nmrsim Documentation

Release 0.6.0

Geoffrey M. Sametz

Feb 17, 2023

Contents:

1 Introduction to nmrsim **1**

2 Installation **3**

3 Overview of the nmrsim Top-Level API **5**
- 3.1 Definitions . . . 5
- 3.2 Scenario: user wants to plot a spectrum for an ABX 3-spin system. . . . 6
- 3.3 Scenario: User wants to simulate individual first-order multiplets . . . 8
- 3.4 Scenario: User wants to simulate a spectrum built from individual components . . . 9
- 3.5 Scenario: User wants to model a specific spin system using an explicit (non-qm) solution . . . 10
- 3.6 Scenario: User wants to model DNMR two-spin exchange, without and with coupling . . . 11

4 Interactive NMR Demo **13**

5 Overview of the Lower-Level nmrsim API **17**
- 5.1 Scenario: user wants to plot a spectrum for an ABX 3-spin system. . . . 17
  - 5.1.1 Method 1: using qm_spinsystem . . . 18
  - 5.1.2 Method 2: via the spin Hamiltonian . . . 19
  - 5.1.3 Method 3: using a discrete mathematical solution . . . 20
  - 5.1.4 Method 4: a first-order simulation . . . 21
- 5.2 Scenario: modeling DNMR spectra . . . 21

6 Demo: Simulation of Tyrosine NMR Spectrum **25**

7 Explanation of First-Generation QM Model **31**
- 7.1 Disclaimer . . . 31
- 7.2 Constructing the Hamiltonian From Scratch . . . 32
  - 7.2.1 Step 1: Each spin gets its own L_x, L_y and L_z operators. . . . 32
  - 7.2.2 Step 2: Create the sums of cartesian products of L operators. . . . 33
  - 7.2.3 Step 3: Add the Zeeman (on-diagonal) terms to the Hamiltonian. . . . 34
  - 7.2.4 Step 4: Add the J-coupling (off-diagonal) terms to the Hamiltonian. . . . 34
- 7.3 Extracting Signal Frequencies and Intensities From the Hamiltonian . . . 34

8 Contributing to nmrsim **39**
- 8.1 Use the library, and give feedback . . . 39
- 8.2 Lend Expertise . . . 39
- 8.3 Become a Developer . . . 40

9 Developers Guide **41**
9.1 Creating a Development Environment . . . 42
9.1.1 Set up git and GitHub . . . 42
9.1.2 Forking the repository . . . 42
9.1.3 Cloning the repository . . . 42
9.1.4 Creating the virtual environment . . . 42
9.1.5 Using venv . . . 42
9.1.6 Using conda . . . 43
9.1.7 Installing nmrsim in developer mode . . . 43
9.2 Making a contribution . . . 44
9.3 Submit a pull request . . . 44
9.4 Code Style and Conventions . . . 45
9.4.1 PEP 8 . . . 45
9.4.2 import sorting . . . 45
9.4.3 type annotations . . . 45
9.4.4 documentation . . . 45

10 Acknowledgements **47**

11 nmrsim **49**
11.1 nmrsim package . . . 49
11.1.1 Submodules . . . 49
11.1.2 nmrsim.discrete module . . . 49
11.1.3 nmrsim.dnmr module . . . 51
11.1.4 nmrsim.firstorder module . . . 55
11.1.5 nmrsim.math module . . . 56
11.1.6 nmrsim.plt module . . . 58
11.1.7 nmrsim.qm module . . . 59
11.1.8 Module contents . . . 61
11.1.8.1 nmrsim . . . 61

12 Indices and tables **63**

Python Module Index **65**

Index **67**

CHAPTER 1

Introduction to nmrsim

nmrsim is a library of tools for simulating NMR spectra, starting from parameters provided by the user (e.g. chemical shift; *J* coupling constants; rate constants for DNMR spectra). Currently, the application is limited to spin-$1/2$ nuclei only, but expanding this to other nuclei is feasible.

The target niche for **nmrsim** are users that want to model NMR spectra but who either are not specialists themselves, and/or who want to model NMR spectra and concepts (e.g. spin Hamiltonians) for instructional purposes. If there is a feature that you would like to see, or a barrier to you using this library, feel free to open an issue on GitHub or to send the author email (sametz at udel dot edu).

The project is inspired by the program WINDNMR by Hans Reich. The goal for Version 1.0 of **nmrsim** is to provide Python tools for the same types of simulations that WINDNMR did: first- and second-order simulation of spin-1/2 spin systems, plus simulation of some dynamic NMR (DNMR) lineshapes. A longer-term goal is to expand the toolset (e.g. to allow higher-spin nuclei, or new DNMR models).

CHAPTER 2

Installation

nmrsim can be installed from the command line:

```
pip install nmrsim
```

See the *Developer Page* for details on installing a developer version into a virtual environment.

CHAPTER 3

Overview of the nmrsim Top-Level API

This notebook gives a tour of the top level classes the nmrsim API provides. These are conveniences that abstract away lower-level API functions. Users wanting more control can consult the full API documentation.

[1]:
```
import os
import sys
import numpy as np
import matplotlib as mpl
mpl.rcParams['figure.dpi']= 300
%matplotlib inline
```

[2]:
```
%config InlineBackend.figure_format = 'svg'  # makes inline plot look less blurry
```

[3]:
```
home_path = os.path.abspath(os.path.join('..', '..', '..'))
if home_path not in sys.path:
    sys.path.append(home_path)

tests_path = os.path.abspath(os.path.join('..', '..', '..', 'tests'))
if tests_path not in sys.path:
    sys.path.append(tests_path)
```

3.1 Definitions

In naming classes, functions, methods, data types etc. certain phrases, taken from NMR nomenclature, have the following interpretations:

- **multiplet** (e.g. the `nmrsim.Multiplet` class): a first-order simulation for one signal (i.e. one or more chemical shift-equivalent nuclei). Examples: doublet, triplet, doublet of triplets, but **not** an AB quartet (which is a second-order pattern for two nuclei).
- **spin system** (e.g. the `SpinSystem` class): a simulation of a set of coupled nuclei.

- **spectrum** (e.g. the `Spectrum` class): a complete collection of first- and/or second-order components for simulating a total NMR spectrum. 'Spectrum' can also refer in general to the simulation results for the system, e.g a peaklist or lineshape (see below).
- **peak**: a pair of frequency (Hz), intensity values corresponding to a resonance in an NMR spectrum. For example, a 1H triplet centered at 100 Hz with J = 10 Hz would have the following peaks: (110, 0.25), (100, 0.5), (90, 0.25).
- **peaklist**: a list of peaks (e.g. [(110, 0.25), (100, 0.5), (90, 0.25)] for the above triplet).
- **lineshape**: a pair of [x_coordinates...], [y_coordinates] arrays for plotting the lineshape of a spectrum.

In this notebook the term **list** is interchangeable with other iterables such as numpy arrays or tuples. As much as possible, nmrsim relies on <"duck typing">(https://en.wikipedia.org/wiki/Duck_typing) to accept a variety of iterables as inputs, converting them to specific types such as numpy arrays as needed. The term **matrix** refers to a 2D array-like object in general, e.g. a list of lists or a 2D numpy array. It does *not* refer specifically to the (marked-for-deprecation) `numpy.matrix` class.

The following idioms are used for arguments: * **v** for a frequency or list of frequencies (similar to ν). * **I** for a signal intensity * **J** for coupling constant data (exact format depends on the implementation).

3.2 Scenario: user wants to plot a spectrum for an ABX 3-spin system.

A spin system can be described using a list of frequencies v and J (coupling constant) data . For this example, a function from nmrsim's test suite will provide some example data:

[4]:
```
# This dataset is for the vinyl group of vinyl acetate, as used in:
# http://www.users.csbsju.edu/~frioux/nmr/ABC-NMR-Tensor.pdf
def rioux():
    v = np.array([430.0, 265.0, 300.0])
    J = np.zeros((3, 3))
    J[0, 1] = 7.0
    J[0, 2] = 15.0
    J[1, 2] = 1.50
    J = J + J.T
    return v, J
```

[5]:
```
v, J = rioux()
print('v: ', v)  # frequencies in Hz
print('J: \n', J)  # matrix of coupling constants
```

```
v:  [430. 265. 300.]
J:
 [[ 0.   7.  15. ]
 [ 7.   0.   1.5]
 [15.   1.5  0. ]]
```

The J matrix is constructed so that J[a, b] is the coupling constant between v[a] and v[b]. The diagonal elements should be 0.

The `SpinSystem` class can be used to model a set of coupled nuclei.

[6]:
```
from nmrsim import SpinSystem
```

[7]:
```
abx_system = SpinSystem(v, J)
```

```
From hamiltonian_sparse:
Lz is type:  <class 'sparse._coo.core.COO'>
Lproduct is type:  <class 'sparse._coo.core.COO'>
```

The `SpinSystem.peaklist()` method returns the peaklist for the simulation:

[8]:
```
abx_system.peaklist()
```

```
From hamiltonian_sparse:
Lz is type:  <class 'sparse._coo.core.COO'>
Lproduct is type:  <class 'sparse._coo.core.COO'>
```

[8]:
```
[(260.6615285748296, 0.23011249131787795),
 (291.3191136690316, 0.22882003310401866),
 (419.5193577561387, 0.29107244545559474),
 (292.8468885409388, 0.2138123115725174),
 (426.4877446901904, 0.26629867696733883),
 (262.18930344673686, 0.24876061726413431),
 (434.5231959501799, 0.2300458619680737),
 (267.62991550888137, 0.24855578963215708),
 (306.32295186307283, 0.29251680284079634),
 (441.49158288423155, 0.21257278181929304),
 (307.85072673497996, 0.2648680204713065),
 (269.1576903807885, 0.27256416758689195)]
```

You can plot this data with the visualization library of your choice. However, the `nmrsim.plt` library has functions for convenient plotting of common nmrsim data types. The `plt.mplplot` function will take a peaklist and use matplotlib to plot the corresponding lineshape. The optional keyword argument `y_max` can be used to set the maximum for the y-axis (and `y_min` for the minimum).

[9]:
```
from nmrsim.plt import mplplot
```

[10]:
```
mplplot(abx_system.peaklist(), y_max=0.2);
```

```
From hamiltonian_sparse:
Lz is type:  <class 'sparse._coo.core.COO'>
Lproduct is type:  <class 'sparse._coo.core.COO'>
[<matplotlib.lines.Line2D object at 0x7fb911655610>]
```

To plot the spectra as a "stick" style plot (single lines for each peak, rather than a simulated lineshape), you can use the mplplot_stick function instead of mplplot:

[11]:
```
from nmrsim.plt import mplplot_stick
```

[12]:
```
# The range of the x axis can be specified using the 'limits' keyword argument:
mplplot_stick(abx_system.peaklist(), y_max=0.3, limits=(250, 320));
```

```
From hamiltonian_sparse:
Lz is type:  <class 'sparse._coo.core.COO'>
Lproduct is type:  <class 'sparse._coo.core.COO'>
```

SpinSystem defaults to second-order simulation of a spin system. If the SpinSystem object is instantiated with the `second_order=False` keyword argument, or if the SpinSystem.second_order attribute is set to `False`, first-order simulation will be performed instead.

[13]:
```
abx_system.second_order = False
mplplot(abx_system.peaklist(), y_max=0.2);
```

```
[<matplotlib.lines.Line2D object at 0x7fb90ef0d890>]
```

Depending on the resolution of the plot and how the data points for the lineshape are interpolated, the peak heights may not look identical. The correct relative intensities can be seen in the stick plot, however:

[14]:
```
mplplot_stick(abx_system.peaklist(), y_max=0.3);
```

3.3 Scenario: User wants to simulate individual first-order multiplets

The Multiplet class can be used to represent an individual first-order multiplet.

[15]:
```
from nmrsim import Multiplet
```

Required arguments for Multiplet are the central frequency `v`, the intensity `I` ("integration") in the absence of coupling, and a list of coupling data `J`. These arguments become attributes of Multiplet. Each list entry is a tuple of (J value in Hz, number of nuclei causing the coupling). For example, the following Multiplet represents: 1200 Hz, 2H, td, J = 7.1, 1.1 Hz.

[16]:
```
# 1200 Hz, 2H, td, J= 7.1, 1.1 Hz
td = Multiplet(1200.0, 2, [(7.1, 2), (1.1, 1)])
print(td.v)
print(td.I)
print(td.J)
```

```
1200.0
2
[(7.1, 2), (1.1, 1)]
```

The `Multiplet.peaklist()` method returns the peaklist for the multiplet:

[17]:
```
mplplot_stick(td.peaklist());
```

[18]:
```
mplplot(td.peaklist());
```

```
[<matplotlib.lines.Line2D object at 0x7fb911703990>]
```

Multiplet attributes can be modified.

[19]:
```
td2 = Multiplet(1200.0, 2, [(7.1, 2), (1.1, 1)])
td2.v = 1100
mplplot(td2.peaklist());
```

```
[<matplotlib.lines.Line2D object at 0x7fb91181d050>]
```

If a Multiplet is multiplied by a scalar, a new Multiplet is returned that has all intensities multiplied by the scalar. In-place multiplication (`*=`) modifies the original Multiplet object.

[20]:
```
td3 = td2 * 2
td2 *= 2
assert td2 is not td3
mplplot(td2.peaklist());
```

```
[<matplotlib.lines.Line2D object at 0x7fb911879810>]
```

Multiplets are equal to each other if their peaklists are equal.

[21]:
```
assert td2 == td3
```

Division and division in place is also possible:

[22]:
```
td4 = td2 / 2
td2 /= 2
assert td4 == td2
```

If two multiplets are added together, the result is a `Spectrum` object. See the next Scenario for the usage of `Spectrum`.

3.4 Scenario: User wants to simulate a spectrum built from individual components

Any object that has a `.peaklist()` method can be used to create a Spectrum object.

A Spectrum object can be specifically created by providing a list of components as the first argument:

[23]:
```
from nmrsim import Spectrum
```

[24]:
```
two_td = Spectrum([td, td3])
```

[25]:
```
mplplot(two_td.peaklist());
```

```
[<matplotlib.lines.Line2D object at 0x7fb913ab15d0>]
```

A Spectrum object is also returned from certain binary operations, such as addition:

[26]:
```
td3.v = 1000
td4.v = 900

all_tds = td + td2 + td3 + td4
mplplot(all_tds.peaklist());
```

```
[<matplotlib.lines.Line2D object at 0x7fb9139fa4d0>]
```

A Spectrum can be composed from both first- and second-order components:

[27]:
```
combo_spectrum = abx_system + td3 + td4

# mplplot has an optional y_max keyword argument to set the max range of the y-axis
mplplot(combo_spectrum.peaklist(), y_max=0.4);
```

```
[<matplotlib.lines.Line2D object at 0x7fb913ac6b90>]
```

3.5 Scenario: User wants to model a specific spin system using an explicit (non-qm) solution

The nmrsim.partial module contains "canned" mathematical solutions for second-order systems.

Example: simulate the AB part of an ABX3 system

[28]:
```
from nmrsim.discrete import ABX3
```

[29]:
```
help(ABX3)
```

```
Help on function ABX3 in module nmrsim.discrete:

ABX3(Jab, Jax, Jbx, Vab, Vcentr)
    Simulation of the AB part of an ABX3 spin system.

    Parameters
    ---------
    Jab : float
        the Ha-Hb coupling constant (Hz).
    Jax : float
        the Ha-Hb coupling constant (Hz).
    Jbx : float
        the Ha-Hb coupling constant (Hz).
    Vab : float
        the difference in the frequencies (Hz) of Ha and Hb in the absence of
        coupling. Positive when vb > va.
    Vcentr : float
        the frequency (Hz) for the center of the AB signal.

    Returns
    -------
    [(float, float)...]
        a list of (frequency, intensity) tuples.
```

[30]:
```
abx3_peaklist = ABX3(-12, 7, 7, 14, 150)
mplplot(abx3_peaklist, y_max=0.25);
```

```
[<matplotlib.lines.Line2D object at 0x7fb913f5b590>]
```

Here is an alternate, non-qm simulation for the ABX system from the SpinSystem demonstration:

[31]:
```
from nmrsim.discrete import ABX
```

[32]:
```
help(ABX)
```

```
Help on function ABX in module nmrsim.discrete:

ABX(Jab, Jax, Jbx, Vab, Vcentr, vx, normalize=True)
    Non-QM approximation for an ABX spin system. The approximation assumes
    that Hx is very far away in chemical shift from Ha/Hb.

    Parameters
    ---------
    Jab : float
```

(continues on next page)

(continued from previous page)

```
        The Ha-Hb coupling constant (Hz).
    Jax : float
        The Ha-Hx coupling constant (Hz).
    Jbx : float
        The Hb-Hx coupling constant (Hz).
    Vab : float
        The difference in the frequencies (in the absence of
        coupling) of Ha and Hb (Hz).
    Vcentr : float
        The frequency (Hz) for the center of the AB signal.
    vx : float
        The frequency (Hz) for Hx in the absence of coupling.

    normalize: bool (optional)
        whether the signal intensity should be normalized. If false, the total
        signal intensity happens to be ~12.

    Returns
    -------
    [(float, float)...]
        a list of (frequency, intensity) tuples.
```

[33]:
```
abx_peaklist = ABX(1.5, 7, 15, 35, 282.5, 430)
mplplot(abx_peaklist, y_max=0.4);
```

```
[<matplotlib.lines.Line2D object at 0x7fb91421dc50>]
```

3.6 Scenario: User wants to model DNMR two-spin exchange, without and with coupling

The nmrsim.dnmr library provides functions for calculating DNMR lineshapes, and classes to describe these systems. Currently, models for two uncoupled nuclei and two coupled nuclei are provided.

[34]:
```
from nmrsim.dnmr import DnmrTwoSinglets, DnmrAB
```

For: va = 165 Hz, vb = 135 Hz, k = 65.9 s-1, line widths (at the slow exchange limit) wa and wb = 0.5 Hz, and population of state a = 0.5 (i.e. 50%):

[35]:
```
two_singlet_system = DnmrTwoSinglets(165.00, 135.00, 65.9, 0.50, 0.50, 0.50)
```

[36]:
```
from nmrsim.plt import mplplot_lineshape
```

[37]:
```
mplplot_lineshape(*two_singlet_system.lineshape());
```

Class attributes can be changed. In the previous case, k = 65.9 -1 corresponds to the point of coalescence. When the rate of exchange is lower, two separate peaks are observed.

[38]:
```
two_singlet_system.k = 5
```

[39]:
```
mplplot_lineshape(*two_singlet_system.lineshape());
```

What if the relative populations of states a and b are 75% and 25%, respectively?

[40]:
```
two_singlet_system.pa = 0.75
mplplot_lineshape(*two_singlet_system.lineshape());
```

To model an AB-like system of two coupled nuclei undergoing exchange, use the DnmrAB class. In the following example, the frequencies are the same as for the previous system. J = 5 Hz, k = 12 -1, and the line width (at the slow exchange limit) is 0.5 Hz.

[41]:
```
from nmrsim.dnmr import DnmrAB
```

[42]:
```
AB = DnmrAB(165, 135, 5, 10, 0.5)
```

[43]:
```
mplplot_lineshape(*AB.lineshape());
```

CHAPTER 4

Interactive NMR Demo

There are many ways to create interactive plots in a Juypter notebook, and the visualization ecosystem is constantly changing. For example the Holoviz tool suite (http://holoviz.org/) looks promising (especially the possibility of creating a web application using Panel). Another interesting option is nbinteract (https://www.nbinteract.com/).

This notebook currently uses ipywidgets and bokeh to create some simple NMR demonstrations.

[1]:
```
import os
import sys
module_path = os.path.abspath(os.path.join('..'))
if module_path not in sys.path:
    sys.path.append(module_path)
```

[2]:
```
from nmrsim.dnmr import dnmr_AB
help(dnmr_AB)
```

```
Help on function dnmr_AB in module nmrsim.dnmr:

dnmr_AB(va, vb, J, k, w, limits=None, points=800)
    Simulate the DNMR lineshape for two coupled nuclei undergoing exchange
    (AB or AX pattern at the slow-exchange limit).

    Parameters
    ---------
    va, vb : float
        frequencies of a and b nuclei (at the slow exchange limit,
        in the absence of coupling)
    J : float
        the coupling constant between the two nuclei.
    k : float
        rate constant for state A--> state B
    w : float
        peak widths at half height (at the slow-exchange limit).
    limits : (int or float, int or float), optional
        The minimum and maximum frequencies (in any order) for the simulation.
    points : int
```

(continues on next page)

(continued from previous page)

```
        The length of the returned arrays (i.e. the number of points plotted).

    Returns
    -------
    x, y : numpy.array, numpy.array
        Arrays for the x (frequency) and y (intensity) lineshape data points.

    See Also
    --------
    DnmrAB : A class representation for this simulation.

    References
    ----------
    See the documentation for the nmrsim.dnmr module.
```

[3]:
```
args = (
    200,  # va
    100,  # vb
    10,   # J
    0.1,  # k
    0.5   # w
)
```

[4]:
```
from ipywidgets import interact
```

[5]:
```
from bokeh.io import push_notebook, show, output_notebook
from bokeh.plotting import figure
output_notebook()
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_load.v0+json

[6]:
```
# get initial xy data
x, y = dnmr_AB(*args)
```

[7]:
```
p = figure(title = 'DNMR AB Interactive Plot',
           height=300,
           width=600)
r = p.line(x, y)
```

[8]:
```
def interactive_ab(va=110, vb=100, J=10, k=0.1, w=0.5):
    args = (va, vb, J, k, w)
    x, y = dnmr_AB(*args)
    r.data_source.data['y'] = y
    r.data_source.data['x'] = x
    push_notebook()
```

[9]:
```
show(p, notebook_handle=True)
interact(interactive_ab, k=(0.1, 100))
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

```
interactive(children=(IntSlider(value=110, description='va', max=330, min=-110),␣
↪IntSlider(value=100, descript...
```

[9]:
```
<function __main__.interactive_ab(va=110, vb=100, J=10, k=0.1, w=0.5)>
```

[ ]:

CHAPTER 5

Overview of the Lower-Level nmrsim API

This notebook gives a tour of some of the lower-level API functions. We recommend that you start with the ***API Introduction*** notebook for a higher-level overview.

[1]:
```
import os
import sys
import numpy as np
import matplotlib as mpl
mpl.rcParams['figure.dpi']= 300
%matplotlib inline
```

[2]:
```
%config InlineBackend.figure_format = 'svg'  # makes inline plot look less blurry
```

[3]:
```
home_path = os.path.abspath(os.path.join('..', '..', '..'))
if home_path not in sys.path:
    sys.path.append(home_path)
```

[4]:
```
tests_path = os.path.abspath(os.path.join('..', '..', '..', 'tests'))
if tests_path not in sys.path:
    sys.path.append(tests_path)
```

[5]:
```
from nmrsim import plt, qm
```

5.1 Scenario: user wants to plot a spectrum for an ABX 3-spin system.

The API-Introduction notebook shows a simulation of an ABX 3-spin system using the SpinSystem class. Here, the simulation will be performed first with higher-level functions that take frequency(v) and intensity(J) arguments and return peaklists.

[6]:
```
# This dataset is for the vinyl group of vinyl acetate, as used in:
# http://www.users.csbsju.edu/~frioux/nmr/ABC-NMR-Tensor.pdf
def rioux():
    v = np.array([430.0, 265.0, 300.0])
    J = np.zeros((3, 3))
    J[0, 1] = 7.0
    J[0, 2] = 15.0
    J[1, 2] = 1.50
    J = J + J.T
    return v, J
```

[7]:
```
v, J = rioux()
print('v: ', v)  # frequencies in Hz
print('J: \n', J)  # matrix of coupling constants
```

```
v:  [430. 265. 300.]
J:
 [[ 0.   7.  15. ]
 [ 7.   0.   1.5]
 [15.   1.5  0. ]]
```

The J matrix is constructed so that J[a, b] is the coupling constant between v[a] and v[b]. The diagonal elements should be 0.

5.1.1 Method 1: using qm_spinsystem

[8]:
```
abx_system = qm.qm_spinsystem(v, J)
abx_system
```

```
From hamiltonian_sparse:
Lz is type:  <class 'sparse._coo.core.COO'>
Lproduct is type:  <class 'sparse._coo.core.COO'>
```

[8]:
```
[(260.6615285748296, 0.23011249131787795),
 (291.3191136690316, 0.22882003310401866),
 (419.5193577561387, 0.29107244545559474),
 (292.8468885409388, 0.2138123115725174),
 (426.4877446901904, 0.26629867696733883),
 (262.18930344673686, 0.24876061726413431),
 (434.5231959501799, 0.2300458619680737),
 (267.62991550888137, 0.24855578963215708),
 (306.32295186307283, 0.29251680284079634),
 (441.49158288423155, 0.21257278181929304),
 (307.85072673497996, 0.2648680204713065),
 (269.1576903807885, 0.27256416758689195)]
```

[9]:
```
plt.mplplot(abx_system, y_max=0.2);
```

```
[<matplotlib.lines.Line2D object at 0x7fd178d846d0>]
```

{''qm_spinsystem'' is a wrapper that selects one of two functions to perform the calculation: ''qm.secondorder_dense'' and ''qm.secondorder_sparse''. With the default qm_spinsystem keyword arguments ''cache=True'' and ''sparse=True'', the faster function ''secondorder_sparse'' is used. However, if at some point the sparse library becomes unavailable, or if caching of partial solutions is not possible, the slower ''secondorder_dense'' function will be used. These functions can also be used as direct swap-ins for ''qm_spinsystem''.}

5.1.2 Method 2: via the spin Hamiltonian

This is not recommended for casual users, but may be of interest for teaching NMR theory, or if you want to take control of the process (e.g. obtain a Hamiltonian, and then simulate a spin pulse with it {a feature not currently implemented in `nmrsim`}). A description of the math behind the qm simulations is in the **qm_explanation.ipynb notebook** (currently under construction).

There are two versions of the Hamiltonian constructor. `qm.hamiltonian_sparse` uses cached sparse arrays for faster speed, and `qm.hamiltonian_dense` does not. Here we will use the former.

[10]:
```
H = qm.hamiltonian_sparse(v, J)
print(H)
print(H.todense())
```

```
From hamiltonian_sparse:
Lz is type:  <class 'sparse._coo.core.COO'>
Lproduct is type:  <class 'sparse._coo.core.COO'>
<COO: shape=(8, 8), dtype=complex128, nnz=20, fill_value=0j>
[[ 503.375+0.j    0.   +0.j    0.   +0.j    0.   +0.j    0.   +0.j
     0.   +0.j    0.   +0.j    0.   +0.j]
 [   0.   +0.j  195.125+0.j    0.75 +0.j    0.   +0.j    7.5  +0.j
     0.   +0.j    0.   +0.j    0.   +0.j]
 [   0.   +0.j    0.75 +0.j  234.125+0.j    0.   +0.j    3.5  +0.j
     0.   +0.j    0.   +0.j    0.   +0.j]
 [   0.   +0.j    0.   +0.j    0.   +0.j  -72.625+0.j    0.   +0.j
     3.5  +0.j    7.5  +0.j    0.   +0.j]
 [   0.   +0.j    7.5  +0.j    3.5  +0.j    0.   +0.j   62.375+0.j
     0.   +0.j    0.   +0.j    0.   +0.j]
 [   0.   +0.j    0.   +0.j    0.   +0.j    3.5  +0.j    0.   +0.j
  -230.875+0.j    0.75 +0.j    0.   +0.j]
 [   0.   +0.j    0.   +0.j    0.   +0.j    7.5  +0.j    0.   +0.j
     0.75 +0.j -199.875+0.j    0.   +0.j]
 [   0.   +0.j    0.   +0.j    0.   +0.j    0.   +0.j    0.   +0.j
     0.   +0.j    0.   +0.j -491.625+0.j]]
```

SpinSystem defaults to second-order simulation of a spin system. If the SpinSystem object is instantiated with the `second_order=False` keyword argument, or if the SpinSystem.second_order attribute is set to `False`, first-order simulation will be performed instead.

`qm.solve_hamilton` accepts a *dense* Hamiltonian array and the number of spins in the system, to give a peaklist:

[11]:
```
peaklist = qm.solve_hamiltonian(H.todense(), nspins=3)
peaklist
```

[11]:
```
array([[260.66152857,   0.92044387],
       [291.31911367,   0.91527407],
       [419.51935776,   1.16428207],
       [292.84688854,   0.85524358],
       [426.48774469,   1.06518765],
       [262.18930345,   0.99503588],
       [434.52319595,   0.92017735],
       [267.62991551,   0.99421657],
       [306.32295186,   1.17005946],
       [441.49158288,   0.85028549],
       [307.85072673,   1.05946506],
       [269.15769038,   1.09024945]])
```

To normalize the intensities so that they add up to 3 (the number of nuclei in the spin system), use `nmrsim.math.normalize_peaklist`:

[12]:
```
from nmrsim.math import normalize_peaklist
plist_normalized = normalize_peaklist(peaklist, 3)
plist_normalized
```

[12]:
```
[(260.6615285748296, 0.23011249131787795),
 (291.3191136690316, 0.22882003310401866),
 (419.5193577561387, 0.29107244545559474),
 (292.8468885409388, 0.2138123115725174),
 (426.4877446901904, 0.26629867696733883),
 (262.18930344673686, 0.24876061726413431),
 (434.5231959501799, 0.2300458619680737),
 (267.62991550888137, 0.24855578963215708),
 (306.32295186307283, 0.29251680284079634),
 (441.49158288423155, 0.21257278181929304),
 (307.85072673497996, 0.2648680204713065),
 (269.1576903807885, 0.27256416758689195)]
```

[13]:
```
plt.mplplot(plist_normalized, y_max=0.2);
```

```
[<matplotlib.lines.Line2D object at 0x7fd178cfd610>]
```

5.1.3 Method 3: using a discrete mathematical solution

The `nmrsim.discrete` module has discrete solutions for some common spin systems. Some are exact (such as discrete.AB for AB quartets) while others are approximations (e.g. `partial.ABX` for an ABX system) or return only part of the solution (e.g. `partial.AAXX` for an AA'XX' system).

The `partial.ABX` function uses an approximation that assumes the X nucleus is very far away in chemical shift from A and B. If accuracy is required, use a second-order calculation instead.

The functions in `nmrsim.discrete` also take different arguments than those usual throughout the rest of the nmrsim library. They are derived from similar functions in Hans Reich's WINDNMR program and use similar inputs.

[14]:
```
from nmrsim.discrete import ABX
help(ABX)
```

```
Help on function ABX in module nmrsim.discrete:

ABX(Jab, Jax, Jbx, Vab, Vcentr, vx, normalize=True)
    Non-QM approximation for an ABX spin system. The approximation assumes
    that Hx is very far away in chemical shift from Ha/Hb.

    Parameters
    ---------
    Jab : float
        The Ha-Hb coupling constant (Hz).
    Jax : float
        The Ha-Hx coupling constant (Hz).
    Jbx : float
        The Hb-Hx coupling constant (Hz).
    Vab : float
        The difference in the frequencies (in the absence of
        coupling) of Ha and Hb (Hz).
    Vcentr : float
        The frequency (Hz) for the center of the AB signal.
    vx : float
```

(continues on next page)

(continued from previous page)

```
        The frequency (Hz) for Hx in the absence of coupling.

    normalize: bool (optional)
        whether the signal intensity should be normalized. If false, the total
        signal intensity happens to be ~12.

    Returns
    -------
    [(float, float)...]
        a list of (frequency, intensity) tuples.
```

[15]:
```
peaklist = ABX(1.5, 7, 15, (265-300), ((265+300)/2), 430)  # JAB, JAX, JBX, Vab,␣
↪Vcentr, vx
plt.mplplot(peaklist, y_max=0.2);
```

```
[<matplotlib.lines.Line2D object at 0x7fd178c32e90>]
```

5.1.4 Method 4: a first-order simulation

The same v/J arguments can be used by `nmrsim.firstorder.first_order_spin_system` to return a peaklist for a first-order simulation:

[16]:
```
from nmrsim.firstorder import first_order_spin_system
peaklist = first_order_spin_system(v, J)
plt.mplplot(peaklist, y_max = 0.2);
```

```
[<matplotlib.lines.Line2D object at 0x7fd178bc21d0>]
```

Individual multiplets can also be modeled using `nmrsim.firstorder.multiplet`. For example, for the X part of the ABX system as a first-order signal, i.e. 430 Hz, 1H, dd, J = 15, 7 Hz:

[17]:
```
from nmrsim.firstorder import multiplet
X = multiplet((430, 1), [(15, 1), (7, 1)])  # args (frequency, integration), [(J, #␣
↪of couplings)...]
print(X)
plt.mplplot(X, y_max=0.2);
```

```
[(419.0, 0.25), (426.0, 0.25), (434.0, 0.25), (441.0, 0.25)]
[<matplotlib.lines.Line2D object at 0x7fd178b5ffd0>]
```

5.2 Scenario: modeling DNMR spectra

The nmrsim.dnmr module provides functions as well as classes for the computation of DNMR lineshapes. Currently there are models for two systems: two uncoupled spins (`dnmr.dnmr_two_singlets`), and two coupled spins (`dnmr.dnmr_AB`, i.e an AB or AX system at the slow-exchange limit).

[18]:
```
from nmrsim.dnmr import dnmr_two_singlets
help(dnmr_two_singlets)
```

```
Help on function dnmr_two_singlets in module nmrsim.dnmr:

dnmr_two_singlets(va, vb, ka, wa, wb, pa, limits=None, points=800)
    Create a the lineshape for a DNMR spectrum of two uncoupled spin-half nuclei.

    Parameters
    ----------
    va, vb : int or float
        The frequencies (Hz) of nuclei 'a' and 'b' at the slow exchange limit.
    ka : int or float
        The rate constant (Hz) for state a--> state b
    wa, wb : int or float
        The peak widths at half height for the 'a' and 'b' singlets at the
        slow-exchange limit.
    pa : float (0 <= pa <= 1)
        The fraction of the population in state a
    limits : (int or float, int or float), optional
        The minimum and maximum frequencies (in any order) for the simulation.
    points : int
        The length of the returned arrays (i.e. the number of points plotted).

    Returns
    -------
    x, y : numpy.array, numpy.array
        Arrays for the x (frequency) and y (intensity) lineshape data points.

    See Also
    --------
    DnmrTwoSinglets : A class representation for this simulation.

    References
    ----------
    See the documentation for the nmrsim.dnmr module.
```

[19]:
```
frequency, intensity = dnmr_two_singlets(165, 135, 1.5, 0.5, 0.5, 0.5) # va, vb, ka,␣
↪wa, wb, pa
frequency[:10], intensity[:10]
```

[19]:
```
(array([85.        , 85.16270338, 85.32540676, 85.48811014, 85.65081352,
        85.8135169 , 85.97622028, 86.13892365, 86.30162703, 86.46433041]),
 array([1.21342220e-05, 1.22103165e-05, 1.22871542e-05, 1.23647452e-05,
        1.24430994e-05, 1.25222271e-05, 1.26021387e-05, 1.26828448e-05,
        1.27643561e-05, 1.28466836e-05]))
```

To plot lineshape data such as the above (a pair of lists, one for all x coordinates and one for the corresponding y coordinates), you can use the visualization library of your choice. For a quick matplotlib representation, you can use `nmrsim.plt.mplplot_lineshape`:

[20]:
```
from nmrsim.plt import mplplot_lineshape
mplplot_lineshape(frequency, intensity);
```

Coalescence for this system occurs at k ~= 65.9 s-1:

[21]:
```
mplplot_lineshape(*dnmr_two_singlets(165, 135, 65.9, 0.5, 0.5, 0.5));
```

CHAPTER 6

Demo: Simulation of Tyrosine NMR Spectrum

This notebook shows how the **nmrsim** library can be used to compose an entire 1H NMR spectrum from scratch.

The nmrsim.plt routines are convenient for quick plots, but for entire spectrums their small size and low resolution is noticeable (e.g. misleading signal intensities).

{TODO: provide ways to customize the plots (e.g. have ‘‘plt.mplplot‘‘ return the actual matplotlib object for customization, or use the peaklist data in another visualization library).}

This tutorial is adapted from the nmrmint tutorial.

(If you’re interested in an app for the simulation of a complete NMR spectrum, see the ‘nmrmit project <https://github.com/sametz/nmrmint>‘__.)

[1]:
```
import os
import sys
import numpy as np
import matplotlib as mpl
mpl.rcParams['figure.dpi']= 300
%config InlineBackend.figure_format = 'svg'  # makes inline plot look less blurry
%matplotlib inline
home_path = os.path.abspath(os.path.join('..', '..', '..'))
if home_path not in sys.path:
    sys.path.append(home_path)

tests_path = os.path.abspath(os.path.join('..', '..', '..', 'tests'))
if tests_path not in sys.path:
    sys.path.append(tests_path)
```

Here is the data for the spectrum of tyrosine in D2O:

```
1H NMR (500 MHz, Deuterium Oxide) δ 7.18 (d, J = 8.5 Hz, 1H), 6.89 (d, J = 8.5 Hz, 1H),
↪ 3.93 (dd, J = 7.7, 5.1 Hz, 1H),
3.19 (dd, J = 14.7, 5.1 Hz, 1H), 3.05 (dd, J = 14.7, 7.8 Hz, 1H).
```

Data is provided in ppm on a 500 MHz spectrometer. We’ll create a function to perform ppm-to-Hz conversions for us:

[2]:
```
def ppm_to_hz(ppm, spec_freq):
    """Given a chemical shift in ppm and spectrometer frequency in MHz, return the␣
↪corresponding chemical shift in Hz."""
    return [d * spec_freq for d in ppm]
```

The two "doublets" in the aromatic region actually comprise an AA'XX' system. This 4-nuclei spin system can be modeled using the SpinSystem class:

[3]:
```
from nmrsim import SpinSystem
```

Create a frequency list (in Hz) for the A, A', X, and X' nuclei:

[4]:
```
v_aaxx = ppm_to_hz([7.18, 7.18, 6.89, 6.89], 500)
v_aaxx
```

[4]:
```
[3590.0, 3590.0, 3445.0, 3445.0]
```

For the *J* values, as a first approximation we'll assume JAX (an JA'X') are close to the faux-doublet splitting of 8.5 Hz. We'll estimate that JAA' and JXX' are about 2 Hz, and that the JAX' and JA'X couplings are about 0 Hz.

[5]:
```
j_aaxx = [[0, 2, 8.5, 0],
          [2, 0, 0, 8.5],
          [8.5, 0, 0, 2],
          [0, 8.5, 2, 0]]
```

[6]:
```
aaxx = SpinSystem(v_aaxx, j_aaxx)
```

```
From hamiltonian_sparse:
Lz is type:  <class 'sparse._coo.core.COO'>
Lproduct is type:  <class 'sparse._coo.core.COO'>
```

[7]:
```
from nmrsim.plt import mplplot, mplplot_lineshape
```

[8]:
```
mplplot(aaxx.peaklist());
```

```
From hamiltonian_sparse:
Lz is type:  <class 'sparse._coo.core.COO'>
Lproduct is type:  <class 'sparse._coo.core.COO'>
[<matplotlib.lines.Line2D object at 0x7ff213d83790>]
```

Next, we'll create the ABX system for the aliphatic protons. For this exercise, we are assuming that the coupling constants that the first-order analysis provided are close enough.

(If accuracy is critical, there are methods for solving the ABX system. For example, see https://www.chem.wisc.edu/areas/reich/nmr/05-hmr-12-abx.htm#solving%20ABX)

[9]:
```
v_abx = ppm_to_hz([3.93,3.19, 3.05], 500)
j_abx = [[0, 5.1, 7.75],
         [5.1, 0, -14.7],  # geminal Js should be negative
         [7.75, -14.7, 0]]
abx = SpinSystem(v_abx, j_abx)
```

```
From hamiltonian_sparse:
Lz is type:  <class 'sparse._coo.core.COO'>
Lproduct is type:  <class 'sparse._coo.core.COO'>
```

[10]:
```
mplplot(abx.peaklist(), y_max=0.2);
```

```
From hamiltonian_sparse:
Lz is type:  <class 'sparse._coo.core.COO'>
Lproduct is type:  <class 'sparse._coo.core.COO'>
[<matplotlib.lines.Line2D object at 0x7ff2115fd210>]
```

These spin systems can be combined into a spectrum:

[11]:
```
tyr_spectrum = aaxx + abx
mplplot(tyr_spectrum.peaklist(), y_max=0.2)
type(tyr_spectrum)
```

```
From hamiltonian_sparse:
Lz is type:  <class 'sparse._coo.core.COO'>
Lproduct is type:  <class 'sparse._coo.core.COO'>
From hamiltonian_sparse:
Lz is type:  <class 'sparse._coo.core.COO'>
Lproduct is type:  <class 'sparse._coo.core.COO'>
[<matplotlib.lines.Line2D object at 0x7ff211580950>]
```

[11]:
```
nmrsim._classes.Spectrum
```

Addition of the two SpinSystem objects returned a Spectrum object.

If peak intensities look off, try using more data points for the lineshape. Here is the same example with ~ 10 data points per Hz:

[12]:
```
points=int((tyr_spectrum.vmax - tyr_spectrum.vmin) * 10)
print(points)
mplplot(tyr_spectrum.peaklist(), y_max=0.5, points=points);
```

```
21838
[<matplotlib.lines.Line2D object at 0x7ff211520310>]
```

The Spectrum class can also provide lineshape data for the spectrum:

[13]:
```
mplplot_lineshape(*tyr_spectrum.lineshape(points=points));
```

```
From hamiltonian_sparse:
Lz is type:  <class 'sparse._coo.core.COO'>
Lproduct is type:  <class 'sparse._coo.core.COO'>
From hamiltonian_sparse:
Lz is type:  <class 'sparse._coo.core.COO'>
Lproduct is type:  <class 'sparse._coo.core.COO'>
```

The Spectrum.linewidth() method has an advantage over the .peaklist() method: it can take into account the linewidths specified by its component Multiplet/SpinSystem objects. The default value is 0.5 Hz, but this can be set to other values.

In D2O, the -OH and -NH protons are exchanged for D and are not seen in the spectrum. If we wanted to include these in the spectrum for pedagogical reasons, we could create broad singlets with the Multiplet class:

[14]:
```
from nmrsim import Multiplet
```

[15]:
```
# frequency in Hz, integration, [empty list for no coupling constants], peakwidth =
↪20 Hz
nh3 = Multiplet(8.3 * 500, 3, [], 20)
tyr_oh = Multiplet(9.8 * 500, 1, [], 10)
tyr_spectrum2 = tyr_spectrum + nh3 + tyr_oh
```

```
From hamiltonian_sparse:
Lz is type:  <class 'sparse._coo.core.COO'>
Lproduct is type:  <class 'sparse._coo.core.COO'>
From hamiltonian_sparse:
Lz is type:  <class 'sparse._coo.core.COO'>
Lproduct is type:  <class 'sparse._coo.core.COO'>
From hamiltonian_sparse:
Lz is type:  <class 'sparse._coo.core.COO'>
Lproduct is type:  <class 'sparse._coo.core.COO'>
From hamiltonian_sparse:
Lz is type:  <class 'sparse._coo.core.COO'>
Lproduct is type:  <class 'sparse._coo.core.COO'>
```

A Spectrum can have its .vmin and .vmax attributes reset to give a full spectral window (defaults are to provide a 50 Hz margin):

[16]:
```
tyr_spectrum2.default_limits()  # resets limits, and returns vmin, vmax tuple
```

[16]:
```
(1462.9870013439968, 4950.0)
```

[17]:
```
points2 = int((tyr_spectrum2.vmax - tyr_spectrum2.vmin) * 10)
mplplot_lineshape(*tyr_spectrum2.lineshape(points=points2));
```

```
From hamiltonian_sparse:
Lz is type:  <class 'sparse._coo.core.COO'>
Lproduct is type:  <class 'sparse._coo.core.COO'>
From hamiltonian_sparse:
Lz is type:  <class 'sparse._coo.core.COO'>
Lproduct is type:  <class 'sparse._coo.core.COO'>
```

What if you want the x axis to be in ppm?

[18]:
```
# A future version of nmrsim should extend the API to facilitate using ppm in
↪simulations.
# For now, simulations use Hz only, and ppm conversions need to be done manually.

tyr_spectrum2.vmin = -0.5 * 500
tyr_spectrum2.vmax = 10.5 * 500
x, y = tyr_spectrum2.lineshape(points=50000)
x_ppm = x / 500
mplplot_lineshape(x_ppm, y, limits=(-0.5, 10.5));
```

```
From hamiltonian_sparse:
Lz is type:  <class 'sparse._coo.core.COO'>
Lproduct is type:  <class 'sparse._coo.core.COO'>
From hamiltonian_sparse:
Lz is type:  <class 'sparse._coo.core.COO'>
Lproduct is type:  <class 'sparse._coo.core.COO'>
```

[ ]:
```
```

CHAPTER 7

Explanation of First-Generation QM Model

A description of the algorithms for computing second-order spectra follows.

7.1 Disclaimer

The author is not an NMR spectroscopist by training. Ultimately, I would like to understand, and be able to explain, the quantum mechanics behind the entire process of simulating an NMR spectrum. For now, here is a "recipe" of the steps to arrive at the spin Hamiltonian, and how its eigensolution can be used to calculate frequencies and intensities.

Two sources in particular enabled this: 1. Materials by Ilya Kuprov at SpinDynamics.org, particularly Module I, Lecture 5 and the Matlab code of Module II, Lecture 05 and 06. 2. Materials by Frank Rioux at St. John's University and College of St. Benedict. In particular, *ABC Proton NMR Using Tensor Algebra* was very helpful.

[1]:
```
from IPython.core.display import display, HTML
display(HTML("<style>.container { width:100% !important; }</style>"))
```

```
<IPython.core.display.HTML object>
```

[2]:
```
import os
import sys

import numpy as np
from math import sqrt
from scipy.linalg import eigh
from scipy.sparse import kron, csc_matrix, csr_matrix, lil_matrix, bmat

# import bokeh.io
# import bokeh.plotting
```

[3]:
```
home_path = os.path.abspath(os.path.join('..'))
if home_path not in sys.path:
    sys.path.append(home_path)
```

(continues on next page)

(continued from previous page)

```
tests_path = os.path.abspath(os.path.join('..', 'tests'))
if tests_path not in sys.path:
    sys.path.append(tests_path)
```

7.2 Constructing the Hamiltonian From Scratch

Start with the Pauli matrices:

$$\sigma_x = \begin{pmatrix} 0 & \frac{1}{2} \\ \frac{1}{2} & 0 \end{pmatrix}, \sigma_y = \begin{pmatrix} 0 & -\frac{i}{2} \\ \frac{i}{2} & 0 \end{pmatrix}, \sigma_z = \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & -\frac{1}{2} \end{pmatrix} \tag{7.1}$$

plus the identity matrix $I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$

[4]:
```
sigma_x = np.array([[0, 1 / 2], [1 / 2, 0]])
sigma_y = np.array([[0, -1j / 2], [1j / 2, 0]])
sigma_z = np.array([[1 / 2, 0], [0, -1 / 2]])
unit = np.array([[1, 0], [0, 1]])
```

The required inputs are a list of frequencies ν_i and a matrix of J_{ij} coupling constants:

[5]:
```
v = [10.0, 20.0]
J = np.array([[0, 5], [5, 0]])
```

From these "ingredients", the steps for computing the spin Hamiltonian are:

7.2.1 Step 1: Each spin gets its own L_x, L_y and L_z operators.

These are formed from Kronecker products between $\sigma_{x/y/z}$ and I operators.

Each individual product, for n spins, uses $1\sigma_{x/y/z}$ and (n - 1) I operators. They all differ in where in the sequence the $\sigma_{x/y/z}$ operator is placed.

For 3 spins, and using L_z for example:

$$L_{z_1} = \sigma_z \otimes I \otimes I \tag{7.2}$$
$$L_{z_2} = I \otimes \sigma_z \otimes I \tag{7.3}$$
$$L_{z_3} = I \otimes I \otimes \sigma_z \tag{7.4}$$

These operators are stored in a 4-D numpy array of shape (3, n, 2n, 2n). This can be visualized as a 2D matrix of 2D matrices, where every column contains a set of $L_{x/y/z}$ for one of the spins.

$$L_{col} = \begin{pmatrix} L_{x_1} & L_{x_2} & \dots & L_{x_n} \\ L_{y_1} & L_{y_2} & \dots & L_{y_n} \\ L_{z_1} & L_{z_2} & \dots & L_{z_n} \end{pmatrix} \tag{7.5}$$

[6]:
```
nspins = len(v)
L = np.empty((3, nspins, 2 ** nspins, 2 ** nspins), dtype=np.complex128)
for n in range(nspins):
```

(continues on next page)

(continued from previous page)

```
    Lx_current = 1
    Ly_current = 1
    Lz_current = 1

    for k in range(nspins):
        if k == n:
            Lx_current = np.kron(Lx_current, sigma_x)
            Ly_current = np.kron(Ly_current, sigma_y)
            Lz_current = np.kron(Lz_current, sigma_z)
        else:
            Lx_current = np.kron(Lx_current, unit)
            Ly_current = np.kron(Ly_current, unit)
            Lz_current = np.kron(Lz_current, unit)

    L[0][n] = Lx_current
    L[1][n] = Ly_current
    L[2][n] = Lz_current
```

7.2.2 Step 2: Create the sums of cartesian products of L operators.

The off-diagonal components of the Hamiltonian H require calculating Cartesian products of the L operators. Python is a slow language, but numpy/sparse calculations use faster C/Fortran libraries. "Vectorizing" computations to use these math routines as much as possible (e.g. to avoid "for" loops and "if" statements) can dramatically speed up the computations, which otherwise become quite slow for 8+ spins. The fastest nmrsim computations pre-solve parts of the solutions (spin operators, their products, and transition matrices) as sparse matrices.

All possible (Cartesian) products of spin operators can effectively be generated by taking the product of L_{col} matrix with its transpose L_{row}:

$$L_{row} = \begin{pmatrix} L_{x_1} & L_{y_1} & L_{z_1} \\ L_{x_2} & L_{y_2} & L_{z_2} \\ \vdots & \vdots & \vdots \\ L_{x_n} & L_{y_n} & L_{z_n} \end{pmatrix} \tag{7.6}$$

$$L_{product} = L_{row} \cdot L_{col} \tag{7.7}$$

$$= \begin{pmatrix} L_{x_1}L_{x_1} + L_{y_1}L_{y_1} + L_{z_1}L_{z_1} & L_{x_1}L_{x_2} + L_{y_1}L_{y_2} + L_{z_1}L_{z_2} & \dots & L_{x_1}L_{x_n} + L_{y_1}L_{y_n} + L_{z_1}L_{z_n} \\ L_{x_2}L_{x_1} + L_{y_2}L_{y_1} + L_{z_2}L_{z_1} & L_{x_2}L_{x_2} + L_{y_2}L_{y_2} + L_{z_2}L_{z_2} & \dots & L_{x_2}L_{x_n} + L_{y_2}L_{y_n} + L_{z_2}L_{z_n} \\ \vdots & & \ddots & \\ L_{x_n}L_{x_1} + L_{y_n}L_{y_1} + L_{z_n}L_{z_1} & L_{x_n}L_{x_2} + L_{y_n}L_{y_2} + L_{z_n}L_{z_2} & \dots & L_{x_n}L_{x_n} + L_{y_n}L_{y_n} + L_{z_n}L_{z_n} \end{pmatrix} \tag{7.8}$$

This may be computationally wasteful, since not many of these products are likely to be required. However, they can all be calculated "outside of Python", and then saved as sparse arrays for future retrieval, largely negating the computational expense.

How to accomplish this in numpy is not straightforward to a novice (such as the author of this notebook), but fortunately there's StackOverflow:

https://stackoverflow.com/questions/47752324/matrix-multiplication-on-4d-numpy-arrays

[7]:
```
L_T = L.transpose(1, 0, 2, 3)
Lproduct = np.tensordot(L_T, L, axes=((1, 3), (0, 2))).swapaxes(1, 2)
```

7.2.3 Step 3: Add the Zeeman (on-diagonal) terms to the Hamiltonian.

$$H_{Zeeman} = \sum_{i=1}^{n} \nu_i L_{z_i} \tag{7.9}$$

[8]:
```
Lz = L[2]  # array of Lz operators
H = np.tensordot(v, Lz, axes=1)
H
```

[8]:
```
array([[ 15.+0.j,   0.+0.j,   0.+0.j,   0.+0.j],
       [  0.+0.j,  -5.+0.j,   0.+0.j,   0.+0.j],
       [  0.+0.j,   0.+0.j,   5.+0.j,   0.+0.j],
       [  0.+0.j,   0.+0.j,   0.+0.j, -15.+0.j]])
```

7.2.4 Step 4: Add the J-coupling (off-diagonal) terms to the Hamiltonian.

$$H_J = \sum_{i=1}^{n} \sum_{j=1}^{n} \frac{J_{ij}}{2} (L_{x_i} L_{x_j} + L_{y_i} L_{y_j} + L_{z_i} L_{z_j}) \tag{7.10}$$

$$H = H_{Zeeman} + H_J \tag{7.11}$$

The magical numpy calculation that achieves this looks a lot less scary than the math notation!

[9]:
```
J = np.array(J)  # convert to numpy array first
scalars = 0.5 * J
H += np.tensordot(scalars, Lproduct, axes=2)
H
```

[9]:
```
array([[ 16.25+0.j,   0.  +0.j,   0.  +0.j,   0.  +0.j],
       [  0.  +0.j,  -6.25+0.j,   2.5 +0.j,   0.  +0.j],
       [  0.  +0.j,   2.5 +0.j,   3.75+0.j,   0.  +0.j],
       [  0.  +0.j,   0.  +0.j,   0.  +0.j, -13.75+0.j]])
```

7.3 Extracting Signal Frequencies and Intensities From the Hamiltonian

To simulate a "modern" NMR experiment, a 90° pulse and FID acquisition is simulated, followed by Fourier transform. This is the approach used in Kuprov's Matlab code, and should be the required approach for any experiment requiring a more elaborate pulse sequence.

For a simple NMR spectrum, we can adopt a "continuous wave spectrometer" approach. We can find the resonance frequencies and their relative intensities directly from the spin Hamiltonian. The time-independent Schrodinger equation $H\Psi = E\Psi$ is solved for eigenvectors and corresponding eigenvalues.

For each ψ_i, the eigenvectors are the coefficients c_n for each pure spin state. For a two-spin system, for example,

$\psi_i = c_1\alpha\alpha + c_2\alpha\beta + c_3\beta\alpha + c_4\beta\beta.$

and the corresponding eigenvector would be $\begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix}$ For a one-spin system, the two states for "spin-up" ($\uparrow$ or α) and

for "spin-down" ($\downarrow$ or β) are represented by vectors $\begin{bmatrix}1\\0\end{bmatrix}$ and $\begin{bmatrix}0\\1\end{bmatrix}$, respectively. For "pure" multiple-spin states, their vectors are obtained by taking tensor products of these vectors. For example:

$$\alpha\alpha = \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}1\\0\\0\\0\end{bmatrix} \tag{7.12}$$

$$\alpha\beta = \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\1\\0\\0\end{bmatrix} \tag{7.13}$$

$$\beta\alpha = \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0\\0\\1\\0\end{bmatrix} \tag{7.14}$$

$$\beta\beta = \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\0\\0\\1\end{bmatrix} \tag{7.15}$$

A (coincidental?) consequence of this is that the index for H, expressed in binary form as a series of 0s and 1s, is the eigenvector for the associated pure spin state (cf. Rioux's *ABC Proton NMR Using Tensor Algebra*).

Since allowed transitions change the total spin of a system by $\pm$ 1, this is analogous to transitions only being allowed between spin states whose binary indices only differ at one bit. In computing terms, if the Hamming weight of the two indices differ by exactly 1, the transition is allowed.

Knowing this, we can create a transition probability matrix T, where $T_{ij} = 1$ if a transition between states i and j are allowed, and 0 if not.

[10]:
```
# function was optimized by only calculating upper triangle and then adding
# the lower.
n = 2 ** nspins
T = np.zeros((n, n))
for i in range(n - 1):
    for j in range(i + 1, n):
        if bin(i ^ j).count('1') == 1:
            T[i, j] = 1
T += T.T
T
```

[10]:
```
array([[0., 1., 1., 0.],
       [1., 0., 0., 1.],
       [1., 0., 0., 1.],
       [0., 1., 1., 0.]])
```

The eigenvector solutions for the Hamiltonian include two pure states ("all-up/α" and "all-down/β", plus mixed states. We can construct a matrix V_{col} where each column of the matrix is an eigenvector solution, in their indexed order:

$$V_{col} = \begin{pmatrix}\psi_1 & \psi_2 & \dots & \psi_n\end{pmatrix} = \left(\begin{bmatrix}c_1\\c_2\\\vdots\\c_n\end{bmatrix}_1 \quad \begin{bmatrix}c_1\\c_2\\\vdots\\c_n\end{bmatrix}_2 \quad \dots \quad \begin{bmatrix}c_1\\c_2\\\vdots\\c_n\end{bmatrix}_n \right) \tag{7.16}$$

and where its transpose $V_{row} = V_{col}^T$ has an eigenvector for each row:

$$V_{row} = \begin{pmatrix} \psi_1 \\ \psi_2 \\ \vdots \\ \psi_n \end{pmatrix} = \begin{pmatrix} \begin{bmatrix} c_1 & c_2 & \dots & c_n \end{bmatrix}_1 \\ \begin{bmatrix} c_1 & c_2 & \dots & c_n \end{bmatrix}_2 \\ \vdots \\ \begin{bmatrix} c_1 & c_2 & \dots & c_n \end{bmatrix}_n \end{pmatrix} \tag{7.17}$$

The intensity matrix I can be obtained by taking $V_{row} \cdot T \cdot V_{col}$ and squaring it element-wise, so that I_{ij} is the relative probability of a transition between the ψ_i and ψ_j states. The difference in energy between the two states gives the frequency in Hz.

The numpy procedures to calculate (frequency, intensity) signals from the eigensolution of H is again rather magical:

[11]:
```
E, V = np.linalg.eigh(H)
V = V.real
I = np.square(V.T.dot(T.dot(V)))
I_upper = np.triu(I)  # symmetry makes it possible to use only one half of the matrix␣
↪for faster calculation
E_matrix = np.abs(E[:, np.newaxis] - E)
E_upper = np.triu(E_matrix)
combo = np.stack([E_upper, I_upper])
iv = combo.reshape(2, I.shape[0] ** 2).T
cutoff = 0.001  # an arbitrary cutoff where peaks below this intensity are filtered␣
↪out of the solution
peaklist =  iv[iv[:, 1] >= cutoff]
peaklist
```

[11]:
```
array([[ 6.90983006,  0.5527864 ],
       [18.09016994,  1.4472136 ],
       [23.09016994,  0.5527864 ],
       [11.90983006,  1.4472136 ]])
```

Currently the mplplot function requires a list of tuples. An easy way to convert is to make sure that this simulation is normalized in intensity for 2H:

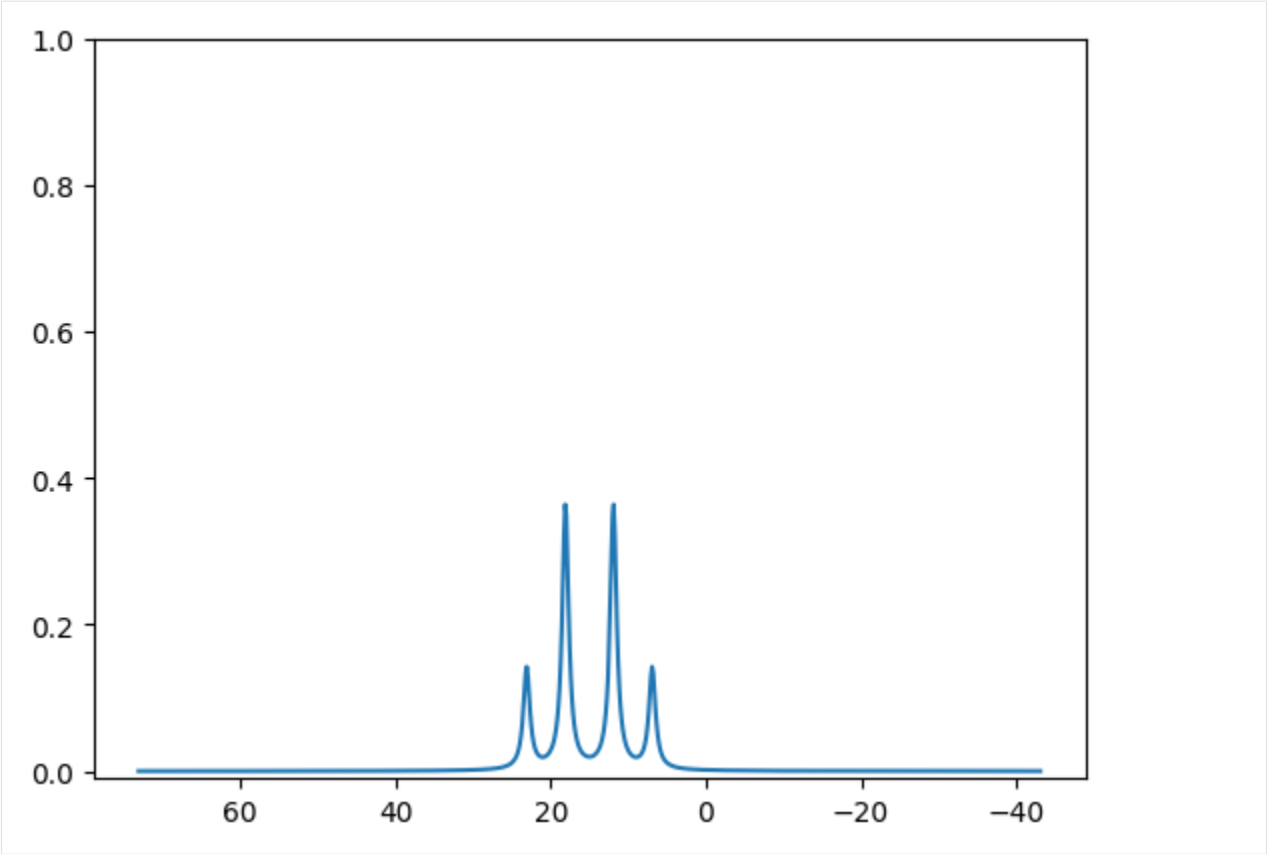
[12]:
```
from nmrsim.math import normalize_peaklist
normalized_plist = normalize_peaklist(peaklist, 2)
normalized_plist
```

[12]:
```
[(6.9098300562505255, 0.276393202250021),
 (18.090169943749473, 0.7236067977499789),
 (23.090169943749473, 0.276393202250021),
 (11.909830056250525, 0.7236067977499789)]
```

[13]:
```
from nmrsim.plt import mplplot
```

[14]:
```
mplplot(normalized_plist);
```

```
[<matplotlib.lines.Line2D object at 0x7f777cff2d50>]
```


1.0
0.8
0.6
0.4
0.2
0.0
60
40
20
0
−20
−40

CHAPTER 8

Contributing to nmrsim

Code of Conduct

You don't need to be an experienced programmer, or NMR spectroscopist, to contribute to nmrsim– the creator of the project is neither! Here are some ways that you can contribute to the project:

8.1 Use the library, and give feedback

Prior to the release of a Version 1, the easiest way to contribute is to use the library and give your feedback. If you are a GitHub user, you can open an issue; if not, you can email the creator: sametz at udel dot edu.

Feedback includes bug reports, feature suggestions and such, but in particular feedback on *how* you use nmrsim (or what *keeps you* from using nmrsim) is valuable. Is the API awkward, non-intuitive, or un-Pythonic? Is the documentation for a feature unclear (or even missing)?

It is a best practice (see: Semantic Versioning) to not break backwards compatability within a major version, because it also breaks user trust. A Version 1 release is a promise that no updates will break your programs that use Version 1, but a future Version 2 may break things. User feedback prior to a Version 1 release can help create a user-friendly library API before it is "set in stone".

8.2 Lend Expertise

If you have experience in NMR spectroscopy, but not necessarily programming, you may be able to describe how the library can be improved. Could the calculations be faster? Are there other models (such as other DNMR systems) that can be added? Much of the work that has gone into nmrsim involved creating and then speed optimizing, the quantum-mechanical calculations for second-order systems (`nmrsim.qm`). There is probably room for improvement here.

If you have experience with Python projects, but not necessarily with NMR spectroscopy, you may help create a "professional" library. The creator of the project is learning Python best practices "on the fly". This includes:

- testing (pytest),

- documentation (Sphinx),
- python packaging,
- automation (e.g. tox or nox),
- continuous integration (CI)

Also, Pythonistas can point out any "broken windows" the project may have (bad docstrings, anti-patterns, etc.)

8.3 Become a Developer

You can also contribute by forking the project and making a pull request. Contributions to the `nmrsim` core code are welcome, but you can also contribute to:

- tests (we aim for 100% code coverage)
- documentation
- tutorials
- jupyter notebooks

To get started, see the *Developer Page*.

CHAPTER 9

Developers Guide

Code of Conduct

This is a basic guide to setting up a development environment on your computer, to get you up and running with the nmrsim code base. It also provides brief guidelines for how to make a pull request. If you would like more information or guidance, or if these instructions are not working for you, you can create an Issue on GitHub and this guide can be improved.

Contents

- *Developers Guide*
  - *Creating a Development Environment*
    - *Set up git and GitHub*
    - *Forking the repository*
    - *Cloning the repository*
    - *Creating the virtual environment*
    - *Using venv*
    - *Using conda*
    - *Installing nmrsim in developer mode*
  - *Making a contribution*
  - *Submit a pull request*
  - *Code Style and Conventions*
    - *PEP 8*
    - *import sorting*
    - *type annotations*

* *documentation*

9.1 Creating a Development Environment

9.1.1 Set up git and GitHub

If you are completely new to Github, you may have to set up an account and install git. See the GitHub Help on Git Setup for more information.

9.1.2 Forking the repository

Navigate to the nmrsim GitHub page, and click the "Fork" icon in the upper right. This will create your own copy of the repository that you can feel free to alter.

See the related GitHub Help on Forking a Repo for more info.

9.1.3 Cloning the repository

Using a terminal (e.g. Command Prompt on Windows; // on Mac), navigate to the directory where you would like to create the nmrsim project folder, then enter:

```
git clone https://github.com/your_GitHub_name_here/nmrsim.git
cd nmrsim
```

The URL can be obtained from your fork's GitHub page by clicking the "Clone or Download" button and copying the URL.

See the GitHub Help on Forking a Repo for more information.

9.1.4 Creating the virtual environment

For any Python project, you don't want to install into your operating system's Python. Instead, you should create a custom Python environment and install nmrsim plus dependencies there.

This can be tricky, and especially for Windows users may pose a significant barrier to entry. The instructions below have been tested for Mac and for Windows 10 machines that had an Anaconda install (and thus Python 3.6+). Linux users should be able to follow the Mac instructions, but can give feedback if they have trouble developing on a Linux machine. The examples use Python 3.7, but nmrsim should be compatible for 3.6+ (i.e. since the introduction of f-string syntax).

9.1.5 Using venv

If your system already has a python version of 3.6 or higher, you can create a virtual environment from the command line with:

```
python -m venv env
```

This creates an 'env' folder with the python environment. Note that on your system you may need to use 'python3', 'python3.7' etc. instead of 'python' if you have more than one version of python installed (e.g. 'python3' if 'python' refers to version 2.7).

You can activate the environment with:

```
source env/bin/activate (Mac)
env\Scripts\activate.bat (Windows)
```

NOTE: If you have an Anaconda install, and try to run the tests, it may fail. If you see two indicators for the environment in parentheses in the terminal, e.g.:

```
(env) (base) ...$
```

enter `conda deactivate` to make sure the conda environment isn't superseding the venv environment. You should see the (base) indicator disappear.

If your system does *not* have a Python version 3.6+ already installed, or if you want to have more than one version of Python on your system, look into the pyenv (Mac/Linux) or pyenv-win (Windows) libraries. Note that Windows 10 users can now get Python 3.7+ via the Microsoft store.

If you wish to deactivate the venv at any point, enter `deactivate` from the command line.

9.1.6 Using conda

It's common for scientists to use an Anaconda or miniconda installation to manage their Python (or other software) dependencies. However, you will be installing packages in development mode with *pip*, and sometimes there are conflicts with pip- and conda-installed packages. This may work on your system, but if there are problems with package conflicts it may be best to use the venv option.

For example, the instructions worked on a 2019 Macbook Pro, but failed on a Windows 10 machine (despite the conda 3.7 environment being activated, the system Python 3.5 installation was still used to run pytest, causing any code with an f-string to fail).

To create a new Python environment named "nmrsim" and activate it, use the command line:

```
conda create --name nmrsim python=3.7
conda activate nmrsim (Mac)
activate nmrsim (Windows)
```

If you later want to exit this environment, you can activate another environment, or enter:

```
conda deactivate (Mac)
deactivate (Windows)
```

9.1.7 Installing nmrsim in developer mode

If you were to just install nmrsim directly from PyPI (by "pip install"), or if you just ran *setup.py*, the current nmrsim would be installed into your python environment *immutably*. Any changes you made to the code would not be noticed by you or the tests. Instead, you will install the package in "developer mode". This will install nmrsim, plus dependencies. It will also install the developer dependencies, which are not required by casual nmrsim users, but are required for developers to run tests, check formatting and so on. From the command line, in the top nmrsim directory that contains setup.py, enter:

```
pip install -e ".[dev]"
```

To check your installation, run the tests using pytest, then navigate to the *docs* directory and build the documentation:

```
pytest
cd docs
make html
```

There will be several pop-up plots that are visual tests for correct behavior; close these windows as they pop up to proceed through the tests.

As an extra test of a correct installation, you can deliberately break some of the code you're working on and re-run the tests to see the tests fail (assuming the code was covered by the tests).

9.2 Making a contribution

Create a git branch with a descriptive name for your contribution, e.g.

```
git checkout -b add_dnmr_tutorial
```

Make your changes, and then:

```
pytest
flake8
```

When these tests both pass, navigate to the docs directory, and build the html documentation:

```
make html
```

Open the docs/build/html/index.html page in your browser. If you made changes to the documentation, including public docstrings, navigate to where the change should appear and check that it looks OK. After you're done with the documentation, run:

```
make clean
```

to delete the contents of the build directory prior to publishing your work.

Commit and push to your fork of nmrsim:

```
git status  # check that your work is staged to commit
git commit -m "Brief description of the change you made"
git push
```

9.3 Submit a pull request

See the GitHub Help on creating a pull request from a fork.

Pull requests should be made to nmrsim's 'develop' branch, and not directly to 'master'.

From your GitHub page for your fork, select the name of your working branch from the 'branch' drop-down menu (e.g. "add_dnmr_tutorial" using the above example). Click 'New pull request'.

You should check that 'base repository' is 'sametz/nmrsim', 'base' is 'develop', 'head repository' is 'yourGitHub-name/nmrsim', and 'compare' is your branch name (e.g. 'add_dnmr_tutorial'). Check that you have a commit message (a longer message in the "Leave a comment" text field is optional) and click "Create pull request" when ready.

The package maintainer will respond via GitHub notification. If there is no response after a week, feel free to email them (sametz at udel dot edu) with 'nmrsim' somewhere in the subject line… they may be busy, on vacation or just distracted :) but will eventually respond.

9.4 Code Style and Conventions

If your code is passing the flake8 test, and if the html documentation looks OK, then it should be acceptable. Here are some of the guidelines:

9.4.1 PEP 8

PEP 8 (the Python style guide) is followed, with the following exceptions:

- The max line length is 119, the width of a GitHub preview. This can be exceeded with good reason. The PEP 8 guideline of 79 characters is a good goal, but readability (e.g. splitting up long URLs) shouldn't be sacrificed.
- Some naming conventions are violated for consistency with NMR terms and with pre-existing code. For example, many variables are upper-case single letters, including H for Hamiltonian, J for coupling constant(s), and (scandalously) I (upper-case 'i') for signal intensity.

The project's .flake8 file makes accomodations for these and other exceptions.

9.4.2 import sorting

imports should be sorted into three categories, with a blank line separating the categories:

- standard library
- third-party libraries
- nmrsim modules

Within each, they should be sorted alphabetically (ignoring "from").

9.4.3 type annotations

We currently don't use type annotations, because this is difficult to implement with numpy and related packages.

9.4.4 documentation

The project follows PEP 257's guidelines for docstrings, and adopts Numpy-style docstrings.

Docstrings are only required for public classes and functions (i.e. not for those whose name begins in a single underscore, e.g. _normalize). However, you may document private classes and functions if you wish– it can make the code's purpose clearer to others, and it's possible that private code may at some point be 'promoted' to the public API.

Currently, "test docstrings" are not used. If you think they should, feel free to make a case for them.

The nmrsim project uses Sphinx for documentation, and restructuredtext (.rst) for content. Semantic line breaks are encouraged– they make editing and formatting easier.

CHAPTER 10

Acknowledgements

This project is inspired by Hans Reich's WINDNMR application. I thank him for our conversations, and his sharing of WINDNMR's Visual Basic 6 code.

CHAPTER 11

nmrsim

11.1 nmrsim package

11.1.1 Submodules

11.1.2 nmrsim.discrete module

Non-quantum mechanical solutions for specific second-order patterns.

These are adapted from the routines from WINDNMR[1] by Hans Reich, U. Wisconsin, and equations from Pople, Schneider and Bernstein[2]. Note that many of the names for arguments, etc. are non-Pythonic but chosen to match the WINDNMR interface and source code (for now).

The *partials* module provides the following functions:

- AB: simulates an AB quartet.
- AB2: simulates an AB2 system.
- ABX: simulates an ABX system.
- ABX3: simulates an ABX3 system.
- AAXX: simulates an AA'XX' system.
- AABB: simulates an AA'BB' system.

References

nmrsim.discrete.**AABB**(*Vab*, *Jaa*, *Jbb*, *Jab*, *Jab_prime*, *Vcentr*, *normalize=True*, ***kwargs*)

A wrapper for a second-order AA'BB' calculation, but using the same arguments as WINDNMR.

Parameters

[1] WINDNMR-Pro home page: https://www.chem.wisc.edu/areas/reich/plt/windnmr.htm

[2] Pople, J.A.; Schneider, W.G.; Bernstein, H.J. *High-Resolution Nuclear Magnetic Resonance.* New York: McGraw-Hill, 1959.

- **Vab** (*float*) – the difference in frequency (Hz) between Ha and Hb in the absence of coupling. A positive number indicates vb > va.
- **Jbb, Jab, Jab_prime** (*Jaa,*) – Jaa is the JAA' coupling constant; Jxx the JXX'; Jax the JAX; and JAX_prime the JAX'.
- **Vcentr** (*float*) – the frequency for the center of the signal.
- **normalize** (*bool*) – whether the signal intensity should be normalized.

Returns a list of (frequency, intensity) tuples.

Return type [(float, float)..]

nmrsim.discrete.**AAXX**(*Jaa*, *Jxx*, *Jax*, *Jax_prime*, *Vcentr*, *normalize=True*)

Simulates one half ('A' part) of an AA'XX' spin system.

All frequencies are in Hz.

Parameters

- **Jax, Jax, Jax_prime** (*Jaa,*) – Jaa is the JAA' coupling constant; Jxx the JXX'; Jax the JAX; and JAX_prime the JAX'.
- **Vcentr** (*float*) – the frequency for the center of the signal.
- **normalize** (*bool*) – whether the signal intensity should be normalized (to 2).

Returns a list of (frequency, intensity) tuples.

Return type [(float, float)..]

nmrsim.discrete.**AB**(*Jab*, *Vab*, *Vcentr*, *normalize=True*)

Calculates the signal frequencies and intensities for two strongly coupled protons (Ha and Hb).

Parameters

- **Jab** (*float*) – The coupling constant (Hz) between Ha and Hb
- **Vab** (*float*) – The chemical shift difference (Hz) between Ha and Hb in the absence of coupling.
- **Vcentr** (*float*) – The frequency (Hz) for the center of the AB quartet.
- **normalize** (*bool*) – Whether the signal intensity should be normalized.

Returns A list of four (frequency, intensity) tuples.

Return type [(float, float)..]

nmrsim.discrete.**AB2**(*Jab*, *Vab*, *Vcentr*, *normalize=True*)

Calculates signal frequencies and intensities for an AB2 spin system.

Parameters

- **Jab** (*float*) – the Ha-Hb coupling constant (Hz).
- **Vab** (*float*) – the difference in the frequencies (Hz). A positive value means vb > va; negative means va > vb.
- **Vcentr** (*float*) – the frequency (Hz) for the center of the AB2 signal.
- **normalize** (*bool*) – whether the signal intensity should be normalized.

Returns a list of (frequency, intensity) tuples.

Return type [(float, float)..]

nmrsim.discrete.**ABX**(*Jab*, *Jax*, *Jbx*, *Vab*, *Vcentr*, *vx*, *normalize=True*)

Non-QM approximation for an ABX spin system. The approximation assumes that Hx is very far away in chemical shift from Ha/Hb.

Parameters

- **Jab** (*float*) – The Ha-Hb coupling constant (Hz).
- **Jax** (*float*) – The Ha-Hx coupling constant (Hz).
- **Jbx** (*float*) – The Hb-Hx coupling constant (Hz).
- **Vab** (*float*) – The difference in the frequencies (in the absence of coupling) of Ha and Hb (Hz).
- **Vcentr** (*float*) – The frequency (Hz) for the center of the AB signal.
- **vx** (*float*) – The frequency (Hz) for Hx in the absence of coupling.
- **normalize** (*bool (optional)*) – whether the signal intensity should be normalized. If false, the total signal intensity happens to be ~12.

Returns a list of (frequency, intensity) tuples.

Return type [(float, float)..]

nmrsim.discrete.**ABX3**(*Jab*, *Jax*, *Jbx*, *Vab*, *Vcentr*)

Simulation of the AB part of an ABX3 spin system.

Parameters

- **Jab** (*float*) – the Ha-Hb coupling constant (Hz).
- **Jax** (*float*) – the Ha-Hb coupling constant (Hz).
- **Jbx** (*float*) – the Ha-Hb coupling constant (Hz).
- **Vab** (*float*) – the difference in the frequencies (Hz) of Ha and Hb in the absence of coupling. Positive when vb > va.
- **Vcentr** (*float*) – the frequency (Hz) for the center of the AB signal.

Returns a list of (frequency, intensity) tuples.

Return type [(float, float)..]

11.1.3 nmrsim.dnmr module

The *dnmr* module provides functions for calculating DNMR line shapes, and classes to describe DNMR systems.

The dnmr module provides the following classes:

- *DnmrTwoSinglets*: a sumulation of the lineshape for two uncoupled nuclei undergoing exchange.
- *DnmrAB*: a simulation of the lineshape for two coupled nuclei undergoing exchange (i.e. an AB (or AX) pattern at the slow exchange limit).

The *dnmr* module provides the following functions:

- *dnmr_two_singlets*: for simulating the lineshape for two uncoupled nuclei undergoing exchange[3].
- *dnmr_AB* : for simulating the lineshape for two coupled nuclei undergoing exchange (i.e. an AB (or AX) pattern at the slow exchange limit)[4].

[3] Sandström, J. Dynamic NMR Spectroscopy; Academic Press: New York, 1982.

[4]

References

class nmrsim.dnmr.**DnmrAB**(*va=165.0*, *vb=135.0*, *J=12.0*, *k=12.0*, *w=0.5*, *limits=None*, *points=800*)
Bases: object

Simulate the DNMR lineshape for two coupled nuclei undergoing exchange (AB or AX pattern at the slow-exchange limit).

Parameters

- **vb** (*va,*) – frequencies of a and b nuclei (at the slow exchange limit, in the absence of coupling)
- **J** (*int or float*) – the coupling constant between the two nuclei.
- **k** (*int or float*) – rate constant for state A–> state B
- **w** (*int or float*) – peak widths at half height (at the slow-exchange limit).
- **limits** (*(int or float, int or float), optional*) – The minimum and maximum frequencies (in any order) for the simulation.
- **points** (*int*) – The length of the returned arrays (i.e. the number of points plotted).

See also:

DnmrAB A class representation for this simulation.

References

See the documentation for the nmrsim.dnmr module.

J
The coupling constant (Hz) between the two nuclei.

Returns

Return type int or float

k
The rate constant (Hz) for state A–> state B (must be >0).

Returns

Return type int or float

limits
Give minimum and maximum frequencies for the simulated lineshape.

Returns

Return type (int or float, int or float)

lineshape()
Return the x, y lineshape data for the simulation.

Returns **x, y** – Arrays for the x (frequency) and y (intensity) lineshape data points.

Return type numpy.array, numpy.array

a) Brown, K.C.; Tyson, R.L.; Weil, J.A. J. Chem. Educ. 1998, 75, 1632.
b) an important math correction to the previous reference:
TODO: add reference to correction

points
Give the length of the returned arrays (i.e. the number of points plotted).

Returns

Return type int

va
The frequency of nucleus "a" (Hz) at the slow-exchange limit, in the absence of coupling.

Returns

Return type int or float

vb
The frequency of nucleus "b" (Hz) at the slow-exchange limit, in the absence of coupling.

Returns

Return type int or float

w
The peak width (Hz) at half height (at the slow-exchange limit).

Returns

Return type int or float

class nmrsim.dnmr.**DnmrTwoSinglets**(*va=1*, *vb=0*, *k=0.01*, *wa=0.5*, *wb=0.5*, *pa=0.5*, *limits=None*, *points=800*)

Bases: object

A DNMR simulation for two uncoupled nuclei undergoing exchange.

Parameters

- **vb** (*va,*) – The frequencies (Hz) of nuclei 'a' and 'b' at the slow exchange limit.
- **k** (*int or float*) – The rate constant (Hz) for state a–> state b
- **wb** (*wa,*) – The peak widths at half height for the 'a' and 'b' singlets at the slow-exchange limit.
- **pa** (*float (0 <= pa <= 1)*) – The fraction of the population in state a
- **limits** (*(int or float, int or float), optional*) – The minimum and maximum frequencies (in any order) for the simulation.
- **points** (*int*) – The length of the returned arrays (i.e. the number of points plotted).

See also:

DnmrTwoSinglets A class representation for this simulation

k
The rate constant (Hz) for state A–> state B (must be >0).

Returns

Return type int or float

limits
The minimum and maximum frequencies for the simulated lineshape.

Returns

Return type (int or float, int or float)

lineshape()
Calculate and return the lineshape for the DNMR spectrum.

Returns **x, y** – Arrays for the x (frequency) and y (intensity) lineshape data points.

Return type numpy.array, numpy.array

pa
The fraction of the population in state a. Must be >=0 and <=1.

Returns

Return type float

points
The length of the returned arrays (i.e. the number of points plotted).

Returns

Return type int

va
The frequency of nucleus "a" (Hz) at the slow-exchange limit.

Returns

Return type int or float

vb
The frequency of nucleus "b" (Hz) at the slow-exchange limit.

Returns

Return type int or float

wa
The peak width at half height (Hz) for the 'a' singlet at the slow-exchange limit.

Returns

Return type int or float

wb
The peak width at half height (Hz) for the 'b' singlet at the slow-exchange limit.

Returns

Return type int or float

nmrsim.dnmr.**dnmr_AB**(*va*, *vb*, *J*, *k*, *w*, *limits=None*, *points=800*)
Simulate the DNMR lineshape for two coupled nuclei undergoing exchange (AB or AX pattern at the slow-exchange limit).

Parameters

- **vb** (*va,*) – frequencies of a and b nuclei (at the slow exchange limit, in the absence of coupling)
- **J** (*float*) – the coupling constant between the two nuclei.
- **k** (*float*) – rate constant for state A–> state B
- **w** (*float*) – peak widths at half height (at the slow-exchange limit).
- **limits** (*(int or float, int or float), optional*) – The minimum and maximum frequencies (in any order) for the simulation.
- **points** (*int*) – The length of the returned arrays (i.e. the number of points plotted).

Returns **x, y** – Arrays for the x (frequency) and y (intensity) lineshape data points.

Return type numpy.array, numpy.array

See also:

DnmrAB() A class representation for this simulation.

References

See the documentation for the nmrsim.dnmr module.

nmrsim.dnmr.**dnmr_two_singlets**(*va*, *vb*, *ka*, *wa*, *wb*, *pa*, *limits=None*, *points=800*)
Create a the lineshape for a DNMR spectrum of two uncoupled spin-half nuclei.

Parameters

- **vb** (*va,*) – The frequencies (Hz) of nuclei 'a' and 'b' at the slow exchange limit.
- **ka** (*int or float*) – The rate constant (Hz) for state a–> state b
- **wb** (*wa,*) – The peak widths at half height for the 'a' and 'b' singlets at the slow-exchange limit.
- **pa** (*float (0 <= pa <= 1)*) – The fraction of the population in state a
- **limits** (*(int or float, int or float), optional*) – The minimum and maximum frequencies (in any order) for the simulation.
- **points** (*int*) – The length of the returned arrays (i.e. the number of points plotted).

Returns **x, y** – Arrays for the x (frequency) and y (intensity) lineshape data points.

Return type numpy.array, numpy.array

See also:

DnmrTwoSinglets() A class representation for this simulation.

References

See the documentation for the nmrsim.dnmr module.

11.1.4 nmrsim.firstorder module

"Functions for calculating first-order spectra.

The nmrsim.firstorder module provides the following functions:

- multiplet: performs first-order splitting of a signal into multiple signals.
- **first_order_spin_system: provides a peaklist for several nuclei, using the** same v/J parameters that are used for second-order spin systems. See nmrsim.qm for details on these parameters.

nmrsim.firstorder.**first_order_spin_system**(*v*, *J*)
Create a first-order peaklist of several multiplets from the same v/J arguments used for qm calculations.

This allows a user to model several multiplets at once, rather than creating each multiplet individually. It also provides a "toggle" where the user, or a higher-level function/class (such as nmrsim.SpinSystem) can decide whether a spin system is modeled as first order or second order.

Parameters

- **v** (*array-like [float...]*) – an array of frequencies
- **J** (*2D array-like (square)*) – a matrix of J coupling constants

Returns a combined peaklist of signals for all the multiplets in the spin system.

Return type [(float, float)..]

nmrsim.firstorder.**multiplet**(*signal*, *couplings*)

Splits a set of signals into first-order multiplets.

Parameters

- **signal** (*(float, float)*) – a (frequency (Hz), intensity) tuple;
- **couplings** (*[(float, int)..]*) – A list of (*J*, # of nuclei) tuples. The order of the tuples in couplings does not matter. e.g. to split a signal into a *dt,* J = 8, 5 Hz, use: `couplings = [(8, 2), (5, 3)]`

Returns a sorted peaklist for the multiplet that results from splitting the signal by each J.

Return type [(float, float)..]

11.1.5 nmrsim.math module

A collection of functions for processing simulated NMR spectra.

Terms used: signal: a pair (e.g. tuple) of frequency, intensity values peaklist: a list (or 1D-array-like) of signals.

Provides the following functions:

- add_peaks: combines a list of signals into one signal of average frequency and summed intensity.
- reduce_peaks: processes a peaklist so that signals within a frequency tolerance are added together.
- normalize_peaklist: scales a peaklist so that intensities add to a specific value.
- lorentz: given a frequency, a signal and a linewidth, calculates an intensity. Used to calculate Lorentzian lineshapes for signals.
- get_intensity: given a lineshape and a frequency, find the intensity at the datapoint closest to that frequency.

nmrsim.math.**add_lorentzians**(*linspace*, *peaklist*, *w*)

Given a numpy linspace, a peaklist of (frequency, intensity) tuples, and a linewidth, returns an array of y coordinates for the total line shape.

Parameters

- **linspace** (*array-like*) – Normally a numpy.linspace of x coordinates corresponding to frequency in Hz.
- **peaklist** (*[(float, float)..]*) – A list of (frequency, intensity) tuples.
- **w** (*float*) – Peak width at half maximum intensity.

Returns an array of y coordinates corresponding to intensity.

Return type [float...]

nmrsim.math.**add_peaks**(*plist*)

Reduces a list of (frequency, intensity) tuples to an (average frequency, total intensity) tuple.

Parameters **plist** (*[(float, float)..]*) – a list of (frequency, intensity) tuples

Returns a tuple of (average frequency, total intensity)

Return type (float, float)

nmrsim.math.**get_intensity**(*lineshape*, *x*)
A crude method to find the intensity of data point closest to frequency x. Better: interpolate between two data points if match isn't exact (TODO?)

Parameters

- **lineshape** (*tuple of (x, y) arrays for frequency, intensity data*) –
- **x** (*frequency lookup*) –

Returns float

Return type the intensity at that frequency

nmrsim.math.**get_maxima**(*lineshape*)
Crude function that returns maxima in the lineshape.

Parameters **lineshape** (*tuple of frequency, intensity arrays*) –

Returns

Return type a list of (frequency, intensity) tuples for individual maxima.

nmrsim.math.**lorentz**(*v*, *v0*, *I*, *w*)
A lorentz function that takes linewidth at half intensity (w) as a parameter.

When *v* = *v0*, and *w* = 0.5 (Hz), the function returns intensity I.

Parameters

- **v** (*float*) – The frequency (x coordinate) in Hz at which to evaluate intensity (y coordinate).
- **v0** (*float*) – The center of the distribution.
- **I** (*float*) – the relative intensity of the signal
- **w** (*float*) – the peak width at half maximum intensity

Returns the intensity (y coordinate) for the Lorentzian distribution evaluated at frequency *v*.

Return type float

nmrsim.math.**normalize_peaklist**(*peaklist*, *n=1*)
Normalize the intensities in a peaklist so that total intensity equals value n (nominally the number of nuclei giving rise to the signal).

Parameters

- **peaklist** (*[(float, float)..]*) – a list of (frequency, intensity) tuples.
- **n** (*int or float (optional)*) – total intensity to normalize to (default = 1).

nmrsim.math.**reduce_peaks**(*plist_*, *tolerance=0*)
Takes a list of (x, y) tuples and adds together tuples whose values are within a certain tolerance limit.

Parameters

- **plist** (*[(float, float)..]*) – A list of (x, y) tuples
- **tolerance** (*float*) – tuples that differ in x by <= tolerance are combined using *add_peaks*

Returns a list of (x, y) tuples where all x values differ by > *tolerance*

Return type [(float, float)..]

11.1.6 nmrsim.plt module

The plt module provides convenience functions for creating matplotlib plots, plus applying Lorentzian distributions about signals.

The plt module provides the following functions:

- add_lorentzians: Creates lineshape data from a provided linspace (array of x coordinates) and peaklist).
- mplplot: Creates a lineshape plot from a peaklist and returns the x, y plot data.
- mplplot_stick: Creates a "stick" (matplotlib "stem" plot) plot from a peaklist and returns the x, y plot data.
- mplplot_lineshape: Creates a lineshape plot from provided x, y lineshape data and returns the x, y plot data.

nmrsim.plt.**mplplot**(*peaklist*, *w=1*, *y_min=-0.01*, *y_max=1*, *points=800*, *limits=None*, *hidden=False*)
A matplotlib plot of the simulated lineshape for a peaklist.

Parameters

- **peaklist** (*[(float, float)..]*) – A list of (frequency, intensity) tuples.
- **w** (*float*) – Peak width at half height
- **y_min** (*float or int*) – Minimum intensity for the plot.
- **y_max** (*float or int*) – Maximum intensity for the plot.
- **points** (*int*) – Number of data points.
- **limits** (*(float, float)*) – Frequency limits for the plot.
- **hidden** (*bool*) – Whether showing the plot should be omitted (e.g. to not block CI tests)

Returns **x, y** – Arrays for frequency (x) and intensity (y) for the simulated lineshape.

Return type numpy.array

nmrsim.plt.**mplplot_lineshape**(*x*, *y*, *y_min=None*, *y_max=None*, *limits=None*, *hidden=False*)
A matplotlib plot that accepts arrays of x and y coordinates.

Parameters

- **x** (*array-like*) – The list of x coordinates.
- **y** (*array-like*) – The list of y coordinates.
- **y_min** (*float or int*) – Minimum intensity for the plot. Default is -10% max y.
- **y_max** (*float or int*) – Maximum intensity for the plot. Default is 110% max y.
- **limits** (*(float, float)*) – Frequency limits for the plot.
- **hidden** (*bool*) – Whether showing the plot should be omitted (e.g. to not block CI tests)

Returns **x, y**

Return type The original x, y arguments.

nmrsim.plt.**mplplot_stick**(*peaklist*, *y_min=-0.01*, *y_max=1*, *limits=None*, *hidden=False*)
A matplotlib plot of a spectrum in "stick" (stem) style.

Parameters

- **peaklist** (*[(float, float)..]*) – A list of (frequency, intensity) tuples.

- **y_min** (*float or int*) – Minimum intensity for the plot.
- **y_max** (*float or int*) – Maximum intensity for the plot.
- **limits** (*(float, float)*) – Frequency limits for the plot.
- **hidden** (*bool*) – Whether showing the plot should be omitted (e.g. to not block CI tests)

Returns The arrays of x and y coordinates used for the plot.

Return type numpy.array, numpy.array

11.1.7 nmrsim.qm module

qm contains functions for the quantum-mechanical (second-order) calculation of NMR spectra.

The qm module provides the following attributes:

- **CACHE** [bool (default True)] Whether saving to disk of partial solutions is allowed.
- **SPARSE** [bool (default True)] Whether the sparse library can be used.

The qm module provides the following functions:

- qm_spinsystem: The high-level function for computing a second-order simulation from frequency and J-coupling data.
- hamiltonian_dense: Calculate a spin Hamiltonian using dense arrays (slower).
- hamiltonian_sparse: Calculate a spin Hamiltonian using cached sparse arrays (faster).
- solve_hamiltonian: Calculate a peaklist from a spin Hamiltonian.
- secondorder_dense: Calculate a peaklist for a second-order spin system, using dense arrays (slower).
- secondorder_sparse: Calculate a peaklist for a second-order spin system, using cached sparse arrays (faster).

Notes

Because numpy.matrix is marked as deprecated, starting with Version 0.2.0 the qm code was refactored to a) accommodate this deprecation and b) speed up the calculations. The fastest calculations rely on:

1. the pydata/sparse library. SciPy's sparse depends on numpy.matrix, and they currently recommend that pydata/sparse be used for now.

2. Caching partial solutions for spin operators and transition matrices as .npz files.

If the pydata/sparse package is no longer available, and/or if distributing the library with .npz files via PyPI is problematic, then a backup is required. The qm module for now provides two sets of functions for calculating second-order spectra: one using pydata/sparse and caching, and the other using neither.

nmrsim.qm.**hamiltonian_dense**(*v*, *J*)

Calculate the spin Hamiltonian as a dense array.

Parameters

- **v** (*array-like*) – list of frequencies in Hz (in the absence of splitting) for each nucleus.
- **J** (*2D array-like*) – matrix of coupling constants. J[m, n] is the coupling constant between v[m] and v[n].

Returns **H** – a sparse spin Hamiltonian.

Return type numpy.ndarray

nmrsim.qm.**hamiltonian_sparse**(*v*, *J*)
Calculate the spin Hamiltonian as a sparse array.

Parameters

- **v** (*array-like*) – list of frequencies in Hz (in the absence of splitting) for each nucleus.
- **J** (*2D array-like*) – matrix of coupling constants. J[m, n] is the coupling constant between v[m] and v[n].

Returns **H** – a sparse spin Hamiltonian.

Return type sparse.COO

nmrsim.qm.**qm_spinsystem**(**args*, *cache=True*, *sparse=True*, ***kwargs*)
Calculates second-order spectral data (frequency and intensity of signals) for *n* spin-half nuclei.

Currently, n is capped at 11 spins.

Parameters

- **freqs** (*[float...]*) – a list of *n* nuclei frequencies in Hz.
- **couplings** (*array-like*) – An *n, n* array of couplings in Hz. The order of nuclei in the list corresponds to the column and row order in the matrix, e.g. couplings[0][1] and [1]0] are the J coupling between the nuclei of freqs[0] and freqs[1].
- **normalize** (*bool (optional keyword argument; default = True)*) – True if the intensities should be normalized so that total intensity equals the total number of nuclei.

Returns **peaklist** – of [frequency, intensity] pairs.

Return type [[float, float]..] numpy 2D array

Other Parameters

- **cache** (*bool (default = nmrsim.qm.CACHE)*) – Whether caching of partial solutions (for acceleration) is allowed. Currently CACHE = True, but this provides a hook to modify nmrsim for platforms such as Raspberry Pi where storage space is a concern.
- **sparse** (*bool (default = nmrsim.qm.SPARSE)*) – Whether the pydata sparse library for sparse matrices is available. Currently SPARSE = True, but this provides a hook to modify nmrsim should the sparse library become unavailable (see notes).
- **cutoff** (*float*) – The intensity cutoff for reporting signals (default is 0.001).

Notes

With numpy.matrix marked for deprecation, the scipy sparse array functionality is on shaky ground, and the current recommendation is to use the pydata sparse library. In case a problem arises in the numpy/scipy/ sparse ecosystem, SPARSE provides a hook to use a non-sparse-dependent alternative.

nmrsim.qm.**secondorder_dense**(*freqs*, *couplings*, *normalize=True*, ***kwargs*)
Calculates second-order spectral data (freqency and intensity of signals) for *n* spin-half nuclei.

Parameters

- **freqs** (*[float...]*) – a list of *n* nuclei frequencies in Hz
- **couplings** (*array-like*) – an *n, n* array of couplings in Hz. The order of nuclei in the list corresponds to the column and row order in the matrix, e.g. couplings[0][1] and [1]0] are the J coupling between the nuclei of freqs[0] and freqs[1].

- **normalize** (*bool*) – True if the intensities should be normalized so that total intensity equals the total number of nuclei.

Returns **peaklist** – numpy 2D array of [frequency, intensity] pairs.

Return type [[float, float]..]

Other Parameters **cutoff** (*float*) – The intensity cutoff for reporting signals (default is 0.001).

nmrsim.qm.**secondorder_sparse**(*freqs*, *couplings*, *normalize=True*, ***kwargs*)

Calculates second-order spectral data (frequency and intensity of signals) for *n* spin-half nuclei.

Parameters

- **freqs** (*[float...]*) – a list of *n* nuclei frequencies in Hz
- **couplings** (*array-like*) – an *n, n* array of couplings in Hz. The order of nuclei in the list corresponds to the column and row order in the matrix, e.g. couplings[0][1] and [1]0] are the J coupling between the nuclei of freqs[0] and freqs[1].
- **normalize** (*bool*) – True if the intensities should be normalized so that total intensity equals the total number of nuclei.

Returns **peaklist** – of [frequency, intensity] pairs.

Return type [[float, float]..] numpy 2D array

Other Parameters **cutoff** (*float*) – The intensity cutoff for reporting signals (default is 0.001).

nmrsim.qm.**solve_hamiltonian**(*H*, *nspins*, ***kwargs*)

Calculates frequencies and intensities of signals from a spin Hamiltonian and number of spins.

Parameters

- **H** (*numpy.ndarray (2D)*) – The spin Hamiltonian
- **nspins** (*int*) – The number of spins in the system

Returns

Return type [[float, float]..] numpy 2D array of frequency, intensity pairs.

Other Parameters **cutoff** (*float*) – The intensity cutoff for reporting signals (default is 0.001).

11.1.8 Module contents

11.1.8.1 nmrsim

The nmrsim package provides tools for simulating nuclear magnetic resonance (NMR) spectra.

The API is still in flux. Currently, it includes the following modules:

- dnmr: for modeling Dynamic NMR systems
- firstorder: for modeling first-order spectra
- math: core math routines for handling NMR data
- partial: uses non-quantum-mechanical solutions for common second-order NMR patterns
- plt: convenience plotting routines for NMR results
- qm: quantum-mechanical second-order simulation of spin systems (currently capped at 11 nuclei)

Currently, only spin-1/2 nuclei are accommodated.

The top-level nmrsim namespace provides the following classes:

- Multiplet: a representation of a first-order multiplet (e.g. quartet; doublet of triplets).
- SpinSystem: a representation of a set of coupled nuclei (modeled as either first-order or second-order).
- Spectrum: a collection of components such as Multiplets or SpinSystems that contribute to a total NMR spectrum simulation.

Definitions of Terms Used

In naming classes, functions, methods, data types etc. certain phrases, taken from NMR nomenclature, have the following interpretations:

- **multiplet** (e.g. the *nmrsim.Multiplet* class): a first-order simulation for one signal (i.e. one or more chemical shift-equivalent nuclei). Examples: doublet, triplet, doublet of triplets, but **not** an AB quartet (which is a second-order pattern for two nuclei).
- **spin system** (e.g. the *SpinSystem* class): a simulation of a set of coupled nuclei.
- **spectrum** (e.g. the *Spectrum* class): a complete collection of first- and/or second-order components for simulating a total NMR spectrum. 'Spectrum' can also refer in general to the simulation results for the system, e.g a peaklist or lineshape (see below).
- **peak**: a pair of frequency (Hz), intensity values corresponding to a resonance in an NMR spectrum. For example, a 1H triplet centered at 100 Hz with J = 10 Hz would have the following peaks: (110, 0.25), (100, 0.5), (90, 0.25).
- **peaklist**: a list of peaks (e.g. [(110, 0.25), (100, 0.5), (90, 0.25)] for the above triplet).
- **lineshape**: a pair of [x coordinates. . .], [y coordinates] arrays for plotting the lineshape of a spectrum.

The following idioms are used for arguments: * **v** for a frequency or list of frequencies (similar to the Greek lowercase "nu" character). * **I** for a signal intensity (despite being a PEP8 naming violation) * **J** for coupling constant data (exact format depends on the implementation).

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

Python Module Index

n

nmrsim, 61
nmrsim.discrete, 49
nmrsim.dnmr, 51
nmrsim.firstorder, 55
nmrsim.math, 56
nmrsim.plt, 58
nmrsim.qm, 59

Index

A

AABB() (*in module nmrsim.discrete*), 49
AAXX() (*in module nmrsim.discrete*), 50
AB() (*in module nmrsim.discrete*), 50
AB2() (*in module nmrsim.discrete*), 50
ABX() (*in module nmrsim.discrete*), 50
ABX3() (*in module nmrsim.discrete*), 51
add_lorentzians() (*in module nmrsim.math*), 56
add_peaks() (*in module nmrsim.math*), 56

D

dnmr_AB() (*in module nmrsim.dnmr*), 54
dnmr_two_singlets() (*in module nmrsim.dnmr*), 55
DnmrAB (*class in nmrsim.dnmr*), 52
DnmrTwoSinglets (*class in nmrsim.dnmr*), 53

F

first_order_spin_system() (*in module nmrsim.firstorder*), 55

G

get_intensity() (*in module nmrsim.math*), 57
get_maxima() (*in module nmrsim.math*), 57

H

hamiltonian_dense() (*in module nmrsim.qm*), 59
hamiltonian_sparse() (*in module nmrsim.qm*), 59

J

J (*nmrsim.dnmr.DnmrAB attribute*), 52

K

k (*nmrsim.dnmr.DnmrAB attribute*), 52
k (*nmrsim.dnmr.DnmrTwoSinglets attribute*), 53

L

limits (*nmrsim.dnmr.DnmrAB attribute*), 52
limits (*nmrsim.dnmr.DnmrTwoSinglets attribute*), 53
lineshape() (*nmrsim.dnmr.DnmrAB method*), 52
lineshape() (*nmrsim.dnmr.DnmrTwoSinglets method*), 53
lorentz() (*in module nmrsim.math*), 57

M

mplplot() (*in module nmrsim.plt*), 58
mplplot_lineshape() (*in module nmrsim.plt*), 58
mplplot_stick() (*in module nmrsim.plt*), 58
multiplet() (*in module nmrsim.firstorder*), 56

N

nmrsim (*module*), 61
nmrsim.discrete (*module*), 49
nmrsim.dnmr (*module*), 51
nmrsim.firstorder (*module*), 55
nmrsim.math (*module*), 56
nmrsim.plt (*module*), 58
nmrsim.qm (*module*), 59
normalize_peaklist() (*in module nmrsim.math*), 57

P

pa (*nmrsim.dnmr.DnmrTwoSinglets attribute*), 54
points (*nmrsim.dnmr.DnmrAB attribute*), 52
points (*nmrsim.dnmr.DnmrTwoSinglets attribute*), 54

Q

qm_spinsystem() (*in module nmrsim.qm*), 60

R

reduce_peaks() (*in module nmrsim.math*), 57

S

secondorder_dense() (*in module nmrsim.qm*), 60
secondorder_sparse() (*in module nmrsim.qm*), 61
solve_hamiltonian() (*in module nmrsim.qm*), 61

V

va (*nmrsim.dnmr.DnmrAB attribute*), 53

va (*nmrsim.dnmr.DnmrTwoSinglets attribute*), 54
vb (*nmrsim.dnmr.DnmrAB attribute*), 53
vb (*nmrsim.dnmr.DnmrTwoSinglets attribute*), 54

W

w (*nmrsim.dnmr.DnmrAB attribute*), 53
wa (*nmrsim.dnmr.DnmrTwoSinglets attribute*), 54
wb (*nmrsim.dnmr.DnmrTwoSinglets attribute*), 54