paramagpy Documentation

Release 0.44

Henry Orton

Nov 04, 2019

CONTENTS

Release 0.44

Date Nov 04, 2019

CHAPTER

ONE

INTRODUCTION

paramagpy is a python module for calculating paramagnetic effects in NMR spectra of proteins. This currently includes fitting of paramagnetic susceptibility tensors to experimental data associated with pseudocontact shifts (PCS) residual dipolar couplings (RDC), paramagnetic relaxation enhancements (PRE) and cross-correlated relaxation (CCR). A GUI allows easy viewing of data and seamless transition between PCS/RDC/PRE/CCR calculations.

Fig. 1: *Please, not the eyes!* - Canberra cyclist

CHAPTER

TWO

FEATURES

- Support for PDB protein structures with models
- Combined SVD gridsearch and gradient descent algorithms for solving PCS tensors
- Optional fitting of reference offset parameter for PCS datasets
- Support for Residual Anisotropic Chemical Shielding (RACS) and Residual Anisotropic Dipolar Shielding (RADS) corrections to PCS
- Lanthanide parameter templates available
- Plotting of correlation between experiment/calculated values
- Plotting of tensor isosurfaces compatible with PyMol
- Q-factor calculations
- Error analysis of tensor fit quality by Monte-Carlo or Bootstrap methods
- Optimisation of multiple PCS/PRE/CCR datasets to a common position
- Unique tensor representation compatible with Numbat (program)
- Fitting of RDC tensor by SVD algorithm
- PRE calculations by Solomon and Curie spin mechanisms
- Spectral power density tensor fitting for anisotropic dipolar PREs
- CSA cross-correlation correction to PRE calculations
- Dipole-dipole/Curie spin cross-correlated relaxation calculations
- Fitting of tensor parameters to PRE/CCR data
- Macro scripts for integration with CCPNMR and Sparky

CHAPTER

THREE

DOCUMENTATION

- https://henryorton.github.io/paramagpy/

CHAPTER

FOUR

CITING PARAMAGPY

The paramagpy preprint can be found on ChemArXiv https://doi.org/10.26434/chemrxiv.9643154.v1

CHAPTER

FIVE

CONTENTS

5.1 Installation Guide

5.1.1 Requirements

Paramagpy is written for python 3. It requires packages:

- NumPy
- SciPy
- matplotlib
- BioPython

5.1.2 Unix/OSX Installation

Install directly using pip:

```
$ pip install paramagpy
```

Or, download the source code and run:

```
$ python setup.py install
```

within the source directory.

5.1.3 Windows Installation

Paramagpy has never been tested on windows, but theoretically it should work. Good luck!

5.1.4 Running the GUI

Once you have installed paramagpy, see *Graphic User Interface (GUI)* for how to run the GUI.

5.2 Examples

5.2.1 PCS data

Fit Tensor to PCS Data

This example shows how to fit a $\Delta\chi$-tensor to experimental PCS data for the protein calbindin D9k. These data contain amide 1H and 15N chemical shifts between diamagnetic and paramagnetic states with the lanthanide Er3+ bound.

Downloads

- Download the data files `4icbH_mut.pdb` and `calbindin_Er_HN_PCS.npc` from here:
- Download the script `pcs_fit.py`

Script + Explanation

Firstly, the necessary modules are imported from paramagpy.

```
from paramagpy import protein, fit, dataparse, metal
```

The protein is then loaded from a PDB file using *paramagpy.protein.load_pdb()* into the variable `prot`. This returns a `CustomStructure` object which is closely based on the `Structure` object from BioPython and contains the atomic coordinates. The object, and how to access atomic coordinates is discussed at this link.

```
# Load the PDB file
prot = protein.load_pdb('../data_files/4icbH_mut.pdb')
```

The PCS data is then loaded from a `.npc` file using the function *paramagpy.dataparse.read_pcs()* into the variable `rawData`. This is a dictionary of `(PCS, Error)` tuples which may be accessed by `rawData[(seq, atom)]` where `seq` is an integer specifying the sequence and `atom` is the atom name e.g `(3,'HA')`. Note that these should match the corresponding sequence and atom in the PDB file.

```
# Load the PCS data
rawData = dataparse.read_pcs('../data_files/calbindin_Er_HN_PCS.npc')
```

To associate the experimental PCS value with atoms of the PDB structure, the method *paramagpy.protein.CustomStructure.parse()* is called on `rawData`. The new list `parsedData` contains elements `[atom, PCS, Error]`, where `atom` is now an atom object from the PDB.

```
# Associate PCS data with atoms of the PDB
parsedData = prot.parse(rawData)
```

An initial $\Delta\chi$-tensor is defined by initialising a *paramagpy.metal.Metal* object. The initial position is known to be near the binding site, which is set to the CA atom of residue 56. Note that the `position` attribute is always in Angstrom units.

```
# Define an initial tensor
mStart = metal.Metal()

# Set the starting position to an atom close to the metal
mStart.position = prot[0]['A'][56]['CA'].position
```

A quick gridsearch is conducted in a sphere of 10 Angstrom with 10 points per radius using the function *paramagpy.fit.svd_gridsearch_fit_metal_from_pcs()*. This requires two lists containing the starting metals `mStart` and parsed experimental data `parsedData`. This function return lists containing a new fitted metal object, the calculated PCS values from the fitted model, and the Q-factor.

```
# Calculate an initial tensor from an SVD gridsearch
mGuess, calc, qfac = fit.svd_gridsearch_fit_metal_from_pcs(
        [mStart],[parsedData], radius=10, points=10)
```

This is then refined using a non-linear regression gradient descent with the function *paramagpy.fit.nlr_fit_metal_from_pcs()*.

```
# Refine the tensor using non-linear regression
mFit, calc, qfac = fit.nlr_fit_metal_from_pcs(mGuess, [parsedData])
```

The fitted tensor parameters are saved by calling the method *paramagpy.metal.Metal.save()*. Alternatively they may be displayed using `print(mFit[0].info())`

```
# Save the fitted tensor to file
mFit[0].save('calbindin_Er_HN_PCS_tensor.txt')
```

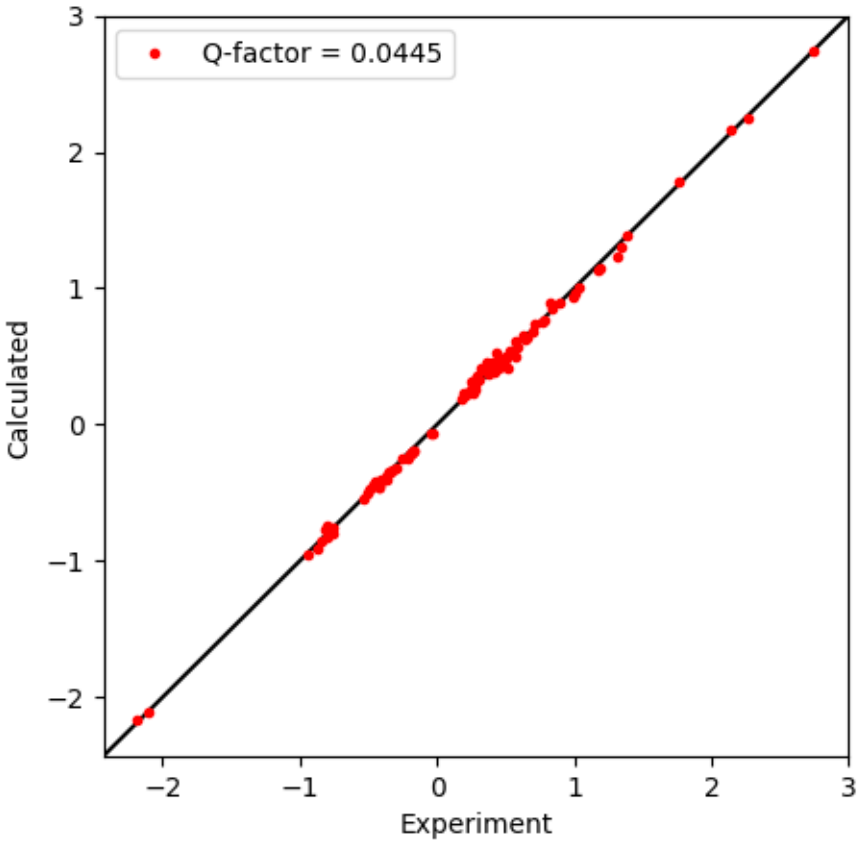
Output: [calbindin_Er_HN_PCS_tensor.txt]

```
ax     | 1E-32 m^3 :    -8.688
rh     | 1E-32 m^3 :    -4.192
x      |   1E-10 m :    25.517
y      |   1E-10 m :     8.652
z      |   1E-10 m :     6.358
a      |       deg :   116.011
b      |       deg :   138.058
g      |       deg :    43.492
mueff  |        Bm :     0.000
shift  |       ppm :     0.000
B0     |         T :    18.790
temp   |         K :   298.150
t1e    |        ps :     0.000
taur   |        ns :     0.000
```

These experimental/calculated PCS values are then plotted in a correlation plot to assess the fit. This is achieved using standard functions of the plotting module matplotlib.

```
#### Plot the correlation ####
from matplotlib import pyplot as plt
fig, ax = plt.subplots(figsize=(5,5))

# Unpack the experimental values
atoms, experiment, errors = zip(*parsedData)

# Plot the data
ax.plot(experiment, calc[0], marker='o', lw=0, ms=3, c='r',
        label="Q-factor = {:5.4f}".format(qfac[0]))

# Plot a diagonal
l, h = ax.get_xlim()
ax.plot([l,h],[l,h],'-k',zorder=0)
ax.set_xlim(l,h)
ax.set_ylim(l,h)

# Make axis labels and save figure
ax.set_xlabel("Experiment")
ax.set_ylabel("Calculated")
ax.legend()
fig.savefig("pcs_fit.png")
```

Output: [pcs_fit.png]



Plot PCS isosurface (PyMol view)

This example shows how to plot the PCS isosurface of a fitted $\Delta\chi$-tensor for data from the example *Fit Tensor to PCS Data*. The isosurface can be viewed in PyMol.

Downloads

- Download the data files `4icbH_mut.pdb` and `calbindin_Er_HN_PCS_tensor.txt` from here:
- Download the script pcs_plot_isosurface.py

Explanation

The protein and tensor are loaded as described previously in.

The isosurface files are generated using the function *`paramagpy.metal.Metal.isomap()`*. The contour level can be chosen by setting the `isoval` argument. A larger `density` value will result in a smoother surface. This function writes two files `isomap.pml` and `isomap.pml.ccp4` which are the PyMol script and PCS grid files respectively.

The isosurface can be displayed by executing `pymol isomap.pml` from a terminal, or by selecting `File>Run` and navigating to the script `isomap.pml`.

Script

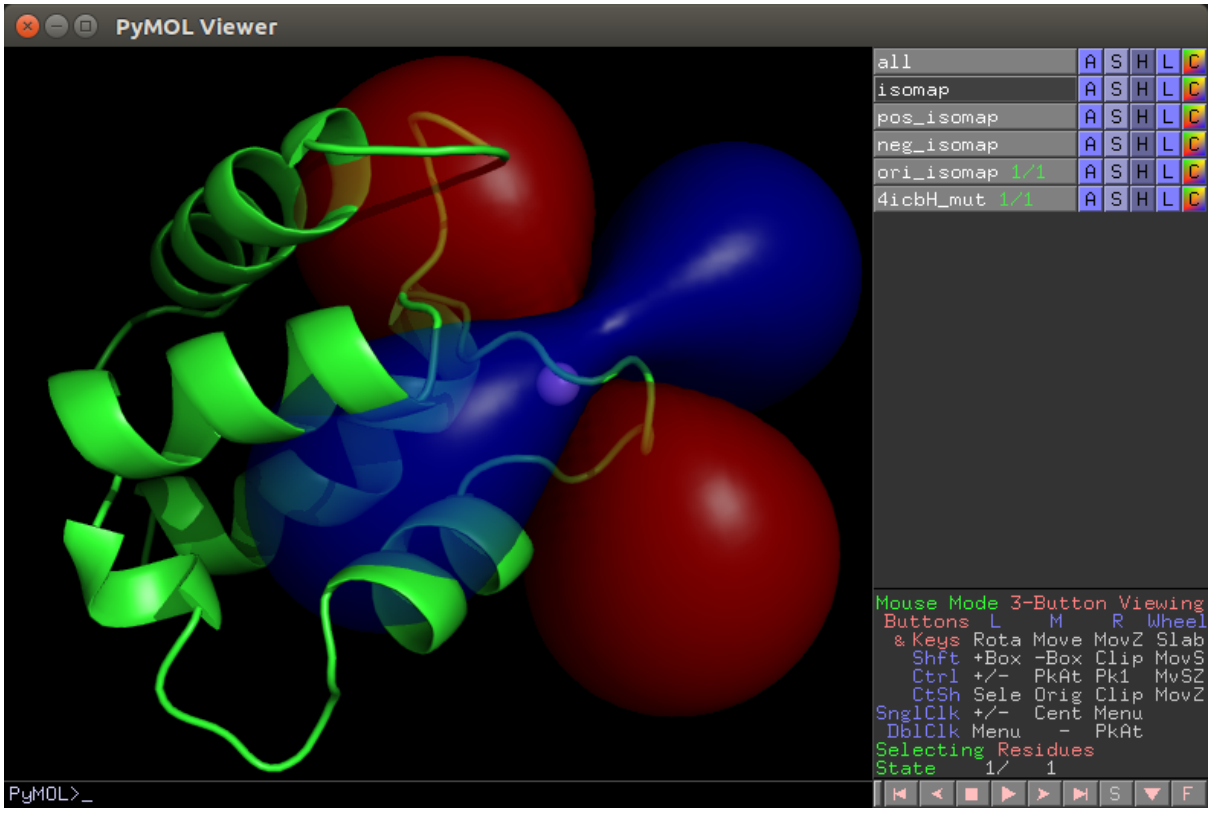
[pcs_plot_isosurface.py]

```
from paramagpy import protein, fit, dataparse, metal

# Load the PDB file
prot = protein.load_pdb('../data_files/4icbH_mut.pdb')

# Load the fitted tensor
met = metal.load_tensor('../data_files/calbindin_Er_HN_PCS_tensor.txt')

# Plot the isosurface to be opened in PyMol
met.isomap(prot.id, density=1, isoval=1.0)
```

Output

PyMol view of isosurface

[pcs_plot_isosurface.png]



Fit multiple PCS datasets to common position

This example shows how to fit multiple $\Delta\chi$-tensors to their respective datasets with a common position, but varied magnitude and orientation. This may arise if several lanthanides were investigated at the same binding site, and the data may be used simultaneously to fit a common position. Data from several PCS datasets for calbindin D9k were used here, and is a generalisation of the previous example: *Fit Tensor to PCS Data*.

Downloads

- Download the data files `4icbH_mut.pdb`, `calbindin_Tb_HN_PCS.npc`, `calbindin_Er_HN_PCS.npc` and `calbindin_Yb_HN_PCS_tensor.txt` from here:
- Download the script pcs_fit_multiple.py

Explanation

The protein and PCS datasets are loaded and parsed. These are placed into a list `parsedData`, for which each element is a PCS dataset of a given lanthanide.

The two fitting functions:

- *paramagpy.fit.svd_gridsearch_fit_metal_from_pcs()*
- *paramagpy.fit.nlr_fit_metal_from_pcs()*

can accept a list of metal objects and a list of datasets with arbitrary size. If this list contains more than one element, fitting will be performed to a common position. The starting position is taken only from the first metal of the list.

After fitting, a list of fitted metals is returned. The fitted tensor are then written to files and a correlation plot is made.

Script

[pcs_fit_multiple.py]

```
from paramagpy import protein, fit, dataparse, metal

# Load the PDB file
prot = protein.load_pdb('../data_files/4icbH_mut.pdb')

# Load the PCS data
rawData1 = dataparse.read_pcs('../data_files/calbindin_Tb_HN_PCS.npc')
rawData2 = dataparse.read_pcs('../data_files/calbindin_Er_HN_PCS.npc')
rawData3 = dataparse.read_pcs('../data_files/calbindin_Yb_HN_PCS.npc')

# Associate PCS data with atoms of the PDB
parsedData = []
for rd in [rawData1, rawData2, rawData3]:
        parsedData.append(prot.parse(rd))

# Make a list of starting tensors
mStart = [metal.Metal(), metal.Metal(), metal.Metal()]

# Set the starting position to an atom close to the metal
mStart[0].position = prot[0]['A'][56]['CA'].position

# Calculate initial tensors from an SVD gridsearch
mGuess = fit.svd_gridsearch_fit_metal_from_pcs(
        mStart, parsedData, radius=10, points=10)

# Refine the tensors using non-linear regression
fitParameters = ['x','y','z','ax','rh','a','b','g']
mFit = fit.nlr_fit_metal_from_pcs(mGuess, parsedData, fitParameters)

# Save the fitted tensors to files
for name, metal in zip(['Tb','Er','Yb'], mFit):
        metal.save("tensor_{}.txt".format(name))
```

(continues on next page)

(continued from previous page)

```
# Make experimental and calculated PCS lists
exp = []
cal = []
for metal, data in zip(mFit, parsedData):
        ex = []
        ca = []
        for atom, exp_pcs, error in data:
                ex.append(exp_pcs)
                ca.append(metal.atom_pcs(atom))
        exp.append(ex)
        cal.append(ca)

#### Plot the correlation ####
from matplotlib import pyplot as plt
fig, ax = plt.subplots(figsize=(5,5))

# Plot the data
for e, c, name, colour in zip(exp, cal, ['Tb','Er','Yb'],['r','g','b']):
        qfactor = fit.qfactor(e,c)
        ax.plot(e, c, marker='o', lw=0, ms=1, c=colour,
                label="{0:} - {1:5.3f}".format(name, qfactor))

# Plot a diagonal
l, h = ax.get_xlim()
ax.plot([l,h],[l,h],'-k',zorder=0)
ax.set_xlim(l,h)
ax.set_ylim(l,h)

# Axis labels
ax.set_xlabel("Experiment")
ax.set_ylabel("Calculated")
ax.legend()
fig.savefig("pcs_fit_multiple.png")
```

Outputs

Tb fitted tensor

[tensor_Tb.txt]

```
ax     | 1E-32 m^3 :    31.096
rh     | 1E-32 m^3 :    12.328
x      |   1E-10 m :    25.937
y      |   1E-10 m :     9.481
z      |   1E-10 m :     6.597
a      |       deg :   151.053
b      |       deg :   152.849
g      |       deg :    69.821
mueff  |        Bm :     0.000
shift  |       ppm :     0.000
B0     |         T :    18.790
temp   |         K :   298.150
t1e    |        ps :     0.000
```

Er fitted tensor

[tensor_Er.txt]

```
ax     | 1E-32 m^3 :    -8.421
rh     | 1E-32 m^3 :    -4.886
x      |   1E-10 m :    25.937
y      |   1E-10 m :     9.481
z      |   1E-10 m :     6.597
a      |       deg :   126.015
b      |       deg :   142.899
g      |       deg :    41.040
mueff  |        Bm :     0.000
shift  |       ppm :     0.000
B0     |         T :    18.790
temp   |         K :   298.150
t1e    |        ps :     0.000
```

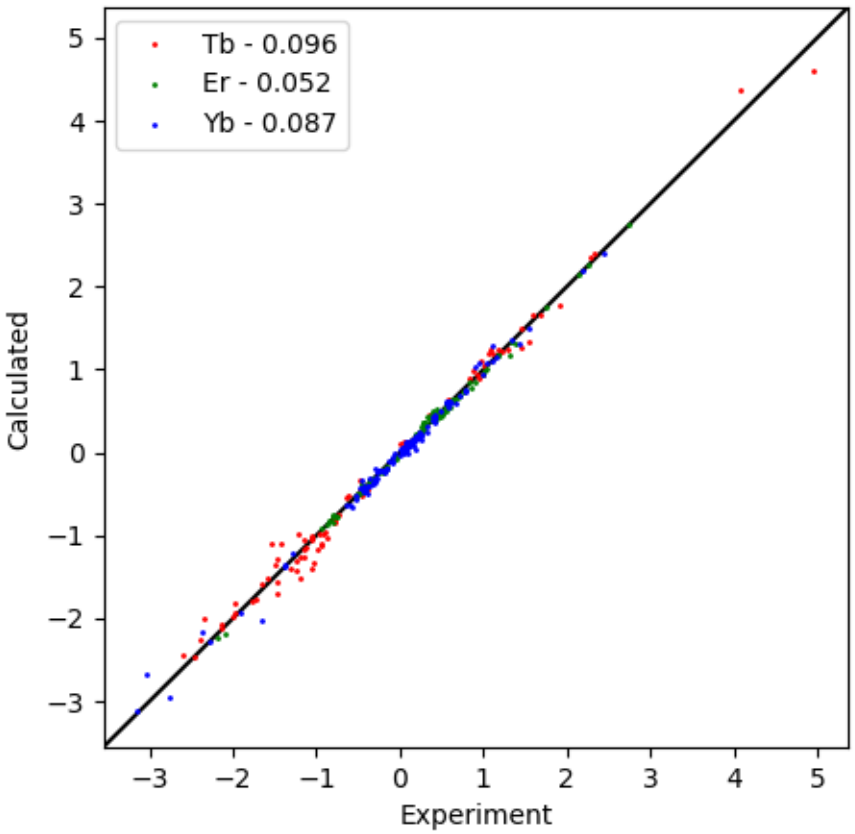
Yb fitted tensor

[tensor_Yb.txt]

```
ax     | 1E-32 m^3 :    -5.392
rh     | 1E-32 m^3 :    -2.490
x      |   1E-10 m :    25.937
y      |   1E-10 m :     9.481
z      |   1E-10 m :     6.597
a      |       deg :   129.650
b      |       deg :   137.708
g      |       deg :    88.796
mueff  |        Bm :     0.000
shift  |       ppm :     0.000
B0     |         T :    18.790
temp   |         K :   298.150
t1e    |        ps :     0.000
```

Correlation Plot

[pcs_fit_multiple.png]



Fit Tensor to PDB with Models

This example shows how to fit a $\Delta\chi$-tensor to experimental PCS data using an NMR structure that contains many models. The tensor can be fit to ensemble averaged PCS values, or to individual models. An ensemble averaged PCS is the mean calculated PCS of all models. No structural averages are ever taken.

Data for calbindin D9k are used as in the previous example *Fit Tensor to PCS Data*.

Downloads

- Download the data files `2bcb.pdb` and `calbindin_Er_HN_PCS.npc` from here:
- Download the script `pcs_fit_models.py`

Script + Explanation

Firstly, the standard preamble and loading of data.

```
from paramagpy import protein, fit, dataparse, metal

# Load data
prot = protein.load_pdb('../data_files/2bcb.pdb')
rawData = dataparse.read_pcs('../data_files/calbindin_Er_HN_PCS.npc')
mStart = metal.Metal()
mStart.position = prot[0]['A'][56]['CA'].position
```

The default method of fitting is to minimise the difference to the experimental values of the ensemble average of the calculated values. The default behaviour is to average atoms with the same serial number in the PDB file. To manipulate ensemble averaging, you can specify the `sumIndices` argument of any fitting function such as *paramagpy.fit.nlr_fit_metal_from_pcs()*. This array contains common integers for corresponding atoms to be averaged. To remove ensemble averaging completely, just specify a list of unique integers with length equal to the data such as `sumIndices=list(range(len(parsedData)))`.

```
#### Ensemble average fitting ####
parsedData = prot.parse(rawData)
mGuess, _, _ = fit.svd_gridsearch_fit_metal_from_pcs(
        [mStart],[parsedData], radius=10, points=10)
mFit, calc, qfac = fit.nlr_fit_metal_from_pcs(mGuess, [parsedData])
mFit[0].save('calbindin_Er_HN_PCS_tensor_ensemble.txt')
```

If desired, you can also fit a separate tensor to each model of the PDB and the compare them. In this case, we loop over each model, fit a tensor, then keep the one with the smallest Q-factor. Selected models can be parsed by specifying the `models` argument of *paramagpy.protein.CustomStructure.parse()*.

```
#### Single model fitting ####
# Loop over models, fit tensor and keep one with best Q-factor
minQfacMod = 1E50
for model in prot:
        parsedDataMod = prot.parse(rawData, models=model.id)
        mFitMod, calcMod, qfacMod = fit.nlr_fit_metal_from_pcs(
                mGuess, [parsedDataMod])
        if qfacMod[0] < minQfacMod:
                minMod = model.id
                minParsedDataMod = parsedDataMod
                minmFitMod = mFitMod
                mincalcMod = calcMod
                minQfacMod = qfacMod
```

Finally we plot three sets of data:

- The ensemble average fit calculated for each model (green)
- The ensemble average of the calculated values of the ensemble fit (red)
- The best fitting single model (blue)

Note that to calculate the ensemble average of the calculated values we use the function *paramagpy.fit.ensemble_average()*. This can take any number of arguments, and will average values based on common serial numbers of the list of atoms in the first argument.

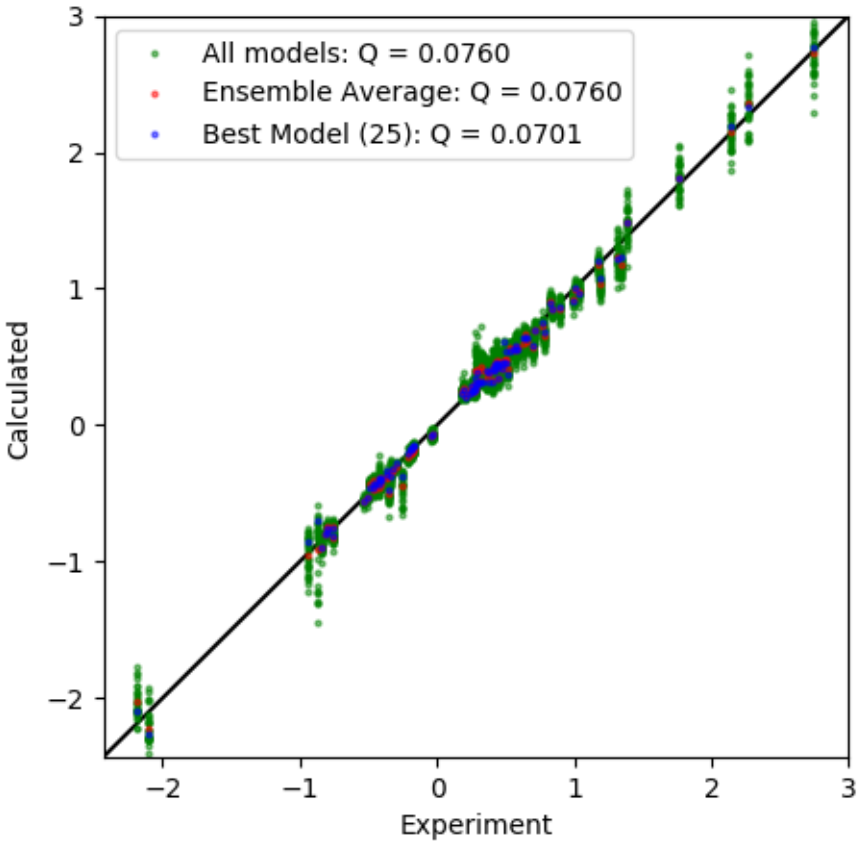
```
# #### Plot the correlation ####
from matplotlib import pyplot as plt
fig, ax = plt.subplots(figsize=(5,5))

# Unpack the experimental values
atoms, exp, err = zip(*parsedData)
expEnsemble, calcEnsemble = fit.ensemble_average(atoms, exp, calc[0])
atomsMod, expMod, errMod = zip(*minParsedDataMod)

# Plot all models
ax.plot(exp, calc[0], marker='o', lw=0, ms=2, c='g',
        alpha=0.5, label="All models: Q = {:5.4f}".format(qfac[0]))

# Plot the ensemble average
ax.plot(expEnsemble, calcEnsemble, marker='o', lw=0, ms=2, c='r',
        alpha=0.5, label="Ensemble Average: Q = {:5.4f}".format(qfac[0]))

# Plot the model with minimum Q-factor
ax.plot(expMod, mincalcMod[0], marker='o', lw=0, ms=2, c='b',
```

(continues on next page)

(continued from previous page)

```
        alpha=0.5, label="Best Model ({0:}): Q = {1:5.4f}".format(
                minMod, minQfacMod[0]))

# Plot a diagonal
l, h = ax.get_xlim()
ax.plot([l,h],[l,h],'-k',zorder=0)
ax.set_xlim(l,h)
ax.set_ylim(l,h)

# Make axis labels and save figure
ax.set_xlabel("Experiment")
ax.set_ylabel("Calculated")
ax.legend()
fig.savefig("pcs_fit_models.png")
```

Output: [pcs_fit_models.png]



Constrained Fitting

This example shows how to fit a $\Delta\chi$-tensor with constraints applied. The two cases here constrain position to fit a tensor to a known metal ion position form an X-ray structure, and fit an axially symmetric tensor with only 6 of the usual 8 parameters.

Downloads

- Download the data files 4icbH_mut.pdb and calbindin_Er_HN_PCS.npc from here:

- Download the script `pcs_fit_constrained.py`

Script + Explanation

The necessary modules are imported and data is loaded

```
from paramagpy import protein, fit, dataparse, metal

# Load data
prot = protein.load_pdb('../data_files/4icbH_mut.pdb')
rawData = dataparse.read_pcs('../data_files/calbindin_Er_HN_PCS.npc')
parsedData = prot.parse(rawData)
mStart = metal.Metal()
```

The calcium ion from the X-ray structure is contained in a heteroatom of the PDB file. We set the starting position of the tensor to this position.

```
# Set the starting position to Calcium ion heteroatom in PDB
mStart.position = prot[0]['A'][('H_ CA', 77, ' ')]['CA'].position
```

To fit the the anisotropy and orientation without position, the linear PCS equation can be solved analytically by the SVD gridsearch method but using only one point with a radius of zero. This tensor is then saved.

```
# Calculate tensor by SVD
mFit, calc, qfac = fit.svd_gridsearch_fit_metal_from_pcs(
        [mStart],[parsedData], radius=0, points=1)

mFit[0].save('calbindin_Er_HN_PCS_tensor_position_constrained.txt')
```

Output: [`pcs_fit_constrained.png`]

```
ax     | 1E-32 m^3 :    -8.152
rh     | 1E-32 m^3 :    -4.911
x      |   1E-10 m :    25.786
y      |   1E-10 m :     9.515
z      |   1E-10 m :     6.558
a      |       deg :   125.841
b      |       deg :   142.287
g      |       deg :    41.758
mueff  |        Bm :     0.000
shift  |       ppm :     0.000
B0     |         T :    18.790
temp   |         K :   298.150
t1e    |        ps :     0.000
```

To fit an axially symmetric tensor, we can used the Non-linear regression method and specify exactly which parameters we want to fit. This will be the axiality `ax`, two Euler angles `b` and `g` and the position coordinates. Note that in the output, the rhombic `rh` and alpha `a` parameters are redundant.

```
# Calculate axially symmetric tensor by NRL
mFitAx, calcAx, qfacAx = fit.nlr_fit_metal_from_pcs(
        [mStart], [parsedData], params=('ax','b','g','x','y','z'))

mFitAx[0].save('calbindin_Er_HN_PCS_tensor_axially_symmetric.txt')
```

Output: [`pcs_fit_constrained.png`]

```
ax     | 1E-32 m^3 :     9.510
rh     | 1E-32 m^3 :     0.000
x      |   1E-10 m :    24.948
```

(continues on next page)

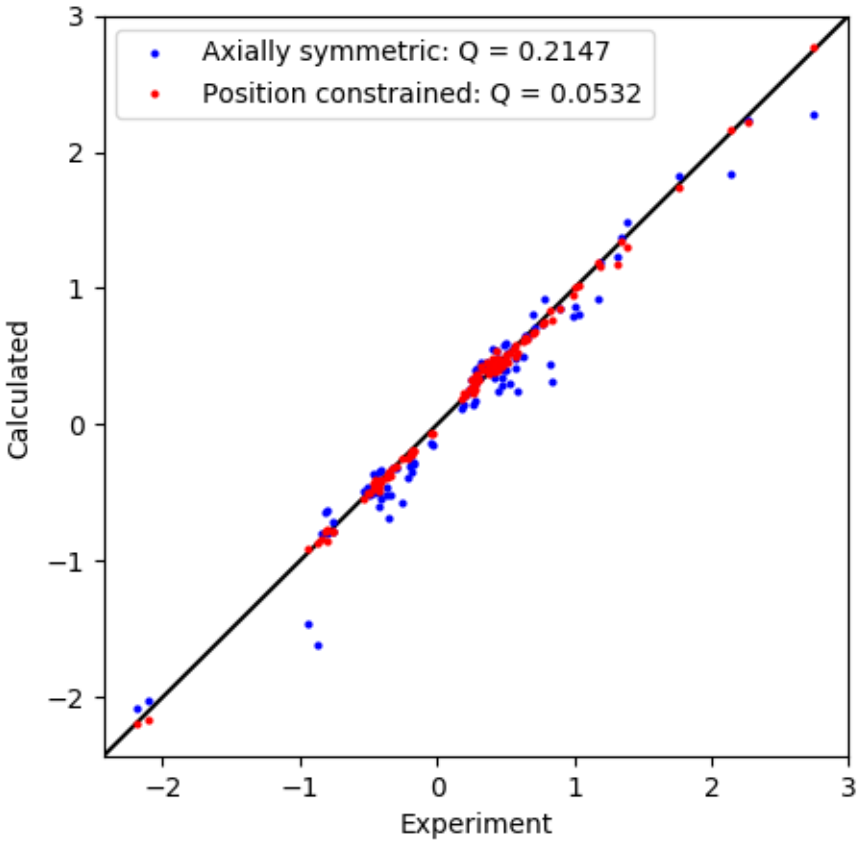
(continued from previous page)

```
y      |   1E-10 m :     8.992
z      |   1E-10 m :     3.205
a      |       deg :     0.000
b      |       deg :   134.697
g      |       deg :   180.000
mueff  |        Bm :     0.000
shift  |       ppm :     0.000
B0     |         T :    18.790
temp   |         K :   298.150
t1e    |        ps :     0.000
```

Finally we plot the data.

```
#### Plot the correlation ####
from matplotlib import pyplot as plt
fig, ax = plt.subplots(figsize=(5,5))

# Unpack the experimental values
atoms, experiment, errors = zip(*parsedData)

# Plot the data
ax.plot(experiment, calc[0], marker='o', lw=0, ms=2, c='r',
        label="Position constrained: Q = {:5.4f}".format(qfac[0]))

ax.plot(experiment, calcAx[0], marker='o', lw=0, ms=2, c='b',
        label="Axially symmetric: Q = {:5.4f}".format(qfacAx[0]))

# Plot a diagonal
l, h = ax.get_xlim()
ax.plot([l,h],[l,h],'-k',zorder=0)
ax.set_xlim(l,h)
ax.set_ylim(l,h)

# Make axis labels and save figure
ax.set_xlabel("Experiment")
ax.set_ylabel("Calculated")
ax.legend()
fig.savefig("pcs_fit_constrained.png")
```

Output: [pcs_fit_constrained.png]



Fit a tensor to PCS data with uncertainties

This example shows how to conduct a weighted fit of a $\Delta\chi$-tensor to experimental PCS data with experimental errors.

Downloads

- Download the data files `4icbH_mut.pdb` and `calbindin_Er_HN_PCS_errors.npc` from here:
- Download the script `pcs_fit_error.py`

Script + Explanation

This script follows very closely the script *Fit Tensor to PCS Data*. The only difference being that errors are included in the fourth column of the .npc file and errorbars are included in the plotting routine.

```
from paramagpy import protein, fit, dataparse, metal

# Load the PDB file
prot = protein.load_pdb('../data_files/4icbH_mut.pdb')

# Load the PCS data
rawData = dataparse.read_pcs('../data_files/calbindin_Er_HN_PCS_errors.npc')

# Associate PCS data with atoms of the PDB
```

(continues on next page)

(continued from previous page)

```
parsedData = prot.parse(rawData)

# Define an initial tensor
mStart = metal.Metal()

# Set the starting position to an atom close to the metal
mStart.position = prot[0]['A'][56]['CA'].position

# Calculate an initial tensor from an SVD gridsearch
mGuess, calc, qfac = fit.svd_gridsearch_fit_metal_from_pcs(
        [mStart],[parsedData], radius=10, points=10)

# Refine the tensor using non-linear regression
mFit, calc, qfac = fit.nlr_fit_metal_from_pcs(mGuess, [parsedData])

# Save the fitted tensor to file
mFit[0].save('calbindin_Er_HN_PCS_tensor_errors.txt')

#### Plot the correlation ####
from matplotlib import pyplot as plt
fig, ax = plt.subplots(figsize=(5,5))

# Unpack the experimental values
atoms, experiment, errors = zip(*parsedData)

# Plot the data
ax.errorbar(experiment, calc[0], xerr=errors, fmt='o', c='r', ms=2,
        ecolor='k', capsize=3, label="Q-factor = {:5.4f}".format(qfac[0]))

# Plot a diagonal
l, h = ax.get_xlim()
ax.plot([l,h],[l,h],'grey',zorder=0)
ax.set_xlim(l,h)
ax.set_ylim(l,h)

# Make axis labels and save figure
ax.set_xlabel("Experiment")
ax.set_ylabel("Calculated")
ax.legend()
fig.savefig("pcs_fit_error.png")
```

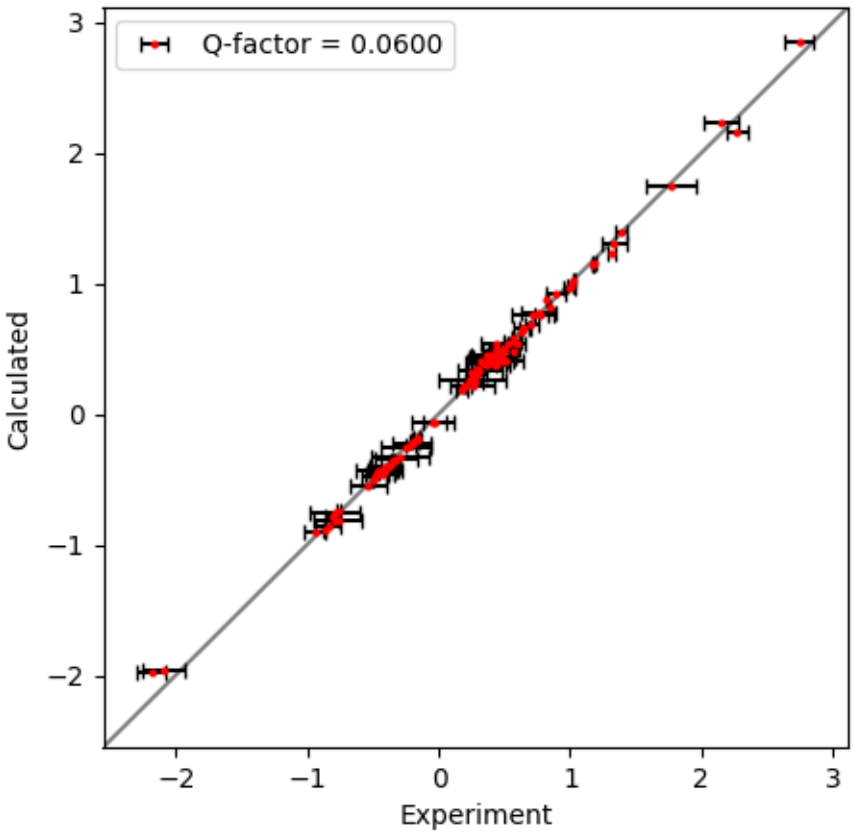
The fitted tensor:

Output: [calbindin_Er_HN_PCS_tensor_errors.txt]

```
ax     | 1E-32 m^3 :    -8.012
rh     | 1E-32 m^3 :    -4.125
x      |   1E-10 m :    24.892
y      |   1E-10 m :     8.456
z      |   1E-10 m :     6.287
a      |       deg :   112.440
b      |       deg :   135.924
g      |       deg :    46.210
mueff  |        Bm :     0.000
shift  |       ppm :     0.000
B0     |         T :    18.790
temp   |         K :   298.150
t1e    |        ps :     0.000
taur   |        ns :     0.000
```

And correlation plot:

Output: [pcs_fit_error.png]



Fit to PCS data with offset, RACS and RADS corrections

5.2.2 RDC data

Fit Tensor to RDC Data

This example shows how to fit a $\Delta\chi$-tensor or equivalently, and alignment tensor to experimental RDC data. These data are taken from a Tb3+ tagged ubiquitin mutant:

Benjamin J. G. Pearce, Shereen Jabar, Choy-Theng Loh, Monika Szabo, Bim Graham, Gottfried Otting (2017) Structure restraints from heteronuclear pseudocontact shifts generated by lanthanide tags at two different sites *J. Biomol. NMR* 68:19-32

Downloads

- Download the data files `2kox.pdb`, `ubiquitin_a28c_c1_Tb_HN.rdc` and `ubiquitin_s57c_c1_Tb_HN.rdc` from here:
- Download the script `rdc_fit.py`

Script + Explanation

Firstly, the necessary modules are imported from paramagpy. And the two RDC datasets are loaded. Because this PDB contains over 600 models, loading may take a few seconds

```
from paramagpy import protein, fit, dataparse, metal

# Load the PDB file
prot = protein.load_pdb('../data_files/2kox.pdb')

# Load the RDC data
rawData1 = dataparse.read_rdc('../data_files/ubiquitin_a28c_c1_Tb_HN.rdc')
rawData2 = dataparse.read_rdc('../data_files/ubiquitin_s57c_c1_Tb_HN.rdc')

# Associate RDC data with atoms of the PDB
parsedData1 = prot.parse(rawData1)
parsedData2 = prot.parse(rawData2)
```

Two starting metals are initialised. It is important here to set the magnetic field strength and temperature.

```
# Define an initial tensor
mStart1 = metal.Metal(B0=18.8, temperature=308.0)
mStart2 = metal.Metal(B0=18.8, temperature=308.0)
```

The alignment tensor is solved using the function *paramagpy.fit.svd_fit_metal_from_rdc()* which return a tuple of `(metal, calculated, qfactor)`, where metal is the fitted metal, calculated is the calculated RDC values. These tensor are then saved.

```
# Calculate the tensor using SVD
sol1 = fit.svd_fit_metal_from_rdc(mStart1, parsedData1)
sol2 = fit.svd_fit_metal_from_rdc(mStart2, parsedData2)

# Save the fitted tensor to file
sol1[0].save('ubiquitin_a28c_c1_Tb_tensor.txt')
sol2[0].save('ubiquitin_s57c_c1_Tb_tensor.txt')
```

Output: [ubiquitin_a28c_c1_Tb_tensor.txt]

```
ax     | 1E-32 m^3 :    -4.776
rh     | 1E-32 m^3 :    -1.397
x      |   1E-10 m :     0.000
y      |   1E-10 m :     0.000
z      |   1E-10 m :     0.000
a      |       deg :    16.022
b      |       deg :    52.299
g      |       deg :    83.616
mueff  |        Bm :     0.000
shift  |       ppm :     0.000
B0     |         T :    18.800
temp   |         K :   308.000
t1e    |        ps :     0.000
taur   |        ns :     0.000
```

Output: [ubiquitin_s57c_c1_Tb_tensor.txt]

```
ax     | 1E-32 m^3 :    -5.930
rh     | 1E-32 m^3 :    -1.899
x      |   1E-10 m :     0.000
y      |   1E-10 m :     0.000
z      |   1E-10 m :     0.000
a      |       deg :     9.976
b      |       deg :    99.463
g      |       deg :    37.410
mueff  |        Bm :     0.000
shift  |       ppm :     0.000
B0     |         T :    18.800
temp   |         K :   308.000
```

(continues on next page)

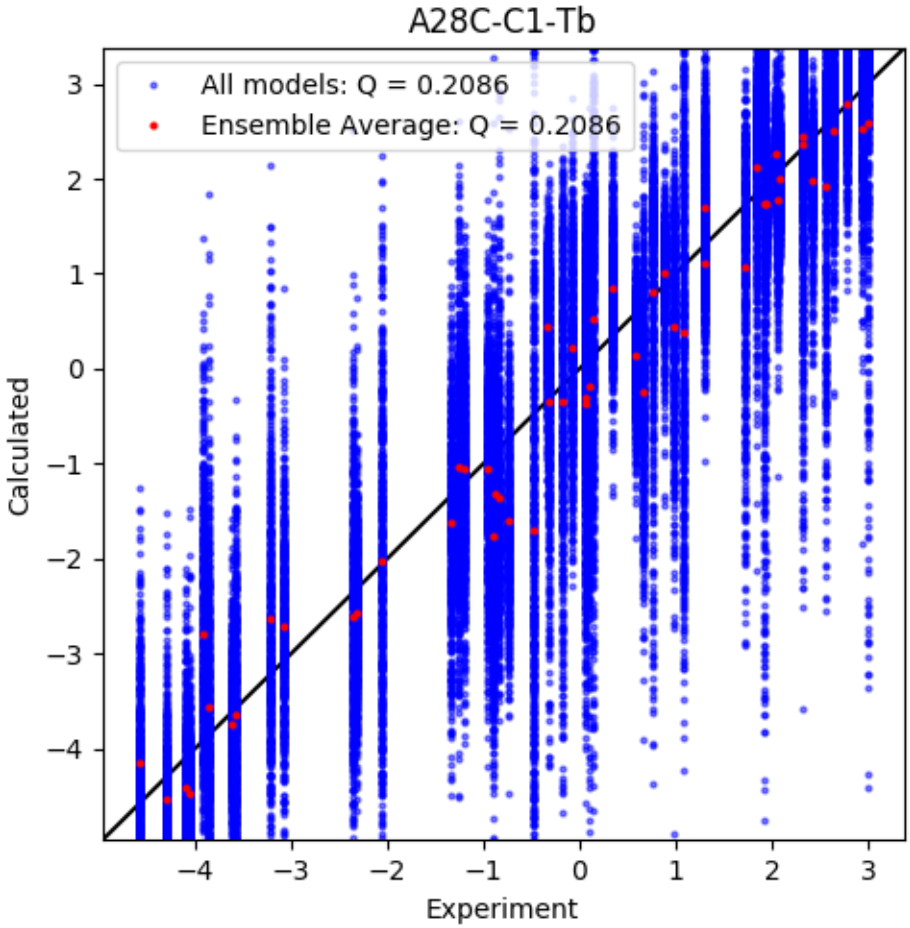
(continued from previous page)

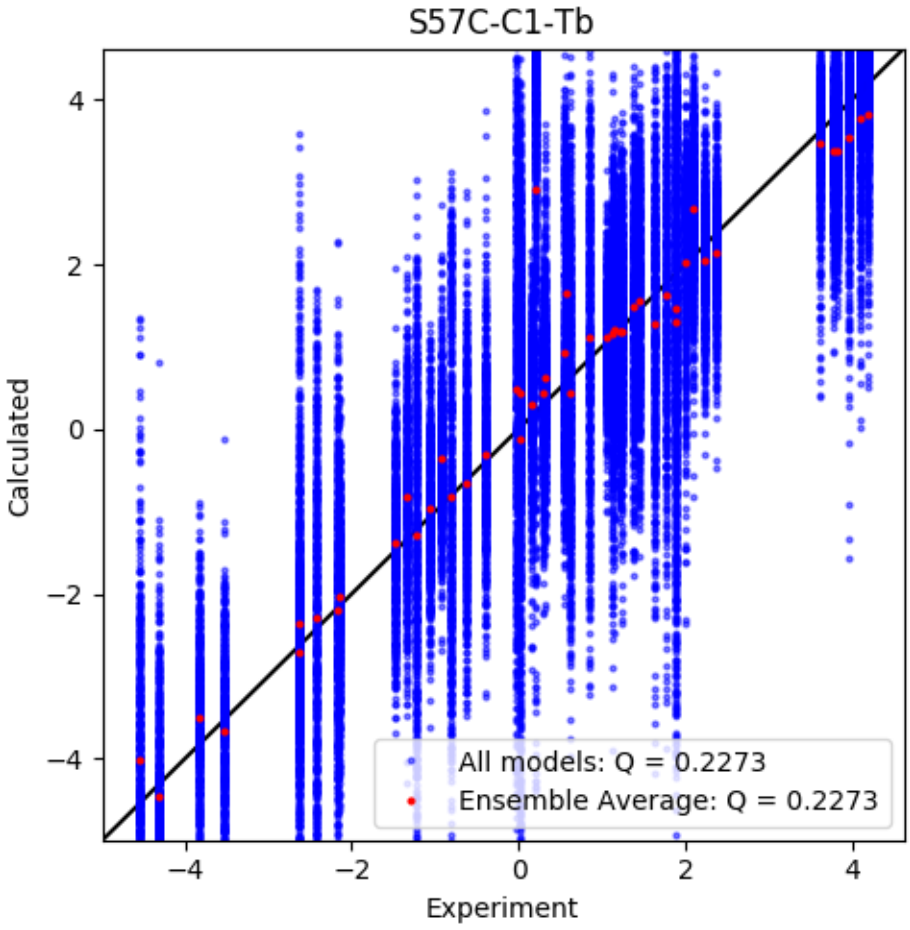
```
t1e   |        ps :     0.000
taur  |        ns :     0.000
```

The experimental/calculated correlations are then plotted. The tensor is by default fitted to the ensemble averaged calculated values. Backcalculation of all models is shown here, as well as the ensemble average.

```
#### Plot the correlation ####
from matplotlib import pyplot as plt
fig = plt.figure(figsize=(5,10))
ax1 = fig.add_subplot(211)
ax2 = fig.add_subplot(212)
ax1.set_title('A28C-C1-Tb')
ax2.set_title('S57C-C1-Tb')

for sol, ax, pdata in zip([sol1,sol2], [ax1,ax2],
        [parsedData1,parsedData2]):

        # Unpack the experimental values
        atoms1, atoms2, exp, err = zip(*pdata)
        atoms = list(zip(atoms1, atoms2))
        metal, calc, qfac = sol
        expEnsemble, calcEnsemble = fit.ensemble_average(atoms, exp, calc)

        # Plot all models
        ax.plot(exp, calc, marker='o', lw=0, ms=2, c='b',
                alpha=0.5, label="All models: Q = {:5.4f}".format(qfac))

        # Plot the ensemble average
        ax.plot(expEnsemble, calcEnsemble, marker='o', lw=0, ms=2, c='r',
                label="Ensemble Average: Q = {:5.4f}".format(qfac))

        # Plot a diagonal
        l, h = ax.get_xlim()
        ax.plot([l,h],[l,h],'-k',zorder=0)
        ax.set_xlim(l,h)
        ax.set_ylim(l,h)

        # Make axis labels and save figure
        ax.set_xlabel("Experiment")
        ax.set_ylabel("Calculated")
        ax.legend()

fig.tight_layout()
fig.savefig("rdc_fit.png")
```

Output: [rdc_fit.png]


A28C-C1-Tb
All models: Q = 0.2086
Ensemble Average: Q = 0.2086
Calculated
Experiment
3
2
1
0
−1
−2
−3
−4
−4
−3
−2
−1
0
1
2
3


S57C-C1-Tb
All models: Q = 0.2273
Ensemble Average: Q = 0.2273
Calculated
Experiment
4
2
0
−2
−4
−4
−2
0
2
4

Calculate RDC from a known Tensor

This example shows how to calculate theoretical RDC values from a known $\Delta\chi$-tensor which has been fitted from PCS data. Paramagpy allows seamless calculation of one PCS/PRE/RDC/CCR effect from a tensor fitted from another effect.

Downloads

- Download the data files `4icbH_mut.pdb` and `calbindin_Er_HN_PCS_tensor.txt` from here:
- Download the script `rdc_calculate.py`

Script + Explanation

First the relevant modules are loaded, the protein is loaded and the metal is loaded from file. The magnetic field strength and temperature are also set.

```
from paramagpy import protein, metal

# Load the PDB file
prot = protein.load_pdb('../data_files/4icbH_mut.pdb')

# Load the fitted tensor
met = metal.load_tensor('../data_files/calbindin_Er_HN_PCS_tensor.txt')
met.B0 = 18.8
```

A loop is made over the atoms of the protein. The amide H and N atoms are selected and then the RDC value is calculated. Finally the formated data is appended to list `forFile`.

```
forFile = []
for atom in prot.get_atoms():
        if atom.name == 'H':
                residue = atom.parent
                seq = residue.id[1]
                if 'N' in residue:
                        H = atom
                        N = residue['N']
                        rdc = met.atom_rdc(H, N)
                        line = "{0:2d} {1:^3s} {2:2d} {3:^3s} {4:6.3f} 0.0\n".
↪format(
                                seq, H.name, seq, N.name, rdc)
                        forFile.append(line)
```

The formatted data is written to file:

```
with open("calbindin_Er_RDC_calc.rdc", 'w') as f:
        f.writelines(forFile)
```

Output: [`calbindin_Er_RDC_calc.rdc`]

```
 0  H   0  N  -1.724 0.0
 1  H   1  N  -6.196 0.0
 2  H   2  N  -4.993 0.0
 4  H   4  N  -0.922 0.0
 5  H   5  N   1.783 0.0
 6  H   6  N   0.280 0.0
 7  H   7  N  -1.906 0.0
 8  H   8  N   1.056 0.0
 9  H   9  N   0.713 0.0
```

(continues on next page)

(continued from previous page)

```
10  H  10  N   0.213 0.0
11  H  11  N  -0.881 0.0
12  H  12  N   2.712 0.0
13  H  13  N   0.614 0.0
14  H  14  N  -2.346 0.0
15  H  15  N   1.659 0.0
16  H  16  N   0.648 0.0
17  H  17  N   0.383 0.0
18  H  18  N   0.420 0.0
19  H  19  N  -7.863 0.0
21  H  21  N   0.973 0.0
22  H  22  N   1.026 0.0
23  H  23  N  -0.613 0.0
24  H  24  N  -5.847 0.0
25  H  25  N   1.761 0.0
26  H  26  N   6.470 0.0
27  H  27  N   5.541 0.0
28  H  28  N  -0.334 0.0
29  H  29  N   3.624 0.0
30  H  30  N   6.673 0.0
31  H  31  N   3.952 0.0
32  H  32  N   1.658 0.0
33  H  33  N   5.449 0.0
34  H  34  N   7.370 0.0
35  H  35  N   1.033 0.0
36  H  36  N   1.136 0.0
38  H  38  N  -7.378 0.0
39  H  39  N  -6.979 0.0
40  H  40  N  -4.810 0.0
41  H  41  N  -3.187 0.0
42  H  42  N   2.415 0.0
43  H  43  N   1.710 0.0
44  H  44  N  -5.977 0.0
45  H  45  N  -5.467 0.0
46  H  46  N   3.243 0.0
47  H  47  N   3.937 0.0
48  H  48  N   7.047 0.0
49  H  49  N   4.577 0.0
50  H  50  N   3.718 0.0
51  H  51  N   4.519 0.0
52  H  52  N   6.077 0.0
53  H  53  N   2.940 0.0
54  H  54  N   2.541 0.0
55  H  55  N  -7.493 0.0
56  H  56  N  -7.159 0.0
57  H  57  N   4.948 0.0
58  H  58  N  -1.078 0.0
59  H  59  N  -0.759 0.0
60  H  60  N   0.161 0.0
61  H  61  N  -1.132 0.0
62  H  62  N  -5.719 0.0
63  H  63  N   4.025 0.0
64  H  64  N   5.929 0.0
65  H  65  N   2.363 0.0
66  H  66  N   2.477 0.0
67  H  67  N   8.265 0.0
68  H  68  N   5.078 0.0
69  H  69  N   3.724 0.0
70  H  70  N   7.743 0.0
71  H  71  N   2.188 0.0
72  H  72  N   4.911 0.0
```

(continues on next page)

(continued from previous page)

```
73  H  73  N   7.514 0.0
74  H  74  N  -0.001 0.0
75  H  75  N   1.119 0.0
```

5.2.3 PRE data

Fit Tensor to PRE Data

This example demonstrates fitting of the rotational correlation time τ_r to 1H PRE data of calbindin D9k. You can fit any parameters of the χ-tensor you desire, such as position or magnitude as well.

Downloads

- Download the data files `4icbH_mut.pdb`, `calbindin_Er_H_R2_600.npc` and `calbindin_Tb_H_R2_800.npc` from here:
- Download the script `pre_fit_proton.py`

Script + Explanation

Firstly, the necessary modules are imported from paramagpy.

```
from paramagpy import protein, fit, dataparse, metal
```

The protein is then loaded from a PDB file.

```
# Load the PDB file
prot = protein.load_pdb('../data_files/4icbH_mut.pdb')
```

The PRE data is loaded. Note that the Er data was recorded at 600 MHz and the Tb data was recorded at 800 MHz.

```
rawData_er = dataparse.read_pre('../data_files/calbindin_Er_H_R2_600.pre')
rawData_tb = dataparse.read_pre('../data_files/calbindin_Tb_H_R2_800.pre')
```

The $\Delta\chi$-tensors that were fitted from PCS data are loaded from file and the relevant B_0 magnetic field strengths are set.

```
mStart_er = metal.load_tensor('../data_files/calbindin_Er_HN_PCS_tensor.txt')
mStart_tb = metal.load_tensor('../data_files/calbindin_Tb_HN_PCS_tensor.txt')
mStart_er.B0 = 14.1
mStart_tb.B0 = 18.8
```

Fitting of the rotational correlation time is done with the function *paramagpy.fit.nlr_fit_metal_from_pre()*. To fit position or χ-tensor magnitude, you can change the `params` argument.

```
(m_er,), (cal_er,), qfac = fit.nlr_fit_metal_from_pre(
        [mStart_er], [data_er], params=['taur'], rtypes=['r2'])
(m_tb,), (cal_tb,), qfac = fit.nlr_fit_metal_from_pre(
        [mStart_tb], [data_tb], params=['taur'], rtypes=['r2'])
```

The fitted tensors are saved to file. Note that the Er dataset gives a reasonable τ_r of around 4 ns which is close to the literature value of 4.25 ns. However, the Tb dataset gives an unreasonably large value of 18 ns. This is due to magnetisation attenuation due to 1H-1H RDCs present during the relaxation evolution time as discussed in literature giving rise to artificially large measured PREs for lanthanides with highly anisotropic $\Delta\chi$-tensors. This is also reflected in the correlation plot below.

```
m_er.save('calbindin_Er_H_R2_600_tensor.txt')
m_tb.save('calbindin_Tb_H_R2_800_tensor.txt')
```

Output: [calbindin_Er_H_R2_600_tensor.txt]

```
ax     | 1E-32 m^3 :    -8.152
rh     | 1E-32 m^3 :    -4.911
x      |   1E-10 m :    25.786
y      |   1E-10 m :     9.515
z      |   1E-10 m :     6.558
a      |       deg :   125.841
b      |       deg :   142.287
g      |       deg :    41.758
mueff  |        Bm :     9.581
shift  |       ppm :     0.000
B0     |         T :    14.100
temp   |         K :   298.150
t1e    |        ps :     0.189
taur   |        ns :     3.923
```

Output: [calbindin_Tb_H_R2_800_tensor.txt]

```
ax     | 1E-32 m^3 :    30.375
rh     | 1E-32 m^3 :    12.339
x      |   1E-10 m :    25.786
y      |   1E-10 m :     9.515
z      |   1E-10 m :     6.558
a      |       deg :   150.957
b      |       deg :   152.671
g      |       deg :    70.311
mueff  |        Bm :     9.721
shift  |       ppm :     0.000
B0     |         T :    18.800
temp   |         K :   298.150
t1e    |        ps :     0.251
taur   |        ns :    18.917
```

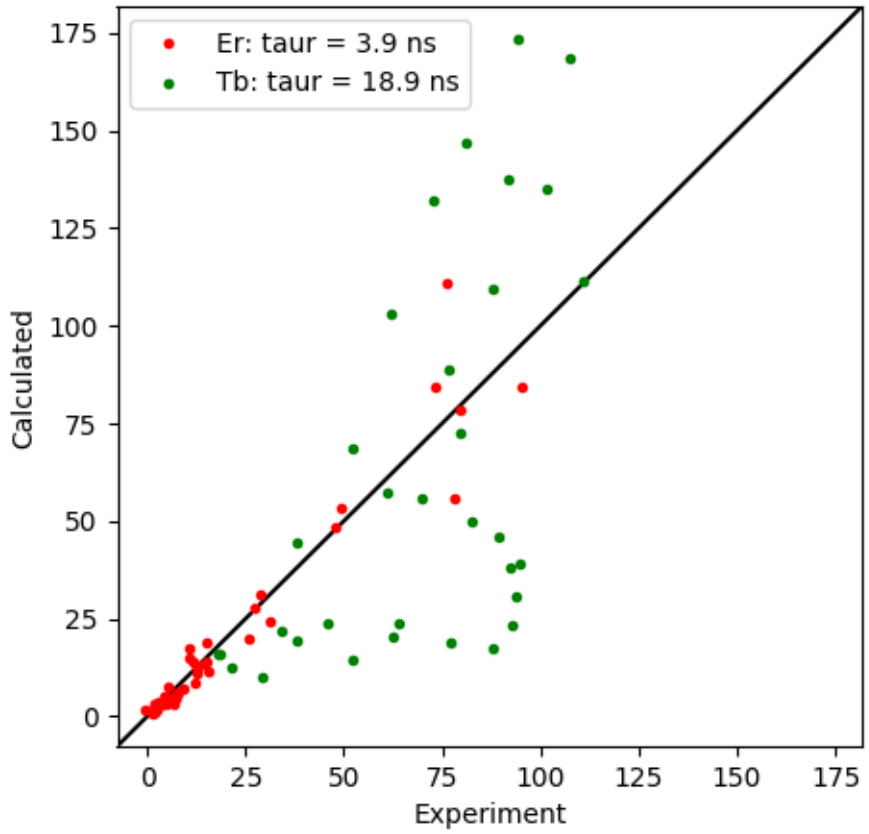
And the results are plotted.

```
from matplotlib import pyplot as plt
fig, ax = plt.subplots(figsize=(5,5))

# Unpack the experimental values
atm_er, exp_er, err_er = zip(*data_er)
atm_tb, exp_tb, err_tb = zip(*data_tb)

# Plot the data
ax.plot(exp_er, cal_er, marker='o', lw=0, ms=3, c='r',
        label="Er: taur = {:3.1f} ns".format(1E9*m_er.taur))
ax.plot(exp_tb, cal_tb, marker='o', lw=0, ms=3, c='g',
        label="Tb: taur = {:3.1f} ns".format(1E9*m_tb.taur))

# Plot a diagonal
l, h = ax.get_ylim()
ax.plot([l,h],[l,h],'-k',zorder=0)
ax.set_xlim(l,h)
ax.set_ylim(l,h)

# Make axis labels and save figure
ax.set_xlabel("Experiment")
ax.set_ylabel("Calculated")
ax.legend()
```

(continues on next page)

(continued from previous page)

```
fig.savefig("pre_fit_proton.png")
```



Calculate 15N PREs with cross-correlation effects

This example shows how to conduct a weighted fit of a $\Delta\chi$-tensor to experimental PCS data with experimental errors.

Downloads

- Download the data files `4icbH_mut.pdb`, `calbindin_Tb_N_R1_600.pre` and `calbindin_Tb_HN_PCS_tensor.txt` from here:
- Download the script `pre_calc_nitrogen.py`

Script + Explanation

First the relevant modules are loaded, the protein and data are read and the data is parsed by the protein.

```
from paramagpy import protein, metal, dataparse

# Load the PDB file
prot = protein.load_pdb('../data_files/4icbH_mut.pdb')
```

(continues on next page)

(continued from previous page)

```
# Load PRE data
rawData = dataparse.read_pre('../data_files/calbindin_Tb_N_R1_600.pre')

# Parse PRE data
data = prot.parse(rawData)
```

The Tb tensor fitted from PCS data is loaded and the relevant parameters, in this case the magnetic field strength, temperature and rotational correlation time are set.

```
met = metal.load_tensor('../data_files/calbindin_Tb_HN_PCS_tensor.txt')
met.B0 = 14.1
met.T = 298.0
met.taur = 4.25E-9
```

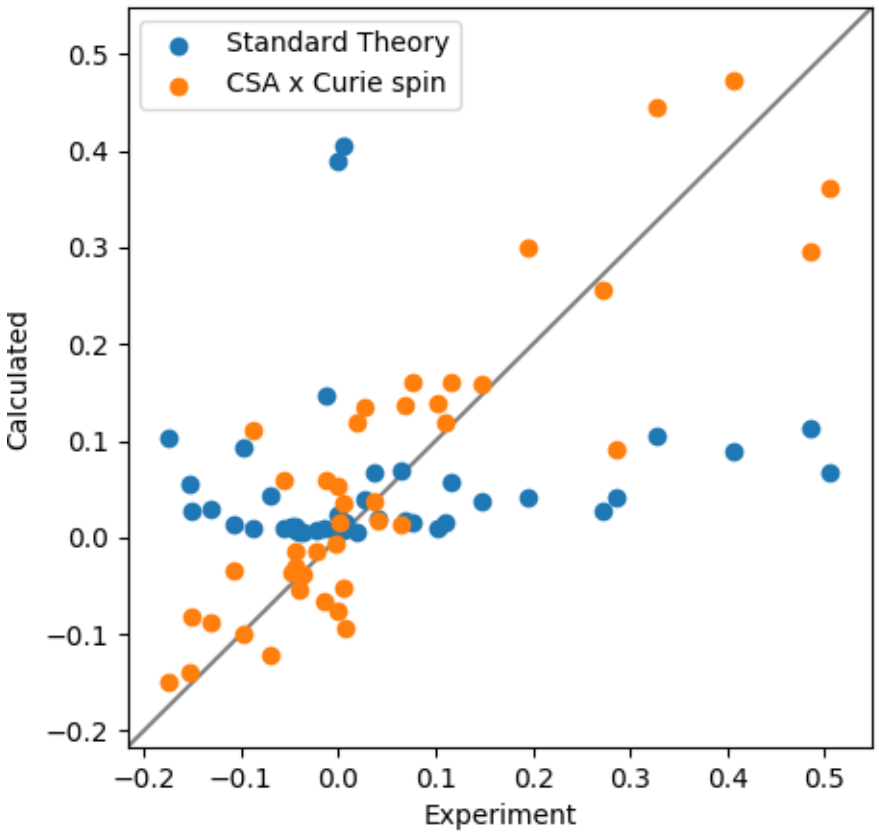
A loop is conducted over the nitrogen atoms that are present in the experimental data. The PRE is calculated using the function `paramagpy.metal.atom_pre()`. Calculations without CSA are appended to the list `cal` and calculations including CSA cross-correlation with the Curie-spin relaxation are appended to the list `cal_csa`.

```
exp = []
cal = []
cal_csa = []
for atom, pre, err in data:
        exp.append(pre)
        cal.append(met.atom_pre(atom, rtype='r1'))
        cal_csa.append(met.atom_pre(atom, rtype='r1', csa=atom.csa))
```

Finally the data are plotted. Clearly CSA cross-correlation is a big effect for backbone nitrogen atoms and should always be taken into account for Curie-spin calculations. Also note the existence and correct prediction of negative PREs!

```
from matplotlib import pyplot as plt
fig, ax = plt.subplots(figsize=(5,5))

# Plot the data
ax.scatter(exp, cal, label="Standard Theory")
ax.scatter(exp, cal_csa, label="CSA x Curie spin")

# Plot a diagonal
l, h = ax.get_xlim()
ax.plot([l,h],[l,h],'grey',zorder=0)
ax.set_xlim(l,h)
ax.set_ylim(l,h)

# Make axis labels and save figure
ax.set_xlabel("Experiment")
ax.set_ylabel("Calculated")
ax.legend()
fig.savefig("pre_calc_nitrogen.png")
```

Output: [`pre_calc_nitrogen.png`]



Fit spectral power density tensor

This example shows how to fit the spectral power density tensor to anisotropic PREs. The data and theory are derived from https://doi.org/10.1039/C8CP01332B.

Downloads

- Download the data files `parashift_Tb.pdb` and `parashift_Tb_R1_exp.pre` from here:
- Download the script `pre_fit_aniso_dipolar.py`

Script + Explanation

Load the relevant modules, read the PDB coordinates and experimental PRE values. Parse the values.

```
from paramagpy import protein, metal, fit, dataparse
from matplotlib import pyplot as plt
import numpy as np

prot = protein.load_pdb('../data_files/parashift_Tb.pdb')
pre_exp = dataparse.read_pre('../data_files/parashift_Tb_R1_exp.pre')
exp = prot.parse(pre_exp)
```

The spectral power density tensor is written here explicitly and set to the attribute `g_tensor`. The values here are sourced from the original paper, and arise from the robust linear fit to the experimental data. We will use this tensor for comparison to the fit achieved by paramagpy.

```
m = metal.Metal(taur=0.42E-9, B0=1.0, temperature=300.0)
m.set_lanthanide('Tb')

m.g_tensor = np.array([
        [1754.0, -859.0, -207.0],
        [-859.0, 2285.0, -351.0],
        [-207.0, -351.0, -196.0]]) * 1E-60
```

An starting tensor with no parameters is also initialised and will be used for fitting to the exerimental data with paramagpy.

```
m0 = metal.Metal(taur=0.42E-9, B0=1.0, temperature=300.0)
m0.set_lanthanide('Tb')
```

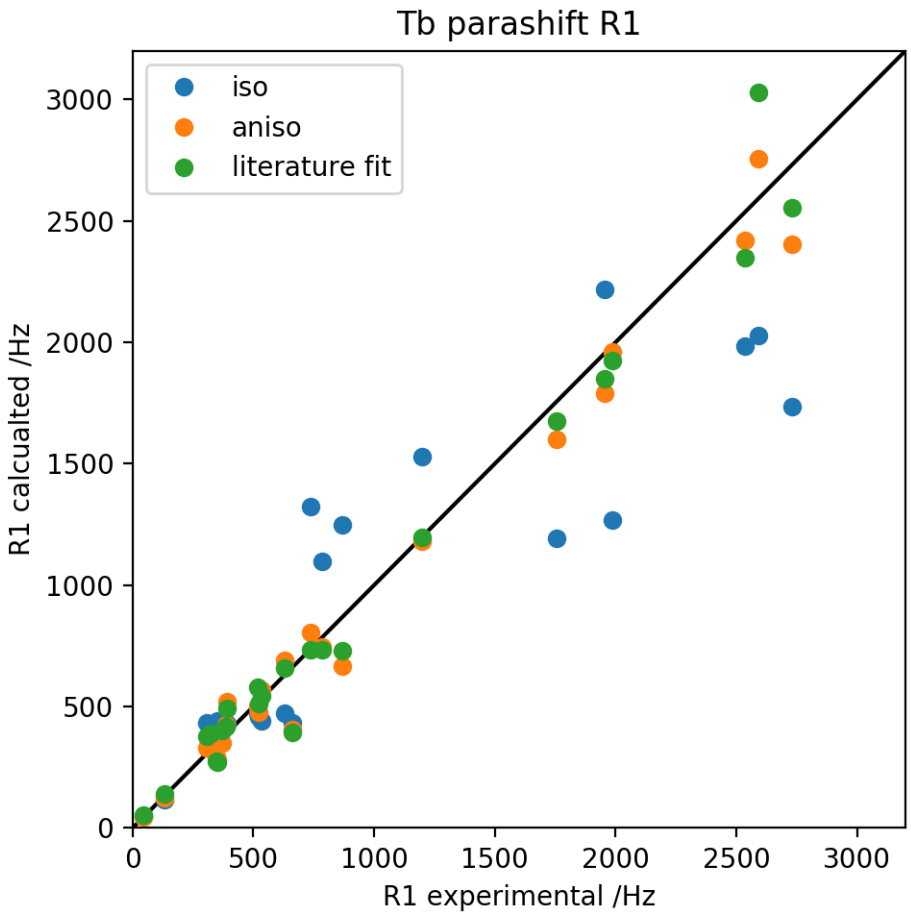
The fit is conducted by setting the `usegsbm` flag to `True`. This uses anisotropic SBM theory to fit the spectral power density tensor in place of the isotropic SBM theory. The relevant fitting parameters must be specified as `'t1e', 'gax', 'grh', 'a','b','g'` which represent the electronic relaxation time, the axial and rhombic componenets of the power spectral density tensor and the 3 Euler angles alpha, beta and gamma respectively. Note that the fitted `t1e` parameter is only an estimate of the electronic relaxation time.

```
[mfit], [calc], [q] = fit.nlr_fit_metal_from_pre([m0], [exp], params=('t1e', 'gax',
↪ 'grh', 'a','b','g'),
```

Finally the results of the fit are plotted alongside the isotropic theory and the literature fit. Note that the difference in the fit from paramagpy is small, and probably arises because the original paper uses a *Robust* linear fit, which may include weighting with experimental uncertainties. However paramagpy weights values evely here because the experimental uncertainties are unknown.

```
atoms, r1, err = zip(*exp)
pos = np.array([a.position for a in atoms])
gam = np.array([a.gamma for a in atoms])

fig = plt.figure(figsize=(5,5))
ax = fig.add_subplot(111)
ax.plot([0,3200],[0,3200], '-k')
ax.plot(r1, mfit.fast_sbm_r1(pos, gam), marker='o', lw=0, label='iso')
ax.plot(r1, mfit.fast_g_sbm_r1(pos, gam), marker='o', lw=0, label='aniso')
ax.plot(r1, m.fast_g_sbm_r1(pos, gam), marker='o', lw=0, label='literature fit')
ax.set_xlim(0,3200)
ax.set_ylim(0,3200)
ax.set_xlabel("R1 experimental /Hz")
ax.set_ylabel("R1 calcualted /Hz")
ax.set_title("Tb parashift R1")
ax.legend()
fig.tight_layout()
fig.savefig("pre_fit_aniso_dipolar.png", dpi=200)
```

Output: [pre_fit_aniso_dipolar.png]



5.2.4 CCR data

Calculate Cross-correlated Relaxation

This example shows how to calculate dipole-dipole/Curie-spin cross-correlated relaxation as measured for data in the literature by Pintacuda et. al.

Downloads

- Download the data files `1bzrH.pdb`, `myoglobin_cn.ccr` and `myoglobin_f.ccr` from here:
- Download the script `ccr_calculate.py`

Script + Explanation

First the relevant modules are loaded, and the iron atom (paramagnetic centre) is identified as the variable `ironAtom`.

```
from paramagpy import protein, fit, dataparse, metal
import numpy as np

# Load the PDB file and get iron centre
prot = protein.load_pdb('../data_files/1bzrH.pdb')
ironAtom = prot[0]['A'][("H_HEM",154," ")]['FE']
```

Two paramagnetic centres are defined for the high and low spin iron atom. The positions are set to that of the iron centre along with other relevant parameters. The measured isotropic χ-tensor magnitudes are also set.

```
met_cn = metal.Metal(position=ironAtom.position,
                                          B0=18.79,
                                          temperature=303.0,
                                          taur=5.7E-9)
met_f = met_cn.copy()
met_cn.iso = 4.4E-32
met_f.iso = 30.1E-32
```

The experimental data are loaded and parsed by the protein.

```
data_cn = prot.parse(dataparse.read_ccr("../data_files/myoglobin_cn.ccr"))
data_f = prot.parse(dataparse.read_ccr("../data_files/myoglobin_f.ccr"))
```

A loop is conducted over the atoms contained in the experimental data and the CCR rate is calculated using the function *paramagpy.metal.Metal.atom_ccr()*. These are appended to lists `compare_cn` and `compare_f`.

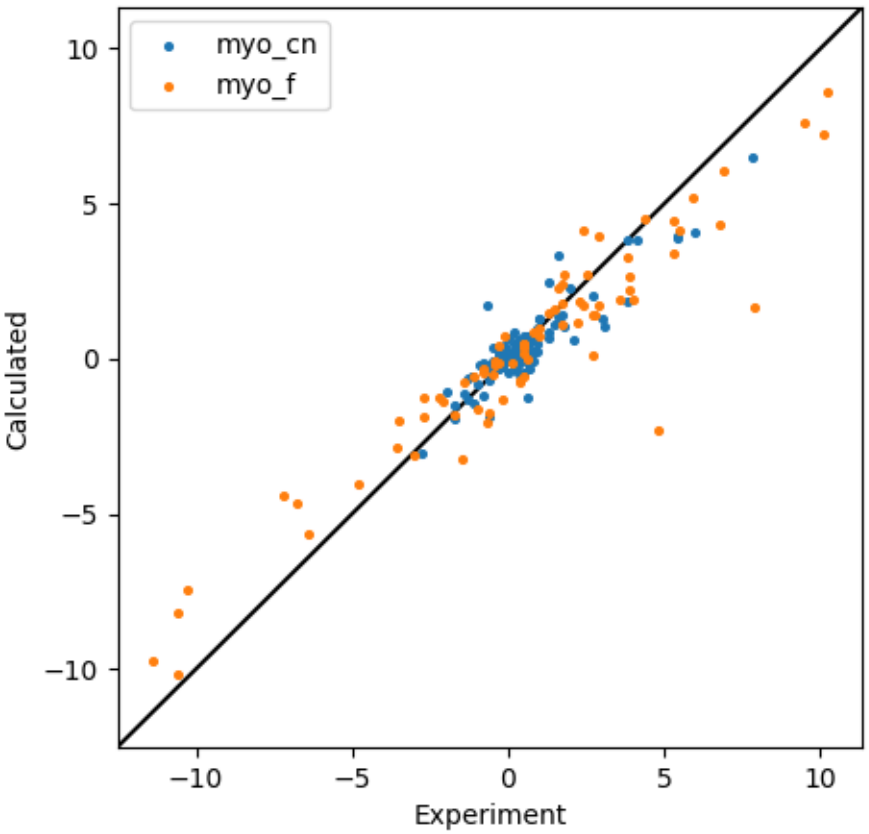
Note that the two H and N atoms are provided. The first atom is the nuclear spin undergoing active relaxation. The second atom is the coupling partner. Thus by swapping the H and N atoms to give `atom_ccr(N, H)`, the differential line broadening can be calculated in the indirect dimension.

```
# Calculate the cross-correlated realxation
compare_cn = []
for H, N, value, error in data_cn:
        delta = met_cn.atom_ccr(H, N)
        compare_cn.append((value, delta*0.5))

compare_f = []
for H, N, value, error in data_f:
        delta = met_f.atom_ccr(H, N)
        compare_f.append((value, delta*0.5))
```

Finally a correlation plot is made.

```
#### Plot the correlation ####
from matplotlib import pyplot as plt
fig, ax = plt.subplots(figsize=(5,5))

# Plot the data correlations
ax.scatter(*zip(*compare_cn), s=7, label="myo_cn")
ax.scatter(*zip(*compare_f), s=7, label="myo_f")

# Plot a diagonal
l, h = ax.get_xlim()
ax.plot([l,h],[l,h],'-k',zorder=0)
ax.set_xlim(l,h)
ax.set_ylim(l,h)

# Make axis labels and save figure
ax.set_xlabel("Experiment")
ax.set_ylabel("Calculated")
ax.legend()
fig.savefig("ccr_calculate.png")
```

Output: [`ccr_calculate.png`]



5.3 Graphic User Interface (GUI)

Paramagpy is equipped with a GUI which is cross-platform and contains most of the functionality of the scripted module. This gives a rapid way for new users to fit and compare PCS, RDC and PRE effects.

5.3.1 YouTube Tutorial

Check out the tutorial on YouTube

5.3.2 Running the GUI

To run the GUI, first open the python inperpreter in the terminal

```
user@computer:~$ python3
Python 3.5.2 (default, Nov 23 2017, 16:37:01)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Then import paramagpy and start the gui with `paramagpy.gui.run()`.

```
user@computer:~$ python3
Python 3.5.2 (default, Nov 23 2017, 16:37:01)
[GCC 5.4.0 20160609] on linux
```

(continues on next page)

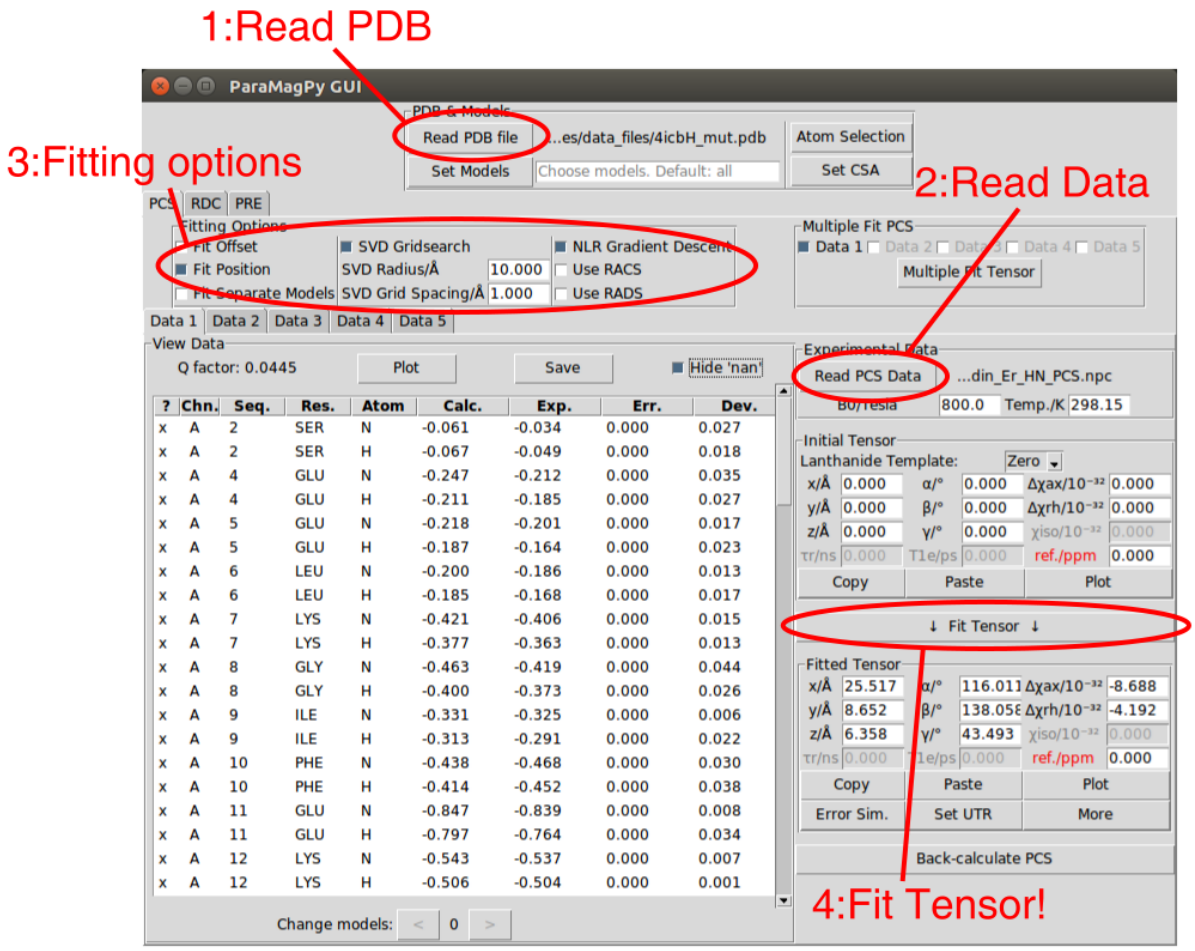
(continued from previous page)

```
Type "help", "copyright", "credits" or "license" for more information.
>>> import paramagpy
>>> paramagpy.gui.run()
```

Alternatively you can simply execute the following from the command line

```
user@computer:~$ echo "import paramagpy; paramagpy.gui.run()" | python3
```

If all this fails, you can contact the author for a prebuilt executable at henry.orton@anu.edu.au



5.4 NMR Software Macros

Paramagpy includes scripts for reading/writing PCS values directly from popular NMR software. This drastically improves the iterative process of tensor fitting.

5.4.1 CCPNMR Analysis 2.4

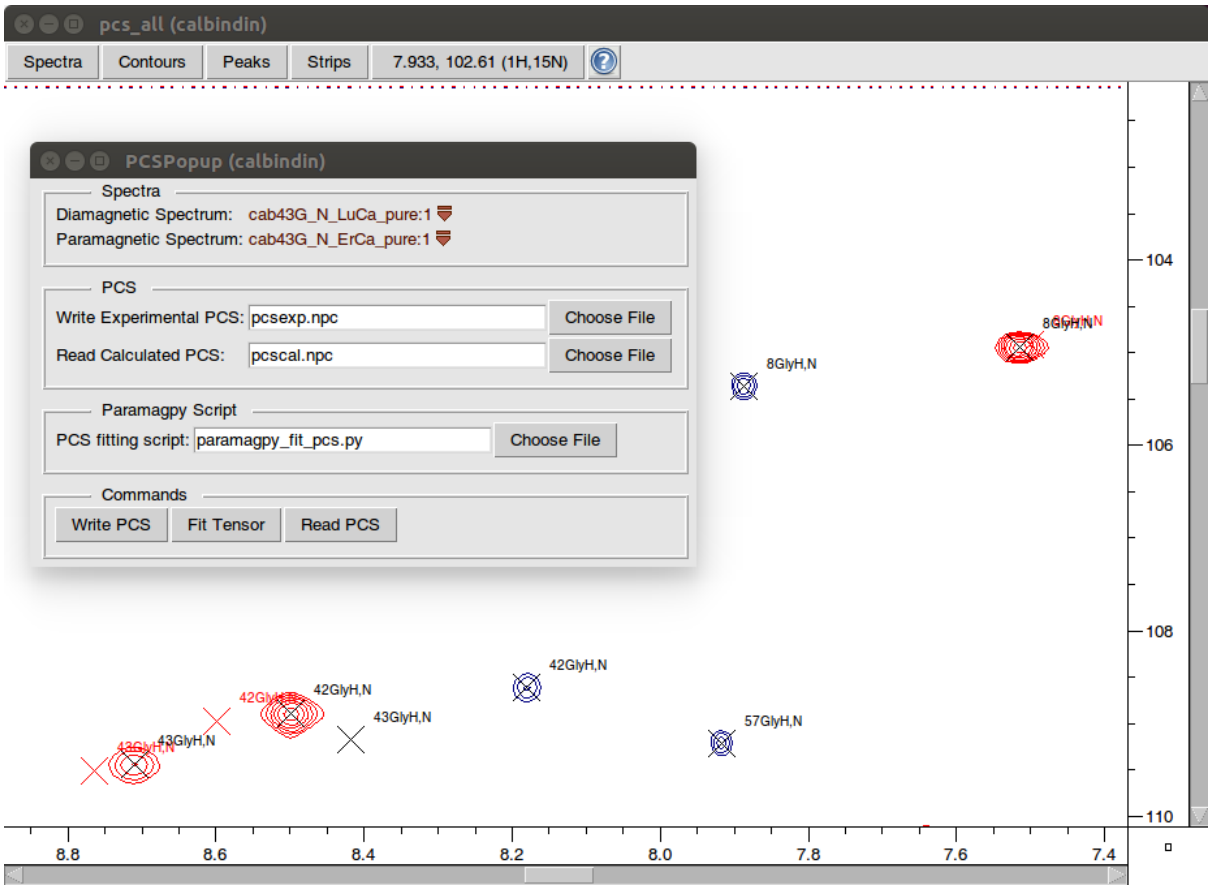
Download the two scripts:

- `paramagpy_ccpnmr_macro.py`
- `paramagpy_fit_pcs.py`

In the first line of the script *paramagpy_fit_pcs.py*, replace the shebang with the path to the python version on your machine that contains the paramagpy installation. On my computer this is set to.

```
#!/usr/bin/python3
```

Open CCPNMR analysis and navigate to Menu->Macro->Organise Macros. At the lower right click *Add Macro* and open the script *paramagpy_ccpnmr_macro.py*, then select *paramagpyMACRO* and click *Load Macro*. You can then select if from the list and click *Run* to reveal the screen below.



The popup window allows you to select a diamagnetic and paramagnetic spectrum and has 3 available buttons:

- Write PCS: This will calculate the difference between assigned peaks in the paramagnetic and diamagnetic spectra and write them to a .npc file (as specified in the relevant field).
- Fit Tensor: This will call the paramagpy script *paramagpy_fit_pcs.py* to fit the tensor the the written PCS values.
- Read PCS: This will read back-calculated PCS values from file (as specified in the relevant field) and plot the expected peaks on the paramagnetic spectrum in red.

Note, to alter the parameters for fitting of the PCS tensor, you can change the values within the script *paramagpy_fit_pcs.py*.

5.4.2 Sparky

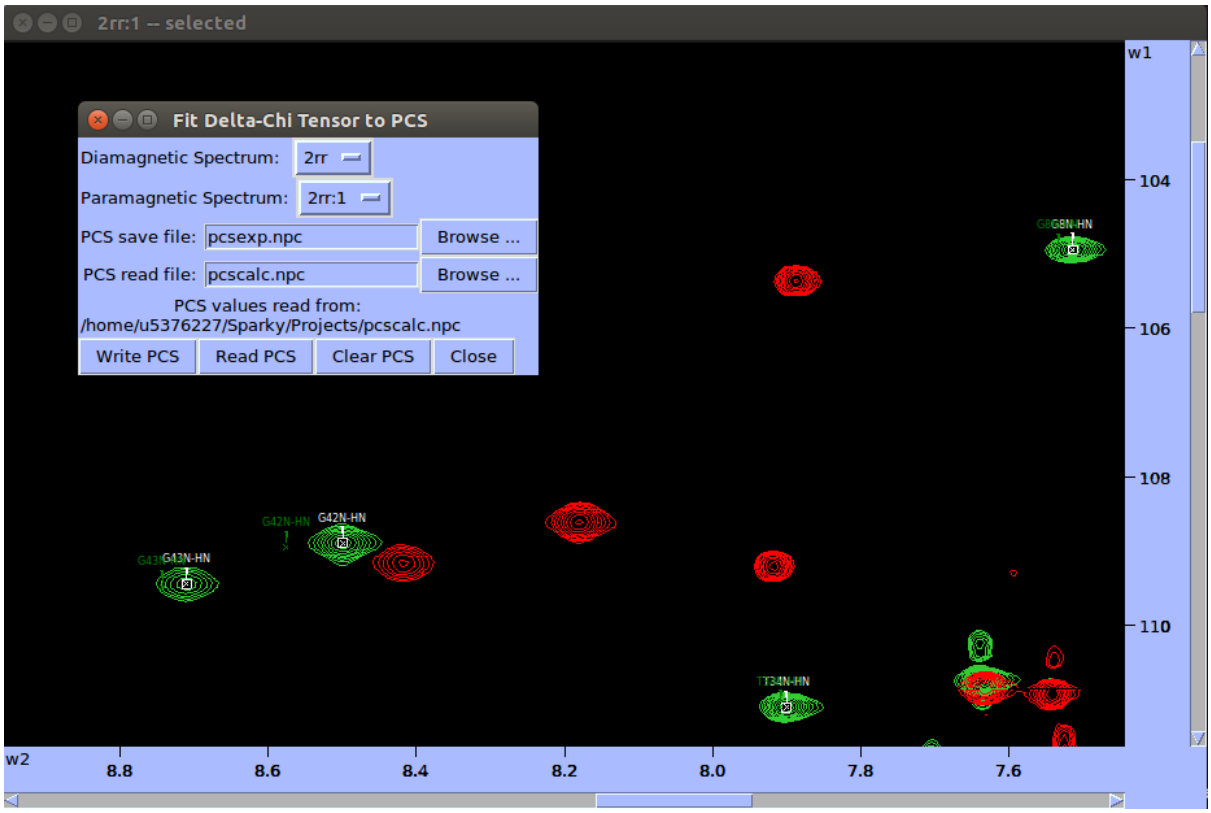
Download the 3 scripts:

- `paramagpy_sparky_macro.py`
- `sparky_init.py`
- `paramagpy_fit_pcs.py`

Place the first two scripts *paramagpy_sparky_macro.py* and *sparky_init.py* in the Sparky directory *~/Sparky/Python*. Note that the Sparky directory usually contains the *Projects*, *Lists* and *Save* folders. You may need to create the *Python* directory here in which to place the two scripts.

Place the third script *paramagpy_fit_pcs.py* in your home directory.

Open Sparky and navigate to Extensions->Read and write PCS files.



The popup window allows you to select a diamagnetic and paramagnetic spectrum and has 3 available buttons:

- Write PCS: This will calculate the difference between assigned peaks in the paramagnetic and diamagnetic spectra and write them to a .npc file (as specified in the relevant field).
- Read PCS: This will read back-calculated PCS values from file (as specified in the relevant field) and plot the expected peaks on the paramagnetic spectrum in green.
- Clear PCS: This will remove all calculated PCS peaks from the spectrum.

Note, to perform the tensor fitting, you will need to execute the paramagpy script in a separate terminal including an argument with the experimental PCS file such as:

```
user@computer:~$ ./paramagpy_fit_pcs.py pcsexp.npc
```

To alter the parameters for fitting of the PCS tensor, you can change the values within the script *paramagpy_fit_pcs.py*.

5.5 Reference Guide

5.5.1 Paramagnetic module

This module handles the paramagnetic centre by defining the magnetic susceptibility tensor and methods for PCS, RDC and PRE calculations.

paramagpy.metal

Functions

euler_to_matrix(eulers)	Calculate a rotation matrix from euler angles using ZYZ convention
matrix_to_euler(M)	Calculate Euler angles from a rotation matrix using ZYZ convention
unique_eulers(eulers)	Calculate Euler angles in unique tensor representation.
make_tensor(x, y, z, axial, rhombic, alpha, ...)	Make a ChiTensor isntance from given parameters.

paramagpy.metal.euler_to_matrix

paramagpy.metal.**euler_to_matrix**(*eulers*)

Calculate a rotation matrix from euler angles using ZYZ convention

Parameters **eulers** (*array of floats*) – the euler angles [alpha,beta,gamma] in radians by ZYZ convention.

Returns **matrix** – the rotation matrix

Return type 3x3 numpy ndarray

Examples

```
>>> eulers = np.array([0.5,1.2,0.8])
>>> euler_to_matrix(eulers)
array([[-0.1223669 , -0.5621374 ,  0.81794125],
       [ 0.75057357,  0.486796  ,  0.44684334],
       [-0.64935788,  0.66860392,  0.36235775]])
```

paramagpy.metal.matrix_to_euler

paramagpy.metal.**matrix_to_euler**(*M*)

Calculate Euler angles from a rotation matrix using ZYZ convention

Parameters **M** (*3x3 numpy ndarray*) – a rotation matrix

Returns **eulers** – the euler angles [alpha,beta,gamma] in radians by ZYZ convention

Return type array of floats

Examples

```
>>> matrix = array([[-0.1223669 , -0.5621374 ,  0.81794125],
                    [ 0.75057357,  0.486796  ,  0.44684334],
                    [-0.64935788,  0.66860392,  0.36235775]])
>>> matrix_to_euler(matrix)
np.array([0.5,1.2,0.8])
```

paramagpy.metal.unique_eulers

paramagpy.metal.**unique_eulers**(*eulers*)

Calculate Euler angles in unique tensor representation.

Given general Euler angles by ZYZ convention, this function accounts for the symmetry of a second rank symmetric tensor to map all angles within the range [0, pi].

Parameters **eulers** (*array of float*) – the three Euler angles in radians

Returns **eulers_utr** – the euler angles [alpha,beta,gamma] in radians by ZYZ convention

Return type array of floats

Examples

```
>>> eulers = np.array([-5.2,10.3,0.1])
>>> unique_eulers(eulers)
np.array([1.08318531 0.87522204 3.04159265])
```

paramagpy.metal.make_tensor

paramagpy.metal.**make_tensor**(*x*, *y*, *z*, *axial*, *rhombic*, *alpha*, *beta*, *gamma*, *lanthanide=None*, *temperature=298.15*)

Make a ChiTensor isntance from given parameters. This is designed to use pdb coordinates (x, y, z) and euler angles from an output like Numbat.

Parameters

- **y, z** (*x,*) – tensor position in pdb coordiante in Angstroms
- **rhombic** (*axial,*) – the tensor anisotropies in units 10^-32
- **beta, gamma** (*alpha,*) – the euler angles in degrees that maps the tensor to the pdb (I think?)

Returns **ChiTensor** – a tensor object for calulating paramagnetic effects on nuclear spins in the pdb coordinate

Return type object *paramagpy.metal.Metal*

Classes

Metal([position, eulers, axrh, mueff, ...])	An object for paramagnetic chi tensors and delta-chi tensors.

paramagpy.metal.Metal

class paramagpy.metal.**Metal**(*position=(0, 0, 0)*, *eulers=(0, 0, 0)*, *axrh=(0, 0)*, *mueff=0.0*, *g_axrh=(0, 0)*, *t1e=0.0*, *shift=0.0*, *temperature=298.15*, *B0=18.79*, *taur=0.0*)

An object for paramagnetic chi tensors and delta-chi tensors. This class has basic attributes that specify position, axiality/rhombicity, isotropy and euler angles. It also has methods for calculating PCS, RDC, PRE and CCR values.

__init__(*position=(0, 0, 0)*, *eulers=(0, 0, 0)*, *axrh=(0, 0)*, *mueff=0.0*, *g_axrh=(0, 0)*, *t1e=0.0*, *shift=0.0*, *temperature=298.15*, *B0=18.79*, *taur=0.0*)

Instantiate ChiTensor object

Parameters

- **position** (*array of floats, optional*) – the (x,y,z) position in meters. Default is (0,0,0) stored as a np.matrix object.
- **eulers** (*array of floats, optional*) – the euler angles [alpha,beta,gamma] in radians by ZYZ convention. Defualt is (0,0,0)
- **axrh** (*array of floats, optional*) – the axial and rhombic values defining

the magnetic susceptibility anisotropy

- **mueff** (*float*) – the effective magnetic moment in units of A.m^2
- **shift** (*float*) – a bulk shift value applied to all PCS calculations. This is a correction parameter that may arise due to an offset between diamagnetic and paramagnetic PCS datasets.
- **temperature** (*float*) – the temperature in Kelvin
- **t1e** (*float*) – the longitudinal electronic relaxation time
- **B0** (*float*) – the magnetic field in Telsa
- **taur** (*float*) – the rotational correlation time in seconds

Methods

atom_ccr(atom, atomPartner)	Calculate R2 cross-corelated relaxation due to DDxDSA
atom_pcs(atom[, racs, rads])	Calculate the psuedo-contact shift at the given atom
atom_pre(atom[, rtype, dsa, sbm, csa])	Calculate the PRE for an atom
atom_rdc(atom1, atom2)	Calculate the residual dipolar coupling between two atoms
atom_set_position(atom)	Set the position of the Metal object to that of an atom
ccr(position, gamma, dipole_shift_tensor)	Calculate R2 cross-corelated relaxation due to DDxDSA
copy()	Copy the current Metal object to a new instance
dipole_shift_tensor(position)	Calculate the chemical shift tensor at the given postition
dsa_r1(position, gamma[, csa])	Calculate R1 relaxation due to Curie Spin
dsa_r2(position, gamma[, csa])	Calculate R2 relaxation due to Curie Spin
fast_ccr(posarray, gammaarray, dstarray)	Vectorised version of *paramagpy.metal.Metal.ccr()*
fast_dipole_shift_tensor(posarray)	A vectorised version of *paramagpy.metal.Metal.dipole_shift_tensor()*
fast_dsa_r1(posarray, gammaarray[, csaarray])	Vectorised version of *paramagpy.metal.Metal.dsa_r1()*
fast_dsa_r2(posarray, gammaarray[, csaarray])	Vectorised version of *paramagpy.metal.Metal.dsa_r2()*
fast_first_invariant_squared(t)	Vectorised version of *paramagpy.metal.Metal.first_invariant_squared()*
fast_g_sbm_r1(posarray, gammaarray)	Vectorised version of *paramagpy.metal.Metal.g_sbm_r1()*
fast_pcs(posarray)	A vectorised version of *paramagpy.metal.Metal.pcs()*
fast_pre(posarray, gammaarray, rtype[, dsa, ...])	Calculate the PRE for a set of spins using Curie and or SBM theory
fast_racs(csaarray)	A vectorised version of *paramagpy.metal.Metal.racs()*
fast_rads(posarray)	A vectorised version of *paramagpy.metal.Metal.rads()*
fast_rdc(vecarray, gammaProdArray)	A vectorised version of *paramagpy.metal.Metal.rdc()* method.
fast_sbm_r1(posarray, gammaarray)	Vectorised version of *paramagpy.metal.Metal.sbm_r1()*

Continued on next page

Table 3 – continued from previous page

fast_sbm_r2(posarray, gammaarray)	Vectorised version of *paramagpy.metal.Metal.sbm_r2()*
fast_second_invariant_squared(t)	Vectorised version of *paramagpy.metal.Metal.second_invariant_squared()*
first_invariant_squared(t)	Calculate the antisymmetric contribution to relaxation via the first invariant of a tensor.
g_sbm_r1(position, gamma)	Calculate R1 relaxation due to Solomon-Bloembergen-Morgan theory from anisotropic power spectral density tensor
get_params(params)	Get tensor parameters that have been scaled appropriately
info([comment])	Get basic information about the Metal object
isomap([protein, isoval])	
make_mesh([density, size])	Construct a 3D grid of points to map an isosurface
pcs(position)	Calculate the psuedo-contact shift at the given postition
pcs_mesh(mesh)	Calculate a PCS value at each location of cubic grid of points
pre(position, gamma, rtype[, dsa, sbm, ...])	Calculate the PRE for a set of spins using Curie and or SBM theory
pre_mesh(mesh[, gamma, rtype, dsa, sbm])	Calculate a PRE value at each location of cubic grid of points
racs(csa)	Calculate the residual anisotropic chemical shift at the given postition.
rads(position)	Calculate the residual anisotropic dipolar shift at the given postition.
rdc(vector, gammaProd)	Calculate Residual Dipolar Coupling (RDC)
save([fileName])	
sbm_r1(position, gamma)	Calculate R1 relaxation due to Solomon-Bloembergen-Morgan theory
sbm_r2(position, gamma)	Calculate R2 relaxation due to Solomon-Bloembergen-Morgan theory
second_invariant_squared(t)	Calculate the second invariant squared of a tensor.
set_Jg(J, g)	Set the magnetic susceptibility absolute magnitude from J/g.
set_lanthanide(lanthanide[, set_dchi])	Set the anisotropy, isotropy and T1e parameters from literature values
set_params(paramValues)	Set tensor parameters that have been scaled appropriately
set_utr()	Modify current tensor parameters to unique tensor representation (UTR)
spec_dens(tau, omega)	A spectral density function with Lorentzian shape:
write_isomap(mesh, bounds[, fileName])	Write a PyMol script to file which allows loading of the isosurface file
write_pymol_script([isoval, surfaceName, ...])	Write a PyMol script to file which allows loading of the isosurface file

paramagpy.metal.Metal.atom_ccr

Metal.**atom_ccr**(*atom*, *atomPartner*)

Calculate R2 cross-corelated relaxation due to DDxDSA

Parameters

- **atom** (paramagpy.protein.CustomAtom) – the active nuclear spin for which relaxation will be calculated must have attributes 'position' and 'gamma'
- **atomPartner** (paramagpy.protein.CustomAtom) – the coupling parnter nuclear spin must have method 'dipole_shift_tensor'

Returns **value** – the CCR differential line broadening in Hz
Return type float

paramagpy.metal.Metal.atom_pcs

Metal.**atom_pcs**(*atom*, *racs=False*, *rads=False*)
Calculate the psuedo-contact shift at the given atom
Parameters
- **atom** (*biopython atom object*) – must have 'position' attribute
- **racs** (*bool (optional)*) – when True, RACS (residual anisotropic chemical shielding) correction is included. Default is False
- **rads** (*bool (optional)*) – when True, RADS (residual anisotropic dipolar shielding) correction is included. Defualt is False

Returns **pcs** – the pseudo-contact shift in parts-per-million (ppm)
Return type float

paramagpy.metal.Metal.atom_pre

Metal.**atom_pre**(*atom*, *rtype='r2'*, *dsa=True*, *sbm=True*, *csa=0.0*)
Calculate the PRE for an atom
Parameters
- **atom** (paramagpy.protein.CustomAtom) – the active nuclear spin for which relaxation will be calculated must have attributes 'position' and 'gamma'
- **rtype** (*str*) – either 'r1' or 'r2', the relaxation type
- **dsa** (*bool (optional)*) – when True (defualt), DSA or Curie spin relaxation is included
- **sbm** (*bool (optional)*) – when True (defualt), SBM spin relaxation is included
- **csa** (*array with shape (3,3) (optional)*) – CSA tensor of the spin. This defualts to 0.0, meaning CSAxDSA crosscorrelation is not accounted for.

Returns **rate** – The PRE rate in /s
Return type float

paramagpy.metal.Metal.atom_rdc

Metal.**atom_rdc**(*atom1*, *atom2*)
Calculate the residual dipolar coupling between two atoms
Parameters
- **atom1** (*biopython atom object*) – must have 'position' and 'gamma' attribute
- **atom1** – must have 'position' and 'gamma' attribute

Returns **rdc** – the RDC values in Hz
Return type float

paramagpy.metal.Metal.atom_set_position

Metal.**atom_set_position**(*atom*)
Set the position of the Metal object to that of an atom
Parameters **atom** (*biopython atom object*) – must have 'position' attribute

paramagpy.metal.Metal.ccr

Metal.**ccr**(*position*, *gamma*, *dipole_shift_tensor*)
Calculate R2 cross-corelated relaxation due to DDxDSA

If the metal has an anisotropic magnetic susceptibility, this is taken into account.

Parameters
- **position** (*array of floats*) – three coordinates (x,y,z) this is the position of the nuclear spin
- **gamma** (*float*) – the gyromagnetic ratio of the relaxing spin
- **dipole_shift_tensor** (*3x3 array of floats*) – this is the dipole shift tensor arising from the nuclear spin of the coupling partner

Returns **value** – The R2 differential line broadening rate in /s
Return type float

paramagpy.metal.Metal.copy

Metal.**copy**()
Copy the current Metal object to a new instance
Returns **new_tensor** – a new Metal instance with the same parameters
Return type Metal object

paramagpy.metal.Metal.dipole_shift_tensor

Metal.**dipole_shift_tensor**(*position*)
Calculate the chemical shift tensor at the given postition

This arises due to the paramagnetic dipole tensor field
Parameters **position** (*array floats*) – the position (x, y, z) in meters
Returns **dipole_shift_tensor** – the tensor describing chemical shift at the nuclear position
Return type 3x3 array

paramagpy.metal.Metal.dsa_r1

Metal.**dsa_r1**(*position*, *gamma*, *csa=0.0*)
Calculate R1 relaxation due to Curie Spin

If the metal has an anisotropic magnetic susceptibility, this is taken into account, resulting in orientation dependent PRE as predicted by Vega and Fiat. CSA cross-correlated relaxation may be included by providing an appropriate CSA tensor.

Parameters
- **position** (*array of floats*) – three coordinates (x,y,z) in meters
- **gamma** (*float*) – the gyromagnetic ratio of the spin
- **csa** (*3x3 matrix (optional)*) – the CSA tensor of the given spin. This defualts to 0.0, meaning CSAxDSA crosscorrelation is not accounted for.

Returns **value** – The R1 relaxation rate in /s
Return type float

paramagpy.metal.Metal.dsa_r2

Metal.**dsa_r2**(*position*, *gamma*, *csa=0.0*)
Calculate R2 relaxation due to Curie Spin

If the metal has an anisotropic magnetic susceptibility, this is taken into account, resulting in orientation dependent PRE as predicted by Vega and Fiat. CSA cross-correlated relaxation may be included by providing an appropriate CSA tensor.

Parameters

- **position** (*array of floats*) – three coordinates (x,y,z)
- **gamma** (*float*) – the gyromagnetic ratio of the spin
- **csa** (*3x3 matrix (optional)*) – the CSA tensor of the given spin. This defualts to 0.0, meaning CSAxDSA crosscorrelation is not accounted for.

Returns **value** – The R2 relaxation rate in /s

Return type float

paramagpy.metal.Metal.fast_ccr

Metal.**fast_ccr**(*posarray*, *gammaarray*, *dstarray*)

Vectorised version of *paramagpy.metal.Metal.ccr()*

This is generally used for speed in fitting DDxDSA data

If the metal has an anisotropic magnetic susceptibility, this is taken into account.

Parameters

- **posarray** (*array with shape (n,3)*) – array of positions in meters
- **gammaarray** (*array with shape (n,3)*) – array of gyromagnetic ratios of the spins
- **dstarray** (*array with shape (n,3,3)*) – array of nuclear dipole shift tensors arising from the coupling partners

Returns **rates** – The R2 differential line broadening rates in /s

Return type array with shape (n,1)

paramagpy.metal.Metal.fast_dipole_shift_tensor

Metal.**fast_dipole_shift_tensor**(*posarray*)

A vectorised version of *paramagpy.metal.Metal.dipole_shift_tensor()*

This is generally used for fast calculations.

Parameters **posarray** (*array*) – an array of positions with shape (n,3)

Returns **dipole_shift_tensor_array** – and array of dipole shift tensors at corresponding positions. This has shape (n,3,3)

Return type array

paramagpy.metal.Metal.fast_dsa_r1

Metal.**fast_dsa_r1**(*posarray*, *gammaarray*, *csaarray=0.0*)

Vectorised version of *paramagpy.metal.Metal.dsa_r1()*

This is generally used for speed in fitting PRE data

Parameters

- **posarray** (*array with shape (n,3)*) – array of positions in meters
- **gammaarray** (*array with shape (n,3)*) – array of gyromagnetic ratios of the spins
- **csaarray** (*array with shape (m,3,3) (optional)*) – array of CSA tensors of the spins. This defualts to 0.0, meaning CSAxDSA crosscorrelation is not accounted for.

Returns **rates** – The R1 relaxation rates in /s

Return type array with shape (n,1)

paramagpy.metal.Metal.fast_dsa_r2

Metal.**fast_dsa_r2**(*posarray*, *gammaarray*, *csaarray=0.0*)

Vectorised version of *paramagpy.metal.Metal.dsa_r2()*

This is generally used for speed in fitting PRE data.

Parameters

- **posarray** (*array with shape (n,3)*) – array of positions in meters
- **gammaarray** (*array with shape (n,3)*) – array of gyromagnetic ratios of the spins
- **csaarray** (*array with shape (m,3,3) (optional)*) – array of CSA tensors of the spins. This defualts to 0.0, meaning CSAxDSA crosscorrelation is not accounted for.

Returns **rates** – The R2 relaxation rates in /s

Return type array with shape (n,1)

paramagpy.metal.Metal.fast_first_invariant_squared

static Metal.**fast_first_invariant_squared**(*t*)

Vectorised version of *paramagpy.metal.Metal.first_invariant_squared()*

This is generally used for speed in fitting PRE data

Parameters **tensorarray** (*array with shape (n,3,3)*) – array of shielding tensors

Returns **firstInvariantSquared** – the first invariants squared of the tensors

Return type array with shape (n,1)

paramagpy.metal.Metal.fast_g_sbm_r1

Metal.**fast_g_sbm_r1**(*posarray*, *gammaarray*)

Vectorised version of *paramagpy.metal.Metal.g_sbm_r1()*

This is generally used for speed in fitting PRE data

Parameters

- **posarray** (*array with shape (n,3)*) – array of positions in meters
- **gammaarray** (*array with shape (n,3)*) – array of gyromagnetic ratios of the spins

Returns **rates** – The R1 relaxation rates in /s

Return type array with shape (n,1)

paramagpy.metal.Metal.fast_pcs

Metal.**fast_pcs**(*posarray*)

A vectorised version of *paramagpy.metal.Metal.pcs()*

This efficient algorithm calculates the PCSs for an array of positions and is best used where speed is required for fitting.

Parameters **posarray** (*array with shape (n,3)*) – array of 'n' positions (x, y, z) in meters

Returns **pcs** – the peudo-contact shift in parts-per-million (ppm)

Return type array of floats with shape (n,1)

paramagpy.metal.Metal.fast_pre

Metal.**fast_pre**(*posarray*, *gammaarray*, *rtype*, *dsa=True*, *sbm=True*, *gsbm=False*, *csaarray=0.0*)

Calculate the PRE for a set of spins using Curie and or SBM theory

Parameters

- **posarray** (*array with shape (n,3)*) – array of positions in meters
- **gammaarray** (*array with shape (n,3)*) – array of gyromagnetic ratios of the spins
- **rtype** (*str*) – either 'r1' or 'r2', the relaxation type
- **dsa** (*bool (optional)*) – when True (defualt), DSA or Curie spin relaxation is included
- **sbm** (*bool (optional)*) – when True (defualt), SBM spin relaxation is included
- **gsbm** (*bool (optional)*) – when True (default=False), anisotropic dipolar relaxation is included using the spectral power density gensor <g_tensor> NOTE: when true, ignores relaxation of type SBM NOTE: only implemented for R1 relaxation calculations
- **csaarray** (*array with shape (m,3,3) (optional)*) – array of CSA tensors of the spins. This defualts to 0.0, meaning CSAxDSA crosscorrelation is not accounted for.

Returns **rates** – The PRE rates in /s

Return type array with shape (n,1)

paramagpy.metal.Metal.fast_racs

Metal.**fast_racs**(*csaarray*)

A vectorised version of *paramagpy.metal.Metal.racs()*

This is generally used when speed is required for fitting

Parameters **csaarray** (*array with shape (n,3,3)*) – array of chemical shift anisotropy tensors

Returns **racs_array** – the residual anisotropic chemical shift in parts-per-million (ppm)

Return type array of floats with shape (n,1)

paramagpy.metal.Metal.fast_rads

Metal.**fast_rads**(*posarray*)

A vectorised version of *paramagpy.metal.Metal.rads()*

This is generally used when speed is required for fitting

Parameters **posarray** (*array with shape (n,3)*) – an array of 'n' positions (x, y, z) in meters

Returns **rads_array** – the residual anisotropic dipole shift in parts-per-million (ppm)

Return type array of floats with shape (n,1)

paramagpy.metal.Metal.fast_rdc

Metal.**fast_rdc**(*vecarray*, *gammaProdArray*)

A vectorised version of *paramagpy.metal.Metal.rdc()* method.

This is generally used for speed in fitting RDC data

Parameters

- **vecarray** (*array with shape (n,3)*) – array of internuclear vectors in meters
- **gammaProdArray** (*array with shape (n,1)*) – the products of gyromagnetic ratios of spins A and B where each has units of rad/s/T

Returns **rdc_array** – the RDC values in Hz
Return type array with shape (n,1)

paramagpy.metal.Metal.fast_sbm_r1

Metal.**fast_sbm_r1**(*posarray*, *gammaarray*)
Vectorised version of *paramagpy.metal.Metal.sbm_r1()*

This is generally used for speed in fitting PRE data
Parameters
- **posarray** (*array with shape (n,3)*) – array of positions in meters
- **gammaarray** (*array with shape (n,3)*) – array of gyromagnetic ratios of the spins

Returns **rates** – The R1 relaxation rates in /s
Return type array with shape (n,1)

paramagpy.metal.Metal.fast_sbm_r2

Metal.**fast_sbm_r2**(*posarray*, *gammaarray*)
Vectorised version of *paramagpy.metal.Metal.sbm_r2()*

This is generally used for speed in fitting PRE data
Parameters
- **posarray** (*array with shape (n,3)*) – array of positions in meters
- **gammaarray** (*array with shape (n,3)*) – array of gyromagnetic ratios of the spins

Returns **rates** – The R2 relaxation rates in /s
Return type array with shape (n,1)

paramagpy.metal.Metal.fast_second_invariant_squared

static Metal.**fast_second_invariant_squared**(*t*)
Vectorised version of *paramagpy.metal.Metal.second_invariant_squared()*

This is generally used for speed in fitting PRE data
Parameters **tensorarray** (*array with shape (n,3,3)*) – array of shielding tensors
Returns **secondInvariantSquared** – the second invariants squared of the tensors
Return type array with shape (n,1)

paramagpy.metal.Metal.first_invariant_squared

static Metal.**first_invariant_squared**(*t*)
Calculate the antisymmetric contribution to relaxation via the first invariant of a tensor.

This is required for PRE calculations using the shilding tensor
Parameters **tensor** (*3x3 matrix*) – a second rank tensor
Returns **firstInvariantSquared** – the first invariant squared of the shift tensor
Return type float

paramagpy.metal.Metal.g_sbm_r1

Metal.**g_sbm_r1**(*position*, *gamma*)
Calculate R1 relaxation due to Solomon-Bloembergen-Morgan theory from anisotropic power spectral density tensor

Parameters
- **position** (*array of floats*) – three coordinates (x,y,z)
- **gamma** (*float*) – the gyromagnetic ratio of the spin

Returns **value** – The R1 relaxation rate in /s
Return type float

paramagpy.metal.Metal.get_params

Metal.**get_params**(*params*)
Get tensor parameters that have been scaled appropriately

This is often used to get parameter values during fitting where floating point errors would otherwise occur on the small values encountered.

Parameters **params** (*list of str*) – each element of the list is a string that corresponds to an attribute of the Metal to be retrieved.
Returns **scaled_params** – a list with respective scaled parameter values from the input.
Return type list

Examples

```
>>> metal = Metal(axrh=[20E-32, 3E-32],position=[0.0,10E-10,-5E-10])
>>> metal.get_params(['ax','rh','x','y','z'])
[20.0, 3.0, 0.0, 10.0, -5.0]
```

paramagpy.metal.Metal.info

Metal.**info**(*comment=True*)
Get basic information about the Metal object

This is returned as a string in human readable units This is also the file format for saving the tensor

Parameters **comment** (*bool (optional)*) – if True, each line has a '#' placed at the front
Returns **information** – a string containing basic information about the Metal
Return type str

Examples

```
>>> metal = Metal()
>>> metal.set_lanthanide('Er')
>>> metal.info()
# ax     | 1E-32 m^3 :   -11.600
# rh     | 1E-32 m^3 :    -8.600
# x      |   1E-10 m :     0.000
# y      |   1E-10 m :     0.000
# z      |   1E-10 m :     0.000
# a      |       deg :     0.000
# b      |       deg :     0.000
# g      |       deg :     0.000
# mueff  |        Bm :     9.581
# shift  |       ppm :     0.000
# B0     |         T :    18.790
# temp   |         K :   298.150
# t1e    |        ps :     0.189
# taur   |        ns :     0.000
```

paramagpy.metal.Metal.isomap

Metal.**isomap**(*protein=None*, *isoval=1.0*, ***kwargs*)

paramagpy.metal.Metal.make_mesh

Metal.**make_mesh**(*density=2*, *size=40.0*)

Construct a 3D grid of points to map an isosurface

This is contained in a cube

Parameters

- **density** (*int (optional)*) – the points per Angstrom in the grid
- **size** (*float (optional)*) – the length of one edge of the cube

Returns

- **mesh** (*cubic grid array*) – This has shape (n,n,n,3) where n is the number of points along one edge of the grid. Units are meters
- **bounds** (*tuple (origin, low, high, points)*) – This tuple has information about the bounding box

  origin : array of floats, the (x,y,z) location of mesh vertex
  low [array of ints, the integer location of the first] point in each dimension
  high [array of ints, the integer location of the last] point in each dimension
  points : array of ints, the number of points along each dimension

paramagpy.metal.Metal.pcs

Metal.**pcs**(*position*)

Calculate the psuedo-contact shift at the given postition

Parameters **position** (*array floats*) – the position (x, y, z) in meters

Returns **pcs** – the pseudo-contact shift in parts-per-million (ppm)

Return type float

Examples

```
>>> metal = Metal()
>>> metal.set_lanthanide('Er')
>>> metal.pcs([0.,0.,10E-10])
-6.153991132886608
```

paramagpy.metal.Metal.pcs_mesh

Metal.**pcs_mesh**(*mesh*)

Calculate a PCS value at each location of cubic grid of points

Parameters **mesh** (*array with shape (n,n,n,3)*) – a cubic grid as generated by the method <make_mesh>

Returns **pcs_mesh** – The same grid shape, with PCS values at the respective locations

Return type array with shape (n,n,n,1)

paramagpy.metal.Metal.pre

Metal.**pre**(*position*, *gamma*, *rtype*, *dsa=True*, *sbm=True*, *gsbm=False*, *csa=0.0*)

Calculate the PRE for a set of spins using Curie and or SBM theory

Parameters

- **position** (*array of floats*) – position in meters
- **gamma** (*float*) – gyromagnetic ratio of the spin
- **rtype** (*str*) – either 'r1' or 'r2', the relaxation type
- **dsa** (*bool (optional)*) – when True (defualt), DSA or Curie spin relaxation is included
- **sbm** (*bool (optional)*) – when True (defualt), SBM spin relaxation is included
- **gsbm** (*bool (optional)*) – when True (default=False), anisotropic dipolar relaxation is included using the spectral power density gensor <g_tensor> NOTE: when true, ignores relaxation of type SBM NOTE: only implemented for R1 relaxation calculations
- **csa** (*array with shape (3,3) (optional)*) – CSA tensor of the spin. This defualts to 0.0, meaning CSAxDSA crosscorrelation is not accounted for.

Returns **rate** – The PRE rate in /s

Return type float

paramagpy.metal.Metal.pre_mesh

Metal.**pre_mesh**(*mesh*, *gamma=267512897.63847807*, *rtype='r2'*, *dsa=True*, *sbm=True*)

Calculate a PRE value at each location of cubic grid of points

Parameters

- **mesh** (*array with shape (n,n,n,3)*) – a cubic grid as generated by the method <make_mesh>
- **gamma** (*float*) – the gyromagnetic ratio of the spin
- **rtype** (*str*) – either 'r1' or 'r2', the relaxation type
- **dsa** (*bool (optional)*) – when True (defualt), DSA or Curie spin relaxation is included
- **sbm** (*bool (optional)*) – when True (defualt), SBM spin relaxation is included

Returns **pre_mesh** – The same grid shape, with PRE values at the respective locations

Return type array with shape (n,n,n,1)

paramagpy.metal.Metal.racs

Metal.**racs**(*csa*)

Calculate the residual anisotropic chemical shift at the given postition.

The partial alignment induced by an anisotropic magnetic susecptiblity causes the chemical shift tensor at a nuclear position to average to a value different to the isotropic value.

Parameters **csa** (*3 x 3 array*) – the chemical shift anisotropy tensor

Returns **racs** – the residual anisotropic chemical shift in parts-per-million (ppm)

Return type float

paramagpy.metal.Metal.rads

Metal.**rads**(*position*)

Calculate the residual anisotropic dipolar shift at the given postition.

The partial alignment induced by an anisotropic magnetic susecptiblity causes the dipole shift tensor at a nuclear position to average to a value different to the PCS.

Parameters **position** (*array floats*) – the position (x, y, z) in meters

Returns **rads** – the residual anisotropic dipole shift in parts-per-million (ppm)

Return type float

paramagpy.metal.Metal.rdc

Metal.**rdc**(*vector*, *gammaProd*)

Calculate Residual Dipolar Coupling (RDC)

Parameters

- **vector** (*array of floats*) – internuclear vector (x,y,z) in meters
- **gammaProd** (*float*) – the product of gyromagnetic ratios of spin A and B where each has units of rad/s/T

Returns **rdc** – the RDC in Hz

Return type float

paramagpy.metal.Metal.save

Metal.**save**(*fileName='tensor.txt'*)

paramagpy.metal.Metal.sbm_r1

Metal.**sbm_r1**(*position*, *gamma*)

Calculate R1 relaxation due to Solomon-Bloembergen-Morgan theory

Parameters

- **position** (*array of floats*) – three coordinates (x,y,z)
- **gamma** (*float*) – the gyromagnetic ratio of the spin

Returns **value** – The R1 relaxation rate in /s

Return type float

paramagpy.metal.Metal.sbm_r2

Metal.**sbm_r2**(*position*, *gamma*)

Calculate R2 relaxation due to Solomon-Bloembergen-Morgan theory

Parameters

- **position** (*array of floats*) – three coordinates (x,y,z)

- **gamma** (*float*) – the gyromagnetic ratio of the spin

Returns **value** – The R2 relaxation rate in /s

Return type float

paramagpy.metal.Metal.second_invariant_squared

static Metal.**second_invariant_squared**(*t*)

Calculate the second invariant squared of a tensor.

This is required for PRE calculations using the shilding tensor

Parameters **tensor** (*3x3 matrix*) – a second rank tensor

Returns **secondInvariantSquared** – the second invariant squared of the shift tensor

Return type float

paramagpy.metal.Metal.set_Jg

Metal.**set_Jg**(*J*, *g*)

Set the magnetic susceptibility absolute magnitude from J/g.

This is achieved using the following formula:

$$\mu_{eff} = g\mu_B\sqrt{J(J+1)}$$

Parameters

- **J** (*str*) – the total spin angular momentum quantum number
- **g** (*bool, optional*) – the Lande g-factor

paramagpy.metal.Metal.set_lanthanide

Metal.**set_lanthanide**(*lanthanide*, *set_dchi=True*)

Set the anisotropy, isotropy and T1e parameters from literature values

Parameters

- **lanthanide** (*str*) –

  one of ['Ce','Pr','Nd','Pm','Sm','Eu','Gd', 'Tb','Dy','Ho','Er','Tm','Yb']

- **set_dichi** (*bool (optional)*) – if True (default), the tensor anisotropy is set. Otherwise only the isotropy and T1e values are set

paramagpy.metal.Metal.set_params

Metal.**set_params**(*paramValues*)

Set tensor parameters that have been scaled appropriately

This is the inverse of the method <get_params>

Parameters **paramValues** (*list of tuple*) – each element is a tuple (variable, value) where 'variable' is the string indentifying the attribute to be set, and 'value' is the corresponding value

Examples

```
>>> metal = Metal()
>>> metal.set_params([('ax',20.0),('rh',3.0)])
>>> metal.axrh
[2.e-31 3.e-32]
```

paramagpy.metal.Metal.set_utr

Metal.**set_utr**()

Modify current tensor parameters to unique tensor representation (UTR)

Note that multiple axial/rhombic and euler angles can give congruent tensors. This method ensures that identical tensors may always be compared by using Numbat style representation.

paramagpy.metal.Metal.spec_dens

static Metal.**spec_dens**(*tau*, *omega*)

A spectral density function with Lorentzian shape:

$$\mathbb{J}(\tau, \omega) = \frac{\tau}{1 + (\omega\tau)^2}$$

Parameters

- **tau** (*float*) – correaltion time
- **omega** (*float*) – frequency

Returns **value** – the value of the spectral denstiy

Return type float

paramagpy.metal.Metal.write_isomap

Metal.**write_isomap**(*mesh*, *bounds*, *fileName='isomap.pml.ccp4'*)

Write a PyMol script to file which allows loading of the isosurface file

Parameters

- **mesh** (*3D scalar np.ndarray of floats*) – the scalar field of PCS or PRE values in a cubic grid
- **bounds** (*tuple (origin, low, high, points)*) – as generated by *paramagpy.metal.Metal.make_mesh()*
- **fileName** (*str (optional)*) – the filename of the isosurface file

paramagpy.metal.Metal.write_pymol_script

Metal.**write_pymol_script**(*isoval=1.0*, *surfaceName='isomap'*, *scriptName='isomap.pml'*, *meshName='./isomap.pml.ccp4'*, *pdbFile=None*)

Write a PyMol script to file which allows loading of the isosurface file

Parameters

- **isoval** (*float (optional)*) – the contour level of the isosurface
- **surfaceName** (*str (optional)*) – the name of the isosurface file within PyMol

- **scriptName** (*str (optional)*) – the name of the PyMol script to load the tensor isosurface
- **meshName** (*str (optional)*) – the name of the binary isosurface file
- **pdbFile** (*str (optional)*) – if not <None>, the file name of the PDB file to be loaded with the isosurface.

Attributes

B0_MHz	1H NMR frequency for the given field in MHz
GAMMA	
HBAR	
K	
MU0	
MUB	
a	alpha euler anglue
alignment_factor	Factor for conversion between magnetic susceptibility and alignment tensors
ax	axiality
b	beta euler anglue
eigenvalues	The eigenvalues defining the magnitude of the principle axes
fit_scaling	
g	gamma euler anglue
g_eigenvalues	The eigenvalues defining the magnitude of the principle axes
g_isotropy	Estimate of the spectral power density tensor isotropy
g_tensor	The magnetic susceptibility tensor matrix representation
gax	axial componenet of spectral power density tensor
grh	axial componenet of spectral power density tensor
iso	isotropy
isotropy	The magnidue of the isotropic component of the tensor
lanth_axrh	
lanth_lib	
lower_coords	
rh	rhombicity
rotationMatrix	The rotation matrix as defined by the euler angles
saupe_factor	Factor for conversion between magnetic susceptibility and saupe tensors
tauc	The effective rotational correlation time.
tensor	The magnetic susceptibility tensor matrix representation
tensor_alignment	The alignment tensor matrix representation
tensor_saupe	The saupe tensor matrix representation
tensor_traceless	The traceless magnetic susceptibility tensor matrix representation
upper_coords	
upper_triang	Fetch 5 unique matrix element defining the magnetic susceptibility tensor

Continued on next page

Table 4 – continued from previous page

upper_triang_alignment	Fetch 5 unique matrix element defining the alignment tensor
upper_triang_saupe	Fetch 5 unique matrix element defining the magnetic susceptibility tensor
x	x coordinate
y	y coordinate
z	z coordinate

paramagpy.metal.Metal.B0_MHz

Metal.**B0_MHz**
1H NMR frequency for the given field in MHz

paramagpy.metal.Metal.GAMMA

Metal.**GAMMA = 176085964400.0**

paramagpy.metal.Metal.HBAR

Metal.**HBAR = 1.0546e-34**

paramagpy.metal.Metal.K

Metal.**K = 1.381e-23**

paramagpy.metal.Metal.MU0

Metal.**MU0 = 1.2566370614359173e-06**

paramagpy.metal.Metal.MUB

Metal.**MUB = 9.274e-24**

paramagpy.metal.Metal.a

Metal.**a**
alpha euler anglue

paramagpy.metal.Metal.alignment_factor

Metal.**alignment_factor**
Factor for conversion between magnetic susceptibility and alignment tensors

paramagpy.metal.Metal.ax

Metal.**ax**
axiality

paramagpy.metal.Metal.b

Metal.**b**
beta euler anglue

paramagpy.metal.Metal.eigenvalues

Metal.**eigenvalues**
The eigenvalues defining the magnitude of the principle axes

paramagpy.metal.Metal.fit_scaling

Metal.**fit_scaling = {'a': 57.29577951308232, 'ax': 1e+32, 'b': 57.2957795130**

paramagpy.metal.Metal.g

Metal.**g**
gamma euler anglue

paramagpy.metal.Metal.g_eigenvalues

Metal.**g_eigenvalues**
The eigenvalues defining the magnitude of the principle axes

paramagpy.metal.Metal.g_isotropy

Metal.**g_isotropy**
Estimate of the spectral power density tensor isotropy

paramagpy.metal.Metal.g_tensor

Metal.**g_tensor**
The magnetic susceptibility tensor matrix representation

paramagpy.metal.Metal.gax

Metal.**gax**
axial componenet of spectral power density tensor

paramagpy.metal.Metal.grh

Metal.**grh**
axial componenet of spectral power density tensor

paramagpy.metal.Metal.iso

Metal.**iso**
isotropy

paramagpy.metal.Metal.isotropy

Metal.**isotropy**
The magnidue of the isotropic component of the tensor

paramagpy.metal.Metal.lanth_axrh

Metal.**lanth_axrh = {'Ce': (2.1, 0.7), 'Dy': (34.7, 20.3), 'Er': (-11.6, -8.6)**

paramagpy.metal.Metal.lanth_lib

Metal.**lanth_lib = {'Ce': (2.5, 0.8571428571428571, 1.33e-13), 'Dy': (7.5, 1.33**

paramagpy.metal.Metal.lower_coords

Metal.**lower_coords = ((0, 1, 1, 2, 2), (0, 1, 0, 0, 1))**

paramagpy.metal.Metal.rh

Metal.**rh**
rhombicity

paramagpy.metal.Metal.rotationMatrix

Metal.**rotationMatrix**
The rotation matrix as defined by the euler angles

paramagpy.metal.Metal.saupe_factor

Metal.**saupe_factor**
Factor for conversion between magnetic susceptibility and saupe tensors

paramagpy.metal.Metal.tauc

Metal.**tauc**
The effective rotational correlation time.

This is calculated by combining the rotational correaltion time and the electronic relaxation time:

$$\tau_c = \frac{1}{\frac{1}{\tau_r} + \frac{1}{T_{1e}}}$$

paramagpy.metal.Metal.tensor

Metal.**tensor**
The magnetic susceptibility tensor matrix representation

paramagpy.metal.Metal.tensor_alignment

Metal.**tensor_alignment**
The alignment tensor matrix representation

paramagpy.metal.Metal.tensor_saupe

Metal.**tensor_saupe**
The saupe tensor matrix representation

paramagpy.metal.Metal.tensor_traceless

Metal.**tensor_traceless**
The traceless magnetic susceptibility tensor matrix representation

paramagpy.metal.Metal.upper_coords

Metal.**upper_coords = ((0, 1, 0, 0, 1), (0, 1, 1, 2, 2))**

paramagpy.metal.Metal.upper_triang

Metal.**upper_triang**
Fetch 5 unique matrix element defining the magnetic susceptibility tensor

paramagpy.metal.Metal.upper_triang_alignment

Metal.**upper_triang_alignment**
Fetch 5 unique matrix element defining the alignment tensor

paramagpy.metal.Metal.upper_triang_saupe

Metal.**upper_triang_saupe**
Fetch 5 unique matrix element defining the magnetic susceptibility tensor

paramagpy.metal.Metal.x

Metal.**x**
x coordinate

paramagpy.metal.Metal.y

Metal.**y**
y coordinate

paramagpy.metal.Metal.z

Metal.**z**
z coordinate

5.5.2 Protein module

This module handles the protein structure coordinates and includes methods for loading a PDB file and calculating atomic properites such as CSA or gyromagnetic ratio

paramagpy.protein

Functions

load_pdb(fileName[, ident])	Read PDB from file into biopython structure object
rotation_matrix(axis, theta)	Return the rotation matrix associated with counterclockwise rotation about the given axis by theta radians.

paramagpy.protein.load_pdb

paramagpy.protein.**load_pdb**(*fileName*, *ident=None*)

Read PDB from file into biopython structure object

Parameters

- **fileName** (*str*) – the path to the file
- **ident** (*str (optional)*) – the desired identity of the structure object

Returns **values** – a structure object containing the atomic coordinates

Return type *paramagpy.protein.CustomStructure*

paramagpy.protein.rotation_matrix

paramagpy.protein.**rotation_matrix**(*axis*, *theta*)

Return the rotation matrix associated with counterclockwise rotation about the given axis by theta radians.

Parameters **axis** (*array of floats*) – the [x,y,z] axis for rotation.

Returns **matrix** – the rotation matrix

Return type numpy 3x3 matrix object

Classes

CustomAtom(*arg, **kwargs)	
CustomStructure(*arg, **kwargs)	This is an overload hack of the BioPython Structure object
CustomStructureBuilder(*arg, **kwargs)	This is an overload hack of BioPython's CustomStructureBuilder

paramagpy.protein.CustomAtom

class paramagpy.protein.**CustomAtom**(**arg*, ***kwargs*)

__init__(**arg*, ***kwargs*)

Create Atom object.

The Atom object stores atom name (both with and without spaces), coordinates, B factor, occupancy,

alternative location specifier and (optionally) anisotropic B factor and standard deviations of B factor and positions.

Parameters

- **name** (*string*) – atom name (eg. "CA"). Note that spaces are normally stripped.
- **coord** (*Numeric array (Float0, size 3)*) – atomic coordinates (x,y,z)
- **bfactor** (*number*) – isotropic B factor
- **occupancy** (*number*) – occupancy (0.0-1.0)
- **altloc** (*string*) – alternative location specifier for disordered atoms
- **fullname** (*string*) – full atom name, including spaces, e.g. " CA ". Normally these spaces are stripped from the atom name.
- **element** (*uppercase string (or None if unknown)*) – atom element, e.g. "C" for Carbon, "HG" for mercury,

Methods

copy()	Create a copy of the Atom.
detach_parent()	Remove reference to parent.
dipole_shift_tensor(position)	Calculate the magnetic field shielding tensor at the given postition due to the nuclear dipole
flag_disorder()	Set the disordered flag to 1.
get_altloc()	Return alternative location specifier.
get_anisou()	Return anisotropic B factor.
get_bfactor()	Return B factor.
get_coord()	Return atomic coordinates.
get_full_id()	Return the full id of the atom.
get_fullname()	Return the atom name, including leading and trailing spaces.
get_id()	Return the id of the atom (which is its atom name).
get_level()	
get_name()	Return atom name.
get_occupancy()	Return occupancy.
get_parent()	Return parent residue.
get_serial_number()	
get_sigatm()	Return standard deviation of atomic parameters.
get_siguij()	Return standard deviations of anisotropic temperature factors.
get_vector()	Return coordinates as Vector.
is_disordered()	Return the disordered flag (1 if disordered, 0 otherwise).
set_altloc(altloc)	
set_anisou(anisou_array)	Set anisotropic B factor.
set_bfactor(bfactor)	
set_coord(coord)	
set_occupancy(occupancy)	
set_parent(parent)	Set the parent residue.
set_serial_number(n)	
set_sigatm(sigatm_array)	Set standard deviation of atomic parameters.

Continued on next page

Table 7 – continued from previous page

set_siguij(siguij_array)	Set standard deviations of anisotropic temperature factors.
top()	
transform(rot, tran)	Apply rotation and translation to the atomic coordinates.

paramagpy.protein.CustomAtom.copy

CustomAtom.**copy**()
Create a copy of the Atom.

Parent information is lost.

paramagpy.protein.CustomAtom.detach_parent

CustomAtom.**detach_parent**()
Remove reference to parent.

paramagpy.protein.CustomAtom.dipole_shift_tensor

CustomAtom.**dipole_shift_tensor**(*position*)
Calculate the magnetic field shielding tensor at the given postition due to the nuclear dipole

Assumes nuclear spin 1/2

Parameters **position** (*array floats*) – the position (x, y, z) in meters

Returns **dipole_shielding_tensor** – the tensor describing magnetic shielding at the given position

Return type 3x3 array

paramagpy.protein.CustomAtom.flag_disorder

CustomAtom.**flag_disorder**()
Set the disordered flag to 1.

The disordered flag indicates whether the atom is disordered or not.

paramagpy.protein.CustomAtom.get_altloc

CustomAtom.**get_altloc**()
Return alternative location specifier.

paramagpy.protein.CustomAtom.get_anisou

CustomAtom.**get_anisou**()
Return anisotropic B factor.

paramagpy.protein.CustomAtom.get_bfactor

CustomAtom.**get_bfactor**()
Return B factor.

paramagpy.protein.CustomAtom.get_coord

CustomAtom.**get_coord**()
Return atomic coordinates.

paramagpy.protein.CustomAtom.get_full_id

CustomAtom.**get_full_id**()
Return the full id of the atom.

The full id of an atom is the tuple (structure id, model id, chain id, residue id, atom name, altloc).

paramagpy.protein.CustomAtom.get_fullname

CustomAtom.**get_fullname**()
Return the atom name, including leading and trailing spaces.

paramagpy.protein.CustomAtom.get_id

CustomAtom.**get_id**()
Return the id of the atom (which is its atom name).

paramagpy.protein.CustomAtom.get_level

CustomAtom.**get_level**()

paramagpy.protein.CustomAtom.get_name

CustomAtom.**get_name**()
Return atom name.

paramagpy.protein.CustomAtom.get_occupancy

CustomAtom.**get_occupancy**()
Return occupancy.

paramagpy.protein.CustomAtom.get_parent

CustomAtom.**get_parent**()
Return parent residue.

paramagpy.protein.CustomAtom.get_serial_number

CustomAtom.**get_serial_number**()

paramagpy.protein.CustomAtom.get_sigatm

CustomAtom.**get_sigatm**()
Return standard deviation of atomic parameters.

paramagpy.protein.CustomAtom.get_siguij

CustomAtom.**get_siguij**()
Return standard deviations of anisotropic temperature factors.

paramagpy.protein.CustomAtom.get_vector

CustomAtom.**get_vector**()
Return coordinates as Vector.

Returns coordinates as 3D vector

Return type Bio.PDB.Vector class

paramagpy.protein.CustomAtom.is_disordered

CustomAtom.**is_disordered**()
Return the disordered flag (1 if disordered, 0 otherwise).

paramagpy.protein.CustomAtom.set_altloc

CustomAtom.**set_altloc**(*altloc*)

paramagpy.protein.CustomAtom.set_anisou

CustomAtom.**set_anisou**(*anisou_array*)
Set anisotropic B factor.

Parameters **anisou_array** (*Numeric array (length 6)*) – anisotropic B factor.

paramagpy.protein.CustomAtom.set_bfactor

CustomAtom.**set_bfactor**(*bfactor*)

paramagpy.protein.CustomAtom.set_coord

CustomAtom.**set_coord**(*coord*)

paramagpy.protein.CustomAtom.set_occupancy

CustomAtom.**set_occupancy**(*occupancy*)

paramagpy.protein.CustomAtom.set_parent

CustomAtom.**set_parent**(*parent*)
Set the parent residue.

Parameters **parent - Residue object** (–) –

paramagpy.protein.CustomAtom.set_serial_number

CustomAtom.**set_serial_number**(*n*)

paramagpy.protein.CustomAtom.set_sigatm

CustomAtom.**set_sigatm**(*sigatm_array*)
Set standard deviation of atomic parameters.

The standard deviation of atomic parameters consists of 3 positional, 1 B factor and 1 occupancy standard deviation.

Parameters **sigatm_array** (*Numeric array (length 5)*) – standard deviations of atomic parameters.

paramagpy.protein.CustomAtom.set_siguij

CustomAtom.**set_siguij**(*siguij_array*)
Set standard deviations of anisotropic temperature factors.

Parameters **siguij_array** (*Numeric array (length 6)*) – standard deviations of anisotropic temperature factors.

paramagpy.protein.CustomAtom.top

CustomAtom.**top**()

paramagpy.protein.CustomAtom.transform

CustomAtom.**transform**(*rot*, *tran*)
Apply rotation and translation to the atomic coordinates.

Parameters

- **rot** (*3x3 Numeric array*) – A right multiplying rotation matrix
- **tran** (*size 3 Numeric array*) – the translation vector

Examples

```
>>> rotation=rotmat(pi, Vector(1, 0, 0))
>>> translation=array((0, 0, 1), 'f')
>>> atom.transform(rotation, translation)
```

Attributes

HBAR	
MU0	
csa	Get the CSA tensor at the nuclear position This uses the geometry of neighbouring atoms and a standard library from Bax J.
csa_lib	docstring for CustomAtom
gyro_lib	

Continued on next page

Table 8 – continued from previous page

position	

paramagpy.protein.CustomAtom.HBAR

CustomAtom.**HBAR = 1.0546e-34**

paramagpy.protein.CustomAtom.MU0

CustomAtom.**MU0 = 1.2566370614359173e-06**

paramagpy.protein.CustomAtom.csa

CustomAtom.**csa**
Get the CSA tensor at the nuclear position This uses the geometry of neighbouring atoms and a standard library from Bax J. Am. Chem. Soc. 2000

Returns **matrix** – the CSA tensor in the PDB frame if appropriate nuclear positions are not available <None> is returned.

Return type 3x3 array

paramagpy.protein.CustomAtom.csa_lib

CustomAtom.**csa_lib = {'C': (array([-8.65e-05, 1.18e-05, 7.47e-05]), 0.663225115**
docstring for CustomAtom

paramagpy.protein.CustomAtom.gyro_lib

CustomAtom.**gyro_lib = {'C': 67261498.71335746, 'H': 267512897.63847807, 'N': -2**

paramagpy.protein.CustomAtom.position

CustomAtom.**position**

paramagpy.protein.CustomStructure

class paramagpy.protein.**CustomStructure**(**arg*, ***kwargs*)
This is an overload hack of the BioPython Structure object

__init__(**arg*, ***kwargs*)
Initialize the class.

Methods

add(entity)	Add a child to the Entity.
copy()	
detach_child(id)	Remove a child.
detach_parent()	Detach the parent.
get_atoms()	

Continued on next page

Table 9 – continued from previous page

get_chains()	
get_full_id()	Return the full id.
get_id()	Return the id.
get_iterator()	Return iterator over children.
get_level()	Return level in hierarchy.
get_list()	Return a copy of the list of children.
get_models()	
get_parent()	Return the parent Entity object.
get_residues()	
has_id(id)	Check if a child with given id exists.
insert(pos, entity)	Add a child to the Entity at a specified position.
parse(dataValues[, models])	
set_parent(entity)	Set the parent Entity object.
transform(rot, tran)	Apply rotation and translation to the atomic coordinates.

paramagpy.protein.CustomStructure.add

CustomStructure.**add**(*entity*)
Add a child to the Entity.

paramagpy.protein.CustomStructure.copy

CustomStructure.**copy**()

paramagpy.protein.CustomStructure.detach_child

CustomStructure.**detach_child**(*id*)
Remove a child.

paramagpy.protein.CustomStructure.detach_parent

CustomStructure.**detach_parent**()
Detach the parent.

paramagpy.protein.CustomStructure.get_atoms

CustomStructure.**get_atoms**()

paramagpy.protein.CustomStructure.get_chains

CustomStructure.**get_chains**()

paramagpy.protein.CustomStructure.get_full_id

CustomStructure.**get_full_id**()
Return the full id.

The full id is a tuple containing all id's starting from the top object (Structure) down to the current object. A full id for a Residue object e.g. is something like:

("1abc", 0, "A", (" ", 10, "A"))

This corresponds to:

Structure with id "1abc" Model with id 0 Chain with id "A" Residue with id (" ", 10, "A")

The Residue id indicates that the residue is not a hetero-residue (or a water) because it has a blank hetero field, that its sequence identifier is 10 and its insertion code "A".

paramagpy.protein.CustomStructure.get_id

CustomStructure.**get_id**()
Return the id.

paramagpy.protein.CustomStructure.get_iterator

CustomStructure.**get_iterator**()
Return iterator over children.

paramagpy.protein.CustomStructure.get_level

CustomStructure.**get_level**()
Return level in hierarchy.

A - atom R - residue C - chain M - model S - structure

paramagpy.protein.CustomStructure.get_list

CustomStructure.**get_list**()
Return a copy of the list of children.

paramagpy.protein.CustomStructure.get_models

CustomStructure.**get_models**()

paramagpy.protein.CustomStructure.get_parent

CustomStructure.**get_parent**()
Return the parent Entity object.

paramagpy.protein.CustomStructure.get_residues

CustomStructure.**get_residues**()

paramagpy.protein.CustomStructure.has_id

CustomStructure.**has_id**(*id*)
Check if a child with given id exists.

paramagpy.protein.CustomStructure.insert

CustomStructure.**insert**(*pos*, *entity*)
Add a child to the Entity at a specified position.

paramagpy.protein.CustomStructure.parse

CustomStructure.**parse**(*dataValues*, *models=None*)

paramagpy.protein.CustomStructure.set_parent

CustomStructure.**set_parent**(*entity*)
Set the parent Entity object.

paramagpy.protein.CustomStructure.transform

CustomStructure.**transform**(*rot*, *tran*)
Apply rotation and translation to the atomic coordinates.

Parameters

- **rot** (*3x3 Numeric array*) – A right multiplying rotation matrix
- **tran** (*size 3 Numeric array*) – the translation vector

Examples

```
>>> rotation = rotmat(pi, Vector(1, 0, 0))
>>> translation = array((0, 0, 1), 'f')
>>> entity.transform(rotation, translation)
```

Attributes

id

paramagpy.protein.CustomStructure.id

CustomStructure.**id**

paramagpy.protein.CustomStructureBuilder

class paramagpy.protein.**CustomStructureBuilder**(**arg*, ***kwargs*)
This is an overload hack of BioPython's CustomStructureBuilder

__init__(**arg*, ***kwargs*)
Initialize the class.

Methods

get_structure()	Return the structure.

Continued on next page

Table 11 – continued from previous page

init_atom(name, coord, b_factor, occupancy, ...)	Create a new Atom object.
init_chain(chain_id)	Create a new Chain object with given id.
init_model(model_id[, serial_num])	Create a new Model object with given id.
init_residue(resname, field, resseq, icode)	Create a new Residue object.
init_seg(segid)	Flag a change in segid.
init_structure(structure_id)	Initialize a new Structure object with given id.
set_anisou(anisou_array)	Set anisotropic B factor of current Atom.
set_header(header)	
set_line_counter(line_counter)	Tracks line in the PDB file that is being parsed.
set_sigatm(sigatm_array)	Set standard deviation of atom position of current Atom.
set_siguij(siguij_array)	Set standard deviation of anisotropic B factor of current Atom.
set_symmetry(spacegroup, cell)	

paramagpy.protein.CustomStructureBuilder.get_structure

CustomStructureBuilder.**get_structure**()
Return the structure.

paramagpy.protein.CustomStructureBuilder.init_atom

CustomStructureBuilder.**init_atom**(*name*, *coord*, *b_factor*, *occupancy*, *altloc*, *fullname*, *serial_number=None*, *element=None*)
Create a new Atom object. :param - name - string, atom name, e.g. CA, spaces should be stripped: :param - coord - Numeric array: :type - coord - Numeric array: Float0, size 3 :param - b_factor - float, B factor: :param - occupancy - float: :param - altloc - string, alternative location specifier: :param - fullname - string, atom name including spaces, e.g. " CA ": :param - element - string, upper case, e.g. "HG" for mercury:

paramagpy.protein.CustomStructureBuilder.init_chain

CustomStructureBuilder.**init_chain**(*chain_id*)
Create a new Chain object with given id.

Parameters **chain_id – string** (–) –

paramagpy.protein.CustomStructureBuilder.init_model

CustomStructureBuilder.**init_model**(*model_id*, *serial_num=None*)
Create a new Model object with given id.

Parameters

- **id – int** (–) –
- **serial_num – int** (–) –

paramagpy.protein.CustomStructureBuilder.init_residue

CustomStructureBuilder.**init_residue**(*resname*, *field*, *resseq*, *icode*)
Create a new Residue object.

Parameters

- **resname – string, e.g. "ASN"** (–) –
- **field – hetero flag, "W" for waters, "H" for** (–) – hetero residues, otherwise blank.
- **resseq – int, sequence identifier** (–) –
- **icode – string, insertion code** (–) –

paramagpy.protein.CustomStructureBuilder.init_seg

CustomStructureBuilder.**init_seg**(*segid*)
Flag a change in segid.

Parameters **segid – string** (–) –

paramagpy.protein.CustomStructureBuilder.init_structure

CustomStructureBuilder.**init_structure**(*structure_id*)
Initialize a new Structure object with given id.

Parameters **id – string** (–) –

paramagpy.protein.CustomStructureBuilder.set_anisou

CustomStructureBuilder.**set_anisou**(*anisou_array*)
Set anisotropic B factor of current Atom.

paramagpy.protein.CustomStructureBuilder.set_header

CustomStructureBuilder.**set_header**(*header*)

paramagpy.protein.CustomStructureBuilder.set_line_counter

CustomStructureBuilder.**set_line_counter**(*line_counter*)
Tracks line in the PDB file that is being parsed.

Parameters **line_counter – int** (–) –

paramagpy.protein.CustomStructureBuilder.set_sigatm

CustomStructureBuilder.**set_sigatm**(*sigatm_array*)
Set standard deviation of atom position of current Atom.

paramagpy.protein.CustomStructureBuilder.set_siguij

CustomStructureBuilder.**set_siguij**(*siguij_array*)
Set standard deviation of anisotropic B factor of current Atom.

paramagpy.protein.CustomStructureBuilder.set_symmetry

CustomStructureBuilder.**set_symmetry**(*spacegroup*, *cell*)

5.5.3 Data I/O module

This module handles the reading and writing of experimental data.

paramagpy.dataparse

Functions

read_pcs(fileName)	Read pseudo contact shift values from file.
read_rdc(fileName)	Read residual dipolar coupling values from file.
read_pre(fileName)	Read paramagnetic relaxation enhancement values from file.
read_ccr(fileName)	Read cross-correlated relaxation values from file.

paramagpy.dataparse.read_pcs

paramagpy.dataparse.**read_pcs**(*fileName*)

Read pseudo contact shift values from file. The returned object is a dicationary. They keys are tuples of (sequence, atomName) The values are tuples of (value, error)

Parameters **fileName** (*str*) – the path to the file

Returns **values** – a dictionary containing the parsed data

Return type *paramagpy.dataparse.DataContainer*

Examples

```
>>> values = paramagpy.dataparse.read_pcs("calbindin_Er_HN_PCS_errors.npc")
>>> for v in values.items():
...     print(v)
...
((2, 'H'), (-0.04855485, 0.0016))
((2, 'N'), (-0.03402764, 0.0009))
((4, 'H'), (-0.18470315, 0.0004))
...
((75, 'H'), (0.19553661, 0.0005))
((75, 'N'), (0.17840666, 0.0004))
```

paramagpy.dataparse.read_rdc

paramagpy.dataparse.**read_rdc**(*fileName*)

Read residual dipolar coupling values from file. The returned object is a dicationary. They keys are frozensets of tuples of the form: frozenset({(sequence1, atomName1), (sequence2, atomName2)}) The frozenset only allows unordered unique atom identification pairs The values are tuples of (value, error)

Parameters **fileName** (*str*) – the path to the file

Returns **values** – a dictionary containing the parsed data

Return type *paramagpy.dataparse.DataContainer*

Examples

```
>>> values = paramagpy.dataparse.read_rdc("ubiquitin_a28c_c1_Tb_HN.rdc")
>>> for v in values.items():
...     print(v)
...
(frozenset({(2, 'N'), (2, 'H')}), (-2.35, 0.32))
(frozenset({(3, 'N'), (3, 'H')}), (-4.05, 0.38))
(frozenset({(4, 'H'), (4, 'N')}), (-3.58, 0.42))
...
(frozenset({(73, 'N'), (73, 'H')}), (-0.47, 0.75))
(frozenset({(76, 'H'), (76, 'N')}), (0.14, 0.3))
```

paramagpy.dataparse.read_pre

paramagpy.dataparse.**read_pre**(*fileName*)

Read paramagnetic relaxation enhancement values from file. The returned object is a dicationary. They keys are tuples of (sequence, atomName) The values are tuples of (value, error)

Parameters **fileName** (*str*) – the path to the file

Returns **values** – a dictionary containing the parsed data

Return type *paramagpy.dataparse.DataContainer*

Examples

see *paramagpy.dataparse.read_pcs()* which has the same file structure

paramagpy.dataparse.read_ccr

paramagpy.dataparse.**read_ccr**(*fileName*)

Read cross-correlated relaxation values from file. These are typically Curie-spin cross Dipole-dipole relaxation rates The returned object is a dicationary. They keys are tuples of the form: ((sequence1, atomName1), (sequence2, atomName2)) Note that the first column is for the active nucleus undergoing relaxation and the second column is for the partner spin. The values are tuples of (value, error)

Parameters **fileName** (*str*) – the path to the file

Returns **values** – a dictionary containing the parsed data

Return type *paramagpy.dataparse.DataContainer*

Examples

see *paramagpy.dataparse.read_rdc()* which has the similar file structure

Classes

DataContainer(*args, **kwargs)	A dictionary-like container for storing PCS, RDC, PRE and CCR data Has an additional attribute 'dtype' to define datatype

paramagpy.dataparse.DataContainer

class paramagpy.dataparse.**DataContainer**(**args*, ***kwargs*)

A dictionary-like container for storing PCS, RDC, PRE and CCR data Has an additional attribute 'dtype' to

define datatype

__init__ (**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

clear()	
copy()	
fromkeys(S[, v])	If not specified, the value defaults to None.
get(k[,d])	
items()	
keys()	
move_to_end	Move an existing element to the end (or beginning if last==False).
pop(k[,d])	value.
popitem()	Pairs are returned in LIFO order if last is true or FIFO order if false.
setdefault(k[,d])	
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values()	

paramagpy.dataparse.DataContainer.clear

DataContainer.**clear**() → None. Remove all items from od.

paramagpy.dataparse.DataContainer.copy

DataContainer.**copy**() → a shallow copy of od

paramagpy.dataparse.DataContainer.fromkeys

DataContainer.**fromkeys**(*S*[, *v*]) → New ordered dictionary with keys from S.
If not specified, the value defaults to None.

paramagpy.dataparse.DataContainer.get

DataContainer.**get**(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

paramagpy.dataparse.DataContainer.items

DataContainer.**items**() → a set-like object providing a view on D's items

paramagpy.dataparse.DataContainer.keys

DataContainer.**keys**() → a set-like object providing a view on D's keys

paramagpy.dataparse.DataContainer.move_to_end

DataContainer.**move_to_end**()
Move an existing element to the end (or beginning if last==False).

Raises KeyError if the element does not exist. When last=True, acts like a fast version of self[key]=self.pop(key).

paramagpy.dataparse.DataContainer.pop

DataContainer.**pop**(k[, d]) → v, remove specified key and return the corresponding value. If key is not found, d is returned if given, otherwise KeyError is raised.

paramagpy.dataparse.DataContainer.popitem

DataContainer.**popitem**() → (k, v), return and remove a (key, value) pair.
Pairs are returned in LIFO order if last is true or FIFO order if false.

paramagpy.dataparse.DataContainer.setdefault

DataContainer.**setdefault**(k[, d]) → od.get(k,d), also set od[k]=d if k not in od

paramagpy.dataparse.DataContainer.update

DataContainer.**update**([E], **F) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

paramagpy.dataparse.DataContainer.values

DataContainer.**values**() → an object providing a view on D's values

5.5.4 Fitting module

This module handles the fitting of paramagnetic objects to experimental data.

paramagpy.fit

paramagpy.fit

Functions

cantor_pairing(a, b)	Map two integers to a single integer.
clean_indices(indices)	Uniquely map a list of integers to their smallest size.
ensemble_average(atoms, *values)	
extract_ccr(data)	Extract values required for CCR calculations
extract_csa(data)	Extract CSA tensors from atoms
extract_pcs(data)	Extract values required for PCS calculations

Continued on next page

Table 16 – continued from previous page

extract_pre(data)	Extract values required for PRE calculations
extract_rdc(data)	Extract values required for RDC calculations
nlr_fit_metal_from_ccr(initMetals, ccrs[, ...])	Fit deltaChi tensor to CCR values using non-linear regression.
nlr_fit_metal_from_pcs(initMetals, pcss[, ...])	Fit deltaChi tensor to PCS values using non-linear regression.
nlr_fit_metal_from_pre(initMetals, pres[, ...])	Fit deltaChi tensor to PRE values using non-linear regression.
nlr_fit_metal_from_rdc(metal, rdc[, params, ...])	Fit deltaChi tensor to RDC values using non-linear regression.
pcs_fit_error_bootstrap(initMetals, pcss, ...)	Analyse uncertainty of PCS fit by Bootstrap methods.
pcs_fit_error_monte_carlo(initMetals, pcss, ...)	Analyse uncertainty of PCS fit by Monte-Carlo simulation This repeatedly adds noise to experimental PCS data and fits the tensor.
qfactor(experiment, calculated[, sumIndices])	Calculate the Q-factor to judge tensor fit quality
sphere_grid(origin, radius, points)	Make a grid of cartesian points within a sphere
svd_calc_metal_from_pcs(pos, pcs, idx, errors)	Solve PCS equation by single value decomposition.
svd_calc_metal_from_pcs_offset(pos, pcs, ...)	Solve PCS equation by single value decomposition with offset.
svd_calc_metal_from_rdc(vec, ...)	Solve RDC equation by single value decomposition.
svd_fit_metal_from_rdc(metal, rdc[, sumIndices])	Fit deltaChi tensor to RDC values using SVD algorithm.
svd_gridsearch_fit_metal_from_pcs(metals, pcss)	Fit deltaChi tensor to PCS values using Single Value Decomposition over a grid of points in a sphere.
unique_pairing(a, b)	Bijectively map two integers to a single integer.

paramagpy.fit.cantor_pairing

paramagpy.fit.**cantor_pairing**(*a*, *b*)

Map two integers to a single integer. The mapped space is minimum size. Ordering matters in this case. see Bijective mapping f:ZxZ->N.

Parameters

- **a** (*int*) –
- **b** (*int*) –

Returns **c** – bijective mapping (a, b) -> c

Return type int

paramagpy.fit.clean_indices

paramagpy.fit.**clean_indices**(*indices*)

Uniquely map a list of integers to their smallest size. For example: [7,4,7,9,9,10,1] -> [4 2 4 0 0 1 3]

Parameters **indices** (*array-like integers*) – a list of integers

Returns **new_indices** – the mapped integers with smallest size

Return type array-like integers

paramagpy.fit.ensemble_average

paramagpy.fit.**ensemble_average**(*atoms*, **values*)

paramagpy.fit.extract_ccr

paramagpy.fit.**extract_ccr**(*data*)
Extract values required for CCR calculations

Parameters **data** (*list of lists*) – A list with elements [Atom1, Atom2, value, error], where Atom is an Atom object, value is the CCR value, and error is the uncertainty

Returns **tuple** – all information required for CCR calculations

Return type dict

paramagpy.fit.extract_csa

paramagpy.fit.**extract_csa**(*data*)
Extract CSA tensors from atoms

Parameters **data** (*list of lists*) – A list with elements [Atom, value, error], where Atom is an Atom object, value is the PCS/RDC/PRE value, and error is the uncertainty

Returns **csas** – an array of each CSA tensor

Return type array of 3x3 arrays

paramagpy.fit.extract_pcs

paramagpy.fit.**extract_pcs**(*data*)
Extract values required for PCS calculations

Parameters **data** (*list of lists*) – A list with elements [Atom, value, error], where Atom is an Atom object, value is the PCS value, and error is the uncertainty

Returns **tuple** – all information required for PCS calculations

Return type (atom coordinates, PCS values, PCS errors, atom indices)

paramagpy.fit.extract_pre

paramagpy.fit.**extract_pre**(*data*)
Extract values required for PRE calculations

Parameters **data** (*list of lists*) – A list with elements [Atom, value, error], where Atom is an Atom object, value is the PRE value, and error is the uncertainty

Returns **tuple** – all information required for PRE calculations

Return type (atom coordinates, PRE values, PRE errors, atom indices)

paramagpy.fit.extract_rdc

paramagpy.fit.**extract_rdc**(*data*)
Extract values required for RDC calculations

Parameters **data** (*list of lists*) – A list with elements [Atom1, Atom2, value, error], where Atom is an Atom object, value is the RDC value, and error is the uncertainty

Returns **tuple** – RDC errors, atom indices) all information required for RDC calculations and fitting

Return type (inter-atomic vector, gamma values, RDC values,

paramagpy.fit.nlr_fit_metal_from_ccr

paramagpy.fit.**nlr_fit_metal_from_ccr**(*initMetals*, *ccrs*, *params=('x', 'y', 'z')*, *sumIndices=None*, *progress=None*)

Fit deltaChi tensor to CCR values using non-linear regression.

Parameters

- **initMetals** (*list of Metal objects*) – a list of metals used as starting points for fitting. a list must always be provided, but may also contain only one element. If multiple metals are provided, each metal is fitted to their respective CCR dataset by index, but all are fitted to a common position.
- **ccrs** (*list of CCR datasets*) – each CCR dataset must correspond to an associated metal for fitting. each CCR dataset has structure [Atom, value, error], where Atom is an Atom object, value is the PCS/RDC/PRE/CCR value and error is the uncertainty
- **params** (*list of str*) – the parameters to be fit. For example ['x','y','z','ax','rh','a','b','g','shift'] This defaults to ['x','y','z']
- **sumIndices** (*list of arrays of ints, optional*) – each index list must correspond to an associated ccr dataset. each index list contains an index assigned to each atom. Common indices determine summation between models for ensemble averaging. If None, defaults to atom serial number to determine summation between models.
- **progress** (*object, optional*) – to keep track of the calculation, progress.set(x) is called each iteration and varies from 0.0 -> 1.0 when the calculation is complete.

Returns

- **metals** (*list of metals*) – the metals fitted by NLR to the CCR data provided
- **calc_ccrs** (*list of lists of floats*) – the calculated CCR values
- **qfactors** (*list*)

paramagpy.fit.nlr_fit_metal_from_pcs

paramagpy.fit.**nlr_fit_metal_from_pcs**(*initMetals*, *pcss*, *params=('x', 'y', 'z', 'ax', 'rh', 'a', 'b', 'g')*, *sumIndices=None*, *userads=False*, *useracs=False*, *progress=None*)

Fit deltaChi tensor to PCS values using non-linear regression.

Parameters

- **initMetals** (*list of Metal objects*) – a list of metals used as starting points for fitting. a list must always be provided, but may also contain only one element. If multiple metals are provided, each metal is fitted to their respective PCS dataset by index, but all are fitted to a common position.
- **pcss** (*list of PCS datasets*) – each PCS dataset must correspond to an associated metal for fitting. each PCS dataset has structure [Atom, value, error], where Atom is an Atom object, value is the PCS/RDC/PRE value and error is the uncertainty
- **params** (*list of str*) – the parameters to be fit. For example ['x','y','z','ax','rh','a','b','g','shift']
- **sumIndices** (*list of arrays of ints, optional*) – each index list must correspond to an associated pcs dataset. each index list contains an index assigned to each atom. Common indices determine summation between models for

ensemble averaging. If None, defaults to atom serial number to determine summation between models.

- **userads** (*bool, optional*) – include residual anisotropic dipolar shielding (RADS) during fitting
- **useracs** (*bool, optional*) – include residual anisotropic chemical shielding (RACS) during fitting. CSA tensors are taken using the <csa> method of atoms.
- **progress** (*object, optional*) – to keep track of the calculation, progress.set(x) is called each iteration and varies from 0.0 -> 1.0 when the calculation is complete.

Returns

- **metals** (*list of metals*) – the metals fitted by NLR to the PCS data provided
- **calc_pcss** (*list of lists of floats*) – the calculated PCS values

paramagpy.fit.nlr_fit_metal_from_pre

paramagpy.fit.**nlr_fit_metal_from_pre**(*initMetals*, *pres*, *params=('x', 'y', 'z')*, *sumIndices=None*, *rtypes=None*, *usesbm=True*, *usegsbm=False*, *usedsa=True*, *usecsa=False*, *progress=None*)

Fit deltaChi tensor to PRE values using non-linear regression.

Parameters

- **initMetals** (*list of Metal objects*) – a list of metals used as starting points for fitting. a list must always be provided, but may also contain only one element. If multiple metals are provided, each metal is fitted to their respective PRE dataset by index, but all are fitted to a common position.
- **pres** (*list of PRE datasets*) – each PRE dataset must correspond to an associated metal for fitting. each PRE dataset has structure [Atom, value, error], where Atom is an Atom object, value is the PCS/RDC/PRE value and error is the uncertainty
- **params** (*list of str*) – the parameters to be fit. For example ['x','y','z','ax','rh','a','b','g','iso','taur','t1e']
- **sumIndices** (*list of arrays of ints, optional*) – each index list must correspond to an associated pcs dataset. each index list contains an index assigned to each atom. Common indices determine summation between models for ensemble averaging. If None, defaults to atom serial number to determine summation between models.
- **rtypes** (*list of str, optional*) – the relaxtion type, either 'r1' or 'r2'. A list must be provided with each element corresponding to an associated dataset. Defaults to 'r2' for all datasets of None is specified.
- **usesbm** (*bool, optional*) – include Solomon-Bloemenbergen-Morgan (Dipole-dipole) relaxation theory. default is True
- **usegsbm** (*bool, optional*) – include anisotropic dipolar relaxation theory. note that the g-tensor must be set for this default is False
- **usedsa** (*bool, optional*) – include Dipolar-Shielding-Anisotropy (Curie Spin) relaxation theory. default is True
- **usecsa** (*bool, optional*) – include Chemical-Shift-Anisotropy cross-correlated realxation theory. default is False

- **progress** (*object, optional*) – to keep track of the calculation, progress.set(x) is called each iteration and varies from 0.0 -> 1.0 when the calculation is complete.

Returns **metals** – the metals fitted by NLR to the PRE data provided

Return type list of metals

paramagpy.fit.nlr_fit_metal_from_rdc

paramagpy.fit.**nlr_fit_metal_from_rdc**(*metal*, *rdc*, *params=('ax', 'rh', 'a', 'b', 'g')*, *sumIndices=None*, *progress=None*)

Fit deltaChi tensor to RDC values using non-linear regression.

Parameters

- **metal** (*Metal object*) – the starting metal for fitting
- **rdc** (*the RDC dataset*) – each RDC dataset has structure [Atom, value, error], where Atom is an Atom object, value is the PCS/RDC/PRE value and error is the uncertainty
- **params** (*list of str, optional*) – the parameters to be fit. this defaults to ['ax','rh','a','b','g']
- **sumIndices** (*array of ints, optional*) – the list contains an index assigned to each atom. Common indices determine summation between models for ensemble averaging. If None, defaults to atom serial number to determine summation between models.
- **progress** (*object, optional*) – to keep track of the calculation, progress.set(x) is called each iteration and varies from 0.0 -> 1.0 when the calculation is complete.

Returns

- **fitMetal** (*Metal object*) – the fitted metal by NLR to the RDC data provided
- **calculated** (*array of floats*) – the calculated RDC values
- **qfac** (*float*) – the qfactor judging the fit quality

paramagpy.fit.pcs_fit_error_bootstrap

paramagpy.fit.**pcs_fit_error_bootstrap**(*initMetals*, *pcss*, *iterations*, *fraction*, *params=('x', 'y', 'z', 'ax', 'rh', 'a', 'b', 'g')*, *sumIndices=None*, *userads=False*, *useracs=False*, *progress=None*)

Analyse uncertainty of PCS fit by Bootstrap methods. This repeats the tensor fitting, but each time samples a fraction of the data randomly. The standard deviation in fitted parameters over each iteration is then reported.

Parameters

- **initMetals** (*list of Metal objects*) – a list of metals used as starting points for fitting. a list must always be provided, but may also contain only one element. If multiple metals are provided, each metal is fitted to their respective PCS dataset by index, but all are fitted to a common position.
- **pcss** (*list of PCS datasets*) – each PCS dataset must correspond to an associated metal for fitting. each PCS dataset has structure [Atom, value, error], where Atom is an Atom object, value is the PCS/RDC/PRE value and error is the uncertainty
- **iterations** (*int*) – the number of Monte Carlo iterations to perform

- **fraction** (*float*) – must be between 0 and 1 the proportion of data to be sample for fitting with each iteration of the bootstrap method.
- **params** (*list of str*) – the parameters to be fit. For example ['x','y','z','ax','rh','a','b','g','shift']
- **sumIndices** (*list of arrays of ints, optional*) – each index list must correspond to an associated pcs dataset. each index list contains an index assigned to each atom. Common indices determine summation between models for ensemble averaging. If None, defaults to atom serial number to determine summation between models.
- **userads** (*bool, optional*) – include residual anisotropic dipolar shielding (RADS) during fitting
- **useracs** (*bool, optional*) – include residual anisotropic chemical shielding (RACS) during fitting. CSA tensors are taken using the <csa> method of atoms.
- **progress** (*object, optional*) – to keep track of the calculation, progress.set(x) is called each iteration and varies from 0.0 -> 1.0 when the calculation is complete.

Returns

- **sample_metals** (*list of list of metals*) – the metals fitted by NLR to the PCS data with noise at each iteration
- **std_metals** (*list of metals*) – the standard deviation in fitted parameters over all iterations of the Monte Carlo simulation. These are stored within the metal object. All unfitted parameters are zero.

paramagpy.fit.pcs_fit_error_monte_carlo

paramagpy.fit.**pcs_fit_error_monte_carlo**(*initMetals*, *pcss*, *iterations*, *params=('x', 'y', 'z', 'ax', 'rh', 'a', 'b', 'g')*, *sumIndices=None*, *userads=False*, *useracs=False*, *progress=None*)

Analyse uncertainty of PCS fit by Monte-Carlo simulation This repeatedly adds noise to experimental PCS data and fits the tensor. The standard deviation of the fitted parameters across each iteration is then reported.

Parameters

- **initMetals** (*list of Metal objects*) – a list of metals used as starting points for fitting. a list must always be provided, but may also contain only one element. If multiple metals are provided, each metal is fitted to their respective PCS dataset by index, but all are fitted to a common position.
- **pcss** (*list of PCS datasets*) – each PCS dataset must correspond to an associated metal for fitting. each PCS dataset has structure [Atom, value, error], where Atom is an Atom object, value is the PCS/RDC/PRE value and error is the uncertainty
- **iterations** (*int*) – the number of Monte Carlo iterations to perform
- **params** (*list of str*) – the parameters to be fit. For example ['x','y','z','ax','rh','a','b','g','shift']
- **sumIndices** (*list of arrays of ints, optional*) – each index list must correspond to an associated pcs dataset. each index list contains an index assigned to each atom. Common indices determine summation between models for ensemble averaging. If None, defaults to atom serial number to determine summation between models.
- **userads** (*bool, optional*) – include residual anisotropic dipolar shielding (RADS) during fitting

- **useracs** (*bool, optional*) – include residual anisotropic chemical shielding (RACS) during fitting. CSA tensors are taken using the <csa> method of atoms.
- **progress** (*object, optional*) – to keep track of the calculation, progress.set(x) is called each iteration and varies from 0.0 -> 1.0 when the calculation is complete.

Returns

- **sample_metals** (*list of list of metals*) – the metals fitted by NLR to the PCS data with noise at each iteration
- **std_metals** (*list of metals*) – the standard deviation in fitted parameters over all iterations of the Monte Carlo simulation. These are stored within the metal object. All unfitted parameters are zero.

paramagpy.fit.qfactor

paramagpy.fit.**qfactor**(*experiment*, *calculated*, *sumIndices=None*)

Calculate the Q-factor to judge tensor fit quality

A lower value indicates a better fit. The Q-factor is calculated using the following equation:

$$Q = \sqrt{\frac{\sum_i \left[\left(\sum_m \left[PCS_{m,i}^{exp} - PCS_{m,i}^{calc} \right] \right)^2 \right]}{\sum_i \left[\left(\sum_m \left[PCS_{m,i}^{exp} \right] \right)^2 \right]}}$$

where m and i are usually indexed over models and atoms respectively.

Parameters

- **experiment** (*list of floats*) – the experimental values
- **calculated** (*list of floats*) – the corresponding caluclated values from the fitted model
- **sumIndices** (*list ints, optional*) – Common indices determine summation between models for ensemble averaging. If None, no ensemble averaging is conducted

Returns **qfactor** – the Q-factor

Return type float

paramagpy.fit.sphere_grid

paramagpy.fit.**sphere_grid**(*origin*, *radius*, *points*)

Make a grid of cartesian points within a sphere

Parameters

- **origin** (*float*) – the centre of the sphere
- **radius** (*float*) – the radius of the sphere
- **points** (*int*) – the number of points per radius

Returns **array** – the points within the sphere

Return type array of [x,y,z] coordinates

paramagpy.fit.svd_calc_metal_from_pcs

paramagpy.fit.**svd_calc_metal_from_pcs**(*pos*, *pcs*, *idx*, *errors*)

Solve PCS equation by single value decomposition. This function is generally called by higher methods like <svd_gridsearch_fit_metal_from_pcs>

Parameters

- **pos** (*array of [x,y,z] floats*) – the atomic positions in meters
- **pcs** (*array of floats*) – the PCS values in ppm
- **idx** (*array of ints*) – an index assigned to each atom. Common indices determine summation between models for ensemble averaging.
- **errors** (*array of floats*) – the standard deviation representing experimental uncertainty in the measured value

Returns **tuple** – calc are the calculated PCS values from the fitted tensor sol is the solution to the linearised PCS equation and consists of the tensor matrix elements

Return type (calc, sol)

paramagpy.fit.svd_calc_metal_from_pcs_offset

paramagpy.fit.**svd_calc_metal_from_pcs_offset**(*pos*, *pcs*, *idx*, *errors*)

Solve PCS equation by single value decomposition with offset. An offset arising from referencing errors between diamagnetic and paramagnetic datasets can be accounted for using this method. This function is generally called by higher methods like <svd_gridsearch_fit_metal_from_pcs>

NOTE: the factor of 1E26 is required for floating point error mitigation

Parameters

- **pos** (*array of [x,y,z] floats*) – the atomic positions in meters
- **pcs** (*array of floats*) – the PCS values in ppm
- **idx** (*array of ints*) – an index assigned to each atom. Common indices determine summation between models for ensemble averaging.
- **errors** (*array of floats*) – the standard deviation representing experimental uncertainty in the measured value

Returns **tuple** – calc are the calculated PCS values from the fitted tensor sol is the solution to the linearised PCS equation and consists of the tensor matrix elements and offset

Return type (calc, sol)

paramagpy.fit.svd_calc_metal_from_rdc

paramagpy.fit.**svd_calc_metal_from_rdc**(*vec*, *rdc_parameterised*, *idx*, *errors*)

Solve RDC equation by single value decomposition. This function is generally called by higher methods like <svd_fit_metal_from_rdc>

Parameters

- **vec** (*array of [x,y,z] floats*) – the internuclear vectors in meters
- **rdc_parameterised** (*array of floats*) – the experimental RDC values, normalised by a prefactor
- **idx** (*array of ints*) – an index assigned to each atom. Common indices determine summation between models for ensemble averaging.

- **errors** (*array of floats*) – the standard deviation representing experimental uncertainty in the measured value

Returns

- **calc** (*array of floats*) – the calculated RDC values from the fitted tensor
- **sol** (*array of floats*) – sol is the solution to the linearised PCS equation and consists of the tensor matrix elements

paramagpy.fit.svd_fit_metal_from_rdc

paramagpy.fit.**svd_fit_metal_from_rdc**(*metal*, *rdc*, *sumIndices=None*)

Fit deltaChi tensor to RDC values using SVD algorithm. Note this is a weighted SVD calculation which takes into account experimental errors. Ensemble averaging defaults to atom numbers.

Parameters

- **metal** (*Metal object*) – the starting metal for fitting
- **rdc** (*the RDC dataset*) – each RDC dataset has structure [(Atom1, Atom2), value, error], where Atom is an Atom object, value is the RDC value and error is the uncertainty
- **sumIndices** (*array of ints, optional*) – the list contains an index assigned to each atom. Common indices determine summation between models for ensemble averaging. If None, defaults to atom serial number to determine summation between models.

Returns

- **fitMetal** (*Metal object*) – the fitted metal by NLR to the RDC data provided
- **calculated** (*array of floats*) – the calculated RDC values
- **qfac** (*float*) – the qfactor judging the fit quality

paramagpy.fit.svd_gridsearch_fit_metal_from_pcs

paramagpy.fit.**svd_gridsearch_fit_metal_from_pcs**(*metals*, *pcss*, *sumIndices=None*, *origin=None*, *radius=20.0*, *points=16*, *offsetShift=False*, *progress=None*)

Fit deltaChi tensor to PCS values using Single Value Decomposition over a grid of points in a sphere. Note this uses a weighted SVD fit which takes into account experimental errors Ensemble averaging is determined by atom number.

Parameters

- **metals** (*list of Metal objects*) – a list of metals used as starting points for fitting. a list must always be provided, but may also contain only one element. If multiple metals are provided, each metal is fitted to their respective PCS dataset by index, but all are fitted to a common position.
- **pcss** (*list of PCS datasets*) – each PCS dataset must correspond to an associated metal for fitting. each PCS dataset has structure [Atom, value, error], where Atom is an Atom object, value is the PCS/RDC/PRE value and error is the uncertainty
- **sumIndices** (*list of arrays of ints, optional*) – each index list must correspond to an associated pcs dataset. each index list contains an index assigned to each atom. Common indices determine summation between models for ensemble averaging. If None, defaults to atom serial number to determine summation between models.

- **origin** (*float, optional*) – the centre of the gridsearch of positions in Angstroms. If None, the position of the first metal is used
- **radius** (*float, optional*) – the radius of the gridsearch in Angstroms.
- **points** (*int, optional*) – the number of points per radius in the gridsearch
- **offsetShift** (*bool, optional*) – if True, an offset value added to all PCS values is included in the SVD fitting. This may arise due to a referencing error between diamagnetic and paramagnetic PCS datasets and may be used when many data points are available. Default False, no offset is included in the fitting.
- **progress** (*object, optional*) – to keep track of the calculation, progress.set(x) is called each iteration and varies from 0.0 -> 1.0 when the calculation is complete.

Returns **minmetals** – the metals fitted by SVD to the PCS data provided

Return type list of metals

paramagpy.fit.unique_pairing

paramagpy.fit.**unique_pairing**(*a*, *b*)

Bijectively map two integers to a single integer. The mapped space is minimum size. The input is symmetric. see Bijective mapping f:ZxZ->N.

Parameters

- **a** (*int*) –
- **b** (*int*) –

Returns **c** – bijective symmetric mapping (a, b) | (b, a) -> c

Return type int

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

N

P

Q

R

S

T

U

V

W

X

Y

Z