mrinversion Documentation

Release 0.2.0

Deepansh J. Srivastava

Jun 07, 2021

GETTING STARTED

About

The `mrinversion` python package is based on the statistical learning technique for determining the distribution of the magnetic resonance (NMR) tensor parameters from two-dimensional NMR spectra correlating the isotropic to anisotropic frequencies. The library utilizes the mrsimulator package for generating solid-state NMR spectra and scikit-learn package for statistical learning.

Features

The `mrinversion` package includes the **inversion of a two-dimensional solid-state NMR spectrum of dilute spin-systems to a three-dimensional distribution of tensor parameters**. At present, we support the inversion of

- **Magic angle turning (MAT), Phase adjusted spinning sidebands (PASS)**, and similar spectra correlating the isotropic chemical shift resonances to pure anisotropic spinning sideband resonances into a three-dimensional distribution of nuclear shielding tensor parameters, $\rho(\delta_{\text{iso}}, \zeta_\sigma, \eta_\sigma)$, where δ_{iso} is the isotropic chemical shift, and ζ_σ and η_σ, are the shielding anisotropy and asymmetry parameters, respectively, defined using the Haeberlen convention.
- **Magic angle flipping (MAF)** spectra correlating the isotropic chemical shift resonances to pure anisotropic resonances into a three-dimensional distribution of nuclear shielding tensor parameters, $\rho(\delta_{\text{iso}}, \zeta_\sigma, \eta_\sigma)$, where δ_{iso} is the isotropic chemical shift, and ζ_σ and η_σ, are the shielding anisotropy and asymmetry parameters, respectively, defined using the Haeberlen convention.

CHAPTER

ONE

GETTING STARTED

1.1 Installation

1.1.1 Requirements

`mrinversion` has the following strict requirements:

- Python 3.6 or later
- Numpy 1.17 or later

See *Package dependencies* (page 4) for a full list of requirements.

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

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

```
$ python --version
```

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

1.1.2 Installing `mrinversion`

On Google Colab Notebook

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

```
!pip install mrinversion
```

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

On Local machine (Using pip)

The `mrinversion` package utilizes the mrsimulator package for generating the NMR line-shapes.

For **Linux** and **Mac** users, type the following in the terminal to install the package.

```
$ pip install mrinversion
```

For **Windows** users, first, install the mrsimulator package and then install the mrinversion package using the above command.

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

```
$ pip install mrinversion --user
```

Upgrading to a newer version

To upgrade, type the following in the terminal/Prompt,

```
$ pip install mrinversion -U
```

1.2 Package dependencies

The `mrinversion` library depends on the following packages:

Required packages

- numpy>=1.17
- csdmpy>=0.4
- mrsimulator>=0.6 (for generating the NMR line-shape)
- scikit-learn>=0.22.1 (for statistical leaning methods)

Other packages

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

1.3 Introduction

1.3.1 Objective

In `mrinversion`, we solve for the distribution of the second-rank traceless symmetric tensor principal components, through an inversion of a pure anisotropic NMR spectrum.

In the case of the shielding tensors, the pure anisotropic frequency spectra corresponds the cross-sections of the 2D isotropic *v.s.* anisotropic correlation spectrum, such as the 2D One Pulse (TOP) MAS, phase adjusted spinning sidebands (PASS), magic-angle turning (MAT), extended chemical shift modulation (XCS), magic-angle hopping (MAH), magic-angle flipping (MAF), and Variable Angle Correlation Spectroscopy (VACSY). A key feature of all these 2D isotropic/anisotropic correlation spectra——either as acquired or after a shear transformation——is that the anisotropic cross-section can be modeled as a linear combination of subspectra,

$$s(\nu|\delta_{\text{iso}}) = \int_{\mathbf{R}} \mathcal{K}(\nu, \mathbf{R}) f(\mathbf{R}|\delta_{\text{iso}}) d\mathbf{R}, \tag{1.1}$$

where $s(\nu|\delta_{\text{iso}})$ is the observed anisotropic cross-section at a given isotropic shift, δ_{iso}, $\mathcal{K}(\nu, \mathbf{R})$ represents a simulated subspectrum of a nuclear spin system with a given set of parameters, $\mathbf{R}$, and $f(\mathbf{R}|\delta_{\text{iso}})$ is the probability of the respective set of parameters. In Eq. (1.1), $\mathbf{R}$ represents the anisotropic and asymmetry parameters of the shielding tensor.

Note, Eq. (1.1) is a Fredholm integral of the first kind.

1.3.2 Generic Linear problem

Linear inverse problems on Fredholm integral of the first kind are frequently encountered in the scientific community and have the following generic form

$$\mathbf{s} = \mathbf{K} \cdot \mathbf{f}, \tag{1.2}$$

where $\mathbf{K} \in \mathbb{R}^{m \times n}$ is the transforming kernel (matrix), $\mathbf{f} \in \mathbb{R}^n$ is the unknown and desired solution, and $\mathbf{s} \in \mathbb{R}^m$ is the known signal, which includes the measurement noise. When the matrix $\mathbf{K}$ is non-singular and $m = n$, the solution to the problem in Eq. (1.2) has a simple closed-form solution,

$$\mathbf{f} = \mathbf{K}^{-1} \cdot \mathbf{s}. \tag{1.3}$$

The deciding factor whether the solution $\mathbf{f}$ exists in Eq. (1.3) is whether or not the kernel $\mathbf{K}$ is invertible. Often, most scientific problems with practical applications suffer from singular, near-singular, or ill-conditioned kernels, where $\mathbf{K}^{-1}$ doesn't exist. Such types of problems are termed as *ill-posed*. The inversion of a purely anisotropic NMR spectrum to the distribution of the tensorial parameters is one such ill-posed problem.

Regularized linear problem

A common approach in solving ill-posed problems is to employ the regularization methods of form

$$\mathbf{f}^{\dagger} = \underset{\mathbf{f}>0}{\operatorname{argmin}} \left(\|\mathbf{K} \cdot \mathbf{f} - \mathbf{s}\|_2^2 + g(\mathbf{f}) \right), \tag{1.4}$$

where $\|\mathbf{z}\|_2$ is the *l-2* norm of $\mathbf{z}$, $g(\mathbf{f})$ is the regularization term, and $\mathbf{f}^{\dagger}$ is the regularized solution. The choice of the regularization term, $g(\mathbf{f})$, is often based on prior knowledge of the system for which the linear problem is defined. For anisotropic NMR spectrum inversion, we choose the smooth-LASSO regularization.

Smooth-LASSO regularization

Our prior assumption for the distribution of the tensorial parameters is that it should be smooth and continuous for disordered and sparse and discrete for crystalline materials. Therefore, we employ the smooth-lasso method, which is a linear model that is trained with the combined l1 and l2 priors as the regularizer. The method minimizes the objective function,

$$\|\mathbf{K}\cdot\mathbf{f}-\mathbf{s}\|_2^2+\alpha\sum_{i=1}^{d}\|\mathbf{J}_i\cdot\mathbf{f}\|_2^2+\lambda\|\mathbf{f}\|_1, \tag{1.5}$$

where α and λ are the hyperparameters controlling the smoothness and sparsity of the solution $\mathbf{f}$. The matrix $\mathbf{J}_i$ typically reflects some underlying geometry or the structure in the true solution. Here, $\mathbf{J}_i$ is defined to promote smoothness along the i$^{\text{th}}$ dimension of the solution $\mathbf{f}$ and is given as

$$\mathbf{J}_i=\mathbf{I}_{n_1}\otimes\cdots\otimes\mathbf{A}_{n_i}\otimes\cdots\otimes\mathbf{I}_{n_d}, \tag{1.6}$$

where $\mathbf{I}_{n_i}\in\mathbb{R}^{n_i\times n_i}$ is the identity matrix, and $\mathbf{A}_{n_i}$ is the first difference matrix given as

$$\mathbf{A}_{n_i}=\begin{pmatrix}1&-1&0&\cdots&\vdots\\0&1&-1&\cdots&\vdots\\\vdots&\vdots&\vdots&\vdots&0\\0&\cdots&0&1&-1\end{pmatrix}\in\mathbb{R}^{(n_i-1)\times n_i}. \tag{1.7}$$

The symbol $\otimes$ is the Kronecker product. The terms, $(n_1, n_2, \cdots, n_d)$, are the number of points along the respective dimensions, with the constraint that $\prod_{i=1}^{d} n_i = n$, where d is the total number of dimensions in the solution $\mathbf{f}$, and n is the total number of features in the kernel, $\mathbf{K}$.

1.3.3 Understanding the *x-y* plot

A second-rank symmetric tensor, $\mathbf{S}$, in a three-dimensional space, is described by three principal components, s_{xx}, s_{yy}, and s_{zz}, in the principal axis system (PAS). Often, depending on the context of the problem, the three principal components are expressed with three new parameters following a convention. One such convention is the Haeberlen convention, which defines δ_{iso}, ζ, and η, as the isotropic shift, anisotropy, and asymmetry parameters, respectively. Here, the parameters ζ and η contribute to the purely anisotropic frequencies, and determining the distribution of these two parameters is the focus of this library.

Defining the inverse grid

When solving any linear inverse problem, one needs to define an inverse grid before solving the problem. A familiar example is the inverse Fourier transform, where the inverse grid is defined following the Nyquist–Shannon sampling theorem. Unlike inverse Fourier transform, however, there is no well-defined sampling grid for the second-rank traceless symmetric tensor parameters. One obvious choice is to define a two-dimensional ζ-η Cartesian grid.

As far as the inversion problem is concerned, ζ and η are just labels for the subspectra. In simplistic terms, the inversion problem solves for the probability of each subspectrum, from a given pre-defined basis of subspectra, that describes the observed spectrum. If the subspectra basis is defined over a ζ-η Cartesian grid, multiple (ζ, η) coordinates points to the same subspectra. For example, the subspectra from coordinates $(\zeta, \eta = 1)$ and $(-\zeta, \eta = 1)$ are identical, therefore, distinguishing these coordinates from the subspectra becomes impossible.

The issue of multiple coordinates pointing to the same object is not new. It is a common problem when representing polar coordinates in the Cartesian basis. Try describing the coordinates of the south pole using latitudes and longitudes. You can define the latitude, but describing longitude becomes problematic. A similar situation arises in the context of second-rank traceless tensor parameters when the anisotropy goes to zero. You can specify the anisotropy as zero, but defining asymmetry becomes problematic.

Introducing the x-y grid

A simple fix to this issue is to define the (ζ, η) coordinates in a polar basis. We, therefore, introduce a piece-wise polar grid representation of the second-rank traceless tensor parameters, ζ-η, defined as

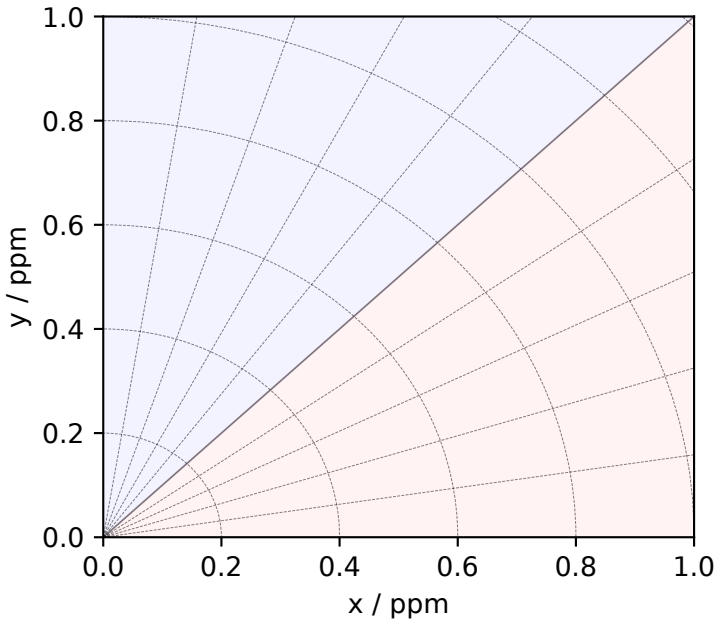
$$r_\zeta = |\zeta| \quad \text{and} \quad \theta = \begin{cases} \frac{\pi}{4}\eta & : \zeta \leq 0, \\ \frac{\pi}{2}\left(1 - \frac{\eta}{2}\right) & : \zeta > 0. \end{cases} \tag{1.8}$$

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

$$x = r_\zeta \cos\theta \quad \text{and} \quad y = r_\zeta \sin\theta. \tag{1.9}$$

In the x-y grid system, the basis subspectra are relatively distinguishable. The `mrinversion` library provides a utility function to render the piece-wise polar grid for your matplotlib figures. Copy-paste the following code in your script.

```
>>> import matplotlib.pyplot as plt
>>> from mrinversion.utils import get_polar_grids
...
>>> plt.figure(figsize=(4, 3.5))
>>> ax=plt.gca()
>>> # add your plots/contours here.
>>> get_polar_grids(ax)
>>> ax.set_xlabel('x / ppm')
>>> ax.set_ylabel('y / ppm')
>>> plt.tight_layout()
>>> plt.show()
```



If you are familiar with the matplotlib library, you may notice that most code lines are the basic matplotlib statements, except for the line that says *get_polar_grids(ax)*. The `get_polar_grids()` (page 24) is a utility function that generates the piece-wise polar grid for your figures.

Here, the shielding anisotropy parameter, ζ, is the radial dimension, and the asymmetry parameter, η, is the angular dimension, defined using Eqs. (1.8) and (1.9). The region in blue and red corresponds to the positive and negative values of ζ, where the

Figure 1.1: The figure depicts the piece-wise polar ζ-η grid represented on an x-y grid. The radial and angular grid lines represent the magnitude of ζ and η, respectively. The blue and red shading represents the positive and negative values of ζ, respectively. The radian grid lines are drawn at every 0.2 ppm increments of ζ, and the angular grid lines are drawn at every 0.2 increments of η. The x and y-axis are $\eta = 0$, and the diagonal $x = y$ is $\eta = 1$.

magnitude of the anisotropy increases radially. The *x* and the *y*-axis are $\eta = 0$ for the negative and positive ζ, respectively. When moving towards the diagonal from *x* or *y*-axes, the asymmetry parameter, η, uniformly increase, where the diagonal is $\eta = 1$.

1.4 Before getting started

1.4.1 Prepping the 2D dataset for inversion

The following is a list of some requirements and recommendations to help prepare the 2D dataset for inversion.

Common recommendations/requirements

	Dataset shear The inversion method assumes that the 2D dataset is sheared, such that one of the dimensions corresponds to a pure anisotropic spectrum. The anisotropic cross-sections are centered at 0 Hz. **Required**: Apply a shear transformation before proceeding.
	Calculate the noise standard deviation Use the noise region of your spectrum to calculate the standard deviation of the noise. You will require this value when implementing cross-validation.

Spinning Sideband correlation dataset specific recommendations

	Data-repeat operation A data-repeat operation on the time-domain signal corresponding to the sideband dimension makes the spinning sidebands look like a stick spectrum after a Fourier transformation, a visual, which most NMR spectroscopists are familiar from the 1D magic-angle spinning spectrum. Like a zero-fill operation, a spinning sideband data-repeat operation is purely cosmetic and adds no information. In terms of computation, however, a data-repeated spinning-sideband spectrum will take longer to solve. **Strongly recommended**: Avoid data-repeat operation.

Magic angle flipping dataset specific recommendations

	Isotropic shift correction along the anisotropic dimension Ordinarily, after shear, a MAF spectrum is a 2D isotropic *v.s.* pure anisotropic frequency correlation spectrum. In certain conditions, this is not true. In a MAF experiment, the sample holder (rotor) physically swaps between two angles ($90° \leftrightarrow 54.735°$). It is possible to have a slightly different external magnetic fields at the two angles, in which case, there is an isotropic component along the anisotropic dimension, which is not removed by the shear transformation. **Required**: Correct for the isotropic offset along the anisotropic dimension by adding an appropriate coordinates-offset.
	Zero-fill operation Zero filling the time domain dataset is purely cosmetic. It makes the spectrum look visually appealing, but adds no information, that is, a zero-filled dataset contains the same information as a non-zero filled dataset. In terms of computation, however, a zero-filled spectrum will take longer to solve. **Recommendation**: If zero-filled, try to keep the total number of points along the anisotropic dimension in the range of 120 - 150 points.
	Sinc wiggles artifacts Kernel correction for spectrum with sinc wiggle artifacts is coming soon.

1.5 Getting started with `mrinversion`

We have put together a set of guidelines for using the *mrinversion* package. We encourage our users to follow these guidelines for consistency.

Let's examine the inversion of a purely anisotropic MAS sideband spectrum into a 2D distribution of nuclear shielding anisotropy parameters. For illustrative purposes, we use a synthetic one-dimensional purely anisotropic spectrum. Think of this as a cross-section of your 2D MAT/PASS dataset.

Import relevant modules

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from matplotlib import rcParams
>>> from mrinversion.utils import get_polar_grids
...
>>> rcParams['pdf.fonttype'] = 42 # for exporting figures as illustrator editable pdf.
...
>>> # a function to plot the 2D tensor parameter distribution
>>> def plot2D(ax, csdm_object, title=''):
...     # convert the dimension from `Hz` to `ppm`.
...     _ = [item.to('ppm', 'nmr_frequency_ratio') for item in csdm_object.dimensions]
...
...     levels = (np.arange(9)+1)/10
...     ax.contourf(csdm_object, cmap='gist_ncar', levels=levels)
...     ax.grid(None)
...     ax.set_title(title)
...     ax.set_aspect("equal")
...
...     # The get_polar_grids method place a polar zeta-eta grid on the background.
...     get_polar_grids(ax)
```

1.5.1 Import the dataset

The first step is getting the sideband spectrum. In this example, we get the data from a CSDM[1] compliant file-format. Import the csdmpy module and load the dataset as follows,

---

Note: The CSDM file-format is a new open-source universal file format for multi-dimensional datasets. It is supported by NMR programs such as SIMPSON[2], DMFIT[3], and RMN[4]. A python package supporting CSDM file-format, csdmpy, is also available.

```
>>> import csdmpy as cp
...
>>> filename = "https://osu.box.com/shared/static/xnlhecn8ifzcwx09f83gsh27rhc5i5l6.csdf"
>>> data_object = cp.load(filename) # load the CSDM file with the csdmpy module
```

Here, the variable *data_object* is a CSDM object. The NMR spectroscopic dimension is a frequency dimension. NMR spectroscopists, however, prefer to view the spectrum on a dimensionless scale. If the dataset dimension within the CSDM object is in frequency, you may convert it into *ppm* as follows,

```
>>> # convert the dimension coordinates from `Hz` to `ppm`.
>>> data_object.dimensions[0].to('ppm', 'nmr_frequency_ratio')
```

In the above code, we convert the dimension at index 0 from *Hz* to *ppm*. For multi-dimensional datasets, use the appropriate indexing to convert individual dimensions to *ppm*.

For comparison, let's also include the true probability distribution from which the synthetic spinning sideband dataset is derived.

```
>>> datafile = "https://osu.box.com/shared/static/lufeus68orw1izrg8juthcqvp7w0cpzk.csdf"
>>> true_data_object = cp.load(datafile) # the true solution for comparison
```

The following is the plot of the spinning sideband spectrum as well as the corresponding true probability distribution.

```
>>> _, ax = plt.subplots(1, 2, figsize=(9, 3.5), subplot_kw={'projection': 'csdm'})
>>> ax[0].plot(data_object)
>>> ax[0].set_xlabel('frequency / ppm')
>>> ax[0].invert_xaxis()
>>> ax[0].set_title('Pure anisotropic MAS spectrum')
...
>>> plot2D(ax[1], true_data_object, title='True distribution')
>>> plt.tight_layout()
>>> plt.savefig('filename.pdf') # to save figure as editable pdf
>>> plt.show()
```

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

[2] Bak M., Rasmussen J. T., Nielsen N.C., SIMPSON: A General Simulation Program for Solid-State NMR Spectroscopy. J Magn Reson. 147, 296330, (2000). DOI:10.1006/jmre.2000.2179

[3] Massiot D., Fayon F., Capron M., King I., Le Calvé S., Alonso B., et al. Modelling one- and two-dimensional solid-state NMR spectra. Magn Reson Chem. 40, 7076, (2002) DOI:10.1002/mrc.984

[4] PhySy Ltd. RMN 2.0; 2019. Available from: https://www.physyapps.com/rmn.

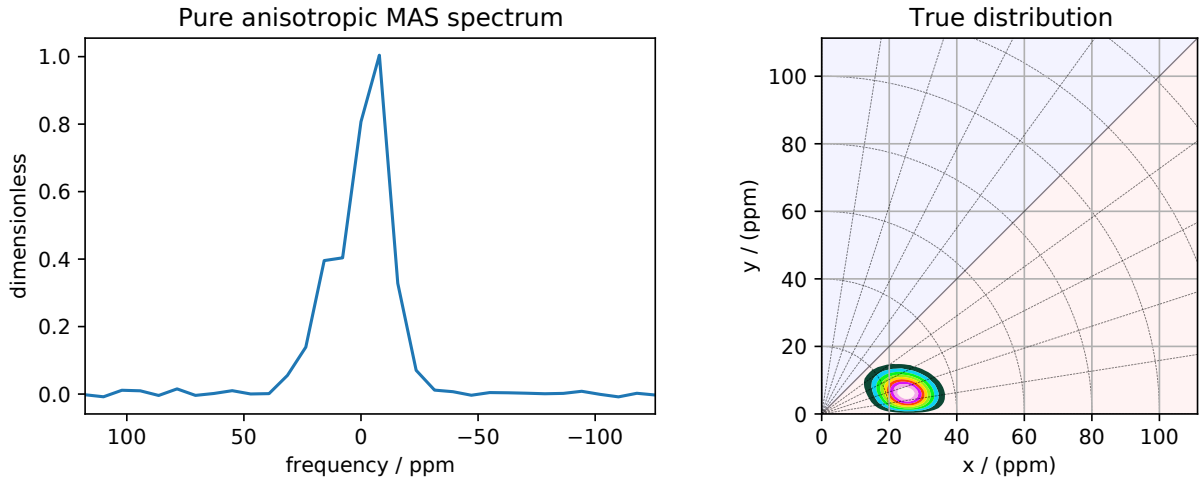


Figure 1.2: The figure on the left is the pure anisotropic MAS sideband amplitude spectrum corresponding to the nuclear shielding tensor distribution shown on the right.

1.5.2 Dimension Setup

For the inversion, we need to define (1) the coordinates associated with the pure anisotropic dimension, and (2) the two-dimensional x-y coordinates associated with the anisotropic tensor parameters, i.e., the inversion solution grid.

In `mrinversion`, the anisotropic spectrum dimension is initialized with a Dimension object from the csdmpy package. This object holds the frequency coordinates of the pure anisotropic spectrum. Because the example NMR dataset is imported as a CSDM object, the anisotropic spectrum dimension is already available as a CSDM Dimension object in the CSDM object and can be copied from there. Alternatively, we can create and initialize a anisotropic spectrum dimension using the csdmpy library as shown below:

```
>>> anisotropic_dimension = cp.LinearDimension(count=32, increment='625Hz', coordinates_
↪offset='-10kHz')
>>> print(anisotropic_dimension)
LinearDimension([-10000.  -9375.  -8750.  -8125.  -7500.  -6875.  -6250.  -5625.  -5000.
  -4375.  -3750.  -3125.  -2500.  -1875.  -1250.   -625.      0.    625.
   1250.   1875.   2500.   3125.   3750.   4375.   5000.   5625.   6250.
   6875.   7500.   8125.   8750.   9375.] Hz)
```

Here, the anisotropic dimension is sampled at 625 Hz for 32 points with an offset of -10 kHz.

Similarly, we can create the CSDM dimensions needed for the *x*-*y* inversion grid as shown below:

```
>>> inverse_dimension = [
...     cp.LinearDimension(count=25, increment='370 Hz', label='x'),   # the x-coordinates
...     cp.LinearDimension(count=25, increment='370 Hz', label='y')    # the y-coordinates
... ]
```

Both dimensions are sampled at every 370 Hz for 25 points. The inverse dimension at index 0 and 1 are the *x* and *y* dimensions, respectively.

1.5.3 Generating the kernel

Import the *ShieldingPALineshape* (page 17) class and generate the kernel as follows,

```
>>> from mrinversion.kernel.nmr import ShieldingPALineshape
>>> lineshapes = ShieldingPALineshape(
...     anisotropic_dimension=anisotropic_dimension,
...     inverse_dimension=inverse_dimension,
...     channel='29Si',
...     magnetic_flux_density='9.4 T',
...     rotor_angle='54.735°',
...     rotor_frequency='625 Hz',
...     number_of_sidebands=32
... )
```

In the above code, the variable `lineshapes` is an instance of the *ShieldingPALineshape* (page 17) class. The three required arguments of this class are the *anisotropic_dimension*, *inverse_dimension*, and *channel*. We have already defined the first two arguments in the previous subsection. The value of the channel attribute is the observed nucleus. The remaining optional arguments are the metadata that describes the environment under which the spectrum is acquired. In this example, these arguments describe a ^{29}Si pure anisotropic spinning-sideband spectrum acquired at 9.4 T magnetic flux density and spinning at the magic angle ($54.735°$) at 625 Hz. The value of the *rotor_frequency* argument is the effective anisotropic modulation frequency. For measurements like PASS[5], the anisotropic modulation frequency is the physical rotor frequency. For measurements like the extended chemical shift modulation sequences (XCS)[6], or its variants, where the effective anisotropic modulation frequency is lower than the physical rotor frequency, then it should be set accordingly.

The argument *number_of_sidebands* is the maximum number of sidebands that will be computed per line-shape within the kernel. For most two-dimensional isotropic vs. pure anisotropic spinning-sideband correlation spectra, the sampling along the sideband dimension is the rotor or the effective anisotropic modulation frequency. Therefore, the *number_of_sidebands* argument is usually the number of points along the sideband dimension. In this example, this value is 32.

Once the *ShieldingPALineshape* instance is created, use the *kernel()* (page 18) method of the instance to generate the spinning sideband kernel, as follows,

```
>>> K = lineshapes.kernel(supersampling=1)
>>> print(K.shape)
(32, 625)
```

Here, `K` is the 32×625 kernel, where the 32 is the number of samples (sideband amplitudes), and 625 is the number of features (subspectra) on the 25×25 *x*-*y* grid. The argument *supersampling* is the supersampling factor. In a supersampling scheme, each grid cell is averaged over a $n \times n$ sub-grid, where n is the supersampling factor. A supersampling factor of 1 is equivalent to no sub-grid averaging.

1.5.4 Data compression (optional)

Often when the kernel, K, is ill-conditioned, the solution becomes unstable in the presence of the measurement noise. An ill-conditioned kernel is the one whose singular values quickly decay to zero. In such cases, we employ the truncated singular value decomposition method to approximately represent the kernel K onto a smaller sub-space, called the *range space*, where the sub-space kernel is relatively well-defined. We refer to this sub-space kernel as the *compressed kernel*. Similarly, the measurement data on the sub-space is referred to as the *compressed signal*. The compression also reduces the time for further computation. To compress the kernel and the data, import the *TSVDCompression* (page 23) class and follow,

[5] Dixon, W. T., Spinning sideband free and spinning sideband only NMR spectra in spinning samples. J. Chem. Phys, 77, 1800, (1982). DOI:10.1063/1.444076

[6] Gullion, T., Extended chemical shift modulation. J. Mag. Res., 85, 3, (1989). DOI:10.1016/0022-2364(89)90253-9

```
>>> from mrinversion.linear_model import TSVDCompression
>>> new_system = TSVDCompression(K=K, s=data_object)
compression factor = 1.032258064516129
>>> compressed_K = new_system.compressed_K
>>> compressed_s = new_system.compressed_s
```

Here, the variable `new_system` is an instance of the *TSVDCompression* (page 23) class. If no truncation index is provided as the argument, when initializing the `TSVDCompression` class, an optimum truncation index is chosen using the maximum entropy method[7], which is the default behavior. The attributes *compressed_K* (page 23) and *compressed_s* (page 23) holds the compressed kernel and signal, respectively. The shape of the original signal *v.s.* the compressed signal is

```
>>> print(data_object.shape, compressed_s.shape)
(32,) (31,)
```

1.5.5 Setting up the inverse problem

When setting up the inversion, we solved the smooth LASSO[8] problem. Read the *Smooth-LASSO regularization* (page 6) section for further details.

Import the *SmoothLasso* (page 19) class and follow,

```
>>> from mrinversion.linear_model import SmoothLasso
>>> s_lasso = SmoothLasso(alpha=0.01, lambda1=1e-04, inverse_dimension=inverse_dimension)
```

Here, the variable `s_lasso` is an instance of the *SmoothLasso* (page 19) class. The required arguments of this class are *alpha* and *lambda1*, corresponding to the hyperparameters α and λ, respectively, in the Eq. (1.5). At the moment, we don't know the optimum value of the *alpha* and *lambda1* parameters. We start with a guess value.

To solve the smooth lasso problem, use the *fit()* (page 20) method of the `s_lasso` instance as follows,

```
>>> s_lasso.fit(K=compressed_K, s=compressed_s)
```

The two arguments of the *fit()* (page 20) method are the kernel, *K*, and the signal, *s*. In the above example, we set the value of *K* as `compressed_K`, and correspondingly the value of *s* as `compressed_s`. You may also use the uncompressed values of the kernel and signal in this method, if desired.

The solution to the smooth lasso is accessed using the *f* (page 20) attribute of the respective object.

```
>>> f_sol = s_lasso.f
```

The plot of the solution is

```
>>> _, ax = plt.subplots(1, 2, figsize=(9, 3.5), subplot_kw={'projection': 'csdm'})
>>> plot2D(ax[0], f_sol/f_sol.max(), title='Guess distribution')
>>> plot2D(ax[1], true_data_object, title='True distribution')
>>> plt.tight_layout()
>>> plt.show()
```

You may also evaluate the residuals corresponding to the solution using the *residuals()* (page 20) method of the object as follows,

[7] Varshavsky R., Gottlieb A., Linial M., Horn D., Novel unsupervised feature filtering of biological data. Bioinformatics, 22, e507e513, (2006). DOI:10.1093/bioinformatics/btl214.

[8] Hebiri M, Sara A. Van De Geer, The Smooth-Lasso and other l1+l2-penalized methods, arXiv, (2010). arXiv:1003.4885v2

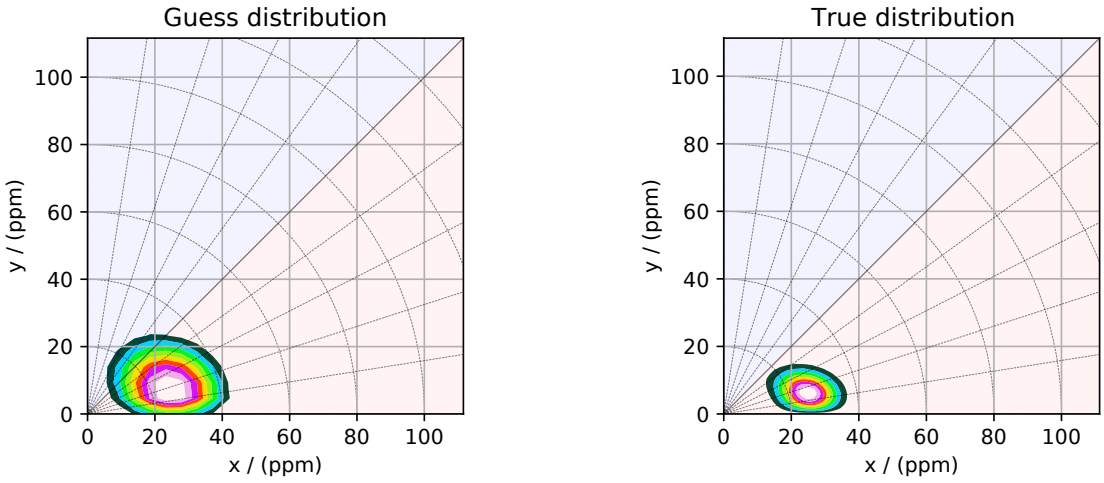


Figure 1.3: The figure on the left is the guess solution of the nuclear shielding tensor distribution derived from the inversion of the spinning sideband dataset. The figure on the right is the true nuclear shielding tensor distribution.

```
>>> residuals = s_lasso.residuals(K=K, s=data_object)
>>> # the plot of the residuals
>>> plt.figure(figsize=(5, 3.5))
>>> ax = plt.subplot(projection='csdm')
>>> ax.plot(residuals, color='black')
>>> plt.tight_layout()
>>> plt.show()
```

The argument of the *residuals* method is the kernel and the signal data. We provide the original kernel, K, and signal, s, because we desire the residuals corresponding to the original data and not the compressed data.

1.5.6 Statistical learning of tensor parameters

The solution from a linear model trained with the combined l1 and l2 priors, such as the smooth LASSO estimator used here, depends on the choice of the hyperparameters. The solution shown in the above figure is when $\alpha = 0.01$ and $\lambda = 1 \times 10^{-4}$. Although it's a solution, it is unlikely that this is the best solution. For this, we employ the statistical learning-based model, such as the *n*-fold cross-validation.

The *SmoothLassoCV* (page 21) class is designed to solve the smooth-lasso problem for a range of α and λ values and determine the best solution using the *n*-fold cross-validation. Here, we search the best model on a 10×10 pre-defined α-λ grid, using a 10-fold cross-validation statistical learning method. The λ and α values are sampled uniformly on a logarithmic scale as,

```
>>> lambdas = 10 ** (-4 - 2 * (np.arange(10) / 9))
>>> alphas = 10 ** (-3 - 2 * (np.arange(10) / 9))
```

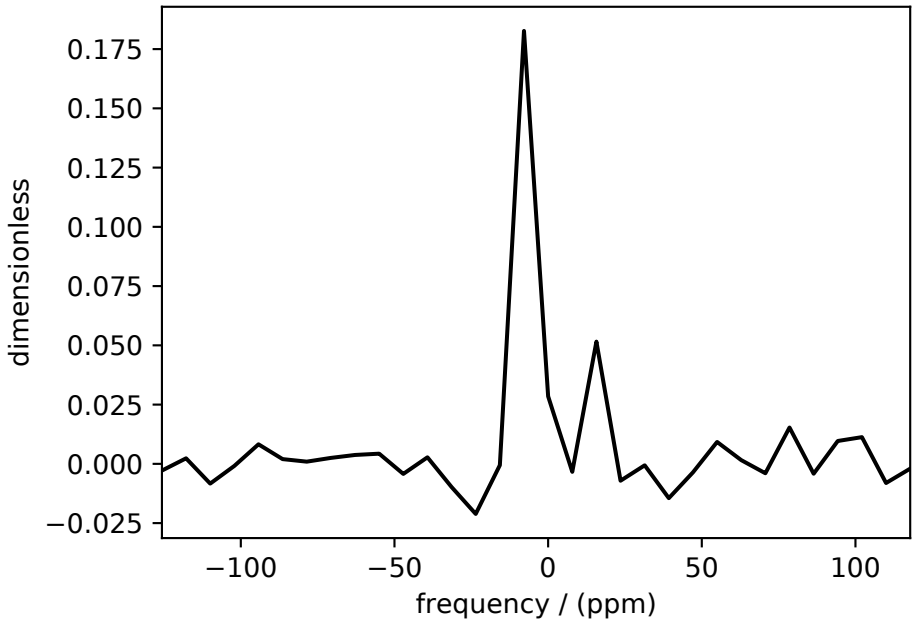


Figure 1.4: The residuals between the 1D MAS sideband spectrum and the predicted spectrum from the guess shielding tensor parameter distribution.

Smooth-LASSO CV Setup

Setup the smooth lasso cross-validation as follows

```
>>> from mrinversion.linear_model import SmoothLassoCV
>>> s_lasso_cv = SmoothLassoCV(
...     alphas=alphas,
...     lambdas=lambdas,
...     inverse_dimension=inverse_dimension,
...     sigma=0.005,
...     folds=10
... )
>>> s_lasso_cv.fit(K=compressed_K, s=compressed_s)
```

The arguments of the *SmoothLassoCV* (page 21) is a list of the *alpha* and *lambda* values, along with the standard deviation of the noise, *sigma*. The value of the argument *folds* is the number of folds used in the cross-validation. As before, to solve the problem, use the *fit()* (page 22) method, whose arguments are the kernel and signal.

The optimum hyperparameters

The optimized hyperparameters may be accessed using the *hyperparameters* (page 22) attribute of the class instance,

```
>>> alpha = s_lasso_cv.hyperparameters['alpha']
>>> lambda_1 = s_lasso_cv.hyperparameters['lambda']
```

The cross-validation surface

The cross-validation error metric is the mean square error metric. You may access this data using the *cross_validation_curve* (page 22) attribute.

```
>>> plt.figure(figsize=(5, 3.5))
>>> ax = plt.subplot(projection='csdm')
>>> ax.contour(np.log10(s_lasso_cv.cross_validation_curve), levels=25)
>>> ax.scatter(-np.log10(s_lasso_cv.hyperparameters['alpha']),
...            -np.log10(s_lasso_cv.hyperparameters['lambda']),
...            marker='x', color='k')
>>> plt.tight_layout()
>>> plt.show()
```

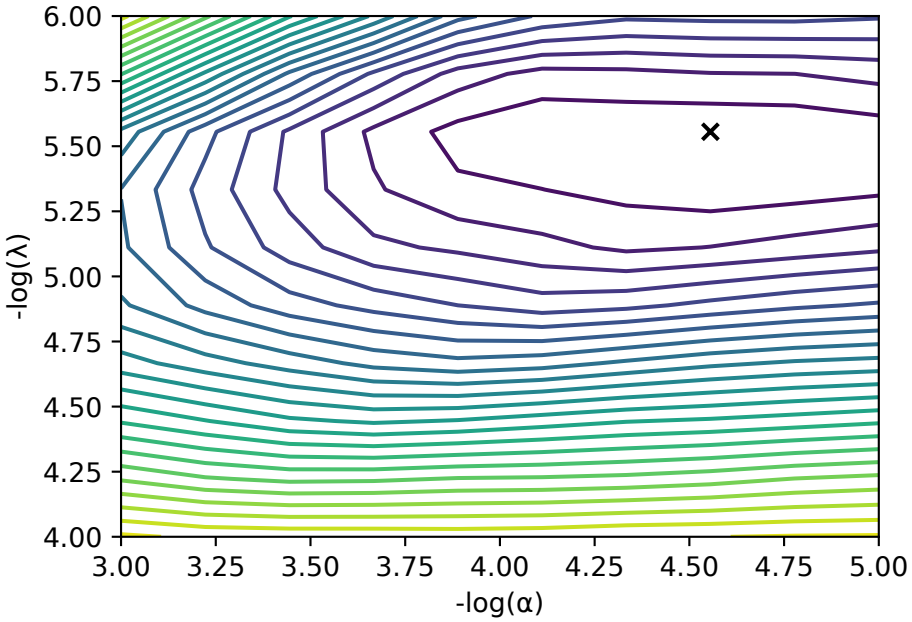


Figure 1.5: The ten-folds cross-validation prediction error surface as a function of the hyperparameters α and β.

The optimum solution

The best model selection from the cross-validation method may be accessed using the *f* (page 22) attribute.

```
>>> f_sol_cv = s_lasso_cv.f  # best model selected using the 10-fold cross-validation
```

The plot of the selected tensor parameter distribution is shown below.

```
>>> _, ax = plt.subplots(1, 2, figsize=(9, 3.5), subplot_kw={'projection': 'csdm'})
>>> plot2D(ax[0], f_sol_cv/f_sol_cv.max(), title='Optimum distribution')
>>> plot2D(ax[1], true_data_object, title='True distribution')
>>> plt.tight_layout()
>>> plt.show()
```

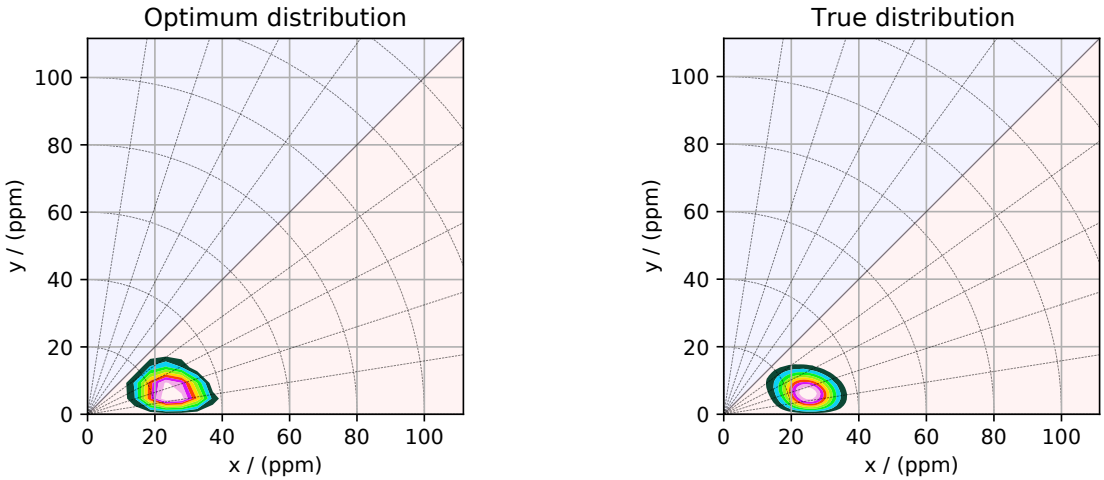


Figure 1.6: The figure on the left is the optimum solution selected by the 10-folds cross-validation method. The figure on the right is the true model of the nuclear shielding tensor distribution.

See also:

csdmpy, Quantity, numpy array, Matplotlib library

1.6 API-Reference

1.6.1 Pure anisotropic Nuclear Shielding Kernel

Generalized Class

class mrinversion.kernel.nmr.**ShieldingPALineshape**(*anisotropic_dimension*, *inverse_dimension*, *channel*, *magnetic_flux_density='9.4 T'*, *rotor_angle='54.735 deg'*, *rotor_frequency='14 kHz'*, *number_of_sidebands=1*)

Bases: mrinversion.kernel.base.LineShape

A generalized class for simulating the pure anisotropic NMR nuclear shielding line-shape kernel.

Parameters

- **anisotropic_dimension** – A Dimension object, or an equivalent dictionary object. This dimension must represent the pure anisotropic dimension.
- **inverse_dimension** – A list of two Dimension objects, or equivalent dictionary objects representing the *x*-*y* coordinate grid.
- **channel** – The channel is an isotope symbol of the nuclei given as the atomic number followed by the atomic symbol, for example, *1H*, *13C*, and *29Si*. This nucleus must correspond to the recorded frequency resonances.
- **magnetic_flux_density** – The magnetic flux density of the external static magnetic field. The default value is 9.4 T.
- **rotor_angle** – The angle of the sample holder (rotor) relative to the direction of the external magnetic field. The default value is 54.735 deg (magic angle).
- **rotor_frequency** – The effective sample spin rate. Depending on the NMR sequence, this value may be less than the physical sample rotation frequency. The default is 14 kHz.
- **number_of_sidebands** – The number of sidebands to simulate along the anisotropic dimension. The default value is 1.

kernel (*supersampling=1*)

Return the NMR nuclear shielding anisotropic line-shape kernel.

Parameters **supersampling** – An integer. Each cell is supersampled by the factor *supersampling* along every dimension.

Returns A numpy array containing the line-shape kernel.

Specialized Classes

Magic Angle Flipping

class mrinversion.kernel.nmr.**MAF**(*anisotropic_dimension*, *inverse_dimension*, *channel*, *magnetic_flux_density='9.4 T'*)

Bases: *mrinversion.kernel.csa_aniso.ShieldingPALineshape* (page 17)

A specialized class for simulating the pure anisotropic NMR nuclear shielding line-shape kernel resulting from the 2D MAF spectra.

Parameters

- **anisotropic_dimension** – A Dimension object, or an equivalent dictionary object. This dimension must represent the pure anisotropic dimension.
- **inverse_dimension** – A list of two Dimension objects, or equivalent dictionary objects representing the *x*-*y* coordinate grid.
- **channel** – The isotope symbol of the nuclei given as the atomic number followed by the atomic symbol, for example, *1H*, *13C*, and *29Si*. This nucleus must correspond to the recorded frequency resonances.
- **magnetic_flux_density** – The magnetic flux density of the external static magnetic field. The default value is 9.4 T.

Assumptions: The simulated line-shapes correspond to an infinite speed spectrum spinning at 90°.

kernel (*supersampling=1*)

Return the NMR nuclear shielding anisotropic line-shape kernel.

Parameters **supersampling** – An integer. Each cell is supersampled by the factor *supersampling* along every dimension.

Returns A numpy array containing the line-shape kernel.

Spinning Sidebands

class mrinversion.kernel.nmr.**SpinningSidebands**(*anisotropic_dimension*, *inverse_dimension*, *channel*, *magnetic_flux_density='9.4 T'*)

Bases: *mrinversion.kernel.csa_aniso.ShieldingPALineshape* (page 17)

A specialized class for simulating the pure anisotropic spinning sideband amplitudes of the nuclear shielding resonances resulting from a 2D sideband separation spectra.

Parameters

- **anisotropic_dimension** – A Dimension object, or an equivalent dictionary object. This dimension must represent the pure anisotropic dimension.
- **inverse_dimension** – A list of two Dimension objects, or equivalent dictionary objects representing the *x*-*y* coordinate grid.
- **channel** – The isotope symbol of the nuclei given as the atomic number followed by the atomic symbol, for example, *1H*, *13C*, and *29Si*. This nucleus must correspond to the recorded frequency resonances.
- **magnetic_flux_density** – The magnetic flux density of the external static magnetic field. The default value is 9.4 T.

Assumption: The simulated line-shapes correspond to a finite speed spectrum spinning at the magic angle, $54.735°$, where the spin rate is the increment along the anisotropic dimension.

kernel(*supersampling=1*)

Return the NMR nuclear shielding anisotropic line-shape kernel.

Parameters **supersampling** – An integer. Each cell is supersampled by the factor *supersampling* along every dimension.

Returns A numpy array containing the line-shape kernel.

1.6.2 Smooth Lasso

class mrinversion.linear_model.**SmoothLasso**(*alpha*, *lambda1*, *inverse_dimension*, *max_iterations=10000*, *tolerance=1e-05*, *positive=True*, *method='gradient_decent'*)

Bases: mrinversion.linear_model._base_l1l2.GeneralL2Lasso

The linear model trained with the combined l1 and l2 priors as the regularizer. The method minimizes the objective function,

$$\|\mathbf{Kf} - \mathbf{s}\|_2^2 + \alpha \sum_{i=1}^{d} \|\mathbf{J}_i\mathbf{f}\|_2^2 + \lambda\|\mathbf{f}\|_1, \tag{1.10}$$

where $\mathbf{K} \in \mathbb{R}^{m\times n}$ is the kernel, $\mathbf{s} \in \mathbb{R}^{m\times m_{\text{count}}}$ is the known (measured) signal, and $\mathbf{f} \in \mathbb{R}^{n\times m_{\text{count}}}$ is the desired solution. The parameters, α and λ, are the hyperparameters controlling the smoothness and sparsity of the solution $\mathbf{f}$. The matrix $\mathbf{J}_i$ is given as

$$\mathbf{J}_i = \mathbf{I}_{n_1} \otimes \cdots \otimes \mathbf{A}_{n_i} \otimes \cdots \otimes \mathbf{I}_{n_d}, \tag{1.11}$$

where $\mathbf{I}_{n_i} \in \mathbb{R}^{n_i \times n_i}$ is the identity matrix,

$$\mathbf{A}_{n_i} = \begin{pmatrix} 1 & -1 & 0 & \cdots & \vdots \\ 0 & 1 & -1 & \cdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & 0 \\ 0 & \cdots & 0 & 1 & -1 \end{pmatrix} \in \mathbb{R}^{(n_i-1)\times n_i}, \tag{1.12}$$

and the symbol $\otimes$ is the Kronecker product. The terms, $(n_1, n_2, \cdots, n_d)$, are the number of points along the respective dimensions, with the constraint that $\prod_{i=1}^{d} n_i = n$, where d is the total number of dimensions.

Parameters

- **alpha** (*float*) – The hyperparameter, α.
- **lambda1** (*float*) – The hyperparameter, λ.
- **inverse_dimension** (*list*) – A list of csdmpy Dimension objects representing the inverse space.
- **max_iterations** (*int*) – The maximum number of iterations allowed when solving the problem. The default value is 10000.
- **tolerance** (*float*) – The tolerance at which the solution is considered converged. The default value is 1e-5.
- **positive** (*bool*) – If True, the amplitudes in the solution, **f**, is contrained to only positive values, else the solution may contain positive and negative amplitudes. The default is True.

f

A ndarray of shape (*m_count*, *nd*, …, *n1*, *n0*) representing the solution, $\mathbf{f} \in \mathbb{R}^{m_{\text{count}} \times n_d \times \cdots n_1 \times n_0}$.

Type ndarray or CSDM object.

n_iter

The number of iterations required to reach the specified tolerance.

Type int

Methods Documentation

fit (*K*, *s*)

Fit the model using the coordinate descent method from scikit-learn.

Parameters

- **K** (*ndarray*) – The $m \times n$ kernel matrix, **K**. A numpy array of shape (m, n).
- **s** (*ndarray or CSDM object.*) – A csdm object or an equivalent numpy array holding the signal, **s**, as a $m \times m_{\text{count}}$ matrix.

predict (*K*)

Predict the signal using the linear model.

Parameters **K** (*ndarray*) – A $m \times n$ kernel matrix, **K**. A numpy array of shape (m, n).

Returns A numpy array of shape (m, m_count) with the predicted values

Return type ndarray

residuals(*K*, *s*)

Return the residual as the difference the data and the prediced data(fit), following

$$\text{residuals} = \mathbf{s} - \mathbf{K}\mathbf{f}^* \tag{1.13}$$

where $\mathbf{f}^*$ is the optimum solution.

Parameters

- **K** (*ndarray.*) – A $m \times n$ kernel matrix, $\mathbf{K}$. A numpy array of shape (m, n).
- **s** (*ndarray ot CSDM object.*) – A csdm object or a $m \times m_{\text{count}}$ signal matrix, $\mathbf{s}$.

Returns If *s* is a csdm object, returns a csdm object with the residuals. If *s* is a numpy array, return a $m \times m_{\text{count}}$ residue matrix. csdm object

Return type ndarray or CSDM object.

score(*K*, *s*, *sample_weights=None*)

The coefficient of determination, R^2, of the prediction. For more information, read scikit-learn documentation.

1.6.3 Smooth Lasso cross-validation

class mrinversion.linear_model.**SmoothLassoCV**(*alphas*, *lambdas*, *inverse_dimension*, *folds=10*, *max_iterations=10000*, *tolerance=1e-05*, *positive=True*, *sigma=0.0*, *randomize=False*, *times=2*, *verbose=False*, *n_jobs=- 1*, *method='gradient_decent'*)

Bases: mrinversion.linear_model._base_l1l2.GeneralL2LassoCV

The linear model trained with the combined l1 and l2 priors as the regularizer. The method minimizes the objective function,

$$\|\mathbf{K}\mathbf{f} - \mathbf{s}\|_2^2 + \alpha \sum_{i=1}^{d} \|\mathbf{J}_i\mathbf{f}\|_2^2 + \lambda\|\mathbf{f}\|_1, \tag{1.14}$$

where $\mathbf{K} \in \mathbb{R}^{m\times n}$ is the kernel, $\mathbf{s} \in \mathbb{R}^{m\times m_{\text{count}}}$ is the known signal containing noise, and $\mathbf{f} \in \mathbb{R}^{n\times m_{\text{count}}}$ is the desired solution. The parameters, α and λ, are the hyperparameters controlling the smoothness and sparsity of the solution $\mathbf{f}$. The matrix $\mathbf{J}_i$ is given as

$$\mathbf{J}_i = \mathbf{I}_{n_1} \otimes \cdots \otimes \mathbf{A}_{n_i} \otimes \cdots \otimes \mathbf{I}_{n_d}, \tag{1.15}$$

where $\mathbf{I}_{n_i} \in \mathbb{R}^{n_i\times n_i}$ is the identity matrix,

$$\mathbf{A}_{n_i} = \begin{pmatrix} 1 & -1 & 0 & \cdots & \vdots \\ 0 & 1 & -1 & \cdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & 0 \\ 0 & \cdots & 0 & 1 & -1 \end{pmatrix} \in \mathbb{R}^{(n_i-1)\times n_i}, \tag{1.16}$$

and the symbol $\otimes$ is the Kronecker product. The terms, $(n_1, n_2, \cdots, n_d)$, are the number of points along the respective dimensions, with the constraint that $\prod_{i=1}^{d} n_i = n$, where d is the total number of dimensions.

The cross-validation is carried out using a stratified splitting of the signal.

Parameters

- **alphas** (*ndarray*) – A list of α hyperparameters.

- **lambdas** (*ndarray*) – A list of λ hyperparameters.
- **inverse_dimension** (*list*) – A list of csdmpy Dimension objects representing the inverse space.
- **folds** (*int*) – The number of folds used in cross-validation.The default is 10.
- **max_iterations** (*int*) – The maximum number of iterations allowed when solving the problem. The default value is 10000.
- **tolerance** (*float*) – The tolerance at which the solution is considered converged. The default value is 1e-5.
- **positive** (*bool*) – If True, the amplitudes in the solution, **f**, is contrained to only positive values, else the solution may contain positive and negative amplitudes. The default is True.
- **sigma** (*float*) – The standard deviation of the noise in the signal. The default is 0.0.
- **sigma** – The standard deviation of the noise in the signal. The default is 0.0.
- **randomize** (*bool*) – If true, the folds are created by randomly assigning the samples to each fold. If false, a stratified sampled is used to generate folds. The default is False.
- **times** (*int*) – The number of times to randomized n-folds are created. Only applicable when *randomize* attribute is True.
- **verbose** (*bool*) – If true, prints the process.
- **n_jobs** (*int*) – The number of CPUs used for computation. The default is -1, that is, all available CPUs are used.

f
A ndarray of shape (m_count, nd, …, n1, n0). The solution, $\mathbf{f} \in \mathbb{R}^{m_{\text{count}} \times n_d \times \cdots n_1 \times n_0}$ or an equivalent CSDM object.

Type ndarray or CSDM object.

n_iter
The number of iterations required to reach the specified tolerance.

Type int.

hyperparameters
A dictionary with the α and :math:lambda‘ hyperparameters.

Type dict.

cross_validation_curve
The cross-validation error metric determined as the mean square error.

Type CSDM object.

Methods Documentation

fit(*K*, *s*)
Fit the model using the coordinate descent method from scikit-learn for all alpha anf lambda values using the *n*-folds cross-validation technique. The cross-validation metric is the mean squared error.

Parameters

- **K** – A $m \times n$ kernel matrix, **K**. A numpy array of shape (m, n).
- **s** – A $m \times m_{\text{count}}$ signal matrix, **s** as a csdm object or a numpy array or shape (m, m_count).

predict(*K*)
Predict the signal using the linear model.

Parameters **K** – A $m \times n$ kernel matrix, **K**. A numpy array of shape (m, n).

Returns A numpy array of shape (m, m_count) with the predicted values.

residuals(*K*, *s*)

Return the residual as the difference the data and the prediced data(fit), following

$$\text{residuals} = \mathbf{s} - \mathbf{K}\mathbf{f}^* \tag{1.17}$$

where $\mathbf{f}^*$ is the optimum solution.

Parameters

- **K** – A $m \times n$ kernel matrix, **K**. A numpy array of shape (m, n).
- **s** – A csdm object or a $m \times m_{\text{count}}$ signal matrix, **s**.

Returns If *s* is a csdm object, returns a csdm object with the residuals. If *s* is a numpy array, return a $m \times m_{\text{count}}$ residue matrix.

1.6.4 TSVDCompression

class mrinversion.linear_model.**TSVDCompression**(*K*, *s*, *r=None*)

Bases: object

SVD compression.

Parameters

- **K** – The kernel.
- **s** – The data.
- **r** – The number of singular values used in data compression.

truncation_index

The number of singular values retained.

Type int

compressed_K

The compressed kernel.

Type ndarray

compressed_s

The compressed data.

Type ndarray of CSDM object

1.6.5 Utils

mrinversion.kernel.utils.**x_y_to_zeta_eta**(*x*, *y*)

Convert the coordinates (x, y) to (ζ, η) using the following definition,

$$\left.\begin{aligned} \zeta &= \sqrt{x^2 + y^2}, \\ \eta &= \tfrac{4}{\pi} \tan^{-1} \left|\tfrac{x}{y}\right| \end{aligned}\right\} \quad |x| \leq |y|. \tag{1.18}$$

$$\left.\begin{aligned} \zeta &= -\sqrt{x^2 + y^2}, \\ \eta &= \tfrac{4}{\pi} \tan^{-1} \left|\tfrac{y}{x}\right| \end{aligned}\right\} \quad |x| > |y|. \tag{1.19}$$

Parameters

- **x** – floats or Quantity object. The coordinate x.
- **y** – floats or Quantity object. The coordinate y.

Returns A list of two ndarrays. The first array is the ζ coordinates. The second array is the η coordinates.

mrinversion.kernel.utils.**zeta_eta_to_x_y**(*zeta*, *eta*)

Convert the coordinates (ζ, η) to (x, y) using the following definition,

$$\left.\begin{aligned} x &= |\zeta| \sin\theta, \\ y &= |\zeta| \cos\theta \end{aligned}\right\} \quad \zeta \geq 0 \tag{1.20}$$

$$\left.\begin{aligned} x &= |\zeta| \cos\theta, \\ y &= |\zeta| \sin\theta \end{aligned}\right\} \quad \zeta < 0 \tag{1.21}$$

where $\theta = \frac{\pi}{4}\eta$.

Parameters

- **x** – ndarray or list of floats. The coordinate x.
- **y** – ndarray or list of floats. The coordinate y.

Returns A list of ndarrays. The first array holds the coordinate x. The second array holds the coordinates y.

mrinversion.utils.**get_polar_grids**(*ax*, *ticks=None*, *offset=0*)

Generate a piece-wise polar grid of Haeberlen parameters, zeta and eta.

Parameters

- **ax** – Matplotlib Axes.
- **ticks** – Tick coordinates where radial grids are drawn. The value can be a list or a numpy array. The default value is None.
- **offset** – The grid is drawn at an offset away from the origin.

mrinversion.utils.**to_Haeberlen_grid**(*csdm_object*, *zeta*, *eta*, *n=5*)

Convert the three-dimensional p(iso, x, y) to p(iso, zeta, eta) tensor distribution.

Parameters

- **csdm_object** (*CSDM*) – A CSDM object containing the 3D p(iso, x, y) distribution.
- **zeta** (*CSDM.Dimension*) – A CSDM dimension object describing the zeta dimension.
- **eta** (*CSDM.Dimension*) – A CSDM dimension object describing the eta dimension.
- **n** (*int*) – An interger used in linear interpolation of the data. The default is 5.

mrinversion.utils.**plot_3d**(*ax*, *csdm_object*, *elev=28*, *azim=-150*, *x_lim=None*, *y_lim=None*, *z_lim=None*, *max_2d=None*, *max_1d=None*, *cmap=<matplotlib.colors.LinearSegmentedColormap object>*, *box=False*, *clip_percent=0.0*, *linewidth=1*, *alpha=0.15*, ***kwargs*)

Generate a 3D density plot with 2D contour and 1D projections.

Parameters

- **ax** – Matplotlib Axes to render the plot.
- **csdm_object** – A 3D{1} CSDM object holding the data.
- **elev** – (optional) The 3D view angle, elevation angle in the z plane.
- **azim** – (optional) The 3D view angle, azimuth angle in the x-y plane.
- **x_lim** – (optional) The x limit given as a list, [x_min, x_max].

- **y_lim** – (optional) The y limit given as a list, [y_min, y_max].
- **z_lim** – (optional) The z limit given as a list, [z_min, z_max].
- **max_2d** – (Optional) The normalization factor of the 2D contour projections. The attribute is meaningful when multiple 3D datasets are viewed on the same plot. The value is given as a list, [*yz*, *xz*, *xy*], where *ij* is the maximum of the projection onto the *ij* plane, $i, j \in [x, y, z]$.
- **max_1d** – (Optional) The normalization factor of the 1D projections. The attribute is meaningful when multiple 3D datasets are viewed on the same plot. The value is given as a list, [*x*, *y*, *z*], where *i* is the maximum of the projection onto the *i* axis, $i \in [x, y, z]$.
- **cmap** – (Optional) The colormap used in rendering the volumetric plot. The same colormap is used for the 2D contour projections. For 1D plots, the first color in the colormap scheme is used for the line color.
- **box** – (Optional) If True, draw a box around the 3D data region.
- **clip_percent** – (Optional) The amplitudes of the dataset below the given percent is made transparent for the volumetric plot.
- **linewidth** – (Optional) The linewidth of the 2D countours, 1D plots and box.
- **alpha** – (Optional) The amount of alpha(transparency) applied in rendering the 3D volume.

CHAPTER

TWO

EXAMPLES

2.1 Example Gallery

The following are the examples of the statistical learning of nuclear shielding tensor parameters from pure anisotropic NMR spectrum.

2.1.1 One-dimensional synthetic datasets

This sub-section is for illustration only. For the practical application of the inversion method, refer to the next sub-section.

Unimodal distribution

The following example demonstrates the statistical learning based determination of the nuclear shielding tensor parameters from a one-dimensional cross-section of a spinning sideband correlation spectrum. In this example, we use a synthetic sideband amplitude spectrum from a unimodal tensor distribution.

Before getting started

Import all relevant packages.

```
import csdmpy as cp
import matplotlib.pyplot as plt
import numpy as np

from mrinversion.kernel.nmr import ShieldingPALineshape
from mrinversion.linear_model import SmoothLasso, SmoothLassoCV, TSVDCompression
from mrinversion.utils import get_polar_grids

# Setup for the matplotlib figures


# function for 2D x-y plot.
def plot2D(ax, csdm_object, title=""):
    # convert the dimension coordinates of the csdm_object from Hz to pmm.
    _ = [item.to("ppm", "nmr_frequency_ratio") for item in csdm_object.dimensions]

    levels = (np.arange(9) + 1) / 10
    plt.figure(figsize=(4.5, 3.5))
    ax.contourf(csdm_object, cmap="gist_ncar", levels=levels)
```

(continues on next page)

(continued from previous page)

```
    ax.grid(None)
    ax.set_title(title)
    get_polar_grids(ax)
    ax.set_aspect("equal")
```

Dataset setup

Import the dataset

Load the dataset. Here, we import the dataset as a CSDM data-object.

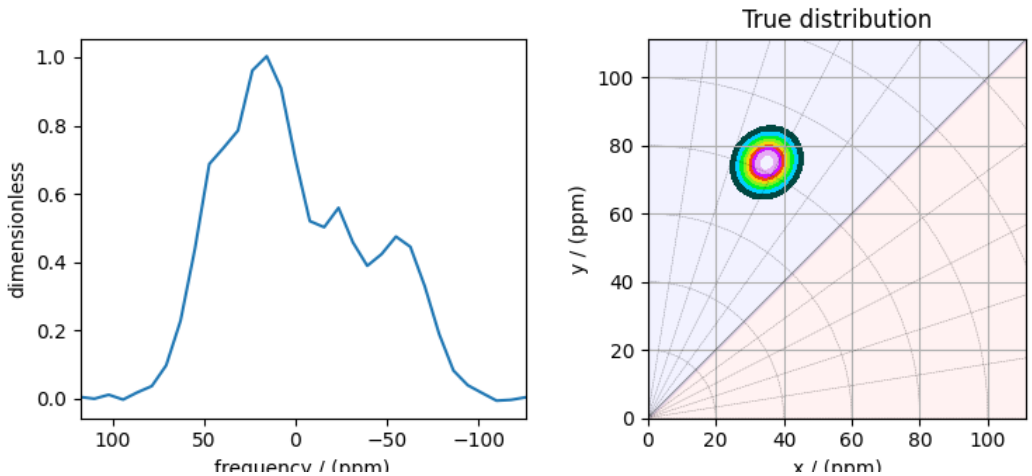
```
# the 1D spinning sideband cross-section data in csdm format
filename = "https://osu.box.com/shared/static/kehokr5op0amkfp5auyd498nblcdr1xy.csdf"
data_object = cp.load(filename)

# convert the data dimension from `Hz` to `ppm`.
data_object.dimensions[0].to("ppm", "nmr_frequency_ratio")
```

The variable `data_object` holds the 1D dataset. For comparison, let's also import the true tensor parameter distribution from which the synthetic 1D pure anisotropic spinning sideband cross-section amplitudes is simulated.

```
datafile = "https://osu.box.com/shared/static/s5wpm26w4cv3w64qjhouqu458ch4z0nd.csdf"
true_data_object = cp.load(datafile)
```

The plot of the 1D sideband cross-section along with the 2D true tensor parameter distribution of the synthetic dataset is shown below.

```
# the plot of the 1D MAF cross-section dataset.
_, ax = plt.subplots(1, 2, figsize=(9, 3.5), subplot_kw={"projection": "csdm"})
ax[0].plot(data_object)
ax[0].invert_xaxis()

# the plot of the true tensor distribution.
plot2D(ax[1], true_data_object, title="True distribution")
plt.tight_layout()
plt.show()
```



-

Linear Inversion setup

Dimension setup

Anisotropic-dimension: The dimension of the dataset that holds the pure anisotropic spinning sidebands.

```
anisotropic_dimension = data_object.dimensions[0]
```

x-y dimensions: The two inverse dimensions corresponding to the *x* and *y*-axis of the *x*-*y* grid.

```
inverse_dimension = [
    cp.LinearDimension(count=25, increment="370 Hz", label="x"),  # the `x`-dimension.
    cp.LinearDimension(count=25, increment="370 Hz", label="y"),  # the `y`-dimension.
]
```

Generating the kernel

For sideband datasets, the line-shape kernel corresponds to the pure anisotropic nuclear shielding spinning sideband spectra. Use the *ShieldingPALineshape* (page 17) class to generate the sideband kernel.

```
lineshape = ShieldingPALineshape(
    anisotropic_dimension=anisotropic_dimension,
    inverse_dimension=inverse_dimension,
    channel="29Si",
    magnetic_flux_density="9.4 T",
    rotor_angle="54.735 deg",
    rotor_frequency="625 Hz",
    number_of_sidebands=32,
)
```

Here, `lineshape` is an instance of the *ShieldingPALineshape* (page 17) class. The required arguments of this class are the *anisotropic_dimension*, *inverse_dimension*, and *channel*. We have already defined the first two arguments in the previous sub-section. The value of the *channel* argument is the observed nucleus. In this example, this value is '29Si'. The remaining arguments, such as the *magnetic_flux_density*, *rotor_angle*, and *rotor_frequency*, are set to match the conditions under which the spectrum was acquired. Note, the rotor frequency is the effective anisotropic modulation frequency, which may be less than the physical rotor frequency. The number of sidebands is usually the number of points along the sideband dimension.

Once the ShieldingPALineshape instance is created, use the *kernel()* (page 18) method of the instance to generate the sideband kernel.

```
K = lineshape.kernel(supersampling=1)
```

Data Compression

Data compression is optional but recommended. It may reduce the size of the inverse problem and, thus, further computation time.

```
new_system = TSVDCompression(K, data_object)
compressed_K = new_system.compressed_K
compressed_s = new_system.compressed_s

print(f"truncation_index = {new_system.truncation_index}")
```

Out:

```
compression factor = 1.032258064516129
truncation_index = 31
```

Solving the inverse problem

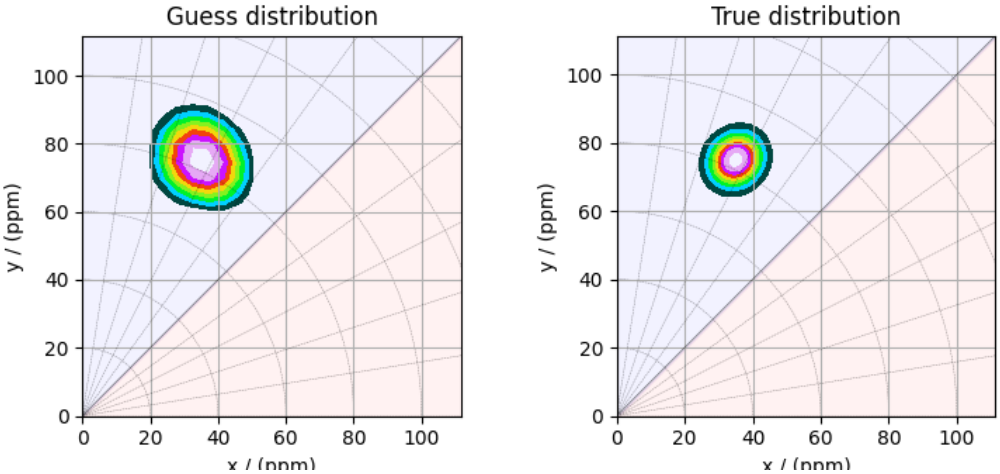
Smooth-LASSO problem

Solve the smooth-lasso problem. You may choose to skip this step and proceed to the statistical learning method. Usually, the statistical learning method is a time-consuming process that solves the smooth-lasso problem over a range of predefined hyperparameters. If you are unsure what range of hyperparameters to use, you can use this step for a quick look into the possible solution by giving a guess value for the α and λ hyperparameters, and then decide on the hyperparameters range accordingly.

```
# guess alpha and lambda values.
s_lasso = SmoothLasso(alpha=5e-5, lambda1=5e-6, inverse_dimension=inverse_dimension)
s_lasso.fit(K=compressed_K, s=compressed_s)
f_sol = s_lasso.f
```

Here, `f_sol` is the solution corresponding to hyperparameters $\alpha = 5 \times 10^{-5}$ and $\lambda = 5 \times 10^{-6}$. The plot of this solution is

```
_, ax = plt.subplots(1, 2, figsize=(9, 3.5), subplot_kw={"projection": "csdm"})

# the plot of the guess tensor distribution solution.
plot2D(ax[0], f_sol / f_sol.max(), title="Guess distribution")

# the plot of the true tensor distribution.
plot2D(ax[1], true_data_object, title="True distribution")
plt.tight_layout()
plt.show()
```

- 
Guess distribution
True distribution
y / (ppm)
x / (ppm)
0
20
40
60
80
100

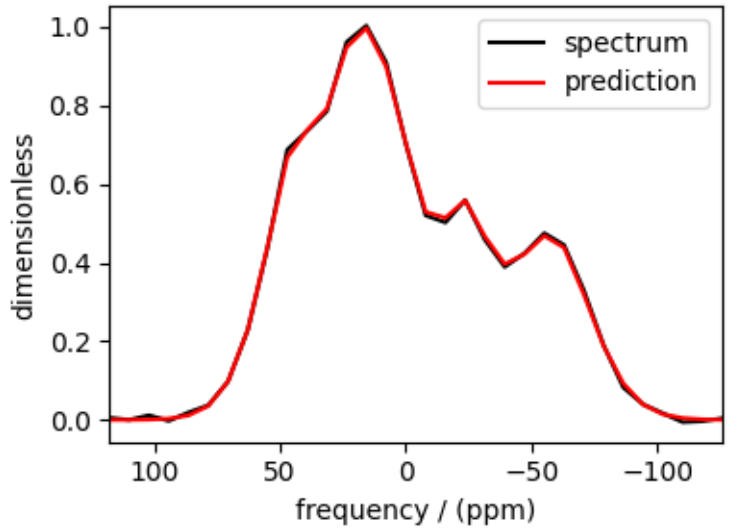
-

-

Predicted spectrum

You may also evaluate the predicted spectrum from the above solution following

```
residuals = s_lasso.residuals(K, data_object)
predicted_spectrum = data_object - residuals

plt.figure(figsize=(4, 3))
plt.subplot(projection="csdm")
plt.plot(data_object, color="black", label="spectrum")  # the original spectrum
plt.plot(predicted_spectrum, color="red", label="prediction")  # the predicted spectrum
plt.gca().invert_xaxis()
plt.legend()
plt.tight_layout()
plt.show()
```



As you can see from the predicted spectrum, our guess isn't far from the optimum hyperparameters. Let's create a search grid about the guess hyperparameters and run a cross-validation method for selection.

Statistical learning of the tensors

Smooth LASSO cross-validation

Create a guess range of values for the α and λ hyperparameters. The following code generates a range of λ and α values that are uniformly sampled on the log scale.

```
lambdas = 10 ** (-5 - 1 * (np.arange(6) / 5))
alphas = 10 ** (-4 - 2 * (np.arange(6) / 5))

# set up cross validation smooth lasso method
s_lasso_cv = SmoothLassoCV(
    alphas=alphas,
    lambdas=lambdas,
    inverse_dimension=inverse_dimension,
    sigma=0.005,
    folds=10,
)
# run the fit using the compressed kernel and compressed signal.
s_lasso_cv.fit(compressed_K, compressed_s)
```

The optimum hyper-parameters

Use the *hyperparameters* (page 22) attribute of the instance for the optimum hyper-parameters, α and λ, determined from the cross-validation.

```
print(s_lasso_cv.hyperparameters)
```

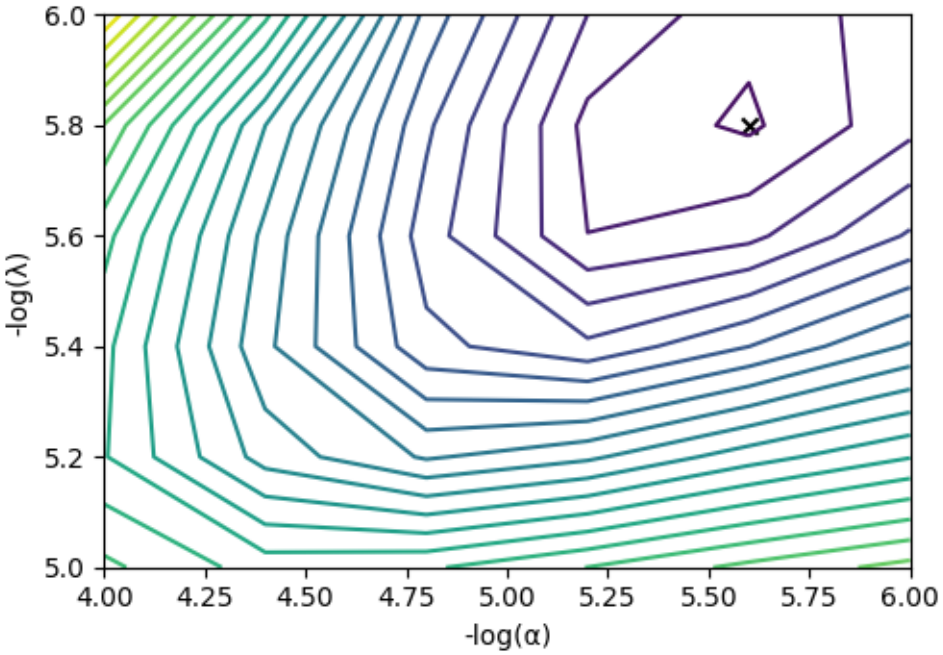
Out:

```
{'alpha': 2.5118864315095823e-06, 'lambda': 1.584893192461114e-06}
```

The cross-validation surface

Optionally, you may want to visualize the cross-validation error curve/surface. Use the *cross_validation_curve* (page 22) attribute of the instance, as follows. The cross-validation metric is the mean square error (MSE).

```
cv_curve = s_lasso_cv.cross_validation_curve

# plot of the cross-validation curve
plt.figure(figsize=(5, 3.5))
ax = plt.subplot(projection="csdm")
ax.contour(np.log10(s_lasso_cv.cross_validation_curve), levels=25)
ax.scatter(
    -np.log10(s_lasso_cv.hyperparameters["alpha"]),
    -np.log10(s_lasso_cv.hyperparameters["lambda"]),
    marker="x",
    color="k",
)
plt.tight_layout(pad=0.5)
plt.show()
```



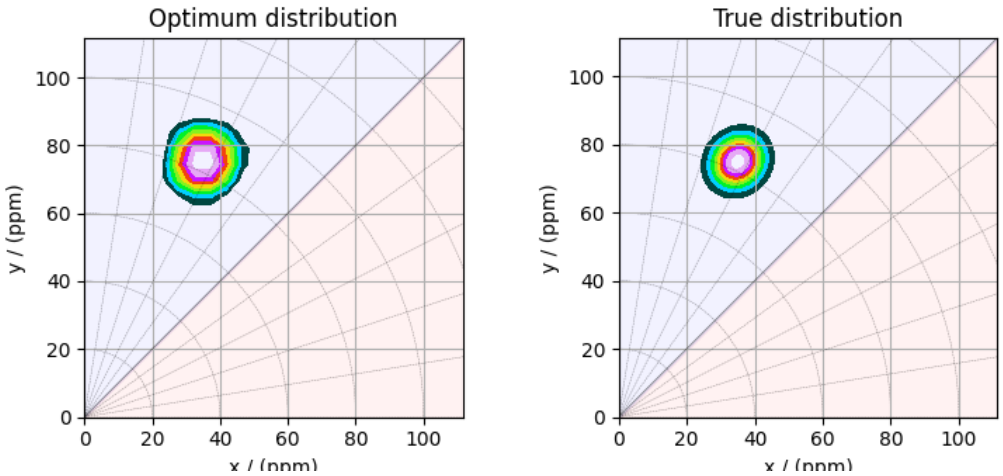
The optimum solution

The *f* (page 22) attribute of the instance holds the solution.

```
f_sol = s_lasso_cv.f
```

The corresponding plot of the solution, along with the true tensor distribution, is shown below.

```
_, ax = plt.subplots(1, 2, figsize=(9, 3.5), subplot_kw={"projection": "csdm"})

# the plot of the tensor distribution solution.
plot2D(ax[0], f_sol / f_sol.max(), title="Optimum distribution")

# the plot of the true tensor distribution.
plot2D(ax[1], true_data_object, title="True distribution")
plt.tight_layout()
plt.show()
```

- 

-

-

Total running time of the script: (0 minutes 35.753 seconds)

Bimodal distribution

The following example demonstrates the statistical learning based determination of nuclear shielding tensor parameters from a one-dimensional cross-section of a spinning sideband correlation spectrum. In this example, we use a synthetic sideband amplitude spectrum from a bimodal tensor distribution.

Before getting started

Import all relevant packages.

```
import csdmpy as cp
import matplotlib.pyplot as plt
import numpy as np

from mrinversion.kernel.nmr import ShieldingPALineshape
from mrinversion.linear_model import SmoothLasso, SmoothLassoCV, TSVDCompression
from mrinversion.utils import get_polar_grids

# Setup for the matplotlib figures


# function for 2D x-y plot.
def plot2D(ax, csdm_object, title=""):
    # convert the dimension coordinates of the csdm_object from Hz to pmm.
    _ = [item.to("ppm", "nmr_frequency_ratio") for item in csdm_object.dimensions]

    levels = (np.arange(9) + 1) / 10
    plt.figure(figsize=(4.5, 3.5))
    ax.contourf(csdm_object, cmap="gist_ncar", levels=levels)
    ax.grid(None)
```

(continues on next page)

(continued from previous page)

```
    ax.set_title(title)
    get_polar_grids(ax)
    ax.set_aspect("equal")
```

Dataset setup

Import the dataset

Load the dataset. Here, we import the dataset as a CSDM data-object.

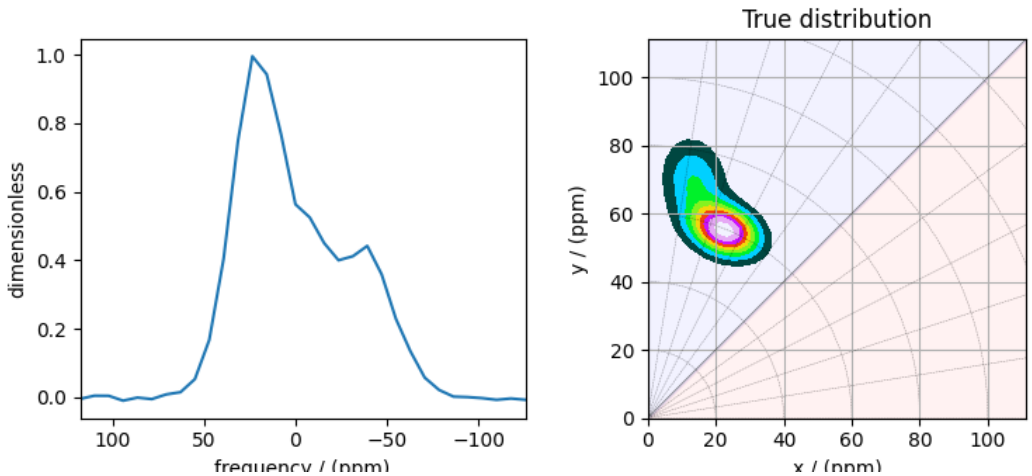
```
# the 1D spinning sideband cross-section data in csdm format
filename = "https://osu.box.com/shared/static/wjbhb6sif76mxfgndetew8mnrq6pw4pj.csdf"
data_object = cp.load(filename)

# convert the data dimension from `Hz` to `ppm`.
data_object.dimensions[0].to("ppm", "nmr_frequency_ratio")
```

The variable `data_object` holds the 1D dataset. For comparison, let's also import the true tensor parameter distribution from which the synthetic 1D pure anisotropic spinning sideband cross-section amplitudes is simulated.

```
datafile = "https://osu.box.com/shared/static/xesah85nd2gtm9yefazmladi697khuwi.csdf"
true_data_object = cp.load(datafile)
```

The plot of the 1D sideband cross-section along with the 2D true tensor parameter distribution of the synthetic dataset is shown below.

```
# the plot of the 1D MAF cross-section dataset.
_, ax = plt.subplots(1, 2, figsize=(9, 3.5), subplot_kw={"projection": "csdm"})
ax[0].plot(data_object)
ax[0].invert_xaxis()

# the plot of the true tensor distribution.
plot2D(ax[1], true_data_object, title="True distribution")
plt.tight_layout()
plt.show()
```



•

Linear Inversion setup

Dimension setup

Anisotropic-dimension: The dimension of the dataset that holds the pure anisotropic spinning sidebands.

```
anisotropic_dimension = data_object.dimensions[0]
```

x-y dimensions: The two inverse dimensions corresponding to the *x* and *y*-axis of the *x*-*y* grid.

```
inverse_dimension = [
    cp.LinearDimension(count=25, increment="370 Hz", label="x"),  # the `x`-dimension.
    cp.LinearDimension(count=25, increment="370 Hz", label="y"),  # the `y`-dimension.
]
```

Generating the kernel

For sideband datasets, the line-shape kernel corresponds to the pure anisotropic nuclear shielding spinning sideband spectra. Use the *ShieldingPALineshape* (page 17) class to generate the sideband kernel.

```
lineshape = ShieldingPALineshape(
    anisotropic_dimension=anisotropic_dimension,
    inverse_dimension=inverse_dimension,
    channel="29Si",
    magnetic_flux_density="9.4 T",
    rotor_angle="54.735 deg",
    rotor_frequency="625 Hz",
    number_of_sidebands=32,
)
K = lineshape.kernel(supersampling=1)
```

Data Compression

Data compression is optional but recommended. It may reduce the size of the inverse problem and, thus, further computation time.

```
new_system = TSVDCompression(K, data_object)
compressed_K = new_system.compressed_K
compressed_s = new_system.compressed_s

print(f"truncation_index = {new_system.truncation_index}")
```

Out:

```
compression factor = 1.032258064516129
truncation_index = 31
```

Solving the inverse problem

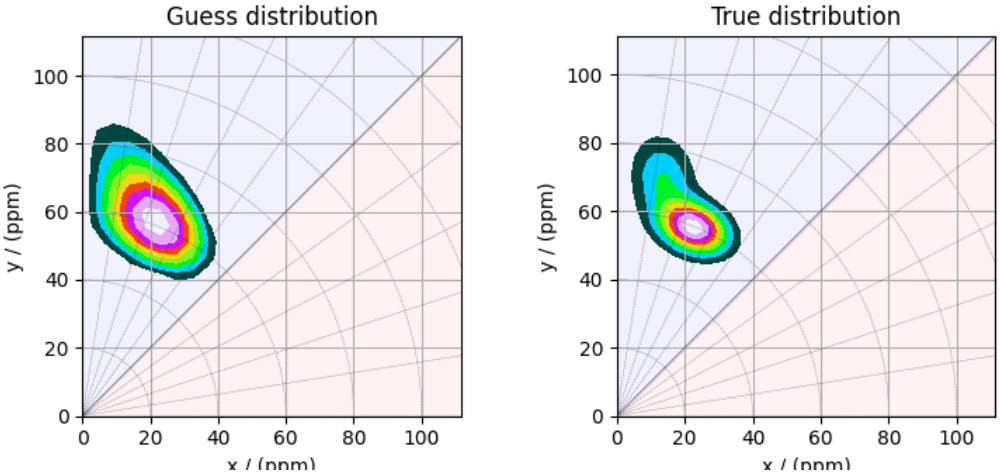
Smooth-LASSO problem

Solve the smooth-lasso problem. You may choose to skip this step and proceed to the statistical learning method. Usually, the statistical learning method is a time-consuming process that solves the smooth-lasso problem over a range of predefined hyperparameters. If you are unsure what range of hyperparameters to use, you can use this step for a quick look into the possible solution by giving a guess value for the α and λ hyperparameters, and then decide on the hyperparameters range accordingly.

```
# guess alpha and lambda values.
s_lasso = SmoothLasso(alpha=5e-5, lambda1=5e-6, inverse_dimension=inverse_dimension)
s_lasso.fit(K=compressed_K, s=compressed_s)
f_sol = s_lasso.f
```

Here, `f_sol` is the solution corresponding to hyperparameters $\alpha = 5 \times 10^{-5}$ and $\lambda = 5 \times 10^{-6}$. The plot of this solution is

```
_, ax = plt.subplots(1, 2, figsize=(9, 3.5), subplot_kw={"projection": "csdm"})

# the plot of the guess tensor distribution solution.
plot2D(ax[0], f_sol / f_sol.max(), title="Guess distribution")

# the plot of the true tensor distribution.
plot2D(ax[1], true_data_object, title="True distribution")
plt.tight_layout()
plt.show()
```

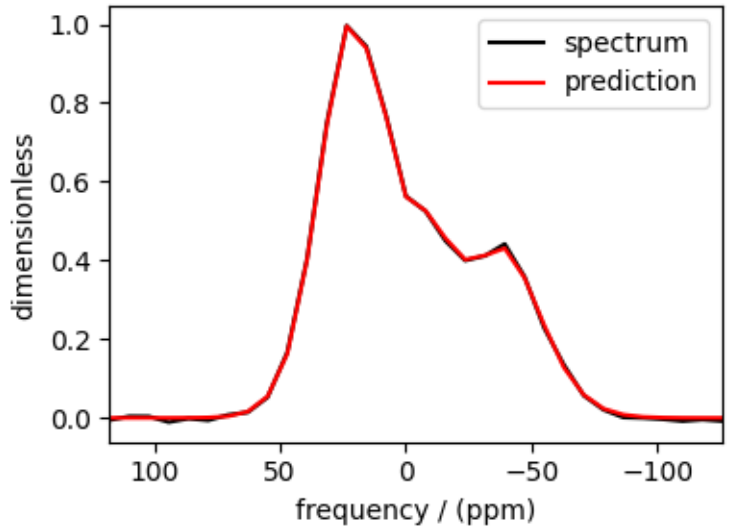
- 
Guess distribution
True distribution
y / (ppm)
x / (ppm)
0
20
40
60
80
100

-

Predicted spectrum

You may also evaluate the predicted spectrum from the above solution following

```
residuals = s_lasso.residuals(K, data_object)
predicted_spectrum = data_object - residuals

plt.figure(figsize=(4, 3))
plt.subplot(projection="csdm")
plt.plot(data_object, color="black", label="spectrum")  # the original spectrum
plt.plot(predicted_spectrum, color="red", label="prediction")  # the predicted spectrum
plt.gca().invert_xaxis()
plt.legend()
plt.tight_layout()
plt.show()
```



As you can see from the predicted spectrum, our guess isn't far from the optimum hyperparameters. Let's create a search grid about the guess hyperparameters and run a cross-validation method for selection.

Statistical learning of the tensors

Smooth LASSO cross-validation

Create a guess range of values for the α and λ hyperparameters. The following code generates a range of λ and α values that are uniformly sampled on the log scale.

```
lambdas = 10 ** (-5 - 1 * (np.arange(6) / 5))
alphas = 10 ** (-4 - 2 * (np.arange(6) / 5))

# set up cross validation smooth lasso method
s_lasso_cv = SmoothLassoCV(
    alphas=alphas,
    lambdas=lambdas,
    inverse_dimension=inverse_dimension,
    sigma=0.005,
    folds=10,
)
# run the fit using the compressed kernel and compressed signal.
s_lasso_cv.fit(compressed_K, compressed_s)
```

The optimum hyper-parameters

Use the *hyperparameters* (page 22) attribute of the instance for the optimum hyper-parameters, α and λ, determined from the cross-validation.

```
print(s_lasso_cv.hyperparameters)
```

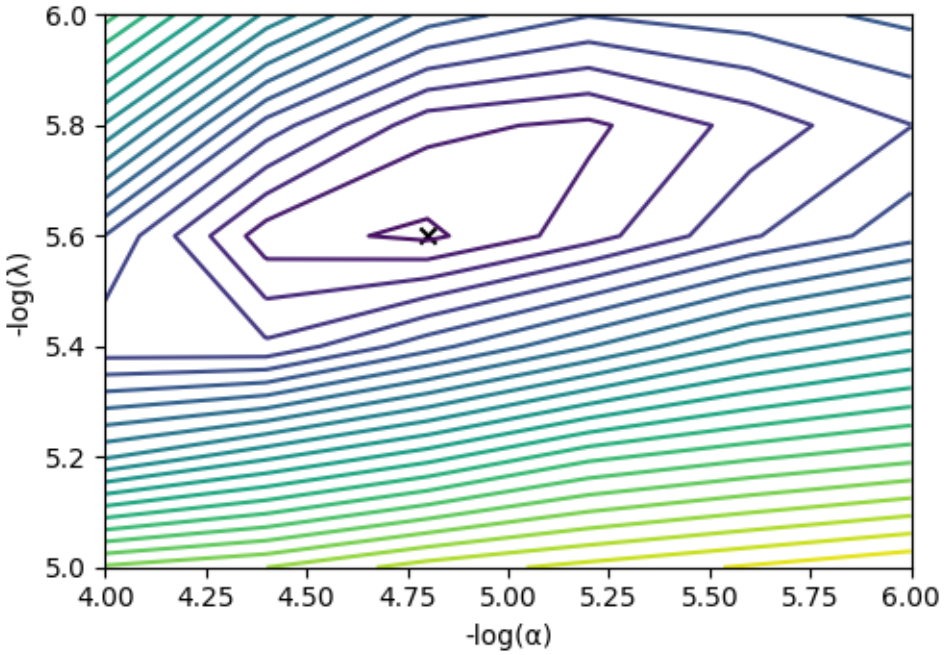
Out:

```
{'alpha': 1.584893192461114e-05, 'lambda': 2.5118864315095823e-06}
```

The cross-validation surface

Optionally, you may want to visualize the cross-validation error curve/surface. Use the *cross_validation_curve* (page 22) attribute of the instance, as follows. The cross-validation metric is the mean square error (MSE).

```
cv_curve = s_lasso_cv.cross_validation_curve

# plot of the cross-validation curve
plt.figure(figsize=(5, 3.5))
ax = plt.subplot(projection="csdm")
ax.contour(np.log10(s_lasso_cv.cross_validation_curve), levels=25)
ax.scatter(
    -np.log10(s_lasso_cv.hyperparameters["alpha"]),
    -np.log10(s_lasso_cv.hyperparameters["lambda"]),
    marker="x",
    color="k",
)
plt.tight_layout(pad=0.5)
plt.show()
```



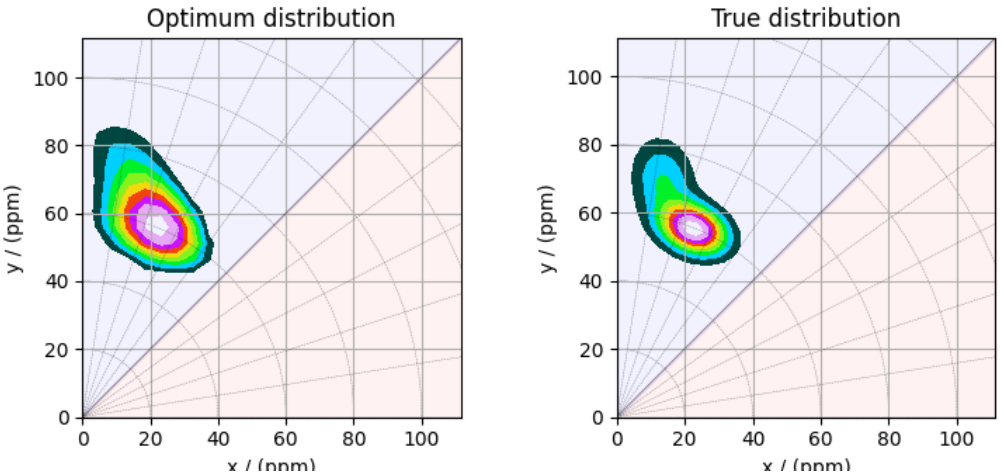
The optimum solution

The *f* (page 22) attribute of the instance holds the solution.

```
f_sol = s_lasso_cv.f
```

The corresponding plot of the solution, along with the true tensor distribution, is shown below.

```
_, ax = plt.subplots(1, 2, figsize=(9, 3.5), subplot_kw={"projection": "csdm"})

# the plot of the tensor distribution solution.
plot2D(ax[0], f_sol / f_sol.max(), title="Optimum distribution")

# the plot of the true tensor distribution.
plot2D(ax[1], true_data_object, title="True distribution")
plt.tight_layout()
plt.show()
```

- 

-

Total running time of the script: (0 minutes 31.414 seconds)

Unimodal distribution

The following example demonstrates the statistical learning based determination of the nuclear shielding tensor parameters from a one-dimensional cross-section of a magic-angle flipping (MAF) spectrum. In this example, we use a synthetic MAF lineshape from a unimodal tensor distribution.

Before getting started

Import all relevant packages.

```
import csdmpy as cp
import matplotlib.pyplot as plt
import numpy as np

from mrinversion.kernel.nmr import ShieldingPALineshape
from mrinversion.linear_model import SmoothLasso, SmoothLassoCV, TSVDCompression
from mrinversion.utils import get_polar_grids

# Setup for the matplotlib figures


# function for 2D x-y plot.
def plot2D(ax, csdm_object, title=""):
    # convert the dimension coordinates of the csdm_object from Hz to pmm.
    _ = [item.to("ppm", "nmr_frequency_ratio") for item in csdm_object.dimensions]

    levels = (np.arange(9) + 1) / 10
    plt.figure(figsize=(4.5, 3.5))
    ax.contourf(csdm_object, cmap="gist_ncar", levels=levels)
    ax.grid(None)
```

(continues on next page)

(continued from previous page)

```
    ax.set_title(title)
    get_polar_grids(ax)
    ax.set_aspect("equal")
```

Dataset setup

Import the dataset

Load the dataset. Here, we import the dataset as a CSDM data-object.

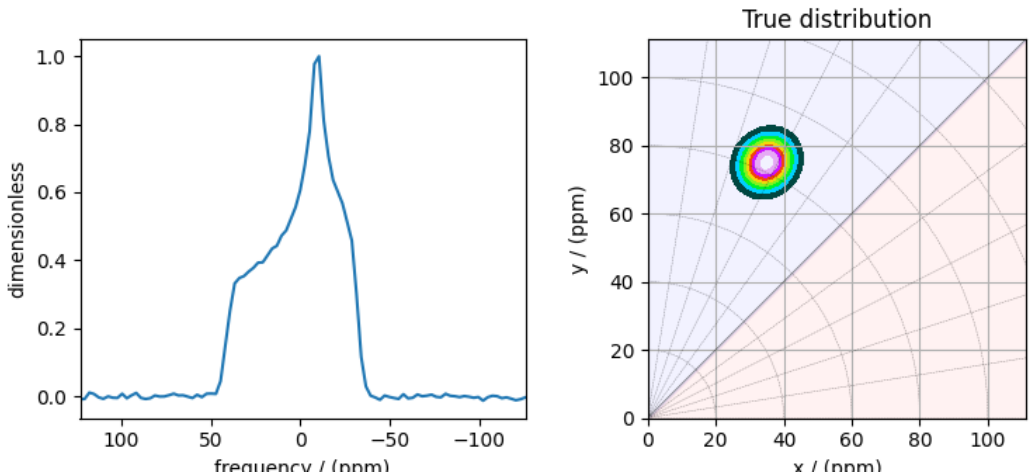
```
# the 1D MAF cross-section data in csdm format
filename = "https://osu.box.com/shared/static/puxfgdh25rru1q3li124anylkgup8rdp.csdf"
data_object = cp.load(filename)

# convert the data dimension from `Hz` to `ppm`.
data_object.dimensions[0].to("ppm", "nmr_frequency_ratio")
```

The variable `data_object` holds the 1D MAF cross-section. For comparison, let's also import the true tensor parameter distribution from which the synthetic 1D pure anisotropic MAF cross-section line-shape is simulated.

```
datafile = "https://osu.box.com/shared/static/s5wpm26w4cv3w64qjhouqu458ch4z0nd.csdf"
true_data_object = cp.load(datafile)
```

The plot of the 1D MAF cross-section along with the 2D true tensor parameter distribution of the synthetic dataset is shown below.

```
# the plot of the 1D MAF cross-section dataset.
_, ax = plt.subplots(1, 2, figsize=(9, 3.5), subplot_kw={"projection": "csdm"})
ax[0].plot(data_object)
ax[0].invert_xaxis()

# the plot of the true tensor distribution.
plot2D(ax[1], true_data_object, title="True distribution")
plt.tight_layout()
plt.show()
```



-

Linear Inversion setup

Dimension setup

Anisotropic-dimension: The dimension of the dataset that holds the pure anisotropic frequency contributions, which in this case, is the only dimension.

```
anisotropic_dimension = data_object.dimensions[0]
```

x-y dimensions: The two inverse dimensions corresponding to the *x* and *y*-axis of the *x*-*y* grid.

```
inverse_dimension = [
    cp.LinearDimension(count=25, increment="370 Hz", label="x"),  # the `x`-dimension.
    cp.LinearDimension(count=25, increment="370 Hz", label="y"),  # the `y`-dimension.
]
```

Generating the kernel

For MAF datasets, the line-shape kernel corresponds to the pure nuclear shielding anisotropy line-shapes. Use the *ShieldingPALineshape* (page 17) class to generate a shielding line-shape kernel.

```
lineshape = ShieldingPALineshape(
    anisotropic_dimension=anisotropic_dimension,
    inverse_dimension=inverse_dimension,
    channel="29Si",
    magnetic_flux_density="9.4 T",
    rotor_angle="90 deg",
    rotor_frequency="14 kHz",
    number_of_sidebands=4,
)
```

Here, `lineshape` is an instance of the *ShieldingPALineshape* (page 17) class. The required arguments of this class are the *anisotropic_dimension*, *inverse_dimension*, and *channel*. We have already defined the first two arguments in the previous sub-section. The value of the *channel* argument is the nucleus observed in the MAF experiment. In this example, this value is '29Si'. The remaining arguments, such as the *magnetic_flux_density*, *rotor_angle*, and *rotor_frequency*, are set to match the conditions under which the spectrum was acquired. The value of the *number_of_sidebands* argument is the number of sidebands calculated for each line-shape within the kernel.

Once the ShieldingPALineshape instance is created, use the *kernel()* (page 18) method of the instance to generate the MAF line-shape kernel.

```
K = lineshape.kernel(supersampling=1)
```

Data Compression

Data compression is optional but recommended. It may reduce the size of the inverse problem and, thus, further computation time.

```
new_system = TSVDCompression(K, data_object)
compressed_K = new_system.compressed_K
compressed_s = new_system.compressed_s

print(f"truncation_index = {new_system.truncation_index}")
```

Out:

```
compression factor = 1.5737704918032787
truncation_index = 61
```

Solving the inverse problem

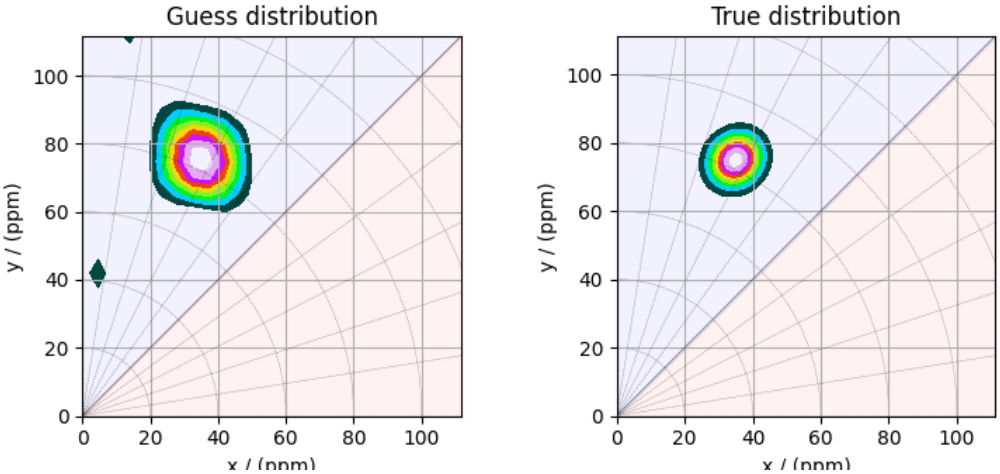
Smooth-LASSO problem

Solve the smooth-lasso problem. You may choose to skip this step and proceed to the statistical learning method. Usually, the statistical learning method is a time-consuming process that solves the smooth-lasso problem over a range of predefined hyperparameters. If you are unsure what range of hyperparameters to use, you can use this step for a quick look into the possible solution by giving a guess value for the α and λ hyperparameters, and then decide on the hyperparameters range accordingly.

```
# guess alpha and lambda values.
s_lasso = SmoothLasso(alpha=5e-5, lambda1=5e-6, inverse_dimension=inverse_dimension)
s_lasso.fit(K=compressed_K, s=compressed_s)
f_sol = s_lasso.f
```

Here, `f_sol` is the solution corresponding to hyperparameters $\alpha = 5 \times 10^{-5}$ and $\lambda = 5 \times 10^{-6}$. The plot of this solution is

```
_, ax = plt.subplots(1, 2, figsize=(9, 3.5), subplot_kw={"projection": "csdm"})

# the plot of the guess tensor distribution solution.
plot2D(ax[0], f_sol / f_sol.max(), title="Guess distribution")

# the plot of the true tensor distribution.
plot2D(ax[1], true_data_object, title="True distribution")
plt.tight_layout()
plt.show()
```

- 
Guess distribution
True distribution
y / (ppm)
x / (ppm)
0
20
40
60
80
100

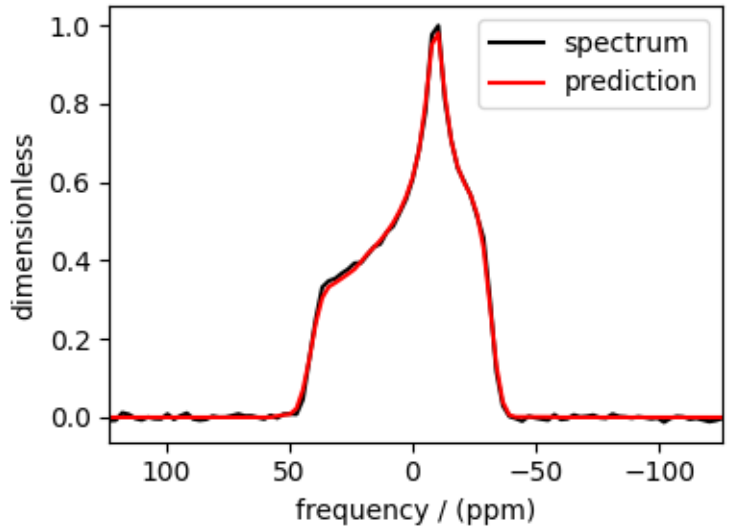
-

-

Predicted spectrum

You may also evaluate the predicted spectrum from the above solution following

```
residuals = s_lasso.residuals(K, data_object)
predicted_spectrum = data_object - residuals

plt.figure(figsize=(4, 3))
plt.subplot(projection="csdm")
plt.plot(data_object, color="black", label="spectrum")  # the original spectrum
plt.plot(predicted_spectrum, color="red", label="prediction")  # the predicted spectrum
plt.gca().invert_xaxis()
plt.legend()
plt.tight_layout()
plt.show()
```



As you can see from the predicted spectrum, our guess isn't far from the optimum hyperparameters. Let's create a search grid about the guess hyperparameters and run a cross-validation method for selection.

Statistical learning of the tensors

Smooth LASSO cross-validation

Create a guess range of values for the α and λ hyperparameters. The following code generates a range of λ and α values that are uniformly sampled on the log scale.

```
lambdas = 10 ** (-5.2 - 1 * (np.arange(6) / 5))
alphas = 10 ** (-4 - 2 * (np.arange(6) / 5))

# set up cross validation smooth lasso method
s_lasso_cv = SmoothLassoCV(
    alphas=alphas,
    lambdas=lambdas,
    inverse_dimension=inverse_dimension,
    sigma=0.005,
    folds=10,
)
# run the fit using the compressed kernel and compressed signal.
s_lasso_cv.fit(compressed_K, compressed_s)
```

The optimum hyper-parameters

Use the *hyperparameters* (page 22) attribute of the instance for the optimum hyper-parameters, α and λ, determined from the cross-validation.

```
print(s_lasso_cv.hyperparameters)
```

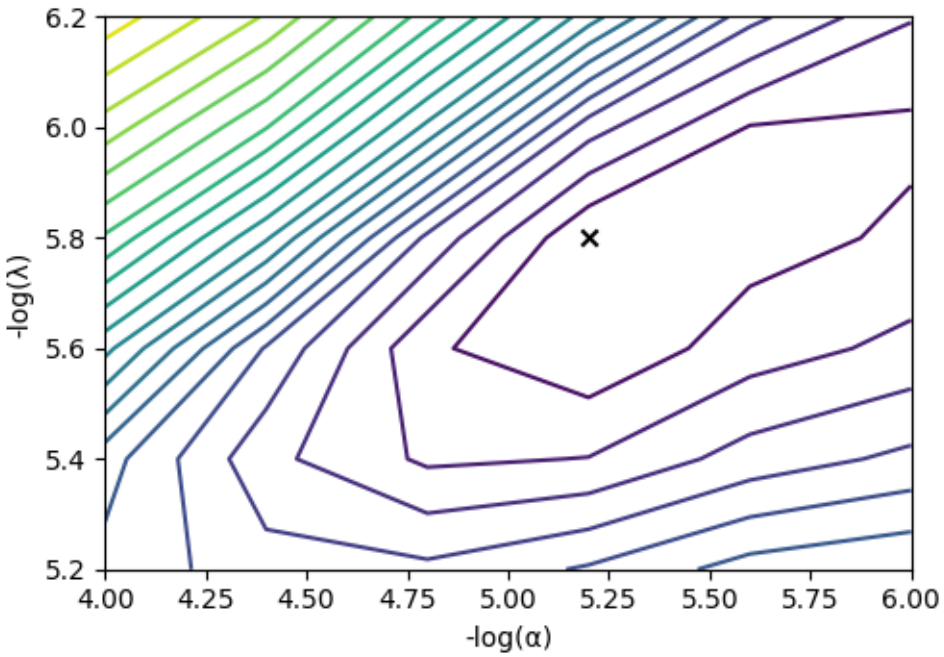
Out:

```
{'alpha': 6.30957344480193e-06, 'lambda': 1.584893192461114e-06}
```

The cross-validation surface

Optionally, you may want to visualize the cross-validation error curve/surface. Use the *cross_validation_curve* (page 22) attribute of the instance, as follows. The cross-validation metric is the mean square error (MSE).

```
cv_curve = s_lasso_cv.cross_validation_curve

# plot of the cross-validation curve
plt.figure(figsize=(5, 3.5))
ax = plt.subplot(projection="csdm")
ax.contour(np.log10(s_lasso_cv.cross_validation_curve), levels=25)
ax.scatter(
    -np.log10(s_lasso_cv.hyperparameters["alpha"]),
    -np.log10(s_lasso_cv.hyperparameters["lambda"]),
    marker="x",
    color="k",
)
plt.tight_layout(pad=0.5)
plt.show()
```



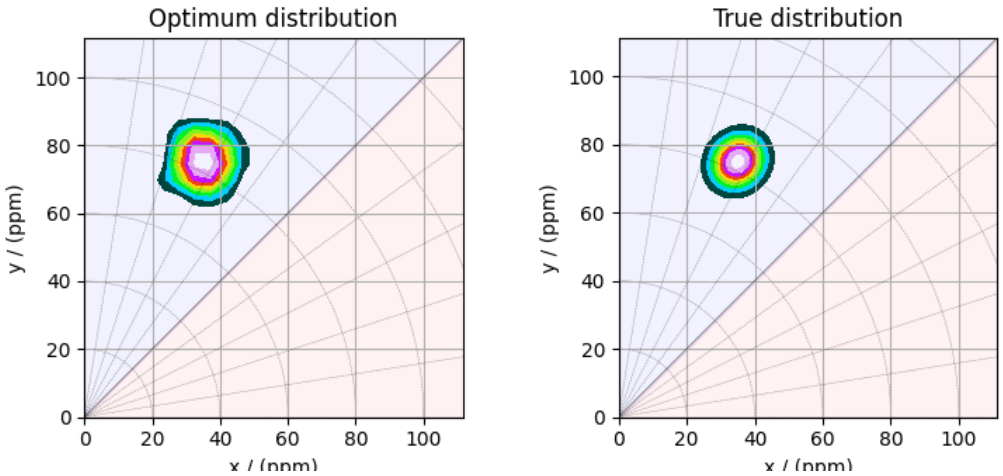
The optimum solution

The *f* (page 22) attribute of the instance holds the solution.

```
f_sol = s_lasso_cv.f
```

The corresponding plot of the solution, along with the true tensor distribution, is shown below.

```
_, ax = plt.subplots(1, 2, figsize=(9, 3.5), subplot_kw={"projection": "csdm"})

# the plot of the tensor distribution solution.
plot2D(ax[0], f_sol / f_sol.max(), title="Optimum distribution")

# the plot of the true tensor distribution.
plot2D(ax[1], true_data_object, title="True distribution")
plt.tight_layout()
plt.show()
```



-

Total running time of the script: (0 minutes 20.184 seconds)

Bimodal distribution

The following example demonstrates the statistical learning based determination of the nuclear shielding tensor parameters from a one-dimensional cross-section of a magic-angle flipping (MAF) spectrum. In this example, we use a synthetic MAF lineshape from a bimodal tensor distribution.

Before getting started

Import all relevant packages.

```
import csdmpy as cp
import matplotlib.pyplot as plt
import numpy as np

from mrinversion.kernel.nmr import ShieldingPALineshape
from mrinversion.linear_model import SmoothLasso, SmoothLassoCV, TSVDCompression
from mrinversion.utils import get_polar_grids

# Setup for the matplotlib figures


# function for 2D x-y plot.
def plot2D(ax, csdm_object, title=""):
    # convert the dimension coordinates of the csdm_object from Hz to pmm.
    _ = [item.to("ppm", "nmr_frequency_ratio") for item in csdm_object.dimensions]

    levels = (np.arange(9) + 1) / 10
    plt.figure(figsize=(4.5, 3.5))
    ax.contourf(csdm_object, cmap="gist_ncar", levels=levels)
    ax.grid(None)
```

(continues on next page)

(continued from previous page)

```
    ax.set_title(title)
    get_polar_grids(ax)
    ax.set_aspect("equal")
```

Dataset setup

Import the dataset

Load the dataset. Here, we import the dataset as a CSDM data-object.

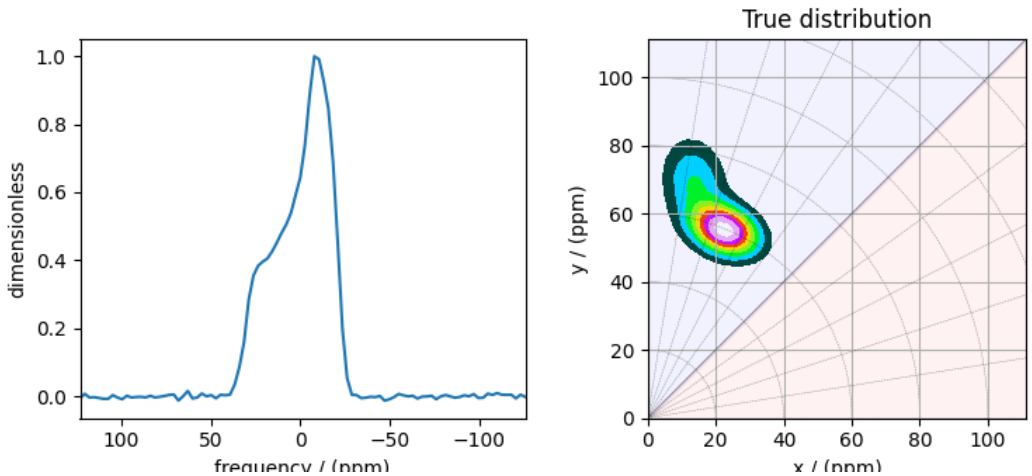
```
# the 1D MAF cross-section data in csdm format
filename = "https://osu.box.com/shared/static/6kcnou9iwqya30utlmzznnbv25iisxxj.csdf"
data_object = cp.load(filename)

# convert the data dimension from `Hz` to `ppm`.
data_object.dimensions[0].to("ppm", "nmr_frequency_ratio")
```

The variable `data_object` holds the 1D MAF cross-section. For comparison, let's also import the true tensor parameter distribution from which the synthetic 1D pure anisotropic MAF cross-section line-shape is simulated.

```
datafile = "https://osu.box.com/shared/static/xesah85nd2gtm9yefazmladi697khuwi.csdf"
true_data_object = cp.load(datafile)
```

The plot of the 1D MAF cross-section along with the 2D true tensor parameter distribution of the synthetic dataset is shown below.

```
# the plot of the 1D MAF cross-section dataset.
_, ax = plt.subplots(1, 2, figsize=(9, 3.5), subplot_kw={"projection": "csdm"})
ax[0].plot(data_object)
ax[0].invert_xaxis()

# the plot of the true tensor distribution.
plot2D(ax[1], true_data_object, title="True distribution")
plt.tight_layout()
plt.show()
```



-

Linear Inversion setup

Dimension setup

Anisotropic-dimension: The dimension of the dataset that holds the pure anisotropic frequency contributions, which in this case, is the only dimension.

```
anisotropic_dimension = data_object.dimensions[0]
```

x-y dimensions: The two inverse dimensions corresponding to the *x* and *y*-axis of the *x*-*y* grid.

```
inverse_dimension = [
    cp.LinearDimension(count=25, increment="370 Hz", label="x"),  # the `x`-dimension.
    cp.LinearDimension(count=25, increment="370 Hz", label="y"),  # the `y`-dimension.
]
```

Generating the kernel

For MAF datasets, the line-shape kernel corresponds to the pure nuclear shielding anisotropy line-shapes. Use the *ShieldingPALineshape* (page 17) class to generate a shielding line-shape kernel.

```
lineshape = ShieldingPALineshape(
    anisotropic_dimension=anisotropic_dimension,
    inverse_dimension=inverse_dimension,
    channel="29Si",
    magnetic_flux_density="9.4 T",
    rotor_angle="90 deg",
    rotor_frequency="14 kHz",
)
K = lineshape.kernel(supersampling=1)
```

Data Compression

Data compression is optional but recommended. It may reduce the size of the inverse problem and, thus, further computation time.

```
new_system = TSVDCompression(K, data_object)
compressed_K = new_system.compressed_K
compressed_s = new_system.compressed_s

print(f"truncation_index = {new_system.truncation_index}")
```

Out:

```
compression factor = 1.5737704918032787
truncation_index = 61
```

Solving the inverse problem

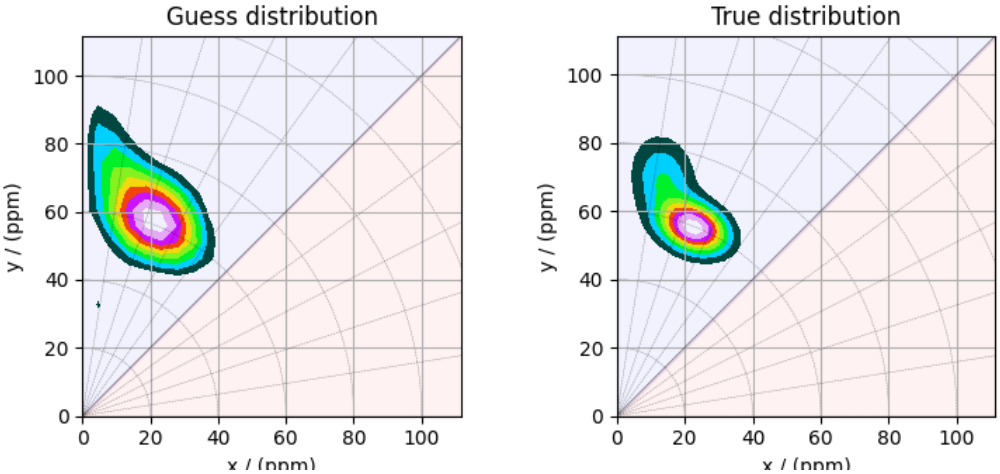
Smooth-LASSO problem

Solve the smooth-lasso problem. You may choose to skip this step and proceed to the statistical learning method. Usually, the statistical learning method is a time-consuming process that solves the smooth-lasso problem over a range of predefined hyperparameters. If you are unsure what range of hyperparameters to use, you can use this step for a quick look into the possible solution by giving a guess value for the α and λ hyperparameters, and then decide on the hyperparameters range accordingly.

```
# guess alpha and lambda values.
s_lasso = SmoothLasso(alpha=5e-5, lambda1=5e-6, inverse_dimension=inverse_dimension)
s_lasso.fit(K=compressed_K, s=compressed_s)
f_sol = s_lasso.f
```

Here, `f_sol` is the solution corresponding to hyperparameters $\alpha = 5 \times 10^{-5}$ and $\lambda = 5 \times 10^{-6}$. The plot of this solution is

```
_, ax = plt.subplots(1, 2, figsize=(9, 3.5), subplot_kw={"projection": "csdm"})

# the plot of the guess tensor distribution solution.
plot2D(ax[0], f_sol / f_sol.max(), title="Guess distribution")

# the plot of the true tensor distribution.
plot2D(ax[1], true_data_object, title="True distribution")
plt.tight_layout()
plt.show()
```

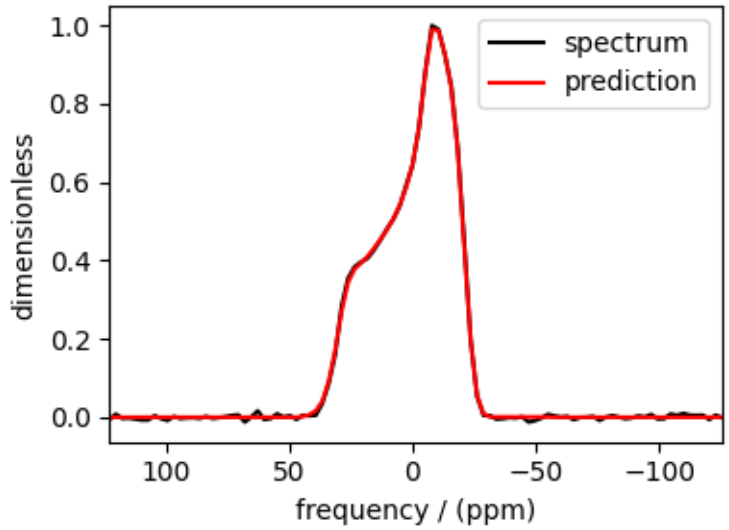
- 
Guess distribution
True distribution
y / (ppm)
x / (ppm)
0
20
40
60
80
100

-

Predicted spectrum

You may also evaluate the predicted spectrum from the above solution following

```
residuals = s_lasso.residuals(K, data_object)
predicted_spectrum = data_object - residuals

plt.figure(figsize=(4, 3))
plt.subplot(projection="csdm")
plt.plot(data_object, color="black", label="spectrum")  # the original spectrum
plt.plot(predicted_spectrum, color="red", label="prediction")  # the predicted spectrum
plt.gca().invert_xaxis()
plt.legend()
plt.tight_layout()
plt.show()
```



As you can see from the predicted spectrum, our guess isn't far from the optimum hyperparameters. Let's create a search grid about the guess hyperparameters and run a cross-validation method for selection.

Statistical learning of the tensors

Smooth LASSO cross-validation

Create a guess range of values for the α and λ hyperparameters. The following code generates a range of λ and α values that are uniformly sampled on the log scale.

```
lambdas = 10 ** (-5.5 - 1 * (np.arange(6) / 5))
alphas = 10 ** (-4 - 2 * (np.arange(6) / 5))

# set up cross validation smooth lasso method
s_lasso_cv = SmoothLassoCV(
    alphas=alphas,
    lambdas=lambdas,
    inverse_dimension=inverse_dimension,
    sigma=0.005,
    folds=10,
)
# run the fit using the compressed kernel and compressed signal.
s_lasso_cv.fit(compressed_K, compressed_s)
```

The optimum hyper-parameters

Use the *hyperparameters* (page 22) attribute of the instance for the optimum hyper-parameters, α and λ, determined from the cross-validation.

```
print(s_lasso_cv.hyperparameters)
```

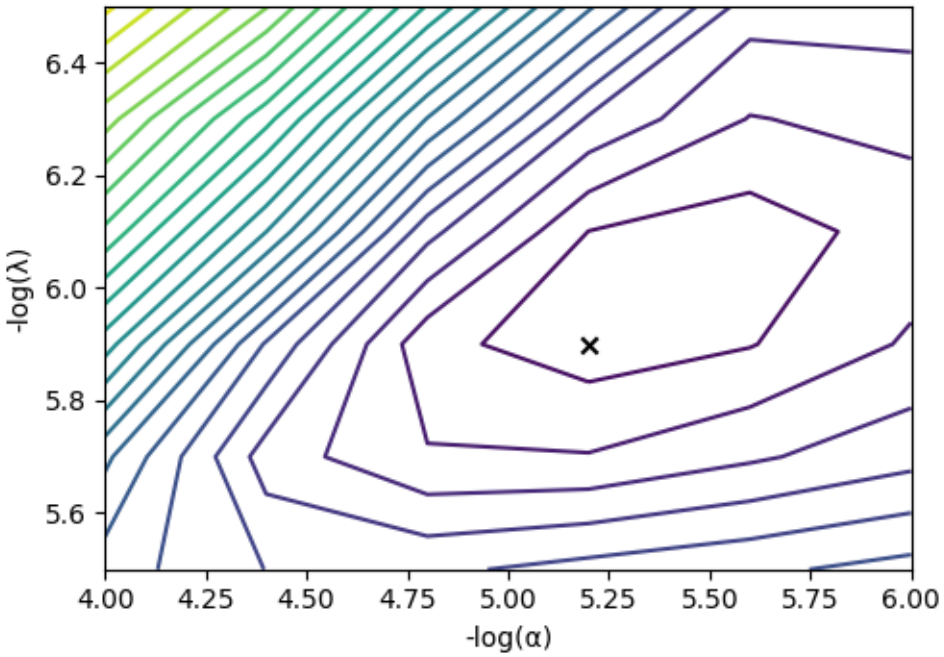
Out:

```
{'alpha': 6.30957344480193e-06, 'lambda': 1.2589254117941661e-06}
```

The cross-validation surface

Optionally, you may want to visualize the cross-validation error curve/surface. Use the *cross_validation_curve* (page 22) attribute of the instance, as follows. The cross-validation metric is the mean square error (MSE).

```
cv_curve = s_lasso_cv.cross_validation_curve

# plot of the cross-validation curve
plt.figure(figsize=(5, 3.5))
ax = plt.subplot(projection="csdm")
ax.contour(np.log10(s_lasso_cv.cross_validation_curve), levels=25)
ax.scatter(
    -np.log10(s_lasso_cv.hyperparameters["alpha"]),
    -np.log10(s_lasso_cv.hyperparameters["lambda"]),
    marker="x",
    color="k",
)
plt.tight_layout(pad=0.5)
plt.show()
```



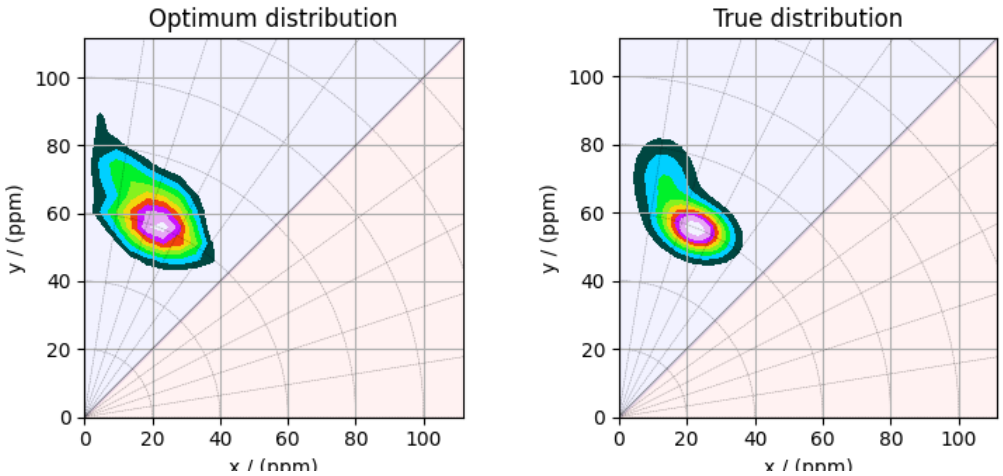
The optimum solution

The *f* (page 22) attribute of the instance holds the solution.

```
f_sol = s_lasso_cv.f
```

The corresponding plot of the solution, along with the true tensor distribution, is shown below.

```
_, ax = plt.subplots(1, 2, figsize=(9, 3.5), subplot_kw={"projection": "csdm"})

# the plot of the tensor distribution solution.
plot2D(ax[0], f_sol / f_sol.max(), title="Optimum distribution")

# the plot of the true tensor distribution.
plot2D(ax[1], true_data_object, title="True distribution")
plt.tight_layout()
plt.show()
```



-

Total running time of the script: (0 minutes 22.935 seconds)

2.1.2 Spinning sideband spectrum (Experiment)

The following are the examples of the statistical learning of nuclear shielding tensor from pure anisotropic spinning sideband spectrum.

2D MAT data of KMg0.5O 4.SiO2 glass

The following example illustrates an application of the statistical learning method applied in determining the distribution of the nuclear shielding tensor parameters from a 2D magic-angle turning (MAT) spectrum. In this example, we use the 2D MAT spectrum[1] of $KMg_{0.5}O \cdot 4SiO_2$ glass.

Setup for the matplotlib figure.

```
import csdmpy as cp
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import cm
from csdmpy import statistics as stats

from mrinversion.kernel.nmr import ShieldingPALineshape
from mrinversion.kernel.utils import x_y_to_zeta_eta
from mrinversion.linear_model import SmoothLassoCV, TSVDCompression
from mrinversion.utils import plot_3d, to_Haeberlen_grid
```

Setup for the matplotlib figures.

[1] Walder, B. J., Dey, K. K., Kaseman, D. C., Baltisberger, J. H., Grandinetti, P. J. Sideband separation experiments in NMR with phase incremented echo train acquisition, J. Chem. Phys. 138, 4803142, (2013). doi:10.1063/1.4803142.

```
# function for plotting 2D dataset
def plot2D(csdm_object, **kwargs):
    plt.figure(figsize=(4.5, 3.5))
    ax = plt.subplot(projection="csdm")
    ax.imshow(csdm_object, cmap="gist_ncar_r", aspect="auto", **kwargs)
    ax.invert_xaxis()
    ax.invert_yaxis()
    plt.tight_layout()
    plt.show()
```

Dataset setup

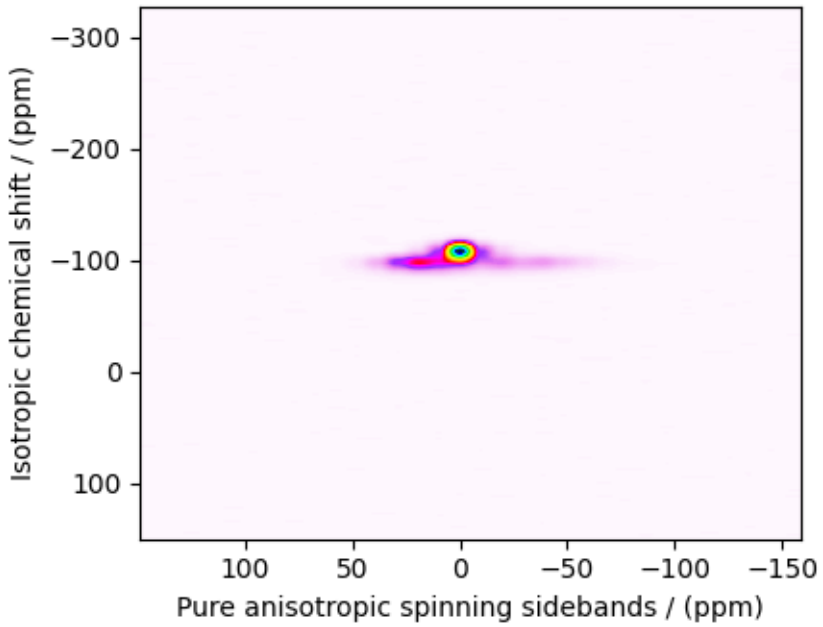
Import the dataset

Load the dataset. Here, we import the dataset as the CSDM data-object.

```
# The 2D MAT dataset in csdm format
filename = "https://zenodo.org/record/3964531/files/KMg0_5-4SiO2-MAT.csdf"
data_object = cp.load(filename)

# For inversion, we only interest ourselves with the real part of the complex dataset.
data_object = data_object.real

# We will also convert the coordinates of both dimensions from Hz to ppm.
_ = [item.to("ppm", "nmr_frequency_ratio") for item in data_object.dimensions]
```

Here, the variable `data_object` is a CSDM object that holds the real part of the 2D MAT dataset. The plot of the MAT dataset is

```
plot2D(data_object)
```



There are two dimensions in this dataset. The dimension at index 0 is the pure anisotropic spinning sideband dimension, while the dimension at index 1 is the isotropic chemical shift dimension.

Prepping the data for inversion

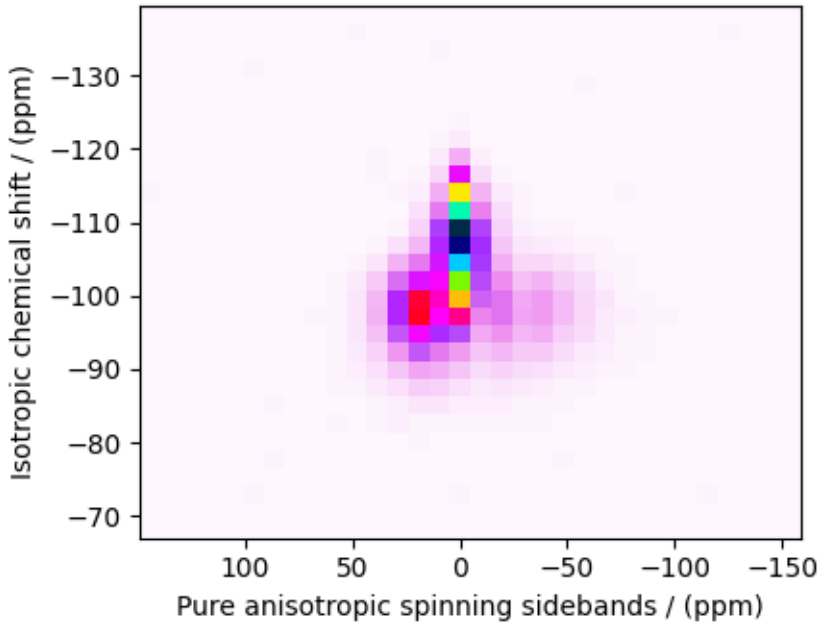
Step-1: Data Alignment

When using the csdm objects with the `mrinversion` package, the dimension at index 0 must be the dimension undergoing the linear inversion. In this example, we plan to invert the pure anisotropic shielding line-shape. In the `data_object`, the anisotropic dimension is already at index 0 and, therefore, no further action is required.

Step-2: Optimization

Also notice, the signal from the 2D MAF dataset occupies a small fraction of the two-dimensional frequency grid. For optimum performance, truncate the dataset to the relevant region before proceeding. Use the appropriate array indexing/slicing to select the signal region.

```
data_object_truncated = data_object[:, 75:105]
plot2D(data_object_truncated)
```



Linear Inversion setup

Dimension setup

Anisotropic-dimension: The dimension of the dataset that holds the pure anisotropic frequency contributions. In `mrinversion`, this must always be the dimension at index 0 of the data object.

```
anisotropic_dimension = data_object_truncated.dimensions[0]
```

x-y dimensions: The two inverse dimensions corresponding to the *x* and *y*-axis of the *x*-*y* grid.

```
inverse_dimensions = [
    cp.LinearDimension(count=25, increment="370 Hz", label="x"),  # the `x`-dimension.
    cp.LinearDimension(count=25, increment="370 Hz", label="y"),  # the `y`-dimension.
]
```

Generating the kernel

For MAF/PASS datasets, the kernel corresponds to the pure nuclear shielding anisotropy sideband spectra. Use the *ShieldingPALineshape* (page 17) class to generate a shielding spinning sidebands kernel.

```
sidebands = ShieldingPALineshape(
    anisotropic_dimension=anisotropic_dimension,
    inverse_dimension=inverse_dimensions,
    channel="29Si",
    magnetic_flux_density="9.4 T",
    rotor_angle="54.735°",
    rotor_frequency="790 Hz",
    number_of_sidebands=anisotropic_dimension.count,
)
```

Here, `sidebands` is an instance of the *ShieldingPALineshape* (page 17) class. The required arguments of this class are the *anisotropic_dimension*, *inverse_dimension*, and *channel*. We have already defined the first two arguments in the previous sub-section. The value of the *channel* argument is the nucleus observed in the MAT/PASS experiment. In this example, this value is '29Si'. The remaining arguments, such as the *magnetic_flux_density*, *rotor_angle*, and *rotor_frequency*, are set to match the conditions under which the 2D MAT/PASS spectrum was acquired, which in this case corresponds to acquisition at the magic-angle and spinning at a rotor frequency of 790 Hz in a 9.4 T magnetic flux density.

The value of the *rotor_frequency* argument is the effective anisotropic modulation frequency. In a MAT measurement, the anisotropic modulation frequency is the same as the physical rotor frequency. For other measurements like the extended chemical shift modulation sequences (XCS)[2], or its variants, the effective anisotropic modulation frequency is lower than the physical rotor frequency and should be set appropriately.

The argument *number_of_sidebands* is the maximum number of computed sidebands in the kernel. For most two-dimensional isotropic v.s pure anisotropic spinning-sideband correlation measurements, the sampling along the sideband dimension is the rotor or the effective anisotropic modulation frequency. Therefore, the value of the *number_of_sidebands* argument is usually the number of points along the sideband dimension. In this example, this value is 32.

Once the ShieldingPALineshape instance is created, use the kernel() method to generate the spinning sideband lineshape kernel.

```
K = sidebands.kernel(supersampling=2)
print(K.shape)
```

Out:

```
(32, 625)
```

The kernel `K` is a NumPy array of shape (32, 625), where the axes with 32 and 625 points are the spinning sidebands dimension and the features (x-y coordinates) corresponding to the 25×25 *x*-*y* grid, respectively.

[2] Gullion, T., Extended chemical-shift modulation, J. Mag. Res., 85, 3, (1989). 10.1016/0022-2364(89)90253-9

Data Compression

Data compression is optional but recommended. It may reduce the size of the inverse problem and, thus, further computation time.

```
new_system = TSVDCompression(K, data_object_truncated)
compressed_K = new_system.compressed_K
compressed_s = new_system.compressed_s

print(f"truncation_index = {new_system.truncation_index}")
```

Out:

```
compression factor = 1.032258064516129
truncation_index = 31
```

Solving the inverse problem

Smooth LASSO cross-validation

Solve the smooth-lasso problem. Use the statistical learning `SmoothLassoCV` method to solve the inverse problem over a range of α and λ values and determine the best nuclear shielding tensor parameter distribution for the given 2D MAT dataset. Considering the limited build time for the documentation, we'll perform the cross-validation over a smaller 5×5 x-y grid. You may increase the grid resolution for your problem if desired.

```
# setup the pre-defined range of alpha and lambda values
lambdas = 10 ** (-5.4 - 1 * (np.arange(5) / 4))
alphas = 10 ** (-4.5 - 1.5 * (np.arange(5) / 4))

# setup the smooth lasso cross-validation class
s_lasso = SmoothLassoCV(
    alphas=alphas,  # A numpy array of alpha values.
    lambdas=lambdas,  # A numpy array of lambda values.
    sigma=0.00070,  # The standard deviation of noise from the MAT dataset.
    folds=10,  # The number of folds in n-folds cross-validation.
    inverse_dimension=inverse_dimensions,  # previously defined inverse dimensions.
    verbose=1,  # If non-zero, prints the progress as the computation proceeds.
    max_iterations=20000,  # The maximum number of allowed interations.
)

# run fit using the compressed kernel and compressed data.
s_lasso.fit(compressed_K, compressed_s)
```

Out:

```
[Parallel(n_jobs=-1)]: Using backend ThreadingBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done   5 out of   5 | elapsed:    7.2s finished
[Parallel(n_jobs=-1)]: Using backend ThreadingBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done   5 out of   5 | elapsed:    9.3s finished
[Parallel(n_jobs=-1)]: Using backend ThreadingBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done   5 out of   5 | elapsed:   11.6s finished
[Parallel(n_jobs=-1)]: Using backend ThreadingBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done   5 out of   5 | elapsed:   16.7s finished
[Parallel(n_jobs=-1)]: Using backend ThreadingBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done   5 out of   5 | elapsed:   24.2s finished
```

The optimum hyper-parameters

Use the *hyperparameters* (page 22) attribute of the instance for the optimum hyper-parameters, α and λ, determined from the cross-validation.

```
print(s_lasso.hyperparameters)
```

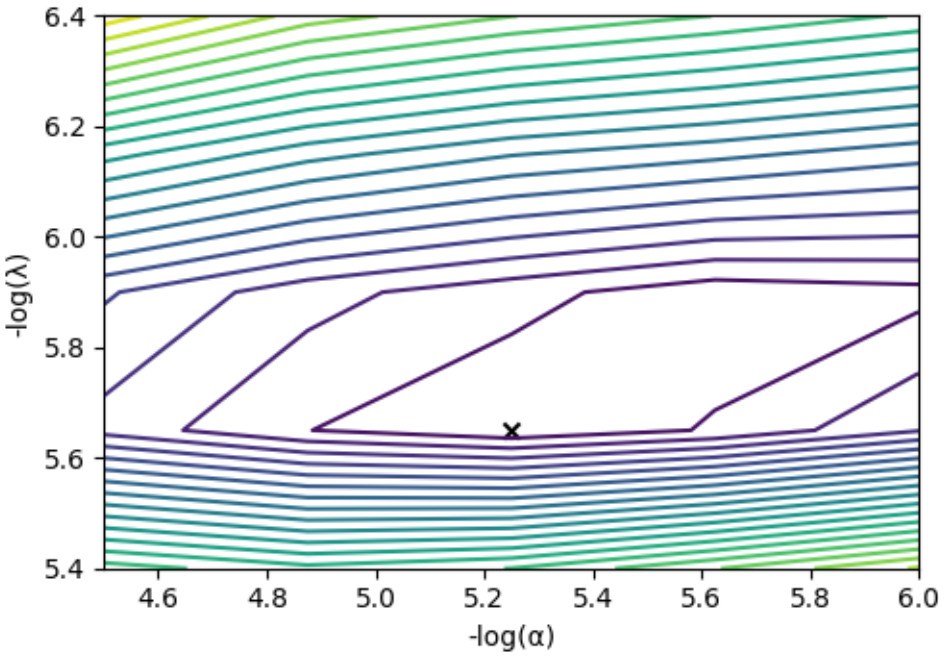
Out:

```
{'alpha': 5.623413251903491e-06, 'lambda': 2.2387211385683376e-06}
```

The cross-validation surface

Optionally, you may want to visualize the cross-validation error curve/surface. Use the *cross_validation_curve* (page 22) attribute of the instance, as follows

```
CV_metric = s_lasso.cross_validation_curve  # `CV_metric` is a CSDM object.

# plot of the cross validation surface
plt.figure(figsize=(5, 3.5))
ax = plt.subplot(projection="csdm")
ax.contour(np.log10(CV_metric), levels=25)
ax.scatter(
    -np.log10(s_lasso.hyperparameters["alpha"]),
    -np.log10(s_lasso.hyperparameters["lambda"]),
    marker="x",
    color="k",
)
plt.tight_layout(pad=0.5)
plt.show()
```



The optimum solution

The *f* (page 22) attribute of the instance holds the solution corresponding to the optimum hyper-parameters,

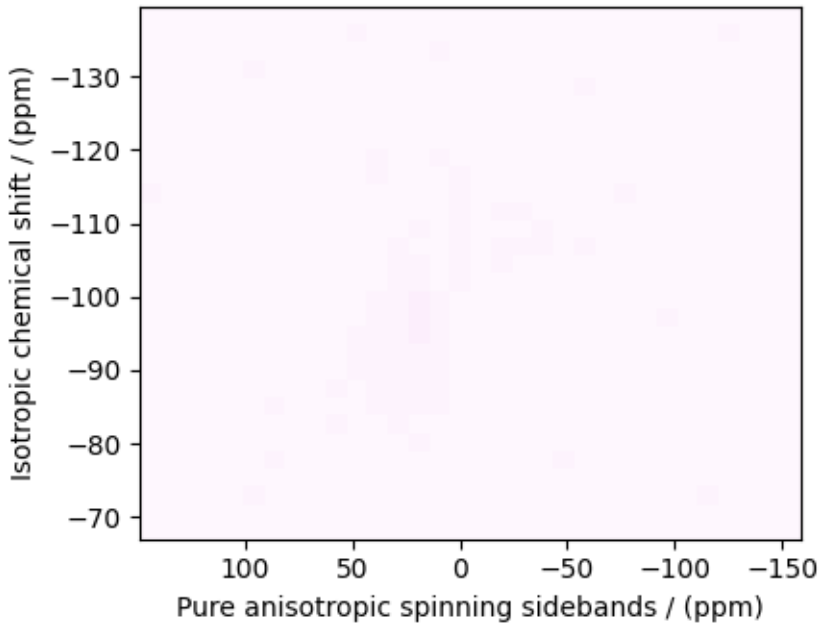
```
f_sol = s_lasso.f  # f_sol is a CSDM object.
```

where `f_sol` is the optimum solution.

The fit residuals

To calculate the residuals between the data and predicted data(fit), use the *residuals()* (page 20) method, as follows,

```
residuals = s_lasso.residuals(K=K, s=data_object_truncated)
# residuals is a CSDM object.

# The plot of the residuals.
plot2D(residuals, vmax=data_object_truncated.max(), vmin=data_object_truncated.min())
```



The standard deviation of the residuals is

```
residuals.std()
```

Out:

```
<Quantity 0.0011629>
```

Saving the solution

To serialize the solution to a file, use the *save()* method of the CSDM object, for example,

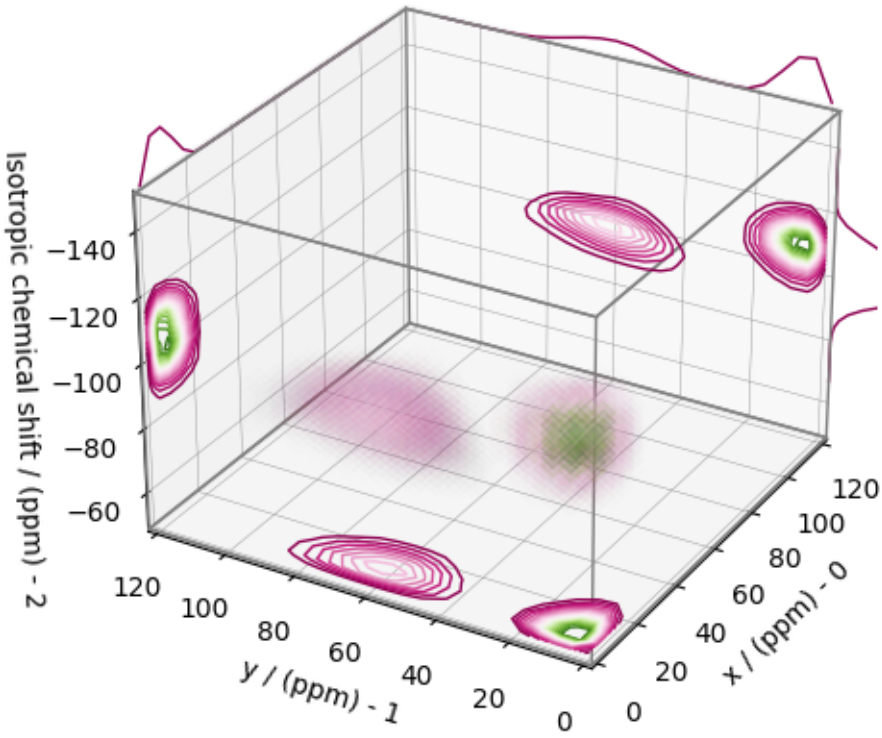
```
f_sol.save("KMg_mixed_silicate_inverse.csdf")  # save the solution
residuals.save("KMg_mixed_silicate_residue.csdf")  # save the residuals
```

Data Visualization

At this point, we have solved the inverse problem and obtained an optimum distribution of the nuclear shielding tensor parameters from the 2D MAT dataset. You may use any data visualization and interpretation tool of choice for further analysis. In the following sections, we provide minimal visualization and analysis to complete the case study.

Visualizing the 3D solution

```
# Normalize the solution
f_sol /= f_sol.max()

# Convert the coordinates of the solution, `f_sol`, from Hz to ppm.
[item.to("ppm", "nmr_frequency_ratio") for item in f_sol.dimensions]

# The 3D plot of the solution
plt.figure(figsize=(5, 4.4))
ax = plt.subplot(projection="3d")
plot_3d(ax, f_sol, x_lim=[0, 120], y_lim=[0, 120], z_lim=[-50, -150])
plt.tight_layout()
plt.show()
```



From the 3D plot, we observe two distinct regions: one for the Q^4 sites and another for the Q^3 sites. Select the respective regions by using the appropriate array indexing,

```
Q4_region = f_sol[0:8, 0:8, 5:25]
Q4_region.description = "Q4 region"

Q3_region = f_sol[0:8, 7:24, 7:25]
Q3_region.description = "Q3 region"
```

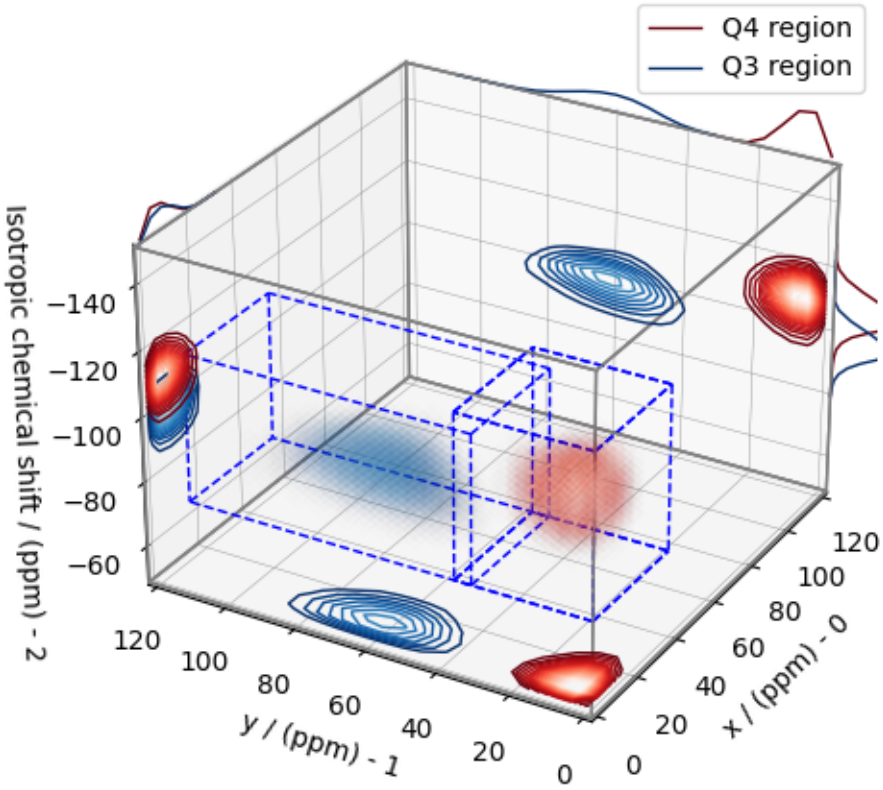
The plot of the respective regions is shown below.

```
# Calculate the normalization factor for the 2D contours and 1D projections from the
# original solution, `f_sol`. Use this normalization factor to scale the intensities
# from the sub-regions.
max_2d = [
    f_sol.sum(axis=0).max().value,
    f_sol.sum(axis=1).max().value,
    f_sol.sum(axis=2).max().value,
]
max_1d = [
    f_sol.sum(axis=(1, 2)).max().value,
    f_sol.sum(axis=(0, 2)).max().value,
    f_sol.sum(axis=(0, 1)).max().value,
]

plt.figure(figsize=(5, 4.4))
ax = plt.subplot(projection="3d")
```

(continues on next page)

(continued from previous page)

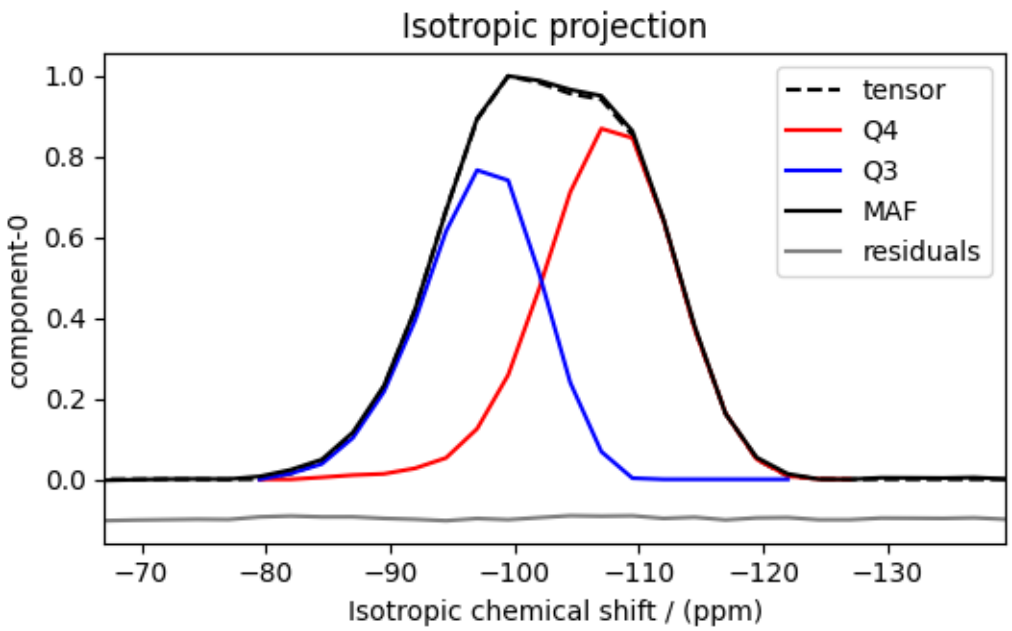
```
# plot for the Q4 region
plot_3d(
    ax,
    Q4_region,
    x_lim=[0, 120],  # the x-limit
    y_lim=[0, 120],  # the y-limit
    z_lim=[-50, -150],  # the z-limit
    max_2d=max_2d,  # normalization factors for the 2D contours projections
    max_1d=max_1d,  # normalization factors for the 1D projections
    cmap=cm.Reds_r,  # colormap
    box=True,  # draw a box around the region
)
# plot for the Q3 region
plot_3d(
    ax,
    Q3_region,
    x_lim=[0, 120],  # the x-limit
    y_lim=[0, 120],  # the y-limit
    z_lim=[-50, -150],  # the z-limit
    max_2d=max_2d,  # normalization factors for the 2D contours projections
    max_1d=max_1d,  # normalization factors for the 1D projections
    cmap=cm.Blues_r,  # colormap
    box=True,  # draw a box around the region
)
ax.legend()
plt.tight_layout()
plt.show()
```



Visualizing the isotropic projections.

Because the Q^4 and Q^3 regions are fully resolved after the inversion, evaluating the contributions from these regions is trivial. For examples, the distribution of the isotropic chemical shifts for these regions are

```
# Isotropic chemical shift projection of the 2D MAT dataset.
data_iso = data_object_truncated.sum(axis=0)
data_iso /= data_iso.max()  # normalize the projection

# Isotropic chemical shift projection of the tensor distribution dataset.
f_sol_iso = f_sol.sum(axis=(0, 1))

# Isotropic chemical shift projection of the tensor distribution for the Q4 region.
Q4_region_iso = Q4_region.sum(axis=(0, 1))

# Isotropic chemical shift projection of the tensor distribution for the Q3 region.
Q3_region_iso = Q3_region.sum(axis=(0, 1))

# Normalize the three projections.
f_sol_iso_max = f_sol_iso.max()
f_sol_iso /= f_sol_iso_max
Q4_region_iso /= f_sol_iso_max
Q3_region_iso /= f_sol_iso_max

# The plot of the different projections.
plt.figure(figsize=(5.5, 3.5))
ax = plt.subplot(projection="csdm")
ax.plot(f_sol_iso, "--k", label="tensor")
ax.plot(Q4_region_iso, "r", label="Q4")
ax.plot(Q3_region_iso, "b", label="Q3")
ax.plot(data_iso, "-k", label="MAF")
ax.plot(data_iso - f_sol_iso - 0.1, "gray", label="residuals")
ax.set_title("Isotropic projection")
ax.invert_xaxis()
plt.legend()
plt.tight_layout()
plt.show()
```



Notice the shape of the isotropic chemical shift distribution for the Q^4 sites is skewed, which is expected.

Analysis

For the analysis, we use the statistics module of the csdmpy package. Following is the moment analysis of the 3D volumes for both the Q^4 and Q^3 regions up to the second moment.

```
int_Q4 = stats.integral(Q4_region)  # volume of the Q4 distribution
mean_Q4 = stats.mean(Q4_region)  # mean of the Q4 distribution
std_Q4 = stats.std(Q4_region)  # standard deviation of the Q4 distribution

int_Q3 = stats.integral(Q3_region)  # volume of the Q3 distribution
mean_Q3 = stats.mean(Q3_region)  # mean of the Q3 distribution
std_Q3 = stats.std(Q3_region)  # standard deviation of the Q3 distribution

print("Q4 statistics")
print(f"\tpopulation = {100 * int_Q4 / (int_Q4 + int_Q3)}%")
print("\tmean\n\t\tx:\t{0}\n\t\ty:\t{1}\n\t\tiso:\t{2}".format(*mean_Q4))
print("\tstandard deviation\n\t\tx:\t{0}\n\t\ty:\t{1}\n\t\tiso:\t{2}".format(*std_Q4))

print("Q3 statistics")
print(f"\tpopulation = {100 * int_Q3 / (int_Q4 + int_Q3)}%")
print("\tmean\n\t\tx:\t{0}\n\t\ty:\t{1}\n\t\tiso:\t{2}".format(*mean_Q3))
print("\tstandard deviation\n\t\tx:\t{0}\n\t\ty:\t{1}\n\t\tiso:\t{2}".format(*std_Q3))
```

Out:

```
Q4 statistics
        population = 55.526502953829876%
        mean
                x:      8.668370723621372 ppm
                y:      8.895931313361551 ppm
                iso:    -107.32391005580547 ppm
```

(continues on next page)

(continued from previous page)

```
        standard deviation
                x:      4.733329589219853 ppm
                y:      4.92857990438073 ppm
                iso:    5.426968389595141 ppm
Q3 statistics
        population = 44.473497046170124%
        mean
                x:      10.133779224057403 ppm
                y:      62.261856413140734 ppm
                iso:    -97.01636670566391 ppm
        standard deviation
                x:      4.540085672075755 ppm
                y:      10.65723728897496 ppm
                iso:    4.741843009042262 ppm
```

The statistics shown above are according to the respective dimensions, that is, the x, y, and the isotropic chemical shifts. To convert the x and y statistics to commonly used ζ and η statistics, use the *x_y_to_zeta_eta()* (page 23) function.

```
mean_ζη_Q3 = x_y_to_zeta_eta(*mean_Q3[0:2])

# error propagation for calculating the standard deviation
std_ζ = (std_Q3[0] * mean_Q3[0]) ** 2 + (std_Q3[1] * mean_Q3[1]) ** 2
std_ζ /= mean_Q3[0] ** 2 + mean_Q3[1] ** 2
std_ζ = np.sqrt(std_ζ)

std_η = (std_Q3[1] * mean_Q3[0]) ** 2 + (std_Q3[0] * mean_Q3[1]) ** 2
std_η /= (mean_Q3[0] ** 2 + mean_Q3[1] ** 2) ** 2
std_η = (4 / np.pi) * np.sqrt(std_η)

print("Q3 statistics")
print(f"\tpopulation = {100 * int_Q3 / (int_Q4 + int_Q3)}%")
print("\tmean\n\t\tζ:\t{0}\n\t\tη:\t{1}\n\t\tiso:\t{2}".format(*mean_ζη_Q3, mean_Q3[2]))
print(
    "\tstandard deviation\n\t\tζ:\t{0}\n\t\tη:\t{1}\n\t\tiso:\t{2}".format(
        std_ζ, std_η, std_Q3[2]
    )
)
```

Out:

```
Q3 statistics
        population = 44.473497046170124%
        mean
                ζ:      63.08115602438252 ppm
                η:      0.20543188190783745
                iso:    -97.01636670566391 ppm
        standard deviation
                ζ:      10.544076189106075 ppm
                η:      0.0968240700650499
                iso:    4.741843009042262 ppm
```

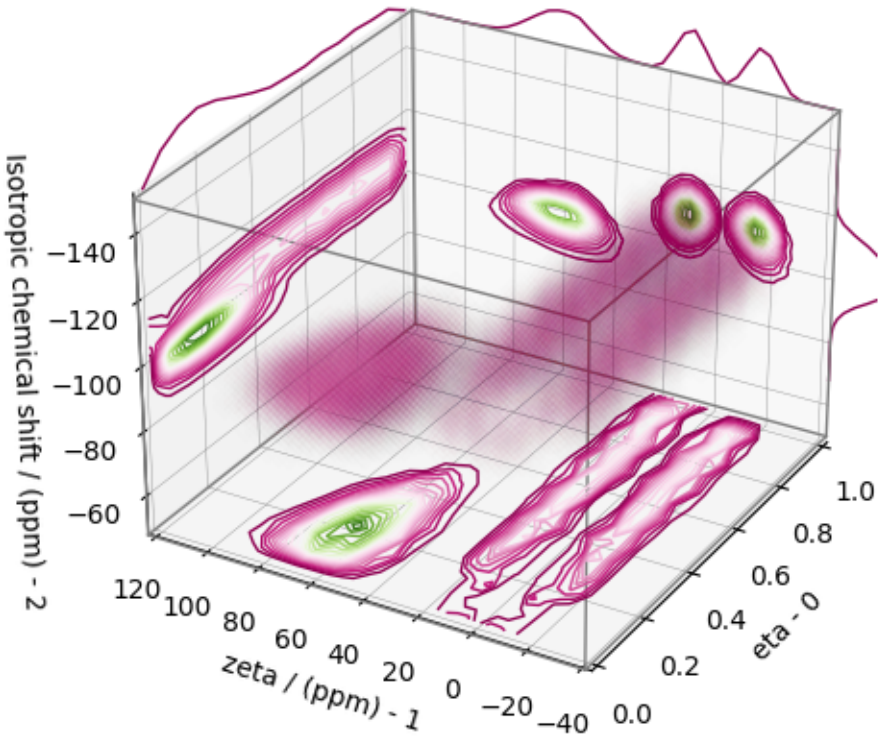
Convert the 3D tensor distribution in Haeberlen parameters

You may re-bin the 3D tensor parameter distribution from a $\rho(\delta_{\text{iso}}, x, y)$ distribution to $\rho(\delta_{\text{iso}}, \zeta_\sigma, \eta_\sigma)$ distribution as follows.

```
# Create the zeta and eta dimensions,, as shown below.
zeta = cp.as_dimension(np.arange(40) * 4 - 40, unit="ppm", label="zeta")
eta = cp.as_dimension(np.arange(16) / 15, label="eta")

# Use the `to_Haeberlen_grid` function to convert the tensor parameter distribution.
fsol_Hae = to_Haeberlen_grid(f_sol, zeta, eta)
```

The 3D plot

```
plt.figure(figsize=(5, 4.4))
ax = plt.subplot(projection="3d")
plot_3d(ax, fsol_Hae, x_lim=[0, 1], y_lim=[-40, 120], z_lim=[-50, -150], alpha=0.4)
plt.tight_layout()
plt.show()
```



References

Total running time of the script: (1 minutes 21.745 seconds)

2.1.3 Magic angle flipping (Experiment)

The following are the examples of the statistical learning method applied in determining a distribution of the nuclear shielding tensor parameters from a 2D MAF NMR spectrum correlating the isotropic to the anisotropic frequency contributions.

2D MAF data of Rb2O.2.25SiO2 glass

The following example is an application of the statistical learning method in determining the distribution of the nuclear shielding tensor parameters from a 2D magic-angle flipping (MAF) spectrum. In this example, we use the 2D MAF spectrum[1] of $Rb_2O \cdot 2.25SiO_2$ glass.

Before getting started

Import all relevant packages.

```
import csdmpy as cp
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import cm
from csdmpy import statistics as stats

from mrinversion.kernel.nmr import ShieldingPALineshape
from mrinversion.kernel.utils import x_y_to_zeta_eta
from mrinversion.linear_model import SmoothLassoCV, TSVDCompression
from mrinversion.utils import plot_3d, to_Haeberlen_grid
```

Setup for the matplotlib figures.

```
# function for plotting 2D dataset
def plot2D(csdm_object, **kwargs):
    plt.figure(figsize=(4.5, 3.5))
    ax = plt.subplot(projection="csdm")
    ax.imshow(csdm_object, cmap="gist_ncar_r", aspect="auto", **kwargs)
    ax.invert_xaxis()
    ax.invert_yaxis()
    plt.tight_layout()
    plt.show()
```

[1] Baltisberger, J. H., Florian, P., Keeler, E. G., Phyo, P. A., Sanders, K. J., Grandinetti, P. J.. Modifier cation effects on 29Si nuclear shielding anisotropies in silicate glasses, J. Magn. Reson. 268 (2016) 95 106. doi:10.1016/j.jmr.2016.05.003.

Dataset setup

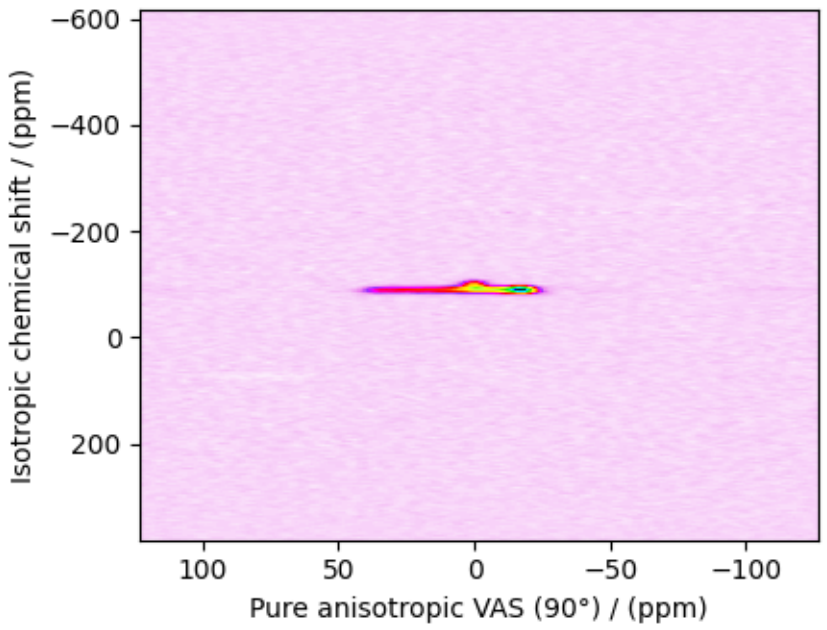
Import the dataset

Load the dataset. In this example, we import the dataset as the CSDM data-object.

```
# The 2D MAF dataset in csdm format
filename = "https://zenodo.org/record/3964531/files/Rb2O-2_25SiO2-MAF.csdf"
data_object = cp.load(filename)

# For inversion, we only interest ourselves with the real part of the complex dataset.
data_object = data_object.real

# We will also convert the coordinates of both dimensions from Hz to ppm.
_ = [item.to("ppm", "nmr_frequency_ratio") for item in data_object.dimensions]
```

Here, the variable `data_object` is a CSDM object that holds the real part of the 2D MAF dataset. The plot of the 2D MAF dataset is

```
plot2D(data_object)
```



There are two dimensions in this dataset. The dimension at index 0, the horizontal dimension in the figure, is the pure anisotropic dimension, while the dimension at index 1, the vertical dimension, is the isotropic chemical shift dimension. The number of coordinates along the respective dimensions is

```
print(data_object.shape)
```

Out:

```
(128, 512)
```

with 128 points along the anisotropic dimension (index 0) and 512 points along the isotropic chemical shift dimension (index 1).

Prepping the data for inversion

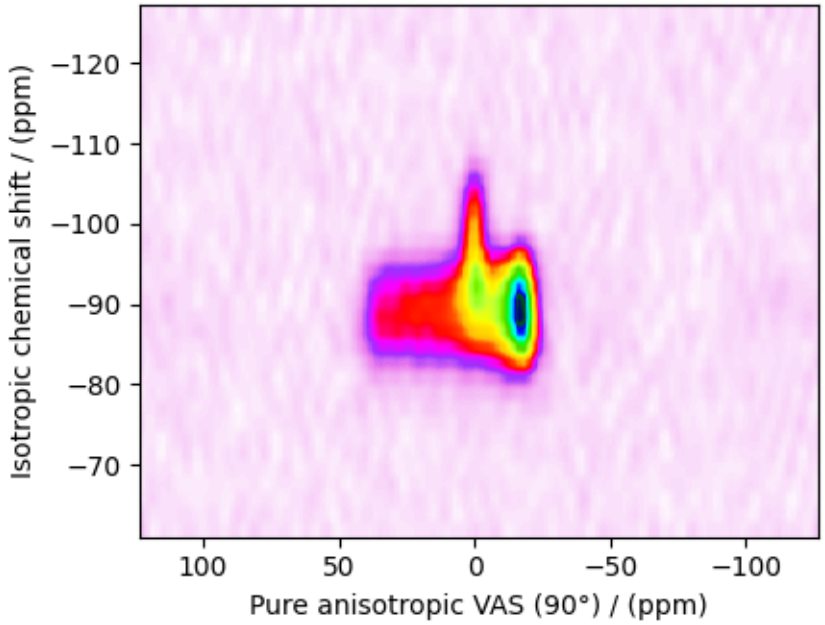
Step-1: Data Alignment

When using the csdm objects with the `mrinversion` package, the dimension at index 0 must be the dimension undergoing the linear inversion. In this example, we plan to invert the pure anisotropic shielding line-shape. Since the anisotropic dimension in `data_object` is already at index 0, no further action is required.

Step-2: Optimization

Notice, the signal from the 2D MAF dataset occupies a small fraction of the two-dimensional frequency grid. Though you may choose to proceed with the inversion directly onto this dataset, it is not computationally optimum. For optimum performance, trim the dataset to the region of relevant signals. Use the appropriate array indexing/slicing to select the signal region.

```
data_object_truncated = data_object[:, 250:285]
plot2D(data_object_truncated)
```



In the above code, we truncate the isotropic chemical shifts, dimension at index 1, to coordinate between indexes 250 and 285. The isotropic shift coordinates from the truncated dataset are

```
print(data_object_truncated.dimensions[1].coordinates)
```

Out:

```
[-127.27782256 -125.3275251  -123.37722764 -121.42693019 -119.47663273
 -117.52633527 -115.57603781 -113.62574035 -111.6754429  -109.72514544
 -107.77484798 -105.82455052 -103.87425306 -101.92395561  -99.97365815
  -98.02336069  -96.07306323  -94.12276577  -92.17246832  -90.22217086
  -88.2718734   -86.32157594  -84.37127848  -82.42098103  -80.47068357
  -78.52038611  -76.57008865  -74.6197912   -72.66949374  -70.71919628
  -68.76889882  -66.81860136  -64.86830391  -62.91800645  -60.96770899] ppm
```

Linear Inversion setup

Dimension setup

In a generic linear-inverse problem, one needs to define two sets of dimensions—the dimensions undergoing a linear transformation, and the dimensions onto which the inversion method transforms the data. In the line-shape inversion, the two sets of dimensions are the anisotropic dimension and the *x*-*y* dimensions.

Anisotropic-dimension: The dimension of the dataset that holds the pure anisotropic frequency contributions. In `mrinversion`, this must always be the dimension at index 0 of the data object.

```
anisotropic_dimension = data_object_truncated.dimensions[0]
```

x-y dimensions: The two inverse dimensions corresponding to the *x* and *y*-axis of the *x*-*y* grid.

```
inverse_dimensions = [
    cp.LinearDimension(count=25, increment="400 Hz", label="x"),  # the `x`-dimension.
    cp.LinearDimension(count=25, increment="400 Hz", label="y"),  # the `y`-dimension.
]
```

Generating the kernel

For MAF datasets, the line-shape kernel corresponds to the pure nuclear shielding anisotropy line-shapes. Use the *ShieldingPALineshape* (page 17) class to generate a shielding line-shape kernel.

```
lineshape = ShieldingPALineshape(
    anisotropic_dimension=anisotropic_dimension,
    inverse_dimension=inverse_dimensions,
    channel="29Si",
    magnetic_flux_density="9.4 T",
    rotor_angle="90°",
    rotor_frequency="13 kHz",
    number_of_sidebands=4,
)
```

Here, `lineshape` is an instance of the *ShieldingPALineshape* (page 17) class. The required arguments of this class are the *anisotropic_dimension*, *inverse_dimension*, and *channel*. We have already defined the first two arguments in the previous sub-section. The value of the *channel* argument is the nucleus observed in the MAF experiment. In this example, this value is '29Si'. The remaining arguments, such as the *magnetic_flux_density*, *rotor_angle*, and *rotor_frequency*, are set to match the conditions under which the 2D MAF spectrum was acquired. Note for the MAF measurements the rotor angle is usually $90°$ for the anisotropic dimension. The value of the *number_of_sidebands* argument is the number of sidebands calculated for each line-shape within the kernel. Unless, you have a lot of spinning sidebands in your MAF dataset, the value of this argument is generally low. Here, we calculate four spinning sidebands per line-shape within in the kernel.

Once the ShieldingPALineshape instance is created, use the *kernel()* (page 18) method of the instance to generate the MAF line-shape kernel.

```
K = lineshape.kernel(supersampling=1)
print(K.shape)
```

Out:

```
(128, 625)
```

The kernel `K` is a NumPy array of shape (128, 625), where the axes with 128 and 625 points are the anisotropic dimension and the features (x-y coordinates) corresponding to the 25×25 *x*-*y* grid, respectively.

Data Compression

Data compression is optional but recommended. It may reduce the size of the inverse problem and, thus, further computation time.

```
new_system = TSVDCompression(K, data_object_truncated)
compressed_K = new_system.compressed_K
compressed_s = new_system.compressed_s

print(f"truncation_index = {new_system.truncation_index}")
```

Out:

```
compression factor = 1.471264367816092
truncation_index = 87
```

Solving the inverse problem

Smooth LASSO cross-validation

Solve the smooth-lasso problem. Use the statistical learning `SmoothLassoCV` method to solve the inverse problem over a range of α and λ values and determine the best nuclear shielding tensor parameter distribution for the given 2D MAF dataset. Considering the limited build time for the documentation, we'll perform the cross-validation over a smaller 5×5 x-y grid. You may increase the grid resolution for your problem if desired.

```
# setup the pre-defined range of alpha and lambda values
lambdas = 10 ** (-5.2 - 1.25 * (np.arange(5) / 4))
alphas = 10 ** (-5.5 - 1.25 * (np.arange(5) / 4))

# setup the smooth lasso cross-validation class
s_lasso = SmoothLassoCV(
    alphas=alphas,  # A numpy array of alpha values.
    lambdas=lambdas,  # A numpy array of lambda values.
    sigma=0.0045,  # The standard deviation of noise from the 2D MAF data.
    folds=10,  # The number of folds in n-folds cross-validation.
    inverse_dimension=inverse_dimensions,  # previously defined inverse dimensions.
    verbose=1,  # If non-zero, prints the progress as the computation proceeds.
)

# run the fit method on the compressed kernel and compressed data.
s_lasso.fit(K=compressed_K, s=compressed_s)
```

Out:

```
[Parallel(n_jobs=-1)]: Using backend ThreadingBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done   5 out of   5 | elapsed:    7.2s finished
[Parallel(n_jobs=-1)]: Using backend ThreadingBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done   5 out of   5 | elapsed:    7.6s finished
[Parallel(n_jobs=-1)]: Using backend ThreadingBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done   5 out of   5 | elapsed:    8.7s finished
[Parallel(n_jobs=-1)]: Using backend ThreadingBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done   5 out of   5 | elapsed:   10.4s finished
[Parallel(n_jobs=-1)]: Using backend ThreadingBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done   5 out of   5 | elapsed:   11.9s finished
```

The optimum hyper-parameters

Use the *hyperparameters* (page 22) attribute of the instance for the optimum hyper-parameters, α and λ, determined from the cross-validation.

```
print(s_lasso.hyperparameters)
```

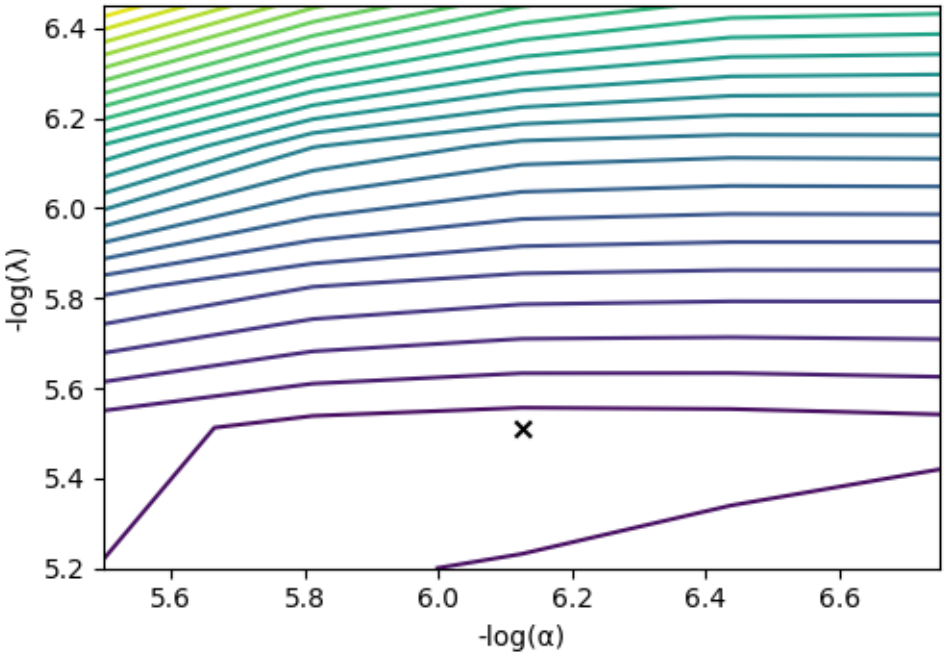
Out:

```
{'alpha': 7.498942093324558e-07, 'lambda': 3.0725573652674454e-06}
```

The cross-validation surface

Optionally, you may want to visualize the cross-validation error curve/surface. Use the *cross_validation_curve* (page 22) attribute of the instance, as follows

```
CV_metric = s_lasso.cross_validation_curve  # `CV_metric` is a CSDM object.

# plot of the cross validation surface
plt.figure(figsize=(5, 3.5))
ax = plt.subplot(projection="csdm")
ax.contour(np.log10(CV_metric), levels=25)
ax.scatter(
    -np.log10(s_lasso.hyperparameters["alpha"]),
    -np.log10(s_lasso.hyperparameters["lambda"]),
    marker="x",
    color="k",
)
plt.tight_layout(pad=0.5)
plt.show()
```



The optimum solution

The *f* (page 22) attribute of the instance holds the solution corresponding to the optimum hyper-parameters,

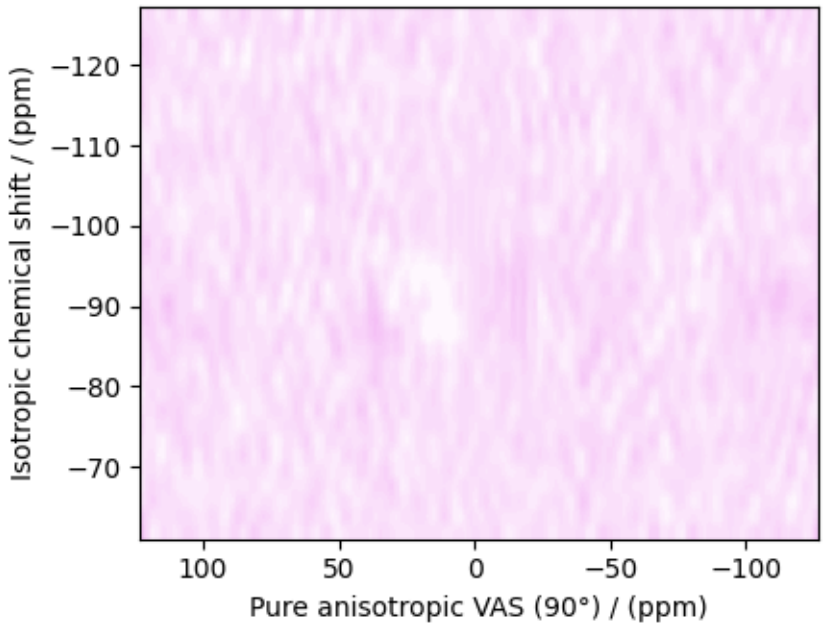
```
f_sol = s_lasso.f  # f_sol is a CSDM object.
```

where `f_sol` is the optimum solution.

The fit residuals

To calculate the residuals between the data and predicted data(fit), use the *residuals()* (page 23) method, as follows,

```
residuals = s_lasso.residuals(K=K, s=data_object_truncated)
# residuals is a CSDM object.

# The plot of the residuals.
plot2D(residuals, vmax=data_object_truncated.max(), vmin=data_object_truncated.min())
```



The standard deviation of the residuals is close to the standard deviation of the noise, $\sigma = 0.0043$.

```
residuals.std()
```

Out:

```
<Quantity 0.0047572>
```

Saving the solution

To serialize the solution (nuclear shielding tensor parameter distribution) to a file, use the *save()* method of the CSDM object, for example,

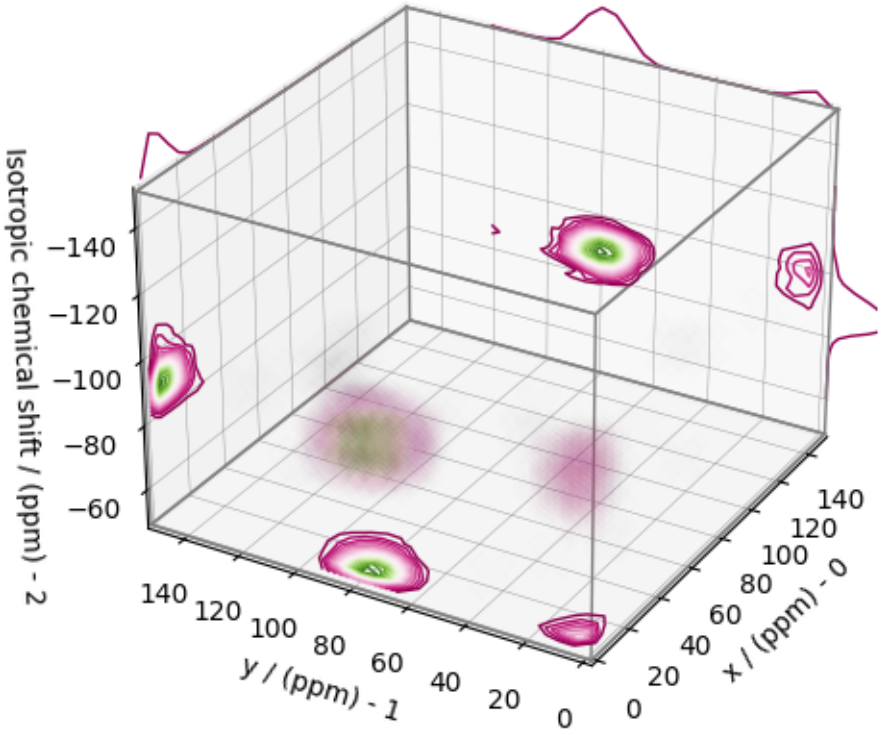
```
f_sol.save("Rb2O.2.25SiO2_inverse.csdf")  # save the solution
residuals.save("Rb2O.2.25SiO2_residue.csdf")  # save the residuals
```

Data Visualization

At this point, we have solved the inverse problem and obtained an optimum distribution of the nuclear shielding tensor parameters from the 2D MAF dataset. You may use any data visualization and interpretation tool of choice for further analysis. In the following sections, we provide minimal visualization and analysis to complete the case study.

Visualizing the 3D solution

```
# Normalize the solution
f_sol /= f_sol.max()

# Convert the coordinates of the solution, `f_sol`, from Hz to ppm.
[item.to("ppm", "nmr_frequency_ratio") for item in f_sol.dimensions]

# The 3D plot of the solution
plt.figure(figsize=(5, 4.4))
ax = plt.subplot(projection="3d")
plot_3d(ax, f_sol, x_lim=[0, 150], y_lim=[0, 150], z_lim=[-50, -150])
plt.tight_layout()
plt.show()
```



From the 3D plot, we observe two distinct regions: one for the Q^4 sites and another for the Q^3 sites. Select the respective regions by using the appropriate array indexing,

```
Q4_region = f_sol[0:7, 0:7, 4:25]
Q4_region.description = "Q4 region"

Q3_region = f_sol[0:8, 10:24, 11:30]
Q3_region.description = "Q3 region"
```

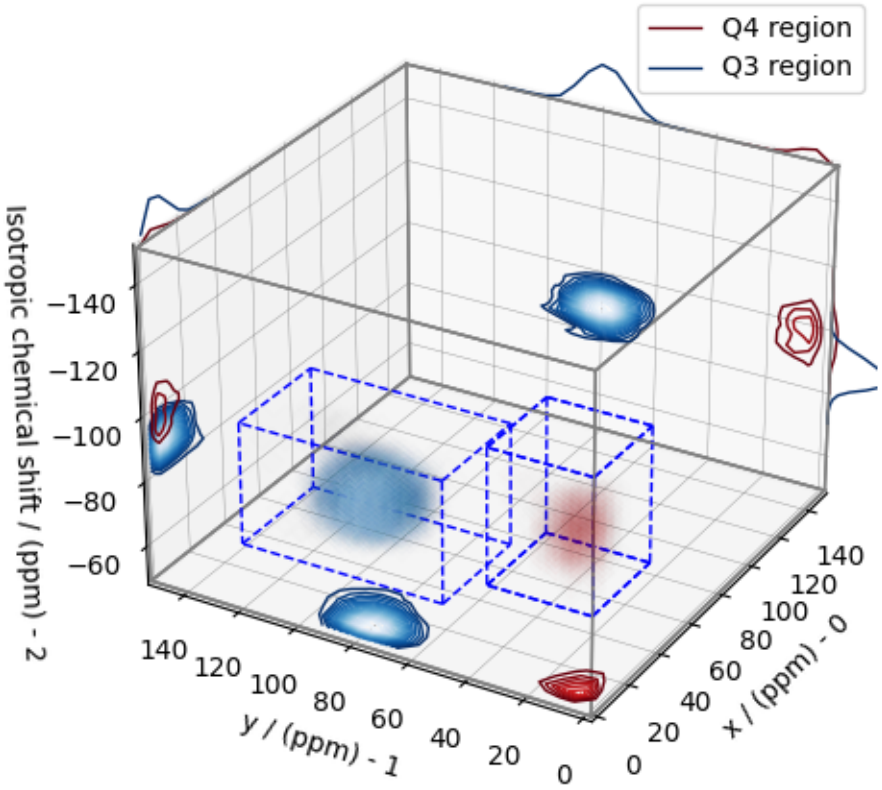
The plot of the respective regions is shown below.

```
# Calculate the normalization factor for the 2D contours and 1D projections from the
# original solution, `f_sol`. Use this normalization factor to scale the intensities
# from the sub-regions.
max_2d = [
    f_sol.sum(axis=0).max().value,
    f_sol.sum(axis=1).max().value,
    f_sol.sum(axis=2).max().value,
]
max_1d = [
    f_sol.sum(axis=(1, 2)).max().value,
    f_sol.sum(axis=(0, 2)).max().value,
    f_sol.sum(axis=(0, 1)).max().value,
]

plt.figure(figsize=(5, 4.4))
ax = plt.subplot(projection="3d")
```

(continues on next page)

(continued from previous page)

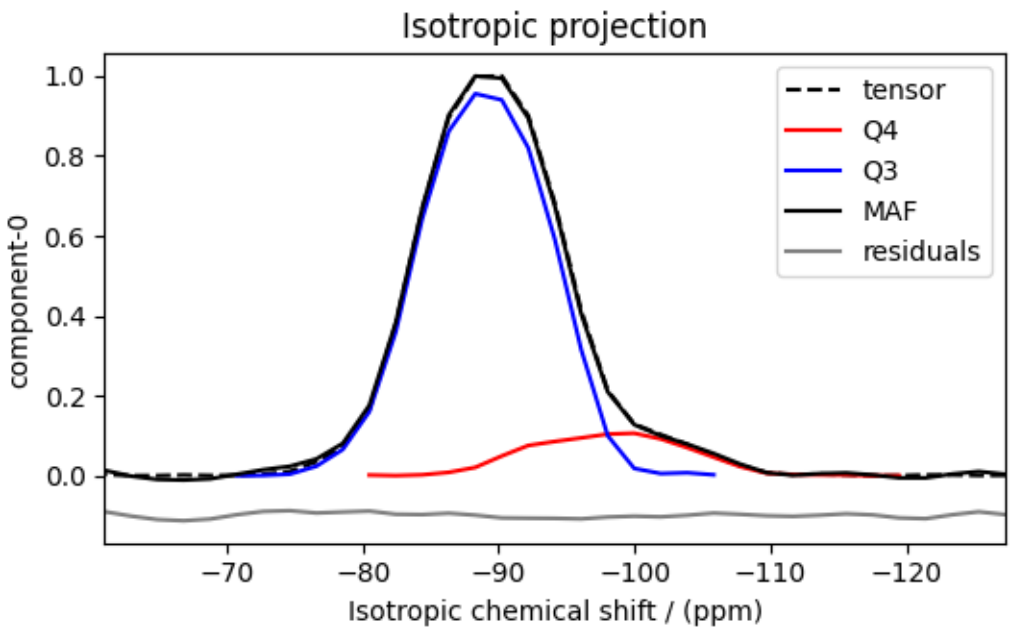
```
# plot for the Q4 region
plot_3d(
    ax,
    Q4_region,
    x_lim=[0, 150],  # the x-limit
    y_lim=[0, 150],  # the y-limit
    z_lim=[-50, -150],  # the z-limit
    max_2d=max_2d,  # normalization factors for the 2D contours projections
    max_1d=max_1d,  # normalization factors for the 1D projections
    cmap=cm.Reds_r,  # colormap
    box=True,  # draw a box around the region
)
# plot for the Q3 region
plot_3d(
    ax,
    Q3_region,
    x_lim=[0, 150],  # the x-limit
    y_lim=[0, 150],  # the y-limit
    z_lim=[-50, -150],  # the z-limit
    max_2d=max_2d,  # normalization factors for the 2D contours projections
    max_1d=max_1d,  # normalization factors for the 1D projections
    cmap=cm.Blues_r,  # colormap
    box=True,  # draw a box around the region
)
ax.legend()
plt.tight_layout()
plt.show()
```



Visualizing the isotropic projections.

Because the Q^4 and Q^3 regions are fully resolved after the inversion, evaluating the contributions from these regions is trivial. For examples, the distribution of the isotropic chemical shifts for these regions are

```
# Isotropic chemical shift projection of the 2D MAF dataset.
data_iso = data_object_truncated.sum(axis=0)
data_iso /= data_iso.max()  # normalize the projection

# Isotropic chemical shift projection of the tensor distribution dataset.
f_sol_iso = f_sol.sum(axis=(0, 1))

# Isotropic chemical shift projection of the tensor distribution for the Q4 region.
Q4_region_iso = Q4_region.sum(axis=(0, 1))

# Isotropic chemical shift projection of the tensor distribution for the Q3 region.
Q3_region_iso = Q3_region.sum(axis=(0, 1))

# Normalize the three projections.
f_sol_iso_max = f_sol_iso.max()
f_sol_iso /= f_sol_iso_max
Q4_region_iso /= f_sol_iso_max
Q3_region_iso /= f_sol_iso_max

# The plot of the different projections.
plt.figure(figsize=(5.5, 3.5))
ax = plt.subplot(projection="csdm")
ax.plot(f_sol_iso, "--k", label="tensor")
ax.plot(Q4_region_iso, "r", label="Q4")
ax.plot(Q3_region_iso, "b", label="Q3")
ax.plot(data_iso, "-k", label="MAF")
ax.plot(data_iso - f_sol_iso - 0.1, "gray", label="residuals")
ax.set_title("Isotropic projection")
ax.invert_xaxis()
plt.legend()
plt.tight_layout()
plt.show()
```



Analysis

For the analysis, we use the statistics module of the csdmpy package. Following is the moment analysis of the 3D volumes for both the Q^4 and Q^3 regions up to the second moment.

```
int_Q4 = stats.integral(Q4_region)  # volume of the Q4 distribution
mean_Q4 = stats.mean(Q4_region)  # mean of the Q4 distribution
std_Q4 = stats.std(Q4_region)  # standard deviation of the Q4 distribution

int_Q3 = stats.integral(Q3_region)  # volume of the Q3 distribution
mean_Q3 = stats.mean(Q3_region)  # mean of the Q3 distribution
std_Q3 = stats.std(Q3_region)  # standard deviation of the Q3 distribution

print("Q4 statistics")
print(f"\tpopulation = {100 * int_Q4 / (int_Q4 + int_Q3)}%")
print("\tmean\n\t\tx:\t{0}\n\t\ty:\t{1}\n\t\tiso:\t{2}".format(*mean_Q4))
print("\tstandard deviation\n\t\tx:\t{0}\n\t\ty:\t{1}\n\t\tiso:\t{2}".format(*std_Q4))

print("Q3 statistics")
print(f"\tpopulation = {100 * int_Q3 / (int_Q4 + int_Q3)}%")
print("\tmean\n\t\tx:\t{0}\n\t\ty:\t{1}\n\t\tiso:\t{2}".format(*mean_Q3))
print("\tstandard deviation\n\t\tx:\t{0}\n\t\ty:\t{1}\n\t\tiso:\t{2}".format(*std_Q3))
```

Out:

```
Q4 statistics
        population = 11.890538538861403%
        mean
                x:      8.278457487623776 ppm
                y:      8.720888069270229 ppm
                iso:    -98.05907744216225 ppm
        standard deviation
                x:      4.367398301090801 ppm
```

(continues on next page)

(continued from previous page)

```
                y:      4.694573289799781 ppm
                iso:    5.396278130701522 ppm
Q3 statistics
        population = 88.1094614611386%
        mean
                x:      10.13012360588423 ppm
                y:      79.93510899144792 ppm
                iso:    -88.9254977266946 ppm
        standard deviation
                x:      6.636438823889573 ppm
                y:      8.12109455821077 ppm
                iso:    4.4050990063512145 ppm
```

The statistics shown above are according to the respective dimensions, that is, the x, y, and the isotropic chemical shifts. To convert the x and y statistics to commonly used ζ_σ and η_σ statistics, use the *x_y_to_zeta_eta()* (page 23) function.

```
mean_ζη_Q3 = x_y_to_zeta_eta(*mean_Q3[0:2])

# error propagation for calculating the standard deviation
std_ζ = (std_Q3[0] * mean_Q3[0]) ** 2 + (std_Q3[1] * mean_Q3[1]) ** 2
std_ζ /= mean_Q3[0] ** 2 + mean_Q3[1] ** 2
std_ζ = np.sqrt(std_ζ)

std_η = (std_Q3[1] * mean_Q3[0]) ** 2 + (std_Q3[0] * mean_Q3[1]) ** 2
std_η /= (mean_Q3[0] ** 2 + mean_Q3[1] ** 2) ** 2
std_η = (4 / np.pi) * np.sqrt(std_η)

print("Q3 statistics")
print(f"\tpopulation = {100 * int_Q3 / (int_Q4 + int_Q3)}%")
print("\tmean\n\t\tζ:\t{0}\n\t\tη:\t{1}\n\t\tiso:\t{2}".format(*mean_ζη_Q3, mean_Q3[2]))
print(
    "\tstandard deviation\n\t\tζ:\t{0}\n\t\tη:\t{1}\n\t\tiso:\t{2}".format(
        std_ζ, std_η, std_Q3[2]
    )
)
```

Out:

```
Q3 statistics
        population = 88.1094614611386%
        mean
                ζ:      80.57444417273476 ppm
                η:      0.16050122245905704
                iso:    -88.9254977266946 ppm
        standard deviation
                ζ:      8.099744401149495 ppm
                η:      0.10528068664136866
                iso:    4.4050990063512145 ppm
```

Result cross-verification

The reported value for the Qn-species distribution from Baltisberger *et. al.*[?] is listed below and is consistent with the above result.

Species	Yield	Isotropic chemical shift, δ_{iso}	Shielding anisotropy, ζ_σ:	Shielding asymmetry, η_σ:
Q4	11.0 ± 0.3 %	-98.0 ± 5.64 ppm	0 ppm (fixed)	0 (fixed)
Q3	89 ± 0.1 %	-89.5 ± 4.65 ppm	80.7 ppm with a 6.7 ppm Gaussian broadening	0 (fixed)

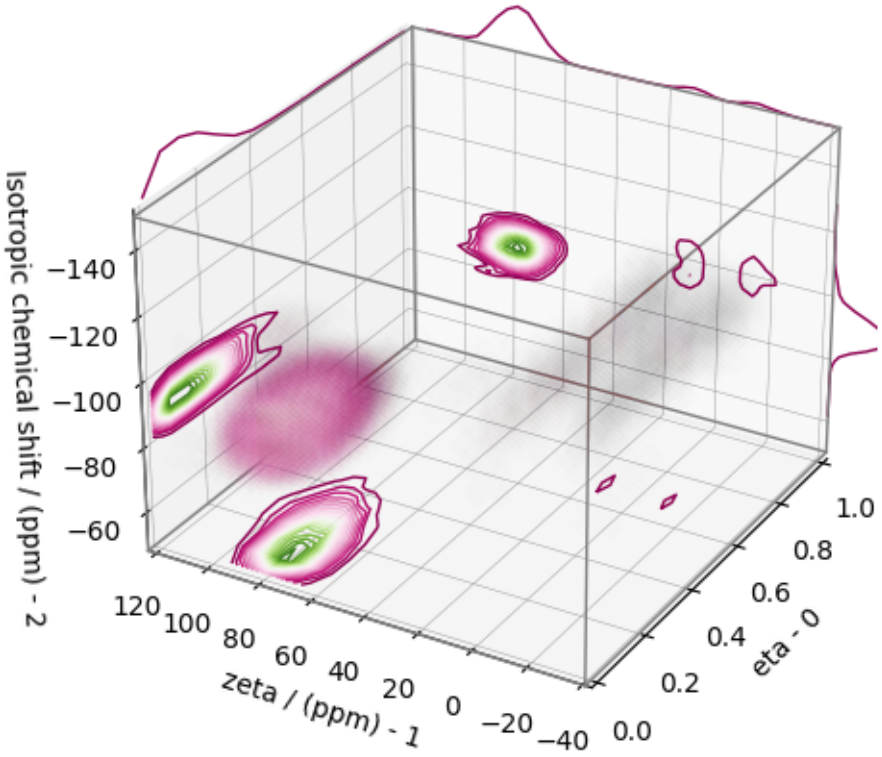
Convert the 3D tensor distribution in Haeberlen parameters

You may re-bin the 3D tensor parameter distribution from a $\rho(\delta_{iso}, x, y)$ distribution to $\rho(\delta_{iso}, \zeta_\sigma, \eta_\sigma)$ distribution as follows.

```
# Create the zeta and eta dimensions,, as shown below.
zeta = cp.as_dimension(np.arange(40) * 4 - 40, unit="ppm", label="zeta")
eta = cp.as_dimension(np.arange(16) / 15, label="eta")

# Use the `to_Haeberlen_grid` function to convert the tensor parameter distribution.
fsol_Hae = to_Haeberlen_grid(f_sol, zeta, eta)
```

The 3D plot

```
plt.figure(figsize=(5, 4.4))
ax = plt.subplot(projection="3d")
plot_3d(ax, fsol_Hae, x_lim=[0, 1], y_lim=[-40, 120], z_lim=[-50, -150], alpha=0.4)
plt.tight_layout()
plt.show()
```



References

Total running time of the script: (0 minutes 52.512 seconds)

2D MAF data of Na2O.4.7SiO2 glass

The following example illustrates an application of the statistical learning method applied in determining the distribution of the nuclear shielding tensor parameters from a 2D magic-angle flipping (MAF) spectrum. In this example, we use the 2D MAF spectrum[1] of $Na_2O \cdot 4.7SiO_2$ glass.

Before getting started

Import all relevant packages.

```
import csdmpy as cp
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import cm
from csdmpy import statistics as stats

from mrinversion.kernel.nmr import ShieldingPALineshape
```

(continues on next page)

[1] Baltisberger, J. H., Florian, P., Keeler, E. G., Phyo, P. A., Sanders, K. J., Grandinetti, P. J.. Modifier cation effects on 29Si nuclear shielding anisotropies in silicate glasses, J. Magn. Reson., 268, (2016), 95 106. doi:10.1016/j.jmr.2016.05.003.

(continued from previous page)

```
from mrinversion.kernel.utils import x_y_to_zeta_eta
from mrinversion.linear_model import SmoothLasso, TSVDCompression
from mrinversion.utils import plot_3d, to_Haeberlen_grid
```

Setup for the matplotlib figures.

```
# function for plotting 2D dataset
def plot2D(csdm_object, **kwargs):
    plt.figure(figsize=(4.5, 3.5))
    ax = plt.subplot(projection="csdm")
    ax.imshow(csdm_object, cmap="gist_ncar_r", aspect="auto", **kwargs)
    ax.invert_xaxis()
    ax.invert_yaxis()
    plt.tight_layout()
    plt.show()
```

Dataset setup

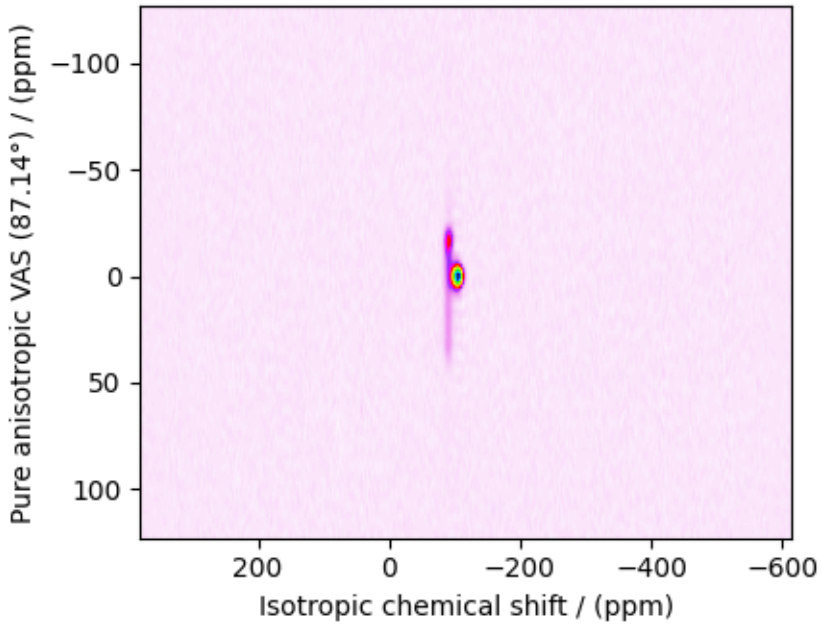
Import the dataset

Load the dataset. Here, we import the dataset as the CSDM data-object.

```
# The 2D MAF dataset in csdm format
filename = "https://zenodo.org/record/3964531/files/Na2O-4_74SiO2-MAF.csdf"
data_object = cp.load(filename)

# For inversion, we only interest ourselves with the real part of the complex dataset.
data_object = data_object.real

# We will also convert the coordinates of both dimensions from Hz to ppm.
_ = [item.to("ppm", "nmr_frequency_ratio") for item in data_object.dimensions]
```

Here, the variable `data_object` is a CSDM object that holds the real part of the 2D MAF dataset. The plot of the 2D MAF dataset is

```
plot2D(data_object)
```



There are two dimensions in this dataset. The dimension at index 0, the horizontal dimension in the figure, is the isotropic chemical shift dimension, while the dimension at index 1 is the pure anisotropic dimension. The number of coordinates along the respective dimensions is

```
print(data_object.shape)
```

Out:

```
(320, 128)
```

with 320 points along the isotropic chemical shift dimension (index 0) and 128 points along the anisotropic dimension (index 1).

Prepping the data for inversion

Step-1: Data Alignment

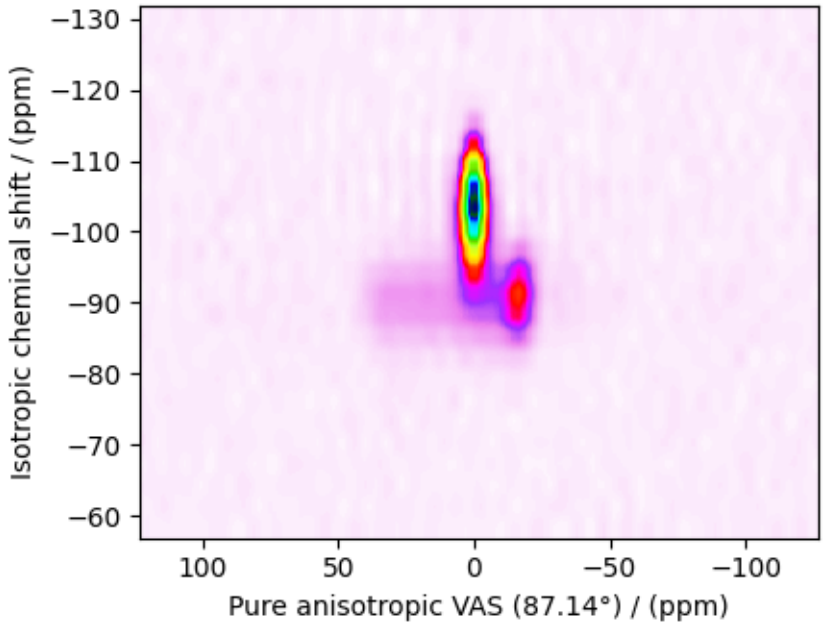
When using the csdm objects with the `mrinversion` package, the dimension at index 0 must be the dimension undergoing the linear inversion. In this example, we plan to invert the pure anisotropic shielding line-shape. In the `data_object`, however, the anisotropic dimension is at index 1. Transpose the dimensions as follows,

```
data_object = data_object.T
```

Step-2: Optimization

Also notice, the signal from the 2D MAF dataset occupies a small fraction of the two-dimensional frequency grid. For optimum performance, truncate the dataset to the relevant region before proceeding. Use the appropriate array indexing/slicing to select the signal region.

```
data_object_truncated = data_object[:, 155:180]
plot2D(data_object_truncated)
```



Linear Inversion setup

Dimension setup

Anisotropic-dimension: The dimension of the dataset that holds the pure anisotropic frequency contributions. In `mrinversion`, this must always be the dimension at index 0 of the data object.

```
anisotropic_dimension = data_object_truncated.dimensions[0]
```

x-y dimensions: The two inverse dimensions corresponding to the *x* and *y*-axis of the *x*-*y* grid.

```
inverse_dimensions = [
    cp.LinearDimension(count=25, increment="400 Hz", label="x"),  # the `x`-dimension.
    cp.LinearDimension(count=25, increment="400 Hz", label="y"),  # the `y`-dimension.
]
```

Generating the kernel

For MAF datasets, the line-shape kernel corresponds to the pure nuclear shielding anisotropy line-shapes. Use the *ShieldingPALineshape* (page 17) class to generate a shielding line-shape kernel.

```
lineshape = ShieldingPALineshape(
    anisotropic_dimension=anisotropic_dimension,
    inverse_dimension=inverse_dimensions,
    channel="29Si",
    magnetic_flux_density="9.4 T",
    rotor_angle="87.14°",
    rotor_frequency="14 kHz",
    number_of_sidebands=4,
)
```

Here, `lineshape` is an instance of the *ShieldingPALineshape* (page 17) class. The required arguments of this class are the *anisotropic_dimension*, *inverse_dimension*, and *channel*. We have already defined the first two arguments in the previous sub-section. The value of the *channel* argument is the nucleus observed in the MAF experiment. In this example, this value is '29Si'. The remaining arguments, such as the *magnetic_flux_density*, *rotor_angle*, and *rotor_frequency*, are set to match the conditions under which the 2D MAF spectrum was acquired. Note for this particular MAF measurement, the rotor angle was set to $87.14°$ for the anisotropic dimension, not the usual $90°$. The value of the *number_of_sidebands* argument is the number of sidebands calculated for each line-shape within the kernel. Unless, you have a lot of spinning sidebands in your MAF dataset, four sidebands should be enough.

Once the ShieldingPALineshape instance is created, use the *kernel()* (page 18) method of the instance to generate the MAF line-shape kernel.

```
K = lineshape.kernel(supersampling=1)
print(K.shape)
```

Out:

```
(128, 625)
```

The kernel `K` is a NumPy array of shape (128, 625), where the axes with 128 and 625 points are the anisotropic dimension and the features (x-y coordinates) corresponding to the 25×25 *x-y* grid, respectively.

Data Compression

Data compression is optional but recommended. It may reduce the size of the inverse problem and, thus, further computation time.

```
new_system = TSVDCompression(K, data_object_truncated)
compressed_K = new_system.compressed_K
compressed_s = new_system.compressed_s

print(f"truncation_index = {new_system.truncation_index}")
```

Out:

```
compression factor = 1.471264367816092
truncation_index = 87
```

Solving the inverse problem

Smooth LASSO cross-validation

Solve the smooth-lasso problem. Ordinarily, one should use the statistical learning method to solve the inverse problem over a range of α and λ values and then determine the best nuclear shielding tensor parameter distribution for the given 2D MAF dataset. Considering the limited build time for the documentation, we skip this step and evaluate the distribution at pre-optimized α and λ values. The optimum values are $\alpha = 2.07 \times 10^{-7}$ and $\lambda = 7.85 \times 10^{-6}$. The following commented code was used in determining the optimum α and λ values.

```
# from mrinversion.linear_model import SmoothLassoCV

# # setup the pre-defined range of alpha and lambda values
# lambdas = 10 ** (-4 - 3 * (np.arange(20) / 19))
# alphas = 10 ** (-4 - 3 * (np.arange(20) / 19))
```

(continues on next page)

(continued from previous page)

```
# # setup the smooth lasso cross-validation class
# s_lasso = SmoothLassoCV(
#     alphas=alphas,  # A numpy array of alpha values.
#     lambdas=lambdas,  # A numpy array of lambda values.
#     sigma=0.003,  # The standard deviation of noise from the MAF data.
#     folds=10,  # The number of folds in n-folds cross-validation.
#     inverse_dimension=inverse_dimensions,  # previously defined inverse dimensions.
#     verbose=1,  # If non-zero, prints the progress as the computation proceeds.
#     max_iterations=20000,  # maximum number of allowed iterations.
# )

# # run fit using the compressed kernel and compressed data.
# s_lasso.fit(compressed_K, compressed_s)

# # the optimum hyper-parameters, alpha and lambda, from the cross-validation.
# print(s_lasso.hyperparameters)
# # {'alpha': 2.06913808111479e-07, 'lambda': 7.847599703514622e-06}

# # the solution
# f_sol = s_lasso.f

# # the cross-validation error curve
# CV_metric = s_lasso.cross_validation_curve
```

If you use the above `SmoothLassoCV` method, skip the following code-block. The following code-block evaluates the smooth-lasso solution at the pre-optimized hyperparameters.

```
# Setup the smooth lasso class
s_lasso = SmoothLasso(
    alpha=2.07e-7, lambda1=7.85e-6, inverse_dimension=inverse_dimensions
)
# run the fit method on the compressed kernel and compressed data.
s_lasso.fit(K=compressed_K, s=compressed_s)
```

The optimum solution

The *f* (page 20) attribute of the instance holds the solution,

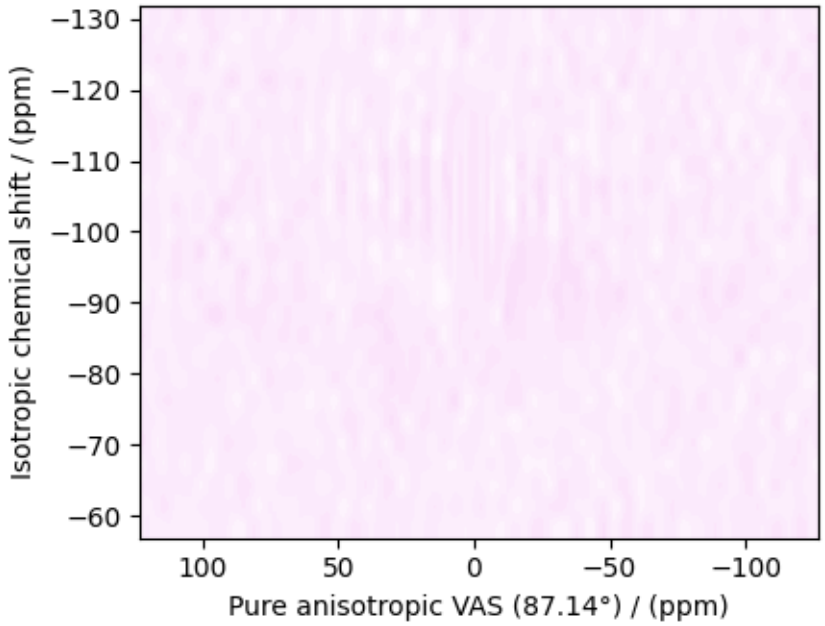
```
f_sol = s_lasso.f  # f_sol is a CSDM object.
```

where `f_sol` is the optimum solution.

The fit residuals

To calculate the residuals between the data and predicted data(fit), use the *residuals()* (page 20) method, as follows,

```
residuals = s_lasso.residuals(K=K, s=data_object_truncated)
# residuals is a CSDM object.

# The plot of the residuals.
plot2D(residuals, vmax=data_object_truncated.max(), vmin=data_object_truncated.min())
```



The mean and standard deviation of the residuals are

```
residuals.mean(), residuals.std()
```

Out:

```
(<Quantity 0.00082075>, <Quantity 0.00353748>)
```

Saving the solution

To serialize the solution to a file, use the *save()* method of the CSDM object, for example,

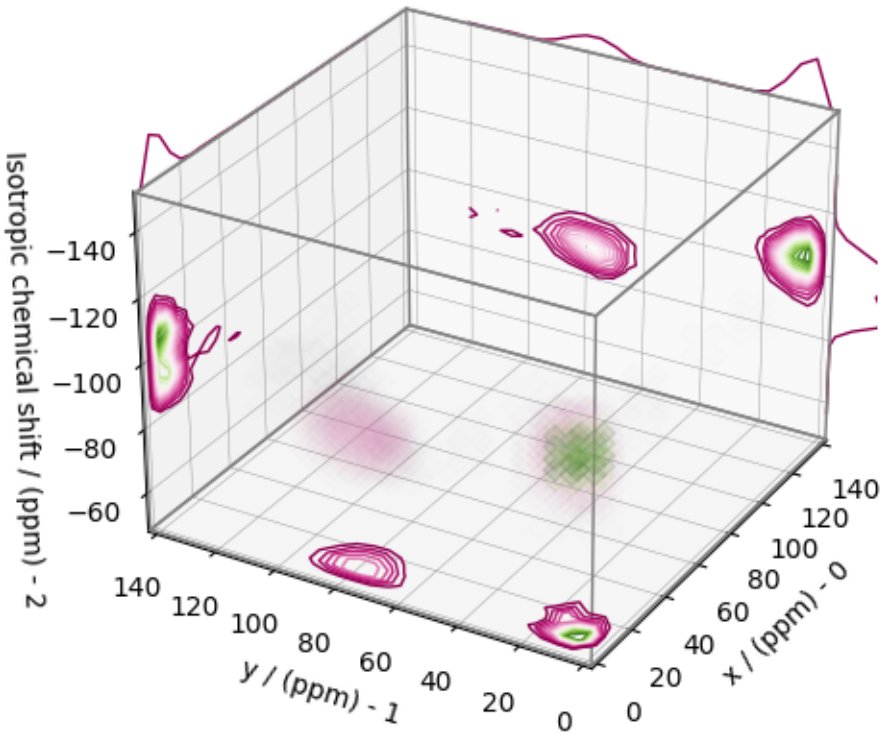
```
f_sol.save("Na2O.4.7SiO2_inverse.csdf")  # save the solution
residuals.save("Na2O.4.7SiO2_residue.csdf")  # save the residuals
```

Data Visualization

At this point, we have solved the inverse problem and obtained an optimum distribution of the nuclear shielding tensor parameters from the 2D MAF dataset. You may use any data visualization and interpretation tool of choice for further analysis. In the following sections, we provide minimal visualization and analysis to complete the case study.

Visualizing the 3D solution

```
# Normalize the solution
f_sol /= f_sol.max()

# Convert the coordinates of the solution, `f_sol`, from Hz to ppm.
[item.to("ppm", "nmr_frequency_ratio") for item in f_sol.dimensions]

# The 3D plot of the solution
plt.figure(figsize=(5, 4.4))
ax = plt.subplot(projection="3d")
plot_3d(ax, f_sol, x_lim=[0, 140], y_lim=[0, 140], z_lim=[-50, -150])
plt.tight_layout()
plt.show()
```



From the 3D plot, we observe two distinct regions: one for the Q^4 sites and another for the Q^3 sites. Select the respective regions by using the appropriate array indexing,

```
Q4_region = f_sol[0:8, 0:8, 3:18]
Q4_region.description = "Q4 region"
```

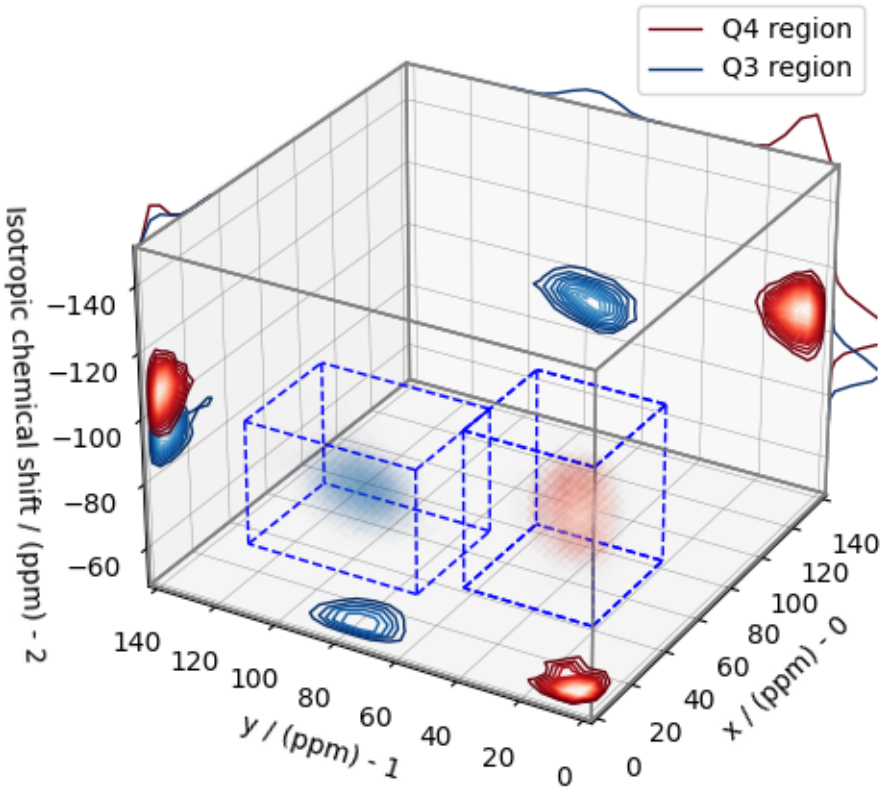
(continues on next page)

(continued from previous page)

```
Q3_region = f_sol[0:8, 11:22, 8:20]
Q3_region.description = "Q3 region"
```

The plot of the respective regions is shown below.

```
# Calculate the normalization factor for the 2D contours and 1D projections from the
# original solution, `f_sol`. Use this normalization factor to scale the intensities
# from the sub-regions.
max_2d = [
    f_sol.sum(axis=0).max().value,
    f_sol.sum(axis=1).max().value,
    f_sol.sum(axis=2).max().value,
]
max_1d = [
    f_sol.sum(axis=(1, 2)).max().value,
    f_sol.sum(axis=(0, 2)).max().value,
    f_sol.sum(axis=(0, 1)).max().value,
]

plt.figure(figsize=(5, 4.4))
ax = plt.subplot(projection="3d")

# plot for the Q4 region
plot_3d(
    ax,
    Q4_region,
    x_lim=[0, 140],  # the x-limit
    y_lim=[0, 140],  # the y-limit
    z_lim=[-50, -150],  # the z-limit
    max_2d=max_2d,  # normalization factors for the 2D contours projections
    max_1d=max_1d,  # normalization factors for the 1D projections
    cmap=cm.Reds_r,  # colormap
    box=True,  # draw a box around the region
)
# plot for the Q3 region
plot_3d(
    ax,
    Q3_region,
    x_lim=[0, 140],  # the x-limit
    y_lim=[0, 140],  # the y-limit
    z_lim=[-50, -150],  # the z-limit
    max_2d=max_2d,  # normalization factors for the 2D contours projections
    max_1d=max_1d,  # normalization factors for the 1D projections
    cmap=cm.Blues_r,  # colormap
    box=True,  # draw a box around the region
)
ax.legend()
plt.tight_layout()
plt.show()
```



Visualizing the isotropic projections.

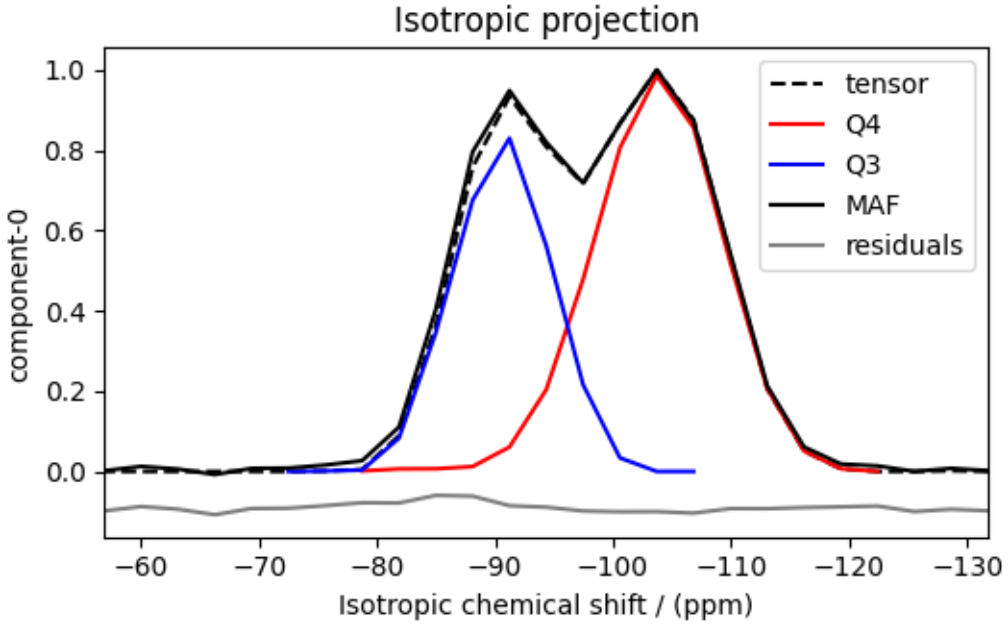
Because the Q^4 and Q^3 regions are fully resolved after the inversion, evaluating the contributions from these regions is trivial. For examples, the distribution of the isotropic chemical shifts for these regions are

```
# Isotropic chemical shift projection of the 2D MAF dataset.
data_iso = data_object_truncated.sum(axis=0)
data_iso /= data_iso.max()  # normalize the projection

# Isotropic chemical shift projection of the tensor distribution dataset.
f_sol_iso = f_sol.sum(axis=(0, 1))

# Isotropic chemical shift projection of the tensor distribution for the Q4 region.
Q4_region_iso = Q4_region.sum(axis=(0, 1))

# Isotropic chemical shift projection of the tensor distribution for the Q3 region.
Q3_region_iso = Q3_region.sum(axis=(0, 1))

# Normalize the three projections.
f_sol_iso_max = f_sol_iso.max()
f_sol_iso /= f_sol_iso_max
Q4_region_iso /= f_sol_iso_max
Q3_region_iso /= f_sol_iso_max

# The plot of the different projections.
plt.figure(figsize=(5.5, 3.5))
ax = plt.subplot(projection="csdm")
```

(continues on next page)

(continued from previous page)

```
ax.plot(f_sol_iso, "--k", label="tensor")
ax.plot(Q4_region_iso, "r", label="Q4")
ax.plot(Q3_region_iso, "b", label="Q3")
ax.plot(data_iso, "-k", label="MAF")
ax.plot(data_iso - f_sol_iso - 0.1, "gray", label="residuals")
ax.set_title("Isotropic projection")
ax.invert_xaxis()
plt.legend()
plt.tight_layout()
plt.show()
```



Analysis

For the analysis, we use the statistics module of the csdmpy package. Following is the moment analysis of the 3D volumes for both the Q^4 and Q^3 regions up to the second moment.

```
int_Q4 = stats.integral(Q4_region)  # volume of the Q4 distribution
mean_Q4 = stats.mean(Q4_region)  # mean of the Q4 distribution
std_Q4 = stats.std(Q4_region)  # standard deviation of the Q4 distribution

int_Q3 = stats.integral(Q3_region)  # volume of the Q3 distribution
mean_Q3 = stats.mean(Q3_region)  # mean of the Q3 distribution
std_Q3 = stats.std(Q3_region)  # standard deviation of the Q3 distribution

print("Q4 statistics")
print(f"\tpopulation = {100 * int_Q4 / (int_Q4 + int_Q3)}%")
print("\tmean\n\t\tx:\t{0}\n\t\ty:\t{1}\n\t\tiso:\t{2}".format(*mean_Q4))
print("\tstandard deviation\n\t\tx:\t{0}\n\t\ty:\t{1}\n\t\tiso:\t{2}".format(*std_Q4))

print("Q3 statistics")
print(f"\tpopulation = {100 * int_Q3 / (int_Q4 + int_Q3)}%")
```

(continues on next page)

(continued from previous page)

```
print("\tmean\n\t\tx:\t{0}\n\t\ty:\t{1}\n\t\tiso:\t{2}".format(*mean_Q3))
print("\tstandard deviation\n\t\tx:\t{0}\n\t\ty:\t{1}\n\t\tiso:\t{2}".format(*std_Q3))
```

Out:

```
Q4 statistics
        population = 60.43132319089674%
        mean
                x:      8.658843513252181 ppm
                y:      9.077692333047839 ppm
                iso:    -103.68408995442384 ppm
        standard deviation
                x:      4.1472630351549045 ppm
                y:      4.30922313805979 ppm
                iso:    5.326098312546948 ppm
Q3 statistics
        population = 39.568676809103266%
        mean
                x:      10.338299897193506 ppm
                y:      78.899061589533 ppm
                iso:    -90.60754829994842 ppm
        standard deviation
                x:      6.055814163126547 ppm
                y:      7.724993691781673 ppm
                iso:    3.9967063660071793 ppm
```

The statistics shown above are according to the respective dimensions, that is, the x, y, and the isotropic chemical shifts. To convert the x and y statistics to commonly used ζ_σ and η_σ statistics, use the *x_y_to_zeta_eta()* (page 23) function.

```
mean_ζη_Q3 = x_y_to_zeta_eta(*mean_Q3[0:2])

# error propagation for calculating the standard deviation
std_ζ = (std_Q3[0] * mean_Q3[0]) ** 2 + (std_Q3[1] * mean_Q3[1]) ** 2
std_ζ /= mean_Q3[0] ** 2 + mean_Q3[1] ** 2
std_ζ = np.sqrt(std_ζ)

std_η = (std_Q3[1] * mean_Q3[0]) ** 2 + (std_Q3[0] * mean_Q3[1]) ** 2
std_η /= (mean_Q3[0] ** 2 + mean_Q3[1] ** 2) ** 2
std_η = (4 / np.pi) * np.sqrt(std_η)

print("Q3 statistics")
print(f"\tpopulation = {100 * int_Q3 / (int_Q4 + int_Q3)}%")
print("\tmean\n\t\tζ:\t{0}\n\t\tη:\t{1}\n\t\tiso:\t{2}".format(*mean_ζη_Q3, mean_Q3[2]))
print(
    "\tstandard deviation\n\t\tζ:\t{0}\n\t\tη:\t{1}\n\t\tiso:\t{2}".format(
        std_ζ, std_η, std_Q3[2]
    )
)
```

Out:

```
Q3 statistics
        population = 39.568676809103266%
        mean
                ζ:      79.57350290437913 ppm
                η:      0.16588999155496023
                iso:    -90.60754829994842 ppm
```

(continues on next page)

(continued from previous page)

```
        standard deviation
                ζ:      7.699821614649238 ppm
                η:      0.09740946211900393
                iso:    3.9967063660071793 ppm
```

Result cross-verification

The reported value for the Qn-species distribution from Baltisberger *et. al.*[?] is listed below and is consistent with the above result.

Species	Yield	Isotropic chemical shift, δ_{iso}	Shielding anisotropy, ζ_σ:	Shielding asymmetry, η_σ:
Q4	57.8 ± 0.1 %	-103.7 ± 5.31 ppm	0 ppm (fixed)	0 (fixed)
Q3	42.2 ± 0.2 %	-90.5 ± 4.29 ppm	79.8 ppm with a 7.1 ppm Gaussian broadening	0 (fixed)

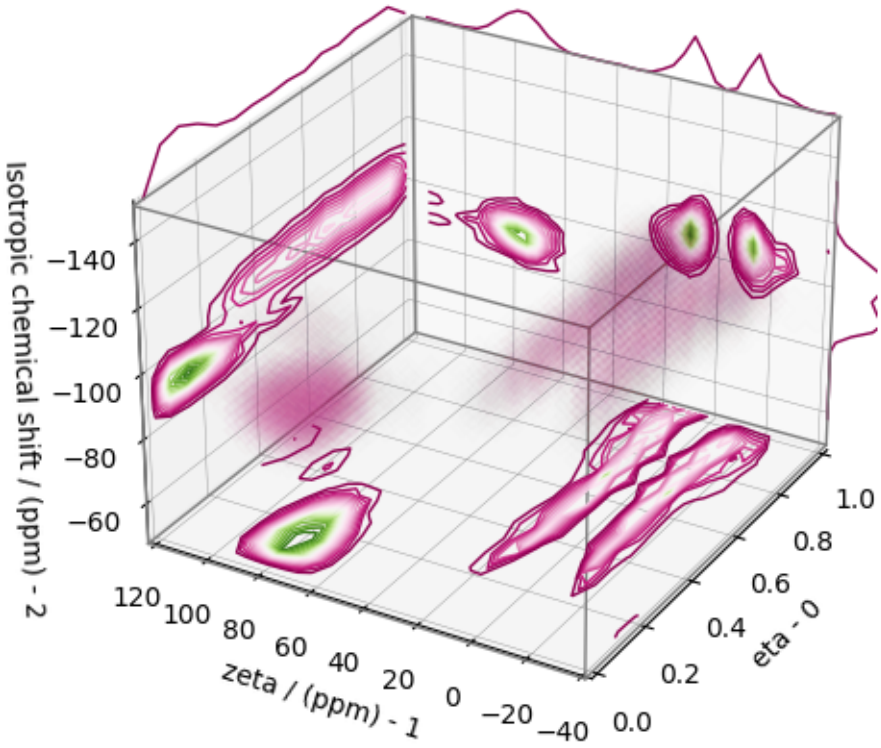
Convert the 3D tensor distribution in Haeberlen parameters

You may re-bin the 3D tensor parameter distribution from a $\rho(\delta_{\text{iso}}, x, y)$ distribution to $\rho(\delta_{\text{iso}}, \zeta_\sigma, \eta_\sigma)$ distribution as follows.

```
# Create the zeta and eta dimensions, as shown below.
zeta = cp.as_dimension(np.arange(40) * 4 - 40, unit="ppm", label="zeta")
eta = cp.as_dimension(np.arange(16) / 15, label="eta")

# Use the `to_Haeberlen_grid` function to convert the tensor parameter distribution.
fsol_Hae = to_Haeberlen_grid(f_sol, zeta, eta)
```

The 3D plot

```
plt.figure(figsize=(5, 4.4))
ax = plt.subplot(projection="3d")
plot_3d(ax, fsol_Hae, x_lim=[0, 1], y_lim=[-40, 120], z_lim=[-50, -150], alpha=0.4)
plt.tight_layout()
plt.show()
```



References

Total running time of the script: (0 minutes 6.109 seconds)

2D MAF data of Cs2O.4.72SiO2 glass

The following example illustrates an application of the statistical learning method applied in determining the distribution of the nuclear shielding tensor parameters from a 2D magic-angle flipping (MAF) spectrum. In this example, we use the 2D MAF spectrum[1] of $Cs_2O \cdot 4.72SiO_2$ glass.

Before getting started

Import all relevant packages.

```
import csdmpy as cp
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import cm
from csdmpy import statistics as stats

from mrinversion.kernel.nmr import ShieldingPALineshape
```

(continues on next page)

[1] Alvarez, D. J., Sanders, K. J., Phyo, P. A., Baltisberger, J. H., Grandinetti, P. J. Cluster formation of network-modifier cations in cesium silicate glasses, J. Chem. Phys. 148, 094502, (2018). doi:10.1063/1.5020986

(continued from previous page)

```
from mrinversion.kernel.utils import x_y_to_zeta_eta
from mrinversion.linear_model import SmoothLasso, TSVDCompression
from mrinversion.utils import plot_3d, to_Haeberlen_grid
```

Setup for the matplotlib figures.

```
# function for plotting 2D dataset
def plot2D(csdm_object, **kwargs):
    plt.figure(figsize=(4.5, 3.5))
    ax = plt.subplot(projection="csdm")
    ax.imshow(csdm_object, cmap="gist_ncar_r", aspect="auto", **kwargs)
    ax.invert_xaxis()
    ax.invert_yaxis()
    plt.tight_layout()
    plt.show()
```

Dataset setup

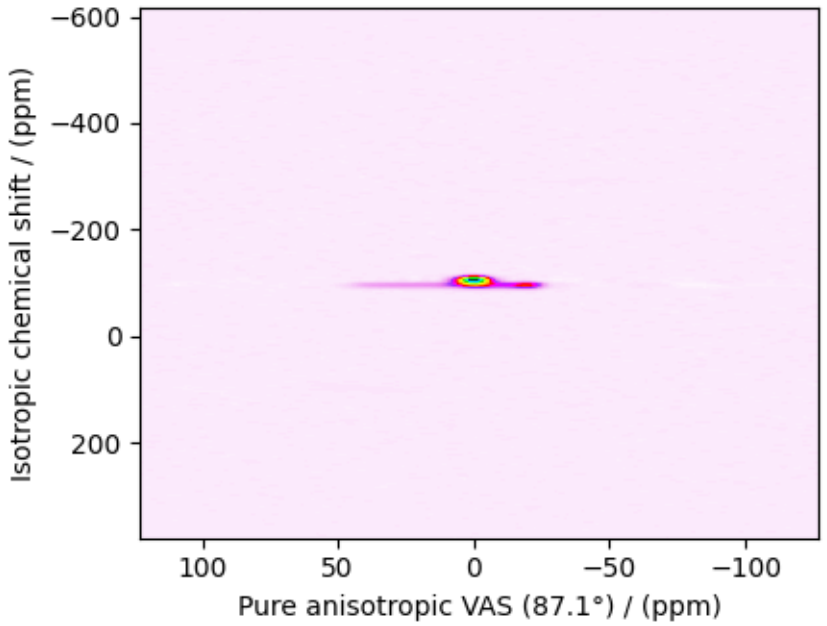
Import the dataset

Load the dataset. Here, we import the dataset as the CSDM data-object.

```
# The 2D MAF dataset in csdm format
filename = "https://zenodo.org/record/3964531/files/Cs2O-4_72SiO2-MAF.csdf"
data_object = cp.load(filename)

# For inversion, we only interest ourselves with the real part of the complex dataset.
data_object = data_object.real

# We will also convert the coordinates of both dimensions from Hz to ppm.
_ = [item.to("ppm", "nmr_frequency_ratio") for item in data_object.dimensions]
```

Here, the variable `data_object` is a CSDM object that holds the real part of the 2D MAF dataset. The plot of the 2D MAF dataset is

```
plot2D(data_object)
```



There are two dimensions in this dataset. The dimension at index 0, the horizontal dimension in the figure, is the pure anisotropic dimension, while the dimension at index 1 is the isotropic chemical shift dimension.

Prepping the data for inversion

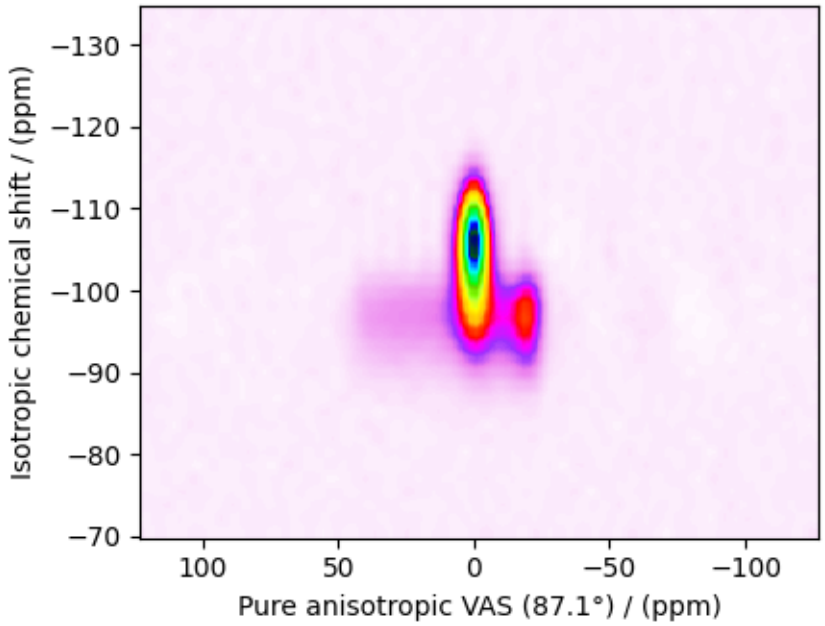
Step-1: Data Alignment

When using the csdm objects with the `mrinversion` package, the dimension at index 0 must be the dimension undergoing the linear inversion. In this example, we plan to invert the pure anisotropic shielding line-shape. In the `data_object`, the anisotropic dimension is already at index 0 and, therefore, no further action is required.

Step-2: Optimization

Also notice, the signal from the 2D MAF dataset occupies a small fraction of the two-dimensional frequency grid. For optimum performance, truncate the dataset to the relevant region before proceeding. Use the appropriate array indexing/slicing to select the signal region.

```
data_object_truncated = data_object[:, 290:330]
plot2D(data_object_truncated)
```



Linear Inversion setup

Dimension setup

Anisotropic-dimension: The dimension of the dataset that holds the pure anisotropic frequency contributions. In `mrinversion`, this must always be the dimension at index 0 of the data object.

```
anisotropic_dimension = data_object_truncated.dimensions[0]
```

x-y dimensions: The two inverse dimensions corresponding to the *x* and *y*-axis of the *x*-*y* grid.

```
inverse_dimensions = [
    cp.LinearDimension(count=25, increment="450 Hz", label="x"),  # the `x`-dimension
    cp.LinearDimension(count=25, increment="450 Hz", label="y"),  # the `y`-dimension
]
```

Generating the kernel

For MAF datasets, the line-shape kernel corresponds to the pure nuclear shielding anisotropy line-shapes. Use the *ShieldingPALineshape* (page 17) class to generate a shielding line-shape kernel.

```
lineshape = ShieldingPALineshape(
    anisotropic_dimension=anisotropic_dimension,
    inverse_dimension=inverse_dimensions,
    channel="29Si",
    magnetic_flux_density="9.4 T",
    rotor_angle="87.14°",
    rotor_frequency="14 kHz",
    number_of_sidebands=4,
)
```

Here, `lineshape` is an instance of the *ShieldingPALineshape* (page 17) class. The required arguments of this class are the *anisotropic_dimension*, *inverse_dimension*, and *channel*. We have already defined the first two arguments in the previous sub-section. The value of the *channel* argument is the nucleus observed in the MAF experiment. In this example, this value is '29Si'. The remaining arguments, such as the *magnetic_flux_density*, *rotor_angle*, and *rotor_frequency*, are set to match the conditions under which the 2D MAF spectrum was acquired. Note for this particular MAF measurement, the rotor angle was set to $87.14°$ for the anisotropic dimension, not the usual $90°$. The value of the *number_of_sidebands* argument is the number of sidebands calculated for each line-shape within the kernel. Unless, you have a lot of spinning sidebands in your MAF dataset, four sidebands should be enough.

Once the ShieldingPALineshape instance is created, use the *kernel()* (page 18) method of the instance to generate the MAF line-shape kernel.

```
K = lineshape.kernel(supersampling=1)
print(K.shape)
```

Out:

```
(128, 625)
```

The kernel `K` is a NumPy array of shape (128, 625), where the axes with 128 and 625 points are the anisotropic dimension and the features (x-y coordinates) corresponding to the 25×25 *x-y* grid, respectively.

Data Compression

Data compression is optional but recommended. It may reduce the size of the inverse problem and, thus, further computation time.

```
new_system = TSVDCompression(K, data_object_truncated)
compressed_K = new_system.compressed_K
compressed_s = new_system.compressed_s

print(f"truncation_index = {new_system.truncation_index}")
```

Out:

```
compression factor = 1.3195876288659794
truncation_index = 97
```

Solving the inverse problem

Smooth LASSO cross-validation

Solve the smooth-lasso problem. Ordinarily, one should use the statistical learning method to solve the inverse problem over a range of α and λ values and then determine the best nuclear shielding tensor parameter distribution for the given 2D MAF dataset. Considering the limited build time for the documentation, we skip this step and evaluate the distribution at pre-optimized α and λ values. The optimum values are $\alpha = 5.62 \times 10^{-7}$ and $\lambda = 3.16 \times 10^{-6}$. The following commented code was used in determining the optimum α and λ values.

```
# from mrinversion.linear_model import SmoothLassoCV

# # setup the pre-defined range of alpha and lambda values
# lambdas = 10 ** (-4 - 3 * (np.arange(20) / 19))
# alphas = 10 ** (-4.5 - 3 * (np.arange(20) / 19))
```

(continues on next page)

(continued from previous page)

```
# # setup the smooth lasso cross-validation class
# s_lasso = SmoothLassoCV(
#     alphas=alphas,  # A numpy array of alpha values.
#     lambdas=lambdas,  # A numpy array of lambda values.
#     sigma=0.002,  # The standard deviation of noise from the MAF data.
#     folds=10,  # The number of folds in n-folds cross-validation.
#     inverse_dimension=inverse_dimensions,  # previously defined inverse dimensions.
#     verbose=1,  # If non-zero, prints the progress as the computation proceeds.
# )

# # run fit using the compressed kernel and compressed data.
# s_lasso.fit(compressed_K, compressed_s)

# # the optimum hyper-parameters, alpha and lambda, from the cross-validation.
# print(s_lasso.hyperparameters)

# # the solution
# f_sol = s_lasso.f

# # the cross-validation error curve
# CV_metric = s_lasso.cross_validation_curve
```

If you use the above `SmoothLassoCV` method, skip the following code-block. The following code-block evaluates the smooth-lasso solution at the pre-optimized hyperparameters.

```
# Setup the smooth lasso class
s_lasso = SmoothLasso(
    alpha=8.34e-7, lambda1=6.16e-7, inverse_dimension=inverse_dimensions
)
# run the fit method on the compressed kernel and compressed data.
s_lasso.fit(K=compressed_K, s=compressed_s)
```

The optimum solution

The *f* (page 20) attribute of the instance holds the solution,

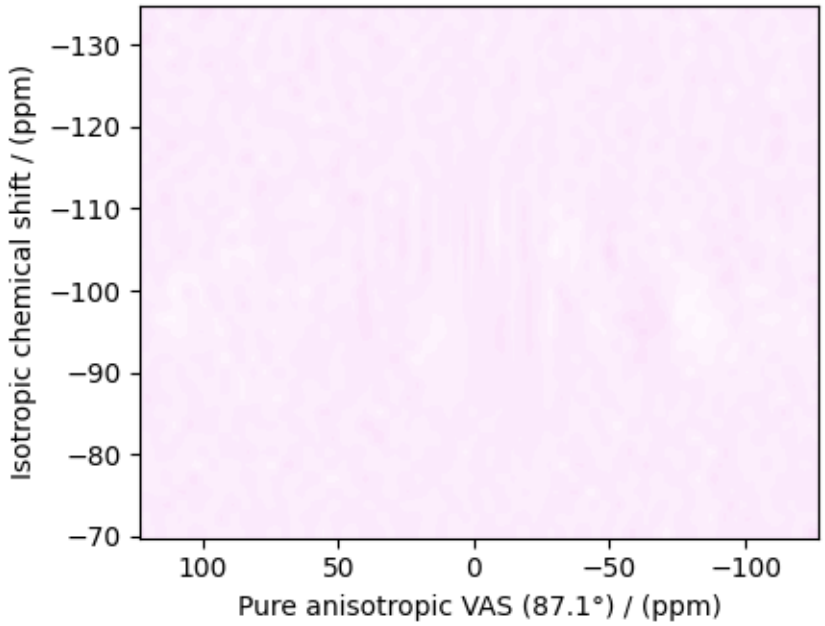
```
f_sol = s_lasso.f  # f_sol is a CSDM object.
```

where `f_sol` is the optimum solution.

The fit residuals

To calculate the residuals between the data and predicted data(fit), use the *residuals()* (page 20) method, as follows,

```
residuals = s_lasso.residuals(K=K, s=data_object_truncated)
# residuals is a CSDM object.

# The plot of the residuals.
plot2D(residuals, vmax=data_object_truncated.max(), vmin=data_object_truncated.min())
```



The standard deviation of the residuals is

```
residuals.std()
```

Out:

```
<Quantity 0.00241182>
```

Saving the solution

To serialize the solution to a file, use the *save()* method of the CSDM object, for example,

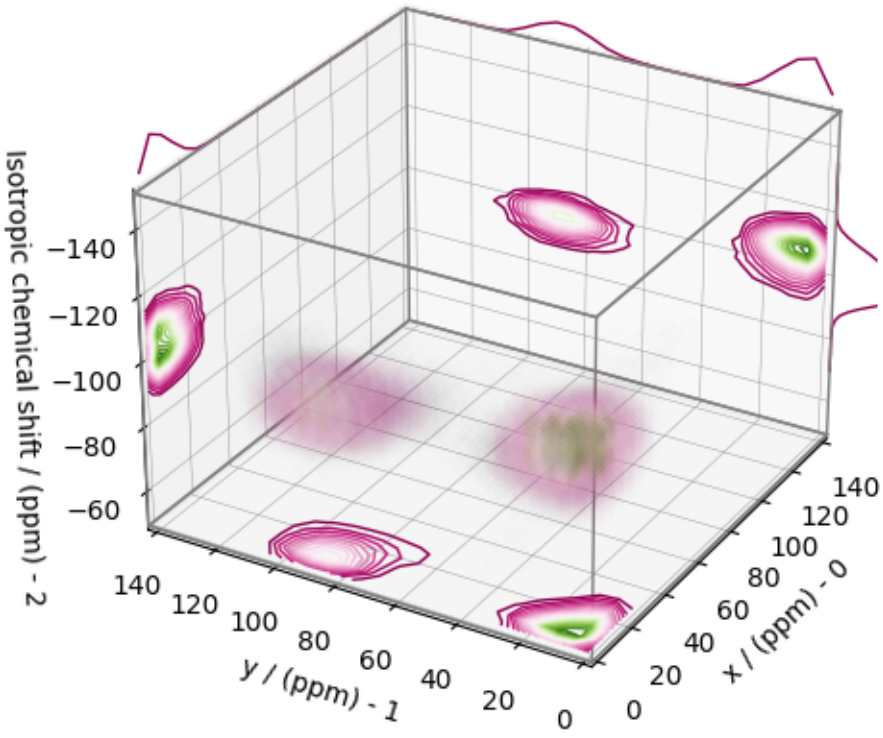
```
f_sol.save("Cs2O.4.72SiO2_inverse.csdf")  # save the solution
residuals.save("Cs2O.4.72SiO2_residue.csdf")  # save the residuals
```

Data Visualization

At this point, we have solved the inverse problem and obtained an optimum distribution of the nuclear shielding tensor parameters from the 2D MAF dataset. You may use any data visualization and interpretation tool of choice for further analysis. In the following sections, we provide minimal visualization and analysis to complete the case study.

Visualizing the 3D solution

```
# Normalize the solution
f_sol /= f_sol.max()

# Convert the coordinates of the solution, `f_sol`, from Hz to ppm.
[item.to("ppm", "nmr_frequency_ratio") for item in f_sol.dimensions]

# The 3D plot of the solution
plt.figure(figsize=(5, 4.4))
ax = plt.subplot(projection="3d")
plot_3d(ax, f_sol, x_lim=[0, 140], y_lim=[0, 140], z_lim=[-50, -150])
plt.tight_layout()
plt.show()
```



From the 3D plot, we observe two distinct regions: one for the Q^4 sites and another for the Q^3 sites. Select the respective regions by using the appropriate array indexing,

```
Q4_region = f_sol[0:7, 0:7, 8:34]
Q4_region.description = "Q4 region"
```

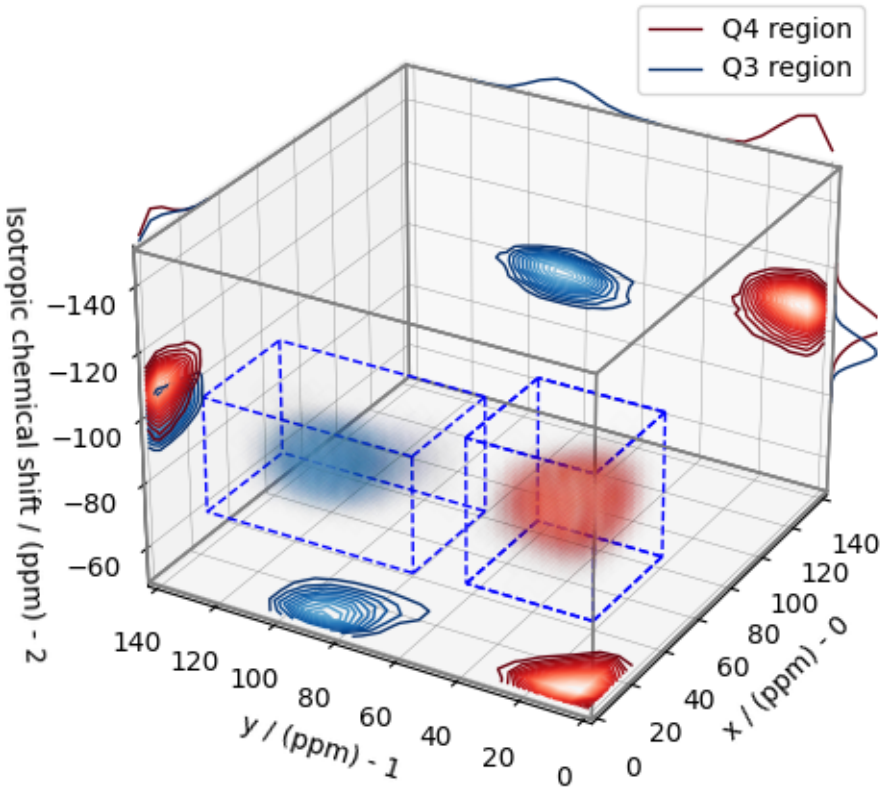
(continues on next page)

(continued from previous page)

```
Q3_region = f_sol[0:7, 10:22, 14:35]
Q3_region.description = "Q3 region"
```

The plot of the respective regions is shown below.

```
# Calculate the normalization factor for the 2D contours and 1D projections from the
# original solution, `f_sol`. Use this normalization factor to scale the intensities
# from the sub-regions.
max_2d = [
    f_sol.sum(axis=0).max().value,
    f_sol.sum(axis=1).max().value,
    f_sol.sum(axis=2).max().value,
]
max_1d = [
    f_sol.sum(axis=(1, 2)).max().value,
    f_sol.sum(axis=(0, 2)).max().value,
    f_sol.sum(axis=(0, 1)).max().value,
]

plt.figure(figsize=(5, 4.4))
ax = plt.subplot(projection="3d")

# plot for the Q4 region
plot_3d(
    ax,
    Q4_region,
    x_lim=[0, 140],  # the x-limit
    y_lim=[0, 140],  # the y-limit
    z_lim=[-50, -150],  # the z-limit
    max_2d=max_2d,  # normalization factors for the 2D contours projections
    max_1d=max_1d,  # normalization factors for the 1D projections
    cmap=cm.Reds_r,  # colormap
    box=True,  # draw a box around the region
)
# plot for the Q3 region
plot_3d(
    ax,
    Q3_region,
    x_lim=[0, 140],  # the x-limit
    y_lim=[0, 140],  # the y-limit
    z_lim=[-50, -150],  # the z-limit
    max_2d=max_2d,  # normalization factors for the 2D contours projections
    max_1d=max_1d,  # normalization factors for the 1D projections
    cmap=cm.Blues_r,  # colormap
    box=True,  # draw a box around the region
)
ax.legend()
plt.tight_layout()
plt.show()
```



Visualizing the isotropic projections.

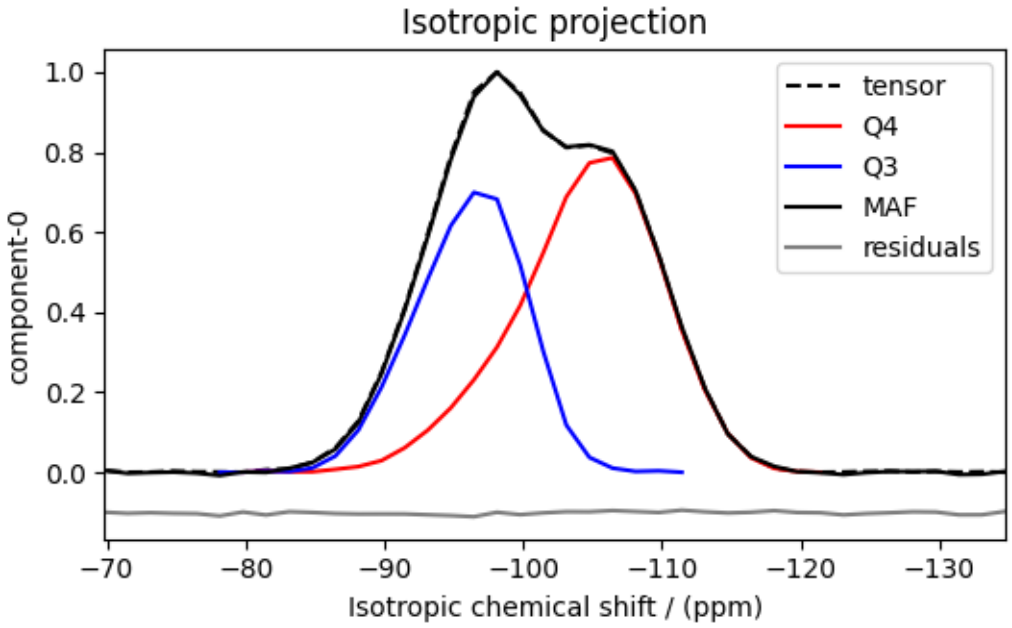
Because the Q^4 and Q^3 regions are fully resolved after the inversion, evaluating the contributions from these regions is trivial. For examples, the distribution of the isotropic chemical shifts for these regions are

```
# Isotropic chemical shift projection of the 2D MAF dataset.
data_iso = data_object_truncated.sum(axis=0)
data_iso /= data_iso.max()  # normalize the projection

# Isotropic chemical shift projection of the tensor distribution dataset.
f_sol_iso = f_sol.sum(axis=(0, 1))

# Isotropic chemical shift projection of the tensor distribution for the Q4 region.
Q4_region_iso = Q4_region.sum(axis=(0, 1))

# Isotropic chemical shift projection of the tensor distribution for the Q3 region.
Q3_region_iso = Q3_region.sum(axis=(0, 1))

# Normalize the three projections.
f_sol_iso_max = f_sol_iso.max()
f_sol_iso /= f_sol_iso_max
Q4_region_iso /= f_sol_iso_max
Q3_region_iso /= f_sol_iso_max

# The plot of the different projections.
plt.figure(figsize=(5.5, 3.5))
ax = plt.subplot(projection="csdm")
```

(continues on next page)

(continued from previous page)

```
ax.plot(f_sol_iso, "--k", label="tensor")
ax.plot(Q4_region_iso, "r", label="Q4")
ax.plot(Q3_region_iso, "b", label="Q3")
ax.plot(data_iso, "-k", label="MAF")
ax.plot(data_iso - f_sol_iso - 0.1, "gray", label="residuals")
ax.set_title("Isotropic projection")
ax.invert_xaxis()
plt.legend()
plt.tight_layout()
plt.show()
```



Notice the skew in the isotropic chemical shift distribution for the Q^4 regions, which is expected.

Analysis

For the analysis, we use the statistics module of the csdmpy package. Following is the moment analysis of the 3D volumes for both the Q^4 and Q^3 regions up to the second moment.

```
int_Q4 = stats.integral(Q4_region)  # volume of the Q4 distribution
mean_Q4 = stats.mean(Q4_region)  # mean of the Q4 distribution
std_Q4 = stats.std(Q4_region)  # standard deviation of the Q4 distribution

int_Q3 = stats.integral(Q3_region)  # volume of the Q3 distribution
mean_Q3 = stats.mean(Q3_region)  # mean of the Q3 distribution
std_Q3 = stats.std(Q3_region)  # standard deviation of the Q3 distribution

print("Q4 statistics")
print(f"\tpopulation = {100 * int_Q4 / (int_Q4 + int_Q3)}%")
print("\tmean\n\t\tx:\t{0}\n\t\ty:\t{1}\n\t\tiso:\t{2}".format(*mean_Q4))
print("\tstandard deviation\n\t\tx:\t{0}\n\t\ty:\t{1}\n\t\tiso:\t{2}".format(*std_Q4))

print("Q3 statistics")
```

(continues on next page)

(continued from previous page)

```
print(f"\tpopulation = {100 * int_Q3 / (int_Q4 + int_Q3)}%")
print("\tmean\n\t\tx:\t{0}\n\t\ty:\t{1}\n\t\tiso:\t{2}".format(*mean_Q3))
print("\tstandard deviation\n\t\tx:\t{0}\n\t\ty:\t{1}\n\t\tiso:\t{2}".format(*std_Q3))
```

Out:

```
Q4 statistics
        population = 59.18635811123303%
        mean
                x:      9.631407520484899 ppm
                y:      10.897557949280454 ppm
                iso:    -104.49449535069108 ppm
        standard deviation
                x:      6.503702326192696 ppm
                y:      6.939954070488857 ppm
                iso:    5.3565480066660065 ppm
Q3 statistics
        population = 40.81364188876696%
        mean
                x:      11.18860038757134 ppm
                y:      87.66280074191147 ppm
                iso:    -96.07798180021605 ppm
        standard deviation
                x:      7.226397705641644 ppm
                y:      10.0498432059067 ppm
                iso:    3.87926866862933 ppm
```

The statistics shown above are according to the respective dimensions, that is, the x, y, and the isotropic chemical shifts. To convert the x and y statistics to commonly used ζ_σ and η_σ statistics, use the *x_y_to_zeta_eta()* (page 23) function.

```
mean_ζη_Q3 = x_y_to_zeta_eta(*mean_Q3[0:2])

# error propagation for calculating the standard deviation
std_ζ = (std_Q3[0] * mean_Q3[0]) ** 2 + (std_Q3[1] * mean_Q3[1]) ** 2
std_ζ /= mean_Q3[0] ** 2 + mean_Q3[1] ** 2
std_ζ = np.sqrt(std_ζ)

std_η = (std_Q3[1] * mean_Q3[0]) ** 2 + (std_Q3[0] * mean_Q3[1]) ** 2
std_η /= (mean_Q3[0] ** 2 + mean_Q3[1] ** 2) ** 2
std_η = (4 / np.pi) * np.sqrt(std_η)

print("Q3 statistics")
print(f"\tpopulation = {100 * int_Q3 / (int_Q4 + int_Q3)}%")
print("\tmean\n\t\tζ:\t{0}\n\t\tη:\t{1}\n\t\tiso:\t{2}".format(*mean_ζη_Q3, mean_Q3[2]))
print(
    "\tstandard deviation\n\t\tζ:\t{0}\n\t\tη:\t{1}\n\t\tiso:\t{2}".format(
        std_ζ, std_η, std_Q3[2]
    )
)
```

Out:

```
Q3 statistics
        population = 40.81364188876696%
        mean
                ζ:      88.37392948459878 ppm
                η:      0.16163254241378167
```

(continues on next page)

(continued from previous page)

```
                iso:    -96.07798180021605 ppm
        standard deviation
                ζ:      10.010868259844177 ppm
                η:      0.10489019772052138
                iso:    3.87926866862933 ppm
```

Result cross-verification

The reported value for the Qn-species distribution from Baltisberger *et. al.*[7] is listed below and is consistent with the above result.

Species	Yield	Isotropic chemical shift, δ_{iso}	Shielding anisotropy, ζ_σ:	Shielding asymmetry, η_σ:
Q4	57.7 ± 0.4 %	-104.7 ± 5.2 ppm	0 ppm (fixed)	0 (fixed)
Q3	42.3 ± 0.4 %	-96.1 ± 4.0 ppm	89.0 ppm	0 (fixed)

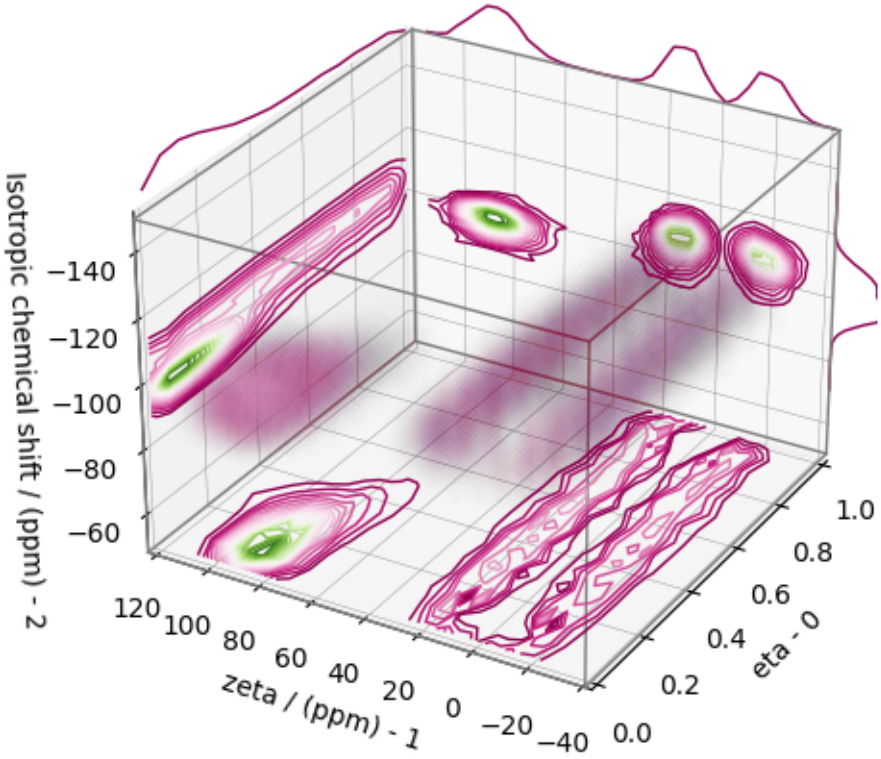
Convert the 3D tensor distribution in Haeberlen parameters

You may re-bin the 3D tensor parameter distribution from a $\rho(\delta_{\text{iso}}, x, y)$ distribution to $\rho(\delta_{\text{iso}}, \zeta_\sigma, \eta_\sigma)$ distribution as follows.

```
# Create the zeta and eta dimensions,, as shown below.
zeta = cp.as_dimension(np.arange(40) * 4 - 40, unit="ppm", label="zeta")
eta = cp.as_dimension(np.arange(16) / 15, label="eta")

# Use the `to_Haeberlen_grid` function to convert the tensor parameter distribution.
fsol_Hae = to_Haeberlen_grid(f_sol, zeta, eta)
```

The 3D plot

```
plt.figure(figsize=(5, 4.4))
ax = plt.subplot(projection="3d")
plot_3d(ax, fsol_Hae, x_lim=[0, 1], y_lim=[-40, 120], z_lim=[-50, -150], alpha=0.2)
plt.tight_layout()
plt.show()
```



References

Total running time of the script: (0 minutes 6.699 seconds)

2D MAF data of 2Na2O.3SiO2 glass

The following example illustrates an application of the statistical learning method applied in determining the distribution of the nuclear shielding tensor parameters from a 2D magic-angle flipping (MAF) spectrum. In this example, we use the 2D MAF spectrum[1] of $2Na_2O \cdot 3SiO_2$ glass.

Before getting started

Import all relevant packages.

```
import csdmpy as cp
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import cm

from mrinversion.kernel.nmr import ShieldingPALineshape
```

(continues on next page)

[1] Zhang, P., Dunlap, C., Florian, P., Grandinetti, P. J., Farnan, I., Stebbins , J. F. Silicon site distributions in an alkali silicate glass derived by two-dimensional 29Si nuclear magnetic resonance, J. Non. Cryst. Solids, 204, (1996), 294300. doi:10.1016/S0022-3093(96)00601-1.

(continued from previous page)

```
from mrinversion.linear_model import SmoothLasso, TSVDCompression
from mrinversion.utils import plot_3d, to_Haeberlen_grid
```

Setup for the matplotlib figures.

```
def plot2D(csdm_object, **kwargs):
    plt.figure(figsize=(4.5, 3.5))
    ax = plt.subplot(projection="csdm")
    ax.imshow(csdm_object, cmap="gist_ncar_r", aspect="auto", **kwargs)
    ax.invert_xaxis()
    ax.invert_yaxis()
    plt.tight_layout()
    plt.show()
```

Dataset setup

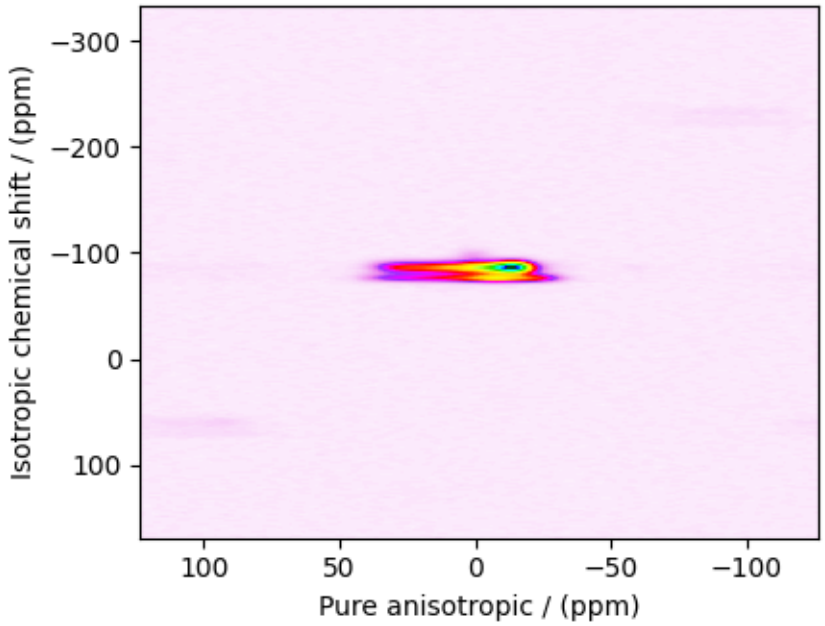
Import the dataset

Load the dataset. Here, we import the dataset as the CSDM data-object.

```
# The 2D MAF dataset in csdm format
filename = "https://osu.box.com/shared/static/k405dsptwe1p43x8mfi1wc1geywrypzc.csdf"
data_object = cp.load(filename)

# For inversion, we only interest ourselves with the real part of the complex dataset.
data_object = data_object.real

# We will also convert the coordinates of both dimensions from Hz to ppm.
_ = [item.to("ppm", "nmr_frequency_ratio") for item in data_object.dimensions]
```

Here, the variable `data_object` is a CSDM object that holds the real part of the 2D MAF dataset. The plot of the 2D MAF dataset is

```
plot2D(data_object)
```



There are two dimensions in this dataset. The dimension at index 0 is the pure anisotropic dimension, while the dimension at index 1 is the isotropic chemical shift dimension.

Prepping the data for inversion

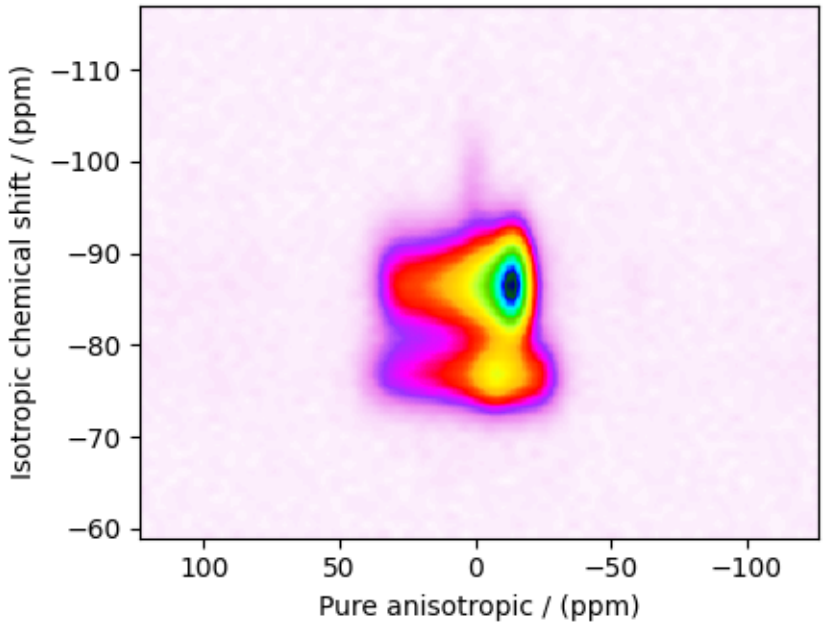
Step-1: Data Alignment

When using the csdm objects with the `mrinversion` package, the dimension at index 0 must be the dimension undergoing the linear inversion. In this example, we plan to invert the pure anisotropic shielding line-shape. In the `data_object`, the anisotropic dimension is already at index 0 and, therefore, no further action is required.

Step-2: Optimization

Also notice, the signal from the 2D MAF dataset occupies a small fraction of the two-dimensional frequency grid. For optimum performance, truncate the dataset to the relevant region before proceeding. Use the appropriate array indexing/slicing to select the signal region.

```
data_object_truncated = data_object[:, 220:280]
plot2D(data_object_truncated)
```



Linear Inversion setup

Dimension setup

Anisotropic-dimension: The dimension of the dataset that holds the pure anisotropic frequency contributions. In `mrinversion`, this must always be the dimension at index 0 of the data object.

```
anisotropic_dimension = data_object_truncated.dimensions[0]
```

x-y dimensions: The two inverse dimensions corresponding to the *x* and *y*-axis of the *x*-*y* grid.

```
inverse_dimensions = [
    cp.LinearDimension(count=25, increment="500 Hz", label="x"),  # the `x`-dimension.
    cp.LinearDimension(count=25, increment="500 Hz", label="y"),  # the `y`-dimension.
]
```

Generating the kernel

For MAF datasets, the line-shape kernel corresponds to the pure nuclear shielding anisotropy line-shapes. Use the *ShieldingPALineshape* (page 17) class to generate a shielding line-shape kernel.

```
lineshape = ShieldingPALineshape(
    anisotropic_dimension=anisotropic_dimension,
    inverse_dimension=inverse_dimensions,
    channel="29Si",
    magnetic_flux_density="9.4 T",
    rotor_angle="90°",
    rotor_frequency="12 kHz",
    number_of_sidebands=4,
)
```

Here, `lineshape` is an instance of the *ShieldingPALineshape* (page 17) class. The required arguments of this class are the *anisotropic_dimension*, *inverse_dimension*, and *channel*. We have already defined the first two arguments in the previous sub-section. The value of the *channel* argument is the nucleus observed in the MAF experiment. In this example, this value is '29Si'. The remaining arguments, such as the *magnetic_flux_density*, *rotor_angle*, and *rotor_frequency*, are set to match the conditions under which the 2D MAF spectrum was acquired. The value of the *number_of_sidebands* argument is the number of sidebands calculated for each line-shape within the kernel. Unless, you have a lot of spinning sidebands in your MAF dataset, four sidebands should be enough.

Once the ShieldingPALineshape instance is created, use the *kernel()* (page 18) method of the instance to generate the MAF line-shape kernel.

```
K = lineshape.kernel(supersampling=1)
print(K.shape)
```

Out:

```
(128, 625)
```

The kernel `K` is a NumPy array of shape (128, 625), where the axes with 128 and 625 points are the anisotropic dimension and the features (x-y coordinates) corresponding to the 25×25 *x*-*y* grid, respectively.

Data Compression

Data compression is optional but recommended. It may reduce the size of the inverse problem and, thus, further computation time.

```
new_system = TSVDCompression(K, data_object_truncated)
compressed_K = new_system.compressed_K
compressed_s = new_system.compressed_s

print(f"truncation_index = {new_system.truncation_index}")
```

Out:

```
compression factor = 1.1851851851851851
truncation_index = 108
```

Solving the inverse problem

Smooth LASSO cross-validation

Solve the smooth-lasso problem. Ordinarily, one should use the statistical learning method to solve the inverse problem over a range of α and λ values and then determine the best nuclear shielding tensor parameter distribution for the given 2D MAF dataset. Considering the limited build time for the documentation, we skip this step and evaluate the distribution at pre-optimized α and λ values. The optimum values are $\alpha = 2.2 \times 10^{-8}$ and $\lambda = 1.27 \times 10^{-6}$. The following commented code was used in determining the optimum α and λ values.

```
# from mrinversion.linear_model import SmoothLassoCV
# import numpy as np

# # setup the pre-defined range of alpha and lambda values
# lambdas = 10 ** (-4 - 3 * (np.arange(20) / 19))
# alphas = 10 ** (-4.5 - 5 * (np.arange(20) / 19))
```

(continues on next page)

(continued from previous page)

```
# # setup the smooth lasso cross-validation class
# s_lasso = SmoothLassoCV(
#     alphas=alphas,  # A numpy array of alpha values.
#     lambdas=lambdas,  # A numpy array of lambda values.
#     sigma=0.003,  # The standard deviation of noise from the MAF data.
#     folds=10,  # The number of folds in n-folds cross-validation.
#     inverse_dimension=inverse_dimensions,  # previously defined inverse dimensions.
#     verbose=1,  # If non-zero, prints the progress as the computation proceeds.
# )

# # run fit using the compressed kernel and compressed data.
# s_lasso.fit(compressed_K, compressed_s)

# # the optimum hyper-parameters, alpha and lambda, from the cross-validation.
# print(s_lasso.hyperparameters)
# # {'alpha': 2.198392648862289e-08, 'lambda': 1.2742749857031348e-06}

# # the solution
# f_sol = s_lasso.f

# # the cross-validation error curve
# CV_metric = s_lasso.cross_validation_curve
```

If you use the above `SmoothLassoCV` method, skip the following code-block.

```
# Setup the smooth lasso class
s_lasso = SmoothLasso(
    alpha=2.198e-8, lambda1=1.27e-6, inverse_dimension=inverse_dimensions
)
# run the fit method on the compressed kernel and compressed data.
s_lasso.fit(K=compressed_K, s=compressed_s)
```

The optimum solution

The *f* (page 20) attribute of the instance holds the solution,

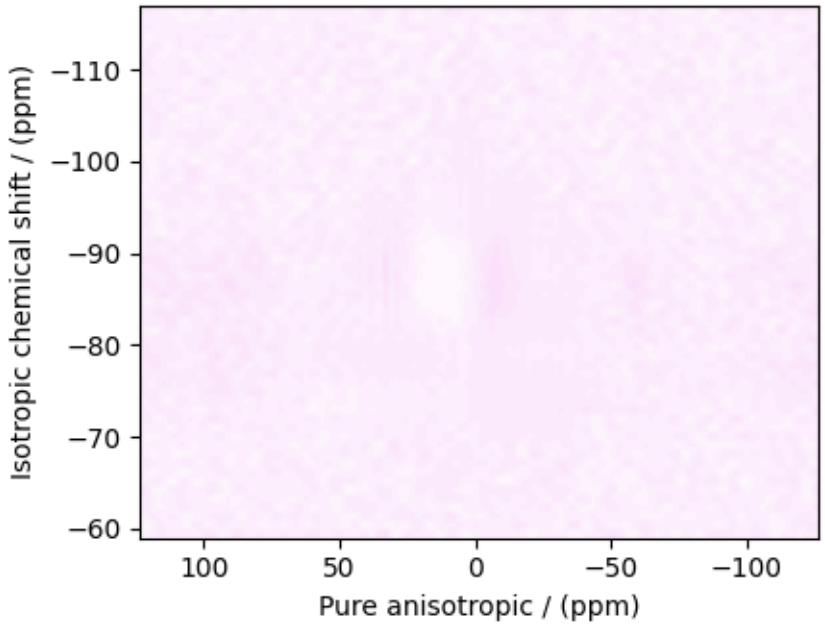
```
f_sol = s_lasso.f  # f_sol is a CSDM object.
```

where `f_sol` is the optimum solution.

The fit residuals

To calculate the residuals between the data and predicted data(fit), use the *residuals()* (page 20) method, as follows,

```
residuals = s_lasso.residuals(K=K, s=data_object_truncated)
# residuals is a CSDM object.

# The plot of the residuals.
plot2D(residuals, vmax=data_object_truncated.max(), vmin=data_object_truncated.min())
```



The standard deviation of the residuals is

```
residuals.std()
```

Out:

```
<Quantity 0.00347761>
```

Saving the solution

To serialize the solution to a file, use the *save()* method of the CSDM object, for example,

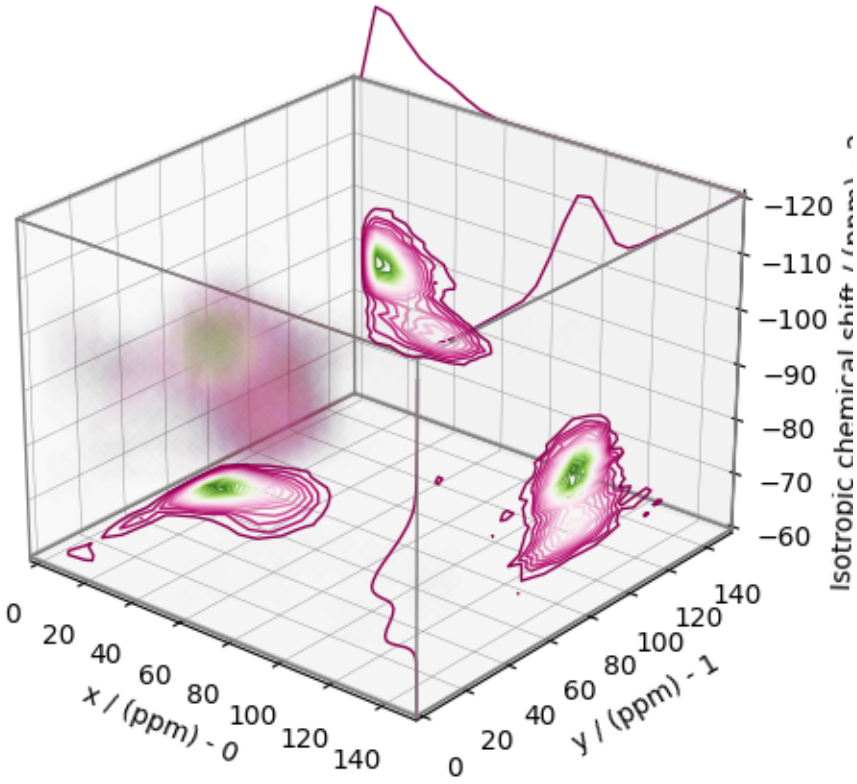
```
f_sol.save("2Na2O.3SiO2_inverse.csdf")  # save the solution
residuals.save("2Na2O.3SiO2_residue.csdf")  # save the residuals
```

Data Visualization

At this point, we have solved the inverse problem and obtained an optimum distribution of the nuclear shielding tensor parameters from the 2D MAF dataset. You may use any data visualization and interpretation tool of choice for further analysis. In the following sections, we provide minimal visualization to complete the case study.

Visualizing the 3D solution

```
# Normalize the solution
f_sol /= f_sol.max()

# Convert the coordinates of the solution, `f_sol`, from Hz to ppm.
[item.to("ppm", "nmr_frequency_ratio") for item in f_sol.dimensions]

# The 3D plot of the solution
plt.figure(figsize=(5, 4.4))
ax = plt.subplot(projection="3d")
plot_3d(ax, f_sol, elev=25, azim=-50, x_lim=[0, 150], y_lim=[0, 150], z_lim=[-60, -120])
plt.tight_layout()
plt.show()
```



From the 3D plot, we observe three distinct regions corresponding to the Q^4, Q^3, and Q^2 sites, respectively. The Q^4 sites are resolved in the 3D distribution; however, we observe partial overlapping Q^3 and Q^2 sites. The following is a naive selection of the three regions. One may also apply sophisticated classification algorithms to better quantify the Q-species.

```
Q4_region = f_sol[0:6, 0:6, 14:35] * 3
Q4_region.description = "Q4 region x 3"

Q3_region = f_sol[0:8, 7:, 20:39]
Q3_region.description = "Q3 region"

Q2_region = f_sol[:10, 6:18, 36:52]
Q2_region.description = "Q2 region"
```

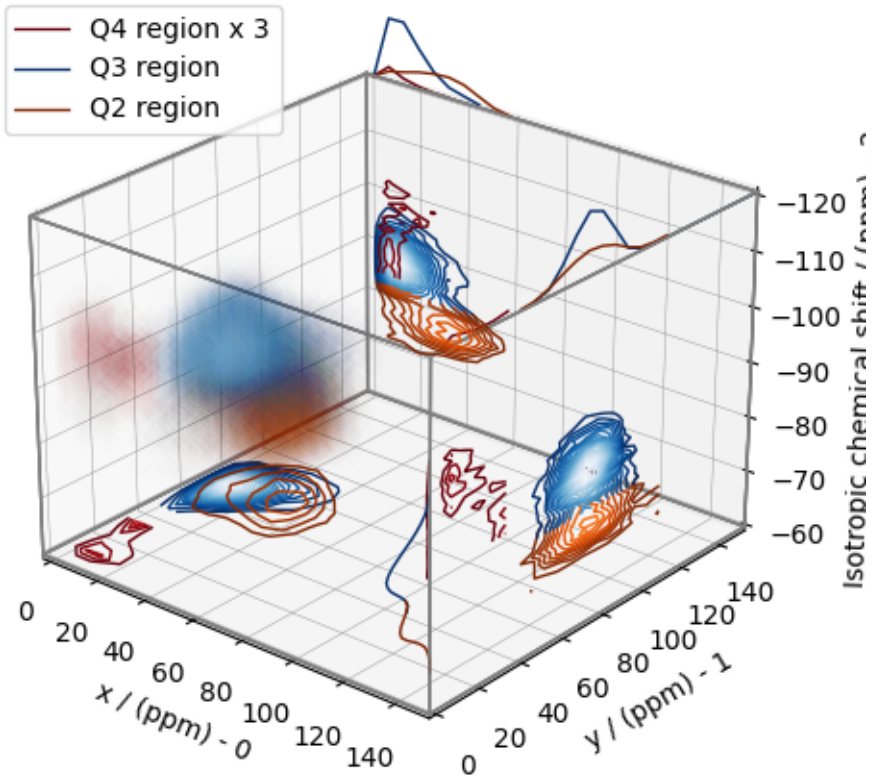
An approximate plot of the respective regions is shown below.

```
# Calculate the normalization factor for the 2D contours and 1D projections from the
# original solution, `f_sol`. Use this normalization factor to scale the intensities
# from the sub-regions.
max_2d = [
    f_sol.sum(axis=0).max().value,
    f_sol.sum(axis=1).max().value,
    f_sol.sum(axis=2).max().value,
]
max_1d = [
    f_sol.sum(axis=(1, 2)).max().value,
    f_sol.sum(axis=(0, 2)).max().value,
    f_sol.sum(axis=(0, 1)).max().value,
]

plt.figure(figsize=(5, 4.4))
ax = plt.subplot(projection="3d")

# plot for the Q4 region
plot_3d(
    ax,
    Q4_region,
    x_lim=[0, 150],  # the x-limit
    y_lim=[0, 150],  # the y-limit
    z_lim=[-60, -120],  # the z-limit
    max_2d=max_2d,  # normalization factors for the 2D contours projections
    max_1d=max_1d,  # normalization factors for the 1D projections
    cmap=cm.Reds_r,  # colormap
)
# plot for the Q3 region
plot_3d(
    ax,
    Q3_region,
    x_lim=[0, 150],  # the x-limit
    y_lim=[0, 150],  # the y-limit
    z_lim=[-60, -120],  # the z-limit
    max_2d=max_2d,  # normalization factors for the 2D contours projections
    max_1d=max_1d,  # normalization factors for the 1D projections
    cmap=cm.Blues_r,  # colormap
)
# plot for the Q2 region
plot_3d(
    ax,
    Q2_region,
    elev=25,  # the elevation angle in the z plane
    azim=-50,  # the azimuth angle in the x-y plane
    x_lim=[0, 150],  # the x-limit
```

(continues on next page)

(continued from previous page)

```
    y_lim=[0, 150],  # the y-limit
    z_lim=[-60, -120],  # the z-limit
    max_2d=max_2d,  # normalization factors for the 2D contours projections
    max_1d=max_1d,  # normalization factors for the 1D projections
    cmap=cm.Oranges_r,  # colormap
    box=False,  # draw a box around the region
)
ax.legend()
plt.tight_layout()
plt.show()
```



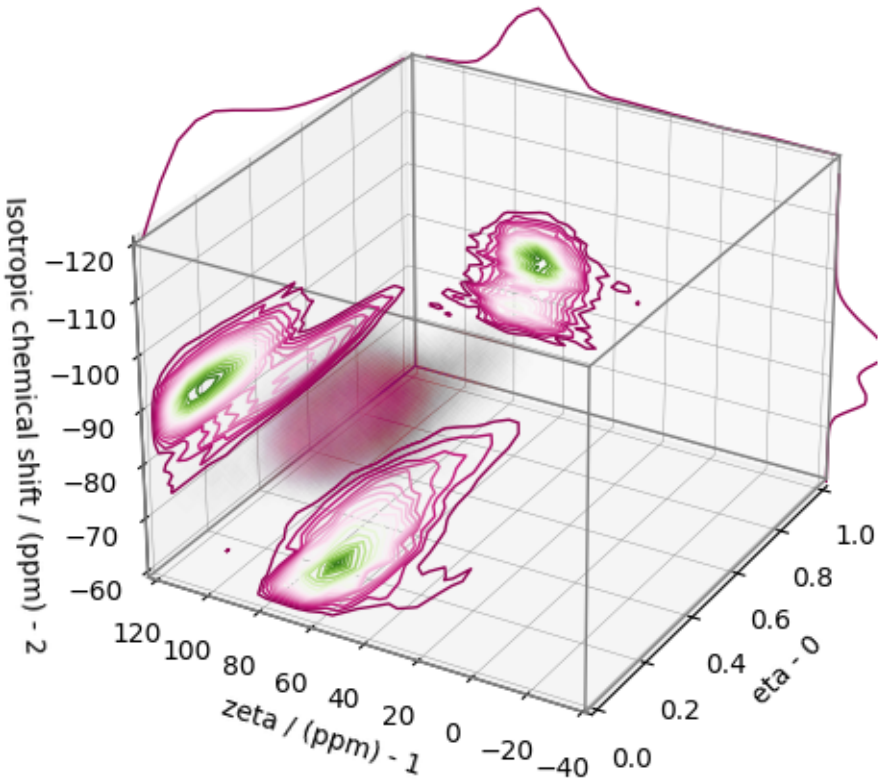
Convert the 3D tensor distribution in Haeberlen parameters

You may re-bin the 3D tensor parameter distribution from a $\rho(\delta_{\text{iso}}, x, y)$ distribution to $\rho(\delta_{\text{iso}}, \zeta_\sigma, \eta_\sigma)$ distribution as follows.

```
# Create the zeta and eta dimensions,, as shown below.
zeta = cp.as_dimension(np.arange(40) * 4 - 40, unit="ppm", label="zeta")
eta = cp.as_dimension(np.arange(16) / 15, label="eta")

# Use the `to_Haeberlen_grid` function to convert the tensor parameter distribution.
fsol_Hae = to_Haeberlen_grid(f_sol, zeta, eta)
```

The 3D plot

```
plt.figure(figsize=(5, 4.4))
ax = plt.subplot(projection="3d")
plot_3d(ax, fsol_Hae, x_lim=[0, 1], y_lim=[-40, 120], z_lim=[-60, -120], alpha=0.1)
plt.tight_layout()
plt.show()
```



References

Total running time of the script: (0 minutes 12.899 seconds)

2D MAF data of MgO.SiO2 glass

The following example illustrates an application of the statistical learning method applied in determining the distribution of the nuclear shielding tensor parameters from a 2D magic-angle flipping (MAF) spectrum. In this example, we use the 2D MAF spectrum[1] of $MgO \cdot SiO_2$ glass.

[1] Davis, M., Sanders, K. J., Grandinetti, P. J., Gaudio, S. J., Sen, S., Structural investigations of magnesium silicate glasses by 29Si magic-angle flipping NMR, J. Non. Cryst. Solids, 357, 27872795, (2011). doi:10.1016/j.jnoncrysol.2011.02.045.

Before getting started

Import all relevant packages.

```
import csdmpy as cp
import matplotlib.pyplot as plt
import numpy as np

from mrinversion.kernel.nmr import ShieldingPALineshape
from mrinversion.linear_model import SmoothLasso, TSVDCompression
from mrinversion.utils import plot_3d, to_Haeberlen_grid
```

Setup for the matplotlib figures.

```
# function for plotting 2D dataset
def plot2D(csdm_object, **kwargs):
    plt.figure(figsize=(4.5, 3.5))
    ax = plt.subplot(projection="csdm")
    ax.imshow(csdm_object, cmap="gist_ncar_r", aspect="auto", **kwargs)
    ax.invert_xaxis()
    ax.invert_yaxis()
    plt.tight_layout()
    plt.show()
```

Dataset setup

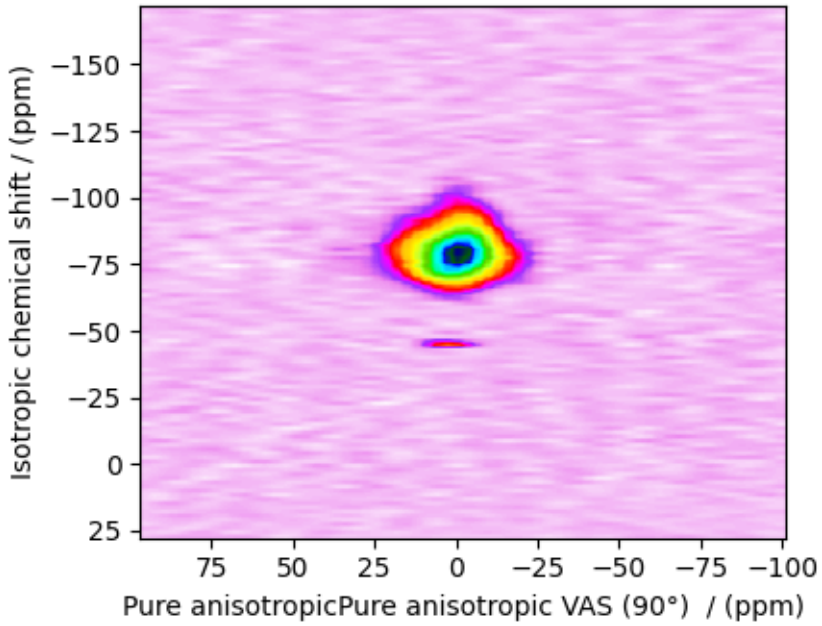
Import the dataset

Load the dataset. Here, we import the dataset as the CSDM data-object.

```
# The 2D MAF dataset in csdm format
filename = "https://zenodo.org/record/3964531/files/MgO-SiO2-MAF.csdf"
data_object = cp.load(filename)

# For inversion, we only interest ourselves with the real part of the complex dataset.
data_object = data_object.real

# We will also convert the coordinates of both dimensions from Hz to ppm.
_ = [item.to("ppm", "nmr_frequency_ratio") for item in data_object.dimensions]
```

Here, the variable `data_object` is a CSDM object that holds the real part of the 2D MAF dataset. The plot of the 2D MAF dataset is

```
plot2D(data_object)
```



There are two dimensions in this dataset. The dimension at index 0 is the pure anisotropic dimension, while the dimension at index 1 is the isotropic chemical shift dimension.

Prepping the data for inversion

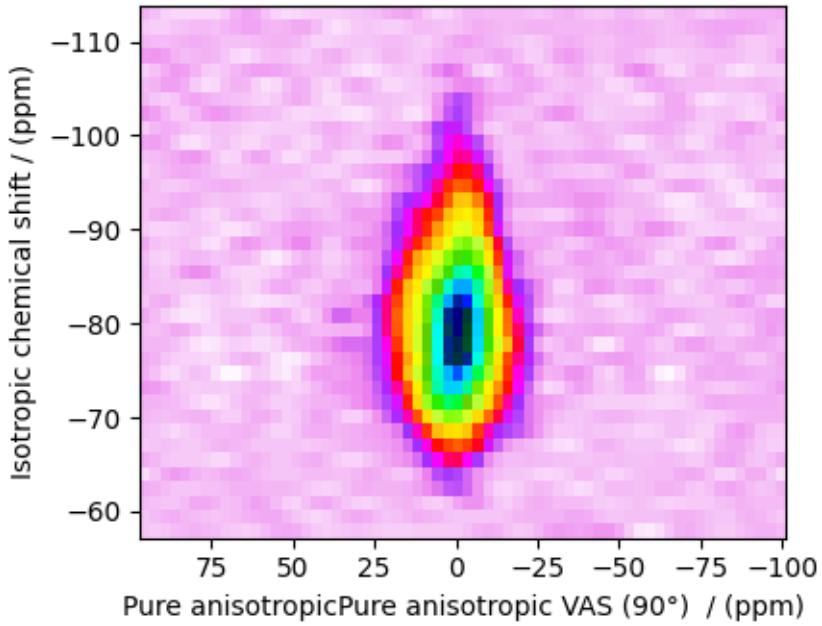
Step-1: Data Alignment

When using the csdm objects with the `mrinversion` package, the dimension at index 0 must be the dimension undergoing the linear inversion. In this example, we plan to invert the pure anisotropic shielding line-shape. In the `data_object`, the anisotropic dimension is already at index 0 and, therefore, no further action is required.

Step-2: Optimization

Also notice, the signal from the 2D MAF dataset occupies a small fraction of the two-dimensional frequency grid. For optimum performance, truncate the dataset to the relevant region before proceeding. Use the appropriate array indexing/slicing to select the signal region.

```
data_object_truncated = data_object[:, 37:74]
plot2D(data_object_truncated)
```



Linear Inversion setup

Dimension setup

Anisotropic-dimension: The dimension of the dataset that holds the pure anisotropic frequency contributions. In `mrinversion`, this must always be the dimension at index 0 of the data object.

```
anisotropic_dimension = data_object_truncated.dimensions[0]
```

x-y dimensions: The two inverse dimensions corresponding to the *x* and *y*-axis of the *x*-*y* grid.

```
inverse_dimensions = [
    cp.LinearDimension(count=28, increment="400 Hz", label="x"),  # the `x`-dimension.
    cp.LinearDimension(count=28, increment="400 Hz", label="y"),  # the `y`-dimension.
]
```

Generating the kernel

For MAF datasets, the line-shape kernel corresponds to the pure nuclear shielding anisotropy line-shapes. Use the *ShieldingPALineshape* (page 17) class to generate a shielding line-shape kernel.

```
lineshape = ShieldingPALineshape(
    anisotropic_dimension=anisotropic_dimension,
    inverse_dimension=inverse_dimensions,
    channel="29Si",
    magnetic_flux_density="9.4 T",
    rotor_angle="90°",
    rotor_frequency="12 kHz",
    number_of_sidebands=4,
)
```

Here, `lineshape` is an instance of the *ShieldingPALineshape* (page 17) class. The required arguments of this class are the *anisotropic_dimension*, *inverse_dimension*, and *channel*. We have already defined the first two arguments in the previous sub-section. The value of the *channel* argument is the nucleus observed in the MAF experiment. In this example, this value is '29Si'. The remaining arguments, such as the *magnetic_flux_density*, *rotor_angle*, and *rotor_frequency*, are set to match the conditions under which the 2D MAF spectrum was acquired. The value of the *number_of_sidebands* argument is the number of sidebands calculated for each line-shape within the kernel. Unless, you have a lot of spinning sidebands in your MAF dataset, four sidebands should be enough.

Once the ShieldingPALineshape instance is created, use the *kernel()* (page 18) method of the instance to generate the MAF line-shape kernel.

```
K = lineshape.kernel(supersampling=1)
print(K.shape)
```

Out:

```
(64, 784)
```

The kernel `K` is a NumPy array of shape (32, 784), where the axes with 32 and 784 points are the anisotropic dimension and the features (x-y coordinates) corresponding to the 28×28 *x*-*y* grid, respectively.

Data Compression

Data compression is optional but recommended. It may reduce the size of the inverse problem and, thus, further computation time.

```
new_system = TSVDCompression(K, data_object_truncated)
compressed_K = new_system.compressed_K
compressed_s = new_system.compressed_s

print(f"truncation_index = {new_system.truncation_index}")
```

Out:

```
compression factor = 1.032258064516129
truncation_index = 62
```

Solving the inverse problem

Smooth LASSO cross-validation

Solve the smooth-lasso problem. Ordinarily, one should use the statistical learning method to solve the inverse problem over a range of α and λ values and then determine the best nuclear shielding tensor parameter distribution for the given 2D MAF dataset. Considering the limited build time for the documentation, we skip this step and evaluate the distribution at pre-optimized α and λ values. The optimum values are $\alpha = 1.2 \times 10^{-4}$ and $\lambda = 4.55 \times 10^{-6}$. The following commented code was used in determining the optimum α and λ values.

```
# from mrinversion.linear_model import SmoothLassoCV
# import numpy as np

# # setup the pre-defined range of alpha and lambda values
# lambdas = 10 ** (-4.5 - 1 * (np.arange(20) / 19))
# alphas = 10 ** (-2.5 - 3 * (np.arange(20) / 19))
```

(continues on next page)

(continued from previous page)

```
# # setup the smooth lasso cross-validation class
# s_lasso = SmoothLassoCV(
#     alphas=alphas,  # A numpy array of alpha values.
#     lambdas=lambdas,  # A numpy array of lambda values.
#     sigma=0.016,  # The standard deviation of noise from the MAF data.
#     folds=10,  # The number of folds in n-folds cross-validation.
#     inverse_dimension=inverse_dimensions,  # previously defined inverse dimensions.
#     verbose=1,  # If non-zero, prints the progress as the computation proceeds.
# )

# # run fit using the compressed kernel and compressed data.
# s_lasso.fit(compressed_K, compressed_s)

# # the optimum hyper-parameters, alpha and lambda, from the cross-validation.
# print(s_lasso.hyperparameters)
# # {'alpha': 3.359818286283781e-05, 'lambda': 5.324953129837531e-06}

# # the solution
# f_sol = s_lasso.f

# # the cross-validation error curve
# CV_metric = s_lasso.cross_validation_curve
```

If you use the above `SmoothLassoCV` method, skip the following code-block.

```
s_lasso = SmoothLasso(
    alpha=1.2e-4, lambda1=4.55e-6, inverse_dimension=inverse_dimensions
)
# run the fit method on the compressed kernel and compressed data.
s_lasso.fit(K=compressed_K, s=compressed_s)
```

The optimum solution

The *f* (page 20) attribute of the instance holds the solution,

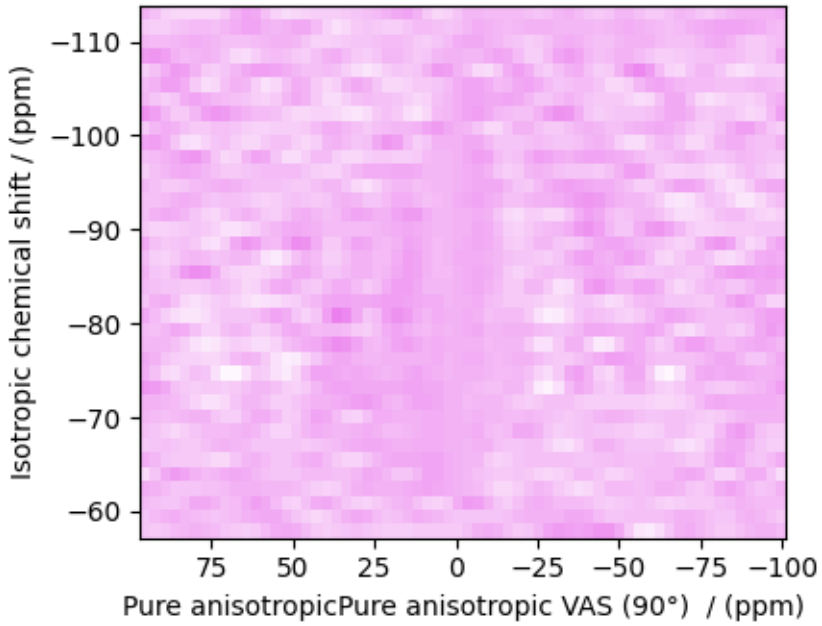
```
f_sol = s_lasso.f  # f_sol is a CSDM object.
```

where `f_sol` is the optimum solution.

The fit residuals

To calculate the residuals between the data and predicted data(fit), use the *residuals()* (page 20) method, as follows,

```
residuals = s_lasso.residuals(K, data_object_truncated)
# residuals is a CSDM object.

# The plot of the residuals.
plot2D(residuals, vmax=data_object_truncated.max(), vmin=data_object_truncated.min())
```



The standard deviation of the residuals is

```
residuals.std()
```

Out:

```
<Quantity 0.01464604>
```

Saving the solution

To serialize the solution to a file, use the *save()* method of the CSDM object, for example,

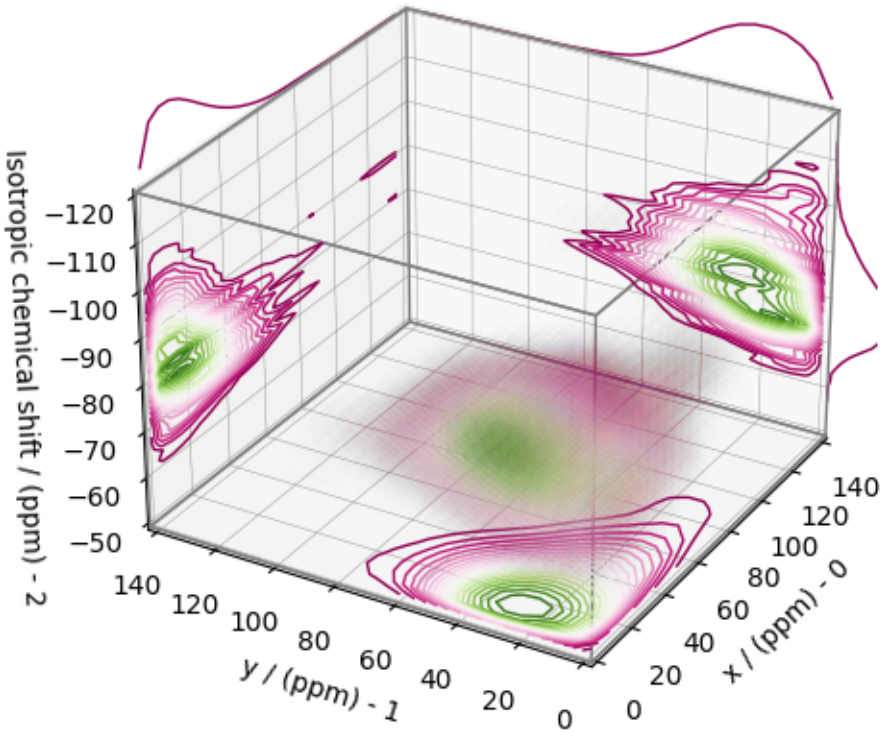
```
f_sol.save("MgO.SiO2_inverse.csdf")  # save the solution
residuals.save("MgO.SiO2_residue.csdf")  # save the residuals
```

Data Visualization

At this point, we have solved the inverse problem and obtained an optimum distribution of the nuclear shielding tensor parameters from the 2D MAF dataset. You may use any data visualization and interpretation tool of choice for further analysis. In the following sections, we provide minimal visualization to complete the case study.

Visualizing the 3D solution

```
# Normalize the solution
f_sol /= f_sol.max()

# Convert the coordinates of the solution, `f_sol`, from Hz to ppm.
[item.to("ppm", "nmr_frequency_ratio") for item in f_sol.dimensions]

# The 3D plot of the solution
plt.figure(figsize=(5, 4.4))
ax = plt.subplot(projection="3d")
plot_3d(ax, f_sol, x_lim=[0, 140], y_lim=[0, 140], z_lim=[-50, -120], alpha=0.05)
plt.tight_layout()
plt.show()
```



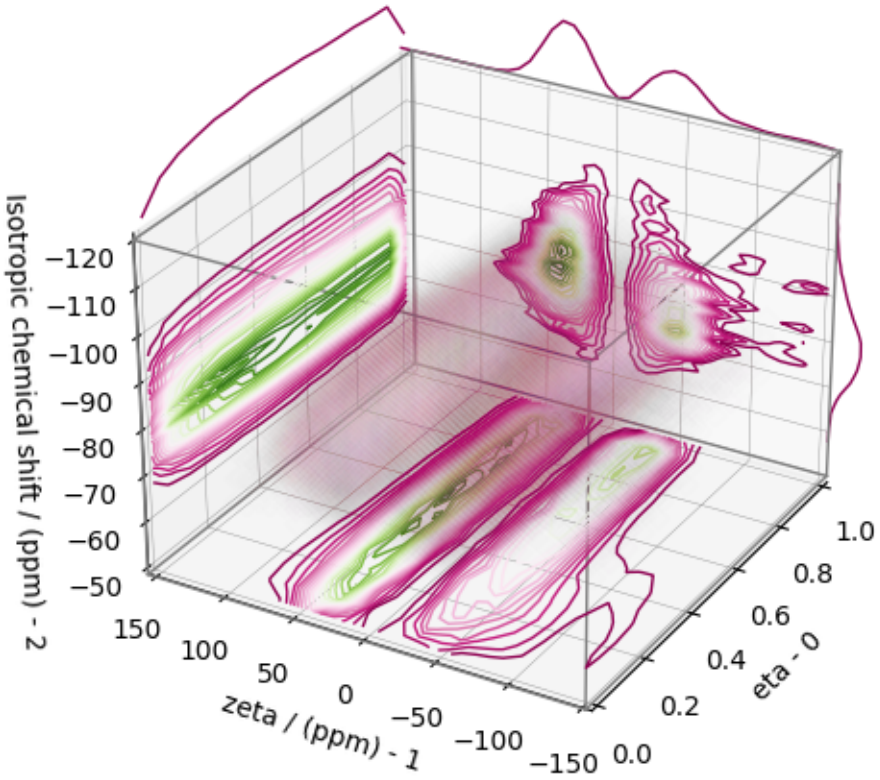
Convert the 3D tensor distribution in Haeberlen parameters

You may re-bin the 3D tensor parameter distribution from a $\rho(\delta_{\text{iso}}, x, y)$ distribution to $\rho(\delta_{\text{iso}}, \zeta_\sigma, \eta_\sigma)$ distribution as follows.

```
# Create the zeta and eta dimensions,, as shown below.
zeta = cp.as_dimension(np.arange(40) * 8 - 150, unit="ppm", label="zeta")
eta = cp.as_dimension(np.arange(16) / 15, label="eta")

# Use the `to_Haeberlen_grid` function to convert the tensor parameter distribution.
fsol_Hae = to_Haeberlen_grid(f_sol, zeta, eta)
```

The 3D plot

```
plt.figure(figsize=(5, 4.4))
ax = plt.subplot(projection="3d")
plot_3d(ax, fsol_Hae, x_lim=[0, 1], y_lim=[-150, 150], z_lim=[-50, -120], alpha=0.05)
plt.tight_layout()
plt.show()
```



References

Total running time of the script: (0 minutes 6.298 seconds)

2D MAF data of CaO.SiO2 glass

The following example illustrates an application of the statistical learning method applied in determining the distribution of the nuclear shielding tensor parameters from a 2D magic-angle flipping (MAF) spectrum. In this example, we use the 2D MAF spectrum[1] of $CaO \cdot SiO_2$ glass.

Before getting started

Import all relevant packages.

```
import csdmpy as cp
import matplotlib.pyplot as plt
import numpy as np

from mrinversion.kernel.nmr import ShieldingPALineshape
from mrinversion.linear_model import SmoothLasso, TSVDCompression
from mrinversion.utils import plot_3d, to_Haeberlen_grid
```

Setup for the matplotlib figures.

```
# function for plotting 2D dataset
def plot2D(csdm_object, **kwargs):
    plt.figure(figsize=(4.5, 3.5))
    ax = plt.subplot(projection="csdm")
    ax.imshow(csdm_object, cmap="gist_ncar_r", aspect="auto", **kwargs)
    ax.invert_xaxis()
    ax.invert_yaxis()
    plt.tight_layout()
    plt.show()
```

Dataset setup

Import the dataset

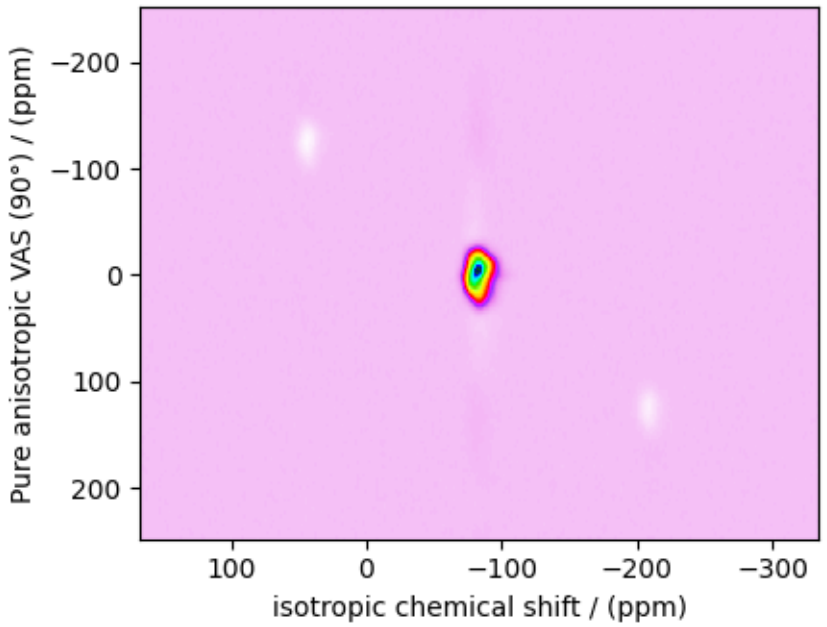
Load the dataset. Here, we import the dataset as the CSDM data-object.

```
# The 2D MAF dataset in csdm format
filename = "https://zenodo.org/record/3964531/files/CaO-SiO2-MAF.csdf"
data_object = cp.load(filename)

# For inversion, we only interest ourselves with the real part of the complex dataset.
data_object = data_object.real

# We will also convert the coordinates of both dimensions from Hz to ppm.
_ = [item.to("ppm", "nmr_frequency_ratio") for item in data_object.dimensions]
```

[1] Zhang, P., Grandinetti, P. J., Stebbins, J. F., Anionic Species Determination in CaSiO3 Glass Using Two-Dimensional 29Si NMR, J. Phys. Chem. B, 101, 4004-4008 (1997). doi:10.1021/jp9700342.

Here, the variable `data_object` is a CSDM object that holds the real part of the 2D MAF dataset. The plot of the 2D MAF dataset is

```
plot2D(data_object)
```



There are two dimensions in this dataset. The dimension at index 0 is the isotropic chemical shift dimension, whereas the dimension at index 1 is the pure anisotropic dimension.

Prepping the data for inversion

Step-1: Data Alignment

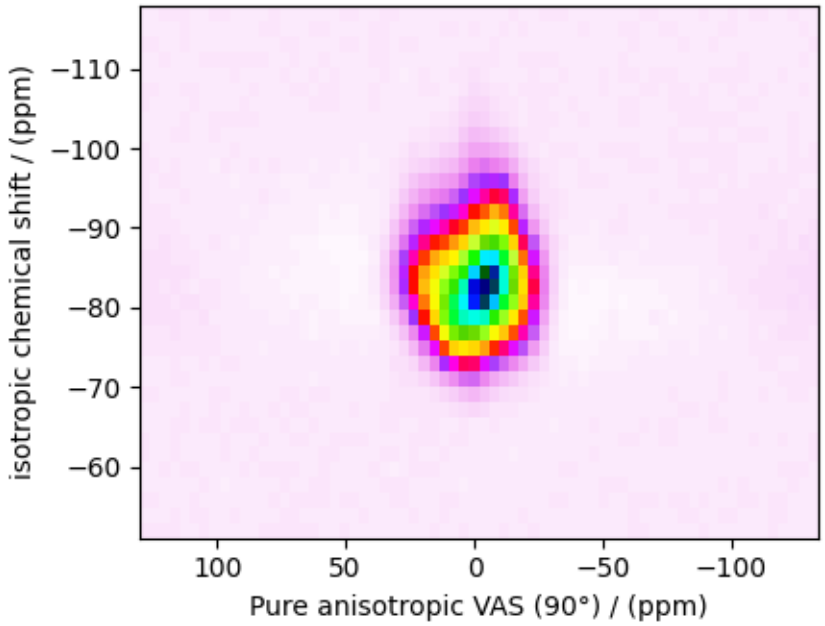
When using the csdm objects with the `mrinversion` package, the dimension at index 0 must be the dimension undergoing the linear inversion. In this example, we plan to invert the pure anisotropic shielding line-shape. In the `data_object`, the anisotropic dimension is at index 1. Transpose the dataset before proceeding.

```
data_object = data_object.T
```

Step-2: Optimization

Also notice, the signal from the 2D MAF dataset occupies a small fraction of the two-dimensional frequency grid. For optimum performance, truncate the dataset to the relevant region before proceeding. Use the appropriate array indexing/slicing to select the signal region.

```
data_object_truncated = data_object[30:-30, 110:145]
plot2D(data_object_truncated)
```



Linear Inversion setup

Dimension setup

Anisotropic-dimension: The dimension of the dataset that holds the pure anisotropic frequency contributions. In `mrinversion`, this must always be the dimension at index 0 of the data object.

```
anisotropic_dimension = data_object_truncated.dimensions[0]
```

x-y dimensions: The two inverse dimensions corresponding to the *x* and *y*-axis of the *x*-*y* grid.

```
inverse_dimensions = [
    cp.LinearDimension(count=25, increment="400 Hz", label="x"),  # the `x`-dimension.
    cp.LinearDimension(count=25, increment="400 Hz", label="y"),  # the `y`-dimension.
]
```

Generating the kernel

For MAF datasets, the line-shape kernel corresponds to the pure nuclear shielding anisotropy line-shapes. Use the *ShieldingPALineshape* (page 17) class to generate a shielding line-shape kernel.

```
lineshape = ShieldingPALineshape(
    anisotropic_dimension=anisotropic_dimension,
    inverse_dimension=inverse_dimensions,
    channel="29Si",
    magnetic_flux_density="9.4 T",
    rotor_angle="90°",
    rotor_frequency="10.4 kHz",
    number_of_sidebands=4,
)
```

Here, `lineshape` is an instance of the *ShieldingPALineshape* (page 17) class. The required arguments of this class are the *anisotropic_dimension*, *inverse_dimension*, and *channel*. We have already defined the first two arguments in the previous sub-section. The value of the *channel* argument is the nucleus observed in the MAF experiment. In this example, this value is '29Si'. The remaining arguments, such as the *magnetic_flux_density*, *rotor_angle*, and *rotor_frequency*, are set to match the conditions under which the 2D MAF spectrum was acquired. The value of the *number_of_sidebands* argument is the number of sidebands calculated for each line-shape within the kernel. Unless, you have a lot of spinning sidebands in your MAF dataset, four sidebands should be enough.

Once the ShieldingPALineshape instance is created, use the *kernel()* (page 18) method of the instance to generate the MAF line-shape kernel.

```
K = lineshape.kernel(supersampling=1)
print(K.shape)
```

Out:

```
(68, 625)
```

The kernel `K` is a NumPy array of shape (32, 784), where the axes with 32 and 784 points are the anisotropic dimension and the features (x-y coordinates) corresponding to the 28×28 *x*-*y* grid, respectively.

Data Compression

Data compression is optional but recommended. It may reduce the size of the inverse problem and, thus, further computation time.

```
new_system = TSVDCompression(K=K, s=data_object_truncated)
compressed_K = new_system.compressed_K
compressed_s = new_system.compressed_s

print(f"truncation_index = {new_system.truncation_index}")
```

Out:

```
compression factor = 1.5454545454545454
truncation_index = 44
```

Solving the inverse problem

Smooth LASSO cross-validation

Solve the smooth-lasso problem. Ordinarily, one should use the statistical learning method to solve the inverse problem over a range of α and λ values and then determine the best nuclear shielding tensor parameter distribution for the given 2D MAF dataset. Considering the limited build time for the documentation, we skip this step and evaluate the distribution at pre-optimized α and λ values. The optimum values are $\alpha = 2.8 \times 10^{-5}$ and $\lambda = 8.85 \times 10^{-6}$. The following commented code was used in determining the optimum α and λ values.

```
# from mrinversion.linear_model import SmoothLassoCV
# import numpy as np

# # setup the pre-defined range of alpha and lambda values
# lambdas = 10 ** (-4 - 2 * (np.arange(20) / 19))
# alphas = 10 ** (-3.5 - 2 * (np.arange(20) / 19))
```

(continues on next page)

(continued from previous page)

```
# # setup the smooth lasso cross-validation class
# s_lasso = SmoothLassoCV(
#     alphas=alphas,  # A numpy array of alpha values.
#     lambdas=lambdas,  # A numpy array of lambda values.
#     sigma=0.0012,  # The standard deviation of noise from the MAF data.
#     folds=10,  # The number of folds in n-folds cross-validation.
#     inverse_dimension=inverse_dimensions,  # previously defined inverse dimensions.
#     verbose=1,  # If non-zero, prints the progress as the computation proceeds.
#     max_iterations=20000,  # maximum number of allowed iterations.
# )

# # run fit using the compressed kernel and compressed data.
# s_lasso.fit(compressed_K, compressed_s)

# # the optimum hyper-parameters, alpha and lambda, from the cross-validation.
# print(s_lasso.hyperparameters)
# # {'alpha': 3.359818286283781e-05, 'lambda': 5.324953129837531e-06}

# # the solution
# f_sol = s_lasso.f

# # the cross-validation error curve
# CV_metric = s_lasso.cross_validation_curve
```

If you use the above `SmoothLassoCV` method, skip the following code-block.

```
s_lasso = SmoothLasso(
    alpha=2.8e-5, lambda1=8.85e-6, inverse_dimension=inverse_dimensions
)
# run the fit method on the compressed kernel and compressed data.
s_lasso.fit(K=compressed_K, s=compressed_s)
```

The optimum solution

The *f* (page 20) attribute of the instance holds the solution,

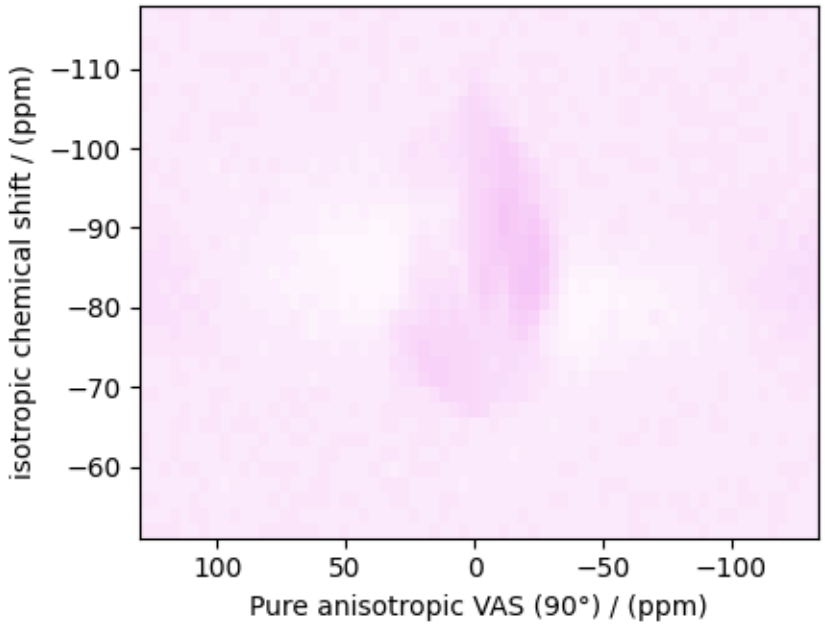
```
f_sol = s_lasso.f  # f_sol is a CSDM object.
```

where `f_sol` is the optimum solution.

The fit residuals

To calculate the residuals between the data and predicted data(fit), use the *residuals()* (page 20) method, as follows,

```
residuals = s_lasso.residuals(K, data_object_truncated)
# residuals is a CSDM object.

# The plot of the residuals.
plot2D(residuals, vmax=data_object_truncated.max(), vmin=data_object_truncated.min())
```



The standard deviation of the residuals is

```
residuals.std()
```

Out:

```
<Quantity 0.00585561>
```

Saving the solution

To serialize the solution to a file, use the *save()* method of the CSDM object, for example,

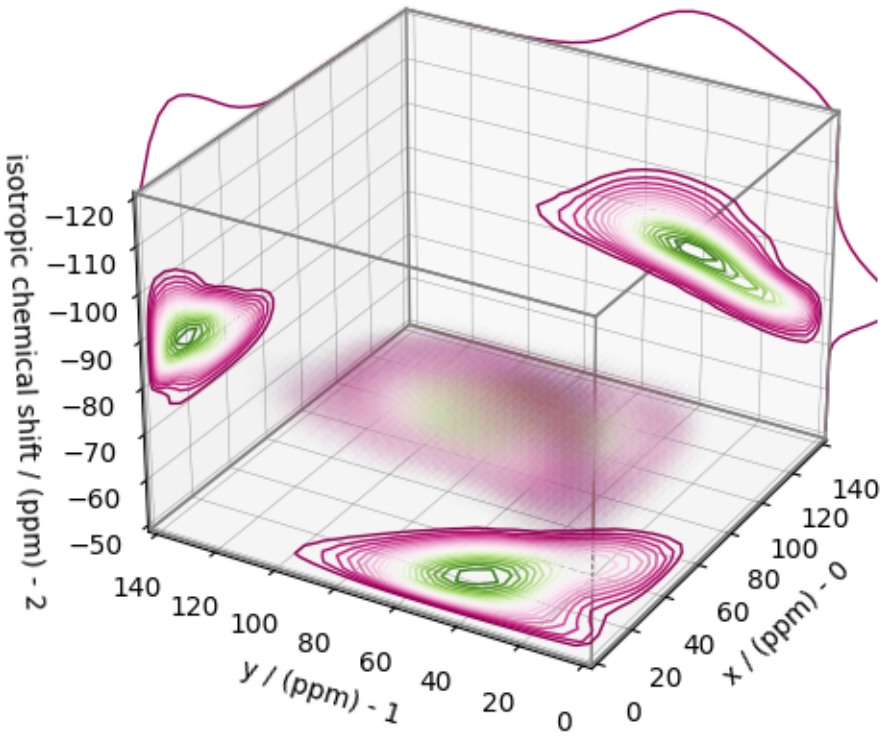
```
f_sol.save("CaO.SiO2_inverse.csdf")  # save the solution
residuals.save("CaO.SiO2_residue.csdf")  # save the residuals
```

Data Visualization

At this point, we have solved the inverse problem and obtained an optimum distribution of the nuclear shielding tensor parameters from the 2D MAF dataset. You may use any data visualization and interpretation tool of choice for further analysis. In the following sections, we provide minimal visualization to complete the case study.

Visualizing the 3D solution

```
# Normalize the solution
f_sol /= f_sol.max()

# Convert the coordinates of the solution, `f_sol`, from Hz to ppm.
[item.to("ppm", "nmr_frequency_ratio") for item in f_sol.dimensions]

# The 3D plot of the solution
plt.figure(figsize=(5, 4.4))
ax = plt.subplot(projection="3d")
plot_3d(ax, f_sol, x_lim=[0, 140], y_lim=[0, 140], z_lim=[-50, -120])
plt.tight_layout()
plt.show()
```



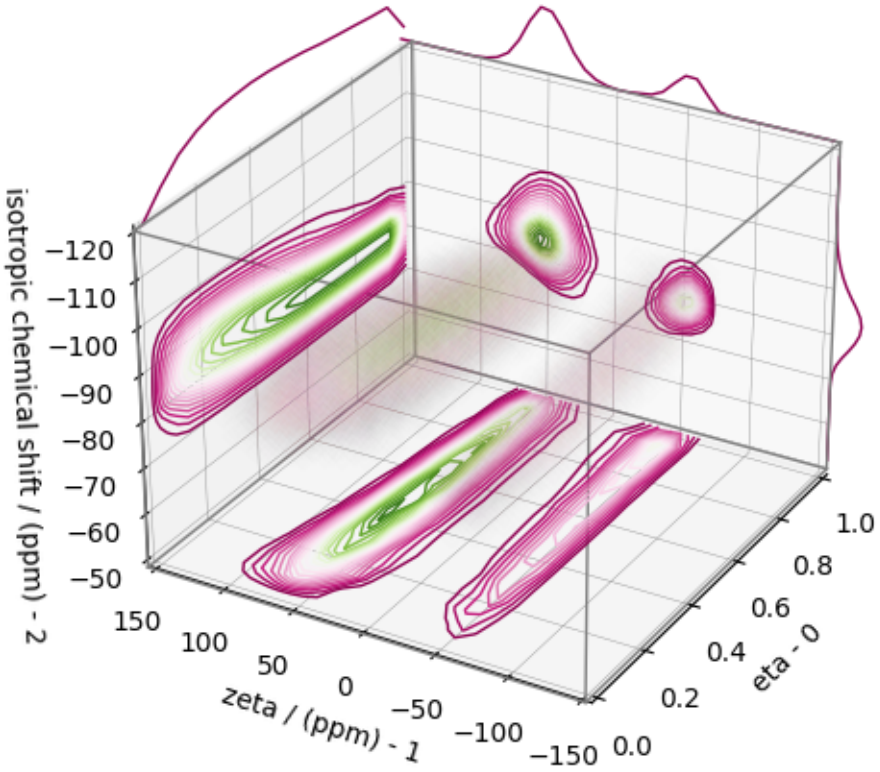
Convert the 3D tensor distribution in Haeberlen parameters

You may re-bin the 3D tensor parameter distribution from a $\rho(\delta_{\text{iso}}, x, y)$ distribution to $\rho(\delta_{\text{iso}}, \zeta_\sigma, \eta_\sigma)$ distribution as follows.

```
# Create the zeta and eta dimensions,, as shown below.
zeta = cp.as_dimension(np.arange(40) * 8 - 150, unit="ppm", label="zeta")
eta = cp.as_dimension(np.arange(16) / 15, label="eta")

# Use the `to_Haeberlen_grid` function to convert the tensor parameter distribution.
fsol_Hae = to_Haeberlen_grid(f_sol, zeta, eta)
```

The 3D plot

```
plt.figure(figsize=(5, 4.4))
ax = plt.subplot(projection="3d")
plot_3d(ax, fsol_Hae, x_lim=[0, 1], y_lim=[-150, 150], z_lim=[-50, -120], alpha=0.05)
plt.tight_layout()
plt.show()
```



References

Total running time of the script: (0 minutes 5.053 seconds)

CHAPTER

THREE

PROJECT DETAILS

3.1 Changelog

3.1.1 v0.2.0

What's new!

- Added *to_Haeberlen_grid()* (page 24) function to convert the 3D $\rho(\delta_{\text{iso}}, x, y)$ distribution to $\rho(\delta_{\text{iso}}, \zeta_\sigma, \eta_\sigma)$ distribution.

Changes

- Update code to comply with updated csdmpy library.

3.2 License

3.2.1 Mrinversion License

Mrinversion is licensed under BSD 3-Clause License.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

3.3 Acknowledgment

The development of the mrinversion library is supported in part by the US National Science Foundation under Grant No. DIBBS OAC 1640899 and Grant No. CHE 1807922.

CHAPTER

FOUR

HOW TO CITE

If you use this work in your publication, please cite the following.

- Srivastava, D. J.; Grandinetti P. J., Statistical learning of NMR tensors from 2D isotropic/anisotropic correlation nuclear magnetic resonance spectra, J. Chem. Phys. **153**, 134201 (2020). https://doi.org/10.1063/5.0023345.
- Deepansh J. Srivastava, Maxwell Venetos, Philip J. Grandinetti, Shyam Dwaraknath, & Alexis McCarthy. (2021, May 26). mrsimulator: v0.6.0 (Version v0.6.0). Zenodo. http://doi.org/10.5281/zenodo.4814638

Additionally, if you use the CSDM data model, please consider citing

- Srivastava DJ, Vosegaard T, Massiot D, Grandinetti PJ (2020) Core Scientific Dataset Model: A lightweight and portable model and file format for multi-dimensional scientific data. PLOS ONE 15(1): e0225953. https://doi.org/10.1371/journal.pone.0225953

INDEX