = np.arange(50) - 25
>>> eta = np.arange(21)/20
>>> amp = cz_model.pdf(pos=[cq, eta])  # returns amp as a CSDM object.
```

`pdf_numerical`(*pos*, *size: int = 400000*)

Generate distribution numerically

`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)
```

`update_lmfit_params`(*params*, *i*)

Create lmfit params for index i

19.2.1 Mini-gallery using extended czjzek distributions

- *Extended Czjzek distribution (Shielding and Quadrupolar)* (page 194)
- *Simulating site disorder (crystalline)* (page 248)
- *Extended Czjzek fitting of ^{139}La MAS NMR of $La_{0.2}Y_{1.8}Si_2O_7$* (page 307)

CHAPTER

TWENTY

FITTING UTILITY API

20.1 LMFIT supplement functions

mrsimulator.utils.spectral_fitting.make_LMFIT_params(*sim: Simulator (page 379) = Simulator(name=None, description=None, label=None, property_units={}, spin_systems=[], spin_system_models=[], methods=[], config=ConfigSimulator(name=None, description=None, label=None, property_units={}, number_of_sidebands=64, number_of_gamma_angles=1, integration_volume='octant', integration_density=70, decompose_spectrum='none', isotropic_interpolation='linear', custom_sampling=None))*, *processors: Optional[list] = None*, *spin_system_models: list = []*, *include={}*)

Parse the Simulator and PostSimulator objects for a list of LMFIT parameters.

Parameters

- `sim` (Simulator (page 379)) – Simulator object.
- `processors` (*list*) – List of SignalProcessor objects. The order must match the order of methods within the simulator object.
- `include` (*set*) – set of keywords from the method object to include as a fitting parameter. Default is {}.

The parameter name associated with the spin system within Simulator object is generated using the following nomenclature- *sys_i_site_j_attribute1_attribute2* for attribute with signature `sim.spin_systems[i].sites[j].attribute1.attribute2`

Here, *sys_i* refers to the spin system at index *i*, *site_j* refers to the site at index *j* with in the :math:i^text{th} spin system, and *attribute1* and *attribute2* are the site attributes.

For examples:

`sim.spin_systems[1].sites[0].isotropic_chemical_shift` parametrizes to *sys_1_site_0_isotropic_chemical_shift* while `sim.spin_systems[0].sites[1].quadrupolar.Cq` to *sys_0_site_1_quadrupolar_Cq*.

Returns

LMFIT Parameters object.

mrsimulator.utils.spectral_fitting.LMFIT_min_function(*params: Parameters*, *sim: Simulator (page 379)*, *processors: Optional[list] = None*, *sigma: Optional[list] = None*, ***sim_kwargs*)

The simulation routine to calculate the vector difference between simulation and experiment based on the parameters update.

Parameters

- `params` – Parameters object containing parameters for OLS minimization.
- `sim` – Simulator object.
- `processors` – A list of :py:class:~`mrsimulator.signal_processor.Processor` objects to apply post-simulation processing to the simulated spectra.
- `sigma` – A list of standard deviations corresponding to the experiments in the :py:attr:~`mrsimulator.Simulator.methods` attribute.
- `sim_kwargs` – Keyword arguments to pass to the :py:mth:~`mrsimulator.Simulator.run()` method.

Returns

Array of the differences between the simulation and the experimental datasets.

`mrsimulator.utils.spectral_fitting.bestfit(`*sim:* Simulator *(page 379)*, *processors: Optional[list] = None*`)`

Return a list of best fit spectrum ordered relative to the methods in the simulator object.

Parameters

- `sim` (Simulator (page 379)) – The simulator object.
- `processors` (*list*) – List of SignalProcessor objects ordered according to the methods in the simulator object.

`mrsimulator.utils.spectral_fitting.residuals(`*sim:* Simulator *(page 379)*, *processors: Optional[list] = None*`)`

Return a list of residuals corresponding to the best fit spectrum. The list is based on the order of methods in the simulator object.

Parameters

- `sim` (Simulator (page 379)) – The simulator object.
- `processors` (*list*) – List of SignalProcessor objects ordered according to the methods in the simulator object.

Part VII

Project details

CHAPTER

TWENTYONE

CHANGELOG

21.1 v1.0.0rc

21.1.1 What's new

Features

- New 2D sideband-sideband correlation simulations.
- Support for gamma angle averaging. *sim.config* now holds a new `number_of_gamma_angles` attribute.
- Support for quadrupolar-shielding cross frequency interactions. *freq_contrib* includes new `Quad_Shielding_cross_0`, `Quad_Shielding_cross_2`, `Quad_Shielding_cross_4` literals.
- Support for user-defined isotopes using the `Isotope.register()` method. See the simulation gallery for use cases and examples.
- Shortcuts for frequency contributions, such as `Shielding`, `Isotropic`, and `cross`. Sets of contributions can also be excluded by placing an exclamation mark in front of the string, for example `"!Shielding"` excludes shielding interactions.
- New functions for fitting Czjzek and Extended Czjzek tensor distribution models to experimental spectra. See the examples gallery for more information.
- Adds `DelayEvent` to the events library.
- Support for python 3.11

Simulator

- New instance method for the *Simulator* class – *.optimize()* – which pre-computes transition pathways before least-squares minimization. This improves the efficiency of least-squares minimizations.

Czjzek and Extended Czjzek - The Czjzek model now uses an analytical expression for calculating the probability distribution greatly improving quality and calculation speed.

Bug Fixes

- Fixed bug where `MixingEnum` class had no attribute `json` (Issue [#260](https://github.com/deepanshs/mrsimulator/issues/260))
- Fixed 0Hz crash issue in apodization.
- Fix bug related to pydantic v2.0 validation. The library is restricted to pydantic<v2.0
- Fix latex build error in pdf docs.
- Fixed the bug where the csdm object origin offset was incorrectly set. The origin offset is now *w_ref*.
- Fix bug when calculating ppm scale for large reference offsets.

Additional updates

- Dropped support for python 3.7

21.2 v0.7.0

21.2.1 What's new

Features

- Support for complex amplitude simulation.
- New isotropic interpolation schemes. Added `isotropic_interpolation` as a *sim.config* parameter. Allowed values are `linear` and `Gaussian`.
- New `larmor_freq(B0)` function added to the *Isotope* class which returns the Larmor frequency of the isotope, given a magnetic flux density. For example, *H1.larmor_freq(B0=9.40)*
- New weak J and dipolar coupling enumerations added to *freq_ contrib*.
- New command-line interface (CLI) tools for mrsimulator.
- Added 200+ NMR active isotopes to the library.
- Support for python 3.10

Method

- New Event classes—`SpectralEvent` and `MixingEvent`. The MixingEvent controls the transition amplitude mixing in a multi-event method.
- New `TotalMixing` and `NoMixing` mixing query enumerations for quick scripting of common mixing events.
- New weights attribute for the `TransitionPathway` object, which holds the probability of the transition pathway based on the mixing events defined within the method.
- New `plot()` function in Method class, which generates a visual representation of the method's events, transition pathways, rotor angle, etc.
- Support for concurrent mixing events.
- Support for negative spectral width in a spectral dimension.
- Deprecated `Method1D` and `Method2D` classes. Use the generic `mrsimulator.method.Method` object for custom 1D and 2D methods.

SpinSystem

- New function `simplify()` to simplifies a spin system object to a list of irreducible spin systems.
- New function `site_generator()` added to the utility collection sub-module, which simplifies the process of creating Site objects in bulk.
- Added gyromagnetic ratio and quadrupole moment metadata for all isotopes, including unstable isotopes.

SignalProcessor

- New *SkewGaussian*, *TopHat*, and *Mask* apodization functions were added to the signal processor module.

Documentation

- Restructured documentation layout
- Improved troubleshooting section.
- Added section *User Guide* detailing the use and attributes of most objects.
- Added section *Method* demonstrating how to create custom **Method** objects.
- Condensed simulation/fitting gallery by removing redundant examples.
- New gallery demonstrating signal processing functions.

21.2.2 Changes

- *reduced_dict* function is deprecated, use *json(units=False)* instead.
- The *mrsimulator.signal_processing* module is renamed to *mrsimulator.signal_processor*
- Drop support for Python version 3.6
- Added Channel validation for named methods #177
- Optimized memory usage and performance of the *single_site_system_generator* utility function.

21.2.3 Bug fixes

- Fixed bug where spectral interpolation resulted in a segmentation fault.
- Fixed memory leak issue in the C code.
- Fixed bug in query combination involving multiple quadrupolar queries. #188.
- Fixed bug related to unsigned/signed integers crashing on M1 macs.

21.2.4 Breaking changes

For most users Mrsimulator is currently in development, and the new release includes breaking changes from v0.6. Please review these changes and make changes according.

- The *mrsimulator.methods* module is renamed as *mrsimulator.method.lib*.
- The *mrsimulator.signal_processing* module is renamed to *mrsimulator.signal_processor*.
- The *data* attribute of *SignalProcessor.apply_operations(data=...)* is renamed to *dataset*. Use *SignalProcessor.apply_operations(dataset=...)*
- The *transition_query* attribute of the *mrsimulator.method.SpectralEvent* class is renamed to *transition_queries*.
- The *mrsimulator.method.query.RotationalQuery* class is renamed to *mrsimulator.method.query.RotationQuery*

For advanced users - Complete redesign of the *TransitionQuery* object. Please refer to the documentation for details.

21.3 v0.6.0

21.3.1 What's new

- ⋆ Improved simulation performance. ⋆ See our benchmark.
- Simulation of one-dimensional spectra of coupled spin systems. The frequency contributions from the coupled sites include weak J-couplings and weak dipolar couplings.
- New *Coupling* (page 399) class.
- Added a new `Sites` class that holds a list of Site objects. The Sites class method, `to_pd()`, exports the sites as a pandas data frame.
- A new method, `sites()`, is added to the Simulator object, which returns a list of unique `Sites` objects within the Simulator object across multiple spin systems.
- Added three new arguments to the `single_site_system_generator()` method, 'site_labels', 'site_names', and 'site_descriptions'.

21.3.2 Changes

- The *get_isotopes()* (page 391) method from the SpinSystem object, will now return *Isotope* (page 479) objects by default. Use the *symbol=True* argument of the method to get a list of string isotopes.
- The `to_freq_dict()` function is deprecated.
- The *D* symmetry of *transition_query* attribute from *Method2D* method is now None by default.
- *BlochDecayCTSpectrum* is an alias for *BlochDecayCentralTransitionSpectrum* class.

21.3.3 Bug fixes

- Fixed a bug related to *get_spectral_dimensions()* utility method in cases when CSDM dimension objects have negative increment.
- Fixed a bug resulting in the non-conserved spectral area after a Gaussian apodization.
- Fixed a bug in Gaussian apodization, which raised an error when the FWHM argument is a scalar.
- Fixed bug causing multi-dataset fit to fail.

21.4 v0.5.1

21.4.1 Bug fixes

- Fixed a bug that was causing incorrect spectral binning when the frequency contribution is pure isotropic.

21.4.2 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.

21.5 v0.5.0

21.5.1 What's new

- ⋆ Improved simulation performance. ⋆ See our benchmark.

The update introduces various two-dimensional methods for simulating NMR spectrum.

- Introduces a generic one-dimensional method, `Method1D`.
- Introduces a generic two-dimensional method, `Method2D`.
- Specialized two-dimensional methods for multi-quantum variable-angle spinning with build-in affine transformations.
  - *ThreeQ_VAS* (page 438),
  - *FiveQ_VAS* (page 444),
  - *SevenQ_VAS* (page 450)
- Specialized two-dimensional methods for satellite-transition variable-angle spinning with build-in affine transformations.
  - *ST1_VAS* (page 457),
  - *ST2_VAS* (page 463),

- Specialized two-dimensional isotropic/anisotropic sideband correlation method, *SSB2D* (page 469).

21.5.2 Other changes

- The *get_transition_pathways()* (page 407) method no longer return a numpy array, instead a python list.
- Renamed *mrsimulator.methods* module to *mrsimulator.method.lib*.

21.6 v0.4.0

21.6.1 What's new!

- ⋆ Improved simulation performance. ⋆ See our benchmark.
- New *CzjzekDistribution* (page 493) and *ExtCzjzekDistribution* (page 495) 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()* (page 484), for generating a list of single-site spin systems from a 1D list/array of respective tensor parameters.

21.7 v0.3.0

21.7.1 What's new!

- ⋆ Improved simulation performance. ⋆ See our benchmark.
- 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.

21.7.2 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.

21.7.3 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`.

21.8 v0.2.x

21.8.1 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.

21.8.2 Bug fixes

- Fixed amplitude normalization. The spectral amplitude no longer change when the *integration_density*, integration_volume`, or the *number_of_sidebands* attributes change.

21.8.3 Other changes

- Removed plotly-dash app to its own repository.
- Renamed the class `Spectrum` to `Dimension`

21.9 v0.1.3

- Fixed missing files from source tar.

21.10 v0.1.2

- Initial release on pypi.

21.11 v0.1.1

- Added solid state quadrupolar spectrum simulation.
- Added mrsimulator plotly-dash app.

21.12 v0.1.0

- Solid state chemical shift anisotropy spectrum simulation.

CHAPTER

TWENTYTWO

AUTHORS AND CREDITS

- Deepansh Srivastava (Ohio State University)
- Maxwell C. Venetos (UC Berkeley)
- Philip J. Grandinetti (Ohio State University)
- Shyam Dwaraknath (LBNL)
- Alexis McCarthy (Ohio State University)
- Matthew Giammar (Ohio State University)

CHAPTER

TWENTYTHREE

LICENSE

23.1 Mrsimulator License

Mrsimulator is licensed under BSD 3-Clause License

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.

CHAPTER

TWENTYFOUR

ACKNOWLEDGMENT

The development of the **mrsimulator** project was supported in part by the US National Science Foundation under Grants DIBBS OAC 1640899, CHE 1807922, and CHE-2107636 (with partial co-funding from the Ceramics program in the Division of Materials Research).

Part VIII

Reporting Bugs

Submit bug reports or feature requests on the Github issue tracker.

Discussions are welcome on the Github discussion page.

Part IX

How to cite

Please refer to mrsimulator Github page for details.

INDEX

D

E

F

G

I

J

L

M

N

O

P

Q

R

S

T

U

V

Z